Ada Boosting: Introduction

Matt Richey

4/15/2020

Introduction

Ada Boosting (Adaptive Boosting) was one of the first boosting algorithms for classification. The idea behind Ada Boosting is to build small trees that classify weighted data. By weighted, we mean that for some observations, the error is more costly than others. Hence, to minimize total error, more emphasis is placed on getting some observations correct over others.

Assume we have a binary classifcation with N observations. That is, for each i=1,2,..N, there is a a response $y_i = \pm 1$.

Ada boosting adaptively (on the fly) builds weights that put extra emphasise on getting misclassifications correct. The underlying model is a very simple modeling object, often just a tree with one branch, ie a "Stump." A single Stump is a very bad performer so Boosting, like Bagging, depends on using a large collection of models. One big difference between Boosting and Bagging is that in Boosting the models themselves are weighted.

Here's the idea at each stage of the process.

and a weight w_i. The weights are normalized, that is,

$$\sum_{i}^{N} w_i = 1.$$

Initially, all the weigths are equal (i.e. $w_i = 1/N$ for all i). Repeat the following process M times.

At stage m, Build a Stump ($Tree_m$) with the current weights (there is an option in the tree function that allows weights). Let $pred_i$ represent the prediction for the i^{th} observation.

Compute the weighted classification error:

$$Err_m = \sum_{i=1}^{N} w_i I[y_i \neq pred_i]$$

where **indicator** function $I[y_i \neq pred_i]$ is simply 1 when $y_i \neq pred_i$ and 0 otherwise. This means Err is simply counting the number of misclassifications, with weights. Highly weighted observations "cost" more.

Consider the value

$$R_m = \frac{1 - Err_m}{Err_m} = \frac{1}{Err_m} - 1$$

Some things to note:

- If Err_m is small, ie we are doing a good job of classification, then R_m is large.

- If $Err_m = 1/2$, we are essentially flipping a coin when it comes to classification. In this case $R_m = 1$.
- If $Err_m > 1/2$, we have more misclassificaitions than correct classification. We should just swap the classes! In this case, $R_m < 1$.

Now adjust the weights:

- For any observation that was misclassified, adjust the weight by

$$w_i \leftarrow R_m * w_i$$

- Renormalize (make sure they add to one).

After repeat this process M times, we have a sequence of models (trees)

$$Tree_1, Tree_2, \ldots, Tree_M$$

Given an value x_0 for which we want a prediction, we run them through all M trees and form a weighted sum. Comput

$$Pred(x_0) = \sum_{m=1}^{M} \alpha_m Tree_m(x_0)$$

where $\alpha_m = \log(R_m) = \log(\frac{1-Err_m}{Err_m}$. Note that

- If R_m is large (and positive), so is α_m. These are good trees, given them lots of weight.
- If $R_m \approx 1/2$, then $\alpha_m \approx 0$. These trees are given very little consideration.
- If $R_m < 1$, then α_m is negative. We really don't like these trees, in fact, we want to negatively weight them, essentially flipping the classes

Notice that since the class predictions are ± 1, $Pred(x_0)$ is a real number. If $Pred(x_0) > 0$, we classify x_0 as $+1$. Similarly, if $Pred(x_0) < 0$, we classify x_0 as -1.

That's Ada Boosting!

Below, as is our wont, we will implement Ada Boosting "by hand" and compare it to some of our other prediction methods.

Data

Let's use some data we've seen before. It's a simple data set with observations x and y and a binary class variable.

```
dataDir <- "~/Dropbox/COURSES/ADM/ADM_S20/CODE/Chap08"
dataFile <- "Classify2D_Data3.csv"
data.df0 <- read.csv(file.path(dataDir,dataFile)) %>%
  ## don't need this
  select(-row) %>%
  ## Convert to +/-1
  mutate(class=ifelse(class=="A",1,-1))
N <- nrow(data.df0)
```

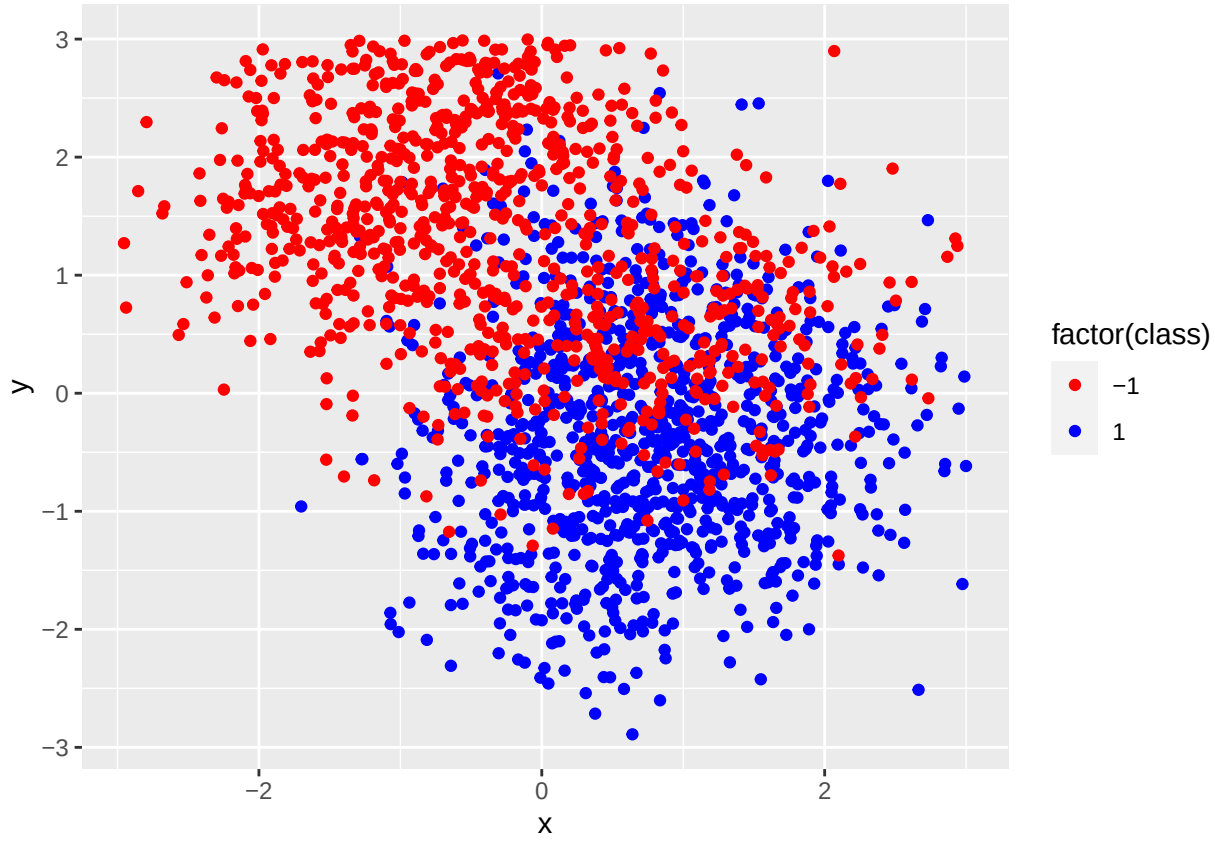
What are we looking at?

```
with(data.df0,table(class))
```

```
## class
##  -1   1
## 917 995
```

Take a quick look at the dispersion of classes

```
cols <- c("red","blue")
data.df0 %>%
  ggplot()+
  geom_point(aes(x,y,color=factor(class)))+
  scale_color_manual(values=cols)
```



Build a smaller data sets to build and verify our model (s).

```
n <- 200
train <- sample(1:N,n,rep=F)
test <- sample(setdiff(1:N,train),n,rep=F)
train.df <- data.df0[train,]
with(train.df,table(class))
```

```
## class
##  -1   1
##  94 106
```

```
test.df <- data.df0[test,]
with(test.df,table(class))
```

```
## class
##  -1   1
## 106  94
```

```
##


# train <- sample(1:N,N/2,rep=F)
# train.df <- data.df0[train,]
# test.df <- data.df0[train,]
# n <- nrow(train.df)
```

Ada Boosting by Hand

Let's implement the algorithm. We'll use 100 Stump trees.

