Predicting Customer Satisfaction

Identifying Dissatisfied Customers for Santander Bank with Machine Learning

George Liu

From frontline support teams to C-suites, customer satisfaction is a key measure of success. Unhappy customers don't stick around. What's more, unhappy customers rarely voice their dissatisfaction before leaving. In a recent Kaggle competition, Santander Bank asks Kagglers to help them identify dissatisfied customers early in their relationship. Doing so would allow Santander to take proactive steps to improve a customer's happiness before it's too late. In this project, we'll tackle this competition and try to predict whether a customer is satisfied or not using customer data.

Definition

1. Project Overview
   To make predictions for specific customers, we are given two datasets: train and test sets. The datasets contain various sorts of customer information such as nationality, number of bank products etc. However, for confidentiality reasons, all features are anonymized. We need to build the model with the train dataset and then make predictions for the test dataset – for each customer, based on the given feature values, predict whether the customer is satisfied or not.
2. Problem Statement
   Based on the situation, we now define the problem as:

   > *Build a predictive model that learns from the train dataset, and makes customer satisfaction predictions with the test dataset after training. The predictions should be in probability form instead of binary classification results.*

   To solve the problem, we need to first examine the data to ensure understanding of the data structure, then based on the understanding choose the most suitable algorithms, preprocess the data, and finally build/tune the model and validate to make sure the model is optimized.

   The final output of predictions will be in csv format containing customer ID's and corresponding probabilities of being dissatisfied customers.
3. Metrics
   To measure the performance of a specific model, we have many metric choices such as accuracy, precision, recall, F1 score, AUC-ROC (Area Under the Receiver operating characteristic Curve) and AUC-PR (Area Under the Precision Recall Curve). In this competition, since AUC-ROC is chosen to evaluate performance, we'll also use it in the model building process. The AUC-ROC score is calculated using the sklearn "roc_auc_score" function which obtains the score from actual labels and predicted probabilities.

   AUC-ROC is a comprehensive metric, because unlike F1 score which calculates one single score based on precision and recall, AUC-ROC is a metric calculated as an area under the ROC curve which comprises various true positive rate (TPR) and false positive rate (FPR) combinations. At

each point on the curve, there is (at least) one corresponding F1 score. To plot the ROC curve, we simply use all possible threshold values to generate different TPR, FPR pairs which are in turn plotted to form the curve.

Part of the reason why AUC-ROC is selected as the metric to use is due to the fact that the output is probabilistic predictions. As such, for each data point, we have a probability instead of a predicted class label produced. Therefore, we can use the AUC-ROC to incorporate more information, i.e. model performance at different discrimination threshold values. Whereas if we have labels like 0 or 1 outputted, it wouldn't be possible to have different TPR, FPR results when varying the threshold value. On the other hand, since the dataset we have is highly imbalanced, AUC-ROC is also a good choice for performance measurement (Chawla, Japkowicz, & Kolcz 2004).

Analysis

4. Data Exploration

The data for this project is provided in the form of a train and a test dataset. The original zipped files are about 4 MB's, after decompressing, the size becomes about 58 MB's. Both datasets are in CSV format and the train dataset has 76020 rows, 371 columns while the test dataset has 75818 rows and 370 columns. The train set has one extra column which is the target column. All the column types are numerical and there are no null values. The dataset is highly imbalanced as the percentage of unsatisfied customers is 3.96%.

Although we do have feature names, they almost provide no meaningful information due to anonymization. Below are the first five rows of the train dataset.

