STAT480 Homework4

Chenz Zhang, NetID chenziz2

2/18/2019

Contents

In this homework, first I run the "HW4Setup.R" to get functions which will be used in this exercise. There are some warnings which are about incomplete final line in 3 emails. Please omit them.

```r
# The following is based on the code for Chapter 3 of
#
# Nolan, Deborah and Temple Lang, Duncan. Data Science in R: A Case Studies Approach to
# Computational Reasoning and Problem Solving. CRC Press, 2015.
# http://rdatasciencecases.org/
#
# the original code is provided on
# http://rdatasciencecases.org/Spam/code.R
#
# Modifications are by Darren Glosemeyer for the Spring 2016 through 2019 sections of
# Stat 480: Data Science Foundations at the University of Illinois at Urbana-Champaign.
#

# This first chunk of code is needed to define a few utility functions and get results for
# Naive Bayes probabilities and logodds based on words in messages. In retrospect, it probably
# would have been useful to store the word lists and the probability and logodds table to
# R data files.
spamPath = "~/Stat480/RDataScience/SpamAssassinMessages"

dirNames = list.files(path = paste(spamPath, "messages",
```

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

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)

splitMessage = function(msg) {
  splitPoint = match("", msg)
  header = msg[1:(splitPoint-1)]
  body = msg[ -(1:splitPoint) ]
  return(list(header = header, body = body))
}

getBoundary = function(header) {
  boundaryIdx = grep("boundary=", header)
  boundary = gsub('"', "", header[boundaryIdx])
  gsub(".*boundary= *([^;]*);?.*", "\\1", boundary)
}

dropAttach = function(body, boundary){

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

  # if there are fewer than 2 beginning boundary strings,
  # there is on attachment to drop
  if (length(bStringLocs) <= 1) return(body)

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

  # if no ending boundary string, grab contents between the first
  # two beginning boundary strings as the message body
  if (length(eStringLoc) == 0)
    return(body[ (bStringLocs[1] + 1) : (bStringLocs[2] - 1)])

  # typical case of well-formed email with attachments
  # grab contents between first two beginning boundary strings and
  # add lines after ending boundary string
  n = length(body)
  if (eStringLoc < n)
    return( body[ c( (bStringLocs[1] + 1) : (bStringLocs[2] - 1),
                     ( (eStringLoc + 1) : n )) ] )

  # fall through case
  # note that the result is the same as the
  # length(eStringLoc) == 0 case, so code could be simplified by
  # dropping that case and modifying the eStringLoc < n check to
  # be 0 < eStringLoc < n
  return( body[ (bStringLocs[1] + 1) : (bStringLocs[2] - 1) ])
```

```
}

library(tm)
```

```
## Loading required package: NLP
```

```
stopWords = stopwords()
cleanSW = tolower(gsub("[[:punct:]0-9[:blank:]]+", " ", stopWords))
SWords = unlist(strsplit(cleanSW, "[[:blank:]]+"))
SWords = SWords[ nchar(SWords) > 1 ]
stopWords = unique(SWords)

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

findMsgWords =
  function(msg, stopWords) {
    if(is.null(msg))
      return(character())

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

    # drop empty and 1 letter words
    # The allowNA=TRUE option is a modification from the text's code.
    # In current versions of R, this is needed to eliminate cases with
    # unrecognized characters which now return NA but previously returned
    # numbers.
    words = words[ nchar(words, allowNA=TRUE) > 1]
    words = words[ !( words %in% stopWords) ]
    invisible(words)
  }

processAllWords = function(dirName, stopWords)
{
  # read all files in the directory
  fileNames = list.files(dirName, full.names = TRUE)
  # drop files that are not email, i.e., cmds
  notEmail = grep("cmds$", fileNames)
  if ( length(notEmail) > 0) fileNames = fileNames[ - notEmail ]

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

  # split header and body
  emailSplit = lapply(messages, splitMessage)
  # put body and header in own lists
  bodyList = lapply(emailSplit, function(msg) msg$body)
  headerList = lapply(emailSplit, function(msg) msg$header)
  rm(emailSplit)

  # determine which messages have attachments
  hasAttach = sapply(headerList, function(header) {
    CTloc = grep("Content-Type", header)
```

