TEAM:5

CREDIT CARD FRAUD DETECTION

DİLDAR YILMAZ
MAHMUT İSMAİL ÖZTÜRK
RANA M S SHEHADA

PROJECT GOAL

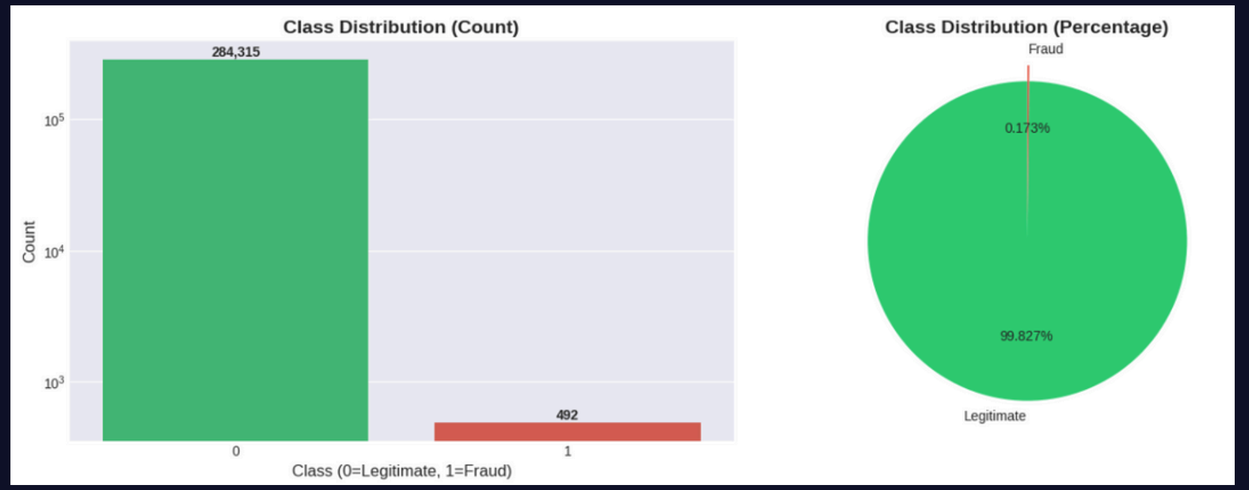
To build and compare machine-learning models for detecting fraudulent credit-card transactions, and to study how preprocessing methods (PCA, SMOTE, imputation, scaling) affect model performance on an imbalanced dataset.

DATASET OVERVIEW

- The dataset is highly imbalanced (99.83% normal vs 0.17% fraud).

- Below is a preview of the first 5 transactions.

- This imbalance makes fraud detection difficult and requires special techniques.



	Time	V1	V2	...	V28	Amount	Class
0	0.0	-1.359807	-0.072781	...	-0.021053	149.62	0
1	0.0	1.191857	0.266151	...	0.014724	2.69	0
2	1.0	-1.358354	-1.340163	...	-0.059752	378.66	0
3	1.0	-0.966272	-0.185226	...	0.061458	123.50	0
4	2.0	-1.158233	0.877737	...	0.215153	69.99	0

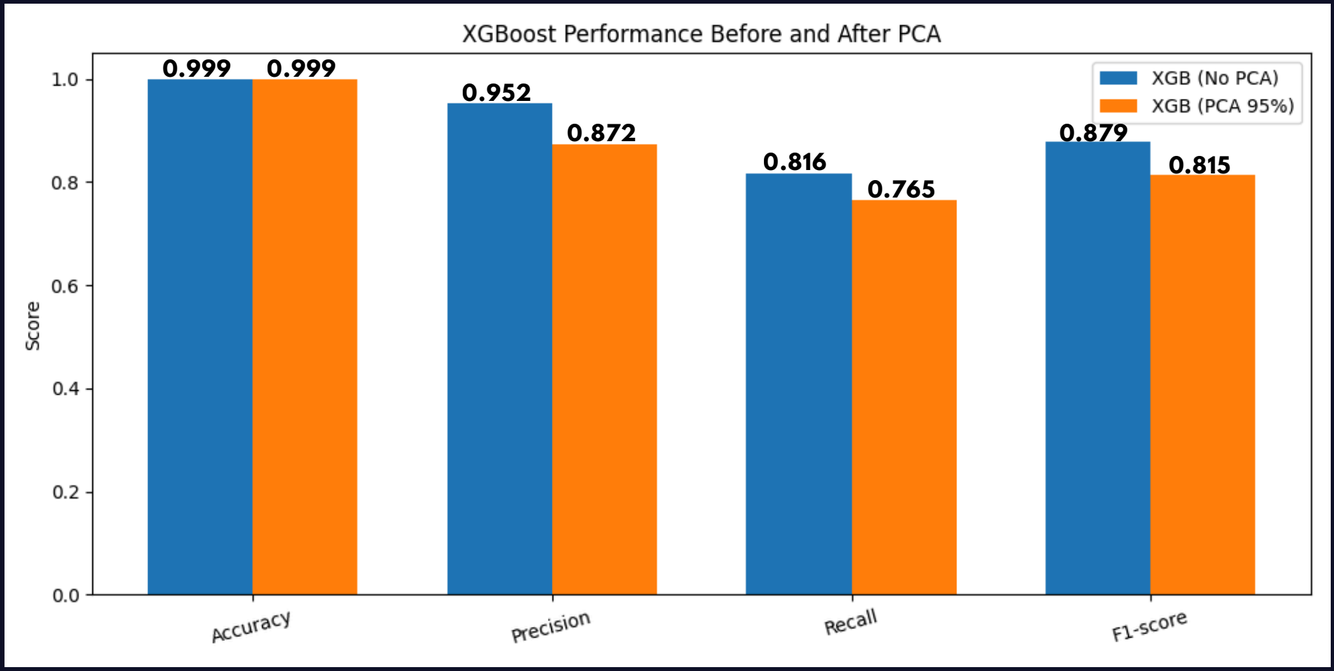
1. PCA
PRINCIPAL COMPONENT ANALYSIS

Principal Component Analysis (PCA) is a dimensionality reduction method that reduces large data sets into fewer variables while preserving key data trends. It simplifies data by identifying uncorrelated components that capture the most variance, making analysis faster and more efficient.

PCA in This Project:

- Applied PCA to 30 original features
- Retained 95% variance, resulting in 27 principal components
- Used PCA with XGBoost, SVM, MLP, and KNN
- Compared model performance before vs. after applying PCA
- Evaluated models using Accuracy, Precision, Recall and F1

XGBoost Comparison: Performance Before and After PCA



Processing time (XGB without PCA): 3.71 seconds
Processing time (XGB with PCA): 3.38 seconds

