hw 5

Rob McCulloch

November 10, 2019

Trees

Let's use the tabloid data.

```
library(MASS)
data(Boston)
trainDf = Boston[,c(8,13,14)]
head(trainDf)
```

```
##      dis lstat medv
## 1 4.0900  4.98 24.0
## 2 4.9671  9.14 21.6
## 3 4.9671  4.03 34.7
## 4 6.0622  2.94 33.4
## 5 6.0622  5.33 36.2
## 6 6.0622  5.21 28.7
```

We will use the `R` package `rpart`.
In the notes, I also use the package `tree`.
`tree` is simpler and easier to use, but if you really want to do trees, you need to use `rpart` so let's try that.

```
library(rpart)
```

First we want to grow a big tree greedily.

We use a "control" data structure to grow the big tree.

For example,

minbucket:

```
the minimum number of observations in any terminal <leaf> node.
If only one of minbucket or minsplit is specified, the code either sets minsplit to minbucket*3
or minbucket to minsplit/3, as appropriate.
```

So, if we choose minbucket and cp small enough, we will get a big tree.

```
set.seed(14)
n = nrow(trainDf)
minob = 3
cntrl = rpart.control(minbucket=minob,minsplit=2*minob,cp=.00005)
bigt = rpart(medv~., data=trainDf, control=cntrl, method="anova") #use method="class" for categorical o
cat("size of big tree: ",length(unique(bigt$where)))
```

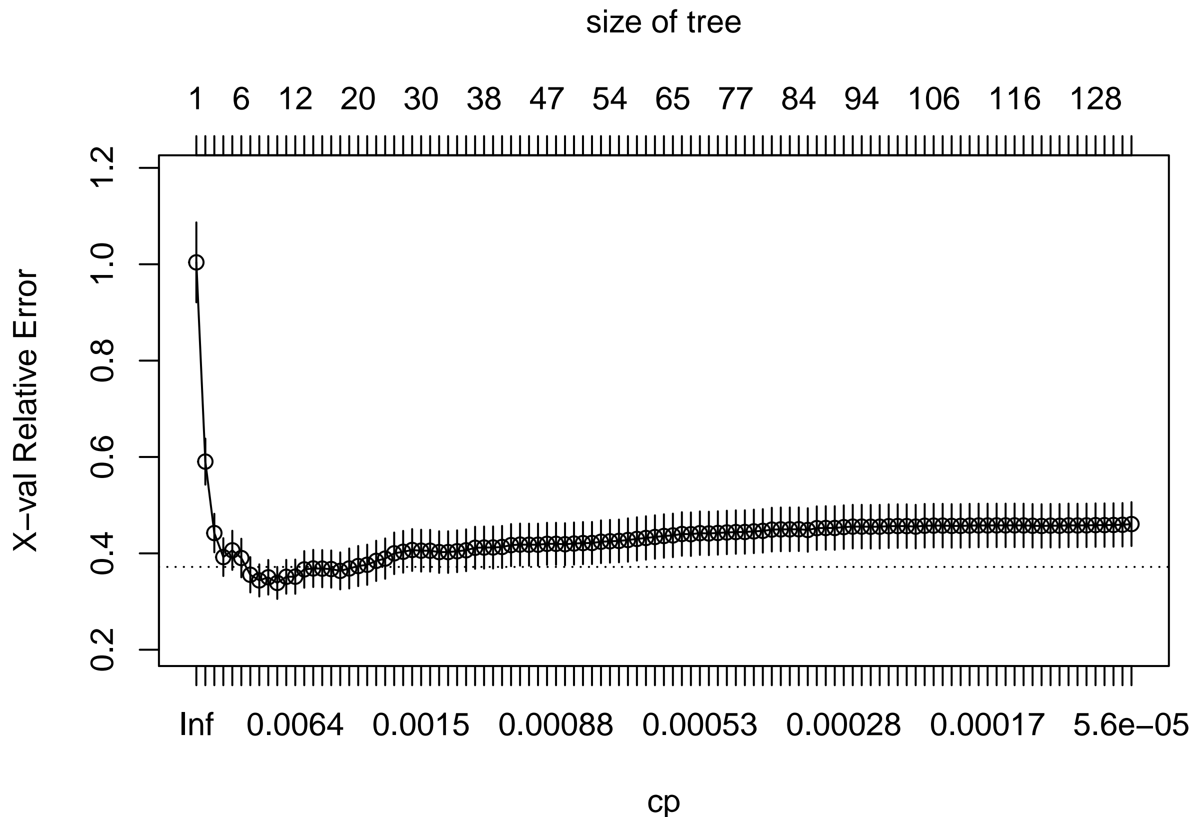
```
## size of big tree:  132
```

```
# where is an index telling which bottom node each observation is in
```

If you want a bigger tree, make cp and/or minob smaller.