	ID	var3	var15	imp_ent_var16_ult1	imp_op_var39_comer_ult1	\
0	1	2	23	0.0	0.0	
1	3	2	34	0.0	0.0	
2	4	2	23	0.0	0.0	
3	8	2	37	0.0	195.0	
4	10	2	39	0.0	0.0	

	imp_op_var39_comer_ult3	imp_op_var40_comer_ult1	imp_op_var40_comer_ult3	\
0	0.0	0.0	0.0	
1	0.0	0.0	0.0	
2	0.0	0.0	0.0	
3	195.0	0.0	0.0	
4	0.0	0.0	0.0	

	imp_op_var40_efect_ult1	imp_op_var40_efect_ult3	...	\
0	0.0	0.0	...	
1	0.0	0.0	...	
2	0.0	0.0	...	
3	0.0	0.0	...	
4	0.0	0.0	...	

	saldo_medio_var33_hace2	saldo_medio_var33_hace3	saldo_medio_var33_ult1	\
0	0.0	0.0	0.0	
1	0.0	0.0	0.0	
2	0.0	0.0	0.0	
3	0.0	0.0	0.0	
4	0.0	0.0	0.0	

	saldo_medio_var33_ult3	saldo_medio_var44_hace2	saldo_medio_var44_hace3	\
0	0.0	0.0	0.0	
1	0.0	0.0	0.0	
2	0.0	0.0	0.0	
3	0.0	0.0	0.0	
4	0.0	0.0	0.0	

	saldo_medio_var44_ult1	saldo_medio_var44_ult3	var38	TARGET
0	0.0	0.0	39205.170000	0
1	0.0	0.0	49278.030000	0
