Random Forest, Bagging and Boosting

Rob Leonard (robleonard@tamu.edu (mailto:robleonard@tamu.edu))

Introduction

For this assignment, let's attempt to make a spam filter. Usually, this would involve a lot of text processing on a huge number of emails. In this case, someone has created a feature matrix for us. The feature matrix has rows given by individual emails and columns given by the number of each word or character that appears in that email, as well as three different numerical measures regarding capital letters (average length of consecutive capitals, longest sequence of consecutive capitals, and total number of capital letters).
The supervisor, Y, is given by the user supplied label marking that email as either spam (Y = 1) or not (Y = 0). Use the missclassification function for this assignment.

```
misClass =function(predClass,trueClass,produceOutput=FALSE){
  confusionMat = table(predClass,trueClass)
  if(produceOutput){
    return(1-sum(diag(confusionMat))/sum(confusionMat))
  }
  else{
    print('misclass')
    print(1-sum(diag(confusionMat))/sum(confusionMat))
    print('confusion mat')
    print(confusionMat)
  }
}
```

Read in the R data set:

```
load("spam.Rdata")
```

Let's make a training and test set.

```
train = spam$train
test  = !train
X     = spam$XdataF[train,]
X_0   = spam$XdataF[test,]
Y     = factor(spam$Y[train])
Y_0   = factor(spam$Y[test])
```

Problem 1: Bagging and random forest

Problem 1.1

To save computations, it is common to iteratively compute batches of random trees until the OOB error rate stabilizes. The following function implements does this as well as demonstrate some typical programming strategies:

```
checkNumberItersF = function(ntrees = 5, tolParm = 1, maxIter = 10, verbose = 0){
  ###
  # tolParm: iterations will continue until the percent decrease
  #          is less than tolParm
  ###
  misClassOut   = list()
  totalTreesOut = list()

  n              = nrow(X)
  votes          = matrix(0,nrow=n,ncol=2)
  totalTrees     = 0
  iterations     = 0
  misClassOld   = 1
  while(iterations < maxIter){
    votes[is.nan(votes)] = 0
    iterations    = iterations + 1
    totalTrees    = totalTrees + ntrees
    if(verbose >= 2){cat('Total trees: ',totalTrees,'\n')}
    outRf        = randomForest(X, Y,ntree = ntrees)

    oob.times        = outRf$oob.times
    votes_iterations = outRf$votes*oob.times
    votes[oob.times>0,] = matrix(votes + votes_iterations,nrow=n)[oob.times>0,]
    if(min(apply(votes,1,sum)) == 0){next}

    Yhat          = apply(votes,1,which.max) - 1
    misClassNew  = misClass(Yhat,Y,produceOutput = TRUE)
    misClassOut[[iterations]]   = misClassNew
    totalTreesOut[[iterations]] = totalTrees
    percentChange = 100*(misClassNew - misClassOld)/misClassOld
    if(verbose >= 1){cat('% change: ',percentChange,'\n')}
    if(percentChange > -tolParm){break}
    misClassOld = misClassNew
  }
  if(iterations == maxIter){
    stop("too many iterations, try a larger ntrees or maxIter value")
  }
  return(list('misClass' = unlist(misClassOut),
              'totalTree' = unlist(totalTreesOut)))
}
```

Comment on the roll of each of these pieces in the above function.

- next: Next will tell R to skip to the next iteration in the while loop without completing the remaining code in the loop. Here, we use next to skip to the next iteration when there are still some values of the test data that have not yet received a classification estimate in the trees created in the previous iterations.
- maxIter: Sets a stopping value / maximum # of iterations for generating new trees in the random forest. This value needs to be set high enough so that we can find the minimum error rate.

- verbose: Verbose is an option that turns on output relating to the total # of trees generated and the percent change in the misclassification rate between iterations. Verbose = 0 will turn off this output.
- while: Loop condition. This allows us to create a random forest via an iterative process where we can create a specified number of trees in each iteration and check to see if we have enough trees after each iteration.
- tolParm: Along with maxIter, this is also a stopping criteria for the while loop. If the change in the misclassification rate shows an improvement (i.e. it's > negative 1% the while loop continues). If the change in the misclassfication rate is too small (below 1% decrease, or is an increase), we stop iterating new trees.
- misClassOld: This is the misclassification rate from the prior iteration. It allows us to see if the new iteration of trees in the random forest improves on this misclassification rate and plays a role in stopping the while loop.
- stop: this stops the while loop if we have reached the maximum number of iterations but we still haven't found our solution that minimizes the misclassification rate. If this occurs, we need to increase the number of iterations to find the lowest misclass rate.