Now we can plot the cross validation to big the cost complexity which indirectly chooses the size of the tree.

```r
plotcp(bigt)
```



We can see that a tree of about size 10 should work.
So, clearly, the "big tree" was big enough.

To get the min size we can look at the `cptable`.

```r
head(bigt$cptable)
```

```
##           CP nsplit rel error    xerror       xstd
## 1 0.44236500      0 1.0000000 1.0038934 0.08295769
## 2 0.15283400      1 0.5576350 0.5903950 0.04781398
## 3 0.06275014      2 0.4048010 0.4421597 0.04000272
## 4 0.02833720      3 0.3420509 0.3922944 0.03929034
## 5 0.02722395      4 0.3137137 0.4063256 0.04061861
## 6 0.01925703      5 0.2864897 0.3902939 0.04003479
```

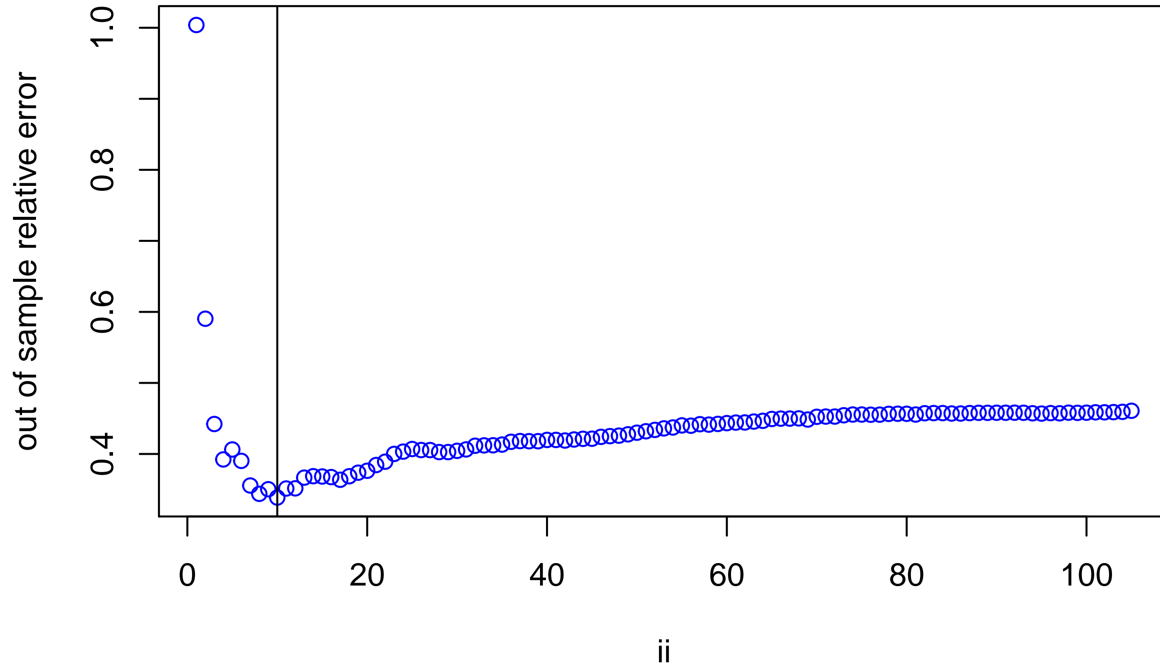
```
# note: in cptable:
# CP is the complexity parameter
# rel error is the average deviance of the current tree
# divided by the average deviance of the null tree (in-sample fit).
# xerror is based on a 10-fold cross-validation and is again measured
# relative to the deviance of the null model (out-of sample fit).
```

