Springleaf Marketing Response

Ji Yang, Ka Ki Lai

Dec 17, 2015

I. Introduction

Springleaf puts the humanity back into lending by offering their customers personal and auto loans that help them take control of their lives and their finances. Direct mail is one important way Springleaf's team can connect with customers whom may be in need of a loan.

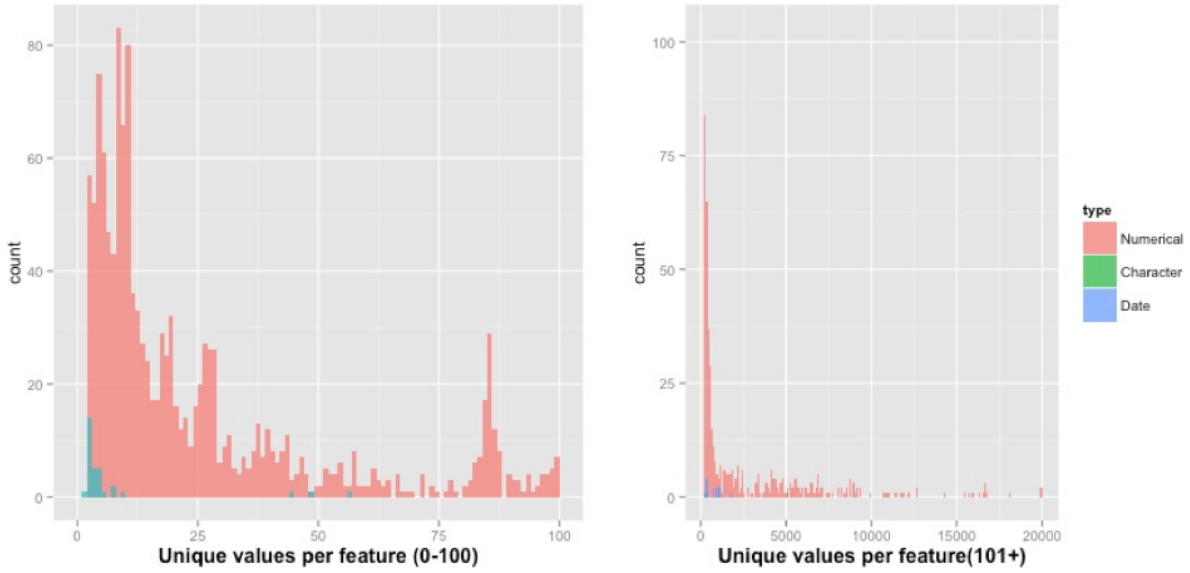
Direct offers provide huge value to customers who need them, and are a fundamental part of Springleaf's marketing strategy. In order to improve their targeted efforts, Springleaf must be sure they are focusing on the customers who are likely to respond and be good candidates for their services.

In this project, using a large set of anonymized features, we aim to predict which customers will respond to a direct mail offer. We need to construct new meta-variables and employ feature-selection methods to approach this dauntingly wide dataset.

