Using Statistics to Identify Spam – Naïve Bayes, CART, XG Boost

Jeffery Lancon, Manisha Pednekar, Andrew Walch, David Stroud
MSDS 7333 - Quantifying the World - Case Study # 6
02/19/2019

Abstract

In this case study, we explore and examine 9,353 electronic messages that have been classified by "SpamAssassin" (http://spamassasin.apache.org) for the purpose of developing and testing spam filters. The study is an extension of work previously conducted by the authors of *Using Statistics to Identify Spam,* Nolan and Lang (NTL)[1][2][3]. NTL's original case study implemented Naïve Bayes and Recursive Partitioning and Classification Trees (CART) algorithms to classify messages as spam (unwanted) or ham (wanted). Our extended analysis will utilize the same dataset and employ alternative classification algorithms, namely Random Forest and XG Boost, to improve the performance of the spam filter algorithm.

Introduction

Electronic messaging had become an essential form of business and personal communication. An estimated 280 Billion electronic messages are sent/received each day[4]. With 3.9 Billion users, that an average of 72 electronic messages per user each day.

Spam messages are unsolicited electronic messages which may be sent in bulk to various recipients who have not granted permission for the message to be sent. Most email spam messages are commercial in nature. According to statistica.com[5], as of September 2018, spam messages accounted for 53.5% of email traffic worldwide. China, the largest contributor, accounting for 14% of spam messages.

Spam messages are a nuisance, at the least, but also can be dangerous, possibly containing links that lead to phishing web sites or malware. Because of the sheer volume of electronic messages sent/received each data, an automated solution to identifying and eliminating spam messages is required.

Automated spam filters examine various characteristics of a message to determine if the message is spam or ham. Message characteristics are examined using statistical analysis, comparing the message against many messages that have been previously classified as spam or ham.

The authors of *Using Statistics to Identify Spam,* explore a data set assembled by SpamAssassin for the purpose of developing and testing spam filters. The messages are processed into a form that can be readily analyzed by statistical software. The messages are first broken down into two (2) parts:

a) Header: Contains routing information, recipients, from, date, time, subject, message-id, etc.
b) Body: Contains message, attachments, images, etc.

Key statistics/characteristics about each part of the message are determined/calculated and compared against known spam and ham messages, to determine what classification should be placed on the message. A few of the message characteristics analyzed are:

> Key words ('cash', 'free', 'Viagra', 'prescription',…), percentage of capital letters, message word count, hour of day sent, is it a reply, number of recipients, sender's address, just to name a few.

NTL's original case study implemented Naïve Bayes and Recursive Partitioning and Classification Trees (CART) algorithms. We will extend the original analysis utilizing alternative classification algorithms, namely Random Forest and XGBoost, in hopes of improving the performance of the spam filter program.

Using Statistics to Identify Spam – Naïve Bayes, CART, XG Boost

Jeffery Lancon, Manisha Pednekar, Andrew Walch, David Stroud

MSDS 7333 - Quantifying the World - Case Study # 6

02/19/2019

Methods

In order to get the data in the proper format for analyzation, we will use text mining so that our data will function properly with our algorithms. Analyzing the header and body of an email brings up a unique set of issues that must be dealt with prior to any modeling.

We begin by analyzing the collection of emails that have already be classified as ham or spam. Our data should be divided in the message, body and attachments. We eliminate attachments for this exercise, as they are not relevant to our analysis.

Data acquisition and data preparation

The message dataset, created by "SpamAssassin" (http://spamassasin.apache.org), is organized in five folders named: easy ham, easy_ham2, hard_ham, spam, spam_2. According to SpamAssassin's website, the messages are named by a message number and their MD5 checksum, see Figure 1.

Figure 1 - SpamAssassin Dataset folder organization and message files naming convention

```
> spamPath = "C:/Manisha/QTW/casestudy6/SpamAssassinMessages"
> list.dirs(spamPath, full.names = FALSE)
[1] ""           "easy_ham"   "easy_ham_2" "hard_ham"   "spam"       "spam_2"
> #Look at some of the files in the easy_ham folder
> head(list.files(path = paste(spamPath, "easy_ham", sep = .Platform$file.sep)))
[1] "00001.7c53336b37003a9286aba55d2945844c" "00002.9c4069e25e1ef370c078db7ee85ff9ac"
[3] "00003.860e3c3cee1b42ead714c5c874fe25f7" "00004.864220c5b6930b209cc287c361c99af1"
[5] "00005.bf27cdeaf0b8c4647ecd61b1d09da613" "00006.253ea2f9a9cc36fa0b1129b04b806608"
> #We check that the names of the messages in the spam directories look the same as those in the ham directories w
> head(list.files(path = paste(spamPath, "spam_2", sep = .Platform$file.sep)))
[1] "00001.317e78fa8ee2f54cd4890fdc09ba8176" "00002.9438920e9a55591b18e60d1ed37d992b"
[3] "00003.590eff932f8704d8b0fcbe69d023b54d" "00004.bdcc075fa4beb5157b5dd6cd41d8887b"
[5] "00005.ed0aba4d386c5e62bc737cf3f0ed9589" "00006.3ca1f399ccda5d897fecb8c57669a283"
```

A function was created to read the messages into R as character vectors. Next, the data needs to be formatted in order to make it useable for naive Bayes analysis which uses only the content of the message. Thus, we need to access the body of the message in order to extract its words. We eliminated the attachments from the body as we are not interested in this portion of the body [1]. Once we have located the relevant portion of the message body, our task is to extract its words. We are using natural language processing package in R to extract important words form the messages body. Thus, we had cleaned and extracted the words from all the email in the Spam Assassin corpus to implement the Naïve Bayes classifier for the text analysis.

Analysis

Our analysis will compare the performance of the Naïve Bayes, CART, and Extreme Gradient Boosting algorithms. Each of these choices are extremely popular in the data science community in measuring the accuracy of binary classification problems. Before we begin our analysis, we examine the list below to determine what algorithm to start with:

- *Number of training examples.*
- *Dimensionality of the feature space.*
- *Do I expect the problem to be linearly separable?*
- *Are features independent?*
- *Are features expected to linearly dependent with the target variable?*
- *Is overfitting expected to be a problem?*

- *What are the system's requirements in terms of speed/performance/memory usage etc?*

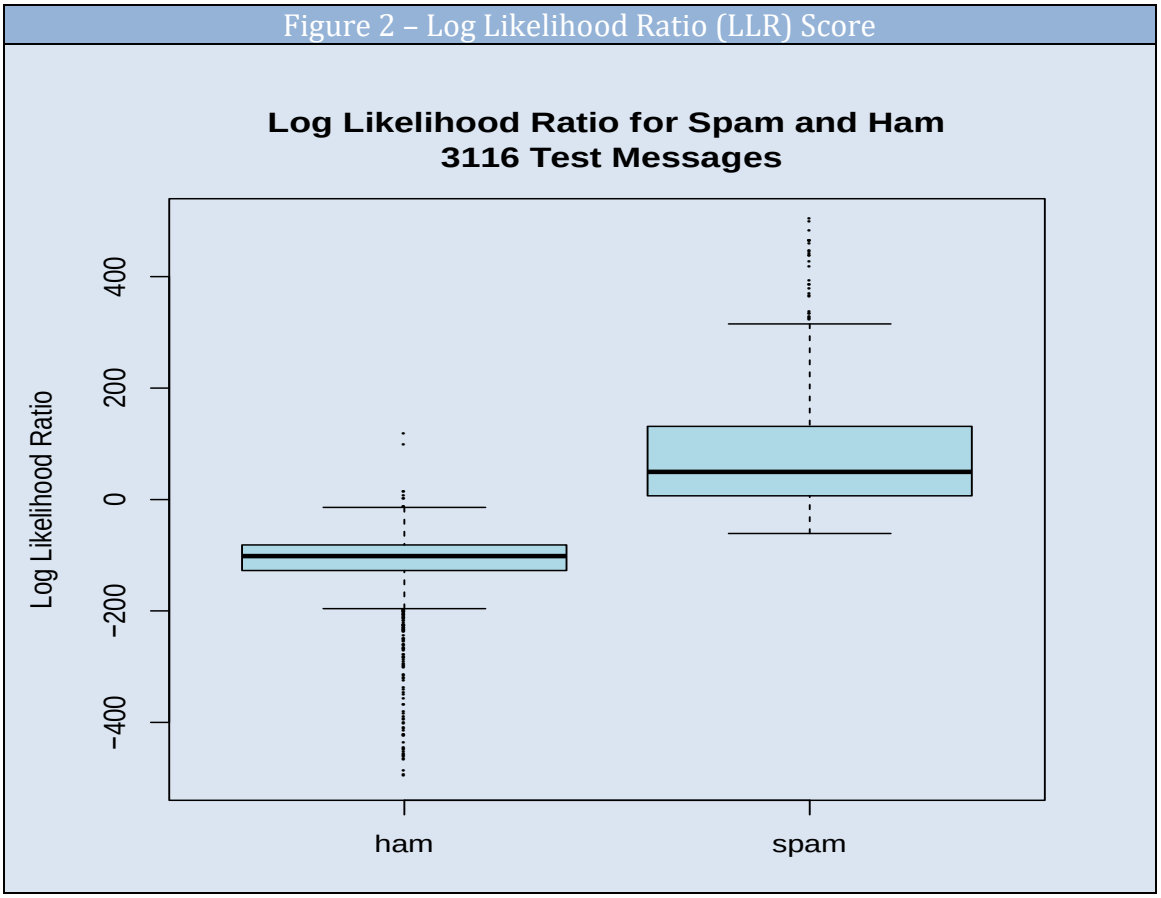
Naïve Bayes

Naïve Bayes is based on applying Bayes theorem to a set of supervised learning algorithms. The term 'Naïve' derives its meaning from the naïve assumption of conditional independence between every pair of features given the value of the class variable.

To properly classify the messages, we used a controlled sampling process to select from spam and ham messages to determine the proportion of spam in the test and training sets corpus. We used 2/3 of the data for training and 1/3 for testing. A value that is positive indicates spam is more likely and a negative value indicates ham is more likely. The number of unique words counted in the training set is 80,059.

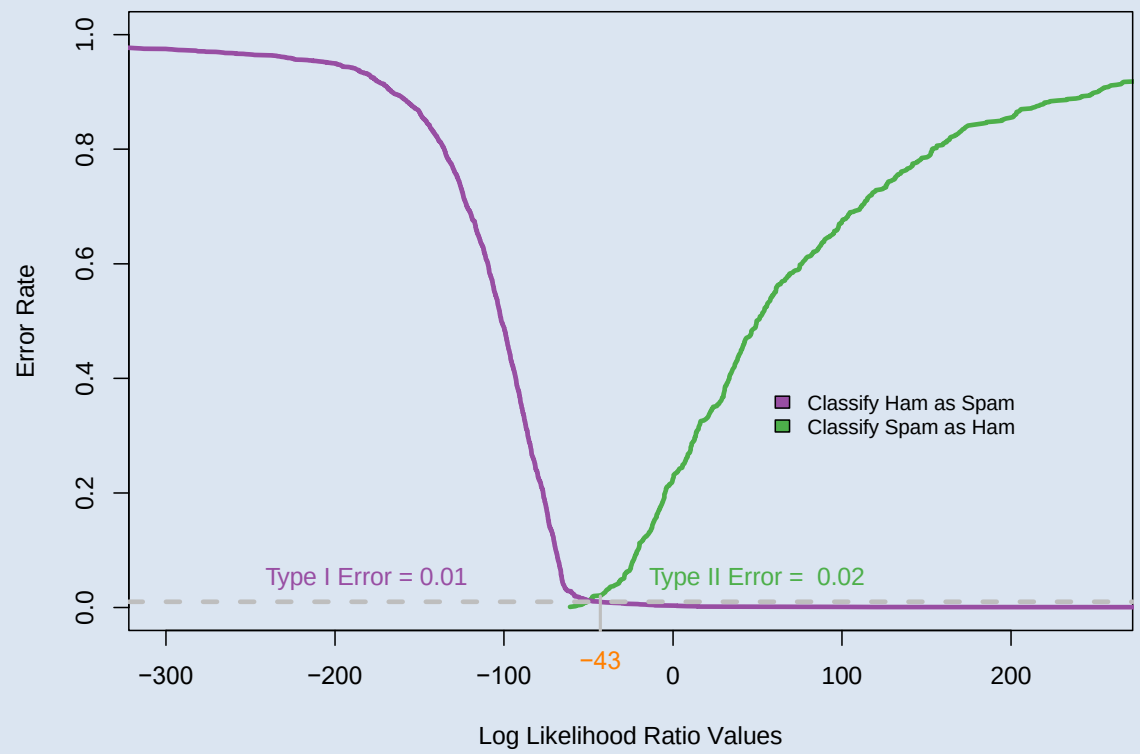
We applied the functions developed to estimate the probabilities that a word occurs in a message given it is spam or ham and used these probabilities to compute the log likelihood ratio for the messages in the training set. The sum of the log likelihood can be calculated by using the bag of words technique by comparing the training set with the test set. Figure 2 depicts the Log Likelihood Ratio (LLR) scores as it classifies spam and ham messages.

Figure 2 – Log Likelihood Ratio (LLR) Score



To determine the threshold of what should be classified as 'Ham or Spam' we need to search our dataset and find a 'cutoff'. Figure 3 indicates that an LLR of -43 would minimize Type I and Type II errors when classifying messages as 'Ham or Spam'. Note that we introduce Type I and Type II errors in the chart. These definitions will be covered more extensively in the section where we discuss confusion matrices.

Fig 3 - Threshold in determining if an email is Ham or Spam



For the Naïve Bayes algorithm, we used 3-fold cross validation on a total of 9,348 samples.

