Boosting 101

Matt Richey

04/10/2018

Boosting

The idea behind boosting is that we employ "weak learning" over and over again in order to get a suitable modelimation to the data. In what follows, the weak learner will involve small trees, depth =1 (stumps) or depth=2

Libraries

```
library(tidyverse)
```

```
## -- Attaching packages --------------------------- tidyverse 1.3.0 --

## v ggplot2 3.3.0     v purrr   0.3.3
## v tibble  2.1.3     v dplyr   0.8.5
## v tidyr   1.0.2     v stringr 1.4.0
## v readr   1.3.1     v forcats 0.5.0

## -- Conflicts ------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

```
library(tree)
```

```
## Registered S3 method overwritten by 'tree':
##   method     from
##   print.tree cli
```

Include a different tree package. The rpart library has works almost the same way, but has a few minor differences.

```
library(rpart)
```

Boosting by hand

Let's explore the idea of boosting by doing it "by hand". To start, create a synthetic dataset builder.

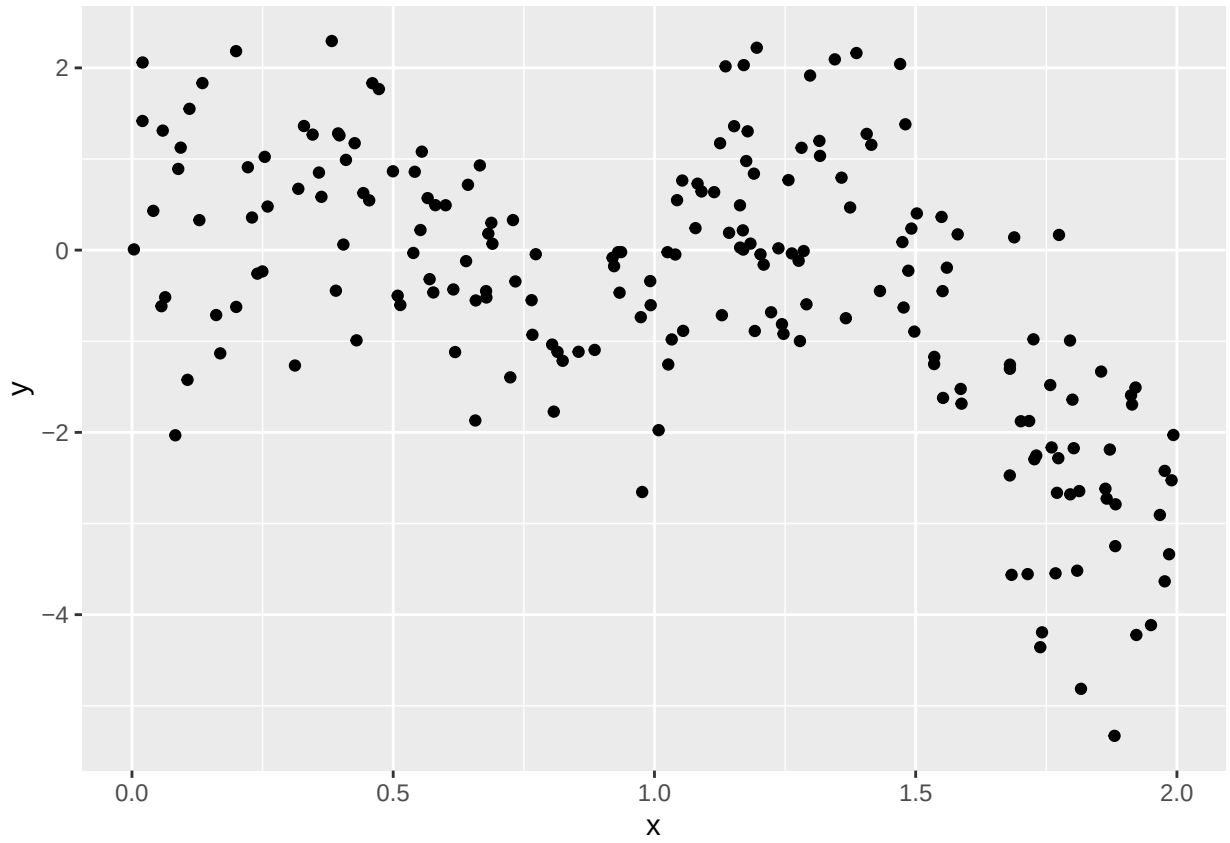
```
buildData<-function(n,sig=.5,S=.1){
  x<-runif(n,0,2)
  y<-x*(1-x)+sin(6*x)
  dat<-y+rnorm(n,0,sig)
  data.frame(x,y=dat)
}
```

Build a synthetic training data set with 200 observationsl

```
n <- 200
sig <- 1.0
train.df <-buildData(n,sig)
```

What are we looking at?

```
train.df %>%
  ggplot()+
  geom_point(aes(x,y),color="black")
```



Not too exciting. It clearly isn't linear and pretty noisy.

The idea behind boosting is to build very simple models (weak learners) and use these to whittle away at the residues.

- The initial model is very simple, namely, the zero model.

- The initial state of the residuals is just the response.

Add this information to the data.

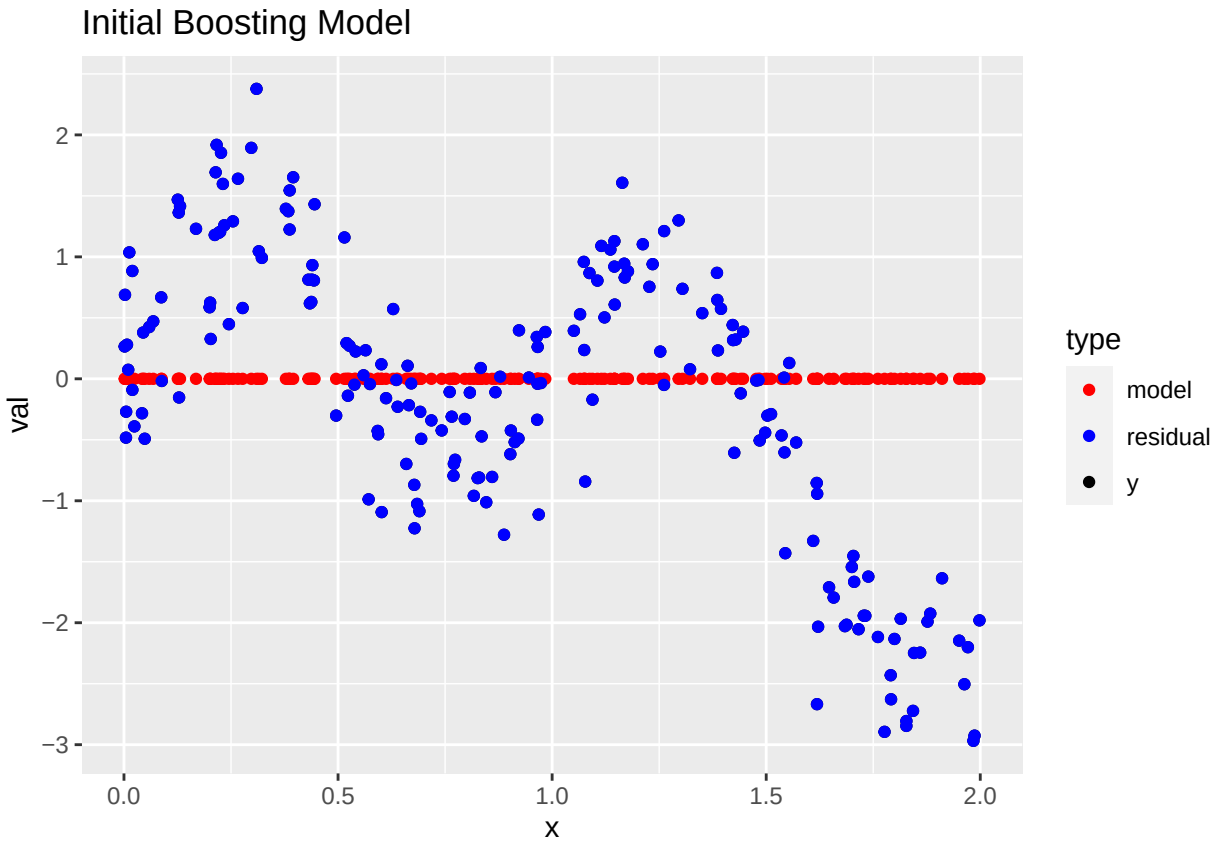
```
train.df <-buildData(n) %>%
  mutate(model  = 0,
           ## initial residuals
           residual=y)
```

As we go along, the model and the residuals will change. The model will (hopefully) get closer to the response. At the same time, the residuals should decay to zero.

We will keep looking at this picture. It has the current modelimation, the target function, and the residuals

Initial state..

