# The use of Machine Learning (ML) Algorithms and Feature Engineering to Predict Fraudulent Transactions.

December 2024

**Module**: D100 & D400

**Candidate Number (BGN)**: 3363D

**Deadline Date**: 19/12/2024 12:00PM

*I confirm that this is entirely my own work and has not previously been submitted for assessment, and I have read and understood the University's and Faculty's definition of Plagiarism*

**ChapGPT 4o was used for the following:**

- Generating ReadMe documentation.
- Tutorials on how to generate pdp plots using matplotlib instead of plotly.
- Tutorial on how to add print statements during unit tests.
- Help with generating configuration files: config, pyproject, environment ...

**Actual Word Count:** 1700

## 1 Introduction

Fraudulent transactions are a major issue in the UK, with over 1.5 million cases and £570 million stolen in the first half of 2024 [2]. In 2022, there were 45.7 billion payments recorded [1]. With an unprecedented number of transactions, we must explore advanced algorithms and unique feature transformations to flag fraud both accurately and quickly. This project focuses on the use of various transformation pipelines, GLMs and LGBMs ML algorithms and hyper-parameter tuning to obtain effective model specifications. My final, tuned model obtains an accuracy score of 96%. A huge volume of transaction data posed a huge constraint in my project and this is something that these algorithms must combat when being deployed in the real world. I use feature importance and partial dependence plots (PDPs) to understand the effect of the various features on the model. A conventional data science repository structure has been used for production, which can be found in **appendix .1**.

# 2 Dataset

The dataset used for this project can be found here[1]. This dataset includes: `transactions_data.csv`, `cards_data.csv` and `users_data.csv`. There were no keys to link `users_data.csv` and `transactions_data.csv`, therefore only a merge between `cards_data.csv` and `transactions_data.csv` was used for the final dataset. The contents of which can be found in **Appendix .2**. Over 13,000,000 transactions, made from over 6,000 cards, were recorded. However, the fraud status is only recorded for 8,901,631 of these. Out of which, only 13,332 are recorded as fraudulent, leading to a rather imbalanced dataset. The package comes with a command line interface (CLI) application that can be used to load the data, once the package has been installed:

1. Run `$ frauddetection --download` to download the dataset from kaggle.
2. Run `$ frauddetection --merge` to merge the transaction and card csv's.
3. Run `$ frauddetection --reduce` to create a reduced data, which is balanced for fraudulent and non-fraudulent transactions.

The Cambridge CSD3 (high-powered computing cluster) was used for the pre-processing and merging as my local machine was not capable. Consequently, a balanced data set called `balanced_data.pkl` comes with the installation. This contains an equal number of fraudulent and non-fraudulent samples and can be found in the `/data` folder. Descriptions for the dataset columns were missing but could be inferred easily. Columns, percentage of missing values and number of unique values can be found in **appendix .2**

# 3 Methodology

## 3.1 Pre-processing

The out-of-the-bag dataset was relatively clean, with a few points to note:

1. Numerical features such as `amount` and `credit_limit` had a $ and thus were coded as objects.
2. Date-like features were also coded as objects.
3. around 12% of `merchant_state` and `zip` were missing.
4. around 98% of `errors` was missing. But this is likely because there was no transaction error.
5. The dataset is varied with a lot of unique values per feature. See **Appendix .2**.

The first step was to deal with the missing values. As this is a fraudulent transaction dataset, missing values raised suspicion and, as categorical encoding was used, these missing values were given their own category. The next step was general pre-processing transformations. In this project, both general transformations and feature engineering were done through scikit-learn pipelines and custom transformers, to streamline the data-wrangling process. These can be found under `Modules/transforming.py` and `Modules/Pipelines.py`. The general transformation pipeline transformers are as follows:

[1] `https://www.kaggle.com/datasets/computingvictor/transactions-fraud-datasets`

| Transformer | Description |
|---|---|
| `Target0_reducer` | Returns a data frame with only a fraction of non-fraudulent samples, appended randomly to all 13,332 fraudulent samples. |
| `TargetBinary` | Encodes 'yes' and 'no' as 1 and 0. |
| `fillna_transformer` | Encodes missing values as their own category |
| `time_series_pipeline` | Returns an ordered time series of date-like features. |
| `DollarToInt` | Features in $ converted from object to int. |

Table 1: General Transformation Pipeline

### 3.2 Exploratory Data Analysis (EDA) & Feature Engineering

Once general transformations were carried out, the next step was to encode variables, through so-called exploration pipelines. The pre-built `OneHotEncoder` was used to apply one-hot encoding to columns with few features: `use_chip`, `card_brand` and `card_type`. `MinMaxScaler` was used to scale `amount` and `credit_limit`. For all other categories a custom target encoder, `CustomTargetEncoder`, was used. Each category in the testing set would be mapped by the mean number of fraudulent transactions for that category in the training set. If a category did not appear in the training set but appeared in the testing set it would be encoded by the overall target mean of the training set. Unit tests for `Target0_reducer` and `CustomTargetEncoder` can be found in `/tests/tests_transforming.py`

A total of 3 different start-to-finish transformation pipelines were used for modelling and their performance compared.

