Credit Card Transaction Fraud Detection Project Report

Baher Alabbar
baher.alabbar@gmail.com

August 4, 2025

Contents

1 Introduction

This project addresses the challenge of detecting fraudulent credit card transactions using machine learning. Since fraudulent activity is extremely rare, the dataset is highly imbalanced, which requires careful handling during both preprocessing and model evaluation. Our goal is to develop a robust classifier by exploring a wide range of techniques to address the imbalance. This includes experimenting with various resampling methods, supervised and unsupervised models, and other strategies tailored for imbalanced classification. In this project, we focus on maximizing both precision and recall, aiming for a high F1-score as our primary evaluation metric.

2 EDA Insights & Feature Discovery

We began our exploration of the dataset by reviewing its structure and understanding the nature of the features provided. The dataset includes the following columns:

- **Time**: The number of seconds elapsed between a given transaction and the first transaction in the dataset.
- **V1 to V28**: These features are the result of a Principal Component Analysis (PCA) transformation applied to the original feature space. They are anonymized to protect user privacy and sensitive information. Each component captures a distinct direction of variance in the original data.
- **Amount**: The monetary value of the transaction.
- **Class**: The binary target variable, where 1 indicates a fraudulent transaction and 0 a legitimate one.

2.1 Data Cleaning and Preprocessing

Initial data cleaning steps included:

- Removing duplicate records to prevent data leakage and reduce model bias.

- Handling outliers using statistical thresholds and visual inspection.
- Applying a $\log(1 + x)$ transformation to skewed numerical features (e.g., `Amount` and possibly some principal components) to reduce the impact of extreme values and improve model convergence.

2.2 Feature Importance Analysis

Since most features are anonymized, we relied on model-driven techniques to assess their importance:

- Extracted and analyzed feature coefficients from the trained Logistic Regression model.
- Computed feature importance scores using a trained Random Forest classifier.

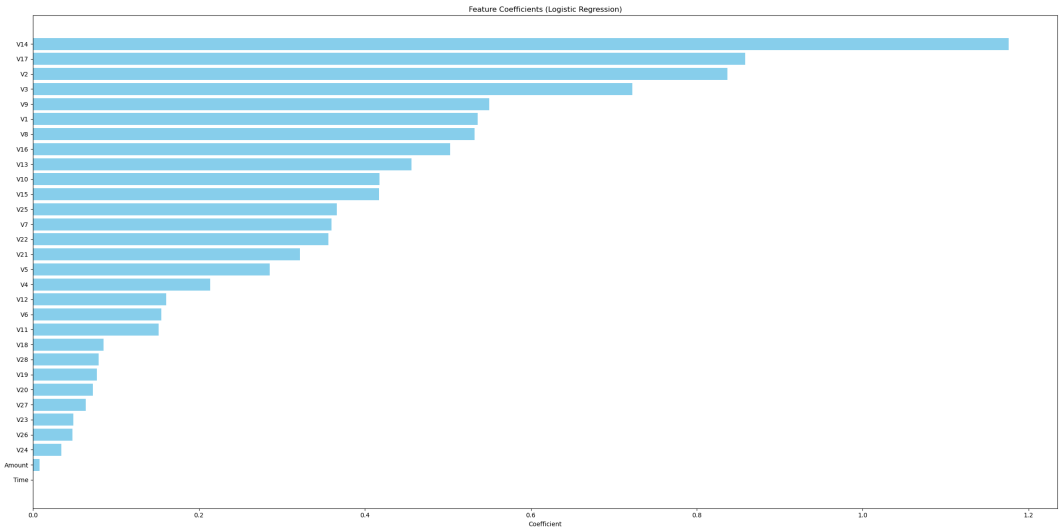

Figure 1: Feature importance from Logistic Regression.

