Predicting Home Credit Default Risk

Clayton Weidinger
Machine Learning Engineer Nanodegree Capstone Project
August 8, 2018

I. Definition

Project Overview

Home loans enable long term wealth building for the borrower and a steady Return On Investment (ROI) for the lender, if the lender can avoid loans that are likely to default (not pay back their loan). Traditional methods of measuring default risk, such as the FHA insurability requirements, have created a gap between the credit haves and havenots. Those meeting FHA requirements can often find a loan that enables home owners to build wealth whereas those who don't are forced to waste money by renting or face the unreasonably high interest rates commonly referred to as predatory lending.

An enterprise that can identify loan applications with low default risk using alternative, perhaps machine learned, criteria stands to profit as they will be able to earn a higher ROI than FHA approved loans and their borrowers can get a loan between the FHA approved loan market rate and the predatory lending market rate. The "Home Credit Group" has seen this opportunity and is leveraging the benefits of competition on kaggle.com to develop a better algorithm to predict home credit default risk.

Problem Statement

The "Home Credit Default Risk" competition on kaggle.com explains to problem I want to solve along with a quantification of what solved looks like. In short, the problem is to use the financial data points provided in the competition to predict default risk ([0,1]). The "Home Credit Group" has provided training data that pairs a borrower id with the default risk target variable (a 0 if they didn't default, a 1 if they defaulted) along with a host of other data that will serve as inputs to an algorithm. The solution will maximize the Area Under the Curve (AUC) of the Receiver Operator Characteristic (ROC). This is a specific way to calculate comparisons between algorithms that prefer True Positives and not False Positives.

This problem will be solved when the default risk for all the applications is generated from a model in a way the distribution of True Positives with respect to default risk is separable from the distribution of False Positives thus maximizing the AUC of the ROC (Area Under the Curve of the Receiver Operator Characteristic). However, a more reasonable goal is to attain an AUC of the ROC higher than the benchmark model built with the FHA industry standard criteria. Additionally, Kaggle maintains the AUC of the ROC for 20% of the test data (where the default information is not publically available) on the public leaderboard (Kaggle 2018), so I can also test my solution against the best machine learning engineers on that platform.

In this project, I follow the CRISP-DM steps that are relevant to this problem statement. To get an understanding of the data, I explore the distribution of values (for different data types) for each column in the dataset with respect to the target value (default risk). I decide what transformation of the data is likely to improve predictability and when to include this variable into the model by using an automated process as making manual decisions for 221 columns would be an onerous and error prone task.

I also choose a model and model parameters based on how well they maximize the AUC of the ROC metric for the validation folds when using k-fold cross validation. First, I compare various models types based on their default parameters and then I tune the top model using a grid search.

Metrics

As mentioned previously, the kaggle competition uses the AUC of the ROC to rank solutions. This is an appropriate measurement because it rewards models that correctly identify applications that will default (True Positives) and punishes models that falsely predict that an application will default (False Positives). Theoretically, the Area Under the Curve is the integral of the function that creates the curve. The Receiver Operator Characteristic is the plot of True Positives (on the vertical axis) with the False Positives (on the horizontal axis) for various cutpoints in the target variable (in this case the default rate).

A lender needs to know the model and the cutpoint to make decisions. For instance, the lender will pick a cutpoint (lets say 0.4) that trades off the lenders appetite for risk (high risk is low True Positive rate) and reward (high reward is low False Positive rate). All applications with default rate predictions above 0.4 will be declined and all loan applications with default rate predictions below 0.4 will be accepted.

II. Analysis

Data Exploration

The datasets provided by the “Home Credit Group” in the kaggle competition contain complex relationships that require significant preprocessing before they can be fed into a machine learning algorithm. First, there is a nesting of the relationships between the datasets and the target variable which requires many aggregations and joins. Second, there are relationships between the columns that are discussed in the data column definitions. Third, there are relationships between columns that only a domain expert would know about. And finally, there are some relationships that even a domain expert wouldn’t know about but are present in the data nonetheless. Because I don’t have access to a domain expert in home lending, I hoped to automate the process of feature transformation and selection. The benefit of this approach is that I may discover relationships that even domain experts wouldn’t expect.

