Unit 6 Case Study

MSDS Fall ‘19

7333 Quantify the World

Daniel Serna, Bruce Granger, and Brandon de la Houssaye

October 8, 2019

**Abstract**

In this paper we provide an in-depth analysis of a machine learning (ML) method that programmatically identifies the difference between emails that are unwanted (spam) from email that are wanted (ham). We will touch upon the history of unwanted emails, the financial impact of malicious spam emails, and who is at risk to spam.

**1 Introduction**

In today’s email reality, people subscribe to email newsletters, forums, alerts, etc. as a means of having information pushed to them automatically. What began as a helpful feature for users slowly turned into a marketing bonanza, as a new form of target marketing. The marketing firms then began to resell email addresses as they learned more about the users, thus creating value to retailers.

Not to be left out of the bonanza, criminals (cyber-criminals) use email as means of attacking new targets through what is known as “phishing” expeditions. The basic aim of a phishing attack is to convince the email receiver to click on an active link that is embedded within the email. This link then leads the victim to a fake website that closely resembles a legitimate company where the user is asked to provide username and password and/or other information. The phishing game is very easy for scammers (criminals) to replicate over and over and as a result is one of the easiest forms of cyber-attack for a cyber-criminal to carry out[[1]](#footnote-1).

The latest form of attack that often begins from email is called “ransomware”. The majority of ransomware is spread via large spam email campaigns involving hundreds of thousands of emails that are sent daily. In a ransomware attack, the receiver is again prompted to click on embedded URLs within the email. Once a user clicks the link, malicious code is downloaded to the user’s machine that encrypts the user’s hard drive. Once encrypted, the cyber-criminal will lock out the user and demand a ransom be paid, typically through bitcoin, before the cyber-criminal will unlock the user’s data. Often times malware will pose as ransomware and when victims pay the ransom to the cyber-criminal, the victims’ data is never “unlocked” by the cyber-criminal. This is known as “wiper” malware[[2]](#footnote-2). These attacks do not typically target an individual or organization, they are attacks of opportunity through economies of scale[[3]](#footnote-3). A report from Cybersecurity Ventures estimated that damages from ransomware attacks cost as much as $8 billion globally in 2018[[4]](#footnote-4).

One agent to combat this growing problem is to make users aware of the dangers of spam email and educate users what to look for and what not to do. Another response is to invest in technologies that stop spam email from ever entering the users inbox, such as implementing machine learning to route spam to the spam folder and the good email to the inbox, which is the point of this exercise. Given what is at stake, the machine learning algorithm should error on the side of caution and false positives, which means we want to focus on precision as a measurement for model effectiveness. Worst case for a false positive, the user can check the emails in their spam folder for an email they are expecting.

**2 Data**

The email messages that will be at the core of this analysis is made available via SpamAssassin (<http://spamassassin.apache.org>). For repeatability and convenience these email examples have been bundled into the RSpamData package in the R programming language. The emails have been organized into five (5) subdirectories, three of which identify the good emails (ham) and two identify the unwanted emails (spam). The names of the spam folders are:

* spam
* spam\_2

The names of the ham folders are:

* easy\_ham
* easy\_ham\_2
* hard\_ham

There are a total of nine-thousand three-hundred and fifty-three (9,353) emails and the distribution of spam to ham is as follows:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Folder Name | Email Type | Email Count | Type Total | Type % of Total |
| spam | Spam | 1,001 | 2,399 | 25% |
| spam\_2 | Spam | 1,398 |
| easy\_ham | Ham | 5,052 | 6,954 | 75% |
| easy\_ham\_2 | Ham | 1,401 |
| hard\_ham | Ham | 501 |

An email message has two parts, the email header and email body, which mirrors the concept of a standard letter that is sent through the United States Postage system. The email header is similar to the envelope that encapsulates the letter and contains metadata about where the email is going, additional recipients that will also receive the email, and routing information. This information is contained in a key: value pair. The body of the email message is analogous to the letter being sent within the postal envelope. The body of the email message is separated from the email header by a single empty line. When an email attachment is added to the email message, the attachment is included in the body of the email message[[5]](#footnote-5). Within the metadata, the Content-Type key communicates if the message has an attachment or not. If the message does not have an attachment, the value of “TEXT/PLAIN” is provided. If the message has an attachment, the value of “multipart” is provided. In the case where there is an attachment, there is a boundary string. After the initial “boundary” is provided, a subsequent boundary string is preceded by two hyphens, except the final boundary where it is preceded and followed by two hyphens.

