Fraud Detection Report for Credit Card Transaction

Group 109
Shih-Ting Liu- 7923289754
Sih-Yu Huang- 6123726133
Chinkai Huang- 6552002967
Yi-Ching Lin- 7613130093

Date: May 5, 2022

Executive Summary

The USA Federal Trade Commission reported that there were 1.7 million frauds with a stunning loss of $1.9 billion in 2019 alone [Reference 1]. Worldwide, the global fraud losses rose up from $7.8 billion in 2010 to $32.82 billion in 2020 for only a decade [Reference 2]. The fraudsters did not let an opportunity go to waste during the pandemic, including situations where the country has shut down and consumers are vulnerable. Besides, our personal data is cheap. From the dark web, you could spend only $1 to own a social security number, and $30 to acquire the information of a credit card. To sum up, credit card fraud remains an important issue. Most of them are either account frauds or transaction frauds. In this report, we are going to focus on the later one and build real-time fraud algorithms that watch all incoming transactions and look for something unusual.

Our optimal goal is to detect fraud on credit card transaction data based on the dataset in 2006 from the government, unveiling and combating the increasingly sophisticated fraud. To achieve our goal, we first examine the dataset we possess and explore the distributions of our fields. We find out the peaks of fraud occurrence rate are in May, August and December. To go further in our analysis, we also clean the data to ensure the data quality is beyond the standard. We then carry out our feature engineering to generate 522 candidate variables. On top of that, we execute the feature selection process through filtering, wrapping and embedded mechanisms to narrow down the important features. After getting the best sets of features, we leverage machine learning algorithms to build 10 different kinds of supervised classification models and tune parameters to conclude an optimized real-time detection model, attempting to identify which transactions are most likely fraudulent.

Our 10 different supervised models explored and compared include Logistic Regression, Decision Tree, Random Forest, Boosted Tree, Gradient Boosting Tree, Catboost, XGboost, KNN, Support Vector Machine and Neural Network. The Neural Network model could detect 60.28% fraudulent transactions in the top 3% of our data, which is the best one among these ten models. After implementing our final model, we take card numbers and merch numbers as time dependence examples to validate our predictions and calculate financial impact at different population cutoffs.

Ideally, our model could help us eliminate 60.28% of the fraud by declining 3% of the transactions and, for a score cutoff at 4%, we could save about $1.25 million annually. Other than being utilized in a credit card fraud detection system, our model could be similarly developed and performed in related business areas, such as telecommunications and insurance, to avoid or detect fraudulent activity.

I. Description of Data

The dataset includes credit card transaction data from a US government organization in 2006. The purpose is to identify transaction fraud. The dataset contains 10 fields and has 96,753 records, with 1,059 records labeled as fraud. Two of the fields are numerical and the others are categorical. Here we have a sneak peek at numerical fields and some fields from categorical ones.

Field Summary Table

1. Numerical Fields

Field Name	% Population	Min	Max	Mean	Stdev	% Zero
date	100%	2006-01-01	2016-12-31	N/A	N/A	0%
Amount	100%	0.01	3,102,045.53	427.89	10,006.14	0%

2. Categorical Fields

Field Name	% Population	# Unique Values	Most Common Value
Recnum	100%	96,753	N/A
Cardnum	100%	1,645	5142148452
Merchnum	***96.51%***	13,091	930090121224
Merch description	100%	13,126	GSA-FSS-ADV
Merch state	***98.76%***	227	TN
Merch zip	***95.19%***	4,567	38118
Transtype	100%	4	P
Fraud	100%	2	0

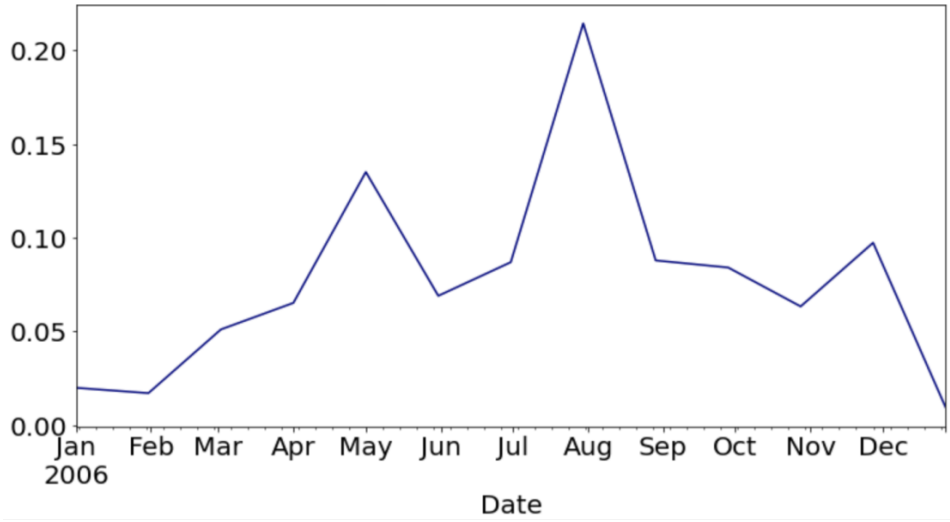
Field Description and Distribution

Below are the fields that we find some insights and would like to elaborate in more detail. The description and distribution of the entire dataset is in the appendix.

1. Date:

It is a numerical field representing the transaction date, with 100% populated. The date ranges from 2006-01-01 to 2006-12-31.The plots below show the fraud rate on a monthly basis. We can tell there are spikes in specific time periods.

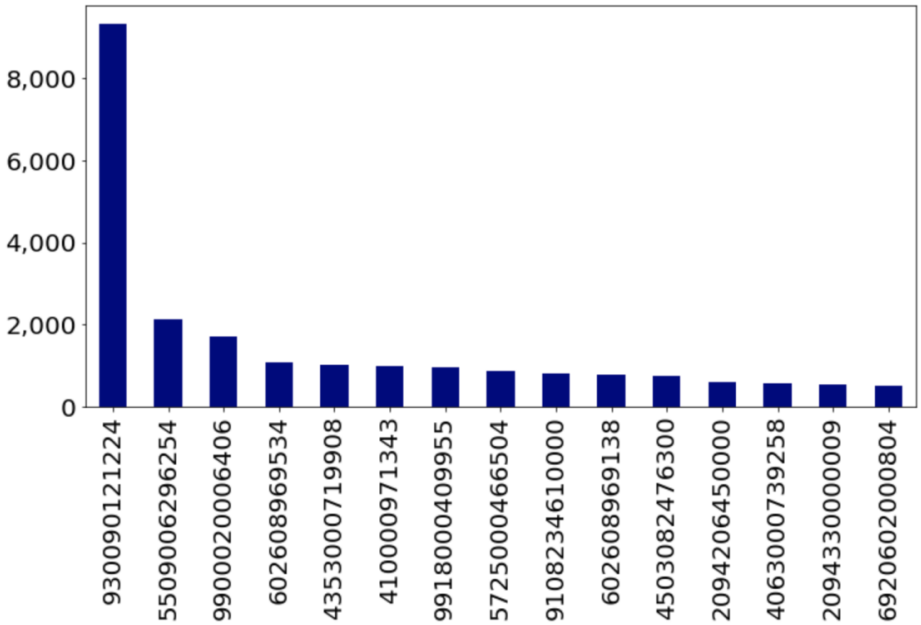
[Figure1.1.1] Monthly Fraud Occurrence Rate



2. Merchnum:

Merchnum is a categorical field and covers 96.51% of records. There are 13,091 unique values, and the most common value is 930090121224.

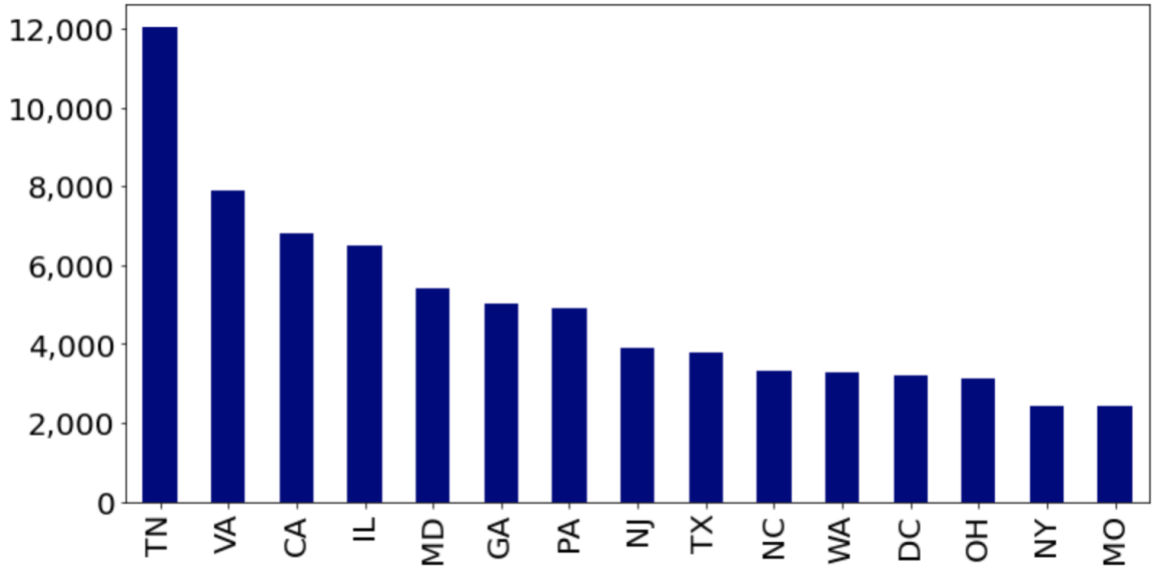
[Figure 1.1.2] The distribution of Merchant's number



3. Merch state:

It is a categorical field representing the merchant's state of the transaction, with 98.76% populated and 227 unique values. The most common value is TN, and the distribution of the top 15 categories of the field is as below.

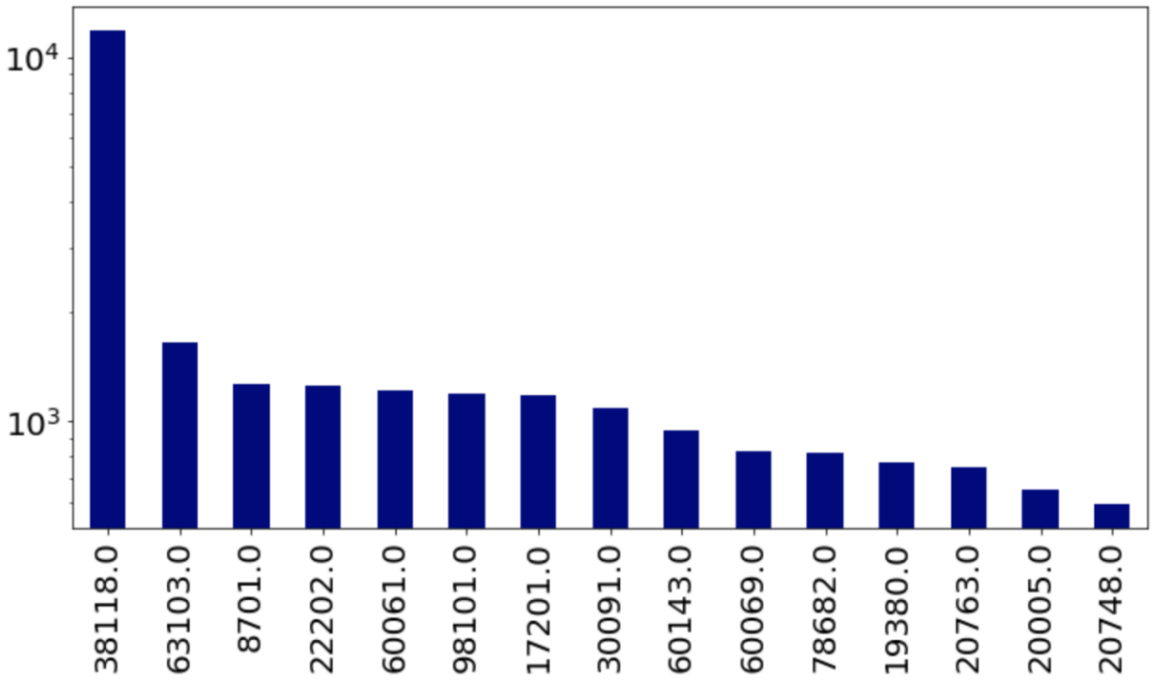
[Figure1.1.3] The distribution of Merchant's state



4. Merch zip:

Merch zip is a numerical field and covers 95.19% of records. There are 4,567 unique values, and the most common value is 38118.

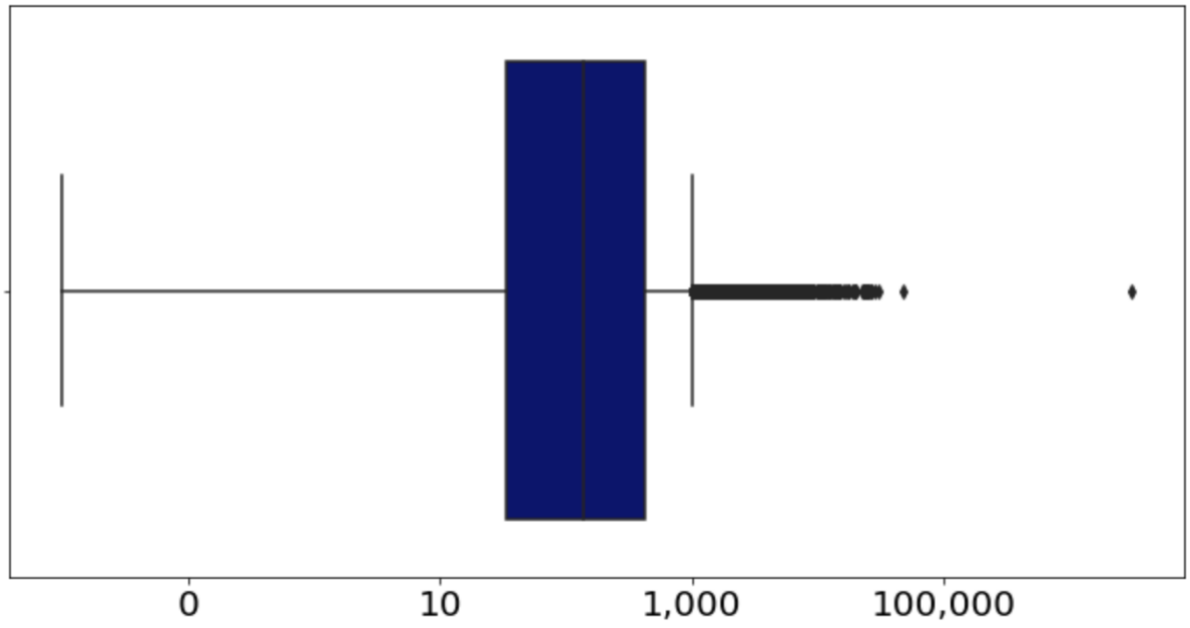
[Figure 1.1.4] The distribution of Merchant's zip



5. Amount:

It is a numeric field representing the transaction amount, with 100% populated and 0% zero. The min and max of the field are 0.01 and 3,102,045.53. From the boxplot below we can tell that there are extreme outliers, which we will deal with in the data cleaning process.

