Udacity Machine Learning Nanodegree Capstone Project

Santander Customer Transaction Prediction

Po-sheng Wang

May 2019

I. Definition

Project Overview:

Santander Customer Transaction Prediction is a project to predict the users' transaction behavior in Banco Santander bank. This project is going to identify which customers will make a specific transaction in the future. The dataset is obtained from Kaggle (https://www.kaggle.com/). It is an anonymized dataset containing numeric feature variables and the binary target. This dataset provides 200000 data with 200 features for each data. Instead of specifying the real meaning of each feature, the dataset only provides the alias name for the features (var_1, var_2, var_3…var_200). There are 179902 data for label 0 and 20098 data or label 1. The figure below shows the example of this dataset:

target	var_0	var_1	var_2	var_3	var_4	var_5	var_6	var_7	...	var_190	var_191	var_192	var_193	var_194	var_195	var_196
1	12.7834	-9.2966	9.5248	4.6642	11.0086	13.6108	5.9264	12.3231	...	-3.1637	6.1416	3.3087	0.4636	23.4027	-2.5640	2.9615
1	11.0450	2.4999	12.8643	8.5602	11.9941	-16.9360	5.7255	12.5728	...	-3.1366	9.5150	-1.0431	0.2986	18.3893	-1.2815	4.5883
0	7.2982	1.0395	13.9573	9.8031	11.1019	3.5735	4.8737	15.1203	...	8.3652	6.1050	2.0829	-0.1308	19.4244	-1.4725	-4.6625

Machine Learning techniques have been used widely on the financial field in terms of helping people understand their financial health and identify which products and services might help them achieve monetary goals. The prediction and classification algorithms could assist the financial organization in providing the proper solutions to the customers. Some related research in this field have been done on [1][2][3][4].

Problem Statement:

The objective was to forecast which clients would make a specific transfer in the future based on the historical data we have. This is a binary prediction. The data given was composed of 200000 rows and 200 anonymized features. The evaluation metric was ROC-AUC.

The most common solution to such problem is the supervised binary classification. Some of the possible algorithms are SVM with different kernels, Neural Network, Random Forest, and Gradient Boosting, which cover both linear and nonlinear classification. The Gradient Boosting is a machine learning algorithm for regression and classification, which produces a prediction model in the form of an ensemble of weak prediction models, typically decision trees.

This project will follow the below tasks to solve this problem:

Tasks and Strategies:

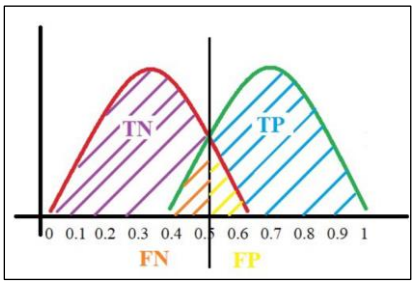
a. Data Exploration: In this stage, it is also called EDA (Exploratory Data Analysis), which means analytically exploring data for finding some hidden information of the data or simply getting more insights on the characteristic of the data. This can be done by visualization of the data.
b. Statistical Tests: Performing some statistical tests to confirm some of our hypotheses and get better intuition on how do we process the data for the next step. We will understand the scale, size, the distribution of the data from quantitative perspective.

c. Data Preprocessing: In most cases, we have to preprocess the dataset before constructing features and feeding into the classification algorithm. Such as outlier detection, encoding categorical variables if necessary, normalizing the dataset.
d. Feature Engineering: Once we have finalized our features in terms of the reliability from data preprocessing, this step is to work on the feature quality. We might use some feature selection or feature extraction algorithm to find the most significant features.
e. Model Selection: Try out different machine learning model to see which one works the best. The Gradient Boost, Random Forest, SVM and Neural Network are the ones I would like to try first.
f. Model Training: For each model, there are some hyper parameters we need to finalize by cross validation technique to build the training model.
g. Testing the model: Testing the model with testing dataset to ensure the quality of the model.

Metrics:

The classification result will be evaluated in ROC Curve. It is a good way to visualize the performance of binary classifier. The basic idea is to use true positive (TP), true negative (TN), false positive (FP) and false negative (FN). The definition is shown below:

		Actual	
		Positives (1)	Negatives (0)
Predicted	Positives (1)	TP	FP
	Negatives (0)	FN	TN



ROC Curve tells us about how good the model can distinguish between two classes. The better models can accurately distinguish between the two. The prediction probability distribution with the cut-off (threshold) is a visualization way to look at those four components. (Image source: https://medium.com/greyatom/lets-learn-about-auc-roc-curve-4a94b4d88152)

Sensitivity (Recall): True positive rate: TP / (TP + FN)

		Actual	
		Positives (1)	Negatives (0)
Predicted	Positives (1)	TP	FP
	Negatives (0)	FN	TN

Specificity: True negative rate: TN / (TN + FP)

		Actual	
		Positives (1)	Negatives (0)
Predicted	Positives (1)	TP	FP
	Negatives (0)	FN	TN

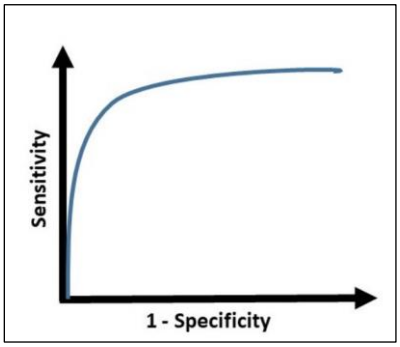
(1 - Specificity): False positive rate: FP / (TN + FP)

		Actual	
		Positives (1)	Negatives (0)
Predicted	Positives (1)	TP	FP
	Negatives (0)	FN	TN

Precision: TP / (TP + FP)

		Actual	
		Positives (1)	Negatives (0)
Predicted	Positives (1)	TP	FP
	Negatives (0)	FN	TN

The trade-off between Sensitivity and Specificity: When the threshold is decreased, the sensitivity will increase while the specificity will decrease. When the threshold is increased the sensitivity will decrease while the specificity will increase. The ROC curve uses (1-specificity) as x axis and sensitivity as y axis for every possible classification threshold. If the classifier is doing a very good job separating the two classes, the ROC curve will be closer to a square shape. If the classifier is doing a poor job separating the two classes, the ROC curve will be close to the diagonal line. As we see above, Specificity gives us the True Negative Rate and (1 - Specificity) gives us the False Positive Rate. By using (1 – specificity), we are just looking at the positives. Thus, AUC ROC indicates how well the probabilities from the positive classes are separated from the negative classes.



AUC, or Are Under Curve, is a metric for binary classification. Accuracy deals with ones and zeros, meaning you either got the class label right or you didn't. But many classifiers are able to quantify their uncertainty about the answer by outputting a probability value. To compute accuracy from probabilities you need a threshold to decide when zero turns into one. The most natural threshold is of course 0.5. That's the whole point of using AUC - it considers all possible thresholds. Various thresholds result in different true positive/false positive rates. As you decrease the threshold, you get more true positives, but also more false positives.

II. Analysis

Data Exploration and Exploratory Visualization:

The dataset of this project has the same structures as the real data and it is provided by the Banco Santander bank. This dataset is an anonymized dataset containing 200 numeric features variables.

- Data description:

The figure below shows the data description of the dataset with the first few features. We can use the comparation of mean value and the median value of each column, or if there is a large difference between 75th% and max values of each columns to see if there is extreme values-outliers in the data set.

	target	var_0	var_1	var_2	var_3	var_4	var_5
count	200000.000000	200000.000000	200000.000000	200000.000000	200000.000000	200000.000000	200000.000000
mean	0.100490	10.679914	-1.627622	10.715192	6.796529	11.078333	-5.065317
std	0.300653	3.040051	4.050044	2.640894	2.043319	1.623150	7.863267
min	0.000000	0.408400	-15.043400	2.117100	-0.040200	5.074800	-32.562600
25%	0.000000	8.453850	-4.740025	8.722475	5.254075	9.883175	-11.200350
50%	0.000000	10.524750	-1.608050	10.580000	6.825000	11.108250	-4.833150
75%	0.000000	12.758200	1.358625	12.516700	8.324100	12.261125	0.924800
max	1.000000	20.315000	10.376800	19.353000	13.188300	16.671400	17.251600

- Data missing:

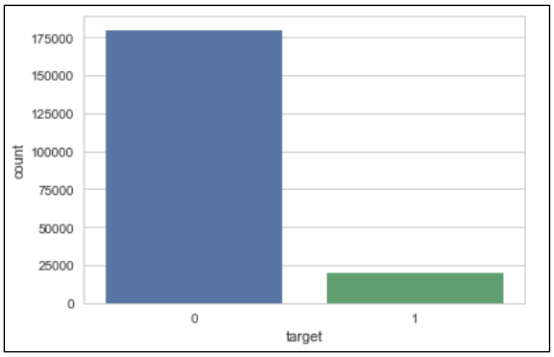
The code below shows that there is no missing data in both training and testing dataset.

```
# check missing data
print("Do we have missing data on the dataset: ", data.isnull().values.any())
```

```
Do we have missing data on the dataset:  False
```

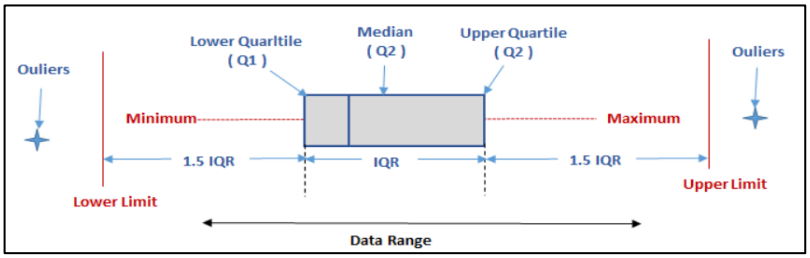
- Check data balance:

The dataset is not balanced according to the target. Around 11.17% of the data is label 1 while 88.83% of the data is label 0.