```r
    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)
}

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

```
## Warning in FUN(X[[i]], ...): incomplete final line found on '/home/
## chenziz2/Stat480/RDataScience/SpamAssassinMessages/messages/hard_ham/
## 00228.0eaef7857bbbf3ebf5edbbdae2b30493'

## Warning in FUN(X[[i]], ...): incomplete final line found on '/home/
## chenziz2/Stat480/RDataScience/SpamAssassinMessages/messages/hard_ham/
## 0231.7c6cc716ce3f3bfad7130dd3c8d7b072'

## Warning in FUN(X[[i]], ...): incomplete final line found on '/home/
## chenziz2/Stat480/RDataScience/SpamAssassinMessages/messages/hard_ham/
## 0250.7c6cc716ce3f3bfad7130dd3c8d7b072'
```

```r
numMsgs = sapply(msgWordsList, length)

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

msgWordsList = unlist(msgWordsList, recursive = FALSE)


# Determine number of spam and ham messages for sampling.
numEmail = length(isSpam)
numSpam = sum(isSpam)
numHam = numEmail - numSpam

# Set a particular seed, so the results will be reproducible.
set.seed(418910)

# Take approximately 1/3 of the spam and ham messages as our test spam and ham messages.
testSpamIdx = sample(numSpam, size = floor(numSpam/3))
testHamIdx = sample(numHam, size = floor(numHam/3))
```

```r
# Use the test indices to select word lists for test messages.
# Use training indices to select word lists for training messages.
testMsgWords = c((msgWordsList[isSpam])[testSpamIdx],
                 (msgWordsList[!isSpam])[testHamIdx] )
trainMsgWords = c((msgWordsList[isSpam])[ - testSpamIdx],
                  (msgWordsList[!isSpam])[ - testHamIdx])

# Create variables indicating which testing and training messages are spam and not.
testIsSpam = rep(c(TRUE, FALSE),
                 c(length(testSpamIdx), length(testHamIdx)))
trainIsSpam = rep(c(TRUE, FALSE),
                  c(numSpam - length(testSpamIdx),
                    numHam - length(testHamIdx)))

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 1/2 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)
  }

# Obtain the probabilities and log odds for the training data.
trainTable = computeFreqs(trainMsgWords, trainIsSpam)
```

```r
# Load the emailStruct and emailDF from the .rda files they were stored in.
load("~/Stat480/RDataScience/Chapter3/emailXX.rda")

load("~/Stat480/RDataScience/Chapter3/spamAssassinDerivedDF.rda")
```

Exercise 1

1) Define my function and apply it:

```r
#define my function
computeMsgLLR2 = 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

  # First part of LLR2
  odd1 = log(prod(freqTable["spam", present])/prod(freqTable["ham", present]))

  # Second part of LLR2
  odd2 = log(prod(1-freqTable["spam",!present])/prod(1-freqTable["ham",!present]))

  # sum LLR2
  odd1 + odd2

}

testLLR2 = sapply(testMsgWords, computeMsgLLR2, trainTable)
head(testLLR2)
```

```
## [1]       NaN 169.60217 113.47581  51.66730  38.58663 -28.36873
```

```r
test2 = tapply(testLLR2, testIsSpam)
tapply(testLLR2, testIsSpam, summary)
```

```
## $`FALSE`
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
##    -Inf -119.88  -97.47     NaN  -79.60     Inf     272
##
## $`TRUE`
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
##  -60.57   -1.70   38.64     Inf   98.68     Inf     113
```

2) Compare the accuracy from `computeMsgLLR`

Cite the `computeMsgLLR` from Chapter 3.6:

```r
computeMsgLLR = function(words, freqTable)
{
  # Discards words not in training data.
  words = words[!is.na(match(words, colnames(freqTable)))]
```

```r
  # Find which words are present
  present = colnames(freqTable) %in% words

  sum(freqTable["presentLogOdds", present]) +
    sum(freqTable["absentLogOdds", !present])
}

testLLR = sapply(testMsgWords, computeMsgLLR, trainTable)
head(testLLR)
```

```
## [1] 255.06434 169.60217 113.47581  51.66730  38.58663 -28.36873
```

```r
test1 = tapply(testLLR, testIsSpam)
tapply(testLLR, testIsSpam, summary)
```

```
## $`FALSE`
##     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
## -1360.82  -126.98  -101.18  -116.16   -81.23   700.27
##
## $`TRUE`
##      Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
##   -60.567     6.076    50.174   137.325   130.638 23567.528
```

