Analysis of the Recursion Partitioning Process Using Rpart.Control

**Abstract**

We followed the code laid out by Nolan and Lang in *Case Studies in Data Science with R* to identify spam emails. One of the steps of this process was creating creating a classification tree to perform regressive partitioning of the data. Using the R package ‘rpart.control’ we look at alternative methods and conditions that could be used to decrease our Type I and Type II errors.

**Introduction**

Spam email is still sent out at a prevalent pace. Depending on its contents, it could clog up one’s inbox or potentially infect a computer with a virus. Spam filters filter out spam email from our inboxes in order to prevent the unwanted messages of polluting our inboxes. Therefore, we must create a procedure that can classify an email as spam or not spam by examining various variables of the emails. The emails used here are from spamassassin that were created in order to test and develop spam filters. The procedure performed using code from Nolan and Lang uses Naive Bayes method in order to classify each message.

The ‘rpart’ package in R is used for recursive partitioning and regression trees. It allows user to visualize the outcomes for categorical or continuous outcomes using either classification or regression trees respectfully. The package can be used to create the decision tree, analyze its results, and edit the tree. The tree can be edited by pruning it. Pruning a decision tree is an excellent way to prevent overfitting to a model.

**Background**

Naive Bayes method calculates the probability of y given x times the probability of y all divided by the probability of x. In this paper’s context, it is calculating the probability of an email is spam given it is not spam multiplied by the probability it is spam all divided by the probability that the email is not spam. These probabilities are calculated by using variables text mined from each email and then classifies the email as spam or not spam.

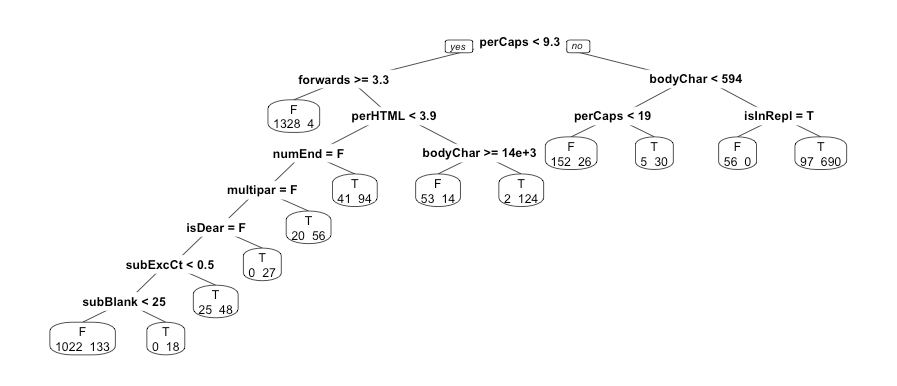
The ‘rpart.control’ package allows for a more in-depth control at the parameters that control the fit of the trees created from the ‘rpart' package. The parameters that can be changed are the minsplit, minbucket, cp, maxcompete, maxsurrogte, usesurrogate, xval, surrogatestyle, and maxdepth.

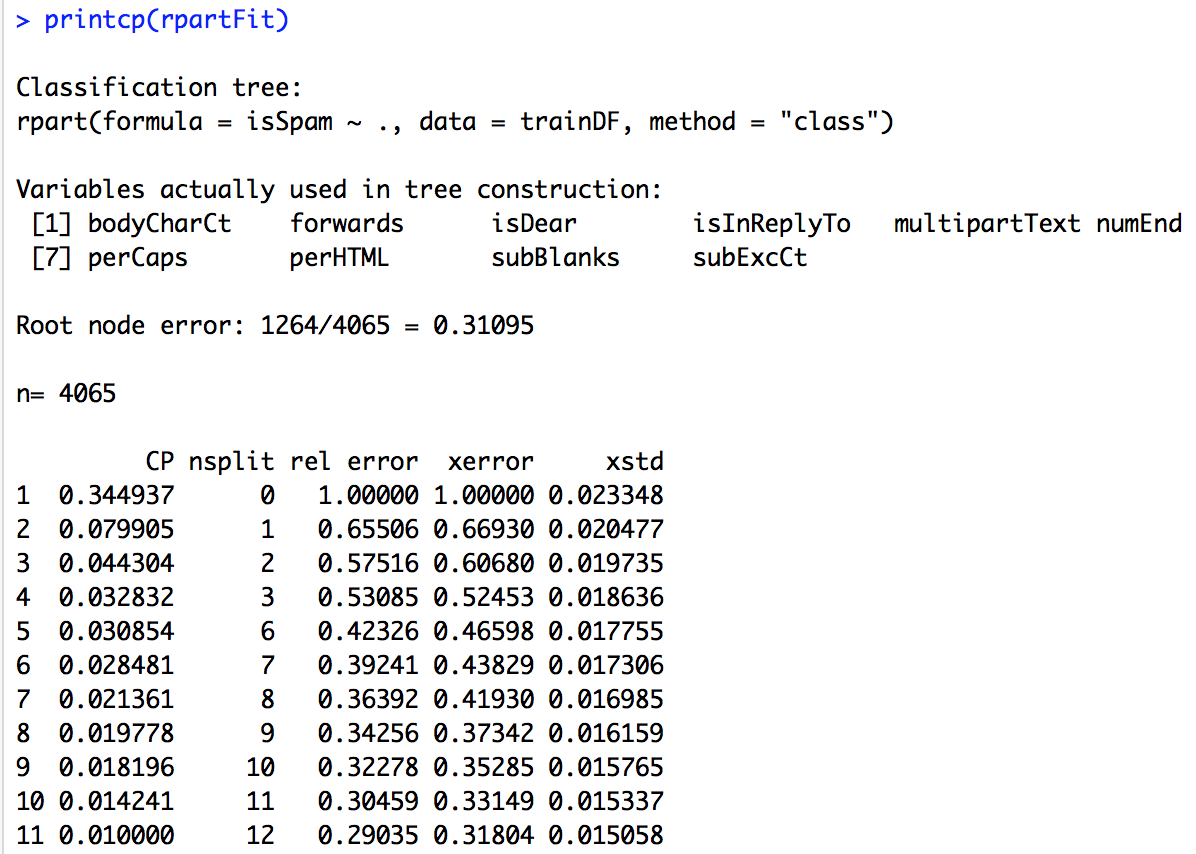
The minsplit is the minimum number of observations required in a node in order for a split to occur. The minbucket is the minimum number of observations in any terminal node. The minsplit and the minbucket are related. If only 1 of the two are specified, R sets the other using the equation minsplit = 3\*minbucket. The cp is the complexity parameter. It prevents R from making splits in the tree that have a insubstantial effect on the lack of fit. The co dictates the minimum decrease in the lack of fit parameter that must occur when a split is made. Otherwise, the split does not occur. The maxcompete setting is the number of splits that competed to be in the output.It provides the inform of what splits were almost made. The maxsurrogate specifies the number of surrogate splits used by the output. Usesurrogate is more definitive in how to split data. If the value is specified to be zero, an observation with missing values is not allowed to move down the tree to other nodes. If the value is specified to be one, surrogate values are used when the chosen variable is missing. If the value is specified to be two, then even if all surrogate values are missing, the observation can still be sent to the next node. in this case, it is sent to the node that has the most observations. The xval is the number of cross-validations performed. Maxdepth is the maximum depth that a node can be from the root node.

There are other parameters that can be tweaked using the ‘rpart’ package besides rpart.control. The are called in the rpart function under the ‘parms’ section. These are advanced parameters specific to different methods. For classification trees, some of the optional parameters can can be used are the vector of prior probabilities, the loss matrix, and the splitting index.

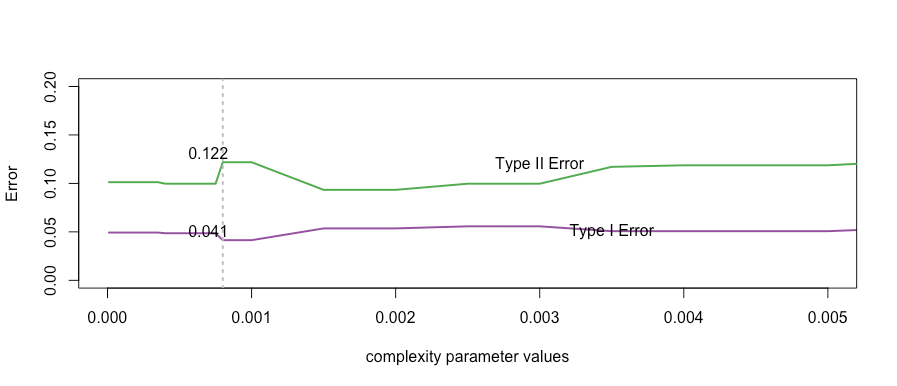
**Methods**

In our initial analysis, the only control that was specified was the complexity parameter, ‘cp’. Our root node first makes a decision based on perCaps. It asks if perCaps is less than 9.3. This variable is the percentage of a message that is capitalized. This is telling us that is less than 9.3 percent of the body is capitalized is a good indicator of whether an email is spam. However, this is not the only time this variable is used. In the case of perCaps is greater than 9.3 percent, it is then asked later down the tree if perCaps is less than 19 percent. This means that for some classification cases it needs perps to be: 9.3 < perCaps < 19.





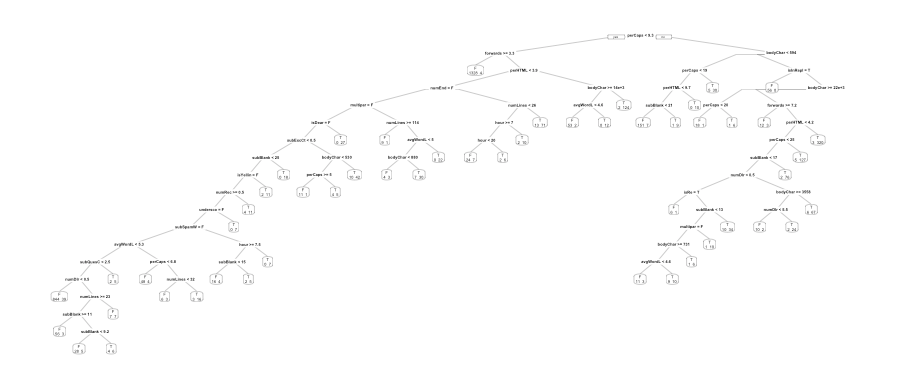
A Type I error is the proportion of messages that were not spam that were classified as spam. The Type I error rate is is 0.074. A Type II error is The Type II error rate is 0.13. A 13% error rate was not good though so we initially changed the complexity parameter value of 0.0001. This lowered both of our error rates. After changing the complexity parameter, our Type I error is 0.039 and our Type II error is 0.105.



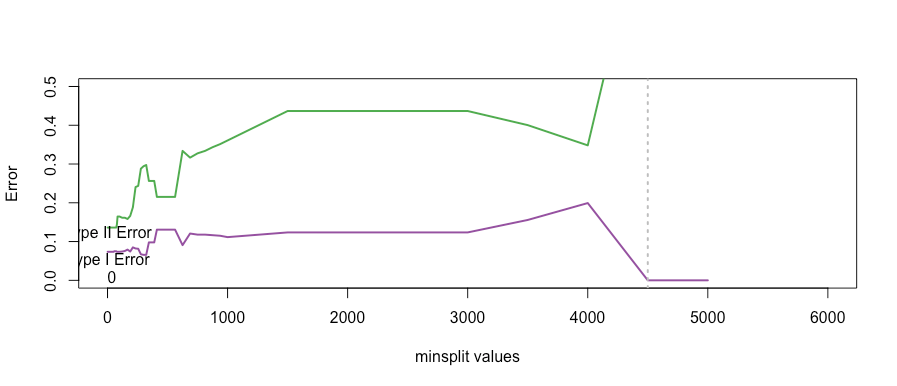
Using the the methods and process described above, we will performed similar processes based on other variables that were available to us in in the rpart.control package. The next section details the results gleaned from the discovery.

