512 Take-Home Final

Brody Vogel

5/1/2018

```r
library(mlbench)
library(tree)
```

```
## Warning: package 'tree' was built under R version 3.4.4
```

```r
library(boot)
library(glmnet)
```

```
## Loading required package: Matrix
```

```
## Loading required package: foreach
```

```
## Loaded glmnet 2.0-13
```

Part A

Preparation)

```r
data(Ozone)

# get the appropriate data
Fixed <- subset(Ozone, select = -c(2, 3))
names(Fixed) <- c('month', 'dailymax', 'pressure', 'windlax', 'humlax', 'tsand', 'telmonte', 'invlax',

#View(Fixed)
```

1)

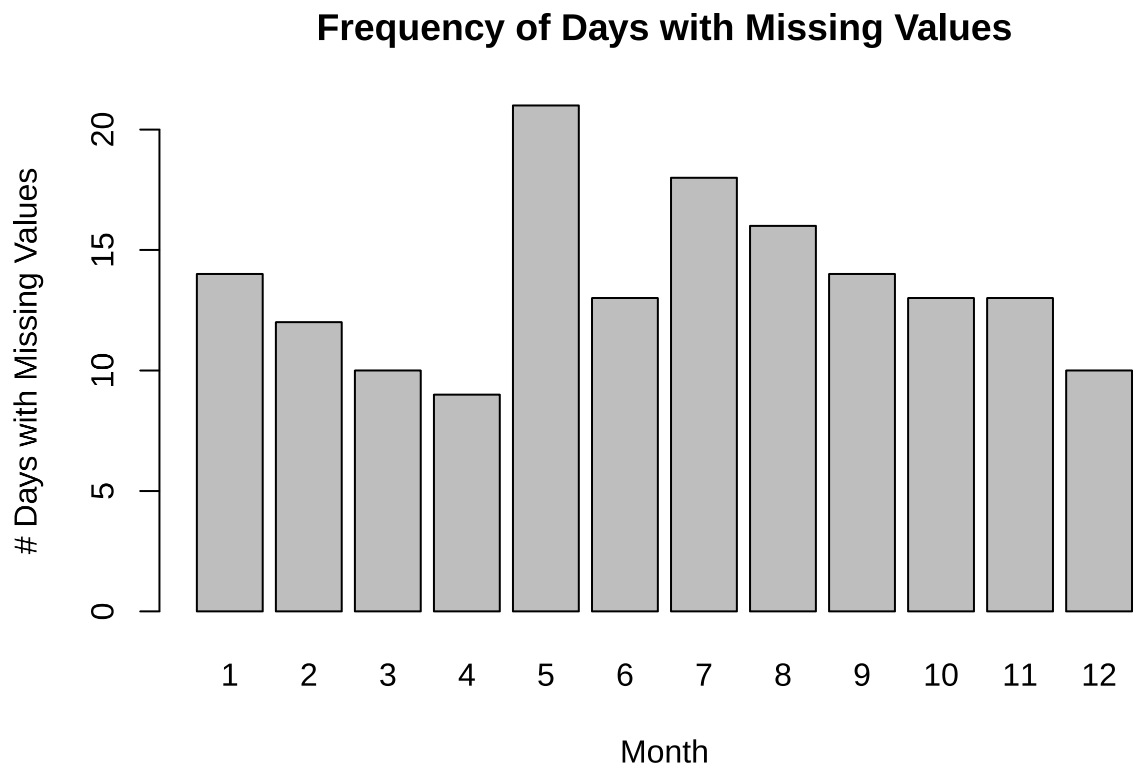
To form the first dataset, I just used *na.omit()* to clean out the rows (days) with any missing observations. For the second, I used *is.na(Fixed$dailymax) == FALSE* as a row filter to throw out the days when the variable *dailymax* is missing (*Fixed* is the name of my prepared dataset).

To determine whether the days with missing data are distributed uniformly over the twelve months, I first used *num_nas <- rowSums(is.na(Fixed))* to get a preliminary indicator of whether a day contains missing data; from there, I just filtered my dataset with *num_nas > 0* when I needed to separate days with complete data from those with missing observations. I then used a bar chart to visualize the frequency of days with missing observations across the months. From this, I would've guessed that - no - those days are not uniformly distributed across the months; it looks like the summer months tend to have more such days.

But, when I next used a chi-squared test to see how likely it is that the days with missing observations are uniformly distributed across the months, I got a p-value of .1293, which is not significant. So, in conclusion - *no* - we cannot say that the days are not uniformly distributed across the months.

I thought about checking whether the days with missing values specifically in the dailymax variable are evenly-distributed across the months, too, but there are only 11 such days. That's far too small of a dataset to reach any kind of conclusion.

```r
# no missing values
no_nas <- na.omit(Fixed)

# no missing dailymax values
no_dm_nas <- Fixed[is.na(Fixed$dailymax) == FALSE, ]

#length(Fixed[is.na(Fixed$dailymax) == TRUE, ])

num_nas <- rowSums(is.na(Fixed))

# frequency of missing values by month
plot(Fixed[num_nas > 0, ]$month, xlab = 'Month', ylab = '# Days with Missing Values', main= 'Frequency 
```



```r
# chi-squared test
chisq.test(num_nas > 0, Fixed$month)
```

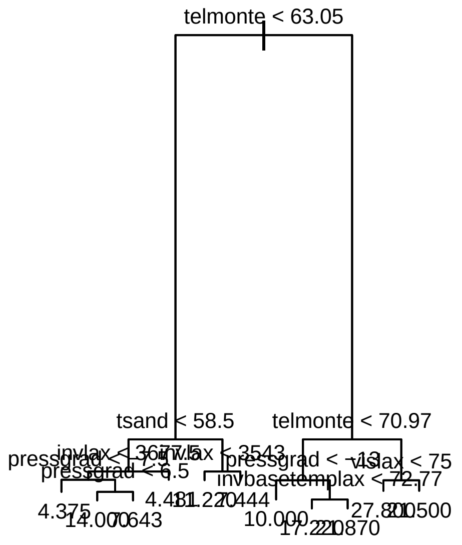
```
##
##  Pearson's Chi-squared test
##
## data:  num_nas > 0 and Fixed$month
## X-squared = 16.331, df = 11, p-value = 0.1293
```

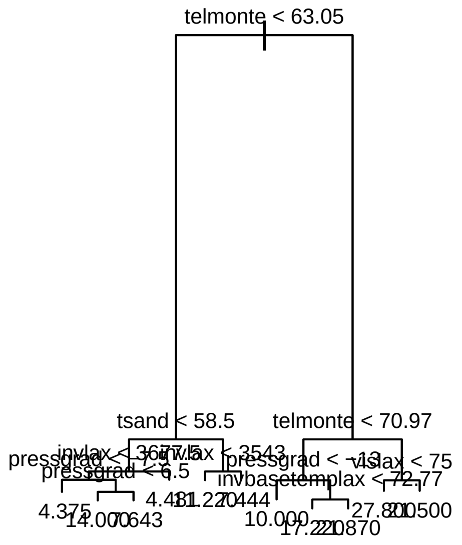
2)

I called the tree fitted from data with no missing values *tree.no_nas* and the one fitted from data with a given dailymax *tree.no_dm_nas*. The trees' corresponding plots make it pretty clear that they are the same (and they have identical residuals), but for further evidence I decided to pass both trees some generated data to see if their predictions are equivalent.