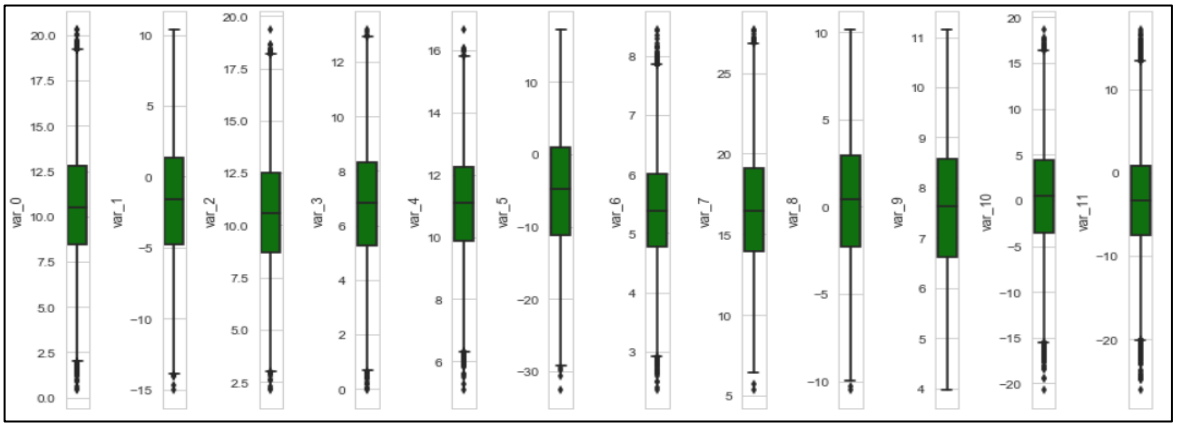


- Data outliers:

An outlier is a data point in a data set that is distant from all other observations. A data point that lies outside the overall distribution of the dataset. The IQR -interquartile range is a standard way to identify any outliers. The definition of outliers:



In this case, for each feature from feature perspective, we can use this IQR-interquartile to identify which data are considered as outliers. The first figure below tells us that from feature 0 to feature 11, only feature 9 does not have outlier data. The second figure shows which exact data is consider as outlier for each feature. Outlier information is useful when performing data preprocessing.



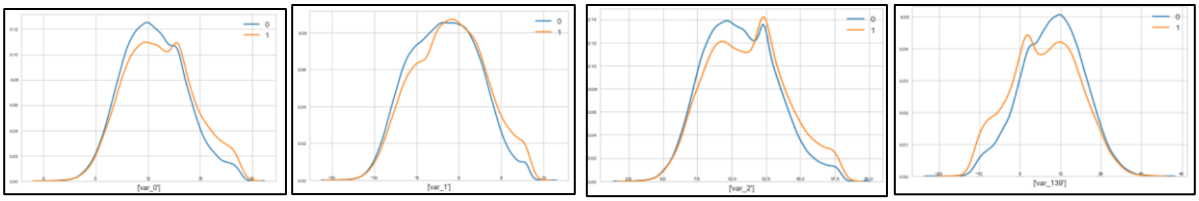
Data points considered outliers for the feature 'var_1':

	ID_code	target	var_0	var_1	var_2	var_3	var_4	var_5	var_6	var_7	...	var_190	var_191	var_192	var_193	var_194	var_195	var_196
17045	train_17045	0	9.9448	-13.9609	9.4189	8.9022	9.9335	2.8718	4.6034	13.0116	...	0.6677	8.8638	1.5104	3.9933	15.4753	-1.9068	4.7601
53322	train_53322	0	9.0616	-14.0325	8.4497	4.5893	13.2734	-16.7324	4.4534	17.0229	...	4.8753	13.0156	1.1313	-0.5577	17.8960	1.2476	2.4324
77939	train_77939	1	9.9115	-14.0370	14.5881	7.8856	13.0283	0.5511	5.9337	13.5277	...	8.6899	6.0020	2.4300	1.1156	18.6356	0.4159	0.9720
128527	train_128527	0	5.8861	-15.0434	6.1810	4.6607	10.5459	-9.9246	4.5121	10.0901	...	4.2344	8.2169	1.2045	6.9338	12.5805	-1.3013	5.8491
130849	train_130849	0	9.5796	-14.6962	9.9887	8.1516	14.1697	-7.6733	3.8098	21.0957	...	-0.0261	5.2231	-0.4406	10.8688	17.6939	0.4001	8.7648
188513	train_188513	0	9.3187	-14.0910	4.7517	7.7008	10.5176	-11.2095	4.7818	20.6821	...	4.3079	3.8301	1.3643	-3.8955	18.5870	1.2193	2.4073

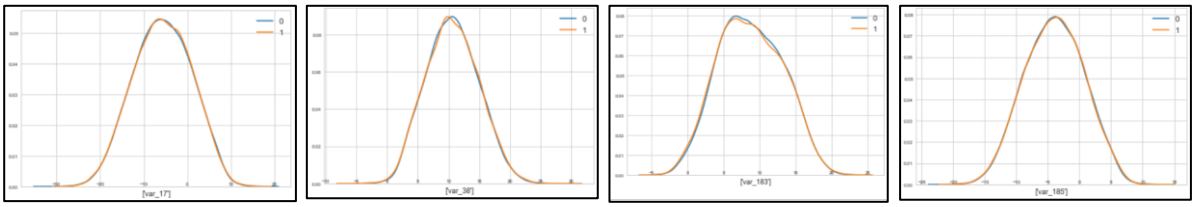
6 rows × 202 columns

- Data feature distributions:

Plot the density plot of feature variables in the dataset and group by target label. We can observe that there are some features with significant different distribution for the two target classes (var_0, var_1, var_2, var_139).



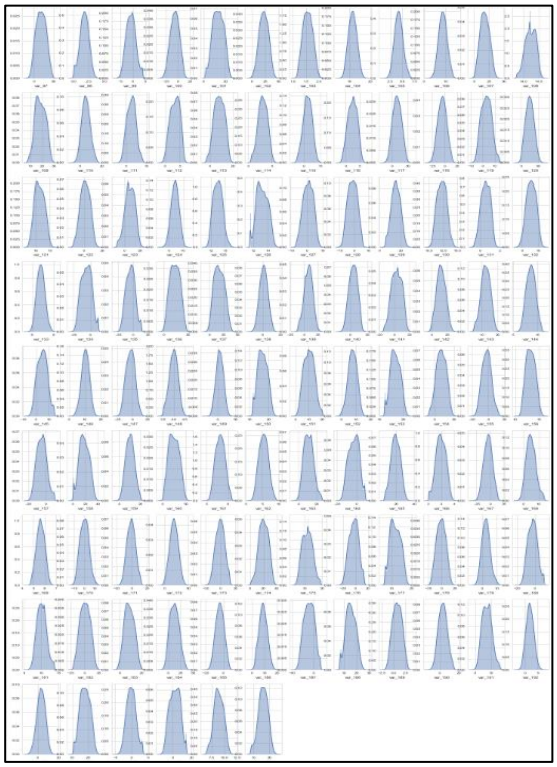
There are also some features with almost the same distribution for the two target classes (var_17, var_38, var_183, var_185)

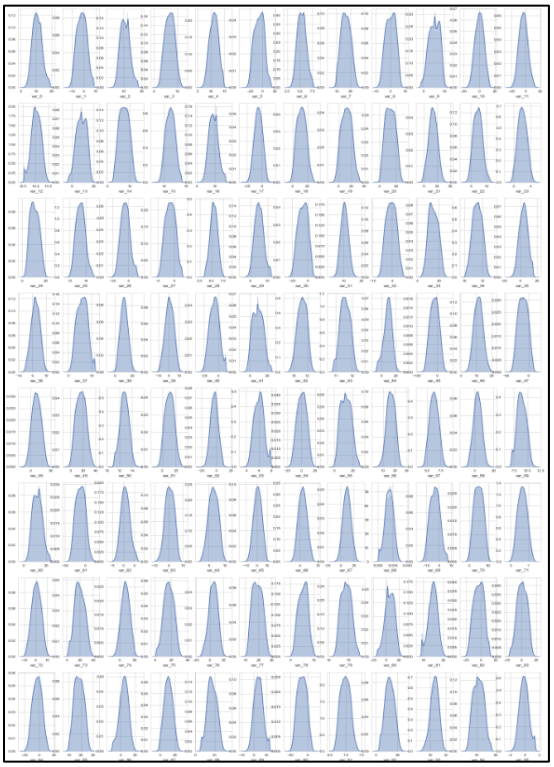


This feature distribution information is important for the feature selection step later for the prediction model.

