Case Study: Spam Email Prediction

Jaclyn Coate, Megan Riley, Reagan Meagher

2/15/2021

1. Introduction

The digital world is something that has increasingly changed the way organizations and businesses communicate with their consumers. As digital memory becomes more cost effective it takes no effort at all for a company to render and send large volumes of emails in an attempt to engage their end users. While is a positive shift for companies in their effort to grow their business, consumers are inundated with spam emails that are problematic for those who use their email for other purposes than just shopping. Individuals can easily spot their spam emails and skip or delete them. But this manual effort can be strenuous and take a long time based on sheer volume of spam emails alone. This may not be a big issue for the casual email users who only use it for online shopping and occasional correspondence. But the individuals who live in the digital world would much rather not spend their time sorting email, but rather leveraging their email for their end goals.

This leads us to the case study we present today; below are our efforts to make sure we have a strong spam email filter to take the burden off of email users for manually filtering their emails manually. The intuitive process starts with reviewing the subject line, and if we fail to identify the email as spam we begin to analyze the body. Between these two elements we are almost guaranteed to know if the email is from someone we do want to engage in correspondence with. Best practice is to never open an attachment in a spam email so even our automated processes would simply strip this from our view. Our research and subsequent automated process for an algorithmic procedure closely resembles the process a human might go through in order to determine if their email is spam. With cleaning and examining the subject and body elements of an email programmatically we can leverage extremely powerful machine learning algorithmic classifiers such as: *NaiveBayes*, *Decisions Tree Classifier*, *Random Forest*, and *XGBoost* to help us automate the classification of emails in our inbox.

```r
library(tm)
library(naivebayes)
library(e1071)
library(rpart)
library(randomForest)
library(xgboost)
library(caret)
library(MLmetrics)
library(RColorBrewer)
library(tidyverse)
Sys.setlocale("LC_ALL", "pt_PT.ISO8859-1")
```

```
## [1] "pt_PT.ISO8859-1/pt_PT.ISO8859-1/pt_PT.ISO8859-1/C/pt_PT.ISO8859-1/en_US.UTF-8"
```

2. Data Cleaning

2.1 Background

These data for this case study consisted of 9000 emails that have been hand classified into *spam* and *non-spam* categories, the *non-spam* categories referred to as *ham*. This was done my Apache SpamAssassin, the premier open source platform for spam detection. We obtained this dataset through the Public Corpus generated by the Apache SpamAssassin.

The data are classified as *ham*(*non-spam*) or *spam* emails. The *non-spam* training data is split into two types; *hard ham* and *easy ham*. *Hard ham* are still *non-spam* emails however they carry some of the same characteristics as *spam* emails and are more difficult to classify.

Overall it is important to note as we build models for the purpose of classifying *spam* emails from this corpus of data that the emails themselves are at minimum nearly fifteen years old. *Spam* techniques along with online communication can change drastically in a year, much less fifteen years. It is most likely any models or techniques built from this corpus of emails will have to be changed and edited in the future for any further potential use at_spam__ classification.

To begin the exploration of the data we begin the process of cleaning and working to explore the distribution of words and other aspects of the messages.

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

Below we build a directory path locally on one machine, with the overall directory, messages can be processed by the previous functions. The stop words of interest are also cleaned and set up using a general stop word lexicon for this process.

```r
#Directory paths
spamPath =
  "/Users/zmartygirl/githubrepos1/7333_Quantifying_The_World/Unit6_CaseStudy3/data"
dirNames = list.files(path = paste(spamPath, sep = .Platform$file.sep))
fullDirNames = paste(spamPath, dirNames, sep = .Platform$file.sep)
#Set up Stop Words
stopWords = stopwords()
```

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

With preparations done, we can load all the available messages into our session and clean and organize them into a set of words for each message. The entirety of this happens within the *processAllWords* function. We can see we have over nine thousand messages, with 5051, 1400, and 500 in separate directories of *non-spam* emails and 998 and 1396 emails in the *spam* category.

```
## [1] 5051 1400  500  998 1396
```

We then add a label of true or false to every message in our email corpus to identify if it is *spam* or *non-spam*. We know our first three directories are *non-spam* and our final two are *spam*, so they are labeled as such below.

```
#Below we are adding the true labels on if messagse are spam or not spam.
#This is our known determination because it was hand labeled.
isSpam = rep(c(FALSE, FALSE, FALSE, TRUE, TRUE), numMsgs)
msgWordsList = unlist(msgWordsList, recursive = FALSE)
```

Next we are building an index of messages and train and test split of the messages and classification labels. We also can see that there are eighty thousand unique words in the set of training data given by the length of our bag of words.

Below we are creating a classification test dataset. There is potential for no words in the training data. Below we are calculating the log odds without new words.

```
## [1] 31.6979
```

```
## [1] -412.5668
```

In reviewing the distributions of the log likelihoods for each of the different classes: *spam* versus *non-spam*. Due to the distribution spread of the *spam* email messages being nearest zero (including the mean) we can state that the data shows our probability is more accurate at identifying the *spam* versus the *non-spam* messages. We also noted that below the quartile ranges of the *non-spam* emails is much smaller than the *spam* emails. This tells us that the repeatability of results for identifying *non-spam* messages is higher than the *spam* messages. Simply stated this indicates, the words unique words identified in *non-spam* messages are not as unique to *non-spam* messages as those words used in *spam* messages. Additionally, the words most used in *non-spam* messages can also appear in *spam* messages.