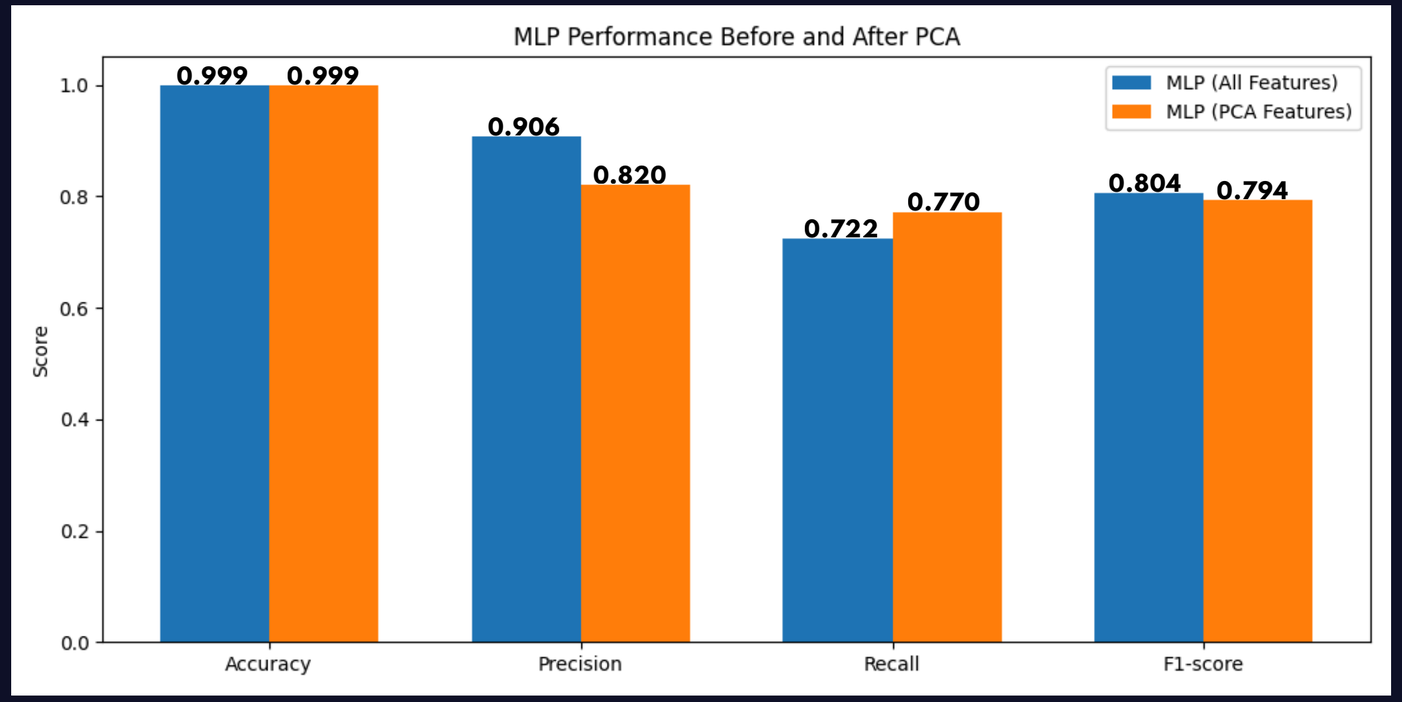
XGBoost performs exceptionally well even without PCA, showing high accuracy and precision. However, the recall value suggests that the model might miss some positive fraud cases, which is why optimizing the threshold becomes crucial.

By applying PCA, the number of features is reduced from 30 to 27, which slightly reduces accuracy, precision, and recall. However, the reduced dimensionality helps speed up the model, with processing time improving marginally.

MLP Comparison: Performance Before and After PCA

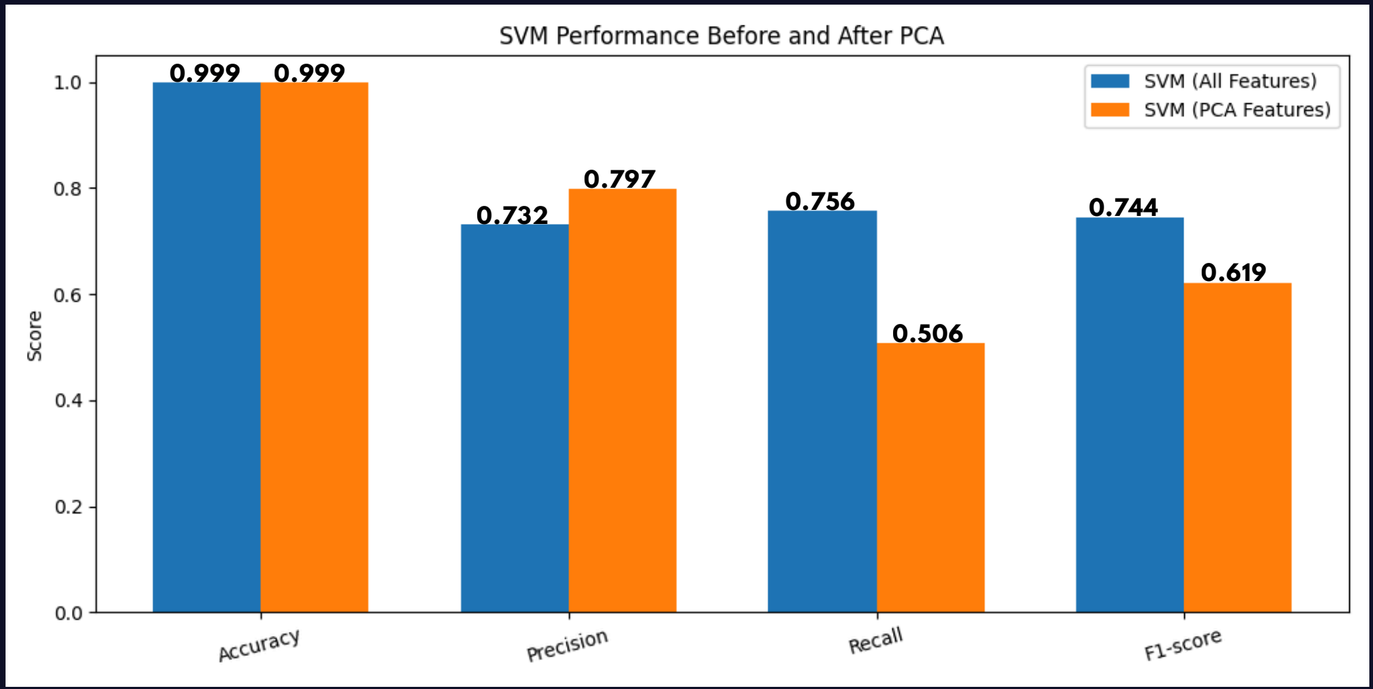
MLP learns patterns through layers of neurons to separate fraud from normal transactions. It tries to capture complex, nonlinear relationships in the data so it can detect subtle fraud signals.

PCA slightly lowers precision but improves recall, meaning it catches more fraud cases. MLP already learns feature combinations internally, so PCA does not speed it up much, but the overall performance remains strong.



Processin time (MLP without PCA): 37.83 seconds
Processing time (MLP with PCA): 41.02 seconds

SVM Comparison: Performance Before and After PCA



Processing time (SVM without PCA): 106.34 seconds

Processing time (SVM with PCA): 109.07 seconds

SVM tries to find a decision boundary (hyperplane) that best separates fraud from normal transactions by maximizing the margin between the two classes.

