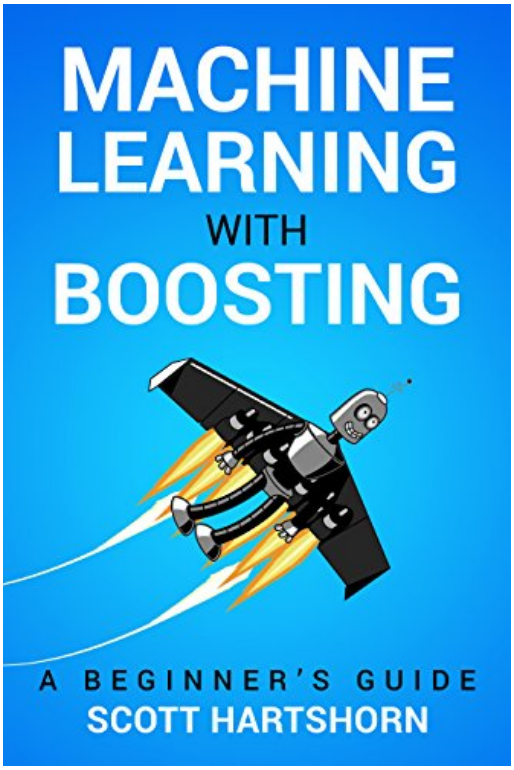
Machine Learning with Boosting - Book Review

RK

June 28, 2018

Abstract

This document contains a brief summary of the book "Machine Learning with Boosting : A Beginner's guide".



Context

The book gives the intuition behind *Gradient Boosting algorithm* via a set of simple examples. In the recent years Gradient Boosting has been widely used in many places, including Kaggle. As things stand, GBM and Deep Learning are two algos that are quintessential tools for an ML engineer. A few years ago, these might have been considered exotic tools but in today's world, they have become a part of default tool set.

The intuition behind *Decision trees* and *Random Forests* is somewhat easy to understand. However the intuition behind Gradient Boosting is not so straightforward. This book does a fantastic job of illustrating all the nuts and bolts of the algorithm.

Can't one read the math and figure out everything about the algo ? You could. However knowing the way it works for simple examples can serve as a nice primer before you head in to the math. In all likelihood, the math that you are going to encounter in the literature will be more clear to you, if you have this sort of background reading.

There are different types of boosting algorithms that are available and all of them are characterized by two key features

- Multiple iterations
- Each subsequent iteration focuses on the parts of the problem that previous iterations got wrong

There are many ways to characterize *Gradient Boosting Algorithms* but the best one liner to summarize this algos is as follows

> Group of weak learners can be combined to form a strong learner

Analogy to understand the algo

The author starts off with an example of an algo followed by a music teacher who wants to improve the average performance of the class. The music teacher is not interested in creating exceptional performers in his class. He is more interested in elevating the average performance of the entire class. At the beginning of his term, there is no choice but to create a standardized lesson for his entire class. Based on the initial feedback from all the class participants, he can customize his lessons so that poor performers can improve. This is in line with his objective function of improving the overall performance of the class. This algo of tweaking his instructions based on the repeated feedback obtained from all his students is something that can be seen in the workings of Gradient Boosting Algorithm.

The above analogy can only take us a few steps. We need to be precise when it comes to any algo. As mentioned earlier, the USP of this algo involves combining a group of weak learners. Plain English can only take us so far. We need to know precisely about what a weak learner is? We need to know a way to combine weak learners. We also need to think of whether some set of weak learners are better than others. Firstly, What is a weak learner ? It is a machine learning algo that gives better accuracy than simple guessing. For boosting problems, the best kind of learners are the ones that are very accurate, even if it is only for a limited

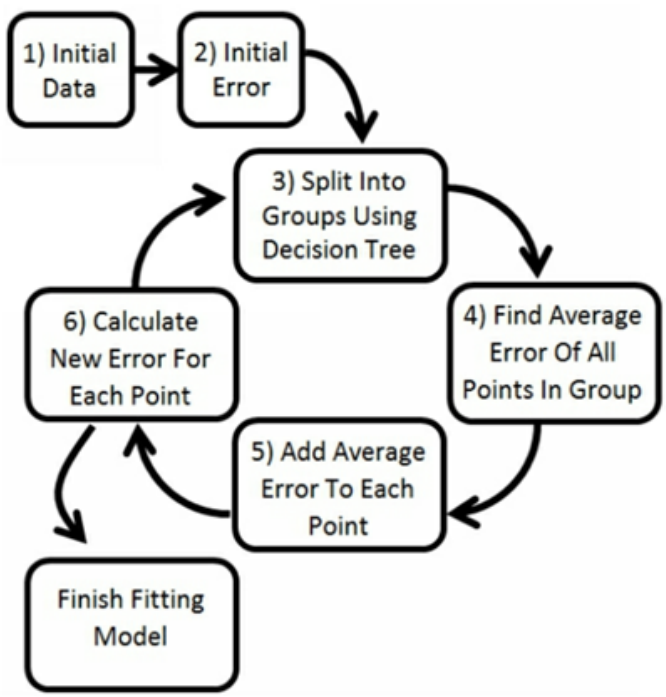
scope of a problem. Now the question of combining learners. How does one combine various learners? Another analogy from the author helps us understand the basic idea