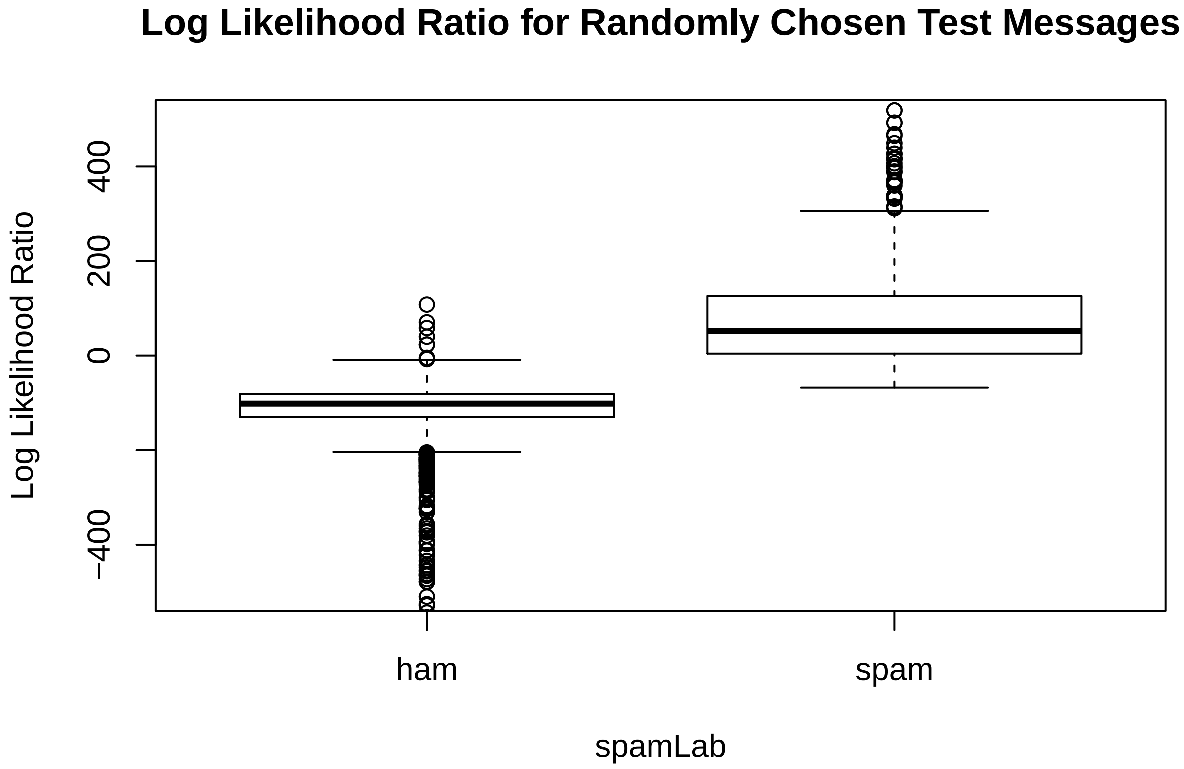
Analyzing from a logical perspective we can note that *spam* emails often contain words that might be difficult for detectors to pick up, such as repetitive letters to change the n_grams of the words and *leet speak*. *Leet speak* is the practice of replacing words with letters and such as an *E* becomes a *3* or an *L* becomes a *1*.

```
# do not run twice
testLLR = sapply(testMsgWords, computeMsgLLR, trainTable)
tapply(testLLR, testIsSpam, summary)
```

```
## $`FALSE`
##     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
## -1375.36  -130.35  -101.54  -118.87   -81.26   107.88
##
## $`TRUE`
##      Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
##   -67.680     4.138    51.738   136.758   126.221 23536.680
```

```
# boxplots of log likelihoods
#pdf("SP_Boxplot.pdf", width = 6, height = 6)
```

```
spamLab = c("ham", "spam")[1 + testIsSpam]
boxplot(testLLR ~ spamLab,
        ylab = "Log Likelihood Ratio",
       main = "Log Likelihood Ratio for Randomly Chosen Test Messages",
        ylim=c(-500, 500))
```



```
#dev.off()
```

Figure 2.1: Distribution: Messages by Classification

Below this code demonstrates misclassifications and calculates type I error rates with a function given the logged likelihood values and true data.

```
typeIErrorRate =
function(tau, llrVals, spam)
{
  classify = llrVals > tau
  sum(classify & !spam)/sum(!spam)
}
#tau 0
typeIErrorRate(0, testLLR,testIsSpam)
```

```
## [1] 0.002589555
```

```
#tau -20
typeIErrorRate(-20, testLLR,testIsSpam)
```

```
## [1] 0.005610703
```

```
typeIErrorRates =
function(llrVals, isSpam)
```

```r
{
  o = order(llrVals)
  llrVals =  llrVals[o]
  isSpam = isSpam[o]
  idx = which(!isSpam)
  N = length(idx)
  list(error = (N:1)/N, values = llrVals[idx])
}
```

Below this code demonstrates misclassifications and calculates type II error rates with a function given the logged likelihood values and true data.

```r
typeIIErrorRates = function(llrVals, isSpam) {

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


  idx = which(isSpam)
  N = length(idx)
  list(error = (1:(N))/N, values = llrVals[idx])
  }
xI = typeIErrorRates(testLLR, testIsSpam)
xII = typeIIErrorRates(testLLR, testIsSpam)
tau01 = round(min(xI$values[xI$error <= 0.01]))
t2 = max(xII$error[ xII$values < tau01 ])
```

