Email Spam Detector

Laura Lazarescou, John Rodgers, Maysam Mansor and Mel Schwan

February 15, 2021

1 Introduction

Spam email is unsolicited and unwanted junk email sent out in bulk to an indiscriminate recipient list. Spam is typically sent for commercial gain. The volume of these types of emails is massive. Many advertisers use this type of efficient communication method and these emails can be classified as spam. Frequent advertising messages irritate targeted email recipients which makes them seek ways to end the stream of advertisements coming to their email address. Automatic filters that detect unwanted email advertisements, called spam filters, use machine learning (ML) and natural language processing (NLP).

We will demonstrate how to build a spam filter using an email spam dataset.

2 Methods

2.1 Data Collection and Preparation

2.1.1 Data Collection

We examined over 9000 messages that have been classified by SpamAssassin (http://spamassassin.apache.org) to develop and test spam filters. This dataset is provided as a public corpus by the Apache Foundation for spam classification model development.
Spam, ham, and hard ham are the three categories that SpamAssassin uses to classify the emails. The emails listed in the hard ham category are legitimate but have characteristics more similar to the spam emails. Experimenting with spam detection using this corpus, which contains emails from 2002-2006, may be useful in developing techniques. However, this dataset may no longer represent today's sophisticated spam and non-spam emails.

2.1.2 Data Preparation

The body of the message will be our main focus for spam analysis. We developed an extraction function for grouping the message body words. We also eliminated the email attachments as we are not interested in this portion of the body. Once we have located the relevant portion of the message body, the words are extracted using a parsing routine. The three main functions that we used sequentially were, splitMessage(), dropAttach(), and findMsgWords().

2.1.3 Feature Engineering

A total of 29 features are extracted from the SpamAssassin email corpus using the included code. We used these extracted features to evaluate three popular ML classifiers, resulting in a better comparison with the results of a similar study. While it might be natural to consider the word frequencies to detect spam, many words of a typical corpus result in a highly dimensional feature matrix. The highly dimensional matrices usually lead to poor modeling results.

Below are five example engineered features (the full list is provided in Appendix A).

- The percentage of capital letters found in the email
- The number of lines in the email
- Whether the letters in the email subject line are all capitalized
- Whether the email is a reply (RE) or not
- Whether the sender name contains an underscore ('_') or not

The **figures 1 - 3** are plots that help determine which engineered features will be of value.