**3 Data Cleansing**

The team performed parsing of the data to transform it into a dataframe consumable by train/test split and cross-validation methods. Using these parsing methods, the team was able to extract targeted aspects of email messages such as the header, body, recipient lists, priority and date/time. Additional metrics such as “percent capitalized”, “number of recipients”, “percent html” and “is pgp signed” were also retrieved. The team also had to deal with NAs created as part of the parsing process.

**4 Methods**

The team initially performed a standard train/test split method utilizing a recursive portioning technique for analysis. Tweaks to the rpart parameters were used to attempt model improvement. This code can be found in Appendix I.

The team ran additional analysis on the spam data by utilizing 5-fold cross-validation techniques in an effort to improve our model. Both the naïve-Bayes and recursive partitioning methods were used for this cross-validation analysis. This code can be found in Appendix II.

The team did attempt to utilize a cross-validation technique on the original msgWordList set, however significant road blocks were encountered, and the method was abandoned for another approach. Code for this effort is found in Appendix III for context.

**5 Results**

The team’s responses to the case study’s questions are as follows:

Question 19) Consider the other parameters that can be used to control the recursive partitioning process. Read the documentation for them in the *rpart.control()* documentation.

**Documentation for rpart.control() was provided in:**

<https://www.rdocumentation.org/packages/rpart/versions/4.1-15/topics/rpart.control>

**Also, carry out an Internet search for more information on how to tweak the *rpart()* tuning parameters. Experiment with values for these parameters.**

<https://www.gormanalysis.com/blog/decision-trees-in-r-using-rpart/>

<https://csantill.github.io/RTuningModelParameters/>

<https://www.mayo.edu/research/documents/rpartminipdf/doc-10027257>

**Do the trees that result make sense with your understanding of how the parameters are used?**

Yes, but only to a certain extent. The complexity parameter (cp) seemed to be the most relevant parameter to adjust in setting the output.

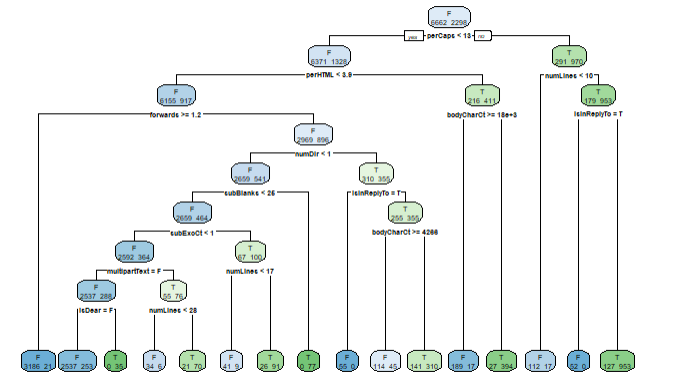
**Can you improve the prediction using them?**

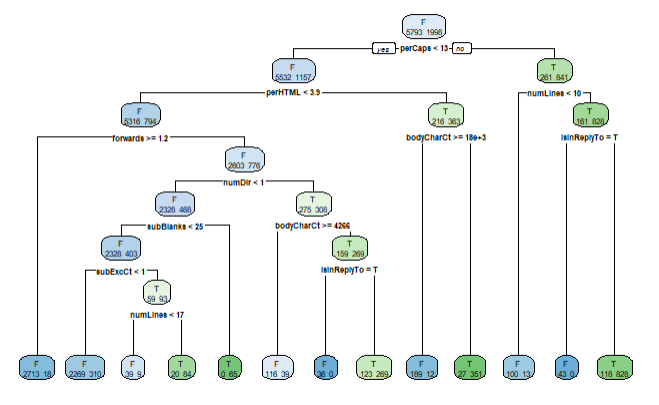
Not necessarily (see below). This was the interesting piece. We changed parameters to account for the sample size, the seed, the maximum depth (i.e., number of possible layers in the tree before splitting stops), cross-validations, minimum size of a bucket (observations in any final leaf), minimum split (the minimum number of splits attempted), a complexity parameter (cp). This parameter (cp) can be a set parameter and is also a model output (as can be seen in the original example). By setting a given cp, we dictate the splitting to continue (or stop) if the relative fit is decreased. That is, the model will stop if performing additional splits does not increase the fit.

We have observed that the cp is clearly the most relevant parameter. We could not set the other parameters to improve upon the fit (as measured by cp). In setting a cp to equal the original model (relatively), we did not find any additional ‘leafs’. Or, put another way, we cannot get to more insight/depth of the data if we are aiming to maintain the fit as measured by cp. i.e., the model would not be able improve with any additional splits.

