DATA3001 Group Report

Yuumi Insurance Fraud Detection

11th August 2019

University of New South Wales

Dean Hou

Neel Iyer

Serena Xu

Introduction

Insurance fraud is an issue with far reaching consequences not only in the insurance industry, but also for the government, corporate sectors and for ordinary consumers. Insurance fraud is estimated to cost Australian insurance companies $2.2 billion every year[15]. Using traditional methods of fraud detection is time consuming, costly and inaccurate, with current systems relying on basic computer algorithms to identify potentially fraudulent claims, which are then investigated by employees who cross check the customer's historical asset information. This is especially costly for larger insurance companies, with statistics in 2014-15 showing that insurers approving 3,361,016 claims from policyholders, and 122,875 claims denied[4].

The goal of this project is to detect fraudulent customers by using log data. We specifically want to see what types of transaction information of a given customer contributes to a customer being fraudulent and if we can build a classification model around this information to detect fraudulent customers given log data.

By allowing an insurance company to detect fraudulent customers earlier on in the fraud detection process using log data, we reduce the required resources and costs of having to later investigate and cross-check claims, as well as reducing the amount of claims paid to fraudulent customers that are detected after a clam has already been paid.

In this project, we were given log data for a hypothetical insurance company called Yuumi insurance. Yuumi insurance covers house and personal possessions against loss or damage caused by the following insured events: storm, fire, lightning, theft and escape of liquid. The insurance policy is priced according to the risk of the customer based off an array of information, done by actuaries within the "company". Irrespective of the price of the insurance all policy has a coverage of at most $25,000.

In this report, we will cover existing research on methods used to detect fraud in financial settings and ways that we could measure the accuracy of our fraud classification models in an Expanded Literature Review. Following the literature review we will go into detail on how we cleaned and transformed the log data into a more readable and usable format in preparation for the modelling process in the Material and methods section. In this section we will also cover results found in the data exploration step, as well as how we dealt the imbalance in the data using under and over sampling. The types of models used in this project will also be discussed in the Material and Methods section. The final section of the report before the conclusion is the Analyses and results section, which will cover the results from the models tested and what features in the data explained why some customers were fraudulent and others were not in the best model. A conclusion to the report will follow along with references to papers and statistics used in the report and an appendix of all our code used in the project.

Expanded Literature Review

Insurance fraud is common, and can occur at almost any stage during the insurance process, such as during application, eligibility, rating, billing and claims processes. The fraud can be committed by a wide variety of people during the various stages, such as customers, agents, company employees, healthcare providers and others **(Ngai, Hu, Wong, Chen, Sun)**[13].

Insurance fraud often occurs in three forms:

- exaggerating or padding an otherwise legitimate claim
- deliberate fabrication or staging of an incident to get a claim
- giving deliberately misleading information in support of a claim or misrepresenting fact material in a claim

Fraud detection is a widely researched area amongst many industries, including credit card fraud, money laundering, telecommunications fraud, computer intrusion, medical and scientific fraud **(Bolton, Hand)**[3], epidemic detection, spam detection and terrorist detection **(Phua, Lee, Smith, Gayler)**[14]. There currently exist many real-time fraud detection services such as Microsoft Azure Stream Analytics, and SAS solutions. However they are paid services and some companies would prefer to use internally developed services for fraud detection to reduce costs. The various types of fraudsters and the affected industries are highlighted in the images below **(Phua, Lee, Smith, Gayler)**[14].

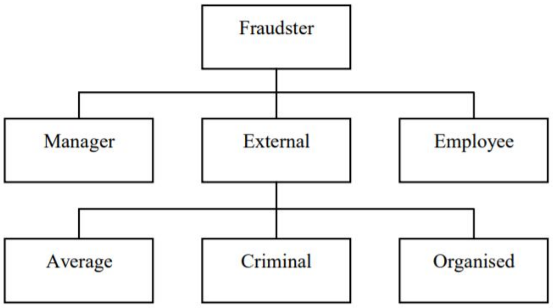


Figure 2.1: Hierarchy chart of white-collar crime perpetrators from both firm-level and community-level perspectives.

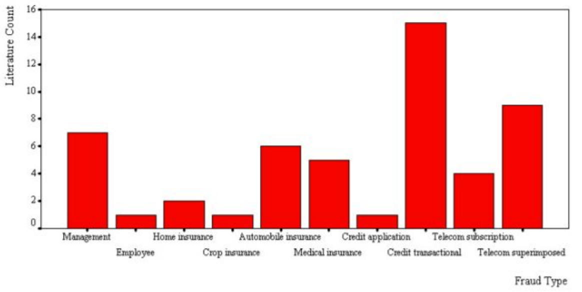


Figure 2.2: Bar chart of fraud types from 51 unique and published fraud detection papers. The most recent publication is used to represent previous similar publications by the same author(s).

Many machine learning techniques are useful for fraud and anomaly detection. Prior to building models, raw data should be cleaned and transformed using techniques such as feature engineering, feature selection and dimensionality reduction using PCA. This allows features found in the raw data that better represent the underlying problem to be fed to the predictive model, and usually results in improved model performance on unseen data. **(Guha, Manjunath, Palepu)**[8].

After the data has been cleaned and transformed, model building and testing occurs. The process of model building involves feeding the data to an initial model, measuring it's performance, then re-engineer necessary features and re-test. Once all the changes and updates are made, the data is re-fed into the model and run. This model is improved by altering parameters, which are tweaked and tested to generate the best model **(Guha, Manjunath, Palepu)**[8].

Some commonly used machne learning models for fraud detection are logistic regression, multivariate gaussian (MVG), boosting and random forest **(Guha, Manjunath, Palepu)**[8], naive Bayes, neural networks, support vector machines **(Phua, Lee, Smith, Gayler)**[14], linear discriminant analysis, logistic discrimination, neural networks, BAYES **(Clark and Niblett, 1989)**, FOIL **(Quinlan, 1990)** and RIPPER **(Cohen, 1995)**, CART **(Breiman, Friedman, Olshen and Stone, 1984)**, C4.5 **(Quinlan, 1993) (Bolton, Hand)**[3], decision trees, K-nearest neighbor, Bayesian belief network, fuzzy logic, and self-organizing map **(Ngai, Hu, Wong, Chen, Sun)**[13]. Some other fraud detection techniques include graph-theoretic anomaly detection, and Inductive Logic Programming. These methods outlined fall under supervised, semi-supervised, unsupervised and hybrid approaches with labelled (structured) data **(Phua, Lee, Smith, Gayler)**[14]. An emphasis on non complex algorithms such as naive Bayes and logistic regression seems to support the fact that they produce better results than non-linear supervised algorithms such as neural networks and support vector machines **(Phua, Lee, Smith, Gayler)**[14].

Model performance is measured using various criteria, such as an error/confusion matrix, precision and recall, ROC and AUC curves **(Guha, Manjunath, Palepu)**[8], the Receiver Operating Characteristic (ROC) analysis (true positive rate versus false positive rate), the Area under the Receiver Operating Curve (AUC) and minimisation of cross entropy (CXE), which measures how close predicted scores are to target scores, the minimization of mean squared error of predictions, the Activity Monitoring Operating Characteristic (AMOC - the average score versus false alarm rate), entropy, information gain and cost **(Phua, Lee, Smith, Gayler)**[14]. Most fraud departments placed monetary value on predictions to maximise savings/profit, and misclassification costs (false positive and false negative error costs) are unequal, uncertain, differs from example to example, and changes over time. In fraud detection, a false negative error is usually more costly that a false positive error **(Phua, Lee, Smith, Gayler)**[14]. (We believe this is a short-term result, as false positive leads to bad customer reviews and a bad reputation for the company, which ultimately results in long term loss).

Model comparison methods include using a contingency matrix which compares various scores such as the number of true/false positives/negatives, recall, precision, F score and the fraud incident rate **(Guha, Manjunath, Palepu)**[8].

Material and methods

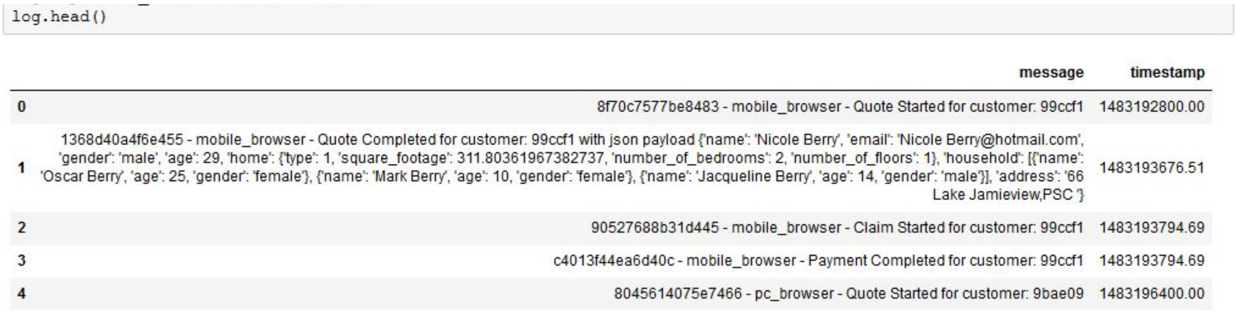
The Data

The data we were given contained two columns: ‘messages’ and ‘timestamp’ with ‘messages’ values being a string type, and ‘timestamp’ values being a float64 type. The initial data contained 3301833 rows.

Each value in the messages column was uniquely contained a string of information regarding the action and inputs the customer made for that transaction, as well as actions made by the system to record what responses the customers got from the system. Each message value contained a unique transaction id, the device they made the transaction with, the action made by the customer, and a customer id.

For different types of actions, additional information would be shown in the message. For example when the action was “Quote Completed for customer ”, a JSON payload which contained information about the occupants in the customers household (occupant names, age, gender, property address), details about the customer (name, email, age, gender), and details about the property (size, type, number of floors, number of bedrooms) that the customer must have inputted to complete the quote was appended to the message. Other actions include ‘Claim Started for customer’, ‘Claim Accepted for customer’ which included the amount paid, ‘Quote Started for customer’, ‘Payment Completed for customer’, ‘Policy Cancelled for customer’, ‘Quote Incomplete for customer’, ‘Claim Denied for customer’ of which the reason was always due to fraud.

Each message also had a timestamp which was a float 64 type value, and the given data was ordered by the timestamp in ascending order. We realised that the timestamp was in UNIX epoch time which meant that the timestamp counted the number of seconds from 00:00:00 UTC on 1 January 1970. Overall the timespan of the data was about 2 years (12/31/2016-02/12/2019 when converted to UTC).



```
log.head()
```

	message	timestamp
0	8f70c7577be8483 - mobile_browser - Quote Started for customer: 99ccf1	1483192800.00
1	1368d40a4f6e455 - mobile_browser - Quote Completed for customer: 99ccf1 with json payload {'name': 'Nicole Berry', 'email': 'Nicole Berry@hotmail.com', 'gender': 'male', 'age': 29, 'home': {'type': 1, 'square_footage': 311.80361967382737, 'number_of_bedrooms': 2, 'number_of_floors': 1}, 'household': [{'name': 'Oscar Berry', 'age': 25, 'gender': 'female'}, {'name': 'Mark Berry', 'age': 10, 'gender': 'female'}, {'name': 'Jacqueline Berry', 'age': 14, 'gender': 'male'}], 'address': '66 Lake Jamieview,PSC '}	1483193676.51
2	90527688b31d445 - mobile_browser - Claim Started for customer: 99ccf1	1483193794.69
3	c4013f44ea6d40c - mobile_browser - Payment Completed for customer: 99ccf1	1483193794.69
4	8045614075e7466 - pc_browser - Quote Started for customer: 9bae09	1483196400.00

Cleaning and Feature Engineering

In order to properly build models to predict fraud the given data needed to be cleaned appropriately first. We used Python in Jupyter Notebook to clean the data and create features from the data that we thought would be useful for the modelling process.

From the ‘messages’ column, we split the message string into distinct categories, transaction id, device, action, customer id and the remaining message using. After this, a unique id was given to each customer based on the ‘Quote Started for customer’ action and change in customer id. This served as the primary key for each customer in the dataset. Originally we had intended to use only customer ids to identify each customer, but sometimes customer ids were reused. We decided to use ‘Quote Started for customer’ action as a significant indicator for a unique customer as we believed that each new customer had to start their application for insurance process by starting a quote. In total there were 632998 unique customers found in the data.

We noticed that for some customers, there was a discrepancy between gender and their first name. A number of customers had a feminine given name but had listed their gender as male and vice versa. To create a feature in the dataset that highlighted this difference, the names for both holders and occupants of the household were split into first name and last name, using a nameparser package in Python and then, using the gender_predictor package, we predicted the gender of a customer based on their first name and found the probability of this predicted gender.

From this predicted gender probability, a column called error in gender was added to the dataframe by taking the absolute difference between predicted gender and actual gender given as input. This was done for each holder of an insurance policy and the occupants in their household and then sum. The sum was then divided by the number of people in the household (including the holder) to find the percentage error in then inputted gender. This predictor formed an influential part of our analysis.

We split up the JSON payload to create features for the details of the household, such as household type, size, number of bedrooms, number of floors. This was achieved by using python’s regular expression package. We also considered using functions native to the JSON library in Python, but these functions took too long to operate on this dataset.

In addition, columns for payment amount was added to the dataset, which detailed the amount paid from the insurance company to the customer for a specified claim.

The time difference between different actions such as quote completed, quote started and payment completed was also found, along with the total, mean, minimum and maximum transaction time difference for each customer. Time difference between Quote started and completed/incomplete and payment and claim started was also found for each customer. We believed that customers whose time difference between actions could potentially be fraudulent.

To determine whether a customer was fraudulent, a boolean column called ‘fraud’ was added to the dataframe based on the message that contained Claim denied due to fraud, with 1 for a fraudulent customer and 0 for a non-fraudulent customer.

After the cleaning and feature engineering processes the new dataset contained 632998 rows and 31 columns. The final columns in the dataset were:
new_id, fraud, device, no_actions, min_trans_diff, max_trans_diff, quote_time_diff, claim_pay_diff, mean_trans_time, total_trans_time, paid_min, paid_max, paid_mean, paid_diff, occupant_count, holder_gender, holder_age, holder_error_in_gender, occupant_error_in_gender, min_occupant_age, max_occupant_age, mean_occupant_age, mean_household_age, house_type, house_square_footage, house_number_of_bedrooms, house_number_of_floors, street, street_number, suburb, email_provider

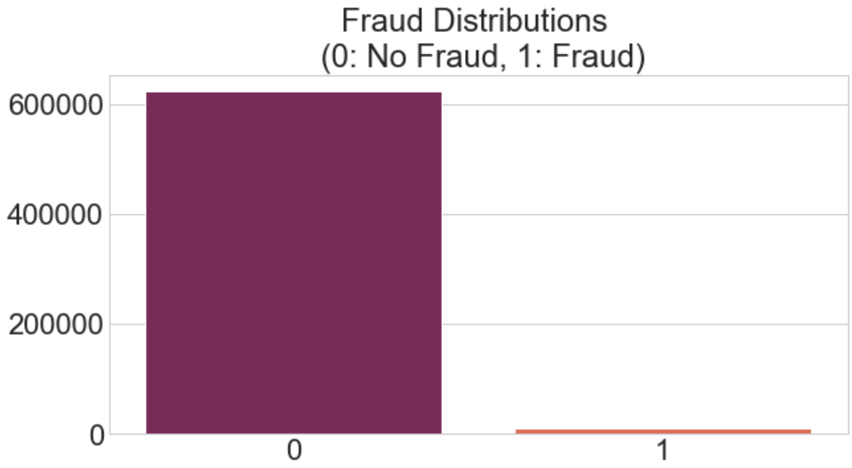
Imputation and Encoding

After the data cleaning and feature engineering, there were a number of columns with significant amounts of missing or undefined values, which could potentially cause problems later in the modelling process. In order to effectively model this data, the missing values had to be filled in. Columns were filled with empty strings, zero values or mean values where appropriate. The mean values were calculated on the mean values for that particular column in accordance with preceding columns in the dataframe. For example, in order to calculate house square footage for a house of type 1, the mean square footages for type 1 houses was calculated.

In order to proceed to modelling, encoding was needed so that the values could be processed by classification algorithms. In order to properly encode such a large dataset, label encoding from sklearn’s preprocessing library in Python was heavily utilized. The possibility of using one hot encoding was also explored but this was thought to add excessive columns to an already large dataframe and may have made modeling a time costly process.

Exploratory Data Analysis

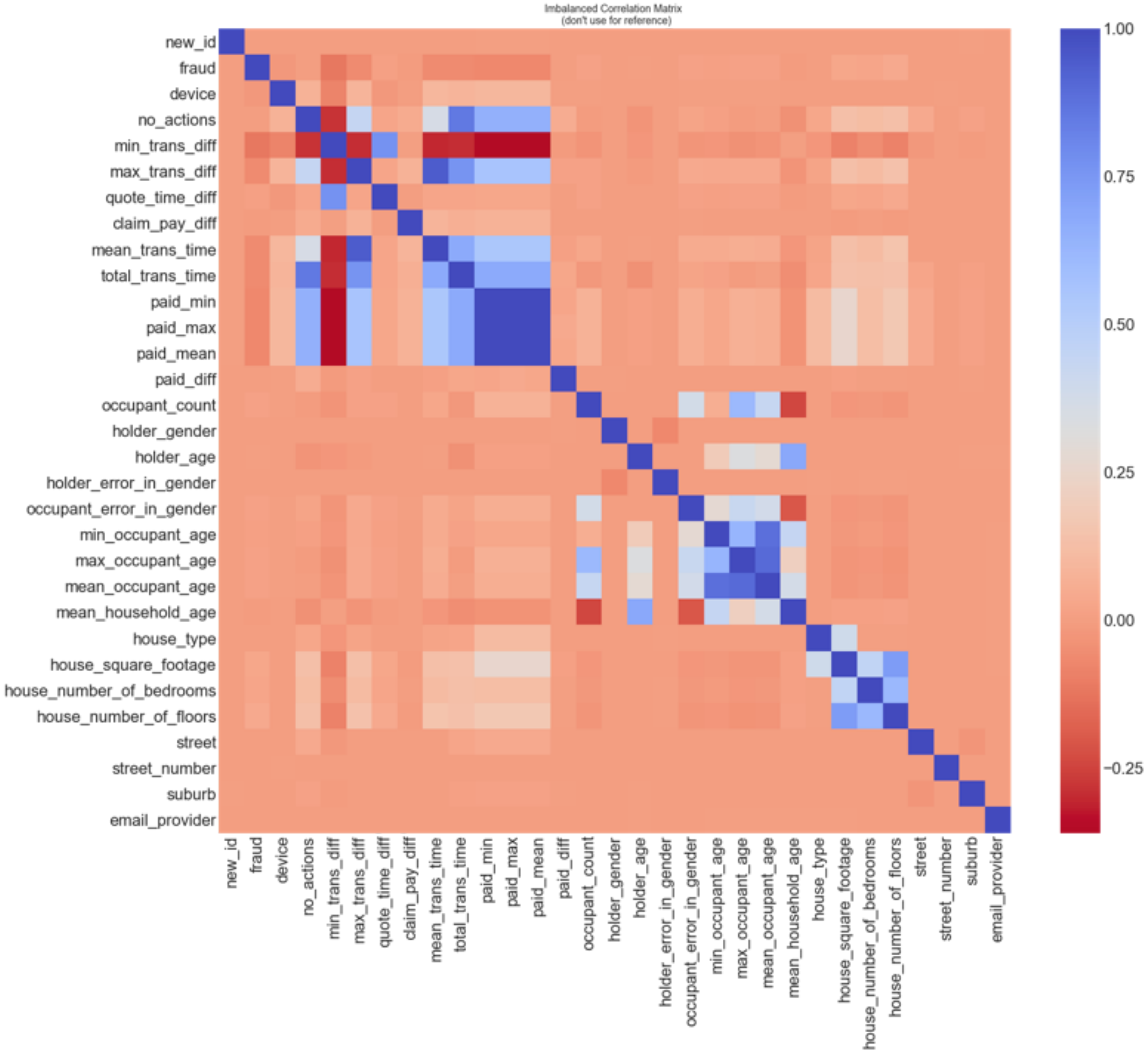
An exploratory data analysis was conducted on the data in order to provide an understanding of the data. It became clear very quickly that the dataset was imbalanced, out of 632998 customers, 98.34% were non-fraudulent customers and 1.66 % were fraudulent. This is best illustrated in the following plot:



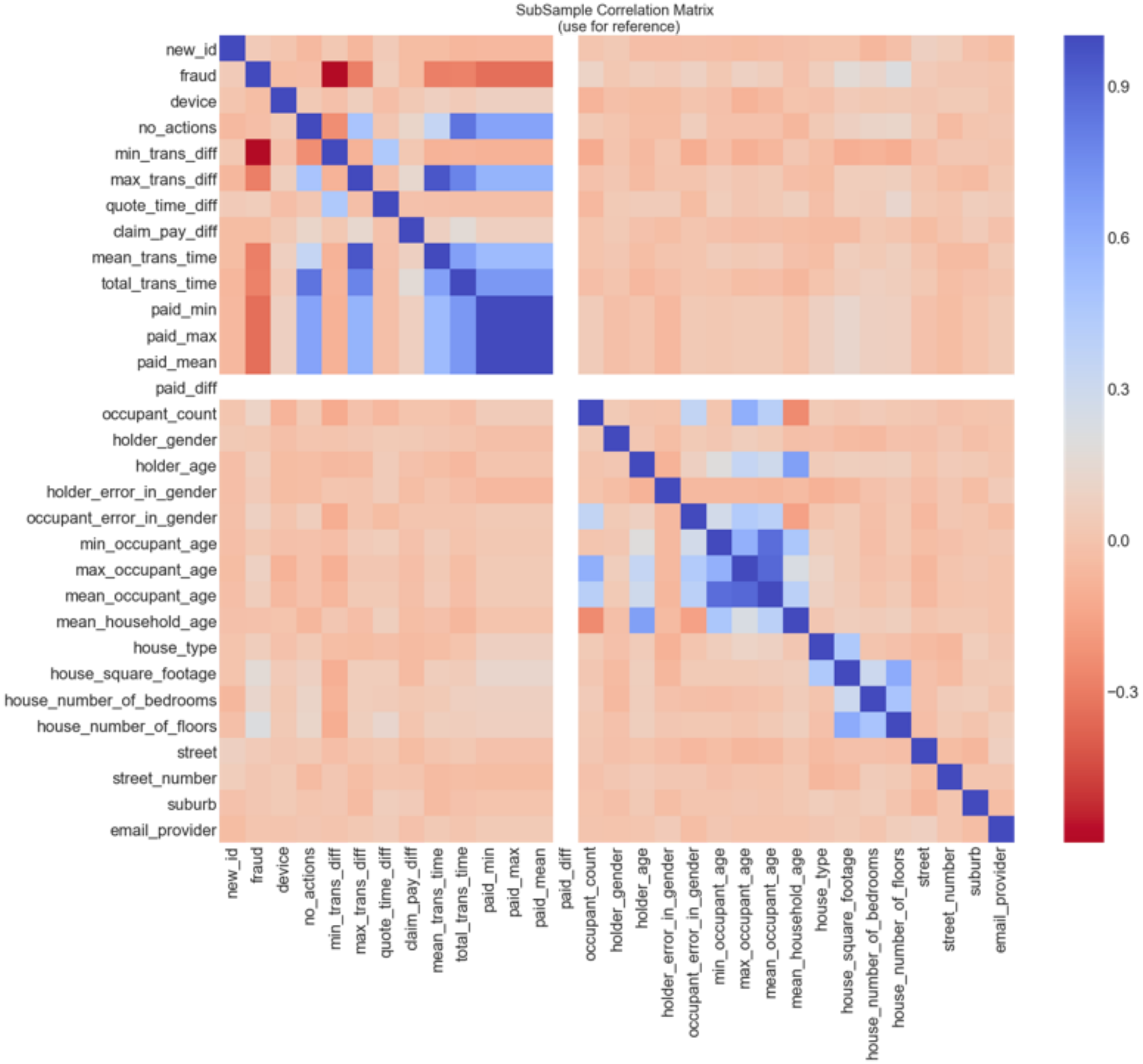
Consequently, a model trained on this dataset may simply predict not fraudulent for each customer and would be still be 99% accurate if accuracy was calculated on how many values the model predicted correctly, as the majority of values would be not fraudulent.

