## **Introduction**

Since the early 2000s, spam has accounted for a significant proportion of email traffic, with most figures online showing between 45-55% of the traffic in 2018 could be classified as spam [1, 2]. Spam is generally defined as unsolicited bulk emails. The intended purpose of these messages varies from simple promotional material to scams designed to extract money from unwitting targets. Given the volume of spam being sent every day, automated means of identification are crucial to help reduce the burden on the recipients.

In this paper, we review a methodology for identifying these unsolicited messages from a corpus of labeled data using recursive partitioning. Specifically, we will focus on parameter tuning of one of these recursive partitioned trees to improve the overall accuracy. This study is based on the exercise 19 from section 3.12 from the Nolan and Lang text:

*Consider the other parameters that can be used to control the recursive partitioning process. Read the documentation for them in the* rpart.control() *documentation. Also, carry out an Internet search for more information on how to tweak the* rpart()*tuning parameters. Experiment with values for these parameters. Do the trees that result make sense with your understanding of how the parameters are used? Can you improve the prediction using them?* [3]

Exploring various parameters of this algorithm, we were able to improve the accuracy of our models against the author’s attempts.

**Background**

For this study we are working with a corpus of 9000 emails messages that have been labeled by SpamAssassin, an Apache project. This corpus was collected and labeled for developing and testing spam filtering techniques. From this data, we manually derived a handful of features we consider to be helpful for the task of classifying spam. The features are derived from the header (subject, to/from), body, and any attachments in the email. The features considered and their definitions are shown in Table 1 below.

For training and testing our model we are using the R package rpart, which was developed based on research conducted by Breiman, Friedman, Olshen and Stone in 1984 [4]. The rpart package allows the customization of the parameters used when fitting the model. Our exercise will focus on the tuning of several of these parameters.

**Table 1:** Variables derived from the corpus for use in our model.

|  |  |
| --- | --- |
| **Variable** | **Definition** |
| **isRe** | TRUE if the message is a reply (contains “RE:” at the start of the message) |
| **numLines** | Count number of lines in the email body. |
| **bodyCharCt** | Number of characters in the body of the email |
| **underscore** | TRUE if from address contains and underscore |
| **subExcCt** | Number of exclamation marks in the subject. |
| **subQuesCt** | Number of question marks in the subject. |
| **numAtt** | Number of attachments |
| **priority** | TRUE if email is marked as high priority |
| **numRec** | Number of recipients in to & cc |
| **perCaps** | Percentage of caps in entire message except attachment |
| **isInReplyTo** | TRUE if the In-Reply-Token is present in the header |
| **sortedRec** | TRUE if words in the subject have punctuation or numbers embedded in them |
| **hour** | Hour of the day in the Date Field |
| **multipartText** | TRUE if the MIME type is multipart/text. |
| **hasImages** | TRUE if the message contains images |
| **isPGPsigned** | TRUE if the message contains a PGP signature |
| **perHTML** | Percentage of characters in HTML tags in the message body in comparison to all characters. |
| **subSpamWords** | TRUE if the subject contains one of the words in a spam word vector. |
| **subBlanks** | Percentage of blanks in the subject |
| **noHost** | TRUE if there is no hostname in the Message-Id key in the header. |
| **numEnd** | TRUE if email senders address ends in a number |
| **isYelling** | TRUE if the subject is all capital letters. |
| **forwards** | Number of forward symbols in a line of the body |
| **isOrigMsg** | TRUE if the message body contains the phrase original message |
| **isDear** | TRUE if the message body contains the word dear |
| **isWrote** | TRUE if the message contains the phrase wrote |
| **avgWordLen** | The average length of the words in a message |
| **numDlr** | Number of dollar signs in the message body |