1. **Pipeline 1**
   (a) Time series transformations for date-like features
   (b) Fixed non-fraudulent sample size as 1% of total non-fraudulent samples.
2. **Pipeline 2**
   (a) More granular decomposition (hour, day of week, month and year) of date-like features: `date`, `acct_open_date` and `expires`.
   (b) `CustomTargetEncoder` was applied to the decomposed date-like features.
   (c) Fixed non-fraudulent sample size as 1% of total non-fraudulent samples.
3. **Pipeline 3**
   (a) Same transformations as Pipeline 2.
   (b) Proportion of non-fraudulent samples used in date frame for modelling could be varied.

Once features were encoded, we could look at correlations with the target column:

There are some redundant features, therefore within all 3 transformation pipelines I add a custom transformer, `RemoveUncorrFeatures` that drops all columns that have less than a 0.05 correlation with the target variable in the training set.

We can see however that, `date`, `acct_open_date` and `expires` have little correlation with the target variable. I put these findings into question as intuitively we might expect more fraudulent transactions at certain hours of the day, such as late at night. Once these variables are decomposed (for pipeline 2) we see more promising results:

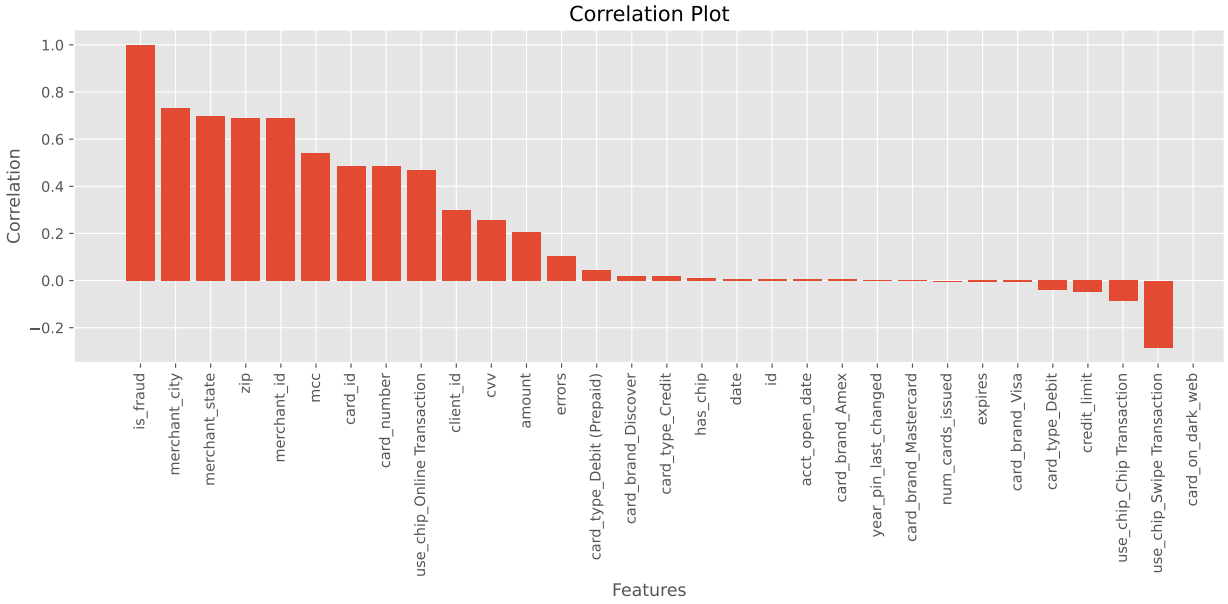


Figure 1: Correlation of target variable with features.

