MSDA 621 Business Analytics Final Project

Sekhar Mekala, John DeBlase and Sonya Hong

Thursday, December 08, 2016

Abstract

Allstate Insurance has published a Kaggle competition challenging interested data scientists to accurately predict the severity of a claim, based on 130 predictor variables. The accurate prediction of the claim severity will help Allstate to assess the impact of the loss on the insured (and his/her family members), so that they can speed up their service to achieve better customer service. In this paper we discuss how Boosting and Rule Fit techniques can be used on this data set to identify important variables affecting the target variable. We will show that these 2 methods have identified almost the same variables and finally show that Boosting has provided a computationally efficient and accurate prediction, when compared to Rule Fit method for this data set.

Key words and Phrases: Allstate loss prediction, Allstate Kaggle competition, Boosting, Rulefit ensemble

Introduction

Allstate is currently developing automated methods of predicting the cost, and hence severity, of claims (Kaggle.com). In this kaggle competition, the participants have to create an algorithm which accurately predicts claims severity.

Allstate has provided 2 data sets: *train* and *test*. The *train* data set must be used to train the models and the *test* data set must be used to make predictions and the obtained predictions will be submitted to Kaggle for scoring the efficiency of the predictions. Since we will be predicting a continuous valued variable (*loss*), this prediction problem can be classified as a regression problem[4].

The main objectives of this project are listed below:

- Identify the significant variables that influence the target variable, using Boosted Regression Trees (BRT[1]) and Rule fit method[2]
- Determine whether the significant variables identified by both the algorithms are the same or different.
- Measure the algorithmic performance of the BRT and Rule fit based on their run times and the cross validation results, and propose the best algorithm for the given data set to accruately predict the target variable.

Literature Review

Regression models belong to a class of machine learning techniques where the dependent variable is quantitative in nature. The two main reasons for developing any predictive model is *prediction* and *inference*. If *inference* is our goal, then we tend to develop less complex models which are highly intrepretable, but lack predictive accuracy. On the other hand if we are more interested in the accuracy of the results and interpretability is not our primary motivation, then we can build more complex models to achieve higher accuracy in our predictions. In the real world we often need to rely on complex models in order to accurately predict the behavior of a complex phenemenon.

For any machine learning algorithm, there are two types of errors: *bias* and *variance*. *Bias* is the error caused due to over simplification of the real world phenemenon and *variance* is the error due to high variability of the model. If the form of the model changes drastically, had the model been trained on a different sample, then such model is said to have high variance. A *Bias-variance trade off*[4] occurs when building any model using machine learning algorithms, since as a model grows more complex, the variance tends to increase while bias decreases, and as the variance decreases, the bias will increase. The optimal strategy when choosing a machine learning method is to choose an approach that will minimize both the bias and variance.

For this project, we will place more emphasis on the prediction accuracy than on the model interpretability and hence we use non-parametric models (Boosting and RuleFit methods) to implement our predictive algorithms. In parametric methods, we assume a functional form of the predictive model, and optimize the coefficients (or parameters) using the given data. In non-parametric methods we do not assume any functional form, and let the machine learning algorithm work on the methodology to achieve the best possible prediction accuracy. Given the nature of the data set for this problem (with 132 variables, and many categorical variables with several levels), we will use the following non-parametric methods for developing the predictive models:

1. Boosted Regression Trees (BRT[1])
2. Rule Fit[2]

The BRT (Boosted Regression Trees) (see J. Elith et al. 2008), is an ensemble machine learning algorithm, which iteratively develops decision trees of a pre-defined depth, in such a way that each tree in the current stage is built using the residuals of the trees built in the previous stages. The final BRT model is a linear combination of many trees that can be thought of as a regression model where each term is a tree. The importance of the tree is controlled by a parameter called the *shrinkage* or *learning rate* that indirectly controls the number of trees needed to minimize the test error. At each stage, observations are randomly sampled from the training data to build a tree. J. Elith et al. (2008), has successfully applied the BRT method on a data set related to ecology. The size of their data set is 13369 observations. The BRT method was applied by randomly choosing 1000 observations from 13369 observations and it was shown how the performance of the BRT models change as parameters like tree depth and learning rate vary. In this paper we will apply the BRT algorithm on the training data set. Since the size of the trainig data set is big (188318 observations), we randomly select 30000 observations to estimate the optimal parameters that can be used to build the final BRT model on the complete training data set. This technique will reduce the computational time to determine the optimal parameters to be used for BRT algorithm. This method is similar to the method used by J. Elith et al. (2008). For our models performance evaluation we use Cross Validation method, while J. Elith et al. (2008) have used a test set method to estimate the test error (they built the BRT model with 1000 observations and the model was evaluated using the remaining 12369 observations in the data set).