To start, unitialize values

```
##
numStumps <- 100
## The weights are initially 1/n.
```

We need to keep track of the alpha values and all the Stumps.

```
alphaVals <- numeric(numStumps)
Stumps <- list()
```

Here comes the main loop. The tricky bit is how to build a good Stump. To do so, we'll use the R rpart again. It is a little more reliable when it comes to building stumps (or stump-like trees).

```
library(rpart)
##
data.df <- train.df
n <- nrow(data.df)
resp <- data.df$class
##
wgts <- rep(1,n)/n
for(m in 1:numStumps){
  ## Build a  big tree
  stump.tree <- rpart(factor(class) ~ x + y,
                      data=data.df,
                      weights=n*wgts,
                      ##weights=round(n*wgts),
                      control=rpart.control(maxdepth=3))
   pred <- predict(stump.tree,
                   newdata=data.df,
                   type="class")
   ## The weighted error..the (resp!= pred) means we just  count the misclassifications
```

```
  err <- sum(wgts*(resp != pred))
  ## this can happpen, if so stop. It rarely happens
  if(err==0){
    numStumps <- m-1
    alphaVals <- alphaVals[1:numStumps]
    break
  }
 ##also stop of err =1/2. At this point, there is no improvement in weights
  if(err== 1/2){
    numStumps <- m-1
    alphaVals <- alphaVals[1:numStumps]
    break
  }
  ## The all-important alpha value!
   alpha <- log((1-err)/err)
  print(sprintf("Error: %s   Alpha: %s",round(err,4),round(alpha,4)))
  ## Adjust the weigthts
  ## This is just adjusting the weights at the misclassifications
  ## by a factor or (1-err)/err.
  wgts <- wgts*exp(alpha*(resp != pred))
  ## Normalize
  wgts <- wgts/sum(wgts)
  ##Keep the alpha values and the stumps
   ##store the resulting stump
  Stumps[[m]] <- stump.tree
  alphaVals[m] <- alpha
}
```

```
## [1] "Error: 0.17   Alpha: 1.5856"
## [1] "Error: 0.2495   Alpha: 1.1014"
## [1] "Error: 0.3255   Alpha: 0.7287"
## [1] "Error: 0.2909   Alpha: 0.8911"
## [1] "Error: 0.3502   Alpha: 0.618"
## [1] "Error: 0.3789   Alpha: 0.494"
## [1] "Error: 0.2806   Alpha: 0.9417"
## [1] "Error: 0.2904   Alpha: 0.8932"
## [1] "Error: 0.3375   Alpha: 0.6744"
## [1] "Error: 0.3332   Alpha: 0.6939"
## [1] "Error: 0.3719   Alpha: 0.524"
## [1] "Error: 0.3618   Alpha: 0.5676"
## [1] "Error: 0.3595   Alpha: 0.5776"
## [1] "Error: 0.3742   Alpha: 0.5142"
## [1] "Error: 0.356   Alpha: 0.5926"
## [1] "Error: 0.3277   Alpha: 0.7187"
## [1] "Error: 0.3898   Alpha: 0.4482"
## [1] "Error: 0.3331   Alpha: 0.694"
## [1] "Error: 0.4266   Alpha: 0.2957"
## [1] "Error: 0.3367   Alpha: 0.678"
## [1] "Error: 0.3615   Alpha: 0.5686"
## [1] "Error: 0.3767   Alpha: 0.5036"
## [1] "Error: 0.3544   Alpha: 0.5999"
## [1] "Error: 0.3266   Alpha: 0.7238"
## [1] "Error: 0.3762   Alpha: 0.5057"
```

```
## [1] "Error: 0.4186   Alpha: 0.3285"
## [1] "Error: 0.3262   Alpha: 0.7252"
## [1] "Error: 0.3706   Alpha: 0.5296"
## [1] "Error: 0.3926   Alpha: 0.4363"
## [1] "Error: 0.4089   Alpha: 0.3686"
## [1] "Error: 0.3451   Alpha: 0.6408"
## [1] "Error: 0.3644   Alpha: 0.5564"
## [1] "Error: 0.3959   Alpha: 0.4225"
## [1] "Error: 0.4155   Alpha: 0.3413"
## [1] "Error: 0.3698   Alpha: 0.5332"
## [1] "Error: 0.3729   Alpha: 0.5197"
## [1] "Error: 0.3526   Alpha: 0.6076"
## [1] "Error: 0.3612   Alpha: 0.5701"
## [1] "Error: 0.4129   Alpha: 0.3521"
## [1] "Error: 0.361   Alpha: 0.571"
## [1] "Error: 0.3609   Alpha: 0.5716"
## [1] "Error: 0.3627   Alpha: 0.5638"
## [1] "Error: 0.3271   Alpha: 0.7212"
## [1] "Error: 0.3908   Alpha: 0.4439"
## [1] "Error: 0.3748   Alpha: 0.5119"
## [1] "Error: 0.3669   Alpha: 0.5456"
## [1] "Error: 0.4015   Alpha: 0.3992"
## [1] "Error: 0.3476   Alpha: 0.6296"
## [1] "Error: 0.3682   Alpha: 0.5401"
## [1] "Error: 0.3495   Alpha: 0.6212"
## [1] "Error: 0.4094   Alpha: 0.3663"
## [1] "Error: 0.3447   Alpha: 0.6423"
## [1] "Error: 0.3088   Alpha: 0.8059"
## [1] "Error: 0.3375   Alpha: 0.6744"
## [1] "Error: 0.3609   Alpha: 0.5717"
## [1] "Error: 0.405   Alpha: 0.3849"
## [1] "Error: 0.4243   Alpha: 0.305"
## [1] "Error: 0.3873   Alpha: 0.4589"
## [1] "Error: 0.3676   Alpha: 0.5423"
## [1] "Error: 0.4248   Alpha: 0.3029"
## [1] "Error: 0.374   Alpha: 0.5152"
## [1] "Error: 0.4121   Alpha: 0.3553"
## [1] "Error: 0.4619   Alpha: 0.1528"
## [1] "Error: 0.4662   Alpha: 0.1354"
## [1] "Error: 0.4817   Alpha: 0.0732"
## [1] "Error: 0.4823   Alpha: 0.0706"
## [1] "Error: 0.4509   Alpha: 0.1969"
## [1] "Error: 0.4711   Alpha: 0.1155"
## [1] "Error: 0.4721   Alpha: 0.1119"
## [1] "Error: 0.4533   Alpha: 0.1874"
## [1] "Error: 0.4326   Alpha: 0.2712"
## [1] "Error: 0.4467   Alpha: 0.2141"
## [1] "Error: 0.4347   Alpha: 0.2627"
## [1] "Error: 0.4148   Alpha: 0.344"
## [1] "Error: 0.4304   Alpha: 0.2804"
## [1] "Error: 0.3888   Alpha: 0.4524"
## [1] "Error: 0.3594   Alpha: 0.578"
## [1] "Error: 0.3657   Alpha: 0.5507"
## [1] "Error: 0.3661   Alpha: 0.549"
```

```
## [1] "Error: 0.3677   Alpha: 0.5419"
## [1] "Error: 0.3775   Alpha: 0.5004"
## [1] "Error: 0.381   Alpha: 0.4853"
## [1] "Error: 0.3775   Alpha: 0.5003"
## [1] "Error: 0.3686   Alpha: 0.5383"
## [1] "Error: 0.4138   Alpha: 0.3482"
## [1] "Error: 0.426   Alpha: 0.298"
## [1] "Error: 0.4275   Alpha: 0.2919"
## [1] "Error: 0.4567   Alpha: 0.1738"
## [1] "Error: 0.4147   Alpha: 0.3444"
## [1] "Error: 0.4017   Alpha: 0.3984"
## [1] "Error: 0.3644   Alpha: 0.5563"
## [1] "Error: 0.3988   Alpha: 0.4103"
## [1] "Error: 0.4222   Alpha: 0.3139"
## [1] "Error: 0.3881   Alpha: 0.4551"
## [1] "Error: 0.369   Alpha: 0.5364"
## [1] "Error: 0.3681   Alpha: 0.5406"
## [1] "Error: 0.3544   Alpha: 0.5997"
## [1] "Error: 0.3742   Alpha: 0.5144"
## [1] "Error: 0.3415   Alpha: 0.6567"
## [1] "Error: 0.4027   Alpha: 0.3943"
```

OK, now let's make predictions on the test data set. This is similar to what we did with Bagging.

