Boosted Trees

CMDA 4654 - Project 2, Group 2

Authors: Lukas Coffey, Ben Zevin, Chandler Crescentini, Juhong Hyun

2019-12-03

Outline

- Summary of boosted trees
- What is boosting?
- Boosted regression theory
- Boosted classification theory
- Description of method parameters
- Using `gbm` on a small dataset
- Using gradient boosting on a large dataset

Summary

Commonly referred to as gradient boosting, boosted trees is a method that works against the common problem among trees - overfitting a dataset. It's called "gradient" boosting because it uses gradient descent to iteratively find the best fit to the data. We are trying to "step" in the right direction, towards the least amount of error. The idea stems from a method called Adaboost - we will not discuss it here but you can find plenty of resources explaining the method online.

There are two use cases for gradient tree boosting, **regression** and **classification**. The concept is the same but the implementation varies slightly depending on the use case. In this presentation we will touch on classification but focus on regression.

Summary

Gradient boost for regression works by creating small regression trees (known as weak learners), calculating the errors from that weak learner, and fitting another tree to predict those errors. These models are then combined to form the gradient boosted model.

Each step of the algorithm creates a new tree that may be small or large using random samples of columns, depending on your tuning parameters. All of the trees are then scaled by a tuning parameter between 0 and 1. *This learning rate parameter basically determines how quickly our boosted trees find a good fit - the smaller the learning rate the better the model.*

The iterations continue for either **A)** a specified number of steps, or **B)** until the errors become sufficiently small.

You can see how this is similar to random forests in that the algorithm creates many trees, and can use random samples of columns. However, the algorithm does not just pick variables at random, it uses the variables that you provide. We can also visualize the importance of the variables we give the algorithm and exclude/include certain columns that provide the most prediction power. We will demonstrate this later.

Pros & Cons

Pros of Gradient Boosting

- Easily interpretable
- Robust to outliers
- Can deal with continuous and categorical data
- Captures non-linear relationships well
- Scales to large datasets

Cons of Gradient Boosting

- Can tend to overfit with predictors that have many categories
- Lacks smoothness
- Can tend to have limited predictive performance
- Long computation times depending on hyperparameters

Terminology

There is quite a bit of mixed jargon with regards to boosted trees. Since we're using gradient descent to "boost" normal decision trees, the method is called "tree boosting." They're all referring to this same idea that we're trying to make regular decision trees better. Below are some of the terms you might see the method referred as:

- Tree boosting
- Boosted trees
- Boosted regression
- Boosted classification
- Gradient boosting
- Gradient boosted regression trees

Boosting & Gradient Descent

What is Boosting?

Key question: Can a set of weak learners create a single strong learner?

In machine learning, boosting is an ensemble$_1$ meta-algorithm$_2$ for primarily reducing bias and variance in supervised learning, and a family of machine learning algorithms that convert weak learners$_3$ to strong learners$_4$.

1. *Meta-algorithm:* higher-level procedure designed to find a heuristic that may provide a sufficiently good solution to an optimization problem especially with incomplete information.
2. *Ensemble:* multi-learning algorithms
3. *Weak learner:* classifier that is only slightly correlated with the true classification
4. *Strong learner:* classifier that is arbitrarily well-correlated with the true classification.

What is Boosting, cont.

Characteristics of Boosting

- Uses lean weak classifiers with respect to a distribution
- Add them to a final strong classifier
- After a weak learner is added, data weights are readjsuted, known as "re-weighting".
- Misclassified input data gain a higher weight and examples that are classified correctly lose weight.
- Thus, future weak learners focus more on the examples that previous weak learners misclassified.

Types of Boosting

- AdaBoost (an adaptive boosting algorithm that won the prestigious Gödel Prize. / historically significant)
- Gradient Boosting (Gradient Descent + Boosting)
- LPBoost, TotalBoost, BrownBoost, xgboost, MadaBoost, etc.

Key Concepts

What is gradient descent?

Suppose you want to optimize a function $f(x)$. Assuming f is differentiable, gradient descent works by iteratively finding

$$x_{t+1} = x_t - \eta \frac{\partial f}{\partial x}\Big|_{x=x_t}$$

where η is called the step size.

What is additive model?

Additive model: $g(x) = f_0(x) + f_1(x) + f_2(x) + \ldots$, where the final classifier g is the sum of simple base classifiers f_i.

For the boosted trees model, each base classifier is a simple decision tree.

Gradient Boosting

Gradient Descent + Boosting = Gradient Boosting

Similarly, if we let $g_t(x) = \sum_{i=1}^{t} f_i(x)$ (additive model; ensemble) be the classifier trained at iteration t, and $L(y_i, g(x_i))$ be the empirical loss function, at each iteration we will move g_t towards the negative gradient direction $-\frac{\partial L}{\partial g}|_{g=g_t}$ by η amount.

Hence, f_t is chosen to be

$$f_t = \underset{t}{argmin} \sum_{i=1}^{N} [\frac{\partial L(y_i, g(x_i))}{\partial g(x_i)}|_{g=g_t} - f(x_i)]^2$$

and the algorithm sets $g_{t+1} = g_t + \eta f_t$.

