# Credit Fraud

Charles Pearlman, Jonah Fallon, Audrey Powers

3/1/24

This comprehensive case study on credit card fraud delves into diverse methodologies for data preprocessing, analysis, and predictive modeling. Leveraging a synthetic credit card fraud dataset sourced from Kaggle, the investigation began with feature selection and data cleansing to enhance usability. Then an array of supervised learning approaches, including support vector machines, logistic regression, and k-nearest neighbors is explored. Augmenting these techniques with advanced data processing methods such as principal component analysis and normalization, refined the selected models. Ultimately, utilizing a neural network bolstered by these preprocessing techniques, obtaining the highest predictive accuracy with an F1 score of 0.79, marking a significant milestone in fraud detection.

# Overview

This study used a synthetic financial data set made specifically for fraud detection. This set was generated from a real mobile money service based in Africa, so while that data itself may not be authentic, the trends and components of each transaction are valid. Some of the important features in this dataset are the old and new balances of the account, the type of transitions, the time of transaction, and whether or not it was fraud. We are all interested in cybersecurity and one important aspect of cybersecurity is fraud detection. The ability to write models that can predict fraud may be a useful skill in a future career, and creating more efficient and accurate fraud detection software is a major goal of many companies. With this data set, we want to be able to identify transactions that are likely to be fraudulent with 95 percent accuracy. Looking at the available data, we can figure out what aspects of a transaction are most likely to raise a flag to fraud. This could also help us in our own lives to be more aware when we could be a victim of fraud ourselves.

# Data Acquisition

The given data contains the features step, denoting the number of hours taken to complete the transaction, the type of payment, the amount of dollars in the transaction, the account number of the origin and destination accounts, the original and new balance of the account that transferred money, the original and new balance of the account that received the money, whether or not the transaction was flagged as fraud, and whether or not the transaction was actually fraud. This data came from a Kaggle repository and was synthetically generated based on actual credit card transaction information. Since this data is synthetically made, there is no guarantee the results will transfer to the real world. The dataset claims to resemble real-world transactions well so there will most likely be some good information found from the dataset, however it is important to be mindful that it may not be perfect.

# Preprocessing

To process the data, the first step taken was to remove columns that couldn't be used in the models. The account numbers of the original and destination accounts are categorical strings that cannot be included in models without processing. The first option considered was the get_dummies() function on these columns in order to extract usable data, however, there are thousands of different account names, which would result in thousands of columns that could potentially skew the findings or slow run speed more than increase accuracy. However, the type

column contains five types of transactions, which was expanded using the get_dummies() function.

When looking at initial visualizations of the data, it seemed insightful to check for a correlation between the step and the number of fraudulent transactions. As seen in figure 1, there was little to no correlation between the number of frauds between different steps.

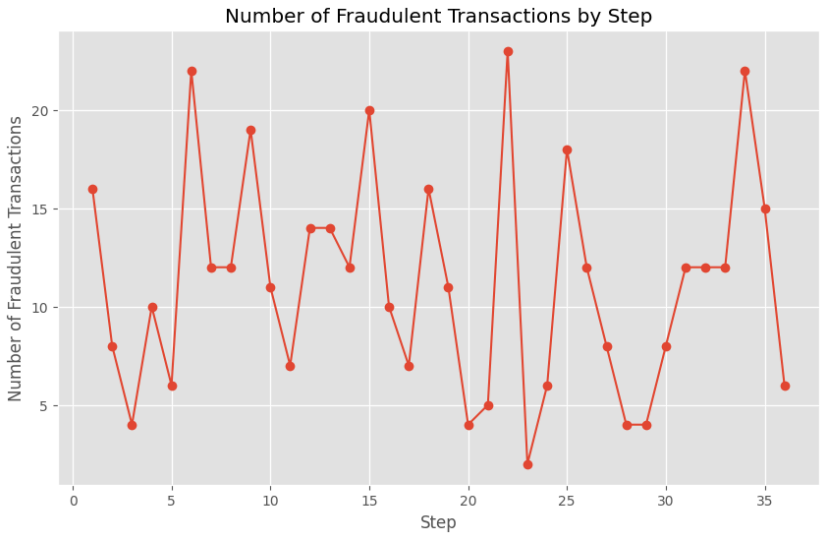


Figure 1

Next, an analysis of the distributions of the account balance of the origin account before and after the transaction was performed. The null values were removed from the dataset (values where the initial balance was 0, meaning no money was transferred), and a histogram of the old and new balances was plotted. From this visualization, it seems that in most transactions, most or all of the money was transferred out of the account, as seen in figure 2.