To generate this test data, I made a 12-value long sequence encompassing the range of each predictor (12 so I could easily turn the months to factors). I then passed this as a dataframe to *predict()*, using each tree as a model once. The results are below; the first 11 columns are the values fed into the predictors, the 12th and 13th are the predictions from the models, and the final column is a logical vector denoting whether the prediction from *tree.no_nas* matches that from *tree.no_dm_nas*. (I didn't output the first 9 columns for printing purposes). As can be seen, the predictions are identical. Because the trees always produce the same predictions, they are composed of the same splits <==> the trees are identical.

```r
# fit the trees
tree.no_nas <- tree(dailymax ~ .-month, data = no_nas)
tree.no_dm_nas <- tree(dailymax ~ .-month, data = no_dm_nas)

par(mfrow = c(1, 2))

# show the trees
plot(tree.no_nas)
text(tree.no_nas, pretty = 0, cex = .7)

plot(tree.no_dm_nas)
text(tree.no_dm_nas, pretty = 0, cex = .7)
```





```r
# identical residuals
extra <- summary(tree.no_nas)$residuals
extra1 <- summary(tree.no_dm_nas)$residuals
FALSE %in% c(extra == extra1)
```

```
## [1] FALSE
```

```r
# produce same predictions
proof_same <- apply(Fixed, 2, range, na.rm = TRUE)

proof_same <- data.frame(apply(proof_same, 2, function(x){seq(from = x[1], to = x[2], length.out = 12)}

names(proof_same) <- names(no_nas)
```

```r
proof_same$month <- as.factor(proof_same$month)

pred1 <- predict(tree.no_nas, proof_same)

pred2 <- predict(tree.no_dm_nas, proof_same)

proof_same$no_nas_pred <- pred1

proof_same$no_dm_nas <- pred2

proof_same$identical <- pred1 == pred2

proof_same[, 10:14]
```

```
##    invbasetemplax    vislax no_nas_pred no_dm_nas identical
## 1        27.50000   0.00000   14.000000 14.000000      TRUE
## 2        33.34182  45.45455    7.642857  7.642857      TRUE
## 3        39.18364  90.90909    7.642857  7.642857      TRUE
## 4        45.02545 136.36364    7.642857  7.642857      TRUE
## 5        50.86727 181.81818    7.642857  7.642857      TRUE
## 6        56.70909 227.27273    7.642857  7.642857      TRUE
## 7        62.55091 272.72727   11.222222 11.222222      TRUE
## 8        68.39273 318.18182   11.222222 11.222222      TRUE
## 9        74.23455 363.63636   21.866667 21.866667      TRUE
## 10       80.07636 409.09091   21.500000 21.500000      TRUE
## 11       85.91818 454.54545   21.500000 21.500000      TRUE
## 12       91.76000 500.00000   21.500000 21.500000      TRUE
```

3)

Basically, we're fitting ten separate step functions to the data and building a cross-validation function (I don't think *cv.glm()* can be made to work because our training and test values for dailymax come from different datasets). To do this, I first randomly shuffled the rows of the dataframe without any missing values in the dailymax column. I then used this shuffling to make K = 10 roughly equally-sized folds. From there, I went through a loop that:) holds out one of the folds,) builds a model that predicts the dailymax to be the mean dailymax - from the non-held-out folds - of the month of that observation,) uses this model to predict the dailymax values for the held-out data, and) adds the Mean Squared Error of this model's predictions to a running list of MSEs.

Then, to calculate the RMSE, I used the function: $RMSE = \sqrt(\frac{1}{K}\sum_{i=1}^{K} MSE_i) \approx 9.786$. So the 10-fold cross-validation estimate for the RMSE is about 9.786.

When I did the same thing with the data with no missing values, I got a RMSE of about 10.472. This seems correct; there are far fewer data points to work with in this set, so it makes sense that the fitted models produced during cross-validation aren't as accurate as those fitted above.

```r
set.seed(36)

# shuffle up the rows
shuffled <- sample(1:nrow(no_dm_nas), nrow(no_dm_nas), replace = FALSE)
```

```r
# get the random folds
folds <- round(seq(length(shuffled)/10, length(shuffled), length.out = 10), 0)

# keep track of the MSEs
MSEs <- c()
# keep track of the fold we're on
last_break <- 0
for (i in folds) {

  # take care of the hold out and training folds
  holdout <- last_break:i

  training <- shuffled[-holdout]
  train <- no_dm_nas[training, ]

  test <- no_dm_nas[-training, ]

  # the average dailymax by month for our non-held-out data
  pass <- data.frame(aggregate(test$dailymax, list(test$month), mean))
  names(pass) <- c('month', 'dailymax')

  pass$month <- as.numeric(pass$month)

  # the model
  pass.glm <- glm(dailymax ~ cut(month, 12) - 1, data = pass)

  # predictions
  preds <- predict(pass.glm, data.frame(month = as.numeric(test$month)))

  # local MSE added to list of MSEs
  SSE_sub <- (no_dm_nas[holdout, ]$dailymax - preds)^2

  MSEs <- c(MSEs, mean(SSE_sub))

  last_break <- i

}

# RMSE
sqrt(mean(MSEs))
```

```
## [1] 9.785937
```

```r
######## Same Thing with no_nas #################

# shuffle up the rows
shuffled1 <- sample(1:nrow(no_nas), nrow(no_nas), replace = FALSE)

# get the random folds
folds1 <- round(seq(length(shuffled1)/10, length(shuffled1), length.out = 10), 0)

# keep track of the MSEs
MSEs1 <- c()
# keep track of the fold we're on
```

```r
last_break1 <- 0
for (i in folds1) {

  # take care of the hold out and training folds
  holdout <- last_break1:i

  training <- shuffled1[-holdout]
  train <- no_nas[training, ]

  test <- no_nas[-training, ]

  # the average dailymax by month for our non-held-out data
  pass <- data.frame(aggregate(test$dailymax, list(test$month), mean))
  names(pass) <- c('month', 'dailymax')

  pass$month <- as.numeric(pass$month)

  # the model
  pass.glm <- glm(dailymax ~ cut(month, 12) - 1, data = pass)

  # predictions
  preds <- predict(pass.glm, data.frame(month = as.numeric(test$month)))

  # local MSE added to list of MSEs
  SSE_sub <- (no_nas[holdout, ]$dailymax - preds)^2

  MSEs1 <- c(MSEs1, mean(SSE_sub))

  last_break1 <- i

}

# RMSE
sqrt(mean(MSEs1))
```

```
## [1] 10.4716
```

4)

I fit a simple linear model. The residuals plots seem to indicate that this is fine - it doesn't look like there are any patterns, and the Q-Q plot hugs the normal line. The summary of this fitted model tells me that the following variables are significant at the $p < .05$ level: humlax, tsand, telmonte, and invlax. Furthermore, pressure is very close to being labeled significant, with a p-value of .0565.