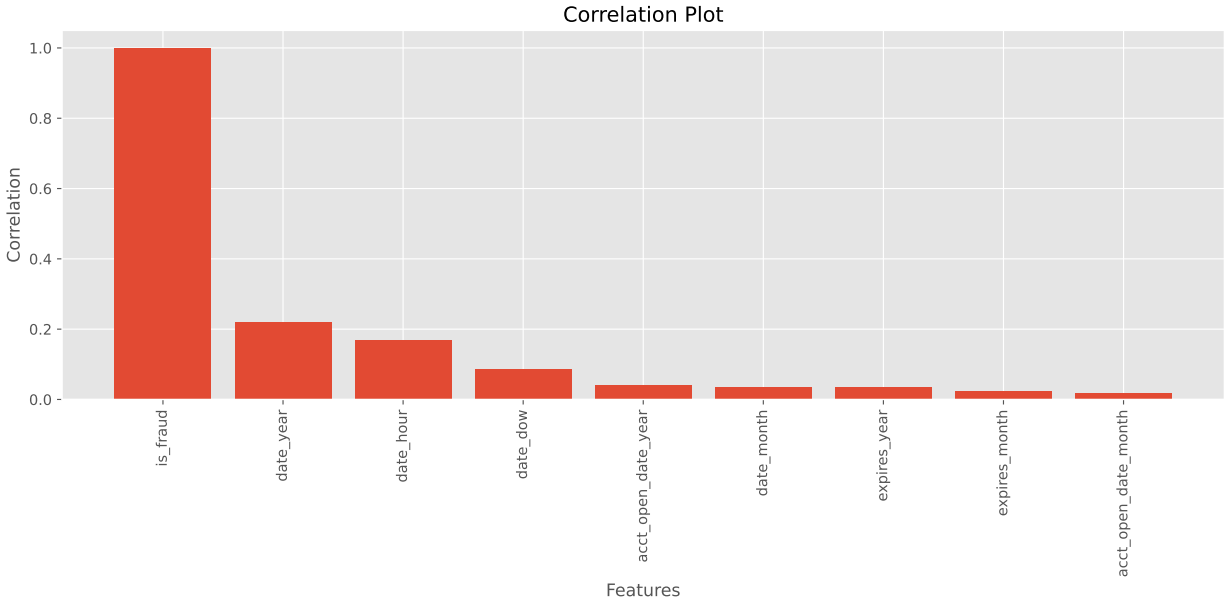


Figure 2: Correlation of target variable with decomposed date-features

The total merged dataset contains 8,901,163 non-fraudulent transactions and only 13,332 fraudulent transactions. Next we look at the distribution of fraudulent transactions across `merchant_city` and `mcc`.

Interestingly, this dataset matches the findings of [1], where the majority of fraudulent transactions are committed online:

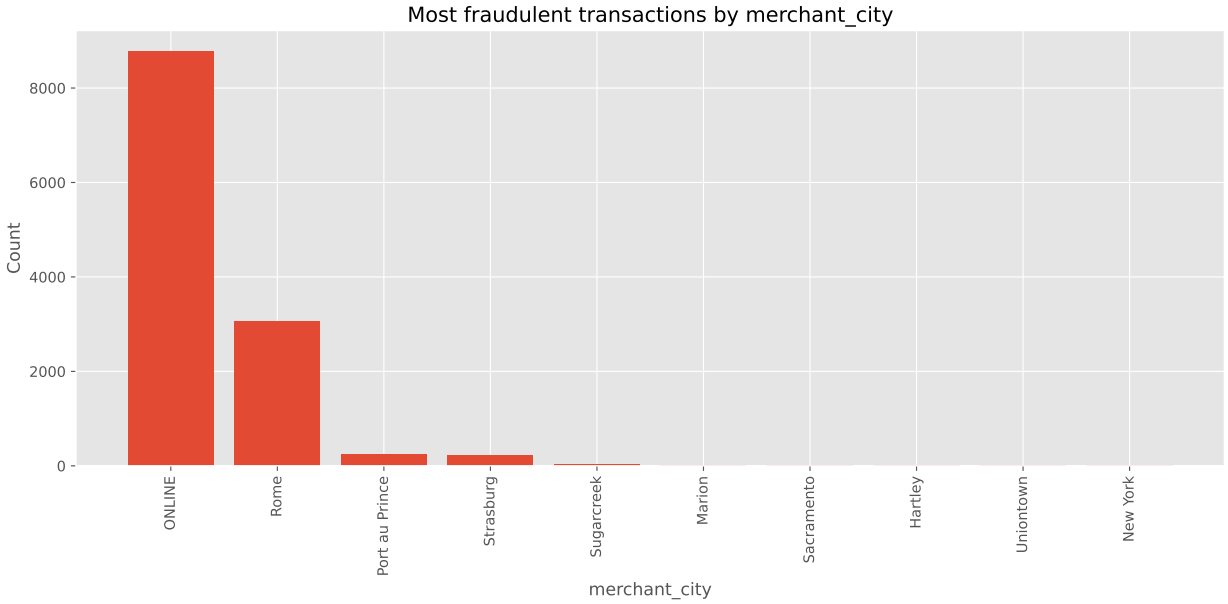


Figure 3: Number of fraudulent transactions by **merchant_city**

We also see the distribution of fraudulent transactions by merchant category code, where department stores have the most fraudulent transactions.

Finally, we can see from **plot 5**, that fraudulent transactions tend to be of a higher value than their non-fraudulent counterparts, with more variation.