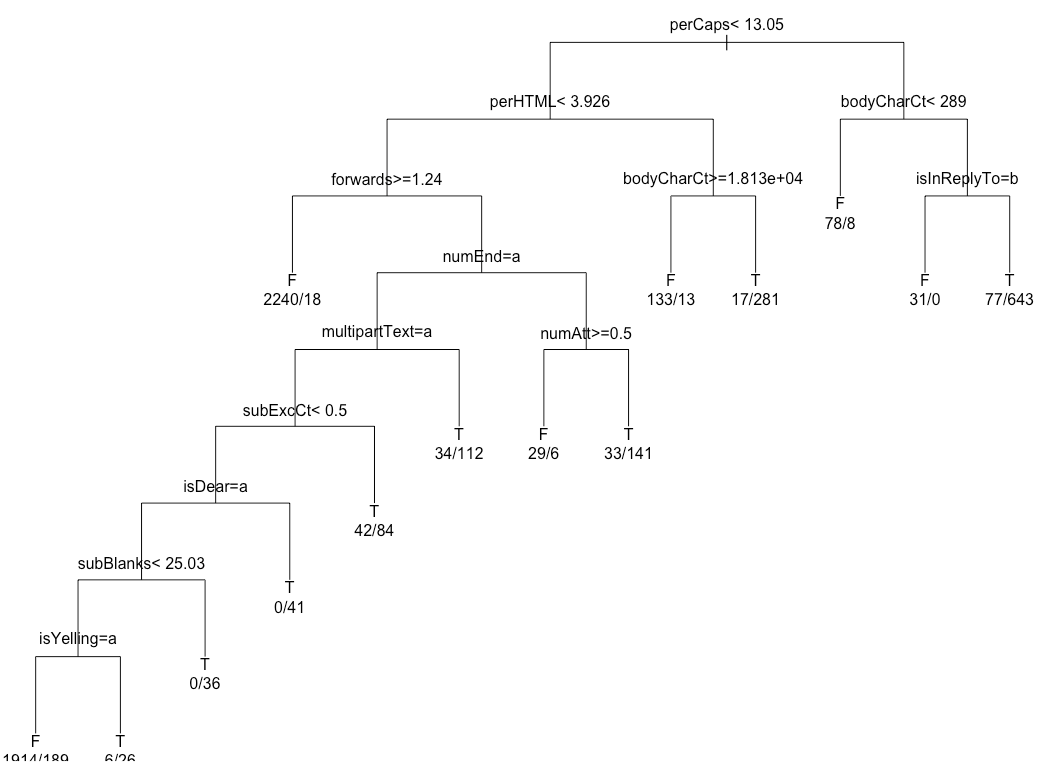
## **Method**

Our data preparation process follows the case study *Using Statistics to Identify Spam* in the Nolan and Lang text [3]. For each message, our first task was to split the message into its header and its message body. This was done by identifying the first empty line in the entire message and returning everything before it as the header and everything after it as the message. Next, since the data in the header is represented as key-value pairs we implemented a function to parse through the header and convert it to named vector where each element’s name is the key from the header. Finally, we implemented a series of functions that examine the message body for the presence of attachments. If attachments are found they are removed from the body and a record of each attachment type and length is stored alongside the header and message body into a data structure suitable for further processing.

Once the data were processed per above, we had the inputs for our feature engineering. The entire data set was run through a series of functions to determine the various features as outlined in Table 1. The output of the preparation functions is a data frame that identified each email uniquely by row name and includes a value for each of the features and whether the email was spam or ham. With this prepared data, we performed recursive partitioning modeling on the data and investigate whether performance can be improved from the basic models performed by the authors.

