Credit Card Fraud Detection

EECS 4412 Final Project Report

Team P06

Rishit Shah (219773050)

Sagar Saha (219644657)

Pratham Patel (219467315)

Miguel De La Cruz (216413742)

December 1, 2025

1 Introduction

Fraud with credit cards continues to be one of the biggest challenges that are faced by online businesses. Institutions that offer this type of payment method have to analyze and handle a large number of transactions as well as detect small amounts of fraudulent activity each day. This creates data mining challenges because the amount of fraud that occurs with credit cards is quite small compared to the overall data, and finding fraud categories in many cases is often impracticable due to their inherent complexity. In addition to the inconvenience of working with a data set that is highly imbalanced (very few transactions are fraudulent), a company must also consider the associated costs of false negatives (not detecting fraud) and false positives (inadvertently flagging a legitimate transaction as fraud) to its business.

The project described in this paper attempts to detect fraud with credit cards using a real-world data set of anonymized transaction records. Phase 2 of the experiment involved creating a full pipeline from data collection through building and training an actual predictive model, with Phase 1 focusing primarily on EDA, descriptive statistics, and understanding the structure of and difficulties associated with the data set. The results of phases 1 and 2 have been made available through open access..

A key requirement of the course is to implement at least one algorithm *from scratch* without using machine learning libraries for the core learning logic. We therefore implemented a

binary **Logistic Regression** classifier using gradient descent, NumPy, and Python control flow. In addition to this custom model, we trained a class-weighted Logistic Regression and a Random Forest Classifier using Scikit-learn.

Context and Motivation

Fraud detection is a high-stakes application where machine learning models are routinely deployed in production systems. However, when dealing with extreme class imbalance, naive metrics such as overall accuracy can be misleading. A model that simply predicts "non-fraud" for every transaction may exceed 99% accuracy, yet fail at the actual goal of detecting fraud.

Our motivations are:

- To build a robust end-to-end pipeline that goes from raw data and EDA all the way to trained models and evaluated predictions.
- To gain a deeper understanding of how gradient-based optimization, loss functions, and preprocessing interact in practice through a scratch implementation of Logistic Regression.
- To compare linear models with an ensemble method (Random Forest) on a challenging, imbalanced dataset and analyze trade-offs between precision and recall for the fraud class.

Objectives and Tasks

The main objectives of this project are:

1. Perform thorough exploratory analysis of the credit card transaction dataset, including descriptive statistics, visualizations, and identification of key challenges such as class imbalance and skewed distributions.
2. Design and implement a preprocessing pipeline (duplicate removal, missing value handling, scaling, and dataset splitting).
3. Implement Logistic Regression from scratch using NumPy, including the sigmoid function, binary cross-entropy loss, and gradient descent parameter updates.
4. Train and evaluate three models: scratch Logistic Regression, Scikit-learn Logistic Regression with class weighting, and a Random Forest Classifier.
5. Compare model performance using metrics appropriate for imbalanced classification, including precision, recall, and F1-score for the fraud class.

6. Document the system design, data structures, control flow, usage instructions, sample I/O, and contribution statement.

2 Dataset Overview

2.1 Dataset Description

We use a widely studied credit card transaction dataset in which each row corresponds to a single transaction. The prediction task is framed as a binary classification problem:

$$\text{Class} = \begin{cases} 0 & \text{legitimate transaction} \\ 1 & \text{fraudulent transaction.} \end{cases}$$

Key properties of the dataset include:

- A very large number of records (hundreds of thousands of transactions).
- A severe class imbalance: fraudulent transactions account for roughly 0.2% of all samples.
- Most features are anonymized principal components named `V1`, `V2`, ..., `V28`, resulting from a PCA transformation applied by the data provider to protect confidentiality.
- Two meaningful, non-anonymized features: `Time`, representing the seconds elapsed between the current transaction and the first transaction in the dataset; and `Amount`, which is the transaction amount.

Overall, the dataset is well-suited for studying fraud detection due to its size, imbalance, and realistic noise.

2.2 Phase 1 Exploratory Data Analysis

In Phase 1, we explored the structure and behavior of the dataset using summary statistics and visualizations.