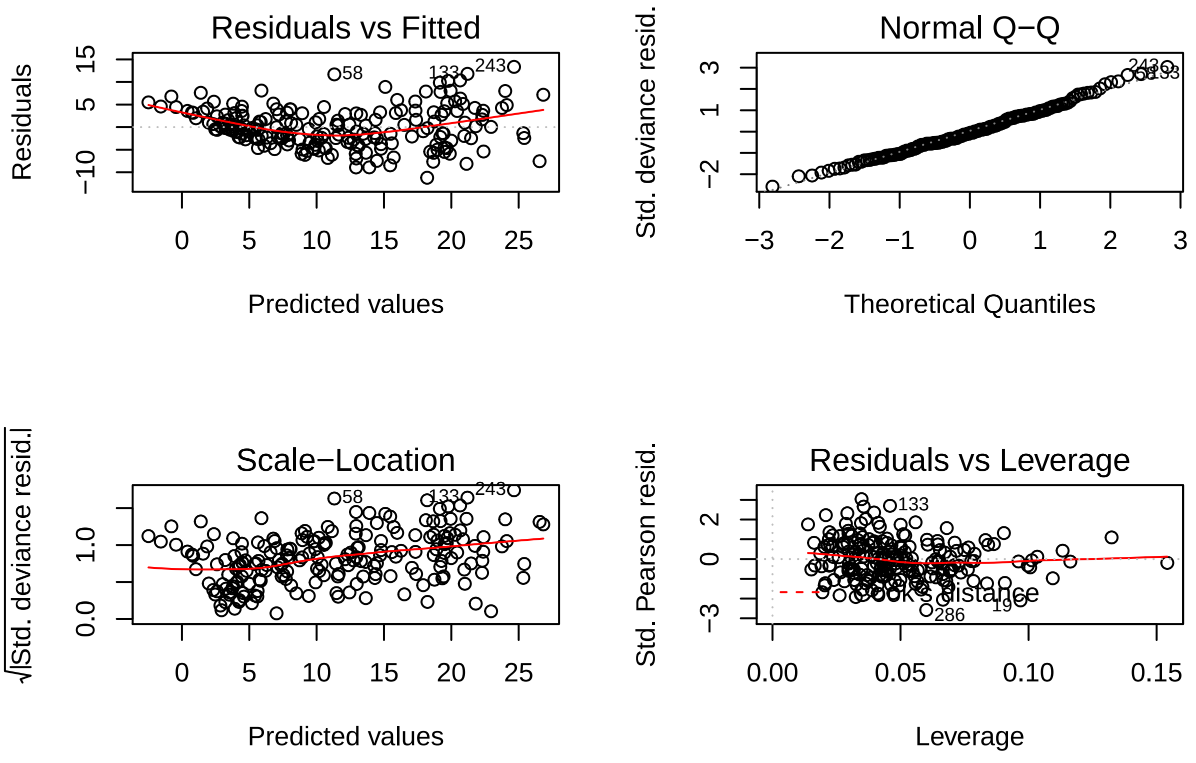
Using 10-fold cross-validation, I got a (non-bias-corrected) CV score of 21.89, which corresponds to the cross validation's estimate of the test MSE. Thus, the estimated test RMSE is about 4.57. This is understandably much lower than the estimate found in #3 that just used monthly means.

```r
set.seed(36)

# fit a linear model
glm.fit <- glm(dailymax ~ .-month, data = no_nas)
```

```r
# get the coefficients and their estimated significances
summary(glm.fit)
```

```
##
## Call:
## glm(formula = dailymax ~ . - month, data = no_nas)
##
## Deviance Residuals:
##      Min        1Q    Median        3Q       Max
## -11.2055   -2.7232   -0.2741    3.0891   13.3442
##
## Coefficients:
##                  Estimate Std. Error t value Pr(>|t|)
## (Intercept)    59.9517553 38.3286940   1.564 0.119421
## pressure       -0.0139111  0.0072511  -1.918 0.056527 .
## windlax         0.0276862  0.1741433   0.159 0.873847
## humlax          0.0808740  0.0237694   3.402 0.000812 ***
## tsand           0.1503404  0.0692994   2.169 0.031272 *
## telmonte        0.5253439  0.1247136   4.212 3.87e-05 ***
## invlax         -0.0010052  0.0003944  -2.549 0.011586 *
## pressgrad       0.0049796  0.0147772   0.337 0.736501
## invbasetemplax -0.1543882  0.1192917  -1.294 0.197140
## vislax         -0.0033951  0.0048963  -0.693 0.488883
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for gaussian family taken to be 20.08543)
##
##     Null deviance: 13549.5  on 202  degrees of freedom
## Residual deviance:  3876.5  on 193  degrees of freedom
## AIC: 1196.8
##
## Number of Fisher Scoring iterations: 2
```

```r
par(mfrow = c(2,2))
plot(glm.fit)
```



```r
# test MSE as estimated by 10-fold cross-validation
MSE_cv <- cv.glm(no_nas, glm.fit, K = 10)$delta[1]

MSE_cv
```

```
## [1] 20.89437
```

```r
# test RMSE as estimated by 10-fold cross-validation
RMSE_cv <- sqrt(MSE_cv)

RMSE_cv
```

```
## [1] 4.571036
```

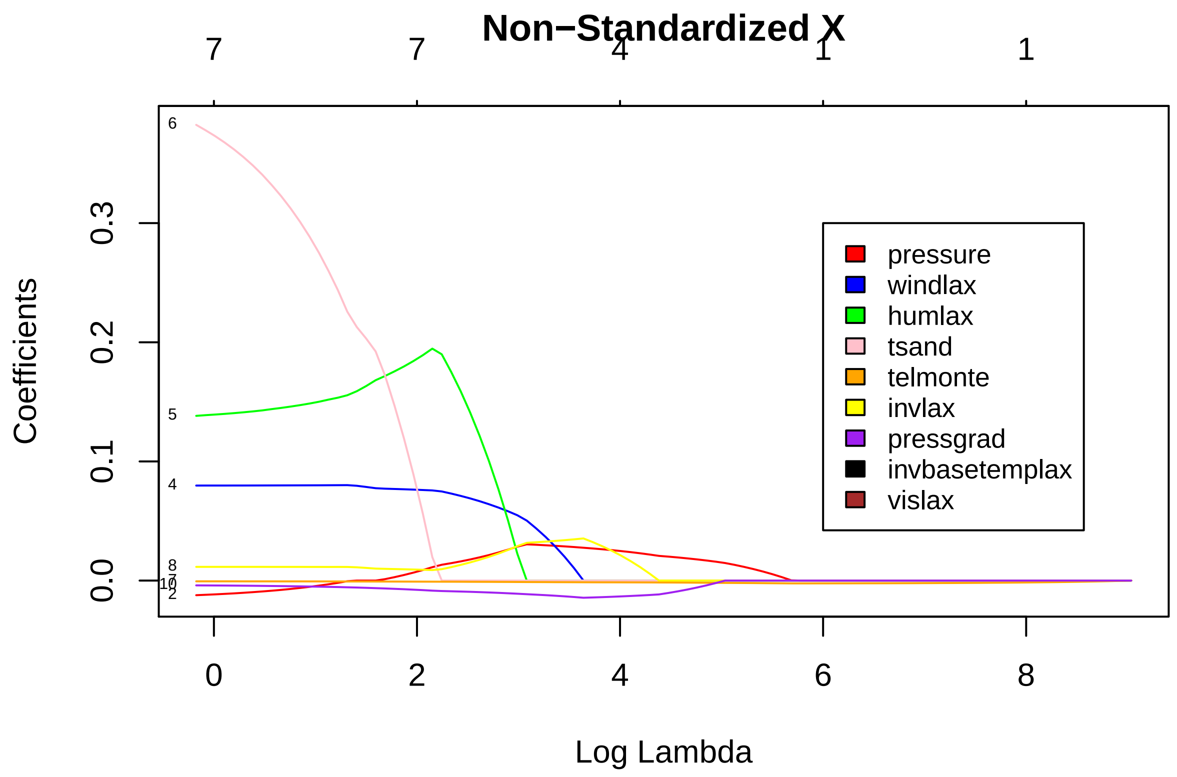
5)

To fit the lasso models, I first had to make a model.matrix with the formula from part (4) above, so as to be able to use the *glmnet()* package. After this, I fit one model with standardized predictors and one without. To find the last 5 predictors thrown out by the two models, I used the *print()* function to show the effective degrees of freedom associated with each tested lambda; there are 5 predictors left in the model the first time this equals 5. In the case of the non-standardized model, this was at $\lambda = 38.08$; for the standardized model it was at $\lambda = .7427$. Already, then, there's an indication that the standardized and non-standardized models might be quite different.