In the Rule fit algorithm (Friedman and Popescu, 2005), we will develop decision trees in an iterative fashion (similar to the BRT), extract rules from each of the decision trees, and finally fit a regression (lasso) on the extracted rules. The extracted rules are considered as the edges of the decision trees, and each rule would output a TRUE/FALSE value, based on the input data. In this algorithm, the tree depth (which represents the interactions) is not fixed, and is randomly generated using exponential distribution (based on a desired mean). This approach will help to average the complexity of the model since each tree can have a random depth. We will use the same rule based method discussed in Friedman and Popescu (2005), to fit a Rule based model to our training data.

Methodology

Data exploration and model building strategy

The training data set has 188318 rows and 132 variables (including the observation's unique identifier, the independent and the dependent variables). The unique identifier variable (named *id* in the training data set) will not be used for developing the predictive models. The training data set has 130 independent variables, out of which 116 are categorical and 14 are continuous. The *loss* variable is the dependent variable, and our goal is to predict the *loss* variable as accurately as possible. The training data set does not have any NA values. The variable details of the training data are displayed in Appendix-A, Figure A-1. Some of the categorical variables in the training data set have more than 2 levels (the highest being 326 levels for *cat116* variable). If we have to use linear regression, then we need to create dummy variables to handle 116 categorical variables. For instance, to handle the 326 levels of *cat116* variable, we have to create 325 dummy variables (one level will be held), each dummy variable representing one of the levels of *cat116*, and if all the dummy variables have 0, then it is regarded as the presence of the held level. Based on this methodology, we would have to create 1023 additional variables to handle 116 categorical variables. This would greatly increase the complexity of the data set to fit parametric models using linear regression. Therefore we will use methods such as *boosting*, and *rulefit*, which can handle categorical variables automatically without creating the dummy variables. If the performance of our predictive models (based on *boosting*, and *rulefit*) is poor, then we will consider using other algorithms such as K Nearest Neighbors, Random Forests etc.

Our strategy to build predictive models is given below:

1. Fit a model using boosting, identify the significant variables and the cross validation (CV) error. Call this model as *Boost-1*
2. Fit a rule fit model, identify the significant variables and get the cross validation error. Call this model as *Rule-Fit-1*
3. Compare the significant variables obtained in *Boost-1* and *Rule-Fit-1* models, and determine if the variables are the same or different
4. Compare the *Boost-1* and *Rule-Fit-1* models performance based on their runtime and CV errors and choose the best model based on these metrics.

The *test* data set has 125546 observations and all the variables of training data set (except the *loss* variable) are also present in the test data set. Like the training data set, the test data set also does not have any NA values. We have to predict the *loss* value using the test data and submit an evaluation file with *id* column of the test data and the predicted value of *loss* variable to Kaggle for scoring. Kaggle evaluates our predictions using mean absolute error.

Experimentation and Results

Building *Boost-1* model

Boosted Regression Trees (BRT) are a linear combination of hundreds to thousands of trees that can be thought of as a regression model where each term is a tree (J. Elith et al. 2008). This machine learning algorithm can be used for both regression and classification problems. Similar to regression models, the BRT models are prone to overfitting. This occurs if a greater number of trees are used than required, similar to how a regression model overfits the data if more terms are included in the model than required. The BRT algorithm has the following important parameters, which need to be carefully chosen to avoid the overfitting problem. We will use Cross Validation(CV) technique to determine these optimal values for our train data.

tc: This parameter refers to the *tree complexity*, which controls the depth of the decision trees. Using this parameter we can state to what extent the interaction between the variables have to be included.

lr: This parameter refers to the *learning rate*, which controls the contribution of each tree to the final model.

nt: This parameter refers to the *number of trees*, which controls the number of trees to be built in the model.

Usually as the *nt* value increases the training error and the test error decreases. But after reaching a certain value, the training error tends to decrease, but the test error will increase. The *nt* value at which the training error decreases, but the test error increases is called the optimal *nt* value. We address this value as *ont* (*optimal number of trees* for the given data) in this discussion. We will choose the *ont* value based on the cross validation technique for the given data set. The *ont* value is based on the *tc* and *lr* values. Different values of *tc* and *nt* will result in different least possible cross validation errors, and hence different *ont* values. If the *lr* is too low, then more number of trees are needed to obtain the least possible CV error, and hence more computation is needed to determine the *ont* value. But if the *lr* is too high, then only a small number of trees are built, and the least CV error is achieved quickly (although this CV error is usually more than the CV error of the model where *lr* is low). Similarly a greater value of *tc* parameter makes the algorithm to achieve the least CV error faster, but this CV value is usually more than the CV value with a less *lr* value. As per J. Elith et al. 2008, it is advised to build at least 1000 trees in the BRT model, by choosing an optimal *lr* and *tc* value. Therefore we will tune the *lr* and *tc* values in such a way that the least possible CV error is achieved for *ont* of greater than 1000. In order to make the algorithm computationally feasible, we will build a maximum of 1500 trees.