- Data feature distributions - skewness:

Skewness is a measure of the asymmetry of the probability distribution, in which the curve appears distorted or skewed either to the left or to the right. A data transformation (Logarithms) may be used to reduce skewness. A data transformation may be used to reduce skewness. From the figure below, it looks like the data appears to be normally distributed without too obvious right/positively skewed.

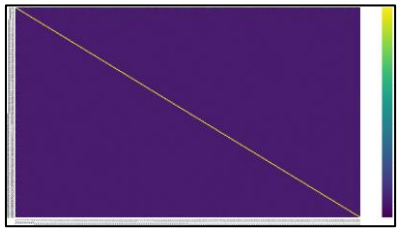




- Feature Correlation:

The figure below shows the first five features pair that have the most correlation and the least correlation. It is shown that even the most correlated features var_165 and var_81 scores only 0.009714. The heatmap figure also tells us the correlation between the features are very small.

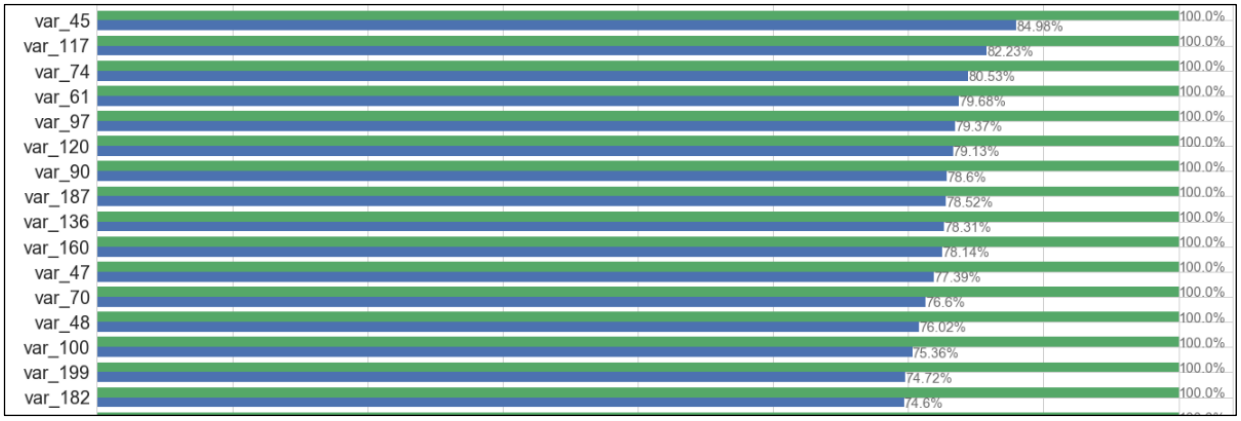
	level_0	level_1	0
39795	var_165	var_81	0.009714
39796	var_53	var_148	0.009788
39797	var_148	var_53	0.009788
39798	var_26	var_139	0.009844
39799	var_139	var_26	0.009844

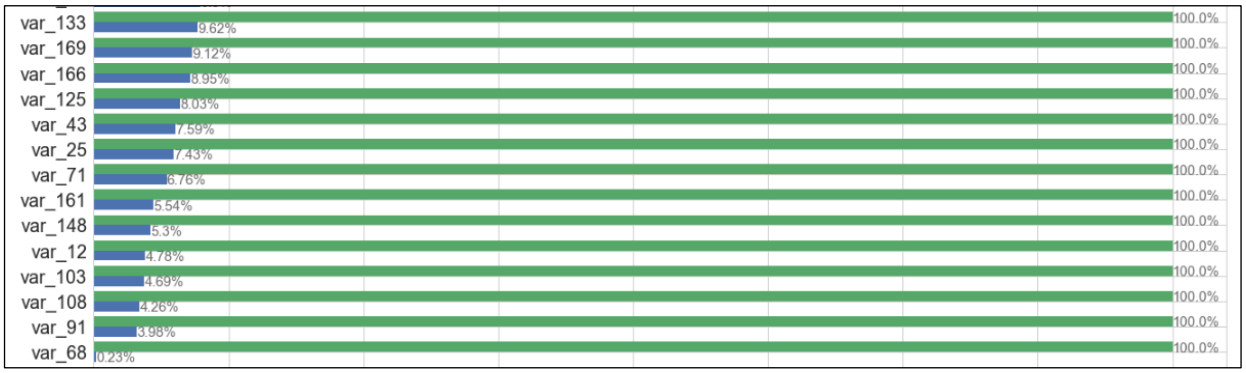


	level_0	level_1	0
0	var_75	var_191	2.703975e-08
1	var_191	var_75	2.703975e-08
2	var_173	var_6	5.942735e-08
3	var_6	var_173	5.942735e-08
4	var_126	var_109	1.313947e-07

- Value duplication and uniqueness:

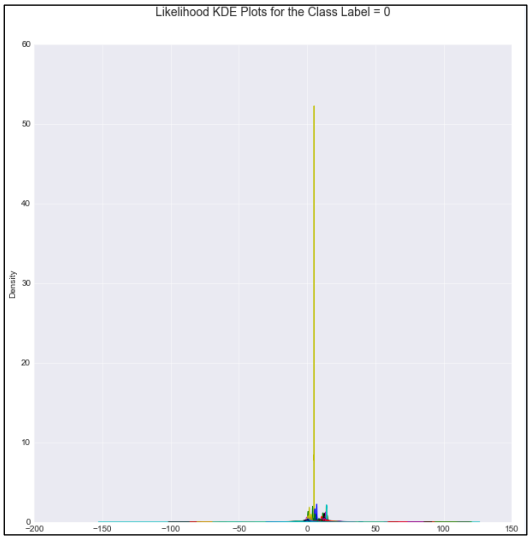
Since the features are all encrypted with code (var_0, var_1…) instead of real feature name, I don't have sense on what do those data values mean. I have a doubt that are they categorical or continuous data, or even mixed. One way that could help me figure out is to count the appearance frequency of each values on each feature to look at the duplication and the uniqueness of the data. The blue bar and the percentage shows how many unique values within that particular feature. Take var_45 as example, which contains the most unique values among all the features. There are 84.98% of the values in var_45 column is unique. For var_68, there is only 0.23% of the values in var_68 is unique, which also means that 99.77% of the values for var_68 are duplicated.