**Results**

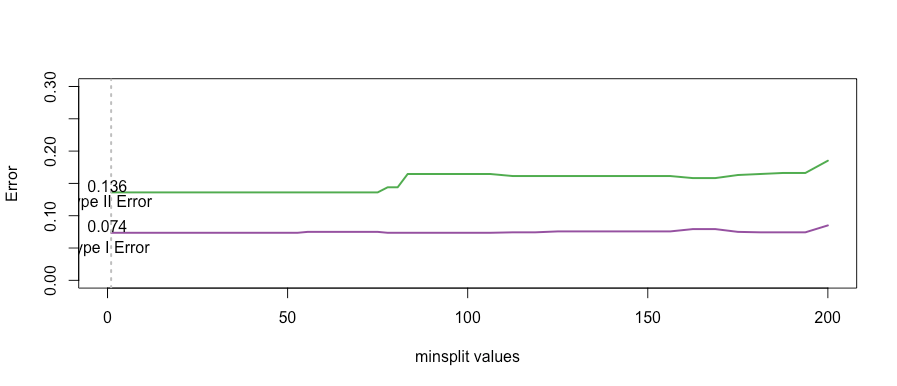
We will now look at how the decision tree looks when the complexity parameters are at 0.0001. This is much more complex than our initial tree above with a lot more nodes and more depth. This tree is so detailed, that there are multiple nodes with a single digit amount of observations in them. Again, very different from our initial tree. You can see why using this tree gives us a lower Type I Error.

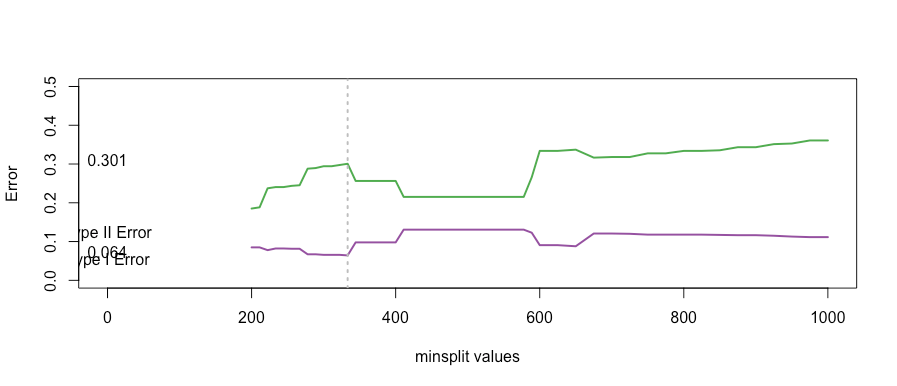


We will now look at when we manipulate the minsplit function. This function dictates how many records must be in a node before there can be a split.

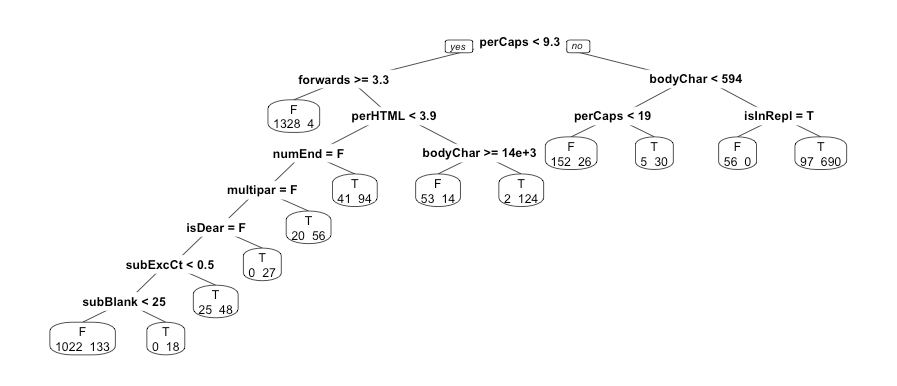


The reason that the Type I error is so small at such a high number is because at that point, there are not enough records for there to be more than 1 node so all of the data would fit into that node. Lets not change our parameters for a maximum minsplit value of 200. Any higher and the Type II error is too high for us.

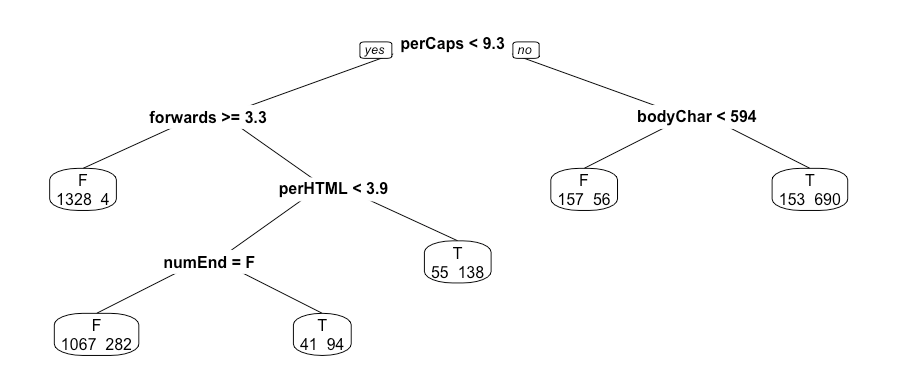


This show that our error is lowest when our minsplit is as low as possible. It gives us a Type I Error of 0.74 and a Type II Error of 0.136. We can see that these values will stay constant until about minsplit = 75. We can see that if we further analyze between x:[200,1000] that the errors are larger. Therefore, we recommend not setting a minsplit value. 

We will now analyze the trees when the minsplit is set at different values. Below is the tree when minsplit is set at 10.

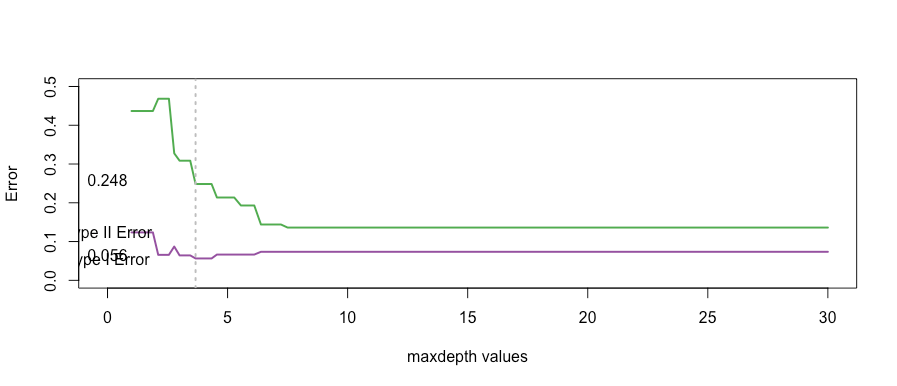


We will now compare this to it being set at 350 and 5000. These are important points because they are possible optimal values for minsplit in terms of minimizing Type I Errors. When minsplit is set at 350, the decision tree is noticeably simpler. This makes sense because more observations are required in order to split nodes. This causes no splitting of nodes to occur where bodyChar, multipar, the second perCaps, and isInRepl were split before. There is no need to show the trees for minsplit = 4000 or 5000. Though it may have the smallest Typle I Error at minsplit=5000, it is because there is not even a split. There’s only a single split at the value 4000. These are not helpful for our analysis.

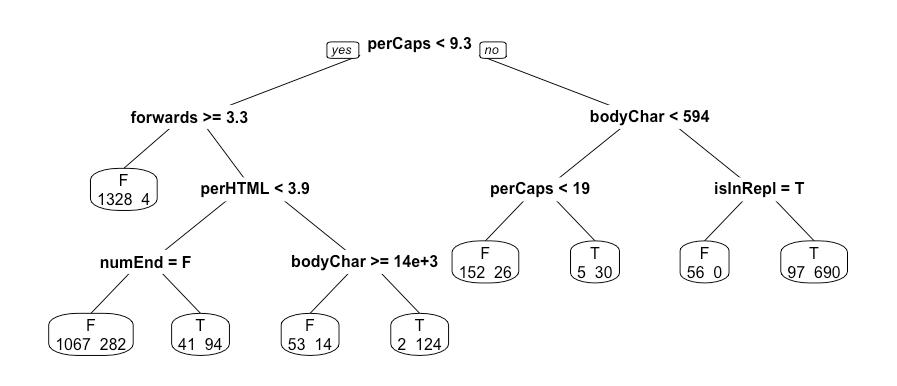


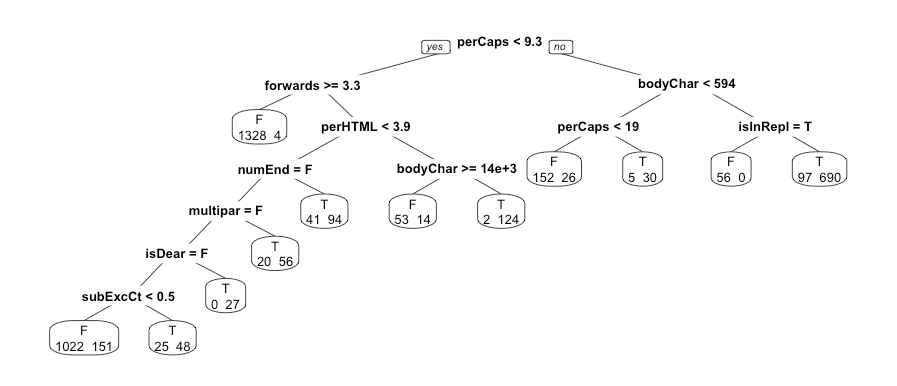
We will not be analyzing the minbucket value as it is directly connected to the minsplit value. R has the relationship between the two as minsplit = 3\*minbucket. Therefore, all of our conclusions about minsplit can be extrapolated to minbucket.

Next, we will look at the maxdepth. This variable is the maximum depth of any node in the tree. The graph below shows that the Type 1 error is lowest with the more freedom it has to go to any depth of nodes in the tree it deems necessary. This is unsurprising considering our initial tree only had a depth of 9. Our lowest Type I Error is 0.056 at maxdepth = 4. However, at that point our Type II Error is too high at 0.248. As we can see on the graph, if we increase the maxdepth value, the Type I Error would slightly rise to about .07 or .08, however the Type II Error will be cut in half. Therefore, I recommend having the maxdepth value of at least 7.

