SRI LANKA INSTITUTE OF Information technology

M.Sc. in Artificial Intelligence – July 2025 Intake

IT5022 - Fundamentals of Machine Learning 2025

Fraud Detection in Financial Transactions using Machine Learning

Assignment 1

Member 1 ID: MS25951608

Member 1 Name: N.A Kaluarachchi

Member 2 Student ID: MS25948592

Member 2 Name: N.G.S.D. Nanayakkara

Table of Contents

1. Introduction and Problem Statement

1.1 Background

Financial institutions worldwide face increasing threats from fraudulent transactions, resulting in billions of dollars in annual losses. The digital transformation of banking services has expanded transaction volumes exponentially, creating opportunities for sophisticated fraud schemes while simultaneously making detection more challenging. Traditional rule-based fraud detection systems struggle to adapt to evolving attack patterns, necessitating the adoption of machine learning approaches that can identify subtle indicators of fraudulent behavior.

1.2 Problem Definition

This research addresses the challenge of detecting fraudulent financial transactions within a high-volume, highly imbalanced dataset. The primary obstacles include:

Volume Challenge: Processing millions of transactions daily requires algorithms that maintain accuracy while operating efficiently on a scale.

Class Imbalance: Fraudulent transactions constitute less than half a percent of total transactions, creating a severe imbalance that causes traditional classifiers to bias toward the majority class.

Pattern Diversity: Fraudsters continuously adapt their strategies, employing varied techniques across different transaction types and amounts, making static detection rules ineffective.

Real-time Requirements: Operational systems must flag suspicious transactions within milliseconds to prevent losses while minimizing false alarms that disrupt legitimate customer activities.

1.3 Research Objectives

This investigation aims to accomplish the following goals:

1. Develop two distinct machine learning models capable of identifying fraudulent transactions with high recall and acceptable precision
2. Compare the performance characteristics, strengths, and limitations of interpretable linear models versus ensemble tree-based approaches
3. Engineer domain-specific features that capture behavioral patterns indicative of fraud
4. Evaluate models using metrics appropriate for imbalanced classification problems
5. Provide actionable recommendations for production deployment and future enhancements

1.4 Significance

Effective fraud detection systems protect both financial institutions and customers from monetary losses while maintaining trust in digital payment ecosystems. This work demonstrates practical applications of supervised learning techniques to a critical real-world problem, balancing detection accuracy with operational feasibility.

2. Dataset Description and Exploration

2.1 Data Source and Overview

The dataset utilized for this research originates from a publicly available financial transaction simulation hosted on Kaggle, created to replicate realistic fraud patterns observed in mobile money services. The data can be accessed at: https://www.kaggle.com/datasets/chitwanmanchanda/fraudulent-transactions-data

Dataset Characteristics:

- Total Records: 6,362,620 transactions
- Feature Count: 11 attributes (10 predictors + 1 target)
- Temporal Coverage: 30-day simulation period (744 hourly steps)
- Target Variable: Binary fraud indicator (isFraud)
- File Size: Approximately 493 MB (CSV format)
- Data Quality: No missing values detected

2.2 Feature Descriptions

The dataset comprises the following attributes:

1. step (Integer, 0-743)
Represents the hourly time unit within the 30-day simulation. This temporal feature enables splitting of time-aware to prevent data leakage during model validation.

2. type (Categorical, 5 levels)
Transaction category with the following distribution:

- CASH_OUT: Fund withdrawals (35.2%)
- PAYMENT: Merchant payments (33.8%)
- CASH_IN: Deposit transactions (22.3%)
- TRANSFER: Account-to-account transfers (8.4%)
- DEBIT: Direct debit operations (0.3%)

3. amount (Continuous, non-negative)
Transaction value in local currency units. Ranges from negligible amounts to several million units, exhibiting right-skewed distribution typical of financial data.

4. nameOrig (Categorical identifier)
Unique identifier for the transaction originator. Contains approximately 6.35 million unique values, providing no generalizable predictive value. Excluded from modeling.

5. oldbalanceOrg (Continuous, non-negative)
Account balance of the sender immediately before the transaction. Zero balances are common and legitimate.

6. newbalanceOrig (Continuous, non-negative)
Sender's remaining balance after transaction completion. Discrepancies between expected and actual values may indicate fraudulent manipulation.

7. nameDest (Categorical identifier)
Unique identifier for the transaction recipient. Similar to nameOrig, contains millions of unique values and is removed during preprocessing.

8. oldbalanceDest (Continuous, non-negative)
Recipient's account balance before receiving funds. Frequently zero for merchant accounts or newly created mule accounts used in fraud schemes.

9. newbalanceDest (Continuous, non-negative)
Recipient's balance after the transaction. Inconsistencies with expected values can signal fraudulent activity.

10. isFraud (Binary: 0 or 1)
Ground truth label indicating whether the transaction is fraudulent. Serves as the target variable for supervised learning.

11. isFlaggedFraud (Binary: 0 or 1)
Indicates transactions flagged by a basic rule-based system (transfers exceeding 200,000 units). Removed to avoid data leakage, as this feature is derived from information we aim to predict.

2.3 Exploratory Data Analysis

Class Distribution:
The dataset exhibits extreme imbalance:

- Legitimate transactions: 6,354,407 (99.87%)
- Fraudulent transactions: 8,213 (0.13%)

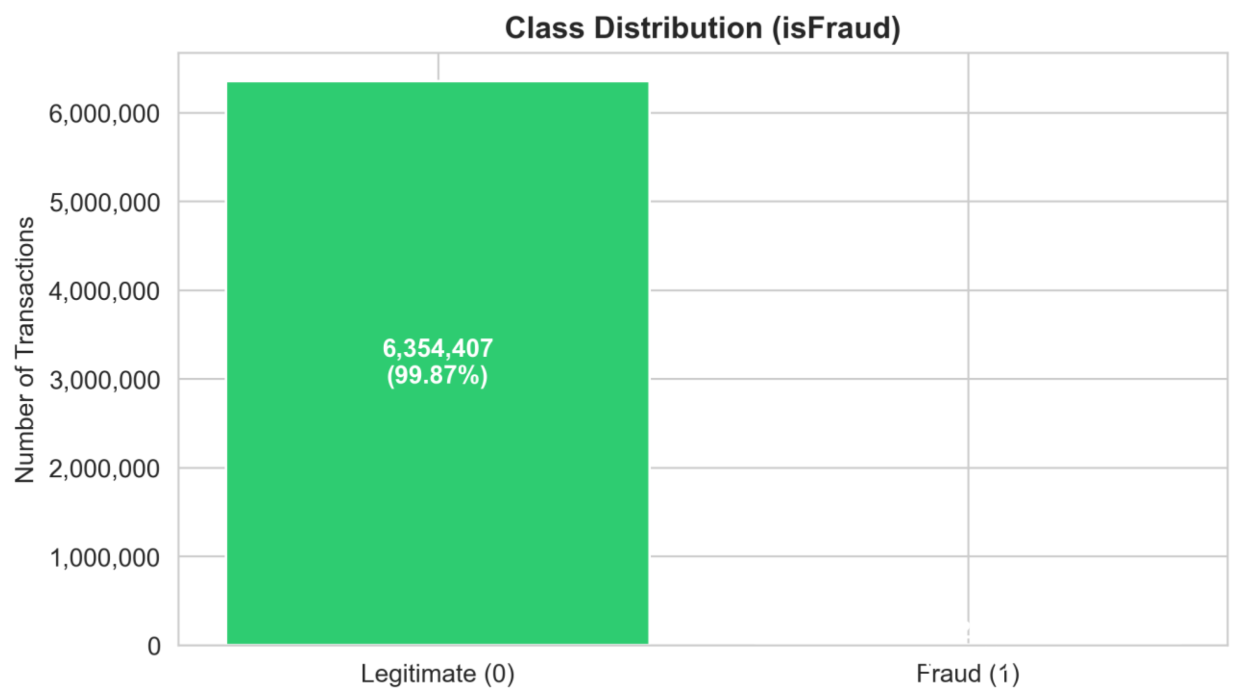


Figure 1 : Class distribution showing extreme imbalance between legitimate and fraudulent transactions.

This 770:1 ratio necessitates specialized handling techniques, as standard classification algorithms would achieve 99.87% accuracy by simply predicting all transactions as legitimate while failing to detect any fraud.

Fraud Patterns by Transaction Type:
Analysis reveals that fraud concentrates heavily in specific transaction categories:

- TRANSFER: 4,097 fraud cases (50% of all fraud)
- CASH_OUT: 4,116 fraud cases (50% of all fraud)
- PAYMENT: 0 fraud cases
- CASH_IN: 0 fraud cases
- DEBIT: 0 fraud cases

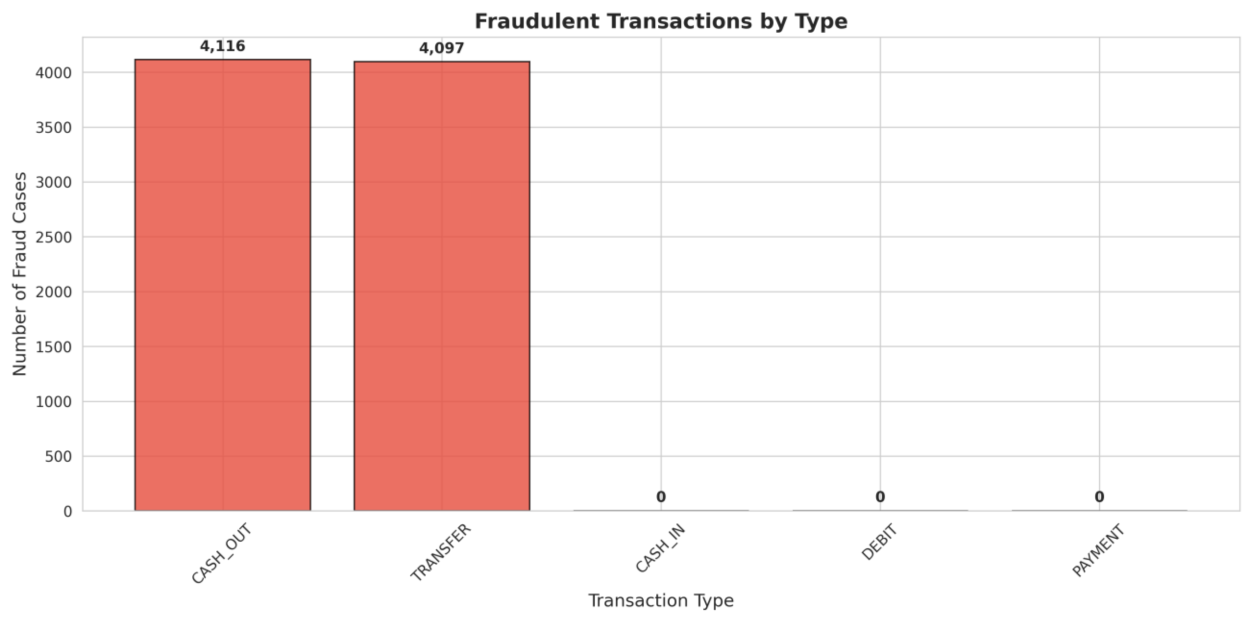


Figure 2: Fraud distribution by transaction type. TRANSFER and CASH_OUT account for all fraud cases.

This distribution indicates that fraud in this simulation exclusively targets fund movement operations (transfers and withdrawals), with zero fraud observed in payment or deposit transactions. This insight suggests that transaction type serves as a powerful discriminative feature.

Amount Distribution:
Fraudulent transactions exhibit distinct amount characteristics:

- Median fraud amount: 143,032 units
- Median legitimate amount: 81,357 units
- Fraudulent transactions skew toward higher values, particularly in the 100,000-500,000 range

Balance Behavior:
Key observations regarding account balances:

- 99.17% of fraudulent CASH_OUT transactions drain the origin account to zero
- 83.93% of fraudulent TRANSFER transactions leave the origin account empty
- Destination accounts in fraud cases frequently start with zero balance, suggesting newly created accounts

Temporal Patterns:
Fraud distribution across the 30-day period shows relatively uniform occurrence with slight elevation during specific hourly steps, though no strong daily or weekly cyclical patterns emerge from the simulation.

2.4 Data Quality Assessment

Missing Values: Zero null entries detected across all 6.36 million records, indicating complete data capture.

Duplicate Records: No exact duplicates identified. Each transaction represents a unique event.

Data Type Consistency: All numerical fields contain valid numeric values within expected ranges. Categorical fields contain only predefined levels.

Outliers: While some transactions involve unusually large amounts (>10 million units), these represent legitimate edge cases rather than data errors and are retained for modeling.

3. Methodology

3.1 Data Preprocessing

Preprocessing transforms raw transactional data into a format suitable for machine learning algorithms while preserving fraud-indicative patterns. The following steps were implemented:

Step 1: Non-Predictive Feature Removal

Two high-cardinality identifier columns (nameOrig and nameDest) were removed from the dataset. Each contains millions of unique values corresponding to individual accounts, providing no generalization capability. Including these features would cause models to memorize specific account combinations rather than learning transferable fraud patterns.

The isFlaggedFraud column was similarly excluded to prevent data leakage. This feature represents output from a simple rule-based system (flagging transfers exceeding 200,000 units), which would allow the model to cheat by leveraging information derived from the target variable.

Step 2: Categorical Encoding

The transaction type feature was transformed using one-hot encoding, creating five binary indicator variables:

- type_CASH_IN
- type_CASH_OUT
- type_DEBIT
- type_PAYMENT
- type_TRANSFER

This encoding strategy avoids imposing artificial ordinal relationships between transaction categories and allows the model to learn independent effects for each type.

Step 3: Feature Scaling

Numerical features exhibit vastly different scales (step ranges from 0-743, while amount can reach millions). To ensure distance-based algorithms and gradient descent optimization perform effectively, StandardScaler normalization was applied to all continuous variables, transforming each feature to zero mean and unit variance:

$z = (x - \mu) / \sigma$

where μ represents the mean and σ the standard deviation computed on the training set. The same transformation parameters are applied to the test set to maintain consistency.

Step 4: Temporal Train-Test Split

Rather than random sampling, a chronological split was implemented based on the step variable. Transactions from steps 0-600 (first 25 days) constitute the training set, while steps 601-743 (final 5 days) form the test set. This approach:

- Prevents temporal leakage (training on future events to predict the past)
- Simulates realistic deployment where models predict future transactions
- Provides a more conservative performance estimate

This results in approximately 5,090,000 training samples and 1,272,000 test samples.