So we need to find the row which has the smallest `xerror`.

```r
iibest = which.min(bigt$cptable[,"xerror"]) #which has the lowest error
bestcp=bigt$cptable[iibest,"CP"]
bestsize = bigt$cptable[iibest,"nsplit"]+1
cat("bestsize: ",bestsize,"\n")
```

```
## bestsize:  10
```

```
nr = nrow(bigt$cptable)
ii = 1:nr
plot(ii,bigt$cptable[ii,"xerror"],col="blue",ylab="out of sample relative error")
abline(v=iibest)
```



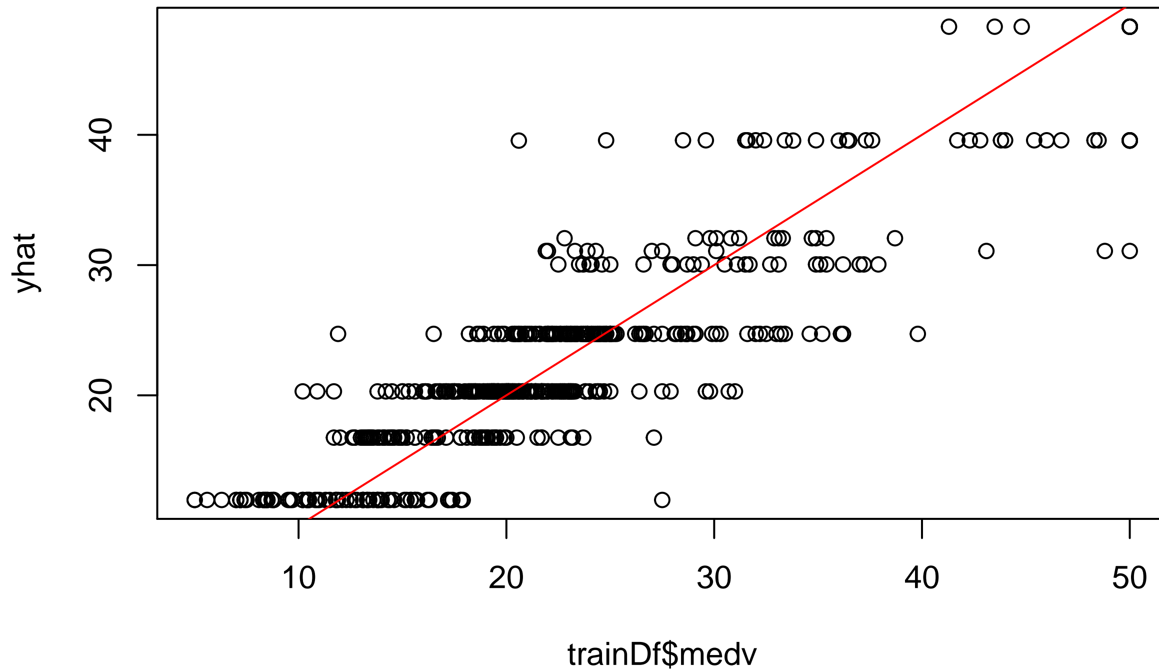
Now we can prune the big tree back using the best CP value.

```
best.tree = prune(bigt,cp=bestcp)
cat("size of best tree: ",length(unique(best.tree$where)))
```

```
## size of best tree:  10
```