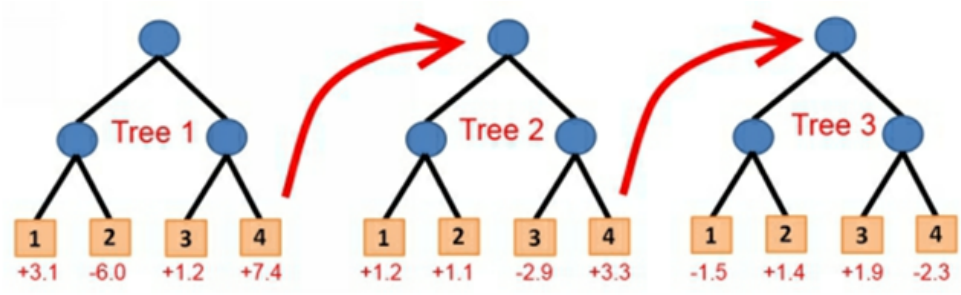
> Imagine you are hiring people to build your house, and you have 10 different big jobs that need to be done. A great way of doing it would be to get someone who is really good at foundations to build the foundation. Then hire a very good carpenter to focus on the framing. Then hire a great roofer and plumber to focus on their sections of the house. At each stage, a small subsection of the project is getting completely solved. The roofer may be completely useless at laying foundations, but as long as you use him at the right time you will be in good shape. The contrast to that method would be to hire 10 people who are all decent at most things, but not great at anything. None of them can build you a good foundation, and if you start with one, the next one will have to come in and fix some problems with it, while at the same time doing a shoddy job framing. The third person will have to come in and make corrections to the errors that the first two left behind. You might get a good product at the end, but more likely you will have adequate results that still have errors in them.

Weak learners are best combined in such a way that allows each one to solve a limited section of the problem

What's the basic idea behind the algo ?

At a 10,000 ft, this is the basic structure of a Gradient Boosting Algo: One needs to split the data in to groups with similar error. One needs to adjust the values in the groups and iterate until some stopping criterion is hit. The computationally expensive part of the algo is determining the best way to group the data. Once the grouping is done, an average error for each group is computed and that value is used to adjust the values of each member in the respective group. In the next boosting cycle, a data point might fall in a group that is different from the previous group. The following visual summarizes this process.





What is the final result of a gradient boosting algo ?It is a set of splits that created various groups for each tree, and the amount of change applied to each group

There are two types of algos that are applied in the context of Gradient Boosting, one for Regression and other for Classification. Even though there are two algos, the method for splitting the data in to various groups is same, i.e. regression decision trees. Both the regression and classification boosting algorithms take their grouped data and calculate a value that gets added to or subtracted from the data points in that group The method for calculating the amount of change to apply is different for regression boosting vs classification boosting. GB for regression task is slightly easier than classification task as the former involves adjustments that are in the same scale as the target value

Since Gradient Boosting creates trees, the author walks the reader through the basics of Decision tree algo in a few steps.

Regression Boosting example

Let's say we want the machine to learn the following function

$$y = \sin(x) + 1 \tag{1}$$

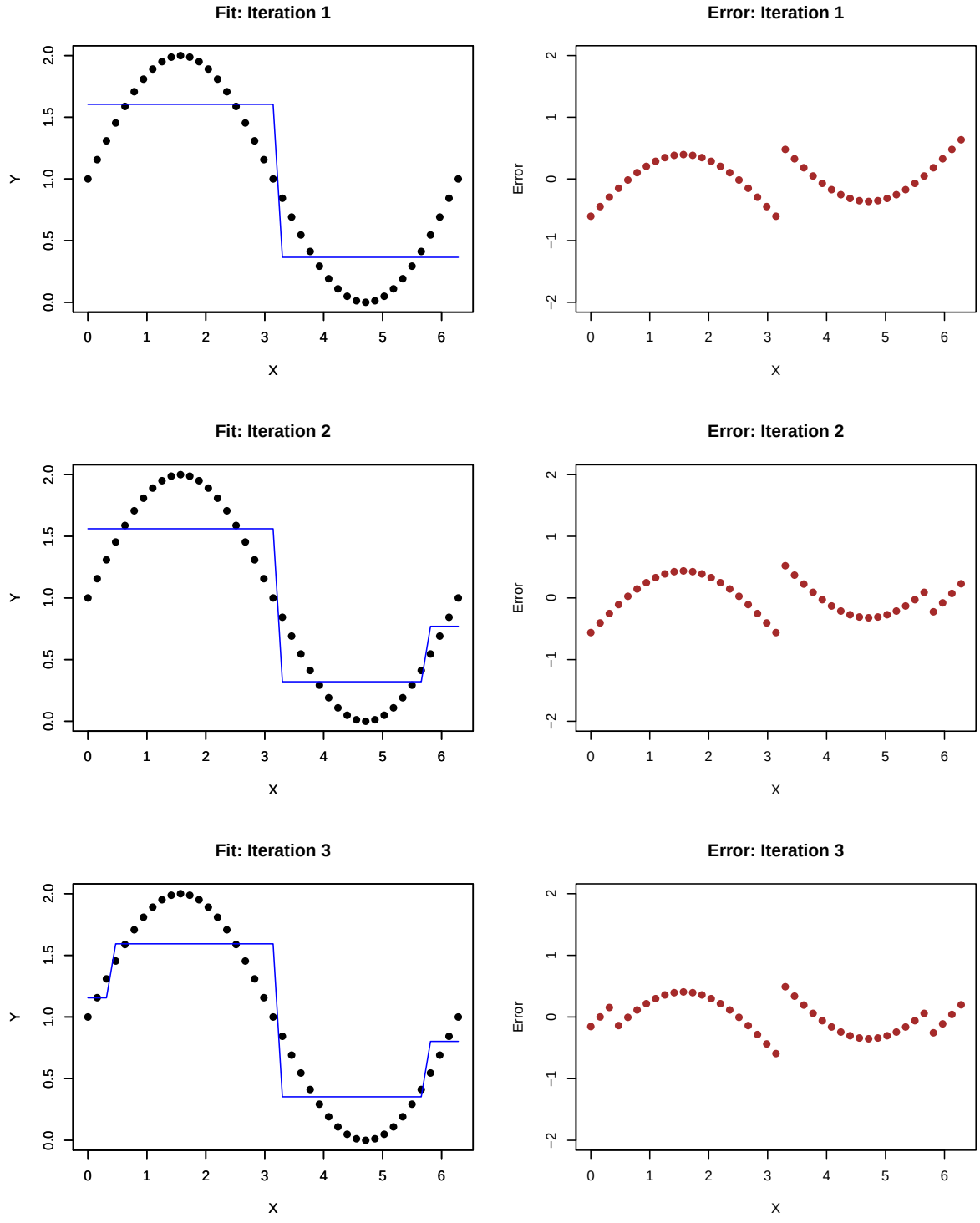
The following are the steps that the algo takes:

1. Calculate Initial Error
2. Split the Data in to Groups
3. For Each Group, find the average error
4. Adjust the current prediction
5. Calculate the New Error
6. Repeat the above cycle

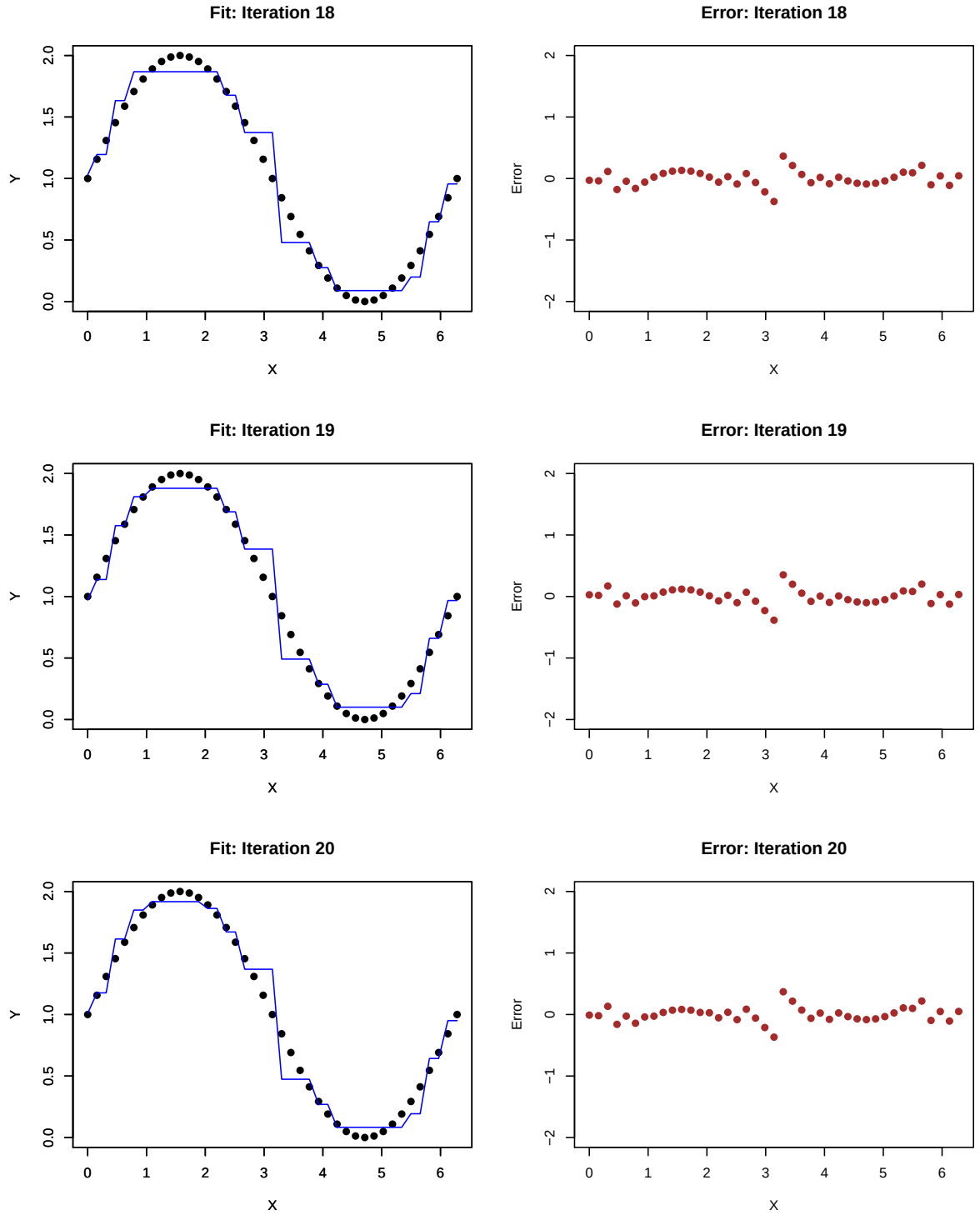
The following code can be used to generate training data :

```
k                       <- seq(0,20, by = 0.5)*pi/10
master_data             <- cbind(k,sin(k) + 1)
master_data             <- as.data.frame(master_data)
colnames(master_data)   <- c("x","y")
master_data$err         <- master_data$y
master_data$pred        <- 0

get_MSE <- function(k0, target){
    k       <- seq(0,20, by = 0.5)*pi/10
    set1    <- master_data[master_data$x<=k0*pi/10,]
    set2    <- master_data[master_data$x>k0*pi/10,]
    mu1     <- mean(set1[,target])
    mu2     <- mean(set2[,target])
    error_1 <- sum((set1[,target] -  mu1)^2)
    error_2 <- sum((set2[,target] -  mu2)^2)
    error_1 + error_2
}
target       <- "err"
iterations   <- 3
```