Boosted Regression

Boosted Regression

References

- Gradient Boosting Explained: http://blog.kaggle.com/2017/01/23/a-kaggle-master-explains-gradient-boosting/
- Gradient Boosting: https://en.wikipedia.org/wiki/Gradient_boosting
- StatQuest - Gradient Boost, Parts 1 & 2: https://www.youtube.com/watch?v=3CC4N4z3GJc&t=582s

Boosted Regression - Main Idea

The main idea behind boosted regression is that we fit a model, then fit another model to the residuals, and then the combination of these becomes the new model. We iterate this idea many times to find the optimal model. In pseudocode:

Step 1: We fit the original model.

$$F_1(x) = y$$

Step 2: We fit a new model to the residuals of the model in Step 1.

$$h_1(x) = y - F_1(x)$$

Step 3: We combine the previous models to find the next steps in the algorithm, then add these to find the final model.

$$F_2(x) = F_1(x) + h_1(x)$$
$$\ldots$$
$$final\ model \rightarrow F_M(x)$$

Boosted Regression - Step 1

To start the regression algorithm, we initialize the model with the average of our y values. This becomes our starting point because initially we want to minimize the squared error.

$$F_0(x) = \underset{\gamma}{argmin} \sum_{i=1}^{n} L(y_i, \gamma) = \underset{\gamma}{argmin} \sum_{i=1}^{n} (\gamma - y_i)^2 = \frac{1}{n} \sum_{i=1}^{n} y_i$$

Now, we can define the following models just like we did before in pseudocode.

$$F_{m+1}(x) = F_m(x) + h_m(x) = y, \ for \ m \geq 0$$

where h_m is a base learner (e.g. weak learner; tree).

Note: How do we select M? This value determines how many times we iterate until we find the best model. This is best done by cross-validation. We'll discuss this later in the regularization section.

Boosted Regression - Step 2

After we initialize the model, $F_0(x)$, we can begin iterating. Each iteration contains 4 steps:

Step 1: Compute the pseudo-residuals (this is the gradient descent part of *gradient* boost).

$$r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)} \quad for\ i = 1, \ldots, n$$

Step 2: Fit a base learner (weak learner, e.g. a tree) to the pseudo-residuals. This is $h_m(x)$.

Step 3: Compute the step magnitude multiplier, γ_m, by solving the 1D optimization problem:

$$\gamma_m = \underset{\gamma}{argmin} \sum_{i=1}^{n} L(y_i,\ F_{m-1}(x_i) + \gamma h_m(x_i))$$

Step 4: Update the model with the new pieces:

$$F_m(x) = F_{m-1}(x) + \gamma_m h_m(x)$$

Boosted Regression - Step 3

After we compute all the iterations of the algorithm, our final model is then:

$$for\ m = 1\ to\ M,$$
$$F_M(x)$$

with inputs:

1. Training data set, $(x_i, y_i)_{i=1}^{n}$
2. Loss function, $L(y, F(x))$
3. Number of iterations, M

We will talk about M and a few more paramaters of the algorithm later.

Boosted Classification

Boosted Classification

References

- Boosting: https://en.wikipedia.org/wiki/Boosting_(machine_learning)#cite_note-9
- Michael Kearns(1988); Thoughts on Hypothesis Boosting, Unpublished manuscript (Machine Learning class project, December 1988)
- Gradient Boosting: https://turi.com/learn/userguide/supervised-learning/boosted_trees_regression.html

Classification vs. Regression

What is different between classification and regression?

With classification, we are not dealing with a continuous response variable. Thus we have to work with proportions and probabilities in our algorithm instead of continuous numbers.

In the classification case, our Loss Function is:

$$L(y_i, \gamma) = -y_i\gamma \ + \ log(1 + e^{\gamma})$$

where y_i is a classification (0:1, yes:no, etc.), and γ is $log(odds)$. This is the function we are trying to minimize and that we find the gradient of in our algorithm.

This changes our initial value when we initialize the algorithm to a log odds value:

$$F_0(x) = log(odds)$$

where $odds$ is the ratio of the sums of the categories (# of yes's/# of no's, # of 1's/# of 0's, etc.).

Classification vs. Regression cont.

The steps for the algorithm are the same - they just use the new Loss function instead. We calculate the residuals for the previous step, and the next step uses the variables from the data to predict those residuals. There is much more deeper theory that goes into the algorithm for classification, which we won't go into here, but the videos below do a great job of explaining.

StatQuest: https://www.youtube.com/watch?v=jxuNLH5dXCs

StatQuest: https://www.youtube.com/watch?v=StWY5QWMXCw&t=1204s

Note:

It is important to note that because we are using gradient descent, we can input *any* differentiable loss function into this algorithm. This could be the mean squared error for regression, the log odds equation in classification, or any other loss function. That is part of what makes gradient boost so powerful!

Regularization

Regularization

One of the main goals in using boosted trees is to avoid overfitting the data. To achieve a boosted model that is well-generalized and not overfitted, we can make use of **regularization**. These are just a few of the regularization parameters that we can use to control effectiveness of the boosting algorithm:

- The number of iterations, M
- Maximum tree depth
- Minimum number of observations in tree leaves
- Penalizing complexity: pruning back the weak learners that don't provide any benefit
- Shrinkage parameter, ν, known as the *"learning rate"*
- Size of the trees

The number of iterations, M

This is the number of times we should run the gradient descent algorithm (basically the number of trees we create). As stated previously, M is most commonly chosen using cross-validation. Increasing M reduces error on the training set, but making M too high can lead to overfitting.

Regularization Parameters

Maximum tree depth

This is the maximum distance from the root node to the furthest leaf node in each of our trees. This helps control how "weak" our weak learners should actually be.

Minimum number of observations in tree leaves

This parameter controls the minimum number of observations within each leaf of the weak learners. This tells the weak learners to basically ignore any splits or new terminal nodes that have fewer observations than this number. This helps to reduce variance in the predictions at the leaves.

