## **Introduction**

Spam identification is a task that is commonplace for our email servers to perform on a daily basis. This is the task of flagging messages that could contain harmful, inappropriate, or unwanted content. Early efforts at filtering spam were centered on the assumed validity of identity of emailers: recipients were expected to “white list” their favored senders, and the senders’ validity would be assumed valid if the email was shown to be from the same address.

However, we know that these filters are not perfect as we still receive phishing or spam messages in our primary inbox. Today, many major email handling systems have moved to an ensemble of factors that can be labeled with a final “TRUE” or “FALSE” to the email’s spam content indication. As email filtering continues to move in the direction of machine learning, development of new variables is likely an area for improvement in accuracy and performance. Training a machine to perform the task of identifying whether an email is spam or not is not a trivial task. Statistical models can be leveraged to aid in this process, which we will investigate in this analysis.

This case study is based on the analysis completed in **Chapter 3** of *Data Science in R: A Case Studies Approach to Computational Reasoning and Problem Solving*. In this chapter, the authors **Deborah Nolan** and **Duncan Temple Lang** provide codes and procedures to train a spam-filtering algorithm on over 9000 email messages [1]. Several text-mining approaches were taken to identify messages that are spam based on characteristics such as the amount of capitalization in the subject line and the frequencies of various words in the body of the message.

For this analysis, we will be expanding on the work done by **Nolan** and **Lang** by selecting a different log odds threshold and performing a different cross validation method to answer **Question 20** from the text. This allows us to compare the Type I and Type II error rates, which is important in identifying spam. Email users do not want important messages to be classified as spam (type I error, false positive) and do not want spam to show up in their primary inboxes (type II error, false negative) [2]. This will help improve spam-filtering algorithms to produce the most accurate method possible.

## **Background**

Spam classification is a task easily done by the human brain. Typically, users will notice a large amount of capital letters, misspelled words, implications of winning a prize, or sexually explicit content and recognize that this message should be deleted. The patterns that we notice in these emails can be used to train machines to perform the same task. This task is increasingly important, as many cyber criminals will deliver viruses via hyperlinks and attachments in email messages. Statistical models such as **Naïve Bayes** can be used to help identify spam messages as accurately as possible.

The **Naïve Bayes** classifier is a relatively simple model based on the **Bayesian** theorem. It is a probabilistic model that aims to determine the probability that an occurrence belongs in each class and return the most likely class. It uses Bayes rule, which is found using the mathematical formula for class A and feature B. The classifier then selects the class with the highest probability calculated [3].

The data used to train this classifier is from previously classified email messages from Spam Assassin [4]. This data was originally part of the RSpamData package, but this has since been removed, and now available at this URL (<http://www.stat.berkeley.edu/~nolan/data/spam/SpamAssassinMessages.zip>). This corpus contains over 9000 emails that have been flagged as either “spam” or “ham” (legitimate emails). This allows us to train a classifier on the data to create an accurate spam identification model.

In the following sections, we will detail how the email messages were cleaned and prepared for analysis, how the classifier was trained, and how choosing thresholds for the log odds affects the Type I and II errors of the model.

## **Method**

**Data Description**

In the dataset used for this analysis, there are 9348 unique emails which were classified as spam or valid. There are 29 predictor variables and 1 response variable (“isSpam”). The detailed information can be derived from an email message and used for classifying spam, as shown in **Table 1** [1].

Briefly, there are in total 13 numeric and 17 Boolean factor variables.

Numeric variables: perCaps, bodyChartCt, numLines, subExcCt, subQuesCt, numAtt, numRec, hour, perHTML, subBlanks, forwards, avgWordLen, numDlr.

Boolean factor variables: isSpam, isRe, isYelling, underscore, priority, isinReplyTo, sortedRec, subPunc, multipartText, isPGPsigned, subSpamWords, noHost, numEnd, isOrigMsg, isDear, isWrote.

### Exploratory Data Analysis

#### **Explanatory variable correlation relationships**

In **Figure1A**, the correlation matrix reveals numLines is positively correlated with bodyCharCt significantly (correlation efficiency is 0.92) between numerical variables. These two variables, bodyChartCt and numLines, represent the number of distinct lines or characters in the body of the message.

|  |
| --- |
| **Table 1: Variable definition table** [1] |
|  |

For Boolean predictor variables, the p-value matrix for dichotomous variables from **Fisher’s** exact tests are shown in **Figure 1B**. Correlation is not an appropriate metric for analyzing nominal or dichotomous relationships. However, Fisher’s exact test allows us to non-parametrically examine the association between categorical variables. Significant non-random dependence between factor variables exists since lower p-values indicate we reject the null hypothesis of random association.