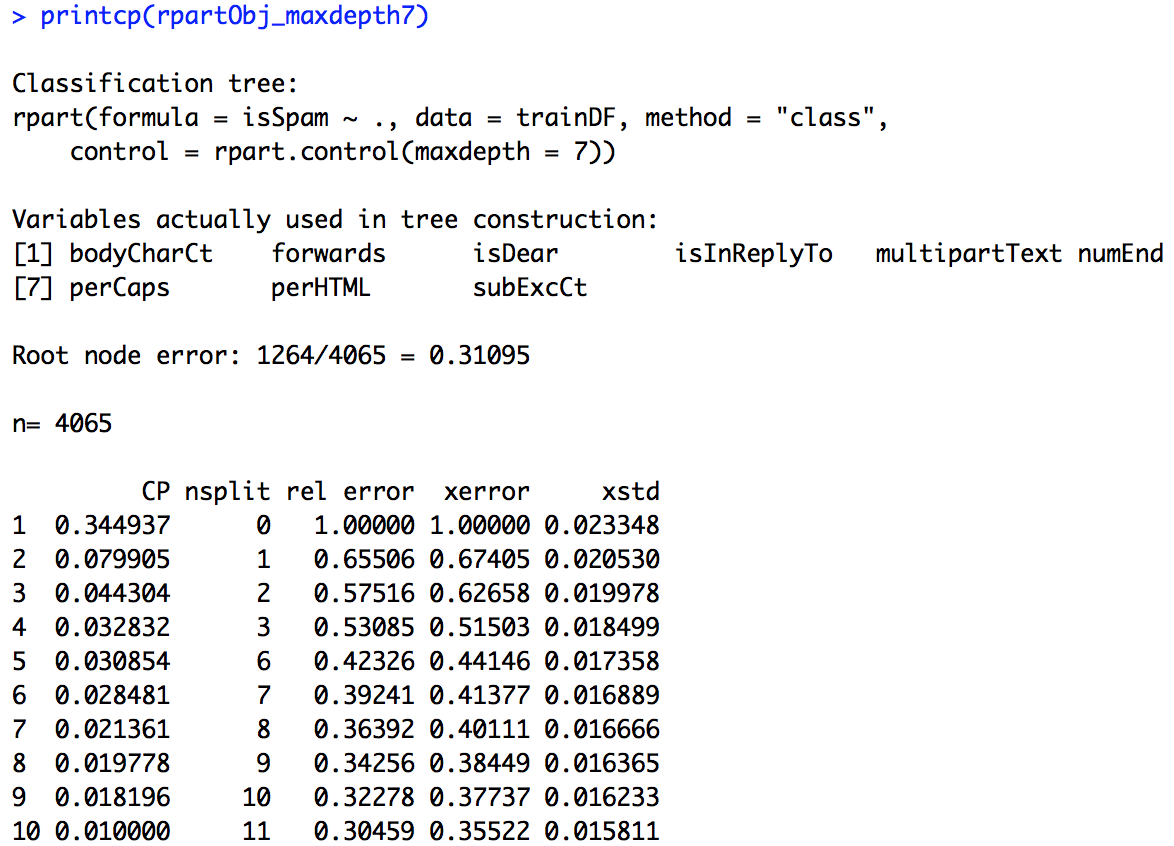
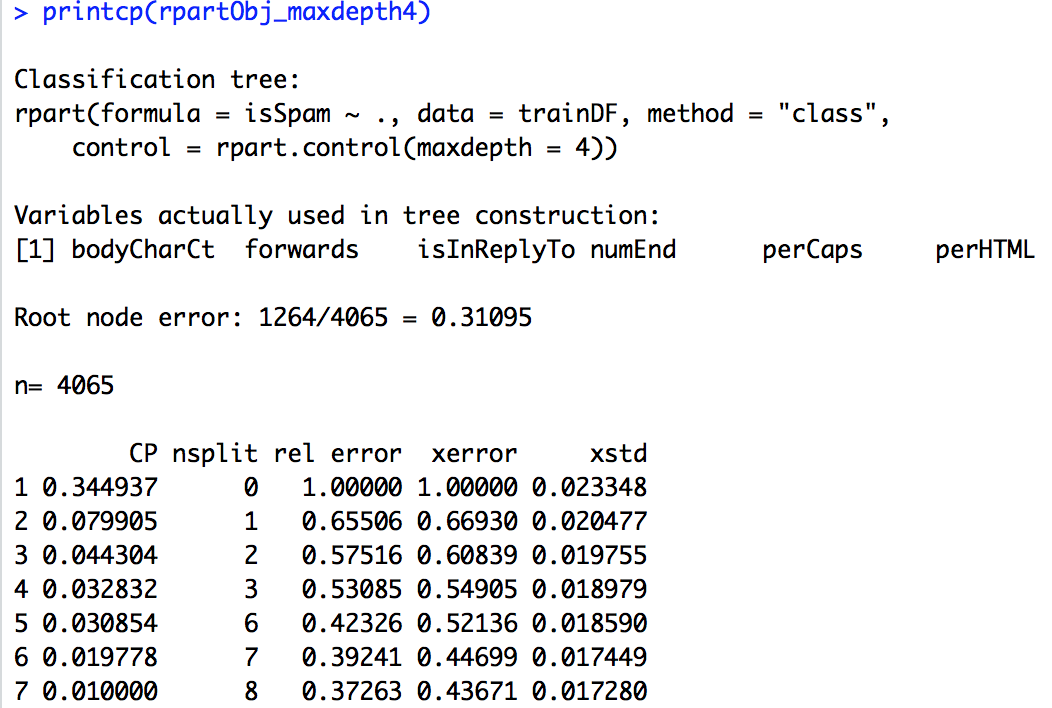


The decision tree for these two ideas look very different. Below are the decision trees for maxdepth=4 and maxdepth=7. Besides the obvious difference of the different depths of the trees we will look at how the areas that have a depth of 7 adapt when they are only allowed to have a depth of 4. The change begins at the numEnd node.

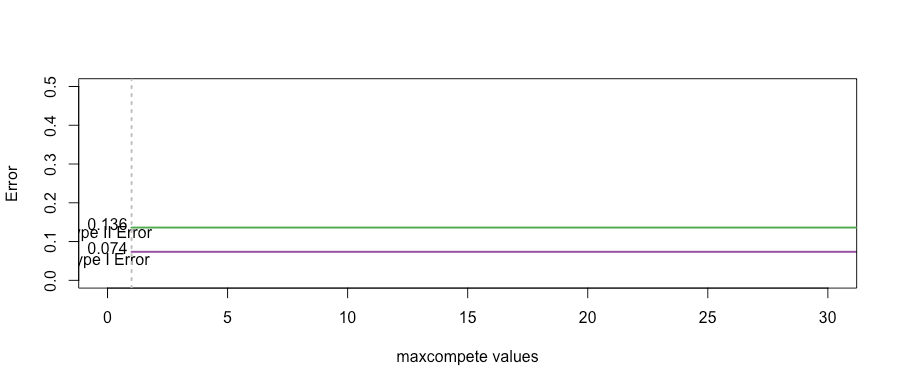




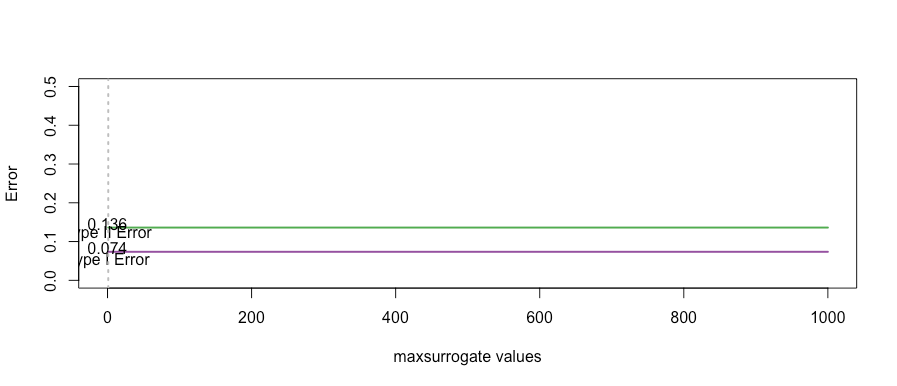
Additionally, allowing for more depth lowered the relative error of the numEnd node slightly. However, the nodes with a depth of 5,6, & 7 all have relative errors themselves.



The next variable we will look at is maxcompete. It informs not just what the optimal variable to split is, but the runner ups that were almost chosen as well. As this information is for our use only and does not actually effect how any decisions are made, it makes sense that any value for maxcompete would not change our Type I or Type II Errors from our initial values.



The maxsurrogate variable is the maximum number of surrogate variables that are retained at each node. Usually, when a node splits, the question is what other splits would have as many correct classifications as the chosen one. For maxsurrogate, it asks what other splits have the same classification patterns. This is again not something that affects the Type I or Type II Errors.



**Conclusion**

We can conclude that the most effective control in the ‘rpart’ package for lowering our Type I Error rate and Type II error rate was the complexity parameter ‘cp’. It had the most dramatic change in our Type I Error Rate from roughly 7% to 3%. Some of these controls such as maxsurrogate, though helpful for looking at potential trees, were not a factor in terms of our error rates.

In the future, one thing we can look at it how using multiple parameters affected the Type I Error Rate and the decision trees. For example, we could look at how inputting parameters for both ‘cp’ and maxdepth affected the Type I Error Rate and the effect it would have on the shape of the corresponding decision tree. In this research we only used one control at a time. Using multiple at a time may bring about even better, more accurate results.

Bibliography

Nolan, D., Lang, D. “Data Science in R: A Case Studies Approach to Computational Reasoning and Problem Solving”. 2015.

Therneau, T., Atkinson, B., Ripley, B. (2018). Rpart: Recursive Partitioning and Regressive Trees. R package version 4.1-13. <https://cran.r-project.org/web/packages/rpart/rpart.pdf>

Therneau, T., Atkinson, B., Ripley, B. (2018). An Introduction to Recursive Partitioning Using Rpart Routines. Mayo Foundation. <https://cran.r-project.org/web/packages/rpart/vignettes/longintro.pdf>

Appendix

# Phillip Efthimion

# Case Study 10

setwd("/Users/Phillip/Documents/SMU/QuantifyingTheWorld/SpamAssassin/")

spamPath = getwd()

list.dirs(spamPath, full.names = FALSE)

length(list.files(paste(spamPath, sep = .Platform$file.sep)))

dirNames = list.files(path = paste(spamPath, sep=.Platform$file.sep))

dirNames

sapply(paste(spamPath, dirNames, sep=.Platform$file.sep), function(dir) length(list.files(dir)))

fullDirNames = paste(spamPath, dirNames, sep=.Platform$file.sep)

fullDirNames

fileNames = list.files(fullDirNames[1], full.names = TRUE)

fileNames[1]

indx = c(1:5,15,27,68,69,329,404,427,516,852,971)

fn = list.files(fullDirNames[1], full.names = TRUE)[indx]

sampleEmail = sapply(fn, readLines)

sampleEmail

length(sampleEmail)

msg = sampleEmail[[1]]

msg

which(msg == "")[1]

match("", msg)

splitPoint <- match("", msg)

splitPoint <- match("", msg)

msg[ (splitPoint - 2): (splitPoint + 6)]

header = msg[1:(splitPoint - 1)]

body = msg[-(1:splitPoint)]

splitMessage = function(msg){}

splitMessage = function(msg){

splitPoint = match("", msg)

header = msg[1:(splitPoint - 1)]

body = msg[-(1:splitPoint)]

return(list(header = header, body = body))

}

sampleSplit = lapply(sampleEmail, splitMessage)

length(sampleSplit)

class(sampleSplit)

sampleSplit[[6]]$header

sampleSplit[[6]]$body

indx = c(indx, 1070,1267,1450,1585,1709,1234,1046,1149,1641,1406)

fn = list.files(fullDirNames[1], full.names = TRUE)[indx]

sampleEmail = sapply(fn, readLines)

sampleSplit = lapply(sampleEmail, splitMessage)

length(sampleSplit)

header = sampleSplit[[1]]$header

grep("Content-Type", header)

header

headerList = lapply(sampleSplit, function(msg) msg$header)

CTloc = sapply(headerList, grep, pattern = "Content-Type")

CTloc

sapply(headerList, function(header) {

CTloc = grep("Content-Type", header)

if (length(CTloc) == 0) return(NA)

CTloc

})

hasAttach = sapply(headerList, function(header) {

CTloc = grep("Content-Type", header)

if (length(CTloc) == 0) return(FALSE)

grepl("multi", tolower(header[CTloc]))

})

header = sampleSplit[[16]]$header

boundaryIdx = grep("boundary=", header)

header[boundaryIdx]

sub(".\*boundary\"(.\*)\";.\*", "\\1", header[boundaryIdx])

sub(".\*boundary=\"(.\*)\";.\*", "\\1", header[boundaryIdx])

header2 = headerList[[17]]

boundaryIdx2 = grep("boundary=", header2)

header2[boundaryIdx2]

sub(".\*boundary=\"(.\*)\";.\*", "\\1", header2[boundaryIdx2])

boundary2 = gsub('"', "", header2[boundaryIdx2])

boundary2

# find portion of string that affects the boundary string

sub(".\*boundary= \*(.\*);?.\*", "\\1", boundary2)

