Spam Message Detection

Stuart Miller, Justin Howard, and Paul Adams

October 6, 2020

1 Introduction

Email usage has become so ubiquitous that having an email address is just as common as having a phone number. A brief examination of the benefits of using email reveals why. Email is a cost effective means of mass communication. Free services, such as gmail, make having an email address possible for anyone with an internet connection and a device that can send the message.

Advertisers gravitate to the use of all accessible and efficient communication methods, and email fits this description well. Some advertisers send messages so frequently, that they irritate address owners, who then seek ways to end the stream of advertisements coming to their email address. Automatic filters that detect unwanted email advertisements, called spam, using machine learning and natural language processing.

This case study will demonstrate how to build a spam filter using an email dataset.

2 Methods

2.1 Data

The data used in this case study consists of 9000 emails classified into spam and non-spam by Apache SpamAssassin (an opensource platform for spam detection)[1]. This dataset is provided as a public corpus by the Apache Foundation for spam classification model development[2]. SpamAssassin classified the emails into three categories spam, ham, and hard ham (non-spam emails are also colloquially referred to as ham). The emails listed in the hard ham category are legitimate emails, but have characteristics more similar to the spam emails. It should be noted that this corpus contains emails from 2002-2006[3]. While the dataset may be a useful source for experimenting with spam detection, this corpus may no longer be representative of spam and non-spam emails found in the wild due to age and spam evolution.

2.1.1 Feature Engineering

While it might be natural to consider the words frequencies of an email to detect spam, there are often a large number of words in a typical corpus vocabulary (10s of thousands), leading to high-dimensional,

[3]https://spamassassin.apache.org/
[3]https://spamassassin.apache.org/old/publiccorpus/
[3]https://spamassassin.apache.org/old/publiccorpus/readme.html

sparse feature matricies. High-dimensional, sparse matricies often do not lead to good modeling results. We engineered 29 features from the emails rather than use word frequencies. Five example engineered features are given below (the full list is provided in appendix A).

- The number of lines in the email
- Whether the email is a reply (RE) or not
- The number of unique characters used in the body
- Whether the sender name contains an underscore ('_') or not
- The number of exclaimation points in the email subject line

2.1.2 An Analysis of Odds

We used log-odds to assess the fit of a probability-based model, where the probability of a message being spam is equal to the count of spam words divided by the count of spam messages plus 0.5 and the probability of a message being not spam is equal to the count of non-spam words divided by the count of non-spam emails plus 0.5. This enabled us to gauge how apt the assumptions of the model are, with respect to spam and not-spam.

Using all training data for each class across all email data types, we gathered raw probability estimations and their mean log-odds per data type.

Class	doctype	html	public	dtd	transitional
Spam	0.057269947	0.5148735	0.09229914	0.056713928	0.050041701
Non-Spam	0.004027232	0.2025122	0.05043628	0.005753188	0.004219005
Spam Log Odds	2.654696773	0.9331210	0.60432398	2.288265654	2.473257439
Non-Spam Log Odds	-0.054939939	-0.4970568	-0.04508776	-0.052615877	-0.047109262

With respect to the following analysis, the null hypothesis states that a given message is not spam:

Based on the distribution of Type I and Type II errors of log likelihood values in **Fig. 1** - where Type I error is rejecting the null hypothesis when the null hypothesis should not be rejected and Type II error is failing to reject the null when there is statistically significant evidence the null should be rejected - the threshold cutoff for determining spam is a log-likelihood value of -48. Translating this to a threshold value meaning, a message with a log-likelihood value of greater than -48 is spam and a message with a log-likelihood value of less than -48 is not spam.

Below, the Type I Error (rejecting the null hypothesis when the null hypothesis should not be rejected) level of significance is 0.01 and the Type II Error (failing to reject the null hypothesis where the null hypothesis is true) is 0.05.

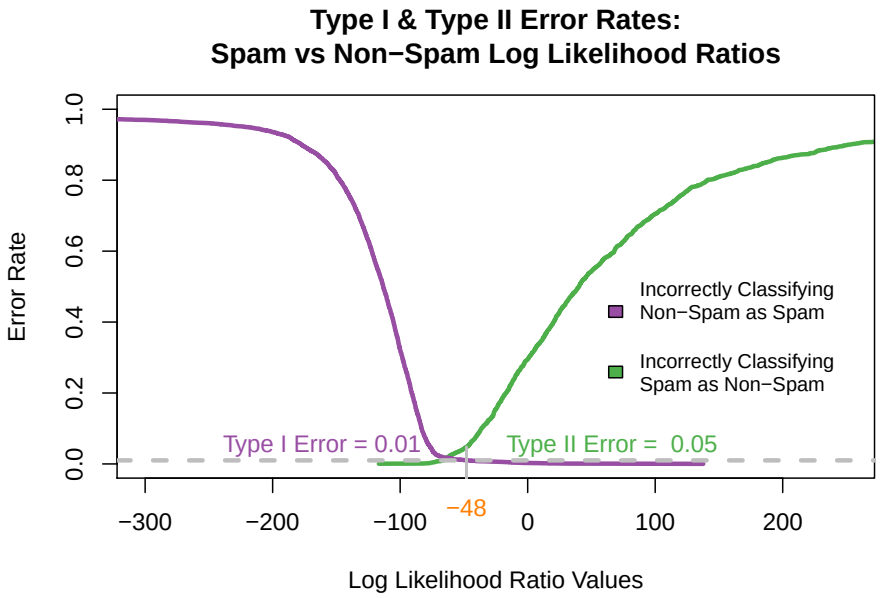

Figure 1: NB Type I and Type II Error

The distributions of log-likelihoods for each class can be inspected in **Fig. 2**. Based on distribution spread of log likelihood ratios for each class, the probability approach appears better at identifying spam than messages that are not spam. This is because the spam class has an overall distribution - including the mean - closer to 0 than the non-spam class, which is indicative of a better fit. However, there is more repeatability of results within the non-spam class words as the interquartile range is much narrower than that of the spam class. This ultimately means words used in non-spam messages are not as unique to non-spam messages as those in spam messages. Intuitively, this makes sense because many of the words used in spam messages are often intentional misspellings - such as words that substitute "3" in place of "e" so spam filters cannot catch them as easily - that do not commonly appear in non-spam messages. Conversely, most words used in non-spam messages also appear in spam messages.