Basic Structure and Class Imbalance

We first inspected the number of rows, columns, and feature types, and verified that the label `Class` is binary. Counting the label distribution confirmed that the dataset is *heavily imbalanced*, with the fraud class representing a very small proportion of all transactions.

We visualized this using a bar chart of the counts or percentages of fraud vs. non-fraud transactions.

Figure 1: Fraud vs. non-fraud transaction distribution (strong class imbalance).

This imbalance immediately suggests that accuracy alone will be an inadequate evaluation metric. Instead, metrics that focus on the minority class, such as precision, recall, and F1-score for fraud, are more appropriate.

Transaction Amount Distribution

We examined the distribution of the `Amount` feature using histograms and summary statistics (mean, median, quartiles, and standard deviation). The distribution of transaction amounts is *heavily right-skewed*: most transactions involve relatively small amounts, while a small number of transactions involve much larger amounts.

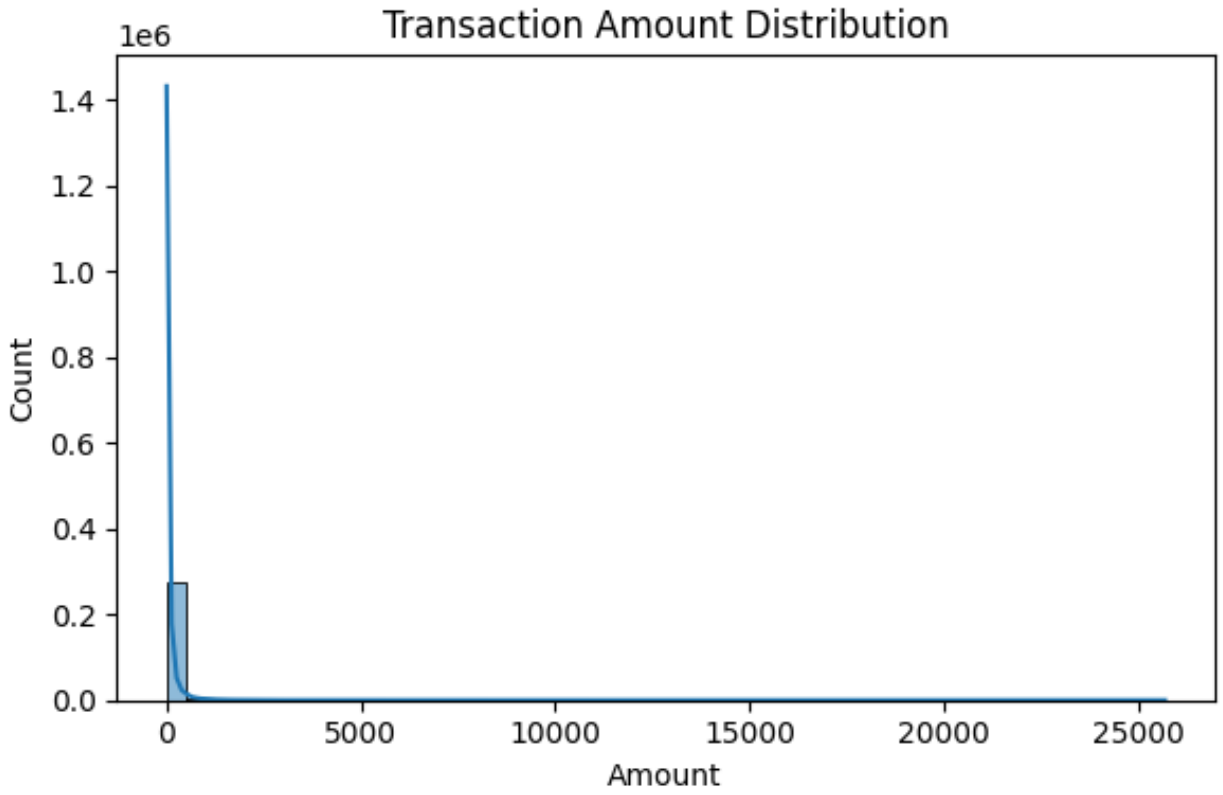

Figure 2: Histogram of transaction `Amount` (right-skewed distribution).