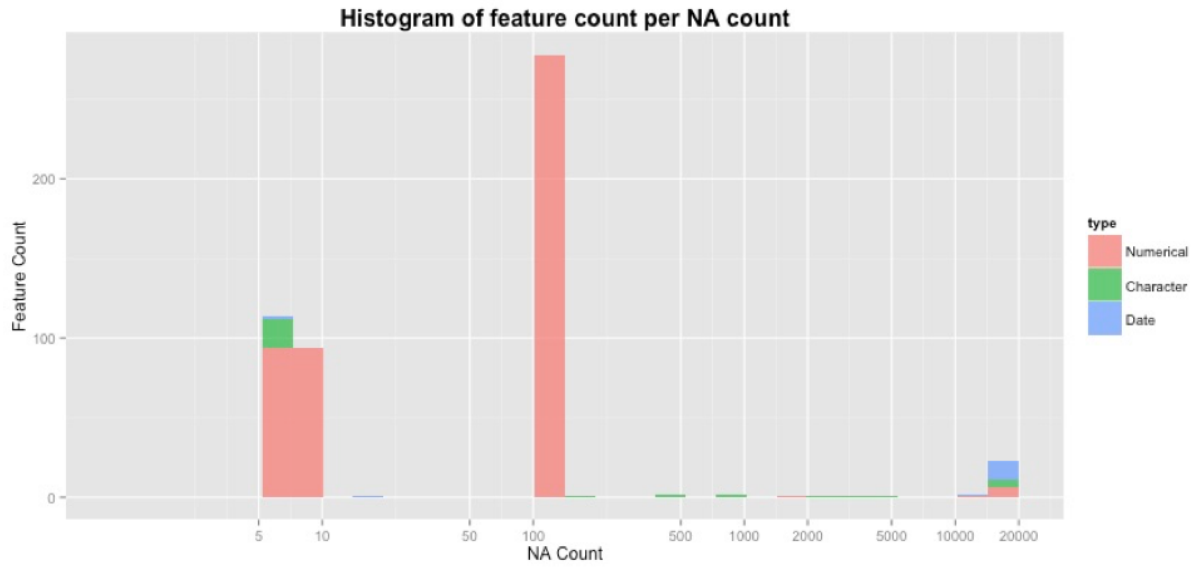
II. Exploratory Data Analysis

We explore and play around with the data before actually doing feature engineering and classification. We compute some important statistics of the dataset. For example, the NA rate over the whole dataset is about 0.57%. There are no duplicated rows. 5 features are constant and can be safely removed. There are 1876 numerical features and 51 character features.

Within the character features, NA values are marked as blank, [] or -1. We can explicitly change them into something else like -99999, just to differentiate them with other meaningful values.



We look at the number of unique values of each feature.



We also look at the number of NA values of each feature.

II. Data preparation (Feature engineering)

1. Categorical feature conversion

There are originally 54 categorical features. After exploring what values the categorical features have, we work on converting categorical features to numeric. The first step we take is to standardize all NA values. As we talked about before, NA values are represented by blank, [] and -1. We firstly change them all into -99999, in order to differentiate them with known values. Then we create dummy variables for each value of each categorical feature. However, we do the low frequency combination, so as to reduce the number of dummy features we create. We set a threshold for low frequency, empirically from 0.001 to 0.05. Specifically, all values that appear less frequent than the threshold will be regarded as the same value in one single dimension of dummy feature. When theta is 0.05, we created 121 dummy variables and when theta is 0.001, we created 1296 dummy variables. This threshold is a tunable parameter.

2. N/A value handling for numeric features

There are 1.6M NA values in the 1878 dimensional numeric features. We create dummy variables for these NA values. Specifically, for each feature, if there is a certain amount of NA values, we create one dummy variable which has 1 for those NA positions and 0 for non-NA positions, and we change NA in the original feature into 0. We also set a threshold for the amount of NA values for creating dummy variables, empirically from 0.001 to 0.05. This is also tunable. When the threshold is 0.05 we get 8 dummy variables and when it is 0.001, we get 286 dummy variables. We also notice that there are some absurd values that are used probably to mark NA values, such as 999999, 9999998, but we have not specifically dealt with them yet.

3. Irrelevant feature removal

We remove irrelevant features. Features with constant values are definitely irrelevant and safe to remove. There are 36 of them to be removed. We also remove features with low variance. However, we consider the std/mean ratio rather than variance, because scales should not influence the decision. We set a threshold theta for this low variance, empirically in the range between 0.001 and 0.05. This is a tunable parameter. When theta is 0.05, we go 109 features to remove. When theta is 0.001 we got 17 features to remove.

4. Dimension reduction through correlation analysis

We conduct pair-wise correlation analysis on features. We set a threshold theta and remove one of the two features if the absolute value of their correlation is larger than 1-theta. Theta is empirically set from 0.001 to 0.05, which is also tunable. When theta is 0.05, we remove 782 features, when theta is 0.001, it is too time consuming and we have not got a result.

5. Summary

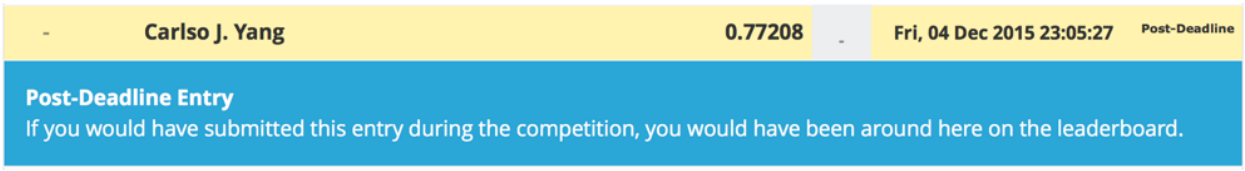
Overall, we have 4 parameters to tune for the feature engineering step, empirically each of which in the range of 0.001 to 0.05. We use cross-validation to tune the parameters within {0.001, 0.005, 0.01, 0.05, 0.1} and found the best value to be 0.05, for all 4 parameters. Due to time limit and the limited influence of feature engineer on this specific dataset, we did not tune each of them separately.