Call this function with a suitable value of maxIter and verbose so that the function produces the minimal amount of output. Report back the number of iterations you find.

```
set.seed(1)
checkNumberIters = checkNumberItersF(ntrees = 5, tolParm = 1, maxIter = 9, verbose = 0 )  # 8 it
erations
```

Random Forest classifications

Fit the random forest with default mtry and include both importance and proximity. What is the test misclassification rate, sensitivity, specificity, precision, recall, and confusion matrix for the chosen random forest? How does the test misclassification rate compare with the OOB misclassification rate?

```
ntrees = max(checkNumberIters$totalTree)
outRf  = randomForest(X, Y, ntree = ntrees,  importance=TRUE, proximity=TRUE) # mtry=sqrt(ncol
(X)),

class.tree = predict(outRf, newdata = X_0, type = 'class')
test.results = misClass(class.tree,Y_0,produceOutput=FALSE)
```

```
## [1] "misclass"
## [1] 0.05691057
## [1] "confusion mat"
##          trueClass
## predClass   0   1
##         0 129  10
##         1   4 103
```

```
testError = round(misClass(class.tree,Y_0,produceOutput=TRUE),4)
testSens  = round(test.results[2,2]/(test.results[1,2]+test.results[2,2]),4)
testSpec  = round(test.results[1,1]/(test.results[1,1]+test.results[2,1]),4)
testPrec  = round(test.results[2,2]/(test.results[2,1]+test.results[2,2]),4)
testRec   = testSens
OOBmiscl  = outRf$err.rate[ntrees,1]
```

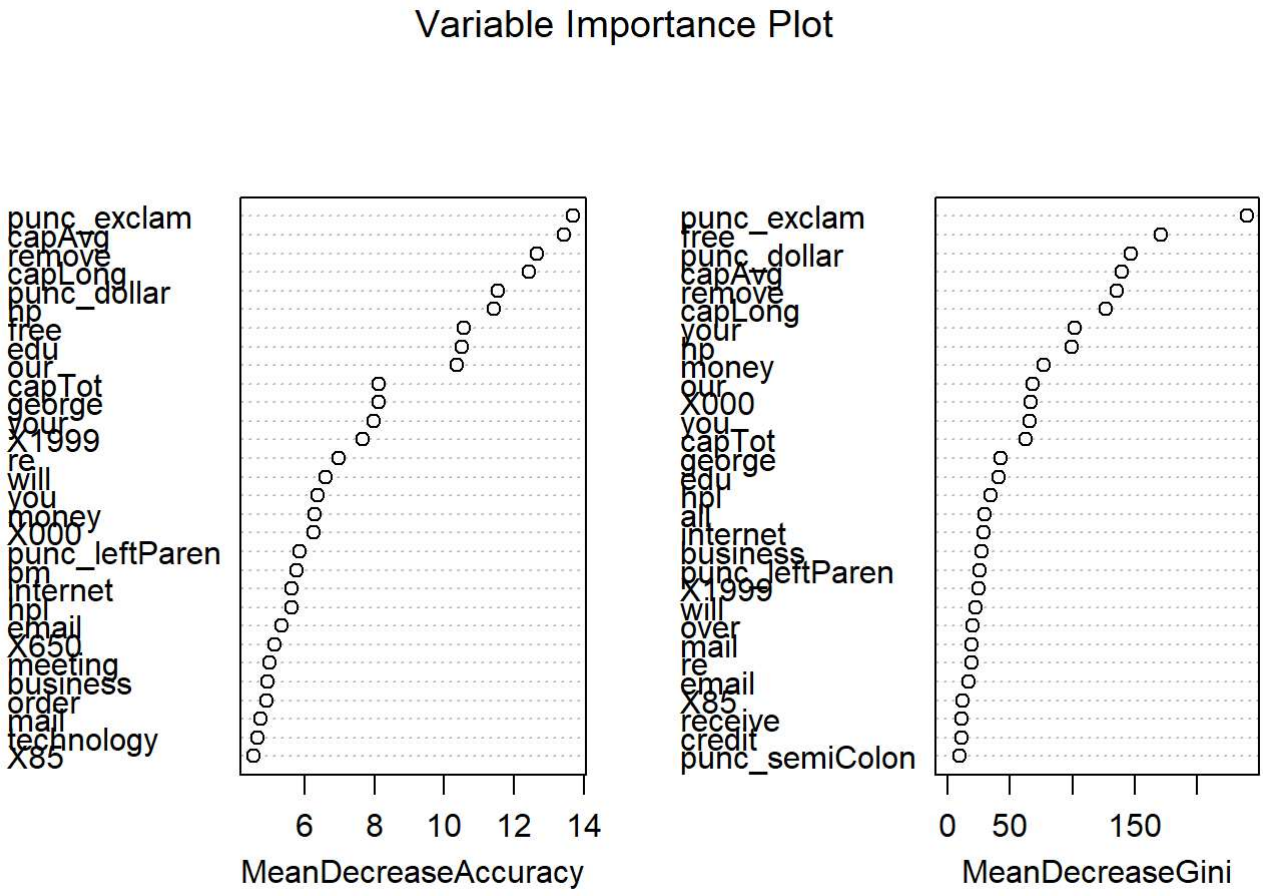
- The test misclassification rate is 0.0569
- The test sensitivity is 0.9115
- The test specificity is 0.9699
- The test precicision is 0.9626
- The test recall is 0.9115
- The test confusion matrix is 129, 4, 10, 103
- The OOB misclassification rate is 0.0509759