```
train.df %>%
  gather(type,val,y:residual) %>%
  ggplot()+
  geom_point(aes(x,val,color=type))+
  scale_color_manual(values=c("red","blue","black"))+
  labs(title="Initial Boosting Model")
```



Weak learning.

For our weak learner, we'll use a basic stump of a tree (one branch, two leaves).

Also, we define a shrinkage factor by which we will scale (down) the weak learner's effect!

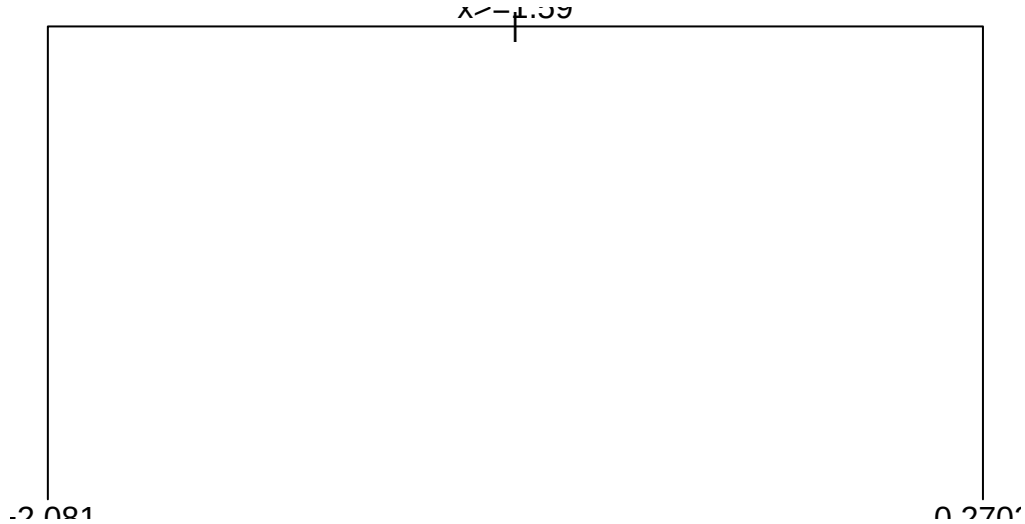
Fix the depth and the shrinkage factors

```
depth <- 1 ##for a stump
lambda <- 0.5 ## shrinkage
```

Use rpart to build a tree (stump in this case).

Notice how we predict the residuals, not the response.

```
tree0 <- rpart(residual ~ x,
               data=train.df,
               control=rpart.control(maxdepth=depth))
{plot(tree0);text(tree0,pretty=0,cex=1)}
```



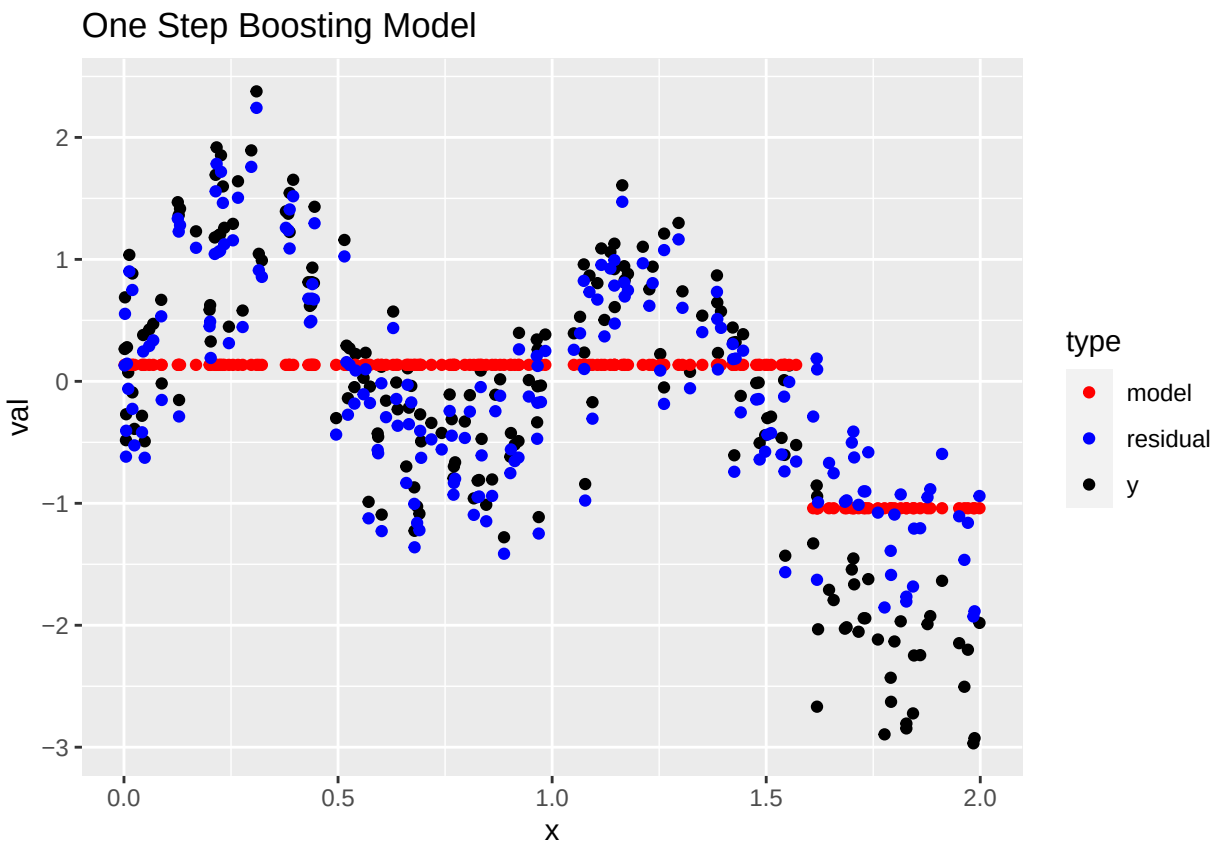
Nice stump.

Now we can make our predictions using the weak learner (stump). After we do so, slice off just a little bit of the predictions off the residuals and add them to the model compute the residuals and current modelimation

```
preds <- as.numeric(predict(tree0))
train.df <- train.df%>%
  ##only use a small about of the residuals..."sanding"
  mutate(residual=residual-lambda*preds,
         ##update the model
         model=model+lambda*preds)
```

The state of affairs right now

```
train.df %>%
  gather(type,val,y:residual) %>%
  ggplot()+
  geom_point(aes(x,val,color=type))+
  scale_color_manual(values=c("red","blue","black"))+
  labs(title="One Step Boosting Model")
```



We see a smaller residual and (ever so) slightly better model. This is a weak learner how a weak learner works.

The plan is to repeat this process a large number of times. That's the idea behind boosting. You do very little at any one step, but you do it a lot, like sanding a piece of furniture.

Boosting Function

Make the boosting a function of the data, lambda, and the depth of the tree. Note that his function returns a list consisting of the new data frame and the model that was used.

```
doBoost<-function(data.df,lambda,depth){
  ##Build a simple tree
    tree0 <- rpart(residual~x,
                   data=data.df,
                   control=rpart.control(maxdepth=depth))
```

```
    ##make the predictions
    preds <- as.numeric(predict(tree0))
    ##update the data
    data.df <- data.df %>%
        mutate(residual=residual-lambda*preds,
               model=model+lambda*preds)
    ##return the data and the model
    list(data=data.df,tree=tree0)
}
```

One step of boosting

Set up the variables and start over with our function.

```
train.df <-buildData(n) %>%
  mutate(model  = 0,
           ## initial residuals
           residual=y)
```

Set up the parameters

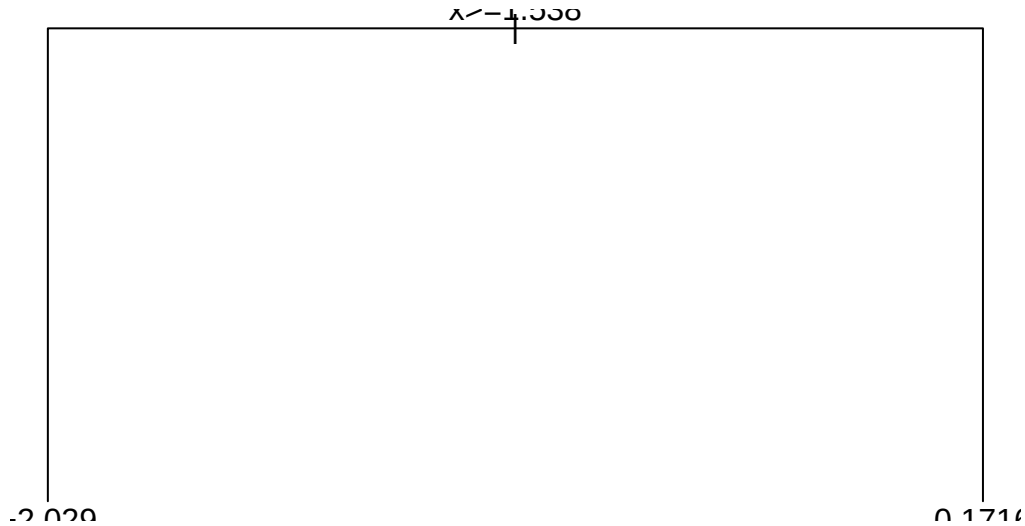
```
lambda<- 0.5
depth <-  1
data.df <- train.df
```

Boost it!