2	0.0	0.0	67333.770000	0
3	0.0	0.0	64007.970000	0
4	0.0	0.0	117310.979016	0

[5 rows x 371 columns]

Anonymized features can be the most important characteristic of the dataset, because without knowing what the features represent, it would be almost impossible to do meaningful outlier detection and processing. For example, for the “var3” column, we have the following value counts:

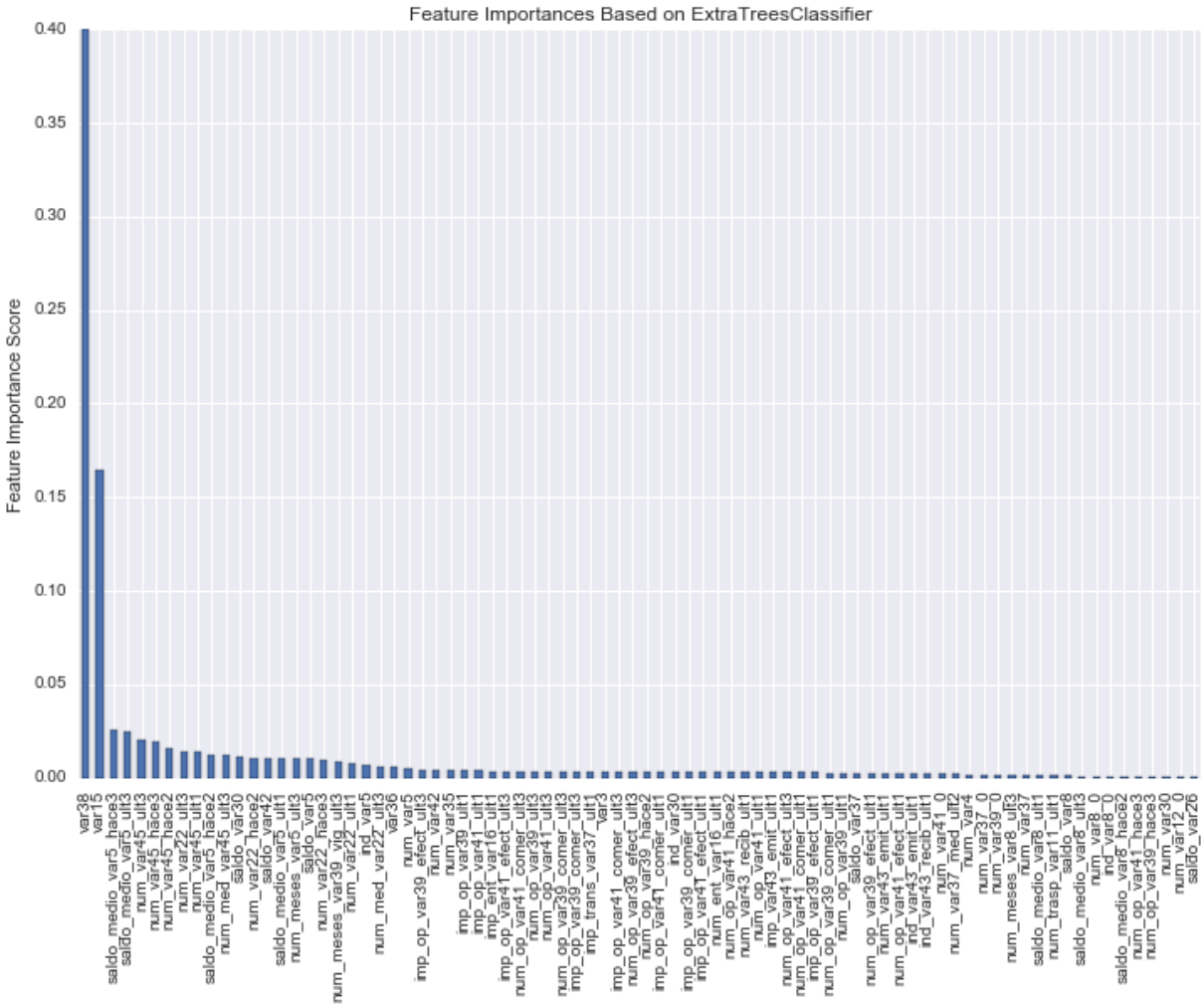
```
2       74165
 8        138
-999999    116
 9        110
 3        108
 1        105
 13        98
 7         97
 4         86
 12        85
 6         82
 0         75
 10        72
 11        66
 5         63
 14        61
 15        34
 18        10
 16         9
 17         7
 23         7
 25         6
 142        6
 154        6
 20         6
 31         6
 38         6
 153        5
 24         5
 91         5
       ...
```

