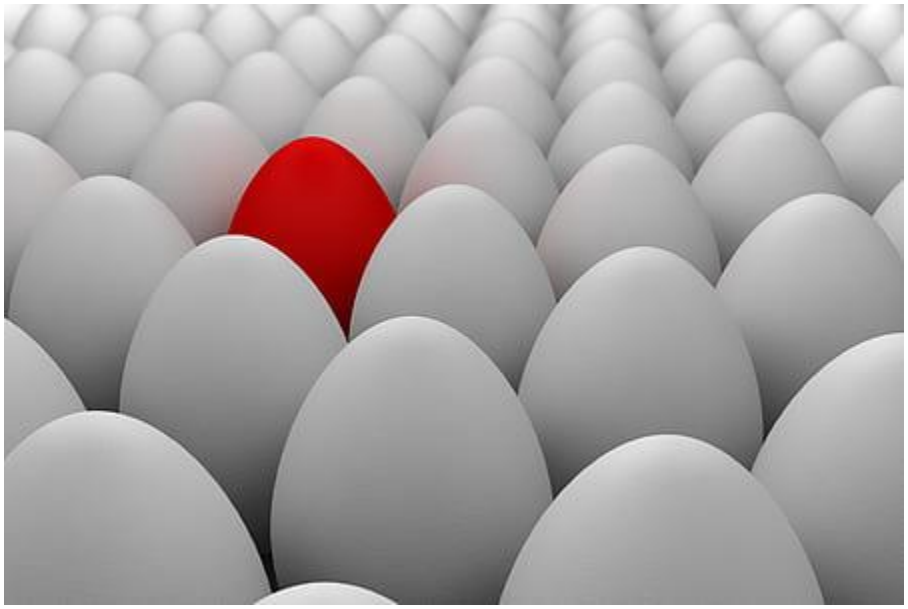
# CREDIT CARD FRAUD DETECTION

BY ARNAV GUPTA

## What is Anomaly Detection?

- An anomaly can be seen as data that deviates substantially from the norm.
- Anomaly detection is the process of identifying rare observations which differ substantially from the majority of the data from where they are drawn
- Applications include intrusion detection, fraud detection, fault detection, healthcare monitoring etc



## What is Fraud Detection?

- Fraud detection is the process of detecting anomalous financial records from within a broader set of normal transactions.
- The data is typically tabular in nature i.e. data sets with rows and columns
- It is important to have access to histrorical instances of confirmed fraudulent behaviour i.e. labels or our target variable, which are often issued by a bank or third party
- Because fraud is by definition less frequent than normal behaviour within a financial services ecosystem, there will be far less confirmed historical instances of fraudulent behaviour compared with the known good/normal behaviour, leading to an imbalance between the fraudulent and non-fraudulent samples

- Feature engineering is crucial, as it involves converting domain knowledge from fraud analysts and investigators into data that can be used to detect suspicious behaviours
- The features/data is typically aggregated at the customer-level, or at the transaction-level, depending on the use-case. Some approaches even combine the two
- Network data i.e. how users within a system are connected to one another (if at all), is normally a strong indicator of fraudulent behaviour
- Data sets for fraud detection are notoriously difficulty to access, due to various issues related to data privacy. There are some popular data sets available online, one of which is the ULB Machine Learning Group credit card fraud data set on Kaggle that we'll be using throughout this course

# Exploring the Credit Card Fraud DataSet

https://www.kaggle.com/datasets/mlg-ulb/creditcardfraud

### Context

It is important that credit card companies are able to recognize fraudulent credit card transactions so that customers are not charged for items that they did not purchase.

### Content

The dataset contains transactions made by credit cards in September 2013 by European cardholders.

This dataset presents transactions that occurred in two days, where we have 492 frauds out of 284,807 transactions. The dataset is highly unbalanced, the positive class (frauds) account for 0.172% of all transactions.

It contains only numerical input variables which are the result of a PCA transformation. Features V1, V2, ... V28 are the principal components obtained with PCA, the only features which have not been transformed with PCA are 'Time' and 'Amount'. Feature 'Time' contains the seconds elapsed between each transaction and the first transaction in the dataset. The feature 'Amount' is the transaction Amount, this feature can be used for example-dependant cost-sensitive learning. Feature 'Class' is the response variable and it takes value 1 in case of fraud and 0 otherwise.

```
[1]: import pandas as pd
```

Collapse Output

```
[4]: df = pd.read_csv('credit_card.csv')
```

```
[5]: df.head()
```

[5]:

| | Time | V1 | V2 | V3 | V4 | V5 | V6 | V7 | V8 | V9 | ... | V21 | V22 | V23 | V24 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0.0 | -1.359807 | -0.072781 | 2.536347 | 1.378155 | -0.338321 | 0.462388 | 0.239599 | 0.098698 | 0.363787 | ... | -0.018307 | 0.277838 | -0.110474 | 0.066928 |
| 1 | 0.0 | 1.191857 | 0.266151 | 0.166480 | 0.448154 | 0.060018 | -0.082361 | -0.078803 | 0.085102 | -0.255425 | ... | -0.225775 | -0.638672 | 0.101288 | -0.339846 |
| 2 | 1.0 | -1.358354 | -1.340163 | 1.773209 | 0.379780 | -0.503198 | 1.800499 | 0.791461 | 0.247676 | -1.514654 | ... | 0.247998 | 0.771679 | 0.909412 | -0.689281 |
| 3 | 1.0 | -0.966272 | -0.185226 | 1.792993 | -0.863291 | -0.010309 | 1.247203 | 0.237609 | 0.377436 | -1.387024 | ... | -0.108300 | 0.005274 | -0.190321 | -1.175575 |
| 4 | 2.0 | -1.158233 | 0.877737 | 1.548718 | 0.403034 | -0.407193 | 0.095921 | 0.592941 | -0.270533 | 0.817739 | ... | -0.009431 | 0.798278 | -0.137458 | 0.141267 |