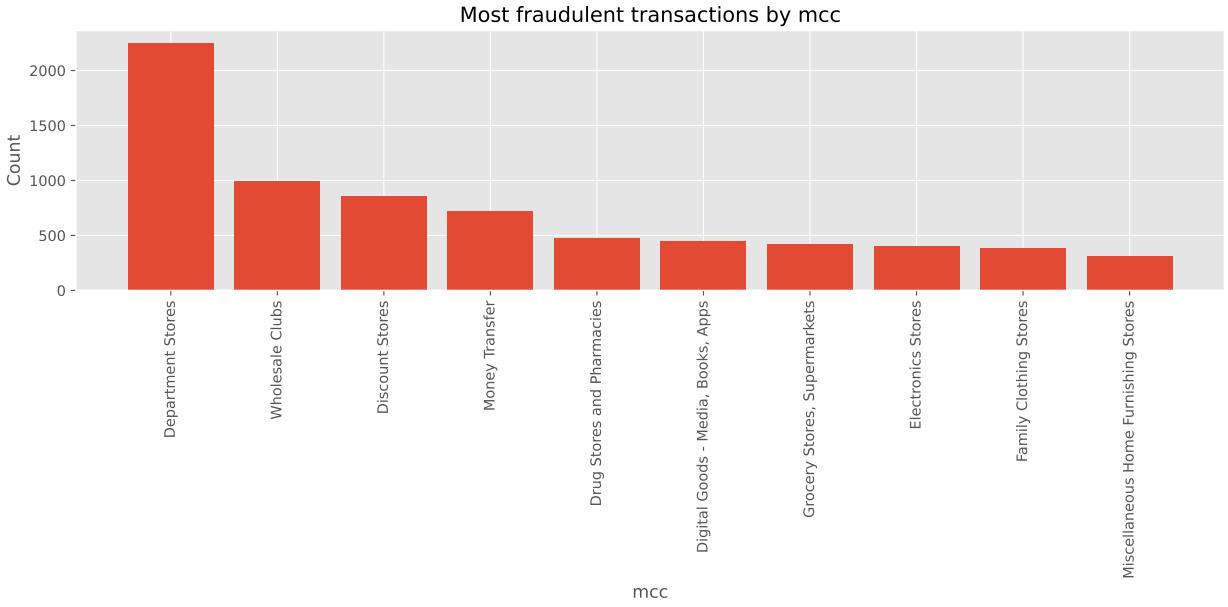


Figure 4: Fraudulent transactions by MCC

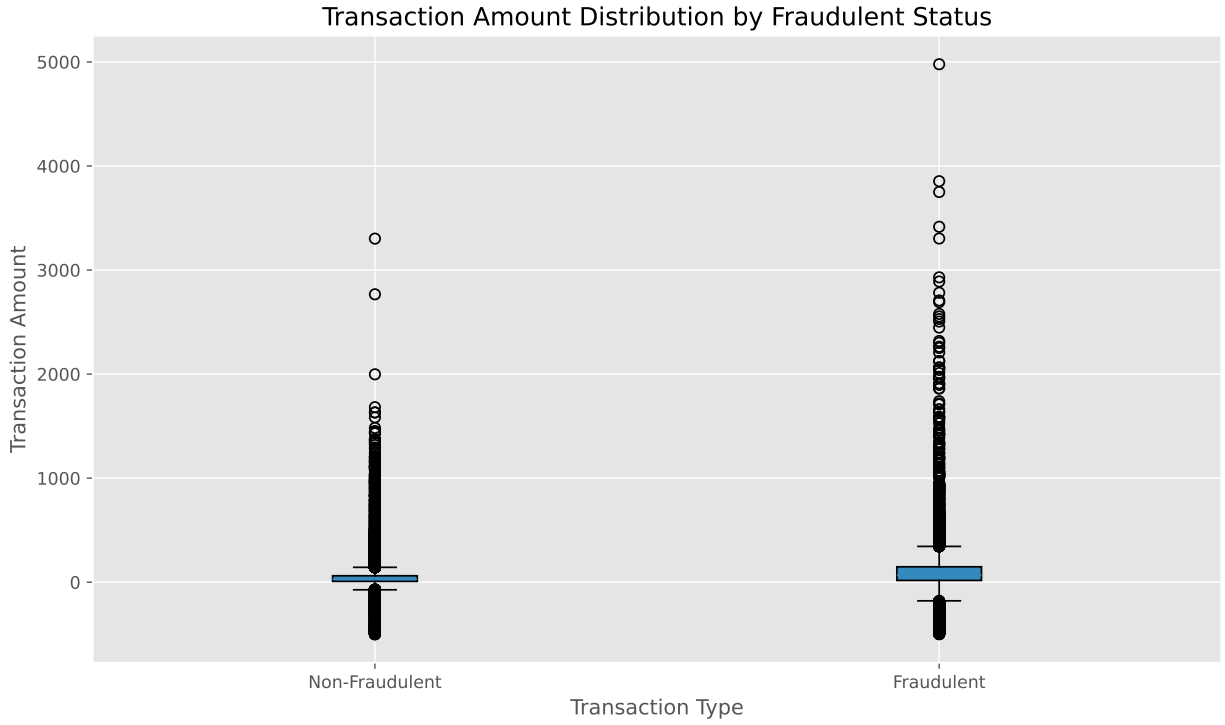


Figure 5: Distribution of transaction amount by fraudulent status.

# 4 Model Training & Results

Please note full model selection was done in `model_selection.ipynb`, however this was carried out using the full `merged_data.pkl`. If the user is not able to obtain this, he/she can replicate the findings of the top models on `balanced_data.pkl` by running `model_training.py`.

As this was a binary classification problem, the following two estimators were used:

1. `LogisticRegression`
2. `LGBMClassifier`

A custom class found in `Modules/modelling.py` called `MLearner` was used to streamline the entire ML process. One can input one of the three transformation pipelines, an estimator, a dictionary of parameters and a scoring metric then call the methods `.fit()` and `.predict()`. The parameter dictionary will be optimised using `GridSearchCV` according to the defined scoring metric. When the parameter dictionary contains only a single set this class reduces to a classic train/ test ML problem. As the dataset was very imbalanced, F1 was used as the desired scoring metric for hyperparameter tuning. When we vary the number of non-fraudulent samples, they become more and more dominant and we see that accuracy increases while F1 falls, as the model misclassifies a larger proportion of the minor class. F1 is a better indicator of how good the model is at identifying the minority class (fraudulent transaction).

The following performance metrics are given by:

$$F_1 = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{1}$$

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}} \tag{2}$$

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}} \tag{3}$$

$$\text{Accuracy} = \frac{\text{True Positives} + \text{True Negatives}}{\text{Total Samples}} \tag{4}$$