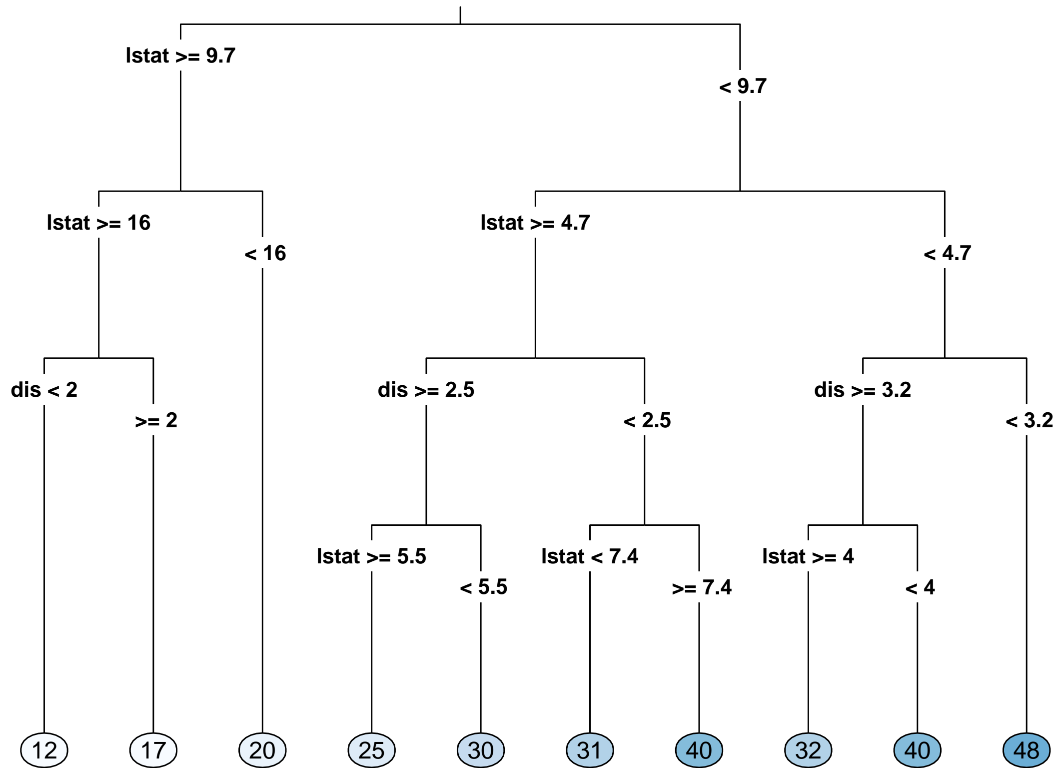
To predict we call `predict`.

```
yhat = predict(best.tree,newdata=trainDf)
plot(trainDf$medv,yhat)
abline(0,1,col="red")
```



To plot the tree, the package `rpart.plot` seems to work.

```
library(rpart.plot)
rpart.plot(best.tree,split.cex=0.9,cex=0.75,type=3,extra=0)
```



And we can get the variable importances:

```
best.tree$variable.importance
```

```
##    lstat      dis
## 30124.84 11628.52
```

In this small tree we can have a look a `where`:

```r
table(best.tree$where)
```

```
##
##   4   5   6  10  11  13  14  17  18  19
##  75  69 150 118  26  11   7  13  25  12
```

```r
best.tree$frame
```

```
##       var   n  wt        dev     yval   complexity ncompete nsurrogate
## 1   lstat 506 506 42716.2954 22.53281 0.4423649998        1          1
## 2   lstat 294 294  7006.2827 17.34354 0.0627501370        1          1
## 4     dis 144 144  2699.2199 14.26181 0.0192570274        1          1
## 8  <leaf>  75  75  1114.6995 11.96933 0.0034979726        0          0
## 9  <leaf>  69  69   761.9316 16.75362 0.0021082271        0          0
## 5  <leaf> 150 150  1626.6094 20.30200 0.0040889341        0          0
## 3   lstat 212 212 16813.8187 29.72925 0.1528339955        1          1
## 6     dis 162 162  6924.4228 26.64630 0.0283371968        1          0
## 12  lstat 144 144  3493.3716 25.67986 0.0141856987        1          0
## 24 <leaf> 118 118  2296.9261 24.71695 0.0070834938        0          0
## 25 <leaf>  26  26   590.4850 30.05000 0.0025371693        0          0
## 13  lstat  18  18  2220.5911 34.37778 0.0071934516        1          1
## 26 <leaf>  11  11  1171.6364 31.08182 0.0045437395        0          0
## 27 <leaf>   7   7   741.6771 39.55714 0.0041006125        0          0
## 7     dis  50  50  3360.8938 39.71800 0.0272239471        1          1
## 14  lstat  38  38  2087.6876 37.00789 0.0113178556        1          0
## 28 <leaf>  13  13   179.3908 32.06154 0.0011284971        0          0
## 29 <leaf>  25  25  1424.8400 39.58000 0.0062580918        0          0
## 15 <leaf>  12  12   110.3000 48.30000 0.0004425419        0          0
```

So `where` tells us which bottom node each training observation is in where the bottom node is index by rows of the `frame`.

The `frame` gives us detailed information on each node of the tree.