The following shows the result of fitting for the first three iterations:



The following is the result of the last three iterations out of a total of 20 iterations:

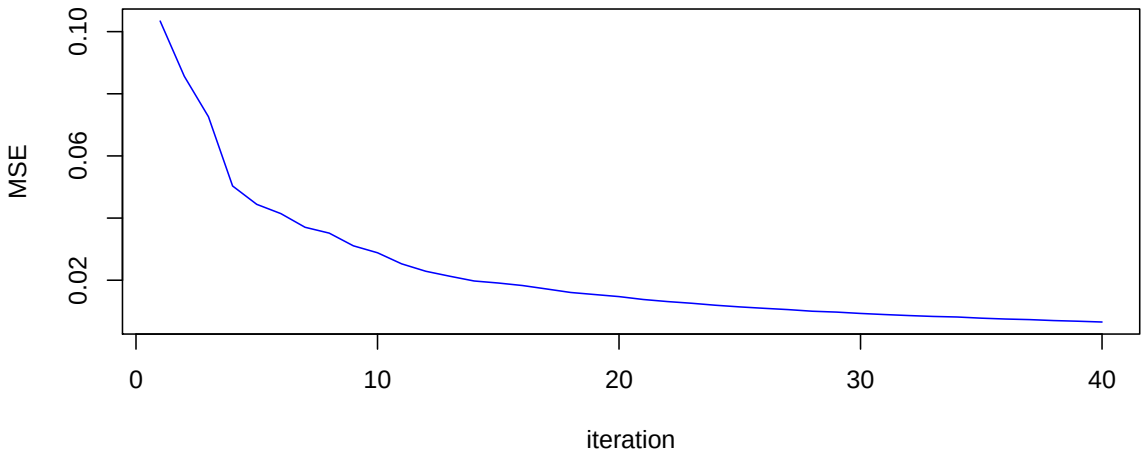


As one can see, the remaining error is more or less close to 0 and the result of further iterations might not fetch great dividends. One must note that all these iterations are being done with a depth level of 1, i.e. for any decision tree in the iteration there is only one cut.

What is the accuracy for the algo if we increase the number of iterations. The following code computes the error for 40 iterations

```
k                      <- seq(0,20, by = 0.5)*pi/10
master_data            <- cbind(k,sin(k) + 1)
master_data            <- as.data.frame(master_data)
colnames(master_data)  <- c("x","y")
master_data$err        <- master_data$y
master_data$pred       <- 0
target                 <- "err"
iterations             <- 40
results                <- rep(0, iterations)
for(i in seq_len(iterations)){
    search_space     <- seq(0,20, by = 1)
    z                <- cbind(search_space, sapply(search_space, get_MSE,target))
    cutoff           <- z[which.min(z[,2])]*pi/10
    set1             <- master_data[master_data$x<=cutoff,]
    set2             <- master_data[master_data$x>cutoff,]
    mu1              <- mean(set1[,target])
    mu2              <- mean(set2[,target])
    master_data$pred <- master_data$pred + ifelse(master_data$x <= cutoff, mu1,mu2)
    master_data$err  <- master_data$y - master_data$pred
    results[i] <- mean(master_data$err^2)
}
```

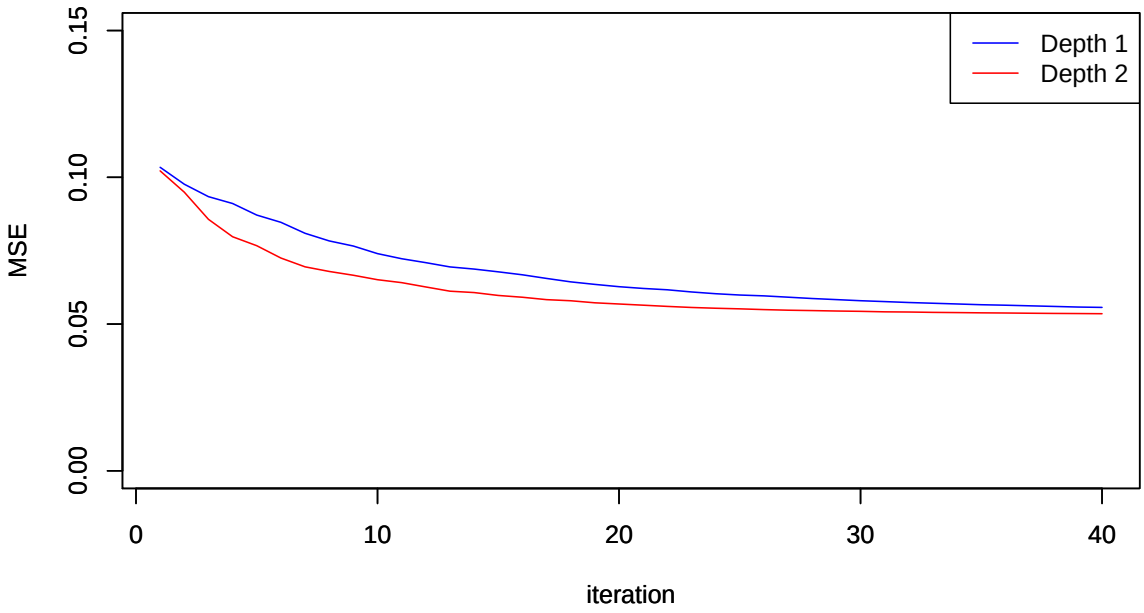
The following shows that the error levels off after certain number of iterations.