Because of this, in addition to the Naive Bayes approach, we constructed additional models to classify spam vs. not spam, including logistic regression and a decision tree classifier.

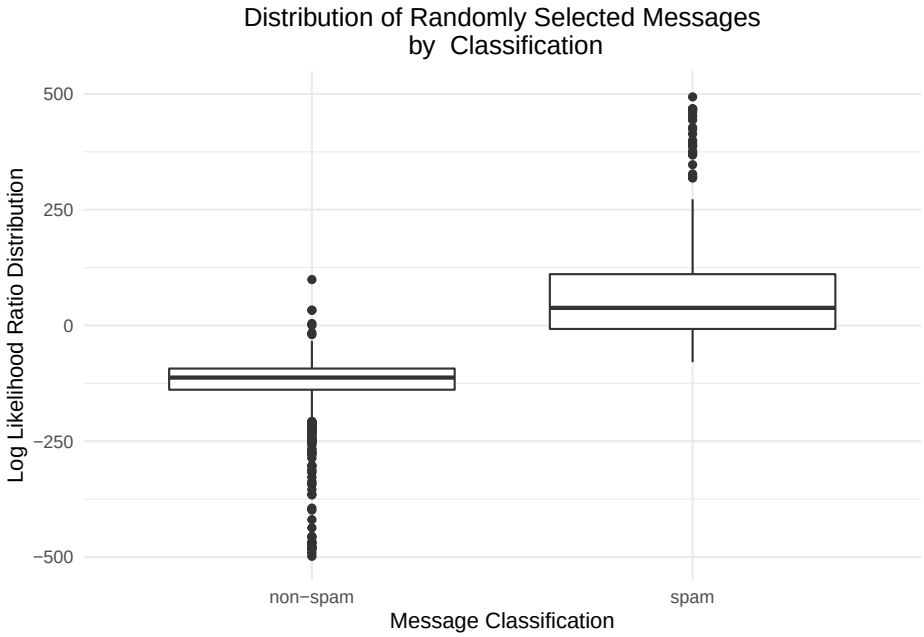

Figure 2: Naive Bayes Log Likelihood Distributions

2.2 Modeling

We experimented with three modeling techniques on this dataset: logistic regression, naive bayes, and parition trees. Logistic regression was primary employed as a baseline model. Since naive bayes and parition trees can describe non-linear decision boundries, these models could provide better performance than logistic regression.

Since hyperparameters must be selected for naive bayes and parition trees, we the data into two sets by random sampling: 70% for training and 30% for test. Each model was fit with 3-fold cross-validation on the training data to select the hyperparameters where F1-score was used as the selection criterion. The hyperparameters selected for each model are shown in table 1 and the F1-scores from cross-validation are shown in table 2. After selecting hyperparameters, we compared the performance of each model on the test set (shown in table 3).

2.2.1 Hyperparameters

We used normal logistic regression, thus there were no hyperparameters to tune.

For naive bayes, we tuned the following parameters:

- `laplace` - laplace smoothing
- `usekernel` - if `TRUE` use a kernel density estimate, else use gaussian density estimate
- `adjust` - if `TRUE` allow the bandwidth of kernel density estimate to be adjusted (more flexiable), else do not allow adjustments

For partition trees, we tuned the following parameters:

- `cp` - minimum decrease in lack of fit for a new split

2.3 Results

2.3.1 Model Results

Using F1 as the baseline metric for error assessment, the Partition Tree outperformed Naive Bayes and Logistic Regression during both internal and external cross-validation. On internal 3-fold cross-validation, the Partition Tree scored an F1 of 0.957, while the Naive Bayes - outperforming the other probability-based model (Logistic Regression) by 0.042 - scored 0.945.

After applying the models to the external test set, error further diverged between the tree-based and probability-based models. Contrary to the internal cross-validation results, Logistic Regression also appears to have generalized better than Naive Bayes, marginally outperforming the Bayesian model by 0.012. However, the tree-based Partition Tree model maintained superiority over the probability-based models; Logistic Regression scored and F1 of 0.911 and the Partition Tree scored an F1 of 0.957.

Multiple hyperparamter configurations were applied during modeling. The Naive Bayes model used a Laplace Correction grid search for values of 0, 0.1, 0.3, 0.5, and 1 to shift probability from zero when there are words that have not been encountered during training. This was applied in combination with adjusted an non-adjusted kernel bandwidth denstiy values. The values we used were 0 for Laplace Smoothing - potentially because the words not trained on were either insignificant or they did not exist - using a bandwidth-adjusted kernel density estimation. Internal cross-validation was performed on three folds, with tuning towards the balanced F1 score.

The partition tree was tuned by a using a complexity parameter range from 0 to 0.01 at intervals of 0.0005. The selected complexity paramter was 0.0015. The precision/recall tuning was balanced with focus on the F1 score.

Table 1. Hyperparameter Tuning Results

Model	Parameter	Value
Naive Bayes	laplace	0
Naive Bayes	usekernal	`True`
Naive Bayes	adjust	`True`
Partition Tree	cp	0.0015

During k-fold cross-validation, Partition Trees were are better for this data set and task than Naive Bayes and Logistic Regression, by F1 score.