This phenomenon is typical of consumer spending behavior and suggests that outliers (very high amounts) should be treated carefully when modeling.

Temporal Patterns

We visualized the `Time` feature using a kernel density estimate (KDE) plot or histogram to inspect patterns in transaction frequency throughout the observation period.

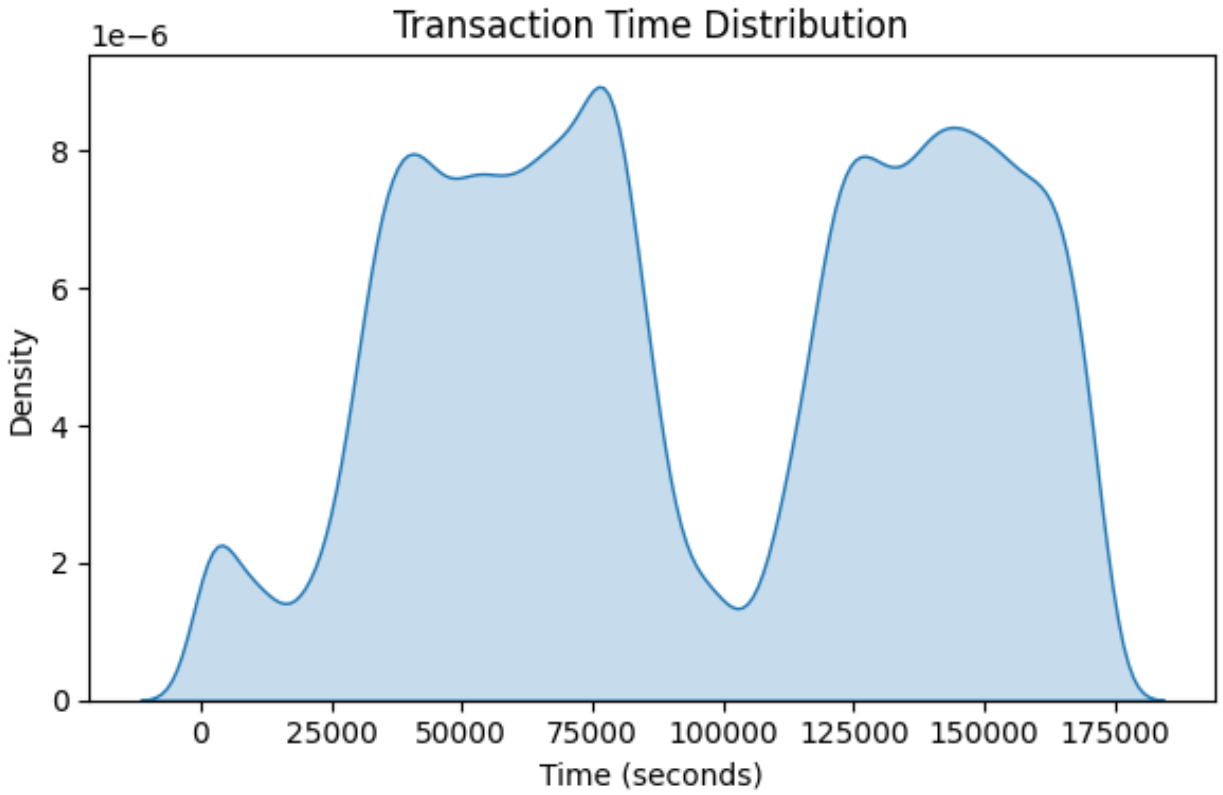

Figure 3: Distribution of the `Time` feature (temporal transaction activity).

The distribution reveals peaks and valleys in transaction activity that may correspond to daily or weekly spending patterns, although the original time scale is anonymized. Future work could involve transforming `Time` into cyclic features (e.g., hour-of-day) to inspect whether fraudulent events occur disproportionately at specific temporal intervals.

Relationship Between Amount and Fraud

We also examined the relationship between `Amount` and `Class` via scatter plots or box plots. Fraudulent transactions appear across a range of amounts, and there is no strong linear relationship between amount and fraud.