We can see that the value “-999999” appears quite often. This definitely looks like an outlier and could denote unknown or missing value. However, since we don’t know exactly what the feature represents and what this value refers to, and furthermore, this feature may not even be selected later on, we’ll leave it as is. The following are the summary statistics about the train set (shown only first six columns due to space limit):

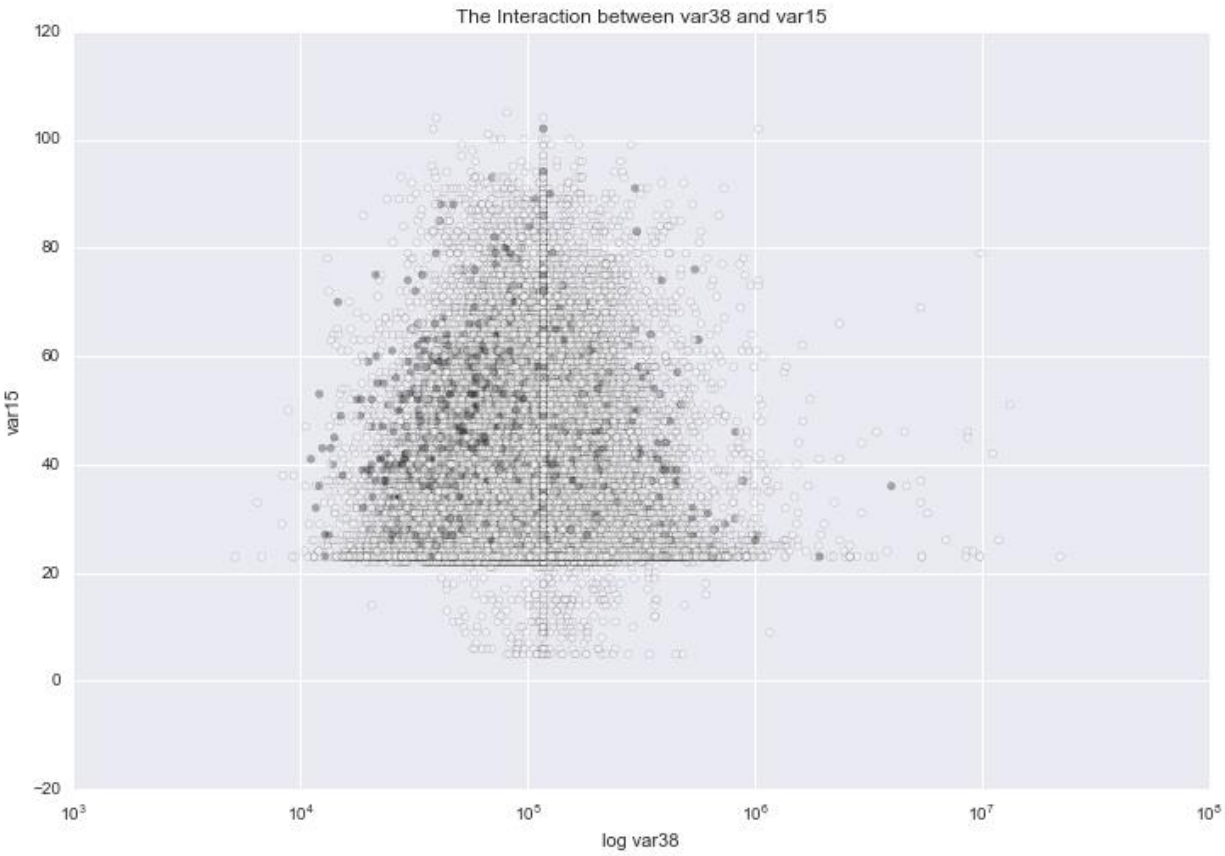
	ID	var3	var15	imp_ent_var16_ult1	imp_op_var39_comer_ult1	imp_op_var39_comer_ult3
count	76020.000000	76020.000000	76020.000000	76020.000000	76020.000000	76020.000000
mean	75964.050723	-1523.199277	33.212865	86.208265	72.363067	119.529632
std	43781.947379	39033.462364	12.956486	1614.757313	339.315831	546.266294
min	1.000000	-999999.000000	5.000000	0.000000	0.000000	0.000000
25%	38104.750000	2.000000	23.000000	0.000000	0.000000	0.000000
50%	76043.000000	2.000000	28.000000	0.000000	0.000000	0.000000
75%	113748.750000	2.000000	40.000000	0.000000	0.000000	0.000000
max	151838.000000	238.000000	105.000000	210000.000000	12888.030000	21024.810000

5. Exploratory Visualization

With an initial feature selection using “feature_importances_” from ExtraTreesClassifier, we have the following graph showing feature importance.



We can see that var38 and var15 are by far the most important features. Below, we provide visualization of the interaction between these two features. It is interesting to see that unhappy customers only happen when var15 is higher than a certain value around 22.



6. Algorithms and Techniques

Since our classification task requires the output of probabilities, natural probabilistic classifiers are first candidates. These include Naïve Bayes and Logistic Regression. In the meantime, any other classifiers with "predict_proba" method are also great candidates, including: K-Nearest Neighbors, Random Forests, AdaBoost, and Gradient Tree Boosting. We'll focus on the last three ensemble methods due to higher prediction accuracy. Additionally, xgboost (Extreme Gradient Boosting) will also be used for model building due to popularity within the Kaggle competition community.

We now provide further explanations for all the algorithms to be used:

Naïve Bayes (NB): NB is a natural probabilistic classifier, it uses Bayes theorem to calculate probability for new data based on probabilities obtained from existing data. Since what it does is just counting numbers and making simple calculation, the algorithm is very fast and normally produces fair prediction results.

Logistic Regression (LR): LR is another simple yet effective algorithm that is also naturally probabilistic. By combining linear regression with logistic function, it connects feature data with target data to make binary predictions. LR runs fast and is not prone to overfitting.

Random Forests (RF): As an ensemble algorithm using averaging method, RF makes predictions by taking average of different estimators. RF can handle complex relationships in data,

correspondingly, it can be susceptible to overfitting. It provides good accuracy with fast running time.