We also inspected the biserial correlation between factors and continuous variables (not shown). All these visualizations are helpful in understanding variable mutual relationships.

|  |  |
| --- | --- |
| **Figure 1: Correlation between numeric or categorical predictor variable pairs** | |
| **A: Numeric variables** | **B: Categorical variables** |

#### **Response variable relationships**

In order to better understand the response variable relationships between spam and valid emails, we visualize the distribution with bar plots for factor predictors (**Figure 2**) and box plots for continuous predictors (**Figure 3**).

In **Figure 2**, 6 boolean predictor variables (isDear, isRe, isWrote, isYelling, numEnd, priority) were investigated the viability for spam prediction.

For example, the variables isRe and isWrote indicate it is TRUE if Re: appears at the start of the subject or Wrote: is in the phrase. Comparatively, other 4 variables (is Dear, isYelling, priority, numEnd) contribute less in classifying spam and valid emails, as based on the counts.

For numeric variables, box plots were used in identifying spam or valid emails by comparing their log values.

As shown in **Figure 3**, the predictor variable forward has a more concentrated distribution of values in the third quartile for messages that are valid, while the predictor variable perCaps shows a larger interquartile range for spam.

Around perCaps values in 80% of valid messages is lower than the median perCaps value for spam messages. PerHTML is also a decent classification variable since the majority of its third quartile occurs specifically with spam predictions. However, for some predictor variables, such as hour and avgWordLen, may not be good candidates for classification.

|  |
| --- |
| **Figure 2: 6 Boolean predictor variables and spam outcomes (Y axis facet: spam as T, valid as F)** |
|  |

|  |
| --- |
| **Figure 3: 10 Continuous predictor variables and spam outcomes** |
|  |

The emails used in this analysis were first cleaned by identifying whether the email had an attachment, separating the message body from the rest of the information, and cleaning the message body by removing punctuation, numbers, and stop words. This is an important step when working with data involving language, so the analysis does not include useless information. Stop words, or frequently used words in the English language, do not provide meaningful insight into the message body as these words are used frequently and without substantial meaning beyond providing grammatical structures. Including these words in the training or testing data set would produce misleading and most likely inaccurate results.

The cleaned messages are then divided into training and testing data sets to train models to classify whether a message is spam or “ham.” The log odds can be used to interpret the performance of the models, with higher values indicating a higher probability of being in the chosen group. We find the log likelihood ratio of a message being spam or “ham.” These results can be viewed in Figure **4**.

|  |
| --- |
| **Figure 4: The log likelihood ratio (LLR) of messages classified as “ham” or “spam”** |
|  |

**Figure 4** reveals that messages classified as spam have much higher log likelihood ratios than those classified as “ham”, the valid emails. This information can be used to classify other email messages with the goal of achieving the most accurate classifications possible.

## **Results**

**Type I and Type II error calculations**

Type I error is the proportion of ham messages that have been misclassified as spam, and type II error rate is the proportion of spam messages that have been misclassified as ham. For boxplots of log likelihood ratios (**LLR**) in **Figure 4**, we use typeIErrorRate() to calculate the Type I and type II error rates. A threshold for the log odds τ is chosen to evaluate the Type I and II error rates with the chosen threshold. When we use a value of τ=0, the type I error rate is 0.003. This means that about 0.3% of messages are being classified as spam when they are actually legitimate messages. Using a threshold of τ=20, the type I error rate is 0.006. This threshold seems to perform slightly worse according to type I error rates. Typically, we want to find the error rate for a vector of τs because we want to find one that provides an acceptable Type I error. We will further investigate the value of τ that optimizes the model according the type I and II error rates.

For selecting a threshold, we need to search possible values of τ in restricted intervals. When any value of τ less than the minimum of the LLR values, we classify all messages as spam and the Type I error rate is 1 [1]. Similarly, any value of τ greater than the maximum of the LLR values implies that we classify every message in our sample as ham, so our Type I error rate is 0.