There is another way to improve the performance of the algorithm, i.e. by increasing the depth of the tree at each iteration. The following code compares the algo with varying depth

```
iterations <- 40
get_mse_gbm <- function(depth){
    fit <- gbm(y ~ x ,data = master_data,
               n.trees = iterations,
               bag.fraction = 1,
               shrinkage = 1,
               interaction.depth = depth)
    n.trees <- seq_len(iterations)
    fitted <- predict(fit, master_data, n.trees=n.trees)
    results <- apply(fitted, 2, function(x) mean((x-master_data$y)^2 ))
    return(results)
}
```

```
## Distribution not specified, assuming gaussian ...
## Distribution not specified, assuming gaussian ...
```



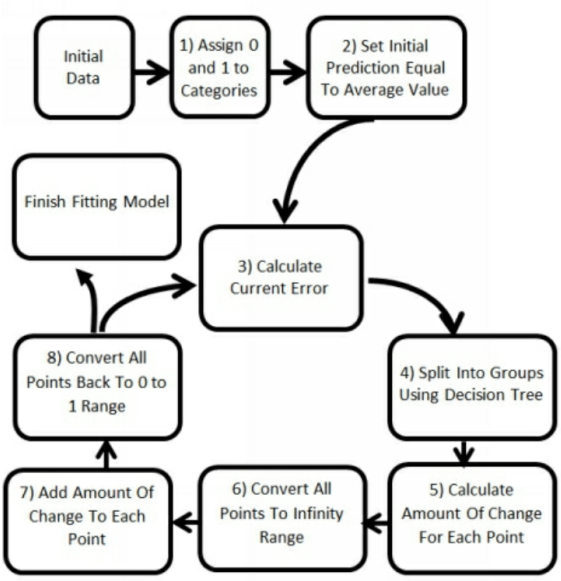
This toy example does make on think about several aspects such as,

- Can one just increase the depth and get a better fit ? Well ,there is a tradeoff. If you increase the depth, the computational time goes up as each iteration, the feature space has to be segregated in to more categories

- What if I don't care about time and just increase the depth ? There is another trade-off, i.e. the higher the depth, the higher the chance of overfitting as weak learns are supposedly shallow trees. By increasing the depth, the weak learners tend to overfit to specific regions
- What should be learning rate that I need to choose ? Again this could be a hyperparameter that can be selected via cross validation. In one sense, you need to separate the dataset in to three categories - Training set, Test set for selecting hyper parameters via Cross validation and another test set to check the out of training sample error. Having said that a smaller learning rate tends to be perform better than higher learning rate for noisy datasets

Classification

Using Gradient Boosting for *Classification* has a few more additional steps as one needs to translate the factor variables to numerical values, adjust the numerical values based on the average error of the group and then rescale the adjusted error back to probabilities.



Also the algorithm is a bit different for multi category response variables from binary response variables.

The following summarizes the algo for two category response variable:

1. Renumber the categories to be 0 & 1 instead of whatever discrete values they have. Their 0 or 1 value will be the True Value for each data point. It doesn't matter which category becomes zero and which becomes one.
2. Make an initial prediction for each data point by dividing the total number of ones in your training data by the total number of data points overall.
3. Subtract your Current Prediction for each data point from the True Value. This is your Current Error
4. Use a Decision Tree regression analysis to fit a minimum mean squared error tree to the Current Error.

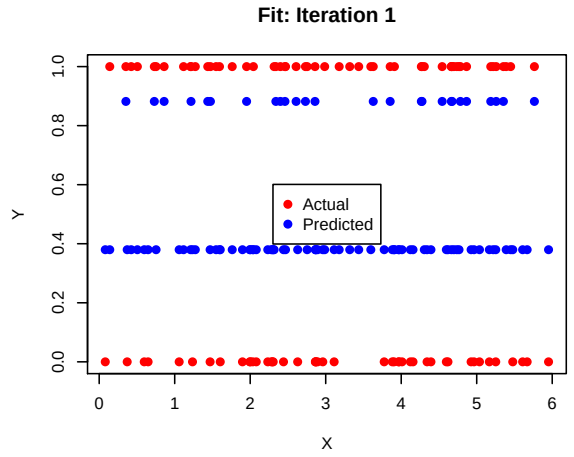
Ignore the actual average values that the regression tree fit to the Current Error, and just keep the groups.

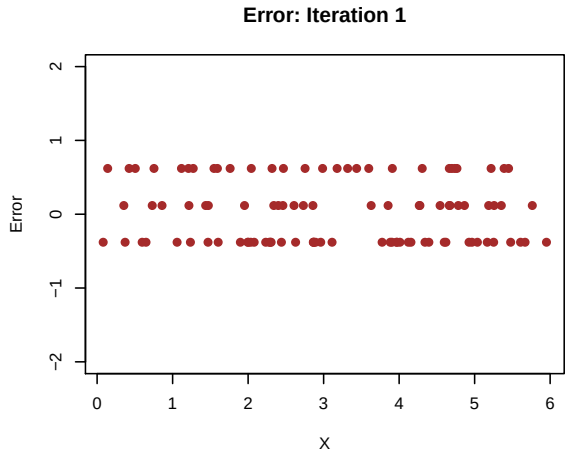
5. For each group, generate an equation based on how many points have positive error, and need to be moved up, and how many have negative error, and need to be moved down. Account for both the quantity of points and the magnitude of the error. This equation will generate either a positive or negative value in the range from negative infinity to positive infinity. Call this the Amount of Change
6. Convert your current prediction for each data point (from step 3 on the first iteration, from step 8 on subsequent iterations) to the negative infinity / positive infinity range. Call this the Modified Current Prediction
7. For each point, add the Modified Current Prediction (step 6) to the Amount of Change(step 5)
8. For each point, convert the result from step 7 from the negative infinity / positive infinity range back to the zero to one range. The is the new Current Prediction
9. Take the new Current Prediction and either start another boosting iteration at step 3 or be done. If you are done, the model has now been fitted, and you can use it to predict new data points.