```
theBoost<-doBoost(data.df,lambda,depth)
```

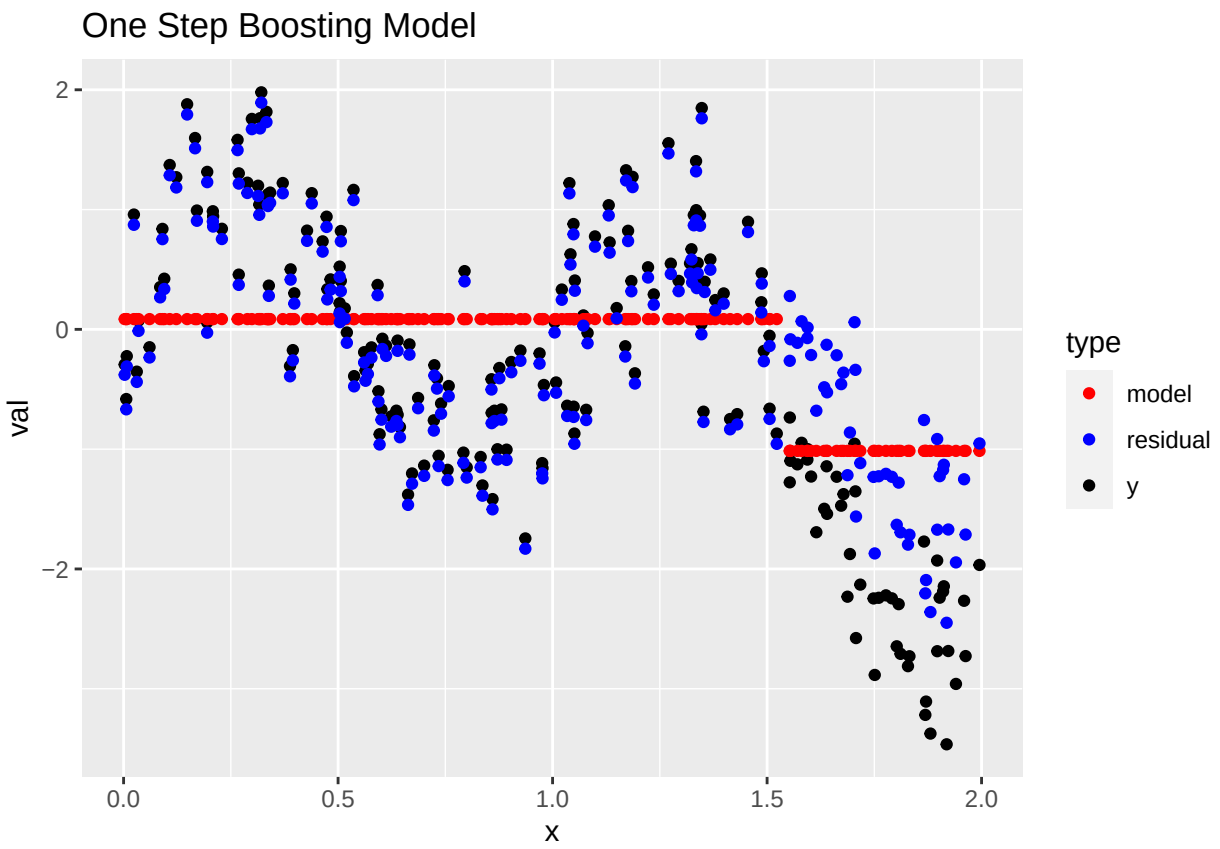
Pull off the data and the tree that was used

```
data.df <- theBoost[["data"]]
tree0 <- theBoost[["tree"]]
{plot(tree0);text(tree0,pretty=0,cex=1)}
```



Plot the data..again, a slightly better modelimation with slightly smaller residuals

```
data.df %>%
  gather(type,val,y:residual) %>%
  ggplot()+
  geom_point(aes(x,val,color=type))+
  scale_color_manual(values=c("red","blue","black"))+
  labs(title="One Step Boosting Model")
```



Looking good

Repeat the Boost

At each step, keep track of the new data frame and the model producted The choice of lambda is important. Here, we'll just use an arbitrary starting value. In general, this is something that is determined by cross-validation.

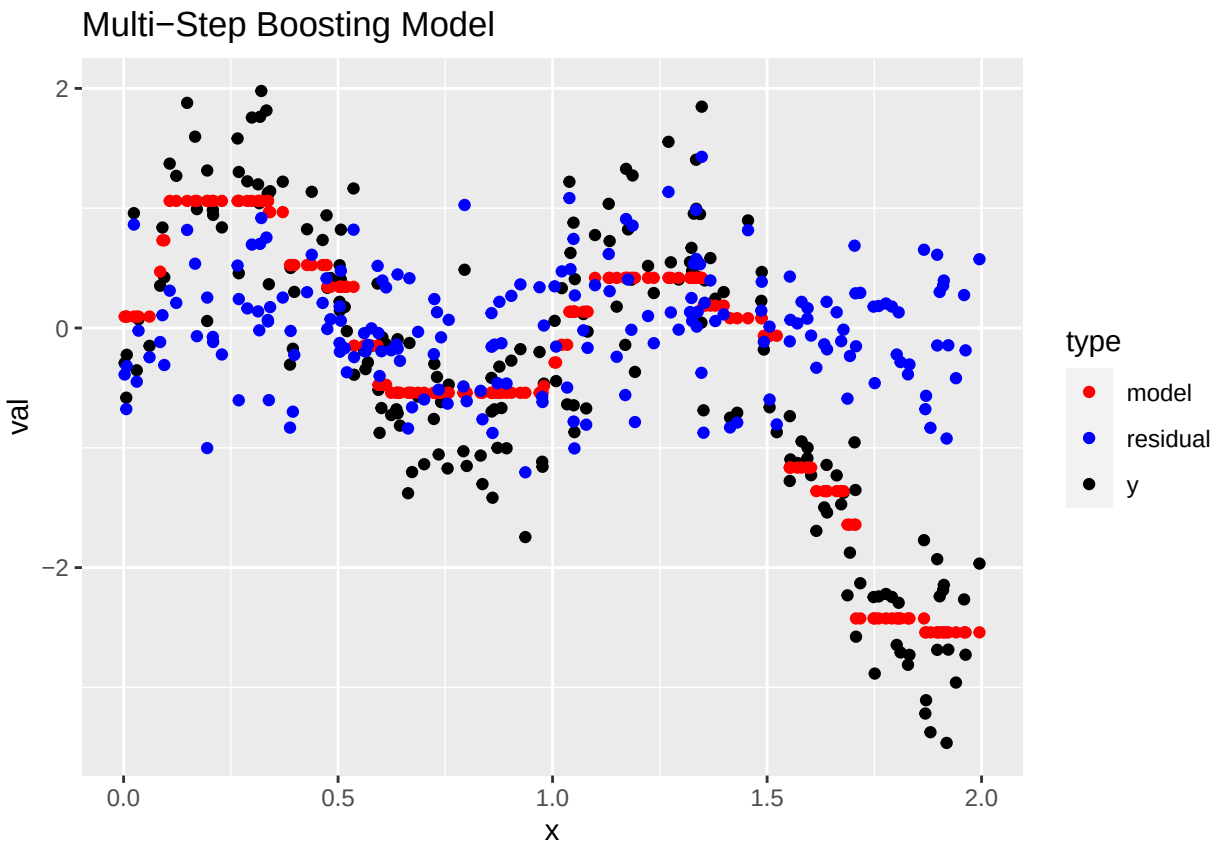
Boost numBoost times. Keep track of all the trees that are built along the way. These are the resulting model used for prediction. Use a list (not an array) to store these.

```
modelList <- list()
numBoost <- 100

for(i in 1:numBoost){
    theBoost<-doBoost(data.df,lambda,depth)
    data.df <- theBoost[["data"]]
    modelList[[i]]<- theBoost[["tree"]]
   ## print(i)
}
```

....and now look at a plot of the data

```
data.df %>%
  gather(type,val,y:residual) %>%
  ggplot()+
```

```
  geom_point(aes(x,val,color=type))+
  scale_color_manual(values=c("red","blue","black"))+
  labs(title="Multi-Step Boosting Model")
```



What does the last weak learner look like?

```
(lastModel <- modelList[[numBoost]])
```

```
## n= 200
##
## node), split, n, deviance, yval
##       * denotes terminal node
##
## 1) root 200 45.16332 5.551115e-19 *
```

The tree is just a root at this point. . . no further effect. Given the limitations of this simple implementation of boosting on a very simple data set, it just can't go any further!

This rarely see this with a a real data set.

Predictions with Boosted Weak Learners

The idea is to run a prediction through all the weak learners, take the sum of the weak predictions weighted by the shrinkage factor.

To do this we, need a simple helper function.

```
## return a list of all the predictions from a data frame
predictFromModels <- function(aModel,someData){
    as.numeric(predict(aModel, newdata=someData))
}
```

Test it out. . . it makes pedictions from a tree.

```
preds <- predictFromModels(modelList[[1]],
                           train.df)
## MSE of this model
with(train.df,mean((y-preds)^2))
```

```
## [1] 0.8943122
```

Here we go. To make predictions from a boosted model, compute the predictions from each of the trees. Then we combine the predictions (scaled by lambda).

```
maxMod <- numBoost
##sapply works best here  since we are re
predsAll.mat <- sapply(1:maxMod,function(i)
  predictFromModels(modelList[[i]],train.df))
## Dimensions
dim(predsAll.mat)
```

```
## [1] 200 100
```

The matrix predsAll.mat has:
* A row for each observation * A column for each weak learner

If we sum over each row and muliply by the shrinkage factor, we get our predictions.

```
predsAll.pred <- lambda*rowSums(predsAll.mat)
```

Put these into the data frame as the prediction.

```
dim(train.df)
```

```
## [1] 200   4
```

```
train.df$pred <- predsAll.pred
with(train.df,mean((y-pred)^2))
```

```
## [1] 0.4771399
```

We cut the error down significantly.

Once again, the plot of the data and the model

```
data.df %>%
  gather(type,val,y:residual) %>%
  ggplot()+
  geom_point(aes(x,val,color=type))+
  scale_color_manual(values=c("red","blue","black"))+
  labs(title="Final Boosting Model")
```

Make predictions on the test data.

If wanted, we could use this approach to make predictions on test data.