In addition, there are also errors in misclassifying spam as ham. The Type II error is 1 when we use the largest observed LLR value in our test set because all spam is classified as ham. The Type I error rate only changes at values of τ that match one of the observed LLR values in our set of messages [1]. For example, for 2 values of τ, if there are no LLR values from the test set between them, then their associated Type I errors must be the same. Likewise, the Type II error rates for them are the same.

**Calculate Type I and Type II errors with 5-fold cross-validation**

In order to control and arrive at better Type I and II error rates, the data had to be split into groups or folds. The data set was split into five folds and each fold had a “turn” being the test set. If fold one was the test set than folds two through five would be used to train the model. After the model was validated on fold one than fold two would now be the test set. Folds one and three through five would now be used to train the model. This process would occur until each fold was used as the test set. For each test set the log likelihood ratio is computed given the probability that a word occurs in a message given it is spam or ham. All of these log likelihood ratios are then pooled together and used to find a Type I error of 1%.

Using the method discussed above, we discover that the value that minimizes both the type I and type Ii error rates is -43. **Figure 5A** displays the log likelihood ratio value necessary to achieve error rates of 1%.

We then split the data using a 5-fold cross validation method. Using this, we discover that the optimal value for τ is -31, which achieves both a type I and type II error rate of 10%. **Figure 5B** displays the values of the log likelihood necessary to achieve optimal error rates.

The results are more favorable using the five-fold cross validation method since the log likelihood ratio is a higher value (-31) than without the cross validation (-43). However, the Type II error increased from 2% to 6%. This means that the probability of classifying an email as a legitimate email when it is actually spam is higher with the cross-validation model. This is not a desirable classification result and could potentially have a large impact for a user. Spam messages can potentially contain harmful viruses or phishing attacks that could affect inexperienced or careless email users.

|  |
| --- |
| **Figure 5. Log Likelihood ratio values vs. error rates (Type I error =0.01)** |
| **A** |
| **B 5-fold cross-validation** |
|  |

In **Figure 6**, when Type I error rate sets to 0.005, the Type II error rate has increased to 0.13 and the τ is -14. For 5-fold cross-validation, Type II error rate has jumped to 0.2 and the τ is -1.

|  |  |
| --- | --- |
| **Figure 6. Log Likelihood ratio values vs. error rates (Type I error = 0.005)** | |
| **A** | **B 5-fold cross-validation** |

**Type I and Type II error rate calculation in recursive partitioning**

With rpart() function, Type I and II Errors are also helpful in understanding the Recursive Partitioning process, as discussed in the textbook. Type I error is the proportion of ham messages that have been misclassified as spam. The complexity parameter is a mechanism for specifying the threshold for choosing a split for a subgroup.

**Figure 7** displays this example of using Type I and II errors for predicting spam as a function of the size of the complexity parameter in the rpart() function. The Type I error are able to achieve around 0.034 which occurs for a complexity parameter value of about 0.001, while the Type II error rate for this complexity parameter value is 0.13.

|  |
| --- |
| **Figure 7: Type I and II Errors for Recursive Partitioning** |
|  |

## **Conclusions**

Recognizing an email as spam or ham is a very important task for email providers. By correctly identifying an email they can improve their customer experience by limiting the number of spam emails a customer receives. However, they must be careful not to over classify too many emails as spam because then users will not be receiving emails they actually should be receiving and depending on the contents of these emails this could have huge impacts.

In this case study a five-fold cross validation approach was used in order to determine the odds that a word would be in an email given that the email was either spam or ham. These probabilities were then pooled together in order to find the log likelihood ratio necessary to achieve Type I and II error rates of 1%. This value was determined to be -35 which is higher, and thus more favorable, than the non-cross validated model’s ratio which was -43. While both these models had a Type I error rate of 1% the cross validated model has a Type II error rate of 6% which is higher than the non-cross validated model’s Type II rate of 2%. This means that the probability of the cross validated model classifying an email as “ham” when it is actually spam is higher than the non-cross validated model. This of course is not ideal, but it is a modest increase and not one that should be overly concerning.

## **Future Work**

The additional improvements to this study could be achieved by providing a more balanced dataset. Oversampling of spam emails could reduce the tendency for rpart models to give false negative results and increase type II error rate.