2.2

Type I versus Type II Error Rates and Log-Likelihood Thresholds

Type I error rate is also known as a false positive error rate and occurs when the predictor incorrectly rejects a true null hypothesis. In our research the null hypothesis states that a message is *not-spam*. Below, in Figure XX, we have displayed the distribution of the Type I and Type II errors of our log likelihood values. Type I error rate is also known as a false positive error rate and occurs when the predictor incorrectly rejects a true null hypothesis. Type II error rate is also known as a false negative error rate and occurs when the predictor incorrectly accepts a false null hypothesis. As indicated by the orange marker, the threshold for determining *spam* is a log likelihood value of -41. Otherwise stated, any message that has a log likelihood value that is greater than -41 would be classified as a *spam* email and any message that has a log-likelihood of less than -41 is not a *spam* email.

```r
#pdf("LinePlotTypeI+IIErrors.pdf", width = 8, height = 6)

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",
     main = "Type I & II Error Rates: Spam vs Non-spam Log Likelihood Ratios")
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])
```

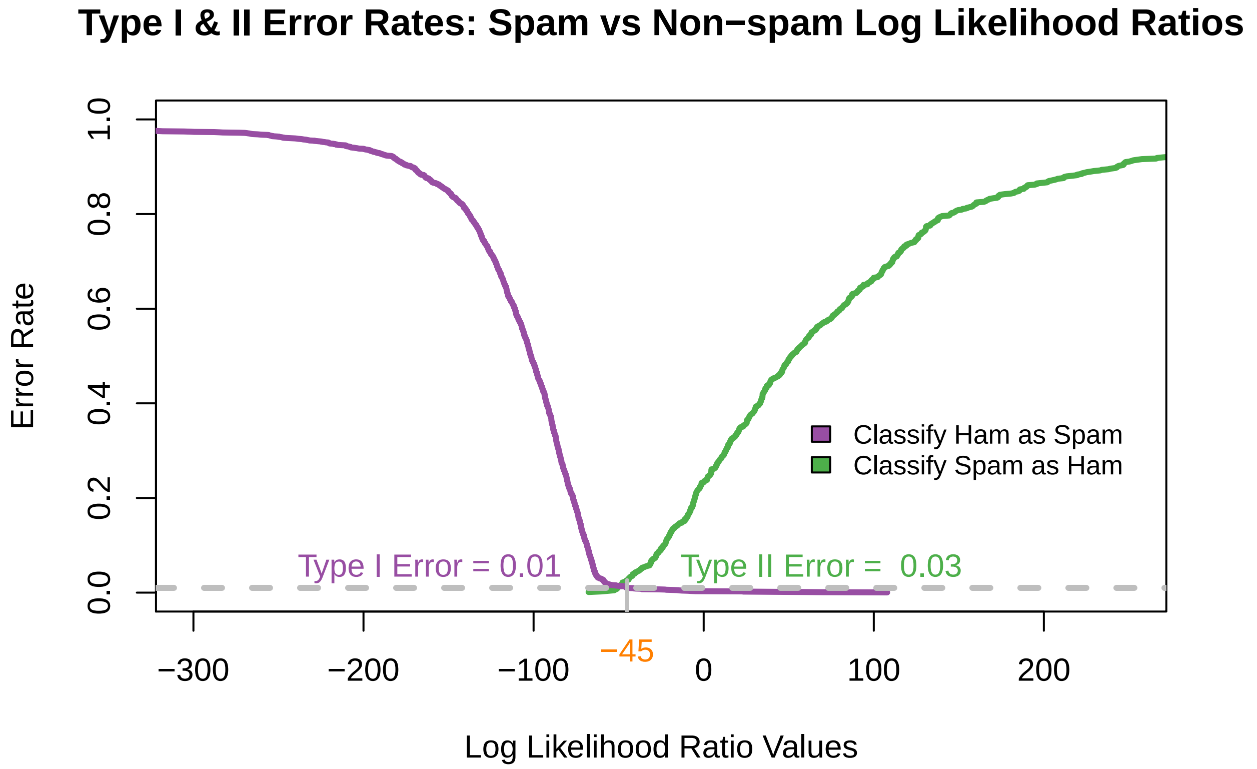


Figure 2.2: Type I and Type II Errors Rates

NOTE: In the above figure the Type I error threshold is 0.01 and the threshold for Type II error rate is 0.04.

```
#Below we are obtaining a tau so that one type of error is withint 1% using our 5-fold cross-validation
k = 5
numTrain = length(trainMsgWords)
partK = sample(numTrain)
tot = k * floor(numTrain/k)
partK = matrix(partK[1:tot], ncol = k)
testFoldOdds = NULL
for (i in 1:k) {
  foldIdx = partK[ , i]
  trainTabFold = computeFreqs(trainMsgWords[-foldIdx], trainIsSpam[-foldIdx])
  testFoldOdds = c(testFoldOdds,
                sapply(trainMsgWords[ foldIdx ], computeMsgLLR, trainTabFold))
}
testFoldSpam = NULL
for (i in 1:k) {
  foldIdx = partK[ , i]
  testFoldSpam = c(testFoldSpam, trainIsSpam[foldIdx])
}
xFoldI = typeIErrorRates(testFoldOdds, testFoldSpam)
xFoldII = typeIIErrorRates(testFoldOdds, testFoldSpam)
tauFoldI = round(min(xFoldI$values[xFoldI$error <= 0.01]))
tFold2 = xFoldII$error[ xFoldII$values < tauFoldI ]
```

2.3 Feature Creation

Borrowing off of the provided exploration of the messages we used code provided to build a thorough list of features and created dataset to run models with.