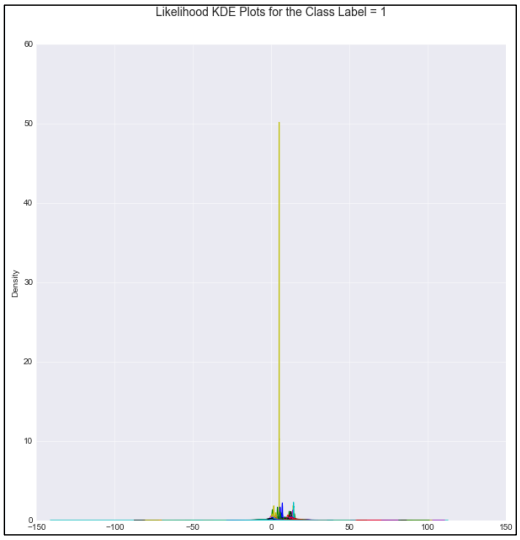




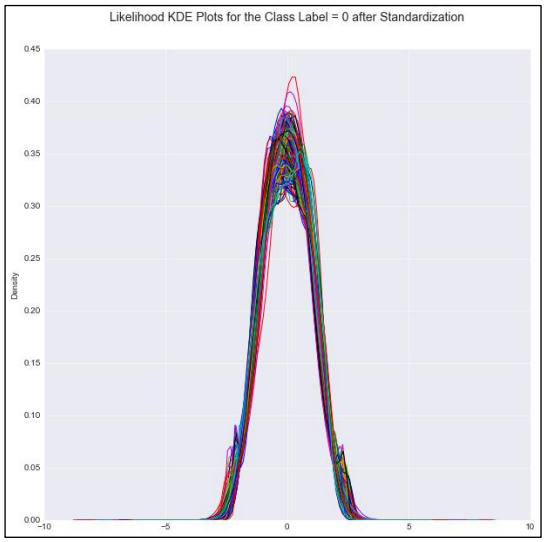
- Feature likelihood distributions for all features:

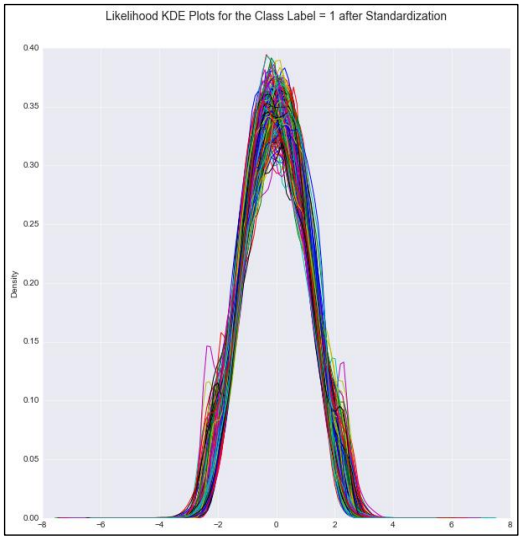
To look at the data from different classes perspective, we plot all the feature distributions for label 0 and label 1.



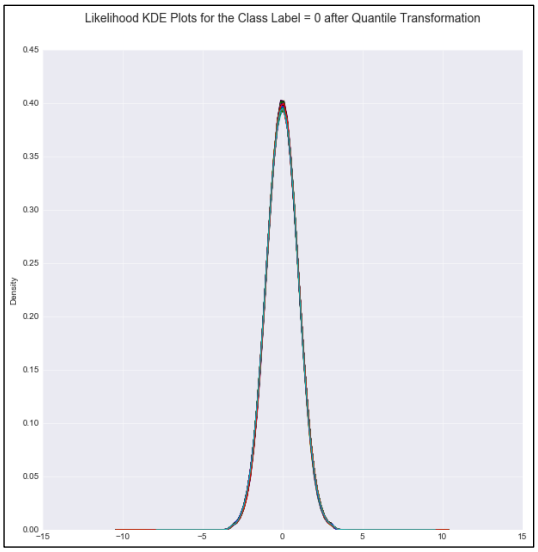


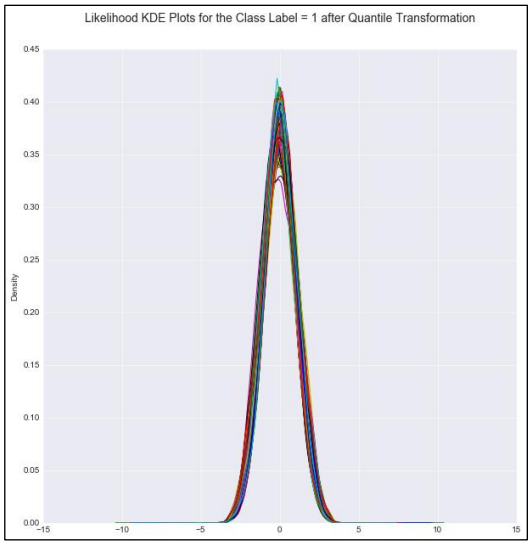
The Kernel Density Estimation (KDE) plots above show that the likelihood distributions have different centers and spread for different features. We can standardize them by subtracting mean and divide by standard deviation so that they have zero mean and unit variance, which result in the plot below.



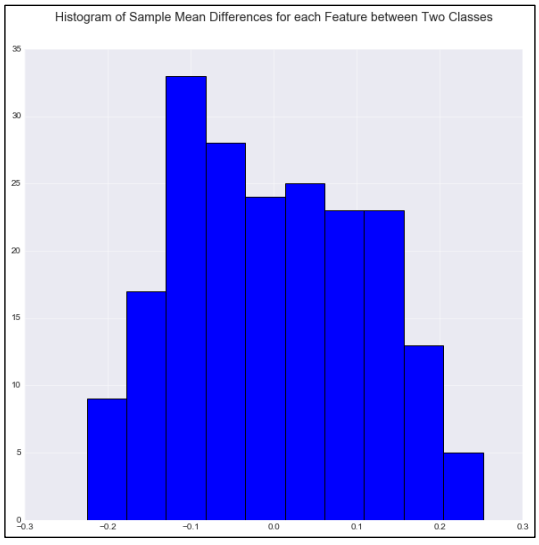


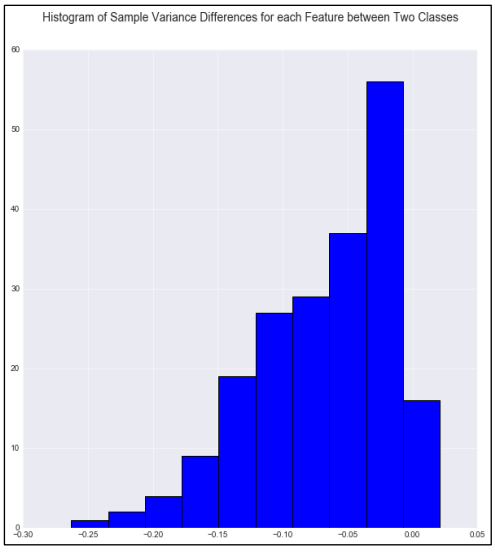
The KDE plots look approximately normal distribution expect that there are some small bumps on the left and right. One approach to remove those small bumps is to apply quantile transformation. After the transformation, the likelihood distribution looks more like normal standard normal distribution.





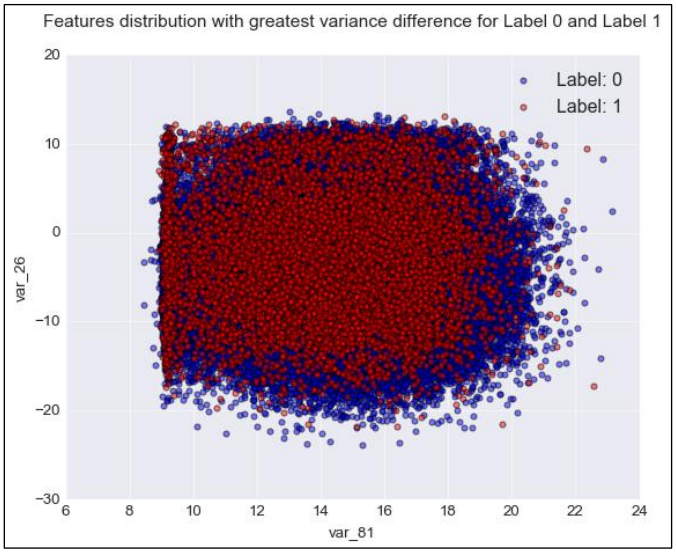
From all the plots above, it's clear to see that label 0 and label 1 behaves differently in terms of distributions, and it's important to know how are the features set different for label 0 and label 1. We plot the sample means and sample variance differ below. The histograms tell us that the sample mean difference are more or less balances around zero. However, the sample variance differences are almost appearing on the negative side. This implies that the label 0 likelihood distribution has less variance, which means that they are more concentrated around their mean value than the label 1 data. This is a good indicator to tell the model that which kind of features belong to which target class. It also implies that the further away the centers of the distributions or the greater the difference in the spread of the distributions, the more it can tell about which class the point is coming from.





In other words, if there are features Xi such that the likelihood distributions are equal for different classes, then their densities will cancel in the numerator and the denominator. These features do not help in classification. I choose two features with greatest sample variance difference and plot the scatter figure

below. The figure shows that the label 0 are more concentrated in a circle while the label 1 points are having a weird boundary concentrated line on the lower and left side.



Algorithms and Techniques:

A Decision Tree can be used both on classification and regression. The nodes in the tree-like model denotes a test on an attribute and each branch represents an outcome of the test, then each leaf node (terminal node) holds a class label. It can predict whether a particular variable would have mattered in the customer's decision to make a certain transaction or not in the future. Since one single tree might be bias, the ensemble methodology is used to improve the result. When considering ensemble learning, there are some primary methods: bagging, boosting, stacking and blending.