**Model expansion**

One area that could be changed in future models is the number of folds chosen for cross validation. In the model used there were five folds selected which was chosen just by random. It would be beneficial to do an analysis to determine what the ideal number of folds would be. There is a tradeoff between a small and large number of folds. If the number of folds is too small than the cross validation will not be useful because the data has not been broken up enough. If the number of groups is too large than overfitting becomes a possibility because there would be so few numbers of observations in each fold. The ideal number of folds will be somewhere in the middle. By determining the ideal number of folds, one could increase the model’s performance and receive a low Type I and II error rate.

**Ethical considerations**

Pretty much every email server today includes a built-in spam classification technique. However, it would be worth investigating how these servers are classifying spam. Methods discussed in this analysis require access to the words in the email bodies. If similar methods are used, this would mean that email servers have access to the messages and words sent between users. Many users are unaware that their messages are being read or can be accessed. However, companies, supervisors, and legal departments often have access to employees’ emails. This could be considered a breach of personal privacy and confidentiality if the user is not aware that their emails are being read or used in ways that they have not explicitly consented to. Users need to be aware and give consent to email servers and supervisors that are using their emails to identify spam or keep an eye on their work or conversations.

**References**

1. Nolan, D. and Temple Lang, D. (2015). Data Science in R: A Case Studies Approach to Computational Reasoning and Problem Solving. Boca Raton, FL: CRC Press (Chapter 3).
2. Berkeley. (n.d.). Multiple Hypothesis Testing and False Discovery Rate. Retrieved February 7, 2019, from https://www.stat.berkeley.edu/~hhuang/STAT141/Lecture-FDR.pdf
3. Soni, D. (2018, May 16). Introduction to Naive Bayes Classification – Towards Data Science. Retrieved February 7, 2019, from <https://towardsdatascience.com/introduction-to-naive-bayes-classification-4cffabb1ae54>
4. Apache SpamAssassin. (n.d.). Retrieved February 7, 2019, from https://spamassassin.apache.org/

## **Appendix - R code**

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

## Section 0: Load the packages for case study 3 from Unit 6

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

Packages <- c('magrittr', 'RColorBrewer', 'ggplot2', 'tidyr', 'dplyr', 'plyr', 'tm', 'NLP',

'rpart', 'caret', 'rpart.plot', 'ltm', 'scales')

lapply(Packages, library, character.only = TRUE)

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

## Section 1: get and check files

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

# spam file path

spamPath = "./SpamAssassinMessages/"

# list files

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

# check easy\_ham

head(list.files(path = paste(spamPath,'messages','easy\_ham', sep=.Platform$file.sep)))

# check spam\_2

head(list.files(path = paste(spamPath, "messages", "spam\_2", sep = .Platform$file.sep)))

# check file length

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

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

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

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

# Pick up 'easy\_ham' from 'messages'

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

fileNames[1]

msg = readLines(fileNames[1])

head(msg)

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)

# Find the words in a message

msg = sampleEmail[[1]]

which(msg == "")[1]

match("", msg)

splitPoint = match("", msg)

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

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

body = msg[ -(1:splitPoint) ]

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)

header = sampleSplit[[1]]$header

grep("Content-Type", header)

grep("multi", tolower(header[46]))

header[46]

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]))

})

hasAttach

header = sampleSplit[[6]]$header

boundaryIdx = grep("boundary=", header)

header[boundaryIdx]

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

header2 = headerList[[9]]

boundaryIdx2 = grep("boundary=", header2)

header2[boundaryIdx2]

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

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

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

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

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

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

getBoundary = function(header) {

boundaryIdx = grep("boundary=", header)

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

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

}

sampleSplit[[6]]$body

boundary = getBoundary(headerList[[15]])

body = sampleSplit[[15]]$body

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

bStringLocs = which(bString == body)

bStringLocs

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

eStringLoc = which(eString == body)

eStringLoc

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

tail(msg)

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

tail(msg)

dropAttach = function(body, boundary){

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

bStringLocs = which(bString == body)

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

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

eStringLoc = which(eString == body)

if (length(eStringLoc) == 0)

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

n = length(body)

if (eStringLoc < n)

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

( (eStringLoc + 1) : n )) ] )