The computation step is to for each of the M models, we have n prediction (each +/-1). For any one observation in the test data, we need to form the sum

$$Pred(x_i) = \sum_{m=1}^{M} \alpha_m Tree_m(x_i)$$

The prediction for x_i, will just be +1 if $Pred(x_i) > 0$ and -1 if $Pred(x_i) < 0$.

Doing this for all the n observations is easily accomplished with matrix multiplication. If $Pred$ is the $n \times M$ matrix of predictions (each row an observation, each column a model) and if α is a $M \times 1$ vector of α values, then we can accomplish all the prediction values in one fell swoop via:

$$Pred * v = P$$

Where P is a $n \times 1$ vector of numbers. The postive ones give us prediction of +1, the negative ones give us predictions of -1.

Let's do it.

```
## the M x n matrix
predMatrix <- matrix(ncol=numStumps,nrow=nrow(test.df))
## Fill  out the columns of the matrix with the predictions from each Stump
for(k in 1:numStumps){
  pred <- predict(Stumps[[k]],newdata=test.df,type="class")
  ## Annoying...convert factors" +/-1" to numbers +/- 1.
  pred <- as.numeric(as.character(pred))
  predMatrix[,k] <- pred
}
```

Compute the predicted values by weighting each tree by the appropriate alpha value. Then decide on the prediction by whether or not the resulting value is positive.

```
predVals <- predMatrix %*% alphaVals
##convert
preds <- ifelse(predVals > 0,1,-1)
```

How did we do? The big reveal....

```
with(test.df,table(class,preds))
```

```
##       preds
## class -1  1
##    -1 78 28
##    1  22 72
```

```
with(test.df,mean((class == 1) != (preds==1)))
```

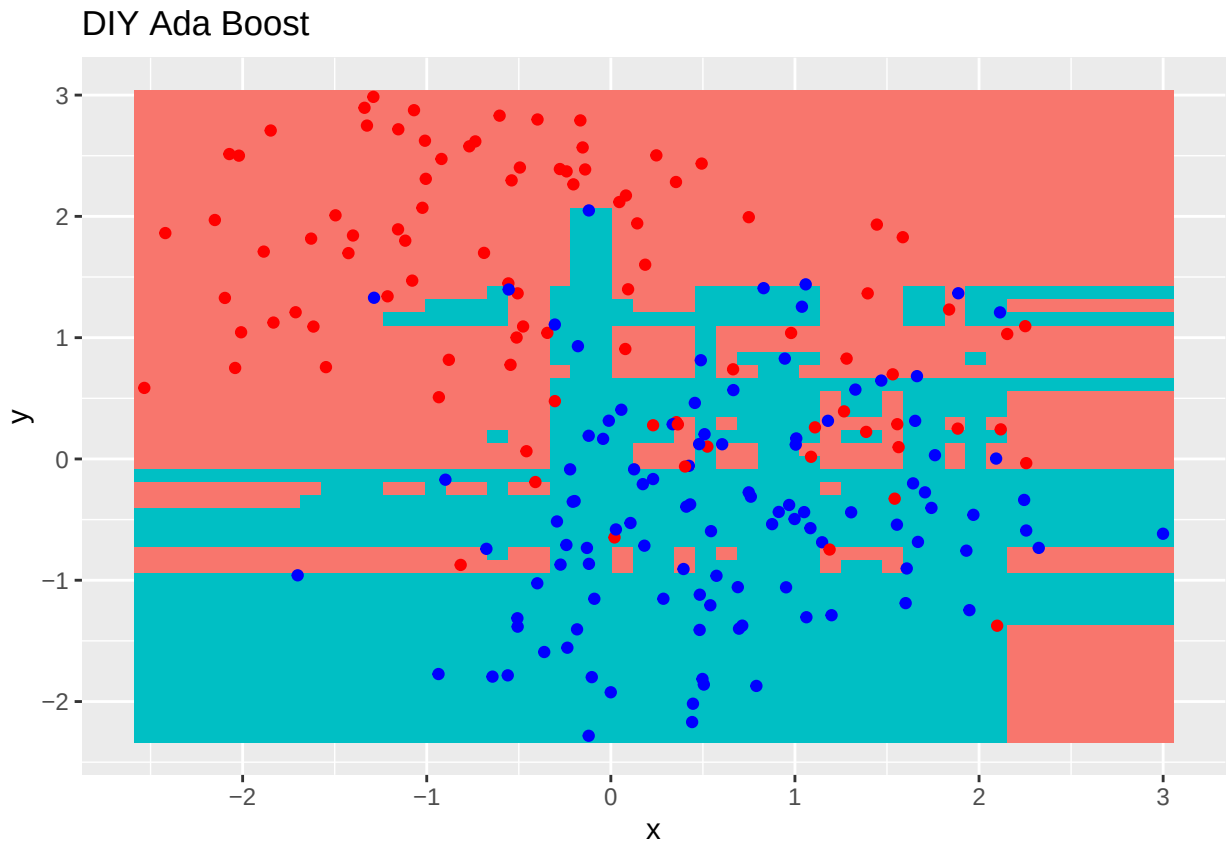
```
## [1] 0.25
```

Hmm.....not bad. Remember that this is simple synthetic dat.

Ada Boost Model on a Gri

For fun, let's visualize our DIY Ada Boost model on a grid.

```
xRng <- with(data.df,range(x))
yRng <- with(data.df,range(y))
gridSize <- 50
grid.df <- data.frame(expand.grid(x=seq(xRng[1],xRng[2],length=gridSize),
                                  y=seq(yRng[1],yRng[2],length=gridSize)))

nGrid <- nrow(grid.df)
gridMatrix <- matrix(ncol=numStumps,nrow=nGrid)
## Fill  out the columns of the matrix with the predictions from each Stump
for(k in 1:numStumps){
  pred <- predict(Stumps[[k]],newdata=grid.df,type="class")
  ## Annoying...convert factors" +/-1" to numbers +/- 1.
  pred <- as.numeric(as.character(pred))
  gridMatrix[,k] <- pred
}

gridVals <- gridMatrix %*% alphaVals
grid.df$pred <- gridVals>0
grid.df %>%
  ggplot()+
  geom_tile(aes(x,y,fill=factor(gridVals > 0)))+
    geom_point(data=data.df,aes(x,y,color=factor(class)))+
   scale_color_manual(values=cols)+
  guides(fill=FALSE,color=FALSE)+
  labs(title="DIY Ada Boost",
       fill="",
       color="")
```



Interesting model!

Ada Boost: The R way

Of course, R has it's own Adaboost built into the gbm function.

One quirk of the gbm adaboost is that the response variable must be 0/1 (not -1/+1). Let's fix up the data to accommodate this quirk.

```
data.df1 <- data.df0 %>%
  mutate(class=ifelse(class > 0,1,0))
with(data.df1,table(class))
```

```
## class
##   0   1
## 917 995
```

```
## Build train/test again
n <- 200
train <- sample(1:N,n,rep=F)
test <- sample(setdiff(1:N,train),n,rep=F)
train.df <- data.df1[train,]
with(train.df,table(class))
```

```
## class
```

```
##   0   1
## 101  99
```

```
test.df <- data.df1[test,]
with(test.df,table(class))
```

```
## class
##   0   1
##  82 118
```

Now we are ready to run the gbm Adaboost. Use distribution="adaboost".