AdaBoost (AB): AB is the first successful boosting algorithm for binary classification. Boosting is a general ensemble method that creates a strong classifier from a number of weak classifiers. Compared with ensemble classifiers that use averaging method, AB leverages boosting, i.e., to use later models to correct the errors from initial model which "boosts" model performance.

Gradient Tree Boosting (GB): A generalized form of AB which combines gradient descent with boosting. It is an "accurate and effective off-the-shelf procedure that can be used for both regression and classification problems" (sklearn). Its advantages include natural handling of mixed type data, predictive power and robustness to outliers in output space. The limitation lies in scalability.

XGBoost (XB): XB has become the most popular algorithm for machine learning competitions on Kaggle. It's based on GB. The developer's goal is "to push the extreme of the computation limits of machines to provide a scalable, portable and accurate library". Compared with GB, xgboost uses a more regularized model formalization to control over-fitting, which gives it better performance (Quora). The algorithm is very fast, accurate and efficient on system resources.

7. Benchmark

In order to measure model performance, we run some initial fittings with un-tuned models and no preprocessing. The AUC-ROC scores and run times are recorded below and will serve as benchmarks for comparison later.

Algorithms	AUC-ROC (based on test split within the original train set)	AUC Score (Kaggle Private Leaderboard)	Time
NB	0.512735990291	----------	2.0s
LR	0.60153907602	----------	10.0s
RF	0.669596357574	----------	3.8s
AB	0.824351815912	----------	17.6s
GB	0.836030403153	----------	3.3min
XB	0.838048399141	0.821234	17.8s

From the table, we can clearly see ensemble methods prevail in terms of prediction accuracy. Among the ensemble methods, boosting methods perform much better than averaging methods. GB produces a great result but does so at the cost of much longer run time. Therefore, overall, XB is the winner, both in terms of prediction accuracy and run time as well. So we'll use XB to further build our model and fine tune the parameters to achieve the optimal result.

Methodology

8. Data Preprocessing

Clearly, we need to do some data preprocessing before model building, be it feature selection, feature scaling or principal component analysis.

We start by removing zero variance features that don't contribute additional information to the model. This is done using VarianceThreshold() in sklearn's feature_selection module.

Next, there are a number of different combinations for feature preprocessing including: scaling, PCA and feature selection. These steps are potentially helpful since:

- Features' scale vary greatly. For example, var15's median is 28 while var38's median is 106409. This may lead to poor PCA performance as PCA is a variance maximizing exercise. Here, we will sklearn's StandardScaler to scale the data by standardizing.
- PCA reduces dimensions and can possibly help avoid curse of "curse of dimensionality". After standardizing the data, we fit a PCA model by using all the features, and then choose the number of PCA's that gives a cumulated explained variance ratio of 0.99.
- Not all features are equally important or useful. To maximize model performance, we need to choose the optimal feature set. Due to the large number of features available even after PCA (142), we use sklearn's SelectPercentile for feature selection. Different percentage points are then cross-validated for performance using GridSearchCV.

To simplify the model tuning process, at this stage, we don't tune any parameter of the algorithm. Depending on whether each step is carried out and what parameters are chosen, we have many different combinations of preprocessing choices. The table below lists all the results.

Zero-Variance Features Removal	Scaling (Standardizing)	PCA	Feature Selection - SelectPercentile	Feature Selection - SelectFromModel	AUC-ROC
Yes	Yes	Yes	Yes (percentile=50)		0.820938320924
Yes	Yes	Yes		Yes (threshold='median')	0.824994846653
Yes	Yes	Yes			0.825247224132
Yes	Yes		Yes(percentile=50)		0.837313961907
Yes	Yes			Yes (threshold='median')	0.838109545363
Yes	Yes				0.838048399141
Benchmark					0.838048399141
Yes			Yes(percentile=50)		0.837313961907
Yes				Yes (threshold='mean')	0.837300765699

The parameters of SelectPercentile are tested using GridSearchCV. Since the scoring function "chi2" requires non-negative input, we'll thus use "f_classif" as the scoring criterion. After doing cross validation with percentiles including 10, 20, 30, 50, 60, we have the best result with 50% of features selected.