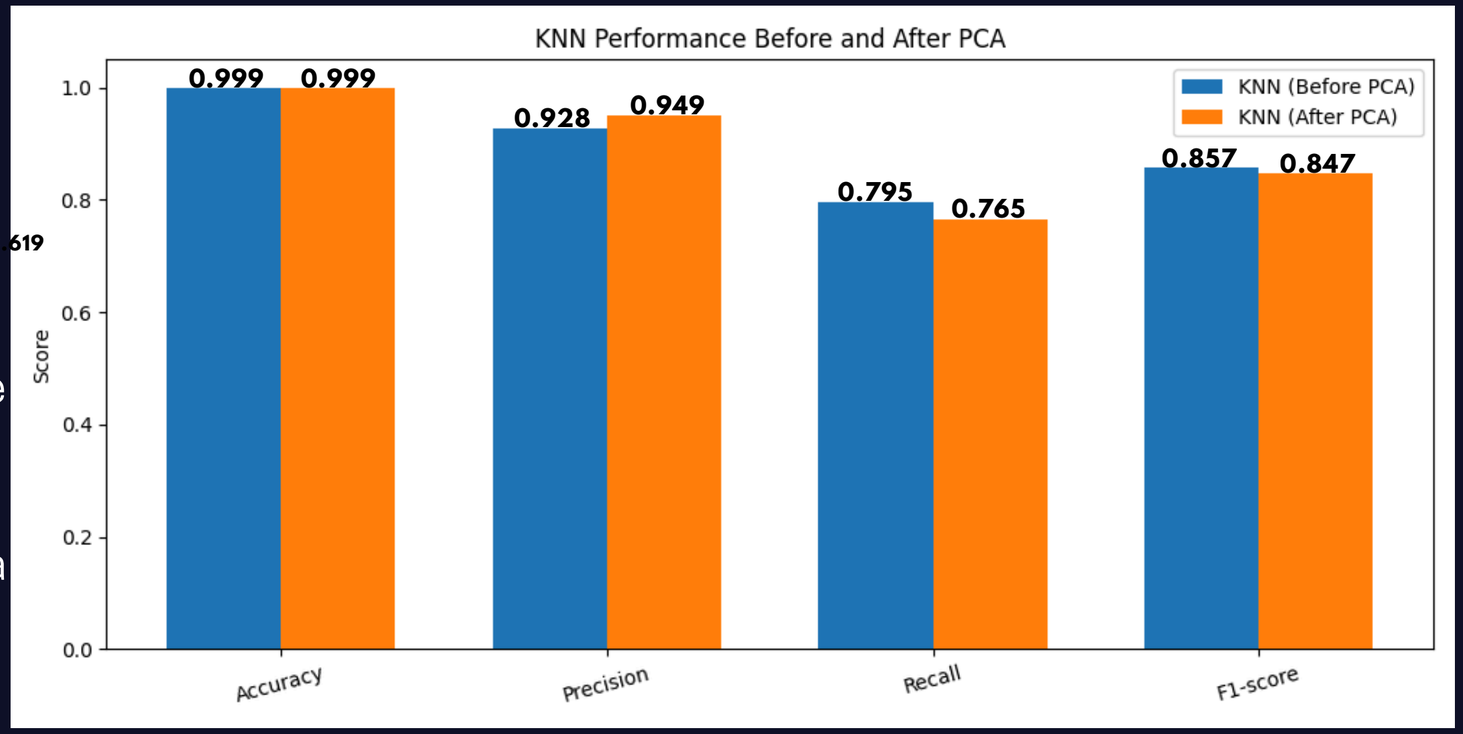
Using all features, SVM has a more balanced precision and recall, giving an F1-score of about 0.74.

With PCA (95% variance), precision increases (fewer false alarms), but recall drops a lot, meaning the model misses more fraudulent transactions. Overall, the PCA version has a lower F1-score and similar (even slightly higher) processing time, so PCA does not improve SVM in this setup.

KNN Comparison: Performance Before and After PCA

KNN classifies each transaction by looking at the closest data points in the feature space. If most of the nearest neighbors are fraud, it predicts fraud; if not, it predicts normal. It relies on similarity rather than learning a model.

After PCA, the model becomes faster because distance calculations happen in fewer dimensions. Precision improves slightly, meaning fewer false alarms, but recall drops a bit, meaning it misses more fraud cases. Overall performance remains strong with only small changes.



Processing time (KNN BEFORE PCA): 11.05 seconds

Processing time (KNN AFTER PCA): 8.57 seconds

2. MISSING VALUE IMPUTATION

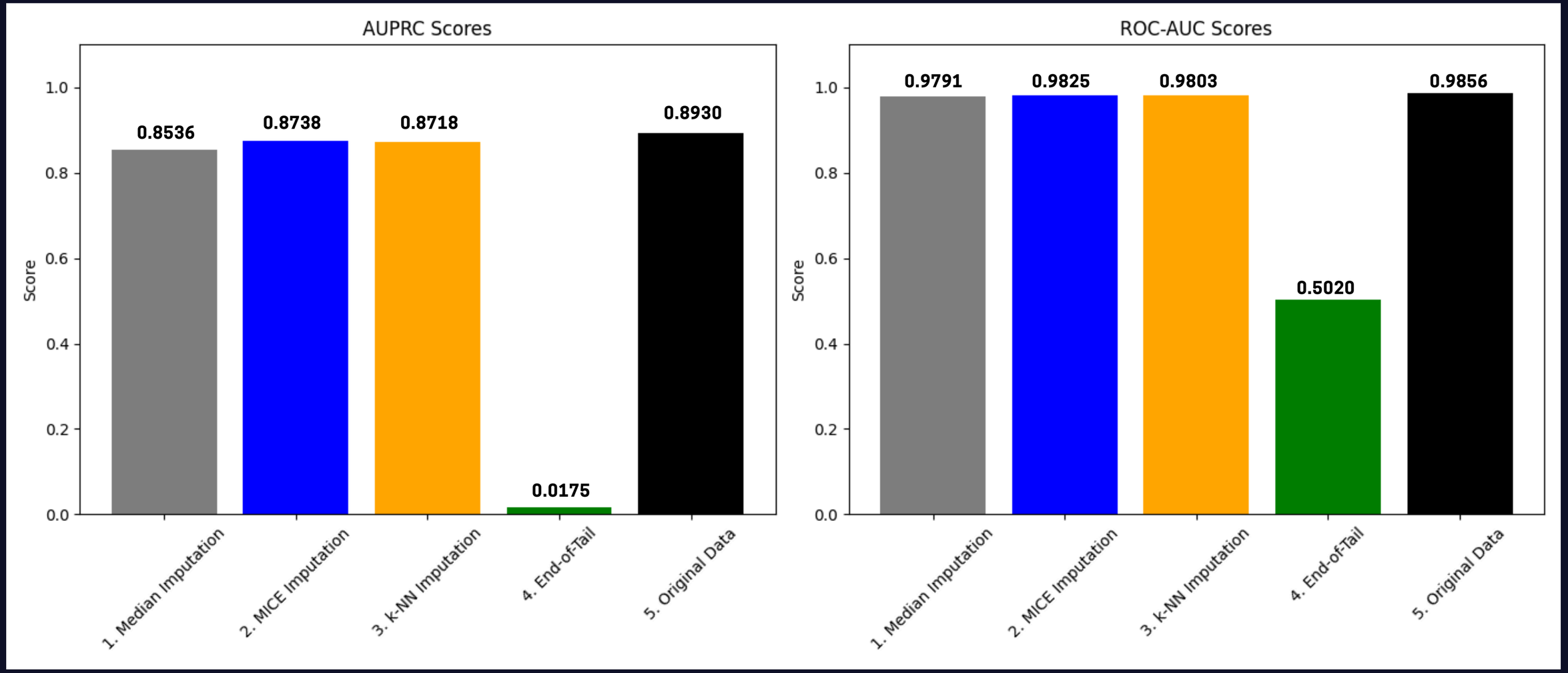
Missing value imputation is a preprocessing technique used to handle incomplete data instead of dropping rows. In this project, we artificially introduced missing values into 15% of the feature matrix and compared several imputation strategies combined with Logistic Regression. The goal is to see which method best recovers the signal and maintains good fraud-detection performance using AUPRC and ROC-AUC.