I created a script called exploration.py which compiles the exploration I’ve done with these datasets. The results are summarized below.

The target variable is 0 (no default) in almost 92% of the training data. Due to the nature of defaults, I would expect this to be representative of the test set too so this is a class imbalance problem. With this in mind, I will likely use a binary classifier. It would make sense to set scale_pos_weight to a high value since there are fewer of these samples in the training set. LightGBM also has an is_unbalance flag that is meant to relieve the user of having to set scale_pos_weight manually. However, through trial and error, I found that the default settings handle the imbalance problem even better than scale_pos_weight or is_unbalance.

There were 38 categorical variables which I one hot encoded. One hot encoding is a technique where each category gets its own feature column. If and only if the base column’s value is equal to that category does the one hot encoding column for that category have a 1 value (otherwise, it gets a 0). One column, ORGANIZATION_TYPE has 58 categories, but the rest have less than thirty. Four of these categorical variables had only 2 categories. I did some analysis that showed that there was no difference between the AUC of the ROC obtained by LightGBM binary categorical variables that were One Hot encoded vs Label encoded. This is probably due to the bucketing nature of LightGBM. Obviously, there is a performance improvement with using less columns but I decided not to write this code since it allowed me to treat all categorical columns alike. Columns with more than 2 categories performed better with One Hot Encoding

perhaps because of the low cardinality of the categories enabled better splitting with simple trees.

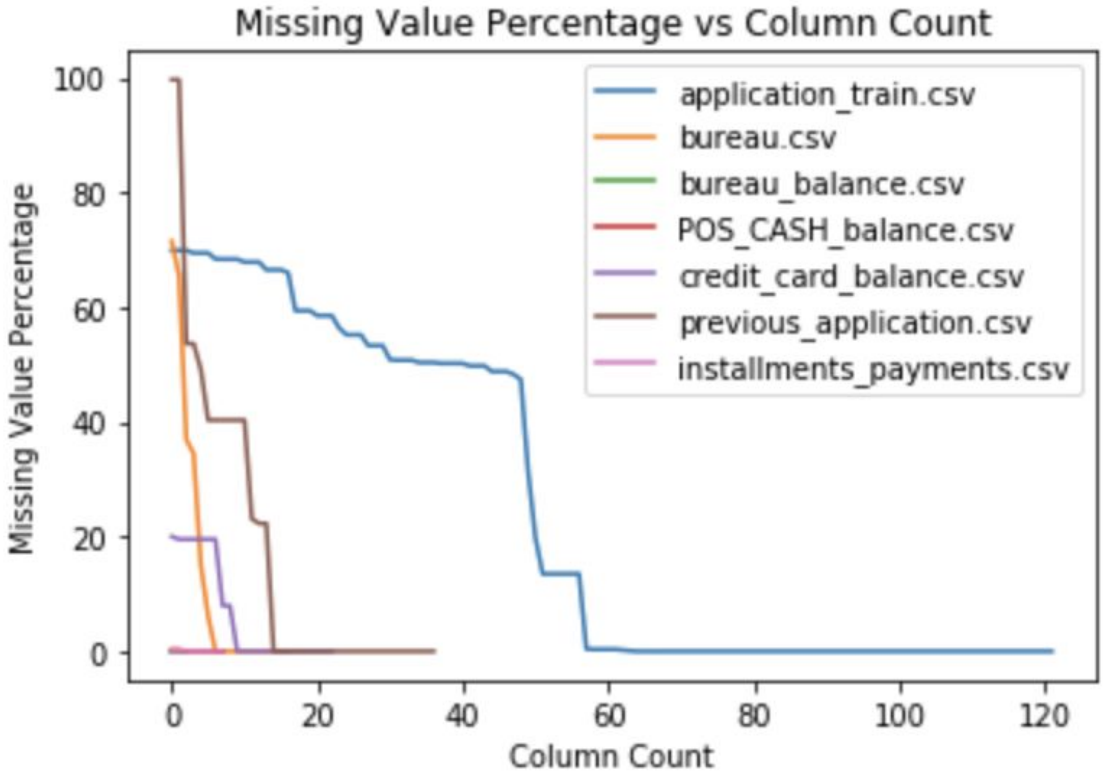
The only non-categorical variable that would probably benefit from explicit bucketing is HOUR_APPR_PROCESS_START but I did not bother to do so because I doubt it is that consequential for the extra effort required to bucket just it.