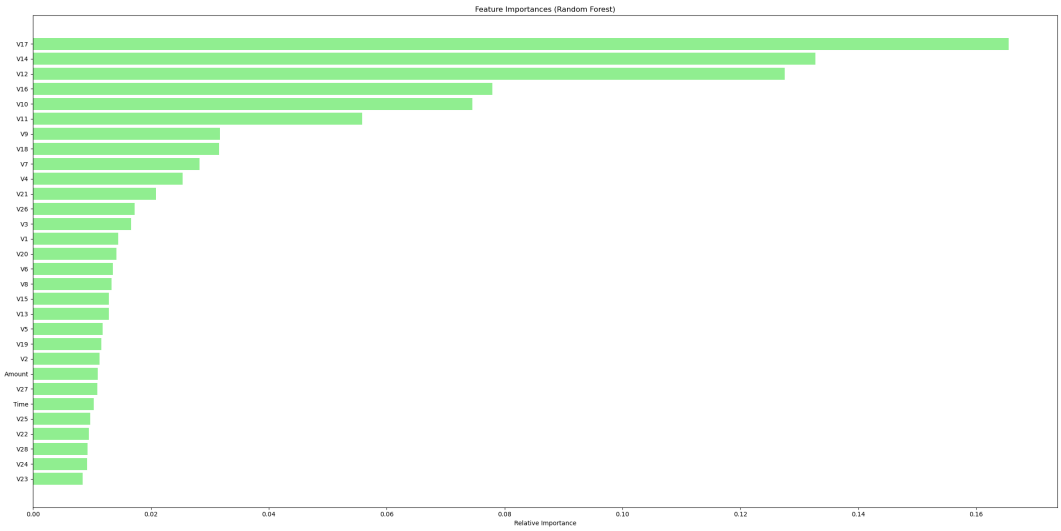

Figure 2: Feature importance from Random Forest.

2.3 Insights on Feature Relevance

From our analyses, we observed the following:

- `Time` and `Amount` were among the least important features in both models. This suggests that these variables are relatively uninformative regarding fraudulent behavior.
- Features `V14` and `V17` consistently ranked highly across both models, indicating strong predictive power and possibly capturing meaningful fraud-related patterns even before the PCA transformation.
- The Logistic Regression model highlighted features with linear relationships to the target, while the Random Forest model surfaced features such as `V10`, `V12`, and `V16`, likely due to its ability to capture non-linear interactions.

These insights lead to the following key takeaways:

- Logistic Regression's linear nature limits its ability to capture non-linear patterns unless polynomial feature expansion is applied.
- Random Forest naturally accounts for complex feature interactions, hence its differing importance rankings.

- Enhancing Logistic Regression performance may involve engineering polynomial features for those identified as important non-linear contributors.
- The agreement between both models on the significance of `V14` and `V17` reinforces their robustness as predictive features.

EDA Summary

The exploratory data analysis (EDA) revealed limited insights due to the anonymized nature of the features, which lack clear semantic meaning. This made it difficult to interpret variable relationships or engineer domain-specific features. However, initial patterns suggest that some features may have non-linear associations with the target variable. As a result, a key next step will be exploring feature expansion techniques to help models like Logistic Regression better capture these complex patterns.