For SelectFromModel, we test model performance with threshold values including mean, median, 1.25mean and 1.25median.

We can see from the table that the best performance happens with standardizing, no PCA, and with feature selection using SelectFromModel. And it is also the only combination that produces higher than benchmark performance.

9. Implementation

   What is worth documenting for the whole process is installation of the xgboost library and the use of it. The xgboost library used to be available through PyPI, but at the time of this project, it was not longer available. After doing a thorough research, I managed to successfully install xgboost on Windows for both IDLE and Anaconda. The following is the procedure I took:

   In Git Bash on Windows:

   git clone --recursive https://github.com/dmlc/xgboost
   git submodule init
   git submodule update

   Then install TDM-GCC.

   alias make='mingw32-make'
   cp make/mingw64.mk config.mk; make -j4

   Last, do the following using anaconda prompt:

   cd xgboost\python-package
   python setup.py install

   When using xgboost with sklearn, it's important to use the sklearn wrapper that can be imported like this: "from xgboost.sklearn import XGBClassifier". Otherwise, xgboost won't work properly with sklearn.

10. Refinement

    To further improve the model's performance, we cross validate different algorithm parameter settings in the xgboost model. Since in previous preprocessing step, we achieve a higher than benchmark performance with standardizing and feature selection using SelectFromModel, we'll base our model tuning on this combination.

    The tuning of the model is done on a train set that is an 80% stratified split from the original "train" set. The final ROC-AUC score is calculated using the other 20% split. As we use the default 3-fold cross-validation built in GridSearchCV, we can make sure that our model is not overfitted to any specific split or pattern.

    Below, detailed explanation about each tuning parameters are provided. This subset of parameters is selected for tuning as they are believed to have higher importance in model performance. The typical final values to be used and default value are put in ().

    - objective: logistic regression for binary classification.
    - learning_rate: (0.01-0.2, default=0.3), improves model by shrinking the weights on each step.
    - max_depth: (3-10, default=6), used to control over-fitting.
    - min_child_weight: (default=1), used to control over-fitting.
    - subsample: (0.5-1, default=1), the fraction of observations to be randomly samples for each tree, controls overfitting.
    - colsample_bytree: (0.5-1, default=1), the fraction of columns to be randomly samples for each tree.
    - n_estimators: number of estimators.

We have a large number of candidate parameters to tune, as a result, when we add a new value for a specific parameter, the overall tuning effort increases very fast. For 5 parameters with 2 values each, we have a total choice of 2^5=32, compared with (2^4)*3=48. Therefore, it would be a huge challenge on system resource when we have many tuning values for each parameter and tune all parameters altogether.

As a result, I decide to start by testing parameters in turn, one at a time while keeping others the same. For example, when testing the learning rate, I used a full spectrum of values in the typical used value range [0.01, 0.05, 0.1, 0.15, 0.2] and chose 0.1 as it gives the highest AUC-ROC score. After running cross-validation with the default 3-fold setting in GridSearchCV for all parameters, we get an AUC score of 0.841678266437, with the following parameters:

> XGBClassifier(base_score=0.5, colsample_bylevel=1, colsample_bytree=0.5, gamma=0, learning_rate=0.1, max_delta_step=0, max_depth=4, min_child_weight=4, missing=None, n_estimators=100, nthread=4, objective='binary:logistic', reg_alpha=0, reg_lambda=1, scale_pos_weight=1, seed=88, silent=1, subsample=1)

Results

11. Model Evaluation and Validation

Since I noticed from previous tests that feature preprocessing generally makes the performance worse, I decided to skip all the preprocessing and feature selection steps and to test the model again. It turns out that the performance does improve again to 0.842057031736 (this is the new value in version 1.0 of this report, the original best was 0.841714669769; the corresponding private leaderbodard score however dropped from 0.823630 to 0.823016, indicating overfitting)! The final best estimator is thus:

> Pipeline(steps=[('xgbclassifier', XGBClassifier(base_score=0.5, colsample_bylevel=1, colsample_bytree=0.5, gamma=0, learning_rate=0.1, max_delta_step=0, max_depth=4, min_child_weight=4, missing=None, n_estimators=100, nthread=4, objective='binary:logistic', reg_alpha=0, reg_lambda=1, scale_pos_weight=1, seed=88, silent=1, subsample=1))])

12. Justification

Recall results from the previous Benchmark section, we have the following AUC scores:

NB	0.512735990291
LR	0.60153907602
RF	0.669596357574
AB	0.824351815912
GB	0.836030403153
XB	0.838048399141 (private leaderboard score 0.821234)

Comparing with our final result of 0.842057031736 (Kaggle private leaderboard AUC score 0.823016), we do see some significant improvement. To put this in context, on the leaderboard

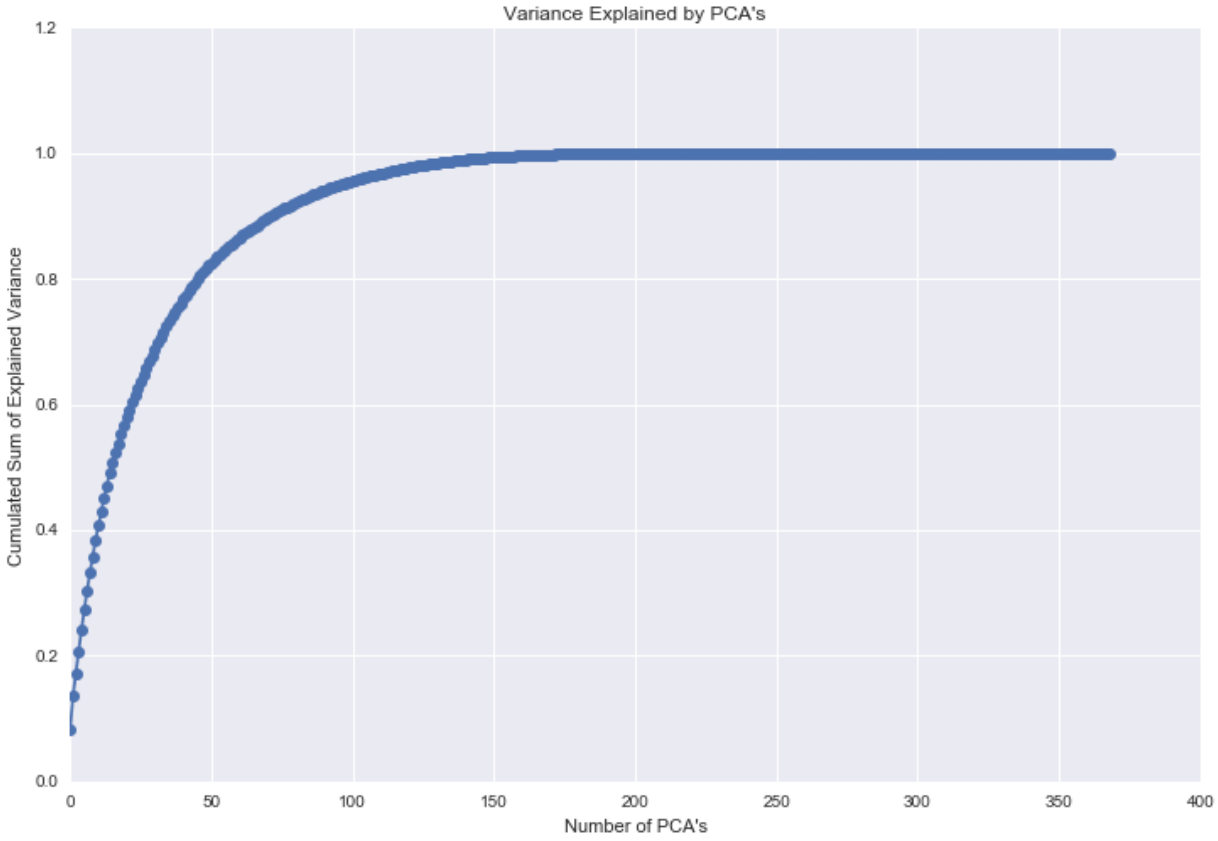
of this Kaggle competition, with a 0.001 improvement in AUC, there is a ranking difference of 135 players! With this result, I was able to achieve a ranking of 2804 out of 5123 teams (this was achieved with the original result of 0.841714669769, private 0.823630).

