Fraud Detection based on Synthetic Financial Datasets

Ling Jiang, Xinyue Jin, Ming Ki Toby Cheng

1. Problem Description

Nowadays, the growing usage of digital payment (mobile money) makes sending and receiving money to and from people extremely fast and convenient. However, it has also increased the likelihood of fraudulent transactions. As e-commerces grows, it becomes more and more likely that a mobile money holder could become a victim of fraud.

Fraudulent transactions can be one of the biggest problems in digital payment. Banks are now under great pressure on perceiving and blocking fraudulent transactions proactively in order to protect their clients and prevent global money laundering on a global scale. If banks are able to accurately detect fraud, they can not only improve their reputation and customer loyalty but also decrease potential risks and increase profitability.

With the large amount of transactions and the changing characteristics carried out by fraudulent agents, the current methods in detecting fraud lack effectiveness. Most of the detection mechanisms are only based on simple assigned thresholds. For example, in this case only transactions over $200,000 would be flagged as fraudulent. Therefore, it is necessary to figure out the key features that are closely related to fraud so that we are able to prevent fraudulent transactions.

We decided to train several different classification models based on the transaction data we have in order to find the best one that can accurately detect fraudulent transactions. To this end, we attempted logistic regression, neural network and XGBoost in both PySpark and SageMaker.

2. Data and Data Preparation Process

We used Synthetic Financial Datasets for Fraud Detection from Kaggle. The data was generated by the PaySim mobile money simulator. Details about this simulator and the assumptions it makes can be found either on the Kaggle Site (https://www.kaggle.com/ntnu-testimon/paysim1) or its research publication (https://www.researchgate.net/publication/313138956_PAYSIM_A_FINANCIAL_MOBILE_MONEY_SIMULATOR_FOR_FRAUD_DETECTION). The reason we used the synthetic dataset is because real financial transaction information is unavailable because of privacy issues. All the real financial transaction data we found had the features masked usually through some dimensionality reduction technique like Principal Component Analysis (PCA). Although the dataset is synthetic, we can still use it to carry out fraud detection and find the best methods and apply it to future real data.

A. Data summary

There are in total 6,362,620 transactions in our dataset which contains the transactions from 6,353,307 customers who started those transactions and 2,722,362 recipients. Each transaction has several features, including step, type, amount, nameOrig, oldbalanceOrg, newbalanceOrig, nameDest, oldbalanceDest, newbalanceDest, isFraud, and isFlaggedFraud. There are no missing values. The description of these features is given in Table 1.

Table 1. Data Summary of Synthetic Financial Datasets for Fraud Detection

Variable	Description
step	Maps a unit of time in real world (1 step is 1 hour of time)
type	Types of Transactions (Cash-in, Cash-out, Debit, Payment, and Transfer)
amount	Amount of the transaction in local currency
nameOrig	Customer who started the transaction
oldbalance	Original balance before the transaction
newbalance	Original customer's balance after the transaction.
nameDest	Recipient ID of the transaction.
oldbalanceDest	Initial recipient balance before the transaction
newbalanceDest	Recipient's balance after the transaction
isFraud	Identifies a fraudulent transaction (1) and non fraudulent (0)
isFlaggedFraud	Flags illegal attempts to transfer more than 200,000 in a single transaction.

B. Data Exploratory Analysis

Before the prediction, we tried to dig out more valuable facts from our dataset, such as user fraud distribution and payment distribution types of all the transactions. For detailed code for the following visualizations and data preparation please refer to the Jupyter Notebook and HTML titled “Data Exploratory Analysis”.

B.1. Fraud distribution

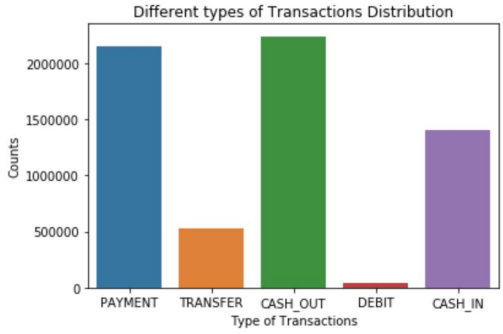
First, we plotted the fraud distribution of the dataset to see the amount of fraudulent cases overall. The result is shown as Figure 1. The graph shows that fraudulent cases are an extremely small part of total transactions. Compared to the total transactions, the amount of fraudulent cases is too small to be observed from the graph. There are 8,213 fraudulent transactions out of a total of 6,362,620 transactions, which is around 0.13% of overall transactions. This indicates that our dataset is extremely imbalanced. It is necessary for us to deal with this imbalanced data when we create models the details of which will be discussed later.