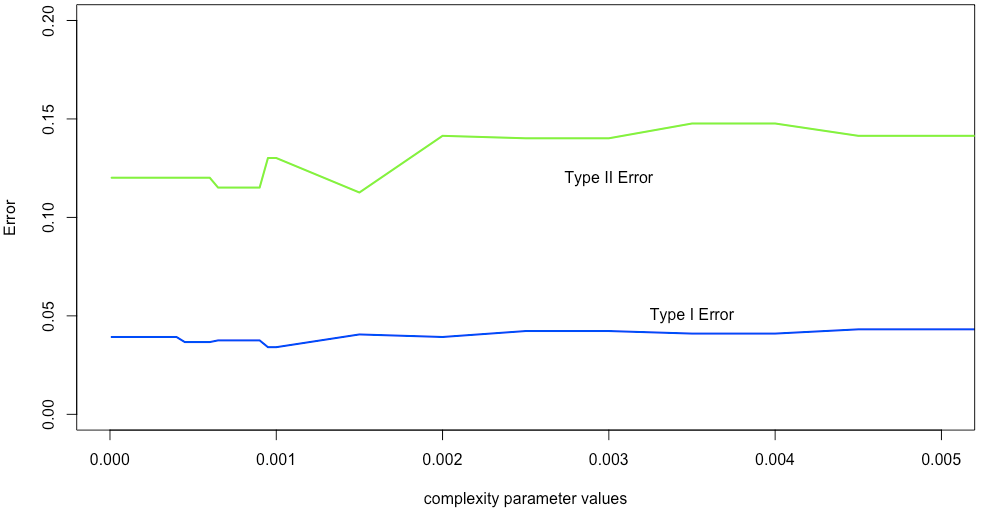
To establish a baseline for our potential accuracy improvements, we proceeded with implementing the same rpart models as the authors. The first model did not customize any of the parameters in rpart.control, which is the list of options that control the details of the rpart algorithm. Notably, this meant the value of the complexity parameter cp was 0.01. The complexity parameter is an algorithm constraint that controls the depth of a recursive partitioning tree. It accomplishes this by not considering any split that does not decrease the lack of fit by more than the value of cp. This model was trained on the same 6232 messages and is graphically displayed in Figure 2. Similar to the authors, our unparameterized rpart model had a Type I error rate of 5.4% and a Type II error rate of 16%. Type I error is defined as the proportion of ham messages in the data set that were misclassified as spam and Type II error is the inverse where spam messages are misclassified as ham. This default rpart model fit had an accuracy of 91.9%.

**Figure 2.** Tree for Partitioning Spam Messages Trained from Default Parameters



We proceeded to implement the authors next set of models by serially updating the value for cp and comparing the Type I and Type II errors in order to optimize the complexity parameter value. We created a vector that is 56 numbers in length ranging from .00001 to the cp default value of .01. We note that 55 of these values are smaller than the default. Figure 3 shows the plots for the Type I and Type II errors for values of cp ranging between 0 and .005 as we found the Type I error is minimized for a complexity parameter of .001. For complexity parameter of .001 we calculated the Type I error rate to be 3.4% and the Type II error rate to be 13%. The accuracy for this model fit was 94.1%. These error rates are both improvements over the default unparameterized rpart model. Next we will explore the other parameters that are available in rpart.control to try and improve the Type I error and Type II error even further.

**Figure 3.** Type I and Type II Errors for Complexity Parameter Values .000 to .005



Upon inspection of the rpart.control documentation we find several parameters related to surrogates, including: maxsurrogate, usesurrogate, and surrogatestyle. We found that rpart includes a clever scheme for handling missing values in the training data. Once a splitting variable has been decided for a node in the tree, it isn’t possible to further progress observations that may be missing that value. Instead of dropping these observations entirely, rpart attempts to estimate the missing value using other independent variables from the observation - these estimated variables are called surrogate variables. In our case - the data we produced from our feature engineering does not contain any missing values, so these rpart.control parameters will not be of any interest. Next we shift our focus to a series of rpart.control parameters that are all associated with tree complexity and depth. These parameters are listed and defined in Table 4 [5].