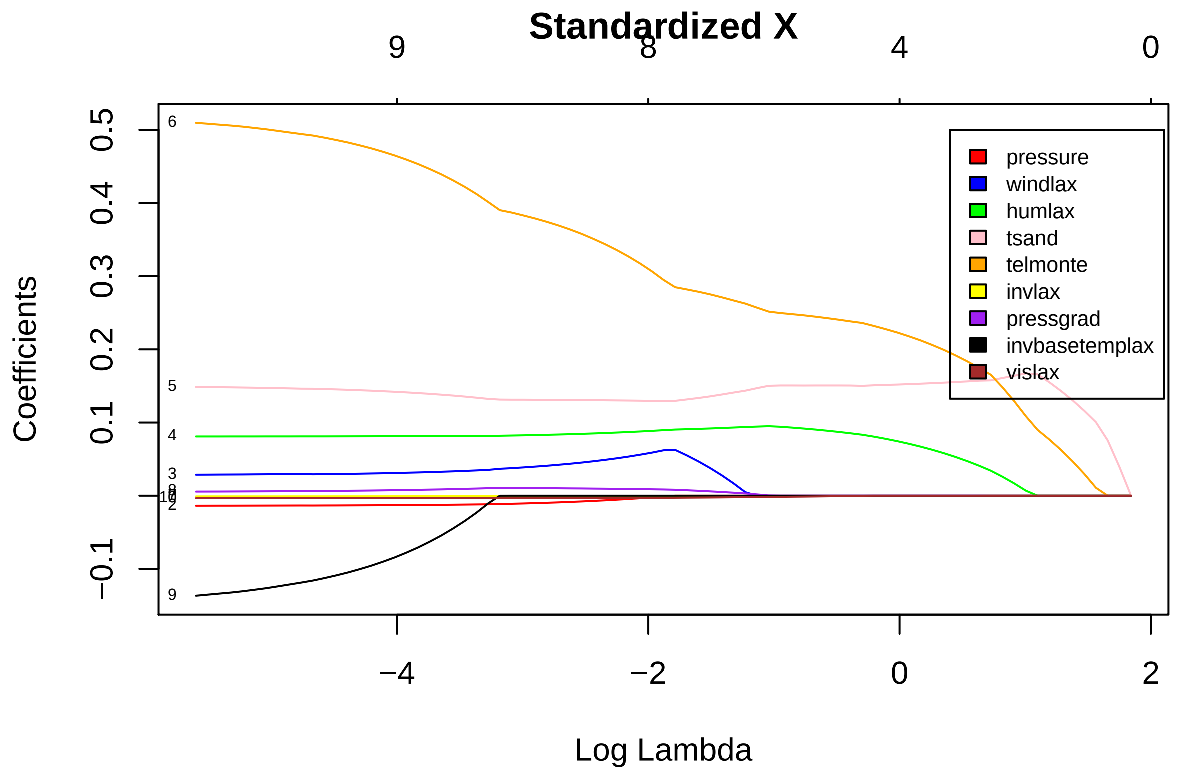
And this turned out to be correct. For the non-standardized model, the last 5 variables thrown out - that is, the 5 variables the model thought were most relevant to the response - were (in order of the last to be thrown out to the 5th from last): invlax, pressure, vislax, pressgrad, and humlax. This is fairly expected: these variables exist on much larger scales than the others, and so the non-standardized lasso model will be dominated by their variances. For the standardized model, though, the equivalent last five variables were: tsand, telmonte, humlax, invlax, and vislax. There's some overlap, but the models produced are very different. In particular, the standardized model seems to agree with the linear model just fit in part (4), which thought the 4 significant predictors were: tsand, humlax, telmonte, and invlax. Each of these were in the last 5 predictors thrown out in the standardized model, but tsand and telmonte were not in the last 5 from the non-standardized model.

The plots echo the ideas above - they're pretty different. On the bottom x-axis is the $log_2\lambda$ sequence; on the top x-axis is the number of predictors left at that level of $lambda$; and on the y-axis is the coefficient weights for each of the predictors. As can be seen, the predictors have a lot of variance between the two plots.

```r
#View(no_nas)
# get things ready for the glmnet() function
x <- model.matrix(dailymax~.-month, data = no_nas)
y <- no_nas$dailymax

# non-standardized and standardized models
lasso.mod <- glmnet(x, y, alpha = 1, standardize = FALSE)
lasso.mod.standardized <- glmnet(x, y, alpha = 1, standardize = TRUE)

# how I got the order of the last 5 coefficients to leave the models - commented so they don't print

#lasso.mod
#lasso.mod.standardized

#coef(lasso.mod, s = 7650)
#coef(lasso.mod, s = 294.8)
#coef(lasso.mod, s = 140.1)
#coef(lasso.mod, s = 73.03)
#coef(lasso.mod, s = 38.08)

#coef(lasso.mod.standardized, s = 5.75)
#coef(lasso.mod.standardized, s = 4.774)
#coef(lasso.mod.standardized, s = 2.732)
#coef(lasso.mod.standardized, s = 2.067)
#coef(lasso.mod.standardized, s = .7427)

# plots of the lasso models

plot(lasso.mod, xvar = 'lambda', label = TRUE, col = c('red','blue','green','pink','orange','yellow','pu
legend(6, .3, legend = c('pressure', 'windlax', 'humlax', 'tsand', 'telmonte', 'invlax', 'pressgrad', 'i
```



```r
plot(lasso.mod.standardized, xvar = 'lambda', label = TRUE, col = c('red','blue','green','pink','orange
legend(.4, .5, legend = c('pressure', 'windlax', 'humlax', 'tsand', 'telmonte', 'invlax', 'pressgrad',
```



6)

From the trees (which were shown to be the same, so we can just consider it to be one tree), we can see that *telmonte, tsand, invlax, pressgrad, invbasetemplax, and vislax* were the variables used. Furthermore, when looking back at the plotted tree, we see that telmonte is involved in two of the three largest splits, with tsand being involved in the other. After that, things get a bit muddled, but it looks - from the tree plot above and the output below - that tsand and telmonte account for most of the splitting after these first three, too, with pressgrad and invlax maybe being the third and fourth most important features, and vislax and invbasetemplax following shortly thereafter.

From the linear model, we remember that the model thinks *telmonte, humlax, invlax, and tsand* are significant (in order of increasing p-value). It also thinks pressure is close to significant ($p = .0565$).

We also remember that the lasso models were quite different depending on whether the x-values were standardized before fitting. The last 5 features to leave the non-standardized lasso model were (in order of the last to be thrown out to the 5th from last): *invlax, pressure, vislax, pressgrad, and humlax*; the last five thrown from the standardized model were: *tsand, telmonte, humlax, invlax, and vislax.*

In aggregate, we see that invlax is the only variable to show up in each of the 4 (5 counting both trees) models. On the other hand, invbasetemplax only shows up (briefly) in the tree model and pressure only in the non-standardized lasso model; pressgrad only pops up in the tree and non-standardized lasso model, too. Without any serious knowledge of the predictors, it's hard to say if we should standardize them before fitting a model. However, the agreement amongst the other methods is a pretty good hint that we should; and, when in doubt, it's usually a good idea to standardize the predictors. If we don't consider the non-standardized lasso model, then, there is much more overlap; each of telmonte, tsand, and invlax are in all of the 3 other models. Moreover, if we use the number of splits responsible for, p-values, and relative time that the predictor left the lasso model as quick measures of relative importance, these 3 models are close to agreement: telmonte and tsand are quite important, with invlax being third. With some quick linear model fitting, we see that leaving out all the variables except these three decreases the R^2 from .714 to .636 - so the other variables are certainly important, but it looks like our models were onto something insofar as they sort of crowd-sourced telmonte, tsand, and invlax as the three most significant features.

```r
# summaries
tree.nonas.sum <- summary(tree.no_nas)
tree.nodmnas.sum <- summary(tree.no_dm_nas)
glm.sum <- summary(glm.fit)

# features used by the trees (same)
tree.nonas.sum$used
```

```
## [1] telmonte       tsand          invlax         pressgrad
## [5] invbasetemplax vislax
## 10 Levels: <leaf> pressure windlax humlax tsand telmonte ... vislax
```

```r
tree.nodmnas.sum$used
```

```
## [1] telmonte       tsand          invlax         pressgrad
## [5] invbasetemplax vislax
## 10 Levels: <leaf> pressure windlax humlax tsand telmonte ... vislax
```

```r
#glm.sum

# show the splits
tree.no_nas
```

```
## node), split, n, deviance, yval
##       * denotes terminal node
##
##  1) root 203 13550.00 11.370
##    2) telmonte < 63.05 142  2734.00  7.268
##      4) tsand < 58.5 88  1119.00  5.614
##        8) invlax < 3677.5 36   684.80  7.250
##         16) pressgrad < -7.5 16    51.75  4.375 *
##         17) pressgrad > -7.5 20   394.90  9.550
##           34) pressgrad < 6.5 6   120.00 14.000 *
##           35) pressgrad > 6.5 14   105.20  7.643 *
##        9) invlax > 3677.5 52   271.00  4.481 *
##      5) tsand > 58.5 54   981.90  9.963
##       10) invlax < 3543 36   674.20 11.220 *
##       11) invlax > 3543 18   136.40  7.444 *
##    3) telmonte > 63.05 61  2846.00 20.930
##      6) telmonte < 70.97 38  1288.00 18.110
##       12) pressgrad < -13 5   102.00 10.000 *
##       13) pressgrad > -13 33   807.30 19.330
##         26) invbasetemplax < 72.77 18   355.10 17.220 *
##         27) invbasetemplax > 72.77 15   275.70 21.870 *
##      7) telmonte > 70.97 23   751.50 25.610
##       14) vislax < 75 15   416.40 27.800 *
##       15) vislax > 75 8   128.00 21.500 *
```

```r
# compare full model with that which just uses telmonte, tsand, and invlax
a <- lm(dailymax~.-month, data = no_nas)
b <- lm(dailymax~telmonte+tsand+invlax, data = no_nas)

summary(a)$r.squared
```

```
## [1] 0.7139027
```

```r
summary(b)$r.squared
```

```
## [1] 0.6358072
```