# see if we can find boundary string in first example

boundary = gsub('"',"", header[boundaryIdx])

sub(".\*boundary= \*(.\*);?.\*", "\\1", boundary)

sub(".\*boundary= \*([^;]\*);?.\*", "\\1", boundary)

# wrap up what we did into one function

getBoundary = function(header) {

boundaryIdx = grep("boundary=", header)

boundary = gsub('"', "", header[boundaryIdx])

gsub(".\*boundary= \*([^;]\*);?.\*", "\\1", boundary)

}

# find what different attachments look like

sampleSplit[[16]]$body

sampleSplit[[21]]$body

sampleSplit[[11]]$body

# find boundary for attachment

boundary = getBoundary(headerList[[15]])

body = sampleSplit[[15]]$body

bstring = paste("--", boundary, sep="")

bstringLocs = which(bstring == body)

bstringLocs

# Try one with an attachment

boundary = getBoundary(headerList[[25]])

body = sampleSplit[[25]]$body

bstring = paste("--", boundary, sep="")

bstringLocs = which(bstring == body)

bstringLocs

# Find ending string

eString = paste("--", boundary, "--", sep="")

eStringLocs = which(eString == body)

eStringLocs

# look at meesage

msg = body[ (bstringLocs[1] + 1) : (bstringLocs[2] - 1)]

tail(msg)

msg = body[ (eStringLocs + 1) : length(body)]

tail(msg)

# create function that drops all of the attachments from code above

dropAttach = function(body, boundary) {

bstring = paste("--", boundary, sep="")

bstringLocs = which(bstring == body)

if (length(bstringLocs) <= 1) return(body)

eString = paste("--", boundary, "--", sep="")

eStringLocs = which(eString == body)

if (length(eStringLocs) ==0)

return(body[ (bstringLocs[1] + 1) : (bstringLocs[2] - 1)])

n = length(body)

if (eStringLocs < n)

return( body[ c( (bstringLocs[1] + 1) : (bstringLocs[2] - 1),

( (eStringLocs + 1) : n )) ] )

return( body[ (bstringLocs[1] + 1) : (bstringLocs[2] - 1) ])

}

# want to seperate into words

# see how we will seperate and what to discard

head(sampleSplit[[1]]$body)

head(sampleSplit[[3]]$body)

# make everything lower case

tolower(gsub("[[:punct:]0-9[:blank:]]+"," ", head(sampleSplit[[3]]$body)))

msg = head(sampleSplit[[3]]$body)

msg[ c(1,3,26,27)]

# Get rid of single letter words

cleanMsg = tolower(gsub("[[:punct:]0-9[:blank:]]+"," ", msg))

cleanMsg[c(1,3,26,27)]

words = unlist(strsplit(cleanMsg, "[[:blank:]]+"))

words

words = words[nchar(words) > 1]

words

library(tm)

stopWords = stopwords()

stopWords

cleanSW = tolower(gsub("[[:punct:]0-9[:blank:]]+"," ", stopWords))

cleanSW

SWords = unlist(strsplit(cleanSW, "[[:blank:]]+"))

SWords = SWords[nchar(SWords) > 1]

SWords

stopWords = unique(SWords)

stopWords

# Match up and create vector of all words that are stopwords

words = words[ !(words %in% stopWords)]

head(words)

# create function that returns vector of unique words in message

findMsgWords = function(msg, stopWords) {

if (is.null(msg)) return(character())

words = unique(unlist(strsplit(cleanText(msg), "[[:blank:]\t]+")))

# drop empty and one letter words

words = words[nchar(words) > 1]

words = words[!(words %in% stopWords)]

invisible(words)

}

cleanText = function(msg) { tolower(gsub("[[:punct:]0-9[:blank:]]+"," ", msg))}

# create fubction to process all of the words of all of the emails

processAllWords = function(dirName, stopWords)

{

# read all files in the directory

fileNames = list.files(dirName, full.names = TRUE)

# drop files that are not email, i.e., cmds

notEmail = grep("cmds$", fileNames)

if ( length(notEmail) > 0) fileNames = fileNames[ - notEmail ]

messages = lapply(fileNames, readLines, encoding = "latin1")

# split header and body

emailSplit = lapply(messages, splitMessage)

# put body and header in own lists

bodyList = lapply(emailSplit, function(msg) msg$body)

headerList = lapply(emailSplit, function(msg) msg$header)

rm(emailSplit)

# determine which messages have attachments

hasAttach = sapply(headerList, function(header) {

CTloc = grep("Content-Type", header)

if (length(CTloc) == 0) return(0)

multi = grep("multi", tolower(header[CTloc]))

if (length(multi) == 0) return(0)

multi

})

hasAttach = which(hasAttach > 0)

# find boundary strings for messages with attachments

boundaries = sapply(headerList[hasAttach], getBoundary)

# drop attachments from message body

bodyList[hasAttach] = mapply(dropAttach, bodyList[hasAttach],

boundaries, SIMPLIFY = FALSE)

# extract words from body

msgWordsList = lapply(bodyList, findMsgWords, stopWords)

invisible(msgWordsList)

}

# warnings not a concern because of incomplete final lines

msgWordsList = lapply(fullDirNames, processAllWords, stopWords = stopWords)

numMsgs = sapply(msgWordsList, length)

numMsgs

fullDirNames

isSpam = rep(c(FALSE, FALSE, FALSE, TRUE, TRUE), numMsgs)

msgWordsFlat = unlist(msgWordsList, recursive = FALSE)

class(msgWordsFlat)

length(msgWordsFlat)

msgWordsList <- msgWordsFlat

# msgWordsFlat is our clean data set for Naive Bayes

# Change to msgWordsList to follow book code

# Now we want to implement the Naive Bayes Classifier

# First we have testing and training data

numEmail = length(isSpam)

numSpam = sum(isSpam)

numHam = numEmail - numSpam

set.seed(418910)

# determine indicies of test spam and not test spam messages

testSpamIdx = sample(numSpam, size = floor(numSpam/3))

testHamIdx = sample(numHam, size = floor(numHam/3))

# indicies then used to select word vectors from msgWordsList

testMsgWords = c((msgWordsList[isSpam])[testSpamIdx],

(msgWordsList[!isSpam])[testHamIdx] )

trainMsgWords = c((msgWordsList[isSpam])[ - testSpamIdx],

(msgWordsList[!isSpam])[ - testHamIdx])

# create test and train versions of isSpam using rep()

testIsSpam = rep(c(TRUE, FALSE),

c(length(testSpamIdx), length(testHamIdx)))

trainIsSpam = rep(c(TRUE, FALSE),

c(numSpam - length(testSpamIdx),

numHam - length(testHamIdx)))

# Probability Estimates from Training Data

# estimate prob. that a certain word is present given message is spam or spam'

# creating our own dictionary from our training data

bow = unique(unlist(trainMsgWords))

length(bow)

# for each word in 'bow', copmute # of spam messages in training set that contains the word

# start with creating a vector to hold counts

spamWordCounts = rep(0, length(bow))

# add names to elements of spamWordsCounts with 'names'

names(spamWordCounts) = bow

# retreive unique words, collapse words on all spam messages, calculate freq table, update spmWordsCounts

tmp = lapply(trainMsgWords[trainIsSpam], unique)

tt = table( unlist(tmp) )

spamWordCounts[ names(tt) ] = tt

# estimate probabilities

# compute freqs

computeFreqs =

function(wordsList, spam, bow = unique(unlist(wordsList)))

{

# create a matrix for spam, ham, and log odds

wordTable = matrix(0.5, nrow = 4, ncol = length(bow),

dimnames = list(c("spam", "ham",

"presentLogOdds",

"absentLogOdds"), bow))

# For each spam message, add 1 to counts for words in message

counts.spam = table(unlist(lapply(wordsList[spam], unique)))

wordTable["spam", names(counts.spam)] = counts.spam + .5

# Similarly for ham messages

counts.ham = table(unlist(lapply(wordsList[!spam], unique)))

wordTable["ham", names(counts.ham)] = counts.ham + .5

# Find the total number of spam and ham

numSpam = sum(spam)

numHam = length(spam) - numSpam

# Prob(word|spam) and Prob(word | ham)

wordTable["spam", ] = wordTable["spam", ]/(numSpam + .5)

wordTable["ham", ] = wordTable["ham", ]/(numHam + .5)

# log odds

wordTable["presentLogOdds", ] =

log(wordTable["spam",]) - log(wordTable["ham", ])

wordTable["absentLogOdds", ] =

log((1 - wordTable["spam", ])) - log((1 -wordTable["ham", ]))

invisible(wordTable)

}

# apply copmuteFreqs to all of training data set

trainTable = computeFreqs(trainMsgWords, trainIsSpam)

# 3.6.3 Classifying New Messages

newMsg = testMsgWords[[1]]

newMsg = newMsg[!is.na(match(newMsg, colnames(trainTable)))]

present = colnames(trainTable) %in% newMsg

sum(trainTable["presentLogOdds", present]) +

sum(trainTable["absentLogOdds", !present])

newMsg = testMsgWords[[ which(!testIsSpam)[1] ]]

newMsg = newMsg[!is.na(match(newMsg, colnames(trainTable)))]

present = (colnames(trainTable) %in% newMsg)

sum(trainTable["presentLogOdds", present]) +

sum(trainTable["absentLogOdds", !present])

# Large negative value indicates that it is not spam

# place all of this code into a function to calculate log liklihood ratio for all test messages

computeMsgLLR = function(words, freqTable)

{

# Discards words not in training data.

words = words[!is.na(match(words, colnames(freqTable)))]

# Find which words are present

present = colnames(freqTable) %in% words

sum(freqTable["presentLogOdds", present]) +

sum(freqTable["absentLogOdds", !present])

}

# apply the function to each message in test set

testLLR = sapply(testMsgWords, computeMsgLLR, trainTable)

# compare summary stats of LLR vs test data values

tapply(testLLR, testIsSpam, summary)

# boxplot

spamLab = c("ham", "spam")[1 + testIsSpam]

boxplot(testLLR ~ spamLab, ylab = "Log Likelihood Ratio",

# main = "Log Likelihood Ratio for Randomly Chosen Test Messages",

ylim=c(-500, 500))

# Type 1 Errors

typeIErrorRate =

function(tau, llrVals, spam)

{

classify = llrVals > tau

sum(classify & !spam)/sum(!spam)

}

typeIErrorRate(0, testLLR,testIsSpam)

typeIErrorRate(-20, testLLR,testIsSpam)

typeIErrorRates =

function(llrVals, isSpam)

{

o = order(llrVals)

llrVals = llrVals[o]

isSpam = isSpam[o]

idx = which(!isSpam)

N = length(idx)

list(error = (N:1)/N, values = llrVals[idx])

}

typeIIErrorRates = function(llrVals, isSpam) {

o = order(llrVals)

llrVals = llrVals[o]

isSpam = isSpam[o]

idx = which(isSpam)

N = length(idx)

list(error = (1:(N))/N, values = llrVals[idx])

}

xI = typeIErrorRates(testLLR, testIsSpam)

xII = typeIIErrorRates(testLLR, testIsSpam)

tau01 = round(min(xI$values[xI$error <= 0.01]))

t2 = max(xII$error[ xII$values < tau01 ])