Penalizing complexity: pruning back the weak learners that don't provide any benefit

This is defined as the "proportional number of leaves in the learned trees." We can employ a post-pruning algorithm that removes branches failing to reduce the loss by a certain threshold. This basically helps to keep the trees from becoming too strong and overfitting.

Regularization Parameters, cont.

Shrinkage parameter, ν, known as the "learning rate"

This is arguably one of the most important parameters. This parameter is a value between 0 and 1, and controls how fast or slow our boosted algorithm finds the optimal model. A lower learning rate (e.g. 0.1) means higher computation time and more iterations, but a better and more generalized model. It is applied to the algorithm as:

$$F_m(x) = F_{m-1}(x) + \nu \cdot \gamma_m h_m(x)$$

Size of the trees

The tree size parameter, J, controls the number of terminal nodes in each weak learner. $J = 2$ would result in each tree being a stump, and no variable-variable interaction. $J = 3$ would allow for a maximum of 2 variables per tree, and a max of 3 leaves. This is a strong controller of the complexity of the trees in the algorithm and coupled with the learning rate can have a big effect on computation time.

Small Example - Boston Housing Data

Boston Housing Data

We will use the `gbm` function to run the gradient boosting algorithm on the Boston Housing dataset.

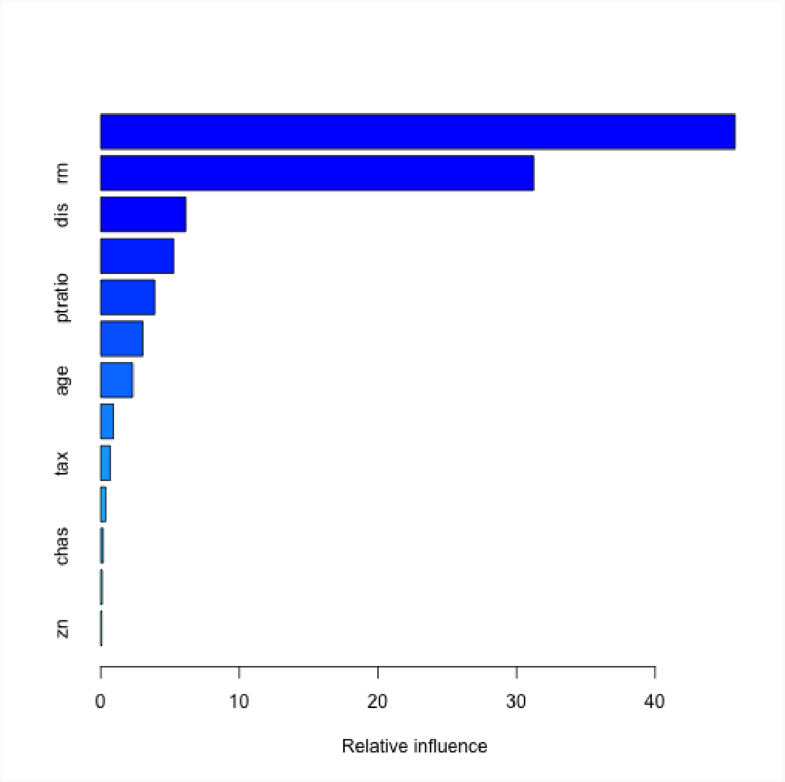
Documentation: https://cran.r-project.org/web/packages/gbm/gbm.pdf

```r
set.seed(101)
train ← sample(1:506, size = 354) # 70% train, 30% test
boston_fit ← gbm(medv ~ ., data = Boston[train,], distribution = "gaussian",
                 n.trees = 10000, shrinkage = 0.01, interaction.depth = 2,
                 bag.fraction = 1.0, train.fraction = 1.0, n.minobsinnode = 5,
                 cv.folds = 5)
boston_fit
```

```
## gbm(formula = medv ~ ., distribution = "gaussian", data = Boston[train,
##     ], n.trees = 10000, interaction.depth = 2, n.minobsinnode = 5,
##     shrinkage = 0.01, bag.fraction = 1, train.fraction = 1, cv.folds = 5)
## A gradient boosted model with gaussian loss function.
## 10000 iterations were performed.
## The best cross-validation iteration was 2239.
## There were 13 predictors of which 13 had non-zero influence.
```

Variable Importance

```r
fit_sum ← summary(boston_fit)
```



```r
data.frame(var = fit_sum$var,
           rel.inf = fit_sum$rel.inf)
```