5 rows × 31 columns

```
[6]: df.describe()
```

[6]:

| | Time | V1 | V2 | V3 | V4 | V5 | V6 | V7 | V8 | V9 |
|---|---|---|---|---|---|---|---|---|---|---|
| count | 284807.000000 | 2.848070e+05 | 2.848070e+05 | 2.848070e+05 | 2.848070e+05 | 2.848070e+05 | 2.848070e+05 | 2.848070e+05 | 2.848070e+05 | 2.848070e+05 |
| mean | 94813.859575 | 1.168375e-15 | 3.416908e-16 | -1.379537e-15 | 2.074095e-15 | 9.604066e-16 | 1.487313e-15 | -5.556467e-16 | 1.213481e-16 | -2.406331e-15 |
| std | 47488.145955 | 1.958696e+00 | 1.651309e+00 | 1.516255e+00 | 1.415869e+00 | 1.380247e+00 | 1.332271e+00 | 1.237094e+00 | 1.194353e+00 | 1.098632e+00 |
| min | 0.000000 | -5.640751e+01 | -7.271573e+01 | -4.832559e+01 | -5.683171e+00 | -1.137433e+02 | -2.616051e+01 | -4.355724e+01 | -7.321672e+01 | -1.343407e+01 |
| 25% | 54201.500000 | -9.203734e-01 | -5.985499e-01 | -8.903648e-01 | -8.486401e-01 | -6.915971e-01 | -7.682956e-01 | -5.540759e-01 | -2.086297e-01 | -6.430976e-01 |
| 50% | 84692.000000 | 1.810880e-02 | 6.548556e-02 | 1.798463e-01 | -1.984653e-02 | -5.433583e-02 | -2.741871e-01 | 4.010308e-02 | 2.235804e-02 | -5.142873e-02 |
| 75% | 139320.500000 | 1.315642e+00 | 8.037239e-01 | 1.027196e+00 | 7.433413e-01 | 6.119264e-01 | 3.985649e-01 | 5.704361e-01 | 3.273459e-01 | 5.971390e-01 |
| max | 172792.000000 | 2.454930e+00 | 2.205773e+01 | 9.382558e+00 | 1.687534e+01 | 3.480167e+01 | 7.330163e+01 | 1.205895e+02 | 2.000721e+01 | 1.559499e+01 |

8 rows × 31 columns

The dataset initially might have contained more than 31 columns i.e dimensions, but by performing data mining techniques like PCA transformation, it has been brought down to 31 dimensions.

```
[2]: df['Class'].value_counts()
```

```
[2]: Class
     0    284315
     1       492
     Name: count, dtype: int64
```

This shows the class imbalance between fraudulent and normal records. Training algorithm would be biased towards majority class. Hence we would not be able to learn anything meaningful from minority fraudulent class. We can cater to this imbalance by:

- Up sample the minority class at training time (synthetic data)
- Down sample the majority class
- Choose an approach better suited to highly imbalanced data i.e. anomaly detection algorithms
- Re-balance the classes at training time using the algorithm's class_weight hyperparameter to penalize the loss function more for misclassifications made on the minority class (hence improving the algorithm's ability to learn the minority class)

## Sampling with a Class Imbalance

In machine learning, there are traditionally two main types of modelling approaches:

- Supervised learning (data has a label or target variable i.e. something to learning and correct itself from)
    - Classification - predicting a categorical value i.e. is fraud yes/no
    - Regression - predicting a continuous value i.e. price
- Unsupervised (data has no label)
    - Clustering - find the natural groupings within the data
    - Dimensionality reduction - reduce higher dimensional data set down to a lower dimensional space i.e. many columns down to fewer columns to potentially help improve model performance

Fraud detection is typically a supervised, binary classification problem, but unsupervised learning (both clustering and PCA) can be used.

This data set represents a supervised learning problem (binary (yes/no) classification).

## Model Validation

- Train set which our model learns from
- Test set (unseen holdout set) which is used to evaluate the effectiveness of the model after training is complete
- Often a 80/20 or 90/10 split depending on the amount of data

```
[3]: from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.1, random_state=42, stratify=y)
print("X_train:", X_train.shape)
print("X_test:", X_test.shape)
print("y_train:", y_train.shape)
print("y_test:", y_test.shape)
```

```
X_train: (256326, 28)
X_test: (28481, 28)
y_train: (256326,)
y_test: (28481,)
```

```
[5]: import numpy as np

print("Fraud in y_train:", len(np.where(y_train == 1)[0]))
print("Fraud in y_test", len(np.where(y_test == 1)[0]))
```

```
Fraud in y_train: 443
Fraud in y_test 49
```

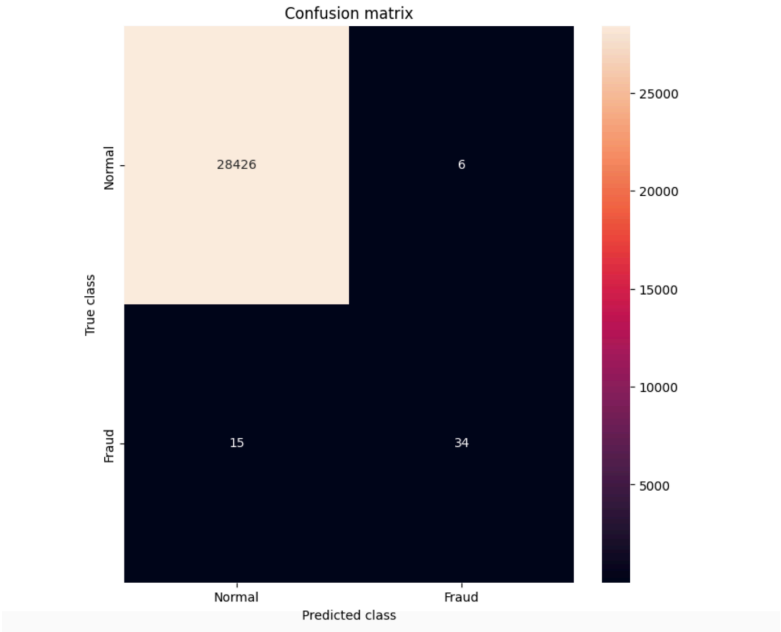
This shows how many fraudulent records are in the training set i.e 443 compared to test set i.e 49

# Training a Baseline Logistic Regression Model using scikit-learn

https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html

## Introducing the Confusion Matrix

- Acts as a cross section between the ground truth and the predictions that the trained model makes
- In a binary classification context, the confusion matrix reports on the classification performance of the model:
    - True negatives - actually non-fraudulent and correctly classified as non-fraudulent
    - False positives - actually non-fraudulent but incorrectly classified as fraud
    - False negatives - actually fraud but incorrectly classified as being non-fraudulent
    - True positives - actually fraud and correctly classified as fraud
- We're trying to detect fraud, so the "positive" relates to fraud while "negative" refers to non-fraudulent.
- "True" relates to correct predictions, whereas "false" refers to incorrect predictions.
- Many performance metrics are derived from the elements of the confusion matrix (Section 4), therefore it is foundational for understanding how to measure performance for classification problems.