First, we compare Pipeline 1 and Pipeline 2 on the testing set. Pipeline 2 seems to perform better, matching our findings in **figure 2**. Using 1% of total non-fraudulent samples means that fraudulent samples make up 13% of the training and test sets. The computational costs of increasing the proportion of non-fraudulent samples is huge, next we see if it brings performance benefits.

| | **Pipeline1** | **Pipeline2** |
|---|---|---|
| **F1** | 0.85 | 0.86 |
| **Precision** | 0.79 | 0.81 |
| **Accuracy** | 0.96 | 0.96 |
| **Recall** | 0.91 | 0.92 |

Table 2: Comparison of Pipeline1 & Pipeline2

We see a general trend here of a fall in Recall, Precision and F1 as we increase the number of non-fraudulent samples and the dataset becomes more imbalanced. Accuracy increases but this is artificial due to an overwhelming majority class. As the general trend was obtained quickly and the computational costs were expensive, using more than 1% of non-fraudulent samples was not tested. Following this experiment, I decided to use a perfectly balanced sample where 13,332 non-fraudulent samples were chosen at random. This also means that accuracy can be used as a valid, more intuitive performance measure.

| **% of non-fraudulent samples** | 0.125% | 0.25% | 0.5% | 1% |
|---|---|---|---|---|
| **F1** | 0.95 | 0.94 | 0.91 | 0.87 |
| **Precision** | 0.94 | 0.93 | 0.89 | 0.81 |
| **Accuracy** | 0.94 | 0.95 | 0.96 | 0.96 |
| **Recall** | 0.95 | 0.94 | 0.93 | 0.93 |

Table 3: Caption

Using the balanced dataset we find the optimum parameters for each estimator to be:

1. `LogisticRegression`[2]
   (a) `C`: 0.89
   (b) `class_weight`: 'balanced'
   (c) `l1_ratio`: 0.77
2. `LGBMClassifier`[3]
   (a) `learning_rate`: 0.1
   (b) `min_child_weight`: 0.001
   (c) `n_estimators`: 1000
   (d) `num_leaves`: 31

However, do not see any difference in performance compared to a standard Logistic Regression with no regularisation.

Next, we look at a performance comparison between `LogisticRegression` and `LGBMClassifier`, with optimised parameters, and see an improvement in using `LGBMClassifier` for all metrics except recall.

If the estimator supports feature importance we can run `.explain()` to obtain feature importance and PDP plots.