Table 2. Cross Validation Results

Model	F1 Score
Naive Bayes	0.903
Partition Tree	0.957
Logistic Regression	0.945

When applied to our test set, Partition Trees continued to perform better than both Naive Bayes and Logistic Regression.

Table 3. Test Results

Model	F1 Score
Naive Bayes	0.899
Partition Tree	0.953
Logistic Regression	0.911

The following results were the comprehensive, summary-level results of our three models:

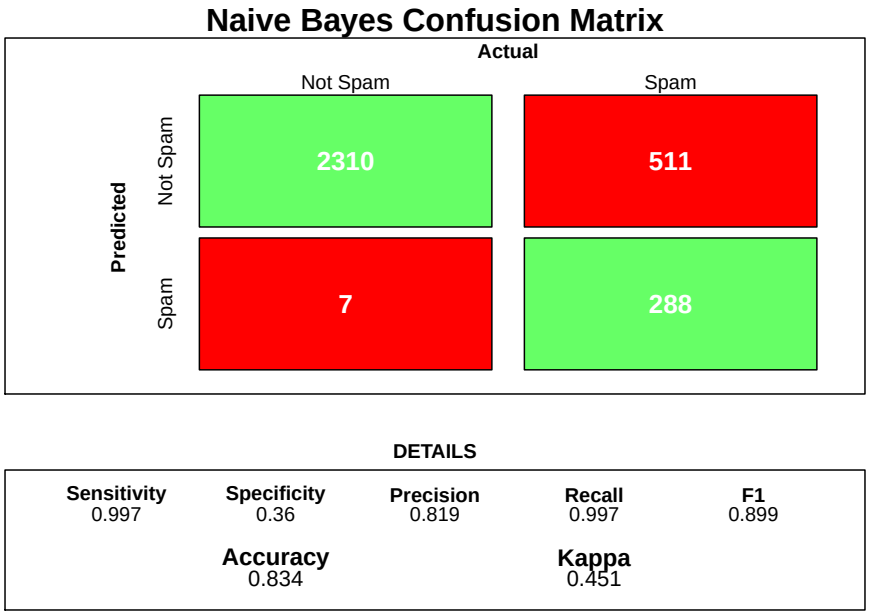

Figure 3: Naive Bayes Confusion Matrix

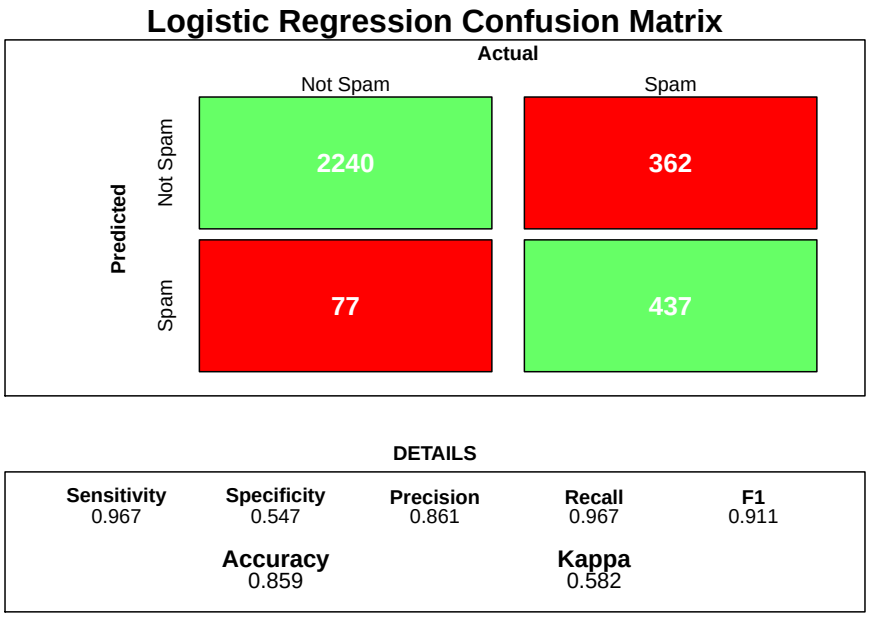

Figure 4: Logistic Regression Confusion Matrix

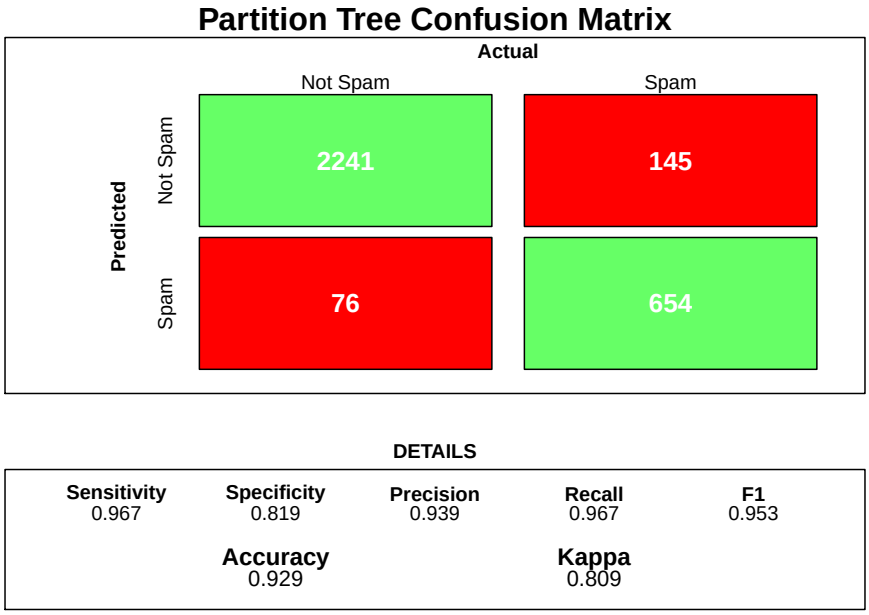

Figure 5: Partition Tree Confusion Matrix

3 Conclusion

Da-dum

A Feature Engineering

A.1 Derived Features

The following boolean values (logical) and numeric features were derived from the emails.