```
test.df <- buildData(400)
```

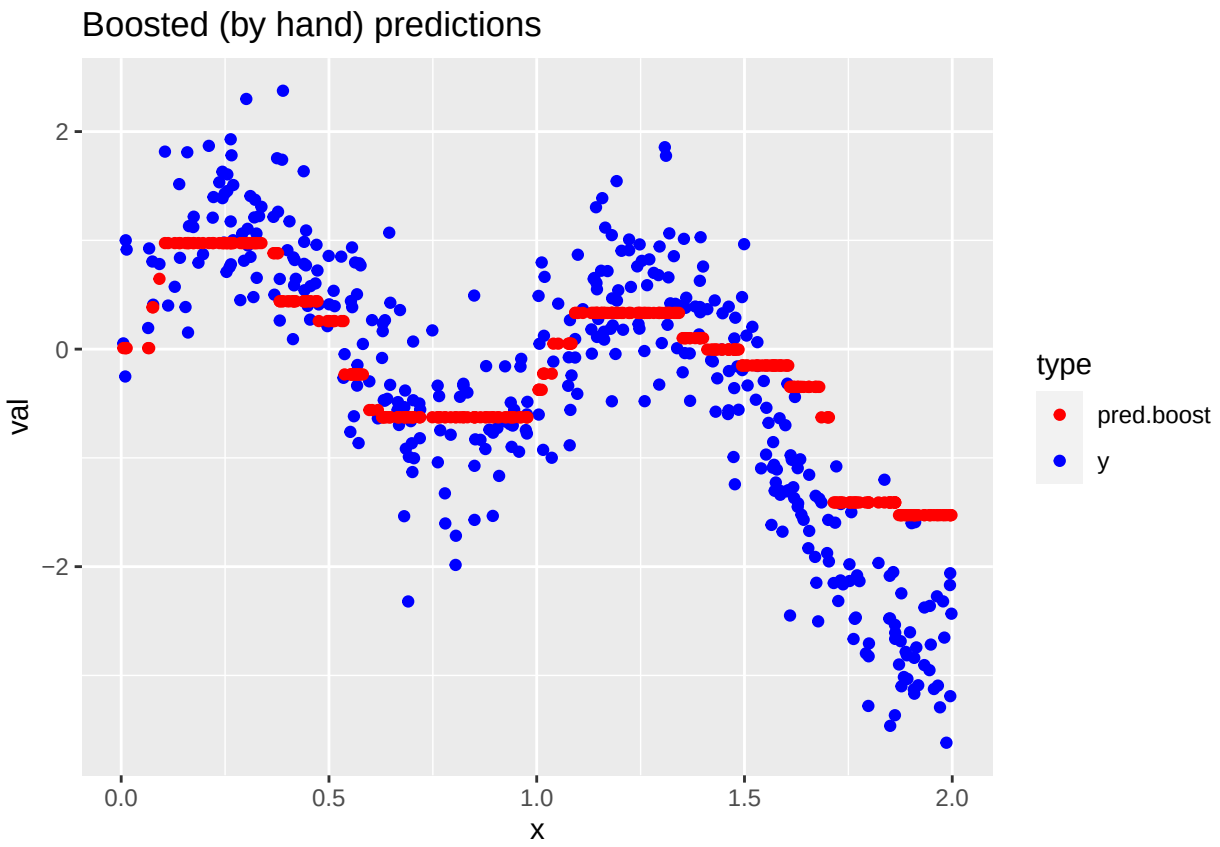
Prediction on the new data

```
predsAll.mat <- sapply(1:maxMod,function(i) predictFromModels(modelList[[i]],test.df))
predsAll.pred <- rowSums(predsAll.mat)*lambda
test.df$pred.boost <- predsAll.pred
```

How's it look?

```
test.df%>%
    gather(type,val,y:pred.boost)%>%
```

```
    ggplot()+
    scale_color_manual(values=c("red","blue","black"))+
    geom_point(aes(x,val,color=type))+
  labs(title="Boosted (by hand) predictions")
```



The MSE

```
(mse.boost <- with(test.df,
                   mean((y-pred.boost)^2)))
```

```
## [1] 0.5719727
```

Comparisons. . .

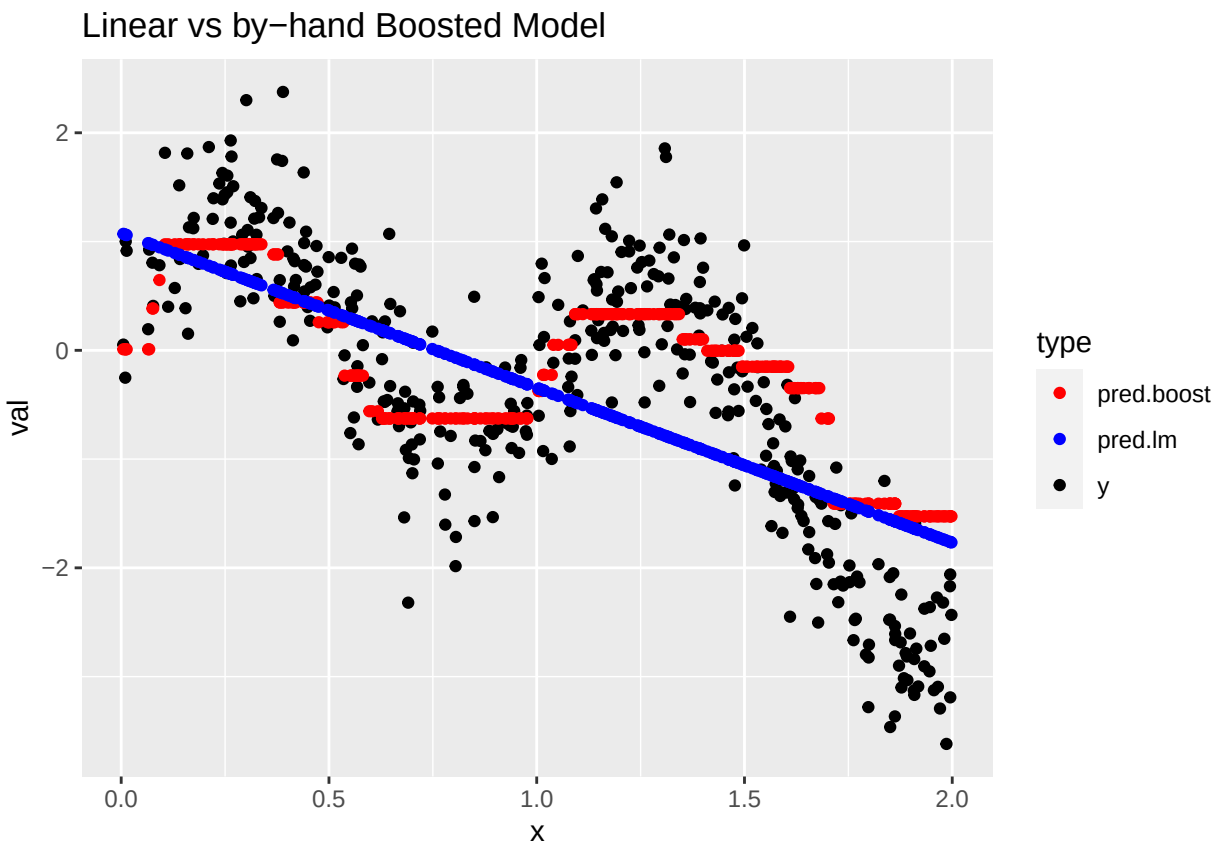
Compared to what?? How about a linear model?

```
mod.lm <- lm(y~x,data=train.df)
test.df$pred.lm <- predict(mod.lm,newdata=test.df)
(mse.lm <- with(test.df,mean((y-pred.lm)^2)))
```

```
## [1] 0.8077875
```

A visual comparison

```
test.df%>%
    gather(type,val,y:pred.lm)%>%
    ggplot()+
    scale_color_manual(values=c("red","blue","black"))+
    geom_point(aes(x,val,color=type))+
  labs(title="Linear vs by-hand Boosted Model")
```



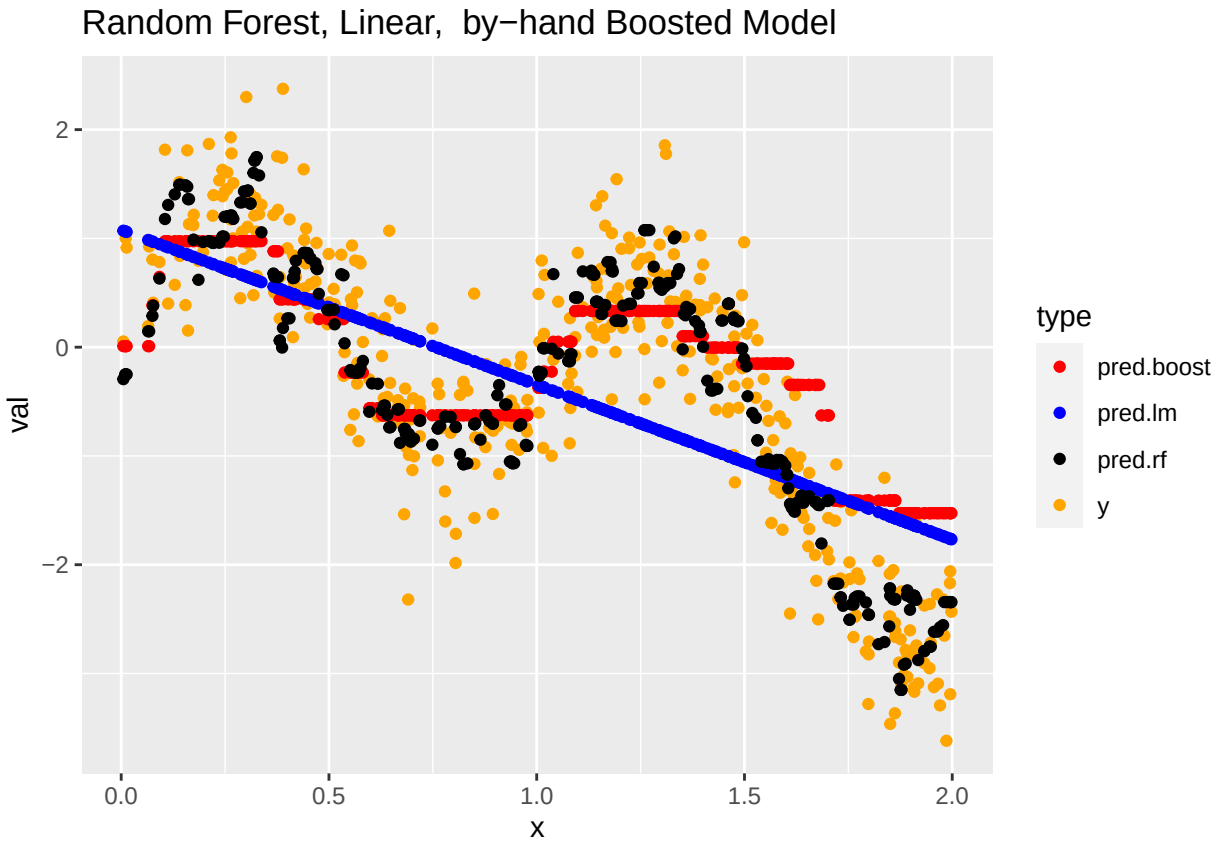
How about Random Forest?