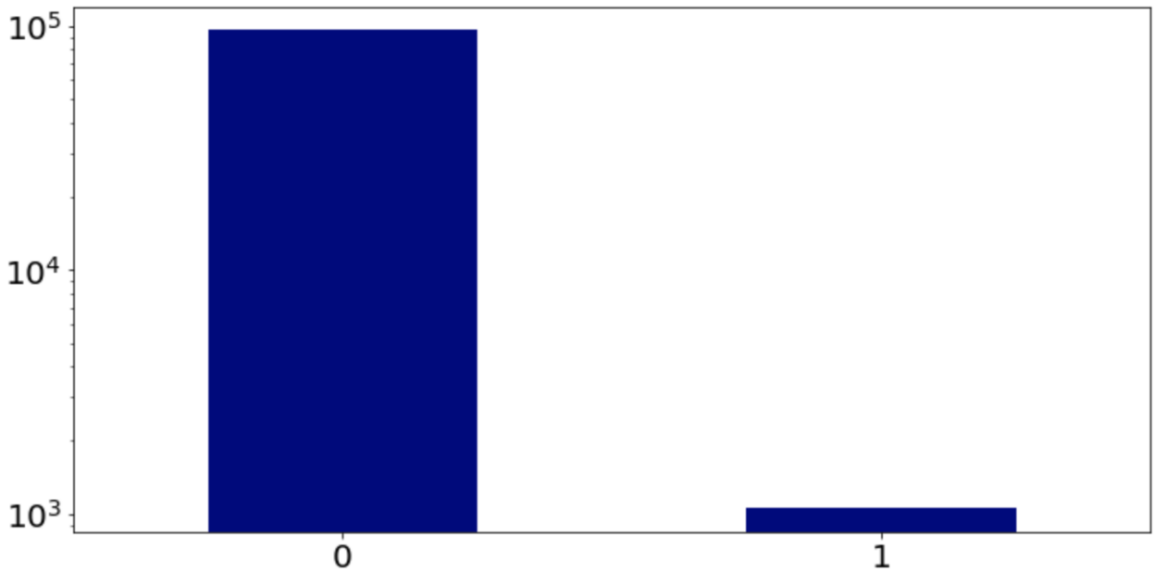
[Figure 1.1.5] The box plot of Amount



6. Fraud

It is a categorical field representing whether the transaction is classified as a fraud, with 100% populated and 2 unique values (1=fraud and 0=non-fraud). About 1.1% of the transactions are classified as fraud.

[Figure 1.1.6] The distribution of whether the transaction is a fraud



II. Data Cleaning

Before diving into data and running the analysis directly, we first need to clear our data to ensure the quality of data is good for further analysis, otherwise, it could be disastrous to our process and analysis.

First, we identify exclusions and bad records. We only keep purchasing data whose transtype is 'P', and remove one single transaction amount which is over 3 million dollars. Secondly, we fill in missing values within Merchnum, Merch state, and Merch zip.

For Merchnum, there are many rows with value 0, and if the Merchnum equals 0, we replace it with a null value. And since we already know that the Merch description will be the same for the same Merchnum, we decide to fill in these missing Marchnum values by mapping with corresponding Merch description. For Merch state, we first look at whether the record has a zip code. If there is a zip code, then we assign the state that zip code belongs to. Otherwise, we filled in these null values by mapping with corresponding Merchnum and Merch description. Similarly, the missing Merch zip values were mapped with linking Merchnum and Merch description. In the end, for these three fields, we then replace adjustments transactions where Merch description equals 'RETAIL CREDIT ADJUSTMENT' or 'RETAIL DEBIT ADJUSTMENT' and the left missing values with "Unknown".

III. Candidate Variables

Feature creation from original data is an important step in the project since it may affect the performance of our models significantly. In this process, we want to create as many candidate variables as possible, and later we will use feature selection methods to reduce the dimension of variables and choose a final set of possible model inputs. To create the candidate model variables, we first think about the signals of potential transaction fraud such as:

- Burst of activity at different merchants
- Larger than normal purchase amounts, same or different merchants
- Used at merchants not used before for that card
- Used at a very different geography
- Used at a high-risk merchant
- Increased usage in card-not-present
- Employee or merchant invents transactions
- Infrequent recurring charges, same amount or same merchant
- Employee or merchant invents a fictitious merchant

Based on these signals, we then create five categories of variables below:

1. New entities:

The first step of our feature creation is to build new entities. We try to see if there are abnormal transactions related to merchants and cardholders so we link Cardnum and Merchnum with Merch state, Merch zip and Amount. These variables are further used in creating categories of variables below.

2. Day-since variables:

These variables measure how much time has passed since a certain transaction last appeared in the dataset. These variables are important. For example, if the last time a card was used was more than half of a year ago, then it might be a fraud when someone else uses it again.

3. Frequency variables:

These variables mean how often a certain transaction over the past 0, 1, 3, 7, 14 and 30 days. These variables are also helpful for detecting fraud. For example, if one person usually does 5 transactions per day. However, he or she does a 10,000 transaction in one day. Then the probability of the cards being stolen is very high.

4. Amount variables:

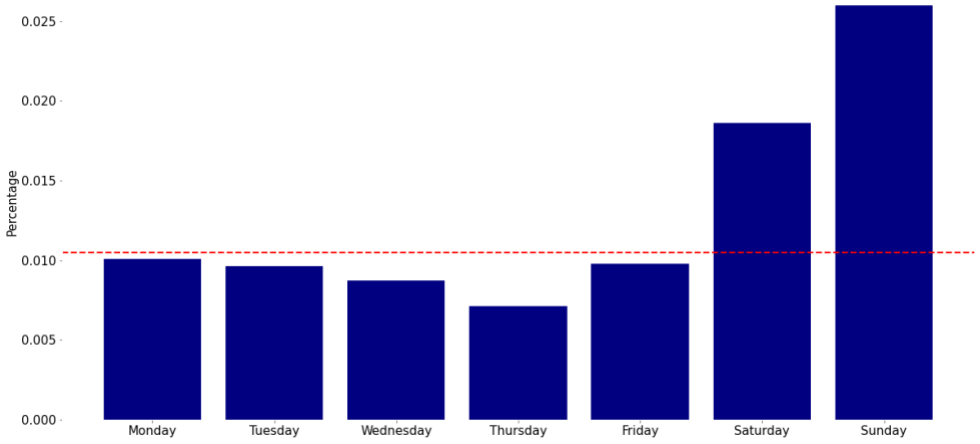
These variables mean the Average, Maximum, Median, Total, Actual/maximum, Actual/median, and Actual/total amount for a certain transaction over the past 0, 1, 3, 7, 14, and 30 days. Seeing the difference in amount spent is an obvious way to detect potential fraud. For example, if a person usually spends less than 500 dollars over a certain time span and one day he or she spends over 500,000 dollars in one transaction, it will be abnormal.

5. Relative velocity variables:

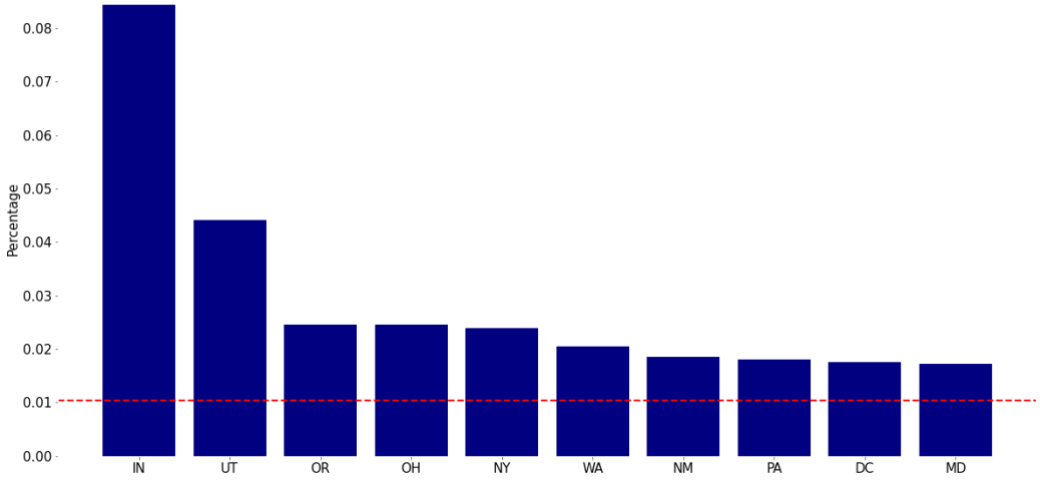
These variables measure the ratio of frequency of a transaction or amount spent in a small time window with a larger time window. By looking at these variables, we can know whether this person has peak frequency of transaction or amount spent over a short-term period compared to his average transactions in the long term.

Besides the five major categories, we also create a day of week and state risk table, which is a measurement of the probability of finding fraud on a certain day of a week or a certain state. These target encodings were created on data in which the most recent two-month data time frame as an out-of-time (OOT) sample was removed and is statistically smoothed by sigmoid function with the overall average to avoid overfitting. Figure 3.1.1 displays the day of week risk table. It shows that Saturday and Sunday have a higher likelihood of fraud. Figure 3.1.2 displays the state risk table of the top ten fraud proportion. We can see that Indiana and Utah state have the highest likelihood of fraud.

[Figure 3.1.1] Day of Week Risk Table



[Figure 3.1.2] State Risk Table



Last, we create Benford's law variables. These variables measure whether the variables violate Benford's law. Based on the intuitive fact that the first digit of many measurements is not uniformly distributed, it will check the level of unusualness of the amounts for Cardnum and Merchnum based on the distributions. Top 5 Cardnum with highest unusualness are 5142253356, 5142299705, 5142197563, 5142194617, and 5142288241. And the top 5 Merchnum with highest unusualness are 991808369338, 8078200641472, 308904389335, 3523000628102, and 808998385332.

After feature creation, we ended up creating 522 new variables. Table 3.1.1 shows the description of new variables that are create:

[Table 3.1.1] Description of New Variables

Description of variables	#Variable
Day of Week target encoding: dow_risk, State target encoding: state_risk	2
New Entities for Variables: {'card_state', 'card_zip', 'card_amount', 'merch_amount', 'card_merch', 'merch_zip', 'merch_state'}	7
Days-since: # days since a transaction with that entity has been seen. Entities are {'Cardnum', 'Merchnum', 'card_state', 'card_zip', 'card_amount', 'merch_amount', 'card_merch', 'merch_zip', 'merch_state'}	9
Frequency: # transactions at that entity over the past n days. Entities are {'Cardnum', 'Merchnum', 'card_state', 'card_zip', 'card_amount', 'merch_amount', 'card_merch', 'merch_zip', 'merch_state', ('Cardnum','Merchnum'), ('Merchnum','Cardnum'), ('Amount','Cardnum'), ('Amount','Merchnum')}, n is {0, 1, 3, 7, 14, 30}	78
Amount: {Average, Maximum, Median, Total, Actual/average, Actual/maximum, Actual/median, Actual/total} amount at that entity over the past n days. Entities are {'Cardnum', 'Merchnum', 'card_state', 'card_zip', 'card_merch', 'merch_zip', 'merch_state'}, n is {0, 1, 3, 7, 14, 30}	336
Relative velocity for number: # transactions at that entity seen in a short time window {0 or past 1 day} compared to # transactions at that entity over a longer time window {past 3, 7, 14, 30 days} Entities are {'Cardnum', 'Merchnum', 'card_state', 'card_zip', 'card_merch', 'merch_zip', 'merch_state'}	72
Relative velocity for amount: Amount of transactions at that {'Cardnum', 'Merchnum'} in a short time window {0 or past 1 day} compared to amount of transactions at that {'Cardnum', 'Merchnum'} over a longer time window {past 3, 7, 14, 30 days}	16
Benford's Law: Use two entities Cardnum and Merchnum to calculate the unusualness, *U**	2
Total	522

IV. Feature Selection

The variables we create from the previous step may contain valuable information that we can utilize. Nevertheless, these could not be directly put into our machine learning models, since the dimensionality would be too high to process with all the variables. This would take an extremely long time, leading our algorithms computationally intractable and may cause overfitting problems for the result. Therefore, we need to perform feature selection. By doing so, we can reduce dimensionality, allow the model to focus only on the important features of the data, and increase the accuracy of detecting fraud in credit card transactions.

There are three major categories of feature selection methods: filter, wrapper and embedded. Filter methods measure general characteristics of the dataset such as the relevance of features by their correlation with dependent variables. Wrapper methods compute models with a certain subset of features and evaluate the importance of each feature. Then they iterate and try a different subset of features until the optimal subset is reached. The most notable wrapper methods include forward selection, backward selection, and SelectKBest. And embedded methods perform feature selection as a part of the model training process. Lasso and Ridge regression are the two most common feature selection methods of this type, intending to shrink many features' coefficients to zero or almost near to zero. In this project, we apply all these three categories of feature selection methods. And we try 6 different assemblies for these methods to get 6 different sets of features to put into our models and see which subsets will have better performance.

1. Filter + Forward Stepwise Wrapper Selection:

The first assembly of feature selection methods is applying filter and forward selection wrapper. During the process of running a filter, we first remove the last two months of records as the out-of-time data (OOT) and remove the first two weeks of records since these were not formed well. In addition, we add a column with a series of random numbers to avoid the model performing below the bar.