Imputation Steps Used:

- Introduce missing values (15%)
- Split into train/test
- Apply 4 methods: Median, MICE, k-NN, End-of-Tail
- Standardize → Train Logistic Regression
- Evaluate with AUPRC & ROC-AUC

Used Methods:

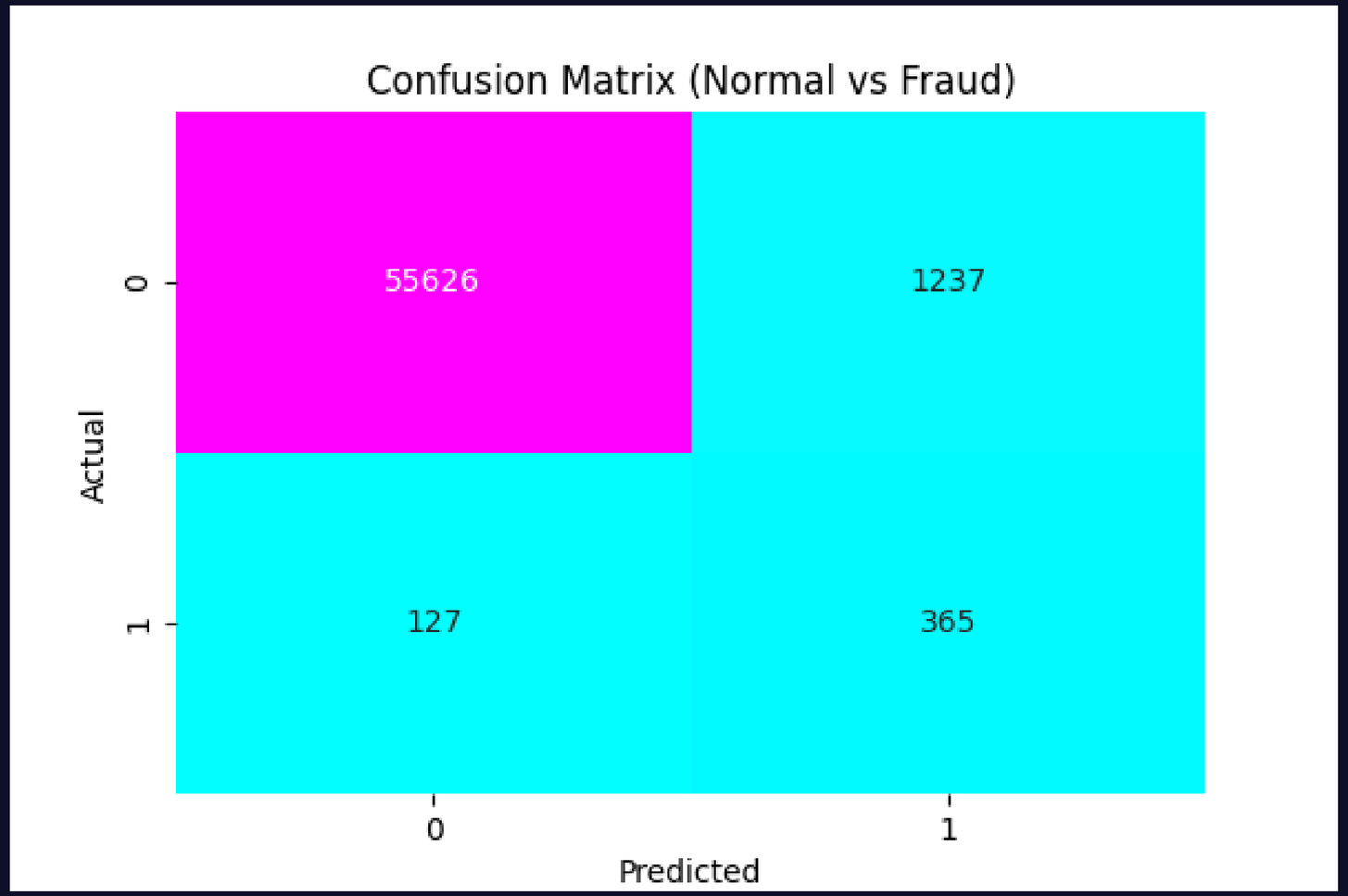
- Median Imputing
- k-NN Imputing
- MICE (Multivariate Imputation by Chained Equations)
- End-of-Tail


AUPRC Scores
Score
1.0
0.8
0.6
0.4
0.2
0.0
0.8536
0.8738
0.8718
0.0175
0.8930
1. Median Imputation
2. MICE Imputation
3. k-NN Imputation
4. End-of-Tail
5. Original Data
ROC-AUC Scores
Score
1.0
0.8
0.6
0.4
0.2
0.0
0.9791
0.9825
0.9803
0.5020
0.9856
1. Median Imputation
2. MICE Imputation
3. k-NN Imputation
4. End-of-Tail
5. Original Data

3. OUTLIER DETECTION

- Outlier detection (sometimes used interchangably with Novelty detection) is an **unsupervised learning** technique for detecting outliers, after training the model on a **clean** dataset.
- In this project; we trained the model on a sub-dataset that only consists of **Normal** credit card transactions, and tested using a dataset with both **Frauds and Normals** mixed in.
- The algorithm we chose was **Isolation Forest**: as it can be understood by its name, this algorithm is best for pointing out the **abnormal** ones in a vast majority of **normals**. (Fraud and Normal)

    - **Evaluation Metrics: Recall and Precision; since our dataset is very imbalanced, using accuracy would be misleading. Hence we used recall and precision, which judge based on the correctly classified.**