The test error is moderately lower than the OOB misclassification rate, but it is in the same neighborhood (below 5% but above 4%). So the OOB misclass rate provides a somewhat reasonable estimate of the test error.

Problem 1.3. Variable importance for random forests

Get a variable importance plot for the permuted OOB importance measure.

```
varImpPlot(outRf, main = "Variable Importance Plot")
```



What is the most important feature? Punc_exclam (exclamation point, !) is the most important feature.

Problem 2: Boosting with gbm

There are 3 main choices involved in boosting trees: the number of boosting iterations, the learning rate, and the tree complexity.

Problem 2.1. Boosting

Using the gbm package, explore a variety of values for these parameters in terms of the associated test misclassifciation rates. Do your conclusions change depending on if you use Adaboost versus logistics loss? (answer below in the appropriate ‘answer’ category. There are of course interactions between these tuning parameters. You can answer qualitatively, for example: ‘the test error increases as lambda increases’). Feel free to change the grids to further your investigation.

```
lambdaGrid            = c(.0001,.1,1,10)
interaction.depthGrid = c(4,5,6)
n.treesGrid           = c(500,1000)
resultsGrid           = array(0,dim=c(length(lambdaGrid),
                                      length(interaction.depthGrid),
                                      length(n.treesGrid)),
                              dimnames = list('lambda'=as.character(lambdaGrid),
                                              'interaction'=as.character(interaction.depthGrid),
                                              'n.trees'=as.character(n.treesGrid)))
resultsBoost          = list('bernoulli' = resultsGrid,
                             'adaboost'  = resultsGrid)

set.seed(1)
verbose = 0
Ychar   = as.character(Y) # gbm doesn't accept factor Y
Ychar_0 = as.character(Y_0)
for(distribution in c('bernoulli','adaboost')){
  lamIter = 0
  for(lambda in lambdaGrid){
    lamIter = lamIter + 1
    intIter = 0
    for(interaction.depth in interaction.depthGrid){
      intIter  = intIter + 1
      treeIter = 0
      for(n.trees in n.treesGrid){
        treeIter = treeIter + 1
        boostOut = gbm(Ychar~.,data=X,
                       n.trees=n.trees, interaction.depth=interaction.depth,
                       shrinkage=lambda, distribution = distribution)
        fHat  = predict(boostOut,X_0,n.trees=n.trees)
        Yhat = rep(0,nrow(X_0))
        Yhat[fHat > 0] = 1

        Yhat = as.factor(Yhat)
        if(verbose > 0){
        cat('lambda = ',lambda,' interaction.depth = ',interaction.depth, ' lambda = ',lambda,'
 n.trees = ',n.trees,'\n')
        }
        if(verbose > 1){
          misClass(Yhat,Ychar_0)
        }
        resultsBoost[[distribution]][lamIter,intIter,treeIter] = misClass(Yhat,Ychar_0, produceO
utput = TRUE)
      }
    }
  }
}
resultsBoost
```

```
## $bernoulli
## , , n.trees = 500
##
##        interaction
## lambda           4          5          6
##   1e-04 0.45934959 0.45934959 0.45934959
##   0.1   0.03658537 0.04065041 0.04065041
##   1     0.08943089 0.06910569 0.08130081
##   10    0.78455285 0.78455285 0.47967480
##
## , , n.trees = 1000
##
##        interaction
## lambda           4          5          6
##   1e-04 0.45934959 0.45934959 0.45934959
##   0.1   0.03658537 0.02439024 0.02439024
##   1     0.07723577 0.07723577 0.07317073
##   10    0.46747967 0.58536585 0.21544715
##
##
## $adaboost
## , , n.trees = 500
##
##        interaction
## lambda           4          5          6
##   1e-04 0.45934959 0.45934959 0.45934959
##   0.1   0.02845528 0.02845528 0.02439024
##   1     0.03658537 0.03658537 0.04065041
##   10    0.47154472 0.45934959 0.35365854
##
## , , n.trees = 1000
##
##        interaction
## lambda           4          5          6
##   1e-04 0.45934959 0.45934959 0.45934959
##   0.1   0.02439024 0.02845528 0.02845528
##   1     0.04471545 0.03252033 0.04471545
##   10    0.36991870 0.28048780 0.13008130
```