Accuracy by $(observed - expected)/expected$:

```r
accuracyLLR = (testLLR2-testLLR)/testLLR
summary(accuracyLLR)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
##    -Inf       0       0     NaN       0     Inf     385
```

Apart from `Inf`, `NaN` and `NA`, we can say over 75% test samples have observed value less than expected value. And from the `Median`, we can say medianly the observed value is 56% less than expected value. But, still we have `Min.` and `Max.` as `Inf`, which makes the accuracy worse.

After cleaning all the `-Inf`, `Inf`, `NaN`, `NA`, we can calculate the clean accuracy as following:

```r
# clean infinite elements
cleanAccLLR = accuracyLLR[!is.infinite(accuracyLLR)]
cleanAccLLR = cleanAccLLR[!is.nan(cleanAccLLR)]
cleanAccLLR = cleanAccLLR[!is.na(cleanAccLLR)]
summary(cleanAccLLR)
```

```
##       Min.    1st Qu.     Median       Mean    3rd Qu.       Max.
## -2.604e-03  0.000e+00  0.000e+00 -7.462e-07  0.000e+00  2.570e-03
```

The result is quite similar. Even though we cleaned the `Inf`, we still get extramely large or small extrame values. So, inaccuracy is obvious.

3) Commends on non-representable numbers

```r
summary(unname(testLLR2))
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
##    -Inf -110.45  -85.63     NaN  -42.47     Inf     385
```

From the `computeMsgLLR2` result, we can find that:

1. `NaN`'s mean 0/0. This comes from the division part inside `log()` function, says, `log(0/0)`. So, the product of probability as numerators and denominators might be very small.
2. `-Inf`'s mean the numerators α in `log()` have $\alpha \to 0$. The product of prob is very small and even close to 0 but the denominators are not close to 0.
3. `Inf`'s mean the denominators in `log()` are close to 0 but the numerators are not close to 0.
4. `NA`'s mean missing values.

Exercise 2

Write a function to handle an alternative approach to measure yelling: count the number of lines in the email message text that consist entirely of capital letters. Carefully consider the case where the message body is empty. How do you modify your code to report a percentage rather than a count? In considering this modification, be sure to make clear how you handle empty lines, lines with no alpha-characters, and messages with no text.

1) Construct yell line couting function:

```
# Construct a function counting the number of yell lines
yell.Lines = function(messages){
  body = messages$body
  # check and drop attachment
  if(is.null(messages$attach) == FALSE){
    boundary = getBoundary(messages$header)
    body = dropAttach(body,boundary)
  }

  if(length(body) == 0) return(NA) #*****Handle Empty Body******

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

  # count entirely capital lines
  count = 0
  check = text
  for(i in 1:length(text)){
    if(nchar(text[i]) > 0){                    #*****Handle Empty Lines******
      check[i] = gsub("[A-Z]","",text[i])
      if(nchar(check[i]) < 1){              #*****Handle lines without alpha-characters
          count = count + 1
      }
    }
  }
    return(count)
}
```

2) Construct yell line percent function:

```
# Construct a function calculating the percentage
yell.percent = function(messages){
```

```
  body = messages$body
  # check and drop attachment
  if(is.null(messages$attach) == FALSE){
    boundary = getBoundary(messages$header)
    body = dropAttach(body,boundary)
  }

  if(length(body) == 0) return(NA) #*****Handle Empty Body******

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

  # count entirely capital lines
  count = rep(0,length(text))
  check = text
  for(i in 1:length(text)){
    if(nchar(text[i]) > 0){
      check[i] = gsub("[A-Z]","",text[i])     #*****Handle Empty Lines******
      if(check[i] < 1){
        count[i] = 1
      }                #*****Handle lines without alpha-characters
    }
  }

  return(round(sum(count, na.rm = TRUE)/length(text)*100,4)) #Denominator includes empty lines

}
```

3) Result display

Next, we try to apply the 1st function onto 15 sample emails.

```
indx = c(1:5, 15, 27, 68, 69, 329, 404, 427, 516, 852, 971)
sampleStruct = emailStruct[ indx ]

unlist(unname(sapply(sampleStruct,yell.Lines)))
```

```
##  [1] 0 0 0 0 0 0 1 0 7 0 0 0 0 0 0
```

```
unlist(unname(sapply(sampleStruct,yell.percent)))
```

```
##  [1]  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  2.8571  0.0000
##  [9] 10.7692  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000
```

We can see 7th and 9th sample email have yell lines:
* 7th: 1 line, 2.8571%. * 9th: 7 lines, 10.7692%.