Problem 1

We only used 4 variables and only the 2 variables lstat and dis came into our final tree.

Try adding a couple more variables and see if they end up in your tree.

Random Forests and Boosting

We will use classic `randomForest` `R` package.
Note that the package `ranger` is faster for larger data sets.

```r
library(randomForest)
```

```
## randomForest 4.6-14
```

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

Let's do binary classification with the tabloid data.
We read in train and test data sets.

```
trainDf = read.csv("http://www.rob-mcculloch.org/data/td1.csv")
trainDf$purchase = as.factor(trainDf$purchase)
testDf = read.csv("http://www.rob-mcculloch.org/data/td2.csv")
testDf$purchase = as.factor(testDf$purchase)
names(trainDf)[1]="y"
names(testDf)[1]="y"
```

We are going store results from various fits on the training data in the `list` `phatL`.

```
phatL = list() #store the test phat for the different methods here
```

We will try a bunch of settings for random forests.
`setrf` will be a matrix where each row corresponds to a choice of `mtry`, the number of variables sampled each time you want to pick a rule, and `ntree=B`, the number of trees fit.

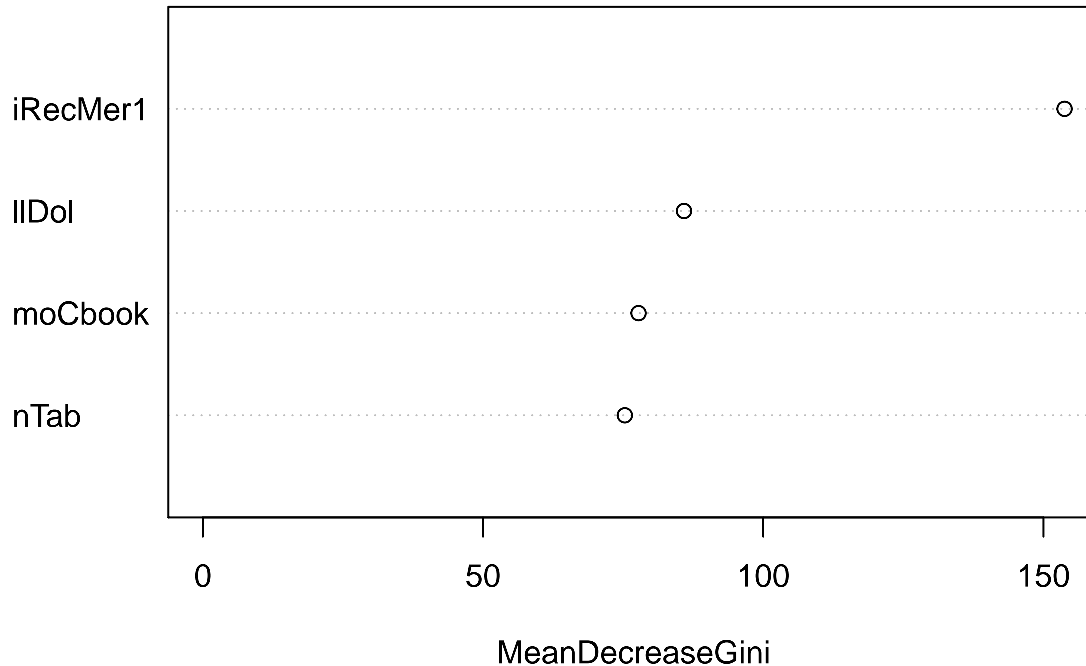
```
##settings for randomForest
p=ncol(trainDf)-1
mtryv = c(p,sqrt(p))
ntreev = c(500,1000)
setrf = expand.grid(mtryv,ntreev)
colnames(setrf)=c("mtry","ntree")
print(setrf) #print out the settings in our grid search
```

```
##   mtry ntree
## 1    4   500
## 2    2   500
## 3    4  1000
## 4    2  1000
```

```
# matrix to store out of sample phats from our random forest models
phatL$rf = matrix(0.0,nrow(testDf),nrow(setrf))
```