Missing values within categorical columns received their own one hot encoding column. Missing values within numerical columns were left as nan (not a number) because the best performing algorithm on this dataset, LightGBM, does better in this case than if the nan values are filled with a value. LightGBM handles nan values well because it uses a bucketed approach rather than a linear approach.

With outliers, I wanted to be careful to preserve any information present in the anomalous values. Because LightGBM bins (buckets) variables based upon frequency, having a few extreme value outliers is not a problem. However, having a huge amount of outliers due to a systematic irregularity would skew the binning. Most variables in the dataset have an abnormally high percentage of values at 0 which is properly interpreted as a missing value by LightGBM's default settings. However there are a few variables where the anomalous value does not occur at such a convenient value. Most notably, DAYS_EMPLOYED has 18% of it's values at 365243 whereas the rest of the values are negative. I wrote the function min_max_flags_and_replacements in transformations.py to detect large frequencies of anomalous values, replacing them with NAN (not a number, a.k.a. A missing value) and add a flag column to mark this replacement. This is more valuable than simply clipping off outliers since they do contain information that needs to be preserved. The AUC of the ROC obtained with this technique is higher than without it.

Exploratory Visualization

As I explored the dataset, I wanted to get a good idea of just how many missing values were present in the dataset. This is important because if my datasets contain many sparse features, it would make sense to choose an algorithm that handles missing values well. The following visualization shows the percentage of missing values (nan) present in each file on the y axis plotted against the number of columns in the x axis. Almost half the columns have missing values. And those columns that have missing values are very often missing half their values. This visualization made it easy to see that having an algorithm that handles missing values well is very important on this dataset.



Algorithms and Techniques

Predicting default would seem like a binary classification task. However, the task is actually to quantify default risk because the measurement that the “Home Credit Group” is hoping to maximize is the AUC of the ROC. Luckily, most classifiers can predict the probability that the sample belongs in each class. That way an organization can choose a cutpoint to divide the 2 groups (high risk of default and low risk of default) based on its risk appetite independently of the model producing its predictions.

Due to the number of raw features in the dataset, the number of derived features after transformation, and the number of samples in the training set, a bucketed gradient boosting machine like LightGBM would probably do the best. It’s been my experience that LightGBM can seldom be beat on structured data. Additionally, how LightGBM works with missing values made it a great choice given the high number of missing values in the dataset. I also used many other standard algorithms like sklearn’s LogisticRegression and GradientBoostingClassifier first and found the scores wanting.

LightGBM is a Gradient Boosting Machine. It’s default boosting method, “gbdt”, uses a decision tree to correct the missed predictions from the previous decision tree. With

each subsequent decision tree, the accuracy is “boosted.” Since decision trees are prone to overfitting, its parameters are set by default to prevent complex models from forming. I have modified the default parameters even more to reduce overfitting and increase cross validation accuracy.

Due to the large number of parameters offered by LightGBM (13 Core Parameters, 30 Learning Control Parameters, 12 Objective Parameters and many other parameters that most likely do not contribute to accuracy), I cannot discuss all the parameters. However, I have included the most noteworthy parameters as well as an extended discussion of parameters I thought important to vary from their default values.

The default behavior of LightGBM is to use to the “gbdt” method of boosting I described above to classify samples. Each variable’s values will be bucketed into 255 bins (max_bin). When a split of more than min_data_in_leaf data samples provides a gain of min_gain_to_split, the decision tree will generate another leaf until num_leaves or max_depth is reached. After no more gains can be found, another decision tree is generated which corrects the inaccuracies in the last decision tree up to num_iterations.