Number	Feature Description	Type
1	The number of lines in the email	Logical
2	Whether the email is a reply (RE) or not	Numeric
3	The number of unique characters used in the body	Numeric
4	Whether the sender name contains an underscore ('_') or not	Logical
5	The number of exclaimation points in the email subject line	Numeric
6	The number of question marks in the email subject line	Numeric
7	The number of attachements in the email	Numeric
8	Whether the email is marked as 'priority' or not	Logical
9	The number of recipients	Numeric
10	The percentage of the email in capital letters	Numeric
11	Whether "In-Reply-To" is used in the email header or not	Numeric
12	Whether the recipients are listed in sorted order or not	Logical
13	Whether punctuation is used in the email subject line or not	Logical
14	The hour the email was sent	Numeric
15	Whether the email contains multipart text	Logical
16	Whether the email contains images or not or not	Logical
17	Whether the email is PGP signed or not	Logical
18	The percentage of the message that is HTML	Numeric
19	Whether the email contains a spam word (see list below)	Logical
20	The number of white space characters in the email subject line	Numeric
21	Whether the email host name is present	Logical
22	Whether there are numbers at the end of an email address	Logical
23	Whether there are letters in the email subject line are all capitalized	Logical
24	The number of foreward symbols in the email	Numeric
25	Whether the email is the original message	Logical
26	Whether the salutation "Dear" is used	Logical
27	Whether "wrote:" appears in the email body	Logical
28	The average length of words in the email body	Numeric
29	The number of dollar signs in the email body	Numeric

A.2 Spam Words

The following words (commonly found in spam emails) were considered spam words for feature 19.

- "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”

B Code

```
library(pacman)
p_load(tidyverse, rpart, rpart.plot, tm, naivebayes, e1071, randomForest, caret, MLmetrics
, RColorBrewer)

# video 1
# list the data diectories
list.dirs('./data/', full.names = F)

# get the number of data directories
length(list.files(paste('./data/', sep=.Platform$file.sep)))

# get all the data directory names
dir.names <- list.files(paste('./data', sep=.Platform$file.sep))

# list the number of files in each data directory
sapply(paste('./data', dir.names, sep = .Platform$file.sep),
       function(dir) length(list.files(dir)))

# get the full data dir names
filldirNames <- paste('./data', dir.names, sep = .Platform$file.sep)

# list full dir names of data files; print sample
fileNames <- list.files(filldirNames[1], full.names = T)
fileNames[1]

# get some example enail files from the first dir
messageIdx <- c(1:5, 15, 27, 68, 69, 329, 404, 427, 516, 852, 971)
fn = list.files(filldirNames[1], full.names = T)[messageIdx]
sampleEmail <- sapply(fn, readLines)
sampleEmail

# video 2
# get a sample email to work with
msg <- sampleEmail[[1]]

# get the blank line as a split point
which(msg == '')[1]
match('', msg)

# save the split point for example message
splitPoint <- match('', msg)
```

```
# view area around split point
msg[ (splitPoint - 2) : (splitPoint + 6) ]

# get the header and body of teh example message
header <- msg[1 : (splitPoint - 1)]
body <- msg[ -( 1:(splitPoint - 1) ) ]

# create a function to split messages on the first empty line
# header is the first element and body is the second element
# of the split
splitMessage <- function(msg){
  splitPoint <- match('', msg)
  header <- msg[1 : (splitPoint - 1)]
  body <- msg[ -( 1:(splitPoint - 1) ) ]
  return(list(header = header, body = body))
}

# apply function to the sample messages
sampleSplit <- lapply(sampleEmail, splitMessage)
length(sampleSplit)

# pring the sample header
sampleSplit[[6]]$header

# video 3
# get some files and split them as before
messageIdx <- c(1:5, 15, 27, 68, 69, 329, 404, 427, 516, 852, 971,
                1070, 1267, 1450, 1585, 1709, 1234, 1046, 1149,
                1641, 1406)
fn = list.files(filldirNames[1], full.names = T)[messageIdx]
sampleEmail <- sapply(fn, readLines)
sampleSplit <- lapply(sampleEmail, splitMessage)
header <- sampleSplit[[1]]$header
grep("Content-Type", header)

# find locations of "Contenct-Type" (where the attachment is)
headerList <- lapply(sampleSplit, function(msg) msg$header)
CTloc <- sapply(headerList, grep, pattern = "Content-Type")
CTloc

# improvement where NA is returned for 0 (no occurance)
sapply(headerList, function(header){
```

```
  CTloc <- grep("Content-Type", header)
  if (length(CTloc) == 0) return (NA)
  CTloc
})

# find locations of attachements
hasAttach <- sapply(headerList, function(header){
  CTloc <- grep("Content-Type", header)
  if (length(CTloc) == 0) return (FALSE)
  grepl('multi', tolower(header[CTloc]))
})
header <- sampleSplit[[12]]$header
boundaryIdx <- grep('boundary=', header)
header[boundaryIdx]
sub(".*boundary=\"? *(.*)\"?;?.*", "\\1", header[boundaryIdx])

# a function to find boundaries
getBoundary <- function(header){
  boundaryIdx <- grep('boundary=', header)
  boundary = gsub('"', "", header[boundaryIdx])
  gsub(".*boundary= *([^;]*);?.*", "\\1", boundary)
}

# sample message with a PGP signature
sampleSplit[[12]]$body

# search for boundary of msg 15
boundary = getBoundary(headerList[[15]])
body = sampleSplit[[15]]$body

# locating beginning string
bString = paste("--", boundary, sep = "")
bStringLocs = which(bString == body)
bStringLocs

# locating ending string
eString = paste("--", boundary, "--", sep = "")
eStringLoc = which(eString == body)
eStringLoc

# separating the body of the message
msg = body[ (bStringLocs[1] + 1) : (bStringLocs[2] - 1)]
tail(msg)
```