We will loop over the settings but lets just try the first one and look at the variable importance.

```
frf = randomForest(y~.,data=trainDf,mtry=setrf[1,1],ntree=setrf[1,2])
varImpPlot(frf) #variable importance
```




```
impvec = frf$importance
cat("the variable importances are:")
```

```
## the variable importances are:
```

```
print(impvec)
```

```
##          MeanDecreaseGini
## nTab             75.30127
## moCbook          77.74451
## iRecMer1        153.76401
## llDol            85.86119
```

Ok, now we will loop over the setting and store predictions on the test data.

```
###fit rf
for(i in 1:nrow(setrf)) {
  cat("on randomForest fit ",i,"\n")
  print(setrf[i,])
  #fit and predict
  frf = randomForest(y~.,data=trainDf,mtry=setrf[i,1],ntree=setrf[i,2])
  phat = predict(frf,newdata=testDf,type="prob")[,2]
  phatL$rf[,i]=phat
}
```

```
## on randomForest fit  1
##   mtry ntree
## 1    4   500
## on randomForest fit  2
##   mtry ntree
## 2    2   500
## on randomForest fit  3
```

```
##   mtry ntree
## 3    4  1000
## on randomForest fit  4
##   mtry ntree
## 4    2  1000
```

Now let's try boosting.
We will use the classic `gbm` `R` package.
Note that `R` package `xgboost` is "an efficient implementation of the gradient boosting".

```
library(gbm)
```

```
## Loaded gbm 2.1.5
```

Again, we will try various settings.
Tree depth: 2,4; number of boosting iterations: 1000 and 5000; shrinkage: .1 and .01.

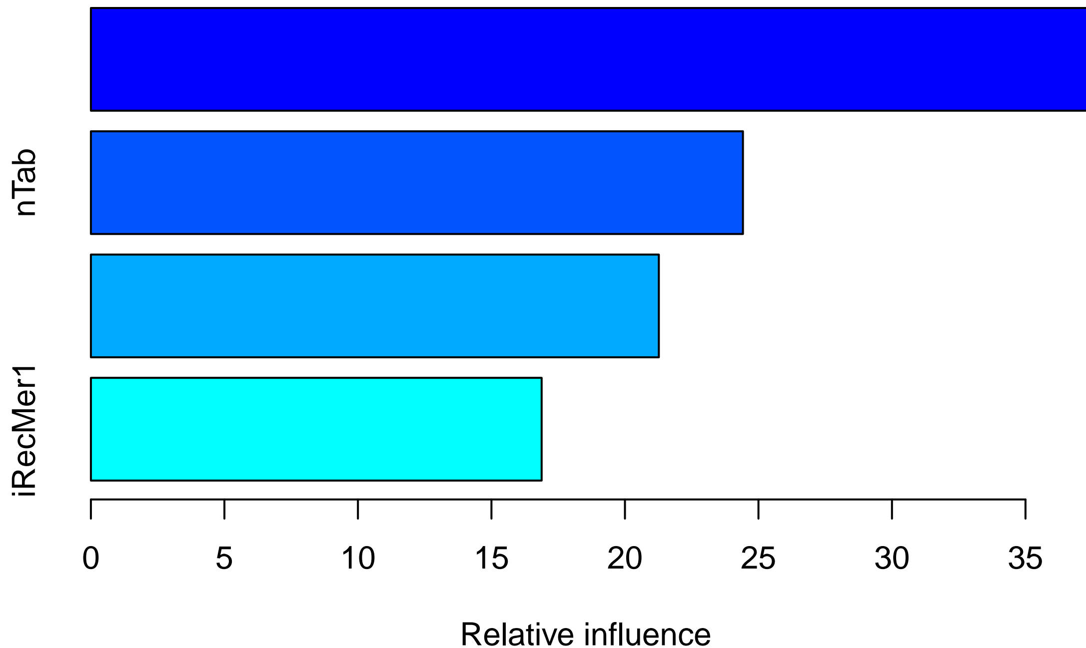
```
##settings for boosting
idv = c(2,4); ntv = c(1000,5000); shv = c(.1,.01)
setboost = expand.grid(idv,ntv,shv)
colnames(setboost) = c("tdepth","ntree","shrink")
print(setboost)
```

```
##   tdepth ntree shrink
## 1      2  1000   0.10
## 2      4  1000   0.10
## 3      2  5000   0.10
## 4      4  5000   0.10
## 5      2  1000   0.01
## 6      4  1000   0.01
## 7      2  5000   0.01
## 8      4  5000   0.01
```

```
#storage for fits
phatL$boost = matrix(0.0,nrow(testDf),nrow(setboost))

#we have to use a numeric y for gbm!
trainDfB = trainDf; trainDfB$y = as.numeric(trainDfB$y)-1
testDfB = testDf; testDfB$y = as.numeric(testDfB$y)-1
```