The first set of parameters tuned were the seed; setting na.action to na.rpart; and decreasing the floor (of the sample set to \*/8). We then increased the sample size, set the min\_split to 10, set the max depth to 30, and x-validations to 80. We then set a minimum number of observations to 2 and then set the complexity parameter to 0.012.

The tree results are as follows:





The code for this analysis is provided in Appendix I.

Regarding question 20, the team’s naïve-Bayes cross-validation analysis allowed us to conclude that utilizing a laplace value of 0 along with usekernel and adjust parameters set to FALSE produced the optimal model. This was determined by the F1 score as see in Figure 1. With this particular model, the team achieved a type I error rate of 0.0463 and type II error rate of 0.0634

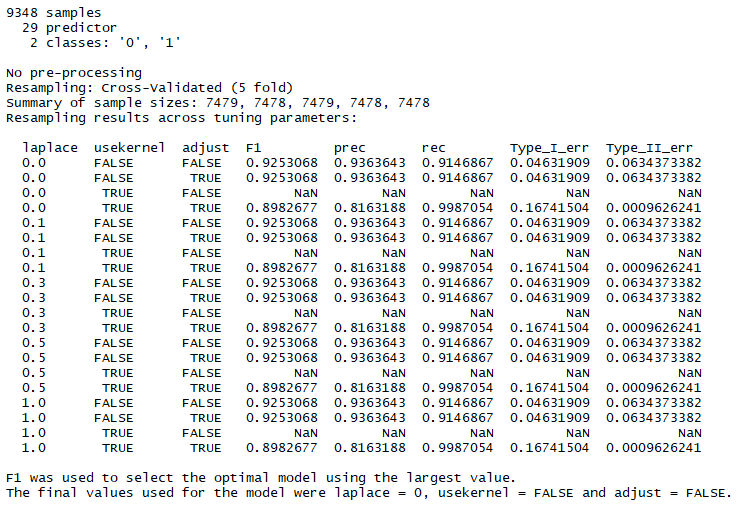


Figure 1 - Naive Bayes CV

The team’s recursive partition cross-validation analysis resulted in an optimal model when a complexity parameter of 0.001 was used. The optimal model was chosen by F1 score as seen in Figure 2. With this model, the team achieved a type I error rate of 0.03 and a type II error rate of 0.026.

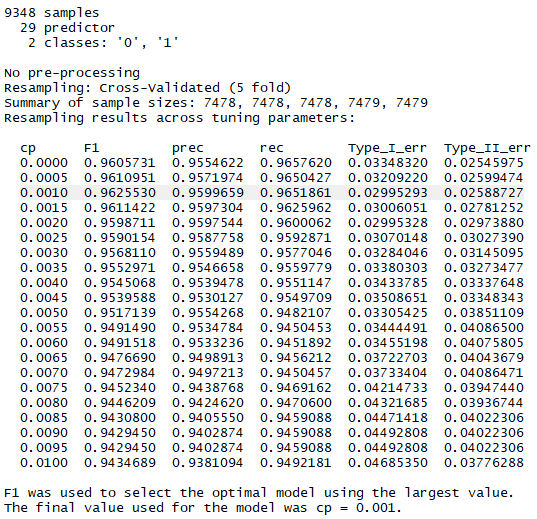


Figure 2 – Recursive Partition CV

The code for this cross-validation analysis is found in Appendix II.

**6 Conclusion**

The team was able to gain much insight from the recursive partition method using the standard train/test split approach. Attempts were made to improve the model by optimizing parameters to the rpart method.

The team had difficulty applying the results from cross-validation methods to the original msgWordList data set. We desired to determine a tau threshold value that we could apply to our original analysis to improve prediction. Ultimately, we chose to go a different route and identify optimal parameters for both naïve-Bayes and recursive partition models. Future work is warranted to expand on this analysis.