Ensemble:

- Use similar classifiers and to combine them together:
  - Bagging: Random Forest
  - Boosting: Adaboost, Gradient Boosting, XGBoost, Light GBM, XGBoost, CatBoost
- Use different classifiers and to stack them together
  - Stacking
  - Blending

- Model algorithm: Light GBM

LightGBM is a gradient boosting framework that uses tree based learning algorithm. It is designed to be faster training speed, better accuracy and able to handle large-scale data. It starts from Decision Tree. When we use only one single tree algorithm, there might be many bias and the classifier might not be strong enough, so we introduce the Boosting algorithm. Boosting is an algorithm to train multiple models sequentially, where each model learns from the errors of the previous model. It starts with a weak base

model, then the following models are trained iteratively. Each time we add the model to the prediction of the previous model to produce a strong overall prediction. In the case of gradient boosted decision trees, successive models are found by applying gradient descent in the direction of the average gradient, calculated with respect to the error residuals of the loss function, of the leaf nodes of previous models. LightGBM is Leaf-wise (Best-first) Tree Growth. Most decision tree learning algorithms grow trees by level (depth)-wise. However, LightGBM grows trees leaf-wise(best-first). It will choose the leaf with max delta loss to grow. Leaf-wise may cause overfitting when data amount is small, so LightGBM includes the max_depth parameter to limit tree depth.

Another famous boosting algorithm is XGBoost. It uses decision trees to split on a variable and exploring different cuts at that variable (the level-wise tree growth strategy). LightGBM concentrates on a split and goes on splitting from there in order to achieve a better fitting (this is the leaf-wise tree growth strategy). This allows LightGBM to reach first and fast a good fit of the data, and to generate alternative solutions compared to XGBoost. It might be good idea to blend those two models together to reduce the variance of the estimated. Algorithmically speaking, figuring out as a graph the structures of cuts operated by a decision tree, XGBoost peruses a breadth-first search (BFS), whereas LightGBM a depthfirst search (DFS).

- K-fold cross validation: Stratified K-fold validation

During the training process, I used k-fold cross validation to estimate accuracy to prevent overfitting and ensure that the model generalize well to new data. This will split our training dataset into k parts. Each time we will be only training on k-1 parts and test on the remaining one part. We repeat this process over and over until all combinations of train-test splits are done. Stratified means we ensure each class is (approximately) equally represented across each test fold. In this project, I have been using 5 folds. For each k-1 model, we also apply the model to real testing dataset. The final prediction result on testing dataset is the average of the 5 results from 5 models.

- Model parameter tuning Bayesian Optimization

For LightGBM, the parameter tuning was performed for looking the best parameter for the trees. Some important parameters are max_depth, number_leaves…etc. For parameter tuning, we perform Bayesian Optimization. This way we could keep tracking of best outcomes of each parameter. Other popular method would be GridSearchCV, BayesSearchCV and RandomizaedSearchCV.

- Model algorithm: Gaussian Naïve Bayes:

Gaussian Naïve Bayes is based on Bayes' Theorem.

$$P(A \mid B) = \frac{P(B \mid A)P(A)}{P(B)}.$$

In the machine learning problems, we have given feature X and our goal is to predict target y, which means we want to calculate following by apply Bayes' Theorem and choose the one with higher probability:

$$\frac{P(y=+1 \mid X=x0)}{P(y=+1 \mid X=x0)} \quad \text{posterior probability}$$

In order to calculate those two probabilities, we have to calculate the probabilities below:

$P(y=+1/-1)$ prior probability

$P(X=x0|y=+1)$

$P(X=x0|y=-1)$ conditional probability: models the distribution of the observation given that we know the class.

The training process is to estimate these two kinds of probability. The Gaussian Naïve Bayes has two assumptions. The likelihood distributions are normal and independent. It means that for each feature, the likelihood distribution is normal, and this feature is independent from other features. The model preparation is actually simple, the fitting is just computing those numbers, and the predicting part is carried out according to the above formula. [5][6]

- Model Blending and Model Stacking:

As part of the ensemble technique, blending and stacking algorithms are using various models to improve the accuracy by looking at the same data from different model's perspective. It's helpful because different classifiers might take care of some particular attributes of the data that other classifiers can't see. Blending is averaging the predictions of multiple models with different weights, which is different from Stacking. Stacking is using the outputs of models as the inputs to other models. Stacking is a process of combining validation outputs from different models to use it as a training set and test set is the combination of test outputs [7].

Benchmark:

The whole dataset (200000 data) was split randomly into training and testing dataset. We ensure that the amount of positive data in the training dataset is the same with testing dataset. Same approach applies to the amount of negative data. We use LightGBM classifier with 5 folds cross validation as benchmark. This benchmark model does not contain any data pre-processing, feature engineering, parameter tuning and model blending/stacking. The parameters are using default without tuning. The model yields ROC AUC score: 0.89177 on training data and ROC AUC score: 0.89056 on the testing data. I consider it as benchmark and will try different methodologies to improve the accuracy.

Split data:

	Training dataset	Testing dataset	Total
Data amount for label 0	89951	89951	179902
Data amount for label 1	10049	10049	20098
Total	100000	100000	200000

Benchmark model (LightGBM without data processing, feature engineering and parameters tuning)

	Training dataset	Testing dataset
ROC AUC score benchmark	0.89056	0.89177

III. Methodology

Data Preprocessing:

The data preprocessing is a technique that involves transforming real-world raw data into an understandable format. It includes data cleaning, data integration, data transformation and reduction [8].

There are some advantages of decision tree algorithm:

1. Many real-world datasets include a mix of continuous and categorical variables and decision tree models are able to operate on both continuous and categorical variables directly. Unlike most of other popular models (e.g., generalized linear models, neural networks) must instead transform categorical variables into some numerical analog by one-hot encoding them to create a new dummy variable for each level of the original variable [9]. In this case, we don't have to worry about the categorical feature transforming. This encoding process is not required for trees model.
2. The scale of the features doesn't matter for the tree based algorithm (Random Forest, LightGBM), because orderable items are ordered and then randomly bisected.
3. The interactions of features don't matter because one of the weak classifiers should find it if it is important.
4. The feature selection isn't important for the tree based algorithm because if the effect is weak then those classifiers will be attenuated.

- Data Cleaning:
    - Data incomplete: Lacking attribute values, lacking certain attributes, containing only aggregate data.
    - Noisy Data: Containing errors or outliers.
    - Inconsistent: Discrepancies in codes or names.
    - Data Balance: Data imbalanced among all classes. Usually handled by data augmentation.
- Data Integration:
    - Combines data from multiple sources into a coherent store and after detecting and resolving data value conflicts.
- Data Transformation:
    - Categorical Values: Convert Categorical variable into Numeric data, and then transfer it to Dummy Variable if it is needed based on the chosen model.
    - Feature Scaling: Limit the range of variable so that they can be compared on common grounds and don't let any particular feature dominates the during the model calculation.
- Data Reduction:
    - Principal Component Analysis.
    - Feature selection: Boruta feature or some feature tools.

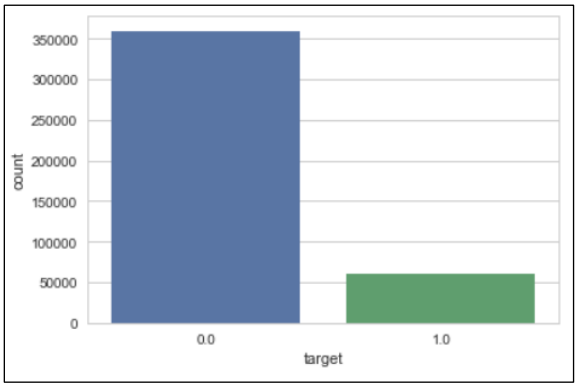
By using the quartile outlier method during the EDA process, I confirmed that there are 24896 data point are outliers out of 200000 data points, which is around 12.4%. After I removed all those outliers. The ratio of target 0 to target 1 become 10.82 and the figure shows us the result. With the outlier removal, the

training data ROC is 0.88768 and the testing data ROC is 0.88987, which is slightly worse than the benchmark model. Based on the result, I decide not to remove outliers for this project.