CART ~ Classification and Regression Trees

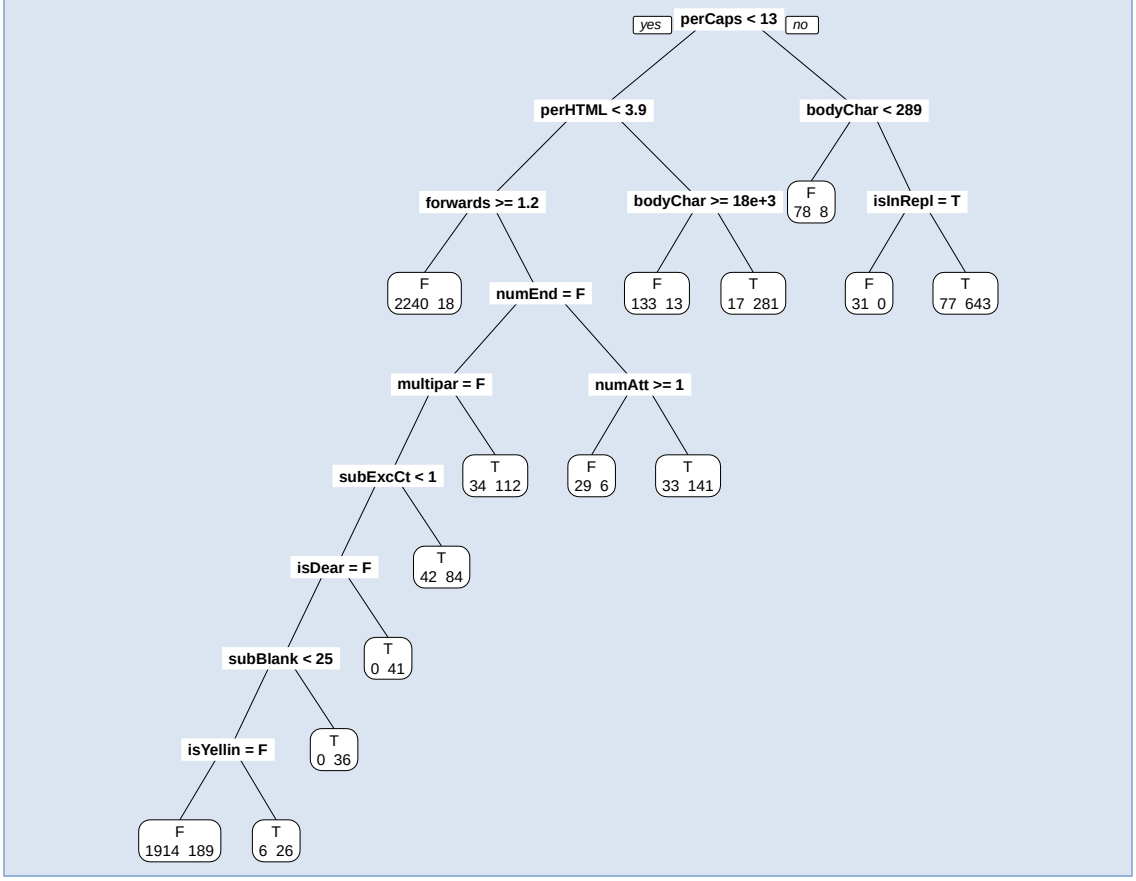
The CART algorithm is used to train Decision Trees, also known as 'growing' trees.

Decision Trees are non-parametric supervised learning methods used for classification and regression. The goal is to create a model that predicts the value of a target variable by learning simple decision rules inferred from the data features. Decision Trees require very little preparation and don't require feature scaling or centering.

The CART algorithm is a greedy algorithm that searches for an optimum split at the top level, then repeats the process at each level. It does not check whether the split will lead to the lower possible impurity several levels down. Figure 4 shows a recursive partition where the root of the tree splits the email into two groups. One reason for the ubiquity of decision trees in machine learning is their ease of interpretation. For example, the binary group in this instance splits where the percentage of

capital letters falls below 13 percent. As you follow the tree observe how the instances are partitioned as yes or no questions. The term decision tree is synonymous with recursion partition.

Fig 4 - Example Tree from a Recursive Partition



For the CART algorithm, we used 5-fold cross-validation on a total of 9,348 samples.

Extreme Gradient Boosting ~ XG Boost

XG Boost is one of the fastest implementations of gradient boosted trees. One of the inefficiencies of Gradient Boosting is the potential loss for all possible splits to create a new branch. XG Boost considers this inefficiency and looks for a distribution of features across all data points in a leaf. This information is used to reduce search space of possible feature splits, thus making XG Boost more efficient, substantially faster and less prone to overfitting.

In order to understand XG Boost, Figure 5, we must understand the format of the decision tree and consider that in using this algorithm that we are taking an ensemble of decision trees to deploy the algorithm. Random forests and boosted trees are technically the same mode and the difference between the two is how we train them.

- Boosting ~ fitting data by utilizing base learners and weak learners.
- Bootstrap Aggregation (Bagging) ~ used to reduce the variance for algorithms that have high variance.

Fig. 5 – Understanding Extreme Gradient Boosting ~ XG Boost

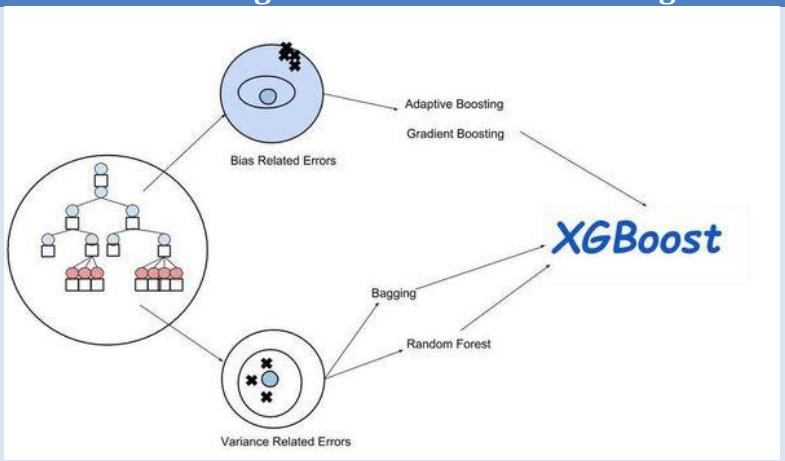


Fig. 6 – Important Variables for Spam Classification (XG Boost)

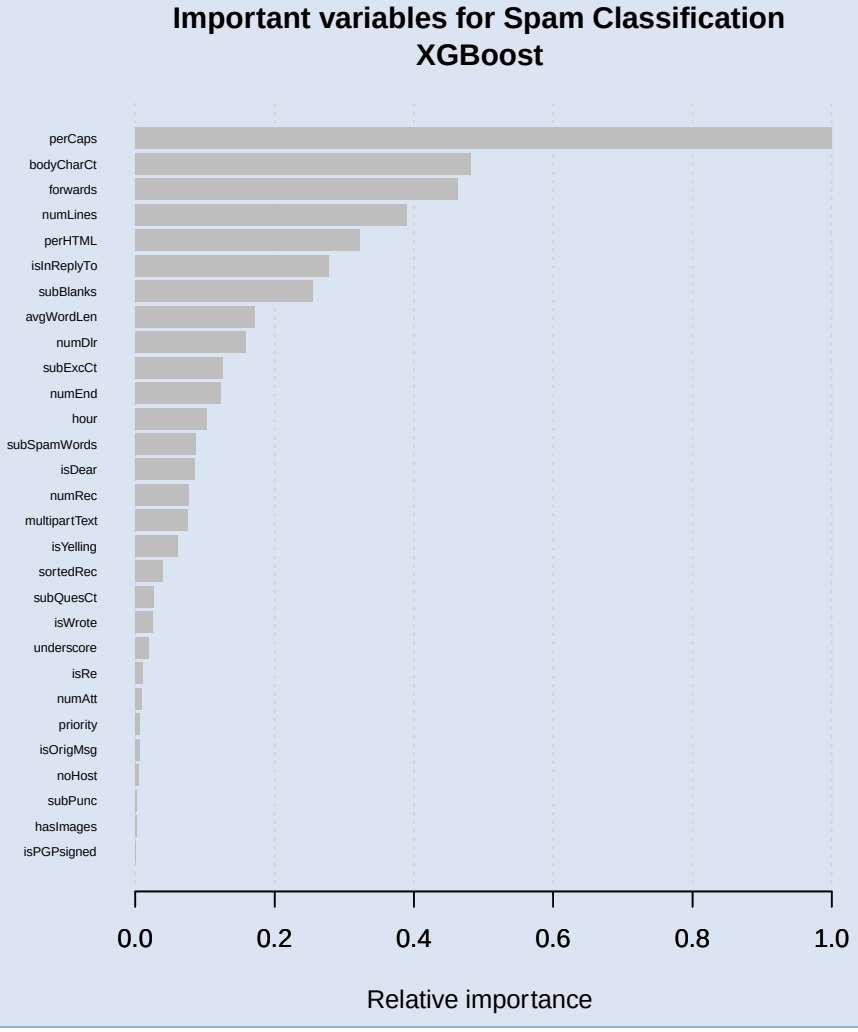
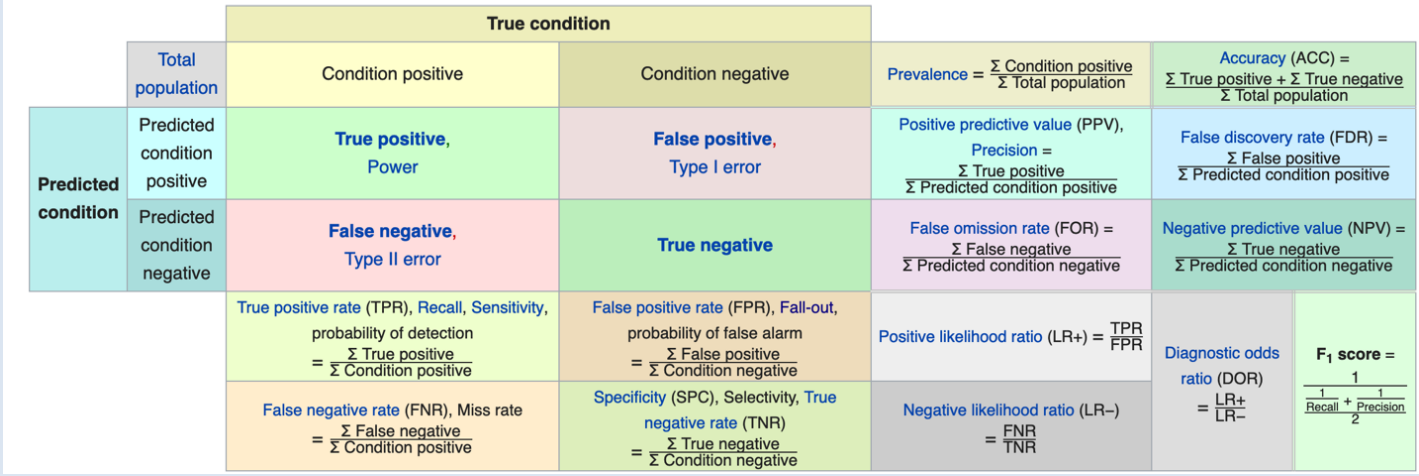


Figure 6 is breakdown of the most important variables used for Spam Classification with the XG Boost package in R. This intuitive chart is particularly welcome as one of the few complaints users of XB Boost have is the results being delivered in a 'black box' format.

Measurement of Analysis ~ Confusion Matrix

A Confusion Matrix is a clean and unambiguous way to present prediction results for a classification problem. The Confusion matrix is not a performance measure as such, but most performance metrics are based on Confusion Matrices and the numbers within them, see Figure 7. We will use the F1 score as the determining factor when considering which model performs best.

Fig. 7 – Model Performance Measures



- Type I Error ~ Ham is caught by spam filter.
- Type II Error ~ Spam is not caught by spam filter.
- Precision ~ The number of correct positive results divided by the number of positive results predicted by the classifier.
- Recall ~ It is the number of correct positive results divided by the number of ***all*** relevant samples.
- F_1 Score ~ The F_1 score is the harmonic average of the precision and recall. An F_1 score of 1 indicates perfect precision and recall. While and F_1 of 0 is the lowest score a model can achieve.

Table 1 – Performance Comparison of Naïve Bayes, CART, and XG Boost

	F_1 Score	Precision	Recall	Type I Error	Type II Error
Bayes Theorem	.9120	.9379	.8899	.0449	.0818
CART	.9612	.9581	.9601	.0313	.0265
XG Boost	.9794	.9733	.9856	.0201	.0106

Looking at the performance comparison table, Table 1, the XG Boost algorithm is the superior solution. While the CART algorithm comes in at a respectable second, there is a gulf of difference

between Bayes Theorem and XB Boost. We leave the reasons for the difference in algorithm performance for further research.

As mentioned previously, we based our conclusions on the algorithm with the highest F_1 score. F_1 scores compare the accuracy of a binary classifier and are often used in the field of text mining as a means of measuring accuracy. The F_1 score is designed to measure precision, how many instances are classified correctly, and robustness, not missing a significant number of instances. Our models have a nice combination of precision and recall in each instance. Therefore, we are comfortable using the F_1 score as our means of measurement and XG Boost as the superior performing algorithm.

Future Work

Case Study – Additional Analysis

Naïve Bayes(NB) – Message Body Analysis

The first part of NTL's case study 'Using Statistics to Identify Spam' uses Naïve Bayes classification analysis to determine if a message was ham vs spam. The analysis separated each individual word in the body of the message and created features using each unique word in the message body. This resulted in (80,000+) features, to determine the likelihood that the message was spam or ham. This analysis could be improved in 2 ways; a) included count of number of times word appears in the body of the message and b) use of short phrases in addition to individual words.

The current analysis only identified unique words present in a message. While this is a good start, this leaves a lot of information 'on-the-table'. Knowing the number of times, a word appears in a message could possible improve the classification process. Take for instance the word 'cash'. This word can easily show up in both ham and spam messages. But, if the word 'cash' appears an excessive amount of times, that could indicate it may be spam. To normalize the number of times a word appears, it should be calculated as a percentage of total words, not an individual count.

The NB analysis of the body text used individual words. Again, this simple analysis leaves a lot of information 'on-the-table'. Use of not only individual words but use of short phrases would improve the analysis. Simple phrases as example: 'Nigerian Prince', 'cash back', 'free shipping', 'as seen on TV' would improve the identification of spam messages. A search of the internet quickly revealed numerous digital marketing sites that advertise lists of words/phrases that marketers should avoid so that their marketing messages have a better chance to not be flagged by anti-spam software [6][7].

Naïve Bayes – Header Analysis

NTL's spam classification case study divided the messages into two broad categories: message body and message header. The case study was performed in 2 separate analyses. The first part of the case study used Naïve Bayes analysis on the body of the message while the 2nd part of the case study used CART analysis on the header of the message. An analysis of both sections of the messages, body and header, concurrently, should improve the classification results. This classification can either be performed as an ensemble of the two methods or performed using a single methodology (Bayes, CART).

CART – Message Body Analysis

Like the NB analysis, CART analysis focused only on one part of the messages, mainly the header. Looking at both the header and body should improve classification accuracy.

Case Study – Future Studies

Creation of White/Black List

A first line of defense would be the creation of White List and Black list for messages. A white list would include known message sender whose messages are known hams and can be forwarded to the recipient without further analysis.

Similarly, the creation of a black list would include message senders, identified by a range of email addresses and/or IP address range, whose messages are suspected and/or know to be spam. These could be flagged as spam prior to any additional analysis.

Only messages that aren't on either the white or black list would be forwarded for further analysis, thus saving time and overhead.

Attachment Analysis

Analyzing attachments can increase the likelihood of detecting spam. Databases of often used imagery have been created (like Google Image search) which can tell you how rampant the use of an image is. If an image tends to be a 'stock' image that is often associated with spam emails, Fig. 8A & B, then we can consider this a flag for this message possibly being spam.

Fig. 8A - Viagra Image – Spam email

Fig. 8B Payday Loan Image – Spam email

Modernize Database

The SpamAssassin database of spam and ham messages is relatively small. The database only includes 9000+ messages. This sample size is probably too small to get a full picture of message traffic that is encountered in a normal application. Additional observations need to be gathered and analyzed for improved performance.