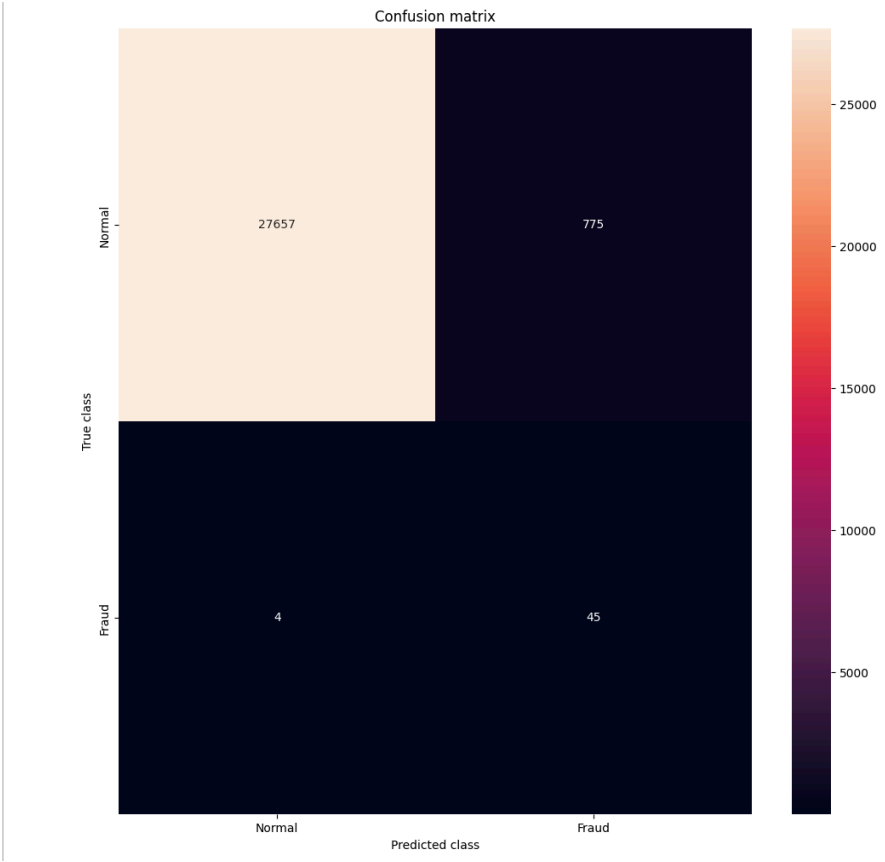
# Improving the Logistic Regression Model through Hyperparameter Selection

## Balanced Class Weight

```
[2]: from sklearn.linear_model import LogisticRegression
     model = LogisticRegression(class_weight='balanced')
     model.fit(X_train, y_train)
[2]: LogisticRegression
     LogisticRegression(class_weight='balanced')
```

Setting class_weight='balanced' automatically adjusts the weights for each class in the training data, so that the model gives more importance to the minority class.

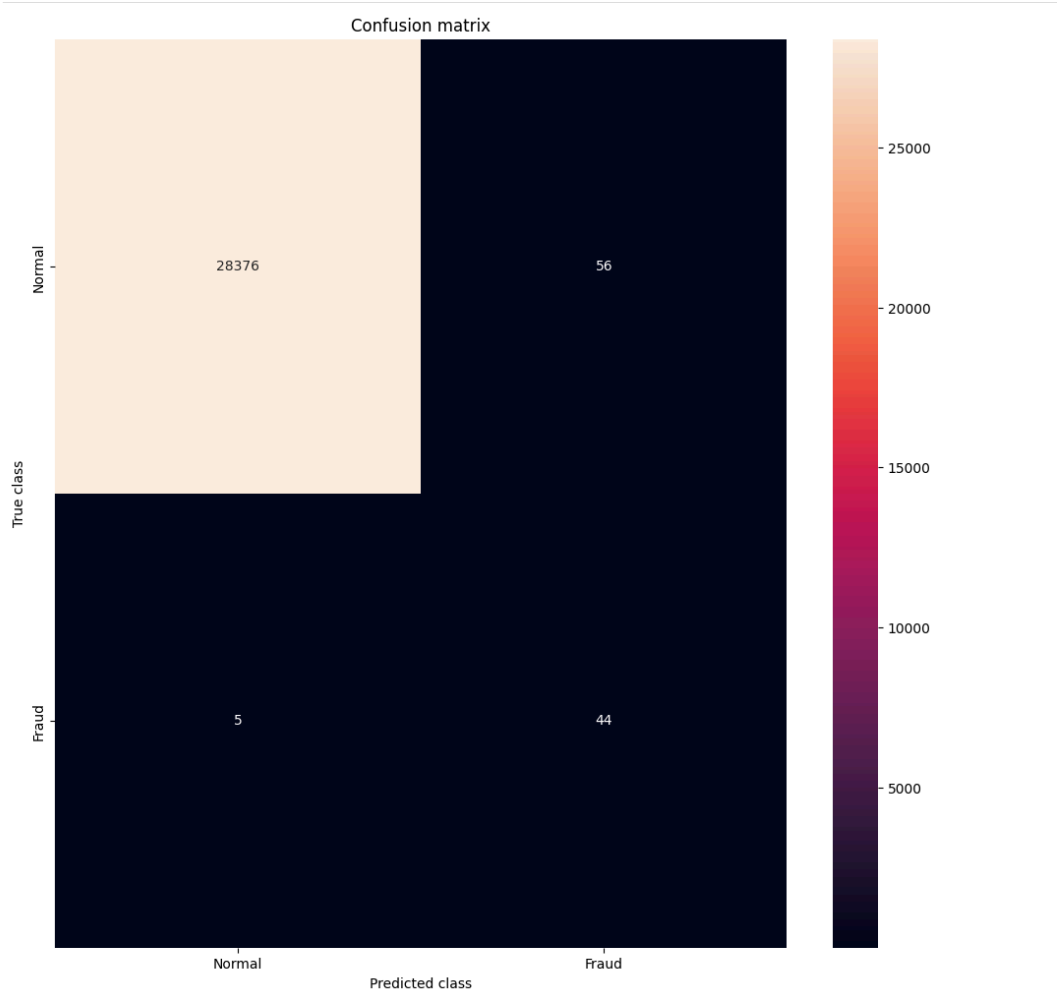
This is useful for preventing the model from being biased toward the majority class when making predictions, especially in cases of class imbalance.