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

}

head(sampleSplit[[1]]$body)

msg = sampleSplit[[3]]$body

head(msg)

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

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

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

stopWords = stopwords()

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

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

SWords = SWords[ nchar(SWords) > 1 ]

stopWords = unique(SWords)

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

words = words[ nchar(words) > 1 ]

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

head(words)

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]

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)

}

# solve the Lat\_1 problems with Sys.setlocale

Sys.setlocale("LC\_ALL", "pt\_PT.ISO8859-1")

# get message words list

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

# check number of messages

numMsgs = sapply(msgWordsList, length)

numMsgs

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

msgWordsList = unlist(msgWordsList, recursive = FALSE)

numEmail = length(isSpam)

numSpam = sum(isSpam)

numHam = numEmail - numSpam

set.seed(418910)

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

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

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

(msgWordsList[!isSpam])[testHamIdx] )

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

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

testIsSpam = rep(c(TRUE, FALSE),

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

trainIsSpam = rep(c(TRUE, FALSE),

c(numSpam - length(testSpamIdx),

numHam - length(testHamIdx)))

bow = unique(unlist(trainMsgWords))

length(bow)

spamWordCounts = rep(0, length(bow))

names(spamWordCounts) = bow

tmp = lapply(trainMsgWords[trainIsSpam], unique)

tt = table( unlist(tmp) )

spamWordCounts[ names(tt) ] = tt

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)

}

# trainTable

trainTable = computeFreqs(trainMsgWords, trainIsSpam)

# Check word frequency

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])

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])

}

testLLR = sapply(testMsgWords, computeMsgLLR, trainTable)

tapply(testLLR, testIsSpam, summary)

# Figure 4 The log likelihood ratio of messages classified as “ham” or “spam”

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

boxplot(testLLR ~ spamLab, ylab = "Log Likelihood Ratio", ylim=c(-500, 500))

dev.copy(png,filename="./LLR\_random\_msg.png", width=400, height=400);

dev.off ();

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

## Section 2: case study part

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

# Type I error rate at tau = 0, or -20

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])

}

# Type II error rate

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])) # <= 0.005 for Figure 6A

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

t2 # Type II error rate

# Figure 5A. Log Likelihood ratio values vs. error rates

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])

dev.copy(png,filename="./Type errors\_cv1.png", width=600, height=400);

dev.off()

# if cv = 5

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])) # change to <=0.005 for Figure 6B

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

## Figure 5B. Log Likelihood ratio values vs. error rates

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

plot(xFoldII$error ~ xFoldII$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(xFoldI$error ~ xFoldI$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(tauFoldI, side = 1, line = 0.5, at = tauFoldI, col = cols[3])

segments(x0 = tauFoldI, y0 = -.50, x1 = tauFoldI, y1 = tFold2,lwd = 2, col = "grey")

text(tauFoldI + 20, 0.05, pos = 4, paste("Type II Error = ", round(tFold2, digits = 1)), col = cols[1])

dev.copy(png,filename="./Type errors\_cv5.png", width=600, height=400);

dev.off()

# Type I error rate for cv = 5

xFoldI=typeIErrorRate(-31, testFoldOdds, testFoldSpam)

xFoldI

# Type I error rate for cv = 5

tFold2 = max(xFoldII$error[ xFoldII$values < tauFoldI ])

tFold2

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

## Section 3: The complete dataset

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

## sample split

sampleSplit = lapply(sampleEmail, splitMessage)

header = sampleSplit[[1]]$header

header[1:12]

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

header[1]

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

headerPieces[, "Delivered-To"]

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)))

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)

}

## Head list

headerList = lapply(sampleSplit,

function(msg) {

processHeader(msg$header)} )

contentTypes = sapply(headerList, function(header)

header["Content-Type"])

names(contentTypes) = NULL

contentTypes

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

hasAttach

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))

# processAttach()

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

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) # will cause an error with the first message. its ok.

attList[[2]]$attachDF

body = bodyList[hasAttach][[2]]

length(body)

body[35:45]

# processAttach()

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

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

emailStruct = mapply(processAllEmail, fullDirNames,

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

emailStruct = unlist(emailStruct, recursive = FALSE)

sampleStruct = emailStruct[ indx ]

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

header = sampleStruct[[1]]$header

subject = header["Subject"]

els = strsplit(subject, "")

all(els %in% LETTERS)