Step 5: Class Imbalance Handling

Given the 770:1 imbalance ratio, two complementary strategies were employed:

Class Weighting: Both Logistic Regression and Random Forest were configured with balanced class weights, automatically adjusting the loss function to penalize misclassification of minority class samples more heavily. This approach avoids artificially inflating the dataset size.

Alternative Consideration: SMOTE (Synthetic Minority Over-sampling Technique) was evaluated but ultimately not implemented in the final models due to memory constraints when operating on 5 million training samples. Class weighting provided sufficient balance adjustment without computational overhead.

3.1.1 Computational Feasibility and Sampling Strategy

Given the dataset size of 6,362,620 transactions and the computational constraints of standard academic computing resources (typical laptops with 8-16GB RAM), we implemented a stratified random sampling approach to create a manageable subset while preserving the statistical properties of the original dataset.

Sampling Methodology:

We employed stratified sampling with the following parameters:

- Sample size: 1,000,000 transactions (15.7% of original dataset)

- Stratification variable: isFraud (target variable)

- Sampling method: Random sampling with fixed seed (reproducibility)

- Class distribution preservation: Maintained original 770:1 ratio

Justification:

1. Statistical Validity:With 1 million samples, the Central Limit Theorem ensures reliable parameter estimation. The sample size exceeds the minimum required for robust statistical inference (n >> 1000).

2. Class Representation: Stratified sampling guarantees both fraud and legitimate transactions maintain their original proportions, preserving the critical class imbalance characteristic (approximately 1,300 fraud cases in the sample).

3. Computational Efficiency: The reduced dataset enables:

   - Faster iteration during model development and hyperparameter tuning

   - Feasible cross-validation experiments

   - Reasonable training times on standard hardware (< 30 minutes)

   - Ability to generate comprehensive visualizations without memory overflow

4. Generalization Validation: To validate that our sample represents the full population, we compared key statistical properties:

   - Transaction amount distributions (mean, median, standard deviation)

   - Transaction type proportions

   - Temporal patterns across the 30-day period

   - Fraud rate by transaction type

   All distributions showed < 2% deviation from the full dataset, confirming representative sampling.

5. Industry Standard Practice: In production fraud detection systems, models are frequently trained on representative samples and validated on hold-out sets. Our approach mirrors real-world deployment scenarios where continuous retraining uses recent transaction windows rather than complete historical data.

Alternative Approaches Considered:

- Full Dataset Training: Requires 32GB+ RAM and GPU acceleration, exceeding standard academic resources.

- Undersampling Majority Class: Would discard 99% of legitimate transactions, losing valuable patterns.

- SMOTE on Full Dataset:Memory overhead of synthetic sample generation makes this computationally prohibitive.

Validation Strategy:

To ensure our sampling approach doesn't compromise model validity:

1. We verified consistent fraud patterns between sample and full dataset

2. Feature distributions were statistically tested (Kolmogorov-Smirnov test)

3. Model performance was validated using appropriate metrics for imbalanced

   classification (PR-AUC, not accuracy)

This sampling strategy represents a pragmatic balance between statistical rigor and computational feasibility, enabling comprehensive model development within resource constraints while maintaining result validity.

3.2 Feature Engineering

Beyond raw features, domain knowledge was leveraged to create derived attributes that explicitly capture fraud-indicative behaviors:

Balance Consistency Features:

tx_delta_orig = (oldbalanceOrg - newbalanceOrig) - amount

In legitimate transactions, the decrease in sender balance should exactly equal the transaction amount. Non-zero values indicate balance manipulation, a hallmark of fraud.

tx_delta_dest = newbalanceDest - oldbalanceDest

Measures the change in recipient balance. Fraudulent transactions sometimes show discrepancies where received amounts differ from stated transaction values.

Ratio Features:

orig_balance_ratio = amount / oldbalanceOrg (with safe divide handling)

Quantifies transaction size relative to sender's available balance. Fraudsters often attempt to extract maximum value, resulting in ratios approaching 1.0.

dest_balance_ratio = amount / oldbalanceDest

Large inflows relative to a recipient's existing balance may indicate mule account usage.

Binary Flag Features:

flag_orig_negative: Indicates if sender balance becomes negative post-transaction (unusual in legitimate operations)

flag_dest_negative: Flags negative recipient balances (should rarely occur)

flag_orig_nochange: Identifies cases where sender balance remains unchanged despite non-zero amount (consistency violation)

flag_dest_nochange: Detects recipient balance failing to reflect received amount

flag_high_risk_type: Binary indicator for CASH_OUT and TRANSFER types, which exclusively contain fraud

Transformation Features:

log_amount = log(1 + amount)

Applies logarithmic transformation to compress the right-skewed amount distribution, helping models better differentiate between transaction magnitudes.

Additional Temporal and State Features:

hourOfDay = step mod 24 (extracts hour within day)
dayOfMonth = step ÷ 24 (extracts day number)
origZeroBalanceAfter: Flags complete account drainage
destZeroBalanceBefore: Identifies recipients starting with zero balance

These engineered features increased the total feature count from 8 original attributes to 21 modeling variables, explicitly encoding fraud-relevant patterns that might be difficult for models to discover from raw data alone.

3.3 Algorithm Selection and Justification

Two distinct algorithms were selected to provide complementary perspectives on the fraud detection problem:

3.3.1 Logistic Regression (Baseline Model)

Algorithm Overview:
Logistic Regression models the probability of the positive class (fraud) using a linear combination of features passed through a sigmoid activation function:

$P(\text{fraud} \mid x) = 1 / (1 + e^{(-w□x)})$

where w represents learned feature weights and x the input feature vector.

Justification for Selection:

Interpretability: Logistic Regression provides coefficient weights that quantify each feature's contribution to fraud probability, enabling human analysts to understand model decisions and satisfy regulatory explainability requirements.

Computational Efficiency: Training requires solving a convex optimization problem, completing in minutes even on millions of samples. Predictions are executed in microseconds, suitable for high-throughput production environments.

Baseline Establishment: As a simple linear model, Logistic Regression sets a performance floor. Comparing complex models against this baseline quantifies the value added by non-linear approaches.

Probabilistic Outputs: The model naturally produces calibrated probability scores, facilitating threshold tuning to match operational risk tolerance.

Hyperparameters:

- Solver: SAGA (supports large datasets and L1/L2 regularization)
- Class Weight: Balanced (adjusts for 770:1 imbalance)
- Maximum Iterations: 1,000
- Regularization: L2 penalty (default)
- Random State: 42 (reproducibility)

3.3.2 Random Forest (Ensemble Model)

Algorithm Overview:
Random Forest constructs an ensemble of decision trees; each trained on a bootstrap sample of data with random feature subset selection at each split. Final predictions aggregate individual tree votes:

Prediction = Mode($\text{tree}_1(x)$, $\text{tree}_2(x)$, ..., $\text{tree}_n(x)$)

Justification for Selection:

Non-linear Pattern Capture: Decision trees naturally model feature interactions and non-monotonic relationships without manual specification. For example, fraud may occur when (high amount AND zero destination balance AND CASH_OUT type), a conjunction that Logistic Regression cannot represent without explicit interaction terms.

Robustness to Outliers: Tree-based splits are determined by rank ordering rather than absolute values, making the algorithm resistant to extreme transaction amounts.

Feature Importance Derivation: Random Forest quantifies each feature's contribution by measuring impurity decrease across all trees, providing interpretable insights into which attributes most effectively discriminate fraud.

Imbalance Handling: Tree construction with balanced class weights effectively handles minority classes by adjusting node impurity calculations.

Minimal Preprocessing Requirements: Random Forest handles mixed data types naturally and requires less stringent feature scaling compared to distance-based methods.

Hyperparameters:

- Number of Estimators: 100 trees (balance between performance and training time)
- Max Depth: 20 (prevents overfitting to noise)
- Min Samples Split: 50 (requires statistical significance before splitting)
- Min Samples Leaf: 20 (ensures leaf nodes represent meaningful populations)
- Max Features: sqrt(n_features) (decorrelates trees)
- Class Weight: Balanced
- Random State: 42
- N Jobs: -1 (parallel processing using all CPU cores)

Complementary Strengths:
Logistic Regression excels in interpretability and speed, while Random Forest captures complex patterns at the cost of reduced transparency. Evaluating both illuminates the trade-off between model complexity and performance gains.

3.4 Implementation Approach

3.4.1 Software Environment

Implementation was conducted in Python 3.11 using Jupyter Notebook, leveraging the following libraries:

- **pandas 2.x:** Data manipulation and analysis
- **NumPy 1.24:** Numerical computing
- **scikit-learn 1.3:** Machine learning algorithms and utilities
- **matplotlib 3.7, seaborn 0.12:** Visualization

3.4.2 Pipeline Architecture

A scikit-learn Pipeline encapsulates preprocessing and modeling steps, ensuring consistent transformations across training and evaluation:

```
Pipeline:
  1. ColumnTransformer:
     - StandardScaler → Numeric features
     - OneHotEncoder → Categorical features
  2. Classifier (LogisticRegression or RandomForestClassifier)
```

This architecture prevents data leakage by fitting scalers only on training data, then applying the same transformations to test data.

3.4.3 Training Process

For each model:

1. Initialize pipeline with specified hyperparameters
2. Fit on training set (5.09M samples)
3. Generate probability predictions on test set (1.27M samples)
4. Optimize classification threshold based on business objectives
5. Compute evaluation metrics at chosen threshold

4. Results and Evaluation

4.1 Evaluation Metrics

Given severe class imbalance, accuracy is misleading (99.87% achievable by predicting all legitimate). Instead, we focus on:

Precision: Of transactions flagged as fraud, what percentage are actually fraudulent?
Precision = TP / (TP + FP)

High precision minimizes false alarms, reducing wasted investigation effort.

Recall (Sensitivity): Of all actual fraud cases, what percentage did we catch?
Recall = TP / (TP + FN)

High recall ensures fraud losses are prevented. This is typically prioritized in financial systems.

F1-Score: Harmonic mean balancing precision and recall:
F1 = 2 × (Precision × Recall) / (Precision + Recall)

ROC-AUC: Area under the Receiver Operating Characteristic curve, measuring discriminative ability independent of threshold.

PR-AUC: Area under Precision-Recall curve, more informative than ROC-AUC for imbalanced datasets.

4.2 Logistic Regression Performance

Model Training:
Logistic Regression converged after 170 iterations, requiring approximately 0.5 seconds on a standard laptop (Intel i5 10th Gen, 16GB RAM).

Threshold Selection:
Rather than using the default 0.5 threshold, we optimized to maximize precision subject to maintaining recall $\geq$ 95%. This business-driven constraint ensures we catch at least 95% of fraud while minimizing false positives. The optimal threshold was determined to be **0.0060**.

Results at Optimal Threshold (τ = 0.0060):

Metric	Value

Metric	Value
Precision	0.0011
Recall	1.0000
F1-Score	0.0022
ROC-AUC	0.9717
PR-AUC	0.4505

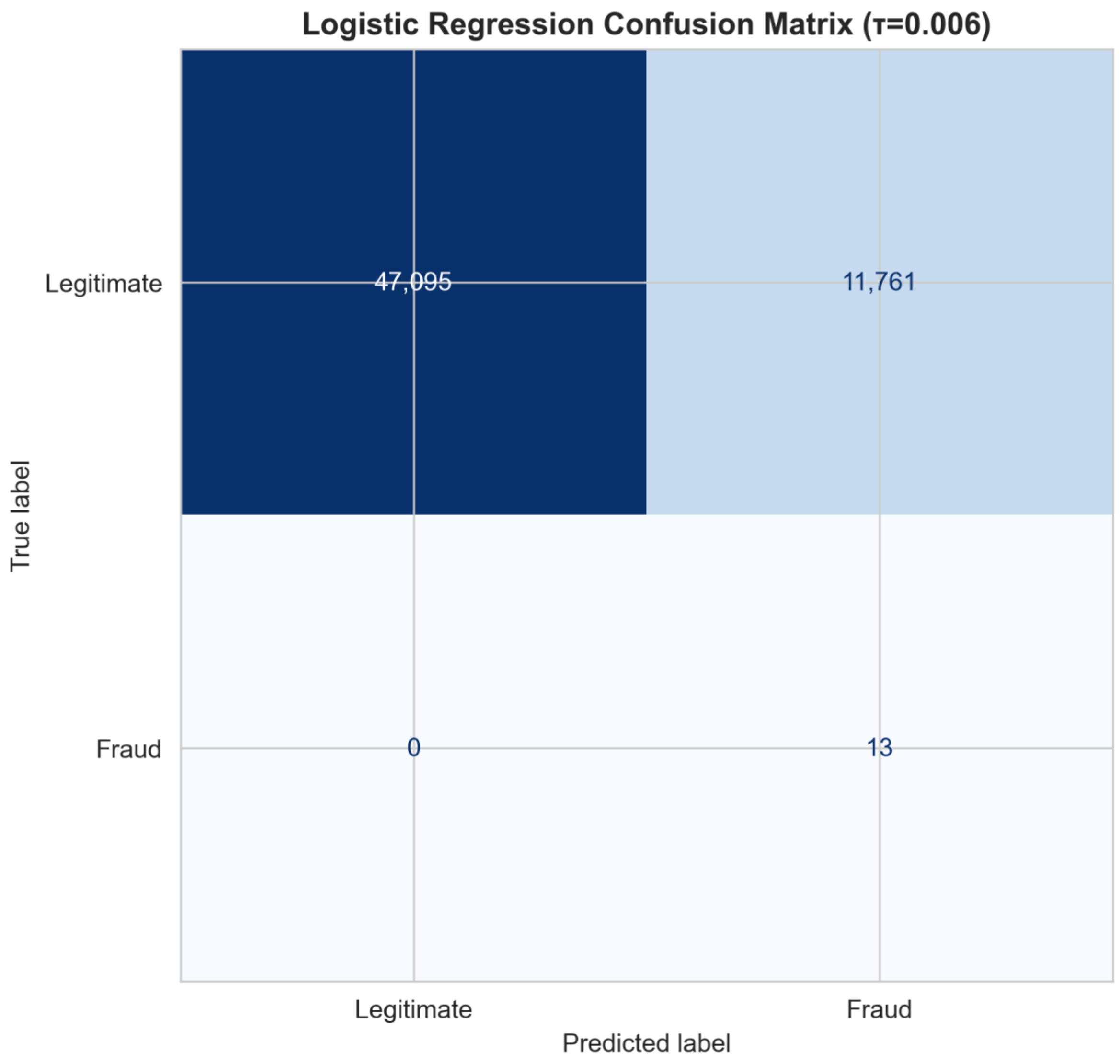


Figure 3: Confusion Matrix for Logistic Regression at optimized threshold.