## Custom Class Weight

```
[6]: model = LogisticRegression(class_weight={0:1, 1:50})
     model.fit(X_train, y_train)
     y_pred = model.predict(X_test)

     LABELS = ["Normal", "Fraud"]
     conf_matrix = confusion_matrix(y_test, y_pred)
     plt.figure(figsize=(12, 12))
     sns.heatmap(conf_matrix, xticklabels=LABELS, yticklabels=LABELS, annot=True, fmt="d")
     plt.title("Confusion matrix")
     plt.ylabel('True class')
     plt.xlabel('Predicted class')
     plt.show()
```

Setting class_weight={0:1, 1:50} keeps majority class as it is, but for minority class, it increases the emphasis of that class by 50 times.

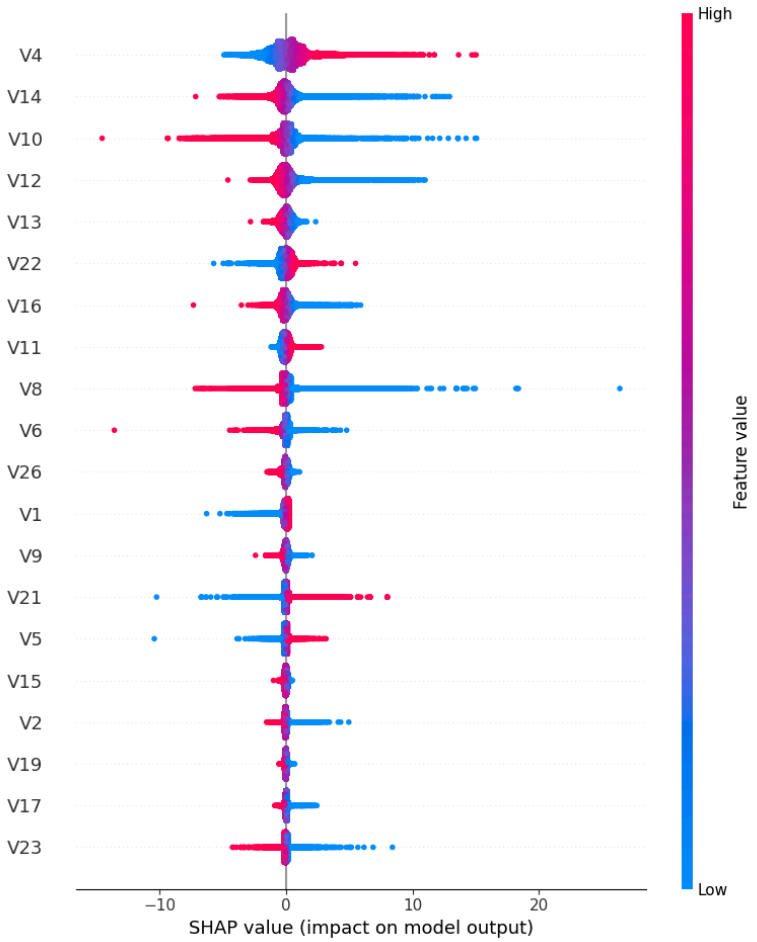


# Interpreting the Logistic Regression Model

## SHAP

The Shapley value is the average expected marginal contribution of one feature after all possible feature combinations have been considered.

Shapley value helps to determine a payoff for all of the features when each feature might have contributed more or less than the others.




This shows descending order of feature importance. Hence V4 has highest importance, next is V14. Moreover, more red corresponds to higher feature value, while more blue corresponds to lower feature value. So V4 is directly proportional to impact on model output (higher the value of V4, higher its positive impact on output) whereas V14 is

inversely proportional to impact on model output (lower the value of V14, higher its positive impact on output).

# Training an XGBoost Model

https://xgboost.readthedocs.io/en/stable/get_started.html

XGBoost is a popular and efficient open-source implementation of the gradient boosted trees algorithm. Gradient boosting is a supervised learning algorithm, which attempts to accurately predict a target variable by combining the estimates of a set of simpler, weaker models.

```
import xgboost as xgb

model = xgb.XGBClassifier()

model.fit(X_train, y_train)
```

[2]:

```
XGBClassifier(base_score=None, booster=None, callbacks=None,
              colsample_bylevel=None, colsample_bynode=None,
              colsample_bytree=None, device=None, early_stopping_rounds=None,
              enable_categorical=False, eval_metric=None, feature_types=None,
              gamma=None, grow_policy=None, importance_type=None,
              interaction_constraints=None, learning_rate=None, max_bin=None,
              max_cat_threshold=None, max_cat_to_onehot=None,
              max_delta_step=None, max_depth=None, max_leaves=None,
              min_child_weight=None, missing=nan, monotone_constraints=None,
              multi_strategy=None, n_estimators=None, n_jobs=None,
              num_parallel_tree=None, random_state=None, ...)
```

[3]:

```
y_pred = model.predict(X_test)
```

[4]:

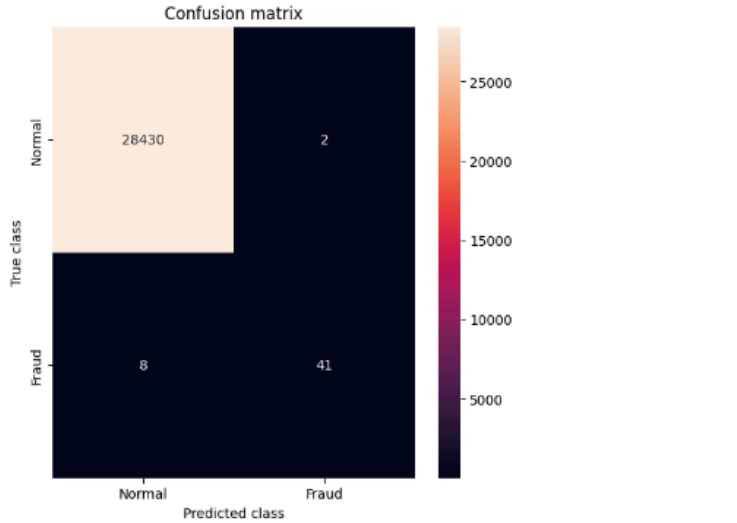
```
from sklearn.metrics import confusion_matrix

confusion_matrix(y_test, y_pred)
```

[4]:

```
array([[28430,     2],
       [    8,    41]])
```

[5]:

```
import matplotlib.pyplot as plt
import seaborn as sns

LABELS = ["Normal", "Fraud"]

conf_matrix = confusion_matrix(y_test, y_pred)
plt.figure(figsize=(6, 6))
sns.heatmap(conf_matrix, xticklabels=LABELS, yticklabels=LABELS, annot=True, fmt="d")
plt.title("Confusion matrix")
plt.ylabel('True class')
plt.xlabel('Predicted class')
plt.show()
```