How does the behavior change with Lambda?

Setting a lambda too close to 0 or too high such as 10 results in high test misclassification rates. Setting lambda to a small constant like 0.1 has the best results, followed by lambda =1. This corresponds to the philosophy of slow learning, of making a large number of smaller improvements rather than making a small number of very large improvements.

How does the behavior change with interaction.depth?

Changing the interaction depth seems to generally have less impact on reducing test misclassification than changing lambda. Increasing the interaction depth has no effect on the amount of test misclassification when lambda is set to the very smallest value close to 0. In most cases, changing the interaction depth results in very

little change in the test misclass There are slight to moderate benefits to increasing interaction depth for cases with the larger number of trees (1000) and with the higher levels of lambda, although it also provides a significant reduction in test misclass when lambda = 0.1 under Bernoulli.

How does the behavior change with n.trees?

Increasing the number of trees is beneficial to reducing the test misclassification rate generally when lambda is set to the higher values of 1 and 10, although there is some slight benefit under both Bernoulli and Adaboost at lambda = .1.

How does the behavior change with adaboost vs. logistic loss??

In most cases of the lambda/interaction/# tree combinations, adaboost results in lower test misclassification rates than Bernoulli, especially with lambda =0.1 and number of trees =500. Also, for larger lambdas and # of trees, adaboost performs better than Bernoulli. Overall, it appears that adaboost is generally providing lower test errors.

Problem 3: Boosting vs. Random Forest

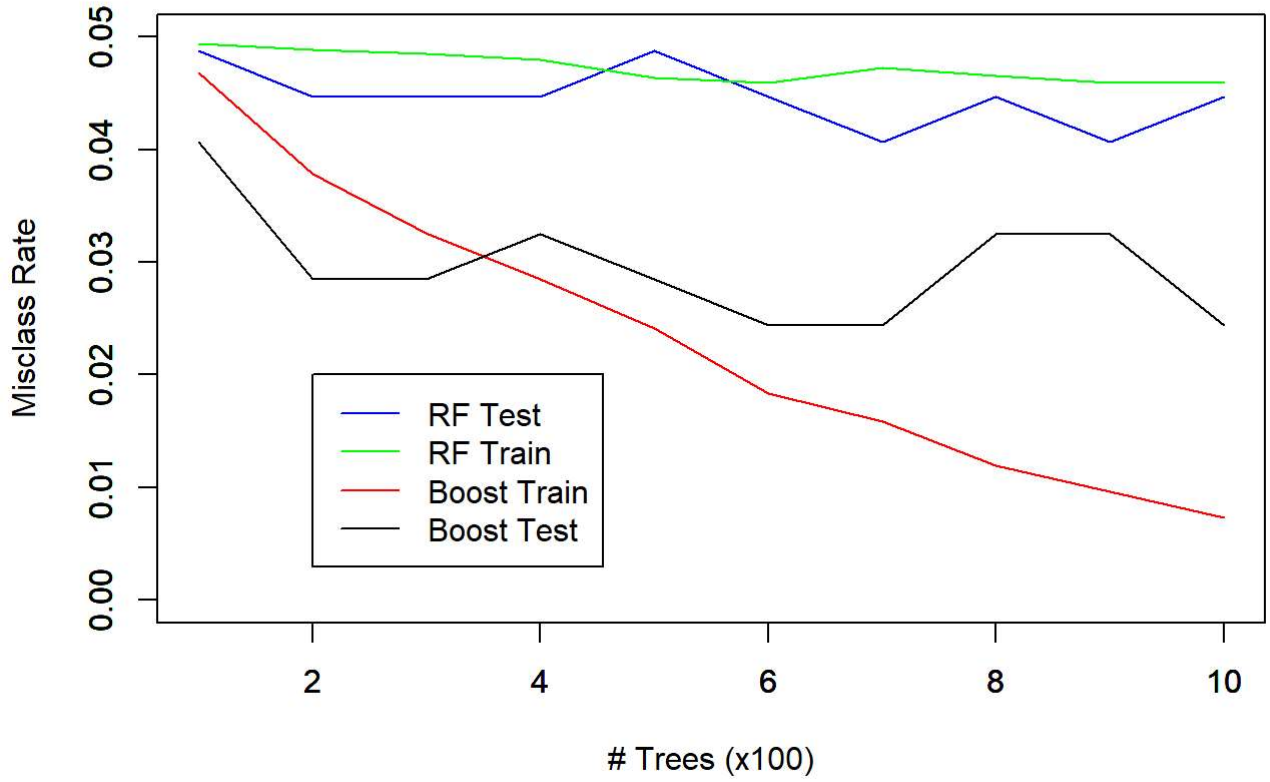
Choose values for the tuning parameters from boosting (except for the n.trees argument) based on your exploration in problem 2.