Part B

Preparation)

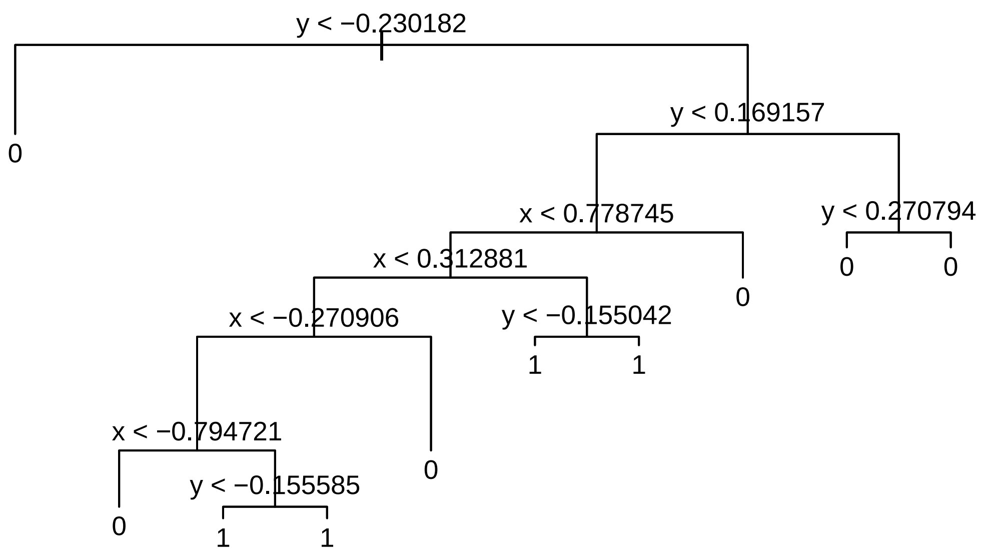
```r
make.eight = function(N, spread = .1, makeplot = T){
  # Make N points:  N/2 points in horizontal figure 8
  # N/4 points each inside the holes of the figure 8
  # spread = noise parameter
  # return data frame with coordinates x, y for each point
  # Classification variables in the data frame:
  #    charlabel =   eight   or   left   or  right
  #    label = 0 (for points on the figure 8) or = 1 (for points inside the holes)
  # plot with marked points if makeplot = T


  circ0 = runif(N/2)*2*pi
  circ = matrix(c(cos(circ0)/(1 + sin(circ0)^2),rep(-.5,N/4),rep(.5,N/4),
        cos(circ0)*sin(circ0)/(1 + sin(circ0)^2),rep(0,N/2)),ncol = 2)
  x = circ + spread*matrix(rnorm(2*N),N,2)
  y=rep(c(0,1),c(N/2,N/2))
  if(makeplot){plot(x,col = c(rep(1,N/2),rep(2,N/4),rep(3,N/4)),pch=19,
        xlab = "x1", ylab = "x2")}
  A = data.frame(x = x[,1], y = x[,2], label = as.factor(y),
        charlabel = c(rep("eight",N/2),rep("left",N/4), rep("right",N/4)))
  return(A)
}

# Try these examples:
#mydf <- make.eight(200, spread = 0)
#mydf <- make.eight(100,spread = .05)
#mydf <- make.eight(300,spread = .1, makeplot = F)

set.seed(2019)

train_data <- make.eight(2000, spread = .1, makeplot = F)
test_data <- make.eight(2000, spread = .1, makeplot = F)

#View(train_data)
#View(test_data)
```

1)

a)

The trained tree (with 10 terminal nodes) did somewhat well on both the training and test data. When used to make predictions on the training data, it produced a false positive rate of .064, a true positive rate of .94, and a total error rate of .062. On the test data, it produced a false positive rate of .088, a true positive rate of .928, and a total error rate of .08.

```r
# fit tree
tree.train <- tree(label ~ . -charlabel, data = train_data)

# show tree
plot(tree.train)

text(tree.train, pretty = 0, cex = .8)
```



```r
# predictions and confusion matrices
train_preds <- predict(tree.train, train_data, type = "class")

table(train_preds, train_data$label)
```

```
##
## train_preds   0   1
##           0 936  60
##           1  64 940
```

```r
test_preds <- predict(tree.train, test_data, type = "class")

table(test_preds, test_data$label)
```

```
##
## test_preds   0   1
##          0 912  72
##          1  88 928
```

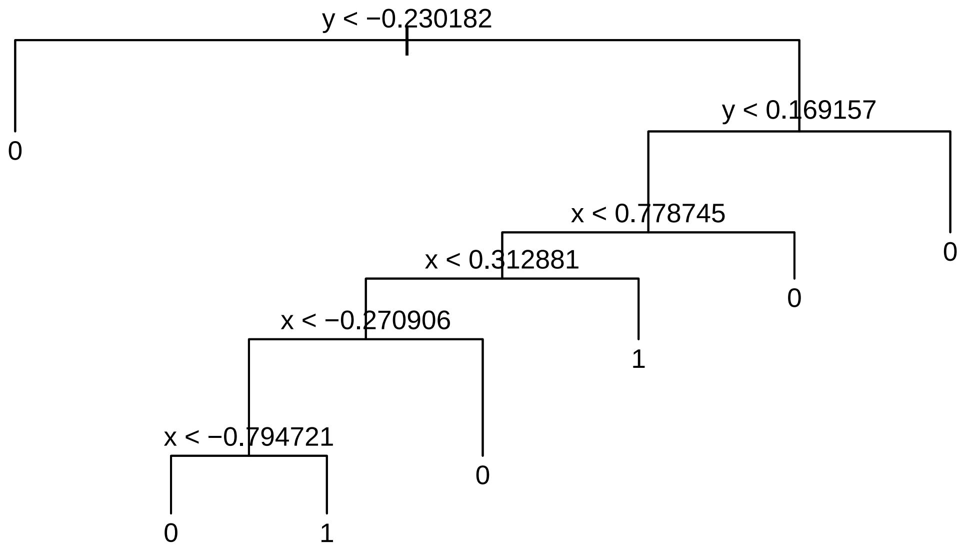
b)

One can tell this tree needs pruned because some of the splits produce terminal nodes that classify to the same label. It turns out that there are three such splits. If we throw out one terminal node for each of these useless splits, we're left with 7 terminal nodes, which is the number that I fit below.