Let's try just one setting first:

```
fboost = gbm(y~.,data=trainDfB,distribution="bernoulli",
               n.trees=setboost[1,2],interaction.depth=setboost[1,1],
               shrinkage=setboost[1,3])
summary(fboost) # will give variable importance
```




```
##                var  rel.inf
## moCbook   moCbook 37.43074
## nTab         nTab 24.41911
## llDol       llDol 21.26968
## iRecMer1 iRecMer1 16.88046
```

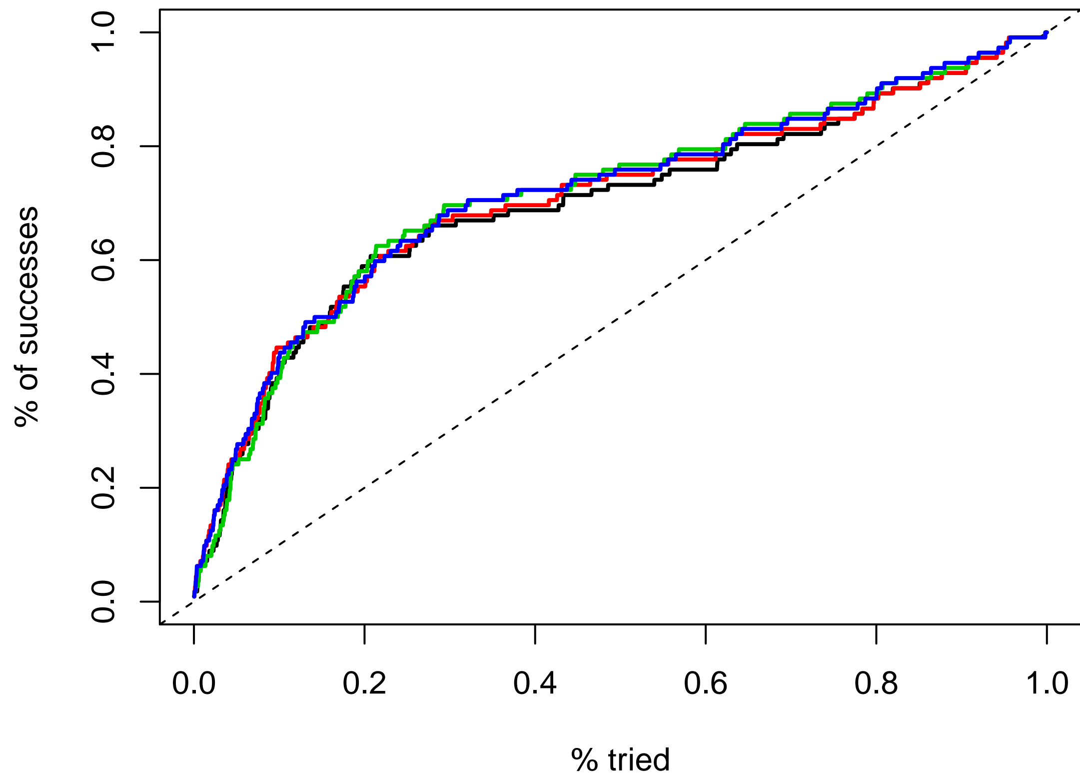
```
##fit boosting
for(i in 1:nrow(setboost)) {
  cat("on boosting fit ",i,"\n")
  print(setboost[i,])
  ##fit and predict
  fboost = gbm(y~.,data=trainDfB,distribution="bernoulli",
               n.trees=setboost[i,2],interaction.depth=setboost[i,1],
               shrinkage=setboost[i,3])
  phat = predict(fboost,newdata=testDfB,n.trees=setboost[i,2],type="response")
  phatL$boost[,i] = phat
}
```

```
## on boosting fit  1
##   tdepth ntree shrink
## 1      2  1000    0.1
## on boosting fit  2
##   tdepth ntree shrink
## 2      4  1000    0.1
## on boosting fit  3
##   tdepth ntree shrink
## 3      2  5000    0.1
## on boosting fit  4
##   tdepth ntree shrink
## 4      4  5000    0.1
## on boosting fit  5
##   tdepth ntree shrink
## 5      2  1000   0.01
## on boosting fit  6
```

```
##   tdepth ntree shrink
## 6      4  1000   0.01
## on boosting fit  7
##   tdepth ntree shrink
## 7      2  5000   0.01
## on boosting fit  8
##   tdepth ntree shrink
## 8      4  5000   0.01
```

Let's compare all the random forests fits by putting them all on the same lift plot.