In the below chunk hidden from written output we build a list of functions used to create a derived dataset. This includes factors such as if the email uses excessive capitalization or if it is a reply to someone, a forward, includes underscores or a dear sir or madam address. All of these factors are fairly simple derivations of the pure messages and able to better describe some unique choices made by the email authors that may give hints as to whether they are *spam* or *non-spam* emails.

```
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")
}

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)
}

processAttach = function(body, contentType){

  boundary = getBoundary(contentType)

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

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

  n = length(body)
  if (length(eStringLoc) == 0) eStringLoc = n + 1
  if (length(bStringLocs) == 1) attachLocs = NULL
  else attachLocs = c(bStringLocs[-1],  eStringLoc)
  msg = body[ (bStringLocs[1] + 1) : min(n, (bStringLocs[2] - 1), na.rm = TRUE)]

  if ( eStringLoc < n )
    msg = c(msg, body[ (eStringLoc + 1) : n ])

  if ( !is.null(attachLocs) ) {
    attachLens = diff(attachLocs, lag = 1)
    attachTypes = mapply(function(begL, endL) {
      contentTypeLoc = grep("[Cc]ontent-[Tt]ype", body[ (begL + 1) : (endL - 1)])
      contentType = body[ begL + contentTypeLoc]
      contentType = gsub('"', "", contentType )
      MIMEType = sub(" *Content-Type: *([^;]*);?.*", "\\1", contentType)
      return(MIMEType)
    }, attachLocs[-length(attachLocs)], attachLocs[-1])
  }

  if (is.null(attachLocs)) return(list(body = msg, attachInfo = NULL) )
  else return(list(body = msg,
                   attachDF = data.frame(aLen = attachLens,
                                         aType = attachTypes,
                                         stringsAsFactors = FALSE)))
}

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)
    )
  }


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)

    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)

  emailList = mapply(function(header, body, isSpam) {
                       list(isSpam = isSpam, header = header,
                            body = body)
                     },
                     headerList, bodyList,
                     rep(isSpam, n), SIMPLIFY = FALSE )
  names(emailList) = fileNames

  invisible(emailList)
}
emailStruct = mapply(processAllEmail, fullDirNames, isSpam = rep( c(FALSE, TRUE), 3:2))
```

```
## Warning in FUN(X[[i]], ...): incomplete final line found on '/Users/
## zmartygirl/githubrepos1/7333_Quantifying_The_World/Unit6_CaseStudy3/data/
## hard_ham/00228.0eaef7857bbbf3ebf5edbbdae2b30493'

## Warning in FUN(X[[i]], ...): incomplete final line found on '/Users/
## zmartygirl/githubrepos1/7333_Quantifying_The_World/Unit6_CaseStudy3/data/
## hard_ham/0231.7c6cc716ce3f3bfad7130dd3c8d7b072'

## Warning in FUN(X[[i]], ...): incomplete final line found on '/Users/
## zmartygirl/githubrepos1/7333_Quantifying_The_World/Unit6_CaseStudy3/data/
## hard_ham/0250.7c6cc716ce3f3bfad7130dd3c8d7b072'
```

```
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)
}

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")


emailDF = createDerivedDF(emailStruct)
```

3. Models

We constructed four different models in order to determine the best classification technique for *spam* emails: *naive bayes*, *partition trees*, *random forest*, and *XGBoost*. Note, we chose to use those classification types that could handle nonlinear relationships between the data. Regularization is not required in *naive bayes* or *partition tree* analysis. They do however require parameters to hyper-tune. We have split the data into a test and train groups. In addition, we have performed a 5-fold cross-validation on the training data. Then leveraging an F1 score as the selection criteria for the best hypertuned parameter.

Below we run with minimal output the creation of several helper functions that will enable us to smoothly run each model.

```
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)
emailDFnum[is.na(emailDFnum)]<-0
cv_folds <- createFolds(emailDFnum$isSpam, k=5, list=TRUE,
                        returnTrain = TRUE)
#Folds should be equal in length about 7500

#Sets up F1 measurement
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
   )
}
```

Naive Bayes Model

Naive Bayes classifiers are a family of simplistic classifiers that are able to predict by applying Bayes' theorem that contain naive independent assumptions between the features. In other words, all *naive bayes* classifiers assume the value of any feature is independent of the value of any other features, given the class variable. Even though these classifiers are some of simplest available, when combined with kernel density estimation they achieve high accuracy levels and easily interpretable models. Given their simplified assumptions and naive design a researcher might think they are not strong enough to accomplish their classification needs. However, this model has proven that it adapts to many complex real-worlds situations and work well. Since they have proven to outperform models in the past we are sure to include is among our tested classifying models.

However, since the *naive bayes* approach cannot easily classify nuanced characteristics that can be apparent in *spam* emails we perform a *decision tree classifier*, *random forest*, and an *XGBoost* analysis to compare results.

List 3.1: Naive Bayes Hyper Parameters

- Laplace: smoothing parameter
- Usekernel: is this is true we should use the *kernel density* estimate, if this is false we should use the *gaussian density* estimate.
- Adjust: if this is true we should allow bandwidth of *kernel density* estimate to be adjusted as needed, if false; do not allow as much flexibility with estimate adjustments.