	Training dataset	Testing dataset
ROC AUC score benchmark	0.89056	0.89177
ROC AUC score with outlier removal	0.88768	0.88987

During the EDA process, I also realized that the data has unbalanced problem. 9.7% of the available data belongs to target 1 while 90.3% of the data belongs to target 0. There are two ways to do this: downsampling the major class or upsampling minor class. The upsampling method is used in this project. After the upsampling process, 14.35% of the available training data belongs to target 1 while 85.64% of the available training data belongs to target 0. The figure below shows the percentage of label 0 and label 1. Actually, another way to make the classes balances is to simply duplicate some of the data points in the minority class, which is called resampling. When you have classes that are imbalanced, this can cause your model to be biased as they will learn to simply guess the most common class. However, resampling the data can cause overfitting. Hense, the optimal approach will probably to be either choosethe metrics that aren't sensitive to class imbalances (AUC, log loss…) or to use a synthetic method to upsample the data (such as SMOTE). You could also try downsapling by removing data points from the major class. However, since data is our most precious resource in Machine Learning, it might not be a good idea to downsample the data.

	Training dataset	Testing dataset	Total
Data amount for label 0	179902	179902	359804
Data amount for label 1	30147	30147	60294
Total	210049	210049	420098



	Training dataset	Testing dataset
ROC AUC score	0.89056	0.89177
ROC AUC score with outlier removal	0.88768	0.88987
ROC AUC score with data augment	0.89602	0.89841

The logic of data augment is to copy each column of the data within specific target and then randomly shuffle and put it back to the columns for the new data. The training accuracy AUC is 0.89602 and the testing accuracy AUC is 0.89841, which is a little bit better than the benchmark.

Feature Engineering:

Feature engineering is about creating new input features from your existing ones. The advantages of using feature engineering is that you can isolate and highlight key information, which helps the algorithms focus on what is most important information. For example, in logistic regression. It is common procedure to add interaction terms. If the relation between target, y, and predictors, x1 x2 is quadratic, then you need to add three new variables x1^2, x2^2, x1*x2.

- Domain Expertise: Feature engineering processes sometimes involves domain expertise on the problems that you are trying to solve.
- Creating Interaction features: Interaction features can be products, sums, or differences between two features.
- Combine Sparse Classes: Grouping sparse classes. It could be used if you have very few total observations for some classes.

There are some feature engineering processes that is good for tree based algorithm:

- Modular arithmetic calculations: Converting a timestamp into day of the week, or time of day. If your model needs to know that something happens on the third Monday of every month, it will be nearly impossible to determine this from timestamps.
- New features: Creating new features from the data you have available. This is because tree based methods can only create splits that are horizontal or vertical.

Here is some feature engineering I tried for this project:

- Boruta feature, which is calculated based on random forest. The Boruta provides you the feature ranking and tell you which features are more important.
- Statistics feature: Performed sum, min, max, mean, std, skew, kurt, med, ma, percentile 5, percentile 10, percentile 25, percentile 50, percentile 75, percentile 85, percentile 95, percentile 99 new feature for each row.
- Frequency counts encoding:
  Feature engineering should be based on what model we are using, which means we should look at the data from the perspective of the model. The tree-based model cannot capture the frequency of values then frequency coding is important information to feed the model. The frequency encoding works often. Frequency encoding is a common encoding technique especially for categorical variable and it helps whenever the target is dependent on how rare or common a variable value is. In this case, we must tell tree model that which variables are unique.

  During the EDA process, we realized that there are tons of duplicate values in the data. By looking at how many duplicate values there was per column, the result shows that many columns had more than 50% of the data value duplicated. This result implies that it's might be a good idea to capture those value counts info in the data.

Given the fact that there are so many values that are duplicated, I started thinking that those values are categorical features instead of continuous features. I started to try frequency count. The regenerating frequency count features is about count of values, especially the fact that some are unique more than half of the features have very high percentage of unique values within a feature. The tree based algorithm can't see the frequency information so we should tell the tree model about this value uniqueness information. The important result of counting is to determine which variable values are unique.

For each variable I add a new feature that states the value's frequency count to represent that if the value is rare or non-rare (frequency count), which means that I created 200 categorical features and each one of them corresponds to an each raw feature. This logic computing frequency applies to both training and testing dataset to calculate the generation of 200 features for both dataset.

After generating the frequency count features, both the training and testing dataset have 400 features. After adding the 200 frequency count features, the score is 0.9001 with benchmark model parameter, which is higher than before. It proves that the frequency account feature is affecting and working.

	Training dataset	Testing dataset
ROC AUC score benchmark	0.89056	0.89177
ROC AUC score with outlier removal	0.88768	0.88987
ROC AUC score with data augment	0.89602	0.89841
ROC AUC score with Feature Engineering - Frequency count	0.88860	0.90010
ROC AUC score with Feature Engineering - Boruta features	0.87521	0.87956
ROC AUC score with Feature Engineering – Statistics	0.88650	0.89391

Model Selection:

- LightGBM

The LightGBM supports many parameters that control various aspects of the algorithm. Some core parameters that should be defined are below. The default value for each parameter is list here: https://github.com/Microsoft/LightGBM/blob/master/docs/Parameters.rst

- Core parameters:
  - Objective
  - Boosting
  - learning_rate
  - num_leaves
  - tree_learner
  - num_threads
  - seed
- Learning Control Parameters:
  - max_depth
  - min_data_in_leaf
  - min_sum_hessian_in_leaf

    - bagging_fraction
    - bagging_freq
    - feature_fraction
    - lambda_l1
    - lambda_l2
    - bagging_seed
- Others:
    - Verbosity
    - Boost_from_average
    - metric

Some highlight important parameters description:

- num_leaves: This is one of the main parameter to control the tree model complexity. Normally we set num_leaves = 2^(max_depth) to obtain the number for depth-wise tree model (xgboost). However, due to the fact that the leaf-wise tree is typically much deeper than the depth-wise tree for a fixed number of leaves, it can end up inducing overfitting. In this case, we should make num_leaves smaller than 2^(max_depth) for LightGBM. The large num_leaves might cause overfitting.
- min_data_in_leaf: The optimal value depends on the number of training samples and num_leaves. Setting it to a large value can avoid growing too deep a tree, but may cause under-fitting. In practice, setting it to hundreds or thousands is enough for a large dataset. This is also an important parameter to prevent over-fitting in a leaf-wise tree.
- max_depth: The parameter to limit the tree depth explicitly.
- learning_rate: The default learning rate value is 0.1. Usually the reasonable range somewhere between 0.05 to 0.3 should work for different problems.

The benchmark model parameters are listed below, which uses default value for most of the parameters:

Parameter	Benchmark model uses	Fix parameter or Tuned parameters later
objective	binary	Fixed
boosting	gbdt	Fixed
learning_rate	0.01	Fixed and tuned
num_leaves	31	Tuned
tree_learner	serial	Fixed
num_threads	0	Fixed
max_depth	-1	Tuned
min_data_in_leaf	20	Tuned
min_sum_hessian_in_leaf	1e-3	Tuned
bagging_fraction	1.0	Tuned
bagging_freq	0	Fixed
feature_fraction	1.0	Fixed
lambda_l1	0	Tuned
lambda_l2	0	Tuned
boost_from_average	True	Fixed
metric	auc	Fixed

After the parameters are specified, we can then call the training API to train a model. The model contains another two parameters: boosting rounds and early stopping rounds. The early stopping occurs when there is no improvement in either the objective evaluations or the metrics we defined as calculated on the validation data.

To find the best parameters, the parameters tuning is an important process. If the function we are dealing with is cheap to evaluate, we could choose Grid Search or Random Search. Otherwise the Bayesian-Optimization is more suitable for complex model [10]. The result suggests Bayesian hyperparameter optimization of machine learning model is more efficient than manual, random, or grid search with better overall performance on the test set and also less time required for optimization.

- Grid search
- Random Search
- Bayesian-optimization (scikit-optimize model or Bayesian-optimization libraries are available)

- Bayesian Optimization Hyper-Parameter Tuning

Bayesian Optimization is a methodology to perform global optimization on the black-box functions (here the boosting algorithm) without knowing much about this function. It attempts to find the maximum value of an unknown function in as few iterations as possible. Optimization is not based on derivatives or the like. Instead, previous outcomes are used to try to find a set of parameters which improve the objective. It is a probabilistic model based approach for finding the minimum of any function that returns a real-value metric. Bayesian optimization utilizes the Bayesian technique of setting a prior over the objective function and combining it with evidence to get a posterior function. The prior belief is our belief in parameters before modeling process. The posterior belief is our belief in our parameters after observing the evidence. Bayesian optimization incorporates prior belief about f and updates the prior with samples drawn from f to get a posterior that better approximates f. Bayesian optimization works by constructing a posterior distribution of functions (gaussian process) that best describes the function you want to optimize. The key is to find a good balance between exploration and exploitation. As the number of observations grow, the posterior distribution improves, and then algorithm becomes more certain of which regions in parameter space are worth exploring and which are not. As you iterate over and over, the algorithm balances its needs of exploration and exploitation by taking into account what it knows about the target function [11] [12].