To experiment with different values of *nt*, *tc*, and *lr*, we will randomly select just 30000 observations from the training data set, since each iteration on the whole data set will run for a considerable amount of time (with *lr*=0.001 and *tc*=5, the algorithm ran for 3.5 hours, when the complete training data was used). Running the algorithms on a fraction of the data will help us to estimate the optimal parameters faster. Therefore, our main aim is to get an estimate of the best parameters using the 30000 observations, and apply the same parameters on the whole data set to build the final boosting model (*Boost-1* model). The *Boost-1* model can also be used to obtain the importance of the variables.

We will examine a set of *lr* and *tc* combinations to obtain the CV and *ont* values in each scenario. The *nt* is assumed to be 1500. If the obtained *ont* is equal to 1500, then it signifies that the least possible CV has not been achieved within 1500 trees, and more number of trees are needed to reduce the CV. We will discard the parameters, which resulted in more than 1500 trees from further consideration.

Table-1: CV values for various combinations of *lr* and *tc*

nt	lr	tc	CV	ont	Imp_var_cnt	Minutes
1500	0.100	1	4807191	126	24	5
1500	0.050	1	4744026	236	24	6
1500	0.010	1	4788385	1322	32	5
1500	0.005	1	6803807	1487	23	5

nt	lr	tc	CV	ont	Imp_var_cnt	Minutes
1500	0.001	1	5907541	1500	6	5
1500	0.100	5	4359167	67	36	21
1500	0.050	5	4287151	144	42	22
1500	0.010	5	4265894	776	60	20
1500	0.005	5	4198831	1499	56	20
1500	0.001	5	5109777	1500	29	23

Table-1 shows that as the *tc* value increases (or tree complexity increases), the CV error decreases. Also we can see that as the *tc* value increases from 1 to 5, the run time also increases (almost by 4 times). Hence we can conclude that as the number of interactions increase, the model building time also increases significantly. For the 30000 observations, there is no significant change in the run-time with the decrease in *lr* value. But as the size of the data increases, the runtime might increase considerably with the decrease in *lr* value and due to limitation in our computational resources we do not investigate this any further in this project.

For *lr*=0.001, the convergence did not happen, and more than 1500 trees are needed to fit the BRT on the data. So we will exclude the *lr*=0.001 option from further consideration, since our requirement is not to build more than 1500 trees, to make our solution computationally feasible. The following plot shows the effect of *lr* and *tc* values on the CV error and the number of important variables determined.

Figure-1: Effect of learning rate (*lr*) and tree complexity (*tc*) on the Cross Validation (*CV*) and the number of significant variables

From Figure-1(a), we can infer the following:

1. As the *tc* (or number of interactions increase) from 1 to 5, the *CV* error has decreased. This suggests that we should use *tc*=5 in our final model, since there might be some interactions between the variables in the data set.

2. For *tc*=1, as the *lr* increases, the *CV* error decreases, suggesting that over fitting is occurring at lower *lr* rates, since more trees will be constructed at lower *lr* values. Hence, if we do not use any interactions (or just build trees with two nodes), then it is suggested to use higher *lr* values, to avoid overfitting.

3. For *tc*=5, as the *lr* decreases, the *CV* error also decreases, suggesting that more number of trees should be built if we use interactions, and a small *lr* value will build more number of trees.

From Figure-1(b), we can infer the following:

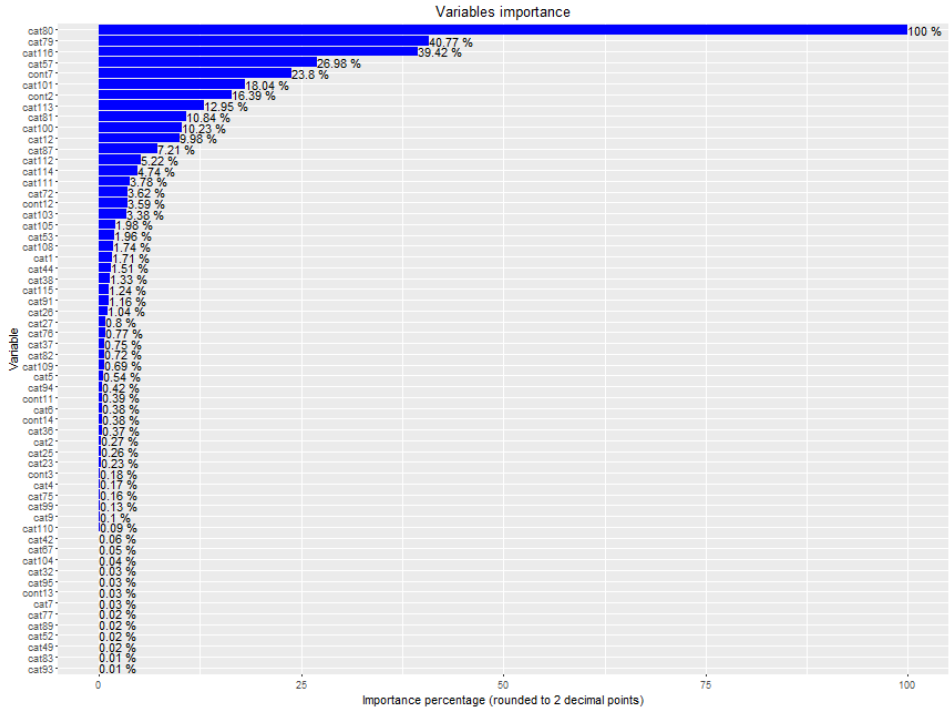
1. As the number of interactions (tc) increase, the number of significant variables also increase, since more interactions would be uncovered resulting in the inclusion of more significant variables in the model.

