Unit 10: Case Study on Using Statistics to Identify Spam

Damon Resnick
Collaborators: Rajni Goyal and Jim Hosker

March 22, 2018

Abstract

A Classification and Regression Trees or CART approach is used with the R function *rpart()* to make determinations about whether email is spam or not. This work focuses only on tweaking the *rpart.control()* parameters in order to find the best combination of parameters to minimize the number of false positives or non-spam messages predicted to be spam. It was found that minimizing the complexity parameter (*cp*) and *minsplit* parameter while nearly maximizing the *maxdepth* parameter gives the best prediction precision and hence minimizes the number of non-spam messages sent to the spam folder. Some evidence of overfitting was noted using this approach.

1 Introduction

This case study focuses on using a Classification and Regression Trees or CART approach with the R function *rpart()* to make determinations about whether email is spam or not. If you want the details about the data and its preparation that is outlined in chapter three of “The R Series” text book “Data Science in R A Case Studies Approach to Computational Reasoning and Problem Solving” by Deborah Nolan and Duncan Temple Lang. [1]

1.1 Email Data

The final data set consists of 9348 email messages 2397 spam messages and 6951 non-spam messages. The messages were processed, parsed, and split in order to create 29 different features that can be quantified or classified giving each individual email message a signature of a sort. There are three types of features. There are 17 logical or true/false variables that consist of whether the email has a certain feature or not. An example of a logical variable would be whether a message is a priority message or not, or if the message is in reply to another message, or if the message has an image attached. There are also six integer variables like the number of characters in the body of the message, or the number of exclamation marks. The last type is the simple numeric variables. There are seven of these numeric variables. An example of a numeric variable would be the percentage of capital letters in the message, or the percentage of blanks in the subject.

The values of each of these different features are determined by using functions to extract that information out of each message. The code from chapter 3 of the textbook details all the functions used to create the features and variables in the modeled data set. Examples of the variables and the functions are provided in a bit more detail below. [1]

2 Classification and Regression Trees (CART)

A function that uses classification and regression trees was used to make the predictions in this study. The *rpart()* function, is a function used in the R coding environment, and is from the rpart package which is an implementation of most of the functionality of the 1984 book by Breiman, Friedman, Olshen, and Stone. [2, 3]

To build a decision tree from derived features, a recursive partitioning method is applied. The way recursive partitioning works is to divide the data into two different groups based on the value of a certain variable. An example of this would be splitting the data based on the amount of capital letters, say if that amount was above or below a set percentage. After the initial split, one group of the data is further divided into two groups. Again, the split is performed according to the value of a specified variable of interest. The data groups are then further divided until all messages have been partitioned into subsets that can be classified as spam or not-spam based on these characteristics. If the resulting sub-groups created from recursive partitioning are drawn as a diagram, a tree shape with many branches is formed, hence the name decision tree.

The overall goal here with this technique is not only to transfer key features into quantifiable measures, but to attempt partitioning the message data into groups that are ultimately as similar as possible, so either all span or all not-spam. Thus, in a single subgroup, observations that comprise a leaf located at the base of the tree are assigned the same classification. If leaf messages are as similar as possible, we can then say misclassification errors have been reduced.

Now that the basics of the recursive partition method has been outlined, we can now identify features of interest and develop functions that process email messages into variables. Let's apply the recursive partitioning method to our constructed variables and determine how well our model predicts email messages as spam or not-spam. We also consider other parameters for controlling the recursive partitioning process, specifically experimenting with values for *rpart()* tuning parameters [2]. Varying parameters in an effort to tune find the best combination will then allow us to more accurately assess the resulting classification trees and if prediction is improved.

3 Methods

In this study the parameters in *rpart.control* are adjusted in order to produce a fit to the data that minimizes the number of false positive or non-spam messages classified as spam. This is the same as minimizing the false discovery rate or maximizing the positive predictive values (PPV) which is another name for the precision.