To verify that the full and pruned tree give the same predictions for the train and test data, I checked to see if there is a 'FALSE' logical value in the vectors produced by *predict(tree.train, train_data, type = 'class') == predict(new.tree.train, train_data, type = 'class')* and *FALSE %in% (predict(tree.train, test_data, type = 'class') == predict(new.tree.train, test_data, type = 'class'))*. In both cases, the answer to this query was 'FALSE' - that is, the classes predicted by the full and pruned trees are always the same for the train and test data.

```r
# pruned tree
new.tree.train <- prune.misclass(tree.train, best = 7)
```

```
# show pruned tree
plot(new.tree.train)
text(new.tree.train, pretty = 0, cex = .8)
```



```
# are they the same predictions? yes
FALSE %in% (predict(tree.train, train_data, type = 'class') == predict(new.tree.train, train_data, type
```

```
## [1] FALSE
```

```
FALSE %in% (predict(tree.train, test_data, type = 'class') == predict(new.tree.train, test_data, type =
```

```
## [1] FALSE
```

c)

Pruning a data with two predictor variables to fewer than six nodes will almost certainly underuse one or both of the variables - one predictor will be used to make at most two splits. If we consider the case in which two splits are associated with variable 1 and two splits with variable 2, for example, the dimension along the variables are only broken into three sections; this is probably far too few sections to accurately capture the variability in the data. Visually, in the best case, you'll get a feature space that looks something like this:

As can be seen, one terminal node will not consider the other variable at all; every value in R5 will be mapped to the same class. But, if this is not a good representation of the data - if there is no way to split one of the variables into thirds such that one of the thirds contains homogeneous classes - this will produce poor results.

Below, the function didn't find a tree with 4 or 5 terminal nodes in the cost-complexity sequence, so I had to go down to 3 terminal nodes. This made the effect even more pronounced - as was foreshadowed, the variable x wasn't considered at all, which means fully half the predictor values were not used by the model. This was reflected in the results: this over-pruned model produced an error rate on the train data of .235 and an error rate on the test data of .247. These are much-higher than the tree pruned to 7 terminal nodes above.

```
# fit and show the bad tree
bad.tree <- prune.misclass(tree.train, best = 3)
plot(bad.tree)
text(bad.tree)
```

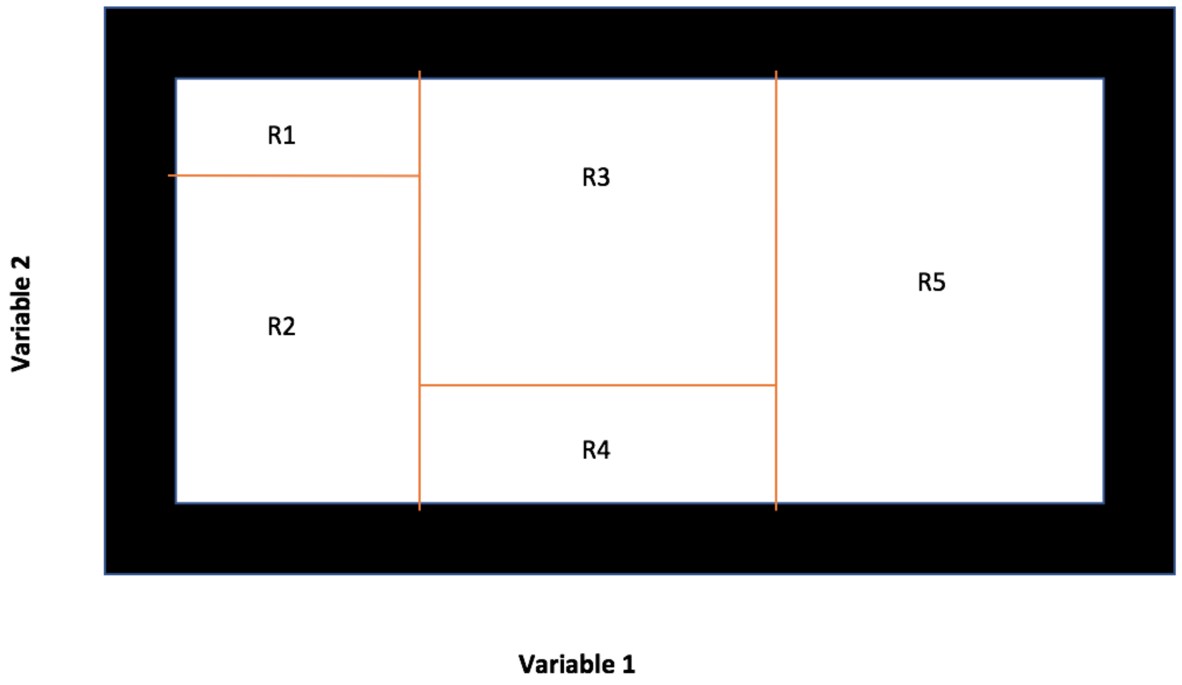
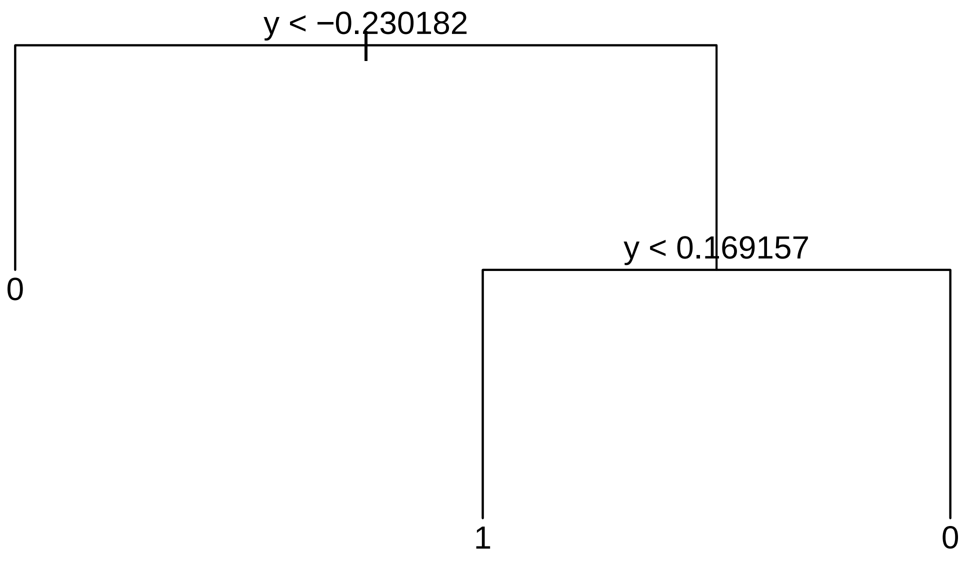


Figure 1: 5 Terminal Nodes Represented by Rx



```r
# make predictions and produce confusion matrices
bad.preds <- predict(bad.tree, train_data, type = 'class')
table(bad.preds, train_data$label)
```

```
##
## bad.preds   0   1
##         0 574  44
##         1 426 956
```

```r
bad.preds1 <- predict(bad.tree, test_data, type = 'class')
table(bad.preds1, test_data$label)
```

```
##
## bad.preds1   0   1
##          0 549  43
##          1 451 957
```

2)

a)

With an interaction depth of less than 3 - so 2, for example - there will always be 3 terminal nodes (one more than a stump). When we consider the noise parameter in the data (spread), this adds an element that a weak learner like a gbm with an interaction depth of 2 will never be able to capture perfectly. The overlap of points on and in the figure eight will throw off the first iterations of the boosting. Since these are remembered in the final prediction - boosting works by adding a shrunken version of each new tree and aggregating them - their errors will be very difficult to fully compensate for, even if we use a massive shrinkage parameter to try to make the boosting learn very quickly from (and thus overfit to) the training data. Even with the very wrong and overfit shrinkage parameter of .9 and 200 trees, I couldn't get the train error to go below .035. I assume one could get the train error a little bit lower if they generated the original data with a smaller spread, but I didn't do that, here; the ultimate conclusions would be the same, I think - with data that's as interspersed as ours, it's very hard to overcome the flaws made by overly-simple trees in the first iterations of the boosting.