With spam traffic accounting for over 50% of all traffic, SpamAssassin's database makeup is not representative of the ratio of spam/ham messages that would be typical in a production environment. The database has a 75:25 ham-to-spam ratio. This lopsided ratio may account for the Low Type I error by sub-par Type II errors.

As we have mentioned, Spammer's tactics are constantly evolving. SpamAssassin's message database is out-of-date. Many of the messages in the database date back to 2002. While some of the common threads are the same, much more advance techniques have developed. To truly attempt to classify spam messages, the database must be updated with recent messages.

Discussions

Identifying spam messages has been an ongoing problem society has been attempting to solve since the invention of email. Both the sophistication of detection algorithms and the skill level and tactics of the spammers have continued to advance.

Spammer's tactics are constantly shape-shifting, to avoid detection, and detection software is adapted to the new threats. This is a classic game of cat-and-mouse. Detection algorithms/systems must be constantly updated to defend against an ever-changing foe. Entire companies/websites[8] are dedicated to helping digital marketers get their messages past the anti-spam software filters, devising ever more sophisticated strategies to defeat the spam filters.

Most efforts in spam detection are defensive in nature. A message is received by a mail service and the service analyzes the message for telltale signs/signatures that this message is either spam or ham.

Detection Tactics – Improvements

Many of the topic covered in this section are beyond the scope of the review of NTL's case study 'Using Statistics to Identify Spam' but should be part of any 'good' spam message detection/reduction strategy. It is often said that 'the best defense is a good offense'. The exact origin of this saying is debated but the ideology can be traced back to the early writings of Sun Tzu's 'The Art of War', some 2500 years ago.

Offensive Strategy

Most corporations do not have official policies and procedures for the proper use of electronic messaging systems, which includes emails. Beyond just the cursory warning not to use the system for non-business uses, most just leave it up to user to figure it out. With the thinking of; the best defense is a good offense; this will need to change. Everyone; IT professionals, users, managers, and support staff all need to contribute to this strategy to help detect and eliminate spam messages. Below are a few policies/procedures that would benefit this strategy:

a) Training employees on how to manage their emails. Simple actions that can reduce the number of spam messages that may target the company's mail system
    a. Don't sign up for every little thing that comes across their email box. Once an email address is on a mailing list, it is almost impossible to truly remove it. Mailing list are often shared/sold to entities that look to profit from sending out spam messages.
    b. Keep email addresses as private as possible. Do not post them online, where scrapping algorithms can easily obtain them. When posting online, say an email marketing campaign, use a designated email address, other than your won, to receive message traffic. Once the marketing campaign is complete, the address can be frozen or terminated.

c. Don't answer survey or polls that ask personal questions. Doing so exposes the user to not only spam but possibly phishing attacks.
d. Teach users how to identify phishing attempts. Phishing attacks can not only put the user at risk but also the entire company. Loss of confidentiality of a password can open the entire network to unforeseen damages. As phishing attacks get increasing difficult to detect, more effort must be made to train the users as well as the system for detecting them before they ever reach the users.

b) Have users actively flag spam messages. Most individuals identify spam messages and quickly delete them. Effort should be made to teach users to flag spam messages instead of deleting them. By flagging the messages, it is possible to improve the quality and quantity of positively identified spam messages. The larger dataset would possibly improve the classification ability of the spam filter algorithm.

Ethics

Several activities associated with spam message detection may run a fowl with the ethics of viewing, processing messages, which may or may not be of a personal nature. While many corporations have policies in place that allow them to monitor activity on company assets (computers), there is a 'grey zone' when it comes to emails. Privacy issues can arise from processing the message's body. While the algorithm doesn't 'read' the messages, some aspects of the message's content could be exposed during the process.

Another ethically suspect scenario could be when creating a white list of cleared message senders. Often, these safe senders list is created by peering at user's contact lists. If a user has a sender's email address in their contacts, it can be assumed that it is a valid sender. This again can lead to privacy violation issues.

Bias

The SpamAssassin database may contain biases. From the literature, it is noted that the SpamAssassin database was assembled for the purpose of developing and testing spam filters. Understanding more about the database may shed light on possible bias issues within the database.

Were these messages selected at random or hand-picked to ensure good results during analysis?

What context were these messages received?

…

Conclusion

Electronic messaging had become an essential form of business and personal communication. Spam messages are unsolicited electronic messages which are sent to recipients. Spam messages are a nuisance, but also can be dangerous, accounting for 53.5% of email traffic worldwide. Because of the sheer volume of electronic messages sent/received each data, an automated solution to identifying and eliminating spam messages is required.

Automated spam filters examine various characteristics of a message to determine if the message is spam or ham. Message characteristics are examined using statistical analysis. NTL's original case study implemented Naïve Bayes and Recursive Partitioning and Classification Trees (CART). Our extended analysis utilized Random Forest and XG Boost.

Our analysis indicates that out of the algorithms we tested, XG Boost algorithm performed best. For comparison we utilized F1 statistics, which is designed to measure precision. F1 statistic for XG Boost was 0.9794 while the original CART analysis was 0.9612. Given this dataset and depth of analysis undertaken, XG Boost algorithm appears to be a good algorithm for automated spam filters.

Spammer's tactics are constantly evolving to avoid detection, and detection software must adapt to new threats. Detection algorithms/systems must be constantly updated to defend against an ever-changing foe.

References

[1] Nolan, Deborah and Lang, Duncan Temple; from: "Data Science in R: A Case Studies Approach to Computational Reasoning and Problem Solving" Chapman & Hall/CRC ©2015

[2] "Using Statistics to Identify Spam"; from: http://rdatasciencecases.org/Spam/code.R

[3] "SpamAssassin Corpus"; from: https://spamassassin.apache.org/old/publiccorpus/

[4] The Radicati Group, Inc; Email Market, 2018-2022; from: https://www.radicati.com/wp/wp-content/uploads/2018/01/Email_Market,_2018-2022_Executive_Summary.pdf

[5] "Global spam volume as percentage of total e-mail traffic from January 2014 to September 2018, by month", from; https://www.statista.com/statistics/420391/spam-email-traffic-share/

[6] "The Ultimate List of Spam Trigger Words"; from https://blog.hubspot.com/blog/tabid/6307/bid/30684/the-ultimate-list-of-email-spam-trigger-words.aspx

[7] "The Ultimate Spam Trigger Words List: 474 Keywords to Avoid in 2019"; from https://www.automational.com/spam-trigger-words-to-avoid/

[8] "11 reasons why your emails go in the spam box and how to make sure they don't"; from: https://optinmonster.com/11-reasons-why-your-emails-go-in-the-spam-box-and-how-to-make-sure-they-dont/