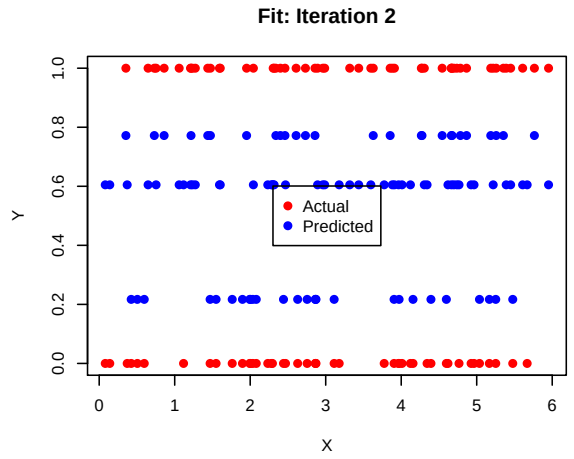
The following code can be used to generate training data :

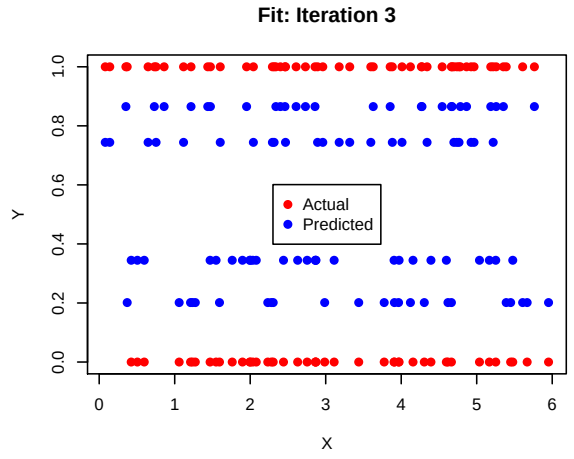
```
set.seed(1)
n                       <- 100
x                       <- runif(n)*6
y                       <- runif(n)*6
z                       <- as.integer(pmax(x,y)%%2)
pch                     <- ifelse(z==0,19,21)
col                     <- ifelse(z==0,"red","blue")
x1                      <- pmax(x,y)
master_data             <- data.frame(x = x1, y = z)
colnames(master_data)   <- c("x","y")
master_data$pred        <- mean(master_data$y)
master_data$err         <- master_data$y - master_data$pred

get_MSE <- function(k0, target){
    k       <- seq(0,6, by = 0.1)
    set1    <- master_data[master_data$x<=k0,]
    set2    <- master_data[master_data$x>k0,]
    mu1     <- mean(set1[,target])
    mu2     <- mean(set2[,target])
    error_1 <- sum((set1[,target] -  mu1)^2)
    error_2 <- sum((set2[,target] -  mu2)^2)
    error_1 + error_2
}

target       <- "err"
iterations   <- 3
```

The following shows the result of fitting for the first three iterations:

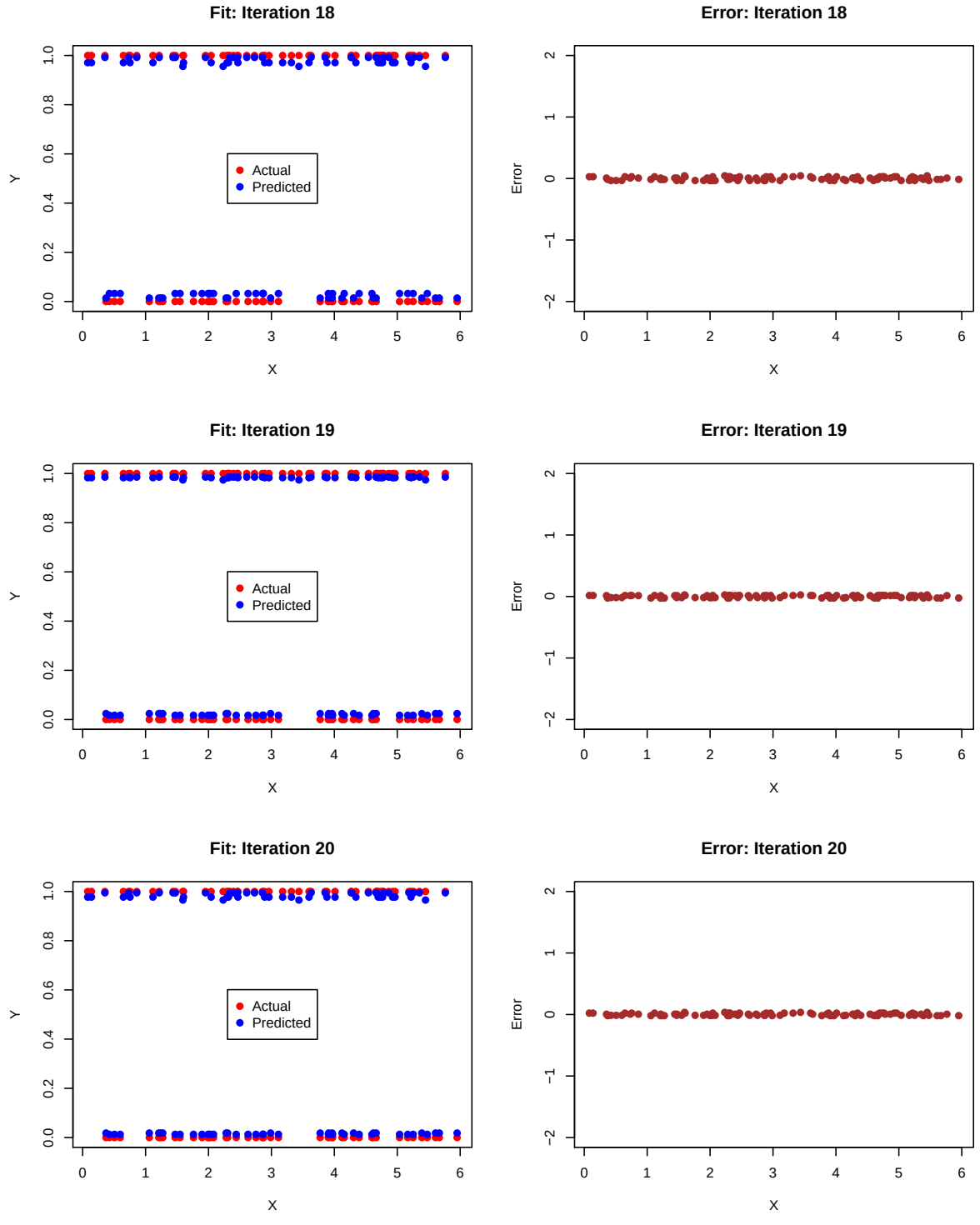








The following is the result of the last three iterations out of a total of 20 iterations:



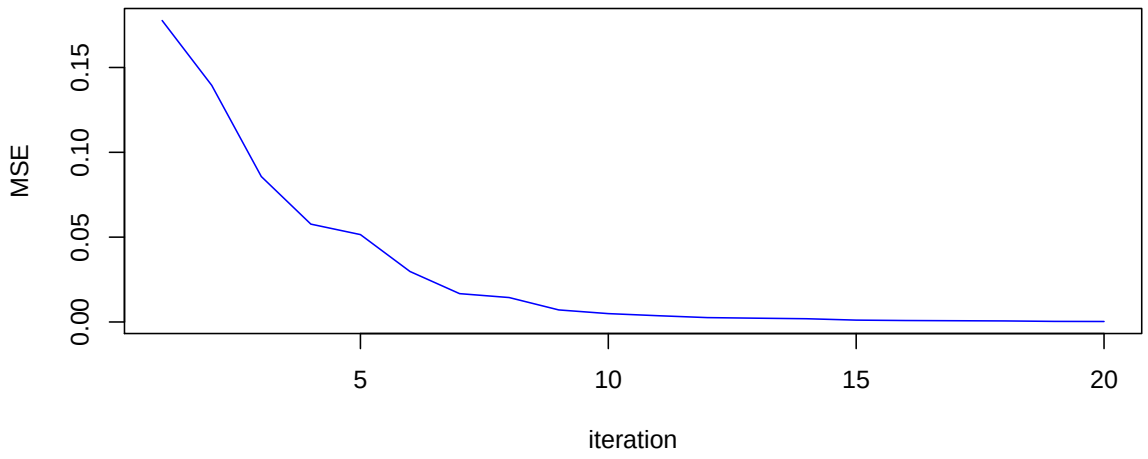
As one can see, the remaining error is more or less close to 0 and the result of further iterations might not fetch great dividends. One must note that all these iterations are being done with a depth level of 1, i.e. for any decision tree in the iteration there is only one cut.

What is the accuracy for the algo if we increase the number of iterations. The following code computes the error for 40 iterations :

```
set.seed(1)
n                       <- 100
x                       <- runif(n)*6
y                       <- runif(n)*6
z                       <- as.integer(pmax(x,y)%%2)
pch                     <- ifelse(z==0,19,21)
col                     <- ifelse(z==0,"red","blue")
x1                      <- pmax(x,y)
master_data             <- data.frame(x = x1, y = z)
colnames(master_data)   <- c("x","y")
master_data$pred        <- mean(master_data$y)
master_data$err         <- master_data$y - master_data$pred

get_MSE <- function(k0, target){
    k       <- seq(0,6, by = 0.1)
    set1    <- master_data[master_data$x<=k0,]
    set2    <- master_data[master_data$x>k0,]
    mu1     <- mean(set1[,target])
    mu2     <- mean(set2[,target])
    error_1 <- sum((set1[,target] -  mu1)^2)
    error_2 <- sum((set2[,target] -  mu2)^2)
    error_1 + error_2
}

target       <- "err"
iterations   <- 20
results                 <- rep(0, iterations)
for(i in seq_len(iterations)){
    search_space     <- seq(0,6, by = 0.1)
    z                <- cbind(search_space, sapply(search_space, get_MSE,target))
    cutoff           <- z[which.min(z[,2])]
    set1             <- master_data[master_data$x<=cutoff,]
    set2             <- master_data[master_data$x>cutoff,]
    num1             <- sum(set1$y - set1$pred)
    num2             <- sum(set2$y - set2$pred)
    den1             <- sum(set1$pred*(1 - set1$pred))
    den2             <- sum(set2$pred*(1 - set2$pred))
    delta1           <- num1/den1
    delta2           <- num2/den2
    cur_pred1        <- log((set1$pred)/(1-set1$pred))
    cur_pred2        <- log((set2$pred)/(1-set2$pred))
```

```
    mod_pred1        <- cur_pred1 + delta1
    mod_pred2        <- cur_pred2 + delta2
    new_cur_pred1    <- 1/(1+exp(-mod_pred1))
    new_cur_pred2    <- 1/(1+exp(-mod_pred2))
    set1$pred        <- new_cur_pred1
    set2$pred        <- new_cur_pred2
    master_data      <- rbind(set1, set2)
    master_data$err  <- master_data$y - master_data$pred
    results[i] <- mean(master_data$err^2)
}
```