III. Learning paradigm and Results

1. Xgboost (R)

Algorithm Introduction

We use **xgboost package in R** for learning the classifier. We find that with default setting of step size, xgboost can keep increasing the performance on training set even after 500 iterations, but the increase on test set stops at around 200. The algorithm runs quite fast, i.e., it takes about one hour to train once on the whole training set, and takes several minutes to predict on the whole test set. The best results with intuitive setting of parameters (0.05 for thresholds and default values for xgb), as we reported in the midterm report is as following.



After midterm, we worked on tuning the parameters of feature engineering as well as xgboost, using cross-validation.

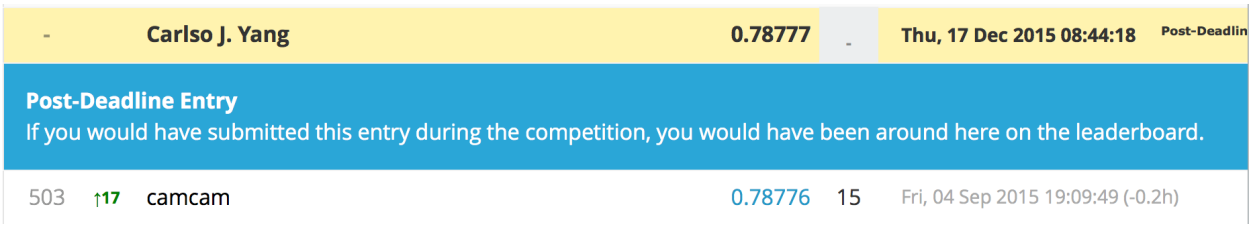
As we discussed before, we use cross-validation to iteratively tune the parameters involved in feature engineer and the parameters of classification, in this case, xgboost. The parameters in feature engineer are theta_1, the threshold of low frequency combination, theta_2, the threshold of the frequency of NA

values, theta_3, the threshold of low variance and theta_4, the threshold of high correlation. We find that feature engineering does not contribute much to the final performance, mainly because the features are very ad hoc and the number of features is quite large. There we tune the 4 parameters together, i.e., we always set them to the same value, rather than tuning each of them separately. The values are {0.001, 0.005, 0.01, 0.05, 0.1}. The smaller the value we set, the less low frequency (variance, correlation) filtering we do, and the more features we end up with. For example, when we set the parameters to 0.001, we get a total of 2510 features, and when we set the parameters to 0.05, we get a total of 930 features. However, during the cross-validation, we find that the correlation analysis is very time consuming, so we commented out it for most cases. Without correlation analysis, we end up with 1858 features with parameters set to 0.05 and 3407 features with parameters set to 0.001. Surprisingly, we found that features without correlation analysis actually lead to better predictions. The reason might be obvious though. When doing correlation analysis, even when the parameter is set to 0.001, we deleted about ⅓ of the features, which still leads to a certain loss of information, thus resulting in poorer prediction power. Therefore, we ended up using the 1858 features generated by all parameters set to 0.05 and without correlation analysis.

For xgboost, the parameters we tuned include the maximum depth of the boosting trees (max.depth), the number of iterations (nround), the step size (eta) and the resampling ratio (subsample). We tune them because we think they are the most influential parameters of xgboost. The default value for max.depth is 6, for nround is 200, for eta is 0.3 and for subsample is 1. We basically want to try smaller step size and larger nround, because reducing the step size might lead to more precise optimals especially when the training set is so large. In order to compensate the smaller step sizes, the number of iterations need to be larger. We also want to try lager depth of the trees, because we have so many features. Nevertheless, we want to try resampling with a ratio smaller than 1, so we can borrow the idea of bootstrap into this bagging algorithm, to add more

randomness and thus reduce prediction variance. However, we found that 5-fold cross-validation on the whole dataset with many values for each parameter is almost impossible in time, so we begin with very few candidate values, i.e. two or three for each parameter. Since each run takes a relatively long time, we need to set those candidate values really wisely.