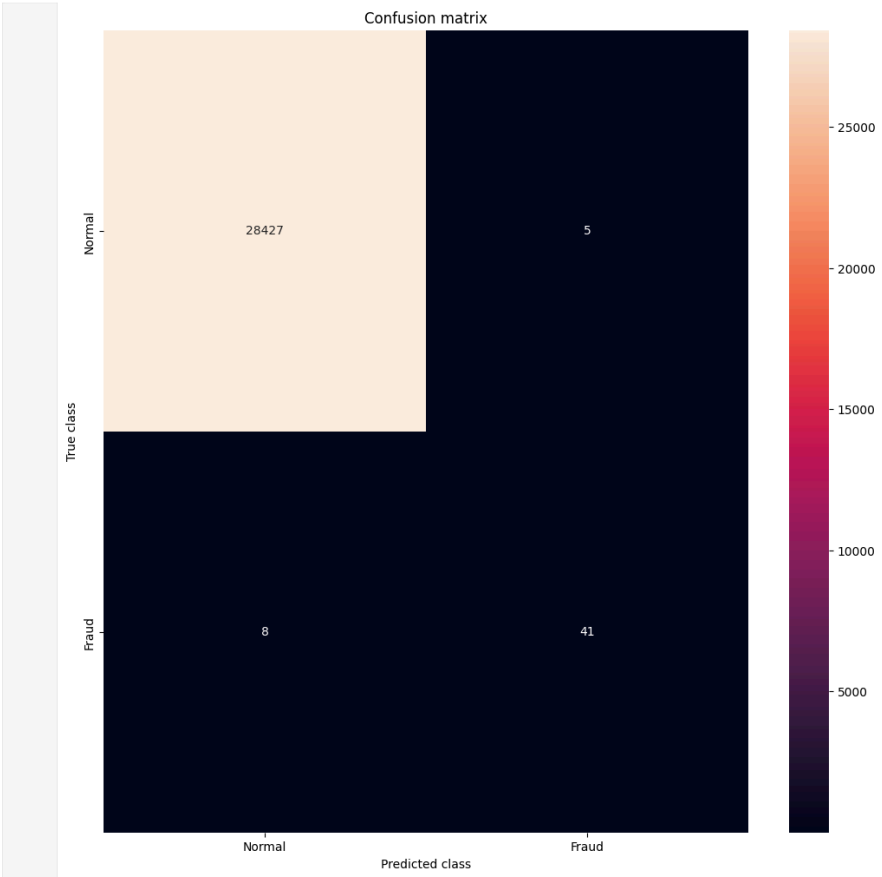
# Improving the XGBoost Model through Hyperparameter Selection

## Scale_pos_weight=100

This hyperparameter penalises misclassifications of positive class more severely by a factor of 100, at training time.

[2]:
```
import xgboost as xgb

model = xgb.XGBClassifier(scale_pos_weight=100)

model.fit(X_train, y_train)
```

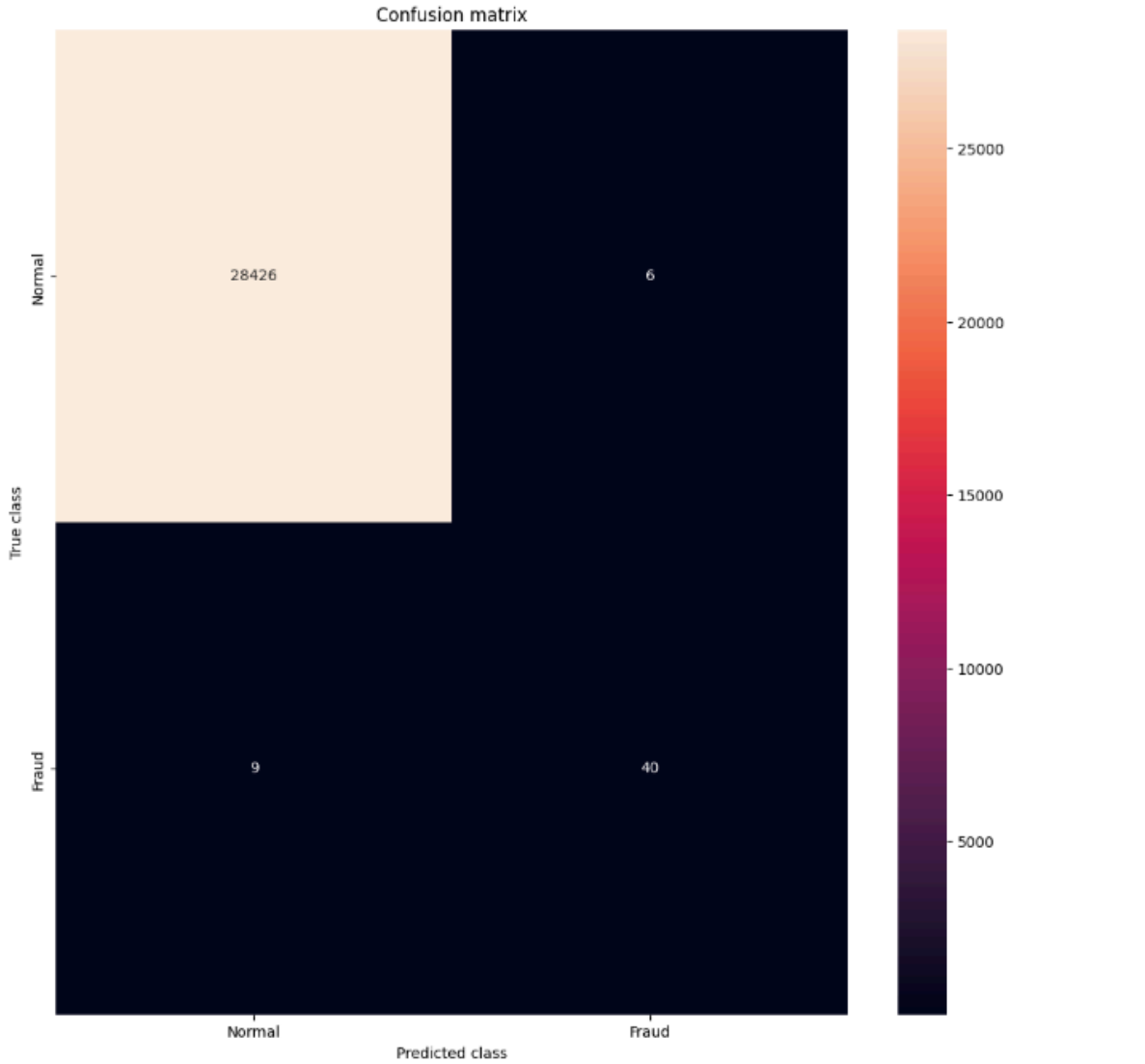
[2]:
```
XGBClassifier(base_score=0.5, booster='gbtree', callbacks=None,
              colsample_bylevel=1, colsample_bynode=1, colsample_bytree=1,
              early_stopping_rounds=None, enable_categorical=False,
              eval_metric=None, feature_types=None, gamma=0, gpu_id=-1,
              grow_policy='depthwise', importance_type=None,
              interaction_constraints='', learning_rate=0.300000012,
              max_bin=256, max_cat_threshold=64, max_cat_to_onehot=4,
              max_delta_step=0, max_depth=6, max_leaves=0, min_child_weight=1,
              missing=nan, monotone_constraints='()', n_estimators=100,
              n_jobs=0, num_parallel_tree=1, predictor='auto', random_state=0, ...)
```