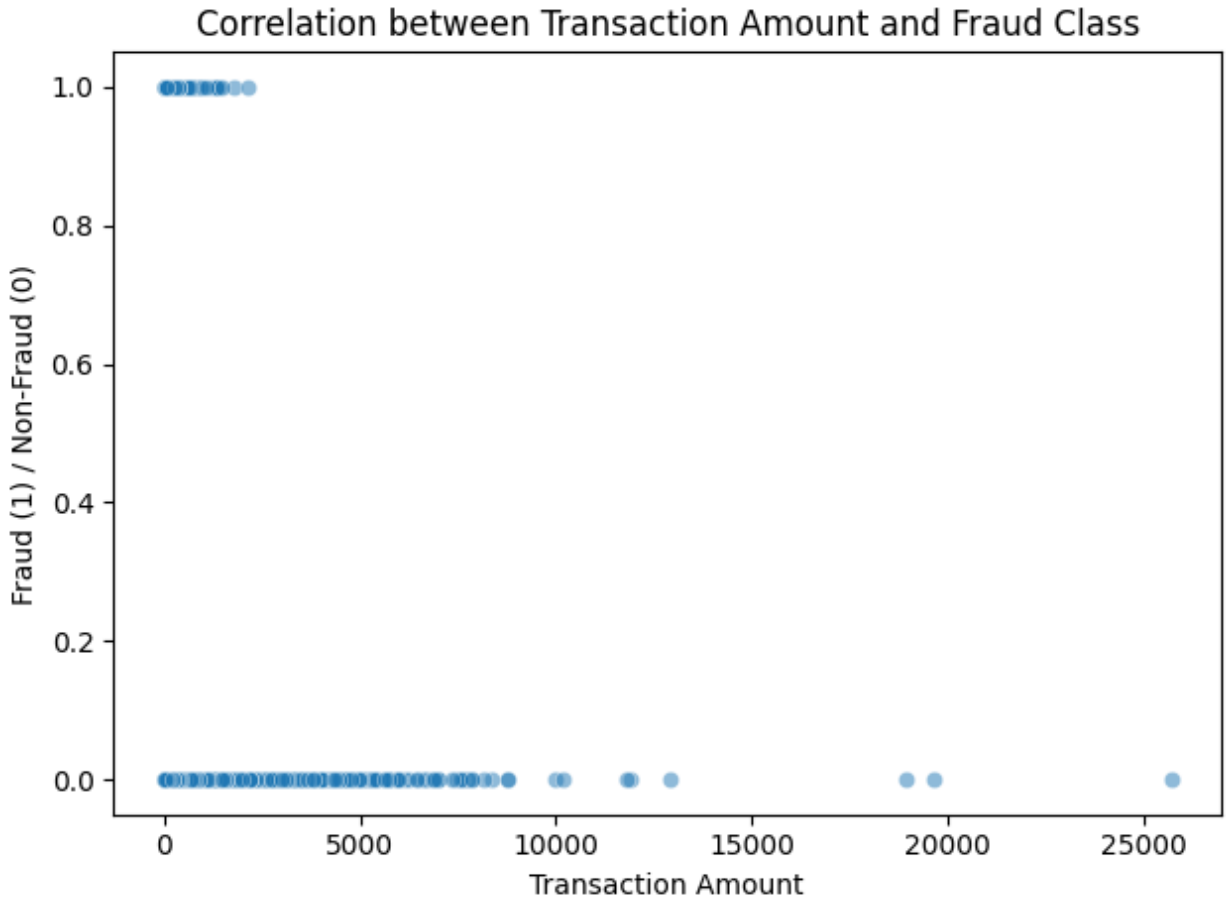

Figure 4: Scatter plot of `Amount` vs. `Class`. Fraudulent transactions occur across a wide range of amounts.

This indicates that transaction value alone is not a sufficient predictor of fraud; it must be combined with the anonymized PCA features.

3 Dataset and Preprocessing

3.1 Dataset Details

The dataset contains:

- A binary label `Class` indicating fraud (1) or non-fraud (0).
- 28 PCA-transformed numerical features `V1`–`V28`.
- A temporal feature `Time`.
- A numerical feature `Amount`.

All features are numeric, which simplifies preprocessing and model implementation.