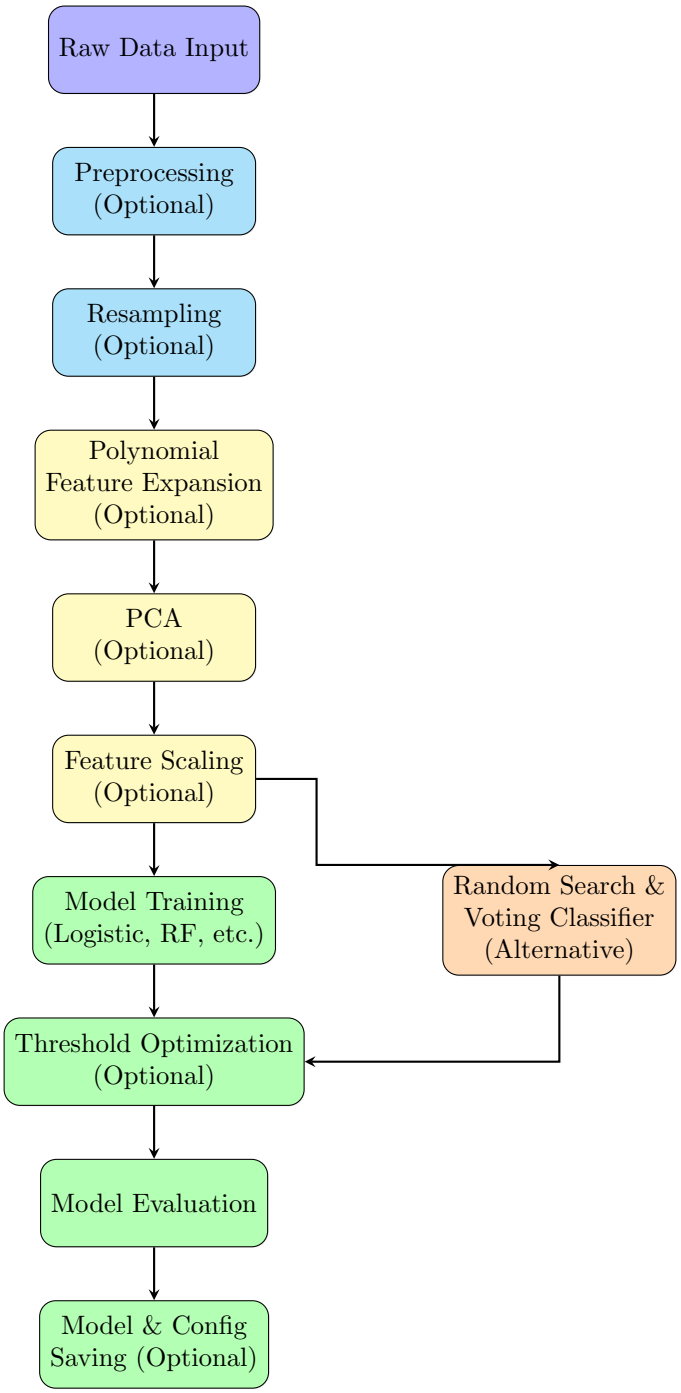

Figure 3: Training pipeline showing optional steps for preprocessing, resampling, feature expansion, dimensionality reduction, and model selection.

Pipeline Step	Approach / Options	Description
Data Input	Full dataset or Sample dataset	Default full train/val/test CSVs or sample data if `--use-sample` flag is set
Preprocessing	Applied or skipped (`--preprocess` flag)	Custom preprocessing pipeline applied if flag is set
Resampling	None, SMOTE oversampling (`over-smote`), SMOTE KMeans oversampling (`over-kmeans`), random undersampling (`under-random`), KMeans undersampling (`under-kmeans`), Combination oversampling + undersampling (`both`, `both-kmeans`)	Used to handle class imbalance if `--resampler` is not `"none"`
Feature Expansion	Polynomial features with degree 0 (disabled) or higher (e.g., 2, 3) (`--expansion`)	Expands selected columns with polynomial features
Dimensionality Reduction	PCA enabled/disabled (`--pca`)	Reduces features to components (default keeps 95% variance or custom components number)
Feature Scaling	None, Min-Max scaling, Standardization (`--scale`)	Scales features after expansion and PCA
Model Selection	Logistic Regression, Neural Network, Random Forest, KNN, Voting Classifier (`--model`)	Model type to train
Hyperparameter Tuning	Random Search enabled/disabled (`--random-search`)	Use random search for hyperparameter tuning
Threshold Optimization	Number of thresholds to evaluate (default 0 disables) (`--thresholds-num`)	Finds best threshold on training data if enabled
Model Saving	Save or not (`--save-model`)	Save trained pipeline, config, and threshold
Test Mode	Normal mode or train on train+val and test on test set (`--test`)	Changes dataset split used for training and evaluation

Table 1: Summary of different approaches and options in each pipeline step

3 Modeling Strategy & Pipeline Design

As illustrated in the pipeline diagram and detailed in Table 1, the modeling strategy involved experimenting with multiple models and pipeline combinations. The primary evaluation metric throughout the process was the F1 score, as justified in the introduction due to the class imbalance in the dataset.