# pdf("LinePlotTypeI+IIErrors.pdf", width = 8, height = 6)

# 3.6.4 Computational Considerations

library(RColorBrewer)

cols = brewer.pal(9, "Set1")[c(3, 4, 5)]

plot(xII$error ~ xII$values, type = "l", col = cols[1], lwd = 3,

xlim = c(-300, 250), ylim = c(0, 1),

xlab = "Log Likelihood Ratio Values", ylab="Error Rate")

points(xI$error ~ xI$values, type = "l", col = cols[2], lwd = 3)

legend(x = 50, y = 0.4, fill = c(cols[2], cols[1]),

legend = c("Classify Ham as Spam",

"Classify Spam as Ham"), cex = 0.8,

bty = "n")

abline(h=0.01, col ="grey", lwd = 3, lty = 2)

text(-250, 0.05, pos = 4, "Type I Error = 0.01", col = cols[2])

mtext(tau01, side = 1, line = 0.5, at = tau01, col = cols[3])

segments(x0 = tau01, y0 = -.50, x1 = tau01, y1 = t2,

lwd = 2, col = "grey")

text(tau01 + 20, 0.05, pos = 4,

paste("Type II Error = ", round(t2, digits = 2)),

col = cols[1])

k = 5

numTrain = length(trainMsgWords)

partK = sample(numTrain)

tot = k \* floor(numTrain/k)

partK = matrix(partK[1:tot], ncol = k)

testFoldOdds = NULL

for (i in 1:k) {

foldIdx = partK[ , i]

trainTabFold = computeFreqs(trainMsgWords[-foldIdx], trainIsSpam[-foldIdx])

testFoldOdds = c(testFoldOdds,

sapply(trainMsgWords[ foldIdx ], computeMsgLLR, trainTabFold))

}

testFoldSpam = NULL

for (i in 1:k) {

foldIdx = partK[ , i]

testFoldSpam = c(testFoldSpam, trainIsSpam[foldIdx])

}

xFoldI = typeIErrorRates(testFoldOdds, testFoldSpam)

xFoldII = typeIIErrorRates(testFoldOdds, testFoldSpam)

tauFoldI = round(min(xFoldI$values[xFoldI$error <= 0.01]))

tFold2 = xFoldII$error[ xFoldII$values < tauFoldI ]

# 3.6.4

smallNums = rep((1/2)^40, 2000000)

largeNum = 10000

print(sum(smallNums), digits = 20)

print(largeNum + sum(smallNums), digits = 20)

for (i in 1:length(smallNums)) {

largeNum = largeNum + smallNums[i]

}

print(largeNum, digits = 20)

# 3.7 Recursive Partitioning and Classification Trees

# 3.8 Organizing an email Message into an R Data Structure

sampleSplit = lapply(sampleEmail, splitMessage)

# 3.8.1 Processing the Header

header = sampleSplit[[1]]$header

header[1:12]

# Check pg 166

header[1] = sub("^From", "Top-From:", header[1])

header[1]

headerPieces = read.dcf(textConnection(header), all = TRUE)

headerPieces[, "Delivered-To"]

# convert headPieces into character vector

headerVec = unlist(headerPieces)

dupKeys = sapply(headerPieces, function(x) length(unlist(x)))

names(headerVec) = rep(colnames(headerPieces), dupKeys)

headerVec[ which(names(headerVec) == "Delivered-To") ]

length(headerVec)

length(unique(names(headerVec)))

# put all of this code into the new processHeader function

processHeader = function(header)

{

# modify the first line to create a key:value pair

header[1] = sub("^From", "Top-From:", header[1])

headerMat = read.dcf(textConnection(header), all = TRUE)

headerVec = unlist(headerMat)

dupKeys = sapply(headerMat, function(x) length(unlist(x)))

names(headerVec) = rep(colnames(headerMat), dupKeys)

return(headerVec)

}

headerList = lapply(sampleSplit,

function(msg) {

processHeader(msg$header)} )

contentTypes = sapply(headerList, function(header)

header["Content-Type"])

names(contentTypes) = NULL

contentTypes

# 3.8.2 Processing Attachments

# determine which messages have attachments. Examine Content-type

hasAttach = grep("^ \*multi", tolower(contentTypes))

hasAttach

# find boundary string

boundaries = getBoundary(contentTypes[ hasAttach ])

boundaries

boundary = boundaries[9]

body = sampleSplit[[15]]$body

bString = paste("--", boundary, sep = "")

bStringLocs = which(bString == body)

bStringLocs

eString = paste("--", boundary, "--", sep = "")

eStringLoc = which(eString == body)

eStringLoc

diff(c(bStringLocs[-1], eStringLoc))

### This code has mistakes in it - and we fix them later!

processAttach = function(body, contentType){

boundary = getBoundary(contentType)

bString = paste("--", boundary, "$", sep = "")

bStringLocs = grep(bString, body)

eString = paste("--", boundary, "--$", sep = "")

eStringLoc = grep(eString, body)

n = length(body)

if (length(eStringLoc) == 0) eStringLoc = n + 1

if (length(bStringLocs) == 1) attachLocs = NULL

else attachLocs = c(bStringLocs[-1], eStringLoc)

msg = body[ (bStringLocs[1] + 1) : min(n, (bStringLocs[2] - 1),

na.rm = TRUE)]

if ( eStringLoc < n )

msg = c(msg, body[ (eStringLoc + 1) : n ])

if ( !is.null(attachLocs) ) {

attachLens = diff(attachLocs, lag = 1)

attachTypes = mapply(function(begL, endL) {

contentTypeLoc = grep("[Cc]ontent-[Tt]ype", body[ (begL + 1) : (endL - 1)])

contentType = body[ begL + contentTypeLoc]

contentType = gsub('"', "", contentType )

MIMEType = sub(" \*Content-Type: \*([^;]\*);?.\*", "\\1", contentType)

return(MIMEType)

}, attachLocs[-length(attachLocs)], attachLocs[-1])

}

if (is.null(attachLocs)) return(list(body = msg, attachInfo = NULL) )

else return(list(body = msg,

attachDF = data.frame(aLen = attachLens,

aType = attachTypes,

stringsAsFactors = FALSE)))

}

bodyList = lapply(sampleSplit, function(msg) msg$body)

attList = mapply(processAttach, bodyList[hasAttach],

contentTypes[hasAttach],

SIMPLIFY = FALSE)

lens = sapply(attList, function(processedA)

processedA$attachDF$aLen)

head(lens)

attList[[2]]$attachDF

body = bodyList[hasAttach][[2]]

length(body)

body[35:45]

processAttach = function(body, contentType){

n = length(body)

boundary = getBoundary(contentType)

bString = paste("--", boundary, sep = "")

bStringLocs = which(bString == body)

eString = paste("--", boundary, "--", sep = "")

eStringLoc = which(eString == body)

if (length(eStringLoc) == 0) eStringLoc = n

if (length(bStringLocs) <= 1) {

attachLocs = NULL

msgLastLine = n

if (length(bStringLocs) == 0) bStringLocs = 0

} else {

attachLocs = c(bStringLocs[ -1 ], eStringLoc)

msgLastLine = bStringLocs[2] - 1

}

msg = body[ (bStringLocs[1] + 1) : msgLastLine]

if ( eStringLoc < n )

msg = c(msg, body[ (eStringLoc + 1) : n ])

if ( !is.null(attachLocs) ) {

attachLens = diff(attachLocs, lag = 1)

attachTypes = mapply(function(begL, endL) {

CTloc = grep("^[Cc]ontent-[Tt]ype", body[ (begL + 1) : (endL - 1)])

if ( length(CTloc) == 0 ) {

MIMEType = NA

} else {

CTval = body[ begL + CTloc[1] ]

CTval = gsub('"', "", CTval )

MIMEType = sub(" \*[Cc]ontent-[Tt]ype: \*([^;]\*);?.\*", "\\1", CTval)

}

return(MIMEType)

}, attachLocs[-length(attachLocs)], attachLocs[-1])

}

if (is.null(attachLocs)) return(list(body = msg, attachDF = NULL) )

return(list(body = msg,

attachDF = data.frame(aLen = attachLens,

aType = unlist(attachTypes),

stringsAsFactors = FALSE)))

}

readEmail = function(dirName) {

# retrieve the names of files in directory

fileNames = list.files(dirName, full.names = TRUE)

# drop files that are not email

notEmail = grep("cmds$", fileNames)

if ( length(notEmail) > 0) fileNames = fileNames[ - notEmail ]

# read all files in the directory

lapply(fileNames, readLines, encoding = "latin1")

}

processAllEmail = function(dirName, isSpam = FALSE)

{

# read all files in the directory

messages = readEmail(dirName)

fileNames = names(messages)

n = length(messages)

# split header from body

eSplit = lapply(messages, splitMessage)

rm(messages)

# process header as named character vector

headerList = lapply(eSplit, function(msg)

processHeader(msg$header))

# extract content-type key

contentTypes = sapply(headerList, function(header)

header["Content-Type"])

# extract the body

bodyList = lapply(eSplit, function(msg) msg$body)

rm(eSplit)

# which email have attachments

hasAttach = grep("^ \*multi", tolower(contentTypes))

# get summary stats for attachments and the shorter body

attList = mapply(processAttach, bodyList[hasAttach],

contentTypes[hasAttach], SIMPLIFY = FALSE)

bodyList[hasAttach] = lapply(attList, function(attEl)

attEl$body)

attachInfo = vector("list", length = n )

attachInfo[ hasAttach ] = lapply(attList,

function(attEl) attEl$attachDF)

# prepare return structure

emailList = mapply(function(header, body, attach, isSpam) {

list(isSpam = isSpam, header = header,

body = body, attach = attach)

},

headerList, bodyList, attachInfo,

rep(isSpam, n), SIMPLIFY = FALSE )

names(emailList) = fileNames

invisible(emailList)

}

emailStruct = mapply(processAllEmail, fullDirNames,

isSpam = rep( c(FALSE, TRUE), 3:2))

emailStruct = unlist(emailStruct, recursive = FALSE)

sampleStruct = emailStruct[ indx ]

save(emailStruct, file="emailXX.rda")

# 3.9 Deriving Variables from the email Message

header = sampleStruct[[1]]$header

subject = header["Subject"]

els = strsplit(subject, "")

all(els %in% LETTERS)

# create test case that code works as expected

testSubject = c("DEAR MADAME", "WINNER!", "")

els = strsplit(testSubject, "")

sapply(els, function(subject) all(subject %in% LETTERS))

# false: there's a space, false: !, true: char(0) & logical(0)

# need to address issue with blanks and punctuation

gsub("[[:punct:] ]", "", testSubject)

gsub("[^[:alpha:]]", "", testSubject)

# isYelling: logical, is everything in subject line uppercase

isYelling = function(msg) {

if ( "Subject" %in% names(msg$header) ) {

el = gsub("[^[:alpha:]]", "", msg$header["Subject"])

if (nchar(el) > 0)

nchar(gsub("[A-Z]", "", el)) < 1

else

FALSE

} else

NA

}

# perCaps : % of capitalized letters in message bosy

perCaps =

function(msg)

{

body = paste(msg$body, collapse = "")

# Return NA if the body of the message is "empty"

if(length(body) == 0 || nchar(body) == 0) return(NA)

# Eliminate non-alpha characters

body = gsub("[^[:alpha:]]", "", body)

capText = gsub("[^A-Z]", "", body)

100 \* nchar(capText)/nchar(body)

}

sapply(sampleStruct, perCaps)

# funcList : combine the last few functions

funcList = list(

isRe = function(msg) {

"Subject" %in% names(msg$header) &&

length(grep("^[ \t]\*Re:", msg$header[["Subject"]])) > 0

},

numLines = function(msg)

length(msg$body),

isYelling = function(msg) {

if ( "Subject" %in% names(msg$header) ) {

el = gsub("[^[:alpha:]]", "", msg$header["Subject"])

if (nchar(el) > 0)

nchar(gsub("[A-Z]", "", el)) < 1

else

FALSE

}

else NA

},

perCaps = function(msg) {

body = paste(msg$body, collapse = "")

# Return NA if the body of the message is "empty"

if(length(body) == 0 || nchar(body) == 0) return(NA)

# Eliminate non-alpha characters

body = gsub("[^[:alpha:]]", "", body)

capText = gsub("[^A-Z]", "", body)

100 \* nchar(capText)/nchar(body)

}

)

lapply(funcList, function(func)

sapply(sampleStruct, function(msg) func(msg)))

# create derived variables

createDerivedDF =

function(email = emailStruct, operations = funcList,

verbose = FALSE)