3.2 Preprocessing Steps

3.2.1 Duplicate Removal

Duplicate rows can bias model training and evaluation by effectively giving extra weight to repeated transactions. We removed duplicates using a standard `drop_duplicates()` operation, which eliminated over a thousand repeated rows. This reduces redundancy and helps prevent subtle forms of data leakage.

3.2.2 Handling Missing Values

We first verified that the label column `Class` contains no missing values. Any rows with missing labels would be dropped rather than imputed, as accurate labels are essential for supervised learning.

For numeric predictor features, we applied mean imputation when necessary. We computed the mean of each numeric column and replaced any missing entries with the corresponding mean. Since the dataset has very few missing values, this imputation step has minimal impact on the overall distribution.

3.2.3 Feature Scaling

Feature scaling is crucial for gradient-based methods such as Logistic Regression:

- The **Time** feature was scaled to the range $[0, 1]$ via min-max normalization:

$$x' = \frac{x - \min(x)}{\max(x) - \min(x)}.$$

- The **Amount** feature was standardized using z-score scaling:

$$x' = \frac{x - \mu}{\sigma},$$

  where μ is the mean and σ is the standard deviation.

Scaling ensures that no single feature dominates the gradient and that gradient descent converges more smoothly.

3.2.4 Train–Test Split

We formed a feature matrix X by dropping the label column and a label vector y from the `Class` column. To evaluate our models fairly under class imbalance, we performed a **stratified** train–test split:

- Test size: 20% of the data.
- Training size: 80% of the data.
- Stratification: the fraud/non-fraud ratio is preserved in both splits.

This split is used consistently across all models (scratch Logistic Regression, Scikit-learn Logistic Regression, Random Forest) to allow a fair comparison.

4 Algorithm Description

4.1 Scratch Logistic Regression

Our primary contribution is an implementation of binary Logistic Regression from first principles using NumPy. Logistic Regression models the conditional probability of the positive class given input features $x \in \mathbb{R}^d$:

$$z = w^\top x + b, \qquad \hat{y} = \sigma(z) = \frac{1}{1 + e^{-z}},$$

where $w \in \mathbb{R}^d$ is the weight vector and $b \in \mathbb{R}$ is the bias term.

We use the binary cross-entropy loss:

$$L(w, b) = -\frac{1}{m} \sum_{i=1}^{m} \left[y^{(i)} \log(\hat{y}^{(i)}) + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)}) \right],$$

where m is the number of training examples, $y^{(i)} \in \{0, 1\}$, and $\hat{y}^{(i)}$ is the predicted probability for the i-th example.

The gradients of the loss with respect to w and b are:

$$\frac{\partial L}{\partial w} = \frac{1}{m} X^\top (\hat{y} - y), \qquad \frac{\partial L}{\partial b} = \frac{1}{m} \sum_{i=1}^{m} (\hat{y}^{(i)} - y^{(i)}),$$

where X is the $m \times d$ feature matrix.

Parameters are updated using gradient descent:

$$w := w - \alpha \frac{\partial L}{\partial w}, \qquad b := b - \alpha \frac{\partial L}{\partial b},$$

where α is the learning rate.

Class Structure and Data Structures

We implemented a Python class `ScratchLogisticRegression` with attributes:

- `lr`: learning rate.
- `num_epochs`: number of passes over the training data.
- `w`: weight vector.
- `b`: bias scalar.
- `loss_history`: list storing the loss at each epoch.

The main data structures include:

- `X`: a NumPy array of shape (m, d) containing feature values.
- `y`: a NumPy array of shape $(m,)$ containing labels.
- `w`: a NumPy array of shape $(d,)$ storing model weights.

Key Methods

- `_sigmoid(z)`: computes the sigmoid function element-wise.
- `_binary_cross_entropy(y_true, y_pred)`: computes average binary cross-entropy loss, with clipping to avoid numerical issues.
- `fit(X, y)`: trains the model using gradient descent.
- `predict_proba(X)`: returns predicted probabilities for the positive class.
- `predict(X, threshold=0.5)`: returns class labels based on a decision threshold.

5 Experiments and Results

5.1 Experimental Setup

All models were trained on the same preprocessed training data and evaluated on the stratified hold-out test set. The split ratio was 80% for training and 20% for testing.