Before the modeling phase began, I created a smaller sample of the dataset to reduce computational time during development. This was achieved using a custom script, `sample_split.py`, which applied stratified sampling to maintain the original class distribution. This was important to preserve the integrity of the classification task. However, during the evaluation phase, it became evident that the sample-based results diverged from those on the full dataset—especially when using resampling techniques. Therefore, after the initial logistic regression tests, I switched back to using the full dataset for all subsequent modeling stages.

To further optimize training efficiency, I used Randomized Search with Stratified K-Fold Cross-Validation rather than Grid Search. While Grid Search is exhaustive, it is computationally expensive. Random Search allowed for broader hyperparameter exploration with significantly less runtime, making it more practical for this large-scale dataset.

The pipeline was carefully designed to reflect a logical and effective processing order. Preprocessing steps (e.g., log transformations, feature selection) were applied first to ensure that only relevant and appropriately engineered features were passed to later steps. Feature expansion, such as polynomial feature generation, was applied next—before Principal Component Analysis (PCA)—to ensure that non-linear relationships identified during EDA were captured before dimensionality reduction. Both feature expansion and PCA were intentionally placed before scaling to prevent post-scaling transformations from altering feature distributions. Feature scaling is crucial for models like neural networks, which rely on gradient descent, to ensure that all features contribute equally during learning. This is less critical for closed-form models like logistic regression but essential for stability in neural networks.

The overall modeling approach began by identifying the best version of each model family: Logistic Regression, Random Forest, Neural Network, and K-Nearest Neighbors (KNN). For each, I compared manual configurations with those produced by Randomized Search. Additionally, I constructed a Voting Classifier that combined multiple base models. Unlike traditional ensembling of pre-tuned models, the Voting Classifier itself was included in the hyperparameter search space and optimized using Randomized Search. Its performance was

then directly compared to the individual best-performing models.

At the final stage, the best two models—selected based on validation performance—were evaluated on the held-out test set. This comparison aimed to assess generalization ability and determine which model performed best in a realistic deployment scenario.

4 Model Selection - Phase 1: Manual Tuning

In this phase, each model was manually tuned and evaluated step-by-step to understand its behavior and the impact of various pipeline components. The objective was to explore individual model performance through iterative adjustments and draw early insights before engaging in automated hyperparameter search.

4.1 Baseline Logistic Regression

We began with a baseline logistic regression model, applied without preprocessing, feature expansion, resampling, or scaling. Performance was evaluated on both a stratified sample and the full dataset.

F1 Score on Sample — Train: 0.7143, Validation: 0.7083
F1 Score on Full Set — Train: 0.7169, Validation: 0.7251

This confirmed that the sample could approximate trends from the full dataset, making it a practical choice for early experimentation. However, significant differences in later stages (especially when resampling) led to abandoning the sample in favor of the full set.

4.2 Threshold Tuning

Replacing the default threshold with the best one based on training PR-AUC improved training performance but slightly worsened validation performance (Train: 0.7385 vs. Val: 0.6829). On the full set, validation improved slightly to 0.7179. This shows that thresholds selected via training data do not always generalize well to unseen data, likely due to class imbalance and overfitting to training distributions.

4.3 Feature Expansion

Motivated by EDA findings where some features had non-linear importance (visible in RF but not logistic), we applied polynomial expansion.

- Degree 2: Train 0.7097, Val 0.5882
- Degree 3: Train 0.6552, Val 0.5000

Higher-degree expansions resulted in overfitting and reduced validation performance. Thus, polynomial expansion was ultimately excluded from the final logistic regression model.