## Max_depth=5, Scale_pos_weight=100

Additionally, max_depth hyperparameter specifies the maximum depth to which each tree will be built. It reduces overfitting.

```
[6]: model = xgb.XGBClassifier(max_depth=5, scale_pos_weight=100)
     # max_depth specifies the maximum depth to which each tree will be built.
     # reduces overfitting
     model.fit(X_train, y_train)
     y_pred = model.predict(X_test)

     LABELS = ["Normal", "Fraud"]
     conf_matrix = confusion_matrix(y_test, y_pred)
     plt.figure(figsize=(12, 12))
     sns.heatmap(conf_matrix, xticklabels=LABELS, yticklabels=LABELS, annot=True, fmt="d")
     plt.title("Confusion matrix")
     plt.ylabel('True class')
     plt.xlabel('Predicted class')
     plt.show()
```



## Interpreting the XGBoost Model

```
[7]: model.feature_importances_
```

```
[7]: array([0.01778892, 0.006522  , 0.01161652, 0.04864641, 0.00839465,
            0.01592866, 0.02007158, 0.02973094, 0.00941193, 0.04264095,
            0.01003185, 0.02240502, 0.01880649, 0.52212244, 0.01488954,
            0.00652269, 0.08942025, 0.00986922, 0.01701785, 0.01133021,
            0.01525105, 0.0066182 , 0.01258562, 0.00372731, 0.00570192,
            0.01185634, 0.00590739, 0.00518408], dtype=float32)
```

Hence according to this model, V14 has the highest importance, with 52.2%, next is V4 with 4.8%. In contrast, the logistic regression model showed that V4 has highest importance, then next was V14.

## Understanding the Cost of Misclassification

```
[2]: conf_matrix_xgb
```

```
[2]: array([[28426,     6],
            [    9,    40]])
```

```
[3]: print(conf_matrix_xgb[0][0])
     print(conf_matrix_xgb[0][1])
     print(conf_matrix_xgb[1][0])
     print(conf_matrix_xgb[1][1])
```

```
28426
6
9
40
```

```
[4]: cost_tn = 1
     cost_fp = 10
     cost_fn = 100
     cost_tp = 1
```

```
[5]: total_cost_of_fraud_xgb = (conf_matrix_xgb[0][0] * cost_tn) + (conf_matrix_xgb[0][1] * cost_fp) + (conf_matrix_xgb[1][0] * cost_fn) + (conf
     total_cost_of_fraud_xgb
```

```
[5]: np.int64(29426)
```

Each type of classification outcome has a predefined cost:

- `cost_tn = 1` → Cost for a **True Negative** (correctly classified negative)
- `cost_fp = 10` → Cost for a **False Positive** (misclassified negative as positive)
- `cost_fn = 100` → Cost for a **False Negative** (misclassified positive as negative)
- `cost_tp = 1` → Cost for a **True Positive** (correctly classified positive)

It applies a weighted cost approach where false negatives (missed frauds) are the most expensive (100 per instance). Final calculated value is 29426. We aim to minimise this cost.

### The Accuracy Paradox

The accuracy paradox refers to situations where a high accuracy metric does not necessarily indicate a good model, especially in imbalanced datasets. In fraud detection, this paradox is particularly relevant because fraud cases are rare compared to non-fraud cases. In our confusion matrix:

**Accuracy = (TN+TP) / Total Samples = (28426+40) / (28426+6+9+40) = 99.95%**

This high accuracy seems amazing but it is misleading.

## Implementing Performance Metrics in scikit-learn

### Precision Score

Precision is the proportion of correctly predicted fraudulent instances among all instances predicted as fraud.

**Precision=TP / (TP+FP) =40 / (40+6) ≈ 86.96%**

### Recall Score

Recall is the proportion of the fraudulent instances that are successfully predicted.

**Recall =TP / (TP+FN) =40 / (40+9) ≈ 81.63%**

### ROC AUC score

Receiver Operating Characteristic - Area Under the Curve plots the TPR (Recall) and FPR at various classification thresholds where **FPR = FP / (FP + TN)**

**ROC_AUC_Score ≈ 90.81%**

### F1 Score

F1-score is the harmonic balance of precision and recall (can be weighted more towards P or R if need be)

**F = 2 * (Precision * Recall)/(Precision + Recall) ≈ 84.19%**

### Average Precision Score

**AUPRC = Area under the Precision-Recall curve**. Better alternative to AUC as doesn't include TN which influences the scores significantly in highly imbalanced data. It calculates the area under the curve at various classification thresholds.

**Average_precision_score ≈ 71.01%**

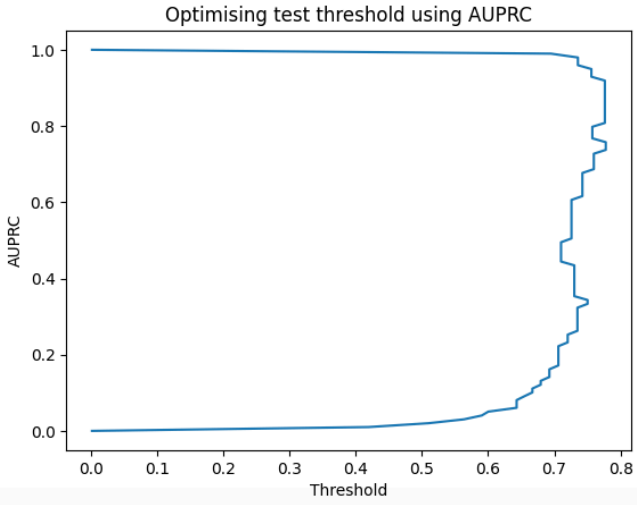
# Threshold Optimization using Performance Metrics

We aim to find the best decision threshold for classification by optimizing one of the performance metrics. Here I am choosing the **Area Under the Precision-Recall Curve (AUPRC).**

## Methodology

- Threshold values between 0 and 1 were generated.
- Binary predictions were created based on each threshold.
- AUPRC was computed for each threshold.
- AUPRC vs. threshold was plotted to visualize performance trends.
- The best-performing thresholds (AUPRC ≥ 0.82) were identified.