[2]Parameter explanations can be found at: `https://scikit-learn.org/1.5/modules/generated/sklearn.linear_model.LogisticRegression.html`

[3]Parameter explanations can be found at: `https://lightgbm.readthedocs.io/en/latest/pythonapi/lightgbm.LGBMClassifier.html`

| | LogisticRegression | LGBMClassifier |
|---|---|---|
| **F1** | 0.95 | 0.96 |
| **Precision** | 0.0.94 | 0.97 |
| **Accuracy** | 0.95 | 0.96 |
| **Recall** | 0.96 | 0.95 |

Table 4: Comparison of LogisticRegression & LGBMClassifier on Pipeline 3.

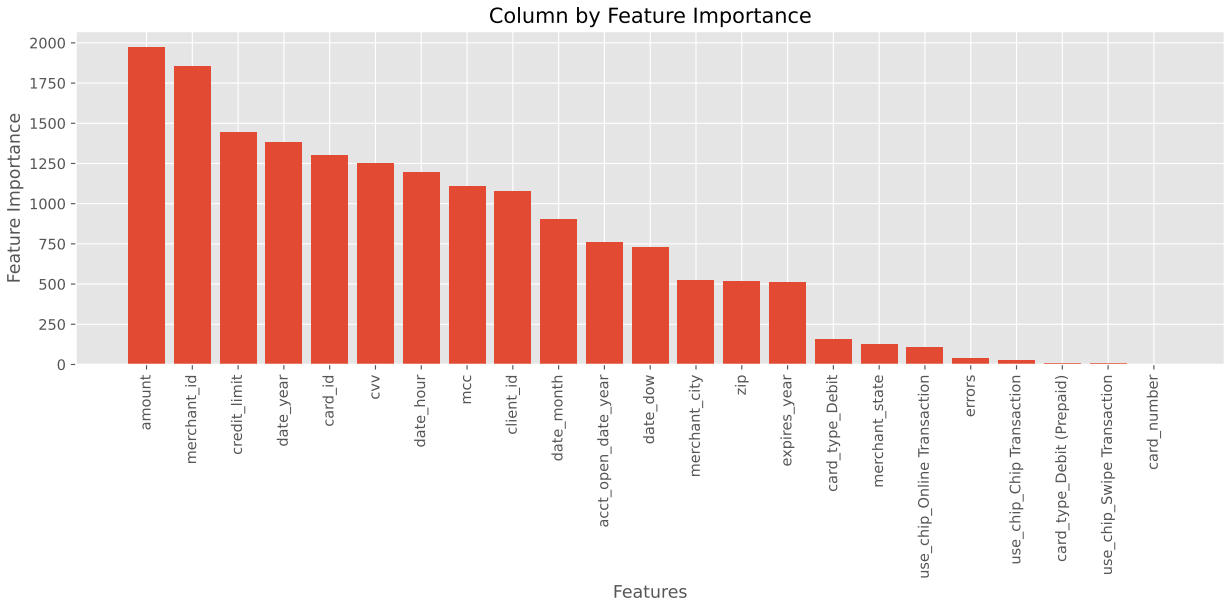


Figure 6: Feature Importance Plots.

Due to a large number of categories and the use of target encoding, it was difficult to generate PDPs for some of the strong predictors such as `merchant_id` and `date_year`. Nevertheless, we see interesting findings for `amount` and `credit_limit`. From **figure 7**, we see that very high credit transactions (>$700) result in a high probability of a transaction being flagged as fraudulent. In addition, debit transactions are also likely to be flagged as fraudulent. Small credit transactions are the least likely to be flagged as fraudulent.

From **figure 8**, we see that transactions were the cardholder has a high credit limit are unlikely to be flagged as fraudulent, this could be due to the fact that high credit limits may only be given out to trustworthy individuals. Lower credit limits result in a high probability of a transaction being flagged.