3.1 False Positives are Bad

In this classification example we attempt to predict whether a message is spam or not. Following this characterization, we get these four possible results:

TP: Correctly identified as Spam; It is Spam and predicted as Spam
TN: Correctly identified as not-Spam; It is not-Spam and predicted as not-Spam

FP: Incorrectly identified as Spam; It is not-Spam but predicted as Spam, (Type I error)
FN: Incorrectly identified as not-Spam; It is Spam but predicted as not-Spam, (Type II error)

It should be mentioned briefly here that the FP and FN results are equivalent to the type I and type II errors respectively, given a specific null hypothesis. Since the outlined model attempts to predict whether an email message is spam or not-spam given a set of features an acceptable null hypothesis is difficult to formulate. The basic alternative hypothesis would be that certain combinations of features at certain levels classify a message as spam, that would lead to a null hypothesis that is opposite this, H_0: Certain combinations of features at certain levels classify a message as not-spam. Using this null hypothesis, it be shown the equivalence of the type I and type II errors to FP and FN results respectively.

Minimizing the number of false positives was chosen as the main goal of this project. People do not want spam in their email folder, therefore you would think the goal of maximizing true positives and/or true negatives would be the goal, however having an email message appear in the spam folder when it is in fact not spam is much less desirable of an outcome than having a few spam messages show up in your email folder. Better to have 10 spam messages in your email folder rather than 1 not-spam message in the spam folder. Therefore, *minimizing* email messages classified as spam, when they are not-spam, is the main goal of this project.

The number of true positives, false positives, true negatives, and false negatives were determined by changing the *errs* function to the *confusion_matrix* function below:

```
confusion_matrix = sapply(fits_b, function(preds) {
  FP = sum(preds[ !spam ] == "T") # Sum of Ham Predictions that are True
  FN = sum(preds[ spam ] == "F") # Sum of Spam Predictions that are False
  TP = sum(preds[ spam ] == "T") # Sum of Spam Predictions that are True
  TN = sum(preds[ !spam ] == "F") # Sum of Ham Predictions that are False
  c(FP = FP, FN = FN, TP = TP, TN = TN)
})
```

In order to determine if the false positives were minimized it was noted that the false discovery rate (FDR) is equal to one minus the precision, or FDR = FP/(FP + TP) = 1 – PPV = 1 – TP/(TP + FP). The output of the *confusion_matrix* function was then used to make plots of precision versus the value of the parameter that was varied.

4 Results

The parameters in the *rpart.contol* are as defined in the documentation. [2] The values that were chosen for these parameters to maximize the precision are listed below along with the documentation descriptions of each parameter. A brief description of how these parameter values were determined is given as well.

minsplit = 2 maximizes the precision
The minimum number of observations that must exist in a node in order for a split to be attempted.

minbucket = 1, must be smaller than minsplit to give the best results the minimum number of observations in any terminal <leaf> node. If only one of minbucket or minsplit is specified, the code either sets minsplit to minbucket*3 or minbucket to minsplit/3, as appropriate.

cp = 0.0001 maximizes the precision
Complexity parameter. Any split that does not decrease the overall lack of fit by a factor of cp is not attempted. For instance, with anova splitting, this means that the overall R-squared must increase by cp at each step. The main role of this parameter is to save computing time by pruning off splits that are obviously not worthwhile. Essentially, the user informs the program that any split which does not improve the fit by cp will likely be pruned off by cross-validation, and that hence the program need not pursue it.

maxcompete = 4 as default, varying doesn't change results at all
The number of competitor splits retained in the output. It is useful to know not just which split was chosen, but which variable came in second, third, etc.

maxsurrogate = 5 as default, varying doesn't change results at all
The number of surrogate splits retained in the output. If this is set to zero the compute time will be reduced, since approximately half of the computational time (other than setup) is used in the search for surrogate splits.

usesurrogate = 2 maximizes the precision
How to use surrogates in the splitting process. 0 means display only; an observation with a missing value for the primary split rule is not sent further down the tree. 1 means use surrogates, in order, to split subjects missing the primary variable; if all surrogates are missing the observation is not split. For value 2, if all surrogates are missing, then send the observation in the majority direction. A value of 0 corresponds to the action of tree, and 2 to the recommendations of Breiman et.al (1984).

xval =10 default, varying doesn't change results of predictions
Number of cross-validations.

surrogatestyle = 0 default, varying doesn't change results of predictions
controls the selection of a best surrogate. If set to 0 (default) the program uses the total number of correct classification for a potential surrogate variable, if set to 1 it uses the percent correct, calculated over the non-missing values of the surrogate. The first option more severely penalizes covariates with a large number of missing values.

maxdepth = 29 maximizes the precision; range of 0 to 30,
Set the maximum depth of any node of the final tree, with the root node counted as depth 0. Values greater than 30 rpart will give nonsense results on 32-bit machines.