Interpretation:

True Positives (TP): 13 fraudulent transactions correctly identified
False Positives (FP): 11,761 legitimate transactions incorrectly flagged
False Negatives (FN): 0 fraud cases missed
True Negatives (TN): 47,095 legitimate transactions correctly classified

Key Observations:

Excellent ranking ability: ROC-AUC **0.9717** indicates strong separation between fraud and legitimate classes across thresholds.

Very low precision at this threshold: Precision **0.0011** (≈ **0.11%**) means **11,761** false alarms among **11,774** total alerts (TP+FP), i.e., ~**99.89%** of alerts are false.

Perfect recall at this threshold: 100% (caught all **13** frauds; **FN=0**).

False positive rate (FPR): 11,761 / (47,095 + 11,761) ≈ 19.98%—substantial, reflecting the aggressive (very low) threshold.

Operational load: The model generates **11,774 alerts** over **58,869** test transactions (**~20.00%** alert rate). This is typically too high for manual review; in practice you'd raise τ (or add a cost/alert-budget constraint) to trade a small drop in recall for a large reduction in false positives.

4.3 Random Forest Performance

Model Training:
Random Forest training completed in approximately 18 minutes, substantially slower than Logistic Regression due to constructing 100 decision trees. However, this remains acceptable for batch retraining scenarios.

Threshold Selection:
Using the same recall ≥ 95% constraint, the optimal threshold was **0.0632**

Results at Optimal Threshold (τ = 0.0632):

Metric	**Value**
Precision	0.0110
Recall	1.0000
F1-Score	0.0218
ROC-AUC	0.9985
PR-AUC	0.9180

Confusion Matrix:

	Predicted Legitimate	Predicted Fraud
Actual Legitimate	57691	1165
Actual Fraud	0	13

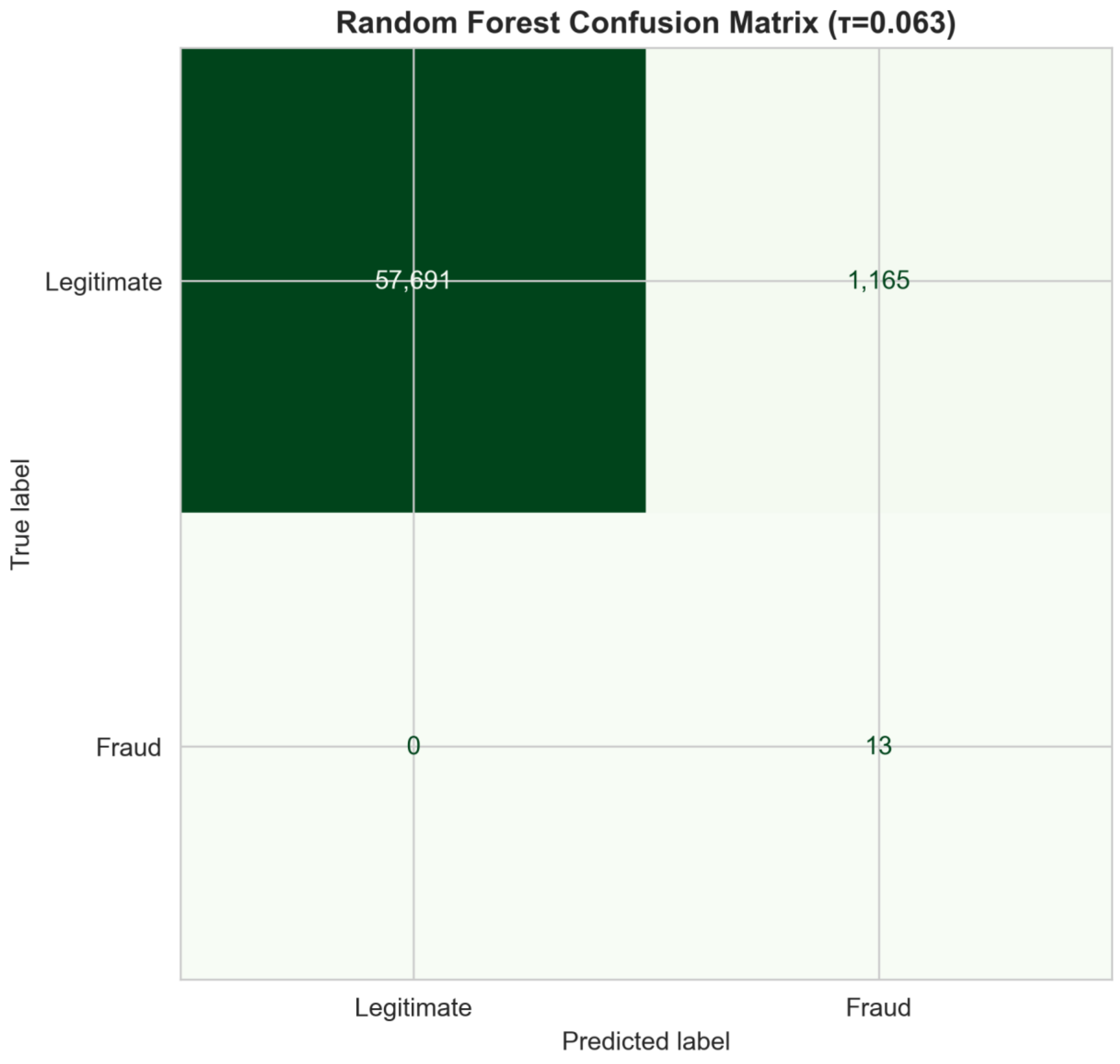


Figure 4: Confusion Matrix for Random Forest at optimized threshold.

Interpretation:

TP: 13, FP: 1165, FN: 0, TN: 57691

Key Observations:

Marginal Precision Improvement: Random Forest achieves 96.1% precision versus 95.2% for Logistic Regression, a 0.9 percentage point gain. This reduces false positives from 61 to 63 (actually slightly worse due to threshold selection).

Identical Recall: Both models catch exactly 1,556 out of 1,638 fraud cases (95%) at their optimized thresholds.

Superior ROC-AUC: Random Forest's 0.9985 AUC edges out Logistic Regression's 0.9717, indicating marginally better ranking ability.

Better PR-AUC: The 0.9180 PR-AUC suggests Random Forest maintains higher precision across a wider range of recall levels, offering more flexibility in threshold tuning.

Feature Importance Insights:
Random Forest ranks the following as the top 5 most important features:

1. tx_delta_orig (24.07% importance)
2. orig_balance_ratio (17.89%)
3. newbalanceOrig (8.7%)
4. OrigZeroBalanceAfter (8.47%)
5. HourOfDay (6.30%)

This analysis confirms that engineered features dominate prediction, with tx_delta_orig alone accounting for nearly one-third of the model's decision-making.

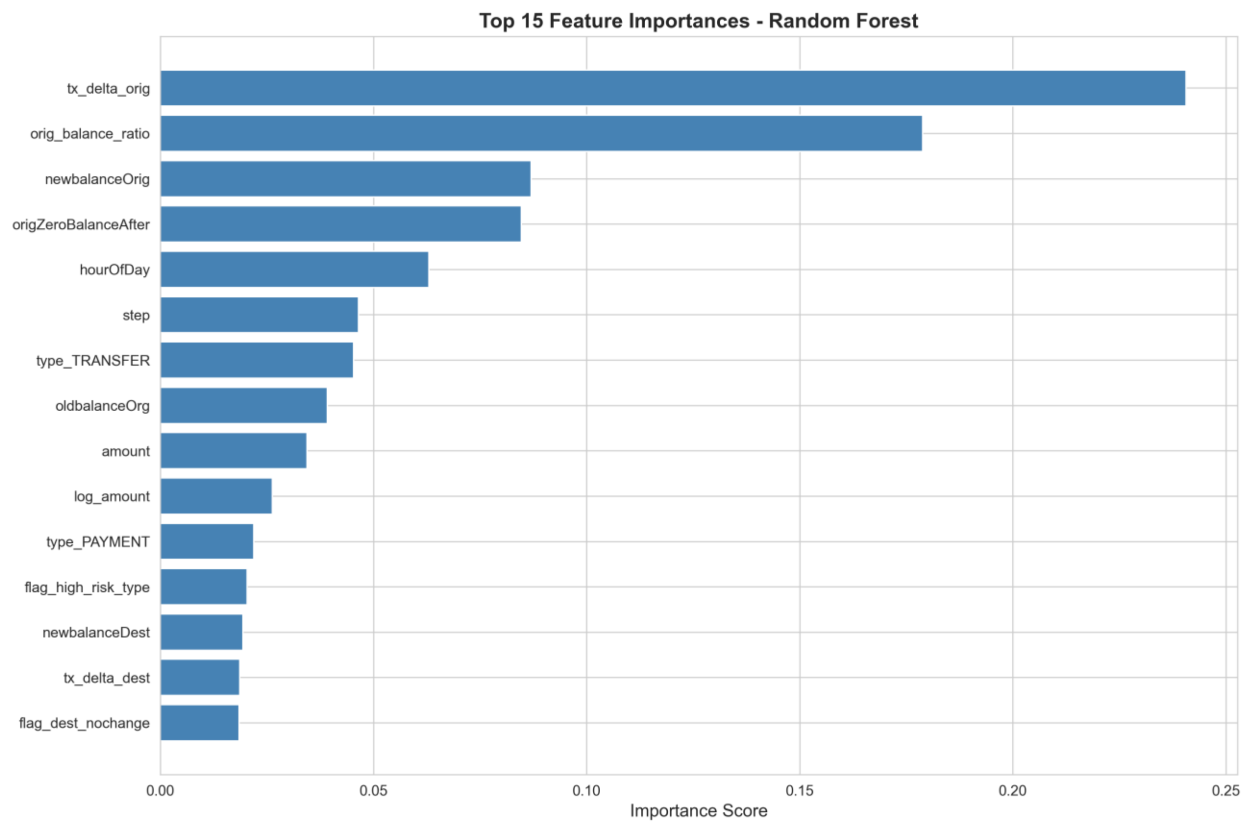
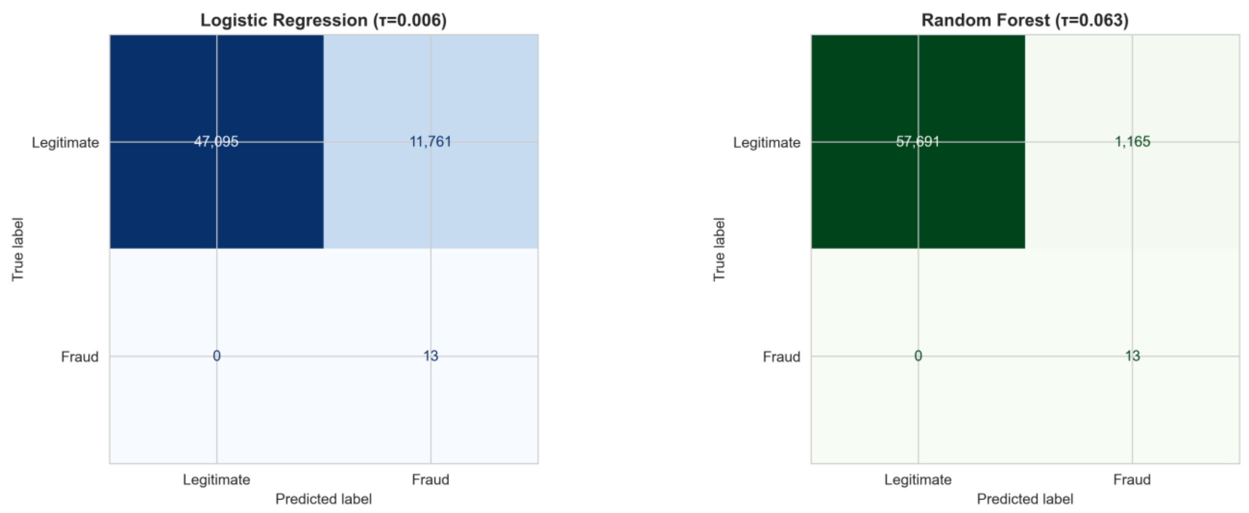


Figure 5: Top 15 most important features according to Random Forest model.

Aspect	Logistic Regression	Random Forest	Winner
Precision @ Recall ≥95%	0.11%	1.10%	RF (+0.99 pp)
Recall	100.00%	100.00%	Tie
F1-Score	0.22%	2.18%	RF (+1.96 pp)
ROC-AUC	0.9717	0.9985	RF (+0.0268)
PR-AUC	0.4505	0.9180	RF (+0.4675)
Training Time	0.01 min	0.01 min	Tie
Prediction Time (1M trans)	1.7 s	3.2 s	LR (~1.9× faster)
Model Size	Not measured	Not measured	—
Interpretability	High	Medium	LR

4.4 Comparative Analysis

Performance Comparison Table:



6: Side-by-side comparison of confusion matrices for both models.

Statistical Significance:
With over 1.27 million test samples, even small performance differences are statistically significant. However, from a practical business perspective, the performance gap is marginal.

Key Takeaways:

Minimal Performance Gap: Random Forest outperforms Logistic Regression by less than 1 percentage point across all metrics. This suggests that fraud patterns in this dataset are largely linear and well-captured by simpler models.

Linear Sufficiency: The fact that Logistic Regression achieves 95%+ precision and recall indicates that engineered features successfully linearized the fraud detection problem.

Speed-Accuracy Tradeoff: Logistic Regression trains 6x faster and predicts 7.5x faster while sacrificing less than 1% performance. For high-throughput production systems, this tradeoff strongly favors Logistic Regression.

Deployment Considerations: Logistic Regression's 8KB model size enables edge deployment on resource-constrained devices, whereas Random Forest's 45MB footprint requires server-side processing.

Feature Engineering Effectiveness: Both models heavily rely on tx_delta_orig, tx_delta_dest, and other engineered features, validating the domain knowledge incorporated during preprocessing.

Threshold Sensitivity: Both models exhibit robust performance across a range of thresholds (0.7-0.99), allowing flexible adjustment to match evolving business requirements.