4.4 Cost-Sensitive Learning

To address class imbalance without data augmentation, class weights were tuned. This approach outperformed even feature expansion: Train F1: 0.7838, Validation F1: 0.8261

This highlights the value of incorporating class importance directly into model training.

4.5 Resampling Techniques

Various resampling methods were tested with logistic regression:

Oversampling:

- SMOTE — Train: 0.7401, Val: 0.7487
- KMeans-SMOTE — Train: 0.6545, Val: 0.7027

Undersampling:

- Random — Train: 0.5473, Val: 0.5414
- KMeans — Train: 0.7524, Val: 0.7805

Combined:

- Random + SMOTE — Train: 0.7717, Val: 0.8111

KMeans undersampling outperformed random undersampling, likely due to more representative sampling. Combining oversampling and undersampling yielded the best results overall. KMeans combinations, however, were too computationally expensive and unstable.

Due to divergence between sample and full dataset performance during resampling, the sample was fully abandoned for the rest of the project.

4.6 PCA Impact

Using PCA with 18 components in logistic regression yielded strong performance, with an F1 score of 0.8034 on the training set and 0.8070 on the validation set. This shows that dimensionality reduction using PCA was effective in retaining important information for classification while reducing feature space complexity.

4.7 Best Logistic Model

The best result was achieved using cost-sensitive logistic regression with threshold tuning (no polynomial expansion):

- F1 Score (Train): 0.8065
- F1 Score (Val): 0.8118
- Precision: 0.8519, Recall: 0.7753, Average Precision: 0.7422

4.8 Other Models - Manual Observations

Neural Network: Scaling was critical. Without it, performance was poor. With standardization: Train F1: 0.9693, Validation F1: 0.8606

This confirms that scaling improves convergence in gradient-based models.

Random Forest: Train F1: 1.0000, Validation F1: 0.8810 — high performance, but possible overfitting.

KNN: KNN was highly sensitive to scaling and dimensionality. PCA with 20 components and standardized data yielded: Train F1: 0.8689, Validation F1: 0.8639

Without PCA, performance was slightly worse, suggesting PCA helped reduce noise when paired with tuning.

4.9 Summary

Manual tuning helped me understand how each model behaves. Logistic regression improved with class weights and threshold adjustment. Resampling gave small improvements. I stopped using polynomial features because they caused overfitting. Scaling was very important for models like neural networks and KNN. PCA gave mixed results—it hurt linear models but helped KNN.

These findings helped guide Phase 2, where I used Randomized Search to automatically find the best settings for each model.

5 Model Selection - Phase 2: Randomized Search

In this phase, I used Randomized Search to automatically tune hyperparameters for each model. The search ranges were chosen based on what worked best during manual tuning in Phase 1.

Logistic Regression: The best model used class weighting, threshold tuning, and scaling. It achieved an F1 score of 0.7950 on the validation set with a threshold of 0.5556. While the results were solid, they were slightly worse than the manually tuned version.

Neural Network: The Randomized Search version scored 0.8383 F1 on the validation set. However, this was still lower than the manually tuned neural network. It shows that while Randomized Search can find decent configurations, manual tuning gave better results in this case.

Random Forest: Validation F1 score reached 0.8475, which is strong, but again not better

than the best manually tuned version. The training score was very high (0.9562), hinting at some overfitting.

KNN: KNN performed better with PCA. With PCA, the validation F1 was 0.7979 compared to 0.7629 without PCA. However, training F1 was 1.0000 in both cases. This is expected—since KNN compares each input to the training data directly, it naturally gets perfect scores on the training set.

Conclusion: For all models tested, Randomized Search gave good results, but none of them outperformed the best manually tuned versions from Phase 1. These findings helped prepare for Phase 3, where a Voting Classifier was built and tuned.