The filter methods we apply here are Kolmogorov-Smirnov (KS). Kolmogorov-Smirnov (KS) test is a measure to examine how well a given variable separates the distribution of frauds and non-frauds. Since the distribution of our data is discrete, we have:

$$KS = max_x \sum_{x_{min}}^{x} (P_1 - P_2)$$

Our KS score is calculated based on the formula above. The higher the score is, the better each feature by itself can potentially separate fraud from non-fraud. We then score and rank each feature by this KS score. The top 80 features are selected for further selection by wrapper methods.

After obtaining the best 80 features from filter selection, we then put them into a forward stepwise wrapper selection for more fine-tuned selection. Forward stepwise selection begins with a model that contains no variables and adds the most significant variables one after the other. It will stop until a pre-specified stopping rule is reached or until all the variables under consideration are included in the model.

Here, we run the selection with the LightGBM classification model and rank the variables by the fraud detection rate (FDR) at 3%, which means the percent of total frauds are caught by the model at 3% of the population, assorted by the fraud score, as our wrapper scores. Twenty best features are selected to build our models. Table 4.1.1 shows the final 20 features by using this assembly.

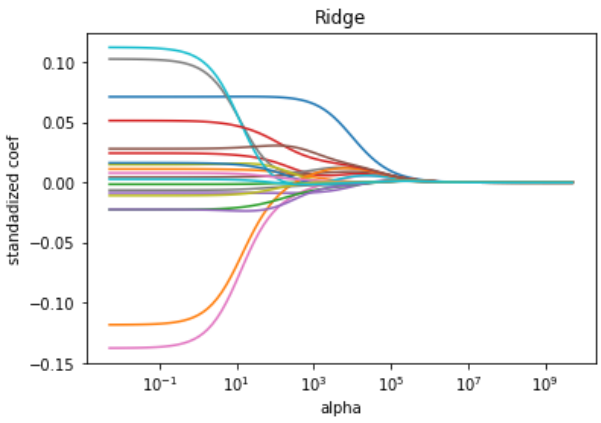
[Table 4.1.1] Top 20 Features

card_zip_total_3	card_state_max_30	card_merch_total_14	card_merch_total_7
card_state_total_30	card_merch_avg_14	merch_state_avg_1	merch_zip_avg_0
Merchnum_avg_0	merch_zip_avg_1	Merchnum_avg_1	card_zip_avg_14
card_merch_avg_7	merch_state_avg_0	Cardnum_total_1	card_merch_total_3
card_merch_avg_3	card_merch_total_0	card_state_total_0	card_zip_total_0

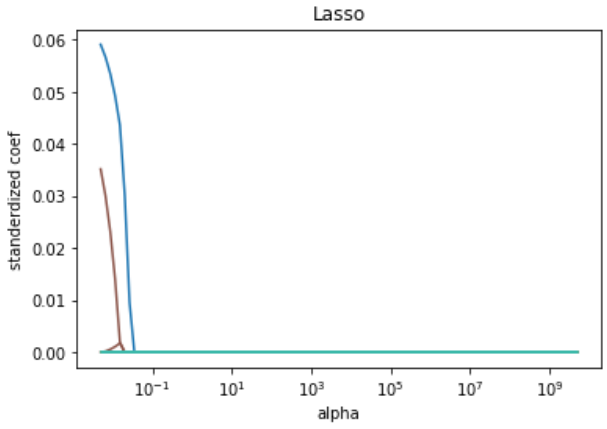
2. Filter + Forward Stepwise Wrapper Selection + Embedded:

After getting variable set from the filter selection and forward stepwise wrapper selection, we also put these features into ridge and lasso regression to see if the coefficients could be further standardized to build a better model, further shrinking coefficients of unimportant features to zero or almost near to zero. Both ridge and lasso regression contain a penalty parameter alpha that controls the amount of shrinkage. The only difference between them is that in ridge regression, the cost function is altered by adding a penalty equivalent to the square of the coefficients and absolute magnitudes are taken into account in lasso regression. Figure 4.2.1 and Figure 4.2.2 display the standardized ridge regression coefficients and lasso coefficients as a function of alpha respectively.

[Figure 4.2.1] The standardized ridge coef



[Figure 4.2.2] The standardized lasso coef



We can see from Figure 2 that some coefficients drop quickly to zero in lasso regression. We thus only keep the variables that their coefficients do not shrink to zero by lasso regression as the model input. The variables include: {card_merch_total_7}.

3. Filter + Backward Stepwise Wrapper Selection:

The next assembly is running filter selection and the backward selection wrapper. We first apply Kolmogorov-Smirnov (KS) in the filter process to rank each feature and choose the top 80 features. We then put them into a backward stepwise wrapper selection for more fine tuned selection. Backward stepwise selection begins with a model that contains all variables under consideration and starts removing the least significant variables one after the other. It will stop until a pre-specified stopping rule is reached or until no variable is left in the model.

Here, we run the selection with the LightGBM classification model and rank the variables by the fraud detection rate (FDR) at 3%. Twenty best features are selected to build our models. Table 4.3.1 shows the final 20 features by using this assembly.

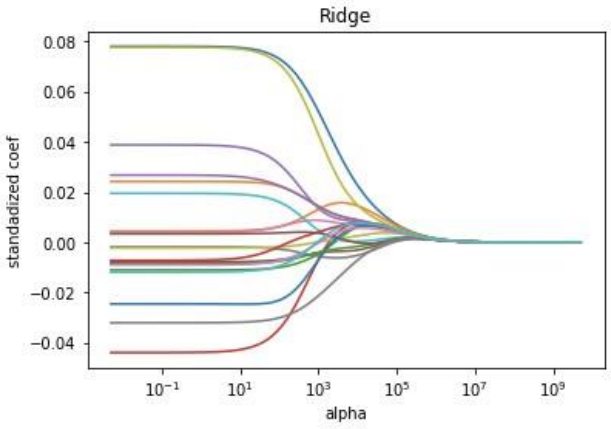
[Table 4.3.1] Top 20 Features

card_merch_total_7	card_zip_total_3	card_merch_total_14	card_zip_total_14
card_state_total_7	card_state_total_14	card_state_total_1	card_zip_max_30
card_state_total_30	card_state_max_14	card_zip_max_1	merch_state_total_3
merch_state_total_1	Cardnum_total_3	Cardnum_total_7	card_state_max_30
card_merch_avg_30	merch_state_max_1	Cardnum_total_1	merch_zip_total_0

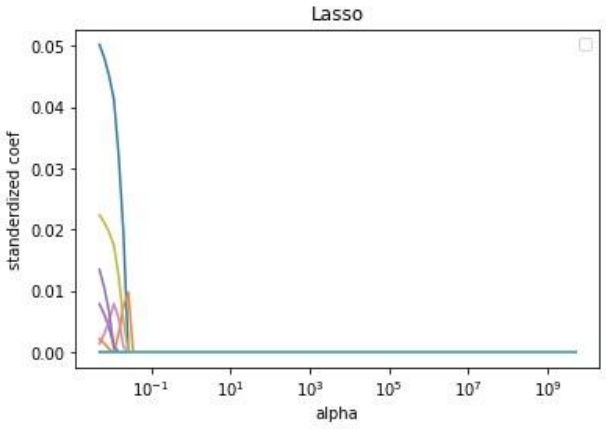
4. Filter + Backward Stepwise Wrapper Selection + Embedded:

After getting variable set from the filter selection and backward stepwise wrapper selection, we also put these features into ridge and lasso regression to see if the coefficients could be further standardized to build a better model, further shrinking coefficients of unimportant features to zero or almost near to zero. Figure 4.4.1 and Figure 4.4.2 display the standardized ridge regression coefficients and lasso coefficients as a function of alpha respectively.

[Figure 4.4.1] The standardized ridge coef



[Figure 4.4.2] The standardized lasso coef



We can see from Figure 4 that some coefficients drop quickly to zero in lasso regression. We thus only keep the variables that their coefficients do not shrink to zero by lasso regression as the model input. The variables include: {Cardnum_total_7, card_state_total_7, card_zip_max_1, card_zip_max_30, merch_zip_total_0}.

5. SelectKbest Wrapper:

Then, we run SelectKbest wrapper. The SelectKBest method selects the features according to the k highest score. We run the selection with the random forest classification model and use Chi-squared statistics as our score function. A large value will mean the feature is non-randomly related to a dependent variable, and so likely to provide important information. Twenty best features are selected to build our models. Table 4.5.1 shows the final 20 features by using this assembly.

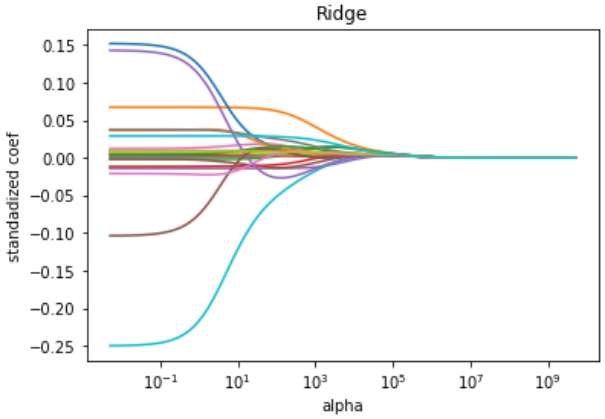
[Table 4.5.1] Top 20 Features

Cardnum_total_0	Cardnum_total_1	Cardnum_total_3	'Cardnum_total_7
Cardnum_total_14	'card_state_total_1	card_state_total_3	'card_state_total_7
card_state_total_14	'card_zip_total_1	card_zip_total_3	'card_zip_total_7
card_zip_total_14	'card_zip_total_30	card_merch_total_1	'card_merch_total_3
card_merch_total_7	'card_merch_total_14	card_merch_total_30	'merch_state_total_1

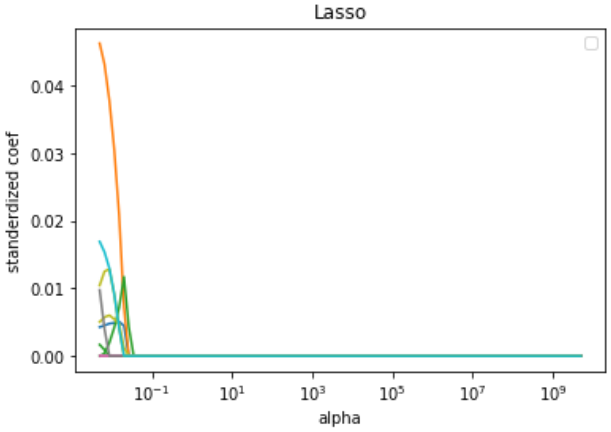
6. SelectKbest Wrapper + Embedded:

After getting variable sets from the SelectKbest wrapper selection, we also put these features into ridge and lasso regression to see if the coefficients could be further standardized to build a better model, further shrinking coefficients of unimportant features to zero or almost near to zero. Figure 4.6.1 and Figure 4.6.2 display the standardized ridge regression coefficients and lasso coefficients as a function of alpha respectively.

[Figure 4.6.1] The standardized ridge coef



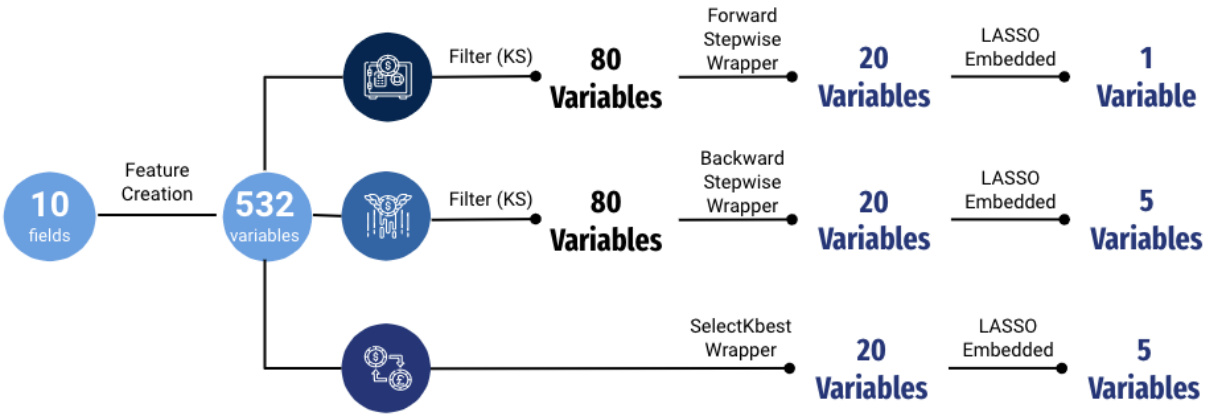
[Figure 4.6.2] The standardized lasso coef



We can see from Figure 4.6.2 that some coefficients drop quickly to zero in lasso regression. We thus only keep the variables that their coefficients do not shrink to zero by lasso regression as the model input. The variables include {card_state_total_3, card_zip_total_3, card_merch_total_3, card_merch_total_7, card_merch_total_14 }.

We begin our feature selection process with 522 expert variables created, and apply different assemblies of filter, wrapper, and embedded methods to reduce our features to 6 different variable sets. Figure 4.7 below shows the pipeline and the flow of how our variables are created and feature selection.

[Figure 4.7] Pipeline of Variables Creation and Feature Selection



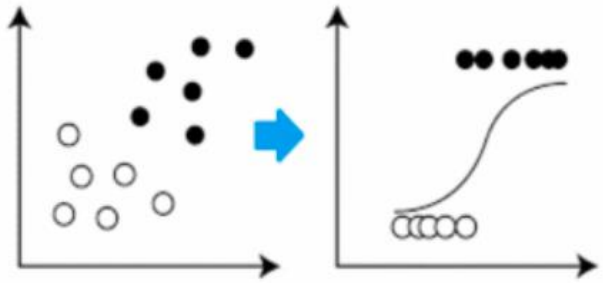
V. Model Algorithms

Since our goal is to predict whether a credit card transaction is fraud, we try 10 classification algorithms with different variable sets to build our classifiers. Moving forward, we would like to input the variable sets selected from our feature selection process to our default models and see which one performs better and then choose the best one to keep tuning the hyperparameters and get our final model. The following are the description of all the model algorithms we are using and the results of our trial.

1. Logistic regression

Logistic regression predicts the probabilities for classification problems with two possible outcomes. Because we have more than one variable, we use Multiple Logistic Regression.