Table 3.1: Hyperparameter Settings for Naive Bayes Model

Hyperparameter	Setting	F1
Laplace	0	0.9197
Usekernel	False	
Adjust	False	

```
set.seed(1263)
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=5,
                            savePredictions = 'final',
                            summaryFunction = f1)
model_nb<-caret::train(as.factor(isSpam) ~ .,
                       data=emailDFnum, trControl = train_control,
                       method='naive_bayes',tuneGrid = nb_grid)
```

```
model_nb
```

```
## Naive Bayes
##
## 9345 samples
##   29 predictor
##    2 classes: '0', '1'
##
## No pre-processing
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 7477, 7475, 7476, 7476, 7476
## Resampling results across tuning parameters:
##
##   laplace  usekernel  adjust  F1         prec       rec        Type_I_err
##   0.0      FALSE      FALSE   0.9198045  0.9144730  0.9254790  0.06463436
##   0.0      FALSE       TRUE   0.9198045  0.9144730  0.9254790  0.06463436
##   0.0       TRUE      FALSE         NaN        NaN        NaN         NaN
##   0.0       TRUE       TRUE   0.8934197  0.8083546  0.9987053  0.17645739
##   0.1      FALSE      FALSE   0.9198045  0.9144730  0.9254790  0.06463436
##   0.1      FALSE       TRUE   0.9198045  0.9144730  0.9254790  0.06463436
##   0.1       TRUE      FALSE         NaN        NaN        NaN         NaN
##   0.1       TRUE       TRUE   0.8934197  0.8083546  0.9987053  0.17645739
##   0.3      FALSE      FALSE   0.9198045  0.9144730  0.9254790  0.06463436
##   0.3      FALSE       TRUE   0.9198045  0.9144730  0.9254790  0.06463436
##   0.3       TRUE      FALSE         NaN        NaN        NaN         NaN
##   0.3       TRUE       TRUE   0.8934197  0.8083546  0.9987053  0.17645739
##   0.5      FALSE      FALSE   0.9198045  0.9144730  0.9254790  0.06463436
##   0.5      FALSE       TRUE   0.9198045  0.9144730  0.9254790  0.06463436
##   0.5       TRUE      FALSE         NaN        NaN        NaN         NaN
##   0.5       TRUE       TRUE   0.8934197  0.8083546  0.9987053  0.17645739
##   1.0      FALSE      FALSE   0.9198045  0.9144730  0.9254790  0.06463436
##   1.0      FALSE       TRUE   0.9198045  0.9144730  0.9254790  0.06463436
##   1.0       TRUE      FALSE         NaN        NaN        NaN         NaN
##   1.0       TRUE       TRUE   0.8934197  0.8083546  0.9987053  0.17645739
##   Type_II_err
##   0.055429916
##   0.055429916
##           NaN
##   0.000963082
##   0.055429916
##   0.055429916
##           NaN
##   0.000963082
##   0.055429916
##   0.055429916
##           NaN
##   0.000963082
##   0.055429916
##   0.055429916
##           NaN
##   0.000963082
##   0.055429916
##   0.055429916
##           NaN
##   0.000963082
##
## F1 was used to select the optimal model using the largest value.
## The final values used for the model were laplace = 0, usekernel =
##  FALSE and adjust = FALSE.
```

```
cmdf = model_nb$pred
nb_cm = confusionMatrix(cmdf$pred,cmdf$obs)
```

Decision Trees and RandomForest Models

In order to fully determine how thoroughly a *decision tree* model can approach classification of *spam* we used both a singular *decision tree* model using the *rpart* toolset. This with a combination of a more thorough *Random Forest* that encompasses a wider random set of potential trees with one selected *decision tree*. Accordingly we expect and do see better results from a *random forest* approach. Our inclusion of a *Random Forest* model was specifically chosen as an extension of the *Decision Tree* model discussion, while our full third model in use will be *XGBoost*.

Decision Trees

Decision Tree models are simple tree-based model. This builds a singular tree based on categorical choices that then categorize each item in the dataset into tighter categories with probabilities indicative of class choice. A singular decision tree should be carefully chosen and pruned in order to build a generalizable model. Decision trees as a rule can be easily overfit to our training data, as we could create a specific tree deeply complex that correctly classifies each element in a training dataset. Therefore our work in choosing hyperparameters is essential to building a decision tree that does not overfit our data. The list below, *List 3.2: Decision Tree Hyper Parameter* contains the hyperparameters we tune towards a best F1 score.

Through a cross validated testing run we found the best value of Cp was .0005 with a F1 score of **.9608**

List 3.2: Decision Tree Hyper Parameter

- Cp: decrease in the lack of fit for any new split

```
val<-seq(from = 0, to=0.01, by=0.0005)
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
##
## 9345 samples
##   29 predictor
##    2 classes: '0', '1'
##
## No pre-processing
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 7477, 7475, 7476, 7476, 7476
## Resampling results across tuning parameters:
```

```
##
##   cp      F1         prec       rec        Type_I_err  Type_II_err
##   0.0000  0.9597621  0.9571102  0.9624511  0.03210336  0.02793006
##   0.0005  0.9608749  0.9572104  0.9646090  0.03210308  0.02632516
##   0.0010  0.9598176  0.9557169  0.9640337  0.03327989  0.02675342
##   0.0015  0.9605369  0.9544804  0.9667659  0.03434918  0.02472117
##   0.0020  0.9579214  0.9515516  0.9644644  0.03659654  0.02643297
##   0.0025  0.9566745  0.9540491  0.9594298  0.03445722  0.03017754
##   0.0030  0.9542744  0.9564891  0.9520921  0.03221026  0.03563466
##   0.0035  0.9528509  0.9533576  0.9523802  0.03467215  0.03542030
##   0.0040  0.9525493  0.9519153  0.9532436  0.03584931  0.03477807
##   0.0045  0.9513396  0.9534994  0.9493615  0.03456595  0.03766577
##   0.0050  0.9509242  0.9526400  0.9493614  0.03520777  0.03766589
##   0.0055  0.9502523  0.9511785  0.9495053  0.03638516  0.03755882
##   0.0060  0.9491240  0.9484824  0.9499370  0.03852534  0.03723780
##   0.0065  0.9468695  0.9465845  0.9472025  0.03980853  0.03927131
##   0.0070  0.9460916  0.9480582  0.9441808  0.03852293  0.04152011
##   0.0075  0.9460471  0.9488360  0.9433175  0.03788088  0.04216216
##   0.0080  0.9460471  0.9488360  0.9433175  0.03788088  0.04216216
##   0.0085  0.9447210  0.9467525  0.9427424  0.03948642  0.04258997
##   0.0090  0.9442283  0.9473546  0.9411597  0.03895137  0.04376707
##   0.0095  0.9442283  0.9473546  0.9411597  0.03895137  0.04376707
##   0.0100  0.9441448  0.9474786  0.9408719  0.03884436  0.04398109
##
## F1 was used to select the optimal model using the largest value.
## The final value used for the model was cp = 5e-04.
```

```r
cmdf = model_rpart$pred
dt_cm = confusionMatrix(cmdf$pred,cmdf$obs)
```

Random Forest

Random Forest models are an aggregated bootstrap tree-based model. This model type builds ensemble models with simple learning decisions that leave wanting for complexity. Each tree within the decision tree is produced by using the values of an independent set of random vectors that stem from a fixed probability distribution. These models are often described as easy to over-fit however are also highly reliable. *Random Forest* models are easily and highly interpretable models that other more advanced algorithms are near impossible to interpret in real terms. *List 3.3: Random Forest Hyper Parameter* contains the hyperparameter that we will tune for an best F1 score with our *random forest* model.