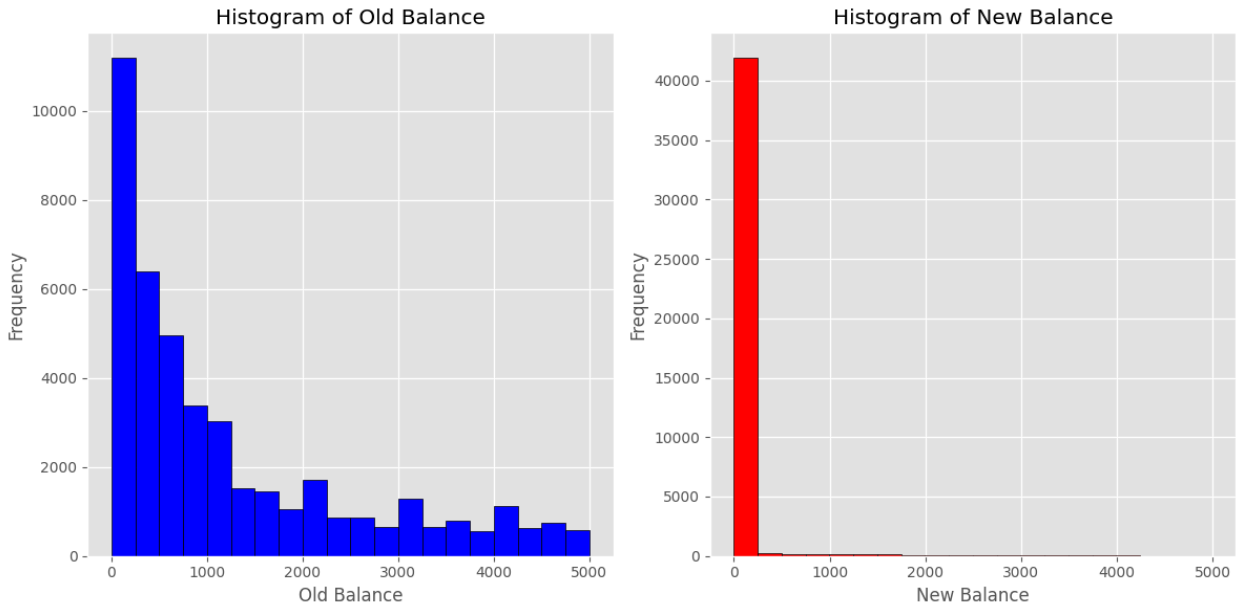


Figure 2

Then, a general overview of the amounts of the transactions in the dataset was generated. Again, rows in which the transfer amount was 0 were removed, and the result was a large spike near 0 with a few massive outliers, skewing the data heavily. In order to better visualize most of the amounts, the range of amounts was limited to $50,000. As seen in figure 3, most of the transactions were under $10,000, and are distributed in a steep decline.

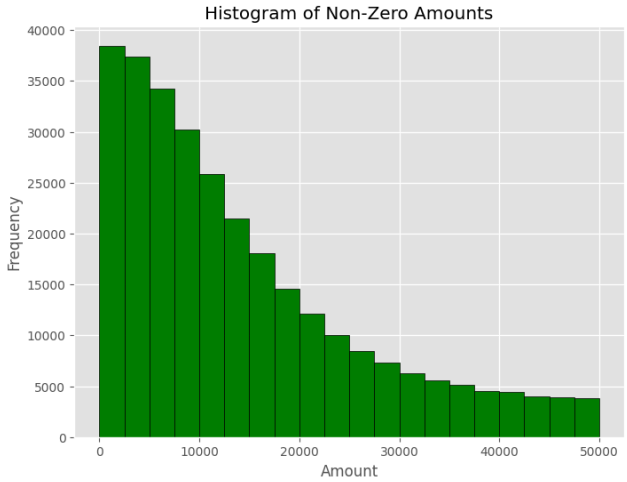


Figure 3

Finally, an analysis of transaction types that had the most fraudulent transactions was performed. The dataset was filtered to include only fraudulent transactions and summed the number of transactions of each type. It was discovered that of the fraudulent transactions, the only transfer types used were cash-outs and transfers, as seen in figure 4. The transaction type will likely be very predictive, as transactions not of this type are very unlikely to be fraud.