We plan to do cross-validation for several times, each time we set the candidate values based on the results of the previous rounds. Specifically, we first intuitively set the candidate values for eta to be {0.01, 0.05, 0.1, 0.2}, for max.depth to be {8, 10, 12}, for subsample to be {0.8,0.9}, for nround to be {300, 400}. Even if we choose candidates wisely, 5-fold cv on the whole dataset is still almost impossible. So we did sampling on the training set, i.e. we randomly sampled 20,000 observations from the training set and we only conducted 2-fold cv. Thus the cv on such a candidate set took about 8 hours, and the best parameter values we get are 0.05 for eta, 12 for max.depth, 0.9 for subsample and 400 for nround. We use this set of parameters to train on the whole training set and predict on the whole test set and pushed our performance to 0.78777, which is around 500th position on the private board of Kaggle. We also used this set of parameters to train on other feature set we got using different parameters for feature engineering, but the results are not as good as this one.



We set the second candidate set of cv based on the best parameters we got from the first round. Specifically, we set the candidate values for eta to be {0.04, 0.06} which covers 0.05, for max.depth to be {10, 14} which covers 12, for subsample to be {0.8,1} which covers 0.9, for nround to be {300, 500} which covers 400. The best values we got are 0.04 for eta, 14 for max.depth, 1 for subsample and 500 for nround. We use this set of parameters to train on the whole training set and

predict on the whole test set and pushed our performance to 0.78377, which is even lower than what we got before.

We set the third candidate set of cv based on the best parameters we got from the first and the second rounds. Specifically, we set the candidate values for eta to be {0.03, 0.04,0.05}, for max.depth to be {12,14,16}, for subsample to be {0.9,1}, for nround to be {300,400,500}. The best values we got are similar to what we got in the second round of cv, which makes us quite confident about the values we choose before.

2.Random Forest (Python)

Algorithm Introduction

Random forests is an ensemble method of classification which combines k base classifiers to create an improved composite model for classification. For a given data set, k training data sets would be generated to return k learnt models. For a given data point, each of the classifier would return a class prediction. The ensemble will then make prediction on the data tuple according to the overall probability of class membership based on this composite classification model.

In the Random forest algorithm, each decision tree is built by a random selection of attributes at each node to find the best split. By introducing randomness in best-split selection, since the split in each node are based on different predictors, correlation between trees built is reduced. Since Random forests would consider fewer features at each split during the process of tree building, the algorithm is more efficient especially toward large datasets.

Methodology

The algorithm is mainly conducted in Python while the feature engineering is conducted in R. To implement the random forest algorithm, the selected training features and the test data were first converted to .csv format in R before being read into Python for implementing the Random forest algorithm. To build the Random forest, the {RandomForestClassifier} from package {sklearn.ensemble } is used. To generate random samples to build tree, {shuffle} from package {sklearn.utils} is applied.

Pseudo code:

for i *in* $1 : N_{tree}$ **do**

Randomly bootstrap sample of data points from *train.csv*.
Build Random Forest Tree, $Tree_i$ based on this subset by repeating following steps in each terminal node:

- select $m*$ features at random from all m features in the data set
- pick the best feature, M_k based on information gain for the best split.
- Split the node by M_K into two daughter nodes

Predict the class probabilities of the test samples where the predicted class probability is the fraction of samples of the same class in a leaf.

end

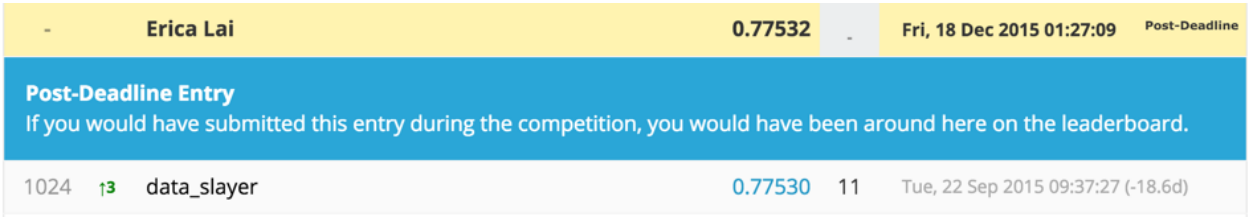
Return the average class probability of all the trees
Classification.

The parameters we tune includes number of tress built (n_estimators), functions to measure the quality of the split (criterion), number of features to consider when looking for the best split (max_features). The result are shown below:

Model	**Criterion**	**N_estimator**	**Max_features**	**Score**
M1	Entropy	500	Log2(p)	0.76940
M2	Gini	500	Sqrt	0.76956

M3	Entropy	800	Sqrt	0.77528
M4	Gini	800	Sqrt	0.77004
M5	**Entropy**	**1000**	**Sqrt**	**0.77532**

The best result is M with a score of 0.77532.