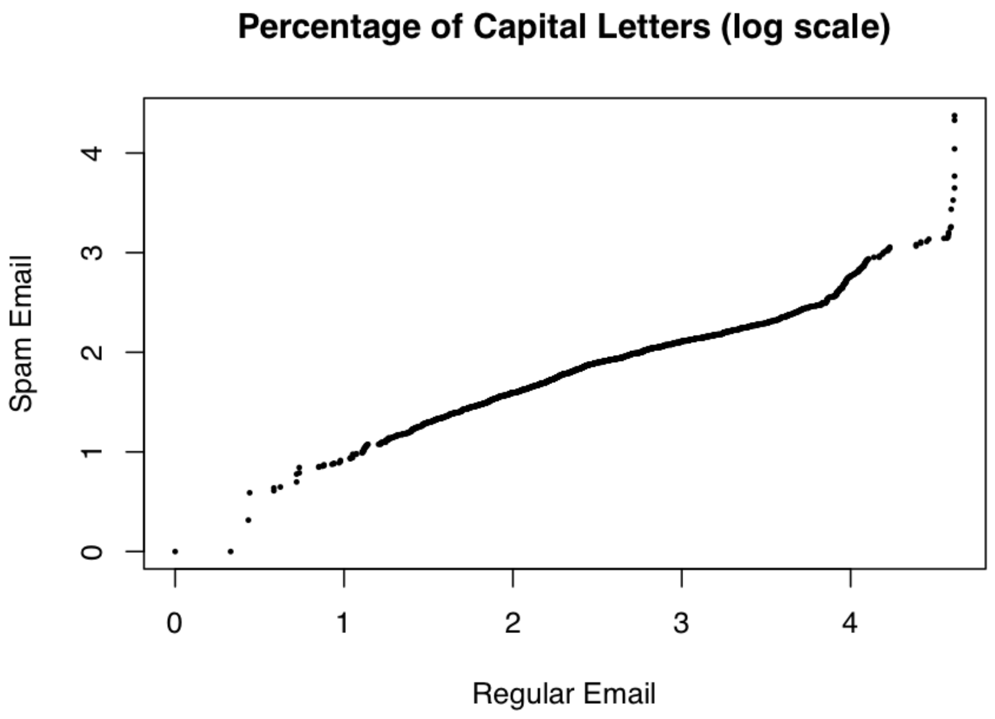

Figure 1: Regular or Spam Using Capital Letters Feature

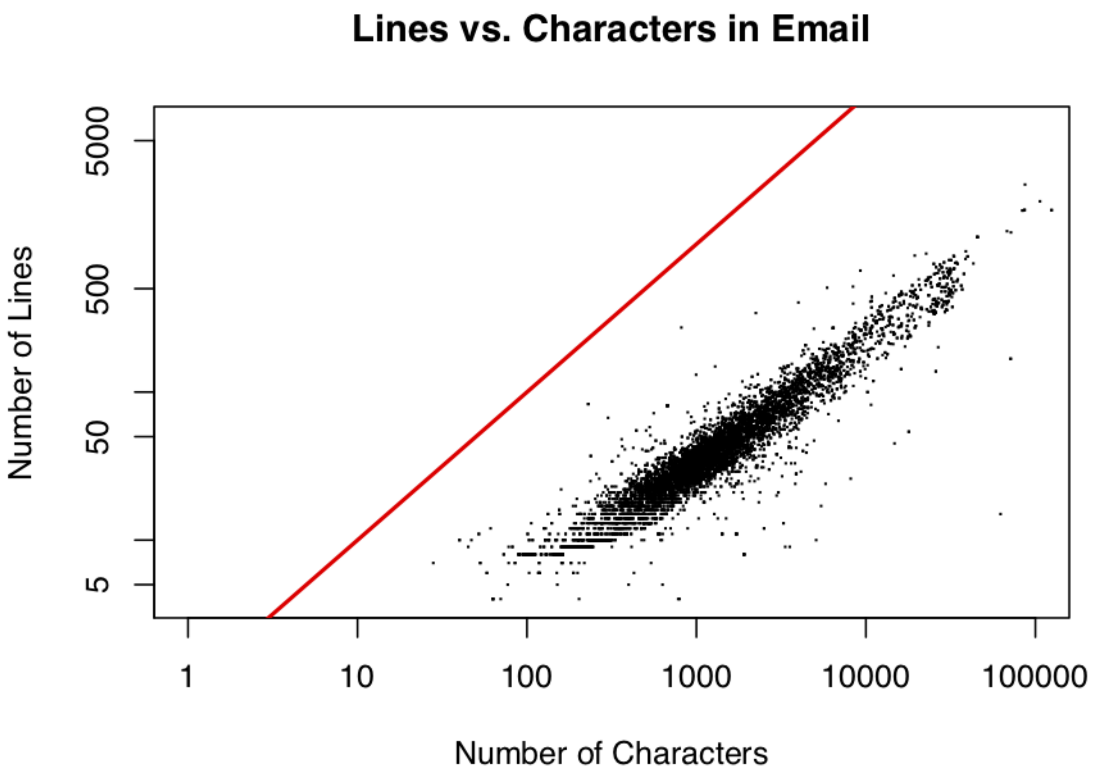

Figure 2: Number of Lines Regular or Spam

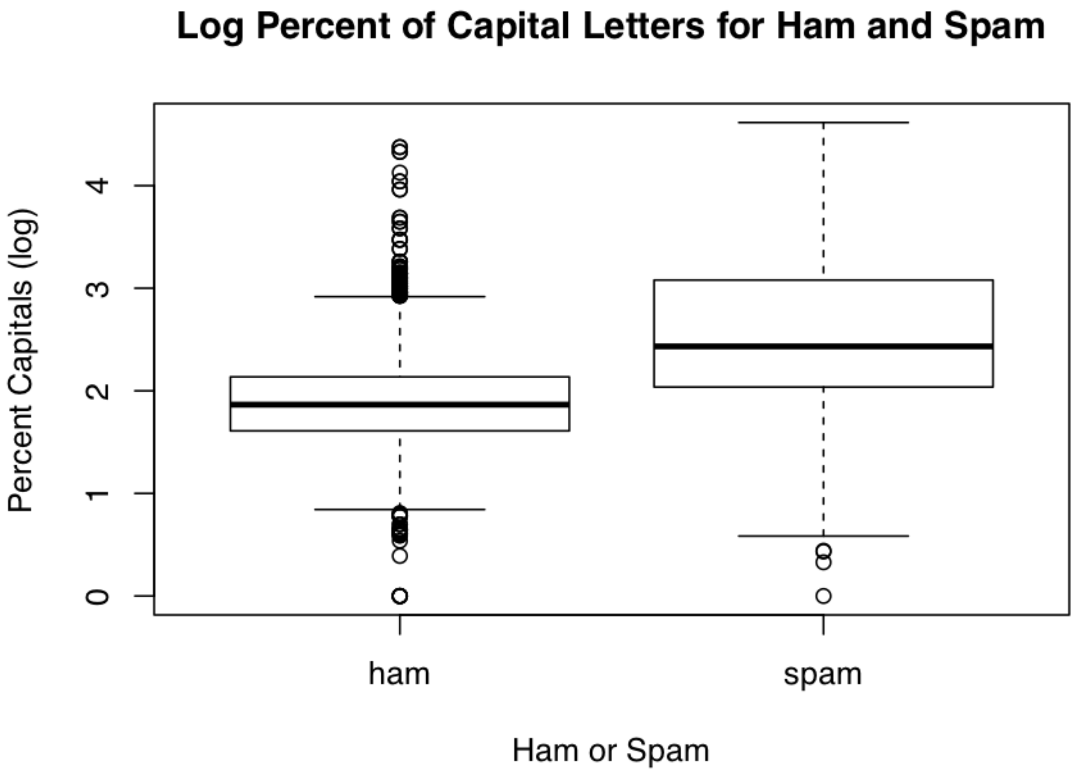

Figure 3: Percentage of Capital Letters for Ham and Spam

2.2 Models

In this case study, we ran the sample code models, which included Naive Bayes, RPart, and XGBoost. Additionally, optimized LogitBoost, SVM, and the GBM Models for comparisons. We will run each of the additional models experimenting with the hyperparameter settings to optimize the F1 measurement.

2.2.1 Naive Bayes Model

In the Naive Bayes model, several parameters were varied: laplace, usekernal True or False, and Adjust True or False. With the given number of example models, the bestTune model has the following values:

laplace = 0, usekernel = FALSE, adjust = FALSE

Best F1 score for Naive-Bayes is .925

2.2.2 RPART Model

This model will not be tuned beyond the example code. cp is varied from 0 to .01 by .0005 The bestTune model has cp=.001 which results in the following F1 score:

Best F1 score for RPart is .959

Below, **figure 4** shows the cross-validation versus complexity for the Rpart model.