```
library(randomForest)
```

```
## randomForest 4.6-14

## Type rfNews() to see new features/changes/bug fixes.

##
## Attaching package: 'randomForest'

## The following object is masked from 'package:dplyr':
##
##     combine

## The following object is masked from 'package:ggplot2':
##
##     margin
```

```
mod.rf <- randomForest(y~x,data=data.df)
test.df$pred.rf <- predict(mod.rf,newdata=test.df,
                ntree=1000)
```

```
test.df%>%
    gather(type,val,y:pred.rf)%>%
    ggplot()+
    scale_color_manual(values=c("red","blue","black","orange"))+
    geom_point(aes(x,val,color=type))+
  labs(title="Random Forest, Linear,  by-hand Boosted Model")
```



Compare the three different models

```
mse.rf <- with(test.df,mean((y-pred.rf)^2))
c(mse.boost,mse.lm,mse.rf)
```

```
## [1] 0.5719727 0.8077875 0.3449105
```

Random Forest wins, in this case.

R's Gradient Boosting Library

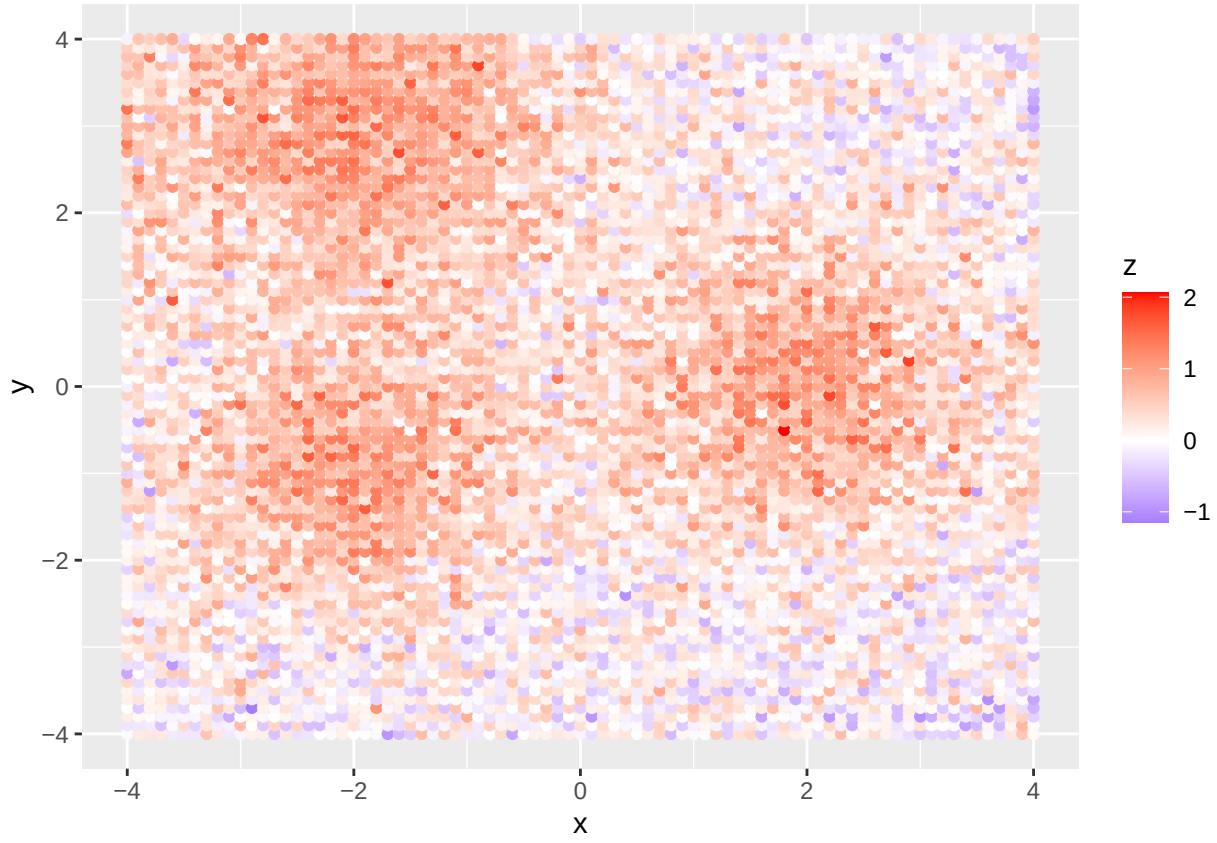
```
library(gbm)
```

```
## Loaded gbm 2.1.5
```

```
library(tidyverse)
```

As well, let's load in a slightly more interesting synthetic data set.

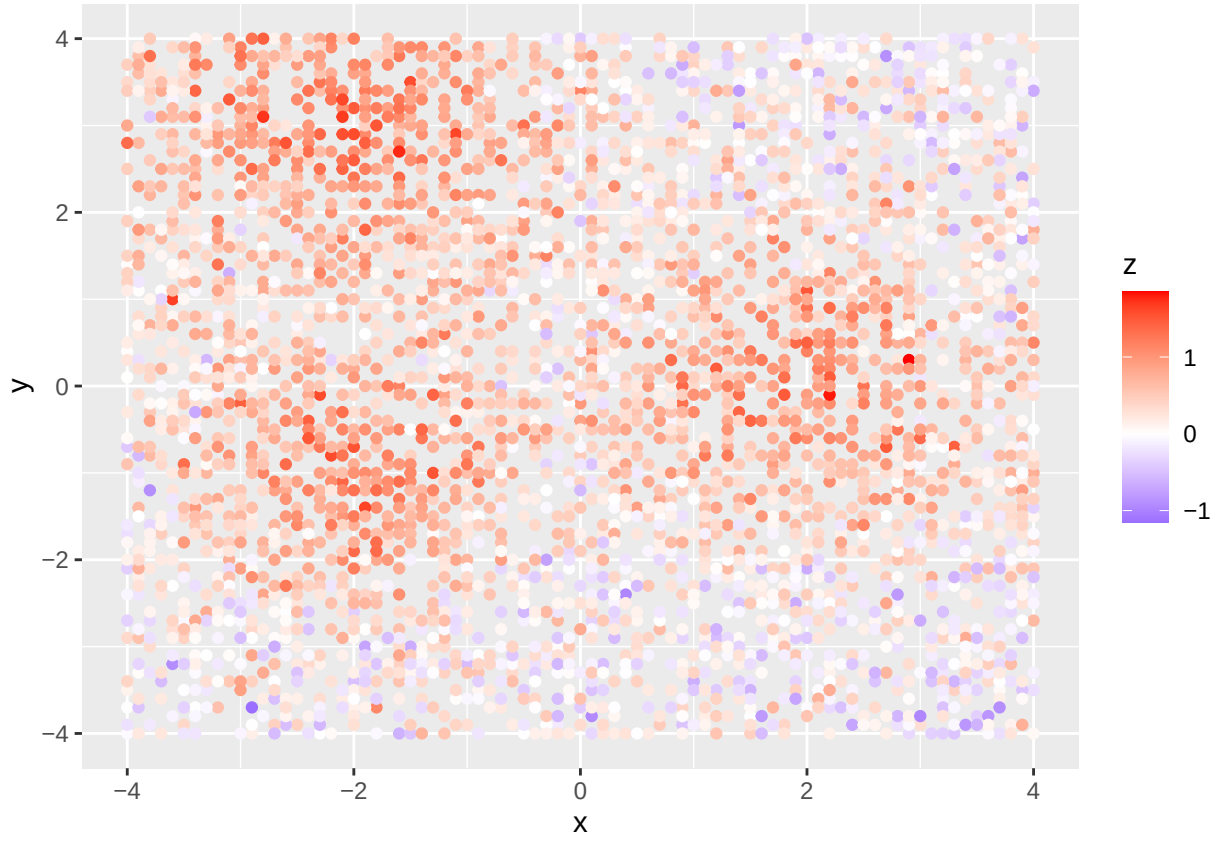
```
dataDir <- "/Users/richeym/Dropbox/COURSES/ADM/ADM_S20/CODE/Chap08"
dataFile <- "RectData_Quant.csv"
data2D.df <- read.csv(file.path(dataDir,dataFile))

data2D.df %>%
  ggplot()+
  geom_point(aes(x,y,color=z))+
  scale_color_gradient2(low="blue",high="red")
```



Build some smaller data sets to work with.

```
N <- nrow(data2D.df)
n <-  N/2
train <- sample(1:N,N/2,rep=F)

train.df <- data2D.df[train,]
```

```
test.df <- data2D.df[-train,]


train.df %>%
  ggplot()+
  geom_point(aes(x,y,color=z))+
  scale_color_gradient2(low="blue",high="red")
```



Now we can use the R version. Of course, this runs much faster with a lot more features. In particular, boosting has a number of control parameters.

- The shrinkage
- The number of learners
- Interaction Depth In practice, all of these parameters are in play and should be tuned via cross-validation.

Use the Boosting package.

Note that for regression, use distribution="gaussian".

Build a boosted model with many trees.

```
numTrees <- 2000
theShrinkage <- 0.1
theDepth <- 2
mod.gbm <- gbm(z ~ x+y,
```

```
                data=train.df,
                distribution="gaussian", ## for regression
                n.trees=numTrees,
                shrinkage=theShrinkage,
                interaction.depth = theDepth)
```

Now let's make predictions and compute the MSE on train and test data.

Like a random forest, you can specify how many learners to use (up to the total from the gbm model).

```
pred <- predict(mod.gbm,
                n.trees=numTrees, ## use them all
                newdata=train.df)
train.df$pred.gbm <- pred
(with(train.df,mean((z-pred.gbm)^2)))
```

```
## [1] 0.1040689
```

A good fit on the training data. Now the testing data.