This model - despite not being very good - can still overfit. As was just noted, to get the training error rate as low as I could, I ramped the shrinkage parameter up to .9, so even the final iterations on the training data were fairly important to the model. This would imply that on data that is *not* the training data - like the test data below - the model would overfit. And we see that it does - on the test data, it produced an error rate of about .075, which is more than double the train error rate of .035.

```r
library(gbm)
```

```
## Loading required package: survival

##
## Attaching package: 'survival'

## The following object is masked from 'package:boot':
##
##     aml

## Loading required package: lattice

##
## Attaching package: 'lattice'

## The following object is masked from 'package:boot':
##
##     melanoma

## Loading required package: splines

## Loading required package: parallel

## Loaded gbm 2.1.3
```

```r
set.seed(36)

# boosted tree with stumps
boost.tree <- gbm(as.numeric(label)~.-charlabel, data = train_data, n.trees = 200, distribution = 'gaus

# predictions and confusion matrices - had to deal with the changes brought about by conveting
# the label to numeric
p <- predict(boost.tree, newdata = test_data, n.trees = 200)
p[p < 1.5] <- 0
p[p > 1.5] <- 1

q <- predict(boost.tree, newdata = train_data, n.trees = 200)
q[q < 1.5] <- 0
q[q > 1.5] <- 1

table(as.factor(p), test_data$label)
```

```
##
##       0   1
##   0 905  55
##   1  95 945
```

```r
table(as.factor(q), train_data$label)
```

```
##
##       0   1
##   0 959  29
##   1  41 971
```

b)

When I ran the interaction.depth up to 15 and included the high shrinkage parameter from (a), the train error immediately went to 0. Because that shrinkage parameter causes the boosting to learn too quickly, however, this still overfits to the training data. Also, as we add to the interation depth, we run the risk of overfitting in that vein, too; a deeper interaction depth allows each of the trees to include more splits and thus fit more specifically to the training data (in fact, the test error was .092, which is even worse than above). This model was quite bad, then, because of the dual-overfitting caused by the large shrinkage parameter and too-deep interaction depth.

So the first model - fit above - would be better for new data. It overfits, but in only one way, whereas the second model just fit overfits in the shrinkage parameter *and* the interaction depth. The best model, then, for new data would be one with a proper depth *and* a normal shrinkage parameter. This would sacrifice some accuracy on the training data, but would theoretically be much better for new, test data. When I did that with a shrinkage parameter of .01 (and 2000 trees to accomodate for the much smaller shrinkage), the train error went up to .0485, but the test error went down to .0655; this still isn't great, but it's an improvement. This, then, is the starting point for a model I'd use for new data (assuming it was also generated with the make.eight function).

If this option weren't available, thoigh, I'd prefer - as mentioned above - the first to the second model, because it only overfits with the shrinkage parameter and not the interaction depth.

```r
# boosted tree with a deeper interaction depth
boost.tree.better <- gbm(as.numeric(label)~.-charlabel, data = train_data, n.trees = 200, distribution =

# predictions and confusion matrices
p <- predict(boost.tree.better, newdata = test_data, n.trees = 200)
p[p < 1.5] <- 0
p[p > 1.5] <- 1

q <- predict(boost.tree.better, newdata = train_data, n.trees = 200)
q[q < 1.5] <- 0
q[q > 1.5] <- 1

table(as.factor(p), test_data$label)
```

```
##
##       0   1
##   0 895  79
##   1 105 921
```

```r
table(as.factor(q), train_data$label)
```

```
##
##        0    1
##   0 1000    0
##   1    0 1000
```

```r
# boosted tree with normal shrinkage
boost.tree.best <- gbm(as.numeric(label)~.-charlabel, data = train_data, n.trees = 2000, distribution =

# predictions and confusion matrices
p <- predict(boost.tree.best, newdata = test_data, n.trees = 2000)
p[p < 1.5] <- 0
p[p > 1.5] <- 1

q <- predict(boost.tree.best, newdata = train_data, n.trees = 2000)
q[q < 1.5] <- 0
q[q > 1.5] <- 1

table(as.factor(p), test_data$label)
```

```
##
##       0   1
##   0 918  51
##   1  82 949
```

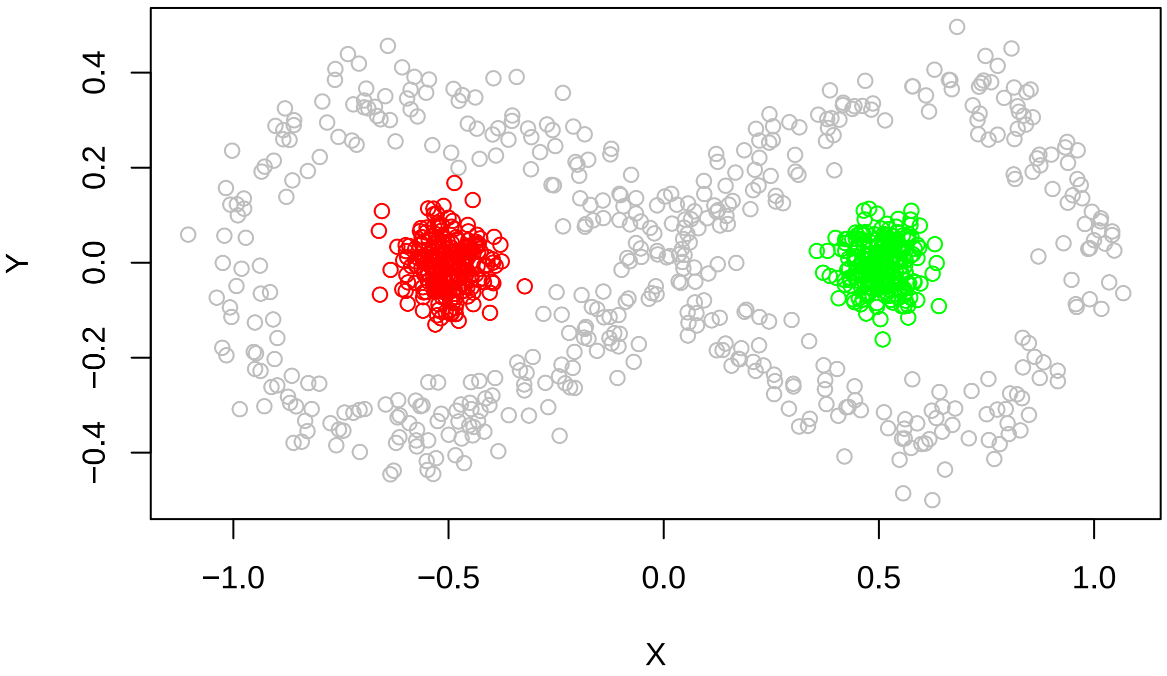
```r
table(as.factor(q), train_data$label)
```

```
##
##       0   1
##   0 948  44
##   1  52 956
```