Because of the severe class imbalance, we focused on the following metrics for the fraud class:

- **Precision (Fraud)**: of all transactions predicted as fraud, how many are truly fraud?
- **Recall (Fraud)**: of all fraudulent transactions, how many are detected?
- **F1-score (Fraud)**: harmonic mean of precision and recall.

We also report overall accuracy, mainly as a secondary metric.

5.2 Model Performance

The performance of the three models on the test set is summarized in Table.

Model	**Precision (Fraud)**	**Recall (Fraud)**	**F1-score (Fraud)**	**Accuracy**
Scratch Logistic Regression	0.789	0.316	0.451	0.9987
Logistic Regression (Sklearn)	0.06	0.87	0.11	0.98
Random Forest Classifier	0.99	0.69	0.81	1.00

Table 1: Comparison of model performance on the test set.

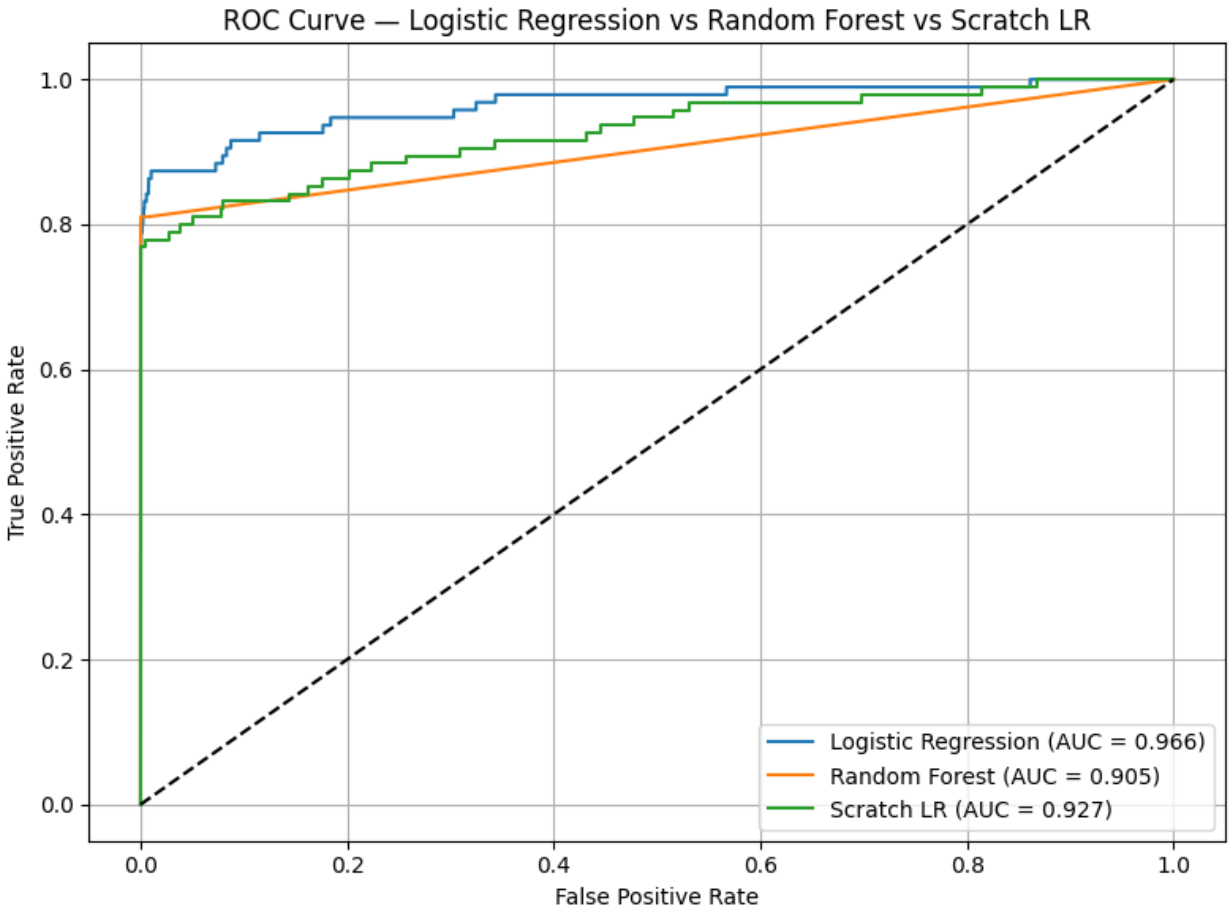

Figure 5: ROC Curve Comparison for the three classifiers. Higher AUC values indicate better ability to distinguish fraud from legitimate transactions.