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

els = strsplit(testSubject, "")

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

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

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

# isYelling()

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)

}

# sampleStruct and perCaps

sapply(sampleStruct, perCaps)

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)

}

)

# createDeriveDF

lapply(funcList, function(func)

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

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)

# FuncList

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))

}

)

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

## Section 4: analyzing the complet dataset

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

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")

#load("spamAssassinDerivedDF.rda")

dim(emailDF)

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 and pC2

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)

all(emailDF$bodyCharCt > emailDF$numLines)

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

# png("ScatterPlotNumLinesNumChars.png")

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

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

box()

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.copy(png,filename="./SPAM\_boxplotsPercentCaps.png", width=400, height=400);

dev.off()

percent = emailDF$perCaps

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

boxplot(log(1 + percent) ~ isSpamLabs, ylab = "Percent Capitals (log)")

dev.copy(png,filename="./ScatterPlotNumLinesNumChars.png", width=400, height=400);

dev.off()

# Percentage of Capital Letters (log scale)

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)

dev.copy(png,filename="./Pct\_Capital\_Letters\_log.png", width=400, height=400);

dev.off()

# png("SPAM\_scatterplotPercentCapsTotChars.png")

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)

dev.copy(png,filename="./SPAM\_scatterplotPercentCapsTotChars.png", width=400, height=400);

dev.off()

table(emailDF$numAtt, isSpamLabs)

# png("SPAM\_mosaicPlots.png")

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)

library(rpart)

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)

dev.copy(png,filename="./SPAM\_mosaicPlots.png", width=400, height=400);

dev.off()

# testDF and trainDF

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, ])

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

# png("SPAM\_rpartTree.png")

prp(rpartFit, extra = 1)

prp(rpartFit, extra = 1)

dev.copy(png,filename="./SPAM\_rpartTree.png", width=400, height=400);

dev.off()

predictions = predict(rpartFit,

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

type = "class")

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

summary(predsForHam)

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

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

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

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))

# complexityVals

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)

})

# png("SPAM\_rpartTypeIandII.png") for Figure 6: Type I and II Errors for Recursive Partitioning

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(0,0.01),

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))

dev.copy(png,filename="./SPAM\_rpartTypeIandII.png", width=600, height=400);

dev.off()

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

## Section 5: EPA

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

load('src/emailDFrp')

# Figure 1A: Correlation between numeric predictor variable pairs

AsVector <- emailDFrp[, c(2:30)]

nums <- sapply(AsVector, is.numeric)

bools <- sapply(AsVector, is.factor)

# correlation matrix for numerical features

cormat <- (round(cor(AsVector[, nums]), 2))

cormat[lower.tri(cormat, diag=TRUE)] <- NA

cormat <- reshape2::melt(cormat, na.rm = TRUE)

# plot correlation matrix

cormat %>% ggplot(aes(x=Var1, y=Var2, fill=value)) +

geom\_tile() +

scale\_fill\_gradient2(low = "green", high = "blue", mid = "white",

midpoint = 0, limit = c(-1,1), space = "Lab",

name="Pearson\nCorrelation")+

geom\_text(aes(Var1, Var2, label = value), color = "white", size = 4)+

theme(legend.position = "right",

axis.text.x = element\_text(angle=90,

vjust=-.5),

legend.text=element\_text(size=10),

text=element\_text(size=14)) +

scale\_x\_discrete("") +

scale\_y\_discrete("") +

guides(fill=guide\_legend(title="Correlation"))

dev.copy(png,filename="./Num\_corr.png", width=400, height=400);

dev.off()

# Figure 1B-pre: Correlation between numeric or categorical predictor variable pairs

Dat <- AsVector

# identify factor and numeric vars for biserial correlation

facs\_indx <- which(lapply(AsVector, is.factor) == TRUE)

facs <- AsVector[,facs\_indx]

nums <- AsVector[,-facs\_indx]

# establish df of correlations

df <- as.data.frame(lapply(nums, function(x) sapply(facs, function(y) biserial.cor(x, y))))

# melt it for viz purposes

df <- reshape::melt(as.matrix(df))

df$value <- round(df$value, 2)