{

els = lapply(names(operations),

function(id) {

if(verbose) print(id)

e = operations[[id]]

v = if(is.function(e))

sapply(email, e)

else

sapply(email, function(msg) eval(e))

v

})

df = as.data.frame(els)

names(df) = names(operations)

invisible(df)

}

sampleDF = createDerivedDF(sampleStruct)

head(sampleDF)

# full function

funcList = list(

isSpam =

expression(msg$isSpam)

,

isRe =

function(msg) {

# Can have a Fwd: Re: ... but we are not looking for this here.

# We may want to look at In-Reply-To field.

"Subject" %in% names(msg$header) &&

length(grep("^[ \t]\*Re:", msg$header[["Subject"]])) > 0

}

,

numLines =

function(msg) length(msg$body)

,

bodyCharCt =

function(msg)

sum(nchar(msg$body))

,

underscore =

function(msg) {

if(!"Reply-To" %in% names(msg$header))

return(FALSE)

txt <- msg$header[["Reply-To"]]

length(grep("\_", txt)) > 0 &&

length(grep("[0-9A-Za-z]+", txt)) > 0

}

,

subExcCt =

function(msg) {

x = msg$header["Subject"]

if(length(x) == 0 || sum(nchar(x)) == 0 || is.na(x))

return(NA)

sum(nchar(gsub("[^!]","", x)))

}

,

subQuesCt =

function(msg) {

x = msg$header["Subject"]

if(length(x) == 0 || sum(nchar(x)) == 0 || is.na(x))

return(NA)

sum(nchar(gsub("[^?]","", x)))

}

,

numAtt =

function(msg) {

if (is.null(msg$attach)) return(0)

else nrow(msg$attach)

}

,

priority =

function(msg) {

ans <- FALSE

# Look for names X-Priority, Priority, X-Msmail-Priority

# Look for high any where in the value

ind = grep("priority", tolower(names(msg$header)))

if (length(ind) > 0) {

ans <- length(grep("high", tolower(msg$header[ind]))) >0

}

ans

}

,

numRec =

function(msg) {

# unique or not.

els = getMessageRecipients(msg$header)

if(length(els) == 0)

return(NA)

# Split each line by "," and in each of these elements, look for

# the @ sign. This handles

tmp = sapply(strsplit(els, ","), function(x) grep("@", x))

sum(sapply(tmp, length))

}

,

perCaps =

function(msg)

{

body = paste(msg$body, collapse = "")

# Return NA if the body of the message is "empty"

if(length(body) == 0 || nchar(body) == 0) return(NA)

# Eliminate non-alpha characters and empty lines

body = gsub("[^[:alpha:]]", "", body)

els = unlist(strsplit(body, ""))

ctCap = sum(els %in% LETTERS)

100 \* ctCap / length(els)

}

,

isInReplyTo =

function(msg)

{

"In-Reply-To" %in% names(msg$header)

}

,

sortedRec =

function(msg)

{

ids = getMessageRecipients(msg$header)

all(sort(ids) == ids)

}

,

subPunc =

function(msg)

{

if("Subject" %in% names(msg$header)) {

el = gsub("['/.:@-]", "", msg$header["Subject"])

length(grep("[A-Za-z][[:punct:]]+[A-Za-z]", el)) > 0

}

else

FALSE

},

hour =

function(msg)

{

date = msg$header["Date"]

if ( is.null(date) ) return(NA)

# Need to handle that there may be only one digit in the hour

locate = regexpr("[0-2]?[0-9]:[0-5][0-9]:[0-5][0-9]", date)

if (locate < 0)

locate = regexpr("[0-2]?[0-9]:[0-5][0-9]", date)

if (locate < 0) return(NA)

hour = substring(date, locate, locate+1)

hour = as.numeric(gsub(":", "", hour))

locate = regexpr("PM", date)

if (locate > 0) hour = hour + 12

locate = regexpr("[+-][0-2][0-9]00", date)

if (locate < 0) offset = 0

else offset = as.numeric(substring(date, locate, locate + 2))

(hour - offset) %% 24

}

,

multipartText =

function(msg)

{

if (is.null(msg$attach)) return(FALSE)

numAtt = nrow(msg$attach)

types =

length(grep("(html|plain|text)", msg$attach$aType)) > (numAtt/2)

}

,

hasImages =

function(msg)

{

if (is.null(msg$attach)) return(FALSE)

length(grep("^ \*image", tolower(msg$attach$aType))) > 0

}

,

isPGPsigned =

function(msg)

{

if (is.null(msg$attach)) return(FALSE)

length(grep("pgp", tolower(msg$attach$aType))) > 0

},

perHTML =

function(msg)

{

if(! ("Content-Type" %in% names(msg$header))) return(0)

el = tolower(msg$header["Content-Type"])

if (length(grep("html", el)) == 0) return(0)

els = gsub("[[:space:]]", "", msg$body)

totchar = sum(nchar(els))

totplain = sum(nchar(gsub("<[^<]+>", "", els )))

100 \* (totchar - totplain)/totchar

},

subSpamWords =

function(msg)

{

if("Subject" %in% names(msg$header))

length(grep(paste(SpamCheckWords, collapse = "|"),

tolower(msg$header["Subject"]))) > 0

else

NA

}

,

subBlanks =

function(msg)

{

if("Subject" %in% names(msg$header)) {

x = msg$header["Subject"]

# should we count blank subject line as 0 or 1 or NA?

if (nchar(x) == 1) return(0)

else 100 \*(1 - (nchar(gsub("[[:blank:]]", "", x))/nchar(x)))

} else NA

}

,

noHost =

function(msg)

{

# Or use partial matching.

idx = pmatch("Message-", names(msg$header))

if(is.na(idx)) return(NA)

tmp = msg$header[idx]

return(length(grep(".\*@[^[:space:]]+", tmp)) == 0)

}

,

numEnd =

function(msg)

{

# If we just do a grep("[0-9]@", )

# we get matches on messages that have a From something like

# " \"marty66@aol.com\" <synjan@ecis.com>"

# and the marty66 is the "user's name" not the login

# So we can be more precise if we want.

x = names(msg$header)

if ( !( "From" %in% x) ) return(NA)

login = gsub("^.\*<", "", msg$header["From"])

if ( is.null(login) )

login = gsub("^.\*<", "", msg$header["X-From"])

if ( is.null(login) ) return(NA)

login = strsplit(login, "@")[[1]][1]

length(grep("[0-9]+$", login)) > 0

},

isYelling =

function(msg)

{

if ( "Subject" %in% names(msg$header) ) {

el = gsub("[^[:alpha:]]", "", msg$header["Subject"])

if (nchar(el) > 0) nchar(gsub("[A-Z]", "", el)) < 1

else FALSE

}

else

NA

},

forwards =

function(msg)

{

x = msg$body

if(length(x) == 0 || sum(nchar(x)) == 0)

return(NA)

ans = length(grep("^[[:space:]]\*>", x))

100 \* ans / length(x)

},

isOrigMsg =

function(msg)

{

x = msg$body

if(length(x) == 0) return(NA)

length(grep("^[^[:alpha:]]\*original[^[:alpha:]]+message[^[:alpha:]]\*$",

tolower(x) ) ) > 0

},

isDear =

function(msg)

{

x = msg$body

if(length(x) == 0) return(NA)

length(grep("^[[:blank:]]\*dear +(sir|madam)\\>",

tolower(x))) > 0

},

isWrote =

function(msg)

{

x = msg$body

if(length(x) == 0) return(NA)

length(grep("(wrote|schrieb|ecrit|escribe):", tolower(x) )) > 0

},

avgWordLen =

function(msg)

{

txt = paste(msg$body, collapse = " ")

if(length(txt) == 0 || sum(nchar(txt)) == 0) return(0)

txt = gsub("[^[:alpha:]]", " ", txt)

words = unlist(strsplit(txt, "[[:blank:]]+"))

wordLens = nchar(words)

mean(wordLens[ wordLens > 0 ])

}

,

numDlr =

function(msg)

{

x = paste(msg$body, collapse = "")

if(length(x) == 0 || sum(nchar(x)) == 0)

return(NA)

nchar(gsub("[^$]","", x))

}

)

# spam checkwords

SpamCheckWords =

c("viagra", "pounds", "free", "weight", "guarantee", "million",

"dollars", "credit", "risk", "prescription", "generic", "drug",

"financial", "save", "dollar", "erotic", "million", "barrister",

"beneficiary", "easy",

"money back", "money", "credit card")

getMessageRecipients =

function(header)

{

c(if("To" %in% names(header)) header[["To"]] else character(0),

if("Cc" %in% names(header)) header[["Cc"]] else character(0),

if("Bcc" %in% names(header)) header[["Bcc"]] else character(0)

)

}

emailDF = createDerivedDF(emailStruct)

dim(emailDF)

# save(emailDF, file = "spamAssassinDerivedDF.rda")

# 3.9.1 Checking our code for errors

# load("Data/spamAssassinDerivedDF.rda")

dim(emailDF)

#perCaps2

perCaps2 =

function(msg)

{

body = paste(msg$body, collapse = "")

# Return NA if the body of the message is "empty"

if(length(body) == 0 || nchar(body) == 0) return(NA)

# Eliminate non-alpha characters and empty lines

body = gsub("[^[:alpha:]]", "", body)

els = unlist(strsplit(body, ""))

ctCap = sum(els %in% LETTERS)

100 \* ctCap / length(els)

}

pC = sapply(emailStruct, perCaps)

pC2 = sapply(emailStruct, perCaps2)

identical(pC, pC2)

indNA = which(is.na(emailDF$subExcCt))

indNoSubject = which(sapply(emailStruct,

function(msg)

!("Subject" %in% names(msg$header))))

all(indNA == indNoSubject)

# indNA==indNoSubject is FALSE need to check into. could be the record Prof talked about

all(emailDF$bodyCharCt > emailDF$numLines)

# a boxplot

x.at = c(1,10,100,1000,10000,100000)

y.at = c(1, 5, 10, 50, 100, 500, 5000)

nL = 1 + emailDF$numLines

nC = 1 + emailDF$bodyCharCt

pdf("ScatterPlotNumLinesNumChars.pdf", width = 6, height = 4.5)

plot(nL ~ nC, log = "xy", pch=".", xlim=c(1,100000), axes = TRUE,

xlab = "Number of Characters", ylab = "Number of Lines")

# box() not sure what this code did but prevent graphs from running from here on

axis(1, at = x.at, labels = formatC(x.at, digits = 0, format="d"))

axis(2, at = y.at, labels = formatC(y.at, digits = 0, format="d"))

abline(a=0, b=1, col="red", lwd = 2)

dev.off()

# 3.10 Exploring the email feature set

percent = emailDF$perCaps

isSpamLabs = factor(emailDF$isSpam, labels = c("ham", "spam"))

boxplot(log(1 + percent) ~ isSpamLabs,

ylab = "Percent Capitals (log)")

# QQ plot

logPerCapsSpam = log(1 + emailDF$perCaps[ emailDF$isSpam ])

logPerCapsHam = log(1 + emailDF$perCaps[ !emailDF$isSpam ])

qqplot(logPerCapsSpam, logPerCapsHam,

xlab = "Regular Email", ylab = "Spam Email",

main = "Percentage of Capital Letters (log scale)",

pch = 19, cex = 0.3)

#

colI = c("#4DAF4A80", "#984EA380")

logBodyCharCt = log(1 + emailDF$bodyCharCt)

logPerCaps = log(1 + emailDF$perCaps)

plot(logPerCaps ~ logBodyCharCt, xlab = "Total Characters (log)",

ylab = "Percent Capitals (log)",

col = colI[1 + emailDF$isSpam],

xlim = c(2,12), pch = 19, cex = 0.5)

table(emailDF$numAtt, isSpamLabs)

# does subject line have RE: in header

oldPar = par(mfrow = c(1, 2), mar = c(1,1,1,1))

colM = c("#E41A1C80", "#377EB880")

isRe = factor(emailDF$isRe, labels = c("no Re:", "Re:"))

mosaicplot(table(isSpamLabs, isRe), main = "",

xlab = "", ylab = "", color = colM)

fromNE = factor(emailDF$numEnd, labels = c("No #", "#"))

mosaicplot(table(isSpamLabs, fromNE), color = colM,

main = "", xlab="", ylab = "")

par(oldPar)