[3]:
```
import numpy as np
from sklearn.metrics import average_precision_score

threshold_list = []
auprc_list = []
thresholds = np.linspace(0, 1, 100)

for threshold in thresholds:
    y_pred_thresh = [1 if e > threshold else 0 for e in y_pred]
    threshold_list.append(threshold)

    # AUPRC
    auprc_score = average_precision_score(y_test, y_pred_thresh)
    auprc_list.append(auprc_score)

# plot curve
threshold_df = pd.DataFrame({'Threshold': threshold_list, 'AUPRC': auprc_list})
threshold_df.columns = ['AUPRC', 'Threshold']

plt.plot(threshold_df['Threshold'], threshold_df['AUPRC'])
plt.title("Optimising test threshold using AUPRC")
plt.xlabel('Threshold')
plt.ylabel('AUPRC')
plt.savefig('Optimising threshold using AUPRC');
plt.show()
```



## Results

- The optimal threshold maximizing AUPRC is around 0.77.
- This selection ensures better precision-recall tradeoff for imbalanced datasets.

[4]:
```
threshold_df.sort_values(by='AUPRC', ascending=False)
```

[4]:

| | AUPRC | Threshold |
|---|---|---|
| **99** | 1.000000 | 0.001720 |
| **98** | 0.989899 | 0.694936 |
| **97** | 0.979798 | 0.735652 |
| **96** | 0.969697 | 0.735652 |
| **95** | 0.959596 | 0.735652 |
| **...** | ... | ... |
| **4** | 0.040404 | 0.590410 |
| **3** | 0.030303 | 0.563415 |
| **2** | 0.020202 | 0.510139 |
| **1** | 0.010101 | 0.419485 |
| **0** | 0.000000 | 0.001720 |

100 rows × 2 columns

[5]:
```
threshold_df.loc[(threshold_df['AUPRC'] >= 0.82)]
```

[5]:

| | AUPRC | Threshold |
|---|---|---|
| **82** | 0.828283 | 0.776372 |
| **83** | 0.838384 | 0.776372 |
| **84** | 0.848485 | 0.776372 |
| **85** | 0.858586 | 0.776372 |
| **86** | 0.868687 | 0.776372 |
| **87** | 0.878788 | 0.776372 |
| **88** | 0.888889 | 0.776372 |
| **89** | 0.898990 | 0.776372 |
| **90** | 0.909091 | 0.776372 |
| **91** | 0.919192 | 0.776372 |
| **92** | 0.929293 | 0.756012 |
| **93** | 0.939394 | 0.756012 |
| **94** | 0.949495 | 0.756012 |
| **95** | 0.959596 | 0.735652 |
| **96** | 0.969697 | 0.735652 |
| **97** | 0.979798 | 0.735652 |
| **98** | 0.989899 | 0.694936 |
| **99** | 1.000000 | 0.001720 |

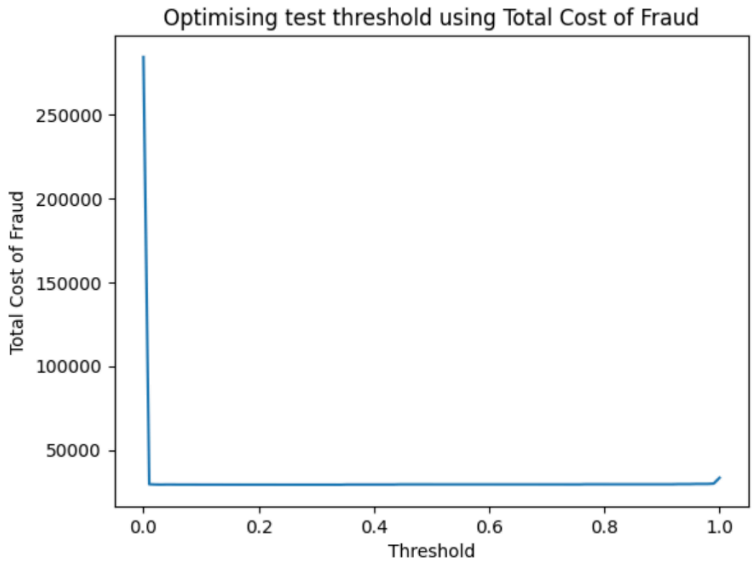
# Threshold Optimization using Total Cost of Fraud

We aim to determine the optimal classification threshold for fraud detection by evaluating how different thresholds impact the **Total Cost of Fraud (TCF)**. By minimizing TCF, we ensure an optimal balance between false positives (FP) and false negatives (FN), which have different cost implications.

## Methodology

- Define a range of thresholds between 0 and 1
- Convert predicted probabilities into binary classifications using each threshold: If y_pred > threshold, classify as fraud (1). Otherwise, classify as non-fraud (0).
- Compute the confusion matrix at each threshold to count: True Negatives (TN), False Positives (FP), False Negatives (FN) & True Positives (TP)
- Calculate the Total Cost of Fraud (TCF) using predefined cost values.
- Plot TCF vs. threshold to visualize cost trends.
- Identify the optimal threshold by selecting the threshold with the lowest TCF.

[3]:
```
import numpy as np
from sklearn.metrics import average_precision_score

threshold_list = []
tcf_list = []
cost_tn = 1
cost_fp = 10
cost_fn = 100
cost_tp = 1
thresholds = np.linspace(0, 1, 100)

for threshold in thresholds:
    y_pred_thresh = [1 if e > threshold else 0 for e in y_pred]
    threshold_list.append(threshold)

    # Total Cost of Fraud
    conf_matrix_xgb = confusion_matrix(y_test, y_pred_thresh)
    tcf_score = (conf_matrix_xgb[0][0] * cost_tn) + (conf_matrix_xgb[0][1] * cost_fp) + (conf_matrix_xgb[1][0] * cost_fn) + (conf_matrix_xgb
    tcf_list.append(tcf_score)

# Create a DataFrame to store results
threshold_df = pd.DataFrame({'Threshold': threshold_list, 'TCF': tcf_list})

# Sort by TCF (ascending order) to find the best threshold
threshold_df.sort_values(by='TCF', ascending=True)


plt.plot(threshold_df['Threshold'], threshold_df['TCF'])
plt.title("Optimising test threshold using Total Cost of Fraud")
plt.xlabel('Threshold')
plt.ylabel('Total Cost of Fraud')
plt.savefig('Optimising threshold using Total Cost of Fraud');
plt.show()
```




## Results

- The optimal threshold minimizing Total Cost of Fraud is around 0.30 to 0.34, where the TCF is around 29228-29237.
- Lower thresholds lead to excessive false positives, increasing the cost. Higher thresholds lead to excessive false negatives, also increasing the cost.
- When threshold = 0, everything is classified as fraud, leading to a very high cost due to false positives. When threshold = 1, almost everything is classified as non-fraud, leading to an even higher cost due to missed fraud cases.