**Table 4.** Parameters of Interest rpart.control

|  |  |
| --- | --- |
| **Parameter** | **Definition** |
| minsplit | The minimum number of observations that must exist in a node in order for a split to be attempted. |
| minbucket | The minimum number of observations in any terminal 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. |
| cp | Complexity parameter. Any split that does not decrease the overall lack of fit by a factor of cp is not attempted. For instance, with anova splitting, this means that the overall R-squared must increase by cp at each step. The main role of this parameter is to save computing time by pruning off splits that are obviously not worthwhile. Essentially,the user informs the program that any split which does not improve the fit by cp will likely be pruned off by cross-validation, and that hence the program need not pursue it. |
| maxdepth | Set the maximum depth of any node of the final tree, with the root node counted as depth 0. Values greater than 30 rpart will give nonsense results on 32-bit machines. |

We then proceeded to perform a grid search on these parameters, which is similar to the author’s approach to investigate accuracy improvements by iterating over adjustments in cp. Grid search is the process of performing hyper parameter tuning in order to determine the optimal values for a given model. We note that minsplit is related to minbucket, so we decided to proceed with a grid search on: cp, minsplit, and maxdepth. We perform a grid search over all permutations of values for the parameters listed in Table 5.

**Table 5.** Grid Search Values for Select rpart.control Parameters

|  |  |
| --- | --- |
| **Parameter** | **Range of Values** |
| minsplit | 5, 7, 9, 11, 13, 15 |
| cp | 0.1, 0.01, 0.001, 0.0001, 0.00001 |
| maxdepth | 5, 10, 15, 20 , 25, 30 |

## **Results**

Our grid search results found an optimal hyper parameter tuning of minsplit = 5, maxdepth = 30, and cp = .0001. This model has an accuracy of 95.2%, a Type I error of 2.7%, and a Type II error of 10.6%. Each of these statistics exhibit improvements over the author’s second model, which left the default parameters and set cp = .001. It’s important to note that when no parameters are provided to rpart.control, they are defaulted. The default values are minsplit = 20, maxdepth = 30, and cp = .01. By setting our minsplit value to 5 and our cp value to .0001 we are allowing our tree to grow in height compared to the author’s second model. We note that we are improving accuracy while also avoiding overfitting, since the accuracy statistics are taken from an unseen test set. In general, we found as we reduced the complexity parameter and increased the max depth, the accuracy of the model increased. In order to visualize this relationship, we set the minsplit parameter to 5 and produced a heat map across all grid search values of cp and maxdepth as shown in Figure 6.

**Figure 6**. Accuracy Heat Map for Complexity Parameter and Max Depth Combinations



## **Conclusions**

Spam email is a nuisance at best and a danger at worst. We know that humans can typically spot spam very quickly, and we want to ensure that valid emails are not being misclassified as spam. To do this, we prioritize a low Type I error over a low Type II error. Our model uses a small minimum split size and limits the maximum depth of the overall tree. The smaller complexity parameter means that we will allow nodes to be split at a lower minimum threshold for increasing the accuracy of the overall model. The combination of a controlled depth tree, with a small minimum node size and a lower complexity parameter gives us a balance of depth and width in the overall tree that controls for overfitting while also improving model performance measures of accuracy and error.

## **Future Work**

Additional work on this topic may include testing different sets of features. As is the case with any engineered features, there is always opportunity to explore different structures of those features or adding or eliminating features based on their impact to the overall model performance. Recursive feature engineering is one approach to evaluating from a large set of features which ones are most valuable and creating the simplest model with the best outcomes.

Another possible next step on this analysis would be employing a random forest approach. The method applied in this case study was to evaluate one tree with the given set of features and find the best tree possible. In a random forest approach, multiple trees are averaged together. This approach would be a good analysis tool, but would not likely be a productionized algorithm, since there are performance considerations when looking at large datasets. For analysis purposes, random forest would help in sampling a variety of features and reduce the risk of overfitting, which is always a concern when working with trees [6].