Figure 1. Fraud Distribution

B.2. The transactions distribution of different types of transaction

We also plotted the distribution of transactions based on the five different types of payment (Cash-in, Cash-out, Debit, Payment, and Transfer). The result is shown in Figure 2. The graph shows that most of the transactions are for Payment and Cash-Out; very small amounts of transactions come from Debit. More specifically, there are 2,237,500 transactions in Cash-out (35.2%), 2,151,495 transactions in Payment (33.8%), 1,399,284 transactions in Cash-in (22.0%), 532,909 transactions in Transfer (8.4%), and 41,432 transactions in Debit (0.65%).

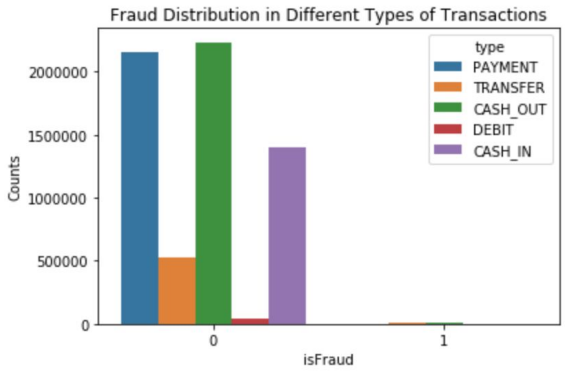
Figure 2. The transactions distribution of different types of transactions



B.3. Fraud distribution in different types of transactions

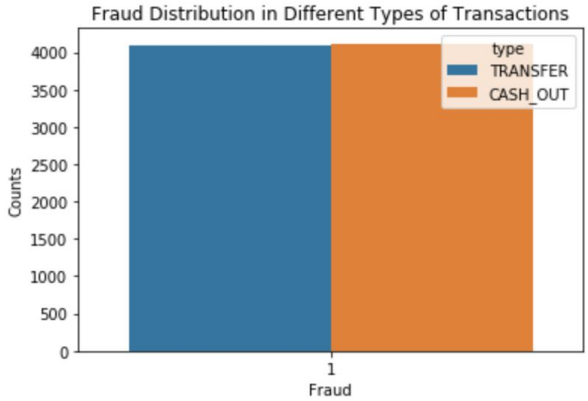
We also tried to analyze the frequency of fraud in these five different types of transactions to see which types of transactions may have a higher frequency of fraud. The result is shown in Figure 3. From the graph, it seems like not all types of transactions have fraud cases.

Figure 3. Fraud distribution in different types of transactions



To be more clear, we plotted the distribution of only fraudulent transactions in each transaction type. The result is shown in Figure 4. From the graph, it shows that only transactions in Cash-out and Transfer have fraud cases. There are 4,116 transactions in Cash-out (50.1%) and 4,097 transactions in Transfer (49.9%).

Figure 4. Fraudulent transaction distribution in different types of transactions



B.4. Fraud distribution in different origins and recipients of transactions

We can get the account types “C” (Customer) and “M” (Merchant) from variables “nameOrig” and “nameDest”, which would be the first character for each value and are not presented directly in the dataset. The feature named ”from_to” was created with two levels ”CC” (Customer to Customer) and “CM”(Customer to Merchant) because all of the transaction initiators in the

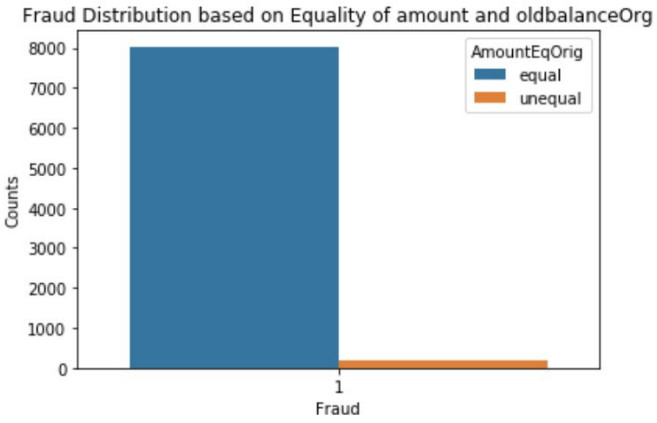
dataset were customers. From Figure 5 shown below, it indicates that fraudulent transactions only occured when the transaction was from customer to customer.