```
pred <- predict(mod.gbm,
                n.trees=numTrees,
                newdata=test.df)
test.df$pred.gbm <- pred
(mse.gbm <- with(test.df,mean((z-pred.gbm)^2)))
```

```
## [1] 0.1273613
```

About the same.

How are doing? Compare with a random forest.

```
library(randomForest)
mod.rf <- randomForest(z ~ x + y,data=train.df,ntree=1000)
test.df$pred.rf <- predict(mod.rf,newdata=test.df,
                ntree=1000)
(mse.rf <-   with(test.df,mean((z-pred.rf)^2)))
```

```
## [1] 0.1329416
```

```
c(mse.rf,mse.gbm)
```

```
## [1] 0.1329416 0.1273613
```

Boosting, with the parameters above appears to be winning. However, there are a number of control parameters in gbm we need to be thinking about

GBM Optimization

First of all: You can overfit a boosted model!

To see this, let's build predictions on both train and test for different numbers of trees. Use th

Let's build our usual cross-validation function.

```
cvGBM <- function(data.df,
                  theShrinkage,
                  theDepth,
                  numTreesPred,
                  numFolds=5){
  n <- nrow(data.df)
  folds <- sample(1:numFolds,n,rep=T)
  errs <- numeric(numFolds)
  for(fold in 1:numFolds){
    train.df.cv <-   data.df[folds != fold,]
    test.df.cv <- data.df[folds == fold,]
    mod.gbm <- gbm(z ~ x + y,
                   data=train.df.cv,
                   distribution="gaussian",
                   interaction.depth = theDepth,
                   shrinkage=theShrinkage,
                   n.minobsinnode=10,
                   n.trees=numTrees)
    pred.gbm <- predict(mod.gbm,
                        newdata=test.df.cv,
                        n.trees=numTreesPred)
    errs[fold] <- with(test.df.cv,mean((z-pred.gbm)^2))
  }
  mean(errs)
}
```

A quick check on the full data set with all the trees and then with fewer trees.

```
cvGBM(data2D.df,theShrinkage, theDepth,  numTrees)
```

```
## [1] 0.12503
```

```
cvGBM(data2D.df,theShrinkage, theDepth,  numTrees/10)
```

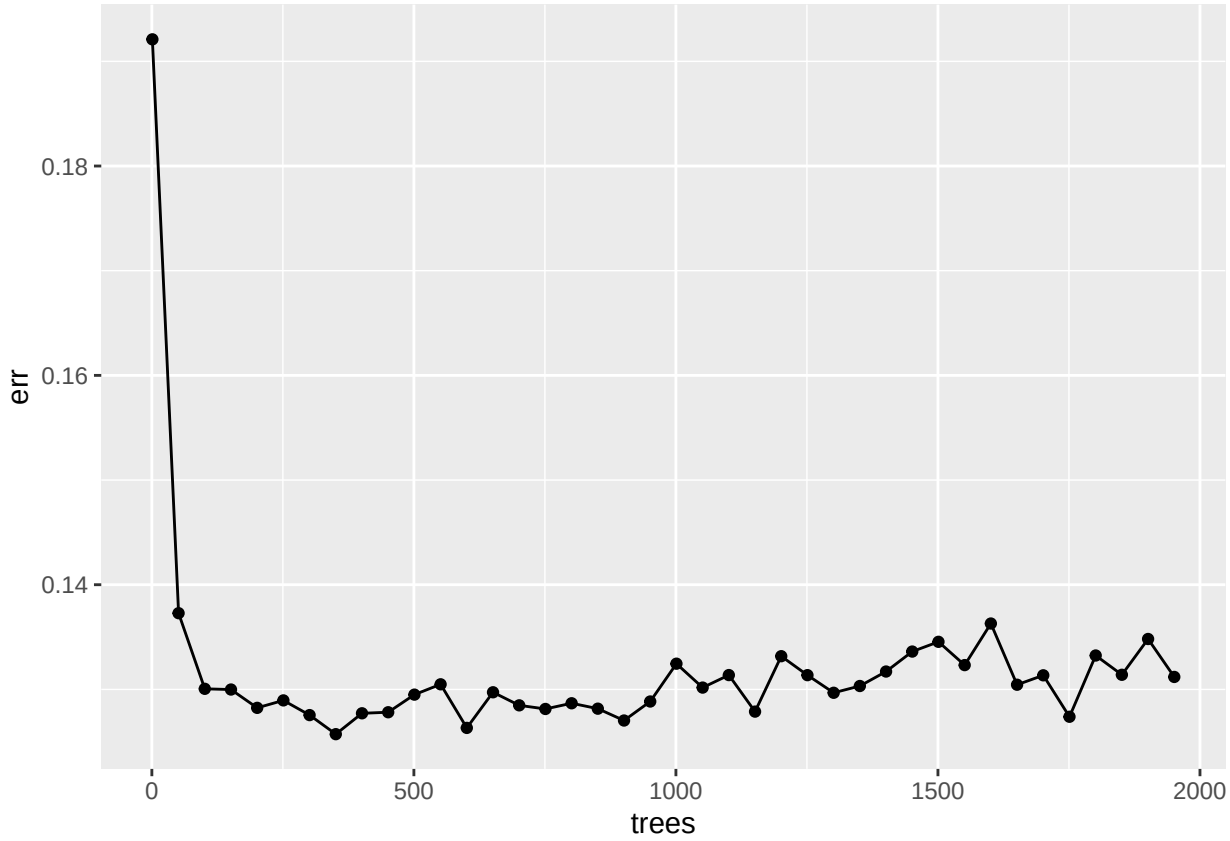
```
## [1] 0.1279657
```

To get an idea of the dependence on the number of trees, let's do a cross-validation on subset of the data (to speed things up).

```
trees <- seq(1,numTrees,by=50)
samp <- sample(1:N,N/5,rep=F) ##<---speed up
allErrs <- map_dbl(trees,
                   function(num){ print(num);cvGBM(data2D.df[samp,],
                                     theShrinkage,
                                     theDepth,  num)}
)
```

```
## [1] 1
## [1] 51
## [1] 101
## [1] 151
## [1] 201
## [1] 251
## [1] 301
## [1] 351
## [1] 401
## [1] 451
## [1] 501
## [1] 551
## [1] 601
## [1] 651
## [1] 701
## [1] 751
## [1] 801
## [1] 851
## [1] 901
## [1] 951
## [1] 1001
## [1] 1051
## [1] 1101
## [1] 1151
## [1] 1201
## [1] 1251
## [1] 1301
## [1] 1351
## [1] 1401
## [1] 1451
## [1] 1501
## [1] 1551
## [1] 1601
## [1] 1651
## [1] 1701
## [1] 1751
## [1] 1801
## [1] 1851
## [1] 1901
## [1] 1951
```

```
data.frame(trees,err=allErrs)  %>%
    ggplot()+
    geom_point(aes(trees,err))+
  geom_line(aes(trees,err))
```



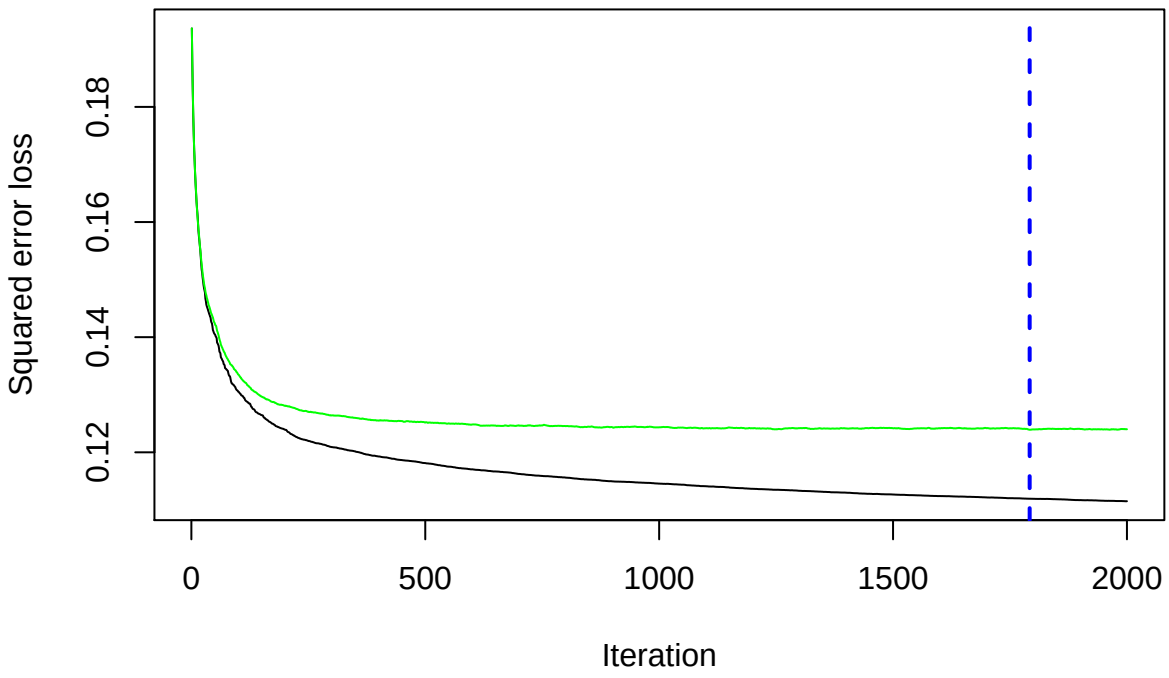
As it turns out, there is a build-in cross-validation for the number of trees (given a specific shrinkage and interation depth). It works like this.

```
mod.gbm.cv <- gbm(z ~ x + y,
                  data=data2D.df,
                  distribution="gaussian", ## for regression
                  n.trees=numTrees,
                  shrinkage=theShrinkage,
                  interaction.depth = theDepth,
                  cv.folds = 5,
                  n.minobsinnode=10,
                  n.cores = 4)  ## <-- use the cores on  your computer
```

Much faster!

Here's what it shows.