After cross validated testing we found the Mtry value was 9 with a peak F1 value of **.984**

List 3.3: Random Forest Hyper Parameter

- Mtry: number of variables randomly sampled as candidates at each split

```r
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_rf
```

```
## Random Forest
##
## 9345 samples
##   29 predictor
##    2 classes: '0', '1'
##
## No pre-processing
## Resampling: Cross-Validated (3 fold)
## Summary of sample sizes: 6230, 6230, 6230
## Resampling results across tuning parameters:
##
##   mtry  F1         prec       rec        Type_I_err  Type_II_err
##    1    0.8840246  0.7933786  0.9981298  0.19347245  0.001391118
##    3    0.9747886  0.9655971  0.9841749  0.02611022  0.011771001
##    5    0.9816787  0.9769346  0.9864768  0.01733547  0.010058855
##    7    0.9831518  0.9801397  0.9861890  0.01487426  0.010272873
##    9    0.9832251  0.9801435  0.9863329  0.01487426  0.010165864
##   11    0.9822295  0.9787342  0.9857574  0.01594436  0.010593900
##   13    0.9816635  0.9779031  0.9854697  0.01658641  0.010807919
##   15    0.9819479  0.9781774  0.9857574  0.01637239  0.010593900
##   17    0.9823802  0.9784695  0.9863329  0.01615837  0.010165864
##   19    0.9816012  0.9775011  0.9857574  0.01690744  0.010593900
##   21    0.9820277  0.9779156  0.9861890  0.01658641  0.010272873
##   23    0.9812992  0.9778879  0.9847504  0.01658641  0.011342964
##   25    0.9815269  0.9773507  0.9857574  0.01701445  0.010593900
##
## F1 was used to select the optimal model using the largest value.
## The final value used for the model was mtry = 9.
```

```
cmdf = model_rf$pred
rf_cm = confusionMatrix(cmdf$pred,cmdf$obs)
```

XGBoost

We chose to leverage *XGBoost* (*eXtreme Gradient Boosting*) as our final model for comparison. *XGBoost* is a sophisticated algorithm that shines when dealing with any irregularities in data. But, while powerful, engineers need to know what hyperparameters to tune in order to get the best model output. The list in *List 3.4: XGBoost Hyper Parameters* outlines those hyperparameters we chose, what they control, and why we chose them. Those being outlined, there are some general reasons why *XGBoost* is an effective algorithm to leverage. For instance, there is standard regularization in the *XGBoost* package. This helps to reduce overfitting and often *XGBoost* is used as a 'regularized boosting' method. Additionally, *XGBoost* offers parallel processing which allows it to be more economical when it comes to processing power in comparison to the other models. With so many hyper parameters to choose from the *XGBoost* package is highly flexible and allows for more advanced customization for those engineers who have deep knowledge of the mathematics behind the algorithm. Lastly, things like tree pruning with *max_depth* in hyperparameters, automatic cross-validation, and the ability to handle missing values *XGBoost* stands alone as one of the more powerful algorithms to leverage for our research purposes.

Through a cross validated testing run we found the best value of hyperparameters are described below with a F1 score of **.9819**

List 3.4: XGBoost Hyper Parameters

- nrounds: Number of boosted trees being fitted to the training data.

- max_depth: controlling the max depth makes this algorithm more conservative. The values can vary depending on the loss function and should always be tuned. This is mainly used to control overfitting as the higher the depth will allow for more models to learn relations for specific samples.
- eta: this is similar to the rate in GBM. The parameter makes the model more robust by decreasing the weights on the individual steps.
- gamma: this identifies the minimum loss reduction necessary to make a split. A node is split solely when the resulting split produces a positive reduction in the loss function. The values can differ all depending on the loss function.