The following is the step to use Bayesian Optimization:

1. Specifying the function to be optimized: Provide a function f that takes a known set of parameters and outputs a real number. If you look at this function from Bayesian Optimization, this function is just like a black box function, which contains unknown internals we wish to maximize.
2. Define the parameters for the function with their corresponding bounds. This is a constrained optimization technique, so you must specify the minimum and maximum values that can be probed for each parameter in order for it to work.
3. Define the parameter for Bayesian Optimization. n_tier controls how many steps of Bayesian Optimization you want to perform. init_points controls how many steps of random exploration you want to perform.
4. Initialize the optimization object and call the maximize function.

5. Find the best combination of parameters and target value found.

The general approach for parameters tuning [13]:

1. Choose a relatively high fixed learning rate and high fixed number of iterations. For the learning rate, use large learning rate 0.3 - 0.5 if you have large data, or 0.1-0.2 if the data is not that many. For the number of iteration, use a large number of 10000+ and make sure to set early_stopping_rounds (100-200 is enough if the learning rate is 0.1 - 0.2) to see how many iterations we get. Try higher learning rate to reduce the number of the iterations. Don't optimize for learning rate or try to find an exact num_boost_round as that lead to overfitting.
2. Tune tree-specific parameters (max_depth, min_data_in_leaf, num_leaves) for this learning rate and number of iterations. We can basically set the range first with wider intervals between values and run the model result. Once we notice the trend, we can focus on some specific values.
3. Tune regularization parameters (lambda, alpha), which can help reduce model complexity and enhance performance.
4. Once you find your best parameters with large learning rate, repeat your prediction with the same parameters but with learning rate at 0.01-0.02 and with early_stopping_round at 200-500. Step6: Reduce learning rate: Lastly, we should lower the learning rate and add more trees.

- Gaussian Naïve Bayes

In order to use Gaussian Naïve Bayes, we have to ensure that the feature likelihood distributions are normal and features are independent. By looking at the correlation matrix plot, it shows very small correlation between the features. Also, it's important that the target y is dependent on features X. If X and y were independent, then the posterior would be equal to the prior $pY|X(y|x)=pY(y)$, and we would not need to do any calculation. By looking at the plots above during EDA session, we have seen that the label 0 and label 1 likelihood distributions are different in the sample means and sample variances.

- Blending model with LightGBM and Naïve Bayes

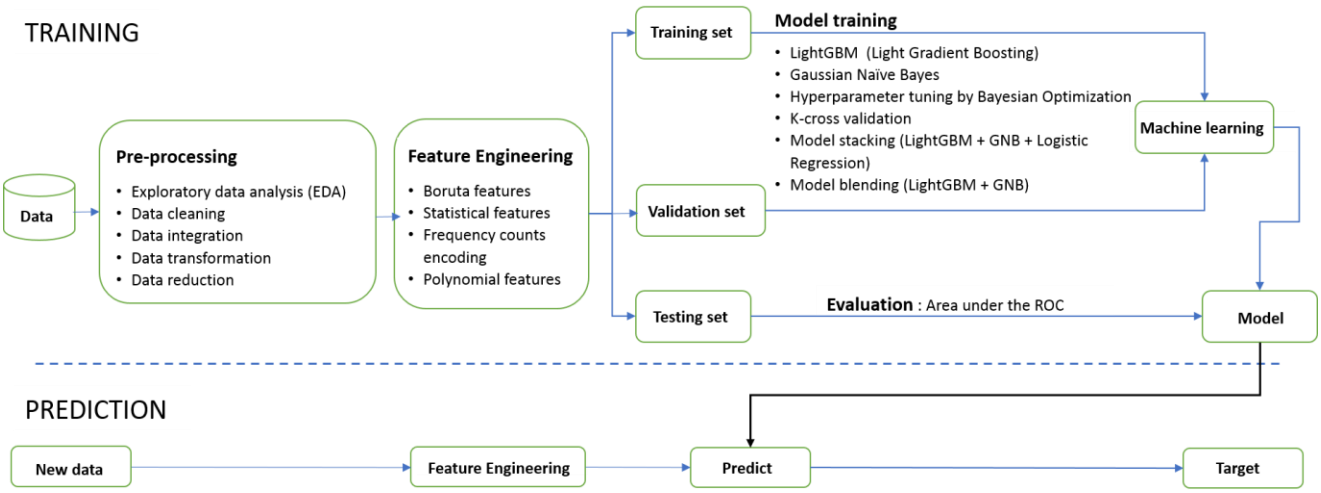
The final testing set prediction is made by blending LightGBM model result and Gaussian Naïve Bayes result. The weights to combine two models require some experiment.

- Stacking model with LightGBM and Naïve Bayes

The LightGBM and Gaussian Naïve Bayes are used as level 1 classifier and the Logistic Regression is used as meta second level classifier. The level 1 classification result will be feed in to the level 2 meta classifier model for final classification [14].

Implementation:

This project follows the standard machine learning analytics flow as shown in the figure below:



Refinement:

To ensure that the model we created is good, we performed Bayes Optimization hyperparameter tuning for LightGBM and splitting the data into training and validation sets to check the accuracy and generalization of the best model. We focused on tuning the important parameters, for example, max_depth. After the lightGBM is tuned, I added Gaussian Naïve Bayes model for further blending and stacking. The blending process requires experiment to find the best weights for two models and the stacking process uses Logistic Regression for the final classifier.

IV. Results

Model Evaluation and Validation:

- **The LightGBM model parameters tuned result is shown in the table below**:

Learning rate: LR. Number round: NR. Early stopping rounds: ESR		LR: 0.5 NR: 15000 ESR: 200	LR: 0.1 NR: 15000 ESR: 200	LR: 0.01 NR: 15000 ESR: 200	LR: 0.5 NR: 1000000 ESR: 3500
Parameters	**Values tested**	**Optimal Value**	**Optimal Value**	**Optimal Value**	**Optimal Value**
num_leaves	(5, 80)	5	73	78	73
max_depth	(3, 30)	12	24	30	24
min_data_in_leaf	(5, 80)	6	5	9	5
min_sum_hessian_in_leaf	(0.00001,100)	99.45	35.74	1.94	35.74
bagging_fraction	(0, 1)	0.002	0.94	0.31	0.94
lambda_l1	(0, 5.0)	0.37	2.97	0	2.97
lambda_l2	(0, 5.0)	2.71	2.14	0.33	2.14
Testing dataset ROC score		0.89738	0.89930	0.89859	0.90042

- **Gaussian Naïve Bayes model result:**

Algorithm	Gaussian Naïve Bayes
Training dataset ROC score	0.8883
Testing dataset ROC score	0.8887

- **Blending models result:**

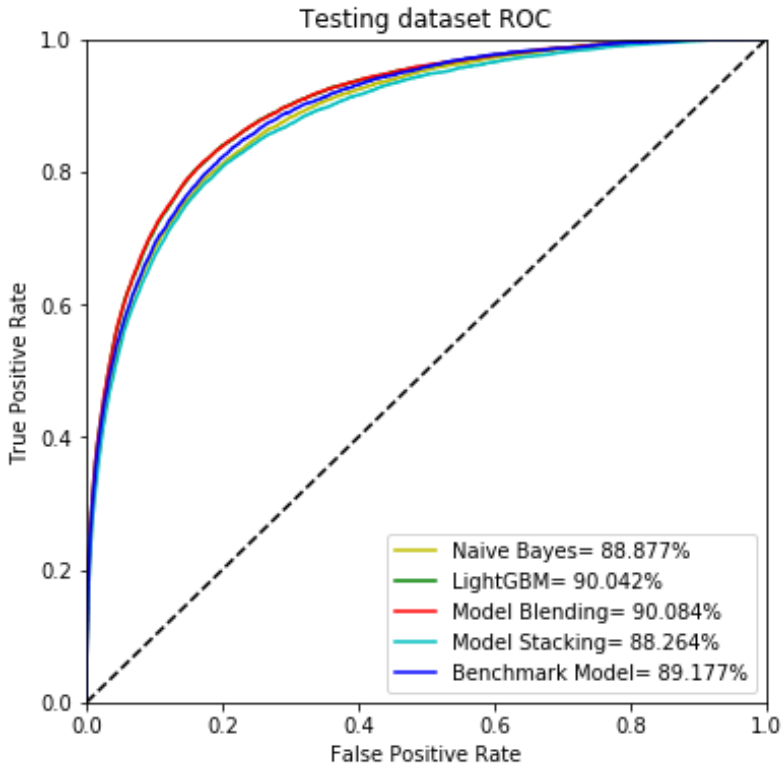
Algorithm	LightGBM * 0.9 + Gaussian Naïve Bayes * 0.1
Testing dataset ROC score	0.90084