The parameter values above were determined to minimize the number of false positives by starting with the default values, getting a precision, then seeing if varying one of the parameters changed the precision. For instance, the default parameters were chosen, then cp was varied from 0 to 1. The cp value that maximized the precision was then chosen. After that cp was fixed along with all the other parameters and then maxdepth was varied. Each parameter was varied in turn and the value that maximized precision was used each time. After the first run through several other iterations of this treatment were performed to explore the parameter space.

Some parameters did not affect the results at all, and a few parameters only had a few value options to choose and the value that maximized precision was always chosen. Ostensibly this means that only a few of the parameters had a significant effect on the results. These most significant parameters are cp, minsplit/minbucket, and maxdepth.

Changing the cp parameter generally effected the number of trees produced:

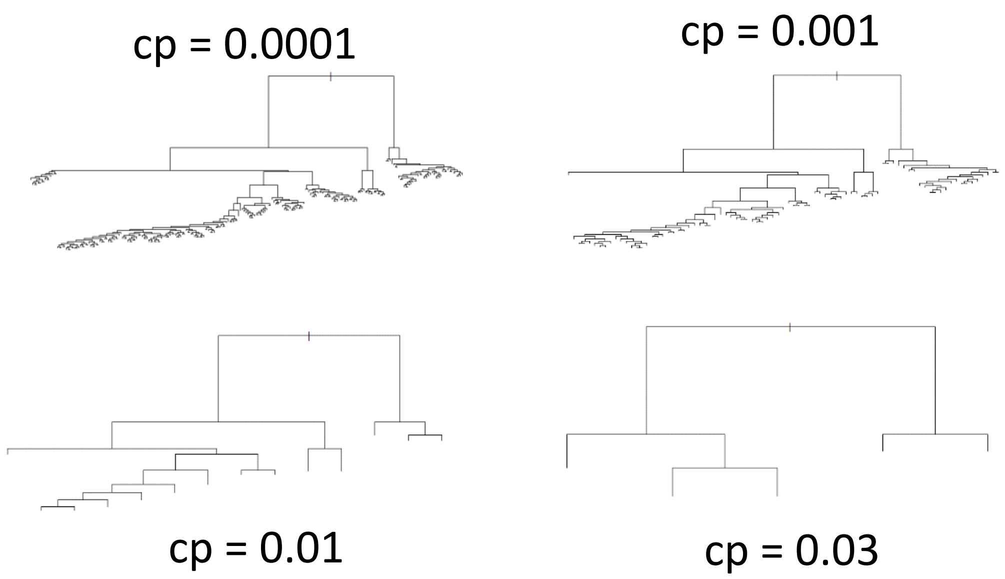

Figure 1: Varying the cp parameter changes the number of trees. The highest precision is achieved for the lowest values of cp that also give the largest number of trees.

A similar reduction of the number of trees was noted with the reduction of the maxdepth parameter as well as with the increase in the minsplit parameter. It was noticed that the number of trees also correlated to the size of the precision and accuracy. When the number of trees was large the precision and accuracy was large. It can be clearly seen in figure 1 that more trees correlate with small cp values.