```
gbm.best <- gbm.perf(mod.gbm.cv,method="cv")
```



We can extract the optimal number of trees.

```
numTreesOpt <- gbm.best
```

Cross validationg to select Boosting control parameters

In boosting, in general, and GBM, in particular, there are three distinct control parameters.

- The interaction depth (sometimes, the number of terminal leaves)
- The shrinkage factor (also called the learning rate)
- The number of trees

As you can imagine, we use cross-validation to estimate the optimal value for each of these parameter. There are packages in R which will simplify this (e.g. caret). Feel free to explore these. I

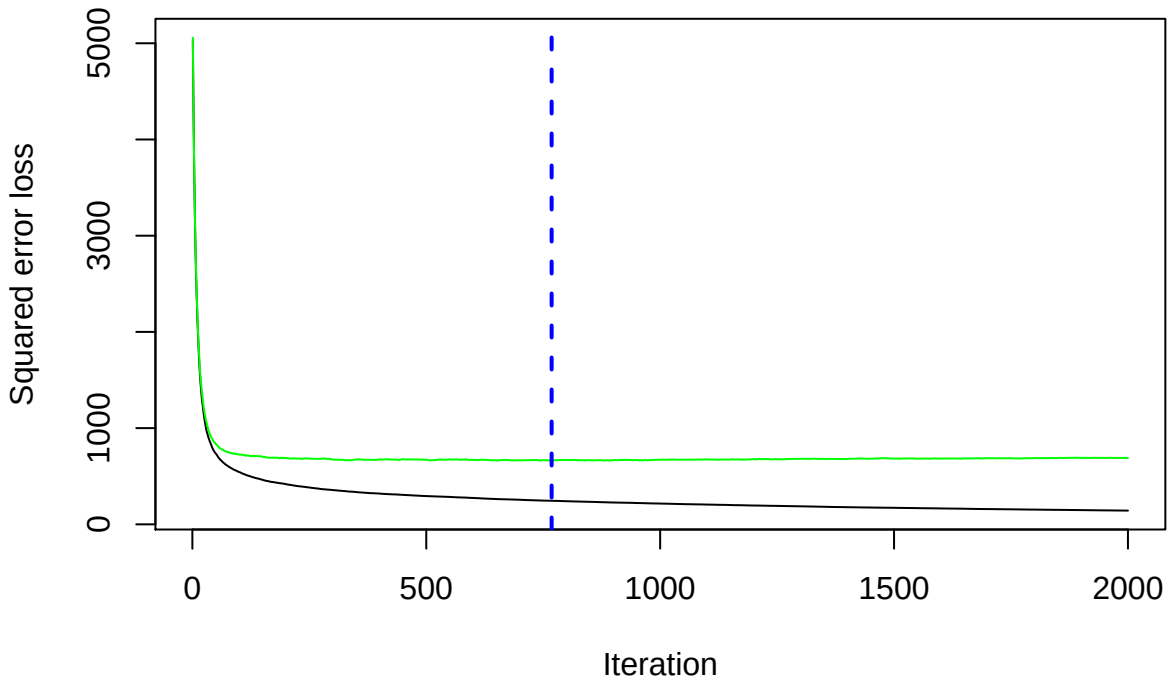
Importance/Influence Plots

Like Bagginng and Random Forests, boosting with trees naturally keeps track of the impact of the different predictors. Let's go back to our housing data and see how this works.

```
dataDir <- "/Users/richeym/Dropbox/COURSES/ADM/ADM_S20/CODE/Chap06"
dataFile <- "HousePrices.csv"
prices.df <- read.csv(file.path(dataDir,dataFile))
```

Now build the model.

```
numPreds <- ncol(prices.df)-1
numTree <- 100
theShrinkage <- 0.1
theDepth <- 2
mod.gbm.cv <- gbm(SalePrice ~ .,
                  data=prices.df,
                  distribution="gaussian", ## for regression
                  n.trees=numTrees,
                  shrinkage=theShrinkage,
                  interaction.depth = theDepth,
                  cv.folds = 5,
                  n.minobsinnode=10,
                  n.cores = 4)
numTreesOpt  <- gbm.perf(mod.gbm.cv,method="cv")
```

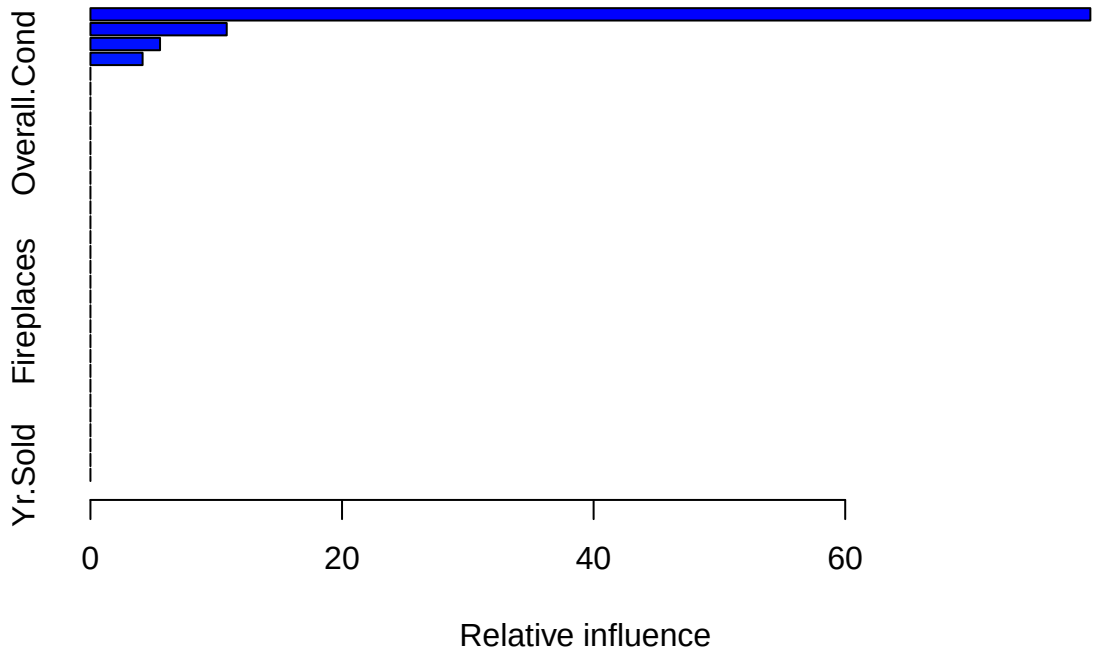


Rebuid with the optimal number of trees.

```
mod.gbm <- gbm(SalePrice ~ .,
                  data=prices.df,
                  distribution="gaussian", ## for regression
                  n.trees=numTreesOpt,
                  shrinkage=theShrinkage,
                  interaction.depth = theDepth
                  )
```

Here's where we can see how the different variables affected the model. The values of rel.inf are total percentage of decrease in the error (MSE).

```
summary(mod.gbm)
```

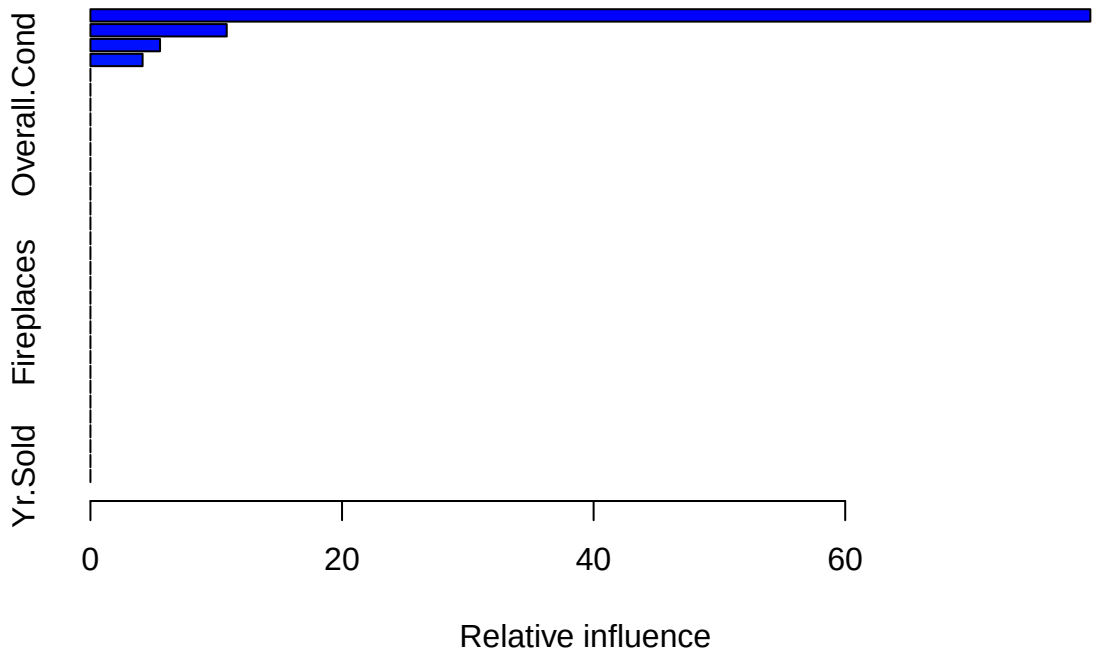



```
##                                 var   rel.inf
## Overall.Qual       Overall.Qual 79.487181
## Garage.Cars         Garage.Cars 10.833487
## Full.Bath             Full.Bath  5.533288
## Year.Built           Year.Built  4.146044
## Lot.Frontage       Lot.Frontage  0.000000
## Lot.Area               Lot.Area  0.000000
## Overall.Cond       Overall.Cond  0.000000
## Year.Remod.Add   Year.Remod.Add  0.000000
## Mas.Vnr.Area       Mas.Vnr.Area  0.000000
## Bsmt.Unf.SF         Bsmt.Unf.SF  0.000000
## Total.Bsmt.SF     Total.Bsmt.SF  0.000000
## X1st.Flr.SF         X1st.Flr.SF  0.000000
## X2nd.Flr.SF         X2nd.Flr.SF  0.000000
## Low.Qual.Fin.SF Low.Qual.Fin.SF  0.000000
## Bsmt.Full.Bath   Bsmt.Full.Bath  0.000000
## Bsmt.Half.Bath   Bsmt.Half.Bath  0.000000
## Half.Bath             Half.Bath  0.000000
## Bedroom.AbvGr     Bedroom.AbvGr  0.000000
## Kitchen.AbvGr     Kitchen.AbvGr  0.000000
```