Note: As with gbm for regression, there is a shrinkage factor. The shrinkage can deflate the alpha values at each step. Above, our shrinkage was 1. In general, this is a parameter that can be tuned for better performance. As well, the interaction depth and the number of trees can be tuned.

```
theDepth <- 2
theShrinkage <- 1
numTrees <- 500
mod.gbm <- gbm(class ~ x + y,
               data=train.df,
               distribution="adaboost", ## <---for adaboost
               interaction.depth = theDepth,
               shrinkage=theShrinkage,  ##adjust to reduce alpha
               n.trees=numTrees)
```

The predictions come out most naturally as probabilities.

```
numTreesPred <- 100
prob.gbm <- predict(mod.gbm,
                    newdata=test.df,
                    n.trees=100,type="response")
pred.gbm <- ifelse(prob.gbm > 0.5,1,0)
```

```
with(test.df,table(class,pred.gbm > 0.5))
```

```
##
## class FALSE TRUE
##     0    63   19
##     1    38   80
```

```
with(test.df,mean((class==1) != pred.gbm))
```

```
## [1] 0.285
```

Cross Validation ADA Boost

Of course, we need to cross validate for find optimal control parameters.

Build a cv function for adaboost.

```
cvADA <- function(data.df,theShrinkage,theDepth,numTrees,numFolds=5){
  n <- nrow(data.df)
  folds <- sample(1:numFolds,n,rep=T)
  errs <- numeric(numFolds)
  for(fold in 1:numFolds){
    train.df <- data.df[folds != fold,]
    test.df <- data.df[folds == fold,]
    mod.gbm <- gbm(class ~ x + y,
                   data=train.df,
                   interaction.depth = theDepth,
                   distribution="adaboost",
                   shrinkage=theShrinkage,
                   n.trees=numTrees)
    prob.gbm <- predict(mod.gbm,
                        newdata=test.df,
                        n.trees=numTrees,type="response")
    errs[fold] <- with(test.df,mean((class == 1) != (prob.gbm > 0.5)))
  }
  mean(errs)
}
```

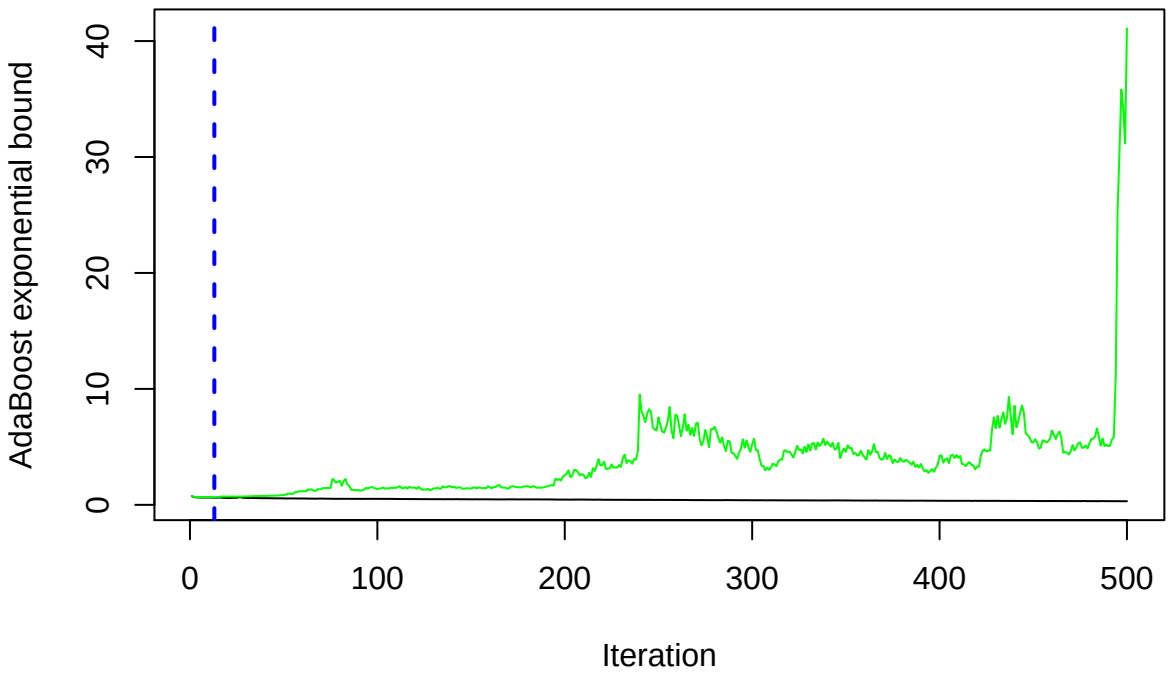
Test it out.

```
theShrinkage <- 1
numTrees <-500
theDept <- 4
(mse.ada <- cvADA(train.df,theShrinkage, theDepth,  numTrees))
```

```
## [1] 0.2597807
```

As we saw before, gbm can cross-validation on the number of trees (given shrinkage and depth values). For this data, using the built-in cv function doesn't reveal much. We'll use it later on some real data.

```
mod.gbm.cv <- gbm(class ~ x + y,
                  data=data.df1,
                  interaction.depth = theDepth,
                  distribution="adaboost",
                  shrinkage=theShrinkage,
                  cv.folds = 10,
                  n.trees=numTrees)
gbm.perf(mod.gbm.cv)
```




```
## [1] 13
```

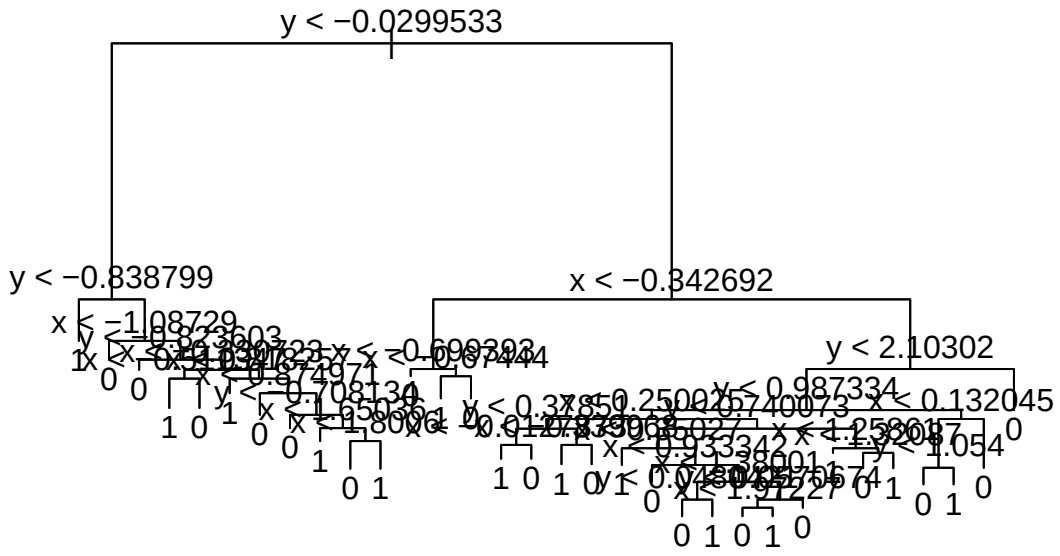
Comparison With Other Models

As a quick check, let's model this data with some other algorithms and see what happens