Figure 5. The fraud distribution in different origins and recipients of transactions

B.5. Fraud distribution based on the equality of amount and original balance

We also checked whether the equality of transaction amount and amount in the original balance of initial customers had an influence on fraudulent transactions. The result is shown as Figure 6. The graph shows that most fraudulent transactions occured when the amount being transacted equaled the original balance of initial customers before the transaction.

Figure 6. The fraud distribution based on the equality of amount and original balance



B.6. The correlations between numerical features

We calculated the correlations between numerical features to check whether there were high correlations between two features. The result is shown in Table 2. The table shows high correlations between oldbalanceOrg and newbalanceOrig, oldbalanceDest and newbalanceDest.

Table 2. Correlation between Numerical Features

	amount	oldbalanceOrg	newbalanceOrig	oldbalanceDest	newbalanceDest
amount	1.000000	-0.002762	-0.007861	0.294137	0.459304
oldbalanceOrg	-0.002762	1.000000	0.998803	0.066243	0.042029
newbalanceOrig	-0.007861	0.998803	1.000000	0.067812	0.041837
oldbalanceDest	0.294137	0.066243	0.067812	1.000000	0.976569
newbalanceDest	0.459304	0.042029	0.041837	0.976569	1.000000

C. Data Preparation

We generated 4 columns based on the original data. These 4 columns were the difference in the original user balance, the difference in the end user balance, whether or not the Amount transferred equaled the original user's starting balance, and whether the transaction was customer to customer (as opposed to customer to merchant). These variables were titled OrigDiff, DestDiff, AmountEqOrig, and CusToCus respectively. OrigDiff and DestDiff were of float type while AmountEqOrig and CusToCus were binary.

OrigDiff and DestDiff were selected because transfer-in and transfer-out amounts in every single transaction should be the same. If there is a difference, it is highly possible that the transaction is fraudulent. In addition, they could help address multicollinearity. Multicollinearity between features will make the estimates become sensitive to minor changes in our model, resulting in a negative impact on model performance.

From Figure 5 shown above, we found out that fraudulent transactions only existed among customer to customer transactions. Thus, CusToCus would be an essential indicator for detecting fraud.

In Figure 6, most of the fraudulent transactions had a common characteristic that the amount of transaction equaled the original balance of the initiator customer. This is because if the fraudulent agent wants to take the money from customers' account, there is a high possibility that he will take all money in the account. As a result, AmountEqOrig is also an important feature.

For PySpark, we randomly split the whole dataset into the training and test sets based on a fraction of 80% and 20% respectively. In SageMaker we split the whole dataset into training, validation, and test sets based on a 60%, 20%, 20% split respectively.

3. Approaches

We decided certain features in the dataset were unnecessary, either simply because it added no value to our modeling or because it was already covered by our feature engineering. As step is simply a way to measure time we decided it was not useful for the model. “nameOrig” and “nameDest” were unique customer and merchant identifiers which is also not useful for modeling. The effect of changing in balance in both the person who started and received the transaction is already captured in our OrigDiff and DestDiff variables so oldbalanceOrg, newbalanceOrig, oldbalanceDest, newbalanceDest aren’t useful and may cause additional issues with multicollinearity if we leave them in. So our final features chosen for further modeling were “type”, “amount”, “OrigDiff”, “DestDiff”, “AmountEqOrig” and “CusToCus”.

A. PySpark

For handling the data in PySpark, we first had to use one-hot encoding to handle the type feature since it was categorical. We used StringIndexer to give each type of transaction an index. “Cash-Out”, “Payment”, “Cash-In”, “Transfer”, and “Debit” were indexed as 0.0 to 4.0 in that order. Then we used OneHotEncoderEstimator to convert those indexes into a type vector called “typeVec” to use with the other features. Next we used VectorAssembler to vectorize all the features we wanted into a new column called “features_unscaled”. Finally we used StandardScalar to scale all the features and put it in a new column simply called “features”. To account for class imbalance in models that could handle class weights, we calculated the imbalance ratio (99.87% of labels were negative) and created a new column called “classWeights”. All non-fraud rows were given 0.00129 and all fraud rows were given 0.9987. For detailed code for all the PySpark Models and data preparation please refer to the Jupyter Notebook and HTML titled “Big-Data-2-Project-Pyspark”.