[Figure 5.1.1]



After running 6 combinations of variables in the default setting, we find that the variable selected from ***forward feature selection with lasso regression*** has the highest FDR for OOT. Then, we use this variable set to tune parameters. There are 3 related hyper-parameters tuned for our Logistic Regression algorithm:

- penalty: Specify the norm of the penalty
- C: Inverse to regularization strength (smaller values specify a stronger regularization)
- solver: Algorithm to use in the optimization problem

According to the table below, we can find that our baseline model performances are not aggressively improved after tuning since there are not many variables included to have an impact.

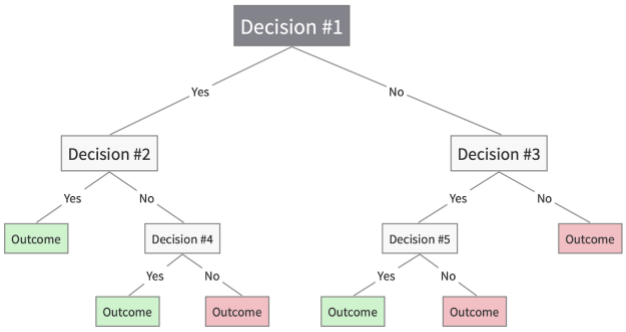
[Table 5.1.1]

Iteration	Parameter				Remark	Average FDR at 3%		
No.	# variable	penalty	C	solver	Fearure Selection method	Train	Test	OOT
1 (default)	20	l2	1	lbfgs	SelectKbest	67.11%	68.24%	33.91%
2 (default)	20	l2	1	lbfgs	Backward	67.00%	67.88%	29.55%
3 (default)	20	l2	1	lbfgs	Stepforward	63.25%	62.60%	24.58%
4 (default)	5	l2	1	lbfgs	SelectKbest_Lasso	63.12%	62.92%	34.25%
5 (default)	5	l2	1	lbfgs	Backward_Lasso	64.84%	65.86%	33.41%
6 (default)	**1**	**l2**	**1**	**lbfgs**	**Stepforward_Lasso**	**63.25%**	**62.29%**	**36.31%**
7	1	l2	0.1	saga	Stepforward_Lasso	63.25%	62.35%	36.31%
8	1	l1	1	saga	Stepforward_Lasso	63.11%	62.51%	36.31%
9	1	i2	0.00001	lbfgs	Stepforward_Lasso	62.82%	63.45%	36.31%

2. Decision Tree

Decision Tree is a non-parametric supervised learning method used for classification. They are a class of machine learning models that can be thought of as a sequence of "if" statements to apply to an input to determine the prediction.

[Figure 5.2.1]



After running 6 combinations of variables in the default setting, we find that the variable selected from ***forward feature selection*** has the highest FDR for OOT. Then, we use the combination of 20 variables to tune parameters. There are 5 related hyper-parameters tuned for our Decision Tree algorithm:

- criterion: The function to measure the quality of a split
- max_depth: The maximum depth of the tree
- min_samples_leaf*:* The minimum number of samples required to be at a leaf node
- min_samples_split: The minimum number of samples required to split an internal node
- splitter*:* The strategy used to choose the split at each node

According to the table below, we can find that the model with 10 splits, 60 leaf nodes and maximum depth of the tree as 8 perform better without overfitting. ***The model can detect 45.14% fraud in the top 3% of our data.***

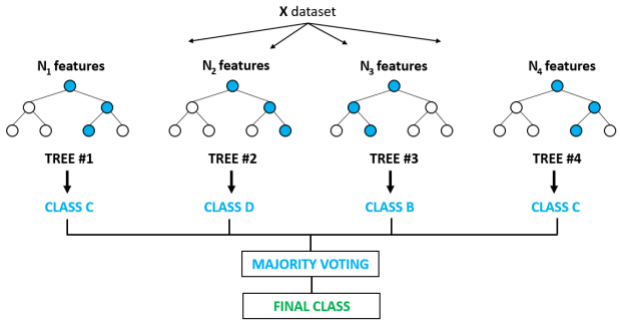
[Table 5.2.1]

Iteration	Parameter						Remark	Average FDR at 3%		
No.	# variable	criterion	max_depth	min_samples_leaf	min_samples_split	splitter	FS method	Train	Test	OOT
1 (default)	20	gini	None	1	2	best	SelectKbest	100.00%	53.09%	25.75%
2 (default)	20	gini	None	1	2	best	Backward	100.00%	67.54%	25.70%
3 (default)	**20**	**gini**	**None**	**1**	**2**	**best**	**Stepforward**	**100.00%**	**62.42%**	**26.42%**
4 (default)	5	gini	None	1	2	best	SelectKbest_Lasso	100.00%	37.63%	21.51%
5 (default)	5	gini	None	1	2	best	Backward_Lasso	100.00%	57.72%	24.13%
6 (default)	1	gini	None	1	2	best	Stepforward_Lasso	100.00%	29.94%	19.94%
7	20	gini	20	60	300	best	Stepforward	77.61%	70.83%	40.34%
8	20	gini	None	500	2	best	Stepforward	67.22%	66.90%	37.77%
9	20	entropy	50	1	10	random	Stepforward	76.61%	69.62%	42.12%
10	**20**	**gini**	**8**	**60**	**10**	**best**	**Stepforward**	**72.15%**	**70.24%**	**45.14%**
11	20	gini	7	65	10	best	Stepforward	73.68%	70.18%	43.13%
12	20	gini	10	1	50	best	Stepforward	72.76%	70.47%	44.92%
13	20	gini	10	1	55	best	Stepforward	74.13%	67.95%	45.47%
14	20	gini	8	1	100	best	Stepforward	70.97%	66.52%	45.14%
15	20	gini	10	1	100	best	Stepforward	71.77%	67.64%	44.92%

3. Random Forest

Random forest is an ensemble of decision trees. It builds a collection of many independent strong trees and makes majority voting across them. Instead of searching for the most important feature while splitting a node, it searches for the best feature among a random subset of features. This results in a wide diversity and generates a better model.

[Figure 5.3.1]



After running 6 combinations of variables in the default setting, we find that the variable selected from ***backward feature selection*** has the highest FDR for OOT. Then, we use the combination of 20 variables to tune parameters. There are 5 related hyper-parameters tuned for our Random Forest algorithm:

- criterion*:* The function to measure the quality of a split
- max_depth: The maximum depth of the tree
- min_samples_leaf: The minimum number of samples required to be at a leaf node
- min_samples_split*:* The minimum number of samples required to split an internal node
- n_estimators: The number of trees in the forest

According to the table below, we can find that the model with 100 trees, 40 splits, 160 leaf nodes and maximum depth of the tree as 50 performs better without overfitting. ***The model can detect 58.83% fraud in the top 3% of our data.***

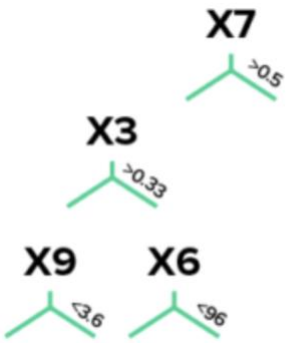
[Table 5.3.1]

Iteration	Parameter						Remark	Average FDR at 3%		
No.	# variable	criterion	max_depth	min_samples_leaf	min_samples_split	n_estimators	FS method	Train	Test	OOT
1 (default)	20	gini	None	1	2	100	SelectKbest	99.98%	70.80%	46.48%
2 (default)	**20**	**gini**	**None**	**1**	**2**	**100**	**Backward**	**100.00%**	**82.77%**	**52.96%**
3 (default)	20	gini	None	1	2	100	Stepforward	100.00%	73.45%	40.56%
4 (default)	5	gini	None	1	2	100	SelectKbest_Lasso	100.00%	53.78%	30.84%
5 (default)	5	gini	None	1	2	100	Backward_Lasso	100.00%	77.42%	37.32%
6 (default)	1	gini	None	1	2	100	Stepforward_Lasso	99.98%	43.89%	25.03%
7	20	gini	None	20	3	50	Backward	100.00%	86.16%	59.66%
8	20	entropy	100	160	40	200	Backward	88.14%	82.54%	57.49%
9	20	gini	10	30	10	50	Backward	86.96%	81.56%	58.66%
10	**20**	**gini**	**50**	**160**	**40**	**100**	**Backward**	**87.54%**	**80.45%**	**58.83%**
11	20	gini	30	50	10	100	Backward	97.33%	85.88%	58.66%
12	20	gini	14	70	10	12	Backward	89.56%	82.21%	57.88%
13	20	gini	30	20	5	50	Backward	99.74%	85.98%	59.11%
14	20	gini	8	60	10	50	Backward	83.48%	79.79%	57.71%
15	20	gini	8	60	10	65	Backward	83.77%	79.42%	57.37%

4. Boosted Tree (LightGBM)

A boosted tree is also an ensemble of decision trees, but it builds a series of many weak trees, then adds them up. It grows trees vertically and chooses the leaf with maximum overall loss to grow. LightGBM also uses the Gradient-based One-Side Sampling technique when finding a split value, filtering the data to keep all the points with large gradients and randomly selecting a certain percentage of the remaining points.

[Figure 5.4.1]



After running 6 combinations of variables in the default setting, we find that the variable selected from ***backward feature selection*** has the highest FDR for OOT. Then, we use the combination of 20 variables to tune parameters. There are 4 related hyper-parameters tuned for our Boosted Tree algorithm:

- learning_rate: Boosting learning rate
- max_depth: Maximum tree depth for base learners
- num_leaves: Maximum tree leaves for base learners
- n_estimators: Number of boosted trees to fit

According to the table below, we can find that the model with 150 trees, 15 leaf nodes, learning rate as 0.01, and maximum depth of the tree as 5 performs better without overfitting. ***The model can detect 57.49% fraud in the top 3% of our data.***

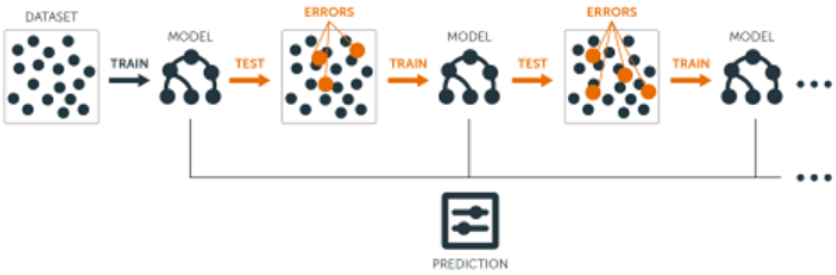
[Table 5.4.1]

Iteration	Parameter					Remark	Average FDR at 3%		
No.	# variable	learning_rate	max_depth	num_leaves	n_estimators	FS method	Train	Test	OOT
1 (default)	20	0.1	-1	31	100	SelectKbest	99.98%	75.52%	47.26%
2 (default)	**20**	**0.1**	**-1**	**31**	**100**	**Backward**	**100.00%**	**84.96%**	**49.44%**
3 (default)	20	0.1	-1	31	100	Stepforward	99.92%	79.52%	43.24%
4 (default)	5	0.1	-1	31	100	SelectKbest_Lasso	81.71%	64.75%	31.01%
5 (default)	5	0.1	-1	31	100	Backward_Lasso	99.39%	75.50%	37.82%
6 (default)	1	0.1	-1	31	100	Stepforward_Lasso	63.42%	63.12%	33.58%
7	20	0.05	3	50	100	Backward	87.04%	82.94%	55.14%
8	20	0.1	5	31	100	Backward	99.28%	86.69%	49.72%
9	20	0.1	3	31	175	Backward	99.98%	87.76%	50.56%
10	20	0.01	5	31	1000	Backward	98.82%	88.32%	56.70%
11	20	0.01	5	31	300	Backward	93.46%	85.57%	57.15%
12	20	0.01	5	31	150	Backward	90.96%	83.72%	57.04%
13	**20**	**0.01**	**5**	**15**	**150**	**Backward**	**88.41%**	**82.90%**	**57.49%**
14	20	0.01	5	15	140	Backward	88.75%	82.21%	56.09%

5. Gradient Boosting Tree

Gradient Boost uses the concept of gradient descent to minimize the error of the predictions. It creates the next learner based on examples whose errors were the greatest.

[Figure 5.5.1]



After running 6 combinations of variables in the default setting, we find that the variable selected from ***backward feature selection*** has the highest FDR for OOT. Then, we use the combination of 20 variables to tune parameters. There are 6 related hyper-parameters tuned for our Gradient Boosting Tree algorithm:

- learning_rate: The learning rate shrinks the contribution of each tree by learning_rate
- max_depth: The maximum depth of the tree
- min_samples_leaf: The minimum number of samples required to be at a leaf node
- min_samples_split*:* The minimum number of samples required to split an internal node
- subsample: The fraction of samples to be used for fitting the individual base learners
- n_estimators: The number of boosting stages to perform

According to the table below, we can find that the model with 50 trees, 10 splits, 50 leaf nodes, maximum depth of the tree as 5 and learning rate as 0.05 performs better without overfitting. ***The model can catch 57.88% fraud in the top 3% of our data.***

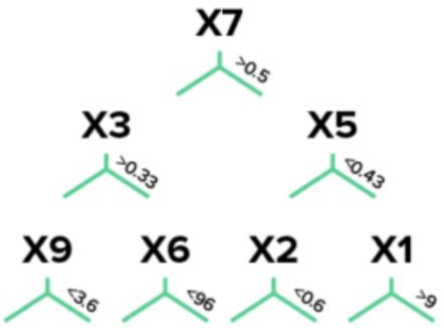
[Table 5.5.1]

Iteration	Parameter						Remark	Average FDR at 3%		
No.	# variable	learning_ rate	max_depth	min_ samples_ leaf	min_samples_split	n_ estimators	FS method	Train	Test	OOT
1 (default)	20	0.1	3	1	2	100	SelectKbest	80.20%	70.88%	52.18%
2 (default)	**20**	**0.1**	**3**	**1**	**2**	**100**	**Backward**	**87.52%**	**81.63%**	**55.53%**
3 (default)	20	0.1	3	1	2	100	Stepforward	83.12%	77.42%	45.98%
4 (default)	5	0.1	3	1	2	100	SelectKbest_Lasso	72.63%	64.74%	36.20%
5 (default)	5	0.1	3	1	2	100	Backward_Lasso	85.68%	76.53%	37.04%
6 (default)	1	0.1	3	1	2	100	Stepforward_Lasso	74.83%	62.02%	35.98%
7	20	0.1	3	1	2	1000	Backward	99.89%	87.19%	54.02%
8	20	0.1	3	1	2	300	Backward	95.15%	86.84%	53.52%
9	20	0.1	3	1	10	100	Backward	88.47%	83.22%	56.59%
10	20	0.01	5	1	2	100	Backward	76.70%	74.98%	52.63%
11	20	0.01	5	1	2	150	Backward	79.21%	77.18%	53.58%
12	20	0.1	5	50	30	100	Backward	91.80%	85.40%	55.03%
13	20	0.1	5	50	10	50	Backward	89.07%	83.55%	57.71%
14	**20**	**0.05**	**5**	**50**	**10**	**50**	**Backward**	**82.62%**	**78.81%**	**57.88%**
15	20	0.05	5	150	30	150	Backward	84.01%	78.17%	56.26%
16	20	0.05	5	60	10	80	Backward	85.44%	79.33%	57.99%
17	20	0.1	10	60	10	150	Backward	98.64%	88.35%	53.18%
18	20	0.1	10	60	10	50	Backward	92.44%	85.13%	52.18%
19	20	0.1	10	150	30	150	Backward	92.85%	86.85%	55.81%
20	20	0.15	10	50	10	100	Backward	98.12%	84.84%	45.81%

6. XGBoost

Extreme Gradient Boosting, as known as XGBoost, is a scalable, distributed gradient-boosted decision tree machine learning library, which provides parallel tree boosting.