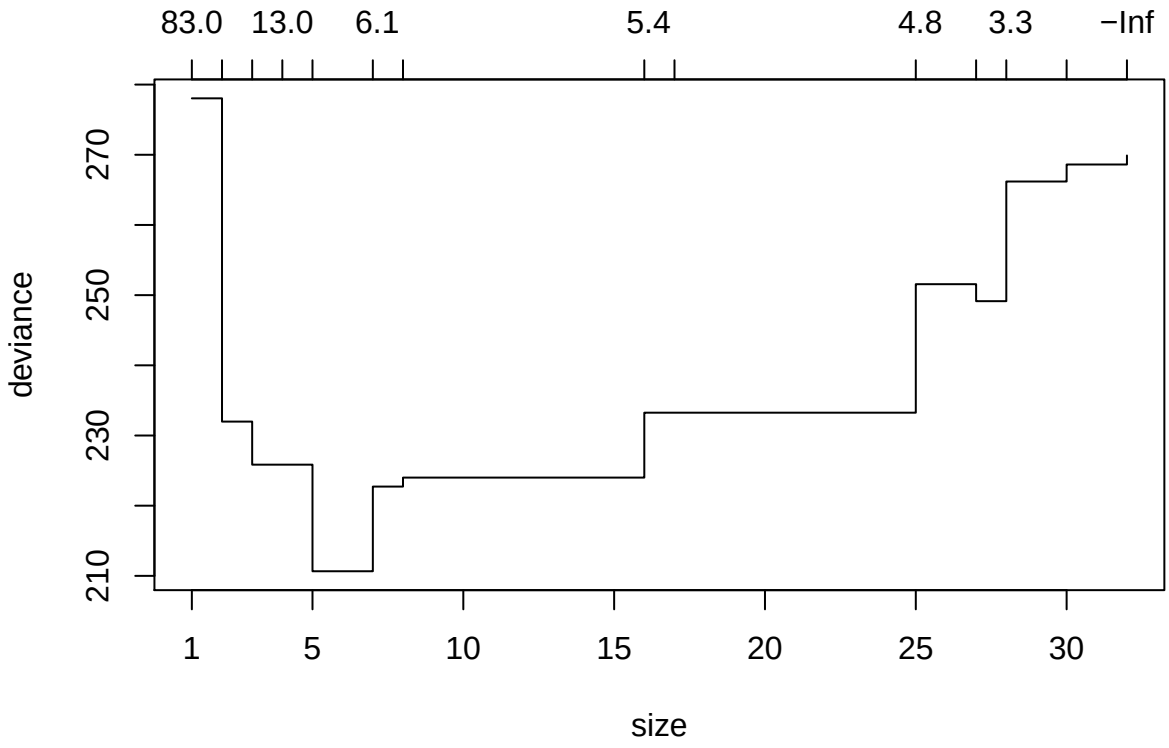
Pruned trees

Here it goes

```
mod.big.tree <- tree(factor(class) ~ x +y,
                     data=train.df,
                     control=tree.control(nrow(data.df),
                                          mindev=.0001,
                                          minsize=2))
{plot(mod.big.tree);text(mod.big.tree)}
```

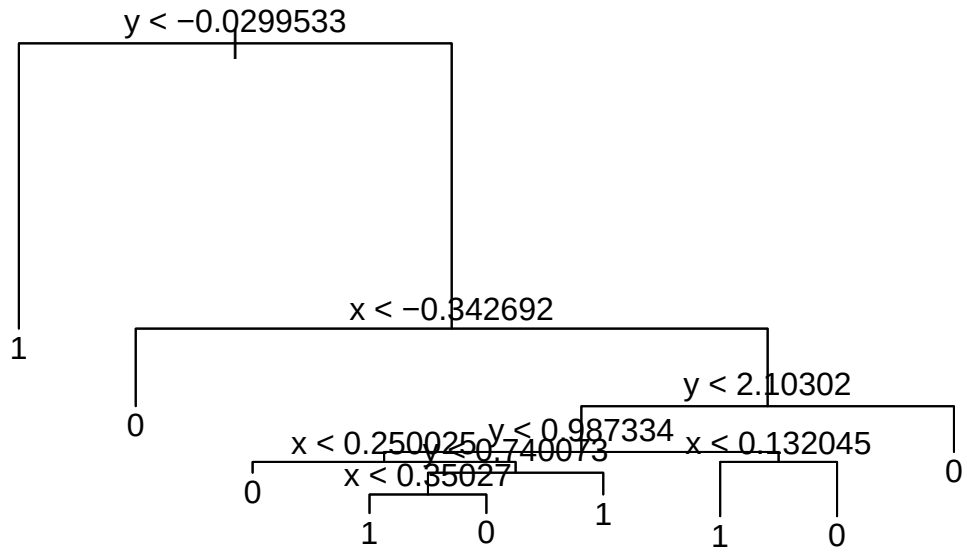



```
mod.cv <- cv.tree(mod.big.tree)
plot(mod.cv)
```



Looks like it best=3-5 will work.

```
mod.prune <- prune.misclass(mod.big.tree,best=5)
{plot(mod.prune);text(mod.prune)}
```



Predict from the pruned tree.

```
preds.prune <- predict(mod.prune,
                       newdata=test.df,
                       type="class")
with(test.df,table(class,preds.prune))
```

```
##       preds.prune
## class  0  1
##     0 69 13
##     1 38 80
```

```
(mse.prune <- with(test.df,mean(class != preds.prune)))
```

```
## [1] 0.255
```

About the same

Logistic Regression

Let's give logistic regression a atry.

```
mod.log <- glm(factor(class) ~ x+y,
               data=train.df,
               family="binomial")
probs <- predict(mod.log,newdata=test.df,family="binomial",type="response")

with(test.df,table(class,probs>0.5))
```

```
##
## class FALSE TRUE
##     0    71   11
##     1    22   96
```

```
(mse.log <- with(test.df,mean((class==1) != (probs>0.5))))
```

```
## [1] 0.165
```

Not bad.

Bagging

Now run the datq a trough Bagging. Since there are only two predictors, Random Forest really isn't an option here.

```
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(factor(class) ~ x+y,
               data=train.df,
               mtry=2)
preds.rf <- predict(mod.rf,newdata=test.df)

with(test.df,table(class,preds.rf))
```

```
##      preds.rf
## class  0  1
##     0 70 12
##     1 30 88
```

```
(mse.rf <- with(test.df,mean(class != preds.rf)))
```

```
## [1] 0.21
```

In summary

```
c(mse.ada,mse.log,mse.prune,mse.rf)
```

```
## [1] 0.2597807 0.1650000 0.2550000 0.2100000
```

Ok, so Boosting is at least competitive with other methods. In this case, the data are too simple to be discriminate between methods. Let's bring in R's Ada boost implementation.

Wisconsin Breast Cancer

The Wisconsin Breast Cancer dataset is available at the UCI Machine Learning Repository.

http://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+%28Original%29

1. Sample code number: id number
2. Clump Thickness: 1 - 10
3. Uniformity of Cell Size: 1 - 10
4. Uniformity of Cell Shape: 1 - 10
5. Marginal Adhesion: 1 - 10
6. Single Epithelial Cell Size: 1 - 10
7. Bare Nuclei: 1 - 10
8. Bland Chromatin: 1 - 10
9. Normal Nucleoli: 1 - 10
10. Mitoses: 1 - 10
11. Class: (2 for benign, 4 for malignant)

Load in the data and construct train/test datasets

```
winBC.df <- read.csv("breastCancer.csv")
names(winBC.df) <- c("ID","ClumpThickness","UnifCellSize","UnifCellShape","MargAdh",
                     "EpthCellSize","BareNuc","BlandChrom","NormChrom","Mitoses","Class")
winBC.df <- winBC.df %>%
  mutate(Class=ifelse(Class==2,0,1))
```

```
with(winBC.df,table(Class))
```

```
## Class
##   0   1
## 457 241
```

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

train.df <- winBC.df[train,]
test.df <- winBC.df[-train,]
with(train.df,table(Class))
```

```
## Class
##   0   1
## 227 122
```

```
with(test.df,table(Class))
```

```
## Class
##   0   1
## 230 119
```

Start with a simple ADA Boost model.

```
numTrees <- 200
theShrinkage <- 1
theDepth <- 2
mod.gbm <- gbm(Class ~ .,
               data=train.df,
               distribution="adaboost",
               shrinkage=theShrinkage,
               n.trees=numTrees,
               interaction.depth = theDepth)
```

Make predictions on the test data.