Conclusion

13. Free-Form Visualization

    Below, we visualize the variance explained by different number of PCA's. The horizontal axis shows the number of PCA's, and the vertical axis represents the cumulated amount of variance in the data explained.

    Originally, there are a total of 369 features, with principal component analysis, we see that 142 PCA's represent 99% of the variance. This is to say that, even after PCA, we still have a lot of features to deal with. This seems to be the defining characteristic of the data and this project – the great number in features and the difficulty in choosing the right ones to use.



14. Reflection

    First, let's summarize the entire process of this project.

We started with data exploration and exploratory visualization. Once we realized the dataset is highly imbalanced, we decided that AUC-ROC may be a more suitable metric to measure model performance. Based on the requirement of the competition, we decided to choose natural probabilistic classifiers and ensemble methods for higher performance, also did spot checks to further decide the algorithm to work with.

Then, for data preprocessing we used a number of different combinations of procedures including: zero variance features removal, feature scaling, PCA and feature selection. We found out that it is featuring scaling and SelectFromModel combined to provide the highest performance.

Finally, we did parameter tuning through cross validation using GridSearchCV and reached the best performance and corresponding parameter set with no preprocessing and feature selection.

What is interesting about the project is the fact that neither PCA nor feature selection helps to achieve the best model performance, which is not the case for normal projects. This may be because of the special structure in the data, or has something to do with xgboost algorithm which is known to work well with large dataset and outliers.

What is difficult about this project is the unfamiliar concept of AUC-ROC and the need for a new tool (xgboost) that is hard to install on Windows machines. Also, the parameter tuning process is tedious but critically important as reflected by the jump in AUC-ROC score in this revision of the report (0.841714669769 to 0.842057031736).

Finally, as a first time player to compete on Kaggle, I am satisfied with the ranking and final solution. Going forward, I can further build on the process developed for this completion and try to have even better results in future competitions.

15. Improvement

Of course, there is room for more improvement and better performance.

First, although the features are anonymized, we can still try some ways to deal with outliers for some of the features. Potentially, we can test by removing all perceived outliers or replacing outliers with other values. However, given the large number of features we have, this may take much more time to complete.

Second, we can also try feature engineering. To do this, we need to further explore the features, particularly the pair relationships among the most important features. Then based on this understanding, we can add new features that are either transformations or combinations of existing features to boost performance.

Third, an even broader selection of parameters can be tuned which may also boost prediction performance. Specifically, we can include the following parameters for tuning: max_leaf_nodes, gamma, max_delta_step, lambda, alpha, scale_pos_weight etc. This will significantly increase the model tuning time given the already large number of parameters for tuning.

Reference

1. Ensemble learning
2. Sklearn ensemble methods
3. Receiver operating characteristic
4. About Feature Scaling and Normalization
5. The Relationship Between Precision-Recall and ROC Curves
6. AUC: A Better Measure than Accuracy in Comparing Learning Algorithms
7. How to interpret almost perfect accuracy and AUC-ROC but zero f1-score, precision and recall
8. Beyond Accuracy, F-score and ROC: a Family of Discriminant Measures for Performance Evaluation
9. Installing XGBoost for Anaconda on Windows
10. Installing Xgboost on Windows
11. Story and Lessons Behind the Evolution of XGBoost
12. Complete Guide to Parameter Tuning in XGBoost (with codes in Python)