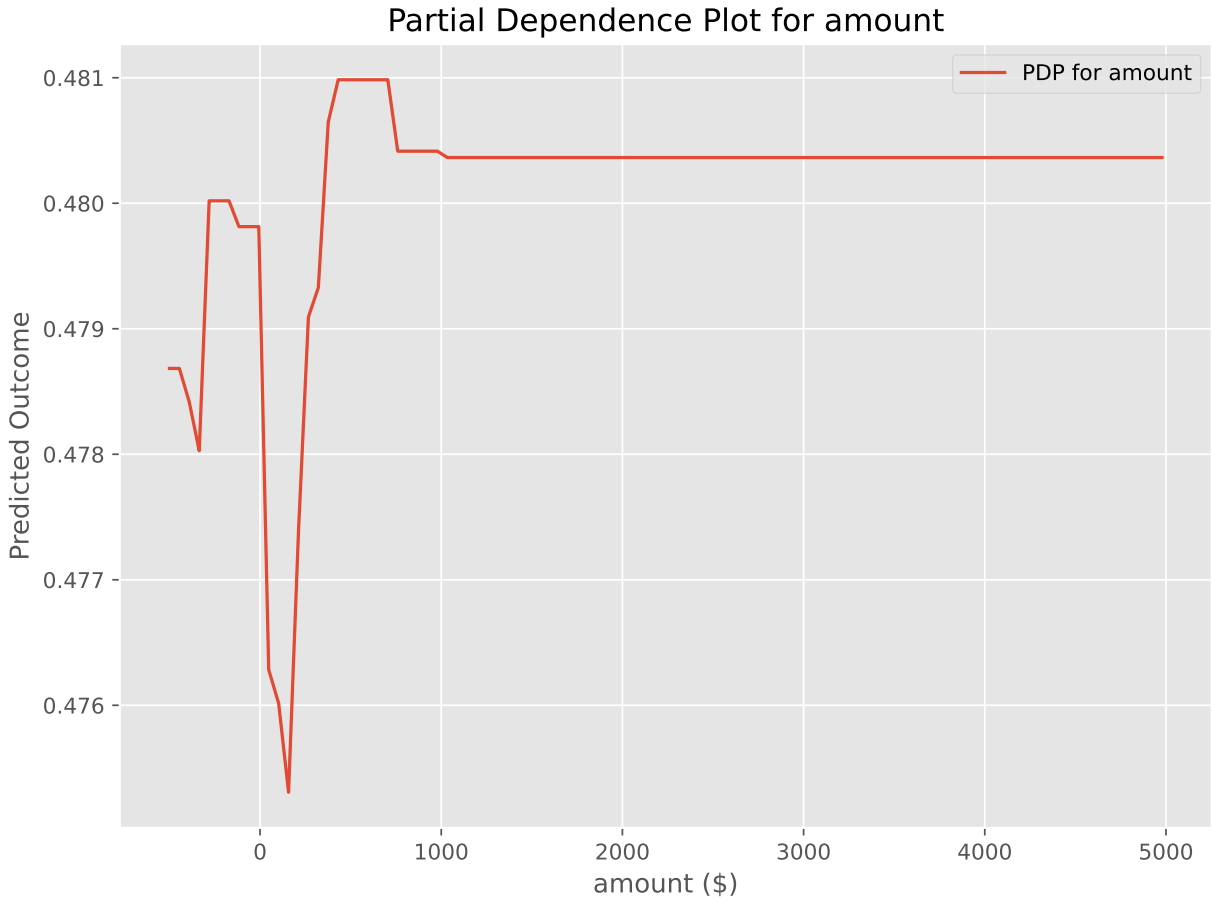


Figure 7: PDP of amount

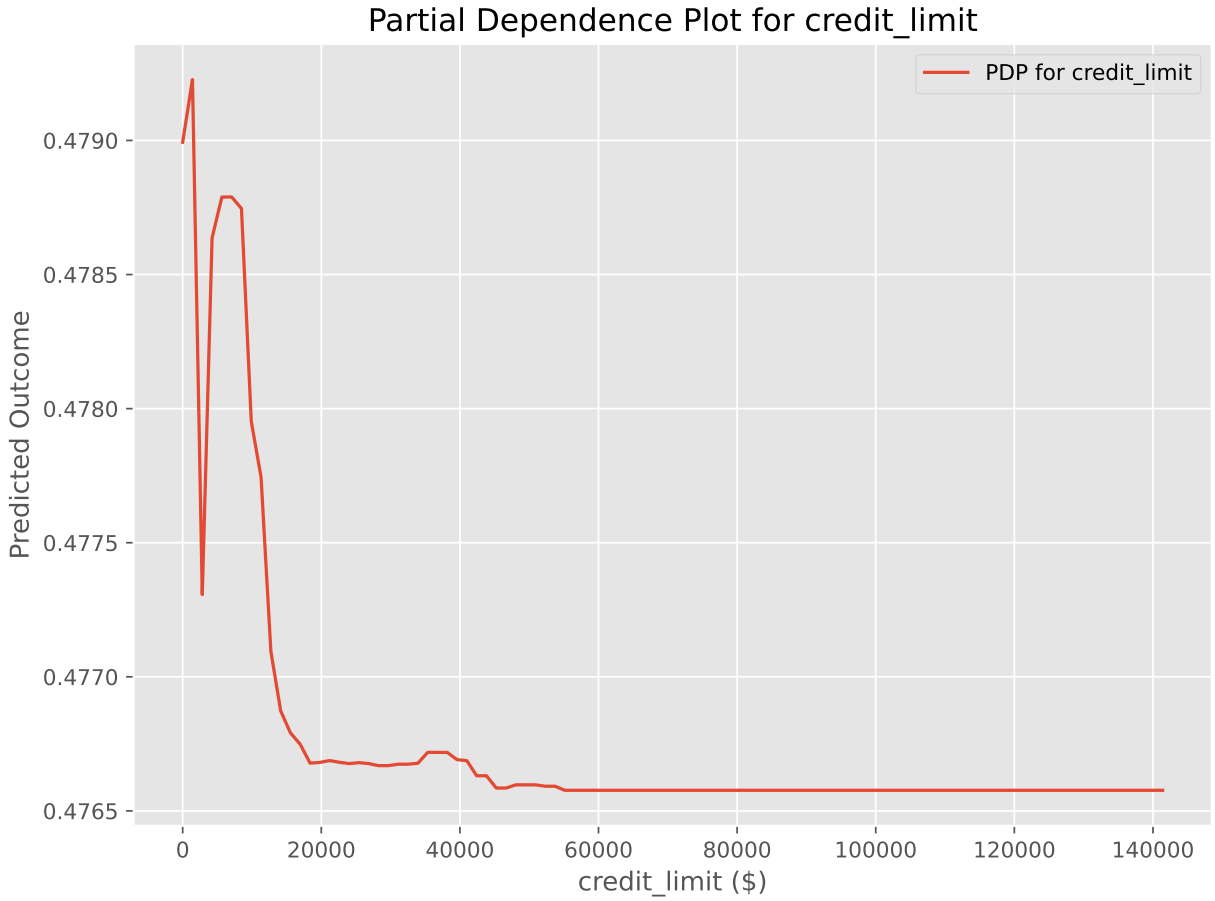


Figure 8: PDP of credit_limit

# 5 Evaluation & Discussion

A large dataset, with `merged_data.pkl` taking 1.4Gb, was a major constraint in this project. Most likely due to the high memory requirements, my local machine could neither merge nor work with large datasets. Consequently, CSD3 had to be used. With more computational resources it would be interesting to experiment with one-hot encoding for all relevant categorical features. This would greatly increase the number of features and so a much larger dataset would have to be used, this could also be optimised. Given more time, I would have liked to experiment with different feature transformations such as interactions and `prev_acct_fraud` that would use the training set to check how many previous fraudulent transactions the card was used in.

Nevertheless, even with target encoding and a smaller, balanced dataset both models were still able to achieve solid performance. PDPs were able to show interesting results too. In hindsight, I would have used one-hot encoding on the coarser date-like column decompositions such as `date_hour` as it would be interesting to see how much more likely a fraudulent transaction is later/ earlier in the day.

I hope for my findings to be a useful starting point in creating fraud detection algorithms.

# References


[1] UK Finance. Uk finance payment markets report 2023 summary. Technical report, 2023. Accessed: 2024-06-17.

[2] UK Finance. Over £570 million stolen by fraudsters in first half of 2024, 2024. Accessed: 2024-06-17.