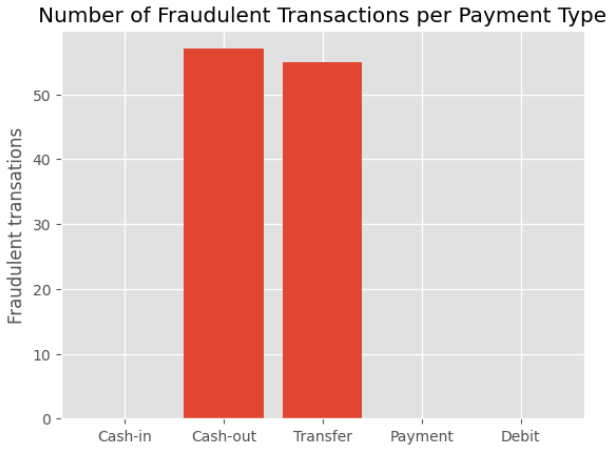


Figure 4

# Model Selection

In order to determine which model should be implemented, a variety of models were tested and evaluated with the f1 score as an indicator of success. The f1 score metric was chosen because it best represents a full range of success, displaying a model's ability to be both precise and accurate. After trials with six models, it was discovered that the best results were with the use of a neural network, giving a f1 score of 0.795. Since the neural network has the best combination of precision and accuracy, this model was chosen for  further evaluation.

**Algorithms Tested/Results (f1-score):**

| Model | Parameters/Manipulation | Score |
|---|---|---|
| Neural Network | Standard scalar, PCA with 8 components, 3 dense layers, 2 'relu' activations, 1 sigmoid, adam optimizer | 0.795 |
| Logistic Regression | None | 0.44 |
| Logistic Regression | Standard scalar | 0.55 |
| Logistic Regression | Standard scalar, PCA with 8 components | 0.61 |
| K-Nearest Neighbors | k=3 | 0.37 |
| SVM | Gamma=0.1 and C=1 | N/A (wouldn't run in reasonable amount of time) |

## SVM

After many trials, it was determined that Support Vector Machine (SVM) is suboptimal for the dataset due to its inefficiency with larger data volumes. Despite cutting the dataset to 30,000 observations for the current test shown in the code below, the time required to run was still around 5 minutes, while still returning a f1 score of 0(probably due to this data trimming). Rather than spending time testing various parameters such as gamma and C values, it was decided to abandon the SVM model altogether.

```
[6]: # Load in data set
     df = shuffle(df)
     df = df[:30000]
     df = pd.get_dummies(df, columns = ['type'])
     df = df.drop(columns=['nameOrig', 'nameDest'])
     df.shape

[6]: (30000, 13)

[7]: df = df.dropna(subset=['isFraud'])
     X = df.drop(columns=['isFraud'])
     y = df['isFraud']
     X_train, X_test, y_train, y_test = train_test_split(X, y)
     model = svm.SVC(gamma = .01, C = 1)
     model.fit(X_train,y_train)
     y_hat = model.predict(X_test)
     print(f1_score(y_test,y_hat))

     0.0
```

Figure 5: SVM code

## K-Nearest Neighbors

The K Nearest Neighbors model also did not function well with such a large dataset. The data was cut down in the same fashion as the SVM model and it was ran with a random set of 30,000 observations. K-values 1 through 20 were tested, and the f1 score increased slightly from values 1 to 3, peaking at an f1 score of 0.37, then continued to decrease for the rest of the values, making the best option k=3.

```
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn import neighbors

inputs = df_subset.drop(columns = ['isFraud'])
target = df_subset['isFraud']
k_values = range(1, 21)

for k in k_values:
    trainX, testX, trainY, testY = train_test_split(inputs, target, test_size = 0.25, shuffle = True, random_state = 13)

    model = neighbors.KNeighborsClassifier(k)
    model.fit(trainX, trainY)
    y_predict = model.predict(testX)
    f_score = f1_score(testY, y_predict)
    print(f_score)

    scorerVar = make_scorer(f1_score, pos_label=1)
    L_scores=model_selection.cross_val_score(model, inputs, target, cv=10, scoring=scorerVar)
    print ("f1 scores for each fold",L_scores)
    print("average f1 score for k value ", k, " : ", L_scores.mean())
```