# 3.11 Fitting the rpart() Model to the email Data

# Where code for Q.19 begins

library(rpart)

# properly format data

setupRpart = function(data) {

logicalVars = which(sapply(data, is.logical))

facVars = lapply(data[ , logicalVars],

function(x) {

x = as.factor(x)

levels(x) = c("F", "T")

x

})

cbind(facVars, data[ , - logicalVars])

}

emailDFrp = setupRpart(emailDF)

# split into training and testing sets

set.seed(418910)

testSpamIdx = sample(numSpam, size = floor(numSpam/3))

testHamIdx = sample(numHam, size = floor(numHam/3))

testDF =

rbind( emailDFrp[ emailDFrp$isSpam == "T", ][testSpamIdx, ],

emailDFrp[emailDFrp$isSpam == "F", ][testHamIdx, ] )

trainDF =

rbind( emailDFrp[emailDFrp$isSpam == "T", ][-testSpamIdx, ],

emailDFrp[emailDFrp$isSpam == "F", ][-testHamIdx, ])

# fit the classification tree

rpartFit = rpart(isSpam ~ ., data = trainDF, method = "class")

library(rpart.plot)

prp(rpartFit, extra = 1)

predictions = predict(rpartFit,

newdata = testDF[, names(testDF) != "isSpam"],

type = "class")

# find out how well our tree has performed

predsForHam = predictions[ testDF$isSpam == "F" ]

summary(predsForHam)

sum(predsForHam == "T") / length(predsForHam)

# Type II Error rate

predsForSpam = predictions[ testDF$isSpam == "T" ]

sum(predsForSpam == "F") / length(predsForSpam)

# Explore complexity parameter

complexityVals = c(seq(0.00001, 0.0001, length=19),

seq(0.0001, 0.001, length=19),

seq(0.001, 0.005, length=9),

seq(0.005, 0.01, length=9))

fits = lapply(complexityVals, function(x) {

rpartObj = rpart(isSpam ~ ., data = trainDF,

method="class",

control = rpart.control(cp=x) )

predict(rpartObj,

newdata = testDF[ , names(testDF) != "isSpam"],

type = "class")

})

# Assess Type I & II errors for these fitted models applied to our test data

spam = testDF$isSpam == "T"

numSpam = sum(spam)

numHam = sum(!spam)

errs = sapply(fits, function(preds) {

typeI = sum(preds[ !spam ] == "T") / numHam

typeII = sum(preds[ spam ] == "F") / numSpam

c(typeI = typeI, typeII = typeII)

})

# Develop a better feature est for predicting spam using rpart

library(RColorBrewer)

cols = brewer.pal(9, "Set1")[c(3, 4, 5)]

plot(errs[1,] ~ complexityVals, type="l", col=cols[2],

lwd = 2, ylim = c(0,0.2), xlim = c(0,0.005),

ylab="Error", xlab="complexity parameter values")

points(errs[2,] ~ complexityVals, type="l", col=cols[1], lwd = 2)

text(x =c(0.003, 0.0035), y = c(0.12, 0.05),

labels=c("Type II Error", "Type I Error"))

minI = which(errs[1,] == min(errs[1,]))[1]

abline(v = complexityVals[minI], col ="grey", lty =3, lwd=2)

text(0.0007, errs[1, minI]+0.01,

formatC(errs[1, minI], digits = 2))

text(0.0007, errs[2, minI]+0.01,

formatC(errs[2, minI], digits = 3))

# create tree

rpartObjcp0001 = rpart(isSpam ~ ., data = trainDF,

method="class",

control = rpart.control(cp=0.0001) )

prp(rpartObjcp0001, extra = 1)

printcp(rpartObjcp0001)

# create tree

rpartObjcp01 = rpart(isSpam ~ ., data = trainDF,

method="class",

control = rpart.control(cp=0.01) )

prp(rpartObjcp01, extra = 1)

printcp(rpartObjcp01)

################################

## Explore minsplit parameter ##

################################

# fit the classification tree

rpartFit = rpart(isSpam ~ ., data = trainDF, method = "class")

library(rpart.plot)

prp(rpartFit, extra = 1)

predictions = predict(rpartFit,

newdata = testDF[, names(testDF) != "isSpam"],

type = "class")

# find out how well our tree has performed

predsForHam = predictions[ testDF$isSpam == "F" ]

summary(predsForHam)

sum(predsForHam == "T") / length(predsForHam)

# Type II Error rate

predsForSpam = predictions[ testDF$isSpam == "T" ]

sum(predsForSpam == "F") / length(predsForSpam)

minSplitVals = c(seq(1, 100, length=19),

seq(100, 500, length=19),

seq(500, 1000, length=9),

seq(1000, 5000, length=9))

min\_split\_fits = lapply(minSplitVals, function(x) {

rpartObj\_minsplit = rpart(isSpam ~ ., data = trainDF,

method="class",

control = rpart.control(minsplit = x) )

predict(rpartObj\_minsplit,

newdata = testDF[ , names(testDF) != "isSpam"],

type = "class")

})

# Assess Type I & II errors for these fitted models applied to our test data

# minsplit edition

spam = testDF$isSpam == "T"

numSpam = sum(spam)

numHam = sum(!spam)

errs\_minsplit = sapply(min\_split\_fits, function(preds) {

typeI = sum(preds[ !spam ] == "T") / numHam

typeII = sum(preds[ spam ] == "F") / numSpam

c(typeI = typeI, typeII = typeII)

})

# Develop a better feature est for predicting spam using rpart

library(RColorBrewer)

cols = brewer.pal(9, "Set1")[c(3, 4, 5)]

plot(errs\_minsplit[1,] ~ minSplitVals, type="l", col=cols[2],

lwd = 2, ylim = c(0,.5), xlim = c(0,6000),

ylab="Error", xlab="minsplit values")

points(errs\_minsplit[2,] ~ minSplitVals, type="l", col=cols[1], lwd = 2)

text(x =c(0.003, 0.0035), y = c(0.12, 0.05),

labels=c("Type II Error", "Type I Error"))

minI = which(errs\_minsplit[1,] == min(errs\_minsplit[1,]))[1]

abline(v = minSplitVals[minI], col ="grey", lty =3, lwd=2)

text(0.0007, errs\_minsplit[1, minI]+0.01,

formatC(errs\_minsplit[1, minI], digits = 2))

text(0.0007, errs\_minsplit[2, minI]+0.01,

formatC(errs\_minsplit[2, minI], digits = 3))

################################

## Explore minsplit parameter ## Part II

################################

# fit the classification tree

rpartFit = rpart(isSpam ~ ., data = trainDF, method = "class")

library(rpart.plot)

prp(rpartFit, extra = 1)

predictions = predict(rpartFit,

newdata = testDF[, names(testDF) != "isSpam"],

type = "class")

# find out how well our tree has performed

predsForHam = predictions[ testDF$isSpam == "F" ]

summary(predsForHam)

sum(predsForHam == "T") / length(predsForHam)

# Type II Error rate

predsForSpam = predictions[ testDF$isSpam == "T" ]

sum(predsForSpam == "F") / length(predsForSpam)

minSplitVals = c(seq(1, 50, length=19),

seq(50, 100, length=19),

seq(100, 150, length=9),

seq(150, 200, length=9))

min\_split\_fits = lapply(minSplitVals, function(x) {

rpartObj\_minsplit = rpart(isSpam ~ ., data = trainDF,

method="class",

control = rpart.control(minsplit = x) )

predict(rpartObj\_minsplit,

newdata = testDF[ , names(testDF) != "isSpam"],

type = "class")

})

# Assess Type I & II errors for these fitted models applied to our test data

# minsplit edition

spam = testDF$isSpam == "T"

numSpam = sum(spam)

numHam = sum(!spam)

errs\_minsplit = sapply(min\_split\_fits, function(preds) {

typeI = sum(preds[ !spam ] == "T") / numHam

typeII = sum(preds[ spam ] == "F") / numSpam

c(typeI = typeI, typeII = typeII)

})

# Develop a better feature est for predicting spam using rpart

# library(RColorBrewer)

cols = brewer.pal(9, "Set1")[c(3, 4, 5)]

plot(errs\_minsplit[1,] ~ minSplitVals, type="l", col=cols[2],

lwd = 2, ylim = c(0,.3), xlim = c(0,200),

ylab="Error", xlab="minsplit values")

points(errs\_minsplit[2,] ~ minSplitVals, type="l", col=cols[1], lwd = 2)

text(x =c(0.003, 0.0035), y = c(0.12, 0.05),

labels=c("Type II Error", "Type I Error"))

minI = which(errs\_minsplit[1,] == min(errs\_minsplit[1,]))[1]

abline(v = minSplitVals[minI], col ="grey", lty =3, lwd=2)

text(0.0007, errs\_minsplit[1, minI]+0.01,

formatC(errs\_minsplit[1, minI], digits = 2))

text(0.0007, errs\_minsplit[2, minI]+0.01,

formatC(errs\_minsplit[2, minI], digits = 3))

################################

## Explore minsplit parameter ## Part III

################################

# fit the classification tree

rpartFit = rpart(isSpam ~ ., data = trainDF, method = "class")

library(rpart.plot)

prp(rpartFit, extra = 1)

predictions = predict(rpartFit,

newdata = testDF[, names(testDF) != "isSpam"],

type = "class")

# find out how well our tree has performed

predsForHam = predictions[ testDF$isSpam == "F" ]

summary(predsForHam)

sum(predsForHam == "T") / length(predsForHam)

# Type II Error rate

predsForSpam = predictions[ testDF$isSpam == "T" ]

sum(predsForSpam == "F") / length(predsForSpam)

minSplitVals = c(seq(200, 400, length=19),

seq(400, 600, length=19),

seq(600, 800, length=9),

seq(800, 1000, length=9))

min\_split\_fits = lapply(minSplitVals, function(x) {

rpartObj\_minsplit = rpart(isSpam ~ ., data = trainDF,

method="class",

control = rpart.control(minsplit = x) )

predict(rpartObj\_minsplit,

newdata = testDF[ , names(testDF) != "isSpam"],

type = "class")

})

# Assess Type I & II errors for these fitted models applied to our test data

# minsplit edition

spam = testDF$isSpam == "T"

numSpam = sum(spam)

numHam = sum(!spam)

errs\_minsplit = sapply(min\_split\_fits, function(preds) {

typeI = sum(preds[ !spam ] == "T") / numHam

typeII = sum(preds[ spam ] == "F") / numSpam

c(typeI = typeI, typeII = typeII)

})

# Develop a better feature est for predicting spam using rpart

# library(RColorBrewer)

cols = brewer.pal(9, "Set1")[c(3, 4, 5)]

plot(errs\_minsplit[1,] ~ minSplitVals, type="l", col=cols[2],

lwd = 2, ylim = c(0,.5), xlim = c(0,1000),

ylab="Error", xlab="minsplit values")

points(errs\_minsplit[2,] ~ minSplitVals, type="l", col=cols[1], lwd = 2)

text(x =c(0.003, 0.0035), y = c(0.12, 0.05),

labels=c("Type II Error", "Type I Error"))

minI = which(errs\_minsplit[1,] == min(errs\_minsplit[1,]))[1]

abline(v = minSplitVals[minI], col ="grey", lty =3, lwd=2)

text(0.0007, errs\_minsplit[1, minI]+0.01,

formatC(errs\_minsplit[1, minI], digits = 2))

text(0.0007, errs\_minsplit[2, minI]+0.01,

formatC(errs\_minsplit[2, minI], digits = 3))

# look at tree for minsplit

rpartObj\_minsplit10 = rpart(isSpam ~ ., data = trainDF,

method="class",

control = rpart.control(minsplit = 10) )

prp(rpartObj\_minsplit10, extra = 1)

printcp(rpartObj\_minsplit10)