# 6 Appendix

## .1 Repository structure.

This is the repository structure:

```
.
 Modules
    Pipelines.py
    __init__.py
    cli.py
    load_data.py
    modelling.py
    plotting.py
    preprocessing.py
    transforming.py
 Plots
    amount.pdf
    amount_box_plot.pdf
    corr_plot1.pdf
    credit_limit.pdf
    date_cols_corrs.pdf
    fi_plots.pdf
    mcc_barplot.pdf
    merch_city_barplot.pdf
```

```
ReadMe.md
config.yaml
data
   balanced_data.pkl
eda_cleaning.ipynb
environment.yaml
model_selection.ipynb
model_training.py
pyproject.toml
tests
    __init__.py
    tests_transforming.py
```

## .2 Description of merged dataset

Description of merged dataset:

| Column | Type | % missing | Num unique |
|---|---|---|---|
| client_id | int64 | 0 | 3,680,988 |
| card_id | int64 | 0 | 1219 |
| amount | object | 0 | 4070 |
| use_chip | object | 0 | 70,639 |
| merchant_id | int64 | 0 | 3 |
| merchant_city | object | 0 | 66,537 |
| merchant_state | object | 11.75 | 12,173 |
| zip | float64 | 12.42 | 199 |
| mcc | int64 | 0 | 24,586 |
| errors | object | 98.41 | 109 |
| card_brand | object | 0 | 22 |
| card_type | object | 0 | 4 |
| card_number | int64 | 0 | 3 |
| expires | object | 0 | 4070 |
| cvv | int64 | 0 | 180 |
| has_chip | object | 0 | 983 |
| num_cards_issued | int64 | 0 | 2 |
| credit_limit | object | 0 | 3 |
| acct_open_date | object | 0 | 2603 |
| year_pin_last_changed | int64 | 0 | 19 |
| card_on_dark_web | object | 0 | 1 |
| id | int64 | 0 | 8,914,963 |
| target | object | 0 | 2 |

Table 5: Merged dataset outline