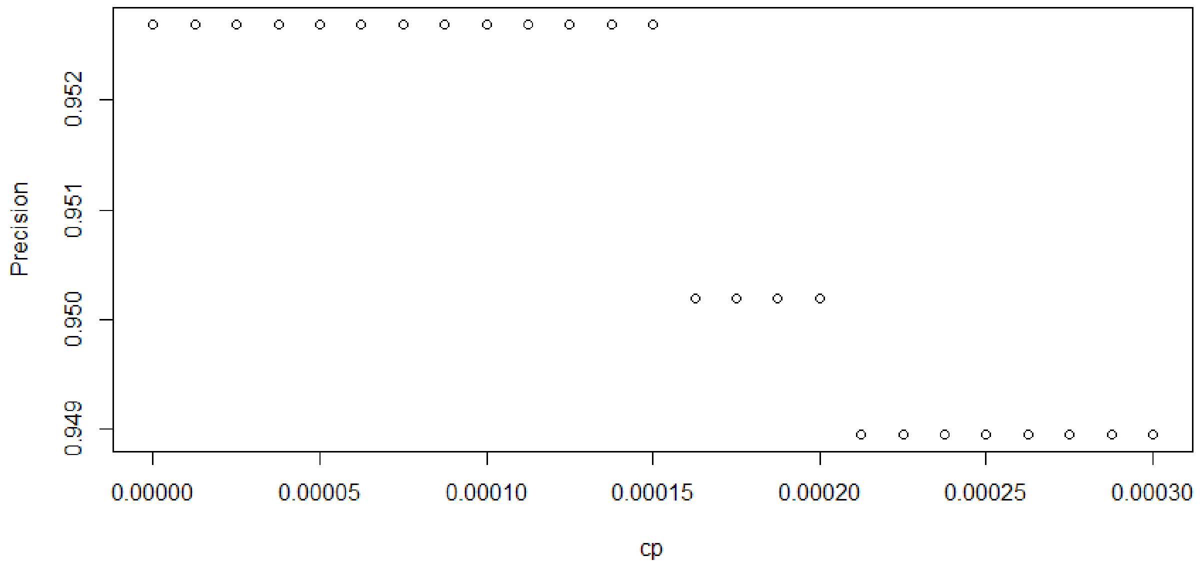

Figure 2: Varying the cp parameter changes the precision. The highest precision is achieved for the lowest values of cp down to roughly 0.00015. A value of 0.0001 was used.

Figure 2 shows what happens when cp is varied. As cp gets larger the number of trees is reduced and the precision and accuracy are both smaller.

Two other things were attempted to increase the precision. First it was noticed that over 200 NA values were scattered throughout the data set in different columns. These were imputed with the median values for each column. Second, there was some evidence for overfitting using this method. To check for that the same parameters were used on different training and test sets. It was noticed that while the accuracy increased for one of these other seed values for the test and training sets, the precision was smaller in each case. This is an indication of overfitting but not conclusive. Therefore, it is suggested that a k-fold approach might be a better way to test the model. Table 1 has the values of the results for different seeds to create the test and training sets.