A.1. Logistic Regression

For Logistic Regression in PySpark we used LogisticRegression from the pyspark.ml.classification package. After training on the training set and predicting on both the training and the test set, we used BinaryClassificationEvaluator to examine the AUC of both the train and test predictions. If the AUC values between the two sets are very different (e.g. training AUC is much higher than test AUC) it is likely a sign of overfitting. The AUCs came out to be 0.99902 and 0.99934 for the training and test set respectively. These AUCs are extremely close and show no sign of overfitting so we continue with our analysis. As a bank we would like to capture as many fraud cases as possible, falsely identifying some cases as fraud is not our main concern. Therefore, we want to minimize false negatives and are not too concerned with our false positive numbers. To see how our model did in this aspect we examined the confusion matrix. On the training set, the model had 22 false positives and 25 false negatives while on the test set, the model had 4 false positives and 8 false negatives.

A.2. Neural Network (Multilayer Perceptron Classifier)

Another model we tried in PySpark was the Multilayer Perceptron Classifier which is most akin to a neural network model. We tried 1 hidden layer with only 1 node to try to avoid overfitting to the training data. Using the same data, the resulting train and test AUC were 0.99876 and 0.99892 respectively, which has no indications of overfitting. This model had 0 false positives on either training or test set but had 29 false negatives on the training set and 8 on the test set. As mentioned previously, we care more about lower false negatives so the logistic regression model performed better than this.

B. SageMaker

We also ran models in SageMaker to get more models to select from. We generated the same features (OrigDiff, DestDiff, AmountEqOrig, CusToCus) we did in PySpark only this time using SageMaker methods. We were able to directly read the file as a Pandas DataFrame and work with it directly. One-Hot-Encoding is much easier in SageMaker, we simply defined columns for each of the 5 types where if that transaction was that type, the relevant column would have value 1. We chose the same features as in the PySpark models and split the dataset into 60%/20%/20% random sets for training, validation, and test sets respectively and uploaded to our S3 bucket. We did the splitting differently with a different seed to PySpark as a type of sanity check that our models and features were robust to changes in the data and would still perform well.

B.1. Logistic Regression

We first tried logistic regression with simply the feature dimension = 10 and binary predictor type. We fit the model by providing the model with the training and validation datasets so the model would minimize loss up to the default threshold. Using the trained model, we used a predict function where the model generated a predicted based on each line of the test file, after which we extracted the “predicted_label”. To evaluate our model, we once again used a confusion matrix. As SageMaker takes much longer to generate predictions, we could only examine the model performance on the test set due to the size of the training set. However, ideally this is what we should be doing anyways since the test performance is most important. This model resulted in 12 false positives and 8 false negatives. For detailed code for this model and data preparation please refer to the Jupyter Notebook and HTML titled “Big-Data-2-Project-Log”.

We then tried a second logistic regression model this time adding the hyperparameters binary_classifier_model_selection_criteria = 'precision_at_target_recall' and target_recall = 0.99. This would allow the training job to select the model that has the best precision at the defined target recall (0.99). We tried to define target_recall as 1.0 but encountered errors. The result of the predictions from this second model had no improvement, it was identical to the first. This makes sense since the recall of the first model was already essentially 1.0. Seeing no way to further optimize our model, we moved on to the next model. For detailed code for this model

and data preparation please refer to the Jupyter Notebook and HTML titled "Big-Data-2-Project-Log-AutoThresh".

B.2. XGBoost

XGBoost with Amazon Sagemaker was also used for this case. The XGBoost (eXtreme Gradient Boosting) is an efficient implementation of the gradient boosted trees algorithm. Gradient boosting attempts to predict a target variable by combining an ensemble of estimates from a set of existing models. The training proceeds iteratively using a gradient descent algorithm to minimize the loss when adding new models. XGBoost is trained to minimize a regularized object function that combines a loss function based on the residuals of existing models and a penalty term for model complexity. It largely optimized execution speed and model performance because it not only can robustly handle various data types, relationships and distributions, but also has a large number of hyperparameters that can be tuned for model improvement. However, XGBoost is not a linear or logistic regression model so we are not able to get a single weight for each feature, and the prediction results from the XGBoost are not as easy to interpret as those from the Logistic Regression. Hence, it would be inappropriate to use it if we have to figure out feature importance to the target variable.