This shows the decrease in the F1 score after the 0.001 values for the complexity parameter.

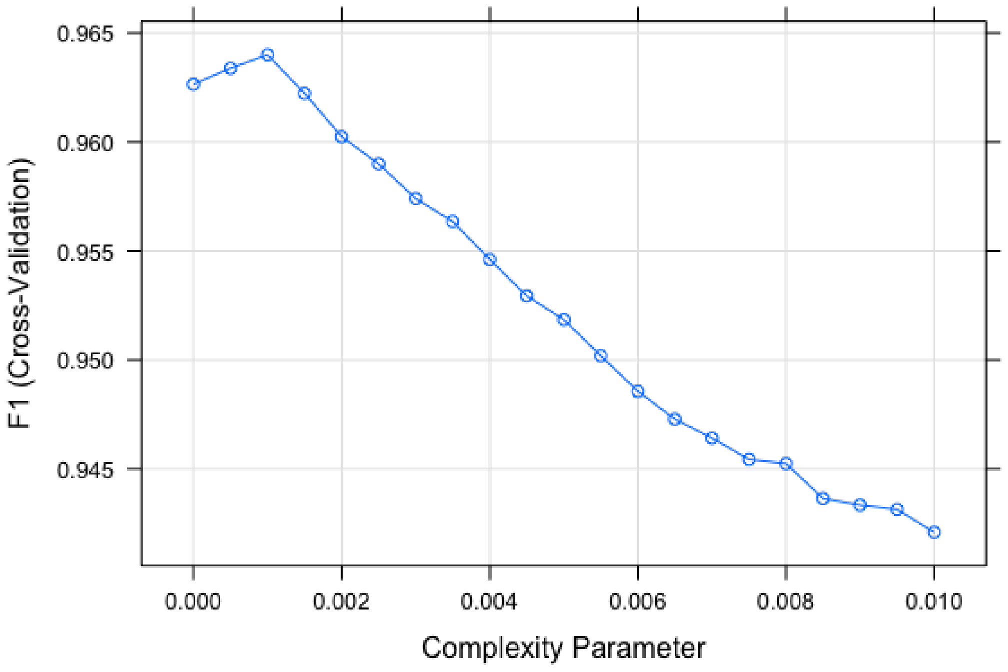

Figure 4: RPart Model Cross-Validation

2.2.3 XGBoost Model

This model was furnished in our example code, so we will not tune it further. The tuneable parameters are the number of rounds, max_depth, eta, and gamma. With the given parameters, we find that the **bestTune parameters** are:

max_depth=11, eta=.1, colsample_bytree=1, min_child_weight=1 subsample = 1

XGBoost bestTune F1 score is .983 which is very high.

This model may warrant further attention if other models do not outperform it.

Below in **figure 5** we show the cross-validation versus tree depth that result from tuning the hyper-perameters.

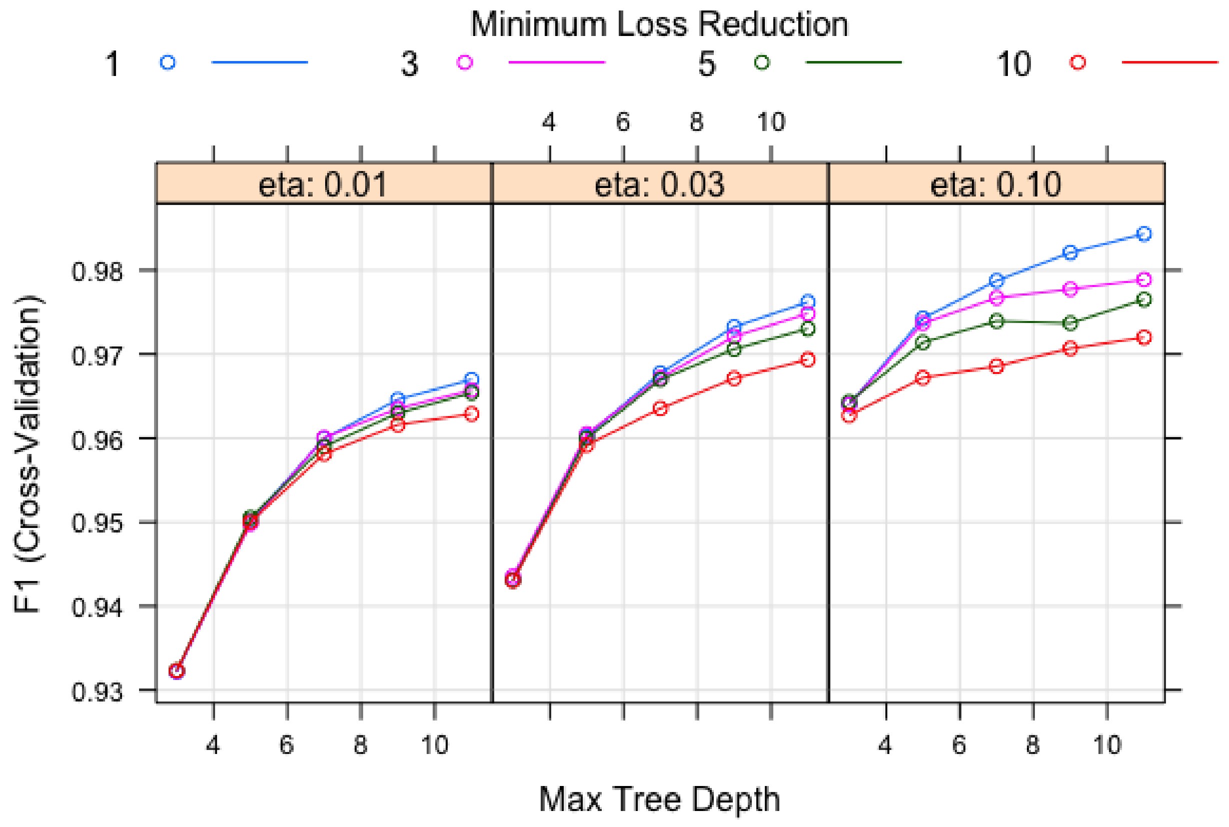

Figure 5: XGBoost Model Cross-Validation

2.2.4 LogitBoost Model - First Additional Model

LogitBoost is a classification algorithm that uses stumps or one-node decision trees. The only tuning parameter is nIter, which represents the number of iterations or number of decision stumps.

