Assignment 10: Using R For Machine Learning

Due Wednesday, May 8 by 11:59pm.

Prediction Using Decision Tree Classifiers

In this worksheet, you'll go over using R to create Decision Tree Classifiers. There's a lot of code here – feel free to copy and paste much of it as you work on the exercises and you do your projects. This is also only barely scratching the surface, and intended to be as transparent of an introduction as possible in terms of showing how everything is done. Of note, the `caret` package in R can be extremely helpful in doing some of what's done here with much less code involved (see https://www.machinelearningplus.com/machine-learning/caret-package/ for more information). However, there's somewhat of a learning curve associated with it, as well as the inability to see how everything works, so I only use the `caret` package to do a few things, like calculating precision and recall.

We'll use the same datasets as the previous assignment, using the spam emails for our example and kickstarter for the exercises.

Motivation: Identifying Spam Email

Suppose you want to decide which emails to treat seriously and which emails to delete as spam (ignoring any spam filters you may already have in place). You might be able to make some heuristic rules about which emails are spam, such as ones that say you're a winner, or aren't responses, but how well can these heuristics actually work? And, is there a better way to identify spam emails?

Let's get started by bringing some packages that we'll use for our classification trees.

```r
require(rpart)
require(rpart.plot)
require(rattle)
require(caret)
require(dplyr)

email <- read.csv('~/Downloads/email.csv', header = TRUE, row.names = 1)
```

Recall that we had to do some data cleaning to make sure we have the right variable types and remove all NA values. First, let's make some variables into factors so that `R` treats them as categorical.

```r
email$spam <- factor(email$spam)
email$re_subj <- factor(email$re_subj)
email$cc <- factor(email$cc)
email$image <- factor(email$image)
```

Note: Here, in `spam`, 0 means not spam and 1 means spam. In `re_subj`, 0 means that it did not have "Re:" in the subject line while 1 means that it did. Similarly, for `cc`, 0 means that it did not have a CC, while 1 means it did, and for `image`, 0 means it had an image while 1 means it did not.

Let's finish up cleaning the dataset by removing all NA rows.

```r
email_clean <- na.omit(email)
```

We want to use validation to make sure our model doesn't overfit. To do this, we first need to create our train and test sets, then run our model with the training sets. Here, we'll demonstrate with just a simple holdout sample, with 10% of the data as our test set.

```r
rows_test <- sample(1:nrow(email_clean), floor(0.1*nrow(email_clean)))
email_test <- email_clean[rows_test,]
email_train <- email_clean[-rows_test,]
```

Now that we have a dataset that we can use, running the actual tree model is actually quite simple. If you have used R for running linear models before, the format is very similar.

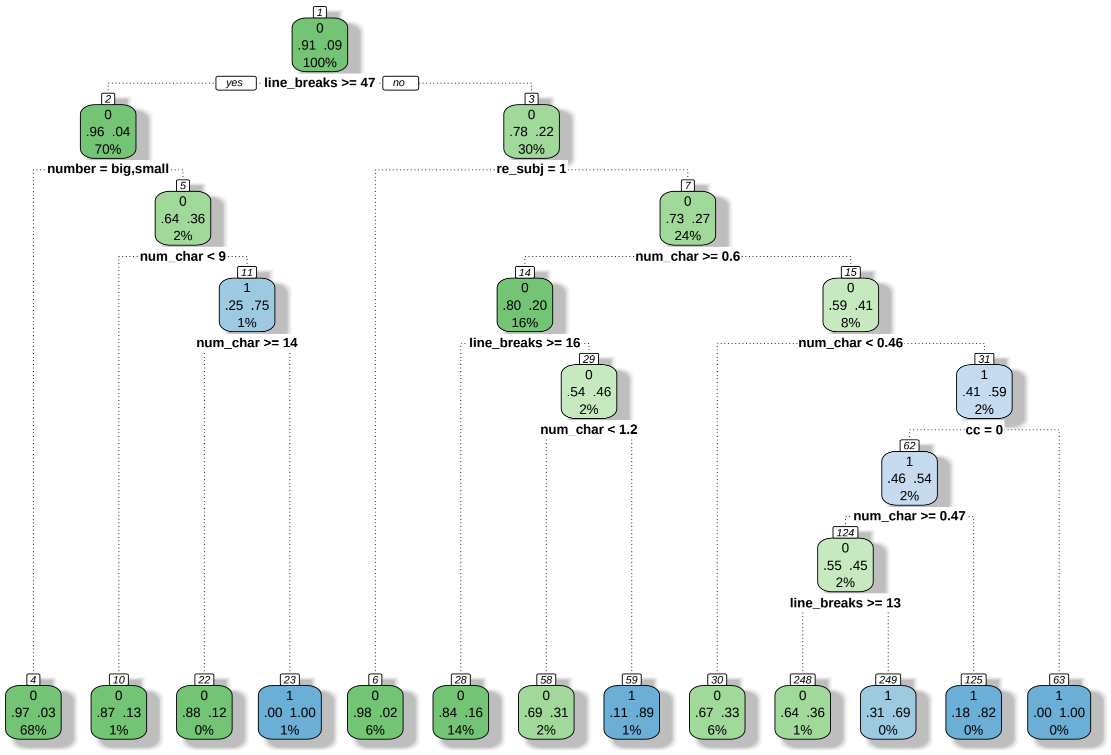
```r
treemod <- rpart(spam ~ num_char + line_breaks + number + winner + re_subj + cc + image,
                 data = email_train,
                 method = 'class',
                 control = rpart.control(minsplit = 25))
```