This kind of accuracy metric would not account for the accuracy of detecting the small percentage of actually fraudulent customers, which is the purpose of this project. Thus, it was necessary to find an accuracy metric that deals with this imbalance. This will be discussed later in the Analyses and Results section.

In order to acquire a broad understanding of how features affected each other in the data, a correlation matrix was created. This allowed for a high level overview of the relationships between predictors and our response in the dataset.

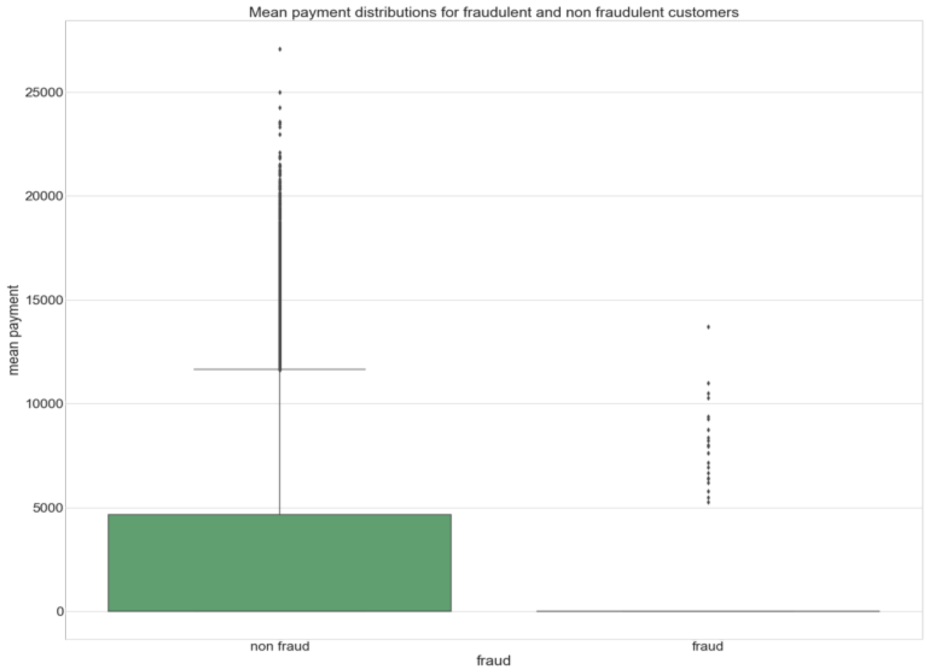


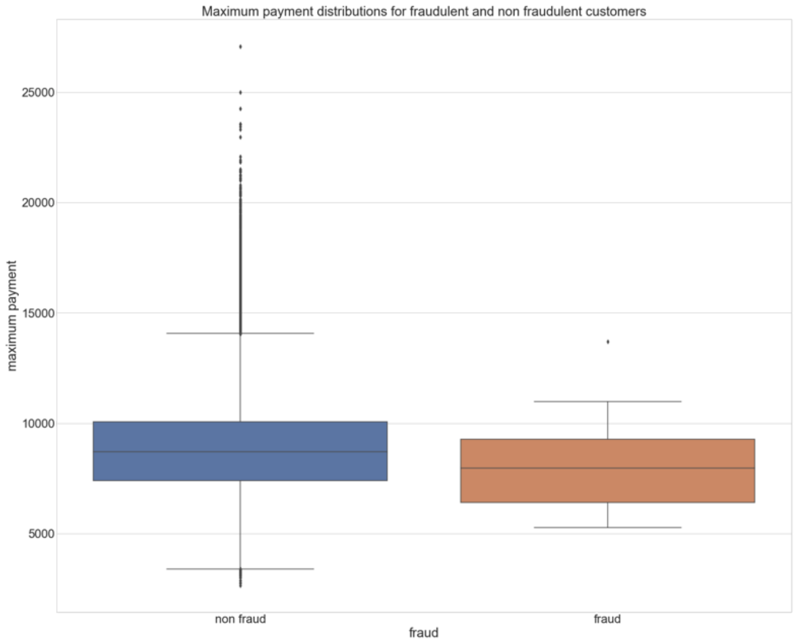
Of particular interest is the relationship between all the variables and fraud. It can be seen that minimum transaction difference, maximum transaction difference, mean transaction time, total transaction time, payment minimum, payment maximum and payment mean are useful predictors. However, this correlation matrix was created on an imbalanced dataset and it is likely that the variables are skewed towards non-fraudulent transactions. As a result, a smaller balanced subsample was created with equitable amounts of frauds and non-frauds in order to better understand the relationships between variables and fraud.


SubSample Correlation Matrix
(use for reference)

In this correlation matrix, made from a balanced dataset, a number of predictors seem to be correlated with fraud. Minimum transaction difference seems to have a strong negative correlation with incidences of fraud, while maximum transaction difference has a weaker negative correlation with fraud. Mean transaction time, total transaction time, payment minimum, payment maximum and payment mean all seem to have a negative correlation with incidences of fraud and may prove to be useful predictors in our analysis. In this matrix, payment difference is highlighted in white. This means that payment difference is 0 for most values and its correlation with other variables cannot be determined.

Minimum, maximum and mean payment were determined to be influential predictors on incidences of fraud. In order to better understand this relationship a boxplot of minimum, maximum and mean payment were constructed for both fraudulent and non-fraudulent transactions.

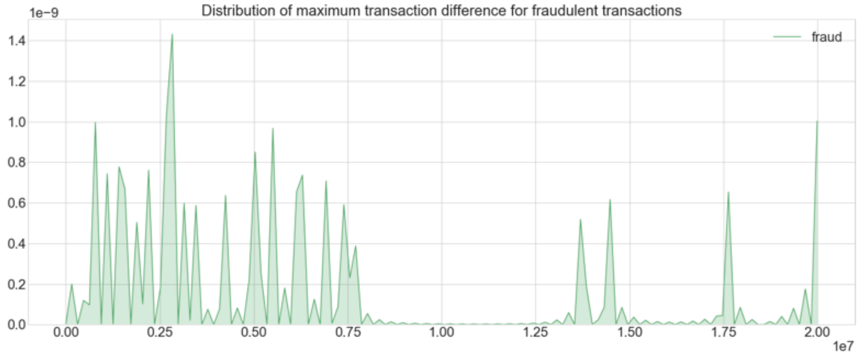


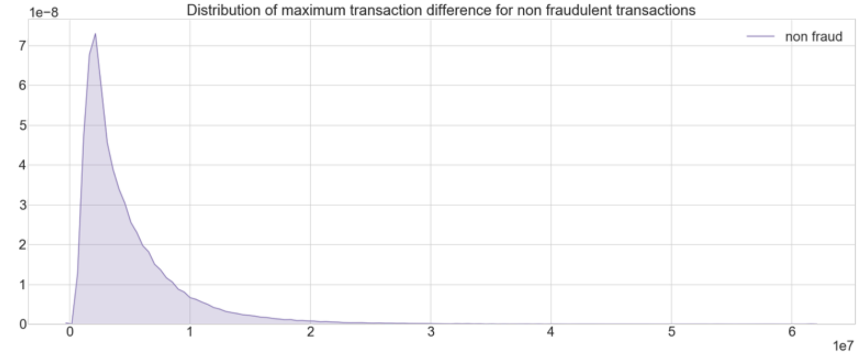


From these boxplots we can observe a few patterns. Firstly, the average fraudulent customer seems to have a mean payment of $0. Interestingly, a significant number of frauds in the dataset do not seem to be attempting to extract funds through false claims. We suspect these frauds are bots trying to establish a trustworthy track record with Yuumi in order to exploit the system further in the future.

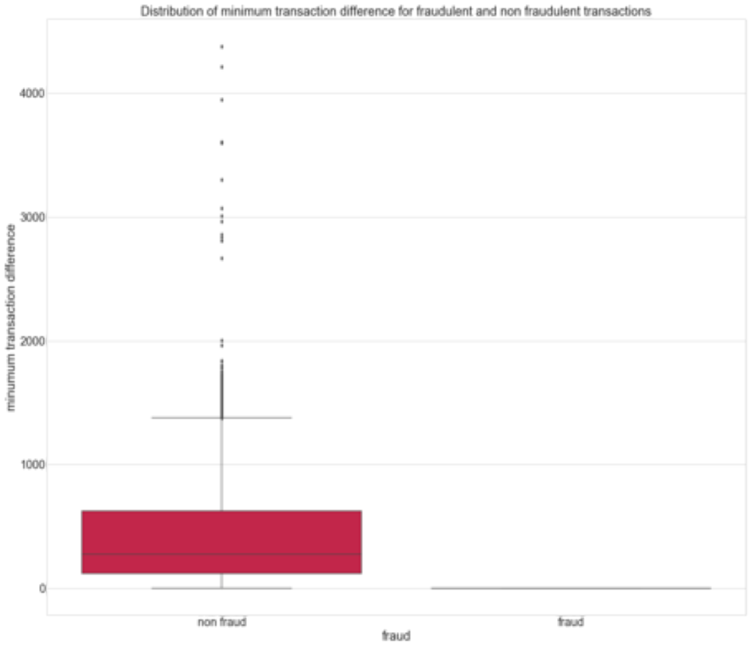
Secondly, the average fraudulent customer has a maximum payment that has a similar distribution to the maximum payments for genuine customers. The fraudsters are careful not to claim excessively high or low amounts of money. However while the interquartile range for fraudsters closely matches that of genuine customers the overall range for genuine customers is far greater than the range for fraudulent customers. In addition there are very few outliers that are fraudulently claiming large sums of money and the fraudulent claims seems to be specified within a strict upper and lower bound. This suggests that there may be a predetermined band in which fraudsters are targeting in order to seem less suspicious. This band seems to begin around the $5000 mark and ends around the $12000 mark.

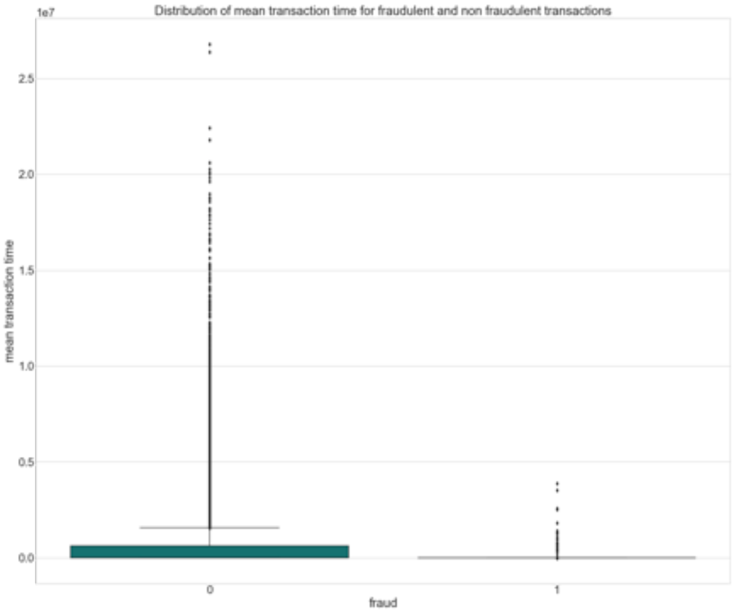
Both minimum and maximum transaction differences were deemed to be strongly negatively correlated with fraud. In order to assess this correlation and their feasibility as predictors density plots of both variables were constructed.





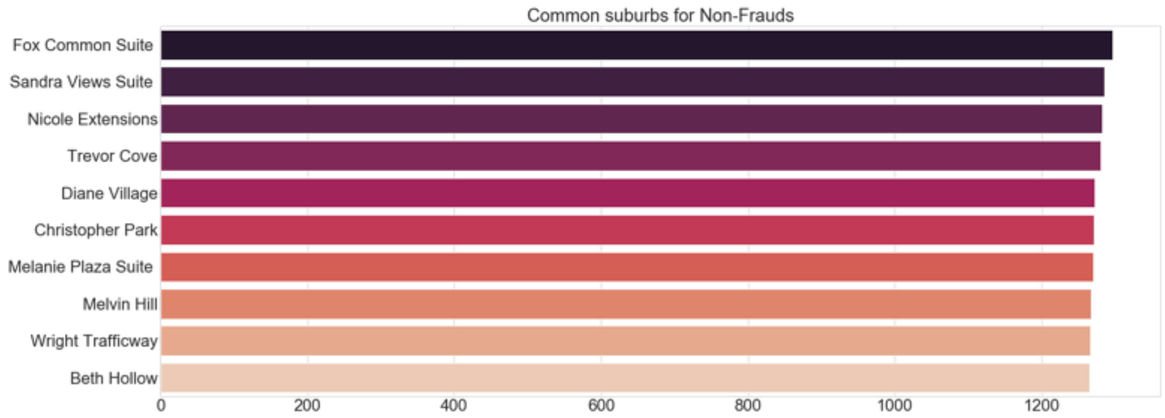
From this graph, maximum transaction difference seems to have a vastly different distribution for frauds and for non frauds. For non frauds, th ere seems to be a right skew to the data while for frauds the maximum transaction difference seems to be contained between 0 and 2.

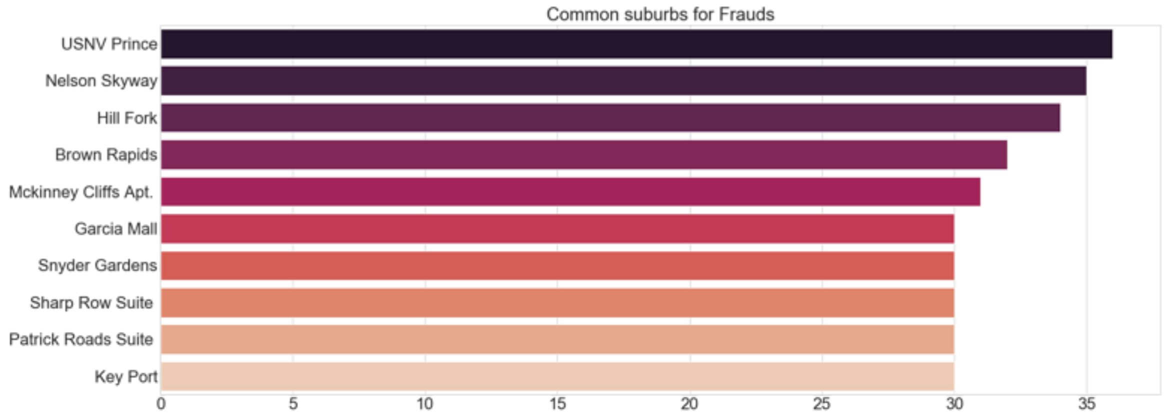




From the graph of minimum transaction difference and mean transaction difference, it is clear that the fraudulent transactions are clustered around 0 while the genuine transactions have a more varied distribution. This clustering around 0 is also prevalent in the plot of mean transaction time. From this information it can be determined that fraudulent transactions are likely to be performed almost instantaneously. **This behaviour is indicative of bots performing transactions near instantly.**

While choice of suburb was not deemed to be an influential predictor from the correlation matrix we believed that location may play a role in determining fraud. In order to assess this assertion an investigation into the suburbs that fraudsters and genuine customers lived in was analysed. This yielded the following plot:





From this graph, it is clear than the common suburbs in which genuine customers live in are not the same as fraudulent customers. On the whole, fraudulent customers seem to come from a select number of locations such as USNV Prince, Nelson Skyway, Hill Fork and Brown Rapids. From this we can determine that there is a correlation between fraudulent activity and these locations.

Dealing with Imbalanced data

When splitting the data into testing and training sets, it was clear that the imbalance of data would potentially cause misleading results in the models as there was over-representation of the non-fraud customers. This would cause models predict a lot of false negatives and provide poor recall, and AUC scores.

Due to the imbalance in the data between fraud and non-fraud values, we needed a way to perform training such that the model being trained could capture features that were characteristic of customers who were fraudulent. We used two methods to deal with this data imbalance when modelling: undersampling and oversampling.

Undersampling

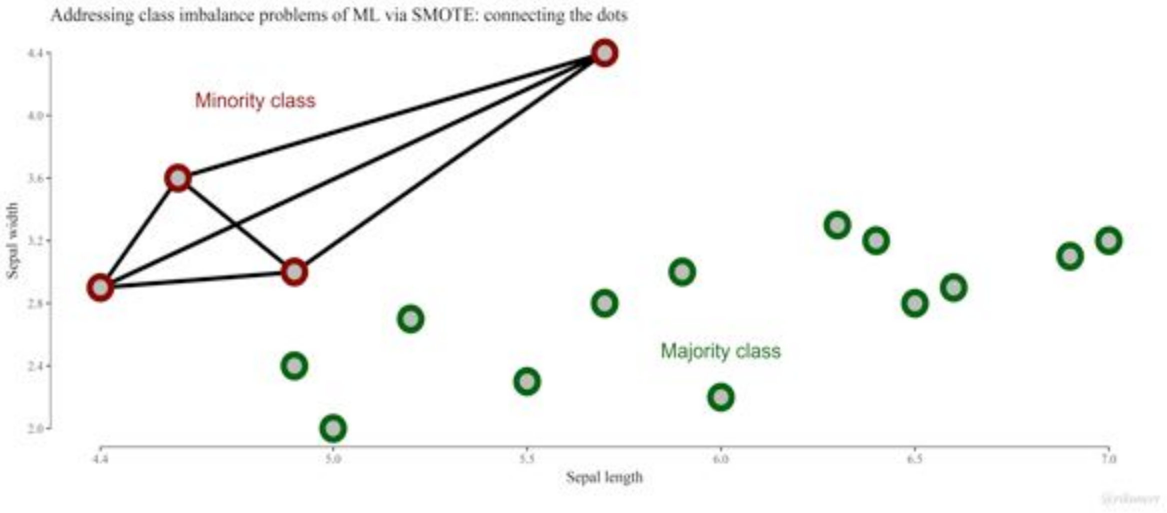
Machine learning models do not cope well with largely unbalanced datasets. The dataset has 1.6% fraud, and 98.4% non-fraud. One method to handle the imbalance is undersampling. Undersampling achieves a balanced dataset by reducing the number of the “over-represented” class in the dataset. The concept of undersampling is as such:

For every observation in the minority class, an observation is taken from the majority class until there are no more observations left in the minority class.

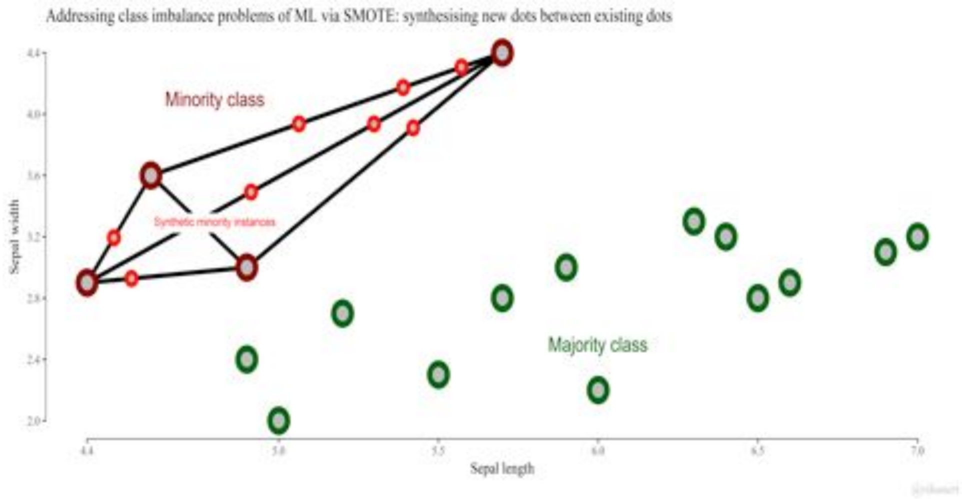
There are two types of undersampling: informed and uninformed. Uninformed undersampling acquires each new observation randomly. Informed undersampling uses calculated metrics such as k-Nearest Neighbours to select an evenly distributed sample from both classes. In our case we will be using uninformed undersampling to create a balanced dataset through the use of Random Undersampling.

Oversampling (SMOTE)

Synthetic Minority Over-sampling Technique (SMOTE) assist in solving the problem of imbalanced data. In order to do this it creates synthetic data for the minority sample based on its k nearest neighbours. For each synthetic data point a vector between the kth nearest neighbours of the current data point is created. This is illustrated in the plot below where the vectors between the neighbours are illustrated in the red minority class.



The result is that imaginary lines are created between data points in the minority class. Then data is instantiantised randomly onto these imaginary lines. In turn this assists in balancing the data.



After the new synthetic minority class is created the data is more balanced than previously. Consequently, the machine learning models created on this dataset are less biased towards the majority class. In the context of insurance fraud the result is that the model is more capable of predicting frauds (the minority class) and in turn fewer frauds go undetected.