Using the 30000 random observations, we found that we should use tc=5 and lr can be 0.01 or 0.005 (since the CV is not very different between lr=0.01 and lr=0.005 for tc=5). But lr=0.01 will run faster than lr=0.005. So we will use tc=5 and lr=0.01 to train our BRT model on the complete training data set. Also, for our BRT model on 30000 observations with lr=0.01 and tc=5, we obtained optimal number of trees (ont) as 776 trees. If we train our model on all the 188318 training observations, we should get more than 1000 trees and as per J. Elith et al. 2008, we should try to build at least 1000 trees in our BRT model for improved model performance.

Using lr=0.01 and tc=5, we built *Boost-1* model on the complete training data set. This model has given a CV error of 3960867 with ont of 1264 trees. The process has ran for *3 hours* approximately. Additionally we ran the BRT algorithm on the complete training data set with tc=6 and lr=0.01, and this parameter combination has given a CV error of approximately 3891753, and ont of 1209, and this iteration has ran for approximately 4.5 hours. Since there is no significant difference between tc=5 and tc=6 for the lr=0.01, we will use tc=5 as the final level of interaction for our *Boost-1* model.

The *gbm* package of R was used to build *Boost-1* model, and this model has identified the following 60 variables as important. These are displayed in Figure-2 below:

Figure-2: Important variables identified by *Boost-1* Model built using *lr=0.01* and *tc=5*



The figure displays the relative importance of the significant variables. The *cat80* variable is the most important variable in the data set that identifies the *loss* variable.

Building *Rule-Fit-1* model

We will now build a Rule Fit model. The Rule fit method, is also an ensemble method, in which a set of rules are generated by building a decision tree in each stage, extracting the rules from the decision tree, and fitting a linear model using the rules extracted in each stage. We will have a learning rate that decides the importance of each tree, and thus the rule. The rule can be considered as an edge in the decision tree, and it represents whether a specific condition is TRUE or FALSE. In Rule Fit method, the tree size represents the number of terminal nodes in the decision tree (which is built at each stage of the training). If the tree size is 2, then it represents a stump, and does not capture any interactions between the predictor variables. But if tree size is larger than 2, then the complexity of the model increases and thus the chance of over fitting. Therefore, unlike in BRT method, Rule Fit method chooses the number of terminal tree nodes randomly, following an exponential distribution with a desired mean (supplied to the algorithm as a parameter).

To fit a rule fit model to the training data, we used the $RuleFit^3$ package on the whole training data set with the following input parameters:

nc: Average terminal Node count of 2

mr: Maximum number of rules to be built. We used 500, to limit the computational time

sf: Sample fraction of 0.6 (60%) to build decision trees at each stage

The algorithm was initially ran on 30000 observations (randomly chosen) and the runtime was approximately 2 hours. This makes the algorithm computationally difficult to test various parameters of the Rule fit algorithm. Hence we directly ran the algorithm on the complete test data with the above listed parameters. The algorithm ran for *6 hours* on the whole data set, and obtained a CV error of 4328873 and the algorithm has identified the variables (in Figure-3) as the important variables:

Figure-3: Important variables identified by *Rule-Fit-1* Model

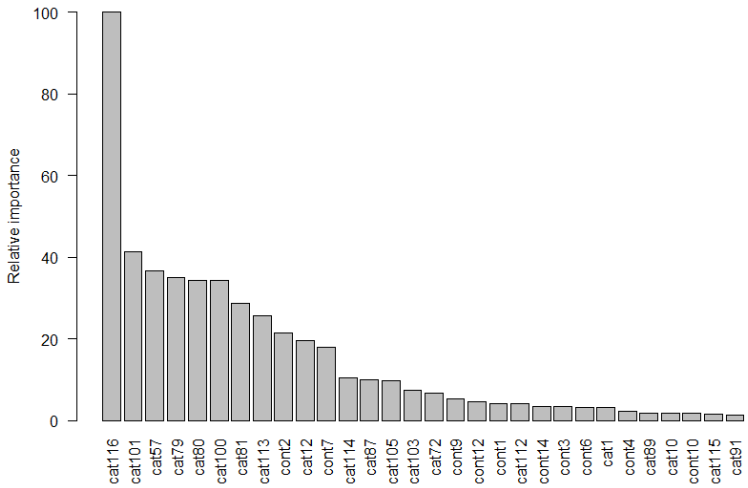


Table-2: Important variables comparison

Variable	BRT Rank (Boost-1 Model)	Rule Fit Rank (Rule-Fit-1 Model)
cat80	1	5
cat79	2	4
cat116	3	1
cat57	4	3
cont7	5	11
cat101	6	2
cont2	7	9
cat113	8	8
cat81	9	7
cat100	10	6
cat12	11	10
cat87	12	13
cat112	13	20
cat114	14	12
cat111	15	NA
cat72	16	16
cont12	17	18
cat103	18	15
cat105	19	14
cat53	20	NA
cont9	NA	17
cont12	NA	19