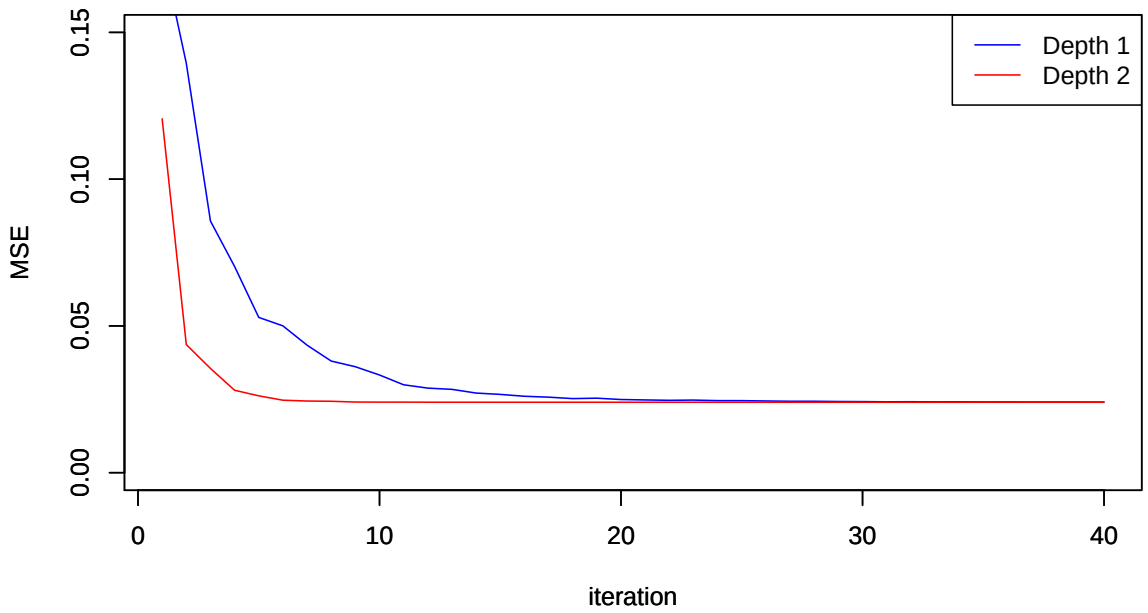
The following shows that the error levels off after certain number of iterations.



There is another way to improve the performance of the algorithm, i.e. by increasing the depth of the tree at each iteration. The following code compares the algo with varying depth

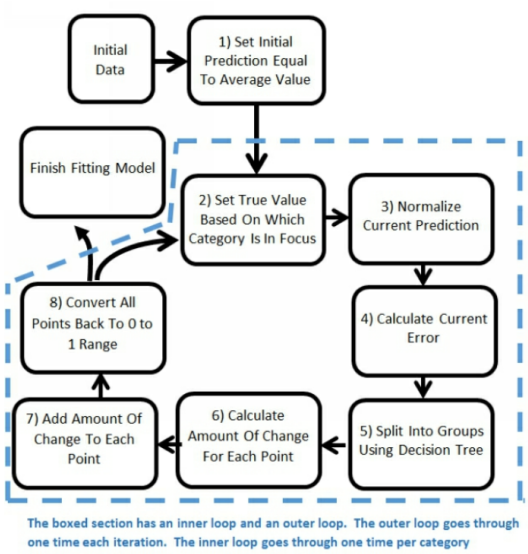
```
iterations <- 40
get_mse_gbm <- function(depth){
    fit <- gbm(y ~ x ,data = master_data,
               n.trees = iterations,
               bag.fraction = 1,
               shrinkage = 1,
               interaction.depth = depth)
    n.trees <- seq_len(iterations)
    fitted <- predict(fit, master_data, n.trees=n.trees)
    fitted <- 1/(1+exp(-fitted))
    results <- apply(fitted, 2, function(x) mean((x-master_data$y)^2 ))
    return(results)
}
```

```
## Distribution not specified, assuming bernoulli ...
## Distribution not specified, assuming bernoulli ...
```



Classification with Multiple Categories

The algo for classifying multiple categories is a little more detailed. The following visual summarizes the basic steps :



Other Topics

The following are some of the other topics covered in the book

- Feature Importances
- Feature Engineering
- Sub-Sampling
- Max Features Hyper parameter
- Various parameters that one need to pass in to the various `Python` or `R` implementation libraries for Gradient Boost algorithm

Takeaway

Gradient Boosting Algorithm is one of the powerful algos out to solve classification and regression problems. This book gives a gentle introduction to the various aspects of the algo with out overwhelming the reader with the detailed math of the algo. The fact that there are a many visuals in the book makes the learning very sticky. Well worth a read for any one who wants to understand the intuition behind the algo.