The example below shows that increasing nIter from 1 to 10 yields increasingly high F1 scores.

If we increase nIter to 50 and 100, F1 is only slightly greater than that of nIter=10. With this in mind, the optimal value of nIter will be a question of cost vs. accuracy.

nIter = 10, F1 = 0.9595785 nIter = 50, F1 = 0.968588 nIter = 100, F1 = 0.9712751

2.2.5 Support Vector Machines (SVM Model) - Second Additional Model

Support Vector Machines are learning methods used for classification, regression, and outlier detection. SVM tries to put the data into different dimensions until it can be distinguished and classified into other groups.

The advantages of support vector machines are:

- Effective in high dimensional spaces.
- Still effective in cases where the number of dimensions is greater than the number of samples.
- Uses a subset of training points in the decision function (called support vectors), so it is also memory efficient.
- Versatile: different Kernel functions can be specified for the decision function. Common kernels are provided, but it is also possible to specify custom kernels. The disadvantages of support vector machines include:
- If the number of features is much greater than the number of samples, avoid over-fitting in choosing Kernel functions and regularization term is crucial.
- SVMs do not directly provide probability estimates; these are calculated using an expensive five-fold cross-validation.
  Here we used this model two times.
  First we tried with loss= 1 which is ‘regularized L2-loss support vector classification’ and then we tried Loss=0 which is ‘regularized logistic regression’.
  We found out ‘regularized L2-loss support vector classification’ would generate higher **F1 score(F1=0.903761)**.

2.2.6 GBM Model - Third Additional Model

In looking at Gradient Boosted Machines (GBM) as an alternative modeling method, a few combinations of modeling hyperparameters will be explored. All models will utilize 5000 trees, but will have variations in interaction-depth and shrinkage . For the first model, an interaction depth of 1 is used with a shrinkage value of .001. This model returns an RMSE value(figure 6) of 0.4313 and an estimation of 5000 trees required to process the model.

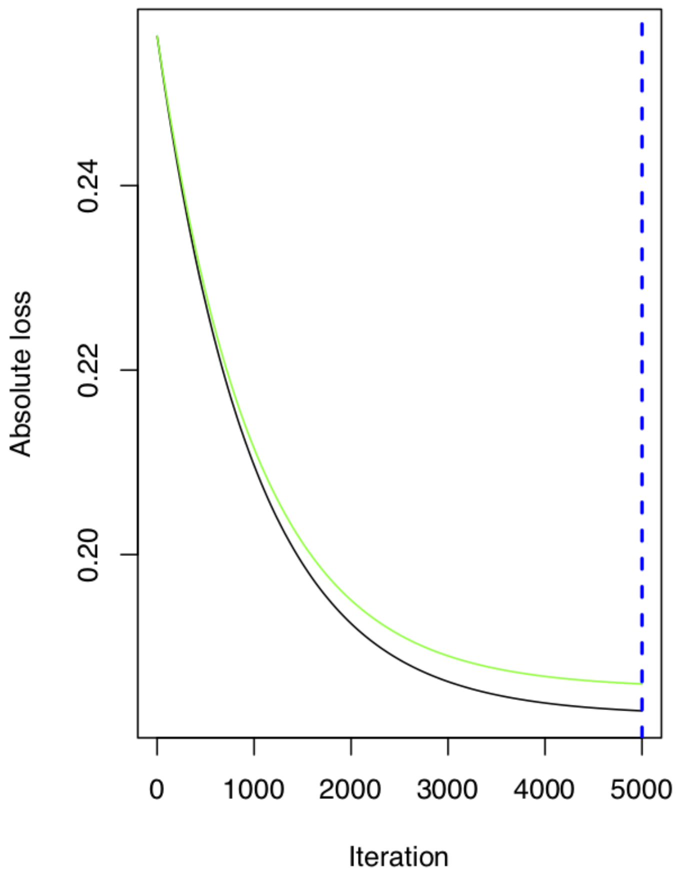

Figure 6: Gradient Boosted Machines Interaction Depth 1

For the second variation, an interaction depth of 1 remains and the shrinkage value was changed to .01. This model returns a small increase in the RMSE value (figure 7) of 0.4314 and a decrease in the estimation of trees required to process the model with 1587.