https://arxiv.org/pdf/1410.5329.pdf

https://homes.cs.washington.edu/~tqchen/pdf/BoostedTree.pdf

https://medium.com/@gabrieltseng/gradient-boosting-and-xgboost-c306c1bcfaf5

https://xgboost.readthedocs.io/en/latest/tutorials/model.html

https://en.wikipedia.org/wiki/Email_spam

Appendix A

R Code

```
#  MSDS 7333 - Quantifying the World - Case Study #6
#  Spam Detection - Using Statistic Analysis
#  Team Members:
#         Jeffery Lancon, Manisha Pednekar, Andrew Walch, David Stroud
#  Date: 02/19/2019
#  Case Study from: Data Science in R: Nolan,Temple,Lang (Ch 3)
#  Initial source code: http://rdatasciencecases.org/code.html


# Setting the current working directory to the current file location
library(rstudioapi)
current_path <- getActiveDocumentContext()$path
setwd(dirname(current_path ))


#------------------ Set-up and Exploration of Data ------------------
#--------------------------------------------------------------------

# To programmatically located the  emails, the files
# are stored in a subdirectory /SpamAssassinMessages/Messages/.....
# There are 5 directories easy_ham,easy_ham_2,hard_ham,spam, and spam_2

spamPath = "./SpamAssassinMessages/"

list.dirs(spamPath, full.names = FALSE)
# [1] ""                "Messages"          "Messages/easy_ham"   "Messages/easy_ham_2"
"Messages/hard_ham"
# [6] "Messages/spam"      "Messages/spam_2"

# Listing folders in Message directory
list.files(path = paste(spamPath, "messages",
                        sep = .Platform$file.sep))
# [1] "easy_ham"   "easy_ham_2" "hard_ham"   "spam"      "spam_2"

# examime the first few file names in spam_2 foloder
# The files are named by message number and their MD5 checksum hash
head(list.files(path = paste(spamPath, "messages", "spam_2",
                             sep = .Platform$file.sep)))
# [1] "00001.317e78fa8ee2f54cd4890fdc09ba8176" "00002.9438920e9a55591b18e60d1ed37d992b"
```

```
# [3] "00003.590eff932f8704d8b0fcbe69d023b54d" "00004.bdcc075fa4beb5157b5dd6cd41d8887b"
# [5] "00005.ed0aba4d386c5e62bc737cf3f0ed9589" "00006.3ca1f399ccda5d897fecb8c57669a283"

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

# Check to see how many files located in the Messages directories
length(list.files(paste(spamPath, "messages", dirNames,
                    sep = .Platform$file.sep)))
# [1] 9353

# Getting a count of files in each of Messages subdirectories
sapply(paste(spamPath, "messages", dirNames,
             sep = .Platform$file.sep),
       function(dir) length(list.files(dir)) )
# ./SpamAssassinMessages//messages/easy_ham ./SpamAssassinMessages//messages/easy_ham_2
# 5052                                 1401
# ./SpamAssassinMessages//messages/hard_ham       ./SpamAssassinMessages//messages/spam
# 501                                  1002
# ./SpamAssassinMessages//messages/spam_2
# 1398

# Creating a list of directory names/paths for use in retrieving data
fullDirNames = paste(spamPath, "messages", dirNames,
                     sep = .Platform$file.sep)
    # ------ Test Code -----
    # List first message/file from messages/easy_ham
    fileNames = list.files(fullDirNames[1], full.names = TRUE)
    fileNames[1] #[1]
"./SpamAssassinMessages//messages/easy_ham/00001.7c53336b37003a9286aba55d2945844c"
    # Read and display the first few lines of the first message in easy_ham
    msg = readLines(fileNames[1])
    head(msg)
    # [1] "From exmh-workers-admin@redhat.com  Thu Aug 22 12:36:23 2002"
    # [2] "Return-Path: <exmh-workers-admin@spamassassin.taint.org>"
    # [3] "Delivered-To: zzzz@localhost.netnoteinc.com"
    # [4] "Received: from localhost (localhost [127.0.0.1])"
    # [5] "\tby phobos.labs.netnoteinc.com (Postfix) with ESMTP id D03E543C36"
    # [6] "\tfor <zzzz@localhost>; Thu, 22 Aug 2002 07:36:16 -0400 (EDT)"

    # Selecting 15 files from easy_ham directory to read
    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) # Creates a list of 15 character vectors

    # Subsetting out first message in easy_ham directory for analysis
    msg = sampleEmail[[1]]
    which(msg == "")[1] # Identifying first blank line in message (This is to find the first blank line,
                # by using the [1], in the message, which indicates the separation between header
                # and body of email) Without index [1], would return a index of all blank lines
    #[1] 63

    match("", msg) # Another way to identifying the first blank line in message/vector)

    # going to create a splitpoint, using the first blank line as the argument
    splitPoint = match("", msg)

    #Splitting message into header/body, using splitPoint index as the argument
    # Creating msg using the last 2 lines of header and first 6 lines of body
    msg[ (splitPoint - 2):(splitPoint + 6) ]
      # [1] "List-Archive: <https://listman.spamassassin.taint.org/mailman/private/exmh-workers/>"
      # [2] "Date: Thu, 22 Aug 2002 18:26:25 +0700"
      # [3] ""
      # [4] "    Date:       Wed, 21 Aug 2002 10:54:46 -0500"
      # [5] "    From:       Chris Garrigues <cwg-dated-1030377287.06fa6d@DeepEddy.Com>"
      # [6] "    Message-ID:  <1029945287.4797.TMDA@deepeddy.vircio.com>"
      # [7] ""
      # [8] ""
      # [9] "  | I can't reproduce this error."

    header = msg[1:(splitPoint-1)] # Creating Header of message: everything above splitPoint
    body = msg[ -(1:splitPoint) ] # Creating Body of message: everything below SplitPoint

#---------- splitMessage function () - Rev A ----------
# Split message into two character vectors (header and body) list, using 1st blank line
# as the split Point
splitMessage = function(msg) {
  splitPoint = match("", msg)
  header = msg[1:(splitPoint-1)]
  body = msg[ -(1:splitPoint) ]
  return(list(header = header, body = body))
}
#---
```

```
# ------ Test Code -----
sampleSplit = lapply(sampleEmail, splitMessage) #Testing code on list of 15 previously select
messages
head(sampleSplit[[1]]$header)
head(sampleSplit[[2]]$body)

header = sampleSplit[[1]]$header
grep("Content-Type", header) #identifying index for line containing 'Content-Type' of message #1
# [1] 46
grep("multi", tolower(header[46]))  # determing if message has attachments (noted by Content-Type
Key value:
                                    # equaling 'multi'), Uses tolower to make text all lowercase.
# integer(0) - 'multi' not found in line 46; therefore, no attachments
header[46]
# [1] "Content-Type: text/plain; charset=us-ascii"

# Function to determines the content type of all 15 test emails
headerList = lapply(sampleSplit, function(msg) msg$header)
CTloc = sapply(headerList, grep, pattern = "Content-Type")
print(unname(CTloc)) # Does not work properly becuase the 7th element does not have a 'content-
Type' Key
# [[1]]
# [1] 46
# [[2]]
# [1] 45
# ......
# [[6]]
# [1] 54
# [[7]]
# integer(0)
# [[8]]
# [1] 21
# ........

# Add a check to deal with missing 'Content-Type' keys
sapply(headerList, function(header) {
  CTloc = grep("Content-Type", header)
  if (length(CTloc) == 0) return(NA)
  CTloc
})
```

```
# Creation of Boolean variable to determine if message contains an attachment
hasAttach = sapply(headerList, function(header) {
  CTloc = grep("Content-Type", header)
  if (length(CTloc) == 0) return(FALSE)
  grepl("multi", tolower(header[CTloc])) # grepl() returns a logic value
})

print(unname(hasAttach)) # Prints out the bollena values for wether it has an attachment or not. Used unname to reduce clutter
# [1] FALSE FALSE FALSE FALSE FALSE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE

#### Attempting to locate 'boundry=' string from those messages that have attachments
header = sampleSplit[[6]]$header # We know message 6 includes an attachment
boundaryIdx = grep("boundary=", header) # Finding 'boundry=' within the header
# [1] 55
header[boundaryIdx]
# [1] "    boundary=\"==_Exmh_-1317289252P\";"

sub(".*boundary=\"(.*)\";.*", "\\1", header[boundaryIdx]) # We want to extract the boundry value, which follows the boundry
# [1] "==_Exmh_-1317289252P"                    # statement and is grouped, using the ( ) which must be followed
#?sub()                                  # by ''; giving access to characters located within the ( ) using
                                        # //1 notatoin.  Complete description on pg 119

# Try regex on other emails to check flexibility
# Using message #9
header2 = headerList[[9]]
boundaryIdx2 = grep("boundary=", header2)
# [1] 17
header2[boundaryIdx2]
# [1] "Content-Type: multipart/alternative; boundary=Apple-Mail-2-874629474"  ## Pattern is different from message #6

sub('.*boundary="(.*)";.*', "\\1", header2[boundaryIdx2])  # Did not find boundry string correctly due to different patterns
# [1] "Content-Type: multipart/alternative; boundary=Apple-Mail-2-874629474"

# Dropping quotation marks from the boundry string, if we eliminate quotation marks, we can remove them from
# our pattern search as well
boundary2 = gsub('"', "", header2[boundaryIdx2])
```

```
    sub(".*boundary= *(.*);?.*", "\\1", boundary2) # modified sub statement from earlier, to not serach for quotation marks
    # [1] "Apple-Mail-2-874629474"

    # Going back and double checking our first one
    boundary = gsub('"', "", header[boundaryIdx])
    sub(".*boundary= *(.*);?.*", "\\1", boundary) # Its not working properly now.
    # [1] "==_Exmh_-1317289252P;"

    #We have to modify search slightly, to correct the pattern issue [^;] means except semicolon
    sub(".*boundary= *([^;]*);?.*", "\\1", boundary)
    # [1] "==_Exmh_-1317289252P"

###### Finalized getBoundry () Function - Rev A
getBoundary = function(header) {
  boundaryIdx = grep("boundary=", header)
  boundary = gsub('"', "", header[boundaryIdx])
  gsub(".*boundary= *([^;]*);?.*", "\\1", boundary)
}
#----


    testingBoundry <- sapply(headerList, getBoundary)
    print(unname(testingBoundry))



    # We are now ready to search through the body of the message
    # for attachments
    # Will use message 6 as example:

    # Dispalying Message 6: Cotains attachment
    sampleSplit[[6]]$body
      # [1] "--==_Exmh_-1317289252P"
      # [2] "Content-Type: text/plain; charset=us-ascii"
      # [3] ""
      # [4] "> From:  Chris Garrigues <cwg-exmh@DeepEddy.Com>"
      # [5] "> Date:  Wed, 21 Aug 2002 10:40:39 -0500"
      # [6] ">"
      # ......
      # [43] "  World War III:  The Wrong-Doers Vs. the Evil-Doers."
```

```
# [44] ""
# [45] ""
# [46] ""
# [47] ""
# [48] "--==_Exmh_-1317289252P"
# [49] "Content-Type: application/pgp-signature"
# [50] ""
# [51] "-----BEGIN PGP SIGNATURE-----"
# [52] "Version: GnuPG v1.0.6 (GNU/Linux)"
# [53] "Comment: Exmh version 2.2_20000822 06/23/2000"
# [54] ""
# [55] "iD8DBQE9ZQJ/K9b4h5R0IUIRAiPuAJwL4mUus5whLNQZC8MsDlGpEdKNrACcDfZH"
# [56] "PcGgN9frLIM+C5Z3vagi2wE="
# [57] "=qJoJ"
# [58] "-----END PGP SIGNATURE-----"
# [59] ""
# [60] "--==_Exmh_-1317289252P--"
# [61] ""
# [62] ""
# [63] ""
# [64] "_____"
# [65] "Exmh-workers mailing list"
# [66] "Exmh-workers@redhat.com"
# [67] "https://listman.redhat.com/mailman/listinfo/exmh-workers"
# [68] ""

sampleSplit[[14]]$body # Another example
# [1] "This is a multi-part message in MIME format."
# [2] ""
# [3] "------=_NextPart_000_0005_01C26412.7545C1D0"
# [4] "Content-Type: text/plain;"
# [5] "\tcharset=\"iso-8859-1\""
# [6] "Content-Transfer-Encoding: 7bit"
# [7] ""
# [8] "liberalism"
# ....
# [27] " http://www.english.upenn.edu/~afilreis/50s/schleslib.html"
# [28] ""
# [29] "------=_NextPart_000_0005_01C26412.7545C1D0"
# [30] "Content-Type: application/octet-stream;"
# [31] "\tname=\"Liberalism in America.url\""
# [32] "Content-Transfer-Encoding: 7bit"
```

```
  # [33] "Content-Disposition: attachment;"
  # [34] "\tfilename=\"Liberalism in America.url\""
  # [35] ""
  # [36] "[DEFAULT]"
  # [37] "BASEURL=http://www.english.upenn.edu/~afilreis/50s/schleslib.html"
  # [38] "[InternetShortcut]"
  # [39] "URL=http://www.english.upenn.edu/~afilreis/50s/schleslib.html"
  # [40] "Modified=E0824ED43364C201DE"
  # [41] ""
  # [42] "------=_NextPart_000_0005_01C26412.7545C1D0--"
  # [43] ""
  # [44] ""
  # [45] ""

sampleSplit[[11]]$body # Does not contain an attachment
  # [1] ""
  # [2] "--------------090602010909000705010009"
  # [3] "Content-Type: text/plain; charset=ISO-8859-1; format=flowed"
  # [4] "Content-Transfer-Encoding: 8bit"
  # [5] ""
  # [6] "Geege wrote:"
  # .....
  # [63] "Check out the pictures."
  # [64] ""
  # [65] ""
  # [66] ""
  # [67] ""
  # [68] "--------------090602010909000705010009--"
  # [69] ""
  # [70] ""

# NOTE: There are 2 occurrences of the boundry string: One before and one after
# the attachment

sampleSplit[[15]]$body
  # [1] ""
  # [2] "--------------080209060700030309080805"
  # [3] "Content-Type: text/plain; charset=US-ASCII; format=flowed"
  # [4] "Content-Transfer-Encoding: 7bit"
  # [5] ""
  # [6] "I actually thought of this kind of active chat at AOL (in 1996 I think), "
  # ...........
```

```
# [34] ""
# [35] "--------------080209060700030309080805"
# [36] "Content-Type: text/html; charset=US-ASCII"
# [37] "Content-Transfer-Encoding: 7bit"
# [38] ""
# [39] "<!DOCTYPE html PUBLIC \"-//W3C//DTD HTML 4.01 Transitional//EN\">"
# ........
# [69] ""
# [70] "Yuck"
# [71] "  </pre>"
# [72] "</blockquote>"
# [73] "<br>"
# [74] "</body>"
# [75] "</html>"
# [76] ""
# [77] "--------------080209060700030309080805--"
# [78] ""
# [79] ""


## We will begin the process of creating a function that does the following:
## a) Drop the blank lines before the first boundry string
## b) Keep the lines following as part of the first portion of the body and not attachment
## c) Use the last line of the email as the end of the attachment if we find no closing boundry string

#Examining message #15 as our test message
boundary = getBoundary(headerList[[15]])
# [1] "------------080209060700030309080805"
body = sampleSplit[[15]]$body

bString = paste("--", boundary, sep = "") # Search body for boundry string preceeded by 2 hyphens
# [1] "--------------080209060700030309080805"
bStringLocs = which(bString == body)
bStringLocs
#[1]  2 35

eString = paste("--", boundary, "--", sep = "") # Search for closing boundry string with preceeding and
post by 2 hyphens
# [1] "--------------080209060700030309080805--"
eStringLoc = which(eString == body)
eStringLoc
# [1] 77
```

```
    # We locate the first part of the message from the body, minus the attachments
    msg = body[ (bStringLocs[1] + 1) : (bStringLocs[2] - 1)]
    tail(msg,5)
     # [28] ">Yuck"
     # [29] ">  "
     # [30] ">"
     # [31] ""
     # [32] ""

    # Adding the lines after the attachment to the message
    msg = c(msg, body[ (eStringLoc + 1) : length(body) ])
    tail(msg,7)
     # [28] ">Yuck"
     # [29] ">  "
     # [30] ">"
     # [31] ""
     # [32] ""
     # [33] ""
     # [34] ""


#---------- dropAttach function () - Rev A ----------
# Function encompasses the techniques shown above for
# removing Attachments from the body portion of the
# message
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) # no exit string
    return(body[ (bStringLocs[1] + 1) : (bStringLocs[2] - 1)])

  n = length(body)
  if (eStringLoc < n) # concatinates bottom of message to body if end string
                      # is not located at end of body
    return( body[ c( (bStringLocs[1] + 1) : (bStringLocs[2] - 1),
```

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

  return( body[ (bStringLocs[1] + 1) : (bStringLocs[2] - 1) ]) #If eStringLoc = n, body is defined as
                                        # message between the bStrings only
}
#--------

    # Trying function on message 15
    msg15 =dropAttach(sampleSplit[[15]]$body, getBoundary(headerList[[15]]) )
      # [1] "Content-Type: text/plain; charset=US-ASCII; format=flowed"
      # [2] "Content-Transfer-Encoding: 7bit"
      # [3] ""
      # [4] "I actually thought of this kind of active chat at AOL (in 1996 I think), "
      # ......
      # [26] ">\"Oh, you're going to Seattle? I can get you airline tickets for less\""
      # [27] ">"
      # [28] ">Yuck"
      # [29] ">  "
      # [30] ">"
      # [31] ""
      # [32] ""
      # [33] ""
      # [34] ""


# ------ 3.5.3 Extracting Words from the Message Body ----------
    # Displaying Message 1
    head(sampleSplit[[1]]$body)
      # [1] "    Date:       Wed, 21 Aug 2002 10:54:46 -0500"
      # [2] "    From:       Chris Garrigues <cwg-dated-1030377287.06fa6d@DeepEddy.Com>"
      # [3] "    Message-ID:  <1029945287.4797.TMDA@deepeddy.vircio.com>"
      # [4] ""
      # [5] ""
      # [6] "  | I can't reproduce this error."

    # Displaying Message 3
    msg = sampleSplit[[3]]$body
    head(msg)
      # [1] "Man Threatens Explosion In Moscow "
      # [2] ""
      # [3] "Thursday August 22, 2002 1:40 PM"
      # [4] "MOSCOW (AP) - Security officers on Thursday seized an unidentified man who"
```

```
    # [5] "said he was armed with explosives and threatened to blow up his truck in"
    # [6] "front of Russia's Federal Security Services headquarters in Moscow, NTV"

  # Selecting lines 1,3,26,27 to extract txt from: testing
  msg[ c(1, 3, 26, 27) ]
    # [1] "Man Threatens Explosion In Moscow "        "Thursday August 22, 2002 1:40 PM"
    # [3] "4 DVDs Free +s&p Join Now"
"http://us.click.yahoo.com/pt6YBB/NXiEAA/mG3HAA/7gSolB/TM"

  # Converting all txt to lowercase; Discord punctuations and numbers and replace them with spaces
  cleanMsg = tolower(gsub("[[:punct:]0-9[:blank:]]+", " ", msg))
  cleanMsg[ c(1, 3, 26, 27) ]
    # [1] "man threatens explosion in moscow "          "thursday august pm"
    # [3] " dvds free s p join now"                     "http us click yahoo com pt ybb nxieaa mg haa gsolb tm"

# Loading 'Text Mining Package'
# https://cran.r-project.org/web/packages/tm/tm.pdf
# Simple example of how to handle plural, possessive and
# tm package includes a list of stopwords' words that do not add value to classification
# and appear often in text (a, an, me, my, we, same,....)
library(tm)
stopWords = stopwords()
cleanSW = tolower(gsub("[[:punct:]0-9[:blank:]]+", " ", stopWords))
SWords = unlist(strsplit(cleanSW, "[[:blank:]]+"))
SWords = SWords[ nchar(SWords) > 1 ]
stopWords = unique(SWords)


  # Creating a vector of words from the cleanMsg. utilizing 'blanks' for the split variable
  words = unlist(strsplit(cleanMsg, "[[:blank:]]+"))
  length(words) #[1] 272
  # Dropping 0 or 1 letter words
  words = words[ nchar(words) > 1 ]
  length(words) #[1] 260
  # Removal of any words that are in the stopWords
  words = words[ !( words %in% stopWords) ]
  length(words) #[1] 173
  head(words)
    # [1] "man"      "threatens" "explosion" "moscow"    "thursday"  "august"