# minsplit=350

rpartObj\_minsplit350 = rpart(isSpam ~ ., data = trainDF,

method="class",

control = rpart.control(minsplit = 350) )

prp(rpartObj\_minsplit350, extra = 1)

printcp(rpartObj\_minsplit350)

# minsplit=5000

rpartObj\_minsplit5000 = rpart(isSpam ~ ., data = trainDF,

method="class",

control = rpart.control(minsplit = 5000) )

prp(rpartObj\_minsplit5000, extra = 1)

printcp(rpartObj\_minsplit5000)

#######################

###### maxdepth #######

#######################

# fit the classification tree

rpartFit = rpart(isSpam ~ ., data = trainDF, method = "class")

library(rpart.plot)

prp(rpartFit, extra = 1)

predictions = predict(rpartFit,

newdata = testDF[, names(testDF) != "isSpam"],

type = "class")

# find out how well our tree has performed

predsForHam = predictions[ testDF$isSpam == "F" ]

summary(predsForHam)

sum(predsForHam == "T") / length(predsForHam)

# Type II Error rate

predsForSpam = predictions[ testDF$isSpam == "T" ]

sum(predsForSpam == "F") / length(predsForSpam)

# Explore complexity parameter

maxdepthVals = c(seq(1, 5, length=19),

seq(5, 10, length=19),

seq(10, 20, length=9),

seq(20, 30, length=9))

fits\_maxdepth = lapply(maxdepthVals, function(x) {

rpartObj = rpart(isSpam ~ ., data = trainDF,

method="class",

control = rpart.control(maxdepth = x) )

predict(rpartObj,

newdata = testDF[ , names(testDF) != "isSpam"],

type = "class")

})

# Assess Type I & II errors for these fitted models applied to our test data

spam = testDF$isSpam == "T"

numSpam = sum(spam)

numHam = sum(!spam)

errs\_maxdepth = sapply(fits\_maxdepth, function(preds) {

typeI = sum(preds[ !spam ] == "T") / numHam

typeII = sum(preds[ spam ] == "F") / numSpam

c(typeI = typeI, typeII = typeII)

})

# Develop a better feature est for predicting spam using rpart

library(RColorBrewer)

cols = brewer.pal(9, "Set1")[c(3, 4, 5)]

plot(errs\_maxdepth[1,] ~ maxdepthVals, type="l", col=cols[2],

lwd = 2, ylim = c(0,.5), xlim = c(0,30),

ylab="Error", xlab="maxdepth values")

points(errs\_maxdepth[2,] ~ maxdepthVals, type="l", col=cols[1], lwd = 2)

text(x =c(0.003, 0.0035), y = c(0.12, 0.05),

labels=c("Type II Error", "Type I Error"))

minI = which(errs\_maxdepth[1,] == min(errs\_maxdepth[1,]))[1]

abline(v = maxdepthVals[minI], col ="grey", lty =3, lwd=2)

text(0.0007, errs\_maxdepth[1, minI]+0.01,

formatC(errs\_maxdepth[1, minI], digits = 2))

text(0.0007, errs\_maxdepth[2, minI]+0.01,

formatC(errs\_maxdepth[2, minI], digits = 3))

# look at tree for maxdepth =

rpartObj\_maxdepth4 = rpart(isSpam ~ ., data = trainDF,

method="class",

control = rpart.control(maxdepth = 4) )

prp(rpartObj\_maxdepth4, extra = 1)

printcp(rpartObj\_maxdepth4)

rpartObj\_maxdepth7 = rpart(isSpam ~ ., data = trainDF,

method="class",

control = rpart.control(maxdepth = 7) )

prp(rpartObj\_maxdepth7, extra = 1)

printcp(rpartObj\_maxdepth7)

#########################

####### maxcompete ######

#########################

# fit the classification tree

rpartFit = rpart(isSpam ~ ., data = trainDF, method = "class")

library(rpart.plot)

prp(rpartFit, extra = 1)

predictions = predict(rpartFit,

newdata = testDF[, names(testDF) != "isSpam"],

type = "class")

# find out how well our tree has performed

predsForHam = predictions[ testDF$isSpam == "F" ]

summary(predsForHam)

sum(predsForHam == "T") / length(predsForHam)

# Type II Error rate

predsForSpam = predictions[ testDF$isSpam == "T" ]

sum(predsForSpam == "F") / length(predsForSpam)

# Explore complexity parameter

maxcompeteVals = c(seq(1, 20, length=19),

seq(20, 80, length=19),

seq(80, 200, length=9),

seq(200, 400, length=9))

fits\_maxcompete = lapply(maxcompeteVals, function(x) {

rpartObj = rpart(isSpam ~ ., data = trainDF,

method="class",

control = rpart.control(maxcompete = x) )

predict(rpartObj,

newdata = testDF[ , names(testDF) != "isSpam"],

type = "class")

})

# Assess Type I & II errors for these fitted models applied to our test data

spam = testDF$isSpam == "T"

numSpam = sum(spam)

numHam = sum(!spam)

errs\_maxcompete = sapply(fits\_maxcompete, function(preds) {

typeI = sum(preds[ !spam ] == "T") / numHam

typeII = sum(preds[ spam ] == "F") / numSpam

c(typeI = typeI, typeII = typeII)

})

# Develop a better feature est for predicting spam using rpart

library(RColorBrewer)

cols = brewer.pal(9, "Set1")[c(3, 4, 5)]

plot(errs\_maxcompete[1,] ~ maxcompeteVals, type="l", col=cols[2],

lwd = 2, ylim = c(0,.5), xlim = c(0,30),

ylab="Error", xlab="maxcompete values")

points(errs\_maxcompete[2,] ~ maxcompeteVals, type="l", col=cols[1], lwd = 2)

text(x =c(0.003, 0.0035), y = c(0.12, 0.05),

labels=c("Type II Error", "Type I Error"))

minI = which(errs\_maxcompete[1,] == min(errs\_maxcompete[1,]))[1]

abline(v = maxcompeteVals[minI], col ="grey", lty =3, lwd=2)

text(0.0007, errs\_maxcompete[1, minI]+0.01,

formatC(errs\_maxcompete[1, minI], digits = 2))

text(0.0007, errs\_maxcompete[2, minI]+0.01,

formatC(errs\_maxcompete[2, minI], digits = 3))

# create tree

rpartObj\_maxcompete = rpart(isSpam ~ ., data = trainDF,

method="class",

control = rpart.control(maxcompete = 10) )

prp(rpartObj\_maxcompete, extra = 1)

printcp(rpartObj\_maxcompete)

#########################

###### maxsurrogate #####

#########################

# fit the classification tree

rpartFit = rpart(isSpam ~ ., data = trainDF, method = "class")

library(rpart.plot)

prp(rpartFit, extra = 1)

predictions = predict(rpartFit,

newdata = testDF[, names(testDF) != "isSpam"],

type = "class")

# find out how well our tree has performed

predsForHam = predictions[ testDF$isSpam == "F" ]

summary(predsForHam)

sum(predsForHam == "T") / length(predsForHam)

# Type II Error rate

predsForSpam = predictions[ testDF$isSpam == "T" ]

sum(predsForSpam == "F") / length(predsForSpam)

# Explore complexity parameter

maxsurrogateVals = c(seq(1, 20, length=19),

seq(20, 100, length=19),

seq(100, 500, length=9),

seq(500, 1000, length=9))

fits\_maxsurrogate = lapply(maxsurrogateVals, function(x) {

rpartObj = rpart(isSpam ~ ., data = trainDF,

method="class",

control = rpart.control(maxsurrogate = x) )

predict(rpartObj,

newdata = testDF[ , names(testDF) != "isSpam"],

type = "class")

})

# Assess Type I & II errors for these fitted models applied to our test data

spam = testDF$isSpam == "T"

numSpam = sum(spam)

numHam = sum(!spam)

errs\_maxsurrogate = sapply(fits\_maxsurrogate, function(preds) {

typeI = sum(preds[ !spam ] == "T") / numHam

typeII = sum(preds[ spam ] == "F") / numSpam

c(typeI = typeI, typeII = typeII)

})

# Develop a better feature est for predicting spam using rpart

library(RColorBrewer)

cols = brewer.pal(9, "Set1")[c(3, 4, 5)]

plot(errs\_maxsurrogate[1,] ~ maxsurrogateVals, type="l", col=cols[2],

lwd = 2, ylim = c(0,.5), xlim = c(0,1000),

ylab="Error", xlab="maxsurrogate values")

points(errs\_maxsurrogate[2,] ~ maxsurrogateVals, type="l", col=cols[1], lwd = 2)

text(x =c(0.003, 0.0035), y = c(0.12, 0.05),

labels=c("Type II Error", "Type I Error"))

minI = which(errs\_maxsurrogate[1,] == min(errs\_maxsurrogate[1,]))[1]

abline(v = maxsurrogateVals[minI], col ="grey", lty =3, lwd=2)

text(0.0007, errs\_maxsurrogate[1, minI]+0.01,

formatC(errs\_maxsurrogate[1, minI], digits = 2))

text(0.0007, errs\_maxsurrogate[2, minI]+0.01,

formatC(errs\_maxsurrogate[2, minI], digits = 3))

#########################

########## xval #########

#########################

# fit the classification tree

rpartFit = rpart(isSpam ~ ., data = trainDF, method = "class")

library(rpart.plot)

prp(rpartFit, extra = 1)

predictions = predict(rpartFit,

newdata = testDF[, names(testDF) != "isSpam"],

type = "class")

# find out how well our tree has performed

predsForHam = predictions[ testDF$isSpam == "F" ]

summary(predsForHam)

sum(predsForHam == "T") / length(predsForHam)

# Type II Error rate

predsForSpam = predictions[ testDF$isSpam == "T" ]

sum(predsForSpam == "F") / length(predsForSpam)

# Explore complexity parameter

xVals = c(seq(0, 5, length=19),

seq(5, 10, length=19),

seq(10, 15, length=9),

seq(15, 20, length=9))

fits\_xval = lapply(xVals, function(x) {

rpartObj = rpart(isSpam ~ ., data = trainDF,

method="class",

control = rpart.control(xval = x) )

predict(rpartObj,

newdata = testDF[ , names(testDF) != "isSpam"],

type = "class")

})

# Assess Type I & II errors for these fitted models applied to our test data

spam = testDF$isSpam == "T"

numSpam = sum(spam)

numHam = sum(!spam)

errs\_xval = sapply(fits\_xval, function(preds) {

typeI = sum(preds[ !spam ] == "T") / numHam

typeII = sum(preds[ spam ] == "F") / numSpam

c(typeI = typeI, typeII = typeII)

})

# Develop a better feature est for predicting spam using rpart

library(RColorBrewer)

cols = brewer.pal(9, "Set1")[c(3, 4, 5)]

plot(errs\_xval[1,] ~ xVals, type="l", col=cols[2],

lwd = 2, ylim = c(0,.5), xlim = c(0,20),

ylab="Error", xlab="cross validation values")

points(errs\_xval[2,] ~ xVals, type="l", col=cols[1], lwd = 2)

text(x =c(0.003, 0.0035), y = c(0.12, 0.05),

labels=c("Type II Error", "Type I Error"))

minI = which(errs\_xval[1,] == min(errs\_xval[1,]))[1]

abline(v = xVals[minI], col ="grey", lty =3, lwd=2)

text(0.0007, errs\_xval[1, minI]+0.01,

formatC(errs\_xval[1, minI], digits = 2))

text(0.0007, errs\_xval[2, minI]+0.01,

formatC(errs\_xval[2, minI], digits = 3))

# look at tree for xval=5

rpartObj\_xval5 = rpart(isSpam ~ ., data = trainDF,

+ method="class",

+ control = rpart.control(xval = 5) )

prp(rpartObj\_xval5, extra = 1)

printcp(rpartObj\_xval5)