From Figure-3, Figure-4 and Table-2, we can infer the following:

- Except 2 variables, all other important variables are the same in both *Boost-1* and *Rule-Fit-1* models.
- None of the variables have the same rank in both the models.
- The *Boost-1* model has identified *cat80* as the most important variable in the model, while the *Rule-Fit-1* model has identified *cat116* as the most significant variable.
- The *Boost-1* model has *cat111* and *cat53* as important variables in the top 20 significant variables. But the *Rule-Fit-1* model does not have these variables as significant variables in its top 20 list.
- The *Rule-Fit-1* model has *cont9* and *cont12* as important variables in the top 20 significant variables. But the *Boost-1* model does not have these variables as significant variables in its top 20 list.

The following table shows the *Boost-1* (based on BRT) and *Rule-Fit-1* (based on Rule Fit) models performance:

Table-3: Performance comparison for BRT and Rule Fit methods

Method	CV	Runtime
Boosted Regression Trees (BRT)	3960867	3 hours
Rule fit	4328873	6 hours

Since the CV error and runtime of the *Rule-Fit-1* model are much higher than the *Boost-1* model, we do not need to build further Rule Fit models for performance evaluation. We will therefore decide to use the *Boost-1* (which is based on the *Boosted Regression Trees*) model. We made a successful submission to Kaggle, and the absolute error error for *Boost-1* model is 1209.29482 while the *Rule-Fit-1* model has an absolute error of 3859.59284. This result has further warranted us to discard Rule fit method of algorithms for our data.

Conclusion

The *Rule fit* method did not perform well on the given data set, since its CV error is higher than the *Boosted Regression Trees*. The Kaggle score of *Rule fit* model (3859.6) is significantly higher than the Kaggle score of *Boosted Regression Trees* (1209.3). The runtime of the *Rule fit* method was another factor to discard *Rule fit* method for our predictive model.

As a part of our future work, we would like to investigate the following:

1. Perform feature engineering on the important variables identified by the Boosted regression trees (based on the interactions identified by *Boost-1* model), and try to improve the *Boost-1* model's performance.
2. We conducted all the computation using a laptop with 6GB of RAM. This computational limitation has inhibited us to evaluate the algorithms performance thoroughly for the given data. We would like to build our predictive models using AWS (https://aws.amazon.com/) and compare the run-time and predictive performances with the models built using our laptop. Also we would like to run Random Forests algorithm on the training data set, and compare its performance with BRT and Rule fit methods. Our preliminary run of random forests on the training data (using our laptop) has ran for 24 hours, without producing any model.
3. Use xgboost (http://xgboost.readthedocs.io/en/latest/model.html) method to build the predictive model. The creators of xgboost claim that the algorithm is significantly faster than the other boosting algorithms.

References

1. J. Elith, J. R. Leathwick, and T. Hastie (2008), *A working guide to boosted regression trees*, Journal of Animal Ecology
2. Jerome H. Friedman and Bogdan E. Popescu (2005), *Predictive Learning via Rule Ensembles*, Stanford University, Department of Statistics. Technical report
3. RuleFit with R (http://statweb.stanford.edu/~jhf/r-rulefit/RuleFit_help.html)
4. Gareth James, Daniela Witten, Trevor Hastie, Robert Tibshirani - *An Introduction to Statistical Learning with Applications in R, pp: 28-29, 33-49*, Sprienger 2013

Appendix-A

Variables of training data set.

Variable	Type	Level
cat1	Factor	2
cat2	Factor	2
cat3	Factor	2
cat4	Factor	2
cat5	Factor	2
cat6	Factor	2
cat7	Factor	2
cat8	Factor	2
cat9	Factor	2
cat10	Factor	2
cat11	Factor	2
cat12	Factor	2
cat13	Factor	2
cat14	Factor	2
cat15	Factor	2
cat16	Factor	2
cat17	Factor	2
cat18	Factor	2
cat19	Factor	2
cat20	Factor	2
cat21	Factor	2
cat22	Factor	2
cat23	Factor	2
cat24	Factor	2
cat25	Factor	2
cat26	Factor	2
cat27	Factor	2
cat28	Factor	2
cat29	Factor	2
cat30	Factor	2
cat31	Factor	2
cat32	Factor	2
cat33	Factor	2
cat34	Factor	2
cat35	Factor	2
cat36	Factor	2
cat37	Factor	2
cat38	Factor	2
cat39	Factor	2
cat40	Factor	2
cat41	Factor	2
cat42	Factor	2
cat43	Factor	2
cat44	Factor	2
cat45	Factor	2
cat46	Factor	2
cat47	Factor	2
cat48	Factor	2