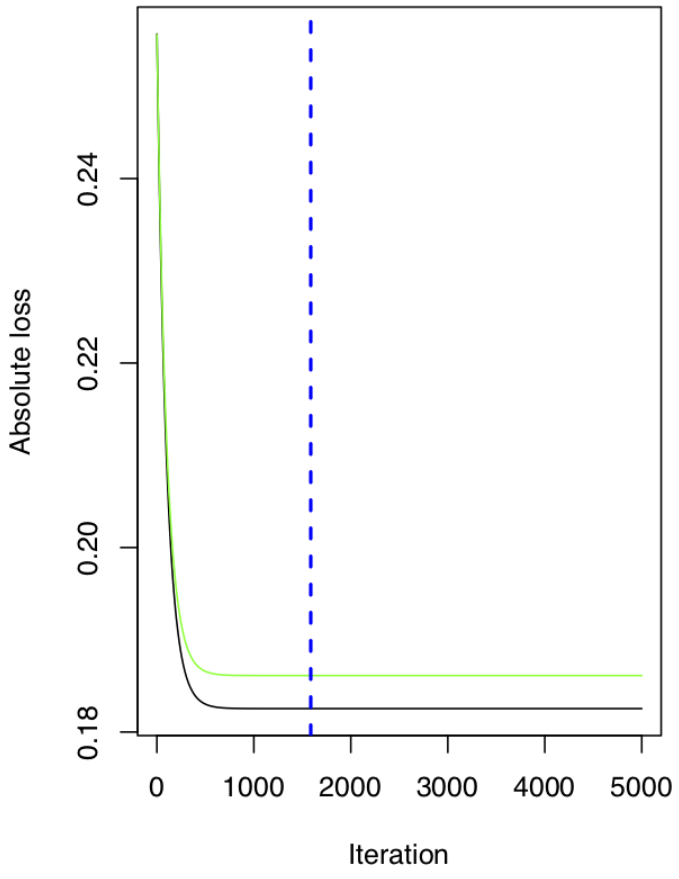

Figure 7: Gradient Boosted Machines Interaction Shrinkage .01

GBM Model - Third Additional Model (figure 8)

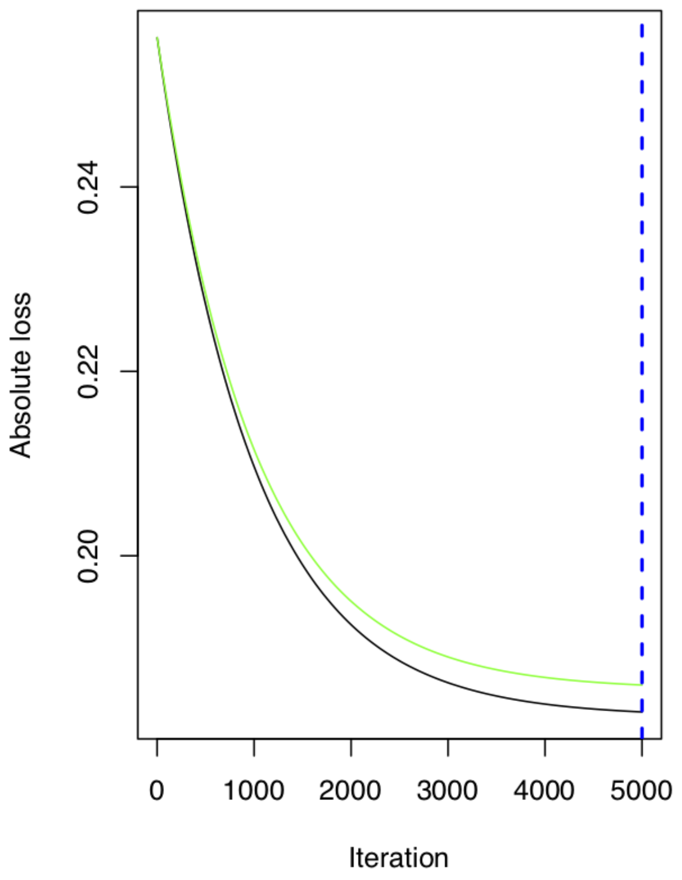

Figure 8: Gradient Boosted Machines Interaction Depth 1, Shrinkage .01

In reviewing the results of these models, a grid of 81 unique models made up of multiple settings for the number of trees (1000, 2000, 5000), the tree interaction depth (1,3,5), shrinkage (.01,.03,.1), and minimum observations in node (5,10,15). From the results of these 81 unique combinations, the following model has been identified as the best model based on minimizing both Type I error and Type II error:

	n.trees	interaction.depth	shrinkage	n.minobsinnode
75	5000	5	0.1	5

	F1	Type_I_err	Type_II_err
75	**0.98719**	0.0133747	0.005778

2.3 Results

2.3.1 Model Results

Our results are based off of the F1 score for each model. The F1-score is a way of combining the precision and recall of the model, and it is defined as the mean of the model's precision and recall. Using this metrics, it shows that the tuned GLB model gave us the best **F1 results of 0.987** .

Model	**F1**
SVM	**0.903**
Naive Bayes	**0.925**
RPart	**0.959**
LogitBoost	**0.971**
XGBoost	**0.983**
GBM	**0.987**

3 Conclusion

We have gained several conclusions from running the six models against the SpamAssassin corpus. If we use the non-biased F1 score, the Gradient Boosted Machines model had the highest score. The fundamental human issue with spam filtering is what kind of email user you are. If you are the type that worries about missing an important email that may end up in your spam folder, instead of using F1 we might choose a model with higher precision to eliminate a false negative. However, if you are an email user who hates spam, we would prefer a model with higher sensitivity, leaning towards classifying more emails as spam.