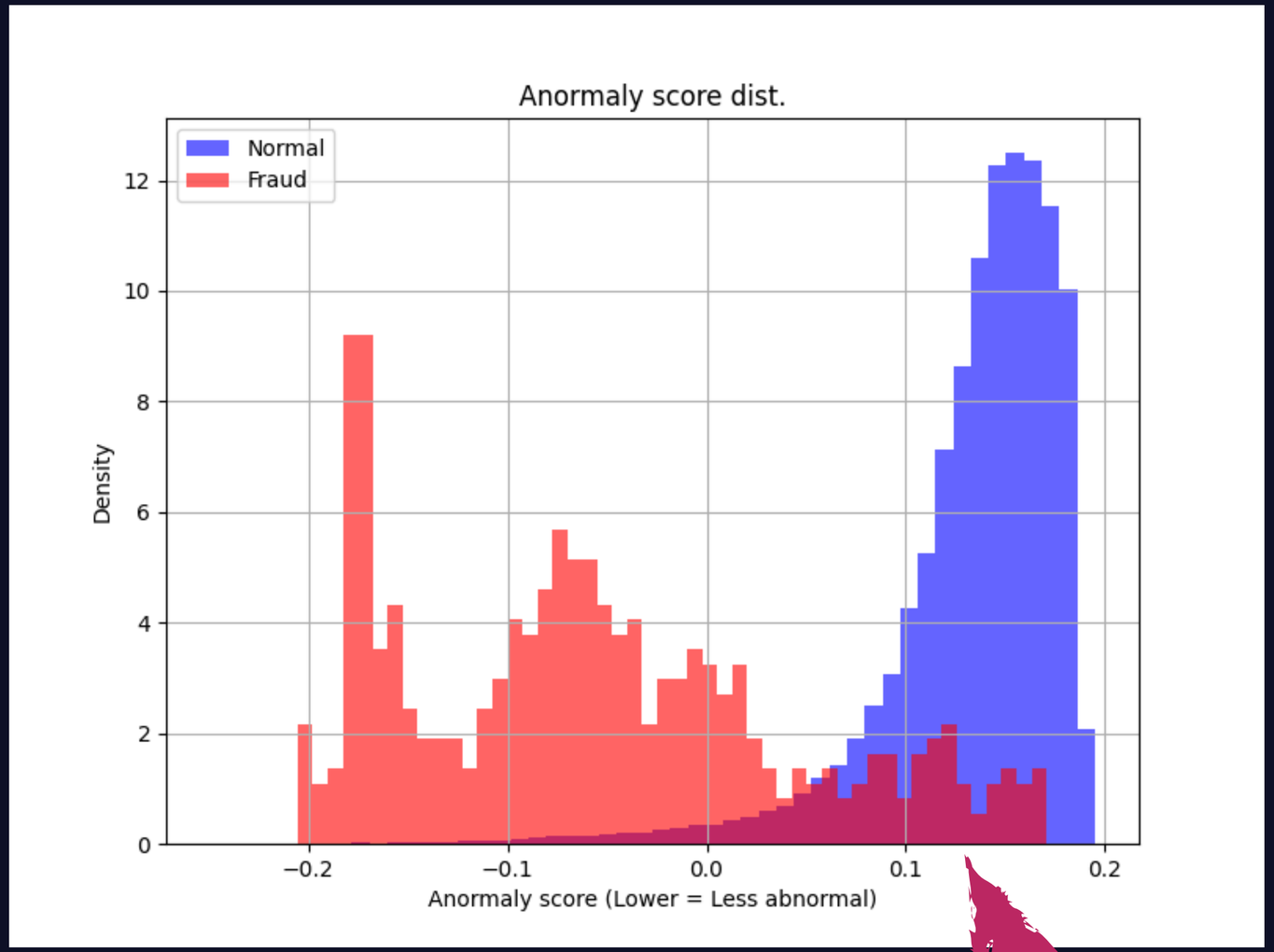
for reference

True Negatives (TN)	False Negatives (FN)
False Positives (FP)	True Positives (TP)

Classification report of outlier

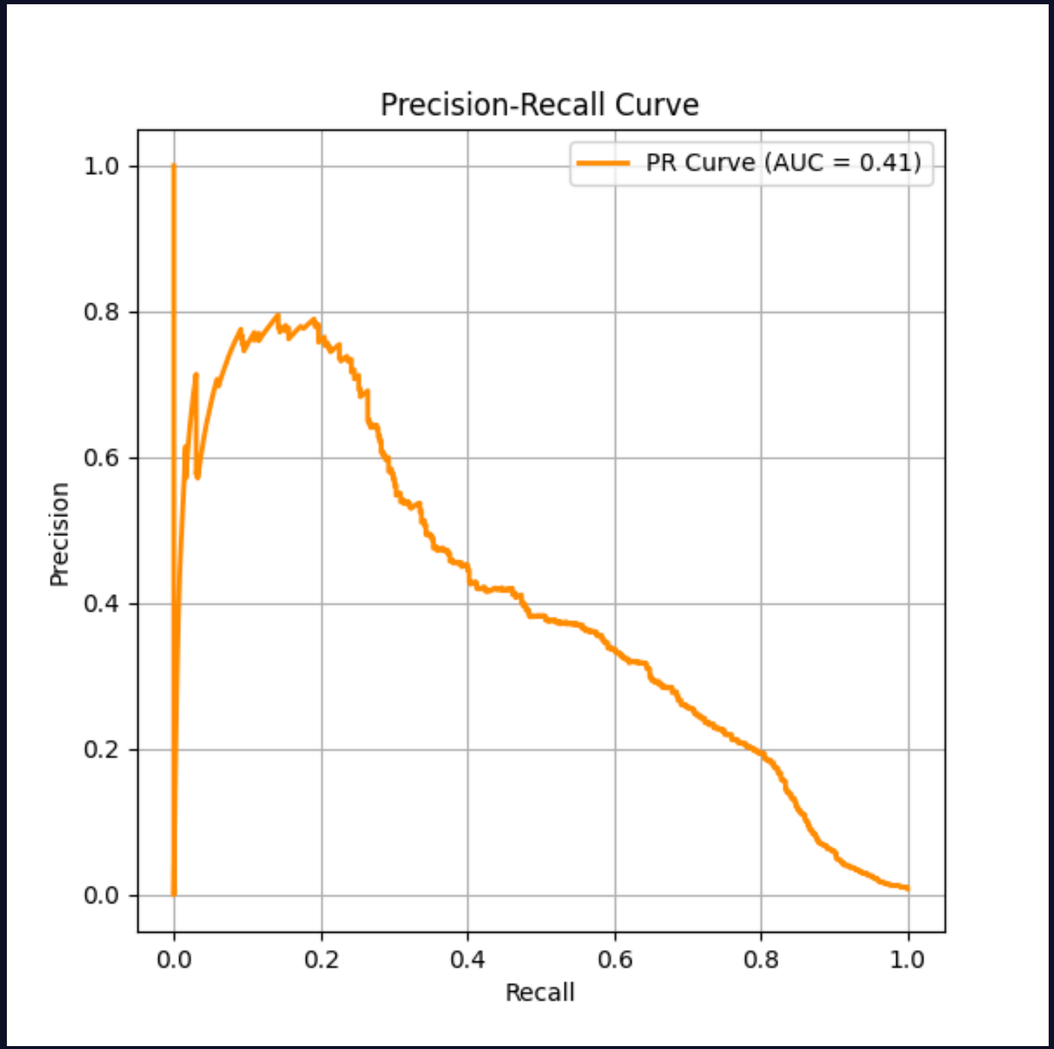
--- Classification Report ---

	precision	recall	f1-score	support
Normal	1.00	0.98	0.99	56863
Fraud	0.23	0.74	0.35	492
accuracy			0.98	57355
macro avg	0.61	0.86	0.67	57355
weighted avg	0.99	0.98	0.98	57355