Table 3.2: Hyperparameter Settings for XGBoost Model

Hyperparameter	Setting	F1
Nrounds	100	.981
MaxDepth	11	
eta	.1	
gamma	1	

```r
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
##
## 9345 samples
##   29 predictor
##    2 classes: '0', '1'
##
## No pre-processing
## Resampling: Cross-Validated (3 fold)
## Summary of sample sizes: 6230, 6230, 6230
## Resampling results across tuning parameters:
##
##   eta   max_depth  gamma  F1         prec       rec        Type_I_err
##   0.01   3          1     0.9311327  0.9057415  0.9579917  0.07415730
##   0.01   3          3     0.9310862  0.9053971  0.9582794  0.07447833
##   0.01   3          5     0.9312456  0.9053136  0.9587110  0.07458534
##   0.01   3         10     0.9316739  0.9056115  0.9592864  0.07437132
##   0.01   5          1     0.9490470  0.9364215  0.9620199  0.04858213
##   0.01   5          3     0.9496398  0.9378518  0.9617321  0.04740503
##   0.01   5          5     0.9493317  0.9367023  0.9623076  0.04836811
##   0.01   5         10     0.9481230  0.9355780  0.9610128  0.04922418
##   0.01   7          1     0.9568252  0.9523923  0.9613005  0.03574104
##   0.01   7          3     0.9566625  0.9526392  0.9607251  0.03552702
##   0.01   7        5   0.9561637  0.9536630  0.9587110  0.03467095
##   0.01   7       10   0.9549158  0.9530213  0.9568407  0.03509898
##   0.01   9        1   0.9650979  0.9557229  0.9746799  0.03360086
##   0.01   9        3   0.9641051  0.9564365  0.9719465  0.03295880
##   0.01   9        5   0.9638660  0.9568132  0.9710833  0.03263777
##   0.01   9       10   0.9605403  0.9579928  0.9631708  0.03146067
##   0.01  11        1   0.9674633  0.9575890  0.9775572  0.03220974
##   0.01  11        3   0.9670312  0.9572971  0.9769817  0.03242376
##   0.01  11        5   0.9661814  0.9582915  0.9742483  0.03156768
##   0.01  11       10   0.9623904  0.9585087  0.9663358  0.03113965
##   0.03   3        1   0.9420901  0.9208744  0.9643217  0.06163724
##   0.03   3        3   0.9422393  0.9209019  0.9646094  0.06163724
##   0.03   3        5   0.9425850  0.9212929  0.9648971  0.06131621
##   0.03   3       10   0.9413306  0.9199469  0.9637462  0.06238630
##   0.03   5        1   0.9588552  0.9490259  0.9689253  0.03873729
##   0.03   5        3   0.9588679  0.9487680  0.9692131  0.03895131
##   0.03   5        5   0.9587192  0.9487518  0.9689253  0.03895131
##   0.03   5       10   0.9569614  0.9465769  0.9676306  0.04066346
##   0.03   7        1   0.9660055  0.9590634  0.9730974  0.03092563
##   0.03   7        3   0.9654034  0.9592991  0.9716588  0.03071161
##   0.03   7        5   0.9642573  0.9581605  0.9705078  0.03156768
##   0.03   7       10   0.9620238  0.9547006  0.9695008  0.03424291
##   0.03   9        1   0.9722306  0.9647446  0.9798590  0.02664526
##   0.03   9        3   0.9713456  0.9648272  0.9779888  0.02653826
##   0.03   9        5   0.9695595  0.9631333  0.9761185  0.02782236
##   0.03   9       10   0.9660962  0.9606492  0.9716588  0.02964152
##   0.03  11        1   0.9746697  0.9667501  0.9827363  0.02514714
##   0.03  11        3   0.9728712  0.9655989  0.9802906  0.02600321
##   0.03  11        5   0.9710621  0.9646926  0.9775572  0.02664526
##   0.03  11       10   0.9677470  0.9620704  0.9735290  0.02857143
##   0.10   3        1   0.9618637  0.9481841  0.9759747  0.03970037
##   0.10   3        3   0.9618330  0.9489459  0.9751115  0.03905832
##   0.10   3        5   0.9608523  0.9459418  0.9762624  0.04151953
##   0.10   3       10   0.9591619  0.9441659  0.9746799  0.04291065
##   0.10   5        1   0.9709520  0.9634840  0.9785642  0.02760835
##   0.10   5        3   0.9706589  0.9633354  0.9781326  0.02771536
##   0.10   5        5   0.9685156  0.9613367  0.9758308  0.02921348
##   0.10   5       10   0.9637744  0.9536877  0.9741044  0.03520599
##   0.10   7        1   0.9766102  0.9712791  0.9820170  0.02161584
##   0.10   7        3   0.9737657  0.9677883  0.9798590  0.02429106
##   0.10   7        5   0.9701021  0.9649064  0.9753992  0.02643125
##   0.10   7       10   0.9663342  0.9602597  0.9725219  0.02996255
##   0.10   9        1   0.9806223  0.9751354  0.9861890  0.01872659
##   0.10   9        3   0.9757012  0.9694955  0.9820170  0.02300696
##   0.10   9        5   0.9732136  0.9665588  0.9800029  0.02525415
##   0.10   9       10   0.9689963  0.9625865  0.9755431  0.02825040
##   0.10  11        1   0.9808193  0.9760867  0.9856136  0.01797753
##   0.10  11        3   0.9775418  0.9721479  0.9830240  0.02097378
##   0.10  11        5   0.9732904  0.9665664  0.9801467  0.02525415
##   0.10  11       10   0.9701301  0.9639789  0.9764063  0.02718031
##   Type_II_err
##   0.03124666
##   0.03103264
##   0.03071161
##   0.03028357
##   0.02825040
##   0.02846442
##   0.02803638
##   0.02899946
##   0.02878545
##   0.02921348
##   0.03071161
##   0.03210273
##   0.01883360
##   0.02086677
##   0.02150883
##   0.02739433
##   0.01669342
##   0.01712146
##   0.01915463
##   0.02504013
##   0.02653826
##   0.02632424
##   0.02611022
##   0.02696629
##   0.02311396
##   0.02289995
##   0.02311396
##   0.02407705
##   0.02001070
##   0.02108079
##   0.02193686
##   0.02268593
##   0.01498127
##   0.01637239
##   0.01776351
##   0.02108079
##   0.01284109
##   0.01466025
##   0.01669342
##   0.01968967
##   0.01787052
##   0.01851257
##   0.01765650
##   0.01883360
##   0.01594436
##   0.01626538
##   0.01797753
##   0.01926164
##   0.01337614
##   0.01498127
##   0.01829856
##   0.02043874
##   0.01027287
##   0.01337614
##   0.01487426
##   0.01819155
##   0.01070091
##   0.01262707
##   0.01476726
##   0.01754949
##
## Tuning parameter 'nrounds' was held constant at a value of 100
##  1
## 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.
```

```
cmdf = model_xgb$pred
xg_cm = confusionMatrix(cmdf$pred,cmdf$obs)
```