This is a topic that must continue to evolve. Our sample dataset is made up of email messages from 2002 to 2006. If we were to rerun the case study with modern email messages, we would find a corpus that has a more current spam terms. The features may change with a more current corpus. We might also create a dynamic machine learning model that is consistently updating with a real-time feedback loop. A real-time learning model would allow the spam filter to morph in parallel with the spam generators

A Appendix

A.1 Feature Engineering

The following 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 exclamation points in the email subject line	Numeric
6	The number of question marks in the email subject line	Numeric
7	The number of attachments 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	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 the letters in the email subject line are all capitalized	Logical
24	The number of forward 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

Number	Feature Description	Type
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”

A.3 All code for this report

```r
knitr::opts_chunk$set(echo = FALSE)
knitr::opts_chunk$set(message = FALSE)
knitr::opts_chunk$set(warning = FALSE)

library(dplyr)
library(MLmetrics)
library(tm)
library(caret)
library(naivebayes)
library(e1071)
library(rpart)
library(gbm)
library(LiblineaR)
library(rpart.plot)
library(RColorBrewer)
library(xgboost)
library(caTools)
library(randomForest)
library(ggplot2)

# Note - Change depending on environment
#spamPath = "C:/Code/DS7333/CS6"
spamPath = "/Users/melschwan/Dropbox/#Masters/QTW/Case_6/spam"
#setwd("C:/SMU_Local/SMU_T5_QTW_CapA/QTW_7333/Unit 5 and 6")
# spamPath = "C:/SMU_Local/SMU_T5_QTW_CapA/QTW_7333/Unit 5 and 6"
# Code for parsing emails

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

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

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){

  n = length(body)
  boundary = getBoundary(contentType)

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

  if (length(eStringLoc) == 0) eStringLoc = n
  if (length(bStringLocs) <= 1) {
    attachLocs = NULL
    msgLastLine = n
    if (length(bStringLocs) == 0) bStringLocs = 0
  } else {
    attachLocs = c(bStringLocs[ -1 ],  eStringLoc)
    msgLastLine = bStringLocs[2] - 1
  }

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

  if ( !is.null(attachLocs) ) {
    attachLens = diff(attachLocs, lag = 1)
    attachTypes = mapply(function(begL, endL) {
      CTloc = grep("^[Cc]ontent-[Tt]ype", body[ (begL + 1) : (endL - 1)])
      if ( length(CTloc) == 0 ) {
        MIMEType = NA
      } else {
        CTval = body[ begL + CTloc[1] ]
        CTval = gsub('"', "", CTval )
        MIMEType = sub(" *[Cc]ontent-[Tt]ype: *([^;]*);?.*", "\\1", CTval)
      }
      return(MIMEType)
    }, attachLocs[-length(attachLocs)], attachLocs[-1])
  }

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

readEmail = function(dirName) {
       # retrieve the names of files in directory
  fileNames = list.files(dirName, full.names = TRUE)
       # drop files that are not email
  notEmail = grep("cmds$", fileNames)
  if ( length(notEmail) > 0) fileNames = fileNames[ - notEmail ]

       # read all files in the directory
  lapply(fileNames, readLines, encoding = "latin1")
}

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)
}

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, fullDirNames,
                     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)
  }

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)
    )
  }

emailDF = createDerivedDF(emailStruct)

setupRnum = function(data) {
  logicalVars = which(sapply(data, is.logical))
```

```r
  facVars = lapply(data[ , logicalVars],
                   function(x) {
                      x = as.numeric(x)
                   })
  cbind(facVars, data[ , - logicalVars])
}

emailDFnum = setupRnum(emailDF)

emailDFnum[is.na(emailDFnum)]<-0

# save(emailDFnum,file="emailDFnum.Rda")  # Used by team to ensure same dataset

# Load Email Dataframe

load("emailDFnum.Rda")
load("emailDF.Rda")
# EDA Visualization of Original Email Messages

perCaps2 =
function(msg)
{
  body = paste(msg$body, collapse = "")

       # Return NA if the body of the message is "empty"
  if(length(body) == 0 || nchar(body) == 0) return(NA)

       # Eliminate non-alpha characters and empty lines
  body = gsub("[^[:alpha:]]", "", body)
  els = unlist(strsplit(body, ""))
  ctCap = sum(els %in% LETTERS)
  100 * ctCap / length(els)
}


pC = sapply(emailStruct, perCaps)
pC2 = sapply(emailStruct, perCaps2)
identical(pC, pC2)

indNA = which(is.na(emailDF$subExcCt))

indNoSubject = which(sapply(emailStruct,
                            function(msg)
```

```
                              !("Subject" %in% names(msg$header))))

all(indNA == indNoSubject)

all(emailDF$bodyCharCt > emailDF$numLines)


x.at = c(1,10,100,1000,10000,100000)
y.at = c(1, 5, 10, 50, 100, 500, 5000)
nL = 1 + emailDF$numLines
nC = 1 + emailDF$bodyCharCt


pdf("ScatterPlotNumLinesNumChars.pdf", width = 6, height = 4.5)
plot(nL ~ nC, log = "xy", pch=".", xlim=c(1,100000), axes = FALSE,
     main = "Lines vs. Characters in Email",
     xlab = "Number of Characters", ylab = "Number of Lines")
box()
axis(1, at = x.at, labels = formatC(x.at, digits = 0, format="d"))