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

# function that drops attachments
dropAttach = function(body, boundary){

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

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

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

  if (length(eStringLoc) == 0)
    return(body[ (bStringLocs[1] + 1) : (bStringLocs[2] - 1)])

  n = length(body)

  if (eStringLoc < n)
     return( body[ c( (bStringLocs[1] + 1) : (bStringLocs[2] - 1),
                    ( (eStringLoc + 1) : n )) ] )

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

head(sampleSplit[[1]]$body)

# load stopwords function
stopWords = stopwords()
# applying same text pre-processing to stopwords as emails
cleanSW = tolower(gsub("[[:punct:]0-9[:blank:]]+", " ", stopWords))
SWords = unlist(strsplit(cleanSW, "[[:blank:]]+"))
SWords = SWords[ nchar(SWords) > 1 ]
stopWords = unique(SWords)

words = SWords[ !( SWords %in% stopWords) ]
words[0:15]

# Function to remove numbers, punctuation, and other undesirables
cleanText = function(msg){
  tolower(gsub("[[:punct:]0-9[:space:][:blank:]]+", " ", msg))
}
```

```
# function that extracts words from message body
findMsgWords =
  function(msg, stopWords){
 if(is.null(msg))
  return(character())

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

 # drop stopwords
 words = words[ nchar(words) > 1]
 words = words[ !( words %in% stopWords) ]
 invisible(words)
  }

# Given directory and stop words, this extracts all email verbiage
processAllWords = function(dirName, stopWords){
  # List all files in the directory
  fileNames = list.files(dirName, full.names = TRUE)

  # Remove all data except the emails
  notEmail = grep("cmds$", fileNames)
  if ( length(notEmail) > 0) fileNames = fileNames[ - notEmail ]
  messages = lapply(fileNames, readLines, encoding = "latin1")

  # break out the email message
  emailSplit = lapply(messages, splitMessage)
  bodyList = lapply(emailSplit, function(msg) msg$body)
  headerList = lapply(emailSplit, function(msg) msg$header)
  rm(emailSplit)

  # Email including attachment
  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)

  # Get boundarys string in the emails
```

```
  boundaries = sapply(headerList[hasAttach], getBoundary)

  # Drop attachments from the email bodies
  bodyList[hasAttach] = mapply(dropAttach, bodyList[hasAttach],
                               boundaries, SIMPLIFY = FALSE)

  # Get the message verbiage
  messageWordsList = lapply(bodyList, findMsgWords, stopWords)

  invisible(messageWordsList)
}
# All valid words from all the emails
messageWordsList = lapply(filldirNames, processAllWords, stopWords = stopWords)
numbaMessages = sapply(messageWordsList, length)
# First three directories are emails and last two are spam. The five lists are unpacked into one.
isSpam = rep(c(FALSE, FALSE, FALSE, TRUE, TRUE), numbaMessages) # Identify the false messages
# All valid words from all the emails
messageWordsList = unlist(messageWordsList, recursive = FALSE)

emailCnt = length(isSpam)
spamCnt = sum(isSpam)
legitCnt = emailCnt - spamCnt
set.seed(43)
### Test and train split
# Random-downsample to 25% without replacement
testSpamIdx = sample(spamCnt, size = floor(spamCnt*0.25))
testLegitIdx = sample(legitCnt, size = floor(legitCnt*0.25))
# split out into test and train data
testMessageWords = c((messageWordsList[isSpam])[testSpamIdx]
,(messageWordsList[!isSpam])[testLegitIdx] )
trainMessageWords = c((messageWordsList[isSpam])[ - testSpamIdx]
,(messageWordsList[!isSpam])[ - testLegitIdx])
testSpam = rep(c(TRUE, FALSE), c(length(testSpamIdx), length(testLegitIdx)))
unique(testSpam)
trainSpam = rep(c(TRUE, FALSE), c(spamCnt - length(testSpamIdx), legitCnt - length(testLegitIdx)))
unique(trainSpam)

# number of unique words in training data
wordsUnique = unique(unlist(trainMessageWords))
length(wordsUnique)
# number of occurrences of each unique word in the spam message
spamWordCounts = rep(0, length(wordsUnique))
names(spamWordCounts) = wordsUnique
```

```
trainWords = lapply(trainMessageWords[trainSpam], unique)
trainTable = table(unlist(trainWords))
spamWordCounts[names(trainTable)] = trainTable
# number of occurrences of spam and non-spam for each word in the list.
spamVnotSpam = function(wordsList, spam, wordsUnique = unique(unlist(wordsList))){
  # table for spam, non-spam, and log-odds
  wordTable = matrix(0.5, nrow = 4, ncol = length(wordsUnique), dimnames = list(c("spam"
  , "non-spam", "presentLogOdds", "absentLogOdds"),  wordsUnique))
  # word count increases by one for each spam message there is
  spamWordCnt = table(unlist(lapply(wordsList[spam], unique)))
  wordTable["spam", names(spamWordCnt)] = spamWordCnt + 1
  # word count increases by one for each non-spam message there is
  legitWordCnt = table(unlist(lapply(wordsList[!spam], unique)))
  wordTable["non-spam", names(legitWordCnt)] = legitWordCnt + 1
  # number of spam and non-spam
  spamCnt = sum(spam)
  hamCnt = length(spam) - spamCnt
  wordTable["spam",] = wordTable["spam",]/(spamCnt + .5) # probability of non-spam given spam
  wordTable["non-spam",] = wordTable["non-spam",]/(hamCnt + .5) # probability of spam given non-spam