Then, we'll apply these two functions onto all emails, and show the result on first 30 emails:

```
yellLines = unlist(unname(lapply(emailStruct,yell.Lines)))
yellPercents = unlist(unname(lapply(emailStruct,yell.percent)))
result2 = cbind(yellLines,yellPercents)
colnames(result2) = c("Yell Lines", "Yell Line Percentage%")
row.names(result2) = seq(1,nrow(result2))
head(result2, n = 30L)
```

```
##    Yell Lines Yell Line Percentage %
## 1           0                0.0000
## 2           0                0.0000
## 3           0                0.0000
## 4           0                0.0000
## 5           0                0.0000
## 6           0                0.0000
## 7           0                0.0000
## 8           0                0.0000
## 9           0                0.0000
## 10          0                0.0000
## 11          0                0.0000
## 12          0                0.0000
## 13          0                0.0000
## 14          0                0.0000
## 15          0                0.0000
## 16          0                0.0000
## 17          0                0.0000
## 18          0                0.0000
## 19          0                0.0000
## 20          0                0.0000
## 21          0                0.0000
## 22          0                0.0000
## 23          0                0.0000
## 24          0                0.0000
## 25          0                0.0000
## 26          1                1.9231
## 27          1                2.8571
## 28          0                0.0000
## 29          0                0.0000
## 30          0                0.0000
```

Exercise 3

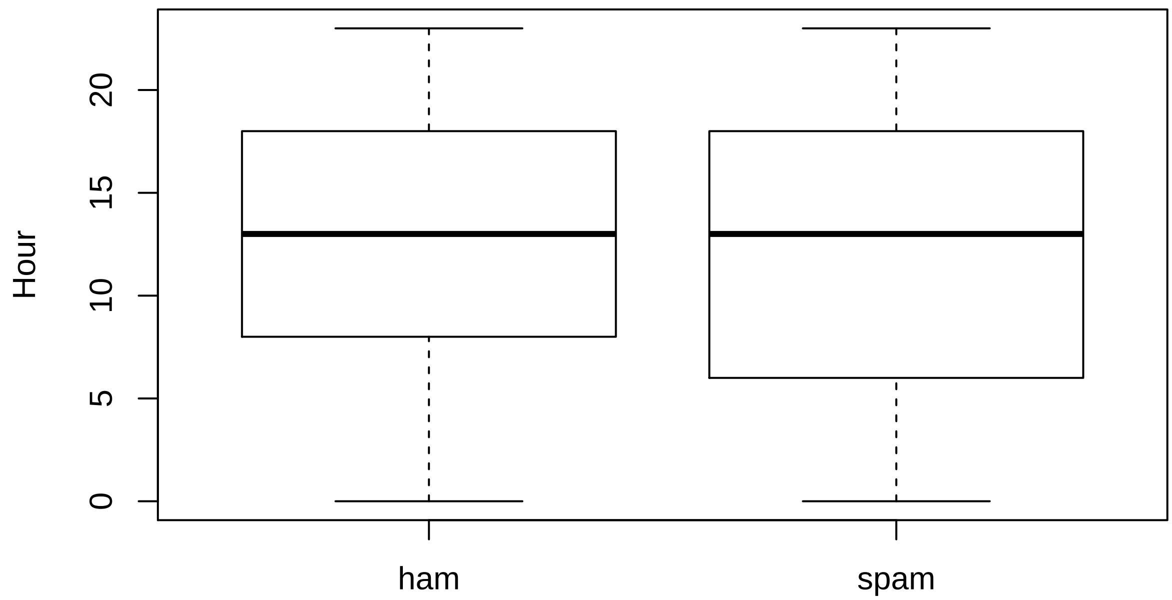
1) Box Plot

```r
length(emailDF$hour) == length(which(emailDF$hour != 'NA'))
```

```
## [1] TRUE
```

```r
#1. Box plot comparing percent capitalization in ham and spam

times = emailDF$hour
isSpamLabs = factor(emailDF$isSpam, labels = c("ham", "spam"))
boxplot(times ~ isSpamLabs,
        ylab = "Hour")
```