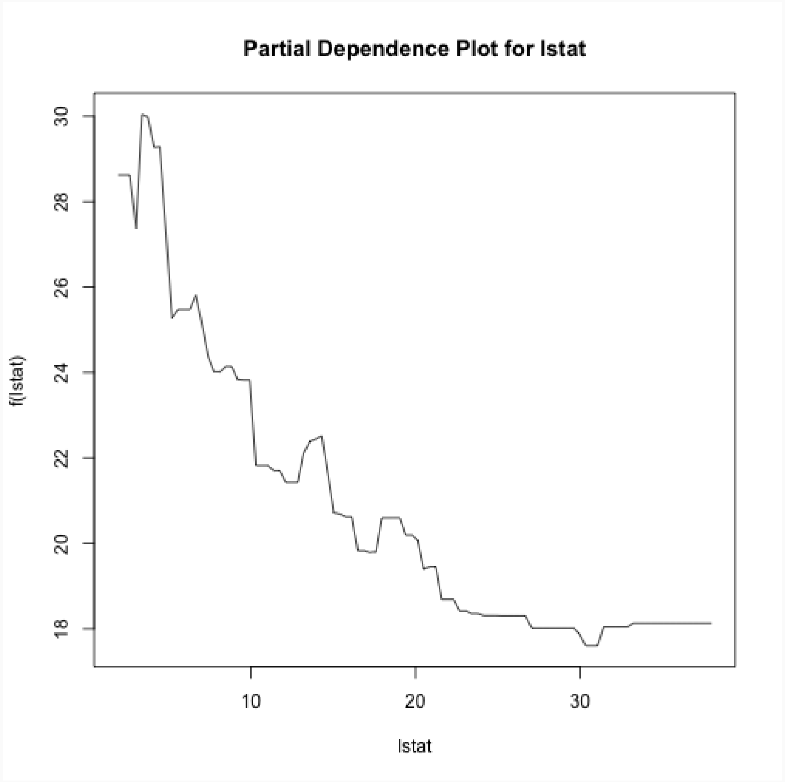
```
##        var     rel.inf
## 1    lstat 45.79230143
## 2       rm 31.23139819
## 3      dis  6.12672951
## 4      nox  5.25810029
## 5  ptratio  3.89834070
## 6     crim  3.04664748
## 7      age  2.31679100
## 8    black  0.93879450
## 9      tax  0.69499335
## 10     rad  0.35895333
## 11    chas  0.17658919
## 12   indus  0.09791081
## 13      zn  0.06245022
```

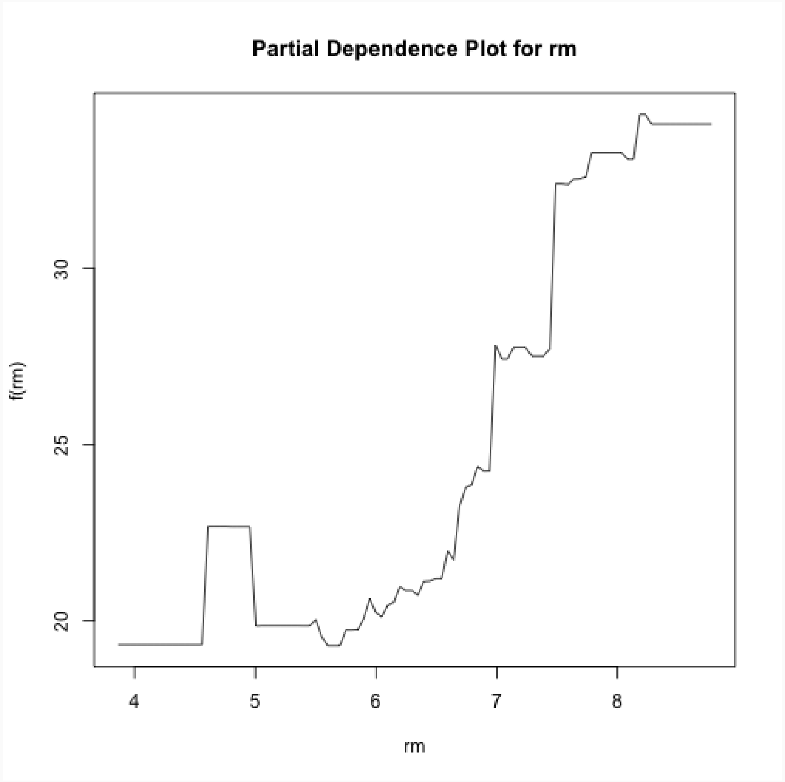
Partial Dependence Plots

Show us the marginal effect of variables on the predicted outcome.

```r
plot(boston_fit, i = "lstat")
```