Grid Search CV

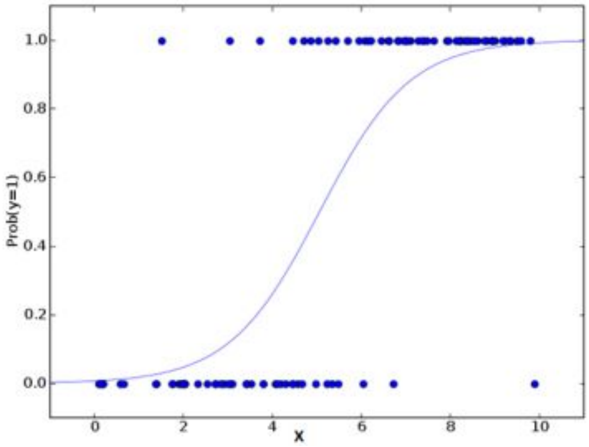
Grid search is the process of hyperparameter tuning. Since our dataset has 31 parameters as columns, it is difficult to find the best coefficient for each parameter using traditional least squares methods. Gridsearch determines the optimal parameters for a particular model. This is necessary as the model is built on the hyperparameter values specified thereby by optimising these parameters the model is improved significantly. In particular GridsearchCV from the sklearn library in python can be used to tune hyperparameters before they are sent to a machine learning model.

Models

In this section we will describe the models we will be using on this dataset to predict fraud. We will go over the results of the models in the Analyses and Results section.

Logistic Regression

Logistic regression is one of the most popular and widely used classification algorithms in statistical learning. A logistic function aims to draw a line of best fit through points that are either 0 or 1. A simple example is shown below, where there is one predictor x.



The goal of this classification problem is to predict fraud, given a number of features. Logistic regression does this by predicting the probability of fraud, P(Y = 1|X), where Y = 1 is fraud and X is all the other features. Then given a threshold, the probability is converted either to a 0 or 1. However logistic regression is not linear, since the predicted probabilities are transformed using a logistic function.

Since

$$F(X) = \frac{e^{b_0+b_1x}}{1+e^{b_0+b_1x}}$$ [the logit function]

$$ln\left(\frac{F(X))}{1-F(X)}\right) = b_0 + b_1x$$

$$ln(odds) = b_0 + b_1x$$

$$odds = e^{b_0+b_1x}$$

The single predictor X can be generalised into multiple predictors X_i.

Thus the odds are the chance that it is a fraud, which is P(Y=1|X). This is calculated by finding the maximum likelihood estimators of the logit function, b_i, then substituting into the odds form. This will return a probability between 0 and 1, in which is fed into the threshold to return the classification outcome.

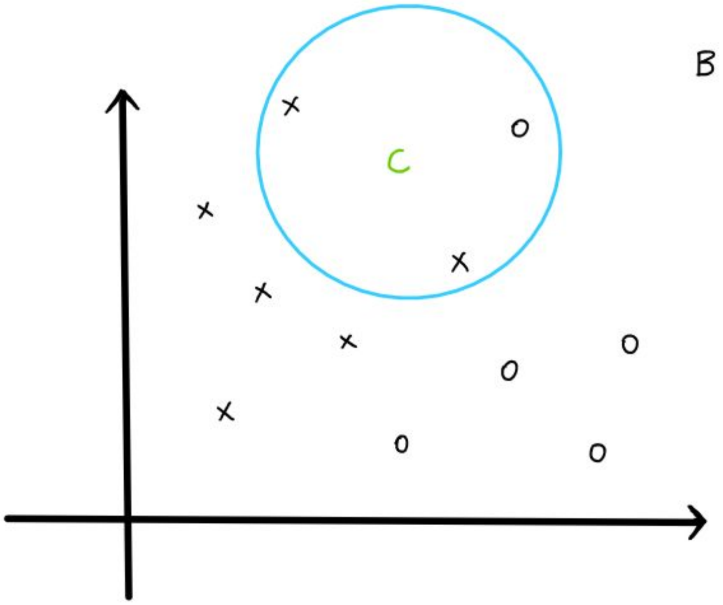
K-Nearest Neighbours

A K-Nearest Neighbours Classification (KNN) model was implemented using R software. This is one of the simpler classification models used in machine learning, and falls under the supervised category. KNN is advantageous for three main reasons:

1) High interpretability (the output is easy to interpret)
2) Low calculation time, the model is fast to run
3) High predictive power

The intuition of KNN is simple. Essentially, given *n* training vectors, in a *B* dimensional feature space, the algorithm identifies the *k* nearest neighbours of the new data point *c*, regardless of labels.

In the diagram below, we set *k* = 3, and find the 3 most closest neighbours of *c*. We have two elements of the class X, and one element of the class O. Thus, we set the class of c = X.



This concept can be generalized to higher dimensions and larger *k*. In R, this is achieved by using the knn function from the “class” package. Some things to keep in mind when choosing the parameter *k*, is that *k* needs to be an odd value for a two-class classification problem, and *k* must not be a multiple of the number of classes.

XGBOOST

XGBoost is a gradient boosting software library that provides a scalable portable and distributable gradient boosting. The software package is contained in a variety of languages and for the purposes of this project the python package was used.

XGBoost was chosen as it based on ensembles of decision trees which in turn allows for optimised performance on imbalanced data. In this respect the model is similar to the decision tree classifier. Undersampling was thought to be an ineffective method of handling imbalanced data as a vast sample the majority class was discarded. In addition XGBoost provides an inbuilt framework for handling missing data and naturally allows for speedup via parallel processing. Lastly, XGBoost allows for a weighting on the minority class compared to the majority class which in turn can account for the skewed nature of the data. As a result, it was expected that XGBoost outperform models trained on SMOTE and Undersampling.

In order to assess the effectiveness of XGBoost the area under the precision-recall curve was used (AUPRC) rather than AUROC. The reason for this is that the area under the precision-recall curve is more sensitive to differences in parameter settings than AUROC and this would be beneficial in handling the skewed data.

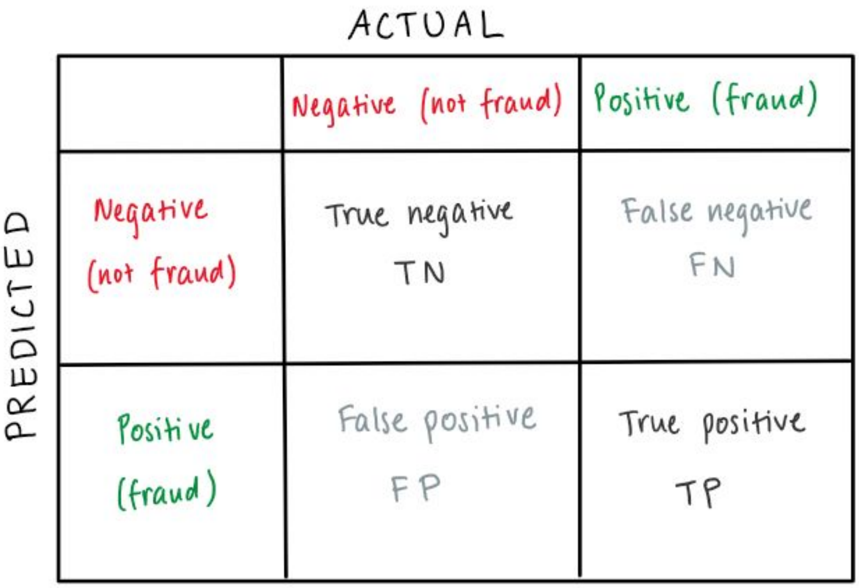
Analyses and results

Metrics

In order to measure the effectiveness of our model, we must take into account the imbalance in our data. Firstly let’s understand some definitions.

- True negative (tn) = customer is not a fraud, and it is predicted as non-fraud.
- True positive (tp) = customer is a fraud, and it is predicted as fraud.
- False negative (fn) = customer is a fraud, but it is predicted as non-fraud.
- False positive (fp) = customer is not a fraud, but it is predicted as fraud.

The table below summarises the these points.



		ACTUAL	
		Negative (not fraud)	Positive (fraud)
PREDICTED	Negative (not fraud)	True negative TN	False negative FN
	Positive (fraud)	False positive FP	True positive TP

As we are dealing with insurance fraud detection, it is necessary to consider what kind of metrics we want to use to measure the accuracy of our models. In insurance fraud detection, we do not want to give away money to customers who are fraudulent. This only occurs when the fraud detection system has failed and the customer is labelled as non-fraud even though they are fraud. Thus, money is lost when there are false-negatives. The number of false negatives is the converse to the number of true-positives, so the more true positives we detect, the less false negatives we have.

On the flip side let us consider what would happen if an insurance company labelled someone as fraud when they are actually non-fraudulent, the false positive case. Someone accused of being fraudulent would probably not likely to come back to the company, and the company would permanently lose a customer, and if the problem is large could lead to damaging the company image, or even lead to being investigated by the Australian Securities and Investments Commission (ASIC) [1]. While for larger companies this may not be that big of an issue the contrary cannot be said for smaller insurance companies. Ideally we would want to have as few false positives as possible, though in our case it would not seem to be too much of an issue.

Thus overall, we aim to maximise the true positive rate and minimise the false positive rate. This would mean that the ordinary accuracy score, below, would not be suitable for the purposes of this project.

$$accuracy\ score\ = \frac{tp+tn}{tp+tn+fn+tn}$$

Instead we will be using the precision, recall and F1-scores to determine how effective our model is at fraud detection.

$$Precision = positive\ predictive\ value\ = \frac{tp}{tp+fp}$$

$$Recall\ = true\ positive\ rate\ = \frac{tp}{tp+fn}$$

$$F1\ score = 2 \times \frac{precision \times recall}{precision + recall}$$

Precision measures how many true positives were predicted over the total amount of positives, and is an indication of how good our model is at predicting true positives while minimising false positives.
Recall measures the amount of true positives over the amount of true positives and false negatives, indicating how well our model is at capturing fraudulent customers.
The F1 score is the harmonic average of the precision and recall and uses both recall and precision to show how well our model is doing.
Overall we want the scores for each of these metrics be as close to one as possible.

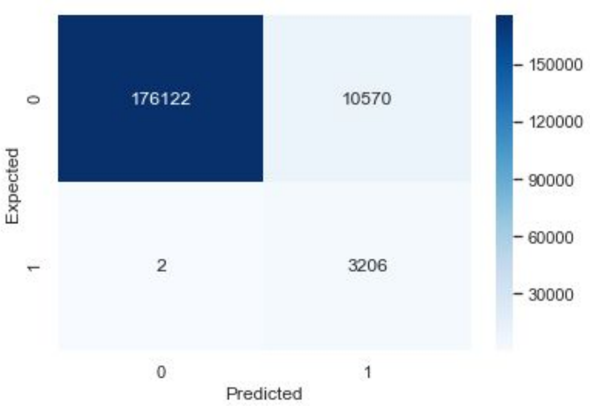
Undersampling Models

We trained logistic regression and K models on the undersampled dataset, where the number of non-fraudulent customers were reduced in the training set to match the number of fraudulent customers.

After undersampling, the dataset was randomly split into 70% training and 30% test set. This resulted in 20988 observations in the training set and 189900 observations in the test set. We note that in the training set there are equal amounts of fraud and non-fraud customers, but in the test set there the proportion of fraud to non-fraud is 97% to 3%.

Logistic Regression

A logistic model was trained and then tested, and the results for model on the test data is shown below.



We can see that the model has quite a large amount of false positives, which is reflected in the F1-score and recall score below.

KNN

For KNN, we will not be weighting the values when calculating the final total accuracy, as we believe the loss to the company of false negatives is equal to the loss from false positives. We justify this as false negative is a short term loss, where the company must pay fraudulent claims, however false positives downgrade the company's reputation and image, thus leads to a long-term loss.

We look at the analysis of models with different parameter *k*. The baseline accuracy is when we predict all results are non-fraudulent, which is 50% due to the method of undersampling.

Model	K parameter	Precision	Recall/TPR	F1-score
Logistic regression	-	0.9993	0.2327	0.38
KNN	3	0.6537	0.7825	0.712324537
KNN	9	0.6494	0.8570	0.73.88951142
KNN	15	0.6469	0.8950	0.75.09897
KNN	19	0.6470	0.9117	0.7568742

KNN	21	0.6450	0.9145	0.7564636

We conclude the KNN model with *k* = 19 gives the best accuracy scores for predicting fraud in undersampling. This is quite an impressive improvement over the baseline. We can also see that despite the Recall improving, meaning the number of false negatives is decreasing, in all the KNN models the precision stays about the same, meaning that the number of false positives is quite high, which could lead to a lot more resources spent on cross checking and investigating, or increase the chance of losing customers.

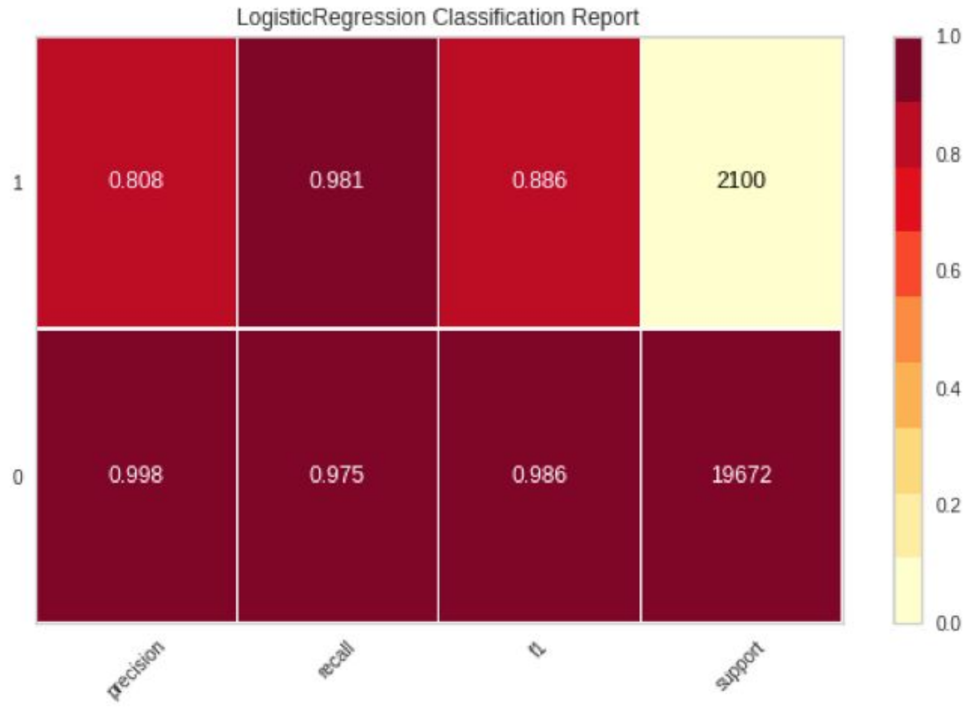
Oversampling (SMOTE Technique)

The data was resampled in order to account for the imbalance. In doing this an approach called SMOTE (Synthetic Minority Over-Sampling Technique) was used. This constructs a new minority class which assists in balancing the data. A grid search technique was used to optimise the hyperparameters before they were fed to the logistic regression model.

The location of these synthetic points is determined by picking the distance between the closest neighbours of the minority (fraud) class. In between these distances a new synthetic data point is created. While in undersampling rows are removed in order to balance the training set, the SMOTE technique retains information as rows are not deleted. However, this requires additional training time and it was expected that this model would be very time inefficient.

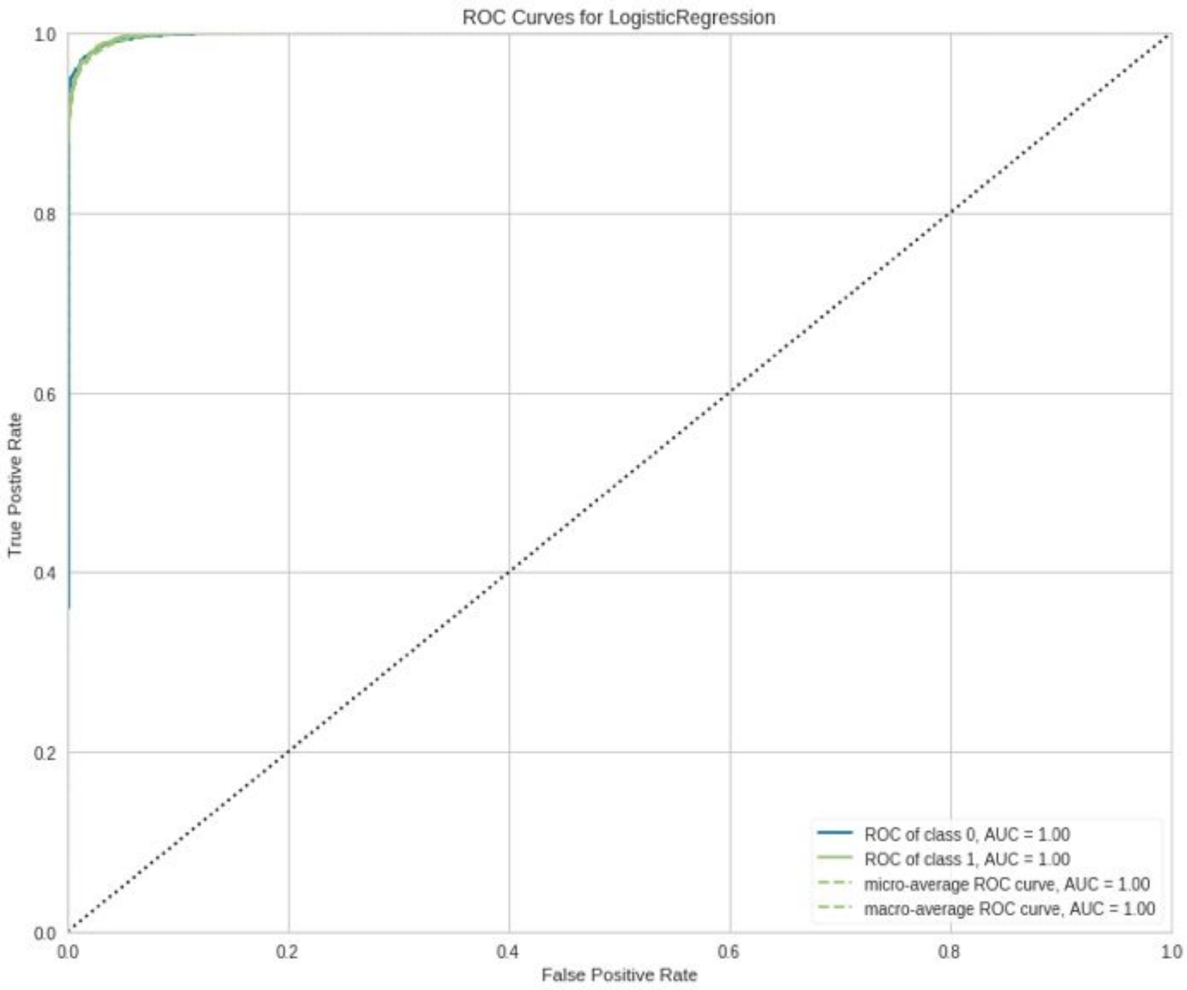
Logistic regression

After the data was oversampled and the logistic regression model was trained the following results were obtained:

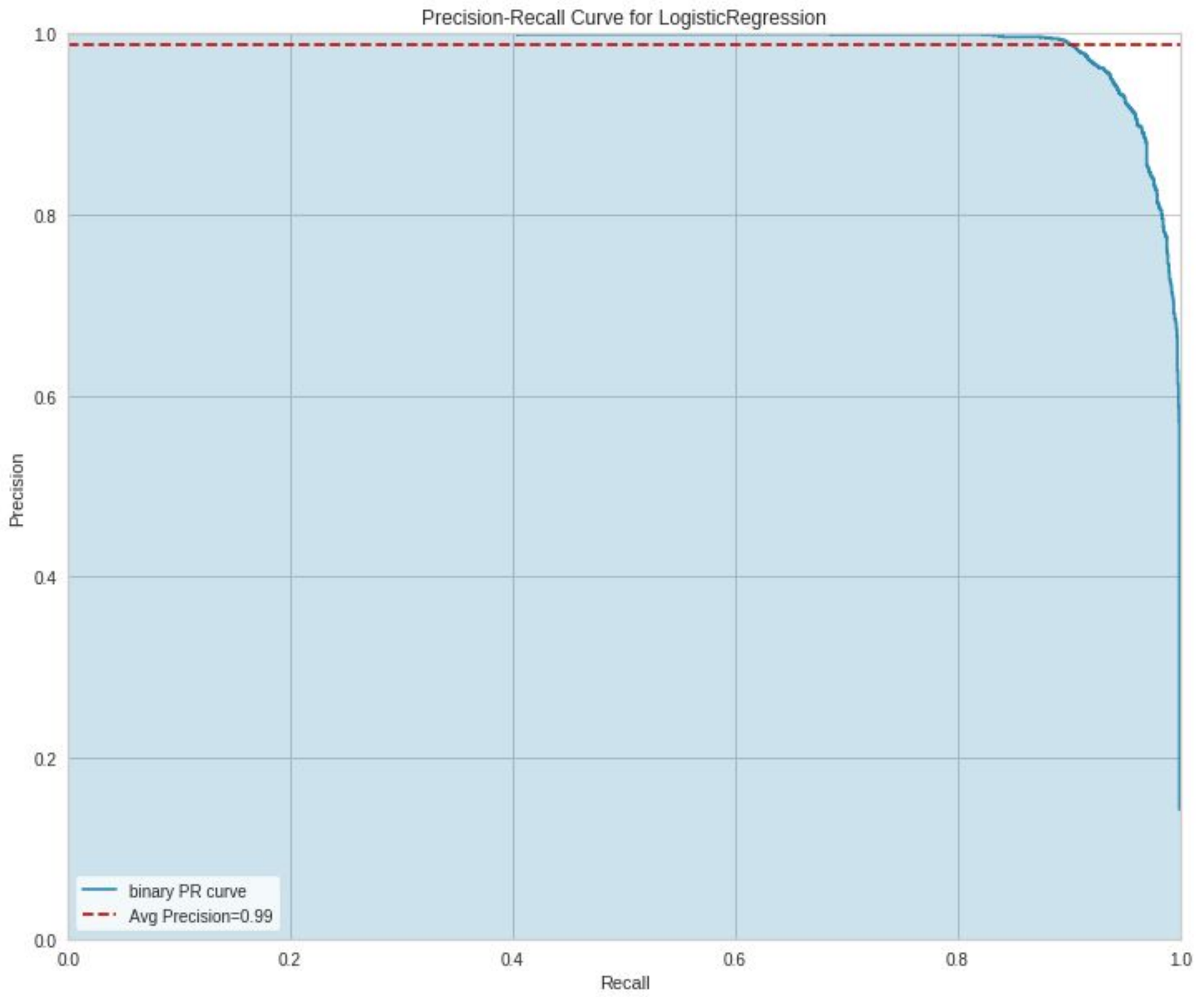


This classification report reveals an interesting pattern: the model has a tendency to over predict frauds. In this way very few fraudsters remain undetected. In the prediction of fraudulent transactions (predicting 1) precision is 0.808 which is moderately high. Therefore, the model does an adequate job of predicting fraud when the customer was actually fraudulent. The recall is 0.981, which is also quite high. Remember that recall measures the proportion of actual positives that was identified correctly. Therefore, the model is very effective at predicting fraudulent customers that actually were fraudulent. The net result is that this oversampled model tends to over predict fraud and let few fraudsters ‘fly under the radar’. This could be extremely useful in an insurance setting where one fraudster could cost the firm thousands of dollars.