**Appendix I Code**

```{r rpartIteration1}

#first iteration: set the seed to 1,000,000; set na.action to equal na.rpart;

#;decrease the floor to /8 (decreases your sample size by four so we get a smaller tree to see)

#

set.seed(10000000)

testSpamIdx = sample(numSpam, size = floor(numSpam/8))

testHamIdx = sample(numHam, size = floor(numHam/8))

testDF =

rbind( emailDFrp[ emailDFrp$isSpam == "T", ][testSpamIdx, ],

emailDFrp[emailDFrp$isSpam == "F", ][testHamIdx, ] )

trainDF =

rbind( emailDFrp[emailDFrp$isSpam == "T", ][-testSpamIdx, ],

emailDFrp[emailDFrp$isSpam == "F", ][-testHamIdx, ])

testcontrolparam <- rpart(isSpam ~., data=trainDF, na.action = na.rpart, method = "class")

#plot the results

rpart.plot(testcontrolparam, extra = 1)

#dev.off()

predictions = predict(testcontrolparam,

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)

```

```{r rpartIteration2}

#not much of a difference in results. Which is good, but we are going to

#increase the sample size in order to fine-tune some control parameters

#set min split to equal 10 (so we stop trying to split at 10)

#set max depth to 30. original model = 9, so lets see what happens when we expand this (especially for the false read-outs)

# set cross-validations to 20. Because I have the processing speed and I want to see...

#the number of surrogate splits was also set to 30..again, lets see processing time

# set min bucket to 2; keep going until the observations get small

#also, set the complexity parameter. this is to determine if a split is to be attempted

#based on whether a split decreases lack of fit. meant to cut down on not-needed splits and save time.

#complexity parameter is the difference maker in the output. this is the parameter that determines

# if there are almost no splits (using the base cp provided in the code) or the current output.

set.seed(10000000)

testSpamIdx = sample(numSpam, size = floor(numSpam/2))

testHamIdx = sample(numHam, size = floor(numHam/2))

testDF =

rbind( emailDFrp[ emailDFrp$isSpam == "T", ][testSpamIdx, ],

emailDFrp[emailDFrp$isSpam == "F", ][testHamIdx, ] )

trainDF =

rbind( emailDFrp[emailDFrp$isSpam == "T", ][-testSpamIdx, ],

emailDFrp[emailDFrp$isSpam == "F", ][-testHamIdx, ])

testcontrolparam1 <- rpart(isSpam ~., data=trainDF, na.action = na.rpart, method = "class",control=rpart.control(minsplit = 10, maxdepth=30, xval = 80,maxsurrogate = 40,minbucket = 2, cp = 0.012))

#plot the results

rpart.plot(testcontrolparam1, extra = 1)

#dev.off()

predictions = predict(testcontrolparam1,

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)

#out of curiosity, let's see those cv results without cp (note: showing 0.012 std on cv 11)

#this tells me that we could probably keep the cp around 0.01.

#when we run a cp = 0.01, the results are very similar to the original model

printcp(testcontrolparam1)

```

**Appendix II Code**

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)

lengths(cv\_folds)

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

)

}

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

table(model\_nb$pred['Resample'])

rpart\_grid<-expand.grid(cp=seq(from=0, to=0.010, by=.0005 ))

train\_control<-trainControl(method="cv", number=5, savePredictions = 'final',summaryFunction = f1)

model\_rp<-caret::train(as.factor(isSpam) ~ .,data=emailDFnum, trControl = train\_control, method='rpart',tuneGrid = rpart\_grid)