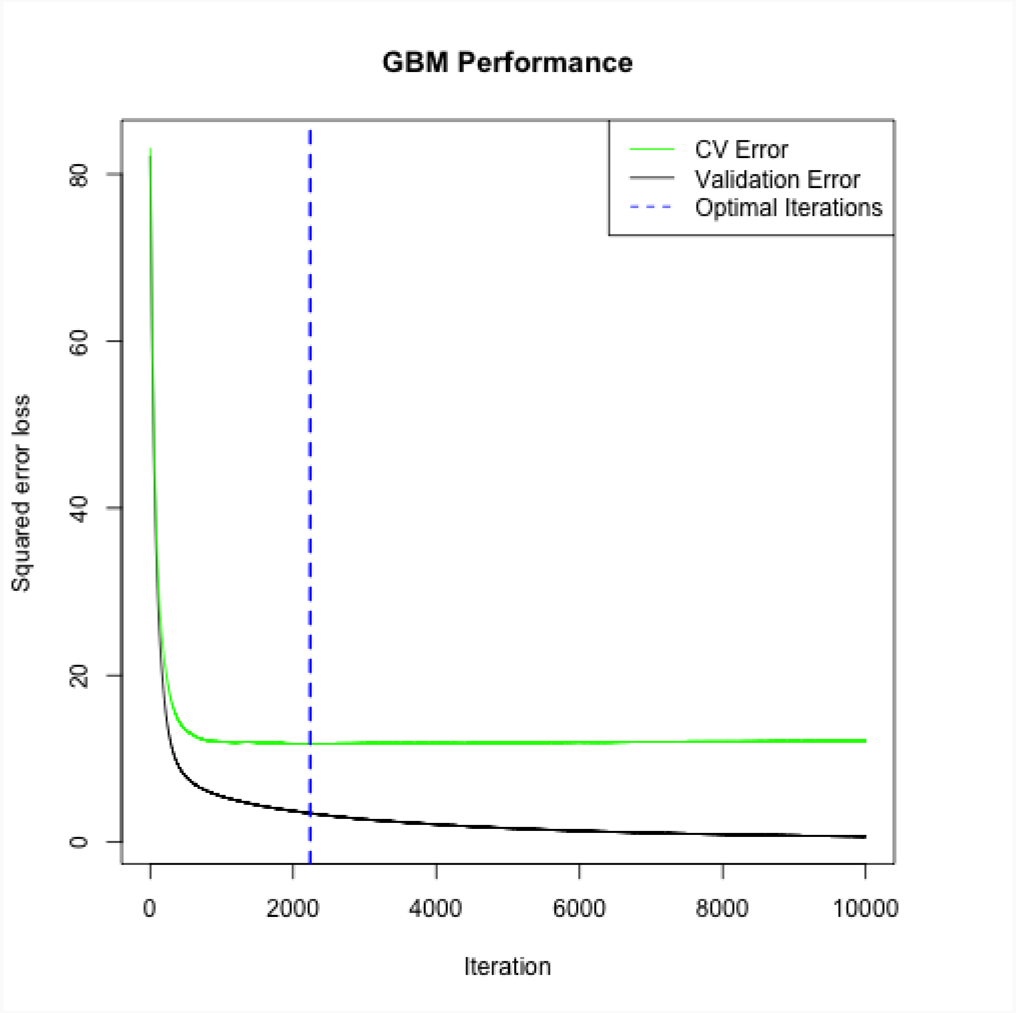
```r
plot(boston_fit, i = "rm")
```





Optimal Number of Boosting Iterations

```r
best.iter ← gbm.perf(boston_fit, method = "cv")
```



Optimal Number, cont.

```
pred ← predict.gbm(boston_fit, Boston[-train,], n.trees = best.iter)
error ← sum((pred - Boston$medv[-train])^2)
print(error)
```

```
## [1] 1968.915
```

Parameter Fine Tuning

```
boston_fit1 ← gbm(medv ~ ., data = Boston[train,], distribution = "gaussian",
                  n.trees = 10000, shrinkage = 0.01, interaction.depth = 2,
                  bag.fraction = 1.0, train.fraction = 1.0, n.minobsinnode = 5,
                  cv.folds = 5)
boston_fit2 ← gbm(medv ~ ., data = Boston[train,], distribution = "gaussian",
                  n.trees = 5000, shrinkage = 0.01, interaction.depth = 2,
                  bag.fraction = 1.0, train.fraction = 1.0, n.minobsinnode = 5,
                  cv.folds = 5)
boston_fit3 ← gbm(medv ~ ., data = Boston[train,], distribution = "gaussian",
                  n.trees = 10000, shrinkage = 0.01, interaction.depth = 2,
                  bag.fraction = 1.0, train.fraction = 1.0, n.minobsinnode = 10,
                  cv.folds = 5)
boston_fit4 ← gbm(medv ~ ., data = Boston[train,], distribution = "gaussian",
                  n.trees = 5000, shrinkage = 0.1, interaction.depth = 2,
                  bag.fraction = 1.0, train.fraction = 1.0, n.minobsinnode = 10,
                  cv.folds = 5)
```

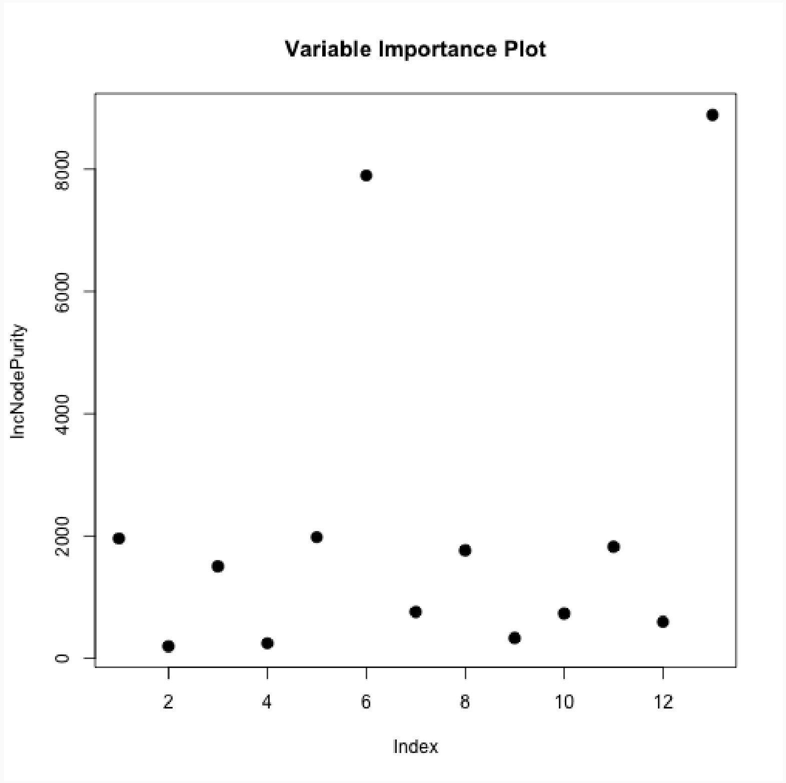
Parameter Fine Tuning, cont.

model	n.trees	shrinkage	n.minobsinnode	best_iter	error
boston_fit1	10000	0.01	5	2239	1968.915
boston_fit2	5000	0.01	5	4998	1909.311
boston_fit3	10000	0.01	10	2189	2045.575
boston_fit4	5000	0.10	10	356	1940.182