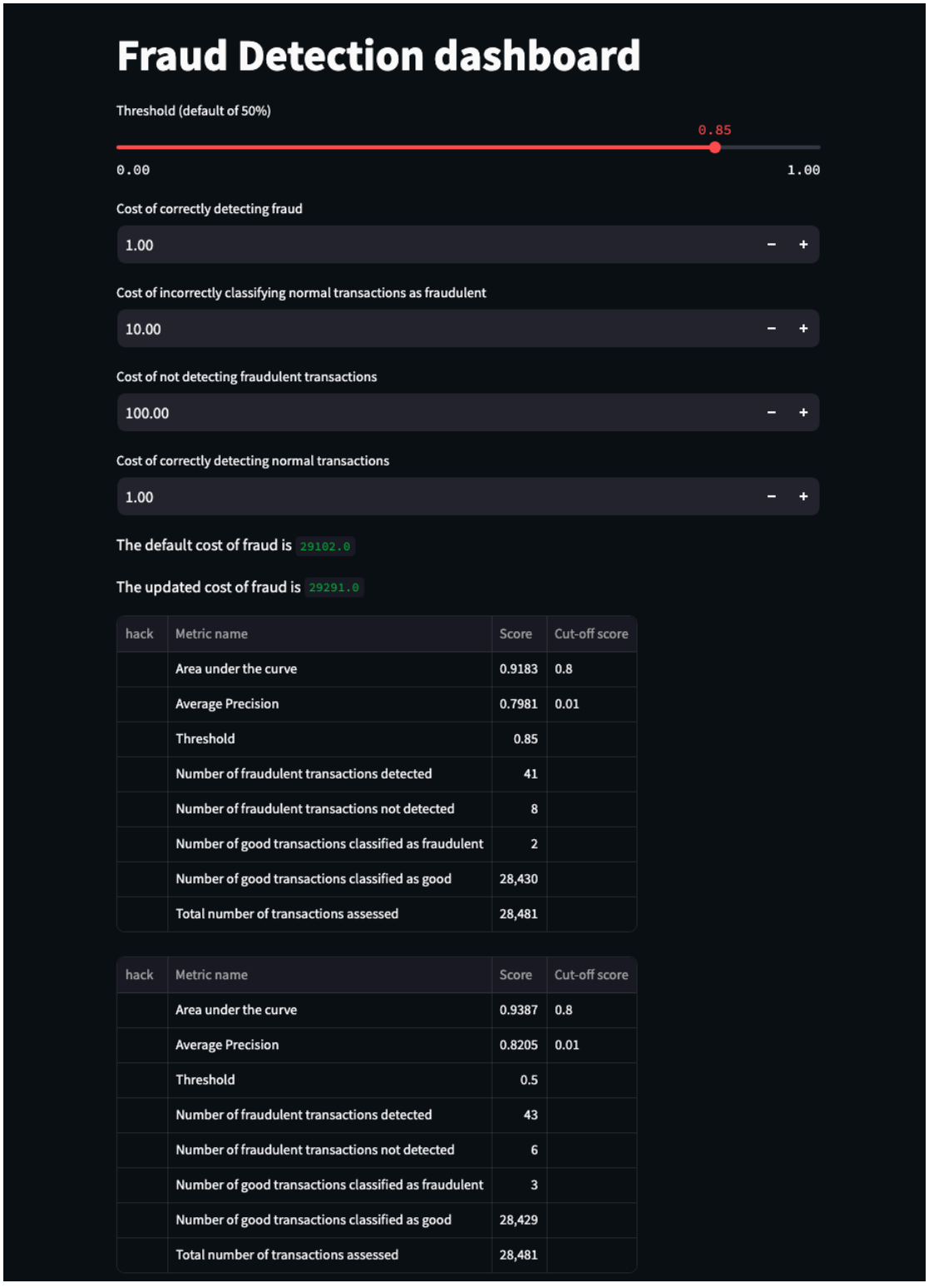
[4]:
```
# if threshold = 0, then everything is fraud (lots of false positives which cost $10 each)
# if threshold = 1, then everything is non-fraudulent (quite a few missed cases of fraud which cost $100 each)
# optimal threshold for this model is around 50% (already well balanced)

threshold_df.sort_values(by='TCF', ascending=True)
```

[4]:

| | Threshold | TCF |
|---|---|---|
| **34** | 0.343434 | 29228 |
| **33** | 0.333333 | 29228 |
| **30** | 0.303030 | 29237 |
| **31** | 0.313131 | 29237 |
| **29** | 0.292929 | 29237 |
| **...** | ... | ... |
| **95** | 0.959596 | 29678 |
| **97** | 0.979798 | 29678 |
| **98** | 0.989899 | 29876 |
| **99** | 1.000000 | 33332 |
| **0** | 0.000000 | 284369 |

100 rows × 2 columns

# Building a Fraud Detection Dashboard using Streamlit




| hack | Metric name | Score | Cut-off score |
|---|---|---|---|
| | Area under the curve | 0.9183 | 0.8 |
| | Average Precision | 0.7981 | 0.01 |
| | Threshold | 0.85 | |
| | Number of fraudulent transactions detected | 41 | |
| | Number of fraudulent transactions not detected | 8 | |
| | Number of good transactions classified as fraudulent | 2 | |
| | Number of good transactions classified as good | 28,430 | |
| | Total number of transactions assessed | 28,481 | |

| hack | Metric name | Score | Cut-off score |
|---|---|---|---|
| | Area under the curve | 0.9387 | 0.8 |
| | Average Precision | 0.8205 | 0.01 |
| | Threshold | 0.5 | |
| | Number of fraudulent transactions detected | 43 | |
| | Number of fraudulent transactions not detected | 6 | |
| | Number of good transactions classified as fraudulent | 3 | |
| | Number of good transactions classified as good | 28,429 | |
| | Total number of transactions assessed | 28,481 | |

# THANKYOU