less overlap here means model is doing good at classifying

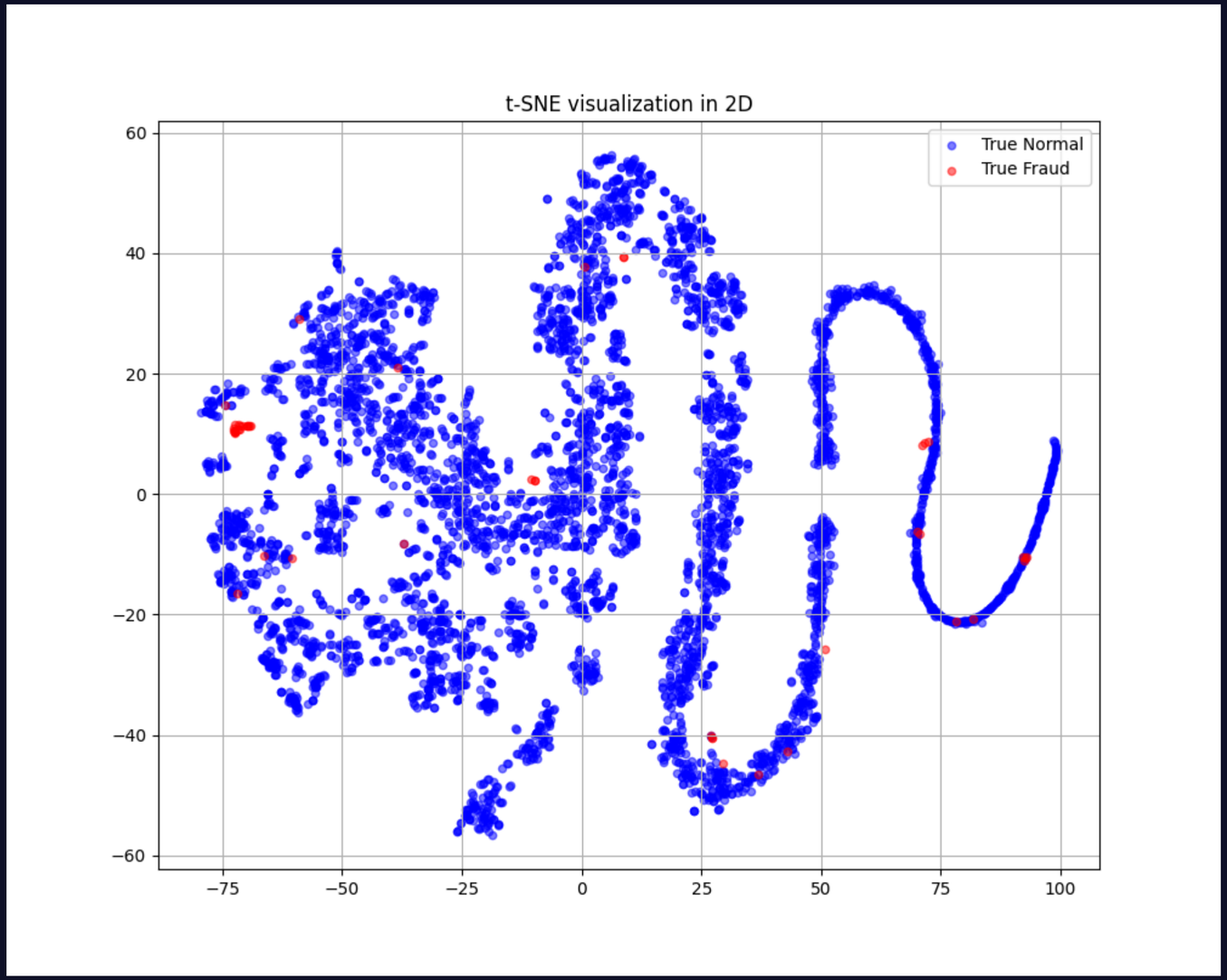
The model’s success curve



$$P = \frac{T_p}{T_p + F_p}$$

$$R = \frac{T_p}{T_p + F_n}$$

Visualization of 30D to 2D



4. SMOTE

SYNTHETIC MINORITY OVERSAMPLING TECHNIQUE

SMOTE (synthetic minority oversampling technique) is one of the most commonly used oversampling methods to solve the imbalance problem. It aims to balance class distribution by randomly increasing minority class examples by replicating them. SMOTE synthesises new minority instances between existing minority instances. It generates the

SMOTE Steps Used:

- Split data (train/test)
- Apply SMOTE on training data only
- Generate synthetic fraud samples
- Rebuild balanced training set
- Train model
- Evaluate on original test set

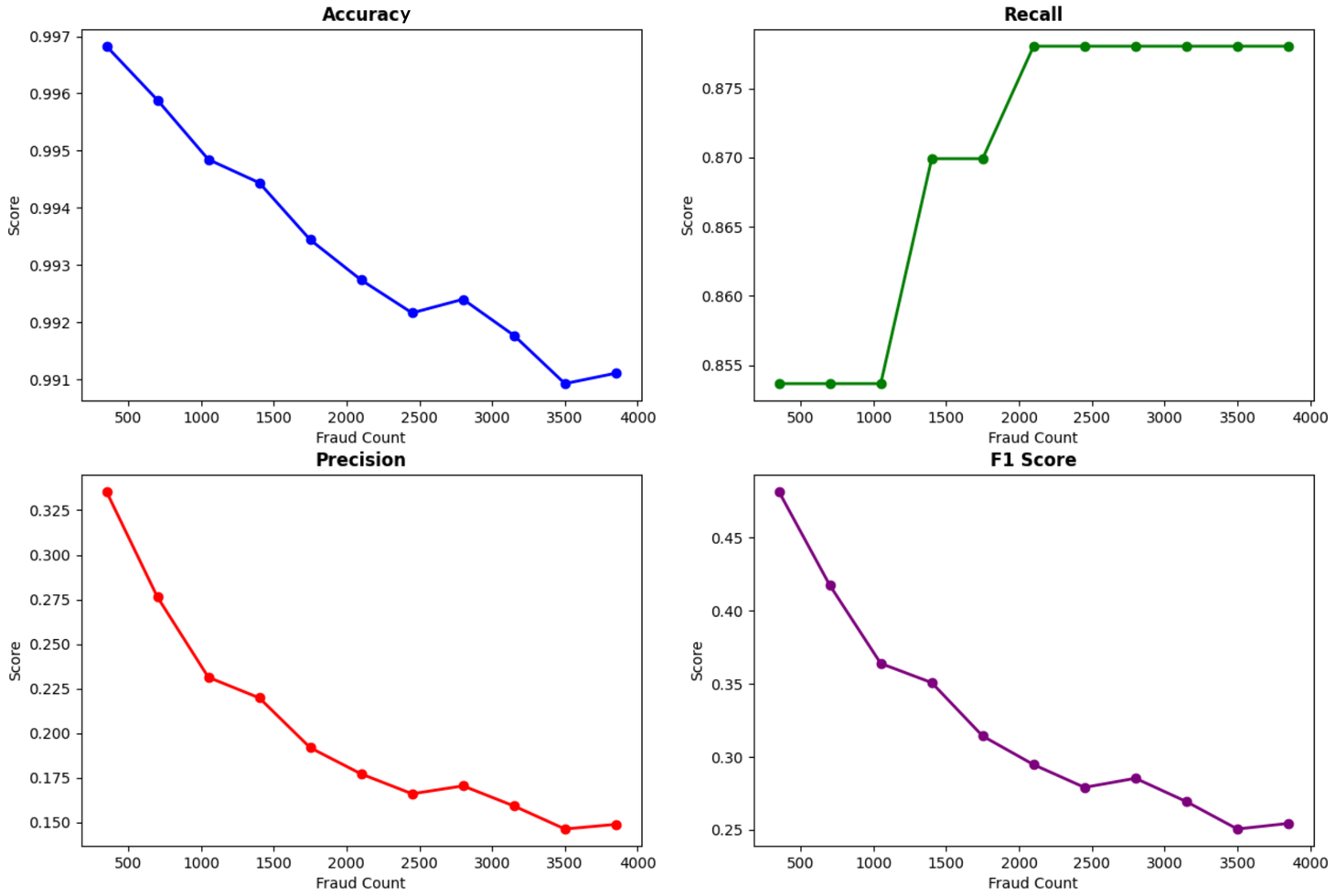
Used Models and SMOTEs

- Models
  - Logistic Regression
  - Random Forest
- SMOTEs
  - SMOTE
  - Borderline-SMOTE