Comparison with Random Forest

```r
rf ← randomForest(medv ~ ., data = Boston[train,], distribution = "gaussian",
                   n.trees = 5000, shrinkage = 0.01, interaction.depth = 2,
                   nodesize = 5)
```

```
##           IncNodePurity
## crim          1957.5808
## zn             194.0213
## indus         1500.8585
## chas           243.5554
## nox           1979.7474
## rm            7894.3595
## age            757.2650
## dis           1762.4770
## rad            328.8089
## tax            730.4825
## ptratio       1822.4640
## black          594.3402
## lstat         8884.3577
```



Comparison with Random Forest, cont.

```r
predRF ← predict(rf, Boston[-train,], type = "response")
errorRF ← sum((predRF - Boston$medv[-train])^2)
print(errorRF)
```

```
## [1] 2020.809
```

Big Example - Economic Freedom Data

Big Example (Economic Freedom)

The following dataset is used to determine the freedom index and was published in the *Economic Freedom of the World* by the *Fraser Institute* measures the degree to which the policies and istitutions of countries are supportive of economic freedom. There are 36 variables in this data set but each of them fit into a one of the five broad areas.

1. Size of Goverment

2. Legal System and Property Rights

3. Sound Money

4. Freedom to Trade Internatioanlly

5. Regulation

- Dataset: https://www.kaggle.com/gsutters/economic-freedom

Economic Freedom (Read and split)

```r
library(gbm)
library(rsample)
library(dplyr)
library(caret)
```

These are the required packages needed to run gradient boosting machines.

Economic Freedom (Initial Model)

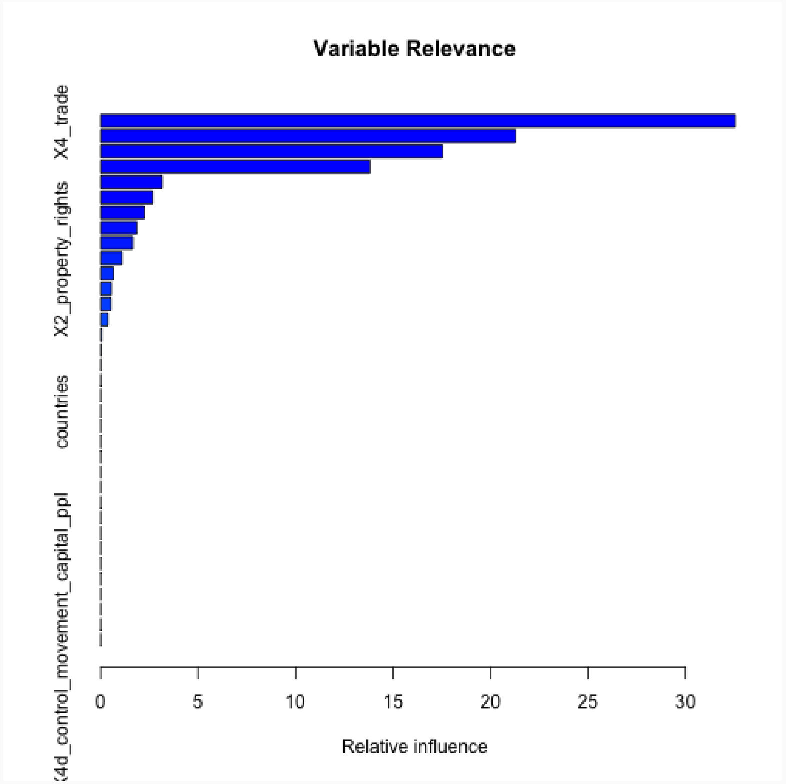
```r
free_split ← initial_split(freedom, prop = .7)
free_train ← training(free_split)
free_test  ← testing(free_split)
gbm.fit ← gbm(formula = ECONOMIC.FREEDOM ~ ., distribution = "gaussian",
                data = free_train, n.trees = 5000, interaction.depth = 1,
                shrinkage = 0.001, cv.folds = 5)
gbm.fit
```

```
## gbm(formula = ECONOMIC.FREEDOM ~ ., distribution = "gaussian",
##     data = free_train, n.trees = 5000, interaction.depth = 1,
##     shrinkage = 0.001, cv.folds = 5)
## A gradient boosted model with gaussian loss function.
## 5000 iterations were performed.
## The best cross-validation iteration was 5000.
## There were 35 predictors of which 18 had non-zero influence.
```

If you wanted to use gradient boosting for classification, all you need to do to implement it is to change the distribution to bernoulli and to make sure your response variable is a column that represents some sort of classification and contains either ones or zeros.

Economic Freedom (Initial Model cont.)

```
fit_sum1 ← summary(gbm.fit)
```



```
head(data.frame(fit_sum1$var,
                fit_sum1$rel.inf),8)
```

```
##        fit_sum1.var fit_sum1.rel.inf
## 1          X4_trade        32.543073
## 2          quartile        21.271054
## 3    X3_sound_money        17.540846
## 4     X5_regulation        13.806717
## 5 X4b_reg_trade_bar         3.157937
## 6              rank         2.658938
## 7          ISO_code         2.243317
## 8  X5c_business_reg         1.858637
```