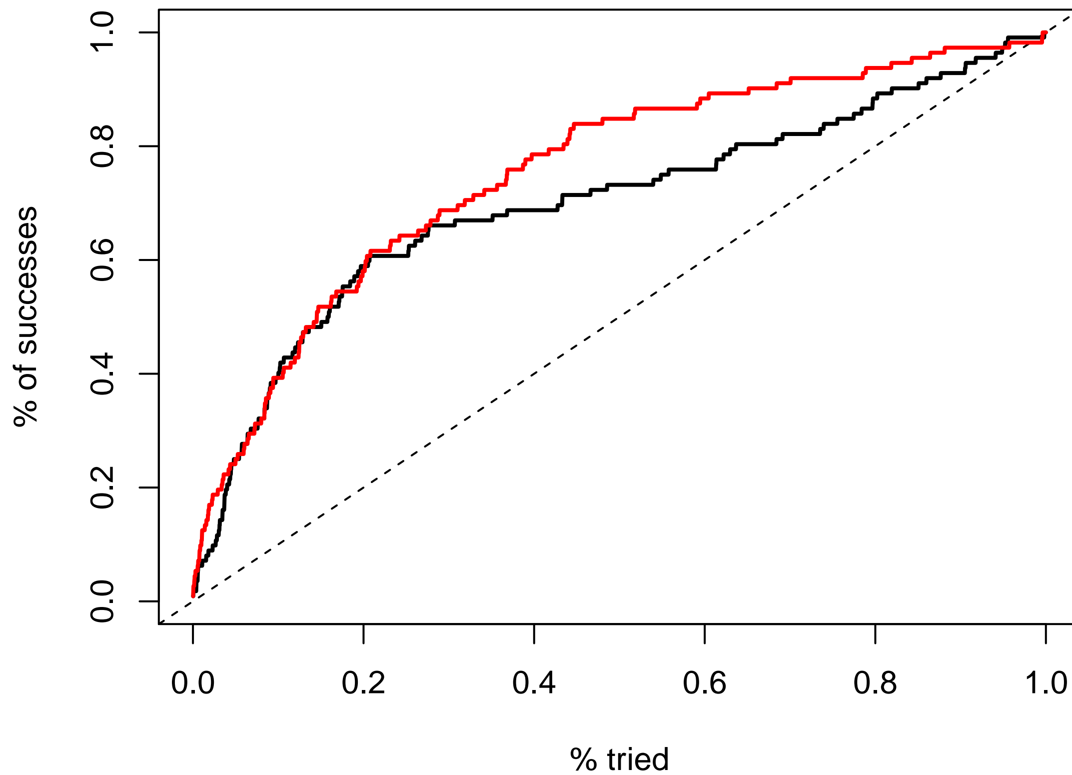
```
source("http://www.rob-mcculloch.org/2019_ml/webpage/notes/lift-loss.R")
nfit = ncol(phatL$rf)

phatLL  = vector("list",nfit)
for(i in 1:nfit) {
  phatLL[[i]] = phatL$rf[,i]
}

lift.many.plot(phatLL,testDf$y)
```



All the random forests fits seem to give similar results!!

Let's compare just the first two boosting fits.

```
phatRFB = vector("list",2)
phatRFB[[1]] = phatL$rf[,1]; phatRFB[[2]] = phatL$boost[,1]
lift.many.plot(phatRFB,testDf$y)
```



The two boosting fits are different after about 20% of the data!

Problem 2

Which boosting fit is the best?

Which is better? Random Forests or boosting?

You can limit yourself to the settings I tried above.
But note that if you really wanted the answer, you might want to do a lot more work trying to tune the boosting.