Let's break down each of the arguments in this function. First, we specify the model, putting the label that we want to predict on the left side of the "~" and all the features we want to include on the right. We include arguments for the dataframe from which we're taking the data, and the tree method we want to use (in this case, since we are doing a classification tree, we use 'class'). Then, we can use the `control` argument to set decision tree parameters. In this case, we are setting the minimum number of observations needed in a node to add a split.

We have stored the model in the `treemod` object. Let's look at what the model gave us. We can use summary to look at the summary of the model, but it might be easier to look a visualization instead.

```r
# You can try running the summary, but it will give a LOT of output
# summary(treemod)

# The fancy tree visualization
fancyRpartPlot(treemod, sub = "")
```



Evaluating the Model

Now that we have a model, we need to test it. We can get predictions using the `predict` function.

```
pred <- predict(treemod, email_test)
```

Note that this gives us the prediction scores for both spam and not spam. They are basically the same thing, since you can get one from taking one minus the other. We'll focus on the spam one, since we want to identify the emails most likely to be spam. Let's attach it to the test set data frame.

```
email_test$spam_score <- pred[,2]
```

Now, we're only really interested in the columns that represent the prediction and the actual value of whether an email was spam or not. So, let's select just the columns of our dataframe that we need.

```
test_pred <- email_test %>% select(spam, spam_score)
```

Finally, we need a way to convert from the scores to an actual prediction. Typically, we use some sort of arbitrary value as a cutoff, or take a certain percentage of the most likely observations (for example, the top 10% most likely are predicted to be spam).

We'll go with the latter for this example. Let's go with a 20% threshold.

```
test_pred <- test_pred %>% arrange(desc(spam_score))
test_pred$pred <- 0

top_scores <- floor(nrow(test_pred)*0.2)
test_pred$pred[1:top_scores] <- 1
```

We can get the values of precision and recall using `confusionMatrix` function from the `caret` package. First, we create a table with the confusion matrix, then run the function with the table as the argument. Note that we specify what the `positive` value is – since we are trying to predict what emails are spam, we have 1 as our positive value. In addition, make sure your predicted values are first in the table, or else you'll get the opposite results as you want!

```
pred_tab <- table(test_pred$pred,test_pred$spam)
confusionMatrix(pred_tab, positive = "1")
```

```
## Confusion Matrix and Statistics
##
##
##       0   1
##   0 300   8
##   1  51  25
##
##                Accuracy : 0.8464
##                  95% CI : (0.8063, 0.8809)
##     No Information Rate : 0.9141
##     P-Value [Acc > NIR] : 1
##
##                   Kappa : 0.385
##  Mcnemar's Test P-Value : 4.553e-08
##
##             Sensitivity : 0.75758
##             Specificity : 0.85470
##          Pos Pred Value : 0.32895
##          Neg Pred Value : 0.97403
##              Prevalence : 0.08594
##          Detection Rate : 0.06510
##    Detection Prevalence : 0.19792
##       Balanced Accuracy : 0.80614
##
##        'Positive' Class : 1
##
```