  #---------- cleanText function () - Rev A ----------
  # Function removes punctuation and numbers from message
```

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

#---------- findMsgWords function () - Rev A ----------
# splits message into individual words and collects unique words
# strips stopWords from words list
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 () - Rev A ----------
# Reads all files, splits message into body,header
# Strips attachments; Eliminates non-email messages
# Splits message into individual words and collects unique words
# strips stopWords from words list.
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)
}
# --------

# Processing all messages from SpamAssain database, using
# messages in directories;creating a words list of 5 character
# vectors, each containing the words list for each message in the subdirectory
msgWordsList = lapply(fullDirNames, processAllWords,
                      stopWords = stopWords)
# Displaying 1st list (easy_ham directory), 12th message [message
00011.fbcde1b4833bdbaaf0ced723edd6e355]
msgWordsList[[1]][12]
 # [1] "hello"       "seen"        "discussed"   "article"     "approach"    "thank"       "http"        "www"
 # [9] "paulgraham"  "com"         "spam"        "html"        "hell"        "rules"       "trying"      "accomplish"
 # [17] "something"  "thomas"      "alva"        "edison"      "sf"          "net"         "email"       "sponsored"
 # [25] "osdn"       "tired"       "old"         "cell"        "phone"       "get"         "new"         "free"
```

```
 # [33] "https"       "inphonic"    "asp"         "sourceforge"  "refcode"     "vs"          "spamassassin"
"devel"
 # [41] "mailing"     "list"        "lists"       "listinfo"

# We want to classify each message as spam or ham
# we create a counter of the number of msgs in each folder
numMsgs = sapply(msgWordsList, length)
numMsgs
 # [1] 5051 1400  500 1000 1397

# Create a Boolean value for is spam for each message
isSpam = rep(c(FALSE, FALSE, FALSE, TRUE, TRUE), numMsgs)

# we flatten the list into a single list
msgWordsList = unlist(msgWordsList, recursive = FALSE)
length(msgWordsList) #[1] 9348
msgWordsList[[8000]]
isSpam[8000]

############################################################
################ Beginning of Naive Bayes Classifier
############################################################

numEmail = length(isSpam) #[1] 9348
numSpam = sum(isSpam) #[1] 2397 (Using True values of isSpam)
numHam = numEmail - numSpam #[1] 6951

set.seed(418910)

# Using sample() function to split the messages into test and traning datasets
# Want to maintain the % split between Spam and Ham that is present in the overall
# dataset
# creating a list of indexes chosen at random for 1/3 of the values of each
# type of message (spam,ham)
testSpamIdx = sample(numSpam, size = floor(numSpam/3))
testHamIdx = sample(numHam, size = floor(numHam/3))

# Note: All spam messages are first then all ham messages are 2nd.  THis will come in handy in just a
little bit
testMsgWords = c((msgWordsList[isSpam])[testSpamIdx],   # msgWordsList[isSpam] only selects from
messages that correspond
                                  # with isSpam is True. We then subdivide these messages by the indexes
```

```
                                    # chosen to be part of the test dataset [testSpamIdx]
          (msgWordsList[!isSpam])[testHamIdx] )  # We repeat this process for the
msgWordsList[!isSpam] from messages
                                    # that are not spam !isSpam and subdivide these ham messages by the
indexes
                                    # chosen to be part of the test dataset [testHamIdx]
length(testMsgWords) #[1] 3116

trainMsgWords = c((msgWordsList[isSpam])[ - testSpamIdx], # Conduct a similar subdivision of
messages, this time taking all of
           (msgWordsList[!isSpam])[ - testHamIdx]) # the remaining spam and ham messages and placing
them in training dataset
length(trainMsgWords) #[1] 6232

#Creating an indicator/classification variable for test messages [Utilizing the Spam than Ham ordering]
testIsSpam = rep(c(TRUE, FALSE), # repeats TRUE for length of testSpamIdx and FALSE for length of
testHamIdx
          c(length(testSpamIdx), length(testHamIdx)))
length(testIsSpam) #[1] 3116

trainIsSpam = rep(c(TRUE, FALSE), # repeats TRUE for remaining spam messasge (numSpam-
length(testSpamIdx)) indexes and FALSE for remaining ham messages
           c(numSpam - length(testSpamIdx),
             numHam - length(testHamIdx)))
length(trainIsSpam) #[1] 6232

# Creating a bag of words (BOW) of all unique words that appear in the messages
bow = unique(unlist(trainMsgWords))
length(bow) # 80059  (slightly different from the book's value 80481)

# Initiating a count list for all unique words that appear in the messages
spamWordCounts = rep(0, length(bow))
length(spamWordCounts) # [1] 80059

# Creating a named numeric vector for each unique word in messages
names(spamWordCounts) = bow
head(spamWordCounts)
  # doctype    html     public       dtd  transitional      en
  # 0          0        0            0    0             0

# We process each Spam message and retrive only the unique words. tmp is a list of 1598 elements of
character vectors (equal
```

```
# to the number of trainIsSpam messages that are categorized as spam)
tmp = lapply(trainMsgWords[trainIsSpam], unique)

# Since we are using unique words, the values in tt represent the frequency of spam messages that the
particular word
# appears in. Using a frequenct table we sums up the number of instances across all messages
tt = table( unlist(tmp) )
tt[1:5]
    # \aÿ□ëa   \033m  \036þw \177ízë  \177ùþ (gibberish becusae we did not prune the words to only be
dictionary words)
    # 1      2      2      1      4

#We update spamWordCount with th frequency values from the frequency table
spamWordCounts[ names(tt) ] = tt
head(spamWordCounts)
    # doctype       html      public       dtd transitional         en
    # 87        811        141         85         75        170

#---------- computeFreqs function () - Rev A ----------
# Compute the frequency of each word's appearance in spam and ham messages.
# Computes the present and absent log odds and place them in a matrix
# Inputs bow, trainIsSpam(noting which messages are spam)
# and  wordsList for all messages in training dataset
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)
 }
#------

# Compute Frequwncy dataframe for messages
trainTable = computeFreqs(trainMsgWords, trainIsSpam)
  class(trainTable) # [1] "matrix"
  dim(trainTable) # [1]    4 80059
  trainTable[1:4,200:205]
      #              linux       ie       irish      users     group     mailman
      # spam          0.04347826  0.06537379  0.04410385  0.07100407  0.0603691  0.08977166
      # ham           0.17013702  0.12482468  0.10648398  0.22753264  0.1992664  0.34577624
      # presentLogOdds -1.36434303 -0.64678885 -0.88144752 -1.16455653 -1.1941652 -1.34852252
      # absentLogOdds   0.14204291  0.06572245  0.06748501  0.18451460  0.1599588  0.33024606

  # ----------------- Test Code ----------------
  # Now we can construct the log likelihood ratio for a message.  We Sum the
  # presentLogOdds for words that appear in the message and the Sum of absentLogOdds for words not
  # in the message
  newMsg = testMsgWords[[1]] #Taking message 1 for example

  newMsg = newMsg[!is.na(match(newMsg, colnames(trainTable)))] #Drop any words that are not in
BOW
  length(newMsg) #[1] 451

  present = colnames(trainTable) %in% newMsg #Boolean variable to indicate if words that are in
message (TRUE or FALSE)
  sum(present == TRUE) # [1] 451          # out of BOW
```

```
  # Compute the log of the probability a message is spam vs ham
  sum(trainTable["presentLogOdds", present]) +
    sum(trainTable["absentLogOdds", !present])
    # [1] 255.0476

  # We know the first message of testMsgWords is spam, so 255.04 seems correct.
  # Now, lets calculate the logliklihood odds of a ham message

  newMsg = testMsgWords[[ which(!testIsSpam)[1] ]] # First ham message in the list (67 words)
  newMsg = newMsg[!is.na(match(newMsg, colnames(trainTable)))] #Drop any words that are not in
BOW
  length(newMsg) #[1] 67
  present = (colnames(trainTable) %in% newMsg)

  # Compute the log of the probability a message is spam vs ham
  sum(trainTable["presentLogOdds", present]) +
    sum(trainTable["absentLogOdds", !present])
    # [1] -124.3275  Very low / negative number; therefore, message has a high probability of being ham



#---------- computeMsgLLR function () - Rev A ----------
# Creating a function to compute the log likelihood of a message being
# spam.
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])
}
#-----

# Calling ComputeMsgLLR function to compute the LLR for all messages in the trainTable
testLLR = sapply(testMsgWords, computeMsgLLR, trainTable)

# Creating summary data for training dataset by spam classification TRUE or FALSE
```

```
tapply(testLLR, testIsSpam, summary)
  # $`FALSE`
  #    Min.  1st Qu.   Median     Mean  3rd Qu.    Max.
  # -1361.89  -127.06  -101.18  -116.25   -81.26   700.23
  #
  # $`TRUE`
  #    Min.   1st Qu.    Median     Mean   3rd Qu.     Max.
  # -60.574    6.369    49.837   137.546   131.719 23518.028

#------- Figure 3.1 - Boxplot LogLikelihood by true Classification ----
# Plots summary data for loglikelihood for Training Dataset
pdf("./Figures/Fig3.1_SP_Boxplot.pdf", width = 6, height = 6)
spamLab = c("ham", "spam")[1 + testIsSpam] #Converting spam T/F label to ham & spam
boxplot(testLLR ~ spamLab, cex = .4, pch=19, col = 'lightblue',
      ylab = "Log Likelihood Ratio",
      main = "Log Likelihood Ratio for Spam and Ham\n 3116 Test Messages",
      ylim=c(-500, 500))

dev.off()
#-----



#---------- typeIErrorRate function () - Rev A ----------
#Create function to determine the Type I error rate for
# certain values of tau
typeIErrorRate =
 function(tau, llrVals, spam)
 {
  classify = llrVals > tau
  sum(classify & !spam)/sum(!spam)
 }
#----

# Try tau of 0 for test dataset
typeIErrorRate(0, testLLR,testIsSpam)
  #[1] 0.003452741

# Try tau of -20 for test dataset
typeIErrorRate(-20, testLLR,testIsSpam)
  #[1] 0.005610703
```

```
#----------- typeIErrorRate function () - Rev B ----------
#Create function to determine the Type I error rate for
# The function looks at the llrVals values for ham messages
# and recognizing that the number of Type 1 errors decrease by
# 1 at each of these values and so i/(number of ham messages)
typeIErrorRates =
  function(llrVals, isSpam)
  {
    o = order(llrVals) #Increasing order of index
    llrVals =  llrVals[o]
    isSpam = isSpam[o]

    idx = which(!isSpam) #is not Spam
    N = length(idx)
    list(error = (N:1)/N, values = llrVals[idx]) # Creats a list of xI$error and xI$values
  }
#-----

#----------- typeIIErrorRate function () - Rev A ----------
#Create function to determine the Type II error rate for
# The function looks at the llrVals values for ham messages
# and recognizing that the number of Type II errors decrease by
# 1 at each of these values and so i/(number of spam messages)
typeIIErrorRates = function(llrVals, isSpam) {

  o = order(llrVals)
  llrVals =  llrVals[o]
  isSpam = isSpam[o]


  idx = which(isSpam) #is Spam
  N = length(idx)
  list(error = (1:(N))/N, values = llrVals[idx]) # Creats a list of xII$error and xII$values
}
#----

xI = typeIErrorRates(testLLR, testIsSpam)
xII = typeIIErrorRates(testLLR, testIsSpam)
tau01 = round(min(xI$values[xI$error <= 0.01])) # Can adjust Type I error rate; tau value will be adjusted
t2 = max(xII$error[ xII$values < tau01 ])
```

```
pdf("./Figures/Fig3.2_LinePlotTypeI+IIErrors.pdf", width = 8, height = 6)
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.off()
#----

####---- K Fold Cross Valaidation -----------------
k = 5
numTrain = length(trainMsgWords)
partK = sample(numTrain)
tot = k * floor(numTrain/k)
partK = matrix(partK[1:tot], ncol = k) # converting to matrix of k columns

testFoldOdds = NULL
for (i in 1:k) {
  foldIdx = partK[ , i]
  trainTabFold = computeFreqs(trainMsgWords[-foldIdx], trainIsSpam[-foldIdx])
  testFoldOdds = c(testFoldOdds,
              sapply(trainMsgWords[ foldIdx ], computeMsgLLR, trainTabFold))
}

testFoldSpam = NULL
for (i in 1:k) {
  foldIdx = partK[ , i]
```

```
  testFoldSpam = c(testFoldSpam, trainIsSpam[foldIdx])
}

xFoldI = typeIErrorRates(testFoldOdds, testFoldSpam)
xFoldII = typeIIErrorRates(testFoldOdds, testFoldSpam)
tauFoldI = round(min(xFoldI$values[xFoldI$error <= 0.01]))
tFold2 = xFoldII$error[ xFoldII$values < tauFoldI ]


######### Should plot results of K-Fold CV into a graph ********
testFoldOdds[2]

        ############## Sample Code - Not implemented
        smallNums = rep((1/2)^40, 2000000)
        largeNum = 10000

        print(sum(smallNums), digits = 20)

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

        for (i in 1:length(smallNums)) {
          largeNum = largeNum + smallNums[i]
        }
        print(largeNum, digits = 20)

        sampleSplit = lapply(sampleEmail, splitMessage)


######################### Section 3.7 Recursive Partitioning and Classification Trees
####################
#############################################################################################
###################
#############################################################################################
###################

# Reprocessing Dataset from Scratch
# We will reuse some of the functions created in the Naive Bayes section
# to process the messages for Recursive Partitioning
#  We will be using the message header for the majority of this analysis section
# see page 140 for additional details about strategy for handling/parsing messages
```

```
    #------- Test Code ---------
  # Reusing sampleSplit variable from NB
  header = sampleSplit[[1]]$header
  header[1:12] # Displaying
   # [1] "From exmh-workers-admin@redhat.com  Thu Aug 22 12:36:23 2002"
   # [2] "Return-Path: <exmh-workers-admin@spamassassin.taint.org>"
   # [3] "Delivered-To: zzzz@localhost.netnoteinc.com"
   # [4] "Received: from localhost (localhost [127.0.0.1])"
   # [5] "\tby phobos.labs.netnoteinc.com (Postfix) with ESMTP id D03E543C36"
   # [6] "\tfor <zzzz@localhost>; Thu, 22 Aug 2002 07:36:16 -0400 (EDT)"
   # [7] "Received: from phobos [127.0.0.1]"
   # [8] "\tby localhost with IMAP (fetchmail-5.9.0)"
   # [9] "\tfor zzzz@localhost (single-drop); Thu, 22 Aug 2002 12:36:16 +0100 (IST)"
   # [10] "Received: from listman.spamassassin.taint.org (listman.spamassassin.taint.org
[66.187.233.211]) by"