axis(2, at = y.at, labels = formatC(y.at, digits = 0, format="d"))
abline(a=0, b=1, col="red", lwd = 2)
dev.off()

pdf("SPAM_boxplotsPercentCaps.pdf", width = 6, height = 4.5)

percent = emailDF$perCaps
isSpamLabs = factor(emailDF$isSpam, labels = c("ham", "spam"))
boxplot(log(1 + percent) ~ isSpamLabs,
        main = "Log Percent of Capital Letters for Ham and Spam",
        ylab = "Percent Capitals (log)", xlab = "Ham or Spam")

dev.off()

logPerCapsSpam = log(1 + emailDF$perCaps[ emailDF$isSpam ])
logPerCapsHam = log(1 + emailDF$perCaps[ !emailDF$isSpam ])

pdf("Regular_or_spam_capital_letters.pdf", width = 6, height = 4.5)
qqplot(logPerCapsSpam, logPerCapsHam,
       xlab = "Regular Email", ylab = "Spam Email",
       main = "Percentage of Capital Letters (log scale)",
       pch = 19, cex = 0.3)
dev.off()
```

```r
#library(rpart.plot)

prp(rpartFit, extra = 1)

pdf("SPAM_rpartTree.pdf", width = 7, height = 7)

prp(rpartFit, extra = 1)
dev.off()

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

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

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

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

complexityVals = c(seq(0.00001, 0.0001, length=19),
                   seq(0.0001, 0.001, length=19),
                   seq(0.001, 0.005, length=9),
                   seq(0.005, 0.01, length=9))


pdf("SPAM_rpartTypeIandII.pdf", width = 8, height = 7)
# library(RColorBrewer)
cols = brewer.pal(9, "Set1")[c(3, 4, 5)]
plot(errs[1,] ~ complexityVals, type="l", col=cols[2],
     lwd = 2, ylim = c(0,0.2), xlim = c(0,0.01),
     ylab="Error", xlab="complexity parameter values")
points(errs[2,] ~ complexityVals, type="l", col=cols[1], lwd = 2)

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

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

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

dev.off()

# Model Development

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

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

<!-- In the Naive Bayes model, several parameters were varied:  laplace, usekernal True or False, and Ad
<!-- With the given number of example models, the bestTune model has the following values: -->
<!-- laplace = 0, usekernel = FALSE, adjust = FALSE -->

<!-- Best F1 score for Naive-Bayes is .925   --># library(naivebayes)
# library(e1071)
nb_grid<-expand.grid(laplace=c(0,0.1,0.3,0.5,1), usekernel=c(T,F), adjust=c(T,F))
train_control<-trainControl(method="cv", number=5, savePredictions = 'final',summaryFunction = f1)
model_nb<-caret::train(as.factor(isSpam) ~ .,data=emailDFnum, trControl = train_control, method='naive_
model_nb$bestTune
nb_best_F1=model_nb$results$F1[1]
nb_best_F1
plot(model_nb)
<!-- RPART Model -->

<!-- This model will not be tuned beyond the example code. -->
<!-- cp is varied from 0 to .01 by .0005 -->
```

```
<!-- The bestTune model has cp=.001 -->

<!-- Best F1 score for RPart is .959 -->

val<-seq(from = 0, to=0.01, by=0.0005)
# library(rpart)
cart_grid<-expand.grid(cp=val)
train_control<-trainControl(method="cv", number =5, savePredictions = 'final',summaryFunction = f1)
model_rpart<-caret::train(as.factor(isSpam) ~ .,data=emailDFnum, trControl = train_control, method='rpa
model_rpart
plot(model_rpart)
<!-- XGBoost Model -->

<!-- This model was furnished in our example code so we will not tune it further. -->
<!-- The tunable parameters are number of rounds, max_depth, eta and gamma. -->
<!-- With the given parameters, we find that the bestTune parameters are  -->
<!-- max_depth=11, eta=.1, colsample_bytree=1, min_child_weight=1 subsample = 1 -->

<!-- XGBoost bestTune F1 score is .983 which is very high.  This model may warrant further attention if
# 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(1,3,5,1
train_control<-trainControl(method="cv", number=5, savePredictions = 'final',summaryFunction = f1)
model_xgb<-caret::train(as.factor(isSpam) ~ .,data=emailDFnum, trControl = train_control,method='xgbTre
model_xgb$bestTune
best_xbg_F1 = model_xgb$results$F1[57]
best_xbg_F1
plot(model_xgb)

<!-- Random Forest Model -->

<!-- This model was furnished with the examples so we will not tune it further. -->
<!-- The only tunable parameter is mTry, which was varied from 1 to 25 by steps of 2. -->

<!-- From bestTune we see the optimal value of mTry = 9 which is the 5th sample. -->
<!-- The best F1 score for our random forest model is .984 which is slightly higher than the XGBoost mod

# library(randomForest)
rf_grid<-expand.grid(mtry=seq(from =1, to = 25, by = 2))
train_control<-trainControl(method="cv", number=5, savePredictions = 'final',summaryFunction = f1)
model_rf<-caret::train(as.factor(isSpam) ~ .,data=emailDFnum, trControl = train_control, ntree=200,meth
model_rf
plot(model_rf)
# LogitBoost Model
```

```
<!-- LogitBoost Model - First Additional Model -->


<!-- LogitBoost is a classification algorithm that uses stumps or one-node decision trees.  The only tu

<!-- In the example below we see that increasing nIter from 1 to 10 yields increasingly high F1 scores.

<!-- If we increase nIter to 50 and 100, F1 is only slightly greater than that of nIter=10. -->
<!-- With this in mind, the optimal value of nIter will be a question of cost vs. accuracy. -->

<!-- nIter = 10, F1 = 0.9595785 -->
<!-- nIter = 50, F1 = 0.968588 -->
<!-- nIter = 100, F1 = 0.9712751 -->

library(caTools)