Variable	Type	Level
cat49	Factor	2
cat50	Factor	2
cat51	Factor	2
cat52	Factor	2
cat53	Factor	2
cat54	Factor	2
cat55	Factor	2
cat56	Factor	2
cat57	Factor	2
cat58	Factor	2
cat59	Factor	2
cat60	Factor	2
cat61	Factor	2
cat62	Factor	2
cat63	Factor	2
cat64	Factor	2
cat65	Factor	2
cat66	Factor	2
cat67	Factor	2
cat68	Factor	2
cat69	Factor	2
cat70	Factor	2
cat71	Factor	2
cat72	Factor	2
cat73	Factor	3
cat74	Factor	3
cat75	Factor	3
cat76	Factor	3
cat77	Factor	4
cat78	Factor	4
cat79	Factor	4
cat80	Factor	4
cat81	Factor	4
cat82	Factor	4
cat83	Factor	4
cat84	Factor	4
cat85	Factor	4
cat86	Factor	4
cat87	Factor	4
cat88	Factor	4
cat89	Factor	8
cat90	Factor	7
cat91	Factor	8
cat92	Factor	7
cat93	Factor	5
cat94	Factor	7
cat95	Factor	5
cat96	Factor	8
cat97	Factor	7
cat98	Factor	5
cat99	Factor	16
cat100	Factor	15

Variable	Type	Level
cat101	Factor	19
cat102	Factor	9
cat103	Factor	13
cat104	Factor	17
cat105	Factor	20
cat106	Factor	17
cat107	Factor	20
cat108	Factor	11
cat109	Factor	84
cat110	Factor	131
cat111	Factor	16
cat112	Factor	51
cat113	Factor	61
cat114	Factor	19
cat115	Factor	23
cat116	Factor	326
cont1	num	–
cont2	num	–
cont3	num	–
cont4	num	–
cont5	num	–
cont6	num	–
cont7	num	–
cont8	num	–
cont9	num	–
cont10	num	–
cont11	num	–
cont12	num	–
cont13	num	–
cont14	num	–
loss	num	–