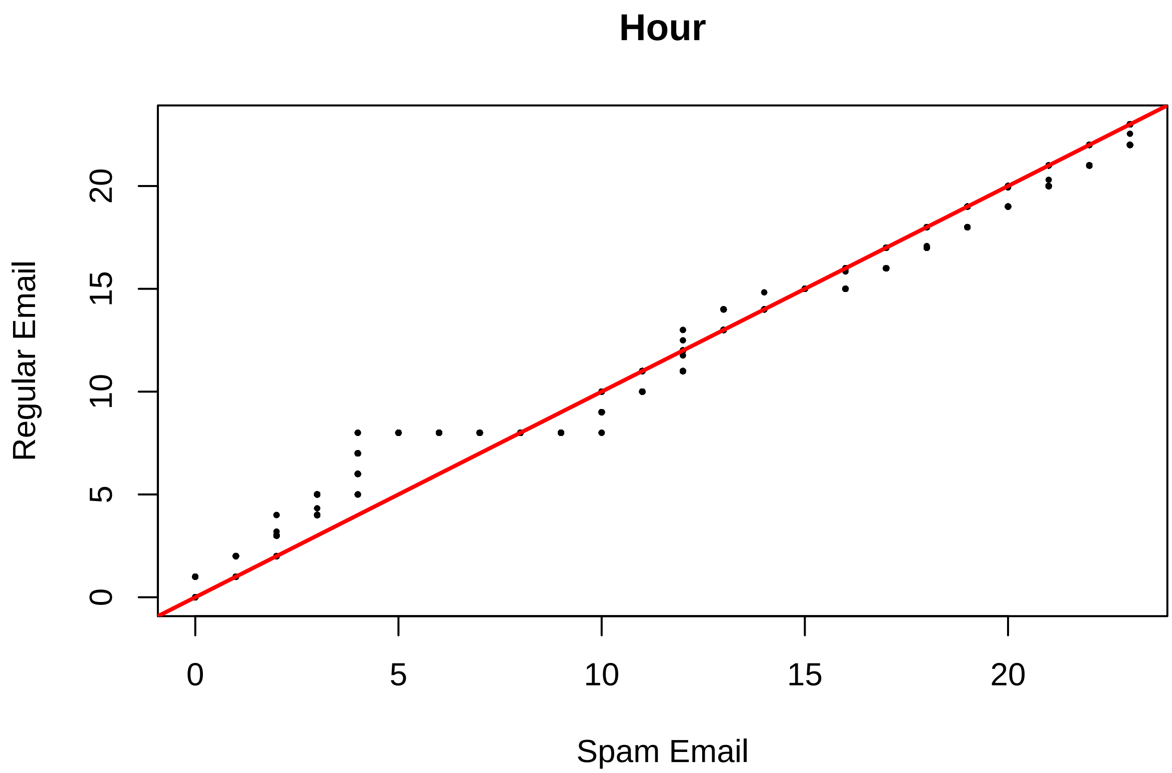


From Box plot, we can see that the middle 50% ranges for ham and spam email hour are roughly similar. Hour range of ham is 8 to 18. Hour range of spam is 5 to 18. Also the medians look the same. So, we cannot use hour as a standard to distinguish spam and ham emails.

2) Other Plot

```
#2. Q-Q plot comparing percent capitalization in ham and spam
timesSpam = emailDF$hour[ emailDF$isSpam ]
timesHam = emailDF$hour[ !emailDF$isSpam ]

# The following corrects the labeling of the x and y axes. The author's
# code has the x and y labeling reversed.
{qqplot(timesSpam, timesHam,
       xlab = "Spam Email", ylab = "Regular Email",
       main = "Hour",
       pch = 19, cex = 0.3)
abline(a=0, b=1, col="red", lwd = 2)}
```