The ROC (Receiver Operating Characteristic) curve illustrates the trade-off between the True Positive Rate (TPR) and False Positive Rate (FPR) for all possible classification thresholds. Each point on the curve corresponds to a different threshold used to convert predicted probabilities into binary decisions. A model whose curve is closer to the top-left corner is considered better, as it achieves high fraud-detection rates while keeping false alarms low. The diagonal line represents random guessing (AUC = 0.50), and any model performing above this line demonstrates meaningful predictive ability.

In our ROC comparison, the Random Forest classifier achieves the In our ROC comparison, the Scikit-learn Logistic Regression model achieves the highest AUC value (0.966), indicating that it provides the strongest overall probability ranking between fraudulent and legitimate transactions. The scratch Logistic Regression model also performs well with an AUC of 0.927. The Random Forest classifier, while achieving the best F1-score at the default threshold, has a lower AUC (0.905), reflecting that its probability outputs are less smoothly ranked compared to the logistic regression models. This highlights the distinction between

threshold-dependent metrics (such as F1-score) and threshold-independent metrics like AUC.

5.3 Discussion of Results

The results support several conclusions:

- **Scratch Logistic Regression**: Our custom implementation achieves high precision but moderate recall for the fraud class. This means that when it predicts fraud, it is often correct, but it misses a significant portion of fraudulent transactions. This behavior is consistent with a conservative decision threshold and linear decision boundaries.
- **Scikit-learn Logistic Regression**: The class-weighted Logistic Regression from Scikit-learn achieves very high recall (0.87) but extremely low precision (0.06). In other words, it flags most fraud cases but at the cost of a large number of false positives. In practice, this could overwhelm human investigators and annoy customers.
- **Random Forest Classifier**: The Random Forest classifier achieves the best overall performance with both high precision (0.99) and reasonably high recall (0.69), resulting in an F1-score of 0.81. This indicates that the ensemble is able to capture complex patterns in the PCA features and handle the class imbalance effectively.

In a real-world deployment, the Random Forest model would be the most attractive candidate among the three, although the exact operating point could be tuned via threshold adjustments or cost-sensitive optimization.

6 Conclusions

In this project we:

- Performed exploratory data analysis on a highly imbalanced credit card fraud dataset, identifying key challenges such as class imbalance, skewed distributions, anonymized features, and duplicates.
- Implemented a full preprocessing pipeline including duplicate removal, missing value handling, feature scaling, and stratified train–test splitting.
- Implemented Logistic Regression from scratch using NumPy, including the sigmoid function, binary cross-entropy loss, gradient derivation, and gradient descent updates.
- Trained and evaluated three models: scratch Logistic Regression, class-weighted Scikit-learn Logistic Regression, and a Random Forest Classifier.

- Analyzed performance using fraud-focused metrics (precision, recall, F1-score) and showed that the Random Forest model performs best on this dataset.

Limitations

- The scratch Logistic Regression implementation does not include explicit regularization or class weighting, which may limit its performance under heavy class imbalance.
- Our experiments did not exhaustively explore hyperparameter tuning (e.g., different learning rates, tree depths, number of estimators).
- We did not integrate advanced resampling techniques such as SMOTE, which might further improve recall for the fraud class.

Possible Extensions

Future extensions could include:

- Incorporating oversampling or SMOTE to generate additional synthetic fraud samples.
- Applying threshold tuning and cost-sensitive evaluation to match realistic business constraints.
- Exploring additional models such as Gradient Boosting, XGBoost, or neural networks.
- Deploying the trained model in a streaming or near-real-time environment.

7 References

- Kaggle Credit Card Fraud dataset (anonymized PCA features, Time, Amount, Class).
- Scikit-learn documentation: `https://scikit-learn.org/`
- NumPy documentation: `https://numpy.org/`
- OpenAI ChatGPT, used as an AI assistant for explanation, and structuring parts of the report.