[Figure 5.6.1]



After running 6 combinations of variables in the default setting, we find that the variable selected from ***backward feature selection*** has the highest FDR for OOT. Then, we use the combination of 20 variables to tune parameters. There are 4 related hyper-parameters tuned for our XGBoost algorithm:

- booster: Type of model to run at each iteration
- max_depth: Maximum depth of a tree
- min_child_weight: The minimum sum of weights of all observations required in a child
- gamma: Minimum loss reduction required to make a further partition on a leaf node of the tree

According to the table below, we can find that the model with booster as 'gbtree', maximum depth of the tree as 8, minimum sum of weights of nodes as 75 and gamma as 0.01 perform better without overfitting. ***The model can detect 56.37% fraud in the top 3% of our data.***

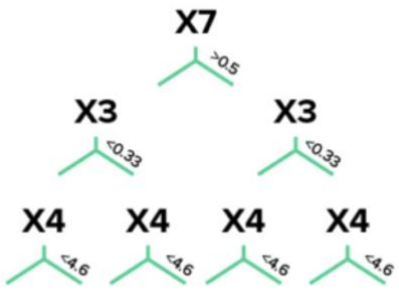
[Table 5.6.1]

Iteration	Parameter					Remark	Average FDR at 3%		
No.	# variable	booster	max_depth	min_child_weight	gamma	FS method	Train	Test	OOT
1 (default)	20	gbtree	6	1	0	SelectKbest	99.89%	75.71%	49.16%
2 (default)	7	gbtree	6	1	0	Stepforward	100.00%	95.67%	37.49%
2 (default)	**20**	**gbtree**	**6**	**1**	**0**	**Backward**	**100.00%**	**89.90%**	**50.95%**
3 (default)	20	gbtree	6	1	0	Stepforward	99.92%	83.28%	38.55%
4 (default)	5	gbtree	6	1	0	SelectKbest_Lasso	84.55%	65.63%	34.36%
5 (default)	5	gbtree	6	1	0	Backward_Lasso	99.02%	81.51%	40.17%
6 (default)	1	gbtree	6	1	0	Stepforward_Lasso	70.84%	62.52%	31.56%
7	20	dart	300	50	0.05	Backward	88.61%	83.82%	54.41%
8	20	gbtree	300	50	0.05	Backward	89.01%	84.64%	54.75%
9	20	gbtree	2	1	0	Backward	91.39%	86.63%	57.26%
10	20	gbtree	5	10	0.01	Backward	96.08%	88.19%	51.90%
11	20	gbtree	5	75	0.01	Backward	83.64%	80.59%	56.26%
12	**20**	**gbtree**	**8**	**75**	**0.01**	**Backward**	**83.54%**	**79.90%**	**56.37%**
13	20	gbtree	8	80	0.03	Backward	82.74%	79.97%	56.48%
14	20	gbtree	10	30	0.01	Backward	93.83%	86.91%	55.08%
15	20	gbtree	10	50	0.01	Backward	89.44%	83.13%	54.53%
16	20	gbtree	10	75	0.01	Backward	84.31%	78.60%	55.87%
17	20	gbtree	10	100	0.01	Backward	80.10%	77.43%	55.47%

7. CatBoost

CatBoost is an algorithm that uses gradient boosting on decision trees library with categorical features support. CatBoost distinguishes itself from LightGBM and XGBoost by building symmetric trees. In every step, leaves from the previous tree are split using the same condition. The feature-split pair that accounts for the lowest loss is selected and used for all the level's nodes.

[Figure 5.7.1]



After running 6 combinations of variables in the default setting, we find that the variable selected from ***select k best selection*** has the highest FDR for OOT. Then, we use the combination of 20 variables to tune parameters. There are 2 related hyper-parameters tuned for our CatBoost algorithm:

- iterations: The maximum number of trees that can be built when solving machine learning problems
- learning_rate: The learning rate used for training

According to the table below, we can find that the model with iterations as 1000 times and learning rate as 0.01 performs better without overfitting. ***The model can detect 56.31% fraud in the top 3% of our data.***

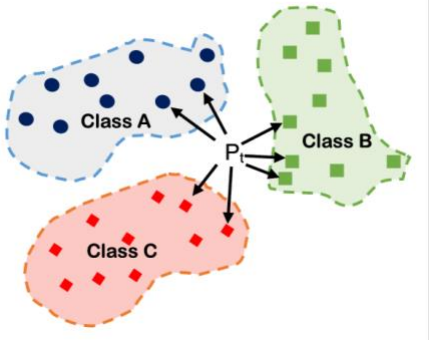
[Table 5.7.1]

Iteration	Parameter			Remark	Average FDR at 3%		
No.	# variable	iterations	learning_rate	FS method	Train	Test	OOT
1 (default)	**20**	**1000**	**0.03**	**SelectKbest**	**93.70%**	**77.53%**	**55.70%**
2 (default)	20	1000	0.03	Backward	97.55%	88.62%	54.25%
3 (default)	20	1000	0.03	Stepforward	95.27%	82.21%	46.87%
4 (default)	5	1000	0.03	SelectKbest_Lasso	79.13%	65.55%	32.79%
5 (default)	5	1000	0.03	Backward_Lasso	93.30%	79.72%	39.55%
6 (default)	1	1000	0.03	Stepforward_Lasso	63.65%	62.05%	33.74%
7	20	100	0.03	SelectKbest	91.83%	77.16%	50.06%
8	**20**	**1000**	**0.01**	**SelectKbest**	**82.75%**	**77.72%**	**56.31%**
9	20	1000	0.05	SelectKbest	92.55%	78.06%	56.03%
10	20	50	0.03	SelectKbest	87.82%	76.31%	53.63%
11	20	100	0.03	SelectKbest	91.58%	76.36%	51.84%

8. K-Nearest Neighbors (KNN)

K-Nearest Neighbors (KNN) is a type of supervised machine learning algorithm to estimate the likelihood that an observation will be a member of one group or another based on what group the data points nearest to it belong to. KNN performs a voting mechanism to determine the class of an unseen observation, which means that the class with the plurality vote will become the class of the data point.

[Figure 5.8.1]



After running 6 combinations of variables in the default setting, we find that the variable selected from ***select k best selection*** has the highest FDR for OOT. Then, we use the combination of 20 variables to tune parameters. There are 2 related hyper-parameters tuned for our K-Nearest Neighbors algorithm:

- n_neighbors: Number of neighbors to use
- leaf_size: Leaf size passed to BallTree or KDTree

According to the table below, we can find that the model with 30 neighbors and leaf size as 50 performs better without overfitting. ***The model can detect 56.2% fraud in the top 3% of our data.***

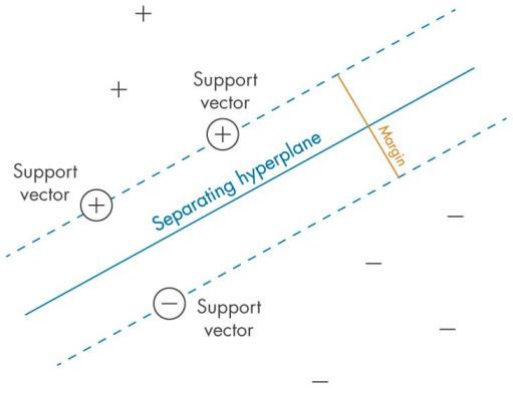
[Table 5.8.1]

Iteration	Parameter			Remark	Average FDR at 3%		
No.	# variable	n_neighbors	leaf_size	FS method	Train	Test	OOT
1 (default)	**20**	**5**	**30**	**SelectKbest**	**100.00%**	**73.44%**	**46.15%**
2 (default)	7	5	30	Stepforward	100.00%	86.62%	17.21%
2 (default)	20	5	30	Backward	100.00%	81.01%	42.63%
3 (default)	20	5	30	Stepforward	100.00%	73.55%	27.71%
4 (default)	5	5	30	SelectKbest_Lasso	95.70%	57.76%	32.40%
5 (default)	5	5	30	Backward_Lasso	100.00%	70.92%	36.76%
6 (default)	1	5	30	Stepforward_Lasso	90.76%	50.11%	25.25%
7	20	15	150	SelectKbest	85.57%	76.24%	48.94%
8	20	25	150	SelectKbest	82.55%	75.73%	54.86%
9	20	25	100	SelectKbest	83.44%	76.14%	57.21%
10	20	25	50	SelectKbest	83.32%	76.48%	57.21%
11	20	28	50	SelectKbest	81.95%	75.63%	56.98%
12	20	29	50	SelectKbest	82.27%	74.21%	55.98%
13	**20**	**30**	**50**	**SelectKbest**	**81.75%**	**76.30%**	**56.20%**
14	20	35	50	SelectKbest	80.98%	76.37%	55.31%
15	20	35	60	SelectKbest	81.31%	74.02%	53.41%

9. Support Vector Machine

Support Vector Machine is a supervised machine learning algorithm. In the SVM algorithm, we plot each data item as a point in n-dimensional space with the value of each feature being the value of a particular coordinate. Then, we perform classification by finding the hyperplane that differentiates the two classes with maximum margin.

[Figure 5.9.1]



After running 6 combinations of variables in the default setting, we find that the variable selected from ***backward feature selection*** has the highest FDR for OOT. Then, we use the combination of 20 variables to tune parameters. There are 3 related hyper-parameters tuned for our Support Vector Classifier:

- C: Regularization parameter
- gamma: Kernel coefficient for 'rbf', 'poly', and 'sigmoid'
- kernel: Specifies the kernel type to be used in the algorithm

According to the table below, we can find that the model with regularization parameter as 0.1, kernel as 'rbf' and kernel efficient as 'auto deprecated' perform better without overfitting. ***The model can detect 55.47% fraud in the top 3% of our data.***

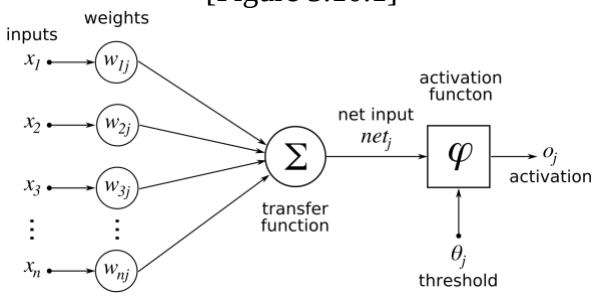
[Table 5.9.1]

Iteration	Parameter				Remark	Average FDR at 3%		
No.	# variable	C	gamma	kernel	FS method	Train	Test	OOT
1 (default)	20	1	auto_deprecated	rbf	SelectKbest	69.16%	65.39%	43.91%
2 (default)	**20**	**1**	**auto_deprecated**	**rbf**	**Backward**	**77.00%**	**74.02%**	**49.39%**
3 (default)	20	1	auto_deprecated	rbf	Stepforward	66.78%	64.47%	35.98%
4 (default)	5	1	auto_deprecated	rbf	SelectKbest_Lasso	42.58%	42.33%	20.06%
5 (default)	5	1	auto_deprecated	rbf	Backward_Lasso	57.33%	53.72%	27.77%
6 (default)	1	1	auto_deprecated	rbf	Stepforward_Lasso	37.78%	37.48%	19.16%
7	20	0.01	1	rbf	Backward	89.26%	79.60%	58.27%
8	**20**	**0.1**	**auto_deprecated**	**rbf**	**Backward**	**72.93%**	**70.33%**	**55.47%**
9	20	10	auto_deprecated	rbf	Backward	81.64%	77.74%	46.54%
10	20	1	auto_deprecated	linear	Backward	62.59%	61.36%	30.17%
11	20	1	auto_deprecated	poly	Backward	71.47%	68.11%	48.27%
12	20	1	auto_deprecated	sigmoid	Backward	15.23%	15.84%	10.95%
13	20	10	100	linear	Backward	63.02%	63.42%	29.66%
14	20	10	auto_deprecated	poly	Backward	73.16%	67.53%	45.70%

10. Neural Network (NN)

Neural network is a series of algorithms that endeavors to recognize underlying relationships in a set of data through a process that mimics the way the human brain operates. It is composed of an input layer, one or more hidden layers, and an output layer. Each node connects to another and has an associated weight and threshold. If the output of any individual node is above the specified threshold value, that node is activated, sending data to the next layer of the network.