I chose to slow down the learning_rate, increase the num_iterations, and set early_stopping_round to find the model that best fits the validation dataset instead of just limiting the num_iterations as a means of preventing overfitting on the training dataset. I set the num_threads to be equal to the number of real CPU cores on my machine as recommended. I lowered the max_depth to 4 to prevent overfitting due to unlimited boosting. I also decreased the bagging_fraction (rows used) and feature_fraction (columns used) to deal with overfitting. I also increased the regularization (both lambda_l1 and lambda_l2) to punish model complexity. I made a min_gain_to_split to prevent splits probably specific to the training set.

In order to generate the predictions for the test data (which lacks a TARGET column), I average the test data predictions generated by each of the folds’ models in the k-fold cross validation. This provides a natural ensemble for the predictions while at the same time generating a cross validation score for the validation set (the training data in the fold left out of training for that split).

Benchmark

It would be ideal to use the FHA loan approval criteria as a benchmark model since it is the industry standard today. Unfortunately, not all the information needed to use that criteria is available in the dataset. However, using just the FHA loan approval criteria

available in the datasets as inputs to the regressor should provide a solid baseline that further models can be compared to.

For a loan to be insured by the FHA, the debtor must have a FICO score > 580 for 3.5% down payment (or between 500-580 for 10% down payment), a debt-to-income ratio > 43%, proof of employment and steady income, to have mortgage insurance premium, and to have the house under loan as their primary residence.

The application's ext_source_1,2, and 3 are presumably the credit scores but they are normalized. AMT_INCOME_TOTAL is the income. Total debt can be added up by summing AMT_CREDIT_SUM_DEBT from bureau.csv and AMT_BALANCE from credit_card_balance.csv. There is no Mortgage Insurance Premium flag or Primary Residence in Home under loan in the dataset.

Since the credit scores are normalized, I cannot use a rule-based classifier for default prediction. Therefore, I used these columns as the input to a Gradient Boosting Regressor (LightGBM) and obtained a a 5-fold cross validated AUC of the ROC of 0.735420. Details of how this was done can be found in benchmark.py. This is a reasonable starting point for comparing more complicated models.

III. Methodology

Data Preprocessing

The first step in preprocessing the data was to explore the relationships between the datasets. Datasets.py contains the script I wrote to open all the datasets provided and deduce the foreign key relationships between them so that their hierarchical structure relative to the TARGET variable could inform how they should be aggregated and joined for training against the target.

I then applied a standard set of transformations to all the columns.

- One Hot Encoding categorical columns
- Adding Flag columns for columns that contained a large proportion of samples at the min or max of the distribution (eg, DAYS_EMPLOYED). This allows the model to correct for outliers in the datasets
- Aggregating columns with similar names (FLAG_DOCUMENT_#, EXT_SOURCE_#)
- Aggregating columns in join tables

- Dividing columns to try to automatically find important relationships between important columns

These steps resulted in a feature set that was entirely numerical and ready to be fed as the input to machine learning algorithms implemented by cutting edge machine learning researches. Furthermore, they contained all the features in all the datasets. In fact, the feature set contained multiple versions of each original column so that a feature selection phase could inform me which version of each column led to the best scores.

Implementation

After preprocessing the data, I fed all my created features into several different standard algorithms using default parameters to get some feedback on my hunch that LightGBM would perform best. I tried LogisticRegression, GradientBoostingRegressor and LightGBM. LightGBM did very well so I decided to focus on it.

The first complication I ran into was installing LightGBM on my mac. Pip install lightgbm gave me a “Library Not Loaded” error so I have cited the Stackoverflow I used to install it below.

I used the LGBMClassifier from LightGBM’s Python API as it has nice interoperability with scikit-learn’s APIs. The predictions from a classifier would yield {0,1} (zero or one). At first this was a problem because it made my ROC curve very jagged and resulted in a poor score. The predict_proba method luckily gives a range of numbers between 0 and 1 inclusive [0,1], which I interpret as the default risk. This gradation of prediction enables many cutpoints to differentiate predictions leading to a nice ROC curve and a better eval_metric=’auc’. I used K-fold cross validation to maximize training data used for fitting the model while retaining a validation set for early stopping to prevent overfitting.