lboost_grid<-expand.grid(nIter=seq(from =1, to = 10))
train_control<-trainControl(method="cv", number=5, savePredictions = 'final',summaryFunction = f1)
model_lboost<-caret::train(as.factor(isSpam) ~ .,data=emailDFnum, trControl = train_control,method='Log
model_lboost

model_lboost_10_F1 = model_lboost$results$F1[10]
model_lboost_10_F1

lboost_grid<-expand.grid(nIter=50)
train_control<-trainControl(method="cv", number=5, savePredictions = 'final',summaryFunction = f1)
model_lboost<-caret::train(as.factor(isSpam) ~ .,data=emailDFnum, trControl = train_control,method='Log
model_lboost_50_F1 = model_lboost$results$F1[1]
model_lboost_50_F1

lboost_grid<-expand.grid(nIter=100)
train_control<-trainControl(method="cv", number=5, savePredictions = 'final',summaryFunction = f1)
model_lboost<-caret::train(as.factor(isSpam) ~ .,data=emailDFnum, trControl = train_control,method='Log
model_lboost_100_F1 = model_lboost$results$F1[1]
model_lboost_100_F1


<!-- SVM Model - Second Additional Model -->

<!-- Support vector machine are learning methods used for classification, regression and outlier detect
```

```
<!-- The advantages of support vector machines are: -->

<!-- - Effective in high dimensional spaces. -->
<!-- - Still effective in cases where number of dimensions is greater than the number of samples. -->
<!-- - Uses a subset of training points in the decision function (called support vectors), so it is also
<!-- - Versatile: different Kernel functions can be specified for the decision function. Common kernels
<!--   The disadvantages of support vector machines include: -->
<!-- - If the number of features is much greater than the number of samples, avoid over-fitting in choos
<!-- - SVMs do not directly provide probability estimates, these are calculated using an expensive five-

<!-- Here we used this model two times. First we tried with loss= 1 which is 'regularized L2-loss suppo

#library(LiblineaR)
ll_grid<-expand.grid(cost=1, Loss=1, weight=1) #change these values and/or change to vectors
train_control<-trainControl(method="cv", number=5, savePredictions = 'final',summaryFunction = f1)
model_ll_1<-caret::train(as.factor(isSpam) ~ .,data=emailDFnum, trControl = train_control,method='svmLi
model_ll_1

ll_grid<-expand.grid(cost=1, Loss=0, weight=1) #change these values and/or change to vectors
train_control<-trainControl(method="cv", number=5, savePredictions = 'final',summaryFunction = f1)
model_ll_2<-caret::train(as.factor(isSpam) ~ .,data=emailDFnum, trControl = train_control,method='svmLi
model_ll_2

<!-- GBM Model - Third Additional Model -->

<!-- In looking at Gradient Boosted Machines (GBM) as an alternative modeling method, a few combination

gbm.fit <- gbm(
  formula = as.factor(isSpam) ~ .,
  distribution = "laplace",
  data = emailDFnum,
  n.trees = 5000,
  interaction.depth = 1,
  shrinkage = 0.001,
  cv.folds = 5,
  verbose = FALSE
  )

save(gbm.fit,file="gbm_fit.Rda")
# load gbm.fit from Rda
load("gbm_fit.Rda")

# get MSE and compute RMSE
```

```
gbm.fit.rmse = sqrt(min(gbm.fit$cv.error))

# plot loss function as a result of n trees added to the ensemble
gbm.perf(gbm.fit, method = "cv")


<!-- For the second variation, an interaction depth of 1 remains and the shrinkage value was changed to

gbm.fit2 <- gbm(
  formula = as.factor(isSpam) ~ .,
  distribution = "laplace",
  data = emailDFnum,
  n.trees = 5000,
  interaction.depth = 1,
  shrinkage = 0.01,
  cv.folds = 5,
  verbose = FALSE
  )

save(gbm.fit2,file="gbm_fit2.Rda")
# load gbm.fit2 from Rda
load("gbm_fit2.Rda")




# get MSE and compute RMSE
gbm.fit2.rmse = sqrt(min(gbm.fit2$cv.error))

# plot loss function as a result of n trees added to the ensemble
gbm.perf(gbm.fit2, method = "cv")


<!-- For the next model, an interaction depth is increased to 3 and the shrinkage value remains at .01.

gbm.fit3 <- gbm(
  formula = as.factor(isSpam) ~ .,
  distribution = "laplace",
  data = emailDFnum,
  n.trees = 5000,
  interaction.depth = 3,
  shrinkage = 0.01,
  cv.folds = 5,
  verbose = FALSE
  )
save(gbm.fit3,file="gbm_fit3.Rda")
# load gbm.fit3 from Rda
load("gbm_fit3.Rda")

# get MSE and compute RMSE
gbm.fit3.rmse = sqrt(min(gbm.fit3$cv.error))

# plot loss function as a result of n trees added to the ensemble
gbm.perf(gbm.fit3, method = "cv")

gbm_grid<-expand.grid(n.trees=c(1000,2000,5000), interaction.depth=c(1,3,5), shrinkage=c(0.01,0.03,0.1)
train_control<-trainControl(method="cv", number=5, savePredictions = 'final',summaryFunction = f1)
model_gbm<-caret::train(as.factor(isSpam) ~ .,data=emailDFnum, trControl = train_control,method='gbm',tu
save(model_gbm,file="model_gbm.Rda")
# load model_gbm from Rda
load("model_gbm.Rda")
#
# In reviewing the results of these models, a grid of 81 unique models made up of multiple settings for
#

# identify unique values from best results
gbm.n.trees = model_gbm$bestTune$n.trees
gbm.interaction.depth = model_gbm$bestTune$interaction.depth
gbm.shrinkage = model_gbm$bestTune$shrinkage
gbm.n.minobsinnode = model_gbm$bestTune$n.minobsinnode

#create dataframe of just the best result
gbm_results = model_gbm$results
gbm_best_result = gbm_results[which(gbm_results$n.trees==gbm.n.trees & gbm_results$interaction.depth==gb

#output the best tuned model configuration
knitr::kable(model_gbm$bestTune)
#output the F1, FP, and FN of model
knitr::kable(gbm_best_result[,c('F1','Type_I_err','Type_II_err')])
```