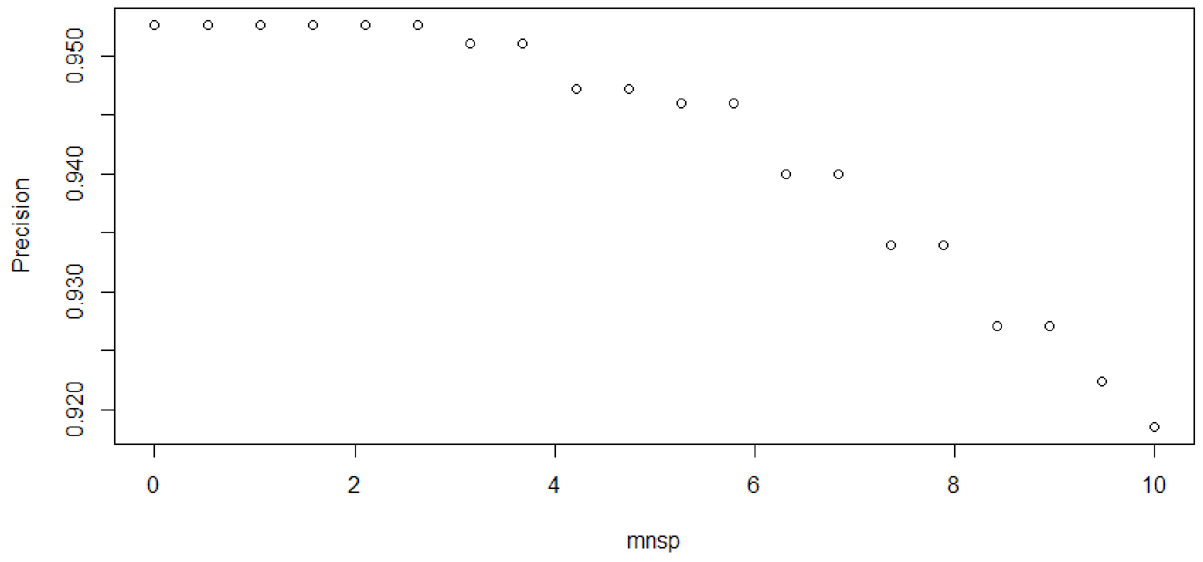

Figure 3: Varying the minsplit parameter changes the precision. The highest precision is achieved for the lowest values of minsplit down to roughly 2. A value of 2 was used.

Figure 3 shows what happens to the precision when minsplit is varied. As the minsplit value increases the precision decreases. This makes sense, as the smaller minsplit is the greater the possibility of more trees.

Figure 4 shows what happens to the precision when maxdepth is varied. As the maxdepth value increases the precision increases. This makes sense, as the larger maxdepth is the greater the possibility of more trees.

Table 1 has the details of the final models' results after the NAs have been imputed with median values and different seeds were used to explore the possibility of overfitting. Table 2 shows a simple confusion matrix outlining the sums of the four possible outcomes: TP, FP, TN, and FN.

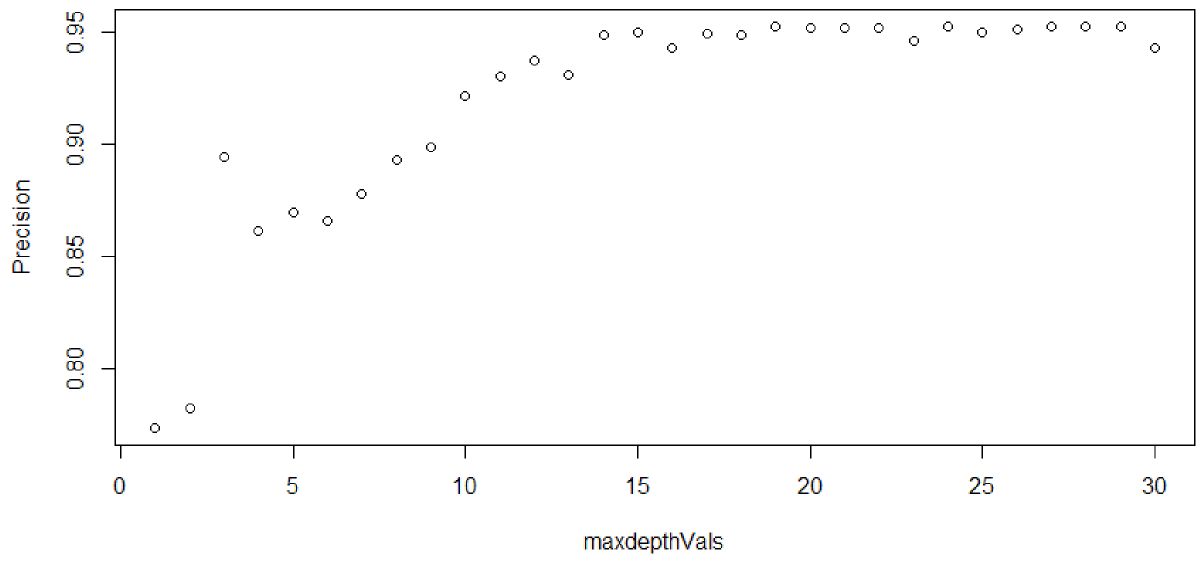

Figure 4: Varying the maxdepth parameter changes the precision. The highest precision is achieved for the highest values of maxdepth up to roughly 29. The values of precision reach a plateau around a maxdepth of 19 and fluctuates up and down around a maximum value.