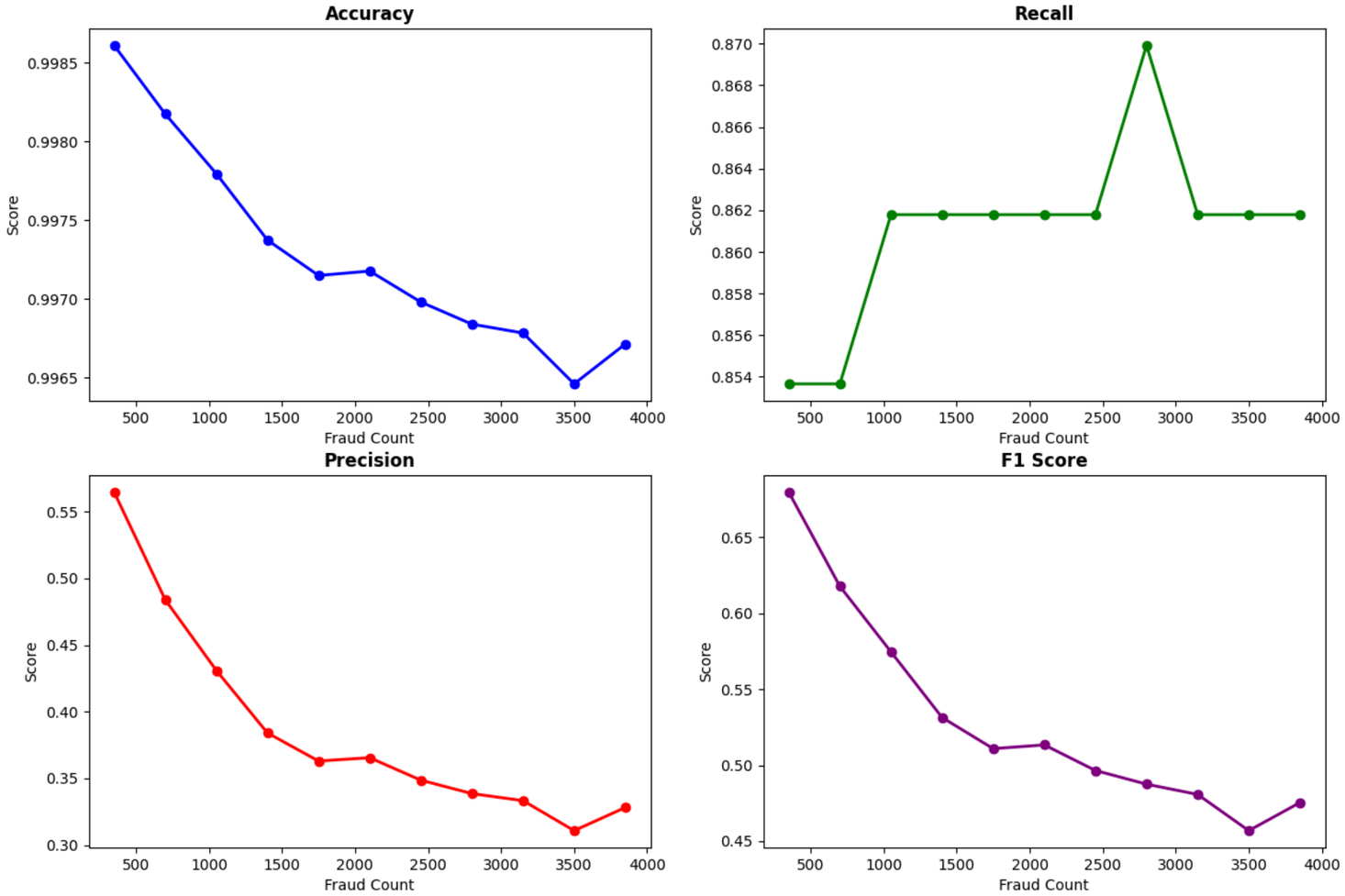
```python
from imblearn.over_sampling import SMOTE
from collections import Counter

You, 3 days ago | 1 author (You)
class SMOTEModel:
    def __init__(self, sampling_strategy='auto', random_state = 42):
        self.sampling_strategy = sampling_strategy
        self.random_state = random_state
        self.smote = SMOTE(sampling_strategy=self.sampling_strategy, random_state=self.random_state)

    def fit_resample(self, X, y):
        resamples = self.smote.fit_resample(X, y)

        X_resampled = resamples[0]
        y_resampled = resamples[1]
              You, 3 days ago • Finished this part of the project …
        return X_resampled, y_resampled

    def set_sampling_strategy(self, target_count, normal_count = 5000):
        ratio = target_count / normal_count

        self.sampling_strategy = ratio

        self.smote = SMOTE(sampling_strategy=self.sampling_strategy, random_state=self.random_state)
        print(f"New normal/fraud ratio: {ratio:.3f}, new fraud count: {target_count}")
```

```python
# Dataset reading and splitting into train and test
whole_set = pd.read_csv("task_SMOTE/creditcard.csv")
handler = dataset_handler(dataframe = whole_set)

X, y = handler.get_features_and_labels(handler.data)

X_train_pool, X_test, y_train_pool, y_test = train_test_split(X, y, test_size=0.25, random_state=42, stratify=y)

ds_pool = pd.concat([X_train_pool, y_train_pool], axis=1)

# Get all the frauds and normals
pool_normals = ds_pool[ds_pool['Class'] == 0]
pool_frauds = ds_pool[ds_pool['Class'] == 1]

# Select a number of fraud and normals for training
selected_normals = pool_normals.sample(n=5000, random_state=42)
selected_frauds = pool_frauds.sample(n=350, random_state=42)

df_train_final = pd.concat([selected_normals, selected_frauds])
df_train_final = df_train_final.sample(frac=1, random_state=42).reset_index(drop=True)

X_train = df_train_final.drop(['Class', 'Time'], axis=1)
y_train = df_train_final['Class']

X_loop_train = X_train
y_loop_train = y_train

fraud_count = y_train[y_train == 1].count()
```