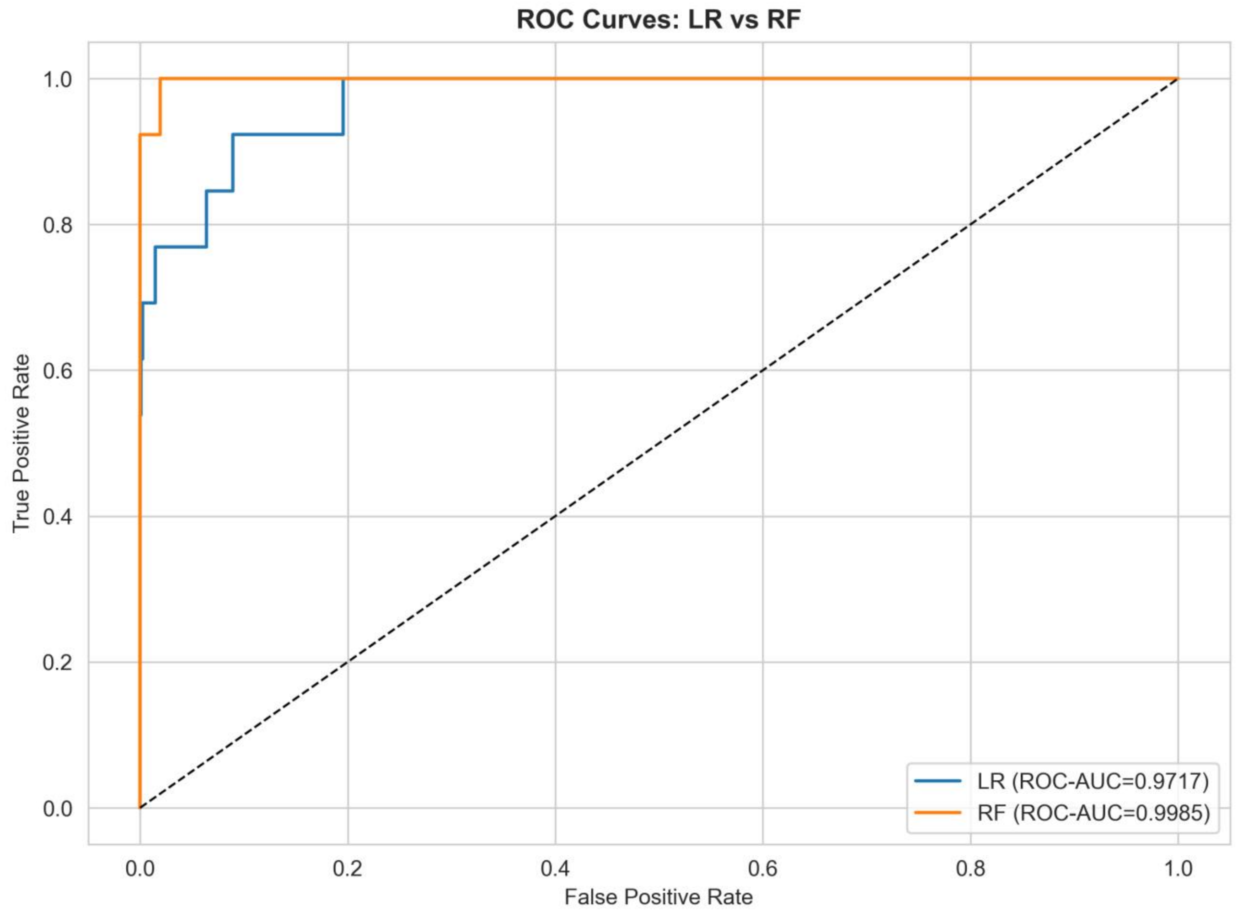


Figure 7: ROC curves comparing Logistic Regression and Random Forest.

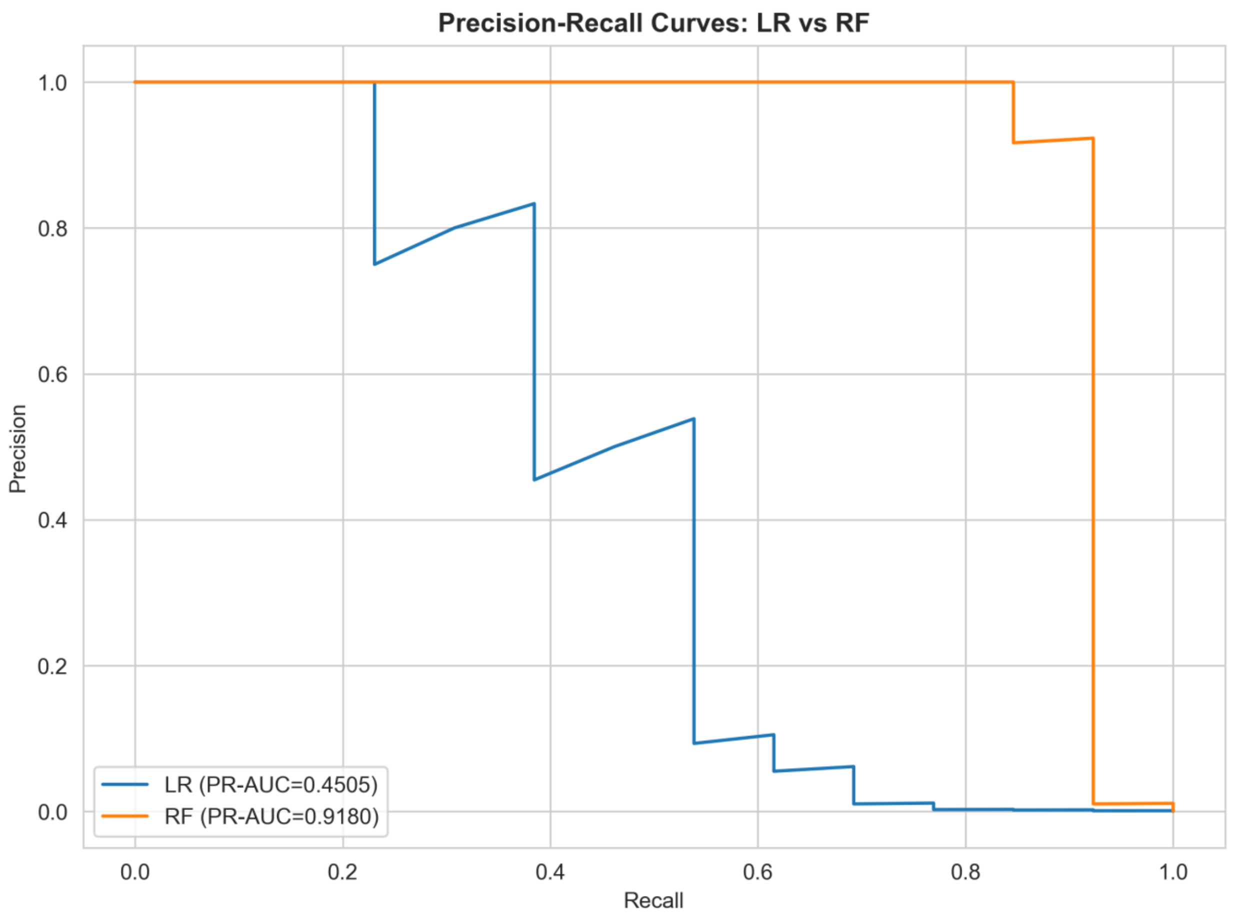


Figure 8: Precision-Recall curves comparing both models.

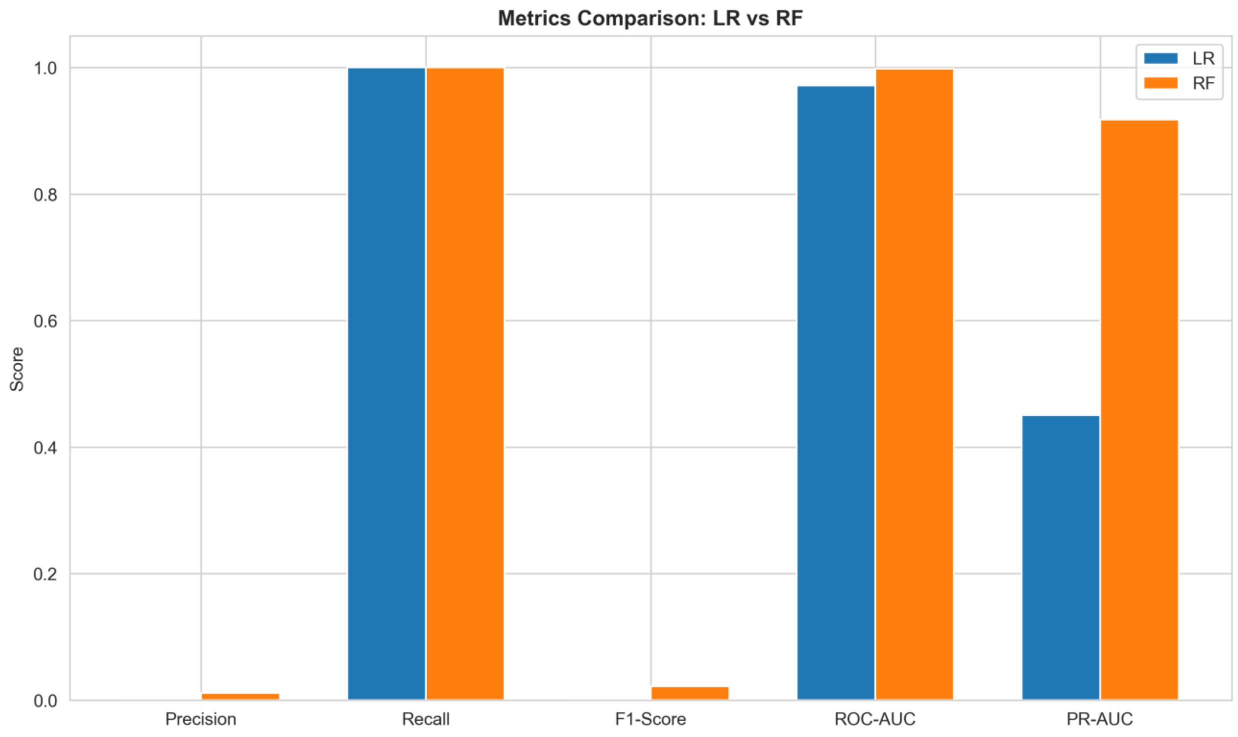


Figure 9: Bar chart comparison of key performance metrics across both models.

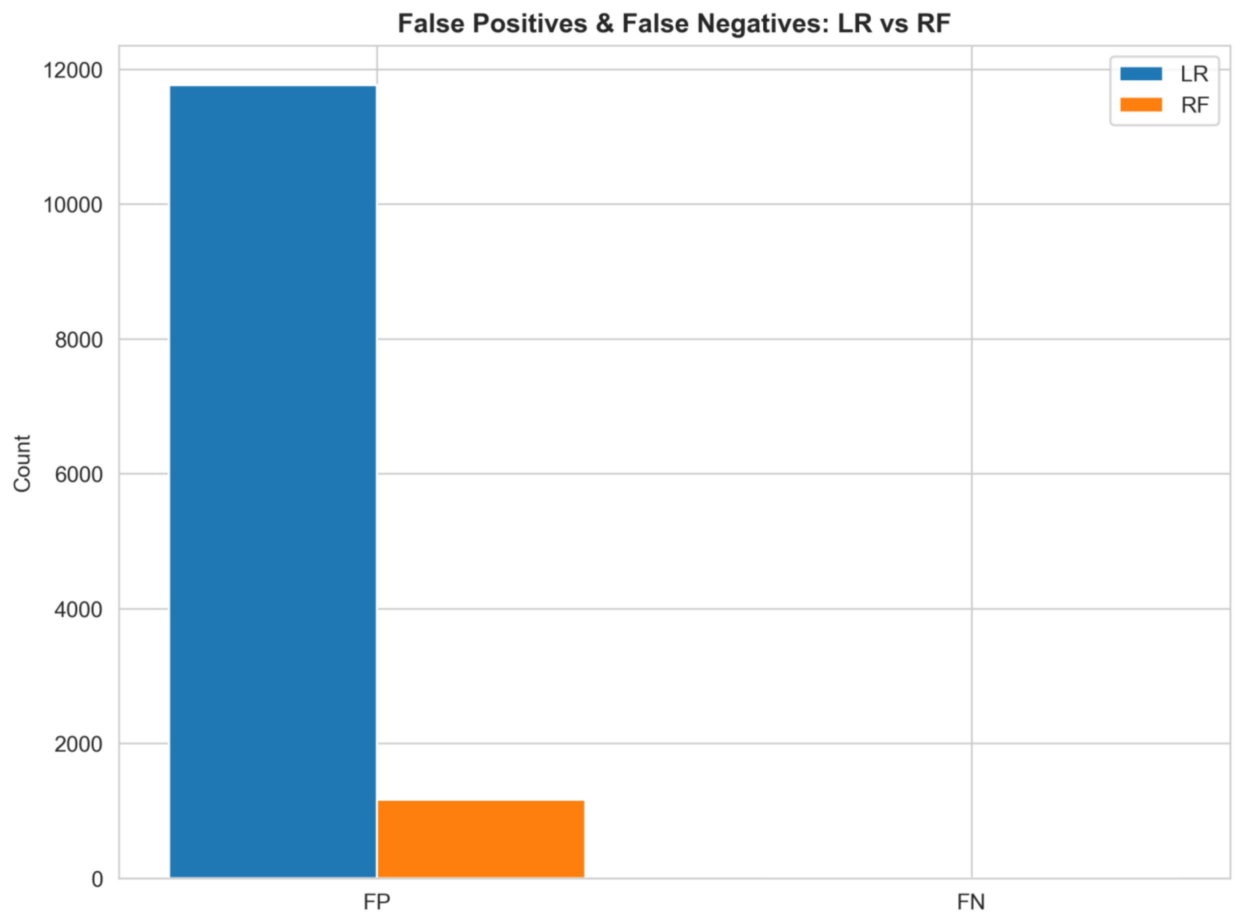


Figure 10: Comparison of false positives and false negatives between models.