# plot the relationships

df %>%

ggplot(aes(x=X1, y=X2, fill=value)) +

geom\_tile() +

scale\_fill\_gradient2(low = "green", high = "blue", mid = "white",

midpoint = 0, limit = c(-1,1), space = "Lab",

name="Pearson\nCorrelation")+

geom\_text(aes(X1, X2, label = value), color = "white", size = 3)+

theme(legend.position = "right",

legend.text=element\_text(size=8),

legend.title = element\_text(size=10),

axis.text.x = element\_text(angle=90, vjust=0.5),

text=element\_text(size=14)) +

scale\_y\_discrete("") + scale\_x\_discrete("") +

ggtitle("")+

guides(fill=guide\_legend(title="Correlation"))

# Figure 1B: Correlation between numeric or categorical predictor variable pairs

# fisher exact matrix for categorical features

# get booleans

Dat <- AsVector[, bools]

# source combos of each var

combos <- combn(ncol(Dat), 2)

# apply fishers to each combo and capture in df

fishers <- adply(combos, 2, function(x) {

test <- fisher.test(Dat[, x[1]], Dat[, x[2]])

out <- data.frame("Row" = colnames(Dat)[x[1]]

, "Column" = colnames(Dat[x[2]])

, "OddsRatio" = test$estimate

, "type"= test$alternative

, "p.value" = round(test$p.value, 2)

)

return(out)

})

# plot fisher matrix

fishers %>%

ggplot(aes(x=Row, y=Column, fill = p.value)) +

geom\_tile() +

scale\_fill\_gradient2(low = "green", high = "blue",mid = "white",

midpoint = 0, limit = c(0,1), space = "Lab",

name="p-value")+

theme(legend.position = "right",

legend.text=element\_text(size=10),

legend.title = element\_text(size=14),

axis.text.x = element\_text(angle=90, vjust=0.5),

text=element\_text(size=14)) +

scale\_y\_discrete("") +

scale\_x\_discrete("")+

ggtitle("")

dev.copy(png,filename="./Fisher\_corr.png", width=400, height=400);

dev.off()

# Figure 2: Boolean predictor variables and spam outcomes (Y axis faceting: spam email as T)

# which factor variables are worth splitting?

# plot counts of each variable based on their boolean status and the counts of spam and valid

emailDFrp[, c(1,which(bools)+1)] %>%

gather(Predictor, Value, 2:ncol(emailDFrp[,c(1, which(bools)+1)])) %>%

filter(Predictor %in% c("isRe", "isYelling","numEnd", "priority", "isDear","isWrote")) %>%

ggplot(aes(x=isSpam, fill = Predictor, color=Predictor)) +

geom\_bar() +

facet\_grid(Value~Predictor) +

theme\_light() +

theme(legend.position = "bottom",

legend.text=element\_text(size=8),

legend.title = element\_text(size=10),

axis.text.x = element\_text(angle=90, vjust=0.5),

text=element\_text(size=14))+

ggtitle("") +

scale\_y\_continuous("Count", labels =comma) +

scale\_x\_discrete("")

dev.copy(png,filename="./Boolean\_spamOutcomes.png", width=600, height=400);

dev.off()

# Figure 3: Continuous predictor variables and spam outcomes (Y axis faceting: spam email as T)

nums <- which(lapply(emailDFrp, is.numeric) ==TRUE)

emailDFrp[,c(1, nums)] %>%

gather(Predictor, Value, 2:ncol(emailDFrp[,c(1, nums)])) %>%

filter(Predictor %in% c('hour', 'forwards', 'perCaps', 'perHTML', 'numLines', 'bodyCharCt',

'subQesCt','numRec', 'subBlanks', 'avgWordLen', 'numAtt')) %>%

ggplot(aes(x=isSpam, y=log(1+Value), color=Predictor)) +

geom\_boxplot(outlier.size=0.25, position="dodge") +

facet\_wrap(~Predictor, scales = "free\_y", ncol=5) +

theme\_light() +

theme(legend.position = "bottom",

legend.text=element\_text(size=8),

legend.title = element\_text(size=10),

axis.text.x = element\_text(angle=90, vjust=0.5),

text=element\_text(size=14))+

scale\_x\_discrete("")+

ylab("Log Value")

dev.copy(png,filename="./Continuous\_spamOutcomes.png", width=600, height=400);

dev.off()

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

### The end of codes for case study 3 (Unit 6)

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