   # [11] "    dogma.slashnull.org (8.11.6/8.11.6) with ESMTP id g7MBYrZ04811 for"
   # [12] "    <zzzz-exmh@spamassassin.taint.org>; Thu, 22 Aug 2002 12:34:53 +0100"

# The plan is to process the message header by converting the key:value pairs in a named vector
# Observations:
  # Some key:value pairs appear on multiple lines
  # First line is not key:value format
  # Colons appear in the value portion of time Key:value pair


  # Processing line 1 of header using regex.  Replacing 'From' with 'Top-From:' using regex
  # Sustituting a value that can be read as a key:value pair
  header[1] = sub("^From", "Top-From:", header[1]) # processing line 1 of header using regex
  header[1] #[1] "Top-From: exmh-workers-admin@redhat.com  Thu Aug 22 12:36:23 2002"


  #?textConnection()
  #?read.dcf()
  headerPieces = read.dcf(textConnection(header), all = TRUE)
  # Using read.dcf() in combo with textConnection() function, converts the header
  # into a dataframe with 26 variables, one for each Key:value pair
  headerPieces[, "Delivered-To"]
  # Delivered-To is a variable in headerPieces dataframe with a list of values that match the key
'Delivered-To'
    #[1] "zzzz@localhost.netnoteinc.com"           "exmh-workers@listman.spamassassin.taint.org"
  headerPieces[,'Delivered-To'][[1]][2]
    #[1] "exmh-workers@listman.spamassassin.taint.org"
```

```
    headerVec = unlist(headerPieces) # converted to a named character vector (When a key has a list of values,
                                     # it breaks them up into separate named character vectors)
    head(headerVec)
      # Top-From
      # "exmh-workers-admin@redhat.com  Thu Aug 22 12:36:23 2002"
      # Return-Path
      # "<exmh-workers-admin@spamassassin.taint.org>"
      # Delivered-To1
      # "zzzz@localhost.netnoteinc.com"
      # Delivered-To2
      # "exmh-workers@listman.spamassassin.taint.org"
      # Received1
      # "from localhost (localhost [127.0.0.1])\nby phobos.labs.netnoteinc.com (Postfix) with ESMTP id D03E543C36\nfor <zzzz@localhost>; Thu, 22 Aug 2002 07:36:16 -0400 (EDT)"
      # Received2
      # "from phobos [127.0.0.1]\nby localhost with IMAP (fetchmail-5.9.0)\nfor zzzz@localhost (single-drop); Thu, 22 Aug 2002 12:36:16 +0100 (IST)"

    # creates a named vector with the number of values(instances) for each Key
    dupKeys = sapply(headerPieces, function(x) length(unlist(x)))
    head(dupKeys)
      # Top-From  Return-Path Delivered-To    Received     From     To
      #   1           1           2          10         1        1

    # Renames headerVec named vector with proper names , using dupKeys values for
    # number of times each name is repeated (removing Deli...To1, Deli...To2)
    names(headerVec) = rep(colnames(headerPieces), dupKeys)

    headerVec[ which(names(headerVec) == "Delivered-To") ]
      # Delivered-To                        Delivered-To
      # "zzzz@localhost.netnoteinc.com" "exmh-workers@listman.spamassassin.taint.org"

    length(headerVec) #[1] 36

    length(unique(names(headerVec))) #[1] 26

#---------- processHeader function () - Rev A ----------
# Create function that performs the initial parsing of the
# message header:
```

```
# Convert 1st line to key:value format
# process header to key:value matrix using read.dcf function
# unlist header into vector matrix and rename the named character
# vectors with proper names (removing Del..To1, Del..To2 in naming convention)
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)
}
#----

# using lapply to process all 15 headers within sampleSplit, breaking them into named character vectors
headerList = lapply(sampleSplit,
                    function(msg) {
                      processHeader(msg$header)} )


   # extracting the contentTypes from the headerList and storing it as a variable
   contentTypes = sapply(headerList, function(header)
     header["Content-Type"])
        #
./SpamAssassinMessages//messages/easy_ham/00001.7c53336b37003a9286aba55d2945844c.Content-
Type
        # "text/plain; charset=us-ascii"
        #
./SpamAssassinMessages//messages/easy_ham/00002.9c4069e25e1ef370c078db7ee85ff9ac.Content-
Type
        # "text/plain; charset=US-ASCII"
        #
./SpamAssassinMessages//messages/easy_ham/00003.860e3c3cee1b42ead714c5c874fe25f7.Content-
Type
        # "text/plain; charset=US-ASCII"

   names(contentTypes) = NULL # removing variable names
```

```
head(contentTypes)
    # [1] "text/plain; charset=us-ascii"
    # [2] "text/plain; charset=US-ASCII"
    # [3] "text/plain; charset=US-ASCII"
    # [4] "text/plain; charset=\"us-ascii\""
    # [5] "text/plain; charset=US-ASCII"
    # [6] "multipart/signed;\nboundary=\"==_Exmh_-1317289252P\";\nmicalg=pgp-
sha1;\nprotocol=\"application/pgp-signature\""


##############################################
### 3.8.2  Processing Attachments #########

  # Identifying which messages have attachments using contentType containing 'multi', similar to NB
  hasAttach = grep("^ *multi", tolower(contentTypes))
  hasAttach
   # [1]  6  8  9 10 11 12 13 14 15

  # Using getBoundry function from NB section, to retrive boundry string
  boundaries = getBoundary(contentTypes[ hasAttach ])
  boundaries
  # [1] "==_Exmh_-1317289252P"    "----=_NextPart_000_00C1_01C25017.F2F04E20"   "Apple-Mail-2-
874629474"
  # [4] "==_Exmh_-518574644P"     "------------090602010909000705010009"       "==_Exmh_-
451422450P"
  # [7] "==_Exmh_267413022P"      "----=_NextPart_000_0005_01C26412.7545C1D0"   "------------
080209060700030309080805"

  boundary = boundaries[9] # Using message #15 (9th in boundries list) as example
  body = sampleSplit[[15]]$body

  # Using same delimiters we used for NB to locate attachments by searching for the beginning boundry
strings
  bString = paste("--", boundary, sep = "") #"--------------080209060700030309080805"
  bStringLocs = which(bString == body)
  bStringLocs #[1]  2 35  (Lines 2 and 35 mark the start of the body and start of the single attachment)

  # We next find the ending boundry string
  eString = paste("--", boundary, "--", sep = "") #[1] "--------------080209060700030309080805--"
  eStringLoc = which(eString == body)
  eStringLoc #[1] 77
```

```
  # Size of Body/Attachment
  diff(c(bStringLocs[-1], eStringLoc)) #[1] 42


#---------- processAttach function () - Rev A ----------
# Create function that returns a list with 2 elements
# a) the first part of the body (message and no attachment)
# b) a dataframe attachDF with 2 variables aLen, aType
#    length and type of attachment

### This code has mistakes in it - and we fix them later!
processAttach = function(body, contentType){

 boundary = getBoundary(contentType)

 bString = paste("--", boundary, "$", sep = "")
 bStringLocs = grep(bString, body)

 eString = paste("--", boundary, "--$", sep = "")
 eStringLoc = grep(eString, body)

 n = length(body)

 if (length(eStringLoc) == 0) eStringLoc = n + 1 #no end strings present
 if (length(bStringLocs) == 1) attachLocs = NULL #no attachments if bstringLoc ==1 (only body)
 else attachLocs = c(bStringLocs[-1],  eStringLoc) # initializing attachment boundry locations

 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)))
}
#------

  # Still using 15 messages in sampleSplit as test code
  bodyList = lapply(sampleSplit, function(msg) msg$body)

  # contains a list of 9
  attList = mapply(processAttach, bodyList[hasAttach],
           contentTypes[hasAttach],
           SIMPLIFY = FALSE)
  # examining the attachment lengths
  lens = sapply(attList, function(processedA)
    processedA$attachDF$aLen)
  head(lens)
    # $`./SpamAssassinMessages//messages/easy_ham/00014.cb20e10b2bfcb8210a1c310798532a57`
    # [1] 12
    # $`./SpamAssassinMessages//messages/easy_ham/00062.009f5a1a8fa88f0b38299ad01562bb37`
    # [1] 44 44
    # $`./SpamAssassinMessages//messages/easy_ham/00063.0acbc484a73f0e0b727e06c100d8df7b`
    # [1] 83
    # $`./SpamAssassinMessages//messages/easy_ham/0030.77828e31de08ebb58b583688b87524cc`
    # [1] 12
    # $`./SpamAssassinMessages//messages/easy_ham/00368.f86324a03e7ae7070cc40f302385f5d3`
    # NULL
    # $`./SpamAssassinMessages//messages/easy_ham/00389.8606961eaeef7b921ce1c53773248d69`
    # [1] 12

  attList[[2]]$attachDF
    #    aLen       aType
    # 1   44     text/html
    # 2   44     <META http-equiv=3DContent-Type content=3Dtext/html; =
  # In message 2, Attachment 2: It appears that the 'Content-Type' appears to be
  # located in an HTML tag, see below (issue needs to addressed in function call)
```

```
body = bodyList[hasAttach][[2]]
length(body)  #[1] 86
body[35:45] #Look at lines [3] and [9]
   # [1] ""
   # [2] "------=_NextPart_000_00C1_01C25017.F2F04E20"
   # [3] "Content-Type: text/html;"
   # [4] "\tcharset=\"Windows-1252\""
   # [5] "Content-Transfer-Encoding: quoted-printable"
   # [6] ""
   # [7] "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 Transitional//EN\">"
   # [8] "<HTML><HEAD>"
   # [9] "<META http-equiv=3DContent-Type content=3D\"text/html; ="
   # [10] "charset=3Dwindows-1252\">"
   # [11] "<META content=3D\"MSHTML 6.00.2716.2200\" name=3DGENERATOR>"



#---------- processAttach function () - Rev B ----------
# updates function processAttach to account for
# issue with 'Content-Type' appearing in multiple places
# and improve error handling/special cases
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 () - Rev A ----------
# Creates a function that encapsulates the tasks of:
# Retrieving names of file directories
# Drops any non emails "cmds' in file name
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")
}
#-------
```

```
#---------- readEmail function () - Rev A ----------
# Creates a function that encapsulates the tasks of:
# Reads all messages in readline format
# Splits messages to Body/Header
# Processes Header
#     Extracts content key:value
# Extracts body
# Identifies which messages have attachments
# Obtain stats for body/attachment
# Envokes functions: readEmail(), splitMessages(),
#       procesHeader(), processAttach(),
# Returns a matrix with appropriate variables
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)
}
#----

# We process all messages in all 5 folders using processAllEmail function
# passing the fullDirNames and isSpam classification to the function
emailStruct = mapply(processAllEmail, fullDirNames,
                 isSpam = rep( c(FALSE, TRUE), 3:2))
emailStruct = unlist(emailStruct, recursive = FALSE)
# Very large list 9348 elements, 101 Mb
emailStruct[[1]]

# Saves dataset into rda file format for use later
save(emailStruct, file="./Data/emailXX.rda")



################### Section 3.9 Deriving Variables from email messages ###

#Used previous 15 message's indexes to help us
# begin to create a feature set
indx #[1]   1 2 3 4 5 15  27  68  69 329 404 427 516 852 971
sampleStruct = emailStruct[ indx ]
```

```
header = sampleStruct[[1]]$header # Named character vector with 36
  headKeys = sapply(header, function(x) length(unlist(x)))
  headKeys
    # Top-From      Return-Path     Delivered-To     Delivered-To         Received
    # 1             1               1                1                1
    # Received      Received        Received         Received         Received
    # 1             1               1                1                1
    # Received      Received        Received         Received         From
    # 1             1               1                1                1
    # To            Cc              Subject          In-Reply-To      References
    # 1             1               1                1                1
    # MIME-Version  Content-Type    Message-Id       X-Loop           Sender
    # 1             1               1                1                1
    # Errors-To     X-Beenthere X-Mailman-Version    Precedence       List-Help
    # 1             1               1                1                1
    # List-Post   List-Subscribe    List-Id  List-Unsubscribe     List-Archive
    # 1             1               1                1                1
    # Date
    # 1

  # Example: Grabbing subject line
  subject = header["Subject"] # "Re: New Sequences Window"
  els = strsplit(subject, "") #Splitting into individual letters

  # Determining if all letters in Subject line are all caps 'LETTERS'
  # returns a T/F boolean
  all(els %in% LETTERS) # [1] FALSE

  # Creating a few test subject lines that we would want to flag having all
  # capital letters
  testSubject = c("DEAR MADAME", "WINNER!", "")

  els = strsplit(testSubject, "")
  sapply(els, function(subject) all(subject %in% LETTERS))
    # [1] FALSE FALSE  TRUE  (Not working properly)

  # Regex replacement of puctuations, replacing with nothing
  gsub("[[:punct:] ]", "", testSubject)
    # [1] "DEARMADAME" "WINNER"     ""

  # Eliminates all non-alpha characters
```

```
  gsub("[^[:alpha:]]", "", testSubject)
    # [1] "DEARMADAME" "WINNER"     ""


#---------- isYelling function () - Rev A ----------
# Creates a function that determines if subject line
# has all Capital letters. Subject line pulled from msg$header["Subject"]
# Returns a logical TRUE/FALSE
# If subject is empty, returns NA
isYelling = function(msg) {
 if ( "Subject" %in% names(msg$header) ) {
  el = gsub("[^[:alpha:]]", "", msg$header["Subject"])
  if (nchar(el) > 0)
   nchar(gsub("[A-Z]", "", el)) < 1
  else
   FALSE
 } else
  NA
}
#----


#---------- perCaps () - Rev A ----------
# Creates a function that determines the percentage
# of capital letters in the body of a message.
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)
 }
#----

# Determining % of alpha characters in the body of a message (using sampleStruct)
sapply(sampleStruct, perCaps)
```

```
  #[1]  4.451039   7.491289   7.436096   5.090909   6.116643   7.732865   5.523256
  #[8]  10.059172  10.885806  6.472492   9.597258   11.970075  9.233792   1.655629
  #[15] 6.417910


# Creating a list of functions that quanity attributes that
# help us classify the messages as either spam or ham
# Placing them in a list make it easier to call them in a functional
# call later
# Includes:  isRe, isYelling, numLines, perCaps
# output is a Data frame
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)
 }
)
#-----
```

```
# Creation of a dataframe with 4 variables that help classify
# the message
lapply(funcList, function(func)
 sapply(sampleStruct, function(msg) func(msg)))

  # $isRe
  # TRUE  FALSE  FALSE  FALSE  TRUE  TRUE  TRUE  FALSE
  # TRUE  TRUE  TRUE  TRUE  TRUE  FALSE  TRUE
  #
  # $numLines
  # 50  26  38  32  31  54  35  36
  # 65  58  70  31  38  28  34
  #
  # $isYelling
  # FALSE  FALSE  FALSE  FALSE  FALSE  FALSE  FALSE  FALSE
  # FALSE  FALSE  FALSE  FALSE  FALSE  FALSE  FALSE
  #
  # $perCaps
  # 4.451039  7.491289  7.436096  5.090909  6.116643  7.732865  5.523256  10.059172
  # 10.885806  6.472492  9.597258  11.970075  9.233792  1.655629  6.417910



#---------- createDerivedDF () - Rev A ----------