Make a plot of the training misclassifications and test misclassifications for a grid of ‘B’ values up to 1000 for both bagging and boosting (so, one plot with 4 curves). If you need to do a loop over a grid of ‘B’ values, you can choose a rather coarse grid (one with maybe 10 grid points). You can add more trees to gbm via gbm.more instead of recomputing the whole solution or look into predict.gbm for an easy way to get predictions over a grid. Also note that the ‘training’ and ‘test’ error from within gbm is in terms of the loss function used, not the 0-1 loss function.

```
rfresult.test = rfresult.train = rep(0,10)
for (i in 1:10) {
  outRfprob3        = randomForest(X, Y, ntree = 100*i,  importance=TRUE, proximity=TRUE)
  rfresult.train[i] = round(misClass(outRfprob3$predicted,Y,produceOutput=TRUE),4)
  class.treeprob3   = predict(outRfprob3, newdata = X_0, type = 'class')
  rfresult.test[i]  = round(misClass(class.treeprob3,Y_0,produceOutput=TRUE),4)
}

# boosting - with 100 trees initially
resultsBoosttest = resultsBoosttrain = rep(0,10)
 boostOutp3 = gbm(Ychar~.,data=X,
                        n.trees=100, interaction.depth=4,
                        shrinkage=0.1, distribution = 'adaboost')
 fHatp3  = predict(boostOutp3,X_0,n.trees=100)
 Yhatp3 = rep(0,nrow(X_0))
 Yhatp3[fHatp3 > 0] = 1
 Yhatp3 = as.factor(Yhatp3)
 resultsBoosttest[1] = misClass(Yhatp3,Ychar_0, produceOutput = TRUE)
# for training data
 fHatp3tr  = predict(boostOutp3,X,n.trees=100)
 Yhatp3tr = rep(0,nrow(X))
 Yhatp3tr[fHatp3tr > 0] = 1
 Yhatp3tr = as.factor(Yhatp3tr)
 resultsBoosttrain[1] = misClass(Yhatp3tr,Ychar, produceOutput = TRUE)
# use gbm.more to add trees, get test error
for (i in 2:10){
  treenumb = (100*i-100)
  boostresult = gbm.more(boostOutp3, n.new.trees = treenumb )
  fHatp3  = predict(boostresult,X_0, n.trees=100*i )
  Yhatp3 = rep(0,nrow(X_0))
  Yhatp3[fHatp3 > 0] = 1
  Yhatp3 = as.factor(Yhatp3)
  resultsBoosttest[i] = misClass(Yhatp3,Ychar_0, produceOutput = TRUE)
  # for training data
  boostresulttr = gbm.more(boostOutp3, n.new.trees = treenumb )
  fHatp3tr  = predict(boostresult,X, n.trees=100*i )
  Yhatp3tr = rep(0,nrow(X))
  Yhatp3tr[fHatp3tr > 0] = 1
  Yhatp3tr = as.factor(Yhatp3tr)
  resultsBoosttrain[i] = misClass(Yhatp3tr,Ychar, produceOutput = TRUE)
  }

plot(x=seq(1:10), y=rfresult.test, col="blue", ylim=c(0,0.05), ylab="Misclass Rate", xlab="# Tre
es (x100)", type='l')
par(new=T)
plot(x=seq(1:10), y=rfresult.train, col="green", ylim=c(0,0.05), ylab="", xlab="", type='l')
par(new=T)
plot(x=seq(1:10), y=resultsBoosttrain, col="red", ylim=c(0,0.05), ylab="", xlab="", type = 'l')
par(new=T)
plot(x=seq(1:10), y=resultsBoosttest, col="black", ylim=c(0,0.05), ylab="", xlab="", type = 'l')
legend(x=2,y=0.02,c("RF Test","RF Train", "Boost Train", "Boost Test"),col=c("blue","green","re
d","black"), lty=c(1,1,1,1))
```