Appendix B

The R Source code used to build the models and graphs is given below:

```
#Including all the required packages
rm(list=ls())
library(ggplot2)
library(gridExtra)
library(tree)
library(ISLR)
library(MASS)
library(randomForest)
library(dplyr)
library(knitr)
library(gbm)
library(grid)
library(png)

set.seed(1)
```

```
#Read the dats sets
setwd("C:/Users/Sekhar/Documents/R Programs/Business Analytics/Final_Project")

train_df <- read.csv("train.csv/train.csv")
test_df <- read.csv("test.csv/test.csv")
```

```
set.seed(1)
train_df_mod <- train_df[sample(1:nrow(train_df),30000),]
tc = c(1,1,1,1,1,5,5,5,5,5)

lr = c(0.1,0.05,0.01, 0.005, 0.001,0.1,0.05,0.01, 0.005, 0.001)

display_df = data.frame(Tree_Complexity=tc,Learning_Rate=lr)
#kable(display_df)
```

```
    print(Sys.time())

    gbm.models_1 <- gbm(loss~., data=train_df_mod,
        distribution="gaussian",
        n.trees=1500,
        interaction.depth=1,
        shrinkage=0.1,
        bag.fraction=0.5,
        keep.data=FALSE,
        cv.folds=5)

    print(Sys.time())

    gbm.models_2 <- gbm(loss~., data=train_df_mod,
        distribution="gaussian",
        n.trees=1500,
        interaction.depth=1,
        shrinkage=0.05,
        bag.fraction=0.5,
        keep.data=FALSE,
        cv.folds=5)

    print(Sys.time())

    gbm.models_3 <- gbm(loss~., data=train_df_mod,
        distribution="gaussian",
        n.trees=1500,
        interaction.depth=1,
        shrinkage=0.01,
        bag.fraction=0.5,
        keep.data=FALSE,
        cv.folds=5)

    print(Sys.time())

    gbm.models_4 <- gbm(loss~., data=train_df_mod,
        distribution="gaussian",
        n.trees=1500,
```

```
    interaction.depth=1,
    shrinkage=,
    bag.fraction=0.005,
    keep.data=FALSE,
    cv.folds=5)

print(Sys.time())

gbm.models_5 <- gbm(loss~., data=train_df_mod,
    distribution="gaussian",
    n.trees=1500,
    interaction.depth=1,
    shrinkage=0.001,
    bag.fraction=0.5,
    keep.data=FALSE,
    cv.folds=5)

print(Sys.time())

gbm.models_6 <- gbm(loss~., data=train_df_mod,
    distribution="gaussian",
    n.trees=1500,
    interaction.depth=5,
    shrinkage=0.1,
    bag.fraction=0.5,
    keep.data=FALSE,
    cv.folds=5)

print(Sys.time())

gbm.models_7 <- gbm(loss~., data=train_df_mod,
    distribution="gaussian",
    n.trees=1500,
    interaction.depth=5,
    shrinkage=0.05,
    bag.fraction=0.5,
    keep.data=FALSE,
    cv.folds=5)

print(Sys.time())

gbm.models_8 <- gbm(loss~., data=train_df_mod,
    distribution="gaussian",
    n.trees=1500,
    interaction.depth=5,
    shrinkage=0.01,
    bag.fraction=0.5,
    keep.data=FALSE,
    cv.folds=5)

print(Sys.time())

gbm.models_9 <- gbm(loss~., data=train_df_mod,
```

```
        distribution="gaussian",
        n.trees=1500,
        interaction.depth=5,
        shrinkage=0.005,
        bag.fraction=0.5,
        keep.data=FALSE,
        cv.folds=5)

    print(Sys.time())

    gbm.models_10 <- gbm(loss~., data=train_df_mod,
        distribution="gaussian",
        n.trees=1500,
        interaction.depth=5,
        shrinkage=0.001,
        bag.fraction=0.5,
        keep.data=FALSE,
        cv.folds=5)
    print(Sys.time())
```

```
Run_Time_Minutes = c(5,6,5,5,5,21,22,20,20,23)
Learning_Rate = lr
Tree_Complexity = tc
Least_CV_Error =  c(4807191,4744026,4788385,6803807,
                    5907541,4359167,4287151,4265894,4198831,5109777)
Optimal_Tree_Count = c(126,236,1322,1487,
                       1500,67,144,776,1499,1500)
Important_Variables_Count = c(24,24,32,23,
                              6,36,42,60,56,29)



display_df = data.frame(nt=1500,
                        lr=Learning_Rate,
                        tc=Tree_Complexity,CV=Least_CV_Error,
                        ont=Optimal_Tree_Count,
                        Imp_var_cnt=Important_Variables_Count,
                        Minutes=Run_Time_Minutes)
display_df$tc = as.factor(display_df$tc)
kable(display_df)
```

```
g1=ggplot(display_df[display_df$ont!=1500,],
          aes(x=lr,y=CV,color=tc))+
 geom_point(size=4)+
 geom_line()+
  labs(title="(a) Learning Rate vs Least CV error",
       x= "Learning Rate", y= "Least Cross validation error")


g2=ggplot(display_df[display_df$ont!=1500,],
          aes(x=lr,y=Imp_var_cnt,color=tc))+
 geom_point(size=4)+
 geom_line()+
```

```
  labs(title="(b) Learning Rate vs Important variables count",
       x= "Learning Rate", y= "# of Important variables")

grid.arrange(g1,g2, ncol=2)
```

```
print(Sys.time())
boost.fit11 = gbm(loss~.,
  data=train_df,distribution="gaussian",n.trees=1500,
  interaction.depth=5,
  shrinkage=0.01,bag.fraction=0.5,
  keep.data=FALSE,cv.folds=5)

    print(Sys.time())
```

```
print(Sys.time())
boost.fit11 = gbm(loss~.,
  data=train_df,distribution="gaussian",
  n.trees=1500,
  interaction.depth=5,
  shrinkage=0.01,bag.fraction=0.5,
  keep.data=FALSE,cv.folds=5)

    print(Sys.time())
```

```
#Rulefit installation
#http://statweb.stanford.edu/~jhf/R_RuleFit.html

platform = "windows"
rfhome = "C:/Users/Sekhar/Documents/R/rulefit"
source("C:/Users/Sekhar/Documents/R/rulefit/rulefit.r")