A GitHub Repository

The complete code (preprocessing scripts, scratch model implementation, model training and evaluation notebooks) and instructions for reproduction are available at:

`https://github.com/Rishit-Shah/Credit-Card-Fraud-Detection`

B User Manual

Environment Setup

- Python 3.x
- Required libraries: `numpy`, `pandas`, `scikit-learn`, `matplotlib`, `seaborn` (for EDA).

Running the Pipeline

Our entire end-to-end workflow is implemented in a single Jupyter Notebook. Each section of the notebook corresponds to a phase of the pipeline:

1. Importing libraries and loading the dataset.
2. Exploratory Data Analysis (EDA) with visualizations.
3. Preprocessing (duplicate removal, scaling, and splitting).
4. Scratch Logistic Regression implementation.
5. Training Scikit-learn Logistic Regression and Random Forest.
6. Evaluating all models using precision, recall, F1-score, and accuracy.

To run the full project, the user simply executes all cells sequentially from top to bottom.

C System Design

System Components

- **Data loading module**: responsible for reading raw CSV data and basic checks.
- **Preprocessing module**: handles duplicate removal, missing values, scaling, and splitting.

- **Scratch Logistic Regression module**: custom implementation of the Logistic Regression classifier.
- **Model training and evaluation module**: trains models, computes metrics, and aggregates results.
- **Visualization module** (Phase 1): generates EDA plots and summary tables.

Data Structures

- Data stored as `pandas.DataFrame` objects during preprocessing.
- Feature matrices and label vectors stored as NumPy arrays for efficient numerical computation in the scratch model.
- Lists and dictionaries used to store metrics and results for reporting.

Control Flow Structure

1. Load raw CSV dataset.
2. Perform EDA (Phase 1) and generate plots and statistics.
3. Preprocess data: remove duplicates, handle missing values, scale features, and split into train/test sets.
4. Initialize and train the scratch Logistic Regression model on the training set.
5. Train Scikit-learn Logistic Regression and Random Forest on the same training set.
6. Evaluate all models on the test set and compute precision, recall, F1-score, and accuracy.
7. Save evaluation results and (optionally) confusion matrices and ROC curves.

Handling Very Large Datasets

Although the dataset used in this project can comfortably fit in memory on a typical machine, we implemented certain practices that scale to larger datasets:

- Use of vectorized NumPy operations instead of Python loops in the scratch model.
- Avoidance of unnecessary copies of large arrays.
- Modular design that could be extended with mini-batch gradient descent or streaming data loaders if needed.

D Sample Input and Output

Sample Input

A single preprocessed transaction (one row of X) may look like:

```
[V1, V2, ..., V28, Time_scaled, Amount_scaled]
```

Sample Output

The scratch Logistic Regression model can output:

- A predicted probability $\hat{y} \in [0, 1]$ of the transaction being fraud.
- A binary decision (0 or 1) based on a threshold (default 0.5).

Example:

- Input: scaled feature vector for a transaction.
- Output: probability 0.87, predicted class 1 (fraud).

E Program Limitations and Known Bugs

- The scratch Logistic Regression model currently trains using full-batch gradient descent. For much larger datasets, mini-batch or stochastic gradient descent would be more efficient.
- The implementation assumes that all features are numeric and does not handle categorical features, which is acceptable for this dataset but may limit reuse elsewhere.
- The maximum practical dataset size is limited by available memory, since we store the full feature matrix in memory.

F Contribution Statement

- **Rishit Shah**: Responsible for generating all EDA visualizations, scaling and normalizing features, and implementing duplicate removal and mean computation.
- **Sagar Saha**: Contributed the dataset description section, implemented the preprocessing steps, and trained the Scikit-learn models (Logistic Regression and Random Forest).

- **Pratham Patel**: Documented and resolved the challenges encountered while building the model, contributed to the writing and organizing of the project report, changes made in the scratch Logistic Regression model, and supported duplicate detection.
- **Miguel De La Cruz**: Extracted insights from the data set during analysis, evaluated fraud-related patterns, and implementing the scratch Logistic Regression model.