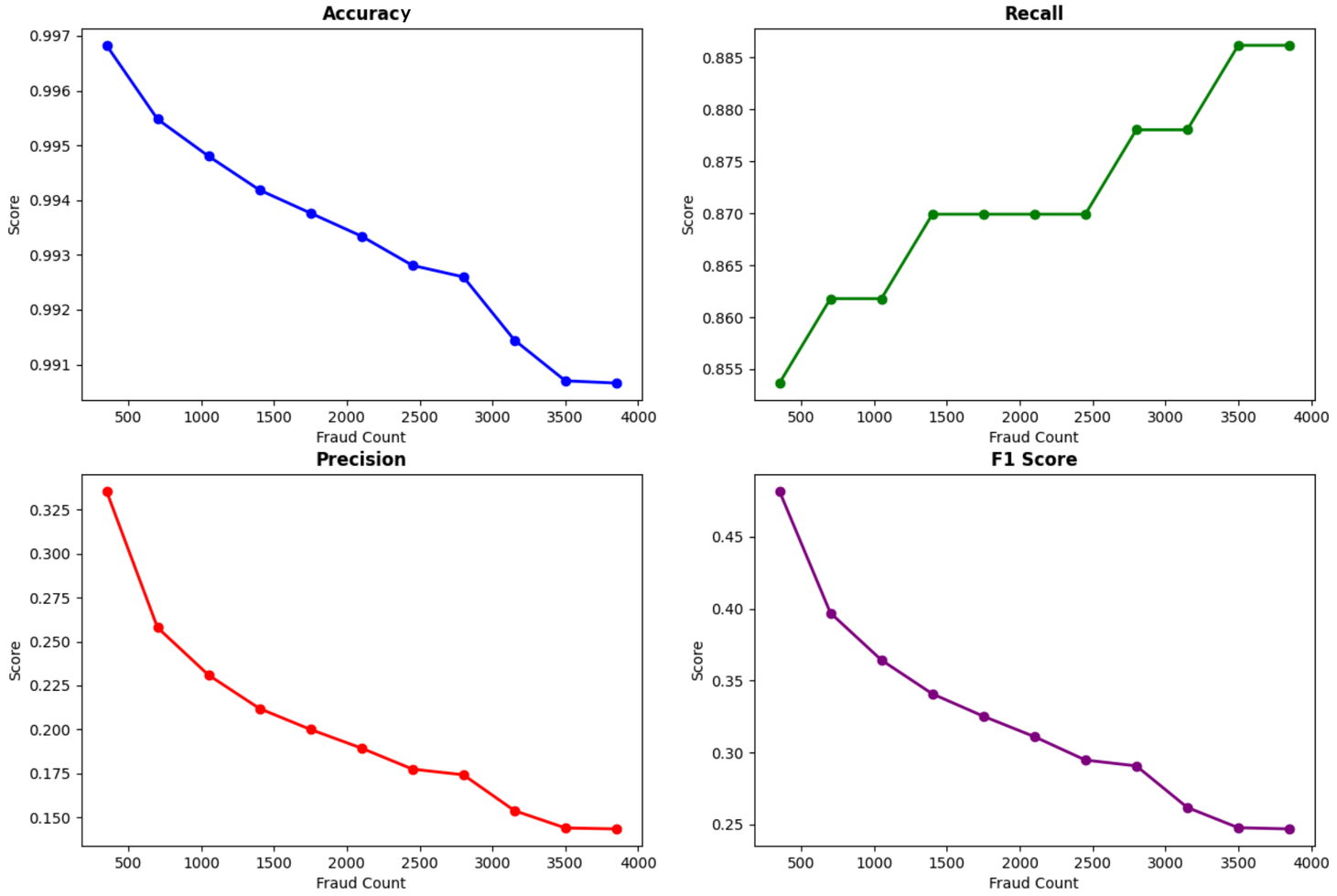
PLOT 1: LOGISTIC REGRESSION + SMOTE



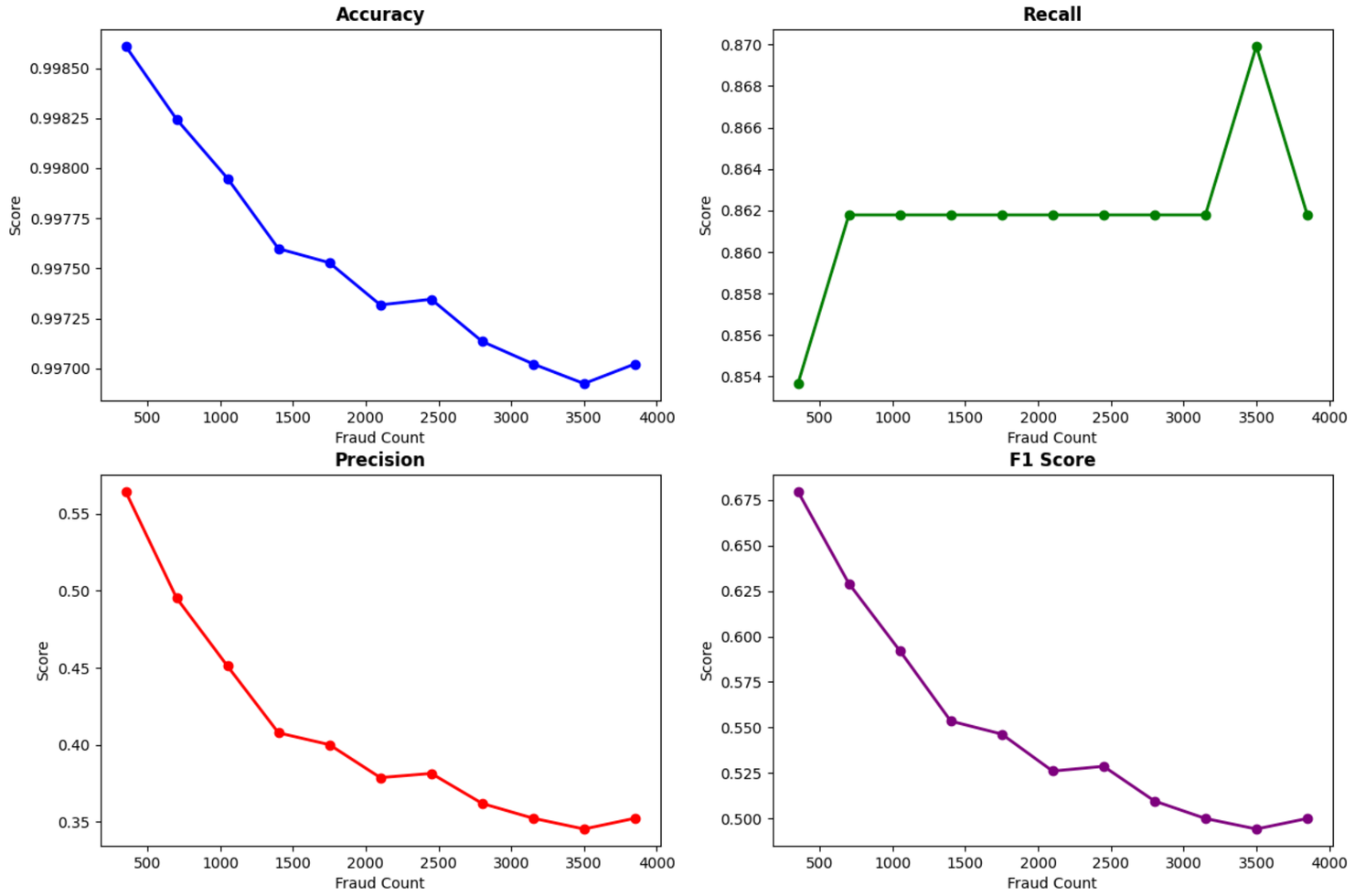
PLOT 2: RANDOM FOREST + SMOTE



PLOT 3: LOGISTIC REGRESSION + BORDERLINE-SMOTE



PLOT 4: RANDOM FOREST + BORDERLINE-SMOTE



THANK YOU