Note that we don't actually see the words "precision" or "recall" here – instead, we can find them by their alternate names: sensitivity (for recall) and positive predictive value (for precision). We can also use the `precision` and `recall` functions (also in the `caret` package). Note that we use `relevant` to specify which outcome we're trying to predict (similar to the `positive` argument above).

```r
precision(pred_tab, relevant = '1')
```

```
## [1] 0.3289474
```

```r
recall(pred_tab, relevant = '1')
```

```
## [1] 0.7575758
```

These numbers don't actually mean that much by themselves – we need a baseline to compare against. Let's look at how much spam there really was in the ttest set.

```r
summary(email_test$spam)
```

```
##   0   1
## 351  33
```

If we had just randomly chosen emails to designate as spam, we would have been correct about 10% of the time.

Finding the Best Model

We've gone through one iteration of running the model. But, remember, there are many different parameters we can adjust – we can change our threshold of what we predict as spam, as well as various characteristics about the decision tree. How do we find the best model? We can use a `for` loop to go through many different models and find the best one that way.

Here, let's look at an example where we go through different minsplit values and maxdepth (how many steps we can go from the root node), as well as choosing a different percentage to predict as spam.

```r
# We will look at minsplit values of 5, 10, 15, 20
splits <- c(5,10,15,20)

# We'll look at maxdepths of 2, 3, 4, 5
depths <- c(2,3,4,5)

# We'll consider predicting the top 5%, 10%, and 20% as spam
percent <- c(.05, .1, .2)

# How many different models are we running?
nmods <- length(splits)*length(depths)*length(percent)

# We will store results in this data frame
results <- data.frame(splits = rep(NA,nmods),
                      depths = rep(NA, nmods),
                      percent = rep(NA,nmods),
                      precision = rep(NA,nmods),
                      recall = rep(NA,nmods))

# The model number that we will iterate on (aka models run so far)
mod_num <- 1

# The loop
for(i in 1:length(splits)){
  for(j in 1:length(depths)){
    s <- splits[i]
    d <- depths[j]
    # Running the model
    treemod <- rpart(spam ~ num_char + line_breaks + number +
                       winner + re_subj + cc + image,
                     data = email_train,method = 'class',
                     control = rpart.control(minsplit = s, maxdepth = d))

    # Find the predictions
    pred <- predict(treemod, email_test)

    # Attach scores to the test set
    # Then sort by descending order
    email_test$spam_score <- pred[,2]
    test_pred <- email_test %>% select(spam, spam_score) %>%
      arrange(desc(spam_score))

    # Make predictions based on scores
    # We loop through each threshold value here.
    for(k in 1:length(percent)){
      p <- percent[k]

      # Predict the top % as 1
      test_pred$pred <- 0
      top_scores <- floor(nrow(test_pred)*p)
      test_pred$pred[1:top_scores] <- 1

      # Confusion Matrix
      pred_tab <- table(test_pred$pred,test_pred$spam)

      # Store results
      results[mod_num,] <- c(s,
                             d,
                             p,
                             precision(pred_tab, relevant = "1"),
                             recall(pred_tab, relevant = "1"))
      # Increment the model number
      mod_num <- mod_num + 1
    }
  }
}

# All results are stored in the "results" dataframe
head(results)
```

```
##   splits depths percent  precision     recall
## 1      5      2    0.05 0.00000000 0.00000000
## 2      5      2    0.10 0.07894737 0.09090909
## 3      5      2    0.20 0.07894737 0.18181818
## 4      5      3    0.05 0.31578947 0.18181818
## 5      5      3    0.10 0.31578947 0.36363636
## 6      5      3    0.20 0.26315789 0.60606061
```

```
# Best recall? Top 5 in descending order
results %>% arrange(desc(recall)) %>% head()
```

```
##   splits depths percent precision    recall
## 1      5      5     0.2 0.3289474 0.7575758
## 2     10      5     0.2 0.3289474 0.7575758
## 3     15      5     0.2 0.3289474 0.7575758
## 4     20      5     0.2 0.3289474 0.7575758
## 5      5      4     0.2 0.2763158 0.6363636
## 6     10      4     0.2 0.2763158 0.6363636
```

```
# Best precision? Top 5 in descending order
results %>% arrange(desc(precision)) %>% head()
```

```
##   splits depths percent precision    recall
## 1      5      5    0.05 0.6842105 0.3939394
## 2     10      5    0.05 0.6842105 0.3939394
## 3     15      5    0.05 0.6842105 0.3939394
## 4     20      5    0.05 0.5789474 0.3333333
## 5      5      4    0.05 0.5263158 0.3030303
## 6      5      5    0.10 0.5263158 0.6060606
```