```
prob.gbm <- predict(mod.gbm,
                    newdata=test.df,
                    n.trees=100,type="response")
pred.gbm <- ifelse(prob.gbm > 0.5,1,0)
with(test.df,table(Class,pred.gbm))
```

```
##      pred.gbm
## Class   0   1
##     0 226   4
##     1  17 102
```

```
(err.gbm <- with(test.df,mean(Class != pred.gbm)))
```

```
## [1] 0.06017192
```

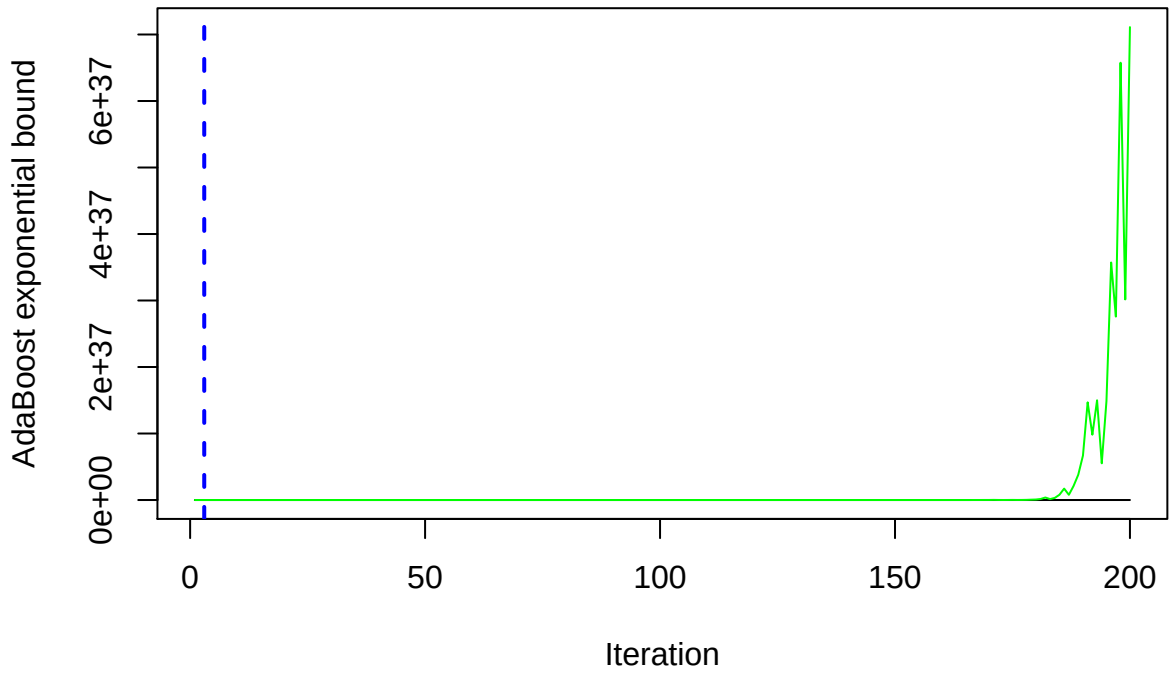
A very good error rate. Can we do better?

Use the built-in gbm cross-validation to look for optimal number of trees.

In this case, we can use the full data set since we cross-validating

```
mod.gbm.cv <- gbm(Class ~ .,
               data=train.df,
               distribution="adaboost",
               shrinkage=theShrinkage,
               n.trees=numTrees,
               interaction.depth = theDepth,
               cv.folds = 10)
```

```
gbm.perf(mod.gbm.cv)
```




```
## [1] 3
```

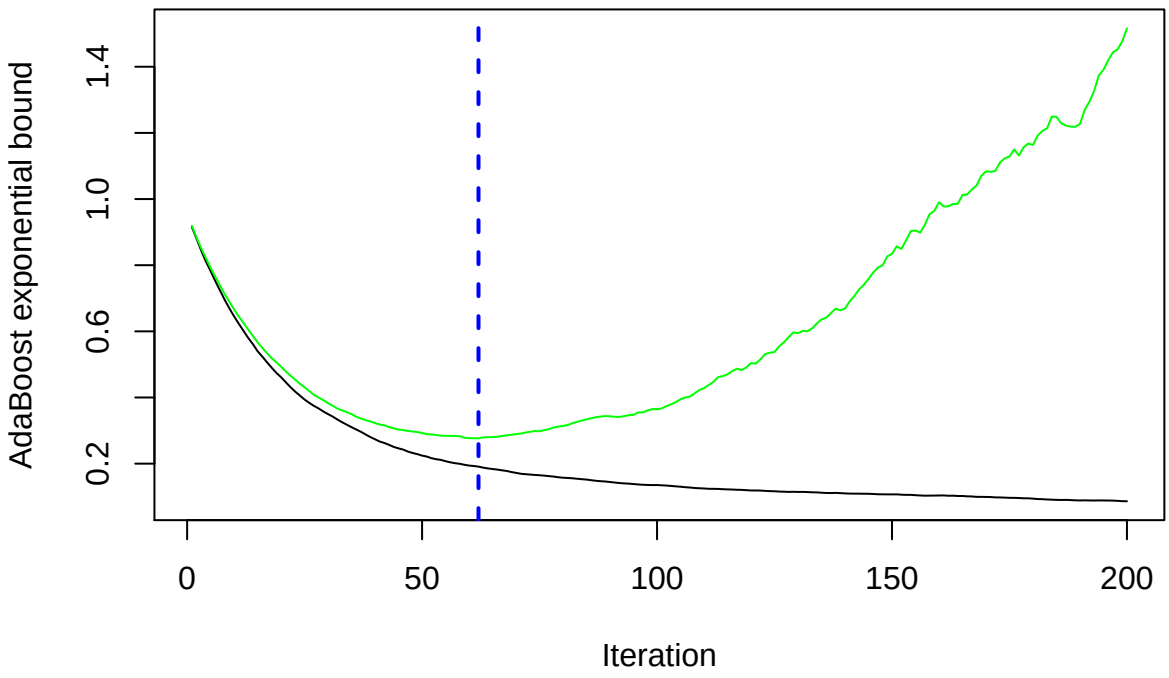
Hmmmm... a plot like this is usually an indication that we don't have good parameters.

Decrease the shrinkage.

```
theShrinkage <- 0.05
mod.gbm.cv <- gbm(Class ~ .,
                  data=train.df,
                  distribution="adaboost",
                  shrinkage=theShrinkage,
                  n.trees=numTrees,
                  interaction.depth = theDepth,
                  cv.folds = 10)
```

And the plot

```
(numTreesOpt <- gbm.perf(mod.gbm.cv))
```




```
## [1] 62
```

How well does this work?

```
mod.gbm <- gbm(Class ~ .,
               data=train.df,
               distribution="adaboost",
               shrinkage=theShrinkage,
               n.trees=numTreesOpt,
               interaction.depth = theDepth)
prob.gbm <- predict(mod.gbm,
                    newdata=test.df,
                    n.trees=numTreesOpt,
                    type="response")
pred.gbm <- ifelse(prob.gbm > 0.5,1,0)
(err.gbm <- with(test.df,mean(Class != pred.gbm)))
```

```
## [1] 0.05157593
```

A distinct improvement in the error rate.

Let's stick with this number of trees. There still remains the question of the best choice of shrinkage and interaction depth. There are R tools that can help with this (a good example is the caret package). Sometimes it's just easier to do it yourself.

Let's build a grid of shrinkage and interaction depth values. We don't have to cut it too fine. We'll just use a good set of values that span a decent range in each dimension.

```
shrinkVals <- c(0.005,0.001,0.05,0.1,0.5)
depthVals <- c(2,3,4,5)
gridVals <- expand.grid(shrinkVals,depthVals)
```

Build a simple function which encapulates the error calculation above.

```
calcErr <- function(shrink,depth){
  mod.gbm <- gbm(Class ~ .,
                 data=train.df,
                 distribution="adaboost",
                 shrinkage=shrink,
                 n.trees=2*numTreesOpt,
                 interaction.depth = depth)
  prob.gbm <- predict(mod.gbm,
                      newdata=test.df,
                      n.trees=numTreesOpt,
                      type="response")
  pred.gbm <- ifelse(prob.gbm > 0.5,1,0)
  with(test.df,mean(Class != pred.gbm))
}
##Test it out
calcErr(theShrinkage,theDepth)
```

```
## [1] 0.04297994
```

Ok, seems to work. Now we can apply it to each row of the grid.

```
gridErrs <- apply(gridVals,1,function(row) calcErr(row[1],row[2]))
```

Pull of the location and values associated with the minimum error.

```
id <- which.min(gridErrs)
(bestVals <- gridVals[id,])
```

```
##    Var1 Var2
## 19  0.1    5
```

```
shrinkOpt <- bestVals[1]
depthOpt <- bestVals[2]
```

This gives us the best gbm ADA Boost Model.