#install.packages("akima", lib=rfhome)
library(akima, lib.loc=rfhome)

#Everytime we run the rulefit program, a model will be stored in the rulefit directory.
#We can retrieve the current model by using getmodel() fun
#But if you rerun the model again, then the existing model will be replaced.
```

```
names(train_df)
train_df = train_df[,-1]



rf.fit1 = rulefit(x=train_df[,-131],y=train_df[,131],
                  rfmode="regress",
                  test.reps=5,test.fract=0.2,
                  tree.size=2,
                  max.rules=500,samp.fract=0.6,
                  quiet = FALSE,mod.sel =1,sparse = 1)
varimp()
interact(1:50)

test_obs = sample(1:188318,30000)
```

```
x_test=x[test_obs,]
y_test = rfpred(x_test)
mean(abs(y[test_obs] - y_test))

y_test = rfpred(test_df[,-131])
eval_df <- data.frame(id=test_df$id,loss=y_test)

write.csv(eval_df,file="eval_rf_1.csv",row.names=FALSE)

#To perform cross val. Do this only after u generate the rfpred()
rfxval (nfold=5, quiet=F)
```

```
display_df = summary(boost.fit11)[1:60,]
display_df$percentage = display_df$rel.inf/(display_df$rel.inf[1]) * 100
display_df = display_df[order(display_df$percentage,decreasing=TRUE),]

ggplot(display_df,aes(x=reorder(var, percentage),
                      y=percentage,label=paste(round(percentage,2),"%")))+
  geom_bar(stat="identity",fill="blue")+
  geom_text( position = position_dodge(1),
             vjust = 0.5,hjust=0)+
  labs(title="Variables importance",x="Variable",
       y="Importance percentage (rounded to 2 decimal points)")+
  coord_flip()




prd = predict(boost.fit11,newdata=train_df_mod,n.trees=1209)
mean(abs(prd-train_df_mod$loss))
test_df <- read.csv("test.csv/test.csv")
boost.fit11.predict = predict(boost.fit11,newdata=test_df,n.trees=1209)
eval_df <- data.frame(id=test_df$id,loss=boost.fit11.predict)
write.csv(eval_df,file="eval_3.csv",row.names=FALSE)
```

```
img <- readPNG("C:/Users/Sekhar/Documents/R Programs/Business Analytics/Final_Project/Rplot01.png")
 grid.raster(img)
```

```
img <- readPNG("C:/Users/Sekhar/Documents/R Programs/Business Analytics/Final_Project/Rplot02.png")
 grid.raster(img)
```

```
img <- readPNG("C:/Users/Sekhar/Documents/R Programs/Business Analytics/Final_Project/Rplot03.png")
 grid.raster(img)
```

```
Method=c("Boosted Regression Trees (BRT)", "Rule fit")
CV = c(3960867,4328873)
Runtime = c("3 hours", "6 hours")
display_df = data.frame(Method,CV, Runtime)
kable(display_df)
```

```
#str(train_df,list.len=132)
Variable=c('cat1','cat2','cat3','cat4',
'cat5','cat6','cat7','cat8',
'cat9','cat10','cat11','cat12',
'cat13','cat14','cat15','cat16',
'cat17','cat18','cat19','cat20',
'cat21','cat22','cat23','cat24',
'cat25','cat26','cat27','cat28',
'cat29','cat30','cat31','cat32',
'cat33','cat34','cat35','cat36',
'cat37','cat38','cat39','cat40',
'cat41','cat42','cat43','cat44',
'cat45','cat46','cat47','cat48',
'cat49','cat50','cat51','cat52',
'cat53','cat54','cat55','cat56',
'cat57','cat58','cat59','cat60',
'cat61','cat62','cat63','cat64',
'cat65','cat66','cat67','cat68',
'cat69','cat70','cat71','cat72',
'cat73','cat74','cat75','cat76',
'cat77','cat78','cat79','cat80',
'cat81','cat82','cat83','cat84',
'cat85','cat86','cat87','cat88',
'cat89','cat90','cat91','cat92',
'cat93','cat94','cat95','cat96',
'cat97','cat98','cat99','cat100',
'cat101','cat102','cat103','cat104',
'cat105','cat106','cat107','cat108',
'cat109','cat110','cat111','cat112',
'cat113','cat114','cat115','cat116',
'cont1','cont2','cont3','cont4',
'cont5','cont6','cont7','cont8',
'cont9','cont10','cont11','cont12',
'cont13','cont14','loss')
Type=c(
  'Factor','Factor','Factor','Factor',
'Factor','Factor','Factor','Factor',
'Factor','Factor','Factor','Factor',
'Factor','Factor','Factor','Factor',
'Factor','Factor','Factor','Factor',
'Factor','Factor','Factor','Factor',
'Factor','Factor','Factor','Factor',
'Factor','Factor','Factor','Factor',
'Factor','Factor','Factor','Factor',
'Factor','Factor','Factor','Factor',
'Factor','Factor','Factor','Factor',
'Factor','Factor','Factor','Factor',
'Factor','Factor','Factor','Factor',
'Factor','Factor','Factor','Factor',
'Factor','Factor','Factor','Factor',
'Factor','Factor','Factor','Factor',
'Factor','Factor','Factor','Factor',
'Factor','Factor','Factor','Factor',
```

```
'Factor','Factor','Factor','Factor',
'Factor','Factor','Factor','Factor',
'Factor','Factor','Factor','Factor',
'Factor','Factor','Factor','Factor',
'Factor','Factor','Factor','Factor',
'Factor','Factor','Factor','Factor',
'Factor','Factor','Factor','Factor',
'Factor','Factor','Factor','Factor',
'Factor','Factor','Factor','Factor',
'Factor','Factor','Factor','Factor',
'num','num','num','num',
'num','num','num','num',
'num','num','num','num',
'num','num','num'),
Levels=c(
2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2,3,3,3,3,4,4,
4,4,4,4,4,4,4,4,
4,4,8,7,8,7,5,7,
5,8,7,5,16,15,19,9,
13,17,20,17,20,11,84,131,
16,51,61,19,23,326,"--","--","--",
"--","--","--","--","--","--","--","--",
"--","--","--","--")
display_df = data.frame(Variable=Variable, Type=Type,Level=Levels)
kable(display_df)
```