## 

## **References**

1. “Global spam volume as percentage of total e-mail traffic from January 2014 to December 2018, by month,” *Statista.* 2019. <https://www.statista.com/statistics/420391/spam-email-traffic-share/>
2. Lardinois, F. "Google says its machine learning tech now blocks 99.9% of Gmail spam and phishing messages – TechCrunch", *TechCrunch*, 31 May 2017. <https://techcrunch.com/2017/05/31/google-says-its-machine-learning-tech-now-blocks-99-9-of-gmail-spam-and-phishing-messages/>
3. Nolan, D. and Lang, D. T. “Data Science in R.” CRC Press, 2015 (Chapter 3)
4. Therneau, T. and Atkinson, B. “Package rpart”.12 April 2019. <https://cran.r-project.org/web/packages/rpart/rpart.pdf>
5. Therneau, T. and Atkinson, B. “An Introduction to Recursive Partitioning Using the RPART Routines”.11 April 2019. <https://cran.r-project.org/web/packages/rpart/vignettes/longintro.pdf>
6. Chilling, M. "Comparison of machine learning methods in email spam detection", *Matchilling.com*, 11 February 2018. <https://www.matchilling.com/comparison-of-machine-learning-methods-in-email-spam-detection/#fn-5.>

## 

## 

## **Appendix - R Code**

library(tm) # import stop words

# set path

spamPath = "/Users/benjaminwilke/Desktop/data"

#spamPath = "/Users/bwilke/Desktop/SpamAssassinMessages"

# get directory names

dirNames = list.files(spamPath)

# display number of files in each sub directory

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

function(dir) length(list.files(dir)) )

# form full path for each sub directory

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

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

# testing set

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)

## input is raw text message from readLines, output is 2 character vectors header and body

splitMessage = function(msg) {

splitPoint = match("", msg)

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

body = msg[ -(1:splitPoint) ]

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

}

# takes a header, searches for index of boundary=, removed quotes, removes trailing semi-colon, return boundary strin

getBoundary = function(header) {

boundaryIdx = grep("boundary=", header)

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

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

}

# drops attachments from Body by passing in the body and the boundary string

dropAttach = function(body, boundary){

bString = paste("--", boundary, sep = "") # form full boundary string

bStringLocs = which(bString == body) # find all boundary indexes

if (length(bStringLocs) <= 1) return(body) # if one or less boundaries, no attachments. return the body

eString = paste("--", boundary, "--", sep = "") # form ending string

eStringLoc = which(eString == body) # find ending boundary line

if (length(eStringLoc) == 0)

return(body[ (bStringLocs[1] + 1) : (bStringLocs[2] - 1)]) # if no ending boundary, return lines between first 2 boundaries, which is the body

n = length(body)

if (eStringLoc < n) # if ending boundary isn't the last line

return( body[ c( (bStringLocs[1] + 1) : (bStringLocs[2] - 1), ( (eStringLoc + 1) : n )) ] ) # return lines between first 2 boundaries, which is the body, and also everything after the ending boundary

return( body[ (bStringLocs[1] + 1) : (bStringLocs[2] - 1) ]) # normal case, ending boundary is last line, return linee between first 2 boundaries, which is the body

}

# takes a message, removes all punctuation, white space, and replace with space

cleanText = function(msg){

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

}

findMsgWords = function(msg, stopWords) {

if(is.null(msg))

return(character())

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

# drop empty and 1 letter words

words = words[ nchar(words) > 1]

# drop words that are in stopwords passed to this function, which will be from tm package

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

invisible(words)

}

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)

}

#-------------------------------------------------#

#----------------PREPROCESS DATA------------------#

#-------------------------------------------------#

#import stopwords from tm package

stopWords = stopwords()

#process data

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

#calc num of msgs

numMsgs = sapply(msgWordsList, length)

numMsgs

#assign labels

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

#flatten to 1 list

msgWordsList = unlist(msgWordsList, recursive = FALSE)