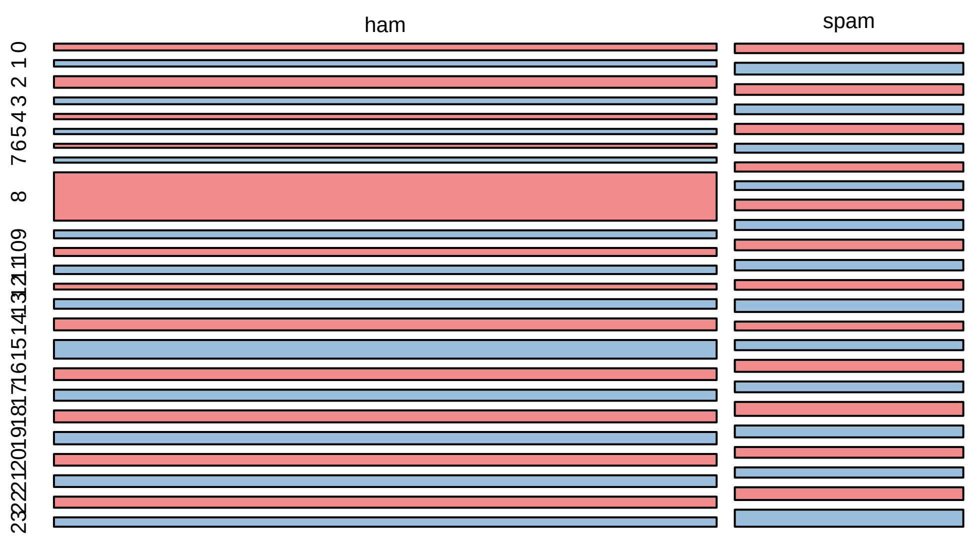

```r
#3. See a tabulation of number of attachments by email type
table(emailDF$hour, isSpamLabs)
```

```
##     isSpamLabs
##       ham spam
##   0   181   87
##   1   172  108
##   2   308   98
##   3   181   89
##   4   142   94
##   5   140   82
##   6   107   85
##   7   138   80
##   8  1288   98
##   9   213   92
##   10  213   98
##   11  224   96
##   12  154   89
##   13  256  115
##   14  318   81
##   15  496   93
##   16  313  110
##   17  294   99
##   18  318  128
##   19  328  109
##   20  311   98
```

```
##   21  311   94
##   22  295  117
##   23  250  157
```

```r
#4. visualize ham and spam counts for hour

colM = c("#E41A1C80", "#377EB880")
times2 = factor(emailDF$hour, labels = as.character(seq(0,23)))
mosaicplot(table(isSpamLabs, times2), main = "",
           xlab = "", ylab = "", color = colM)
```



From these result, we still have the conclusion that hour cannot be chosen as standard for classify ham and spam emails. The only thing we can say is that most of the emails at 8 am are ham emails.

3) Not Necessarily Important Part

```r
#5. Prob
library(rpart)

# Function to replace logical variables with factor 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")
```

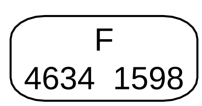
```r
                     x
                   })
  cbind(facVars, data[ , - logicalVars])
}

# Process the email data frame.
emailDFrp = setupRpart(emailDF)

# Get spam and ham indices. These are the same samples chosen in section 3.6.1.
set.seed(418910)
testSpamIdx = sample(numSpam, size = floor(numSpam/3))
testHamIdx = sample(numHam, size = floor(numHam/3))

testDF =
  rbind( emailDFrp[ emailDFrp$isSpam == "T", ][testSpamIdx, ],
         emailDFrp[emailDFrp$isSpam == "F", ][testHamIdx, ] )[,c("isSpam","hour")]
trainDF =
  rbind( emailDFrp[emailDFrp$isSpam == "T", ][-testSpamIdx, ],
         emailDFrp[emailDFrp$isSpam == "F", ][-testHamIdx, ])[,c("isSpam","hour")]
rpartFit = rpart(isSpam ~ hour, data = trainDF, method = "class")


library(rpart.plot)
prp(rpartFit, extra = 1)
```



```r
new <- data.frame(hour = testDF[, "hour"])
predictions = predict(rpartFit,
                      newdata = new,
                      type = "class")


predsForHam = predictions[ testDF$isSpam == "F" ]
summary(predsForHam)
```

```
##    F    T
## 2317    0
```

```r
# Obtain the Type I error rate.
sum(predsForHam == "T") / length(predsForHam)
```

```
## [1] 0
```

```r
# Obtain the Type II error rate.
predsForSpam = predictions[ testDF$isSpam == "T" ]
sum(predsForSpam == "F") / length(predsForSpam)
```

```
## [1] 1
```

Still not work.