We first trained an XGBoost model with static hyperparameters of 5 max_depths, 0.2 eta, 4 gamma, 6 min_child_weight, 6 num_class, 10 num_round and multi:softmax objective. We fitted the model with training and validation datasets to avoid overfitting. Then, we used a Transformer to spawn a Batch-Transform job using the trained model, and predict labels for pre-uploaded testing data on S3. This could largely improve the work efficiency when we had a large amount of data to run inference on. Similar to logistic regression, we also used a confusion matrix as well as F-beta to evaluate the model performance. The model resulted in 0 positives and 8 false negatives, and F1 score was 0.9976. As we would like to focus more on false negatives, we set beta to 2 when calculating F-beta, and the result was 0.9962.

Next, we started a hyperparameter tuning job in order to find a better model. We tried to adjust the continuous hyperparameter eta in the range [0.1,0.5], subsample in the range [0.5, 1], min_child_weight in the range [0,10] and the integer hyperparameter max_depth in the range [1,10]. And we also set Hyperparameter Tuning Job Objective as "validation:f1". The prediction result after the tuning job was almost identical to the previous one, which was reasonable because the F1 score of the first model had been close to 1. The new model resulted in 2 positives and 8 false negatives, and F1 was 0.9976. As we would like to focus more on false negatives, we set beta to 2 when calculating F-beta, and the result was 0.9962.

For detailed code for these models and data preparation please refer to the Jupyter Notebook and HTML titled "Big-Data-2-Project-XGB-best".

4. Challenges and Solutions

A. Imbalanced Data

As mentioned previously, our dataset is extremely imbalanced. If models are fitted with the imbalanced data, the overall accuracy might be high, but the model may miss the concerned rows due to the low count. In this project, the value of finding the minority class (fraud transactions) is much higher than that of finding the majority (normal transactions). If we do not balance our data, we will have very low recall. In order to account for this challenge, we created the classWeights column to set weight when doing the models to balance our dataset.

B. Features Selection and Engineering

We found early on that the features we choose and create from the dataset had a large impact on being able to capture the fraudulent cases. As a result, a large portion of time was spent on understanding the data and engineering features. Additionally, we had to try to keep multicollinearity low between our original features and engineered solutions. Multicollinearity would make the estimates become sensitive to minor changes in our model, which has a bad influence on our model accuracy. This is why we decided to linearly combine the independent variables to address the multicollinearity. For this case, we subtracted old balances from new balances to create two new variables “OrigDiff” and “DestDiff” to avoid multicollinearity between the original balance variables.

C. Working with SageMaker

Unlike the previous assignment, we tried using Logistic Regression through Linear Learner SageMaker in addition to XGBoost. One major challenge we faced was largely due to the fact that SageMaker is relatively new (a little over 2 years old) and a lot of the bugs we ran into were hard to solve by looking at documentation. Additionally, because it’s so new there are not as many relevant posts with helpful answers for debugging online. An example is that it was not clear that all other models other than XGBoost are implemented with Apache MXNet. This means the model needs to be read differently when examining the coefficients since the file type is different and we need a kernel capable of using MXNet to import mxnet.

D. Models Selection

The challenge of model selection was to choose one among a set of candidate models. Models can be chosen based on Mean Squared Errors (MSE), F1 score, AUC, and so on. We needed to decide which factor to use in order to find the best fit model.

In this case, we are trying to detect fraud and banks care more about failing to capture real fraudulent transactions than mistakenly blocking legitimate transactions. So, we decided to select our models based on recall & F1 score in this project. In other words, we focus more on FN (false negative) in our predictions and choose the model with the smallest number of FN.

However, in the case that the models are highly similar in recall & F1 score, we would choose the logistics regression model since it provides interpretable explanations and indicates which features have great influences on predicting fraud.

5. Analysis results

Comparisons of results from different models are described in Table 3. We can see that all models worked well with high F1 scores and only 8 false negatives. Among them, Logistic Regression in PySpark had the highest AUC with 0.9993, while XGBoost in SageMaker and Neural Network in PySpark had the same highest F1 scores 0.9976. But the differences in these two measures among models are extremely small. Besides, all models got the same false negative number of 8. After carefully analyzing the predicted result, we found out that these eight transactions had the same characteristics. Their amounts were comparatively small, and the amounts were not equal to old origin balances. This might be an unique pattern that the model could not identify and something we could improve in the future.