This whole process is quite involved, but very much worth knowing. I would highly suggest working your way through the code above and going through how each part works.

Simplifying the Process with Caret

We've already used the `caret` package for tools like the confusion matrix and precision/recall. However, we can also use it to do other validation methods more easily, such as k-fold cross validation. Here, we'll look at a quick example of using `caret` to find the training and test sets need to do 10-fold cross validation with the email dataset.

```
# Create a list of 10 folds (each element has indices of the fold)
flds <- createFolds(email_clean$spam, k = 10, list = TRUE, returnTrain = FALSE)
str(flds)
```

```
## List of 10
##  $ Fold01: int [1:385] 7 33 35 41 44 66 68 73 75 78 ...
##  $ Fold02: int [1:385] 8 22 34 40 47 51 58 60 89 92 ...
##  $ Fold03: int [1:385] 12 20 23 26 27 32 38 49 65 67 ...
##  $ Fold04: int [1:385] 5 17 25 39 61 69 70 84 87 90 ...
##  $ Fold05: int [1:385] 15 19 28 29 48 50 82 98 101 107 ...
##  $ Fold06: int [1:385] 1 2 18 21 31 37 45 56 63 64 ...
##  $ Fold07: int [1:384] 6 24 46 54 59 71 76 83 91 95 ...
##  $ Fold08: int [1:385] 4 10 11 14 30 52 55 57 80 86 ...
##  $ Fold09: int [1:385] 3 9 13 36 74 77 85 94 100 103 ...
##  $ Fold10: int [1:385] 16 42 43 53 62 102 104 112 121 127 ...
```

As you can see, we created a list of 10 vectors, each containing a fold. So, to do our 10-fold cross validation, we can take the first fold, and create our train and test sets from that, take the second fold and create test and train from that, and so on. For example, to create test and train using the first fold, we can use the following code:

```r
# Create train and test using fold 1 as test
email_test01 <- email_clean[flds$Fold01,]
email_train01 <- email_clean[-flds$Fold01,]
```

You can then use these train and test sets as we have above.

Questions

Download and bring in the kickstarter dataset.

```r
kickstarter <- read.csv('~/Downloads/kickstarter.csv', header = TRUE, row.names = 1)
```

This dataset represents a sample of kickstarter campaigns, including information about the name of the kickstarter, state (whether it failed or was successful), the category, how much money was pledged (in USD), what its initial goal was, and more.

Using the kickstarter dataset and the information above, answer the following questions.

1) Make sure you do the appropriate data cleaning and split the data into 20% test set and 80% train set. Show the code for how you did this. (5 points)

2) Run a tree model to predict state (failed vs successful) using goal, country, and main category as the features. Show the tree plot. (5 points)

3) Find the prediction scores and set the top 40% as the threshold for a "successful" prediction. Display the confusion matrix. (5 points)

4) Calculate the precision and the recall. Interpret what they mean in context. (4 points)

5) Suppose you are investor trying to determine good kickstarters that are likely to succeed. Would you be more interested in precision or in recall? How would you evalute the performance of this model based on your metric of choice (Think about what a "baseline" would be if you were to choose randomly) (5 points)

6) Try running the model with different values of max depth and minimum number of observations needed for a split. You don't necessarily need to run the loop (though it will be faster with it), but run at least 3 additional models and find precision/recall for those. Which version of the model has the best performance? (11 points)