Because I decided to automatically create features, I wrote many functions to do this. They can be found in transformation.py and enumerated in the “Data Preprocessing” section. I am aware of featuretools but wanted to implement this myself to take advantage of some naming conventions in the columns. I also wanted to test out some ideas for automatically generating features not available in featuretools.

Refinement

There were many steps in the refinement of my feature creation, feature selection and model selection processes. Initially, I used only the application_train.csv dataset. I used

only one column, the Target column, to verify that I had everything connected correctly. Using Target as an input resulted in a 1 AUC of the ROC as expected. Then using all features generated from the application_train.csv, the default parameters of LGBMClassifier resulted in an AUC of the ROC of 0.760186.

I then used all the feature columns from all the datasets (files in the input directory). This produced 2856 columns and the AUC of the ROC was 0.781046. This was my initial solution. I knew by the scores reported in the Public Leaderboard (0.809) on Kaggle that I needed to do better. The average score on the training data (0.9330502) was significantly higher than my score on the validation data indicating overfitting. Since I had generated many features from the same column through various aggregation functions, I knew that reducing the total number of features would reduce this overfitting.

The first refinement I tried was making a feature selection process that avoided overfitting. My goals during the feature selection process was to retain or even increase a high roc_auc_score on the cross validation set while decreasing training time. I decided to use my same model on each column individually and store the score in meta/col_effect_full_comma.csv. I then wrote a function called get_best_variants to select the best feature version of each underlying column from the dataset. This reduced my features from 3190 to 446 (86% reduction) also cutting training time by the same fraction without sacrificing much validation score. The AUC of the ROC on the validation set for this model was 0.778485 whereas the average score on the training set was 0.9206322.

Now that I had a reasonably sized feature set, I knew it was time to tune the hyperparameters that defines the behavior of the LGBMClassifier. Even though I reduced overfitting using feature selection, I still noticed the tell-tale mark of overfitting within the debugging output for LightGBM: a large difference between train and validation scores. That’s why the focus of my hyperparameter tuning efforts were to increase validation score by reducing overfitting. A complication that I faced when I tried to use the hyper parameter tuning method sklearn.model_selection.GridSearchCV was that the hyperparameters that it chose were inferior to the hyperparameters I had hand selected using early stopping and a high number of iterations. Because I could not feed the validation set into GridSearchCV, I could not get it to optimize under early stopping conditions. I got around this by using the num_iterations from my early stopping model using a validation set as the num_iterations in GridSearchCV for further hyperparameter tuning. The AUC of the ROC for this model was 0.783868.

A record of these models and their results can be found in implementation-refinement-history.py.

IV. Results

Model Evaluation and Validation

The final model involved incorporating all the refinements that showed a positive effect on the metric important to the business entity backing the competition (AUC of the ROC). All the automatic feature creation discussed in the “Data Preprocessing” section, the LGBMClassifier discussed in the “Algorithm and Techniques” section, and the hyperparameter tuning discussed in the “Refinement” section were used in the final model. I did not use the feature selection discussed in the “Refinement” section because it didn’t result in a higher score on the cross-validation set. I am still glad I did it since it made the hyperparameter tuning complete in a reasonable time.

The model is robust because it was chosen by early stopping on a validation set. The fold scores in a single k-fold cross-validation are not that different from each other showing that small perturbations in the training data don’t change the results much. Early stopping with cross validation gives us confidence that our model can make predictions on data that shares the same source that the training data was derived from (if the training data was randomly sampled).

As a final test, I submitted my solution to the kaggle competition and achieved an AUC of the ROC of 0.777 on the public leaderboard; nearly the same as the score I achieved in the cross validation set. This lends even more credibility to the model.

Justification

The benchmark model I setup achieved a 5-fold cross-validated AUC of the ROC of 0.735420. The final refined model scored 0.783868. This is definitely a statistically significant improvement. It makes intuitive sense that a higher AUC of the ROC enables a business to identify loans with higher default risk enabling them to avoid future default losses (maximize true positives) and similarly recognize applications that seem risky but have a low default risk according to a model that performed well on historical data (minimize false positives).