Seed	FP	FN	TP	TN	Sum	Accuracy	Precision	Recall	F1	NAs Imputed
418910	36	75	724	2281	3116	0.9643774	0.9526316	0.9061327	0.9288005	NO
418910	36	74	725	2281	3116	0.9646983	0.9526938	0.9073842	0.9294872	YES
1000	44	65	734	2273	3116	0.9650193	0.9434447	0.9186483	0.9308814	YES
7	54	70	729	2263	3116	0.9602054	0.9310345	0.9123905	0.9216182	YES
11	44	74	725	2273	3116	0.9621309	0.9427828	0.9073842	0.9247449	YES

Table 1: Both precision and accuracy increased slightly when the NAs were imputed with the median values. Changing the seed for the creation of the test and train sets decrease precision in all cases while accuracy did increase for the 1000 seed.

Confusion Matrix		Actual Classification	
		Spam	not-Spam
Predicted:	Spam	TPs = 725	FPs = 36
Predicted:	not-Spam	FNs = 74	TNs = 2281

Table 2: Simple confusion matrix to show the relative numbers of the four possible outcomes: True Positive (TP), False Positive (FP), True Negative (TN), and False Negative (FN). These are the results of from the original seed value after the NAs were imputed with the median values.

Further studies of this prediction method should next include feature selection. Evidence of overfitting was also noted. Several of the features no doubt explains some of the same variance in the data. It is recommended that a k-fold test and training technique be used for feature selection and then parameter tweaking can again be performed. An alternative to this would be to include a validation set held out of the testing phase to reach a conclusion on overfitting.

5 Conclusion

A Classification and Regression Trees or CART approach is used with the R function *rpart()* to make determinations about whether email is spam or not. This work focused only on tweaking the *rpart.control()* parameters in order to find the best combination of parameters to minimize the number of false positives or non-spam messages predicted to be spam. It was found that minimizing the complexity parameter (*cp*) and *minsplit* parameter while nearly maximizing the *maxdepth* parameter gives the best prediction precision and hence minimizes the number of non-spam messages sent to the spam folder.

It was also found that over 200 NAs were found throughout the original data set. Imputing those values with the median values slightly improved the precision. Some evidence of overfitting was also found using the method described. In order to minimize overfitting a k-fold approach or validation set holdout is recommended for future work but was not used here.

References

[1] "The R Series" text book "Data Science in R A Case Studies Approach to Computational Reasoning and Problem Solving" by Deborah Nolan and Duncan Temple Lang, CRC Press, 2015.

[2] B. Atkinson, B. Ripley, & T. Therneau, Package 'rpart'. Version 4.1-11. Online: 2017-03-12. https://cran.r-project.org/web/packages/rpart/rpart.pdf

[3] Breiman L., Friedman J. H., Olshen R. A., and Stone, C. J. (1984) Classification and Regression Trees. Wadsworth.

Appendix A: Code

Listed here is the code that was changed or added in order to answer the questions. The rest of the code is in an attached .txt file.