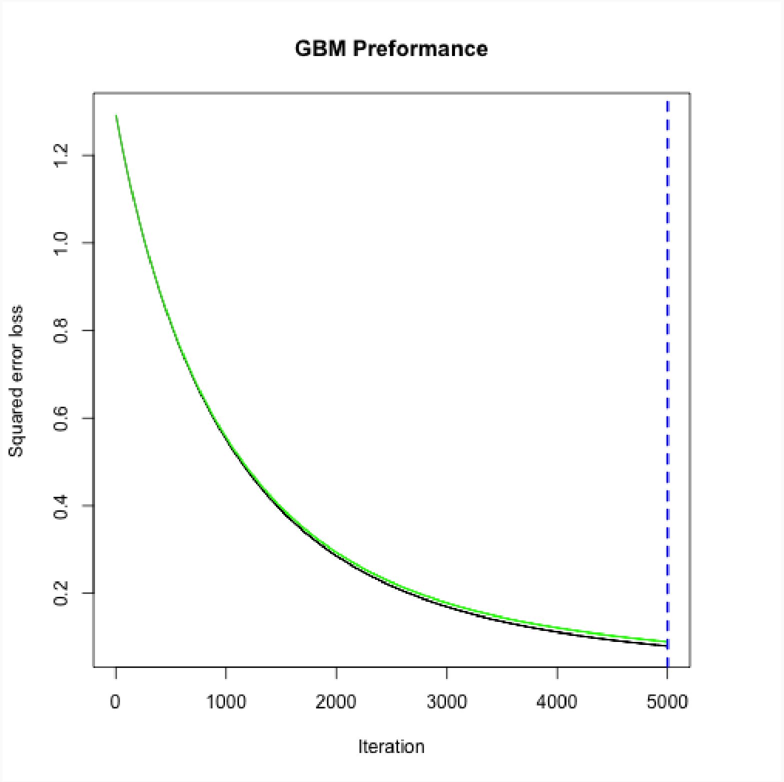
Economic Freedom (MSE and Min Trees)

```r
sqrt(min(gbm.fit$cv.error))
```

```
## [1] 0.2994301
```

```r
gbm.perf(gbm.fit, method = "cv")
```

```
## [1] 5000
```



Economic Freedom (Second Model)

```r
gbm.fit2 <- gbm(formula = ECONOMIC.FREEDOM ~ ., distribution = "gaussian",
                data = free_train, n.trees = 2500, interaction.depth = 1,
                shrinkage = 0.001, cv.folds = 5)
min_MSE <- which.min(gbm.fit2$cv.error)
min_MSE
```

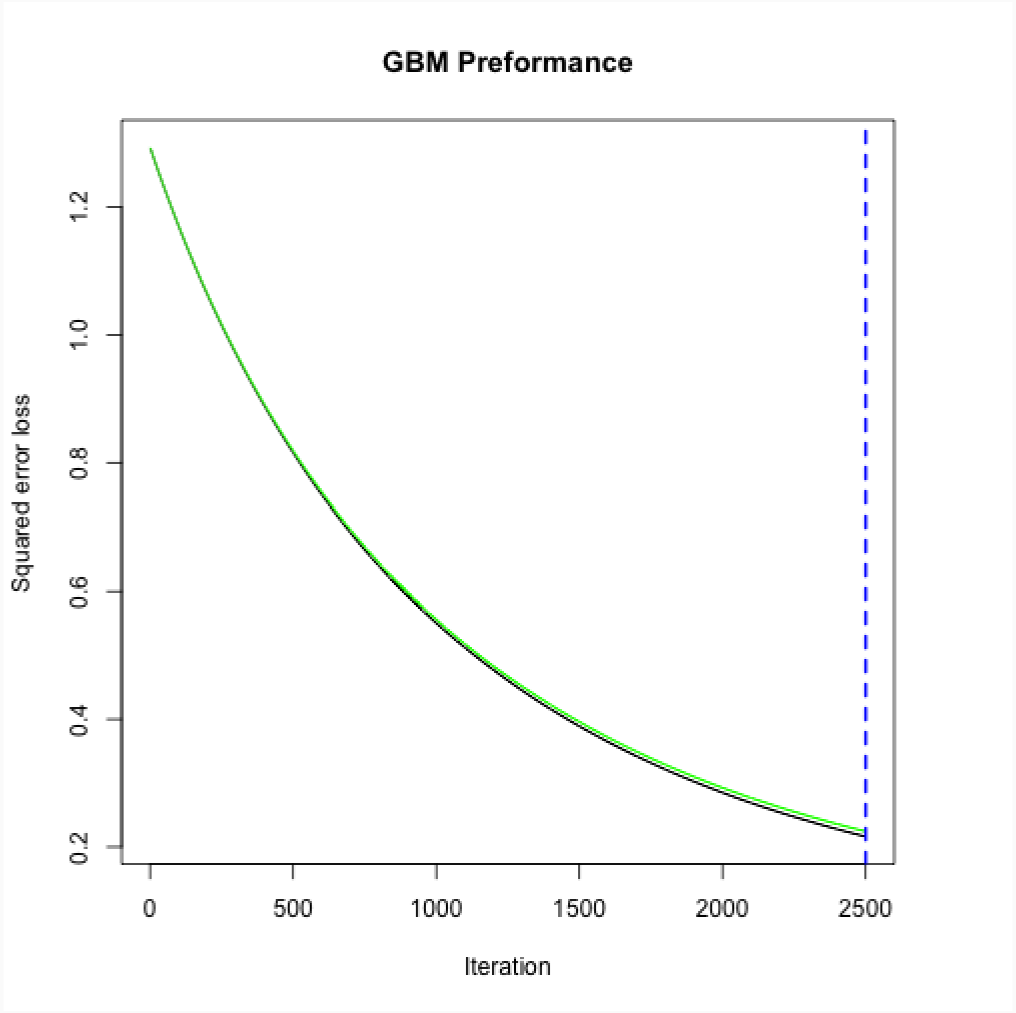
```
## [1] 2500
```

```r
# get MSE and compute RMSE
sqrt(gbm.fit2$cv.error[min_MSE])
```

```
## [1] 0.4736203
```