- **Stacking models result:**

Algorithm	LightGBM and Gaussian Naïve Bayes as level 1 classifiers and logistic regression as level 2 (meta) classifier
Testing dataset ROC score	0.88264

ROC is a useful tool when predicting the probability of a binary outcome. It can be more flexible to predict probabilities of an observation belonging to each class in a classification problem rather than predicting classes directly. We can control the trade-off between true positive and false positive when using the model where the cost of one error outweighs the cost of other types of errors. ROC curve is used as a summary of the model skill. The table below shows the ROC scores for each model selection:

Algorithm	Testing dataset ROC score
Benchmark	89.177%
LightGBM	90.042%
Gaussian Naïve Bayes	88.870%
LightGBM + Gaussian Naïve Bayes stacking	88.264%
LightGBM + Gaussian Naïve Bayes blending	90.084%



From the ROC score result plot above, the lightGBM performed a bitter than the benchmark model. The Naïve Bayes performed not too bad too although it's a simple model. The Naive Bayes is very effective for dataset when features are independent and uncorrelated. The highest one is the blending model for our testing dataset. Our end goal was to have a final model that could beat the benchmark. It shows that the model generalizes well enough since the scores are very close to the validation set.

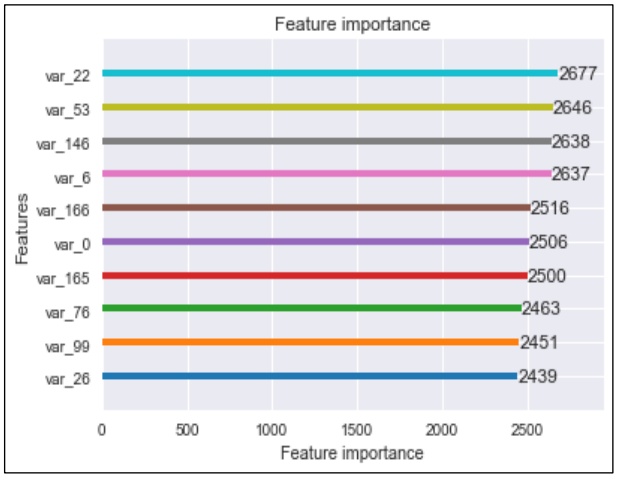
Justification:

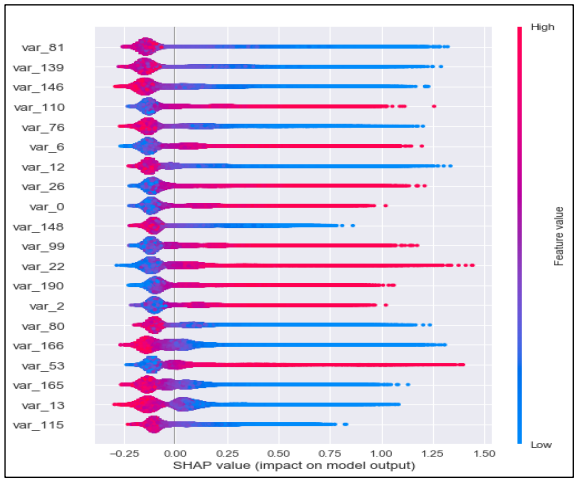
The tuned and ensemble final model made no too much significant improvement over the untuned model. This could mean that we have to work on feature engineering and model ensemble more to extract and generate more powerful features, and also more diversified models. Usually, the tuning of the algorithm will lead you to slightly better result, however the features and model ensemble are the key that could lead you to a significant improvement.

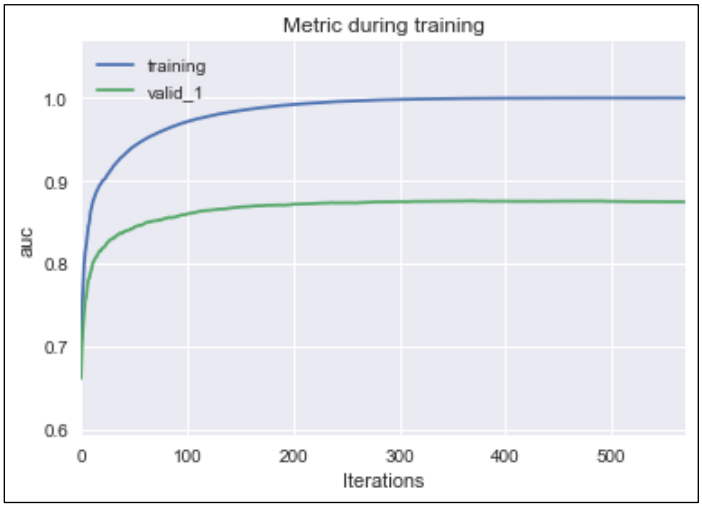
V. Conclusion

Free-Form Visualization

The importance of each feature for lightGBM was visualized with the following plot. The Feature Importance figure shows that var_22 is the most important feature and the SHAP value (impact on model output) figure shows that the higher of var_22, the more impacting on the model. The metric plot below tells us that the validation model stops getting better after around 100 iterations. I think it doesn't mean that the model is overfitting since the validation model does not decrease the performance. It keeps on the same accuracy. But we could have the conclusion that 300 iterations are probably enough already.







Reflection

Overall, from data pre-processing and feature engineering perspective, the data outliers removal and unbalanced data handling doesn't make significant improvement on the benchmark model. From the model perspective, there are still a lot good model that I haven't tired. For example, Neural Network and deep learning network. Also, from implementation perspective, I should try more pipeline structure to make the code more clean and readable.

Improvements

As mentioned on the Reflection session, there are other ways to handle unbalanced data. For example SMOTE. The feature engineering is tied with the model we select. We need to do the feature engineering based on what model can't see. For LightGBM this tree based model, it looks like the frequency count feature engineering I have tried only improved few percent of the accuracy. Next thing I can try would be encoding the frequency count with different categories, for example we can encode the count as 0 and 1, which represent unique or non-unique instead of using the counts value itself. Also, since we can get the feature importance ranking after the LightGBM, it is worth trying to do feature engineering on those important features. For the model, try to think of more diversified models and ensemble them to generate more powerful final classifier. Some simple classifier without too much tuning might come out to be as good as some complicated classifiers. As you can see in this project that the simple Gaussian Naïve Bayes perform almost the same with LightGBM after so much tuning work. Last but not least, in terms of Sensitivity analysis, we should shuffle the data and re-run the model multiple times and then look at the Standard deviation (std) and mean values from those result to see if the model is stable enough. If the std is high, it means that the model is very sensitive to the data, which is not a good model in the end.

VI. Reference

[1] https://www.quantinsti.com/blog/use-decision-trees-machine-learning-predict-stock-movements
[2] https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8212859
[3] https://www.econstor.eu/bitstream/10419/183139/1/1032172355.pdf
[4] https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1631572
[5] https://www.kaggle.com/jiazhuang/demonstrate-naive-bayes
[6] https://www.kaggle.com/blackblitz/gaussian-naive-bayes
[7] https://www.kaggle.com/roydatascience/santander-transaction-stacking-1-0
[8] https://hackernoon.com/what-steps-should-one-take-while-doing-data-preprocessing-502c993e1caa
[9] https://roamanalytics.com/2016/10/28/are-categorical-variables-getting-lost-in-your-random-forests/
[10] http://krasserm.github.io/2018/03/21/bayesian-optimization/ and https://lightgbm.readthedocs.io/en/latest/Parameters.html
[11] https://github.com/fmfn/BayesianOptimization/blob/master/examples/basic-tour.ipynb.
[12] http://krasserm.github.io/2018/03/21/bayesian-optimization/

[13] https://www.kaggle.com/c/santander-customer-transaction-prediction/discussion/84639#latest-493441
[14] https://www.kaggle.com/thomasnelson/simple-stacking-classifier-for-beginners/code
[15] https://www.kaggle.com/cdeotte/200-magical-models-santander-0-920/comments#518661
[16] https://www.kaggle.com/felipemello/why-your-model-is-overfitting-not-making-progress
[17] https://www.kaggle.com/felipemello/step-by-step-guide-to-the-magic-lb-0-922
[18] https://www.kaggle.com/cdeotte/modified-naive-bayes-santander-0-899
[19] https://www.kaggle.com/cdeotte/200-magical-models-santander-0-920
[20] https://www.kaggle.com/peterheindl/lgbm-parameter-tuning-with-bayesian-optimisation
[21] https://www.kaggle.com/c/bnp-paribas-cardif-claims-management/discussion/19083#108783