```
mod.gbm.opt <-  gbm(Class ~ .,
                 data=train.df,
                 distribution="adaboost",
                 shrinkage=shrinkOpt,
                 n.trees=numTreesOpt,
                 interaction.depth = depthOpt)
prob.gbm <- predict(mod.gbm.opt,
                      newdata=test.df,
```

```
                      n.trees=numTreesOpt,
                      type="response")
pred.gbm <- ifelse(prob.gbm > 0.5,1,0)
(err.ada <- with(test.df,mean(Class != pred.gbm)))
```

```
## [1] 0.03724928
```

Comparison with other models

Logistic Regression

```
mod.log <- glm(Class ~ .,
               data=train.df,
               family="binomial")
##
```

The error message is troublesome.

```
prob.log <- predict(mod.log,
               data=test.df,
               family="binomial",
               type="response")
pred.log <- ifelse(prob.log > 0.5, 1,0)
with(test.df,table(Class,pred.log))
```

```
##      pred.log
## Class   0   1
##     0 147  83
##     1  79  40
```

```
(err.log <- with(test.df,mean(Class != pred.log)))
```

```
## [1] 0.4641834
```

Not too impressive.

Ridge/Lasso

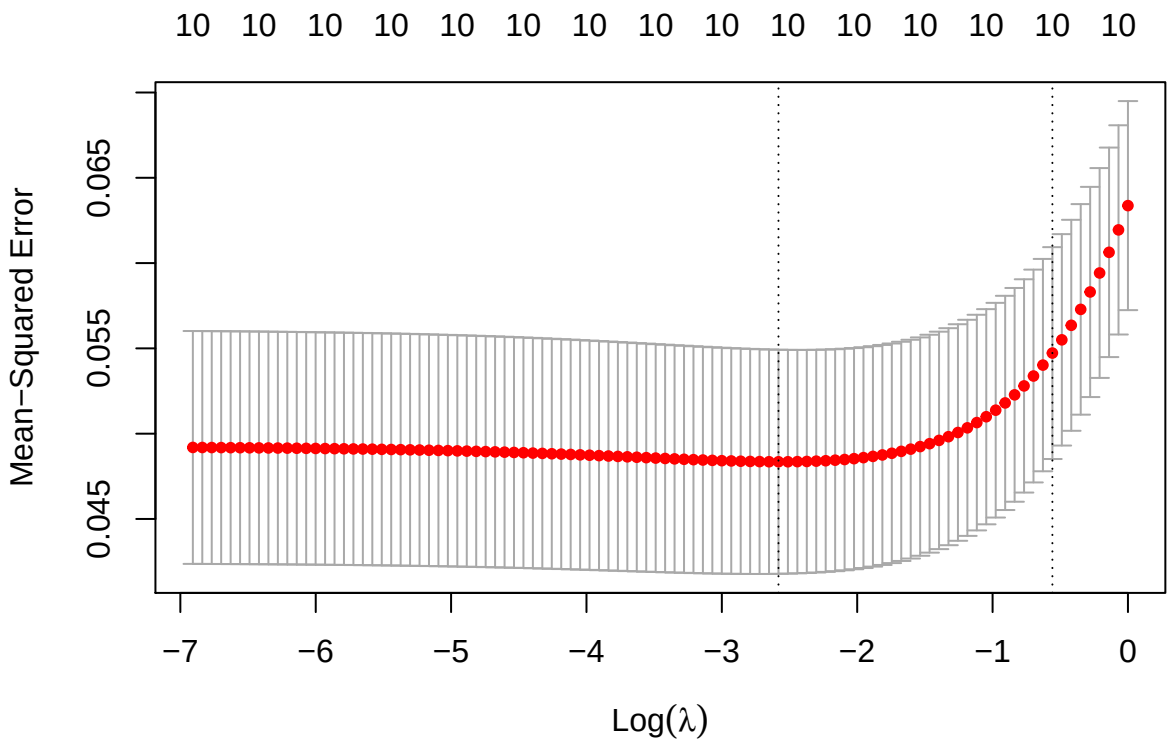
```
library(glmnet)
```

```
## Loading required package: Matrix

##
## Attaching package: 'Matrix'

## The following objects are masked from 'package:tidyr':
##
##     expand, pack, unpack
```

```
## Loaded glmnet 3.0-2
```

```
train.x <- data.matrix(train.df[,-11])
train.y <- data.matrix(train.df[,11])
test.x <- data.matrix(test.df[,-11])
test.y <- data.matrix(test.df[,11])
lambda.grid <- 10^seq(-3,0,length=100)
```

```
mod.ridge.cv <- cv.glmnet(train.x,
                          train.y,
                          lambda=lambda.grid,
                          alpha=0)
plot(mod.ridge.cv)
```



Build the optimal ridge model.

```
lambda.opt <- mod.ridge.cv$lambda.1se
mod.ridge <-  glmnet(train.x,
                     train.y,
                     lambda=lambda.opt,
                     family="binomial",
                     alpha=0)
prob.ridge <- predict(mod.ridge,newx=test.x,type="response")
pred.ridge <- ifelse(prob.ridge > 0.5, 1,0)
table(test.y,pred.ridge)
```

```
##       pred.ridge
## test.y   0   1
##      0 228   2
##      1  24  95
```

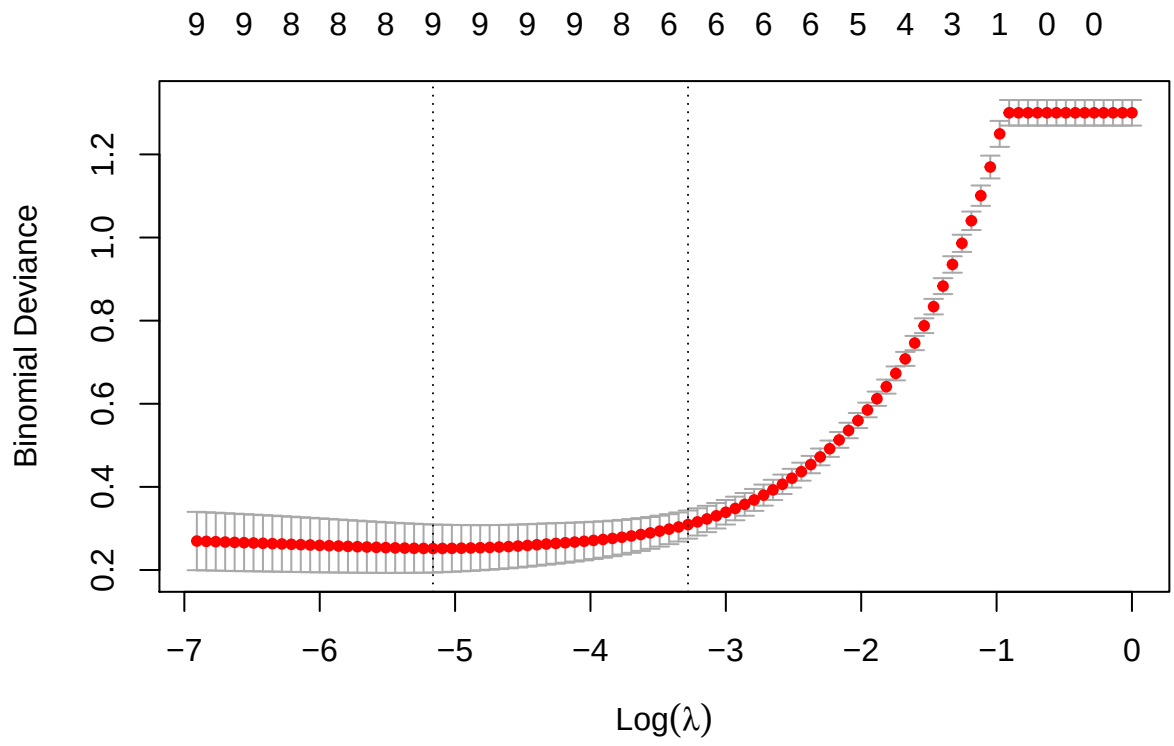
```
(err.ridge <- with(test.df,mean(Class != pred.ridge)))
```

```
## [1] 0.07449857
```

Pretty good

Repeat with Lasso

```
mod.lasso.cv <- cv.glmnet(train.x,
                          train.y,
                          lambda=lambda.grid,
                          family="binomial",
                          alpha=1)
plot(mod.lasso.cv)
```



Build the optimal Lasso model.