Economic Freedom (Min Trees)

```r
best_iter ← gbm.perf(gbm.fit2, method = "cv")
```



Economic Freedom (Multiple Models)

```r
hyper_grid <- expand.grid(shrinkage = c(.01, .1, .3),
                          interaction.depth = c(1, 2, 3),
                          n.minobsinnode = c(5, 10, 15),
                          bag.fraction = c(.65, .8, 1),
                          optimal_trees = 0,min_RMSE = 0)
for(i in 1:nrow(hyper_grid)) {

 # train model
 gbm.tune <- gbm(formula = ECONOMIC.FREEDOM ~ ., distribution = "gaussian",
                data = free_train, n.trees = 2500,
                interaction.depth = hyper_grid$interaction.depth[i],
                shrinkage = hyper_grid$shrinkage[i],
                n.minobsinnode = hyper_grid$n.minobsinnode[i],
                bag.fraction = hyper_grid$bag.fraction[i],
                train.fraction = .75)

 # add min training error and trees to grid
 hyper_grid$optimal_trees[i] <- which.min(gbm.tune$valid.error)
 hyper_grid$min_RMSE[i] <- sqrt(min(gbm.tune$valid.error))

}
```

Economic Freedom (Multiple Models)

```
hyper_grid <- hyper_grid %>% dplyr::arrange(min_RMSE)
knitr::kable(head(hyper_grid,8), format = "html")
```

shrinkage	interaction.depth	n.minobsinnode	bag.fraction	optimal_trees	min_RMSE
0.1	1	5	0.65	2466	0.6767641
0.1	1	5	0.80	2474	0.6846531
0.1	1	10	0.65	2487	0.6863954
0.1	1	15	0.65	1863	0.6939371
0.3	1	15	0.80	2495	0.6980808
0.3	1	10	0.80	2451	0.7039104
0.1	2	5	0.65	2339	0.7045366
0.3	1	5	0.80	2344	0.7165286

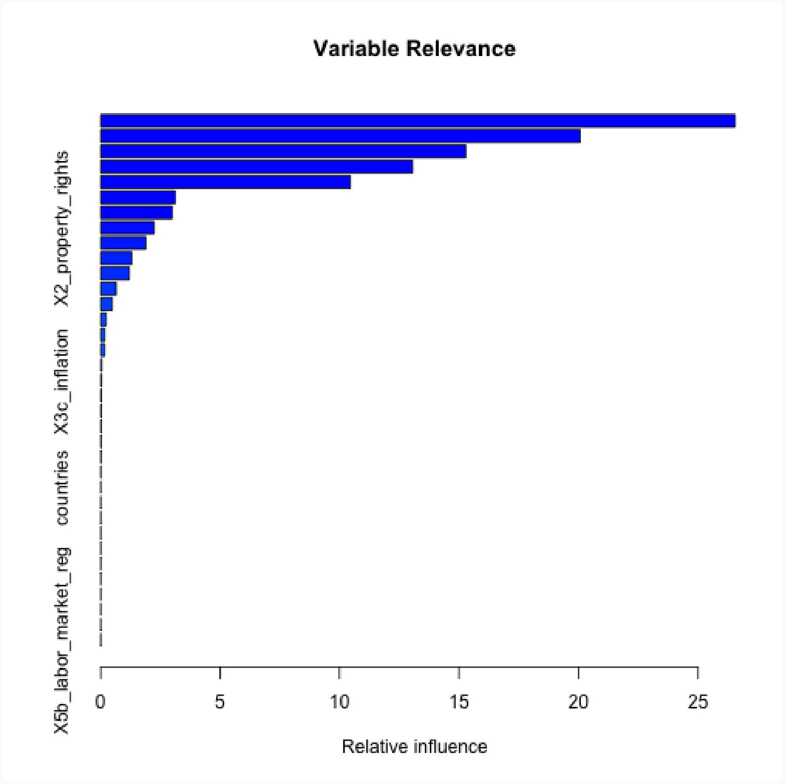
Economic Freedom (Best Model)

```r
gbm.fit.final <- gbm(formula = ECONOMIC.FREEDOM ~ ., distribution = "gaussian",
                     data = free_train, n.trees = 2466, interaction.depth = 1,
                     shrinkage = 0.1, n.minobsinnode = 5, bag.fraction = .65)
# Predictions and RMSE
pred <- predict(gbm.fit.final, n.trees = gbm.fit.final$n.trees, free_test)
caret::RMSE(pred, free_test$ECONOMIC.FREEDOM)
```

```
## [1] 0.1349991
```

Economic Freedom(Best Model cont.)

```
fit_sum ← summary(gbm.fit.final)
```



```
head(data.frame(fit_sum$var,
                fit_sum$rel.inf),8)
```

```
##           fit_sum.var fit_sum.rel.inf
## 1            X4_trade       26.552417
## 2            quartile       20.076450
## 3      X3_sound_money       15.272095
## 4       X5_regulation       13.044592
## 5            ISO_code       10.449143
## 6                rank        3.116889
## 7   X4b_reg_trade_bar        2.985496
## 8 X2_property_rights         2.235198
```