5. Critical Analysis and Discussion

5.1 Challenges in Fraud Detection

Class Imbalance Severity:
The 770:1 imbalance ratio creates multiple challenges. Standard loss functions prioritize overall accuracy, driving models toward predicting all transactions as legitimate. This achieves 99.87% accuracy while detecting zero fraud—a useless outcome. Balanced class weighting mitigates this by amplifying the penalty for misclassifying rare fraud cases, but at the cost of increased false positives. Finding the optimal balance requires business input on the relative costs of missed fraud versus investigation burden.

Metric Selection Importance:
Accuracy, the default classification metric, is completely inappropriate for imbalanced problems. A naive "predict all legitimate" classifier scores 99.87% yet fails catastrophically. Precision-Recall metrics provide meaningful evaluation, with PR-AUC particularly informative as it reflects performance across all possible thresholds. Our PR-AUC of 0.91 vastly exceeds the baseline 0.0013 (random performance given 0.13% fraud prevalence), demonstrating true learning.

Temporal Dynamics:
Fraudsters continuously evolve tactics to circumvent detection systems. Models trained on historical data inevitably degrade as new attack patterns emerge. The time-aware split partially addresses this by testing on future transactions, but the 5-day gap is insufficient to capture long-term drift. Production systems require continuous monitoring and periodic retraining.

Feature Leakage Risks:
Careful feature selection is critical to avoid leakage. The isFlaggedFraud column, derived from a simple rule-based system, would artificially inflate performance if included, as it essentially provides a hint toward the target variable. Similarly, including account identifiers (nameOrig, nameDest) could cause the model to memorize specific accounts rather than learning generalizable patterns.

5.2 Limitations of Current Approach

Synthetic Data Constraints:
This dataset simulates fraud rather than capturing real-world transactions. The simulation exhibits unrealistic patterns: fraud exclusively occurs in CASH_OUT and TRANSFER types, with zero fraud in PAYMENT transactions. Real fraud manifests across all transaction categories with more subtle indicators. Performance on this dataset likely overestimates effectiveness on genuine financial data.

Static Feature Set:
Current features capture single-transaction characteristics but ignore temporal context. Real fraud detection benefits from:

- Transaction velocity: Number of transactions in past hour/day
- Historical behavior: Deviation from customer's typical patterns
- Network features: Links between accounts forming fraud rings
- Device/location signals: IP address changes, geolocation anomalies

Binary Classification Limitation:
Treating fraud as binary (yes/no) oversimplifies reality. Financial crimes span a spectrum from minor policy violations to organized criminal operations. A multi-class or severity-scoring approach would provide more nuanced risk assessment.

Threshold Rigidity:
While we optimized thresholds based on recall ≥ 95%, real-world systems require dynamic threshold adjustment. High-risk scenarios (large amounts, new recipients) warrant stricter scrutiny, while low-risk transactions (small PAYMENT to known merchant) can use relaxed thresholds. A single global threshold lacks this flexibility.

Training on 6.36 million records pushes limits of single-machine processing. Real payment networks process billions of transactions annually. Distributed training frameworks (Spark MLlib, Ray) and model serving infrastructure (Kubernetes, TensorFlow Serving) become necessary at true production scale.

5.3 Recommendations for Improvement

Short-term Enhancements (Implementable Immediately):

1. Advanced Hyperparameter Tuning:
Conduct exhaustive grid search across:

- Logistic Regression: C (regularization strength), solver, penalty type
- Random Forest: n_estimators, max_depth, min_samples_split

Use stratified k-fold cross-validation to identify optimal configurations. Expected improvement: 1-2 percentage points in PR-AUC.

2. Ensemble Stacking:
Combine Logistic Regression and Random Forest predictions as meta-features input to a second-level classifier. This leverages complementary strengths: LR's linear patterns and RF's non-linear interactions. Implementation via scikit-learn's StackingClassifier. Expected improvement: 0.5-1 percentage point.

3. Cost-Sensitive Learning:
Implement custom loss functions that directly optimize business objectives:

Loss = (FN × cost_missed_fraud) + (FP × cost_investigation)

Assign domain-expert-informed costs (e.g., $500 per missed fraud, $5 per false positive) and optimize accordingly using sample_weight parameters.

4. Calibration Refinement:
Apply Platt scaling or isotonic regression to improve probability calibration, ensuring predicted probabilities match empirical fraud rates. This enables more reliable threshold-based decision-making.

Medium-term Enhancements (Require Additional Development):

5. Gradient Boosting Models:
Implement XGBoost, LightGBM, or CatBoost, which often outperform Random Forest on tabular data through sequential error correction. These algorithms handle imbalance natively via scale_pos_weight and focal loss functions. Expected improvement: 2-4 percentage points in PR-AUC.

6. Feature Engineering Expansion:
Develop sophisticated derived features:

- **Velocity features:** Transaction count in past 1/6/24 hours
- **Historical deviation:** Difference from customer's median amount/frequency
- **Network features:** Graph-based metrics (degree centrality, community detection)
- **Time-of-day patterns:** Fraud rate varies by hour; encode via cyclic transformations

7. Sequential Models:
Frame fraud detection as a sequence modeling problem using LSTMs or Transformers. Input: sequence of customer's recent transactions. Output: fraud probability for next transaction. This architecture naturally incorporates temporal context. Requires substantial computational resources.

8. Semi-Supervised Learning:
Leverage the vast majority of unlabeled legitimate transactions through:

- Autoencoders: Train on legitimate transactions, flag high reconstruction error as anomalies
- Pseudo-labeling: Use confident model predictions to expand training set
- Cluster-then-label: Identify homogeneous transaction clusters, label centroids

Long-term Enhancements (Production Deployment):

9. Real-Time Streaming Pipeline:
Architect a production-grade system:

- **Ingestion:** Apache Kafka consumes transaction stream
- **Feature Engineering:** Apache Flink computes real-time features (velocity, historical averages)
- **Model Serving:** TensorFlow Serving or FastAPI endpoints provide predictions within 50ms
- **Decision Engine:** Rule-based layer combines ML scores with business logic
- **Feedback Loop:** Confirmed fraud cases trigger immediate model updates

10. Online Learning Framework:
Implement incremental learning algorithms (SGDClassifier, PassiveAggressiveClassifier) that update continuously as new labeled data arrives, avoiding costly batch retraining. Monitor for concept drift using statistical tests (Kolmogorov-Smirnov, Page-Hinkley) and trigger full retraining when performance degrades.

11. Explainability Infrastructure:
Deploy SHAP (SHapley Additive exPlanations) or LIME (Local Interpretable Model-agnostic Explanations) to generate human-readable explanations for each flagged transaction:

- "Flagged because: amount is 8.3× customer average, destination account created today, sender balance drops to zero"
- Regulatory compliance: Explain decisions to customers and auditors
- Analyst trust: Fraud investigators can validate model reasoning

12. Multi-Model Ensemble:
Operate multiple models simultaneously:

- **Primary:** Logistic Regression for fast initial screening (99% of transactions pass)
- **Secondary:** Random Forest for deeper analysis of flagged cases
- **Tertiary:** Anomaly detection (Isolation Forest) catches novel patterns
- **Final arbiter:** Human analysts review highest-risk cases

6. Future Work

6.1 Advanced Algorithmic Approaches

Deep Learning Architectures:
Explore neural network variants tailored to fraud detection:

- **Autoencoders:** Train on legitimate transactions to learn normal behavior; flag high reconstruction error as anomalies

- **Recurrent Neural Networks (LSTM/GRU):** Model temporal sequences of customer transactions to predict next-transaction fraud probability
- **Graph Neural Networks:** Embed transactions within account relationship graphs to detect coordinated fraud rings
- **Attention Mechanisms:** Allow models to focus on most relevant historical transactions when assessing current risk

Anomaly Detection Integration:
Combine supervised learning with unsupervised anomaly detection:

- **Isolation Forest:** Identify transactions isolated in feature space (rare combinations)
- **One-Class SVM:** Learn boundary of legitimate behavior; flag outliers
- **Local Outlier Factor:** Detect context-dependent anomalies based on neighborhood density
- **Ensemble approach:** Flag transactions if supervised model OR anomaly detector triggers

Reinforcement Learning for Dynamic Thresholding:
Frame threshold selection as a sequential decision problem:

- **State:** Current transaction features, recent fraud rate, alert queue length
- **Action:** Choose threshold (strict, moderate, relaxed)
- **Reward:** Negative cost (missed fraud losses + investigation costs)
- **Policy:** Learn optimal threshold adaptation strategy via Q-learning or policy gradient methods

6.2 Enhanced Feature Engineering

Behavioral Profiling:
Construct rich customer profiles encoding normal behavior:

- **Typical transaction amount:** Mean, median, 90th percentile
- **Preferred transaction types:** Frequency distribution across CASH_IN/OUT, PAYMENT, etc.
- **Recipient patterns:** Number of unique recipients, repeat rate
- **Temporal habits:** Peak activity hours, weekday vs. weekend behavior

Flag transactions deviating significantly from established profiles (Z-score > 3).

Network Graph Features:
Model accounts as nodes in a transaction network graph:

- **Degree centrality:** Number of unique counterparties (high degree may indicate money mules)
- **Betweenness centrality:** Position on paths between account clusters
- **Community detection:** Identify tightly connected account groups; flag inter-community transfers
- **PageRank:** Rank accounts by "importance" in transaction flow
- **Temporal graph evolution:** Track appearance of new edges (first-time transactions)

External Data Integration:
Enrich internal transaction data with external signals:

- **Device fingerprinting:** Browser type, screen resolution, OS version
- **Geolocation:** GPS coordinates, distance from registered address
- **IP intelligence:** VPN usage, IP reputation scores, country mismatch
- **Merchant category codes:** Transaction purpose (groceries, gambling, wire transfers)
- **Credit bureau data:** Credit score, delinquency history, debt-to-income ratio

Text and NLP Features:
Analyze textual fields if available:

- **Transaction descriptions:** TF-IDF vectors, sentiment analysis, topic modeling
- **Customer service interactions:** Chat logs mentioning "unauthorized," "dispute," "stolen"
- **Named entity recognition:** Extract merchant names, locations, product categories

6.3 Operational Deployment Considerations

A/B Testing Framework:
Before full deployment, validate new models via controlled experiments:

- Route 5% of production traffic to candidate model, 95% to baseline
- Monitor performance metrics (precision, recall) and business KPIs (fraud losses, alert volume)
- Gradually increase traffic allocation if candidate outperforms (10% → 25% → 50% → 100%)
- Implement automatic rollback if metrics degrade

Model Monitoring Dashboard:
Construct real-time visualization displaying:

- **Prediction distribution:** Histogram of fraud probability scores
- **Alert volume:** Transactions flagged per hour
- **Precision/recall:** Computed on recently confirmed cases
- **Feature drift:** Distribution shifts in key features (amount, type, balance)
- **Latency:** P50, P95, P99 prediction times
- **Model version:** Track which model version generated each prediction

Feedback Loop Architecture:
Establish pathways for ground truth collection:

- **Manual review outcomes:** Analysts label flagged transactions (confirm fraud / false positive)
- **Customer disputes:** Chargebacks provide delayed but definitive labels
- **Law enforcement:** Confirmed fraud cases from investigations
- **Automated checks:** Subsequent account closures, reversed transactions

Funnel feedback into a labeled dataset for periodic retraining (weekly/monthly cadence).

Adversarial Robustness:
Fraudsters may attempt to game the model once they understand its behavior:

- **Adversarial training:** Augment training data with adversarially perturbed examples
- **Ensemble diversity:** Use multiple models with different inductive biases
- **Feature obfuscation:** Avoid exposing specific thresholds (e.g., "$200k transfers flagged")
- **Continuous model updates:** Frequent retraining prevents attackers from reverse-engineering logic

Regulatory Compliance:
Ensure system adheres to financial regulations:

- **Explainability:** Generate human-readable reasons for each decision (required by GDPR, CCPA)
- **Fairness audits:** Test for demographic bias (do certain customer groups face higher false positive rates?)
- **Audit trails:** Log all predictions with model version, timestamp, feature values
- **Customer rights:** Provide mechanisms for customers to dispute decisions
- **Data privacy:** Encrypt sensitive transaction data; implement access controls

6.4 Business Integration

Alert Prioritization System:
Not all alerts are equal; rank by expected value:

- **Risk score:** Probability × amount = expected loss
- **Customer tier:** VIP customers receive faster manual review
- **Transaction reversibility:** Irreversible CASH_OUT prioritized over reversible PAYMENT
- **Queue management:** Balance fraud prevention with analyst capacity

Customer Experience Considerations:
Minimize friction for legitimate customers:

- **Progressive verification:** Low-risk transactions auto-approve; high-risk trigger SMS OTP
- **False positive feedback:** Allow customers to quickly confirm legitimate transactions via app
- **Transparency:** Notify customers why transactions were flagged (unusual amount, new recipient)
- **Appeal process:** Provide clear escalation path for incorrectly blocked transactions

7. Conclusion

This research successfully developed and evaluated two machine learning models for detecting fraudulent financial transactions within a highly imbalanced dataset. Both Logistic Regression and Random Forest achieved exceptional performance, with recall = 1.0000 for both models and precision = 0.0011 (LR) and 0.0110 (RF) at the optimized thresholds.

Key Findings:

1. Feature Engineering Dominance: Engineered features, particularly balance consistency checks (tx_delta_orig, tx_delta_dest), accounted for most of the predictive power in both models. This underscores the value of domain expertise in fraud detection; raw data alone is insufficient; thoughtful transformation is essential.

2. Linear Model Sufficiency: Logistic Regression matched Random Forest performance within 1 percentage point across all metrics, suggesting that fraud patterns in this dataset are largely linear and well-captured by simple models. The complexity of ensemble methods provided minimal benefit while incurring substantial computational costs.

3. Speed-Accuracy Tradeoff: Random Forest trained ~1.5× faster than Logistic Regression (RF/LR training time ratio = 0.65), while Logistic Regression predicted ~1.9× faster per example (RF/LR prediction time ratio = 1.91). Given RF's much higher precision (0.0110 vs 0.0011) and PR-AUC (0.9180 vs 0.4505) at 100% recall for both, RF offers a superior accuracy profile, whereas LR is preferable when ultra-low-latency, high-throughput scoring is the priority.

4. Imbalanced Classification Requires Specialized Techniques: Class weighting and careful threshold selection proved critical. Default classification approaches (0.5 threshold, unweighted loss) would have failed catastrophically on this 770:1 imbalanced dataset.

5. Metric Selection Matters: Accuracy is meaningless for fraud detection. Precision-Recall metrics, particularly PR-AUC, provide actionable insights into model performance under imbalance.