As mentioned previously, we have to pay more attention to the failures in capturing the real fraudulent transactions in order to decrease the fraud risk. We decided to select our models based on F1 score, which can balance the precision and recall. Thus, after comparing different models, we chose the Logistic Regression model in PySpark as our final model to analyze and get insights in this project. We chose this model not only because of its low False Positive number (ensuring model accuracy), but also it was easier for us to interpret the results and find the important features, compared to Neural Network and XGBoost.

Table 3. Model Comparison

Model Performance Comparison on Testing Dataset					
Models	**Platforms**	**F1 Score**	**AUC**	**False Negative**	**False Positive**
Logistic Regression	PySpark	0.9963	0.9993	8	4
Neural Network (Multilayer Perceptron Classifier)	PySpark	0.9976	0.9989	8	0
Logistic Regression	SageMaker	0.9941	0.9976	8	12
XGBoost	SageMaker	0.9976	0.9976	8	0

We know that OneHotEncoder does not include the last category by default, meaning the effect of the last category of “Debit” transactions is in the intercept of the model. By looking at the remaining 9 feature coefficients in Table 4, we can more clearly see what are the main predictors of fraud. We take the exponent of the coefficients to see the odds ratios of each of the features and get the following:

Table 4. Nine feature coefficients

Feature	**Odds-Ratio**	**Interpretation (**Assuming all else constant)
Cash-Out	1.3319	33.19% higher fraud probability than Debit transactions
Payment	0.8565	14.35% lower fraud probability than Debit transactions
Cash-In	0.4113	58.87% lower fraud probability than Debit transactions
Transfer	1.2521	25.21% higher fraud probability than Debit transactions
Amount	1.5100	51% higher fraud probability per unit increase in amount
OrigDiff	0.9497	5.03% lower fraud probability per unit increase in original balance difference
DestDiff	0.6234	37.66% lower fraud probability per unit increase in destination balance difference
AmountEqOrig	1.500	50% higher fraud probability if Amount transferred equals origin balance
CusToCus	1.1675	16.75% higher fraud probability if the transaction is customer to customer

6. Insights gained

Our results reinforce what we found in our data exploration. Fraudulent transactions occur mostly in Cash-Out or Transfer transactions, when the amount being transferred is high, when the transfer amount is equal to the original balance, and when the transaction is being made from customer to customer.

We can imagine why this is the case from the perspective of the fraudulent agents. They would mostly work in cash out or transfer since they want to get the money out of the account and into their possession as quickly as possible. In fact, if we look at the data we see that the amount being transferred is also being cashed out in a short time frame, usually within the same time step. This also carries over to why customer to customer transactions are more likely fraudulent since they want to first transfer the user's money to an account they fully control before cashing out. Additionally, when they have access to a user's account they would want to empty the balance which is why the amount being transferred usually equals the entire balance and is a high amount.

Using this information, we could preemptively add security measures to our mobile payment application to prevent fraudulent transactions from even going through. This may come in the form of some kind of two-factor verification or other additional barriers for the transaction to go through to make sure it is the account owner. When our model predicts a high probability that a transaction being made is fraudulent we would either block it or require secure verification from the user.

7. Future work

Besides focusing on capturing fraudulent transactions based on our model, we also need to decrease the frequency of perceiving normal transactions as fraud cases mistakenly for our future consideration. Banks do decrease their risk by blocking as many fraud cases as they can; however, if the frequency of falsely regarding normal transactions as fraud is high, it will still hurt their customer loyalty, which leads to a bad influence on the development of banks. So, besides getting the minimum of FN, we also need to consider the number of FP (false positive) and avoid it being too high.

From the results we got from our models, the same number of false negatives in each model indicate that our models captured most fraudulent cases but could not identify certain patterns. In order to improve our model accuracy in the future, we should try to capture these unidentified patterns. For example, regarding the 8 fraud cases the model failed to capture, they contained some unique patterns of feature “amount” we hadn’t accounted for. Some of their amounts were quite high, but not as high as other transactions where transaction amount does not equal original balance. As a result, the resulting probability of fraud was lower and the model failed to identify it. One way we could address it is to split “amount” into bins instead of just putting numerical values into our model.

One last thing we can consider for the future is that our model will fail to capture newer fraud cases if fraudulent agents learn to avoid detection. As the characteristics of fraudulent cases change over time our original model will become less accurate. In order to capture new features that may have an effect on fraud, we would need to update our model to keep up with fraudulent agents.