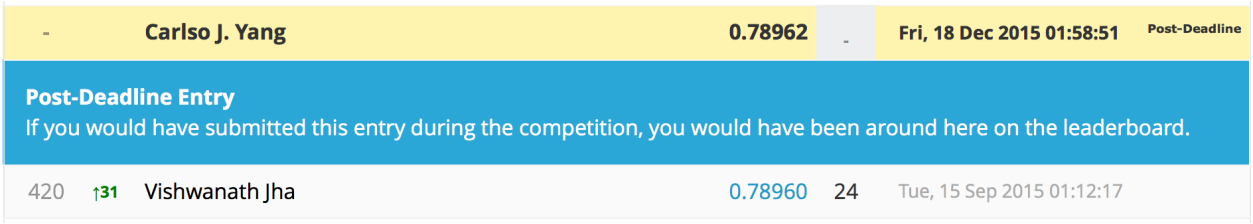


It looks like use of criterion and max_features give highly similar prediction. While higher number of trees built could return a better prediction (i.e. M5 perform better than M3). One of the reason could be due to the large size of training sample, larger number of trees built would be more accurate for prediction of large data set. However, it is worth noting that the large number of trees built would lead to long computation time of the algorithm. On average, it takes roughly 30 minutes for building 500 trees and 120 minutes for building 1000 trees.

3. Ensembling Xgboost and Random Forest

Finally we ensemble the predictions of Xgboost and Random Forest. Ensembling of different algorithms is known to be working well on improving overall predictions, because it combines the advantages of various algorithms. More intuitively speaking, different algorithms have different emphasizes and assumptions, and ensembling them together usually alleviates the extreme situations and can yield better overall results.

We simply ensemble the predictions on test data using weighted sums. Since the results of our xgboost model is better than random forest, we empirically tried the weight combinations of {0.5,0.5}, {0.6,0.4},{0.7,0.3} and (0.8,0.2}, where the former one is for xgb and the latter one is for random forest. We found that the weighting of {0.7,0.3} gives us the best performance.



III. Discussion

The lessons we learned from this project are diverse.

The first thing is about feature engineering. NA values are sometimes useful, so it is usually unwise to ignore them or just set them to zero or mean values. In the real world, NA values are not just NA, but also various absurd values. Categorical values are hard to deal with, because they can explode into lots of dummy variables. A common way is to do low frequency combination. It sometimes also makes sense to just delete some of the categorical values, such as job description, because it is too hard to extract useful information from such features. Date-time features usually need to be specifically taken care of, because they can always be converted to more meaningful formats and due to different datasets and tasks, they can somehow become very important. It is good to start with careful feature engineer. However, sometimes they just do not count much, especially in an ad hoc dataset like the one in our project. It might be easier to just put everything into a black-box complex algorithm and let the random processes and step-wise updating of the algorithm to figure out the importance and weight of each feature. This is especially true for correlation analysis. We spent a lot of time on this part, but we ended up abandoning it completely. An intuitive explanation for it might be, the bagging algorithm can easily deal with the correlations, because they always randomly choose a subset of features to split on, especially when the features are correlated, and overall the correlations are cancelled out. However, this might also lead to a distribution of weights on correlated features, and thus the really important features may not stand out. Therefore, if we want to better interpret the models, rather than just need a final prediction, it is still useful to do feature engineering such as correlation analysis.

The second thing is the parameter tuning. When the data is large, we need to wisely select the candidate values and do cross-validation iteratively. We also need to really know our data and understand the meaning and influence of each parameter in order to tune wisely and efficiently.
The third thing is about iterative tuning of feature engineer and learning algorithm. This two part although are separate, they are actually correlated with each other. So in order to further push up the performance, one may want to iteratively tune the two processes. We simulated this iterative process incompletely by always trying the best parameters returned by cross validation on a specific training set on other training and test set. This process is certainly not totally correct, but it nevertheless gives us more chance to get better results.

IV. References

i. John Duchi, Elad Hazan, and Yoram Singer, Adaptive Subgradient Methods for Online Learning and Stochastic Optimization, JMLR 2011 & COLT 2010

ii. Andy Liaw, Matthew Wiener. Classification and regression by randomforest.

iii. Leo Breiman. Random forests.

iv. Jerome Friedman. Greedy function approximation: a gradient boosting machine.

v. Jerome Friedman. Stochastic gradient boosting.

vi. Scikit learn user guide http://scikit-learn.org/stable/user_guide.html.

vii. pandas: a Foundational Python Library for Data Analysis and Statistics.