Exercise 4

1)`isRe` in Chapter 3.9

```r
isRe = function(msg) {
    "Subject" %in% names(msg$header) &&
      length(grep("^[ \t]*Re:", msg$header[["Subject"]])) > 0
}
```

2) `isRe2` also checks `Fwd: Re:`

```r
isRe2 = function(msg){
  "Subject" %in% names(msg$header) &&
    length(grep("^[ \t]*(Fwd:)?[\t]*Re:", msg$header[["Subject"]])) > 0
}
```

3) `isRe3` checks `Re:` anywhere in the subject

```r
isRe3 = function(msg){
  "Subject" %in% names(msg$header) &&
    length(grep(".*Re:.*", msg$header[["Subject"]])) > 0
}
```

4) Apply to Emails

```
re.1 = unlist(sapply(emailStruct,isRe))
re.2 = unlist(sapply(emailStruct,isRe2))
re.3 = unlist(sapply(emailStruct,isRe3))
```

Display the sum of `TRUE` lines for each function:

```
re = c(sum(re.1),sum(re.2),sum(re.3))
re
```

```
## [1] 3005 3005 3218
```

5) Analyze:

1. The oringinal `isRe` function greps string begin with `Re:`, allowing `Tab`'s before `Re:`, in the Subject line of header. Return logical value for the existance of this string.

2. The second `isRe2` function written by me greps string begin with `Re:` or `Fwd: Re`, allowing `Tab`'s before `Fwd:` and `Re:`, in the Subject line of header. Return logical value for the existance of this string. The count this function returned should be more than or equal to the first function.

3. The third `isRe3` function written by me greps string with `Re:` no matter any string before or after `Re:` in the Subject line of header. Return logical value for the existance of this string. The count this function returned should be more than or equal to the first function.

6) Comparation:

The result from `isRe2 - isRe` and `isRe3 - isRe` are as following:

```
sum(re.2) - sum(re.1)
```

```
## [1] 0
```

```
sum(re.3) - sum(re.1)
```

```
## [1] 213
```

The results from `isRe` and `isRe2` are the same: 0 messages have a return value of TRUE for these alternatives. But those of `isRe` and `isRe3` are different: 213 messages have a return value of TRUE for these alternatives.

Let's look at the difference between the results for `isRe` and `isRe3`. `isRe3` has more emails as following type:

```
diffIndx = which(re.3 != re.1)
sum(re.3[diffIndx] == TRUE) == length(diffIndx) #check if the isRe3 return is TRUE, if TRUE, return TRU
```

```
## [1] TRUE
```

```
diffnum = length(diffIndx)
diffham = diffnum - sum((emailDF$isSpam)[diffIndx])
diffcount = c(diffham,diffnum - diffham)
perdiff = round(diffcount/diffnum,4)*100

diffmatrix = matrix(rbind(diffcount,perdiff),2,2)
rownames(diffmatrix) = c("number","percentage(%)")
colnames(diffmatrix) = c("ham","spam")
print(diffmatrix)
```

```
##                    ham spam
## number          204.00 9.00
## percentage(%)    95.77 4.23
```

95.77% different emails in `isRe3` are ham.

```r
par(mfrow = c(1,2))
colM = c("#E41A1C80", "#377EB880")
isRe = factor(emailDF$isRe, labels = c("no Re:", "Re:"))
mosaicplot(table(isSpamLabs, isRe), main = "Result from original isRe",
xlab = "", ylab = "", color = colM)


isRe3 = factor(re.3, labels = c("no Re:", "Re:"))
mosaicplot(table(isSpamLabs, isRe3), main = "Result from my isRe3",
xlab = "", ylab = "", color = colM)
```

The difference between percentages of `(ham,Re:)` and `(spam, Re:)` gets larger. So, the last isRe function, i.e. `isRe3`, is the most useful one in predicting spam.