  # log-odds
  wordTable["presentLogOdds",] = log(wordTable["spam",]) - log(wordTable["non-spam", ])
  wordTable["absentLogOdds",] = log((1 - wordTable["spam",])) - log((1 -wordTable["non-spam",]))
  invisible(wordTable)
}
### Apply spamVnotSpam function to training data
trainTable = spamVnotSpam(trainMessageWords, trainSpam)
trainTable[,1:5]

# There may be words not in training data. This calculates log odds without new words.
# The first message is spam, and the log likelihood is positive.
newMessage = testMessageWords[[1]]
newMessage = newMessage[!is.na(match(newMessage, colnames(trainTable)))]
present = colnames(trainTable) %in% newMessage
sum(trainTable["presentLogOdds", present]) + sum(trainTable["absentLogOdds", !present])
# A negative value with a large log likelihood ratio is in this non-spam message
newMessage = testMessageWords[[which(!testSpam)[1]]]
newMessage = newMessage[!is.na(match(newMessage, colnames(trainTable)))]
present = (colnames(trainTable) %in% newMessage)
sum(trainTable["presentLogOdds", present]) + sum(trainTable["absentLogOdds", !present])
# function to find log likelihood:
getLogLikelihood = function(words, freqTable){
  # remove words not in the training data
```

```
  words = words[!is.na(match(words, colnames(freqTable)))]
  # get words that DO appear in training data
  present = colnames(freqTable) %in% words
  sum(freqTable["presentLogOdds", present]) + sum(freqTable["absentLogOdds", !present])
}
# box plots of log likelihoods
testLogLikelihood = sapply(testMessageWords, getLogLikelihood, trainTable)
tapply(testLogLikelihood, testSpam, summary)

# misclassification rate and using reference value tau
type_1_Error = function(tau, logLikelihoodVals, spam){
  classify = logLikelihoodVals > tau
  sum(classify & !spam)/sum(!spam)
}
# tau = 0
type_1_Error(0, testLogLikelihood,testSpam)
# tau = -20
type_1_Error(-20, testLogLikelihood,testSpam)
type_1_Error = function(logLikelihoodVals, isSpam){
  orderVals = order(logLikelihoodVals)
  logLikelihoodVals = logLikelihoodVals[orderVals]
  isSpam = isSpam[orderVals]
  idx = which(!isSpam)
  N = length(idx)
  list(error = (N:1)/N, values = logLikelihoodVals[idx])
}
type_2_Error = function(logLikelihoodVals, isSpam){

  orderVals = order(logLikelihoodVals)
  logLikelihoodVals =  logLikelihoodVals[orderVals]
  isSpam = isSpam[orderVals]
  idx = which(isSpam)
  N = length(idx)
  list(error = (1:(N))/N, values = logLikelihoodVals[idx])
}
# Get tau so one type of error is within 1% using 5-fold cross-validation.
k = 5
trainCnt = length(trainMessageWords)
partK = sample(trainCnt)
total = k * floor(trainCnt/k)
partK = matrix(partK[1:total], ncol = k)
testFoldOdds = NULL
for (i in 1:k){
```

```
  foldIdx = partK[,i]
  trainTabFold = spamVnotSpam(trainMessageWords[-foldIdx], trainSpam[-foldIdx])
  testFoldOdds = c(testFoldOdds, sapply(trainMessageWords[foldIdx], getLogLikelihood
  , trainTabFold))
}
testFoldSpam = NULL
for (i in 1:k){
  foldIdx = partK[,i]
  testFoldSpam = c(testFoldSpam, trainSpam[foldIdx])
}
xFoldI = type_1_Error(testFoldOdds, testFoldSpam)
xFoldII = type_2_Error(testFoldOdds, testFoldSpam)
tauFoldI = round(min(xFoldI$values[xFoldI$error <= 0.01]))
t2 = max(xFoldII$error[ xFoldII$values < tauFoldI ])
#tFold2 = xFoldII$error[ xFoldII$values < tauFoldI ]
#max(tFold2)

cols = brewer.pal(9, "Set1")[c(3, 4, 5)]
plot(xFoldII$error ~ xFoldII$values,  type = "l", col = cols[1], lwd = 3,
     xlim = c(-300, 250), ylim = c(0, 1),
     xlab = "Log Likelihood Ratio Values", ylab="Error Rate", main="Type I & Type II
     Error Rates:\nSpam vs Non-Spam Log Likelihood Ratios")
points(xFoldI$error ~ xFoldI$values, type = "l", col = cols[2], lwd = 3)
legend(x = 50, y = 0.6, fill = c(cols[2], cols[1]),
       legend = c("Incorrectly Classifying\nNon-Spam as Spam\n",
                  "Incorrectly Classifying\nSpam as Non-Spam"), cex = 0.8,
       bty = "n")
abline(h=0.01, col ="grey", lwd = 3, lty = 2)
text(-250, 0.05, pos = 4, "Type I Error = 0.01", col = cols[2])
mtext(tauFoldI, side = 1, line = 0.5, at = tauFoldI, col = cols[3])
segments(x0 = tauFoldI, y0 = -.50, x1 = tauFoldI, y1 = t2,
         lwd = 2, col = "grey")
text(tauFoldI + 20, 0.05, pos = 4,
     paste("Type II Error = ", round(t2, digits = 2)),
     col = cols[1])