[Figure 5.10.1]



After running 6 combinations of variables in the default setting, we find that the variable selected from ***select k best selection*** has the highest FDR for OOT. Then, we use the combination of 20 variables to tune parameters. There are 4 related hyper-parameters tuned for our NN Classifier:

- hidden_layer_sizes: ith element represents the number of neurons in the ith hidden layer
- activation*:* Activation function for the hidden layer
- alpha: L2 penalty (regularization term) parameter
- learning_rate*:* Learning rate schedule for weight updates

According to the table below, we can find that the model with 60 layers, activation function as 'relu', penalty parameter as 0.01 and adaptive learning rate perform better without overfitting. ***The model can detect 60.28% fraud in the top 3% of our data.***

[Table 5.10.1]

Iteration	Parameter					Remark	Average FDR at 3%		
No.	# variable	hidden_layer_sizes	activation	alpha	learning_rate	FS method	Train	Test	OOT
1 (default)	**20**	**100**	**relu**	**0.0001**	**constant**	**SelectKbest**	**81.72%**	**76.28%**	**56.31%**
2 (default)	20	100	relu	0.0001	constant	Backward	87.80%	80.55%	55.25%
3 (default)	20	100	relu	0.0001	constant	Stepforward	80.77%	75.58%	48.88%
4 (default)	5	100	relu	0.0001	constant	SelectKbest_Lasso	63.71%	64.55%	36.82%
5 (default)	5	100	relu	0.0001	constant	Backward_Lasso	69.82%	70.93%	46.82%
6 (default)	1	100	relu	0.0001	constant	Stepforward_Lasso	62.88%	63.14%	36.31%
7	20	100	relu	0.001	constant	SelectKbest	81.42%	77.20%	56.54%
8	20	110	relu	0.0001	constant	SelectKbest	82.60%	76.17%	56.03%
9	20	110,1	logistic	0.0001	constant	SelectKbest	77.92%	76.14%	59.44%
10	20	100,1	logistic	0.0001	constant	SelectKbest	77.70%	74.77%	59.05%
11	20	10	relu	0.0001	constant	SelectKbest	73.97%	70.73%	53.52%
12	20	50	relu	0.0001	constant	SelectKbest	78.02%	75.27%	58.21%
13	20	50	relu	0.0001	adaptive	SelectKbest	79.40%	75.65%	58.88%
14	20	50	relu	0.01	constant	SelectKbest	77.01%	72.26%	59.44%
15	20	50	relu	0.01	adaptive	SelectKbest	76.29%	73.98%	60.11%
16	20	50	relu	0.05	adaptive	SelectKbest	74.40%	74.73%	56.20%
17	20	60	relu	0.01	constant	SelectKbest	75.73%	74.43%	58.77%
18	**20**	**60**	**relu**	**0.01**	**adaptive**	**SelectKbest**	**76.23%**	**73.98%**	**60.28%**
19	20	70	relu	0.0001	constant	SelectKbest	80.09%	74.86%	58.32%
20	20	70	relu	0.0001	adaptive	SelectKbest	80.29%	76.94%	57.77%
21	20	80	relu	0.0001	constant	SelectKbest	81.60%	74.61%	56.42%

VI. Results

1. Model Results

Table 6.1.1 shows the mean FDR score at a 3% cutoff for each model we run above. We select our final model based on the FDR performance on OOT data and the Neural Network model with variables from select K best method (Table 6.1.2) has the best performance of 60.28% FDR score at 3% in out-of-time data among all models.

[Table 6.1.1] Mean/Standard Deviation of FDR Score

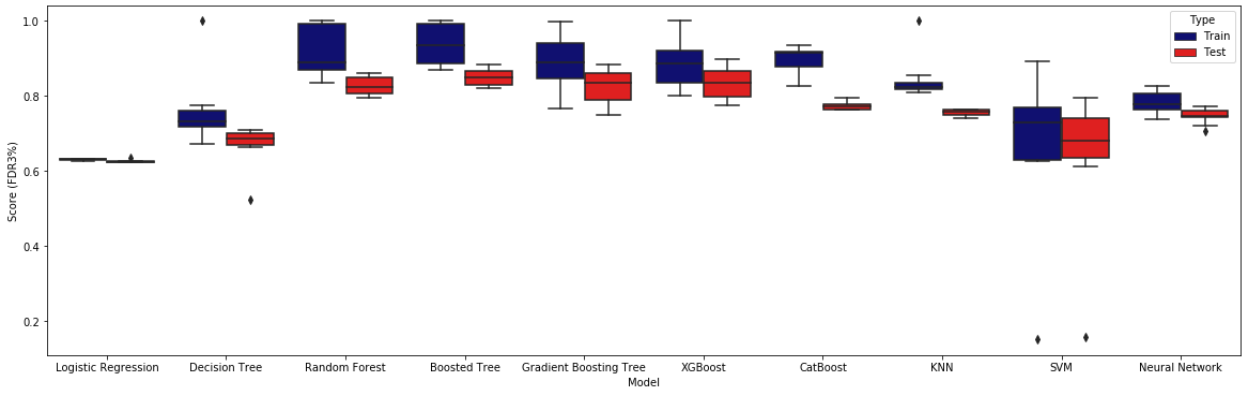
Model	# of Variables	Training Score		Testing Score		OOT Score	
		Mean	STD	Mean	STD	Mean	STD
Logistic Regression	1	63.10%	0.20%	62.65%	0.54%	36.31%	0.00%
Decision Tree	20	75.69%	9.02%	67.29%	5.43%	41.74%	5.31%
Random Forest	20	91.65%	6.84%	82.68%	2.55%	57.83%	1.87%
Boosted Tree	20	94.08%	5.47%	85.01%	2.23%	54.37%	3.43%
Gradient Boosting Tree	20	89.46%	7.03%	82.77%	4.17%	54.51%	3.08%
XG Boost	20	88.56%	6.01%	83.31%	4.05%	54.94%	1.87%
CatBoost	20	89.51%	4.33%	77.40%	1.29%	53.55%	2.66%
KNN	20	84.31%	5.67%	75.59%	0.93%	54.23%	3.74%
SVM	20	67.37%	21.26%	64.22%	19.13%	41.60%	15.12%
Neural Network	**20**	**78.40%**	**2.71%**	**74.87%**	**1.67%**	**57.83%**	**1.84%**

[Table 6.1.2] Final Variables With Their Univariate KS Score

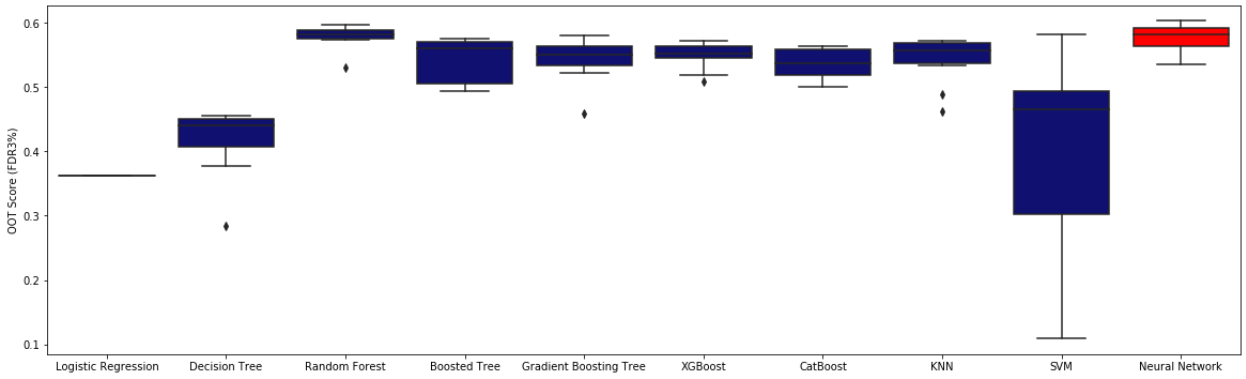
Order	Variable Name	KS Score	Order	Variable Name	KS Score
1	card_zip_total_7	0.684729	11	card_state_total_1	0.659137
2	card_merch_total_7	0.681017	12	card_merch_total_30	0.658264
3	card_zip_total_3	0.677675	13	card_merch_total_1	0.658215
4	card_merch_total_3	0.675054	14	card_zip_total_30	0.656706
5	card_merch_total_14	0.674867	15	merch_state_total_1	0.605706
6	card_state_total_3	0.674178	16	Cardnum_total_3	0.601978
7	card_zip_total_14	0.672277	17	Cardnum_total_7	0.600132
8	card_state_total_7	0.669661	18	Cardnum_total_1	0.576903
9	card_state_total_14	0.668943	19	Cardnum_total_0	0.570686
10	card_zip_total_1	0.660589	20	Cardnum_total_14	0.547659

Figure 6.1.1 and 6.1.2 shows the boxplots of FDR score at a 3% cutoff in our training/ testing data, and out-of-time data.

[Figure 6.1.1] Boxplots of FDR Score at a 3% Cutoff in **Training/Testing Data**
Neural Network has the lowest disparity between training and testing data, which means it might not have



[Figure 6.1.2] Boxplots of FDR Score at a 3% Cutoff in **Out-of-Time Data**
Neural Network has the highest FDR score and low standard deviation on OOT data



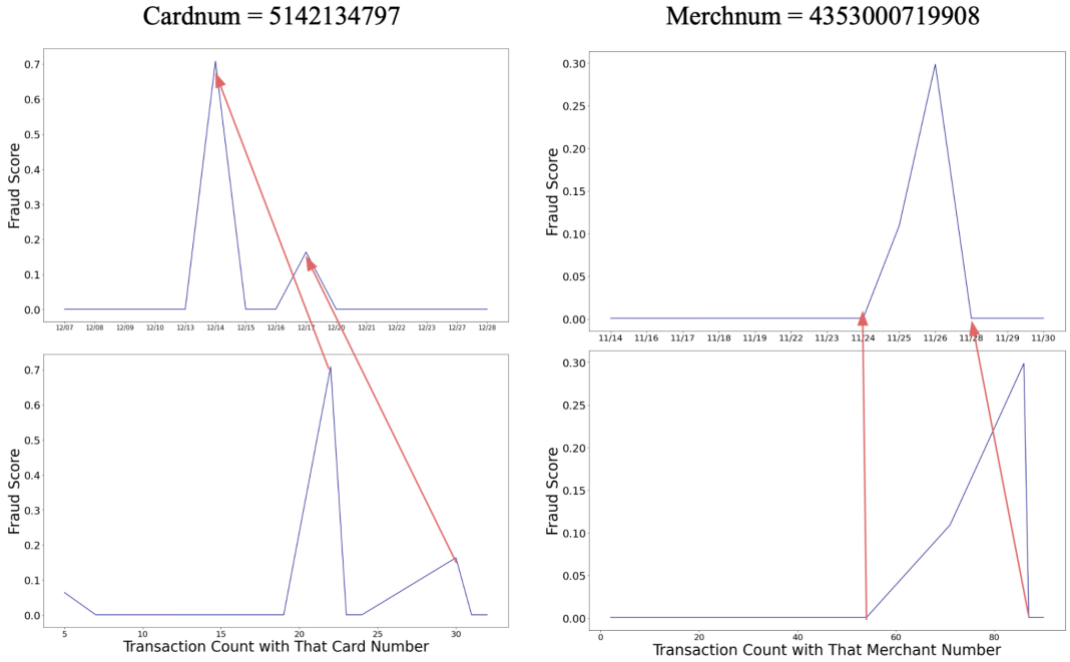
2. Time dependence of fraud score

After selecting the Neural Network as our final model, we take one card number and one merch number as examples to demonstrate the time dependence of the fraud score.

From the card number example, there are 30 transactions that happen in this month, 5 of them happen on December 13&14 and 6 of them happen on December 16, so the fraud score grows rapidly when transactions occur across these three days.

From the merch number example, there are 90 transactions that happen in this month, 17 of them happen on November 25 and 15 of them happen on 26. As the count increases, the fraud score rises. In addition, we notice that November 25 and 26 in 2006 is Saturday and Sunday, which corresponds to our weekday risk table results that Saturday and Sunday have a higher likelihood of fraud.

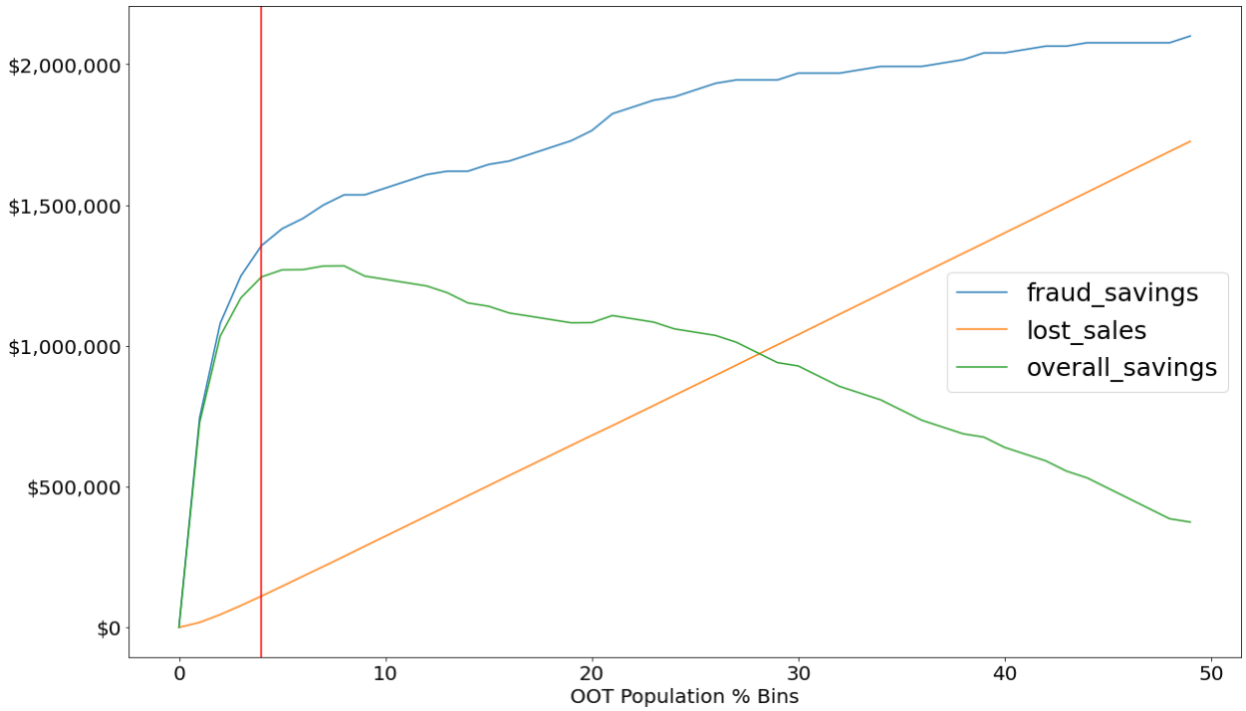
[Figure 6.2.1] Time dependence of fraud score



3. Suggested score cutoff

In order to understand the financial impact of our model, we examine the fraud saving at different population cutoffs. The highest overall saving is $1,284,000 at 8% population cutoff. However, we would like to choose the smallest cutoff that is still close to the highest savings, so we recommend a cutoff at 4% and the expected annual savings is $1,244,700.