Results

The results for our different models by F1 score are below. The data depicts that the Random Forest model is the top performing model over all. While the F1 score is our major score of interest

Table 4.1: Model Results by F1 Score

Hyperparameter	F1 Score
Naive Bayes	0.9197
RandomForest	0.984
DecisionTree	0.9608
XGBoost	0.9819

Naive Bayes

Overall as we expected our Naive Bayes model and approach is the worst model of the four we demonstrated. The model had a best accuracy of about .92 with an average accuracy of .88. We can see with the table the overall confusion matrix is however balanced without a major discrepancy of misclassified *spam* due to the unbalanced nature of the data we explored earlier.

```
print(nb_cm)
```

```
## Confusion Matrix and Statistics
##
##            Reference
## Prediction    0    1
##          0 6433  604
##          1  518 1790
##
##                Accuracy : 0.8799
##                  95% CI : (0.8732, 0.8865)
##     No Information Rate : 0.7438
##     P-Value [Acc > NIR] : < 2e-16
##
```

```
##                   Kappa : 0.6812
##
##  Mcnemar's Test P-Value : 0.01116
##
##             Sensitivity : 0.9255
##             Specificity : 0.7477
##          Pos Pred Value : 0.9142
##          Neg Pred Value : 0.7756
##              Prevalence : 0.7438
##          Detection Rate : 0.6884
##    Detection Prevalence : 0.7530
##       Balanced Accuracy : 0.8366
##
##        'Positive' Class : 0
##
```

Decision Tree Classifier

The decision tree on its own performed better overall than the Naive Bayes classifier. With an overall accuracy of about .94 across all folds and a peak accuracy of .96 this model worked well but did have some problems with balance. While the Naive Bayes classifier produced errors in a balanced way, we can see with the table below that the Decision Tree had a higher misclassification of *spam* emails as *non-spam*.

```r
print(dt_cm)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction    0    1
##          0 6705  300
##          1  246 2094
##
##                Accuracy : 0.9416
##                  95% CI : (0.9366, 0.9462)
##     No Information Rate : 0.7438
##     P-Value [Acc > NIR] : < 2e-16
##
##                   Kappa : 0.8455
##
##  Mcnemar's Test P-Value : 0.02332
##
##             Sensitivity : 0.9646
##             Specificity : 0.8747
##          Pos Pred Value : 0.9572
##          Neg Pred Value : 0.8949
##              Prevalence : 0.7438
##          Detection Rate : 0.7175
##    Detection Prevalence : 0.7496
##       Balanced Accuracy : 0.9196
##
##        'Positive' Class : 0
##
```

Random Forest

An overall best model the Random Forest produced average accuracies of .97 with a peak accuracy of .984. We know that a Random Forest model regularly produces good classification results, and it is not surprising as the generalized version and corrected version of the decision tree model it performs a good deal better. We can also see misclassifications are fairly balanced among *spam* and *non-spam* emails.

```
print(rf_cm)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction    0    1
##          0 6856  139
##          1   95 2255
##
##                Accuracy : 0.975
##                  95% CI : (0.9716, 0.978)
##     No Information Rate : 0.7438
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.9339
##
##  Mcnemar's Test P-Value : 0.004939
##
##             Sensitivity : 0.9863
##             Specificity : 0.9419
##          Pos Pred Value : 0.9801
##          Neg Pred Value : 0.9596
##              Prevalence : 0.7438
##          Detection Rate : 0.7337
##    Detection Prevalence : 0.7485
##       Balanced Accuracy : 0.9641
##
##        'Positive' Class : 0
##
```

XGBoost

Finally XGBoost was the last algorithm we ran. It also produced average classification rates of .97 with peak accuracy of .981. As a model it only performed slightly worse than the Random Forest and has many similar characteristics in the model and predictions in the confusion matrix we can see below.

```
print(xg_cm)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction    0    1
##          0 6851  168
##          1  100 2226
##
##                Accuracy : 0.9713
##                  95% CI : (0.9677, 0.9746)
##     No Information Rate : 0.7438
##     P-Value [Acc > NIR] : < 2.2e-16
##
```

```
##                   Kappa : 0.924
##
##  Mcnemar's Test P-Value : 4.264e-05
##
##             Sensitivity : 0.9856
##             Specificity : 0.9298
##          Pos Pred Value : 0.9761
##          Neg Pred Value : 0.9570
##              Prevalence : 0.7438
##          Detection Rate : 0.7331
##    Detection Prevalence : 0.7511
##       Balanced Accuracy : 0.9577
##
##        'Positive' Class : 0
##
```

Conclusion

Natural language processes are the most versatile when it comes to how text is pre-processed. We had a single method here, but as email tactics and content evolves with character usages, it stands to argue that these models would execute classifications with just as much accuracy but the entirety of our text cleaning methods would likely need to be updated based on recent practices of email spammers.

Additionally, expect the results of these various models to be dated by the origin time of the emails themselves dating back to 2006 and earlier, the techniques may persist as useful ones with more updated lexicons of training emails and spam.

Overall our best models would have been a Random Forest as a better version of the decision tree approach and our XGBoost model. The combination of a Random Forest offering slightly better accuracies with faster run times make this model a slightly more attractive candidate compared to the XGBoost.

References

Nolan, D. and Lang, D.T. (2015). Data Science in R A Case Studies Approach to Computational Reasoning and Problem Solving. CRC Press.

Lim, S., & Chi, S. (2019). XGBoost application on bridge management systems for proactive damage estimation. Advanced Engineering Informatics, 41, 100922–. https://doi.org/10.1016/j.aei.2019.100922.