# -----------------------------------------------------------------------------------

numEmail = length(isSpam)

numSpam = sum(isSpam)

numHam = numEmail - numSpam

# -------------Recursive Partitioning and Classification Trees-----------------------

## this is Laura's updated processHeader

processHeader = function(header) {

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

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

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

close(tc)

headerVec = unlist(headerMat)

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

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

return(headerVec)

}

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)

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)

)

}

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")

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))

}

)

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)

}

emailDF = createDerivedDF(emailStruct)

library(purrr)

library(dplyr)

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")

})

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)

})

plot(errs[1,] ~ complexityVals, type="l", col='blue',

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='green', lwd = 2)

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

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

compute\_accuracy <- function(fit, test\_features, test\_labels) {

predicted <- predict(fit, test\_features, type = "class")

mean(predicted == test\_labels)

}

library(purrr)

library(dplyr)

grid <- list(minsplit = c(5),

maxdepth = c(5, 10, 15, 20 , 25, 30),

cp = c(0.1, 0.01, 0.001, 0.0001, 0.00001)

) %>% cross\_df()

trainModel <- function(...) {

rpart(isSpam ~ ., data = trainDF,

method="class",

control = rpart.control(...) )

}

grid <- grid %>% mutate(fit = pmap(grid, trainModel))

compute\_accuracy <- function(fit, test\_features, test\_labels) {

predicted <- predict(fit, test\_features, type = "class")

mean(predicted == test\_labels)

}

compute\_type1 <- function(fit, test\_features, test\_labels) {

predicted <- predict(fit, test\_features, type = "class")

mean(predicted[ !test\_labels == "T" ] == "T")

}

compute\_type2 <- function(fit, test\_features, test\_labels) {

predicted <- predict(fit, test\_features, type = "class")

mean(predicted[ test\_labels == "T" ] == "F")

}

test\_features <- testDF %>% select(-isSpam)

test\_labels <- testDF$isSpam

grid <- grid %>%

mutate(test\_accuracy = map\_dbl(fit, compute\_accuracy,

test\_features, test\_labels)) %>%

mutate(test\_type1 = map\_dbl(fit, compute\_type1,

test\_features, test\_labels)) %>%

mutate(test\_type2 = map\_dbl(fit, compute\_type2,

test\_features, test\_labels))

grid <- grid %>% arrange(desc(test\_accuracy), desc(test\_type2), desc(test\_type1), desc(maxdepth))

grid

library(ggplot2)

## plot type I data

ggplot(grid, aes(as.factor(maxdepth), as.factor(cp))) +

geom\_tile(aes(fill = test\_type1\*100)) +

geom\_text(aes(label = round(test\_type1\*100, 2))) +

scale\_fill\_gradient(low = "white", high = "red") +

xlab('Max Depth') + ylab('Complexity Parameter') +

guides(fill = guide\_colourbar(title = NULL)) +

ggtitle("Type I Error Rates") +

theme(plot.title = element\_text(hjust = 0.5))

## plot type II

ggplot(grid, aes(as.factor(maxdepth), as.factor(cp))) +

geom\_tile(aes(fill = test\_type2\*100)) +

geom\_text(aes(label = round(test\_type2\*100, 2))) +

scale\_fill\_gradient(low = "white", high = "red") +

xlab('Max Depth') + ylab('Complexity Parameter') +

guides(fill = guide\_colourbar(title = NULL)) +

ggtitle("Type II Error Rates") +

theme(plot.title = element\_text(hjust = 0.5))

## plot accuracy

ggplot(grid, aes(as.factor(maxdepth), as.factor(cp))) +

geom\_tile(aes(fill = test\_accuracy\*100)) +

geom\_text(aes(label = round(test\_accuracy\*100, 2))) +

scale\_fill\_gradient(low = "white", high = "red") +

xlab('Max Depth') + ylab('Complexity Parameter') +

guides(fill = guide\_colourbar(title = NULL)) +

ggtitle("Accuracy") +

theme(plot.title = element\_text(hjust = 0.5))