[Figure 6.3.1] Fraud savings at OOT population % bins



[Table 6.3.1] Time dependence of fraud score

OOT population bins	Fraud Saving	Lost Sales	Overall Savings
1%	744,000	17,700	726,300
2%	1,080,000	45,600	1,034,400
3%	1,248,000	77,700	1,170,300
4%	**1,356,000**	**111,300**	**1,244,700**
5%	1,416,000	146,100	1,269,900
6%	1,452,000	181,500	1,270,500
7%	1,500,000	216,600	1,283,400
8%	1,536,000	252,000	1,284,000
9%	1,536,000	288,300	1,247,700

VII. Conclusion

For this project, we build supervised machine learning models to detect fraud for the credit card transaction data. For the first step, we make a data quality report (DQR) for exploring and visualizing the dataset. Then, we identify bad records and fill in missing values (Merchnum, State, and ZIP). In the variable creation stage, we first build more identity entities, then use those entities to create frequency variables such as day since, frequency, and relative velocity, and also create benford's law variables and the likelihood of fraud for the day of the week and state. The total number of variables we create is 522.

In order to reduce dimensionality and select the most valuable subset, we first try the feature selection process that the professor suggested. We apply a filter method by calculating KS and run a wrapper to get down to 20 variables. Then, we apply different wrapper methods (select K best and backward selection) to select 2 more variable sets. Before putting the variable sets into our model, we also run ridge and lasso regression to see if the coefficients could be further standardized to build a better model. Therefore, we have 6 variable sets in total.

During the modeling process, we first input 6 variable sets to 10 default models and see which one performs better and then choose the best one to keep tuning the hyperparameter to get our final model. The neural network model outperforms other models with the variables from select K best method. It has performance of 60.28% at 3% FDR in out-of-time data. As a result, the model can eliminate 60.28% of the fraud by declining 3% of the transactions.

Our final model can be used in credit card fraud detection systems to improve the accuracy of fraud detection. Compared to traditional rule-based systems, machine learning-based systems allow for creating algorithms that process large datasets with many variables and help find the hidden correlations between user behavior and the likelihood of fraudulent actions. For example, when detecting unusual user behavior during transactions, the algorithm will consider this activity suspicious and assign a higher level of fraud likelihood. Then, the system can send a verification request to a card owner to check this transaction and prevent fraudulent actions.

For future work, we may consult with domain experts to create more effective variables and perform insightful feature engineering. Second, since the fraud label in this dataset is imbalanced, we may try an oversampling method such as SMOTE to create new and artificial records, then our models might perform better. Third, in the feature selection process, we can use different algorithms in the wrapper or try other wrappers such as exhaustive feature selection. In addition, we may also do unsupervised learning because prior labeling heavily relies on human experience and knowledge of fraud, reducing the possibility of detecting new unknown fraud and abuses. By conducting unsupervised learning to recognize fraudulent patterns based on exploration and grouping, we can better distinguish the fraud pattern even if it has been modified or is totally new.

Appendix

Data Quality Report on Credit Card Transaction Data

Field Summary Table

1. Numerical Fields

Field Name	% Population	Min	Max	Mean	Stdev	% Zero
date	100%	2006-01-01	2016-12-31	N/A	N/A	0%
Amount	100%	0.01	3,102,045.53	427.89	10,006.14	0%

2. Categorical Fields

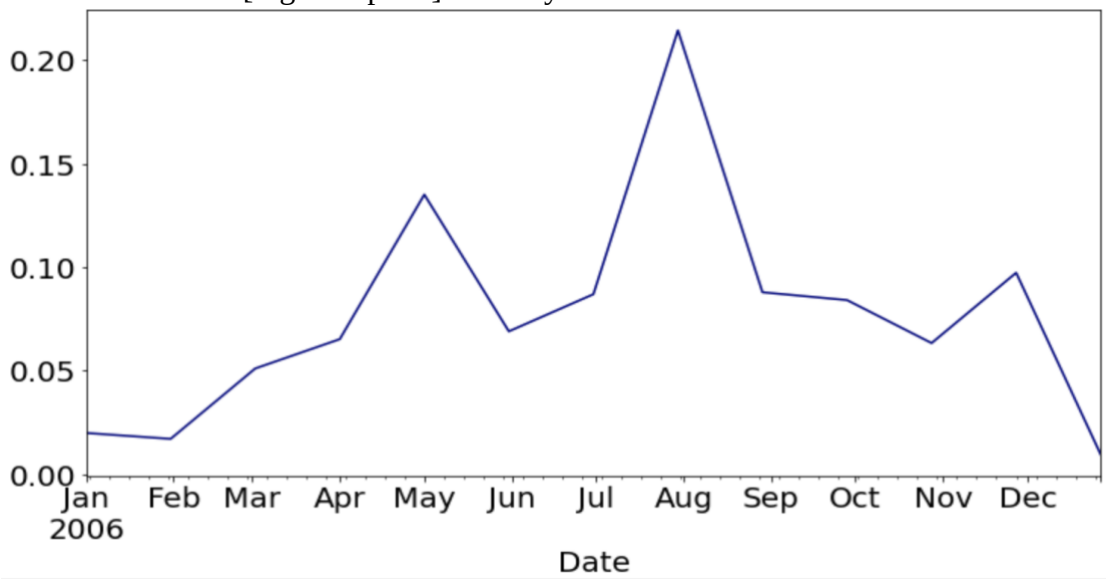
Field Name	% Population	# Unique Values	Most Common Value
Recnum	100%	96,753	N/A
Cardnum	100%	1,645	5142148452
Merchnum	96.51%	13,091	930090121224
Merch description	100%	13,126	GSA-FSS-ADV
Merch state	98.76%	227	TN
Merch zip	95.19%	4,567	38118
Transtype	100%	4	P
Fraud	100%	2	0

Field Description and Distribution

Below are the fields that we find some insights and would like to elaborate in more detail.

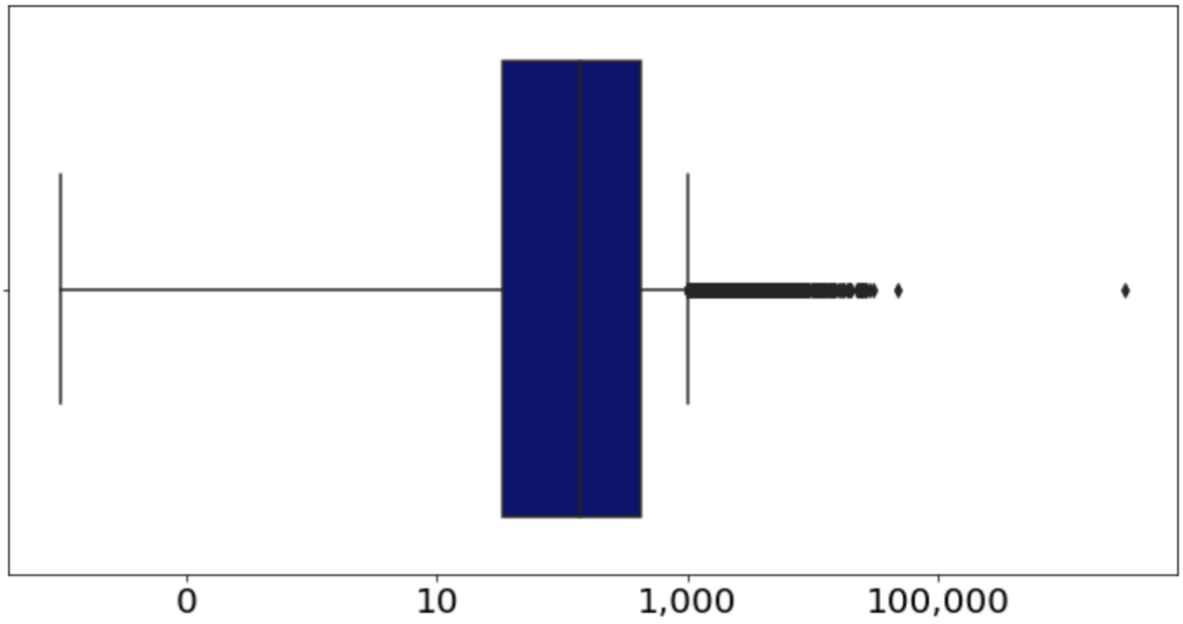
1. Date: It is a numerical field representing the transaction date, with 100% populated. The date ranges from 2006-01-01 to 2006-12-31.The plots below show the fraud rate on a monthly basis. We can tell there are spikes in specific time periods.

[Figure Ap.1.1] Monthly Fraud Occurrence Rate



2. Amount: It is a numeric field representing the transaction amount, with 100% populated and 0% zero. The min and max of the field are 0.01 and 3,102,045.53. From the boxplot below we can tell that there are extreme outliers, which we will deal with in the data cleaning process.

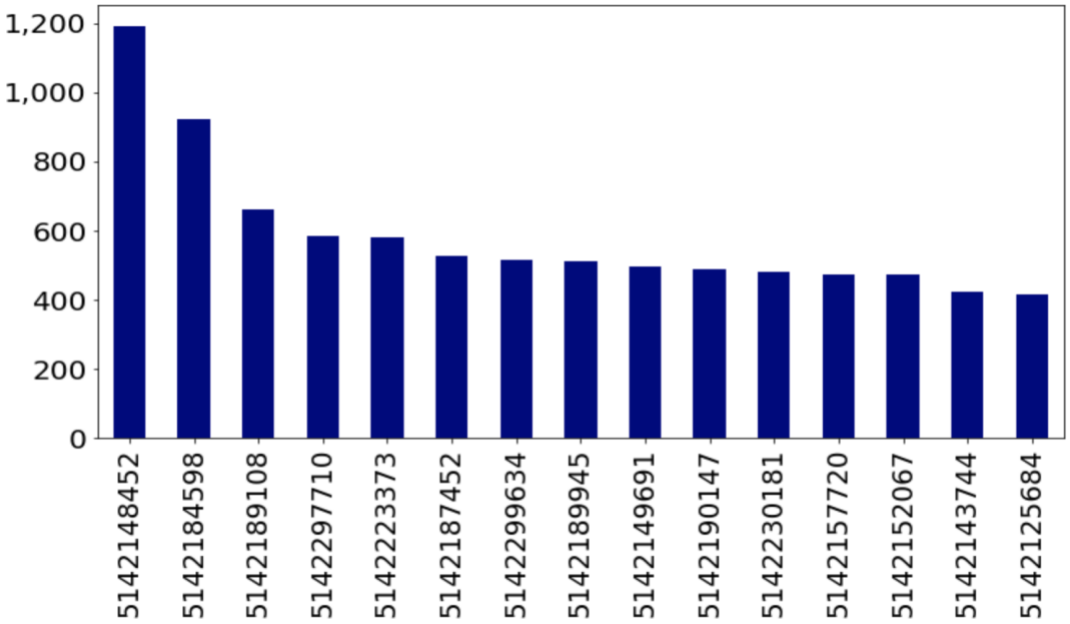
[Figure Ap.1.2] The box plot of Amount



3. Recnum: Recnum is a categorical field and covers 100% of records. There are 96,753 unique values.

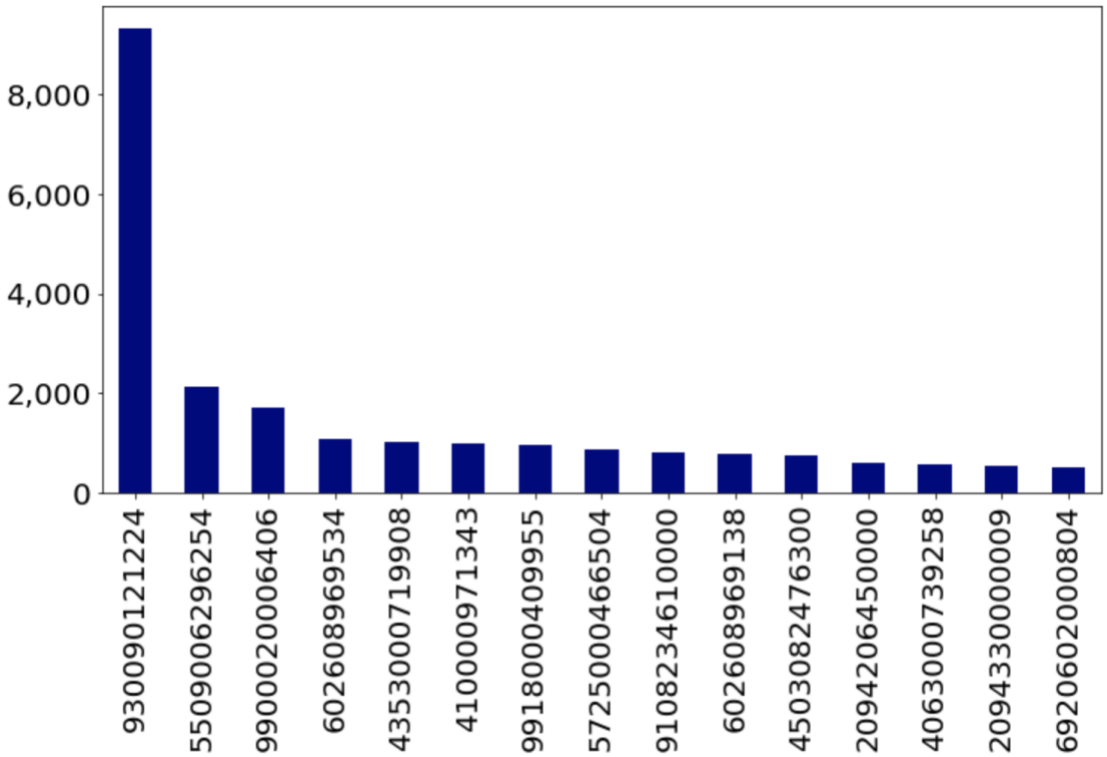
4. Cardnum: Cardnum is a categorical field and covers 100% of records. There are 1,645 unique values, and the most common value is 5142148452.