We note that the SMOTE logistic regression outperformed the the undersampled models significantly. We decided to plot a ROC curve for this model. The ROC curve plots the true positive rate (TPR) against the false positive rate (FPR), and shows at various thresholds and shows how well the model is in terms of these rates. Overall we want the curve to be and increasing logarithmic curve covering as much area as posible



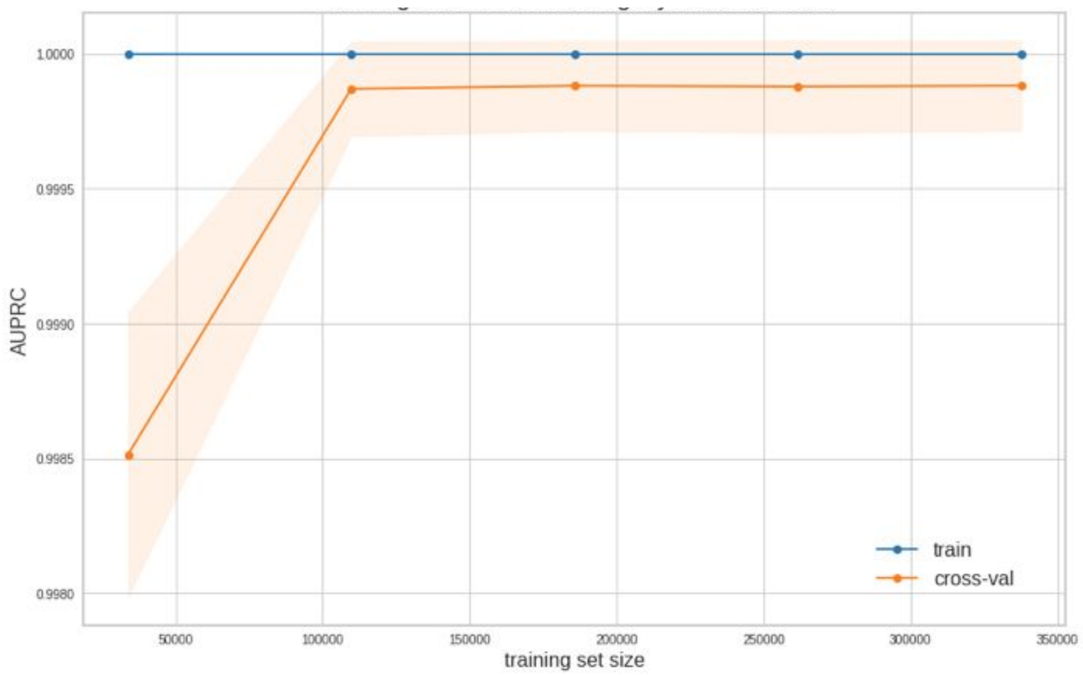
This ROC curve reveals that the model is highly adept at maximising true positives and minimising false positives. The ROC curve remains vertical along the y-axis for most of the plot indicating that false positives are near 0 and begins to curve when true positive rate equals approximately 0.96. At this point the graph becomes horizontal along the true positive rate of 1.0. In addition, the area under the curve for both the class of frauds and non-frauds is 1.00 which is indicative of a very successful model; one that maximises true positives and minimises false positives. In order to further analyse this model a classification report was created:



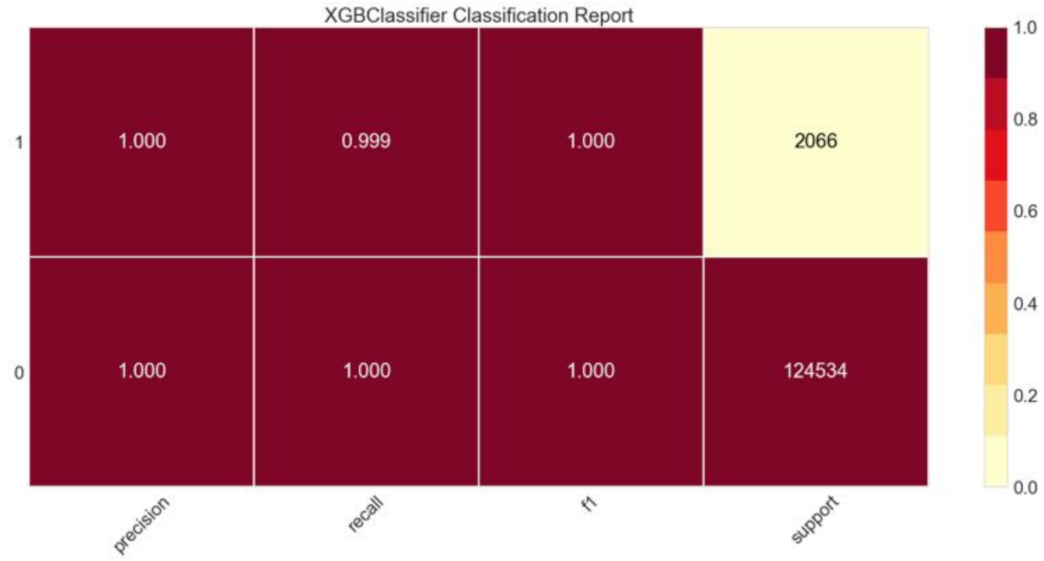
Finally, the precision recall curve seems to suggest that this model has a high degree of precision and an even higher degree of recall. However the area under the curve is less than the AUPRC for XGBoost so it is likely that XGBoost is a better model.

XGBoost

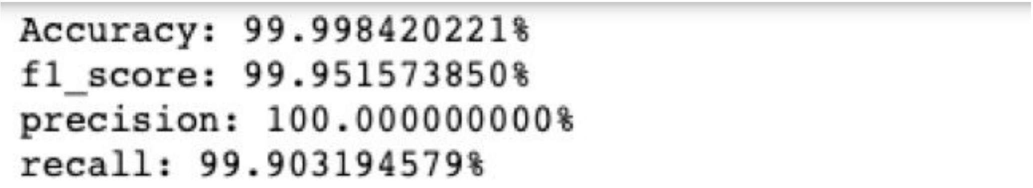
After the XGBoost model was fitted the following results were obtained.



The graph of area under precision-recall curve (AUPRC) vs. training set size for the training and cross validation set provides an indication of the degree of bias and fit. This model has a levelling in the AUPRC curve as the size of the training set increases. From this it can be inferred that the model has a degree of bias and is somewhat underfit. In order to improve this model the max_depth parameter can be adjusted in the XGBoost code. This would increase the amount of time the model spending learning however it would reduce this underfitting. In our case we decided not to do this.



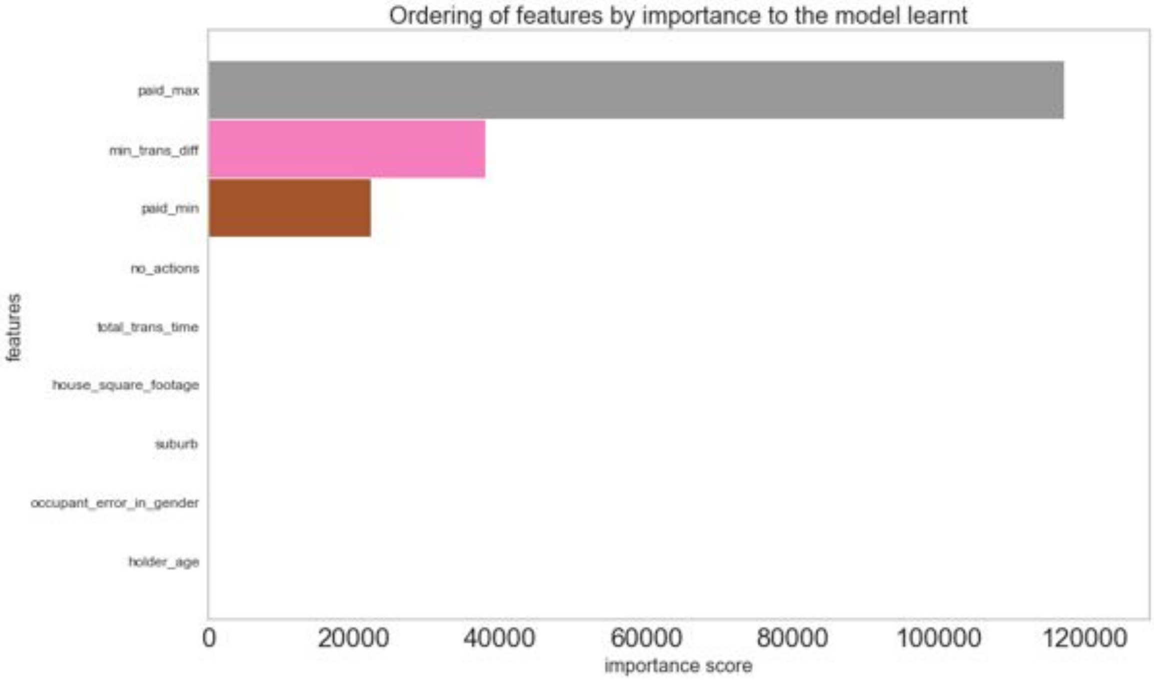
From this classification report it is clear XGBoost has a superior fit. The metrics for precision recall and f1 score have been rounded to 1.000. The following plot provides the unrounded version of the scores.



```
Accuracy: 99.998420221%
f1_score: 99.951573850%
precision: 100.000000000%
recall: 99.903194579%
```

Clearly, this model is a superior fit to all the previous models.

Looking at the model features and finding which features were the most important in the XGBoost model we plot the chart below.



It can be seen that maximum payment is the most influential predictor to the model. The next most important feature is minimum transaction difference and this is backed up from the exploratory data analysis conducted earlier. Maximum payment, minimum transaction difference and minimum payment are extremely influential predictors compared to the other features. This may be due to the imputation conducted earlier in which values were filled with mean values or zeroes appropriately.

Conclusion

The purpose of this report is to detect fraudulent customers based on given log data and finding out which features of the log data contributes the most significantly to a customer being fraudulent. From the report it is clear that the XGBoost model provided the highest predictive accuracy, with a F1-score of 99.95, recall of 99.9, precision of 100.00. Also we notice that the factors that have the strongest influence in detecting whether a customer is fraudulent using the log data is the paid_max columns followed by the min_trans_diff column as well as the max_paid column. It would seem that smaller values of these features often indicate that the customer is fraud. From this project we have learnt which factors in log data makes a customer fraudulent and have successfully predicted fraudulent customers using log data.

There are several limitations to our project that may have impacted the final result. In the imputation stage of data exploration and preprocessing, our methods of imputation were not very complex (using mean, median and setting as 0s) and may have led to some bias in the parameters.

Future research could be done with using different models such as Support Vector Machines (SVMs), Naive Bayes Classifier, neural networks to see if they can create a model that has a greater true positive rate than the one we have created. Also trying out other methods of undersampling (Tomek links) and oversampling (ADASYN which builds on SMOTE) could yield different results. With the goal of preemptively detecting fraudulent customers before they make a claim in mind, we also could potentially seek out more information about customers. More data in the data collection stage such as collecting information about the user's past history or information from cookies could allow us to build a model that has more features. Having more features could potentially give a clearer indication as to which customers are fraudulent. With the future of online interaction expected to increase, extracting data from customers history could be potentially used, though this may run into potential legal issues regarding internet privacy.

References

[1] (2019, February 27). *ASIC targets fraud 'false positives'*. Retrieved from https://www.insurancenews.com.au/daily/asic-targets-fraud-false-positives

[2] Bergstra, J. and Bengio, Y. (2012, February). *Random Search for Hyper-Parameter Optimization*. Retrieved from http://www.jmlr.org/papers/volume13/bergstra12a/bergstra12a.pdf

[3] Bolton, R. J. and Hand, D. J. (2002, August). *Statistical Fraud Detection: A Review*. Retrieved from https://www.jstor.org/stable/pdf/3182781.pdf?refreqid=excelsior%3A9d41ec42a681c7fdb58fbbe40830da07

[4] Briggs, L., Berg, I. and Maron, J. (2016). *The General Insurance Industry Data Report 2014–2015*. Retrieved from : https://www.fos.org.au/custom/files/docs/cgc-20142015-industry-data-report.pdf

[5] Brownlee, J. (2018, August 31). *How and When to Use ROC Curves and Precision-Recall Curves for Classification in Python*. Retrieved from https://machinelearningmastery.com/roc-curves-and-precision-recall-curves-for-classification-in-python/

[6] Davis, J. and Goadrich, M. *The Relationship Between Precision-Recall and ROC Curves*. Retrieved from https://www.biostat.wisc.edu/~page/rocpr.pdf

[7] Ekelund, S. (2017, March). *Precision-recall curves – what are they and how are they used?* Retrieved from https://acutecaretesting.org/en/articles/precision-recall-curves-what-are-they-and-how-are-they-used

[8] Guha, R., Manjunath, S. and Palepu, K. *Comparative Analysis of Machine Learning Techniques for Detecting Insurance Claims Fraud*. Retrieved from https://www.wipro.com/en-AU/analytics/comparative-analysis-of-machine-learning-techniques-for-detectin/

[9] Kunert, R. (2017, November 6). *SMOTE explained for noobs - Synthetic Minority Over-sampling TEchnique line by line*. Retrieved from http://rikunert.com/SMOTE_explained

[10] Malik, U. (2018, July 24). *Cross Validation and Grid Search for Model Selection in Python*. Retrieved from https://stackabuse.com/cross-validation-and-grid-search-for-model-selection-in-python/

[11] Narkhede, S. (2018, May 17). *Understanding Logistic Regression*. Retrieved from https://towardsdatascience.com/understanding-logistic-regression-9b02c2aec102

[12] Narkhede, S. (2018, June 27). *Understanding AUC - ROC Curve*. Retrieved from https://towardsdatascience.com/understanding-auc-roc-curve-68b2303cc9c5

[13] Ngai, E., Hu, Y., Wong, Y., Chen, Y. and Sun, X. (2011, February). *The application of data mining techniques in financial fraud detection: A classification framework and an academic review of literature*. Retrieved from ttps://www.sciencedirect.com/science/article/pii/S0167923610001302

[14] Phua, C., Lee, V., Smith, K. and Gayler, R. *A Comprehensive Survey of Data Mining-based Fraud Detection Research*. Retrieved from https://arxiv.org/ftp/arxiv/papers/1009/1009.6119.pdf

[15] SURETE INVESTIGATION SERVICES. (2019). *The Cost Of Fraudulent Claims For Insurance Companies*. Retrieved from: https://sureteinvestigations.com.au/the-cost-of-fraudulent-claims-for-insurance-companies/

DATA Cleaning

August 11, 2019

[ ]:

```
#!/usr/bin/env python
# coding: utf-8

# # Yuumi Insurance EDA

# In[1]:



#package for cell runtime (dev only)
get_ipython().system('pip install ipython-autotime')
get_ipython().magic('load_ext autotime')


# In[2]:



# import basic libraries
import math
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import time
from datetime import datetime
get_ipython().magic('matplotlib inline')

# data visualisation
import seaborn as sns
sns.set(font_scale=2)
plt.style.use('seaborn-whitegrid')

# preprocessing - encoding
from sklearn.preprocessing import OneHotEncoder, LabelEncoder, label_binarize
from nltk.tokenize import WordPunctTokenizer
import json
from pandas.io.json import json_normalize
#json_normalize(msgLog['message'])
```

```
# make column width bigger when displaying columns
pd.set_option("display.max_colwidth", 1000)
# display more columns
pd.set_option("display.max_columns", None)
# format float display
pd.options.display.float_format = '{:.2f}'.format

#regex
import re


# -----
# # Part 1: Clean/Process Data
# -----

# In[3]:


log = pd.read_csv('Dataset/data.csv')


# In[4]:


# shows columns and the data types of columns
# message       object
# timestamp    float64
# dtype: object
# note object = string
#log.dtypes
# df with 3301834 rows
# len(log)
# look at data, we can check the columns by log.message/log.timestamp
# also log.message[0] shows first value for log.message
log.head(20)


# In[5]:


# the log is sorted in icreasing order
log['timestamp'].is_monotonic


# split the message column into multiple columns, transaction_id, action,␣
 ↪customer_id, action, message
```

```python
# In[6]:


# create msgLog for manipulating message data
msgLog = pd.DataFrame()

#msgLog['message'] = log['message']
msgLog['transaction_id'], msgLog['device'], msgLog['message'] = log['message'].str.split('-',2).str
msgLog['action'], msgLog['message'] = msgLog['message'].str.split(':',1).str
msgLog['customer_id'], msgLog['message'] = msgLog['message'].str.split(n=1).str
msgLog['action'] = msgLog['action'].str.strip()

#msgLog['message'] = msgLog['message'].str.split('json payload',1).str[0]
msgLog['message'], msgLog['json_payload'] = msgLog['message'].str.split('json payload',1).str

msgLog['timestamp'] = log['timestamp']


# In[7]:


msgLog.head()


# In[8]:


msgLog[msgLog['customer_id']=='1c8f1f']


# Drop the transaction column cos its useless, can count number of transactions by actions

# In[9]:


# all transaction id's are unique, not very useful
# msgLog.transaction_id.value_counts()
msgLog = msgLog.drop(columns=['transaction_id'])

# 621123 unique ids, note that some customer ids are repeated
len(msgLog.customer_id.value_counts())
```

```
# Sort values by customer id (thereby grouping by customer id, but not reducing␣
 ↪rows)

# In[10]:


msgLog = msgLog.sort_values(by=['customer_id', 'timestamp'], kind = 'mergesort')
msgLog = msgLog.reset_index(drop=True)


# In[11]:


# add paid dataframe to msgLog
paid = pd.DataFrame()
msgLog['message'], msgLog['paid'] = msgLog['message'].str.split('$').str
msgLog['paid'] = pd.to_numeric(msgLog['paid'])


# In[12]:


msgLog.message.value_counts()


# In[13]:


print(np.nanmin(msgLog['paid']), np.nanmax(msgLog['paid']), np.
 ↪nanmean(msgLog['paid']))


# In[14]:


msgLog.head()


# ## One hot encode the actions and perform functions on time

# In[15]:


# msgLog.action.value_counts()
'''
 Claim Started for customer        826795
 Claim Accepted for customer       816301
 Quote Started for customer        632998
 Quote Completed for customer      523322
 Payment Completed for customer    225921
 Policy Cancelled for customer     156327
 Quote Incomplete for customer     109676
 Claim Denied for customer          10494
 '''

oneHotAction = pd.DataFrame()
oneHotAction =  pd.get_dummies(msgLog['action'])
oneHotAction.columns = ['Cl_A', 'Cl_D', 'Cl_S', 'Pay_Comp', 'Pol_Canc', 'Q_C',␣
 ↪'Q_I', 'Q_S']
#oneHotAction = oneHotAction[['Q_S', 'Q_C', 'Q_I', 'Cl_S', 'Cl_A', 'Cl_D',␣
 ↪'Pay_Comp', 'Pol_Canc']]
oneHotAction['customer_id'], oneHotAction['timestamp'] = msgLog['customer_id'],␣
 ↪msgLog['timestamp']
oneHotAction['no_actions'] = 1
#msgLog = pd.concat([msgLog, one_hot], axis=1)


# ## Out of 632998 customers, 826795 claims, 10494 claim denied (fraud)

# In[16]:


oneHotAction[['Q_C', 'Q_I', 'Q_S', 'Cl_S', 'Cl_D', 'Cl_A']].apply(pd.
 ↪value_counts)


# In[17]:


oneHotAction = oneHotAction.sort_values(by=['customer_id', 'timestamp'], kind =␣
 ↪'mergesort').reset_index(drop=True)
oneHotAction.head()


# Create a function that creates new ids based on Quote Started

# In[18]:


# iterate through quote started column, create new id for each quote started
# note speed - fast cos only binary, slow cos 3 million rows

# checks for customer id changes, if customer id is same, checks for quote␣
 ↪started
```

```
# cases to check for: (same customer id && new quote started)
# (action other than quote started occurs first after customer id change)

def create_new_id(quoteStarted, customerId):
    counter = -1
    array = []
    for i in range(0, len(quoteStarted)):
        if counter != -1:
            # different customer
            if customerId[i] != customerId[i-1]:
                counter += 1
                array.append(counter)
            elif quoteStarted[i] > 0:
                # repeated use of customer id in group
                if customerId[i] == customerId[i-1] and customerId[i] == customerId[i+1] and customerId[i] == customerId[i-2]:
                    counter += 1
                    array.append(counter)
                else:
                    array.append(counter)
            else:
                array.append(counter)
        else:
            counter += 1
            array.append(counter)
    return array


# In[19]:


# create an arsray for the new id, note using vecttorise over np arrays method for speed
oneHotAction['new_id'] = create_new_id(oneHotAction.Q_S.values, oneHotAction.customer_id.values)

# expected minimum id is 0, maximim id is 632997 (= no of 1's in quote started - 1 as index starts from 0)
# note 621123 unique ids
print(min(oneHotAction['new_id']),max(oneHotAction['new_id']))


# In[20]:


# 14 edits to fix new ids as some quote started are after quotes completed
oneHotAction.at[[589335, 589336], 'new_id'] = 113198
oneHotAction.at[[709935, 709936], 'new_id'] = 136438
oneHotAction.at[[888762, 888763], 'new_id'] = 170509
oneHotAction.at[[938775, 938776], 'new_id'] = 180310
oneHotAction.at[[1795406, 1795407], 'new_id'] = 344095
oneHotAction.at[[2778333, 2778334], 'new_id'] = 532807
oneHotAction.at[[2938889, 2938890], 'new_id'] = 563504


# In[21]:


# confirming that new id works for retarded customer ids
oneHotAction[oneHotAction.customer_id == 'd4ec27']


# In[24]:


oneHotAction[oneHotAction['customer_id'] == 'e3dd62']
# oneHotAction[oneHotAction['customer_id'] == '001e75']


# ## get time difference of the quote to quote completed/quote incomplete

# In[25]:


def multiply_col_to_othercol(df, col):
    for i in range(0,len(df.columns)):
        df.iloc[:,i]=df.iloc[:,i]*col
    return df


# In[26]:


quoteTimeDiff = pd.DataFrame()
quoteTimeDiff = pd.concat([quoteTimeDiff, oneHotAction['Q_C'], oneHotAction['Q_I'], oneHotAction['Q_S']], axis=1)
quoteTimeDiff = multiply_col_to_othercol(quoteTimeDiff, oneHotAction['timestamp'])
quoteTimeDiff = pd.concat([quoteTimeDiff, oneHotAction['new_id']], axis=1)
quoteTimeDiff = quoteTimeDiff.groupby(['new_id'], as_index = False).sum()
quoteTimeDiff = quoteTimeDiff.sort_values(by=['new_id'], kind = 'mergesort').reset_index(drop=True)
```

```
# In[27]:


quoteTimeDiff['quote_time_diff'] = quoteTimeDiff[['Q_C', 'Q_S', 'Q_I']].
 ↪sum(axis=1)
quoteTimeDiff['quote_time_diff'] = abs(quoteTimeDiff['quote_time_diff'] -␣
 ↪2*quoteTimeDiff['Q_S'])


# In[28]:


quoteTimeDiff.head()


# In[29]:


#np.mean(quoteTimeDiff.Quote_time_diff.values)
len(quoteTimeDiff[quoteTimeDiff['quote_time_diff'] < 1])


# ## get the time difference between all actions

# In[30]:


def time_difference(time, cid):
    difference = []
    first = 0
    # check if the current id is different to previous id, else get time␣
 ↪difference
    for i in range(0, len(time)):
        if first == 0:
            difference.append(np.nan)
            first = 1
        elif cid[i] != cid[i-1]:
            difference.append(np.nan)
        else:
            difference.append(time[i] - time[i-1])
    return difference


# In[31]:
```

```
oneHotAction['difference'] = time_difference(oneHotAction.timestamp.values,␣
 ↪oneHotAction.new_id.values)


# In[32]:


print(np.nanmin(oneHotAction['difference']), np.
 ↪nanmax(oneHotAction['difference']))


# In[33]:


oneHotAction.difference.value_counts(dropna=False)


# ## get time difference between payment completed and first claim started
#
# Hypothesis/why: if time between payment completed and first claim started ==␣
 ↪0 then high chance of fraud
#
# In that sense we want to be able to distinguish between people who have␣
 ↪started a claim and those who haven't
#
# nan = no claim
#
# time = claim

# In[34]:


def encode_action_customer(value):
    return 1 if value > 0 else 0


# In[35]:


# get payment complete times

firstPayComp = pd.DataFrame()
firstPayComp = pd.concat([firstPayComp, oneHotAction['Pay_Comp']], axis=1)
# check that payment occurs only once (CONFIRMED)
# firstPayComp = pd.concat([firstPayComp, oneHotAction['Pay_Comp'],␣
 ↪oneHotAction['new_id']], axis=1)
# firstPayComp = firstPayComp.groupby(['new_id'], as_index = False).sum()
# firstPayComp.Pay_Comp.value_counts()
firstPayComp = multiply_col_to_othercol(firstPayComp, oneHotAction['timestamp'])
firstPayComp = pd.concat([firstPayComp,␣
 ↪oneHotAction['Pay_Comp'],oneHotAction['new_id']], axis=1)
firstPayComp = firstPayComp.groupby(['new_id'], as_index = False).sum()
firstPayComp.columns = ['new_id', 'Pay_Comp_times', 'Pay_Comp']


# In[36]:


firstPayComp['Pay_Comp_encode'] = firstPayComp['Pay_Comp'].
 ↪apply(encode_action_customer)


# In[37]:


firstPayComp.head()


# In[38]:


# get claim times
claimTime = pd.DataFrame()
claimTime = pd.concat([claimTime, oneHotAction['Cl_S'], oneHotAction['Cl_D']],␣
 ↪axis=1)
claimTime = multiply_col_to_othercol(claimTime, oneHotAction['timestamp'])
claimTime = pd.concat([claimTime, oneHotAction['Cl_S'], oneHotAction['Cl_D'],␣
 ↪oneHotAction['new_id']], axis=1)
claimTime.columns = ['Cl_S_time', 'Cl_D_time', 'Cl_S', 'Cl_D', 'new_id']
claimTime.head()


# In[39]:


claimGrouped = claimTime.groupby('new_id', as_index = False).sum()
claimGrouped['Cl_S_encode'] = claimGrouped['Cl_S'].apply(encode_action_customer)
claimGrouped['Cl_D_encode'] = claimGrouped['Cl_D'].apply(encode_action_customer)
claimGrouped.head()


# In[40]:

```

```
claimGrouped[['Cl_S_encode', 'Cl_D_encode']].apply(pd.value_counts)


# ## of 632998 customers, 168562 have actually made a claim, 464436 didn't make␣
 ↪a claim
# ## of the 10494 denied claims

# In[41]:


claimTimeList = claimTime.groupby(['new_id'], as_index = False)['Cl_S_time'].
 ↪apply(list)
claimTimeList.index


# In[42]:


# claim grouped
claimGrouped['Cl_S_times_array'] = claimTimeList
claimGrouped['Pay_Comp_times'] = firstPayComp['Pay_Comp_times']
claimGrouped['Pay_Comp'] = firstPayComp['Pay_Comp']
claimGrouped.head(113199)


# In[43]:


# gets the first claim time of id, if there is no claim then set as nan
def get_first_claim(value):
    array = []
    for i in range(0, len(value)):
        #print(np.nonzero(value[i])[0])
        #print(np.nonzero(value[i])[0][0])
        #print(len(np.nonzero(value[i])[0]))
        if len(np.nonzero(value[i])[0]) > 0:
            array.append(value[i][np.nonzero(value[i])[0][0]])
        else:
            array.append(np.nan)
    return array


# ## Final claim and pay df

# In[44]:
```

```
claimGrouped['Cl_S_time'] = get_first_claim(claimGrouped.Cl_S_times_array.
 ↪values)
claimGrouped['claim_pay_diff'] = abs(claimGrouped['Cl_S_time'] -␣
 ↪claimGrouped['Pay_Comp_times'])
claimGrouped = claimGrouped.drop(columns=['Cl_S_times_array'])
claimGrouped.head()


# In[45]:


claimGrouped.claim_pay_diff.value_counts()


# ## get the min and max transaction time
# Create data frames for minimum and maximum transaction times

# In[46]:


minTime = pd.DataFrame()
minTime['min_trans_diff'], minTime['new_id'] = oneHotAction['difference'],␣
 ↪oneHotAction['new_id']
# create mask
m = (minTime.reindex(minTime['min_trans_diff'].sort_values(ascending=True,␣
 ↪na_position='last').index).duplicated(['new_id']))
# Apply inverted mask
minTime = minTime.loc[~m].sort_values(by=['new_id'], kind = 'mergesort').
 ↪reset_index(drop=True)
minTime.head()


# In[47]:


len(minTime[minTime['min_trans_diff']  == 0])


# In[48]:


maxTime = pd.DataFrame()
maxTime['max_trans_diff'], maxTime['new_id'] = oneHotAction['difference'],␣
 ↪oneHotAction['new_id']

# create mask
```

```
m = (maxTime.reindex(maxTime['max_trans_diff'].sort_values(ascending=False,␣
 ↪na_position='last').index).duplicated(['new_id']))
# Apply inverted mask
maxTime = maxTime.loc[~m].sort_values(by=['new_id'], kind = 'mergesort').
 ↪reset_index(drop=True)
maxTime.head()


# In[49]:


print(len(maxTime), len(minTime))


# Create data frame for average transaction time

# In[50]:


# sort by new_id, then timestamp, asc and desc, get first value as 2 columns

start = pd.DataFrame()
start['start'], start['new_id'] = oneHotAction['timestamp'],␣
 ↪oneHotAction['new_id']

m = (start.reindex(start['start'].sort_values(ascending=True).index).
 ↪duplicated(['new_id']))
start = start.loc[~m]
start = start.reset_index(drop=True)

end = pd.DataFrame()
end['end'], end['new_id1'] = oneHotAction['timestamp'], oneHotAction['new_id']

m = (end.reindex(end['end'].sort_values(ascending=False).index).
 ↪duplicated(['new_id1']))
end = end.loc[~m]
end = end.reset_index(drop=True)


# In[51]:


grouped = oneHotAction.groupby(['new_id'], as_index = False).sum()
grouped = grouped.sort_values(by=['new_id'], kind = 'mergesort').
 ↪reset_index(drop=True)
grouped[['Q_C', 'Q_I', 'Q_S', 'Cl_D']].apply(pd.value_counts)
```

```
# ## Final time data df

# In[52]:


claimGrouped.head()


# In[53]:


time = pd.DataFrame()
time = pd.concat([grouped['new_id'], grouped['no_actions'],␣
 ↪minTime['min_trans_diff'], maxTime['max_trans_diff'], start['start'],␣
 ↪end['end'], claimGrouped['claim_pay_diff']], axis=1)
time['quote_time_diff'] = quoteTimeDiff['quote_time_diff']
time['total_trans_time'] = (time['end'] - time['start'])
time['mean_trans_time'] = (time['end'] - time['start'])/(time['no_actions']-1)
time = time.drop(columns = ['start', 'end'])
time = time[['new_id', 'no_actions', 'min_trans_diff', 'max_trans_diff',␣
 ↪'quote_time_diff', 'claim_pay_diff', 'mean_trans_time', 'total_trans_time']]
time.head()


# ## Join the new_id from OHA with msgLog

# In[54]:


msgLog = pd.concat([msgLog, oneHotAction['new_id']], axis=1)
msgLog.head()


# In[55]:


len(msgLog)


# # Add Fraud Column
# Add fraud column to msglog

# In[58]:

```

```
#fraud dataframe
fraud_col = pd.DataFrame()
fraud_col['message'], fraud_col['fraud'] = msgLog['message'].str.split('fraud').
 ↪str

#one hot encode
fraud_col = pd.get_dummies(fraud_col, prefix_sep="", columns=['fraud'])

#merge fraud dataframe with msgLog
msgLog['fraud'] = fraud_col['fraud']


# In[59]:


msgLog.head()


# Creating fraud table that shows: customer_id and a 1/0 for fraud. For example,
#
# new_id1     | fraud
#
# ========
#
# 000023 | 0
#
# 356456 | 1

# In[60]:


customer_fraud = msgLog[['new_id', 'fraud']]

#group fraudulent customers by 'customer_id' (probs a better method tbh)
fraudulent = customer_fraud.pivot_table(index=['new_id'], aggfunc=sum)
fraudulent['new_id1'] = fraudulent.index
fraudulent.index = range(fraudulent.shape[0])

fraudulent.head()


# In[61]:


fraudulent.fraud.value_counts()
```

```
# In[63]:


oneHotAction[oneHotAction['new_id'] == 587967]


# Merge this table with msgLog

# In[64]:


#merge df: 'msgLog' and 'fraudulent' on 'customer_id'
merged = pd.merge(msgLog, fraudulent, left_on='new_id', right_on='new_id1',␣
 ↪how='left').drop(['fraud_x', 'new_id1'], axis = 1)
merged = merged.rename(columns={'fraud_y': 'fraud'})
msgLog = merged
msgLog.head()


# In[65]:


#check using nicole cause she's a fraud
msgLog[msgLog['new_id']==380403]


# In[66]:


"""is_fraud = msgLog.action == "Claim Denied for customer"
found_fraud = msgLog[is_fraud]
len(found_fraud)"""
"""#is_fraud = msgLog.customer_id == "d4ec27"
found_fraud = msgLog[msgLog.customer_id == "d4ec27"]
found_fraud.head()"""


# # Payment Dataframe (changes start here)
#
# Create payment dataframe that shows min, max and avg payment for each␣
 ↪customer id

# In[67]:


def payment_df(dataframe):
    """
    function to create a payment dataframe

    Shows new_id and min, max and avg payment for each new_id

    returns this payment dataframe
    """

    #create paid dataframe
    paid_df = dataframe[['new_id', 'paid']]

    #find min, max and avg payment for each new_id
    paid_df = paid_df.groupby('new_id')['paid'].agg([pd.np.min, pd.np.max, pd.
 ↪np.mean])

    #rename col names for clarity
    paid_df = paid_df.rename(index=str, columns={'amin': 'paid_min', 'amax':␣
 ↪'paid_max', 'mean': 'paid_mean',})

    #create new_id col for merging later on
    paid_df['new_id'] = paid_df.index
    paid_df['new_id'] = paid_df['new_id'].astype(int)

    #rename index
    paid_df.index.names = ['']

    return paid_df


# In[68]:


paid_df = payment_df(msgLog)
paid_df['paid_diff'] = abs(paid_df['paid_min'] - paid_df['paid_max'])
paid_df.head()


# # Group by new id
#
# Group customers by new_id. Will do the following things:
#
# 1. merge with time df
# 2. then use the groupby function.
# 3. merge with paid_df

# In[70]:
```

```
backup2 = msgLog.copy()
#msgLog = msgLog2.copy()


# In[71]:


#1. use the groupby function.
msgLog = msgLog.groupby(['device','json_payload','fraud'], as_index =␣
↪False)[['new_id']].sum().sort_values(by=['new_id'])

#2. merge with time df
msgLog = pd.merge(msgLog, time, left_on='new_id', right_on='new_id', how='left')

#3. merge with paid_df
msgLog = pd.merge(msgLog, paid_df, left_on='new_id', right_on='new_id',␣
↪how='left')

msgLog.head()


# In[72]:


msgLog.head()


# # JSON to dataframe
#
# Convert json payload into dataframe and append this dataframe to msgLog

# In[73]:


#create columns (initialise to nan)
msgLog['address'] = np.nan
msgLog.head()


# # REMINDER: This function makes a huge assumption (see the caution in the␣
↪code)

# Faster implementation that uses regex through df.apply
#
# please see: https://regex101.com/r/MrGEDo/1 (string) and https://regex101.com/
↪r/xejhC5/1 (numeric) for details on how regex works here
```

```
#
# This code could use a few helper functions as it's slightly repetitive. I␣
 ↪tried making them initially but ran into too many errors.
#
# Also see caution in the code
#

# In[74]:


import warnings
warnings.filterwarnings('ignore')

def JSON_to_dataframe_regex(dataframe):

    """
    function that uses regex to read json payload and append it into a dataframe

    Needs to be cleaned up. Create a function cause this is repetitive. Also␣
 ↪see caution.

    """

    #replace all single quotes with double quotes
    dataframe['json_payload'] = dataframe['json_payload'].apply(lambda text: np.
 ↪nan if text is np.nan else text.replace("'", '"'))


    #holders
    #holder_name (string)
    dataframe['holder_name'] = dataframe['json_payload'].apply(lambda text: np.
 ↪nan if (text is np.nan or re.search('(?:"name": ")(.*?)(?:")', text) is␣
 ↪None) else re.findall('(?:"name": ")(.*?)(?:")', text)[0])

    #holder_email (string)
    dataframe['holder_email'] = dataframe['json_payload'].apply(lambda text: np.
 ↪nan if (text is np.nan or re.search('(?:"email": ")(.*?)(?:")', text) is␣
 ↪None) else re.findall('(?:"email": ")(.*?)(?:")', text)[0])

    #holder_gender (string)
    dataframe['holder_gender'] = dataframe['json_payload'].apply(lambda text:␣
 ↪np.nan if (text is np.nan or re.search('(?:"gender": ")(.*?)(?:")', text) is␣
 ↪None) else re.findall('(?:"gender": ")(.*?)(?:")', text)[0])

    #holder_age (numeric)
    dataframe['holder_age'] = dataframe['json_payload'].apply(lambda text: np.
↪nan if (text is np.nan or re.search('(?:"age": )(\d+\.?\d*)', text) is None)␣
↪else re.findall('(?:"age": )(\d+\.?\d*)', text)[0])


    #house
    #type (numeric)
    dataframe['house_type'] = dataframe['json_payload'].apply(lambda text: np.
↪nan if (text is np.nan or re.search('(?:"type": )(\d+\.?\d*)', text) is␣
↪None) else re.findall('(?:"type": )(\d+\.?\d*)', text)[0])

    #sqaure feet (numeric)
    dataframe['house_square_footage'] = dataframe['json_payload'].apply(lambda␣
↪text: np.nan if (text is np.nan or re.search('(?:"square_footage": )(\d+\.?
↪\d*)', text) is None) else re.findall('(?:"square_footage": )(\d+\.?\d*)',␣
↪text)[0])

    #house_number_of_bedrooms (numeric)
    dataframe['house_number_of_bedrooms'] = dataframe['json_payload'].
↪apply(lambda text: np.nan if (text is np.nan or re.search('(?:
↪"number_of_bedrooms": )(\d+\.?\d*)', text) is None) else re.findall('(?:
↪"number_of_bedrooms": )(\d+\.?\d*)', text)[0])

    #house_number_of_bedrooms (numeric)
    dataframe['house_number_of_floors'] = dataframe['json_payload'].
↪apply(lambda text: np.nan if (text is np.nan or re.search('(?:
↪"number_of_floors": )(\d+\.?\d*)', text) is None) else re.findall('(?:
↪"number_of_floors": )(\d+\.?\d*)', text)[0])

    #address (string)
    dataframe['address'] = dataframe['json_payload'].apply(lambda text: np.nan␣
↪if (text is np.nan or re.search('(?:"address": ")(.*?)(?:")', text) is None)␣
↪else re.findall('(?:"address": ")(.*?)(?:")', text)[0])


    #occupants
    '''
    CAUTION:
    -------

    will not deal well with missing values as ages/genders in household are␣
↪read in order

    eg.
    household : {[name:_____, gender:______], [name:______, age: 32, gender:␣
↪_______]}
```

```
    Because there is no age entry for the 1st occupant- will think that the 1st occupant is 32 years old.
    '''

    #occupant_name
    dataframe['occupant_name'] = dataframe['json_payload'].apply(lambda text: np.nan if (text is np.nan or re.search('(?:"name": ")(.*?)(?:")', text) is None) else re.findall('(?:"name": ")(.*?)(?:")', text)[1:])

    #occupant_count

    dataframe['occupant_count'] = dataframe['json_payload'].apply(lambda text: 0 if (text is np.nan or re.search('(?:"name": ")(.*?)(?:")', text) is None) else len(re.findall('(?:"name": ")(.*?)(?:")', text)))

    #occupant_age
    dataframe['occupant_age'] = dataframe['json_payload'].apply(lambda text: np.nan if (text is np.nan or re.search('(?:"age": )(\d+\.?\d*)', text) is None) else re.findall('(?:"age": )(\d+\.?\d*)', text)[1:])

    #occupant_gender
    dataframe['occupant_gender'] = dataframe['json_payload'].apply(lambda text: np.nan if (text is np.nan or re.search('(?:"gender": ")(.*?)(?:")', text) is None) else re.findall('(?:"gender": ")(.*?)(?:")', text)[1:])

    return dataframe


# In[75]:


#convert json to df (only takes ~1 minute LOL)
msgLog = JSON_to_dataframe_regex(msgLog)

#drop unnecessary columns
msgLog = msgLog.drop(columns=['json_payload'], axis = 1)

msgLog.head()


# # Seperate columns
#
# Seperate names for holders and occupants into 'first_name' 'last_name'. Uses a nameparser: https://nameparser.readthedocs.io/en/latest/usage.html as this stuff can be very complicated.
# In[76]:


#!pip install nameparser
from nameparser import HumanName

#example usage
#HumanName("Dr. Juan Q. Xavier de la Vega III")['first']


# In[77]:


def first_name_last_name(dataframe):

    """
    function that converts a tuple of names into a tuple of first names and␣
 ↪appends this column to the dataframe
    """

    #occupant
    dataframe['occupant_first_name'] = dataframe['occupant_name'].apply(lambda␣
 ↪tuple1: np.nan if (tuple1 is np.nan or len(tuple1)==0) else␣
 ↪([HumanName(value)['first'] for value in tuple1]))
    #dataframe['occupant_last_name'] = dataframe['occupant_name'].apply(lambda␣
 ↪tuple1: np.nan if (tuple1 is np.nan or len(tuple1)==0) else␣
 ↪([HumanName(value)['last'] for value in tuple1]))

    #holder
    dataframe['holder_first_name'] = dataframe['holder_name'].apply(lambda text:
 ↪ np.nan if (text is np.nan or len(text)==0) else (HumanName(text)['first']))
    #dataframe['holder_last_name'] = dataframe['holder_name'].apply(lambda text:
 ↪ np.nan if (text is np.nan or len(text)==0) else (HumanName(text)['last']))

    #drop names
    dataframe = dataframe.drop('occupant_name', axis = 1)
    dataframe = dataframe.drop('holder_name', axis = 1)

    return dataframe


# In[78]:


msgLog = first_name_last_name(msgLog)
msgLog.head()
```

```
# Seperate address into street number, street name and suburb

# In[79]:


def seperate_address(dataframe):

    #find street, number and suburb from address column
    dataframe['street'] = dataframe['address'].apply(lambda text: np.nan if (text is np.nan or len(text)==0) else " ".join(text.split(',',2)[0].split()[1:]))
    dataframe['street_number'] = dataframe['address'].apply(lambda text: np.nan if (text is np.nan or len(text)==0) else text.split(',',2)[0].split()[0])
    dataframe['suburb'] = dataframe['address'].apply(lambda text: np.nan if (text is np.nan or len(text)==0) else text.split(',',2)[-1])

    #drop address
    dataframe = dataframe.drop('address', axis = 1)

    return dataframe


# In[80]:


msgLog = seperate_address(msgLog)
msgLog.head()


# In[81]:


msgLog['street'] = msgLog['street'].str.strip()
msgLog['street'] = msgLog['street'].str.lower()
msgLog['suburb'] = msgLog['suburb'].str.strip()
msgLog['suburb'] = msgLog['suburb'].str.lower()
msgLog.head()


# Seperate email into email_provider and email_prefix

# In[82]:


def email_seperator(dataframe):
    #get email_prefix and provider
    dataframe['email_prefix'] = dataframe['holder_email'].apply(lambda text: np.
 ↪nan if (text is np.nan or len(text)==0) else text.split('@',2)[0])
    dataframe['email_provider'] = dataframe['holder_email'].apply(lambda text:␣
 ↪np.nan if (text is np.nan or len(text)==0) else "".join(text.split('@',2)[1:
 ↪]))

    #drop email
    dataframe = dataframe.drop('holder_email', axis = 1)

    return dataframe


# In[83]:


msgLog = email_seperator(msgLog)
msgLog.head()


# # Genderize

# Gender discrepancies between people. Found a replacement library that␣
 ↪installs required files locally. It's not exactly space optimised but, it␣
 ↪can handle large datasets.

# In[84]:


#!pip install git+git://github.com/clintval/gender_predictor.git


# In[85]:


from gender_predictor import GenderPredictor

#downloads files for local use
gp = GenderPredictor()
gp.train_and_test()


# In[86]:


def predicted_gender(dataframe):
    """
    function that uses genderpredictor to return a list of predicted genders␣
 ↪for each occupant.

    Used through df.apply for efficiency
    """

    #predict gender for occupants and holder
    dataframe['occupant_pred_gender'] = dataframe['occupant_first_name'].
 ↪apply(lambda list1: np.nan if (list1 is np.nan or len(list1)==0) else [(''␣
 ↪if value == '' else gp.classify(value)) for value in list1])
    dataframe['holder_pred_gender'] = dataframe['holder_first_name'].
 ↪apply(lambda text: np.nan if (text is np.nan or len(text)==0) else gp.
 ↪classify(text))

    return dataframe


# In[87]:


msgLog = predicted_gender(msgLog)
msgLog.head(20)


# # Graphs (unchanged)

# graphs to get an idea of the data

# In[88]:


def bar_plot(count, title):

    """
    function that creates bar plot from value_counts of variable. Just makes␣
 ↪graphing easier.
    """

    x = count.index
    y = count.values
    plt.figure(figsize=(10,5))
    sns.barplot(x,y,palette='rocket')
    plt.title(title)

def bar_plot_horizontal(count, title):
    """
    function that creates a horizontal bar plot from value_counts of variable. Just makes graphing easier.
    """

    y = count.index
    x = count.values
    plt.figure(figsize=(20,10))
    b = sns.barplot(x,y,palette='rocket')
    plt.title(title)



# In[89]:


bar_plot(msgLog['device'].value_counts(ascending = True), '')


# In[90]:


bar_plot(msgLog['fraud'].value_counts(ascending = True), 'Fraud Distributions \n (0: No Fraud, 1: Fraud)')


# Analysing only the fraudulent customers

# In[91]:


#fraudulent customers only
fraud_df = msgLog[msgLog['fraud'] == 1]
fraud_df.index = range(fraud_df.shape[0])


# In[92]:


#graph of males and females
bar_plot(fraud_df['holder_gender'].value_counts(ascending = True), "")


# In[93]:
```

```
#bar_plot_horizontal(fraud_df['suburb'].value_counts(ascending = True), '')


# In[94]:


non_fraud_df = msgLog[msgLog['fraud']==0]


# # Encoding
#
# This section will:
# 1. encode all relevant data using label encoder or one hot encoding as appropriate - complete
# 2. Add (scaled) error between predicted gender and actual gender - complete
#
# Encode stuff in tuples and list. 1 for male 0 for female. Will use self-made encoder.

# # Create 'saved' dataframe to reduce wait time (dev only)

# In[95]:


#cause I ceebs waiting
saved = msgLog.head(1000)
saved.head()


# In[96]:


"""
from sklearn import preprocessing
le = preprocessing.LabelEncoder()
le.fit(('male', 'male', 'female', 'female'))
le.classes_
"""


# In[97]:


#le.fit('female')
#saved['holder_gender'].apply(lambda text: np.nan if (text is np.nan or len(text)==0) else le.transform(text))
```

```
def encode_nested_lists(dataframe):

    """
    function that allows for the encoding of nested lists

    Will use df.apply for efficiency

    male = 1
    female = 0
    M = 1
    F = 0
    """

    dataframe ['occupant_gender'] = dataframe['occupant_gender'].apply(lambda␣
 ↪tuple1: np.nan if (tuple1 is np.nan or len(tuple1)==0) else [(1 if value ==␣
 ↪'male' else 0) for value in tuple1])
    dataframe['occupant_pred_gender'] = dataframe['occupant_pred_gender'].
 ↪apply(lambda list1: np.nan if (list1 is np.nan or len(list1)==0) else [(1 if␣
 ↪value == 'M' else 0) for value in list1])

    return dataframe


# In[98]:


msgLog = encode_nested_lists(msgLog)
msgLog.head()


# label encode gender columns. Use scikit learn label encoder

# In[99]:


backup3 = msgLog.copy()


# In[125]:


#msgLog = backup3.copy()
#msgLog.head()


# In[126]:
```

```
from sklearn.preprocessing import LabelEncoder

def label_encode(dataframe, list_of_encoded_items):

    """
    function to label encode: 'holder_gender', 'holder_pred_gender', 'suburb', 'email_provider', 'device', 'street'

    Male = 1
    Female = 0
    M = 1
    F = 0
    """
    array = []
    for i in list_of_encoded_items:
        enc = LabelEncoder().fit(dataframe[i])
        dataframe[i] = enc.transform(dataframe[i])
        #dataframe[i] = LabelEncoder().fit_transform(dataframe[i])
        array.append(enc.inverse_transform(dataframe[i]))
    """
    dataframe['holder_gender'] = LabelEncoder().fit_transform(dataframe['holder_gender'])
    dataframe['holder_pred_gender'] = LabelEncoder().fit_transform(dataframe['holder_pred_gender'])
    dataframe['suburb'] = LabelEncoder().fit_transform(dataframe['suburb'])
    dataframe['email_provider'] = LabelEncoder().fit_transform(dataframe['email_provider'])

    dataframe['device'] = LabelEncoder().fit_transform(dataframe['device'])
    dataframe['street'] = LabelEncoder().fit_transform(dataframe['street'])
    """

    return dataframe, array


# In[127]:


msgLog, decode = label_encode(msgLog, ['holder_gender', 'holder_pred_gender', 'suburb', 'email_provider', 'device', 'street'])
msgLog.head()


# In[128]:
```

```
decode


# Getting the encoding back

# # Error in Gender
# Find error in gender for each person and their occupants in their house
#
# Error is defined as difference between predicted gender and actual gender.
#
# More details can be found in the code comments

# In[129]:


def list_compare(list1, list2):

    """
    helper function to compare the values two lists

    creates another list of same length. If values are equal 0 is recorded else␣
 ↪1 is recorded. Then summed and divided by length.

    Eg.
    list1 = [0,1,0,0]
    list2 = [0,0,0,0]

    returned_list = [0,1,0,0]
    returned_error = 1/4 = 0.25
    """

    #early exit
    if(len(list1)==0 or len(list2)==0):
        return 0

    #dealing with mismatching lengths (there's a hole in this method - that␣
 ↪links to the caution mentioned above - but I ceebs making this code perfect)
    smallest = min(len(list1), len(list2))

    #new list
    list3 = []

    #print(len(list1))
    #print(len(list2))

    #loop through lists
    for i in range(smallest):
        #if not equal
        if(list1[i]!=list2[i]):
            list3.append(1)

        #if equal
        else:
            list3.append(0)

    #print(sum(list3))
    #print(len(list3))
    #print(list3)

    return sum(list3)/len(list3)



# In[130]:


def gender_error(dataframe):

    """
    driver function to find the error between predicted gender and actual␣
 ↪gender for both occupants and holders
    """

    dataframe['holder_error_in_gender'] = abs(dataframe['holder_gender'] -␣
 ↪msgLog['holder_pred_gender'])
    dataframe['occupant_error_in_gender'] = dataframe.apply(lambda row: np.nan␣
 ↪if (row['occupant_gender'] is np.nan or len(row['occupant_gender'])==0) else␣
 ↪list_compare(row['occupant_gender'], row['occupant_pred_gender']), axis = 1)

    return dataframe


# In[131]:


msgLog = gender_error(msgLog)
msgLog.head()


# In[132]:


#test = msgLog.head(1000)
```

```
# # Occupant Age
#
# get min, max and avg age for occupants

# In[133]:


from statistics import mean

def breakdown_occupant_age(dataframe):

    dataframe['occupant_age'] = dataframe['occupant_age'].apply(lambda list1:␣
 ↪np.nan if (list1 is np.nan or len(list1)==0) else ([int(i) for i in list1]))

    dataframe['mean_occupant_age'] = dataframe['occupant_age'].apply(lambda␣
 ↪list1: 0 if (list1 is np.nan or len(list1)==0) else (mean(list1)))

    dataframe['min_occupant_age'] = dataframe['occupant_age'].apply(lambda␣
 ↪list1: 0 if (list1 is np.nan or len(list1)==0) else (min(list1)))

    dataframe['max_occupant_age'] = dataframe['occupant_age'].apply(lambda␣
 ↪list1: 0 if (list1 is np.nan or len(list1)==0) else (max(list1)))

    dataframe['sum_occupant_age'] = dataframe['occupant_age'].apply(lambda␣
 ↪list1: 0 if (list1 is np.nan or len(list1)==0) else (sum(list1)))


    #dataframe = dataframe.drop('occupant_age', axis = 1)

    return dataframe


# In[134]:


msgLog = breakdown_occupant_age(msgLog)
msgLog.head()


# # REMINDER: CONVERT STRINGS TO NUMERIC
#
# This is a bandaid for a bug that shouldn't exist in the first place.

# In[135]:
```

```
def convert_cols_to_nuermic(dataframe, list_of_cols_to_fix):

    """
    function to fix an issue that shouldn't occur in the first place
    """

    for i in list_of_cols_to_fix:
        dataframe[i] = dataframe[i].astype(float)

    return dataframe

msgLog = convert_cols_to_nuermic(msgLog, ['house_square_footage', 'holder_age',␣
 ↪'house_type', 'house_number_of_bedrooms', 'house_number_of_floors',␣
 ↪'street_number', 'occupant_count'])


# In[136]:


msgLog['mean_household_age'] = (msgLog['sum_occupant_age'] +␣
 ↪msgLog['holder_age'])/msgLog['occupant_count']


# # Drop columns
# drop certain columns. consider keeping these and using them for modelling

# In[137]:


msgLog = msgLog.drop(['occupant_first_name', 'holder_first_name',␣
 ↪'email_prefix', 'occupant_pred_gender', 'occupant_gender', 'occupant_age',␣
 ↪'sum_occupant_age'], axis = 1)
msgLog.head()


# In[138]:


msgLog.dtypes


# In[139]:


len(msgLog.columns)
```

```
# In[140]:


# reordering the table
msgLog = msgLog[['new_id', 'fraud', 'device', 'no_actions', 'min_trans_diff',␣
 ↪'max_trans_diff', 'quote_time_diff', 'claim_pay_diff', 'mean_trans_time',␣
 ↪'total_trans_time', 'paid_min', 'paid_max', 'paid_mean', 'paid_diff',␣
 ↪'occupant_count', 'holder_gender', 'holder_age', 'holder_error_in_gender',␣
 ↪'occupant_error_in_gender', 'min_occupant_age', 'max_occupant_age',␣
 ↪'mean_occupant_age', 'mean_household_age', 'house_type',␣
 ↪'house_square_footage', 'house_number_of_bedrooms',␣
 ↪'house_number_of_floors', 'street', 'street_number','suburb',␣
 ↪'email_provider']]


# In[142]:


msgLog.head()


# ## with NAN Export CSV

# In[143]:


backup4 = msgLog.copy()


# In[144]:


import os
os.getcwd()
msgLog.to_csv('./Dataset/df_nan_with_extra_cols.csv', index = False)
#msgLog.to_csv(os.getcwd()+'df_with_nan.csv',index = False)


# # Imputation
#
# The problem with the data right now is that there are many NA's in some␣
 ↪columns (noticably paid, occupants)
#
# check this out: https://towardsdatascience.com/
 ↪how-to-handle-missing-data-8646b18db0d4
#
```

```
# Paid NA's are considered Mising Not at Random (MNAR): paid depends on whether␣
↪the customer made a claim and only occurs if the claim was accepted.
#
# Fill na's with either 0 or empty string depending on column, filling with 0's␣
↪will skew the data in those columns
#
# Filling with mean will reduce variance in column

# In[145]:


# data visualisation
import missingno
missingno.matrix(msgLog, figsize = (30,10))


# In[146]:


msgLog.isnull().sum()


# # Export to CSV NAN = 0

# In[147]:


def fill_na(dataframe, list_of_cols):
    for col in list_of_cols:
        dataframe[col] = dataframe[col].fillna(0)

    return dataframe


# In[148]:


msgLog = fill_na(msgLog, ['paid_min', 'paid_max', 'paid_mean',␣
↪'house_square_footage', 'house_number_of_bedrooms',␣
↪'house_number_of_floors'])


# In[149]:


import os
os.getcwd()
```

```
msgLog.to_csv('./Dataset/df_nanfilled_0_with_extra_cols.csv', index = False)
#msgLog.to_csv(os.getcwd()+'df_nanfilled_0.csv',index = False)


# In[150]:


msgLog = backup4.copy()
msgLog.head()


# ## Imputation (NAN != all 0)
#
# from the missingno matrix above, we see missing values for claim_pay_diff,␣
 ↪paid_min, paid_max, paid_mean, paid_diff, occupant_error_gender,␣
 ↪house_square_footage, house_number_of_bedrooms and house_number_of_floors

# In[151]:


msgLog.describe()


# In[186]:


len(msgLog[msgLog['fraud'] == 1])


# In[185]:


house_square_dist = msgLog.house_square_footage.dropna()
sns.distplot(house_square_dist)


# In[ ]:


bar_plot(msgLog['house_type'].value_counts(ascend = True), '')


# In[195]:


print(np.nanmean(msgLog['claim_pay_diff']), np.
 ↪nanmedian(msgLog['claim_pay_diff']))
```

```
# It would seem to make sense that those who have not lodged a claim (gone␣
 ↪through payment) would have not been paid, and so have been paid $0

# In[197]:


# filling in paid_min, paid_max, paid_mean with 0's
msgLog['paid_min'] = msgLog['paid_min'].fillna(0)
msgLog['paid_max'] = msgLog['paid_max'].fillna(0)
msgLog['paid_mean'] = msgLog['paid_mean'].fillna(0)
msgLog['occupant_error_in_gender'] = msgLog['occupant_error_in_gender'].
 ↪fillna(np.nanmean(msgLog['occupant_error_in_gender']))
```

DATA3001 Data Modelling

August 10, 2019

# Data Modelling

This section will model the dataframe that was exported in Part 1: Data Cleaning.
It will:

- conduct an EDA into the data
- build a model in XGBoost
- oversampling the data using SMOTE and create a logistic regression classifier on this data