model\_rp

**Appendix III Code**

```{r crossValidationProblem20}

### first identify the "our training data": for this we can use the data already sampled and separated between train and test sets in the original code. The data "msgWordLIst" represents the emails pulled apart from attachments, headers, and non-of-interest words (i.e., it is a string of all emails with the applicable words).

###the original code took this data object and sampled and seperated between Spam and Ham. Next samples were created for test MsgWords and Is/Is Not Spam. Those sub samples were then split into train and test sets. The applicalbe train sets are called "trainMsgWords" and "trainIsSpam" [note, this is a logical vector]. We will use the sample() function on those two train sets to create five sub-samples of equal size in order to cross-fold validation.

# cvTrainDF <- trainDF

#

# #randomize the data frame since all of the isSpam=T is at the beginning.

# cvTrainDF <- cvTrainDF[sample(1:nrow(cvTrainDF)), ]

#

# folds = createFolds(cvTrainDF$isSpam, k=5, list=TRUE)

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

#

# msgWordsList = unlist(msgWordsList, recursive = FALSE)

cvMsgWords1 <- msgWordsList

cvMsgWords1 <- cvMsgWords1[sample(length(cvMsgWords1))]

wordFoldTestIndexes <- createFolds(cvMsgWords1, k=5, list=TRUE)

wordFoldTestIndexes1 = wordFoldTestIndexes$Fold1

wordFoldTestIndexes2 = wordFoldTestIndexes$Fold2

wordFoldTestIndexes3 = wordFoldTestIndexes$Fold3

wordFoldTestIndexes4 = wordFoldTestIndexes$Fold4

wordFoldTestIndexes5 = wordFoldTestIndexes$Fold5

#not sure if this is the correct approach for isSpam vector

IsSpamFold1 = rep(c(TRUE, FALSE), c(length(wordFoldTestIndexes1), length(wordFoldTestIndexes1)))

IsSpamFold2 = rep(c(TRUE, FALSE), c(length(wordFoldTestIndexes2), length(wordFoldTestIndexes2)))

IsSpamFold3 = rep(c(TRUE, FALSE), c(length(wordFoldTestIndexes3), length(wordFoldTestIndexes3)))

IsSpamFold4 = rep(c(TRUE, FALSE), c(length(wordFoldTestIndexes4), length(wordFoldTestIndexes4)))

IsSpamFold5 = rep(c(TRUE, FALSE), c(length(wordFoldTestIndexes5), length(wordFoldTestIndexes5)))

wordTestFold1 <- cvMsgWords1[wordFoldTestIndexes1]

wordTrainFold1 <- cvMsgWords1[-wordFoldTestIndexes1]

wordTestFold2 <- cvMsgWords1[wordFoldTestIndexes2]

wordTrainFold2 <- cvMsgWords1[-wordFoldTestIndexes2]

wordTestFold3 <- cvMsgWords1[wordFoldTestIndexes3]

wordTrainFold3 <- cvMsgWords1[-wordFoldTestIndexes3]

wordTestFold4 <- cvMsgWords1[wordFoldTestIndexes4]

wordTrainFold4 <- cvMsgWords1[-wordFoldTestIndexes4]

wordTestFold5 <- cvMsgWords1[wordFoldTestIndexes5]

wordTrainFold5 <- cvMsgWords1[-wordFoldTestIndexes5]

pooledbow = unique(unlist(wordTrainFold5))

pooledbow = append(pooledbow, unique(unlist(wordTrainFold4)))

pooledbow = append(pooledbow, unique(unlist(wordTrainFold3)))

pooledbow = append(pooledbow, unique(unlist(wordTrainFold2)))

pooledbow = append(pooledbow, unique(unlist(wordTrainFold1)))

spamWordCounts1 = rep(0, length(pooledbow))

names(spamWordCounts1) = pooledbow

trainTable1 = computeFreqs1(wordTrainFold1, IsSpamFold1)

testLLRFold1 = sapply(wordTestFold1, computeMsgLLR, trainTable1)

trainTable2 = computeFreqs1(wordTrainFold2, IsSpamFold2)

testLLRFold2 = sapply(wordTestFold2, computeMsgLLR, trainTable2)

trainTable3 = computeFreqs1(wordTrainFold3, IsSpamFold3)

testLLRFold3 = sapply(wordTestFold3, computeMsgLLR, trainTable3)

trainTable4 = computeFreqs1(wordTrainFold4, IsSpamFold4)

testLLRFold4 = sapply(wordTestFold4, computeMsgLLR, trainTable4)

trainTable5 = computeFreqs1(wordTrainFold5, IsSpamFold5)

testLLRFold5 = sapply(wordTestFold5, computeMsgLLR, trainTable5)

pooledLLR = testLLRFold1

pooledLLR = append(pooledLLR, testLLRFold2, after = length(pooledLLR))

pooledLLR = append(pooledLLR, testLLRFold3, after = length(pooledLLR))

pooledLLR = append(pooledLLR, testLLRFold4, after = length(pooledLLR))

pooledLLR = append(pooledLLR, testLLRFold5, after = length(pooledLLR))

#need to figure out correct isSpam vector to send here.

cvTypeOneErrorRate = typeIErrorRates(pooledLLR, IsSpam)

cvTypeOneErrorRate

cvTau01 = round(min(cvTypeOneErrorRate$values[cvTypeOneErrorRate$error <= 0.01]))

cvTau01

```

1. https://www.forbes.com/sites/kateoflahertyuk/2018/08/17/how-to-survive-a-ransomware-attack-and-not-get-hit-again/#4807db156cd3 [↑](#footnote-ref-1)
2. Ibid [↑](#footnote-ref-2)
3. Ibid [↑](#footnote-ref-3)
4. https://www.newsweek.com/texas-ransomware-bitcoin-hackers-1454865 [↑](#footnote-ref-4)
5. Data Science in R, A Case Studies Approach to Computational Reasoning and Problem Solving Deborah Nolan, Duncan Temple Lang page 107 [↑](#footnote-ref-5)