6 Model Selection - Phase 3: Voting Classifier

In this phase, I used a Voting Classifier to combine the strengths of the best individual models. The idea is to reduce the effect of noise and let each model contribute what it does best. Instead of relying on just one model, this approach makes a final decision based on a majority or weighted vote.

Why use a Voting Classifier? It helps balance out the weaknesses of individual models and makes the overall system more stable. Some models may perform better in certain situations, and combining them can lead to stronger overall results.

Performance (Best threshold = 0.2323):

Metric	Training Set	Validation Set
F1 Score	0.9983	0.8824
Precision	0.9966	0.9259
Recall	1.0000	0.8427
Average Precision	0.9999	0.8601

Table 2: Voting Classifier performance metrics on training and validation sets.

Conclusion: This was the best model so far. It outperformed all previous models on the validation set, making it the top choice for final deployment.

7 Final Testing Results

For the final evaluation, I selected two models to test on the holdout set: the manually tuned K-Nearest Neighbors (KNN) model and the Voting Classifier (VC) built using random search.

Metric	**KNN**	**Voting Classifier**
F1 Score	0.8691	0.8557
Precision	0.8737	0.8469
Recall	0.8646	0.8646
Average Precision	0.8293	0.8694

Table 3: Final test set performance comparison between KNN and Voting Classifier.

The manually tuned KNN model achieved the best F1 score on the test set. However, the difference between KNN and the Voting Classifier is not large. Interestingly, the Voting Classifier had a higher average precision, suggesting it may rank predictions more effectively.

Conclusion: Both models performed well, and I am satisfied with the results. Depending on the final application, either model could be used with confidence.

8 Summary of Findings

I will start with an issue I encountered during modeling and what I learned from it.

At first, I used `imblearn.pipeline.Pipeline` to combine the resampling and modeling steps. I initially thought that all steps in the pipeline, including resampling, would apply to both training and testing data. Later, I discovered that `imblearn`'s pipeline is specifically designed to apply resampling only during training — and skip it during prediction or evaluation. This design helps prevent unintended behavior, such as applying resampling to the test set, which would be meaningless in a real-world setting. In production, we want to evaluate model performance on the original distribution of the data, not on a resampled or altered version.

This clarified an important point: using `sklearn.pipeline.Pipeline` with resampling steps isn't supported — it would raise an error. That's why `imblearn` provides its own pipeline to handle such scenarios properly.

Another issue I faced was related to undersampling. I realized that to apply it correctly, I should have first split the dataset by class. Undersampling should only be applied to the majority class, while the minority class should remain untouched. Applying undersampling to the entire dataset risks deleting minority samples, which defeats the purpose of trying to reduce imbalance.

From the overall modeling process, I learned that:

- Modeling and model selection can sometimes be more effective than exploratory data analysis (EDA). We achieved strong results even without relying heavily on insights from EDA.
- Manual tuning can outperform automated search methods, especially with larger datasets. While random and grid search provide broad coverage, they can be time-consuming. Manual tuning allowed more efficient and focused experimentation.
- Not all preprocessing steps work equally across models. For example, applying feature scaling significantly improved the performance of the Neural Network, while it had little to no effect on tree-based models like Random Forest. The improvement in the Neural Network was expected, as it uses gradient descent to update weights — and without scaling, features with larger values would dominate the learning process. This example highlights the importance of understanding how different models work and tailoring preprocessing accordingly.
- Using a Voting Classifier helped combine strengths from multiple models, leading to balanced and strong performance overall.
- **Exploring different techniques** — such as trying multiple models, tuning methods, and resampling approaches — was essential for finding a high-performing solution.

This project reinforced the importance of understanding pipeline behavior, how to properly handle imbalanced datasets, and the value of methodical experimentation during model development.

9 References

The dataset used in this project is private and not publicly available.

This report is part of a GitHub project, which contains the full exploratory data analysis (EDA), modeling scripts, and the configuration files for all the models discussed in this report.

The repository can be accessed at: https://github.com/b-4her/credit-card-fraud-detection