# generate the box plot
SPAM_CLASS = c("non-spam", "spam")[1 + testSpam]
#boxplot(testLogLikelihood ~ SPAM_CLASS, ylab = "Log Likelihood Ratio Distribution", main =
"Distribution of Randomly #Selected Messages\n by  Classification", ylim=c(-500, 500))
data.frame('testLogLikelihood' = testLogLikelihood,
           'SPAM_CLASS' = SPAM_CLASS) %>%
  ggplot(aes(x = SPAM_CLASS, y = testLogLikelihood)) +
```

```
  geom_boxplot() +
  ylim(-500,500) +
  ylab("Log Likelihood Ratio Distribution") +
  xlab("Message Classification") +
  ggtitle("Distribution of Randomly Selected Messages\n by  Classification") +
  #labs(caption = paste(
  #    'Figure 1:\n',
  #    ' NB', sep='')
  #    ) +
  theme_minimal() +
   theme(
      plot.title = element_text(hjust = 0.5),
      plot.caption = element_text(hjust = 0)
  )

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))
    }
)
SpamCheckWords =
  c("viagra", "pounds", "free", "weight", "guarantee", "million",
    "dollars", "credit", "risk", "prescription", "generic", "drug",
    "financial", "save", "dollar", "erotic", "million", "barrister",
    "beneficiary", "easy",
    "money back", "money", "credit card")
getMessageRecipients =
  function(header)
  {
    c(if("To" %in% names(header))  header[["To"]] else character(0),
      if("Cc" %in% names(header))  header[["Cc"]] else character(0),
      if("Bcc" %in% names(header)) header[["Bcc"]] else character(0)
    )
  }
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)))
}
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)
}
readEmail = function(dirName) {
       # retrieve the names of files in directory
```

```
  fileNames = list.files(dirName, full.names = TRUE)
       # drop files that are not email
  notEmail = grep("cmds$", fileNames)
  if ( length(notEmail) > 0) fileNames = fileNames[ - notEmail ]
       # read all files in the directory
  lapply(fileNames, readLines, encoding = "latin1")
}
processAllEmail = function(dirName, isSpam = FALSE)
{
       # read all files in the directory
  messages = readEmail(dirName)
  fileNames = names(messages)
  n = length(messages)

       # split header from body
  eSplit = lapply(messages, splitMessage)
  rm(messages)
       # process header as named character vector
  headerList = lapply(eSplit, function(msg)
                                processHeader(msg$header))

       # extract content-type key
  contentTypes = sapply(headerList, function(header)
                                     header["Content-Type"])

       # extract the body
  bodyList = lapply(eSplit, function(msg) msg$body)
  rm(eSplit)
       # which email have attachments
  hasAttach = grep("^ *multi", tolower(contentTypes))
       # get summary stats for attachments and the shorter body
  attList = mapply(processAttach, bodyList[hasAttach],
                  contentTypes[hasAttach], SIMPLIFY = FALSE)

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

  attachInfo = vector("list", length = n )
  attachInfo[ hasAttach ] = lapply(attList,
                                function(attEl) attEl$attachDF)

       # prepare return structure
  emailList = mapply(function(header, body, attach, isSpam) {
```

```
                      list(isSpam = isSpam, header = header,
                           body = body, attach = attach)
                    },
                    headerList, bodyList, attachInfo,
                    rep(isSpam, n), SIMPLIFY = FALSE )
  names(emailList) = fileNames

  invisible(emailList)
}
emailStruct = mapply(processAllEmail, filldirNames,
                    isSpam = rep( c(FALSE, TRUE), 3:2))
emailStruct = unlist(emailStruct, recursive = FALSE)
createDerivedDF =
function(email = emailStruct, operations = funcList,
         verbose = FALSE)
{
  els = lapply(names(operations),
               function(id) {
                 if(verbose) print(id)
                 e = operations[[id]]
                 v = if(is.function(e))
                        sapply(email, e)
                      else
                        sapply(email, function(msg) eval(e))
                 v
         })
   df = as.data.frame(els)
   names(df) = names(operations)
   invisible(df)
}
emailDF = createDerivedDF(emailStruct)
#dim(emailDF)
setupRpart = function(data) {
  logicalVars = which(sapply(data, is.logical))
  facVars = lapply(data[ , logicalVars],
                   function(x) {
                      x = as.factor(x)
                      levels(x) = c("F", "T")
                      x
                   })
  cbind(facVars, data[ , - logicalVars])
}
emailDFrp = setupRpart(emailDF)
```

```
set.seed(418910)
testSpamIdx = sample(spamCnt, size = floor(spamCnt/3))
testHamIdx = sample(legitCnt, size = floor(legitCnt/3))
testDF =
  rbind( emailDFrp[ emailDFrp$isSpam == "T", ][testSpamIdx, ],
         emailDFrp[emailDFrp$isSpam == "F", ][testHamIdx, ] )
trainDF =
  rbind( emailDFrp[emailDFrp$isSpam == "T", ][-testSpamIdx, ],
         emailDFrp[emailDFrp$isSpam == "F", ][-testHamIdx, ])
rpartFit = rpart(isSpam ~ ., data = trainDF, method = "class")

predictions = predict(rpartFit,
       newdata = testDF[, names(testDF) != "isSpam"],
       type = "class")

predsForHam = predictions[ testDF$isSpam == "F" ]
#summary(predsForHam)
#sum(predsForHam == "T") / length(predsForHam)
predsForSpam = predictions[ testDF$isSpam == "T" ]
#sum(predsForSpam == "F") / length(predsForSpam)