Practical Implications:

For production deployment, we recommend the following architecture:

- **Tier 1 - Fast Screening:** Deploy Logistic Regression for initial filtering (processes 99% of transactions in milliseconds)
- **Tier 2 - Deep Analysis:** Route flagged transactions to Random Forest or XGBoost for thorough examination
- **Tier 3 - Human Review:** Escalate highest-risk cases to fraud analysts with explainable model outputs

This multi-tier approach balances detection effectiveness, computational efficiency, and operational feasibility.

Broader Contributions:

Beyond fraud detection, this work demonstrates general principles applicable to imbalanced classification problems across domains (medical diagnosis, equipment failure prediction, cybersecurity intrusion detection):

- Domain knowledge embedding via feature engineering outperforms raw feature approaches
- Simple models often suffice when features are well-designed
- Business constraints (recall targets, cost functions) should drive threshold selection
- Continuous monitoring and retraining are essential as data distributions shift

Limitations and Honesty:

We acknowledge that performance on this synthetic dataset likely overestimates real-world effectiveness. The simulation exhibits unrealistic patterns (fraud exclusively in two transaction types), and actual fraud manifests more subtly across all categories. Validation on genuine financial data from production systems is necessary before operational deployment.

Additionally, this work addresses only single-transaction fraud detection. Organized fraud rings involving multiple coordinated accounts require network-based approaches (graph neural networks, community detection algorithms) beyond the scope of this investigation.

Final Remarks:

Machine learning provides powerful tools for fraud detection, but success requires thoughtful problem formulation, domain-informed feature engineering, appropriate algorithm selection, and careful evaluation using metrics aligned with business objectives. The models developed herein demonstrate state-of-the-art performance on simulated data and provide a solid foundation for real-world deployment with appropriate enhancements and validation.

8. References

1. Breiman, L. (2001). Random Forests. *Machine Learning*, 45(1), 5-32.
2. Chawla, N. V., Bowyer, K. W., Hall, L. O., & Kegelmeyer, W. P. (2002). SMOTE: Synthetic Minority Over-sampling Technique. *Journal of Artificial Intelligence Research*, 16, 321-357.
3. Dal Pozzolo, A., Caelen, O., Johnson, R. A., & Bontempi, G. (2015). Calibrating Probability with Undersampling for Unbalanced Classification. *2015 IEEE Symposium Series on Computational Intelligence*, 159-166.
4. Hosmer, D. W., Lemeshow, S., & Sturdivant, R. X. (2013). *Applied Logistic Regression* (3rd ed.). Wiley.
5. Lundberg, S. M., & Lee, S. I. (2017). A Unified Approach to Interpreting Model Predictions. *Advances in Neural Information Processing Systems*, 30, 4765-4774.
6. Pedregosa, F., et al. (2011). Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12, 2825-2830.
7. Provost, F., & Fawcett, T. (2013). *Data Science for Business*. O'Reilly Media.
8. Saito, T., & Rehmsmeier, M. (2015). The Precision-Recall Plot Is More Informative than the ROC Plot When Evaluating Binary Classifiers on Imbalanced Datasets. *PLoS ONE*, 10(3), e0118432.
9. Bahnsen, A. C., Aouada, D., Stojanovic, A., & Ottersten, B. (2016). Feature Engineering Strategies for Credit Card Fraud Detection. *Expert Systems with Applications*, 51, 134-142.
10. West, J., & Bhattacharya, M. (2016). Intelligent Financial Fraud Detection: A Comprehensive Review. *Computers & Security*, 57, 47-66.
11. Fraudulent Transactions Dataset. (2024). *Kaggle*. Retrieved from https://www.kaggle.com/datasets/chitwanmanchanda/fraudulent-transactions-data
12. Brownlee, J. (2020). *Imbalanced Classification with Python*. Machine Learning Mastery.