Despite the logical above, I wanted to empirically justify this hand waving. I wrote the script free-form-visualization.py that attemps a “back of the envelope calculation” of the

gains and losses from every loan in the training set given various models' predictions at many cutpoints. The result of all these calculations is an excellent visualization that you can see in the "Free-Form Visualization" section that shows exactly how this model can profit the business who implements (and, roughly, by how much).

Fortunately, it shows how thousands of dollars per application can be earned by using the model to make underwriting decisions. However, it also shows that almost 10 times as much room for improvement is available between my model and perfect predictions as I gained over the benchmark. I have made an improvement, but this problem is by no stretch of the imagination solved.

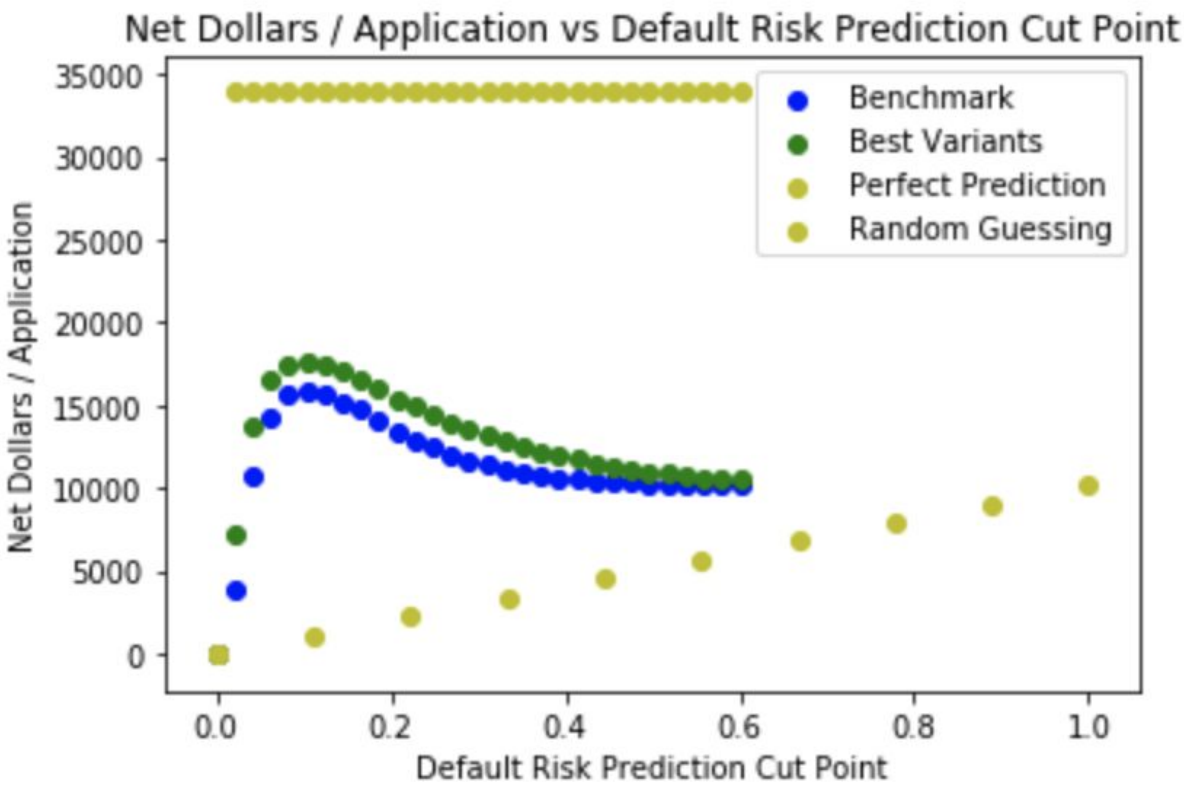
V. Conclusion

Free-Form Visualization

The most important part of this research project is how my work translates into more money for the corporation that uses it.The following visualization shows the net dollars per application as a function of the default risk prediction cut point the corporation can expect if they use a certain model. You can see in the visualization below that perfect prediction yields the maximum net dollars per application regardless of the cut point because all applications that resulted in loans default have default risk of 1 and all applications that resulted in loans that didn't default have a default risk of 0. A randomly chosen default risk yields the best return when all applications are accepted. This is also the asymptote for all prediction models because a default risk cut point of 1 means that all applications are accepted whatever the model.