# create a function to calculate the F1 summary metrics
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
   )
}

# split the dataset into training and test
testDF =
  rbind( emailDF[ emailDF$isSpam == TRUE, ][testSpamIdx, ],
         emailDF[emailDF$isSpam == FALSE, ][testHamIdx, ] )
trainDF =
  rbind( emailDF[emailDF$isSpam == TRUE, ][-testSpamIdx, ],
```

```
         emailDF[emailDF$isSpam == FALSE, ][-testHamIdx, ])
setupRnum = function(data) {
  logicalVars = which(sapply(data, is.logical))
  facVars = lapply(data[ , logicalVars],
                   function(x) {
                      x = as.numeric(x)
                   })
  cbind(facVars, data[ , - logicalVars])
}
testDF = setupRnum(testDF)
testDF[is.na(testDF)]<-0
trainDF = setupRnum(trainDF)
trainDF[is.na(trainDF)]<-0

# naive bayes model
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=trainDF,
                       trControl = train_control,
                       method='naive_bayes',
                       tuneGrid = nb_grid)
# Partition Tree Model
cp_val <- seq(from = 0, to=0.01, by=0.0005)
cart_grid<-expand.grid(cp=cp_val)
train_control<-trainControl(method="cv",
                            number =5,
                            savePredictions = 'final',
                            summaryFunction = f1)
model_rpart <- caret::train(as.factor(isSpam) ~ .,
                          data=trainDF,
                          trControl = train_control,
                          method='rpart',
                          tuneGrid = cart_grid)
# logitsic model
train_control <- trainControl(method="cv",
                            number=3,
                            savePredictions = 'final',
```

```
                            summaryFunction = f1)
model_glm <- caret::train(as.factor(isSpam) ~ .,
                          data=trainDF,
                          trControl = train_control,
                          method='glm',

# Creating a function to build the confusion matrix visuals
draw_confusion_matrix <- function(cm, type) {

  layout(matrix(c(1,1,2)))
  par(mar=c(2,2,2,2))
  plot(c(100, 345), c(300, 450), type = "n", xlab="", ylab="", xaxt='n', yaxt='n')
    title(paste0(type," Confusion Matrix", sep=''), cex.main=2)

  # create the matrix
  rect(150, 430, 240, 370, col='#66ff66')
  text(195, 435, 'Not Spam', cex=1.2)
  rect(250, 430, 340, 370, col='#ff0000')
  text(295, 435, 'Spam', cex=1.2)
  text(125, 370, 'Predicted', cex=1.3, srt=90, font=2)
  text(245, 450, 'Actual', cex=1.3, font=2)
  rect(150, 305, 240, 365, col='#ff0000')
  rect(250, 305, 340, 365, col='#66ff66')
  text(140, 400, 'Not Spam', cex=1.2, srt=90)
  text(140, 335, 'Spam', cex=1.2, srt=90)

  # add in the cm results
  res <- as.numeric(cm$table)
  text(195, 400, res[1], cex=1.6, font=2, col='white')
  text(195, 335, res[2], cex=1.6, font=2, col='white')
  text(295, 400, res[3], cex=1.6, font=2, col='white')
  text(295, 335, res[4], cex=1.6, font=2, col='white')

  # add in the specifics
  plot(c(100, 0), c(100, 0), type = "n", xlab="", ylab="", main = "DETAILS", xaxt='n', yaxt='n')
  text(10, 85, names(cm$byClass[1]), cex=1.2, font=2)
  text(10, 70, round(as.numeric(cm$byClass[1]), 3), cex=1.2)
  text(30, 85, names(cm$byClass[2]), cex=1.2, font=2)
  text(30, 70, round(as.numeric(cm$byClass[2]), 3), cex=1.2)
  text(50, 85, names(cm$byClass[5]), cex=1.2, font=2)
  text(50, 70, round(as.numeric(cm$byClass[5]), 3), cex=1.2)
  text(70, 85, names(cm$byClass[6]), cex=1.2, font=2)
  text(70, 70, round(as.numeric(cm$byClass[6]), 3), cex=1.2)
```

```
  text(90, 85, names(cm$byClass[7]), cex=1.2, font=2)
  text(90, 70, round(as.numeric(cm$byClass[7]), 3), cex=1.2)

  # add in the accuracy information
  text(30, 35, names(cm$overall[1]), cex=1.5, font=2)
  text(30, 20, round(as.numeric(cm$overall[1]), 3), cex=1.4)
  text(70, 35, names(cm$overall[2]), cex=1.5, font=2)
  text(70, 20, round(as.numeric(cm$overall[2]), 3), cex=1.4)
}

# get the results on the test set
model_nb_y_hat <- predict(model_nb, newdata = select(testDF, -isSpam))
model_rpart_y_hat <- predict(model_rpart, newdata = select(testDF, -isSpam))
model_glm_y_hat <- predict(model_glm, newdata = select(testDF, -isSpam))

# Printing the confusion matrix for Naive Bayes using the test data set
cfm_nb <- confusionMatrix(as.factor(model_nb_y_hat), as.factor(testDF$isSpam))
draw_confusion_matrix(cfm_nb, type="Naive Bayes")

# Printing the confusion matrix for Logistic Regression Generalized Linear Model using the test
data set
cfm_logistic <- confusionMatrix(as.factor(model_glm_y_hat), as.factor(testDF$isSpam))
draw_confusion_matrix(cfm_logistic, type="Logistic Regression")


# Printing the confusion matrix for the Partition Tree using the test data set
cfm_rpart <- confusionMatrix(as.factor(model_rpart_y_hat), as.factor(testDF$isSpam))
draw_confusion_matrix(cfm_rpart, type="Partition Tree")
```