Figure 6: KNN Code

## Logistic Regression

The logistic regression model was the best option before implementing the neural network. Logistic Regression started with an f1-score of 0.44, and then after applying a standard scalar its f-1 score increased to 0.55. This scalar was chosen because some of the transactions are much larger than others, and making these features a standard scale can eliminate some of the noise from outliers. Then applying PCA with 8 components was attempted. Some of the features such as amount, and balance before and after are highly correlated. Therefore applying PCA to remove some of this dimensionality seemed beneficial for the algorithm. By applying PCA with standard scaling this model was able to bring the f1-score up to 0.61.

```
from sklearn.model_selection import train_test_split
from sklearn import metrics
from sklearn.metrics import confusion_matrix, f1_score, classification_report, make_scorer
from sklearn import model_selection
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA

#Data splitting


y=df['isFraud']
X=df.drop(columns=['isFraud','step'])

pca=PCA(n_components=8)
pca.fit(X)
X=pca.transform(X)


X_train, X_test, y_train, y_test=train_test_split(X, y, test_size=0.3, shuffle=True, random_state=42)
print(X_train.shape)

scaler = StandardScaler() #scaling data make f1 increase from like .38 to .58
X_train = scaler.fit_transform(X_train)
X_test = scaler.fit_transform(X_test)
```

Figure 7: Scaling, Splitting, and PCA

```
#Log Reg
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import ShuffleSplit
import warnings
warnings.simplefilter(action='ignore', category=FutureWarning)

#create a model object
model=LogisticRegression(max_iter=200)

#train our model
model.fit(X_train, y_train)

#evaluate the model
y_pred = model.predict(X_test)

# Compute F1 score
f1 = f1_score(y_test, y_pred)
print("F1 score on test set:", f1)
```

Figure 8: Logistic Regression Best f1-score: .61

From experience in a previous machine learning class, it was decided to try a neural network (NN). Only 3 epochs were used because the data set was quite large and it would take too long to

run otherwise. All of the activation functions, optimizers, number of neurons, and layers were the default values from a standard NN example. Since it takes a long time to run, further experimentation and parameter tuning were not performed, however, this algorithm was run on the previously altered data with PCA and standard scaling.

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

# Define the number of features
num_features = X_train.shape[1]

# Define the FNN model architecture for binary classification
model = Sequential([
    Dense(units=64, activation='relu', input_shape=(num_features,)),
    Dense(units=32, activation='relu'),
    Dense(units=1, activation='sigmoid')
])

# Compile the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

# Train the model
model.fit(X_train, y_train, epochs=3, batch_size=32)

# Evaluate the model
test_loss, test_accuracy = model.evaluate(X_test, y_test)
print("Test Loss:", test_loss)
print("Test Accuracy:", test_accuracy)



from sklearn.metrics import f1_score

# Predictions on test set
y_pred = (model.predict(X_test) > 0.5).astype("int32")  # Convert probabilities to binary predictions