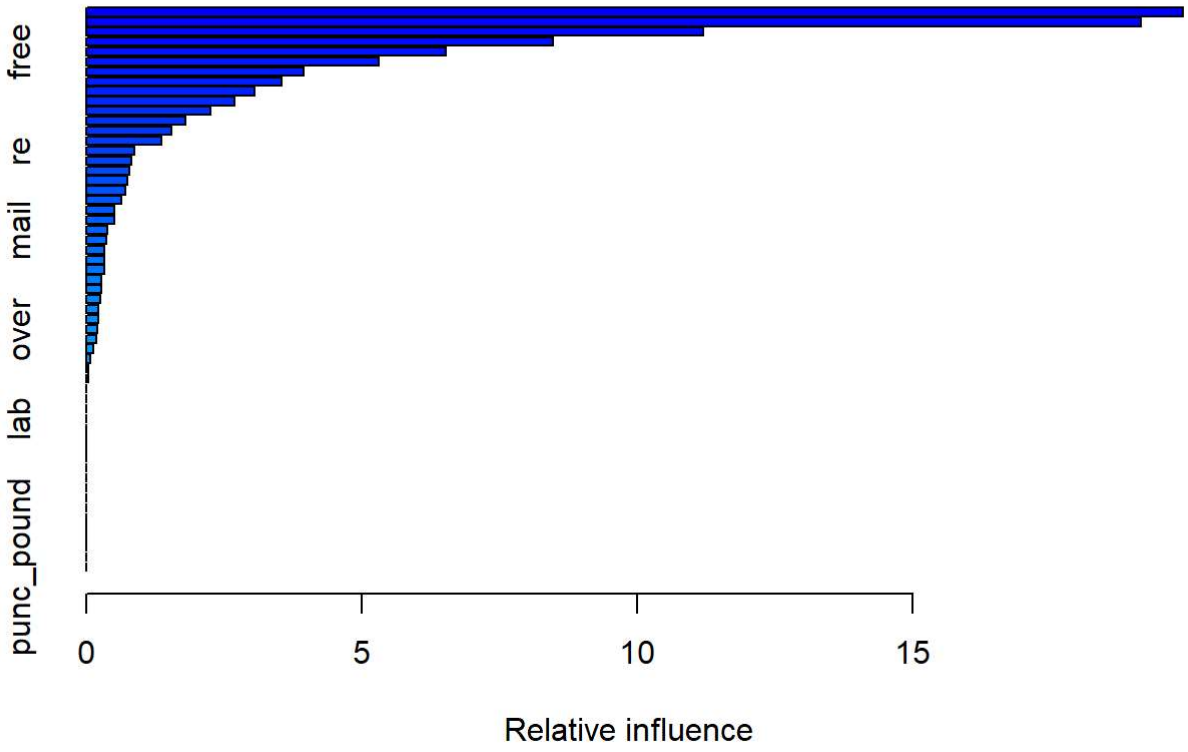


Which method gets the lowest training error? The lowest test error? Boosting gets both the lowest test and training error assuming we use 100 or more trees.

Problem 4: Boosting Importance

Just like in bagging, we can get an idea about variable importance. For each tree in the boosting ensemble, and for each feature, we record the amount in the training misclassification rate is reduced by making a split on the jth feature. We take the average of this over all B trees to get the importance. We can have R give us this information from running summary on the output of gbm and choosing the appropriate number of trees.

```
head(summary(boostOutp3), n=3)
```




```
##                     var  rel.inf
## punc_dollar punc_dollar 19.91023
## punc_exclam punc_exclam 19.14399
## remove           remove 11.20180
```

What are the 3 most important variables and how do they compare to the 3 most important from RF? 2 of the 3 are the same as RF ($ and !).