# Creates a function that evaluates derived variables
# from the message and stores them in a dataframe
# includes all functions listed in funcList()
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)
 }
#-----

  # Using sampleStruct to confirm function works properly
  sampleDF = createDerivedDF(sampleStruct)
  print(head(sampleDF), row.names = FALSE)
    # isRe numLines isYelling  perCaps
    # TRUE      50    FALSE 4.451039
    # FALSE      26    FALSE 7.491289
    # FALSE      38    FALSE 7.436096
    # FALSE      32    FALSE 5.090909
    # TRUE      31    FALSE 6.116643
    # TRUE      54    FALSE 7.732865




################################################################################
############  Creating funcList that encompasses all 29 variables
############   in Table 3.1 pg 151
# Variables:  isSpam; isRe; numLines; bodyCharCt; underscore; subExcCt;
#   subQuesCt;  numAtt; priority; numRec; perCaps;  isInReplyTo;  sortedRec;
#   subPunc;  hour; multipartTxt; hadImage; is PGPsigned; perHTML;  subSpamWords
#   subBlanks;  noHost; numEnd; isYelling;  forwards; isOrigMgs; isDear;
#   isWrote;  avgWordLen; numDlr
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))
  }
)
#--------------

# Wordlist that has a very high probability of being spam message if
# the message includes these words
# NOTE: Only used in Subject of message, not body
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 - Rev A ----------
# Creates a function that create a list of message recipients in
# the TO:  CC:  and BCC: Fields
# Used in a few functions with funcList list
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)
  )
 }
#-------------

#####
##### Creating derived datafram from messages
##### This dataframe is the basis of our analysis.  All analysis will be
##### performed on these 30 variables
emailDF = createDerivedDF(emailStruct)

dim(emailDF)
  #[1] 9348   30 (9348 observations & 30 variables)

print(head(emailDF), row.names = FALSE)
  # isSpam  isRe numLines bodyCharCt underscore subExcCt subQuesCt numAtt priority numRec
perCaps
  # FALSE  TRUE      50      1554     FALSE       0       0     0   FALSE     2 4.451039
  # FALSE FALSE      26       873     FALSE       0       0     0   FALSE     1 7.491289
  # FALSE FALSE      38      1713     FALSE       0       0     0   FALSE     1 7.436096
  # FALSE FALSE      32      1095     FALSE       0       0     0   FALSE     0 5.090909
  # FALSE  TRUE      31      1021     FALSE       0       0     0   FALSE     1 6.116643
  # FALSE  TRUE      25       718     FALSE       0       0     0   FALSE     1 7.625272
  # isInReplyTo sortedRec subPunc hour multipartText hasImages isPGPsigned perHTML subSpamWords
  # TRUE      TRUE   FALSE   11       FALSE     FALSE      FALSE      0       FALSE
  # FALSE     TRUE   FALSE   11       FALSE     FALSE      FALSE      0       FALSE
  # FALSE     TRUE   FALSE   12       FALSE     FALSE      FALSE      0       FALSE
  # FALSE     TRUE   FALSE   13       FALSE     FALSE      FALSE      0       FALSE
  # FALSE     TRUE   FALSE   13       FALSE     FALSE      FALSE      0       FALSE
```

```
  # TRUE     TRUE   FALSE   13       FALSE    FALSE      FALSE      0      FALSE
  # subBlanks noHost numEnd isYelling  forwards isOrigMsg isDear isWrote avgWordLen numDlr
  # 12.50000  FALSE  FALSE    FALSE  0.000000    FALSE  FALSE   FALSE   4.376623     3
  # 8.00000  FALSE  FALSE    FALSE  0.000000    FALSE  FALSE   FALSE   4.555556     0
  # 8.00000  FALSE  FALSE    FALSE  0.000000    FALSE  FALSE   FALSE   4.817164     0
  # 18.91892  FALSE  FALSE    FALSE  3.125000    FALSE  FALSE   FALSE   4.714286     0
  # 15.21739  FALSE  FALSE    FALSE  6.451613    FALSE  FALSE   FALSE   4.234940     0
  # 15.21739  FALSE  FALSE    FALSE 12.000000    FALSE  FALSE   FALSE   3.956897     0



# Saving emailDF data to file for future reference
save(emailDF, file = "./Data/spamAssassinDerivedDF.rda")



###################### Stopping point ###################################
load("./Data/spamAssassinDerivedDF.rda")
dim(emailDF)


### Confirming that the dataset is actually correct.  To
### do this, we want to use a different method to find one of the
### previously processed variables to see if they are identical
### For this exercise, we chose the %capital statistic

# Finding the % capital letters in a message
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)
  }
#---------
```

```
# Testing both methods for finding % capitals in Message
# to see if they are identical
pC = sapply(emailStruct, perCaps) # Original
pC2 = sapply(emailStruct, perCaps2) # New Method
identical(pC, pC2) # [1] TRUE

# Looking for unusual values in the data
# looking for NA values
emailDFna_count = colSums(is.na(emailDF))
print(emailDFna_count)
  # isSpam       isRe     numLines    bodyCharCt    underscore     subExcCt    subQuesCt
  # 0          0          0          0          0         20         20
  # numAtt     priority      numRec     perCaps   isInReplyTo    sortedRec      subPunc
  # 0          0        282          0          0          0          0
  # hour multipartText    hasImages   isPGPsigned      perHTML  subSpamWords     subBlanks
  # 0          0          0          0          0          7         20
  # noHost      numEnd    isYelling     forwards    isOrigMsg      isDear      isWrote
  # 1          0          7          0          0          0          0
  # avgWordLen   numDlr
  # 0          0

# We see the variable with subject line Exclamation points has 20 NA values
# We retrieve the index numbers of these values
indNA = which(is.na(emailDF$subExcCt))
# We check these values against the index values for the messages without a
# subject line
indNoSubject = which(is.na(emailDF$subBlanks))
    #indNoSubject = which(sapply(emailStruct,
    #                    function(msg)
    #                      !("Subject" %in% names(msg$header))))
# Determine if the NA values for subExcCt is the same as the index
# values of the messages without a subject line
all(indNA == indNoSubject) #[1] TRUE


# The number of lines in an email should not be greater than the number
# of characters in the bocy of the email. (detecting empty messages)
all(emailDF$bodyCharCt > emailDF$numLines) #[1] TRUE
```

```
#------- Figure 3.4 - Scatter Plot Number of Characters vs Lines in Body ----
#--- Looking for overall patterns
x.at = c(1,10,100,1000,10000,100000)
y.at = c(1, 5, 10, 50, 100, 500, 5000)
nL = 1 + emailDF$numLines
nC = 1 + emailDF$bodyCharCt
pdf("./Figures/Fig3.4_ScatterPlotNumLinesNumChars.pdf", width = 6, height = 4.5)
plot(nL ~ nC, log = "xy", pch=".", xlim=c(1,100000), axes = FALSE,
    xlab = "Number of Characters", ylab = "Number of Lines",main=" Number of Characters vs Lines\n
Message Body")
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.off()
#------

#--------------------------------------------------------------------------
#-----------  3.10 Exploring email Feature Set
#--------------------------------------------------------------------------


# We will examine some statistical variables relative to their classification
# as either ham or spam

#------- Figure 3.5 - Box Plot Use of Capitalization ham vs spam ----
#--- Message Body
pdf("./Figures/Fig3.5_SPAM_boxplotsPercentCaps.pdf", width = 5, height = 5)

percent = emailDF$perCaps
isSpamLabs = factor(emailDF$isSpam, labels = c("ham", "spam"))
boxplot(log(1 + percent) ~ isSpamLabs,
      ylab = "Percent Capitals (log)",cex=.4,pch=3,
      col = 'lightblue',
      main = "Box-Plot % Capitalization - Message Body")

dev.off()
#-------

#------- Figure 3.5B - Q-Q Plot Use of Capitalization ham vs spam ----
#------- Message Body
```

```
logPerCapsSpam = log(1 + emailDF$perCaps[ emailDF$isSpam ])
logPerCapsHam = log(1 + emailDF$perCaps[ !emailDF$isSpam ])
pdf("./Figures/Fig3.5B_SPAM_Q-QplotPercentCaps.pdf", width = 5, height = 5)
qqplot(logPerCapsSpam, logPerCapsHam,
      xlab = "Regular Email", ylab = "Spam Email",
      main = "Percentage of Capital Letters (log scale)",
      col = 'lightblue',
      pch = 19, cex = 0.3)
dev.off()
#-------


#------- Figure 3.6 - Scatter Plot Use of Capitalization by Classification ----
#------- Message Body
pdf("./Figures/Fig3.6_SPAM_scatterplotPercentCapsTotChars.pdf", width = 8, height = 6)

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.off()
#-----

# looking at the number of attachments for spam vs ham (not very telling)
table(emailDF$numAtt, isSpamLabs)
   # isSpamLabs
   #    ham spam
   # 0  6624 2158
   # 1   314  230
   # 2    11    6
   # 4     0    1
   # 5     1    2
   # 18    1    0

#------- Figure 3.7 - Mosaic Plot Likelyhood of containing Re: by Classification ----
#------- and likelyhood sender's email contains a number in it
pdf("./Figures/Fig3.7_SPAM_mosaicPlots.pdf", width = 8, height = 4)
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)
dev.off()
#-----


#------------------------------------------------------------------------
#-----------  3.11 Fitting the rpart() Model
#------------------------------------------------------------------------

library(rpart)
# within rpart() all variables must either be factors or numeric. We must
# convert our logical variables to factors

#---------- setupRpart () function - Rev A ----------
# Creates logical variables from factor variables
# Binds new variables to dataframe and deleted original variables
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])
}

# Preparing dataset for rpart() Converting to factor variables
emailDFrp = setupRpart(emailDF)
dim(emailDFrp)
set.seed(418910)
```

```
# Split the data set into 2/3 training and 1/3 test sets. This
# is the same split we used for the Naive Bayes dataset, since
# we are using the same seed value.
# First step is to select the Index values that will be used for
# each set of data
testSpamIdx = sample(numSpam, size = floor(numSpam/3))
testHamIdx = sample(numHam, size = floor(numHam/3))

# Creating testDF using the testIdx values. Adding observations
# that are isSpam = T first than isSpam = F after
testDF =
  rbind( emailDFrp[ emailDFrp$isSpam == "T", ][testSpamIdx, ],
       emailDFrp[emailDFrp$isSpam == "F", ][testHamIdx, ] )
dim(testDF)#[1] 3116   30

# Repeat the process for testDF but using all observations that
# were not included in the testDF
trainDF =
  rbind( emailDFrp[emailDFrp$isSpam == "T", ][-testSpamIdx, ],
       emailDFrp[emailDFrp$isSpam == "F", ][-testHamIdx, ])
dim(trainDF)#[1] 6232   30


#-------------- Fitting rpart() nmodel to training dataset --
rpartFit = rpart(isSpam ~ ., data = trainDF, method = "class")


# Library for plotting rpart object  (Trees)
library(rpart.plot)

#------- Figure 3.8 - Recursive Partioning and Classification Tree ----
#------- Training dataset
pdf("./Figures/Fig3.8_SPAM_rpartTree.pdf", width = 7, height = 7)

prp(rpartFit, extra = 1) # Function for plotting rpart tree
dev.off()
#----


# Use predict function on the testDF, using rpartFit model just created
# to determine the predicted classification of the testDF messages
predictions = predict(rpartFit,
```

```
                newdata = testDF[, names(testDF) != "isSpam"],
                type = "class")
predictions[1:5]
summary(predictions)
  # F    T
  # 2317  799

# Finding Type I error for TestDF (predicting T when it is actually F)
# subsetting using the indexes for testDF$isSpam which were classified as F
predsForHam = predictions[ testDF$isSpam == "F" ]
summary(predsForHam)
  # F    T
  # 2192  125

# The error rate for Type I error
sum(predsForHam == "T") / length(predsForHam)
  # [1] 0.05394907

# Repeat for Type II error for TestDF (predicting F when it is actually T)
# subsetting using the indexes for testDF$isSpam which were classified as T
predsForSpam = predictions[ testDF$isSpam == "T" ]
summary(predsForSpam)
  # F   T
  # 125 674

sum(predsForSpam == "F") / length(predsForSpam)
  # [1] 0.1564456

# The model, using default values for rpart() parameters, did fairly well
# on Type I errors but was not very good at Type II errors.
# We will explore some of the variables available in rpart(), to attempt
# to improve results

# Creating a list of complexity Values to be used in rpart() function,
# to see if it improves performance
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))

# Use lapply to create a list of rpart objects, using trainDF, and using
# the different complexity values
```

```
fits = lapply(complexityVals, function(x) {
  rpartObj = rpart(isSpam ~ ., data = trainDF,
                   method="class",
                   control = rpart.control(cp=x) )

# using predict() on the testDF, using the rpart objects
   predict(rpartObj,
        newdata = testDF[ , names(testDF) != "isSpam"],
        type = "class")
})

spam = testDF$isSpam == "T"
numSpam = sum(spam)
numHam = sum(!spam)

# We assign Type I and Type II error rates on the rpartObj
errs = sapply(fits, function(preds) {
  typeI = sum(preds[ !spam ] == "T") / numHam
  typeII = sum(preds[ spam ] == "F") / numSpam
  c(typeI = typeI, typeII = typeII)
})
dim(errs) #[1]  2 56
head(errs[,1:10])
   #            [,1]      [,2]      [,3]      [,4]      [,5]      [,6]      [,7]
   # typeI  0.04272767 0.04272767 0.04272767 0.04272767 0.04272767 0.04272767 0.04272767
   # typeII 0.11639549 0.11639549 0.11639549 0.11639549 0.11639549 0.11639549 0.11639549
   # [,8]      [,9]     [,10]
   # typeI  0.04272767 0.04272767 0.04272767
   # typeII 0.11639549 0.11639549 0.11639549


#------- Figure 3.9 - Type I and II Errors for Recussive Partitioning
pdf("./Figures/Fig3.9_SPAM_rpartTypeIandII.pdf", width = 8, height = 7)
library(RColorBrewer)
cols = brewer.pal(9, "Set1")[c(3, 4, 5)]
plot(errs[1,] ~ complexityVals, type="l", col=cols[2],
    lwd = 2, ylim = c(0,0.2), xlim = c(0,0.005),
    ylab="Error", xlab="complexity parameter values")
points(errs[2,] ~ complexityVals, type="l", col=cols[1], lwd = 2)

text(x =c(0.003, 0.0035), y = c(0.12, 0.05),
    labels=c("Type II Error", "Type I Error"))
```

```
minI = which(errs[1,] == min(errs[1,]))[1]
abline(v = complexityVals[minI], col ="grey", lty =3, lwd=2)

text(0.0007, errs[1, minI]+0.01,
    formatC(errs[1, minI], digits = 2))
text(0.0007, errs[2, minI]+0.01,
    formatC(errs[2, minI], digits = 3))

dev.off()
#-----

save(emailDFrp,file="./Data/data.Rda")

###-------------  Expansion of Case Study6 ----------
### Analyzing case study data using alternative algorithms
###-------------------------------------------------------


library(caret)

# Our data is in T/F 'factors'.
# We need to change it to numbers.  And as it turns out, there are quite
# a few NANs as well.  Let's set those to zero

setupRnum = function(data) {