# Calculate F1 score
f1 = f1_score(y_test, y_pred)

print("F1 Score:", f1)
```

Figure 9: NN: Best f1-score .795

# Results and Evaluation

**Conclusions:**

From analyzing the PCA, it was discovered that the two features with the largest variances are the old and new balances of the destination account. This means that when PCA was performed for the model, these features are likely to be chosen first. Additionally, a confusion matrix was created for the data, displaying that the largest source of error in the model is when a transaction is fraud and is predicted to be not fraudulent. We believe this may be the case because significantly more data points are not fraud in the set and the model was overfitted to assume that a transaction is not fraud. The final F1-score for the best-performing neural network is 0.79. We are happy with this value after trying many models. Due to the size of the dataset, only 3 epochs were used. With just the 3 epochs, the training still took around 28 minutes. Our assumption is if more were added the F1 score could be raised even further. Doing some research and applying lecture materials, 5-10 epochs would most likely yield a better result.

**Visualizations:**

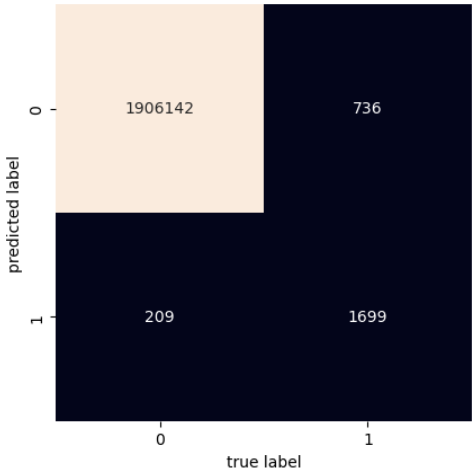


Figure 10: Confusion Matrix

The confusion matrix is a little unclear due to the large amount of non-fraud transactions. One observation taken from this was that most of the incorrect labels were predicted not fraud, but in reality they were. 736/945 incorrect observations fell in this category.

Top 2 PCA features:

Using the PCA model, the loading values for each column name were analyzed. The loading values indicate the variance of that particular feature. By finding the highest loading values, one can see which features will be chosen by performing PCA. As seen in Figures 11 and 12, the highest loading values were the destination account's old and new balances. The features where the get_dummies() function was used had extremely low loading values, likely because they are valued at 0 or 1, leading to very low variances.

| | Feature | Loading |
|---|---|---|
| 0 | amount | 4.651803e-02 |
| 1 | oldbalanceOrg | 9.448760e-02 |
| 2 | newbalanceOrig | 9.600343e-02 |
| 3 | oldbalanceDest | 6.721950e-01 |
| 4 | newbalanceDest | 7.265297e-01 |
| 5 | isFlaggedFraud | -8.435514e-15 |
| 6 | type_CASH_IN | 9.180374e-09 |
| 7 | type_CASH_OUT | 6.348042e-09 |
| 8 | type_DEBIT | 8.262479e-11 |
| 9 | type_PAYMENT | -2.419190e-08 |

Figure 11: Loading values for the data columns

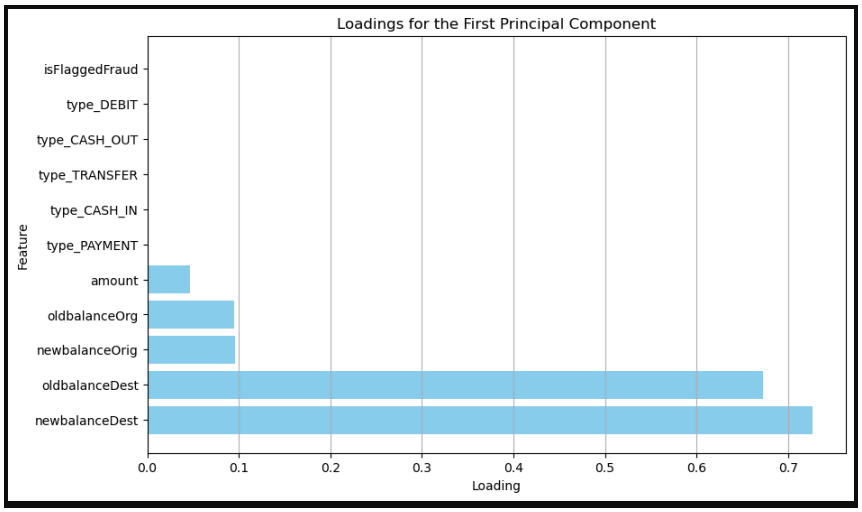


Figure 12: Loadings table plotted

**Validation Metrics:**

```
139183/139183 [==============================] - 451s 3ms/step - loss: 0.00
34 - accuracy: 0.9994
Epoch 2/3
139183/139183 [==============================] - 467s 3ms/step - loss: 0.00
28 - accuracy: 0.9995
Epoch 3/3
139183/139183 [==============================] - 479s 3ms/step - loss: 0.00
25 - accuracy: 0.9995
59650/59650 [==============================] - 156s 3ms/step - loss: 0.0027
- accuracy: 0.9995
Test Loss: 0.002739035990089178
Test Accuracy: 0.9995049238204956
59650/59650 [==============================] - 133s 2ms/step
```

Figure 13: Test Accuracies and Loss from Epocs from Neural Network model

The accuracy and loss metric is a different way to measure the performance of neural networks. However, it was important to analyze all of the models on the same scale, so the the F1 score was used for the final analysis. As seen in Figure 13, there was a final test accuracy of 0.9995, displaying a very high performance in this metric. While this helped validate the model, we chose not to include this metric. Since there were so many transactions that were not fraudulent and likely easy to detect for the neural network due to potential overfitting, we believe this value could be skewed higher than the actual performance of the model.