Please see attachment for entirety of code.

```
# complexityVals_2b = c(seq(0.00001, 0.0003, length=20),
#                   seq(0.0003, 0.002, length=3),
#                   seq(0.002, 0.002, length=5),
#                   seq(0.002, 0.01, length=2))

complexityVals_2b = c(seq(0, 0.0003, length=25))


# cp = 0.001448276 looks to be close to the best F1
# now cp = 0.0013 seems to give a best precision value

#cpvals_3c = c(0.001379310, 0.001413793, 0.001448276, 0.001482759)
minsplitVals = c(seq(0, 10, length=20))
#minsplitVals = 0 to 2 seems to give the best values for precision
maxdepthVals = c(seq(1, 30, length=30))
#maxdepthVals = 29 seems to give the best values for precision
#minsplitVals = 0 to 2 seems to give the best values for precision

#maxdepthVals = c(seq(16, 20, length=20))
# maxdepth = 19 seems to give the best precission value

# surrogatestyle seems to change nothing
usesurrogateVals = c(seq(1, 5, length=3))
#usesurrogate = 3 #seems to be the best F1
#maxsurrogateVals = c(seq(1, 5, length=20))
#maxsurrogate = 1 #at least to get best F1
maxcompeteVals = c(seq(2, 3, length=2))
# maxcompete did not change anything
# xval did not change anything
# minbucket can't figure out minbucket, just seems to make it worse putting values in for it
#
# fits_b = lapply(minsplitVals, function(x) {
#   rpartObj = rpart(isSpam ~ ., data = trainDF,
#                   method="class",
#                   control = rpart.control(maxdepth = 20, xval = 1, maxcompete = 0, maxsurrogate =
1, cp=0.008, usesurrogate = 2, surrogatestyle = 0, minsplit = x) )
#
#   predict(rpartObj,
#           newdata = testDF[ , names(testDF) != "isSpam"],
#           type = "class")
# })


fits_b = lapply(maxdepthVals, function(x) {
  rpartObj = rpart(isSpam ~ ., data = trainDF,
                   method="class",
                   control = rpart.control(maxdepth = x, xval = 10, minbucket = 1,
                     maxcompete = 4, maxsurrogate = 5, cp=0.0001, minsplit = 2,
                     usesurrogate = 2, surrogatestyle = 0) )

  predict(rpartObj,
          newdata = testDF[ , names(testDF) != "isSpam"],
          type = "class")
})


confusion_matrix =  sapply(fits_b, function(preds) {
  FP = sum(preds[ !spam ] == "T") # Sum of Ham Predictions that are True
  FN = sum(preds[ spam ] == "F") # Sum of Spam Predictions that are False
  TP = sum(preds[ spam ] == "T") # Sum of Spam Predictions that are True
  TN = sum(preds[ !spam ] == "F") # Sum of Ham Predictions that are False
  c(FP = FP, FN = FN, TP = TP, TN = TN)
})

trans_confusion = t(confusion_matrix)

df_confusion = as.data.frame(trans_confusion)

df_confusion$Sum = df_confusion$FP + df_confusion$FN + df_confusion$TP +
df_confusion$TN
df_confusion$Accuracy = (df_confusion$TP + df_confusion$TN)/(df_confusion$Sum)
df_confusion$Precision = (df_confusion$TP)/(df_confusion$TP + df_confusion$FP)
df_confusion$Recall = (df_confusion$TP)/(df_confusion$TP + df_confusion$FN)
df_confusion$F1 = 2/((1/df_confusion$Precision) + (1/df_confusion$Recall))

#df_cp = as.numeric(complexityVals_2b)
#df_confusion$cp = df_cp
df_confusion$cp = 0.0001
#df_mnsp = as.numeric(minsplitVals)
#df_confusion$mnsp = df_mnsp
```

```
df_mnsp = 2
#df_confusion$mnsp = 6
df_maxdepthVals = as.numeric(maxdepthVals)
df_confusion$maxdepthVals = df_maxdepthVals
#df_confusion$maxdepthVals = 19
# df_usesurrogateVals = as.numeric(usesurrogateVals)
# df_confusion$usesurrogateVals = df_usesurrogateVals
df_confusion$usesurrogateVals = 3
#df_maxcompeteVals = as.numeric(maxcompeteVals)
#df_confusion$maxcompeteVals = df_maxcompeteVals
df_confusion$maxcompeteVals = 5
#df_maxdepthVals = as.numeric(maxdepthVals)
#df_confusion$maxdepthVals = df_maxdepthVals
#df_confusion$maxdepthVals = 29

# Plot Recall
plot(df_confusion$maxdepthVals, df_confusion$Recall, xlab = "maxdepthVals", ylab =
"Recall")

# max F1
df_confusion[df_confusion$F1 == max(df_confusion$F1),]

# Plot Accuracy
plot(df_confusion$maxdepthVals, df_confusion$Accuracy, xlab = "maxdepthVals", ylab =
"Accuracy")

# Plot F1
plot(df_confusion$maxdepthVals, df_confusion$F1, xlab = "maxdepthVals", ylab = "F1")

# max Precision
df_confusion[df_confusion$Precision == max(df_confusion$Precision),]

# Plot Precision
plot(df_confusion$maxdepthVals, df_confusion$Precision, xlab = "maxdepthVals", ylab =
"Precision")

plotfit <- rpart(isSpam ~ ., data = trainDF,
    method="class",
    control = rpart.control(maxdepth = 29, xval = 10, minbucket = 1,
                    maxcompete = 4, maxsurrogate = 5, cp=0.0001, minsplit = 2,
                    usesurrogate = 2, surrogatestyle = 0))

plot(plotfit)
#rpart.plot(plotfit)
#fancyRpartPlot(plotfit)
```