```
lambda.opt <- mod.lasso.cv$lambda.1se
mod.lasso <-  glmnet(train.x,
                     train.y,
```

```
                    lambda=lambda.opt,
                    alpha=1)
prob.lasso <- predict(mod.lasso,newx=test.x,type="response")
pred.lasso <- ifelse(prob.lasso > 0.5, 1,0)
(err.lasso <- with(test.df,mean(Class != pred.lasso)))
```

```
## [1] 0.05730659
```

Random Forest

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

```
## [1] 349  11
```

Since there are 10 predictors, use mtry=3 o4.

```
mod.rf <- randomForest(factor(Class) ~ .,
               data=train.df,
               mtry=4,
               ntree=1000,
               importance=TRUE)
```

```
pred.rf <- predict(mod.rf,newdata=test.df)
with(test.df,table(Class,pred.rf))
```

```
##      pred.rf
## Class   0   1
##     0 225   5
##     1  10 109
```

```
(err.rf <- with(test.df,mean(Class != pred.rf)))
```

```
## [1] 0.04297994
```

```
c(err.log,err.ridge,err.lasso,err.rf,err.ada)
```

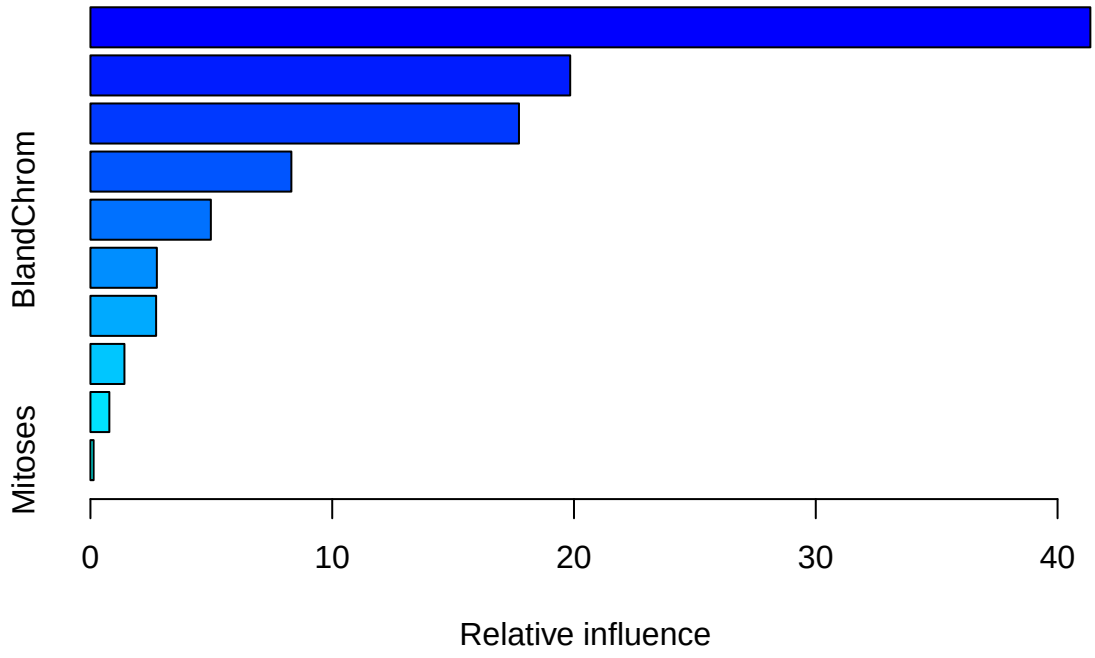
```
## [1] 0.46418338 0.07449857 0.05730659 0.04297994 0.03724928
```

It looks as if Random Forest and ADA Boost are the best methods here, with a small edge going to Random Forest.

Importance Variables

Both GBM and Random Forest have a way of characterizing the importance of the predictors variables. What do they say here?

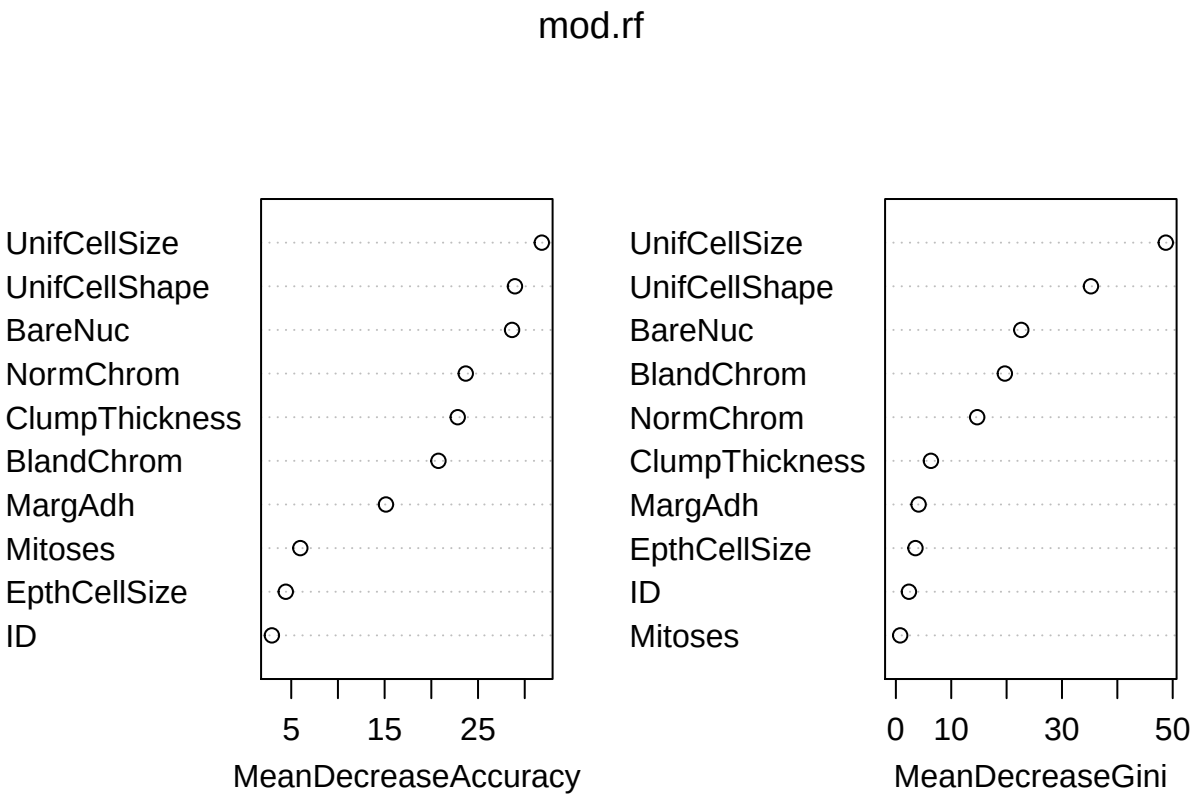
```
mod.gbm.opt <-  gbm(Class ~ .,
                 data=train.df,
                 distribution="adaboost",
                 shrinkage=shrinkOpt,
                 n.trees=numTreesOpt,
                 interaction.depth = depthOpt)

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




```
sum.gbm
```

```
##                              var    rel.inf
## UnifCellSize       UnifCellSize 41.3570716
## UnifCellShape     UnifCellShape 19.8432691
## BareNuc                 BareNuc 17.7247202
## NormChrom             NormChrom  8.3103066
## BlandChrom           BlandChrom  4.9812332
## ClumpThickness ClumpThickness  2.7474883
## MargAdh                 MargAdh  2.7161987
## ID                           ID  1.4059384
## EpthCellSize       EpthCellSize  0.7822523
## Mitoses                 Mitoses  0.1315217
```

```
varImpPlot(mod.rf)
```



Both models identify UnifCellSize as the most important variable. There is some disagreement over what comes next but both agree on the top five predictors. This is interesting information when it comes to understanding the roles played the predictors in this situation. # Assignment Analyze the SPAM data set along these same lines. Make sure you find what you think are the best control parameters (depth, number of trees, shrinkage) for ADA Boost. Compare the results of ADA Boost with a few other methods.