The shape of the benchmark and Best Variants prediction models are skewed left. They show that too low of a risk for defaults is heavily penalized since the vast majority (92%) of applications turn out to be loans that don't default. The bump in the middle shows that there is some optimal cut point such that the most money is made per application. This is because accepting too many applications results in accepting too much default risk and not accepting enough applications results in not enough reward. For example, $1,711 per application is earned by using my best variants model over the benchmark when using the best default risk cut point for each model. However, if perfect predictions can be made $16,396 per application can be earned over the earnings from my model. This shows the theoretical room for improvement that might spur business investment into further model improvements.

This can be reproduced by running free-form-visualization.py.



Reflection

In summary, I predicted “Home Credit Default Risk“ using the related datasets provided on the kaggle competition using a gradient boosting decision tree configured for classification to optimize the Area Under the Curve of the Receiver Operator Characteristic. I chose LightGBM based the sparsity, structure of the dataset, and it’s score on a k-fold cross-validation. I modified columns to handle anomalous values, created polynomial features, and aggregated features from sets of related features, and aggregated features that had foreign key relationships with the TARGET variable.

The aspect of this project that I found most interesting was trying to separate my implementation from this specific dataset so that it could work for any structured datasets. This was difficult to do and I even had to add the ability for the user of my generic process to manually create features since domain knowledge is really critical to feature creation with structured data.

Of course, I had hoped that I would have a top score on kaggle for this competition, but I am pleased that I did so well given that I tried to automate the entire process. Because I built my library to be used in a general setting to solve similar types of problems with

relational and structured data, I think it can both provide a good baseline and benchmark for other problems and as a framework for building more advanced models using domain knowledge to manually create features.

Improvement

It's clear from the AUC of the ROC that I obtained that there is plenty of room for improvement. Not only does it fall short of perfection (AUC of the ROC of 1) but it also falls short of the leaders on public leaderboard for the kaggle competition.

After all my reading about kaggle competitions on structured data, I believe the biggest improvement I could have made was by consulting a domain expert for manual feature creation. During the project I tried to overcome this deficit by using automated techniques, but it is clear that I have not yet achieved that goal.

I have several additional ideas for automatic feature creation that I did not have time to implement. There are clearly relationships between the variables in the datasets implicit within the naming conventions used. I wrote some functions to group columns based on these conventions but had to abandon the last mile of creating features based on the groups due to time constraints. I also wanted to use information present in the variable descriptions to group and aggregate columns to create new features in an automated fashion.

Adding these features would not have significantly changed my feature selection or training process but I believe they would have improved the score.

I could have made my solution more general by making a cleaner separation between the reusable parts and the project specific parts. I would have liked to generalize my solution to regression tasks and different metrics as well. Ideally, the process would be nearly fully automated for future users too.

I would have also liked to use Bayesian optimization for hyperparameter tuning instead of the brute force approach of GridSearchCV. I have never used Bayesian optimization before so I did not have time to learn it as well as implement it.

Finally, I would have liked to use features in the datasets in combination with data outside of the datasets provided in the kaggle competition (alternative data) to create new features. I have never done this before nor have I seen it done for structured data like this. I also felt it required domain expertise that I didn't have.

I think I could easily spend the next few months working through these ideas for improvement. Luckily, I can still test myself against the cross-validation set even after the Kaggle competition and the Udacity term ends. I am excited about what I have achieved in this project and am looking forward to continuing developing these ideas further.

References

Kaggle.com 2018, Home Credit Default Risk Data Description, viewed 30 June, 2018, <https://www.kaggle.com/c/home-credit-default-risk/data>.

FHA.com 2018, FHA Loan Requirements in 2018, viewed 23 July, 2018, <https://www.fha.com/fha_loan_requirements>.

Stackoverflow.com 2018, Lightgbm OSError, Library not loaded, viewed on Aug 5, 2018, <https://stackoverflow.com/questions/44937698/lightgbm-oserror-library-not-loaded>.