[ ]:

```
#!/usr/bin/env python
# coding: utf-8

# -------
# # Part 2: Data Modelling
# -------

# Welcome to the Jungle
#
# This section will model the dataframe that was exported in Part 1: Data Cleaning.
#
# It wil conduct an EDA and build models using SMOTE oversampling and random undersampling. These models will include: XGBoost, KNN, SVM, Log, DTC

# In[1]:


# import basic libraries
import math
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from datetime import datetime
get_ipython().magic('matplotlib inline')
import time
```

```python
# data visualisation
import seaborn as sns
sns.set(font_scale=2)
plt.style.use('seaborn-whitegrid')

import warnings
warnings.simplefilter('ignore')

# preprocessing - encoding
from sklearn.preprocessing import OneHotEncoder, LabelEncoder, label_binarize
from nltk.tokenize import WordPunctTokenizer
import json
from pandas.io.json import json_normalize
#json_normalize(msgLog['message'])

# make column width bigger when displaying columns
pd.set_option("display.max_colwidth", 1000)
# display more columns
pd.set_option("display.max_columns", None)
# format float display
pd.options.display.float_format = '{:.2f}'.format

#package for cell runtime (dev only)
#!pip install ipython-autotime
get_ipython().magic('reload_ext autotime')

#libraries
import re
import tensorflow as tf
from sklearn.manifold import TSNE
from sklearn.decomposition import PCA, TruncatedSVD
import matplotlib.patches as mpatches
import xgboost as xgb
from sklearn.model_selection import train_test_split, learning_curve
from sklearn.metrics import average_precision_score
from sklearn.model_selection import StratifiedShuffleSplit
from sklearn.model_selection import StratifiedKFold


# Classifier Libraries
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
import collections
```

```
# Other Libraries
from sklearn.model_selection import train_test_split
from sklearn.pipeline import make_pipeline
from imblearn.pipeline import make_pipeline as imbalanced_make_pipeline
from imblearn.over_sampling import SMOTE
from imblearn.under_sampling import NearMiss
from imblearn.metrics import classification_report_imbalanced
from sklearn.metrics import precision_score, recall_score, f1_score,␣
 ↪roc_auc_score, accuracy_score, classification_report
from collections import Counter
from sklearn.model_selection import KFold, StratifiedKFold


# # Read Data from CSV

# read data from part 1 into msgLog

# In[ ]:


# Code to read csv file into Colaboratory:
get_ipython().system('pip install -U -q PyDrive')
from pydrive.auth import GoogleAuth
from pydrive.drive import GoogleDrive
from google.colab import auth
from oauth2client.client import GoogleCredentials
# Authenticate and create the PyDrive client.
auth.authenticate_user()
gauth = GoogleAuth()
gauth.credentials = GoogleCredentials.get_application_default()
drive = GoogleDrive(gauth)


# In[ ]:


link = 'https://drive.google.com/file/d/1bC1QPOPkjprDsB6vv9GMUZeLzYsROMsa/view?
 ↪usp=sharing'
id = '1bC1QPOPkjprDsB6vv9GMUZeLzYsROMsa'
print(id)


# In[ ]:
```

```
downloaded = drive.CreateFile({'id':id})
downloaded.GetContentFile('df_unfilled_nan_extra_cols.csv.zip')


# In[ ]:


get_ipython().system('unzip df_unfilled_nan_extra_cols.csv.zip')


# In[ ]:


msgLog = pd.read_csv('df_unfilled_nan_extra_cols.csv')


# # Imputation (change this to mean)

# Fill na's with either 0 or empty string depending on column

# In[3]:


def fill_na(dataframe, list_of_cols):

    for col in list_of_cols:
        dataframe[col] = dataframe[col].fillna(0)

    return dataframe


# In[4]:


msgLog = fill_na(msgLog, ['paid_min', 'paid_max', 'paid_mean',␣
 ↪'house_square_footage', 'house_number_of_bedrooms',␣
 ↪'house_number_of_floors', 'claim_pay_diff', 'paid_diff',␣
 ↪'occupant_error_in_gender'])


# # Graphs

# graphs to get an idea of the data

# In[5]:
```

```
def bar_plot(count, title):

    """
    function that creates bar plot from value_counts of variable. Just makes␣
 ↪graphing easier.
    """

    x = count.index
    y = count.values
    plt.figure(figsize=(10,5))
    sns.barplot(x,y,palette='rocket')
    plt.title(title)

def bar_plot_horizontal(count, title):

    """
    function that creates a horizontal bar plot from value_counts of variable.␣
 ↪Just makes graphing easier.
    """

    y = count.index
    x = count.values
    plt.figure(figsize=(20,10))
    b = sns.barplot(x,y,palette='rocket')
    plt.title(title)



# In[6]:


bar_plot(msgLog['device'].value_counts(ascending = True), '')


# In[7]:


y = msgLog['fraud'].value_counts(ascending = True).index
x = msgLog['fraud'].value_counts(ascending = True).values
plt.figure(figsize=(20,10))
b = sns.barplot(x,y,palette='rocket')
plt.title('Fraud Distributions')
plt.xticks(np.arange(2), ['non fraud', 'fraud' ])


# Analysing only the fraudulent customers
# In[100]:


#fraudulent customers only
fraud_df = msgLog[msgLog['fraud'] == 1]
fraud_df.index = range(fraud_df.shape[0])
fraud_df.head()
len(fraud_df)


# In[9]:


#graph of males and females
bar_plot(fraud_df['holder_gender'].value_counts(ascending = True), "")


# show wordcloud of suburbs for fraudulent customers

# In[10]:


"""suburbs = ", ".join(text for text in msgLog['suburb'])"""


# In[ ]:


"""from wordcloud import WordCloud, STOPWORDS
def show_wordcloud(data, title = None):

    '''funtion to produce and display wordcloud
        taken 2 arguments
        1.data to produce wordcloud
        2.title of wordcloud'''


    wordcloud = WordCloud(
        background_color='white',
        stopwords=set(STOPWORDS),
        max_words=250,
        max_font_size=40,
        scale=3,
        random_state=1 # chosen at random by flipping a coin; it was heads
    ).generate(str(data))

    fig = plt.figure(1, figsize=(12, 12))
    plt.axis('off')
    if title:
        fig.suptitle(title, fontsize=20)
        fig.subplots_adjust(top=2.3)

    plt.imshow(wordcloud)
    plt.show()
show_wordcloud(suburbs,'most common suburbs')"""


# Graphing of payment max. This may be a useful predictor

# In[11]:


non_fraud_df = msgLog[msgLog['fraud']==0]

# plot of 2 variables
plt.figure(figsize=(20,10))
p1=sns.kdeplot(non_fraud_df['paid_max'], shade=True, color="r", label = 'non fraud')
p1=sns.kdeplot(fraud_df['paid_max'], shade=True, color="b", label = 'fraud')
#sns.plt.show()


# In[12]:


non_fraud_df = msgLog[msgLog['fraud']==0]

# plot of 2 variables
f, (ax1, ax2) = plt.subplots(2, 1, figsize=(24,20))

#ax1.title('Maximum Payment distributions for fraudulent and non fraudulent customers')
p1=sns.kdeplot(non_fraud_df['paid_max'], shade=True, color="r", label = 'non fraud', ax = ax2)
p1=sns.kdeplot(fraud_df['paid_max'], shade=True, color="b", label = 'fraud', ax = ax1)
#sns.plt.show()


# In[13]:


f, (ax1) = plt.subplots(1, 1, figsize=(24,20))
sns.boxplot( x=msgLog["fraud"], y=msgLog["paid_max"] , ax = ax1)
```

```
ax1.set_title('Maximum payment distributions for fraudulent and non fraudulent␣
 ↪customers')
plt.xticks(np.arange(2), ['non fraud', 'fraud' ])
plt.ylabel('maximum payment')


# mean payment

# In[14]:


f, (ax1) = plt.subplots(1, 1, figsize=(24,20))
sns.boxplot( x=msgLog["fraud"], y=msgLog["paid_mean"] , ax = ax1, color = 'g')
ax1.set_title('Mean payment distributions for fraudulent and non fraudulent␣
 ↪customers')
plt.xticks(np.arange(2), ['non fraud', 'fraud' ])
plt.ylabel('mean payment')


# In[15]:


f, (ax1) = plt.subplots(1, 1, figsize=(24,20))
sns.boxplot( x=msgLog["fraud"], y=msgLog["paid_min"] , ax = ax1, color =␣
 ↪'purple')
ax1.set_title('Min payment distributions for fraudulent and non fraudulent␣
 ↪customers')
plt.xticks(np.arange(2), ['non fraud', 'fraud' ])
plt.ylabel('min payment')


# plot that shows gender discrepancies between people

# In[16]:


msgLog.columns


# In[17]:


"""f, axes = plt.subplots(2, 2, figsize=(20, 20))

axes[0, 0].set_title("Fraud \n (don't use for reference)", fontsize=14)
sns.distplot( fraud_df['holder_error_in_gender'] , color="skyblue", ax=axes[0,␣
 ↪0])
```

```
sns.distplot( fraud_df['occupant_error_in_gender'] , color="olive", ax=axes[0,␣
 ↪1])
sns.distplot( non_fraud_df['holder_error_in_gender'] , color="gold", ax=axes[1,␣
 ↪0])
sns.distplot( non_fraud_df['occupant_error_in_gender'] , color="teal",␣
 ↪ax=axes[1, 1])"""

plt.figure(figsize=(20,10))
plt.title('Distribution of error in gender for occupants')
p1=sns.kdeplot(fraud_df['occupant_error_in_gender'], shade=True, color="r",␣
 ↪label = 'fraud')
p1=sns.kdeplot(non_fraud_df['occupant_error_in_gender'], shade=True, color="y",␣
 ↪label = 'non fraud')


# mins_trans_diff predictor

# In[18]:


f, (ax1, ax2) = plt.subplots(2, 1, figsize=(24,20))

sns.distplot(fraud_df["max_trans_diff"] , ax = ax1, color = 'forestgreen',␣
 ↪bins=20)
ax1.set_title('Distribution of maximum transaction difference for fraudulent␣
 ↪transactions')
ax1.set_ylabel('frequency')
ax1.set_xlabel('maximum transaction difference')

sns.distplot(non_fraud_df["max_trans_diff"] , ax = ax2, color = 'royalblue',␣
 ↪bins=20)
ax2.set_title('Distribution of maximum transaction difference for non␣
 ↪fraudulent transactions')
ax2.set_ylabel('frequency')
ax2.set_xlabel('maximum transaction difference')


# In[19]:


f, (ax1) = plt.subplots(1, 1, figsize=(24,20))
sns.boxplot( x=msgLog["fraud"], y=msgLog["max_trans_diff"] , ax = ax1, color =␣
 ↪'goldenrod')
#sns.violinplot( x=msgLog["fraud"], y=msgLog["max_trans_diff"] , ax = ax1)
```

```
ax1.set_title('Distribution of maximum transaction difference for fraudulent␣
 ↪and non fraudulent transactions')
plt.xticks(np.arange(2), ['non fraud', 'fraud' ])
plt.ylabel('maximum transaction difference')


# In[20]:


f, (ax1) = plt.subplots(1, 1, figsize=(24,20))
sns.boxplot( x=msgLog["fraud"], y=msgLog["min_trans_diff"] , ax = ax1, color =␣
 ↪'goldenrod')
#sns.violinplot( x=msgLog["fraud"], y=msgLog["max_trans_diff"] , ax = ax1)

ax1.set_title('Distribution of minimum transaction difference for fraudulent␣
 ↪and non fraudulent transactions')
plt.xticks(np.arange(2), ['non fraud', 'fraud' ])
plt.ylabel('minimum transaction difference')


# In[21]:


# plot of 2 variables
f, (ax1, ax2) = plt.subplots(2, 1, figsize=(24,20))

p1=sns.kdeplot(non_fraud_df['max_trans_diff'], shade=True, color="m", label =␣
 ↪'non fraud', ax = ax2)
p1=sns.kdeplot(fraud_df['max_trans_diff'], shade=True, color="g", label =␣
 ↪'fraud', ax = ax1)

ax2.set_title('Distribution of maximum transaction difference for non␣
 ↪fraudulent transactions')
ax1.set_title('Distribution of maximum transaction difference for fraudulent␣
 ↪transactions')
#sns.plt.show()


# In[22]:


# plot of 2 variables
f, (ax1, ax2) = plt.subplots(2, 1, figsize=(24,20))

p1=sns.kdeplot(non_fraud_df['mean_trans_time'], shade=True, color="r", label =␣
 ↪'non fraud', ax = ax2)
```

```
p1=sns.kdeplot(fraud_df['mean_trans_time'], shade=True, color="b", label =␣
 ↪'fraud', ax = ax1)

ax2.set_title('Distribution of mean transaction time for non fraudulent␣
 ↪transactions')
ax1.set_title('Distribution of mean transaction time for fraudulent␣
 ↪transactions')
#sns.plt.show()


# In[23]:


# plot of 2 variables
f, (ax1, ax2) = plt.subplots(2, 1, figsize=(24,20))

p1=sns.kdeplot(non_fraud_df['total_trans_time'], shade=True, color="m", label =␣
 ↪'non fraud', ax = ax2)
p1=sns.kdeplot(fraud_df['total_trans_time'], shade=True, color="g", label =␣
 ↪'fraud', ax = ax1)

ax2.set_title('Distribution of total transaction time for non fraudulent␣
 ↪transactions')
ax1.set_title('Distribution of total transaction time for fraudulent␣
 ↪transactions')
#sns.plt.show()


# In[24]:


f, (ax1) = plt.subplots(1, 1, figsize=(24,20))
sns.boxplot( x=msgLog["fraud"], y=msgLog["mean_trans_time"] , ax = ax1, color =␣
 ↪'teal')
#sns.violinplot( x=msgLog["fraud"], y=msgLog["max_trans_diff"] , ax = ax1)

ax1.set_title('Distribution of mean transaction time for fraudulent and non␣
 ↪fraudulent transactions')
plt.ylabel('mean transaction time')


# In[25]:


# plot of 2 variables
plt.figure(figsize=(20,10))
plt.title('Maximum Payment distributions for fraudulent and non fraudulent␣
 ↪customers')
p1=sns.kdeplot(non_fraud_df['mean_trans_time'], shade=True, color="r", label =␣
 ↪'non fraud')
p1=sns.kdeplot(fraud_df['mean_trans_time'], shade=True, color="b", label =␣
 ↪'fraud')
#sns.plt.show()


# # Correlation Matrix

# To find which features are helpful in predicting fraud

# In[30]:


len(sample.columns)


# In[31]:


f, (ax1) = plt.subplots(1, 1, figsize=(24,20))

#change font size
plt.rcParams.update({'font.size': 20})

# Entire DataFrame
corr = msgLog.corr()
sns.heatmap(corr, cmap='coolwarm_r', annot_kws={'size':20}, ax=ax1)
ax1.set_title("Imbalanced Correlation Matrix \n (don't use for reference)",␣
 ↪fontsize=14)

"""sub_sample_corr = sample.corr()
sns.heatmap(sub_sample_corr, cmap='coolwarm_r', annot_kws={'size':20}, ax=ax2)
ax1.set_title('Subsample Correlation Matrix \n (use for reference)',␣
 ↪fontsize=20)
plt.show()"""


# There seems to be a correlation between using a mobile_app or mobiel browser␣
 ↪and committing insurance fraud. There also seems to be a strong negative␣
 ↪correlation between min_difference and insurance fraud.

# Comparing fradulent customers with genuine customers in order to find which␣
 ↪variables are useful. Will use heatmap to conduct analysis.
```

```
# In[108]:


saved = msgLog.head(100000)
saved = saved.append(fraud_df.head(20000), ignore_index = True)
print(len(saved[saved['fraud']==1]))
saved.drop_duplicates(keep = 'first', inplace = True)
print(len(saved[saved['fraud']==1]))


# # Splitting the Data

# Split into train and test sets

# In[109]:


from sklearn.model_selection import train_test_split
from sklearn.model_selection import StratifiedShuffleSplit

print('No Frauds', round(msgLog['fraud'].value_counts()[0]/len(msgLog) *␣
 ↪100,2), '% of the dataset')
print('Frauds', round(msgLog['fraud'].value_counts()[1]/len(msgLog) * 100,2),␣
 ↪'% of the dataset')

X = saved.drop('fraud', axis=1)
y = saved['fraud']

sss = StratifiedKFold(n_splits=5, random_state=None, shuffle=False)

for train_index, test_index in sss.split(X, y):
    print("Train:", train_index, "Test:", test_index)
    original_Xtrain, original_Xtest = X.iloc[train_index], X.iloc[test_index]
    original_ytrain, original_ytest = y.iloc[train_index], y.iloc[test_index]

# Turn into an array
original_Xtrain = original_Xtrain.values
original_Xtest = original_Xtest.values
original_ytrain = original_ytrain.values
original_ytest = original_ytest.values


# Clearly, there is a huge difference in correlation of variables between␣
 ↪fraudulent and genuine entries. This should be useful in providing a trend␣
 ↪from which we can build ML models
```

```
# # Model 1: XGBoost
#

# create model using XGBoost

# In[110]:


trainX, testX, trainY, testY = train_test_split(X, y, test_size = 0.2, random_state = 5)
# Long computation in this cell (~1.8 minutes)
weights = (y == 0).sum() / (1.0 * (y == 1).sum())
clf = xgb.XGBClassifier(max_depth = 3, scale_pos_weight = weights, n_jobs = 4)
probabilities = clf.fit(trainX, trainY).predict_proba(testX)
print('AUPRC = {}'.format(average_precision_score(testY, probabilities[:, 1])))


# In[111]:


trainX.head()


# In[112]:


fig = plt.figure(figsize = (14, 9))
ax = fig.add_subplot(111)

colours = plt.cm.Set1(np.linspace(0, 1, 9))

ax = xgb.plot_importance(clf, height = 1, color = colours, grid = False, show_values = False, importance_type = 'cover', ax = ax);
for axis in ['top','bottom','left','right']:
            ax.spines[axis].set_linewidth(2)

ax.set_xlabel('importance score', size = 16);
ax.set_ylabel('features', size = 16);
ax.set_yticklabels(ax.get_yticklabels(), size = 12);
ax.set_title('Ordering of features by importance to the model learnt', size = 20);


# Visualisation of ML model
# In[113]:


xgb.to_graphviz(clf)


# In[114]:


#Long computation in this cell (~6 minutes)

trainSizes, trainScores, crossValScores = learning_curve(xgb.
 ↪XGBClassifier(max_depth = 3, scale_pos_weight = weights, n_jobs = 4),␣
 ↪trainX,                                         trainY, scoring =␣
 ↪'average_precision')


# In[115]:


trainScoresMean = np.mean(trainScores, axis=1)
trainScoresStd = np.std(trainScores, axis=1)
crossValScoresMean = np.mean(crossValScores, axis=1)
crossValScoresStd = np.std(crossValScores, axis=1)

colours = plt.cm.tab10(np.linspace(0, 1, 9))

fig = plt.figure(figsize = (14, 9))
plt.fill_between(trainSizes, trainScoresMean - trainScoresStd,
    trainScoresMean + trainScoresStd, alpha=0.1, color=colours[0])
plt.fill_between(trainSizes, crossValScoresMean - crossValScoresStd,
    crossValScoresMean + crossValScoresStd, alpha=0.1, color=colours[1])
plt.plot(trainSizes, trainScores.mean(axis = 1), 'o-', label = 'train',        ␣
 ↪ color = colours[0])
plt.plot(trainSizes, crossValScores.mean(axis = 1), 'o-', label = 'cross-val', ␣
 ↪        color = colours[1])

ax = plt.gca()
for axis in ['top','bottom','left','right']:
    ax.spines[axis].set_linewidth(2)

handles, labels = ax.get_legend_handles_labels()
plt.legend(handles, ['train', 'cross-val'], bbox_to_anchor=(0.8, 0.15),        ␣
 ↪       loc=2, borderaxespad=0, fontsize = 16);
plt.xlabel('training set size', size = 16);
plt.ylabel('AUPRC', size = 16)
plt.title('Learning curves indicate slightly underfit model', size = 20);
```

```
# In[116]:


y_pred_xgb = clf.predict(testX)
predictions = [i for i in y_pred_xgb]

from sklearn.metrics import accuracy_score, f1_score, precision_score, recall_score

accuracy = accuracy_score(testY, predictions)
print("Accuracy: %.9f%%" % (accuracy * 100.0))
print("f1_score: %.9f%%" % (f1_score(testY, predictions) * 100.0))
print("precision: %.9f%%" % (precision_score(testY, predictions) * 100.0))
print("recall: %.9f%%" % (recall_score(testY, predictions) * 100.0))

#roc_auc_score
print('roc_auc_score: ', roc_auc_score(testY, predictions))

#AUPRC
print('AUPRC = {}'.format(average_precision_score(testY, probabilities[:, 1])))

#average precision recall score
average_precision = average_precision_score(testY, predictions)
print('Average precision-recall score: {0:0.9f}'.format(average_precision))

#confusion matrix (graph)

from sklearn.metrics import confusion_matrix

xgb_cm = confusion_matrix(testY, y_pred_xgb)

fig, ax = plt.subplots(1, 1,figsize=(10,5))
sns.heatmap(xgb_cm, ax=ax, cmap=plt.cm.copper)
ax.set_title("XGBoost Classifier \n Confusion Matrix", fontsize=14)

#precision, recall, f1-score

from sklearn.metrics import classification_report

#print('XGBoost Classifier:')
print(classification_report(testY, y_pred_xgb))


# In[117]:
```

```
get_ipython().system('pip install yellowbrick')
from yellowbrick.classifier import ClassificationReport
plt.figure(figsize=(20,10))
visualizer = ClassificationReport(clf, classes=['0', '1'], support=True)
visualizer.fit(trainX, trainY)  # Fit the visualizer and the model
visualizer.score(testX, testY)  # Evaluate the model on the test data
g = visualizer.poof()             # Draw/show/poof the data


# In[118]:


from yellowbrick.classifier import PrecisionRecallCurve
plt.figure(figsize=(20,10))
# Create the visualizer, fit, score, and poof it
viz = PrecisionRecallCurve(clf, classes=['0', '1'], support=True)
viz.fit(trainX, trainY)
viz.score(testX, testY)
viz.poof()


# # Models 2, 3, 4, 5: SVM, KNN, LOG, DTC

# copied from: https://www.kaggle.com/janiobachmann/
 ↪credit-fraud-dealing-with-imbalanced-datasets
#
# Log, KNN, SVM, Decsion tree models
#
# uses gridsearch to find best parameters

# In[119]:


# re-split data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42)


# In[120]:


#preprocessing
from sklearn import preprocessing


# In[121]:
```

```
X_train.head()


# In[122]:


X_train = preprocessing.MinMaxScaler().fit_transform(X_train)
X_test = preprocessing.MinMaxScaler().fit_transform(X_test)
#y_train = preprocessing.MinMaxScaler().fit_transform(y_train)
#y_test = preprocessing.MinMaxScaler().fit_transform(y_test)


# In[123]:


"""X_train = preprocessing.scale(X_train)
X_test = preprocessing.scale(X_test)
X_test"""


# In[124]:


# Turn the values into an array for feeding the classification algorithms.
#X_train = X_train.values
#X_test = X_test.values
y_train = y_train.values
y_test = y_test.values


# In[125]:


# Let's implement simple classifiers

classifiers = {
    "LogisiticRegression": LogisticRegression(),
    "KNearest": KNeighborsClassifier(),
    "Support Vector Classifier": SVC(),
    "DecisionTreeClassifier": DecisionTreeClassifier()
}

# Wow our scores are getting even high scores even when applying cross␣
 ↪validation.
from sklearn.model_selection import cross_val_score
```

```
for key, classifier in classifiers.items():
    classifier.fit(X_train, y_train)
    training_score = cross_val_score(classifier, X_train, y_train, cv=2)
    print("Classifiers: ", classifier.__class__.__name__, "Has a training score␣
 ↪of", round(training_score.mean(), 2) * 100, "% accuracy score")



# In[126]:


# Use GridSearchCV to find the best parameters.
from sklearn.model_selection import GridSearchCV


# Logistic Regression
log_reg_params = {"penalty": ['l1', 'l2'], 'C': [0.001, 0.01, 0.1, 1, 10, 100,␣
 ↪1000]}

grid_log_reg = GridSearchCV(LogisticRegression(), log_reg_params)
grid_log_reg.fit(X_train, y_train)
# We automatically get the logistic regression with the best parameters.
log_reg = grid_log_reg.best_estimator_


# In[127]:


"""knears_params = {"n_neighbors": list(range(2,5,1)), 'algorithm': ['auto',␣
 ↪'ball_tree', 'kd_tree', 'brute']}

grid_knears = GridSearchCV(KNeighborsClassifier(), knears_params)
grid_knears.fit(X_train, y_train)
# KNears best estimator
knears_neighbors = grid_knears.best_estimator_"""


# In[130]:


# Support Vector Classifier
svc_params = {'C': [0.5, 0.7, 0.9, 1], 'kernel': ['rbf', 'poly', 'sigmoid',␣
 ↪'linear']}
grid_svc = GridSearchCV(SVC(), svc_params)
grid_svc.fit(X_train, y_train)
```

```
# SVC best estimator
svc = grid_svc.best_estimator_


# In[131]:


# DecisionTree Classifier
tree_params = {"criterion": ["gini", "entropy"], "max_depth":␣
 ↪list(range(2,4,1)),
              "min_samples_leaf": list(range(5,7,1))}
grid_tree = GridSearchCV(DecisionTreeClassifier(), tree_params)
grid_tree.fit(X_train, y_train)

# tree best estimator
tree_clf = grid_tree.best_estimator_


# In[132]:


"""

X_test not preprocessed:
-gridsearch taks 28.9 seconds


X_test preprocessed with minmax:
-create models: 2.37 s
-LOG best est: 2.88s
-KNN best est: 23.7s
-SVM best est: 6.59s
-implementing SMOTE: 30.1s
-confusion matrix: 2.54s

full dataset
-create models: 51 mins
-LOG best est: 61 mins
-KNN best est: 8.5 hrs
-SVM best est: 2.36 hrs
-implementing SMOTE: 10.79 hrs
-confusion matrix: 54 mins

X_train and X_test preprocssed with scaler:
-gridsearch takes 32.6 seconds
```

```
2.37s = 51*60s
1s = (51*60)/2.37

"""



# # SMOTE Technique (Over-Sampling)

# copied from: https://www.kaggle.com/janiobachmann/
↪credit-fraud-dealing-with-imbalanced-datasets
#
# SMOTE stands for Synthetic Minority Over-sampling Technique.
#
# <img src=SMOTE_R_visualisation_3.png, width=400>
#
# https://raw.githubusercontent.com/rikunert/SMOTE_visualisation/master/
↪SMOTE_R_visualisation_3.png
#
# Unlike Random UnderSampling, SMOTE creates new synthetic points in order to␣
↪have an equal balance of the classes. This is a way for solving the "class␣
↪imbalance problems".

# In[133]:


from imblearn.over_sampling import SMOTE
from sklearn.model_selection import train_test_split, RandomizedSearchCV

print('Length of X (train): {} | Length of y (train): {}'.
↪format(len(original_Xtrain), len(original_ytrain)))
print('Length of X (test): {} | Length of y (test): {}'.
↪format(len(original_Xtest), len(original_ytest)))

# List to append the score and then find the average
accuracy_lst = []
precision_lst = []
recall_lst = []
f1_lst = []
auc_lst = []


# In[134]:


# Classifier with optimal parameters
# log_reg_sm = grid_log_reg.best_estimator_
log_reg_sm = LogisticRegression()
```

```
rand_log_reg = RandomizedSearchCV(LogisticRegression(), log_reg_params,␣
 ↪n_iter=4)


# In[135]:


# Implementing SMOTE Technique
# Cross Validating the right way
# Parameters
log_reg_params = {"penalty": ['l1', 'l2'], 'C': [0.001, 0.01, 0.1, 1, 10, 100,␣
 ↪1000]}
for train, test in sss.split(original_Xtrain, original_ytrain):
    pipeline = imbalanced_make_pipeline(SMOTE(sampling_strategy='minority'),␣
 ↪rand_log_reg) # SMOTE happens during Cross Validation not before..
    model = pipeline.fit(original_Xtrain[train], original_ytrain[train])
    best_est = rand_log_reg.best_estimator_
    prediction = best_est.predict(original_Xtrain[test])

    accuracy_lst.append(pipeline.score(original_Xtrain[test],␣
 ↪original_ytrain[test]))
    precision_lst.append(precision_score(original_ytrain[test], prediction))
    recall_lst.append(recall_score(original_ytrain[test], prediction))
    f1_lst.append(f1_score(original_ytrain[test], prediction))
    auc_lst.append(roc_auc_score(original_ytrain[test], prediction))

print('---' * 45)
print('')
print("accuracy: {}".format(np.mean(accuracy_lst)))
print("precision: {}".format(np.mean(precision_lst)))
print("recall: {}".format(np.mean(recall_lst)))
print("f1: {}".format(np.mean(f1_lst)))
print('---' * 45)


# In[136]:


labels = ['No Fraud', 'Fraud']
smote_prediction = best_est.predict(original_Xtest)
print(classification_report(original_ytest, smote_prediction,␣
 ↪target_names=labels))


# In[137]:
```

```
y_score = best_est.decision_function(original_Xtest)


# In[138]:


average_precision = average_precision_score(original_ytest, y_score)

print('Average precision-recall score: {0:0.2f}'.format(
      average_precision))


# In[139]:


from sklearn.metrics import precision_recall_curve

fig = plt.figure(figsize=(12,6))

precision, recall, _ = precision_recall_curve(original_ytest, y_score)

plt.step(recall, precision, color='r', alpha=0.2,
         where='post')
plt.fill_between(recall, precision, step='post', alpha=0.2,
                 color='#F59B00')

plt.xlabel('Recall')
plt.ylabel('Precision')
plt.ylim([0.0, 1.05])
plt.xlim([0.0, 1.0])
plt.title('OverSampling Precision-Recall curve: \n Average Precision-Recall␣
 ↪Score ={0:0.2f}'.format(
          average_precision), fontsize=16)


# In[140]:


# SMOTE Technique (OverSampling) After splitting and Cross Validating
sm = SMOTE(ratio='minority', random_state=42)
# Xsm_train, ysm_train = sm.fit_sample(X_train, y_train)


# This will be the data were we are going to
Xsm_train, ysm_train = sm.fit_sample(original_Xtrain, original_ytrain)
```

```
# In[141]:


# We Improve the score by 2% points approximately
# Implement GridSearchCV and the other models.

# Logistic Regression
t0 = time.time()
log_reg_sm = grid_log_reg.best_estimator_
log_reg_sm.fit(Xsm_train, ysm_train)
t1 = time.time()
print("Fitting oversample data took :{} sec".format(t1 - t0))


# In[142]:


from sklearn.metrics import confusion_matrix

# Logistic Regression fitted using SMOTE technique
y_pred_log_reg = log_reg_sm.predict(X_test)

# Other models fitted with UnderSampling
#y_pred_knear = knears_neighbors.predict(X_test)
y_pred_svc = svc.predict(X_test)
y_pred_tree = tree_clf.predict(X_test)

log_reg_cf = confusion_matrix(y_test, y_pred_log_reg)
#kneighbors_cf = confusion_matrix(y_test, y_pred_knear)
svc_cf = confusion_matrix(y_test, y_pred_svc)
tree_cf = confusion_matrix(y_test, y_pred_tree)

fig, ax = plt.subplots(2, 2,figsize=(22,12))


sns.heatmap(log_reg_cf, ax=ax[0][0], annot=True, cmap=plt.cm.copper)
ax[0, 0].set_title("Logistic Regression \n Confusion Matrix", fontsize=14)
ax[0, 0].set_xticklabels(['', ''], fontsize=14, rotation=90)
ax[0, 0].set_yticklabels(['', ''], fontsize=14, rotation=360)

"""sns.heatmap(kneighbors_cf, ax=ax[0][1], annot=True, cmap=plt.cm.copper)
ax[0][1].set_title("KNearsNeighbors \n Confusion Matrix", fontsize=14)
ax[0][1].set_xticklabels(['', ''], fontsize=14, rotation=90)
ax[0][1].set_yticklabels(['', ''], fontsize=14, rotation=360)"""

sns.heatmap(svc_cf, ax=ax[1][0], annot=True, cmap=plt.cm.copper)
ax[1][0].set_title("Suppor Vector Classifier \n Confusion Matrix", fontsize=14)
```

```
ax[1][0].set_xticklabels(['', ''], fontsize=14, rotation=90)
ax[1][0].set_yticklabels(['', ''], fontsize=14, rotation=360)

sns.heatmap(tree_cf, ax=ax[1][1], annot=True, cmap=plt.cm.copper)
ax[1][1].set_title("DecisionTree Classifier \n Confusion Matrix", fontsize=14)
ax[1][1].set_xticklabels(['', ''], fontsize=14, rotation=90)
ax[1][1].set_yticklabels(['', ''], fontsize=14, rotation=360)


plt.show()


# In[143]:


from sklearn.metrics import classification_report


print('Logistic Regression:')
print(classification_report(y_test, y_pred_log_reg))

"""print('KNears Neighbors:')
print(classification_report(y_test, y_pred_knear))"""

print('Support Vector Classifier:')
print(classification_report(y_test, y_pred_svc))

print('Support Vector Classifier:')
print(classification_report(y_test, y_pred_tree))


# In[144]:


get_ipython().system('pip install yellowbrick')
from yellowbrick.classifier import ClassificationReport

visualizer = ClassificationReport(LogisticRegression(), classes=['0', '1'],␣
 ↪support=True)
visualizer.fit(X_train, y_train)  # Fit the visualizer and the model
visualizer.score(X_test, y_test)  # Evaluate the model on the test data
g = visualizer.poof()             # Draw/show/poof the data

visualizer = ClassificationReport(SVC(), classes=['0', '1'], support=True)
visualizer.fit(X_train, y_train)  # Fit the visualizer and the model
visualizer.score(X_test, y_test)  # Evaluate the model on the test data
g = visualizer.poof()             # Draw/show/poof the data
```

```
visualizer = ClassificationReport(DecisionTreeClassifier(), classes=['0', '1'],␣
 ↪support=True)
visualizer.fit(X_train, y_train)  # Fit the visualizer and the model
visualizer.score(X_test, y_test)  # Evaluate the model on the test data
g = visualizer.poof()             # Draw/show/poof the data

visualizer = ClassificationReport(KNeighborsClassifier(), classes=['0', '1'],␣
 ↪support=True)
visualizer.fit(X_train, y_train)  # Fit the visualizer and the model
visualizer.score(X_test, y_test)  # Evaluate the model on the test data
g = visualizer.poof()             # Draw/show/poof the data


# In[145]:


# Final Score in the test set of logistic regression
from sklearn.metrics import accuracy_score

# Logistic Regression with Under-Sampling
y_pred = log_reg.predict(X_test)
undersample_score = accuracy_score(y_test, y_pred)

# Logistic Regression with SMOTE Technique (Better accuracy with SMOTE t)
y_pred_sm = best_est.predict(original_Xtest)
oversample_score = accuracy_score(original_ytest, y_pred_sm)

d = {'Technique': ['Oversampling (SMOTE)'], 'Score': [oversample_score]}
final_df = pd.DataFrame(data=d)

# Move column
score = final_df['Score']
final_df.drop('Score', axis=1, inplace=True)
final_df.insert(1, 'Score', score)

# Note how high is accuracy score it can be misleading!
final_df


# In[154]:


from yellowbrick.classifier import ROCAUC

f, (ax1) = plt.subplots(1, 1, figsize=(12,10))
```

```python
# Instantiate the visualizer with the classification model
visualizer = ROCAUC(LogisticRegression(), classes=['0', '1'])

visualizer.fit(X_train, y_train)  # Fit the training data to the visualizer
visualizer.score(X_test, y_test)  # Evaluate the model on the test data
g = visualizer.poof()             # Draw/show/poof the data


# In[151]:


from yellowbrick.classifier import PrecisionRecallCurve

f, (ax1) = plt.subplots(1, 1, figsize=(12,10))

# Create the visualizer, fit, score, and poof it
viz = PrecisionRecallCurve(LogisticRegression(), classes=['0', '1'])
viz.fit(X_train, y_train)
viz.score(X_test, y_test)
viz.poof()


# In[152]:


from sklearn.linear_model import LogisticRegression
from yellowbrick.classifier import DiscriminationThreshold

f, (ax1) = plt.subplots(1, 1, figsize=(12,10))

# Instantiate the classification model and visualizer
logistic = LogisticRegression()
visualizer = DiscriminationThreshold(logistic)

visualizer.fit(X_train, y_train)  # Fit the training data to the visualizer
visualizer.poof()     # Draw/show/poof the data


# In[ ]:
```