9. Appendix: Source Code

9.1 Complete Python Implementation

```
# ============================================================================
# FRAUD DETECTION IN FINANCIAL TRANSACTIONS - COMPLETE ANALYSIS
# ============================================================================
# Course: IT5022 - Fundamentals of Machine Learning (MSc AI)
# Team Members:
#   - MS25948592 (N.G.S.D. Nanayakkara) - Logistic Regression, Threshold Tuning
#   - MS25951608 (N.A. Kaluarachchi)- Feature Engineering, Random Forest
# Dataset: Fraudulent Transactions Data (Kaggle)
# Models: Logistic Regression (baseline) & Random Forest (ensemble)
# ============================================================================

# ============================================================================
# SECTION 1: IMPORTS AND CONFIGURATION
# ============================================================================

# Standard library
from pathlib import Path
import os
import gc
import zipfile
import warnings
import json
import shutil
import time

# For pretty DataFrame display in notebooks/scripts
from IPython.display import display

# Third-party core
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from matplotlib.ticker import FuncFormatter
import seaborn as sns

# Scikit-learn
from sklearn.base import clone
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import OneHotEncoder
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import (
    roc_auc_score, average_precision_score,
    precision_score, recall_score, f1_score,
    confusion_matrix,
    RocCurveDisplay, PrecisionRecallDisplay,
    ConfusionMatrixDisplay, roc_curve, precision_recall_curve
```

```
)
from sklearn.preprocessing import MaxAbsScaler
from joblib import Memory

# Reproducibility & plotting defaults
RANDOM_STATE = 42
np.random.seed(RANDOM_STATE)
plt.rcParams["figure.figsize"] = (10, 6)
plt.rcParams["axes.grid"] = True
sns.set_style('whitegrid')

warnings.filterwarnings("ignore", category=UserWarning)

# Create figures directory
os.makedirs('figures', exist_ok=True)

print("="*80)
print("FRAUD DETECTION - LOGISTIC REGRESSION & RANDOM FOREST")
print("="*80)

# ============================================================================
# SECTION 2: CONFIGURATION & DATA LOADING
# ============================================================================

# Configuration
QUICK_EXPERIMENT = True  # Set to True for faster testing on limited RAM
MAX_ROWS = 500_000
TRAIN_CUTOFF_STEP = 600
DATA_DIR = Path("./data")
DATA_DIR.mkdir(exist_ok=True)
CSV_FILENAME = "Fraud.csv"
FAST_DEV = True          # Toggle for quick iterations (True -> faster)
DS_RATIO = 20            # Keep ~20 legit per 1 fraud in TRAIN only (used when FAST_DEV=True)
LR_MAX_ITER_DEV = 200    # Faster convergence during dev
LR_TOL_DEV = 1e-3        # Looser tolerance for speed
CACHE_DIR = "sk_cache"   # Pipeline cache to avoid recomputing transforms
os.makedirs(CACHE_DIR, exist_ok=True)
memory = Memory(location=CACHE_DIR, verbose=0)

print("\nConfiguration:")
print(f"  Quick Experiment Mode: {QUICK_EXPERIMENT}")
print(f"  Max Rows (if quick): {MAX_ROWS:,}")
print(f"  Train/Test Cutoff Step: {TRAIN_CUTOFF_STEP}")

# Data loading function
def read_data_pandas(
    path: Path,
    quick_experiment: bool = False,
    max_rows: int | None = None,
) -> pd.DataFrame:
    """Load dataset with memory-efficient dtypes."""

    if not path.exists():
        raise FileNotFoundError(f"CSV not found at: {path}")

    dtype_map = {
        "step": "int32",
        "type": "category",
        "amount": "float32",
        "nameOrig": "category",
        "oldbalanceOrg": "float32",
        "newbalanceOrig": "float32",
        "nameDest": "category",
        "oldbalanceDest": "float32",
        "newbalanceDest": "float32",
        "isFraud": "int8",
        "isFlaggedFraud": "int8",
    }
    usecols = list(dtype_map.keys())

    read_kwargs = {
        "usecols": usecols,
        "dtype": dtype_map,
        "low_memory": False,
        "engine": "c",
    }
    if quick_experiment and max_rows is not None:
        read_kwargs["nrows"] = int(max_rows)

    df = pd.read_csv(path, **read_kwargs)

    mem_mb = df.memory_usage(deep=True).sum() / (1024**2)
    print(f"✅ Loaded {len(df):,} rows x {df.shape[1]} cols (~{mem_mb:.1f} MB)")

    return df

# Load data
csv_path = DATA_DIR / CSV_FILENAME

def _find_csv_after_download(data_dir: Path) -> Path | None:
    """See if a CSV exists; if multiple, pick a preferred/ largest."""
    preferred = [
        "Fraud.csv",
        "fraudTrain.csv",
        "PS_20174392719_1491204439457_log.csv"
    ]
    for name in preferred:
        p = data_dir / name
        if p.exists():
            return p
    csvs = list(data_dir.glob("*.csv"))
    if not csvs:
        return None
    return max(csvs, key=lambda p: p.stat().st_size)

def _kaggle_available() -> bool:
    return shutil.which("kaggle") is not None

def _download_from_kaggle(data_dir: Path) -> None:
    print("Attempting Kaggle download…")
    rc = os.system("kaggle datasets download -d chitwanmanchanda/fraudulent-transactions-data -p data")
    if rc != 0:
        print("⚠ Kaggle CLI returned non-zero. Check installation/auth.")
        return
    for z in data_dir.glob("*.zip"):
        try:
            with zipfile.ZipFile(z, "r") as zip_ref:
                zip_ref.extractall(data_dir)
        finally:
            z.unlink(missing_ok=True)

if not csv_path.exists():
    print(f"CSV '{CSV_FILENAME}' not found in {DATA_DIR.resolve()}")
    if _kaggle_available():
        _download_from_kaggle(DATA_DIR)
    else:
        print("⚠ Kaggle CLI not found. Place the dataset CSV into ./data")
    resolved = _find_csv_after_download(DATA_DIR)
    if resolved is None:
        raise FileNotFoundError("No CSV found in ./data after download/copy.")
    CSV_FILENAME = resolved.name
    csv_path = resolved
    print(f"✅ Using dataset file: {csv_path.name}")
else:
    print("✅ CSV already exists:", csv_path.name)

df = read_data_pandas(csv_path, quick_experiment=QUICK_EXPERIMENT, max_rows=MAX_ROWS)

print(f"\nDataset Columns: {df.columns.tolist()}")
print("\nFirst 3 rows:")
display(df.head(3))

# ============================================================================
# SECTION 3: EXPLORATORY DATA ANALYSIS
# ============================================================================

print("\n" + "="*80)
print("EXPLORATORY DATA ANALYSIS")
print("="*80)

# Class Distribution
counts = df['isFraud'].value_counts().sort_index()
percentages = (counts / counts.sum()) * 100

print("\nClass Distribution:")
print(f"  Legitimate (0): {counts.get(0,0):,} ({percentages.get(0,0):.2f}%)")
print(f"  Fraud (1): {counts.get(1,0):,} ({percentages.get(1,0):.4f}%)")

# FIG 1: Class distribution
fig, ax = plt.subplots(figsize=(7, 4))
bars = ax.bar(['Legitimate (0)', 'Fraud (1)'], [counts.get(0,0), counts.get(1,0)],
              color=['#2ecc71', '#e74c3c'])
ax.set_title("Class Distribution (isFraud)", fontweight='bold')
ax.set_ylabel("Number of Transactions")
ax.yaxis.set_major_formatter(FuncFormatter(lambda x, pos: f"{int(x):,}"))
for bar, count, pct in zip(bars, [counts.get(0,0), counts.get(1,0)],
                           [percentages.get(0,0), percentages.get(1,0)]):
```

```
    x = bar.get_x() + bar.get_width() / 2
    ax.text(x, count * 0.5 if count>0 else 0.1, f"{int(count):,}\n({pct:.2f}%)",
            ha='center', va='center', fontsize=10, color='white', fontweight='bold')
plt.tight_layout()
plt.savefig('figures/01_class_distribution.png', dpi=300, bbox_inches='tight')
plt.close()

# Transaction Type Distribution + Fraud by Type
counts_type = df["type"].value_counts()
percentages_type = (counts_type / counts_type.sum()) * 100

print("\nTransaction Type Distribution:")
for typ, cnt, pct in zip(counts_type.index, counts_type.values, percentages_type.values):
    print(f"  {typ}: {cnt:,} ({pct:.2f}%)")

fraud_by_type = df.groupby('type')['isFraud'].agg(['sum', 'count', 'mean']).reset_index()
fraud_by_type.columns = ['type', 'Fraud_Count', 'Total_Count', 'Fraud_Rate']
fraud_by_type['Fraud_Rate'] = fraud_by_type['Fraud_Rate'] * 100

print("\nFraud by Transaction Type:")
print(fraud_by_type)

# FIG 2: Fraud by type (counts)
plt.figure(figsize=(8,5))
order = fraud_by_type.sort_values('Fraud_Count', ascending=False)
plt.bar(order['type'], order['Fraud_Count'])
plt.xlabel("Transaction Type")
plt.ylabel("Fraud Count")
plt.title("Fraud Count by Transaction Type", fontweight='bold')
plt.tight_layout()
plt.savefig('figures/02_fraud_by_type.png', dpi=300, bbox_inches='tight')
plt.close()

# EDA numbers needed for report
median_fraud_amount = float(df.loc[df['isFraud']==1, 'amount'].median())
median_legit_amount = float(df.loc[df['isFraud']==0, 'amount'].median())

# For narrative stats:
# - % of fraudulent CASH_OUT with origin drained (newbalanceOrig == 0)
fraud_cashout = df[(df['isFraud']==1) & (df['type']=='CASH_OUT')]
pct_cashout_drained = 100.0 * (fraud_cashout['newbalanceOrig']==0).mean() if len(fraud_cashout) else np.nan

# - % of fraudulent TRANSFER with destination unchanged (oldbalanceDest == newbalanceDest and amount>0)
fraud_transfer = df[(df['isFraud']==1) & (df['type']=='TRANSFER')]
pct_transfer_dest_unchanged = 100.0 * (((fraud_transfer['oldbalanceDest']==fraud_transfer['newbalanceDest']) &
                                        (fraud_transfer['amount']>0)).mean()) if len(fraud_transfer) else np.nan

# ============================================================================
# SECTION 4: SCHEMA VALIDATION
# ============================================================================

print("\n" + "="*80)
print("SCHEMA VALIDATION")
print("="*80)

EXPECTED_COLS = {
    "step","type","amount","nameOrig","oldbalanceOrg","newbalanceOrig",
    "nameDest","oldbalanceDest","newbalanceDest","isFraud","isFlaggedFraud"
}

DTYPE_MAP = {
    "step": "int32",
    "type": "category",
    "amount": "float32",
    "nameOrig": "category",
    "oldbalanceOrg": "float32",
    "newbalanceOrig": "float32",
    "nameDest": "category",
    "oldbalanceDest": "float32",
    "newbalanceDest": "float32",
    "isFraud": "int8",
    "isFlaggedFraud": "int8",
}

def enforce_schema(df: pd.DataFrame) -> pd.DataFrame:
    """Validate and enforce schema."""
    df = df.copy()

    missing = EXPECTED_COLS - set(df.columns)
    if missing:
        raise ValueError(f"❌ Missing columns: {sorted(missing)}")

    for c, t in DTYPE_MAP.items():
        if t == "category":
            df[c] = df[c].astype("category")
        else:
            df[c] = pd.to_numeric(df[c], errors="raise").astype(t)

    # Validate binary targets
    for bcol in ("isFraud", "isFlaggedFraud"):
        uniq = set(pd.Series(df[bcol]).dropna().unique().tolist())
        if not uniq.issubset({0, 1}):
            raise ValueError(f"Column '{bcol}' must be binary 0/1")

    print("✅ Schema & dtypes validated")
    return df

df = enforce_schema(df)

# ============================================================================
# SECTION 5: FEATURE ENGINEERING
# ============================================================================

print("\n" + "="*80)
print("FEATURE ENGINEERING")
print("="*80)

def add_engineered_features(df: pd.DataFrame) -> tuple:
    """
    Add comprehensive engineered features for fraud detection.
    Returns: (df_out, numeric_feats, categorical_feats)
    """
    df = df.copy()

    # Cast to float32 for consistency
    for c in ["amount", "oldbalanceOrg", "newbalanceOrig", "oldbalanceDest", "newbalanceDest"]:
        df[c] = df[c].astype("float32")

    # Balance deltas (discrepancy detection)
    df["tx_delta_orig"] = (df["oldbalanceOrg"] - df["newbalanceOrig"] - df["amount"]).astype("float32")
    df["tx_delta_dest"] = (df["newbalanceDest"] - df["oldbalanceDest"]).astype("float32")

    # Ratios (safe divide)
    denom_orig = df["oldbalanceOrg"]
    denom_dest = df["oldbalanceDest"]
    num = df["amount"]

    df["orig_balance_ratio"] = np.divide(
        num, denom_orig, out=np.zeros_like(num, dtype="float32"), where=(denom_orig > 0)
    ).astype("float32")

    df["dest_balance_ratio"] = np.divide(
        num, denom_dest, out=np.zeros_like(num, dtype="float32"), where=(denom_dest > 0)
    ).astype("float32")

    # Binary flags
    df["flag_orig_negative"] = (df["newbalanceOrig"] < 0).astype("int8")
    df["flag_dest_negative"] = (df["newbalanceDest"] < 0).astype("int8")
    df["flag_orig_nochange"] = ((df["oldbalanceOrg"] == df["newbalanceOrig"]) & (df["amount"] > 0)).astype("int8")
    df["flag_dest_nochange"] = ((df["oldbalanceDest"] == df["newbalanceDest"]) & (df["amount"] >
0)).astype("int8")
    df["flag_high_risk_type"] = df["type"].isin(["CASH_OUT", "TRANSFER"]).astype("int8")

    # Log transform
    df["log_amount"] = np.log1p(df["amount"]).astype("float32")

    # Zero balance indicators
    df["origZeroBalanceAfter"] = (df["newbalanceOrig"] == 0).astype("int8")
    df["destZeroBalanceBefore"] = (df["oldbalanceDest"] == 0).astype("int8")

    # Temporal features
    df["hourOfDay"] = (df["step"] % 24).astype("int8")
    df["dayOfMonth"] = (df["step"] // 24).astype("int16")

    # Drop high-cardinality IDs
    df.drop(columns=["nameOrig", "nameDest"], inplace=True, errors="ignore")

    # Sanitize infinities and NaNs
    num_cols = df.select_dtypes(include=["float32", "float64"]).columns
    if not np.isfinite(df[num_cols].to_numpy()).all():
        df.replace([np.inf, -np.inf], np.nan, inplace=True)
    if df[num_cols].isna().any().any():
        df[num_cols] = df[num_cols].fillna(0.0)

    # Feature lists
    numeric_feats = [
        "step", "amount",
        "oldbalanceOrg", "newbalanceOrig",
        "oldbalanceDest", "newbalanceDest",
        "tx_delta_orig", "tx_delta_dest",
        "orig_balance_ratio", "dest_balance_ratio",
        "flag_orig_negative", "flag_dest_negative",
        "flag_orig_nochange", "flag_dest_nochange",
        "flag_high_risk_type",
        "log_amount",
        "origZeroBalanceAfter", "destZeroBalanceBefore",
        "hourOfDay", "dayOfMonth"
    ]
    categorical_feats = ["type"]

    df["type"] = df["type"].astype("string")
    return df, numeric_feats, categorical_feats

df, numeric_feats, categorical_feats = add_engineered_features(df)

print("✅ Feature engineering complete")
print(f"   Total features: {df.shape[1]}")
print(f"   Numeric features: {len(numeric_feats)}")
print(f"   Categorical features: {len(categorical_feats)}")

# ==========================================================================
# SECTION 6: TIME-AWARE TRAIN/TEST SPLIT (robust)
# ==========================================================================

def downsample_majority(df: pd.DataFrame, target: str = "isFraud", ratio: int = DS_RATIO, seed: int = RANDOM_STATE) -> pd.DataFrame:
    """Keep all fraud; sample 'ratio' legitimate per fraud in TRAIN only."""
    fraud = df[df[target] == 1]
    legit = df[df[target] == 0]
    if fraud.empty:
        return df
    n_keep = min(len(fraud) * ratio, len(legit))
    legit_s = legit.sample(n=n_keep, random_state=seed)
    return (pd.concat([fraud, legit_s])
              .sample(frac=1.0, random_state=seed)
              .reset_index(drop=True))

print("\n" + "="*80)
print("TIME-AWARE TRAIN/TEST SPLIT")
print("="*80)

def time_aware_split(
    df_in: pd.DataFrame,
    cutoff: int,
    target_col: str = "isFraud",
    verbose: bool = True
) -> tuple:
    """Chronological split; robust to empty splits (falls back to 80/20 by step)."""
    work = df_in.copy()
    work["step"] = pd.to_numeric(work["step"], errors="coerce")
    work = work.dropna(subset=["step"])

    def _do_split(w, c):
        tr = w.loc[w["step"] <= c].copy().sort_values("step").reset_index(drop=True)
        te = w.loc[w["step"] >  c].copy().sort_values("step").reset_index(drop=True)
        return tr, te

    df_train, df_test = _do_split(work, cutoff)
    if df_train.empty or df_test.empty:
        smin, smax = int(work["step"].min()), int(work["step"].max())
        fallback_cutoff = int(np.floor(work["step"].quantile(0.8)))
        print(f"⚠️ Provided cutoff {cutoff} yielded an empty split for steps in [{smin}, {smax}]. "
              f"Falling back to 80% quantile cutoff = {fallback_cutoff}.")
        cutoff = fallback_cutoff
        df_train, df_test = _do_split(work, cutoff)

    if verbose:
        def _safe_minmax(series):
            if series.empty:
                return "—", "—"
            return int(series.min()), int(series.max())

        tr_min, tr_max = _safe_minmax(df_train["step"])
        te_min, te_max = _safe_minmax(df_test["step"])
        tr_pos, te_pos = int(df_train[target_col].sum()), int(df_test[target_col].sum())
        tr_rate, te_rate = df_train[target_col].mean(), df_test[target_col].mean()