[Figure Ap.1.3] The distribution of Cardnum



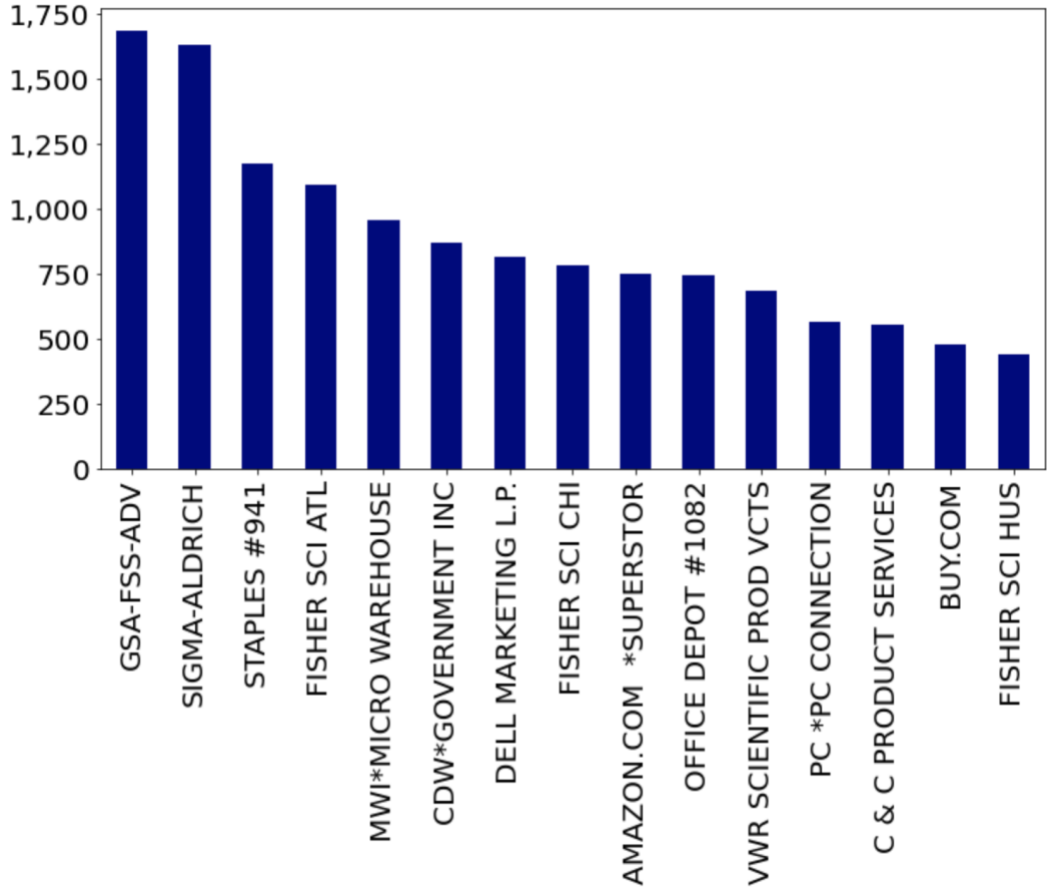
5. Merchnum: Merchnum is a categorical field and covers 96.51% of records. There are 13,091 unique values, and the most common value is 930090121224.

[Figure Ap.1.4] The distribution of Merchant's number



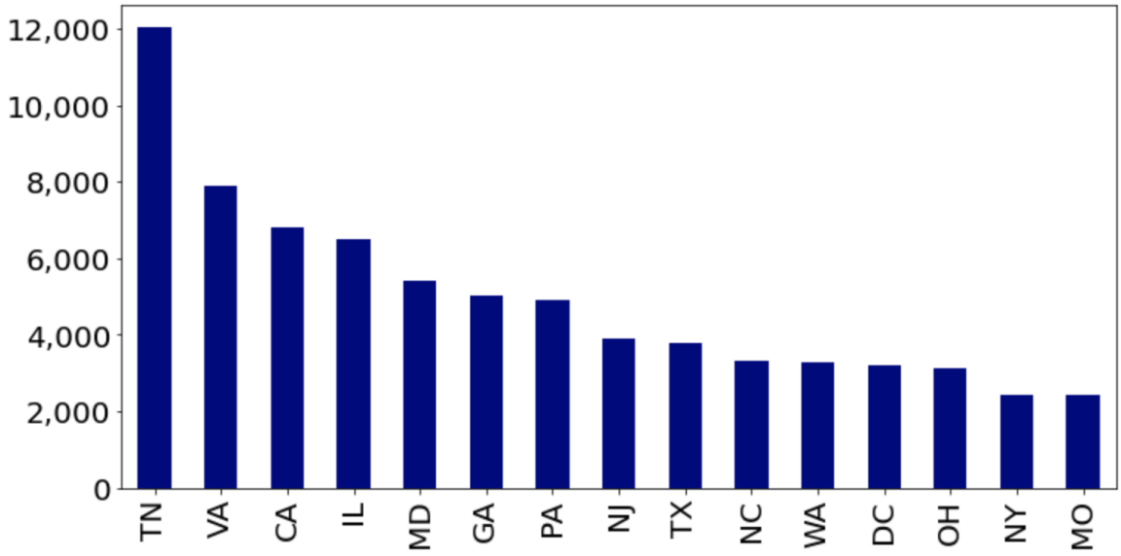
6. Merch description**:** Merch description is a categorical field and covers 100% of records. There are 13,126 unique values, and the most common value is GSA-FSS-ADV.

[Figure Ap.1.5] The distribution of Merch description



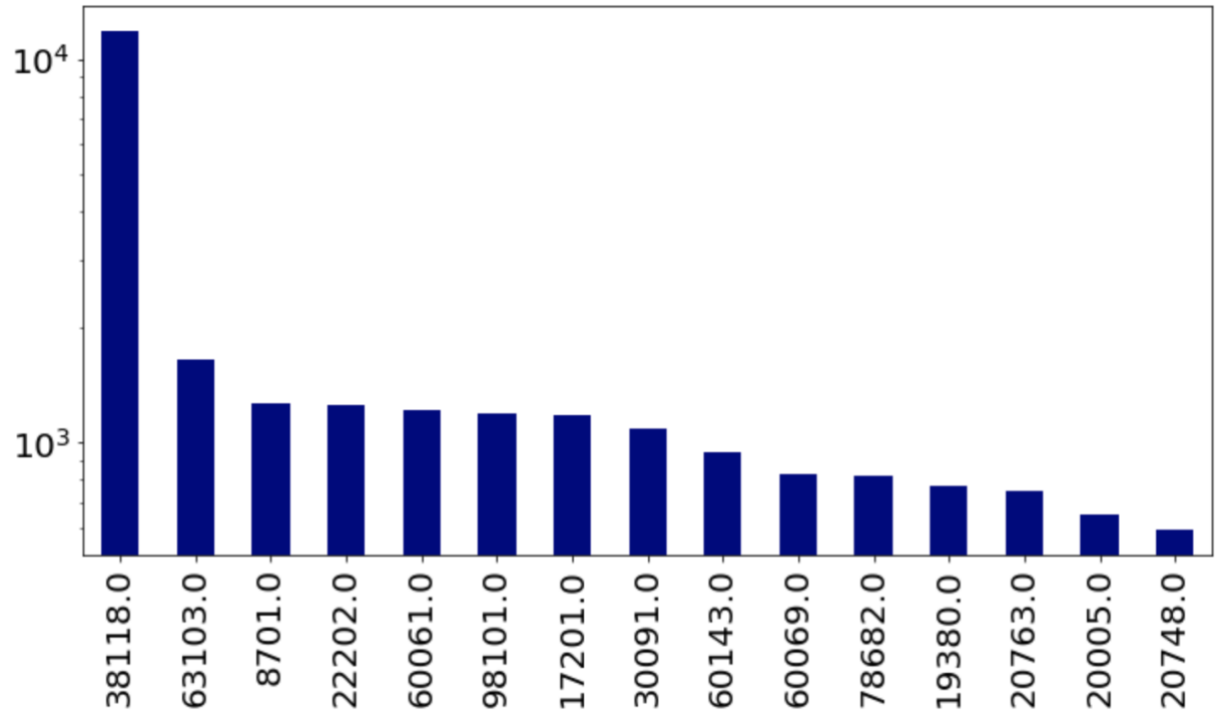
7. Merch state: It is a categorical field representing the merchant's state of the transaction, with 98.76% populated and 227 unique values. The most common value is TN, and the distribution of the top 15 categories of the field is as below.

[FigureAp.1.6] The distribution of Merchant's state



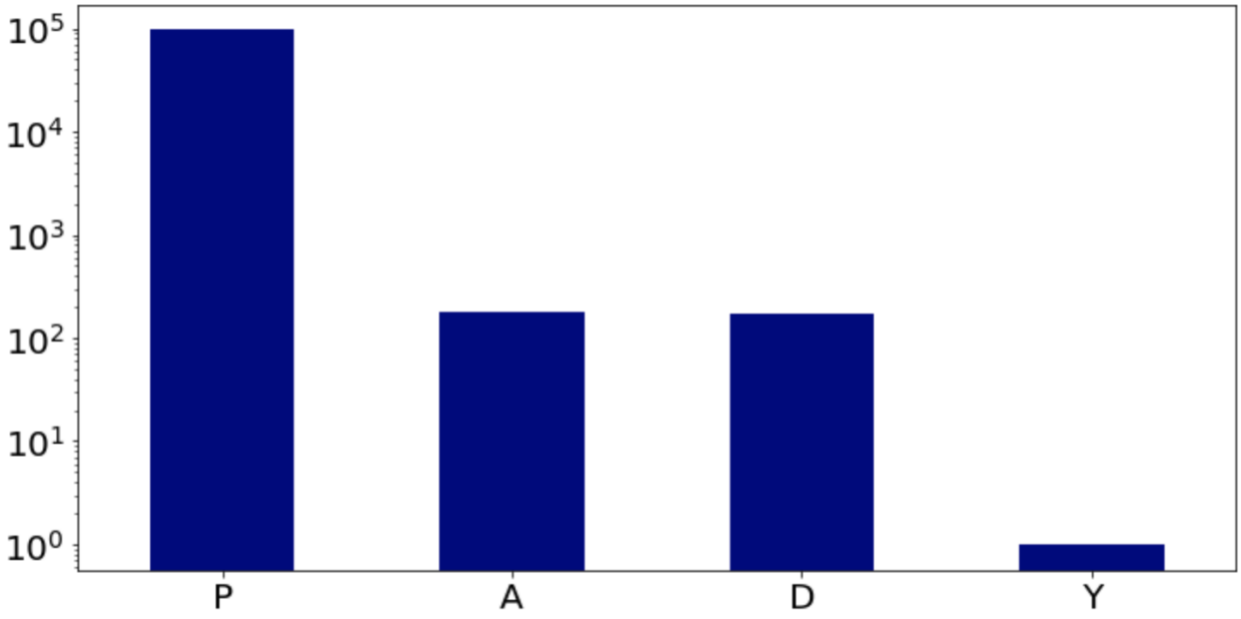
8. Merch zip: Merch zip is a numerical field and covers 95.19% of records. There are 4,567 unique values, and the most common value is 38118.

[Figure Ap.1.7] The distribution of Merchant's zip



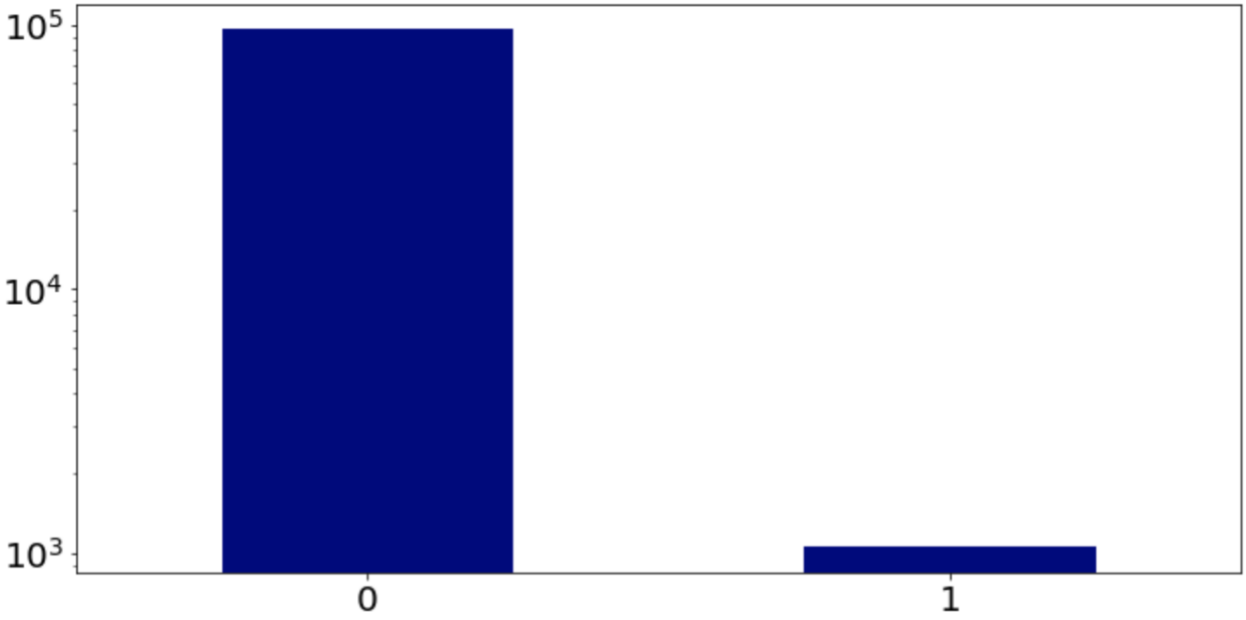
9. Transtype: Transtype is a categorical field and covers 100% of records. There are 4 unique values, and the most common value is P.

[Figure Ap.1.8] The distribution of transtype



10. Fraud: It is a categorical field representing whether the transaction is classified as a fraud, with 100% populated and 2 unique values (1=fraud and 0=non-fraud). About 1.1% of the transactions are classified as fraud.

[Figure Ap.1.9] The distribution of whether the transaction is a fraud



Reference

[1] Federal Trade Commission, 2020, CONSUMER SENTINEL NETWORK DATA BOOK 2019

[2] Abhishek Joshi, Vaibhav Jain, Shreedhar Soni, 2021, An Experimental Study using Unsupervised Machine Learning Techniques for Credit Card Fraud Detection