  logicalVars = which(sapply(data, is.logical))
  facVars = lapply(data[ , logicalVars],
                   function(x) {
                     x = as.numeric(x)
                   })
  cbind(facVars, data[ , - logicalVars])
}
#----

emailDFnum = setupRnum(emailDF)

# Imputing all NA values to 0
emailDFnum[is.na(emailDFnum)]<-0


## I think i may skip this and see if I can get the folds to work
```

```
## internally to caret
#  However, one way to define your folds is to set a seed, and have
#  your folds in a list that you can pass on to others to get the same splits.
cv_folds <- createFolds(emailDFnum$isSpam, k=5, list=TRUE, returnTrain = TRUE)
lengths(cv_folds)



## Finally Metric Stuff
# Because our authors prefer Type I/II errors, but the cool kids
# know that precision/recall/F1 is where its at, while the default of caret
# is accuracy and kappa.  To get us all on the same page, I create a function
# that returns the metrics we want.  However, rather than re-invent the wheel,
# I just install a package.  I am not sure if it had Type I/II errors so those
# I made my self.  \#MLSwag

library(MLmetrics)
f1 <- function(data, lev = NULL, model = NULL) {
  f1_val <- F1_Score(y_pred = data$pred, y_true = data$obs, positive = lev[1])
  p <- Precision(y_pred = data$pred, y_true = data$obs, positive = lev[1])
  r <- Recall(y_pred = data$pred, y_true = data$obs, positive = lev[1])
  fp <-sum(data$pred==0 & data$obs==1)/length(data$pred)

  fn <-sum(data$pred==1 & data$obs==0)/length(data$pred)
  c(F1 = f1_val,
    prec = p,
    rec = r,
    Type_I_err=fp,
    Type_II_err=fn
  )
}
#----

# ok so lets get the naive bayes packages installed. (first 2 lines)
# The next line makes a dataframe of all the parameters to check.
# If you don't know what they are, look them up
# https://topepo.github.io/caret/available-models.html
#
# Then we create a trainControl object.  It tells caret how to train--using a
# cross-validation ('cv') with 3 folds in this case (number = 3).  We want the
# final predictions of the best model and our summary is the custom function from
# above.
```

```
#
# Then we create our model: "model_nb".  We user the caret::train method.  We make
# 'isSpam' a factor because R is dumb and can't figure out that 1 and 0 are classes.
# *as.factor(isSpam) ~ .*  means Y=as.factor(isSpam), X=everything else.
#
# *method* is the package we are using, and we pass our tuning grid.

library(naivebayes)
library(e1071)
nb_grid<-expand.grid(laplace=c(0,0.1,0.3,0.5,1), usekernel=c(T,F), adjust=c(T,F))
train_control<-trainControl(method="cv", number=3, savePredictions = 'final',summaryFunction = f1)
model_nb<-caret::train(as.factor(isSpam) ~ .,data=emailDFnum, trControl = train_control,
method='naive_bayes',tuneGrid = nb_grid)
model_nb

#Did the boss fool us with the folds?  Nope.
table(model_nb$pred['Resample'])
  # Fold1 Fold2 Fold3
  # 3116  3116  3116


val<-seq(from = 0, to=0.01, by=0.0005)
library(rpart)
cart_grid<-expand.grid(cp=val)
train_control<-trainControl(method="cv", number =5, savePredictions = 'final',summaryFunction = f1)
model_rpart<-caret::train(as.factor(isSpam) ~ .,data=emailDFnum, trControl = train_control,
method='rpart',tuneGrid = cart_grid)
model_rpart
  # CART
  #
  # 9348 samples
  # 29 predictor
  # 2 classes: '0', '1'
  #
  # No pre-processing
  # Resampling: Cross-Validated (5 fold)
  # Summary of sample sizes: 7478, 7478, 7478, 7479, 7479
  # Resampling results across tuning parameters:
  #
  #   cp     F1        prec      rec       Type_I_err  Type_II_err
  # 0.0000  0.9581525  0.9561686  0.9601489  0.03273425  0.02963208
  # 0.0005  0.9590785  0.9557457  0.9624508  0.03316246  0.02792033
```

```
  # 0.0010  0.9612105  0.9581377  0.9643216  0.03134382  0.02652927
  # 0.0015  0.9593084  0.9572004  0.9614442  0.03198559  0.02866888
  # 0.0020  0.9581564  0.9592122  0.9571287  0.03027430  0.03187755
  # 0.0025  0.9575135  0.9553423  0.9597183  0.03337671  0.02995230
  # 0.0030  0.9570396  0.9544202  0.9597189  0.03412566  0.02995225
  # 0.0035  0.9556337  0.9538952  0.9574167  0.03444657  0.03166411
  # 0.0040  0.9546036  0.9548451  0.9543958  0.03359096  0.03391050
  # 0.0045  0.9534228  0.9542049  0.9526695  0.03401865  0.03519398
  # 0.0050  0.9518631  0.9533941  0.9503674  0.03455284  0.03690624
  # 0.0055  0.9500561  0.9530936  0.9470580  0.03466019  0.03936641
  # 0.0060  0.9472297  0.9533036  0.9413041  0.03434008  0.04364466
  # 0.0065  0.9453650  0.9486385  0.9421674  0.03797650  0.04300295
  # 0.0070  0.9437633  0.9432875  0.9443253  0.04225572  0.04139799
  # 0.0075  0.9436108  0.9432716  0.9440377  0.04225572  0.04161195
  # 0.0080  0.9436108  0.9432716  0.9440377  0.04225572  0.04161195
  # 0.0085  0.9436108  0.9432716  0.9440377  0.04225572  0.04161195
  # 0.0090  0.9436108  0.9432716  0.9440377  0.04225572  0.04161195
  # 0.0095  0.9438597  0.9406092  0.9472031  0.04450205  0.03925878
  # 0.0100  0.9423884  0.9385597  0.9463394  0.04610684  0.03990049
  #
  # F1 was used to select the optimal model using the largest value.
  # The final value used for the model was cp = 0.001.
```

```
library(randomForest)
rf_grid<-expand.grid(mtry=seq(from =1, to = 25, by = 2))
train_control<-trainControl(method="cv", number=3, savePredictions = 'final',summaryFunction = f1)
model_rf<-caret::train(as.factor(isSpam) ~ .,data=emailDFnum, trControl = train_control,
ntree=200,method='rf',tuneGrid = rf_grid)
model_rpart
  # CART
  #
  # 9348 samples
  # 29 predictor
  # 2 classes: '0', '1'
  #
  # No pre-processing
  # Resampling: Cross-Validated (5 fold)
  # Summary of sample sizes: 7478, 7478, 7478, 7479, 7479
  # Resampling results across tuning parameters:
  #
```

```
#   cp     F1       prec      rec       Type_I_err  Type_II_err
# 0.0000  0.9581525  0.9561686  0.9601489  0.03273425  0.02963208
# 0.0005  0.9590785  0.9557457  0.9624508  0.03316246  0.02792033
# 0.0010  0.9612105  0.9581377  0.9643216  0.03134382  0.02652927
# 0.0015  0.9593084  0.9572004  0.9614442  0.03198559  0.02866888
# 0.0020  0.9581564  0.9592122  0.9571287  0.03027430  0.03187755
# 0.0025  0.9575135  0.9553423  0.9597183  0.03337671  0.02995230
# 0.0030  0.9570396  0.9544202  0.9597189  0.03412566  0.02995225
# 0.0035  0.9556337  0.9538952  0.9574167  0.03444657  0.03166411
# 0.0040  0.9546036  0.9548451  0.9543958  0.03359096  0.03391050
# 0.0045  0.9534228  0.9542049  0.9526695  0.03401865  0.03519398
# 0.0050  0.9518631  0.9533941  0.9503674  0.03455284  0.03690624
# 0.0055  0.9500561  0.9530936  0.9470580  0.03466019  0.03936641
# 0.0060  0.9472297  0.9533036  0.9413041  0.03434008  0.04364466
# 0.0065  0.9453650  0.9486385  0.9421674  0.03797650  0.04300295
# 0.0070  0.9437633  0.9432875  0.9443253  0.04225572  0.04139799
# 0.0075  0.9436108  0.9432716  0.9440377  0.04225572  0.04161195
# 0.0080  0.9436108  0.9432716  0.9440377  0.04225572  0.04161195
# 0.0085  0.9436108  0.9432716  0.9440377  0.04225572  0.04161195
# 0.0090  0.9436108  0.9432716  0.9440377  0.04225572  0.04161195
# 0.0095  0.9438597  0.9406092  0.9472031  0.04450205  0.03925878
# 0.0100  0.9423884  0.9385597  0.9463394  0.04610684  0.03990049
#
# F1 was used to select the optimal model using the largest value.
# The final value used for the model was cp = 0.001.
```

```
#------- Figure 3.Special - Variable Importance Factor for Random Forest
pdf("./Figures/Fig3.Special_VariableImportance_RandomForest.pdf", width = 8, height = 7)
rf1 <- randomForest(as.factor(isSpam) ~ .,cp=0.001,data=emailDFnum)
rf1$importance
varImpPlot(rf1,main="Important variables for Spam Classification")
dev.off()
#-----
```

```
library(xgboost)
xgb_grid<-expand.grid(nrounds = 100, max_depth = c(3,5,7,9,11), eta = c(0.01,0.03,0.1),
gamma=c(1,3,5,10), colsample_bytree=1, min_child_weight=1, subsample=1)
train_control<-trainControl(method="cv", number=3, savePredictions = 'final',summaryFunction = f1)
model_xgb<-caret::train(as.factor(isSpam) ~ .,data=emailDFnum, trControl =
train_control,method='xgbTree',tuneGrid = xgb_grid)
```

```
model_xgb
  # eXtreme Gradient Boosting
  #
  # 9348 samples
  # 29 predictor
  # 2 classes: '0', '1'
  #
  # No pre-processing
  # Resampling: Cross-Validated (3 fold)
  # Summary of sample sizes: 6232, 6232, 6232
  # Resampling results across tuning parameters:
  #
  #  eta  max_depth  gamma  F1        prec      rec       Type_I_err  Type_II_err
  # 0.01  3          1     0.9325138  0.8910818 0.9779888 0.08889602  0.016367137
  # 0.01  3          3     0.9320846  0.8908973 0.9772695 0.08900300  0.016902011
  # 0.01  3          5     0.9321577  0.8909110 0.9774133 0.08900300  0.016795036
  # 0.01  3         10     0.9319668  0.8905632 0.9774133 0.08932392  0.016795036
  # 0.01  5          1     0.9507558  0.9253922 0.9775572 0.05862217  0.016688062
  # 0.01  5          3     0.9509613  0.9256514 0.9777011 0.05840822  0.016581087
  # 0.01  5          5     0.9512952  0.9262849 0.9777011 0.05787334  0.016581087
  # 0.01  5         10     0.9511802  0.9269723 0.9766940 0.05723149  0.017329910
  # 0.01  7          1     0.9591569  0.9425066 0.9764063 0.04428755  0.017543860
  # 0.01  7          3     0.9592800  0.9430139 0.9761185 0.04385965  0.017757809
  # 0.01  7          5     0.9592591  0.9419007 0.9772695 0.04482242  0.016902011
  # 0.01  7         10     0.9586442  0.9424558 0.9753992 0.04428755  0.018292683
  # 0.01  9          1     0.9646711  0.9534898 0.9761185 0.03540864  0.017757809
  # 0.01  9          3     0.9638535  0.9535405 0.9743922 0.03530167  0.019041506
  # 0.01  9          5     0.9634780  0.9536335 0.9735290 0.03519469  0.019683355
  # 0.01  9         10     0.9606418  0.9522226 0.9692131 0.03615747  0.022892597
  # 0.01  11         1     0.9699404  0.9606401 0.9794274 0.02984596  0.015297390
  # 0.01  11         3     0.9692472  0.9614971 0.9771256 0.02909713  0.017008986
  # 0.01  11         5     0.9677136  0.9609908 0.9745360 0.02941806  0.018934531
  # 0.01  11        10     0.9623347  0.9597878 0.9648971 0.03005991  0.026101840
  # 0.03  3          1     0.9425920  0.9119339 0.9753992 0.07006846  0.018292683
  # 0.03  3          3     0.9427004  0.9125171 0.9749676 0.06953359  0.018613607
  # 0.03  3          5     0.9428897  0.9129983 0.9748238 0.06910569  0.018720582
  # 0.03  3         10     0.9422342  0.9117723 0.9748238 0.07017544  0.018720582
  # 0.03  5          1     0.9610995  0.9437162 0.9791397 0.04343175  0.015511339
  # 0.03  5          3     0.9606605  0.9436751 0.9782765 0.04343175  0.016153188
  # 0.03  5          5     0.9605208  0.9435403 0.9781326 0.04353872  0.016260163
  # 0.03  5         10     0.9601336  0.9438599 0.9769817 0.04321780  0.017115961
  # 0.03  7          1     0.9666731  0.9571373 0.9764063 0.03252033  0.017543860
```

```
# 0.03   7       3    0.9661821  0.9564483  0.9761185  0.03305520  0.017757809
# 0.03   7       5    0.9656163  0.9557546  0.9756870  0.03359007  0.018078733
# 0.03   7      10    0.9634656  0.9522349  0.9749676  0.03637142  0.018613607
# 0.03   9       1    0.9712868  0.9621704  0.9805783  0.02866923  0.014441592
# 0.03   9       3    0.9706047  0.9627737  0.9785642  0.02813436  0.015939238
# 0.03   9       5    0.9697869  0.9607502  0.9789958  0.02973898  0.015618314
# 0.03   9      10    0.9663878  0.9587931  0.9741044  0.03112965  0.019255456
# 0.03  11       1    0.9745092  0.9673970  0.9817292  0.02460419  0.013585794
# 0.03  11       3    0.9728531  0.9662287  0.9795713  0.02545999  0.015190415
# 0.03  11       5    0.9720513  0.9659118  0.9782765  0.02567394  0.016153188
# 0.03  11      10    0.9681005  0.9625965  0.9736729  0.02813436  0.019576380
# 0.10   3       1    0.9654720  0.9481370  0.9834556  0.04000856  0.012302097
# 0.10   3       3    0.9645000  0.9468022  0.9828802  0.04107831  0.012729996
# 0.10   3       5    0.9630905  0.9452830  0.9815854  0.04225503  0.013692769
# 0.10   3      10    0.9622365  0.9429776  0.9823047  0.04418057  0.013157895
# 0.10   5       1    0.9741426  0.9648704  0.9835995  0.02663671  0.012195122
# 0.10   5       3    0.9739233  0.9648531  0.9831679  0.02663671  0.012516046
# 0.10   5       5    0.9711921  0.9606069  0.9820170  0.02995293  0.013371844
# 0.10   5      10    0.9651984  0.9513809  0.9794274  0.03722721  0.015297390
# 0.10   7       1    0.9774265  0.9707711  0.9841749  0.02203680  0.011767223
# 0.10   7       3    0.9750744  0.9682310  0.9820170  0.02396234  0.013371844
# 0.10   7       5    0.9721643  0.9647299  0.9797151  0.02663671  0.015083440
# 0.10   7      10    0.9683722  0.9590815  0.9778449  0.03102268  0.016474112
# 0.10   9       1    0.9796257  0.9737140  0.9856136  0.01979033  0.010697475
# 0.10   9       3    0.9753317  0.9695802  0.9811538  0.02289260  0.014013693
# 0.10   9       5    0.9736348  0.9672104  0.9801467  0.02471117  0.014762516
# 0.10   9      10    0.9694661  0.9616518  0.9774133  0.02899016  0.016795036
# 0.10  11       1    0.9810470  0.9755380  0.9866206  0.01839966  0.009948652
# 0.10  11       3    0.9768072  0.9720766  0.9815854  0.02096705  0.013692769
# 0.10  11       5    0.9746716  0.9695395  0.9798590  0.02289260  0.014976466
# 0.10  11      10    0.9712818  0.9646684  0.9779888  0.02663671  0.016367137
#
# Tuning parameter 'nrounds' was held constant at a value of 100
# Tuning
# parameter 'min_child_weight' was held constant at a value of 1
# Tuning
# parameter 'subsample' was held constant at a value of 1
# F1 was used to select the optimal model using the largest value.
# The final values used for the model were nrounds = 100, max_depth = 11, eta =
#   0.1, gamma = 1, colsample_bytree = 1, min_child_weight = 1 and subsample = 1.
```

```
#------- Figure 3.Special2 - Variable Importance Factor for Random Forest
pdf("./Figures/Fig3.Special2_VariableImportance_XGBoost.pdf", width = 7, height = 7)
# Used optimal results from XGBoost matrix for analysis model
bst <- xgboost(data = as.matrix(emailDFnum[ , names(emailDFnum) != "isSpam"]), label =
emailDFnum$isSpam,
        nrounds = 100, max_depth = 11, eta =0.1, gamma = 1, colsample_bytree = 1,
        min_child_weight = 1, subsample = 1, objective = "binary:logistic")
#bstimportance = xgb.importance(model = bst)

importance_matrix <- xgb.importance(colnames(emailDFnum[ , names(emailDFnum) != "isSpam"]),
model = bst)

xgb.plot.importance(importance_matrix, rel_to_first = TRUE,
            xlab = "Relative importance",
            main = 'Important variables for Spam Classification\nXGBoost',
            col = 'blue')
dev.off()
#-----

gg <- xgb.ggplot.importance(importance_matrix, measure = "Frequency", rel_to_first = TRUE)
gg + ggplot2::ylab("Frequency")

#----- End of code ----
```