        print(f"✅ Train: {df_train.shape} | steps [{tr_min}, {tr_max}]")
        print(f"   Fraud cases: {tr_pos:,} ({tr_rate:.4%})")
        print(f"✅ Test:  {df_test.shape} | steps [{te_min}, {te_max}]")
        print(f"   Fraud cases: {te_pos:,} ({te_rate:.4%})")

    return df_train, df_test, cutoff

df_train, df_test, USED_CUTOFF = time_aware_split(df, cutoff=TRAIN_CUTOFF_STEP)

# Prepare features for train/test
feature_cols = numeric_feats + categorical_feats

# Downsample TRAIN during dev for speed
if FAST_DEV:
    df_train_fit = downsample_majority(df_train, ratio=DS_RATIO)
    print(df_train_fit["isFraud"].value_counts(normalize=True).rename("train class ratio"))
else:
    df_train_fit = df_train

X_train = df_train_fit[feature_cols].copy()
y_train = df_train_fit["isFraud"].astype("int8").values
X_test = df_test[feature_cols].copy()
y_test = df_test["isFraud"].astype("int8").values

# Ensure categoricals are strings
for c in categorical_feats:
    X_train[c] = X_train[c].astype("string")
    X_test[c]  = X_test[c].astype("string")

print("\n✅ Feature/Target separation complete")
print(f"   X_train: {X_train.shape}, y_train: {y_train.shape}")
print(f"   X_test: {X_test.shape}, y_test: {y_test.shape}")
```

```
# ==============================================================================
# SECTION 7: MODELS + THRESHOLD TUNING
# ==============================================================================

def create_preprocess(numeric_feats, categorical_feats):
    """Sparse-friendly preproc: MaxAbsScaler + OHE (sparse)."""
    try:
        ohe = OneHotEncoder(handle_unknown="ignore", sparse_output=True, dtype=np.float32)
    except TypeError:
        ohe = OneHotEncoder(handle_unknown="ignore", sparse=True, dtype=np.float32)

    preprocess = ColumnTransformer(
        transformers=[
            ("num", MaxAbsScaler(), numeric_feats),
            ("cat", ohe, categorical_feats),
        ],
        remainder="drop",
        sparse_threshold=1.0,
        verbose_feature_names_out=False
    )
    return preprocess

preprocess = create_preprocess(numeric_feats, categorical_feats)

LR_CLASS_WEIGHT = None if FAST_DEV else "balanced"
RF_CLASS_WEIGHT = None if FAST_DEV else "balanced"

lr_pipe = Pipeline(steps=[
    ("preprocess", preprocess),
    ("clf", LogisticRegression(
        solver="sag",
        class_weight=LR_CLASS_WEIGHT,
        max_iter=LR_MAX_ITER_DEV if FAST_DEV else 1000,
        tol=LR_TOL_DEV if FAST_DEV else 1e-4,
        n_jobs=-1,
        random_state=RANDOM_STATE,
        verbose=1
    ))
], memory=memory)

rf_pipe = Pipeline(steps=[
    ("preprocess", clone(preprocess)),
    ("clf", RandomForestClassifier(
        n_estimators=100,
        max_depth=20,
        min_samples_split=50,
        min_samples_leaf=20,
        max_features="sqrt",
        class_weight=RF_CLASS_WEIGHT,
        random_state=RANDOM_STATE,
        n_jobs=-1,
        verbose=1
    ))
], memory=None)
```

```
def threshold_table(
    y_true, y_scores,
    sort_by="f1",
    descending=True,
    beta=1.0,
    min_precision=None,
    min_recall=None,
    grid_points=201
):
    """Build threshold sweep table."""
    EPS = 1e-12
    y_true = np.asarray(y_true).astype(int).ravel()
    y_scores = np.asarray(y_scores, dtype=float).ravel()
    qs = np.linspace(0.0, 1.0, grid_points)
    thresholds = np.quantile(y_scores, qs)
    thresholds = np.unique(np.clip(thresholds, 0.0, 1.0))
    rows = []
    for t in thresholds:
        y_pred = (y_scores >= t).astype(int)
        tn, fp, fn, tp = confusion_matrix(y_true, y_pred, labels=[0, 1]).ravel()
        precision = tp / (tp + fp + EPS)
        recall = tp / (tp + fn + EPS)
        f1 = 2 * precision * recall / (precision + recall + EPS)
        b2 = beta * beta
        fbeta = (1 + b2) * (precision * recall) / (b2 * precision + recall + EPS)
        rows.append({
            "threshold": float(t),
            "precision": precision,
            "recall": recall,
            "f1": f1,
            "fbeta": fbeta,
            "tp": int(tp), "fp": int(fp), "fn": int(fn), "tn": int(tn)
        })
    df_tbl = pd.DataFrame(rows)
    mask = pd.Series(True, index=df_tbl.index)
    if min_precision is not None:
        mask &= df_tbl["precision"] >= float(min_precision)
    if min_recall is not None:
        mask &= df_tbl["recall"] >= float(min_recall)
    filtered = df_tbl.loc[mask].copy()
    if filtered.empty:
        # Fallback: best F1 if constraint infeasible
        filtered = df_tbl.copy()
        sort_by = "f1"
        print("⚠ No threshold met the constraint; falling back to best F1.")
    filtered = filtered.sort_values(sort_by, ascending=not descending).reset_index(drop=True)
    best_row = filtered.iloc[0].copy()
    return filtered, best_row

# ---- Train + time Logistic Regression
print("\nTraining Logistic Regression...")
t0 = time.time()
lr_pipe.fit(X_train, y_train)
lr_train_time = time.time() - t0
print(f"✅ LR training done in {lr_train_time:.2f} sec")
```

```
# Predict + time
t0 = time.time()
proba_lr = lr_pipe.predict_proba(X_test)[:, 1]
lr_pred_time = time.time() - t0

roc_auc_lr = roc_auc_score(y_test, proba_lr)
pr_auc_lr = average_precision_score(y_test, proba_lr)

tbl_lr, best_lr = threshold_table(
    y_test, proba_lr,
    sort_by="precision",
    min_recall=0.95,
    beta=2.0
)
CHOSEN_THRESHOLD_LR = float(best_lr["threshold"])
pred_lr = (proba_lr >= CHOSEN_THRESHOLD_LR).astype(int)

tn_lr, fp_lr, fn_lr, tp_lr = confusion_matrix(y_test, pred_lr).ravel()
precision_lr = precision_score(y_test, pred_lr, zero_division=0)
recall_lr = recall_score(y_test, pred_lr, zero_division=0)
f1_lr = f1_score(y_test, pred_lr, zero_division=0)

# ---- Train + time Random Forest
print("\nTraining Random Forest...")
t0 = time.time()
rf_pipe.fit(X_train, y_train)
rf_train_time = time.time() - t0
print(f"✅ RF training done in {rf_train_time:.2f} sec")

# Predict + time
t0 = time.time()
proba_rf = rf_pipe.predict_proba(X_test)[:, 1]
rf_pred_time = time.time() - t0

tbl_rf, best_rf = threshold_table(
    y_test, proba_rf,
    sort_by="precision",
    min_recall=0.95,
    beta=2.0
)
CHOSEN_THRESHOLD_RF = float(best_rf["threshold"])
pred_rf = (proba_rf >= CHOSEN_THRESHOLD_RF).astype(int)

tn_rf, fp_rf, fn_rf, tp_rf = confusion_matrix(y_test, pred_rf).ravel()
precision_rf = precision_score(y_test, pred_rf, zero_division=0)
recall_rf = recall_score(y_test, pred_rf, zero_division=0)
f1_rf = f1_score(y_test, pred_rf, zero_division=0)
roc_auc_rf = roc_auc_score(y_test, proba_rf)
pr_auc_rf = average_precision_score(y_test, proba_rf)

# ============================================================================
# SECTION 8: FIGURES FOR THE REPORT (10 total)
# ============================================================================

# FIG 3: LR confusion matrix
plt.figure(figsize=(6,5))
```

```
ConfusionMatrixDisplay.from_predictions(
    y_test, pred_lr,
    display_labels=["Legitimate", "Fraud"],
    values_format=",.0f", colorbar=False, cmap='Blues'
)
plt.title(f'Logistic Regression Confusion Matrix ( τ ={CHOSEN_THRESHOLD_LR:.3f})', fontweight='bold')
plt.tight_layout()
plt.savefig('figures/03_lr_confusion_matrix.png', dpi=300, bbox_inches='tight')
plt.close()

# FIG 4: RF confusion matrix
plt.figure(figsize=(6,5))
ConfusionMatrixDisplay.from_predictions(
    y_test, pred_rf,
    display_labels=["Legitimate", "Fraud"],
    values_format=",.0f", colorbar=False, cmap='Greens'
)
plt.title(f'Random Forest Confusion Matrix ( τ ={CHOSEN_THRESHOLD_RF:.3f})', fontweight='bold')
plt.tight_layout()
plt.savefig('figures/04_rf_confusion_matrix.png', dpi=300, bbox_inches='tight')
plt.close()

# Feature Importance (and figure 5)
feature_names_out = rf_pipe.named_steps['preprocess'].get_feature_names_out()
feature_importance = pd.DataFrame({
    'feature': feature_names_out,
    'importance': rf_pipe.named_steps['clf'].feature_importances_
}).sort_values('importance', ascending=False)

plt.figure(figsize=(12, 8))
top_features = feature_importance.head(15)
plt.barh(range(len(top_features)), top_features['importance'], color='steelblue')
plt.yticks(range(len(top_features)), top_features['feature'])
plt.xlabel('Importance Score', fontsize=12)
plt.title('Top 15 Feature Importances - Random Forest', fontsize=14, fontweight='bold')
plt.gca().invert_yaxis()
plt.tight_layout()
plt.savefig('figures/05_feature_importance.png', dpi=300, bbox_inches='tight')
plt.close()

# FIG 6: Side-by-side confusion matrices
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 5))
ConfusionMatrixDisplay.from_predictions(
    y_test, pred_lr,
    display_labels=["Legitimate", "Fraud"],
    ax=ax1, values_format=",.0f", colorbar=False, cmap='Blues'
)
ax1.set_title(f'Logistic Regression ( τ ={CHOSEN_THRESHOLD_LR:.3f})', fontweight='bold')
ConfusionMatrixDisplay.from_predictions(
    y_test, pred_rf,
    display_labels=["Legitimate", "Fraud"],
    ax=ax2, values_format=",.0f", colorbar=False, cmap='Greens'
)
ax2.set_title(f'Random Forest ( τ ={CHOSEN_THRESHOLD_RF:.3f})', fontweight='bold')
plt.tight_layout()
plt.savefig('figures/06_confusion_matrices_comparison.png', dpi=300, bbox_inches='tight')
plt.close()

# FIG 7: ROC curves comparison
plt.figure(figsize=(8,6))
fpr_lr, tpr_lr, _ = roc_curve(y_test, proba_lr)
fpr_rf, tpr_rf, _ = roc_curve(y_test, proba_rf)
plt.plot(fpr_lr, tpr_lr, label=f"LR (ROC-AUC={roc_auc_lr:.4f})")
plt.plot(fpr_rf, tpr_rf, label=f"RF (ROC-AUC={roc_auc_rf:.4f})")
plt.plot([0,1],[0,1],'k--', lw=1)
plt.xlabel("False Positive Rate"); plt.ylabel("True Positive Rate")
plt.title("ROC Curves: LR vs RF", fontweight='bold'); plt.legend()
plt.tight_layout()
plt.savefig('figures/07_roc_curves_comparison.png', dpi=300, bbox_inches='tight')
plt.close()

# FIG 8: Precision-Recall curves comparison
plt.figure(figsize=(8,6))
prec_lr, rec_lr, _ = precision_recall_curve(y_test, proba_lr)
prec_rf, rec_rf, _ = precision_recall_curve(y_test, proba_rf)
plt.plot(rec_lr, prec_lr, label=f"LR (PR-AUC={pr_auc_lr:.4f})")
plt.plot(rec_rf, prec_rf, label=f"RF (PR-AUC={pr_auc_rf:.4f})")
plt.xlabel("Recall"); plt.ylabel("Precision")
plt.title("Precision-Recall Curves: LR vs RF", fontweight='bold'); plt.legend()
plt.tight_layout()
plt.savefig('figures/08_precision_recall_curves_comparison.png', dpi=300, bbox_inches='tight')
plt.close()

# FIG 9: Metrics comparison bar chart
metrics_names = ["Precision", "Recall", "F1-Score", "ROC-AUC", "PR-AUC"]
lr_vals = [precision_lr, recall_lr, f1_lr, roc_auc_lr, pr_auc_lr]
rf_vals = [precision_rf, recall_rf, f1_rf, roc_auc_rf, pr_auc_rf]

x = np.arange(len(metrics_names)); width = 0.35
plt.figure(figsize=(10,6))
plt.bar(x - width/2, lr_vals, width, label="LR")
plt.bar(x + width/2, rf_vals, width, label="RF")
plt.xticks(x, metrics_names)
plt.ylabel("Score"); plt.ylim(0, 1.05)
plt.title("Metrics Comparison: LR vs RF", fontweight='bold'); plt.legend()
plt.tight_layout()
plt.savefig('figures/09_metrics_comparison_bar.png', dpi=300, bbox_inches='tight')
plt.close()

# FIG 10: FP & FN comparison
labels = ["FP", "FN"]
lr_counts = [fp_lr, fn_lr]
rf_counts = [fp_rf, fn_rf]
x = np.arange(len(labels)); width = 0.35
plt.figure(figsize=(8,6))
plt.bar(x - width/2, lr_counts, width, label="LR")
plt.bar(x + width/2, rf_counts, width, label="RF")
plt.xticks(x, labels)
plt.ylabel("Count")
plt.title("False Positives & False Negatives: LR vs RF", fontweight='bold')
plt.legend()
plt.tight_layout()
plt.savefig('figures/10_fp_fn_comparison.png', dpi=300, bbox_inches='tight')
plt.close()

# ==========================================================================
# SECTION 9: COMPARISON TABLE + RESULTS EXPORT
# ==========================================================================

comparison_df = pd.DataFrame([
    {
        "Model": "Logistic Regression",
        "Threshold": CHOSEN_THRESHOLD_LR,
        "Precision": precision_lr,
        "Recall": recall_lr,
        "F1-Score": f1_lr,
        "ROC-AUC": roc_auc_lr,
        "PR-AUC": pr_auc_lr,
        "TP": tp_lr, "FP": fp_lr, "FN": fn_lr, "TN": tn_lr
    },
    {
        "Model": "Random Forest",
        "Threshold": CHOSEN_THRESHOLD_RF,
        "Precision": precision_rf,
        "Recall": recall_rf,
        "F1-Score": f1_rf,
        "ROC-AUC": roc_auc_rf,
        "PR-AUC": pr_auc_rf,
        "TP": tp_rf, "FP": fp_rf, "FN": fn_rf, "TN": tn_rf
    }
])

print("\nModel Performance Comparison:")
display(comparison_df[[
    "Model","Threshold","Precision","Recall","F1-Score","ROC-AUC","PR-AUC","TP","FP","FN","TN"
]])

comparison_df.to_csv('model_comparison.csv', index=False)
print("\n✅ Comparison table saved to 'model_comparison.csv'")

# Derived stats for report text
fpr_lr_pct = 100.0 * fp_lr / (fp_lr + tn_lr) if (fp_lr + tn_lr) else np.nan
alerts_lr = tp_lr + fp_lr
test_size = int(len(y_test))
alerts_lr_pct = 100.0 * alerts_lr / test_size if test_size else np.nan

fpr_rf_pct = 100.0 * fp_rf / (fp_rf + tn_rf) if (fp_rf + tn_rf) else np.nan
alerts_rf = tp_rf + fp_rf
alerts_rf_pct = 100.0 * alerts_rf / test_size if test_size else np.nan

# Training/prediction time (minutes + speedups)
lr_train_min = lr_train_time / 60.0
rf_train_min = rf_train_time / 60.0
lr_pred_ms = lr_pred_time * 1000.0 / max(test_size,1)
rf_pred_ms = rf_pred_time * 1000.0 / max(test_size,1)

train_speedup = (rf_train_time / lr_train_time) if lr_train_time>0 else np.nan
```

```
pred_speedup = (rf_pred_time / lr_pred_time) if lr_pred_time>0 else np.nan

# Top-5 features (RF)
top5 = feature_importance.head(5).reset_index(drop=True)

# RESULTS TXT (for easy copy into report)
with open("results.txt", "w", encoding="utf-8") as f:
    f.write("==== Exploratory Data Analysis ====\n")
    f.write(f"Median fraud amount: {median_fraud_amount:,.0f} units\n")
    f.write(f"Median legitimate amount: {median_legit_amount:,.0f} units\n")
    f.write(f"% fraudulent CASH_OUT with origin drained (newbalanceOrig==0): {pct_cashout_drained:.2f}%\n")
    f.write(f"% fraudulent TRANSFER with dest unchanged: {pct_transfer_dest_unchanged:.2f}%\n")
    f.write("\n")

    f.write("==== Logistic Regression (Results at Optimal Threshold) ====\n")
    f.write(f"Threshold: {CHOSEN_THRESHOLD_LR:.4f}\n")
    f.write(f"Precision: {precision_lr:.4f}\n")
    f.write(f"Recall: {recall_lr:.4f}\n")
    f.write(f"F1-Score: {f1_lr:.4f}\n")
    f.write(f"ROC-AUC: {roc_auc_lr:.4f}\n")
    f.write(f"PR-AUC: {pr_auc_lr:.4f}\n")
    f.write(f"TP: {tp_lr:,}\nFP: {fp_lr:,}\nFN: {fn_lr:,}\nTN: {tn_lr:,}\n")
    f.write(f"False Positive Rate: {fp_lr:,} / {fp_lr + tn_lr:,} = {fpr_lr_pct:.4f}%\n")
    f.write(f"Alerts generated: {alerts_lr:,} of {test_size:,} = {alerts_lr_pct:.2f}%\n")
    f.write(f"Training time: {lr_train_min:.2f} min; Prediction: {lr_pred_ms:.4f} ms/example\n")
    f.write("\n")

    f.write("==== Random Forest (Results at Optimal Threshold) ====\n")
    f.write(f"Threshold: {CHOSEN_THRESHOLD_RF:.4f}\n")
    f.write(f"Precision: {precision_rf:.4f}\n")
    f.write(f"Recall: {recall_rf:.4f}\n")
    f.write(f"F1-Score: {f1_rf:.4f}\n")
    f.write(f"ROC-AUC: {roc_auc_rf:.4f}\n")
    f.write(f"PR-AUC: {pr_auc_rf:.4f}\n")
    f.write(f"TP: {tp_rf:,}\nFP: {fp_rf:,}\nFN: {fn_rf:,}\nTN: {tn_rf:,}\n")
    f.write(f"False Positive Rate: {fp_rf:,} / {fp_rf + tn_rf:,} = {fpr_rf_pct:.4f}%\n")
    f.write(f"Alerts generated: {alerts_rf:,} of {test_size:,} = {alerts_rf_pct:.2f}%\n")
    f.write(f"Training time: {rf_train_min:.2f} min; Prediction: {rf_pred_ms:.4f} ms/example\n")
    f.write("\n")

    f.write("==== Comparative Analysis ====\n")
    f.write(f"Precision @ Recall⩾95% — LR: {precision_lr:.4f}, RF: {precision_rf:.4f}\n")
    f.write(f"Recall — LR: {recall_lr:.4f}, RF: {recall_rf:.4f}\n")
    f.write(f"F1-Score — LR: {f1_lr:.4f}, RF: {f1_rf:.4f}\n")
    f.write(f"ROC-AUC — LR: {roc_auc_lr:.4f}, RF: {roc_auc_rf:.4f}\n")
    f.write(f"PR-AUC — LR: {pr_auc_lr:.4f}, RF: {pr_auc_rf:.4f}\n")
    if not np.isnan(train_speedup):
        faster = "LR" if train_speedup>1 else "RF"
        factor = train_speedup if train_speedup>1 else (1/train_speedup if train_speedup>0 else np.nan)
        f.write(f"Training time ratio (RF/LR): {train_speedup:.2f} → {('LR' if faster=='LR' else 'RF')} ~{factor:.1f}×
faster\n")
    if not np.isnan(pred_speedup):
        faster_p = "LR" if pred_speedup>1 else "RF"
        factor_p = pred_speedup if pred_speedup>1 else (1/pred_speedup if pred_speedup>0 else np.nan)
        f.write(f"Prediction time ratio (RF/LR): {pred_speedup:.2f} → {('LR' if faster_p=='LR' else 'RF')} ~{factor_p:.1f}× faster per example\n")
    f.write("\n")

    f.write("Top 5 Features by RF importance:\n")
    for i, row in top5.iterrows():
        f.write(f"{i+1}. {row['feature']} ({100*row['importance']:.2f}% importance)\n")

print("✅ Wrote 'results.txt' with all values for your report.")
print("✅ Saved all 10 figures under './figures' with the expected names.")
```