```
## TotRms.AbvGrd     TotRms.AbvGrd  0.000000
## Fireplaces           Fireplaces  0.000000
## Garage.Yr.Blt     Garage.Yr.Blt  0.000000
## Garage.Area         Garage.Area  0.000000
## Wood.Deck.SF       Wood.Deck.SF  0.000000
## Open.Porch.SF     Open.Porch.SF  0.000000
## Enclosed.Porch   Enclosed.Porch  0.000000
## X3Ssn.Porch         X3Ssn.Porch  0.000000
## Screen.Porch       Screen.Porch  0.000000
## Pool.Area             Pool.Area  0.000000
## Misc.Val               Misc.Val  0.000000
## Mo.Sold                 Mo.Sold  0.000000
## Yr.Sold                 Yr.Sold  0.000000
```

If you want to work with the importance variables directly, you can extract them.

```
influence.sum <- summary(mod.gbm)
```




```
influence.df <- data.frame(var=influence.sum[,1],
                           influence=influence.sum[,2])
##Checking...should be 100%
with(influence.df,sum(influence))
```

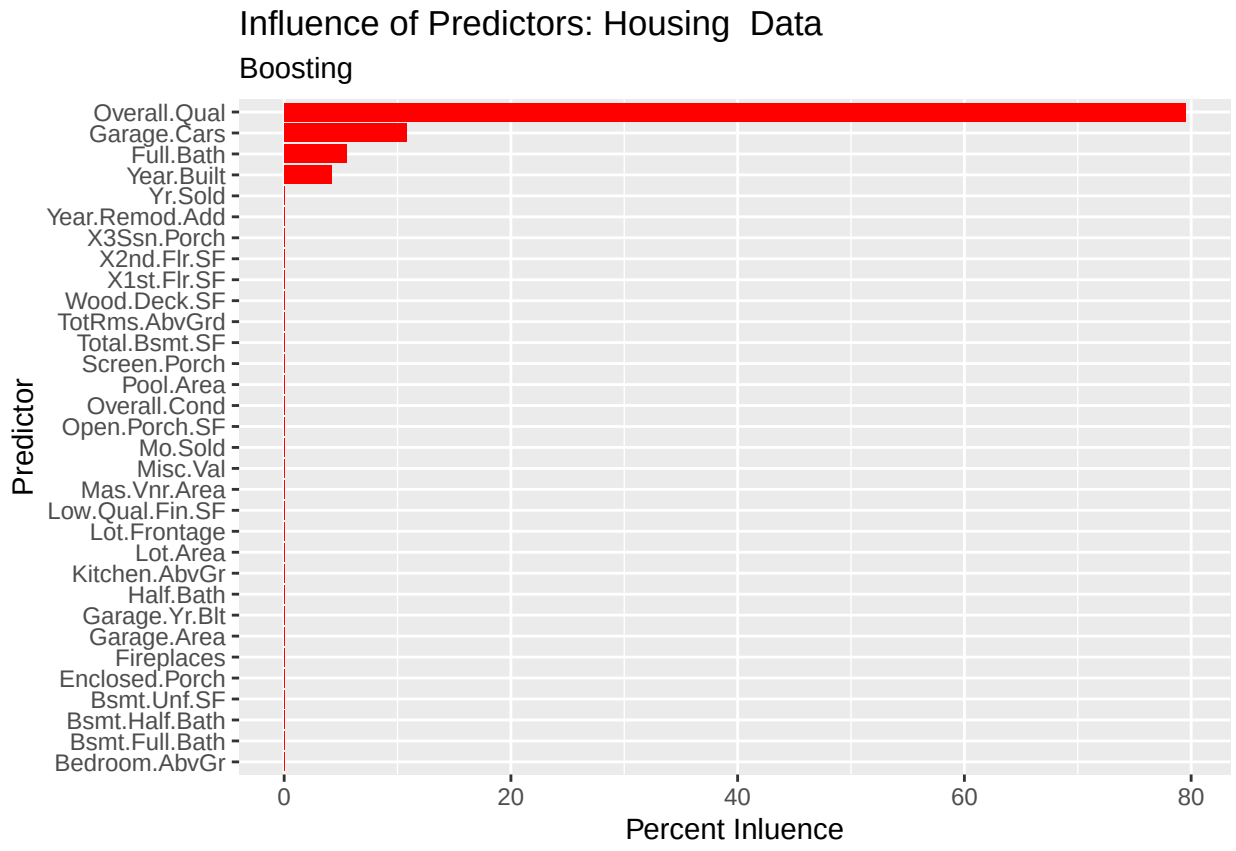
```
## [1] 100
```

Now you have a data frame of the importance. As before, you can play around with this and build a significantly nicer importance plot with ggplot.

Keep the order of the variables (descending order). Here's how to do this using fct_order

```
influence.df <- influence.df %>%
   mutate(var=fct_reorder(var,desc(-influence)))
```

```
influence.df %>%
    ggplot()+
    geom_bar(aes(var,influence),
             stat="identity",
             fill="red")+
    coord_flip()+
    labs(title="Influence of Predictors: Housing  Data",
         subtitle="Boosting",
         x="Predictor",
         y="Percent Inluence")
```



Looks as if Overall.Qual is, by far, the most influential predictor (seems reasonable). This is followed by two less obvious choices: Garage.Cars and Total.Bsmt.SF. Overall, we can see that anything with SF is a big influencer of SalePrice.

Optimizing the Control Parameters via Cross-validation

There are three significant control parameters in play here.

- Number of Trees
- Shrinkage
- Interaction Depth

A complete fit with gbm requires paying attention to all of these. Fortunately, for the most part, the performance is not too sensitive to fine details of these values. By that we I mean if numTree=200 is good, so is numTreess=300. Similarly, a shrinkage factor of 0.1 and 0.2 are probably indistinguishable in practice. Hence you can cross-validate over a broad grid of values.

Let's keep the opitmial number of trees from above and use this to explore the shrinkage and depth values.

```
shrink.vals <- 10^c(-2.0, -1.0, 0.0)
depth.vals <- c(1,2,4,6)
cv.grid <- expand.grid(shrink=shrink.vals,
                       depth=depth.vals)
```

We can use a modest redefinition (SalePrice is the response) of the cvGBM functionfrom earlier.

```
cvGBM <- function(data.df,
                  theShrinkage,
                  theDepth,
                  numTreesPred,
                  numFolds=5){
  N <- nrow(prices.df)
  folds <- sample(1:numFolds,N,rep=T)
  errs <- numeric(numFolds)
  fold <- 1
  for(fold in 1:numFolds){
    train.df.cv <-   prices.df[folds != fold,]
    test.df.cv <- prices.df[folds == fold,]
    mod.gbm <- gbm(SalePrice ~ . ,
                   data=train.df.cv,
                   distribution="gaussian",
                   interaction.depth = theDepth,
                   shrinkage=theShrinkage,
                   n.minobsinnode=10,
                   n.trees=numTrees)
    pred.gbm <- predict(mod.gbm,
                        newdata=test.df.cv,
                        n.trees=numTreesPred)
    errs[fold] <- with(test.df.cv,mean((SalePrice-pred.gbm)^2))
  }
  mean(errs)
}
```

Run a couple quick checks.

```
cvGBM(prices.df,theShrinkage, theDepth,numTreesOpt)
```

```
## [1] 513.2569
```

```
cvGBM(prices.df,theShrinkage/10, 2*theDepth,numTreesOpt)
```

```
## [1] 589.2187
```

Note that cvGBM takes a little while to run.

Now we need to apply it each row of the cv.grid array.Give this some time to run!

```
cv.vals <- apply(cv.grid,1,function(row) {
  print(row)
  cvGBM(prices.df,row[1],row[2],numTreesOpt)
  })
```

```
## shrink  depth
##   0.01   1.00
## shrink  depth
##    0.1    1.0
## shrink  depth
##      1      1
## shrink  depth
##   0.01   2.00
## shrink  depth
##    0.1    2.0
## shrink  depth
##      1      2
## shrink  depth
##   0.01   4.00
## shrink  depth
##    0.1    4.0
## shrink  depth
##      1      4
## shrink  depth
##   0.01   6.00
## shrink  depth
##    0.1    6.0
## shrink  depth
##      1      6
```

Now let's pluck of the location of the minimal MSE and get the corresponding values for shrinkage and depth

```
id <- which.min(cv.vals)
cv.vals[id]
```

```
## [1] 537.8434
```

```
cv.grid[id,]
```

```
##   shrink depth
## 5    0.1     2
```

Hence, it appears that a using 768 trees with a shrinkage of 0.1 and depth of 2 will produce a well performing boosted model.

Assignment: ALS Data Set

Consider the ALS dataset (again) from CASI.

https://web.stanford.edu/~hastie/CASI/data.html

Build an optimal Boosted Model (or as close to optimal as you can get) by considering all three control parameters:

- Number of Trees
- Shrinkage
- Interaction Depth

To the best of your ability, perform a systematic search of this three-dimensional parameter space. Justify your conclusion with cross-validated estimates of the MSE.

How does your Boosted Model perform relative to other algorithms. You have already analyzed ALS with other methods. Refer to this work (or compute anew) to justify your conclusions.