Untitled

Dean Hou, Neel Iyer, Serena Xu

11/08/2019

```
library(data.table)
library(tidyverse)
```

```
## Registered S3 methods overwritten by 'ggplot2':
##   method         from
##   [.quosures     rlang
##   c.quosures     rlang
##   print.quosures rlang
```

```
## -- Attaching packages ------------------------------------------------------------------------------
```

```
## v ggplot2 3.1.1     v purrr   0.3.2
## v tibble  2.1.1     v dplyr   0.8.2
## v tidyr   0.8.3     v stringr 1.4.0
## v readr   1.3.1     v forcats 0.4.0
```

```
## -- Conflicts ---------------------------------------------------------------------------------------
## x dplyr::between()   masks data.table::between()
## x dplyr::filter()    masks stats::filter()
## x dplyr::first()     masks data.table::first()
## x dplyr::lag()       masks stats::lag()
## x dplyr::last()      masks data.table::last()
## x purrr::transpose() masks data.table::transpose()
```

```
data <- read_csv("SandBox/dataframe_filled_na.csv")
```

```
## Parsed with column specification:
## cols(
##   .default = col_double()
## )
```

```
## See spec(...) for full column specifications.
```

```
# Undersampling
library(unbalanced)
```

```
## Warning: package 'unbalanced' was built under R version 3.6.1
```

```
## Loading required package: mlr
```

```
## Warning: package 'mlr' was built under R version 3.6.1
```

```
## Loading required package: ParamHelpers
## Warning: package 'ParamHelpers' was built under R version 3.6.1
```

```
## Loading required package: foreach
```

```
## Warning: package 'foreach' was built under R version 3.6.1
```

```
##
## Attaching package: 'foreach'
```

```
## The following objects are masked from 'package:purrr':
##
##     accumulate, when
```

```
## Loading required package: doParallel
```

```
## Warning: package 'doParallel' was built under R version 3.6.1
```

```
## Loading required package: iterators
```

```
## Warning: package 'iterators' was built under R version 3.6.1
```

```
## Loading required package: parallel
```

```r
n <- ncol(data)
data$fraud <- as.factor(data$fraud)
output <- data$fraud
input <- data %>% select(1, 3:27)
ubData <- ubBalance(X = input, Y = output, type = "ubUnder")
newData <- cbind(ubData$X, ubData$Y)
setnames(newData, "ubData$Y", "fraud")
dataset <- newData


# Normalization
normalize <- function(x) {
  return ((x - min(x)) / (max(x) - min(x)))
}
norm <- as.data.frame(lapply(dataset[,1:26], normalize))


# KNN Classification
# Splitting the data set into train and test
set.seed(2)
part <- sample(2, nrow(norm), replace = TRUE, prob = c(0.7, 0.3))
train <- norm[part == 1,]
test <- norm[part == 2,]

# Get the target variable (fraud) which is not included in our train / test set
train_labels <- dataset$fraud[part == 1]
test_labels <- dataset$fraud[part == 2]


# Training the KNN model
library(class)
```

```
## Warning: package 'class' was built under R version 3.6.1
```

```
test_pred_9 <- knn(train = train, test = test, cl = train_labels, k=9)
test_pred_99 <- knn(train = train, test = test, cl = train_labels, k=99)



# Evaluating the model performance
library(gmodels)
```

```
## Warning: package 'gmodels' was built under R version 3.6.1
```

```
accuracy <- function(x){sum(diag(x)/(sum(rowSums(x)))) * 100}


CrossTable(x = test_labels, y = test_pred_9, prop.chisq=FALSE)
```

```
##
##
##    Cell Contents
## |-------------------------|
## |                       N |
## |           N / Row Total |
## |           N / Col Total |
## |         N / Table Total |
## |-------------------------|
##
##
## Total Observations in Table:  6349
##
##
##               | test_pred_9
##   test_labels |         0 |         1 | Row Total |
## -------------|-----------|-----------|-----------|
##             0 |      1716 |      1498 |      3214 |
##               |     0.534 |     0.466 |     0.506 |
##               |     0.785 |     0.360 |           |
##               |     0.270 |     0.236 |           |
## -------------|-----------|-----------|-----------|
##             1 |       470 |      2665 |      3135 |
##               |     0.150 |     0.850 |     0.494 |
##               |     0.215 |     0.640 |           |
##               |     0.074 |     0.420 |           |
## -------------|-----------|-----------|-----------|
## Column Total |      2186 |      4163 |      6349 |
##               |     0.344 |     0.656 |           |
## -------------|-----------|-----------|-----------|
##
##
```

```
tb_9 <- table(test_pred_9,test_labels)
tb_9
```

```
##            test_labels
## test_pred_9    0    1
##           0 1716  470
##           1 1498 2665
```

```
accuracy(tb_9)
```

```
## [1] 69.00299
```

```
CrossTable(x = test_labels, y = test_pred_99, prop.chisq=FALSE)
```

```
## 
## 
##    Cell Contents
## |-------------------------|
## |                       N |
## |           N / Row Total |
## |           N / Col Total |
## |         N / Table Total |
## |-------------------------|
## 
## 
## Total Observations in Table:  6349 
## 
## 
##              | test_pred_99 
##  test_labels |         0 |         1 | Row Total | 
## -------------|-----------|-----------|-----------|
##            0 |      1167 |      2047 |      3214 | 
##              |     0.363 |     0.637 |     0.506 | 
##              |     0.920 |     0.403 |           | 
##              |     0.184 |     0.322 |           | 
## -------------|-----------|-----------|-----------|
##            1 |       101 |      3034 |      3135 | 
##              |     0.032 |     0.968 |     0.494 | 
##              |     0.080 |     0.597 |           | 
##              |     0.016 |     0.478 |           | 
## -------------|-----------|-----------|-----------|
## Column Total |      1268 |      5081 |      6349 | 
##              |     0.200 |     0.800 |           | 
## -------------|-----------|-----------|-----------|
## 
## 
```

Undersampling

August 11, 2019

[ ]:

```
# -*- coding: utf-8 -*-
"""DATA3001_EDA_Undersampling_dean.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1XQ6WXQ61AVFth4-NHt7HZU-eyfX2q6BL

# DATA3001 Yuumi insurance EDA Undersampling

The goal of this exploratory data analysis is to explore the cleaned data and␣
 ↪use some models to predict insurance fraud.
"""

## visualisation packages
# %matplotlib inline
import pandas as pd
import numpy as np
from sklearn import preprocessing
import matplotlib.pyplot as plt
plt.rc("font", size=14)

import seaborn as sns
sns.set(style="white")
sns.set(style="whitegrid", color_codes=True)

# make column width bigger when displaying columns
pd.set_option("display.max_colwidth", 1000)
# display more columns
pd.set_option("display.max_columns", None)
# format float display
pd.options.display.float_format = '{:.2f}'.format

## modelling pacakges
from sklearn.preprocessing import StandardScaler, RobustScaler

from sklearn.model_selection import train_test_split
from sklearn.model_selection import StratifiedShuffleSplit
from sklearn.model_selection import KFold, StratifiedKFold
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import cross_val_predict
from sklearn.model_selection import ShuffleSplit
from sklearn.model_selection import learning_curve
from scipy.stats import norm

from sklearn.linear_model import LogisticRegression
from sklearn.dummy import DummyClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
from xgboost import XGBClassifier
from imblearn.under_sampling import NearMiss

from sklearn.metrics import␣
 ↪accuracy_score,confusion_matrix,f1_score,recall_score,precision_recall_curve,average_precis
from sklearn.metrics import roc_auc_score
from sklearn.metrics import roc_curve
from sklearn.metrics import auc
from sklearn.metrics import balanced_accuracy_score

from sklearn.utils import resample

"""## Initial Data"""

log = pd.read_csv('Dataset/data.csv')
log.head()

"""From the initial data we can see that the timestamp is sorted in monotonic␣
 ↪order (increasing) which is expected. Each message has a transaction id, a␣
 ↪device which the transaction is performed on, a customer id, an action made␣
 ↪by the customer,

Assumptions made here - each customer has to start a quote and either complete␣
 ↪the quote or have the quote be incompleted

When cleaning the data we noticed that:

- the customer ids would be reused in some cases
- sometimes the quote started action was not the first action made by the␣
 ↪customer
```

```
However using customer id with quote started action yielded the best results␣
 ↪for the re-iding of transactions.

The goal of the data cleaning was to transform the data into a tabular format␣
 ↪and make it more readable. This involved splitting the message string into␣
 ↪columns using methods such as string split in Python and Regex.

## Cleaned Data

This is after data preprocessing and cleaning
"""

data = pd.read_csv('Dataset/df_unfilled_nan_extra_cols.csv')
data.head()

data.dtypes

for i in data.columns:
    print(i, end = ', ')

print(len(data), len(data.columns))
print(data.isnull().sum().max())

"""## Imputation

Currently on fills with 0, to improve models we need better imputation of data,␣
 ↪which means intuitively fill in Na's
"""

def fill_na(dataframe, list_of_cols):

    for col in list_of_cols:
        dataframe[col] = dataframe[col].fillna(0)

    return dataframe

data = fill_na(data, ['paid_min', 'paid_max', 'paid_mean',␣
 ↪'house_square_footage', 'house_number_of_bedrooms',␣
 ↪'house_number_of_floors', 'claim_pay_diff', 'paid_diff',␣
 ↪'occupant_error_in_gender'])

# heavily skewed towards non fraud
print('No Frauds', data['fraud'].value_counts()[0], round(data['fraud'].
 ↪value_counts()[0]/len(data) * 100,2), '% of the dataset')
print('Frauds', data['fraud'].value_counts()[1], round(data['fraud'].
 ↪value_counts()[1]/len(data) * 100,2), '% of the dataset')
```

```
colors = ["#0101DF", "#DF0101"]

sns.countplot('fraud', data = data, palette=colors)
plt.title('Class Distributions \n (0: No Fraud || 1: Fraud)', fontsize=14)

"""## imbalanced data

As we can see the data is imbalanced, 98% of the data is non-fraud, while 2% of the data is fraud. Thus setting all the values in the data to 0 will yield high score, but this will lead to a large amount of false negatives. (below)

Steps:
1. separate data into test and training (train on undersampled data)
2. run models on data, using cross validation

## first testing results with imbalanced data and bad accuracy measure
"""

# Separate input features and target
y = data.fraud
X = data.drop('fraud', axis=1)

# setting up testing and training sets, splitting by 20/80 testing/training
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=27)

"""Dummy classifier (sets everything to 0), using default accuracy score"""

# DummyClassifier to predict only target 0
dummy = DummyClassifier(strategy='most_frequent').fit(X_train, y_train)
dummy_pred = dummy.predict(X_test)

# checking unique labels
print('Unique predicted labels: ', (np.unique(dummy_pred)))

# checking accuracy
print('Test score: ', accuracy_score(y_test, dummy_pred))

# Checking unique values
predictionsDummy = pd.DataFrame(dummy_pred)
predictionsDummy[0].value_counts()

# scoring 0.98344

"""Now comparing this to a logistic regression, an actual trained classifier"""
```

```
# Modeling the data as is
# Train model
lr = LogisticRegression(solver='liblinear').fit(X_train, y_train)

# Predict on training set
lr_pred = lr.predict(X_test)

# Checking accuracy
print(accuracy_score(y_test, lr_pred))
#0.98346

# Checking unique values
predictions = pd.DataFrame(lr_pred)
predictions[0].value_counts()

# 5 frauds
# note we also get a convergance warning

"""It seems to have performed the same though noticably the Logistic regression␣
 ↪has 5 frauds detected, we can see that this method of training (just␣
 ↪splitting data) is not a good method, we need some way to balance the␣
 ↪training data in order for the model to capture features that can help␣
 ↪define observations as fraud.

We also see that the accuracy_score metric is not a good measure as it only␣
 ↪does a fraction of correct predictions thus explaining the all 0 result.
https://scikit-learn.org/stable/modules/model_evaluation.html accuracy_score

Better accuracy scores are F1 score, recall score, and AUC score.
"""

# using sklearn class_weight balanced parameter to try to balance the imbalance␣
 ↪in data
# also applying the better accuracy scores F1 score, recall score, AUC score
logistic = LogisticRegression(class_weight='balanced')
model = logistic.fit(X_train, y_train)

logisticPred = model.predict(X_test)

print(accuracy_score(y_test, logisticPred))

predictionslog = pd.DataFrame(logisticPred)
predictionslog[0].value_counts()
# accuracy = 0.977796
# fraud 4905

# train model
```

```
rfc = RandomForestClassifier(class_weight='balanced', n_estimators=10).
 ↪fit(X_train, y_train)

# predict on test set
rfc_pred = rfc.predict(X_test)

print(accuracy_score(y_test, rfc_pred))

predictionsRFC = pd.DataFrame(rfc_pred)
predictionsRFC[0].value_counts()

"""## Undersampling: code based on link below
https://www.kaggle.com/nitinchan/fraud-detection-for-credit-card-imbalance-data
"""

data1 = data.copy()

# Separate input features and target
Y = data1.fraud
X = data1.drop(['fraud'], axis=1)

# setting up testing and training sets
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.3,␣
 ↪random_state=2727)

# concatenate our training data back together
X = pd.concat([X_train, Y_train], axis=1)

# separate minority and majority classes
not_fraud = data1[data1.fraud==0]
fraud = data1[data1.fraud==1]

len(fraud)

not_fraud_downsampled = resample(not_fraud,
                                replace = False, # sample without replacement
                                n_samples = len(fraud), # match minority n
                                random_state = 27) # reproducible results

# combine minority and downsampled majority
downsampled = pd.concat([not_fraud_downsampled, fraud])

# checking counts
downsampled.fraud.value_counts()

print(len(X_test), len(Y_test))
```

```
Y_test.value_counts()

# trying logistic regression with the balanced dataset
y_train = downsampled.fraud
X_train = downsampled.drop('fraud', axis=1)

underSampledLog = LogisticRegression(solver='liblinear')
underSampledLog.fit(X_train, y_train)

log_reg_pred = cross_val_predict(underSampledLog, X_train, y_train, cv=5,
                             method="decision_function")

print('Logistic Regression: ', roc_auc_score(y_train, log_reg_pred))

# Predict on test
undersampled_pred = log_reg_pred.predict(X_test)
# predict probabilities
probs = log_reg_pred.predict_proba(X_test)
# keep probabilities for the positive outcome only
probs = probs[:, 1]

# Checking accuracy
accuracy = accuracy_score(Y_test, undersampled_pred)
print("Test Accuracy is {:.2f}%".format(accuracy * 100.0))

# f1 score
f1_under = f1_score(Y_test, undersampled_pred)
print("F1 Score is {:.2f}".format(f1_under))

b_score = balanced_accuracy_score(Y_test, undersampled_pred)
b_score

# calculate precision-recall curve
precision, recall, thresholds = precision_recall_curve(Y_test, probs)
# calculate precision-recall AUC
auc_under = auc(recall, precision)
# plot no skill
plt.plot([0, 1], [0.5, 0.5], linestyle='--')
# plot the precision-recall curve for the model
plt.plot(recall, precision, marker='.')
plt.title("Precison-Recall Curve for Logistic with AUC score: {:.3f}".
 ↪format(auc_under))
# show the plot
plt.show()

from sklearn.metrics import precision_recall_curve
log_fpr, log_tpr, log_thresold = roc_curve(y_train, log_reg_pred)
```

```
#l_precision, l_recall, l_threshold = precision_recall_curve(, )

def graph_roc_curve(log_fpr, log_tpr):
    plt.figure(figsize=(16,8))
    plt.title('ROC Curve \n Top 4 Classifiers', fontsize=18)
    plt.plot(log_fpr, log_tpr, label='Logistic Regression Classifier Score: {:.
 ↪4f}'.format(roc_auc_score(y_train, log_reg_pred)))
    plt.plot([0, 1], [0, 1], 'k--')
    plt.axis([-0.01, 1, 0, 1])
    plt.xlabel('False Positive Rate', fontsize=16)
    plt.ylabel('True Positive Rate', fontsize=16)
    plt.annotate('Minimum ROC Score of 50% \n (This is the minimum score to␣
 ↪get)', xy=(0.5, 0.5), xytext=(0.6, 0.3),
                arrowprops=dict(facecolor='#6E726D', shrink=0.05),
                )
    plt.legend()

graph_roc_curve(log_fpr, log_tpr)
plt.show()

# assign cnf_matrix with result of confusion_matrix array
cnf_matrix = confusion_matrix(Y_test,undersampled_pred)
#create a heat map
sns.heatmap(pd.DataFrame(cnf_matrix), annot = True, cmap = 'Blues', fmt = 'd')
plt.xlabel('Predicted')
plt.ylabel('Expected')
plt.show()

"""## we see that while the undersampled logistic regression is good for␣
 ↪insample training, out of sample it performs relatively poorly as it detects␣
 ↪a lot of false positives"""
```