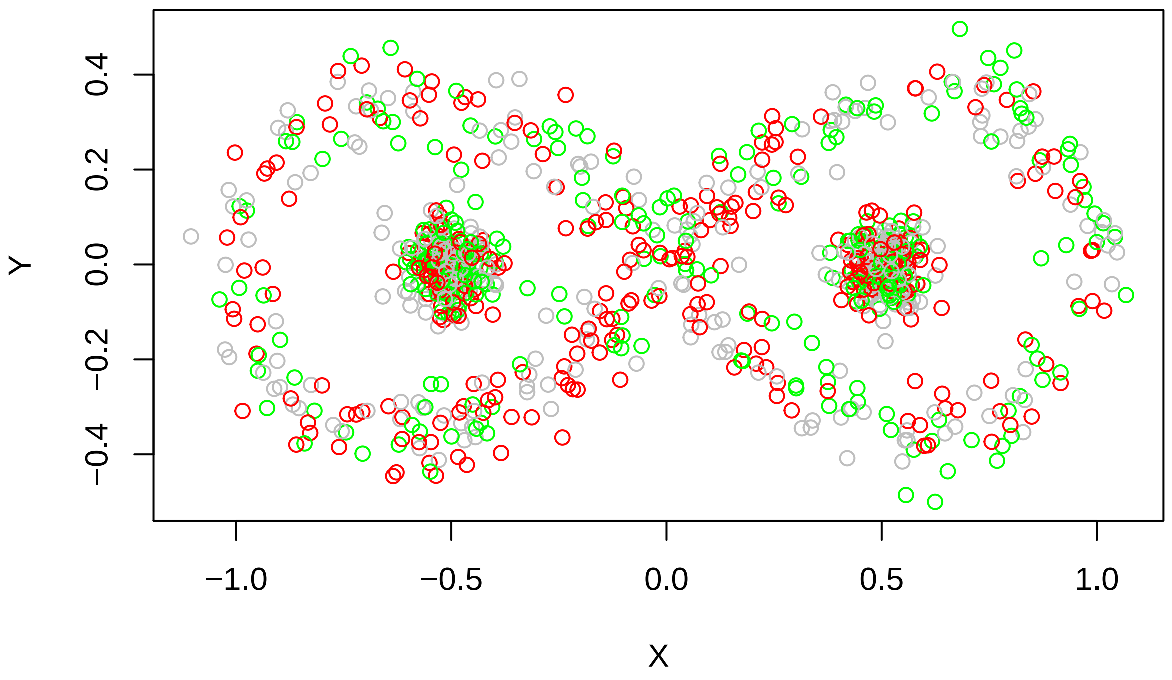
3)

For validation, I'll be using a scatterplot with the points colored by cluster label. If I use a muted color - like gray - for points on the eight and two vibrant colors for the points inside the eights, it should be easy to tell visually if the clustering algorithm got things correct. Conversely, if things go wrong and some of the vibrant-colored points end up on the figure eight, they're sure to stand out against their gray counterparts. This is illustrated below: in the first scatterplot I colored the points correctly, while in the second I colored them at random; it's pretty clear which is which.

```
set.seed(2019)

last_data <- make.eight(1000, spread = .05, makeplot = F)
last_data <- last_data[, -3]

plot(last_data$x, last_data$y, col = c('gray', 'red', 'green')[last_data$charlabel], xlab = 'X', ylab =
```



```
plot(last_data$x, last_data$y, col = c('gray', 'red', 'green')[sample(c(1, 2, 3), 1000, replace = TRUE)]
```



a)

To use my chosen method here, I colored all the produced cluster that did not contain a point inside the eight a uniform 'gray'. This makes it easier to tell if one of the red or green points made its way onto the figure eight.

For hierarchical clustering with complete linkage and K = 10, the method got close, but didn't ultimately capture the structure of the data. As can be seen, both bubbles of points inside the figure eight leaked out onto the eight in one region. Either there weren't enough clusters or there is too much noise in the data, then, to separate the (theoretically) 8 groups on the figure eight from the 2 inside it.

For hierarchical clustering with complete linkage and K = 20, the method perfectly captured the data. With 20 groups, it would seem, the method learns to differentiate between the 2 groups that belong inside the figure eight from those that don't.

Hierarchical clustering with single linkage performed very poorly for both K = 10 and K = 20. As can be seen, both runs matched perfectly the group inside the right bubble of the figure eight, but that in the left bubble leaked out and overtook the outer rungs. With single linkage, then, it would appear that there is too much noise - some points in the left bubble were too close to some points that should be classified as on the figure eight - and this ruined the entire method.

```
hc.complete <- hclust(dist(last_data), method = 'complete')
```

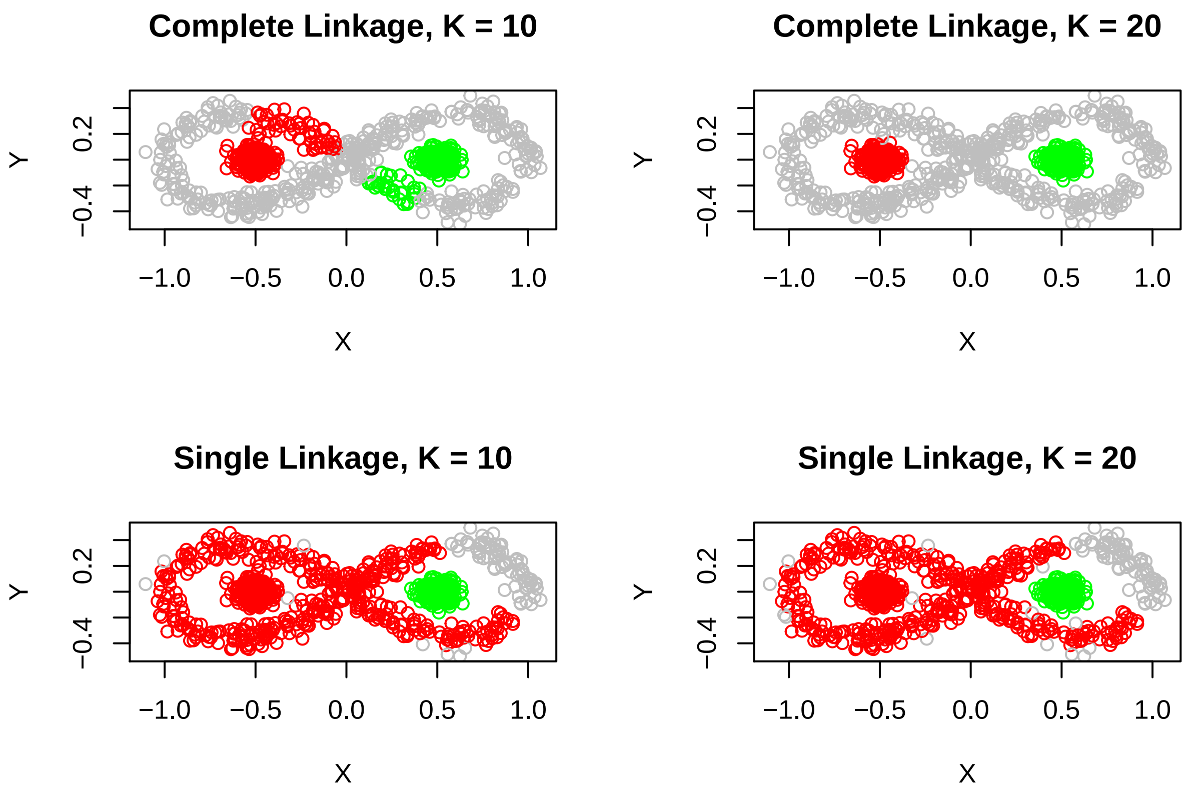
```
## Warning in dist(last_data): NAs introduced by coercion
```

```
hc.single <- hclust(dist(last_data), method = 'single')
```

```
## Warning in dist(last_data): NAs introduced by coercion
```

```
comp10 <- cutree(hc.complete, k = 10)
comp20 <- cutree(hc.complete, k = 20)
```

```
sing10 <- cutree(hc.single, k = 10)
sing20 <- cutree(hc.single, k = 20)

#comp10[501] ---> equals 2
#comp10[751] ---> equals5
#comp20[501] ---> equals 19
#comp20[751] ---> equals 20

#sing10[501] ---> equals 1
#sing10[751] ---> equals 10
#sing20[501] ---> equals 1
#sing20[751] ---> equals 10

par(mfrow = c(2,2))

cols1 <- c('gray', 'red', 'gray', 'gray', 'green', rep('gray', 5))
cols2 <- c(rep('gray', 18), 'red', 'green')

plot(last_data$x, last_data$y, col = cols1[comp10], xlab = 'X', ylab = 'Y', main = 'Complete Linkage, K
plot(last_data$x, last_data$y, col = cols2[comp20], xlab = 'X', ylab = 'Y', main = 'Complete Linkage, K

cols3 <- c('red', rep('gray', 8), 'green')
cols4 <- c('red', rep('gray', 18), 'green')

plot(last_data$x, last_data$y, col = cols3[sing10], xlab = 'X', ylab = 'Y', main = 'Single Linkage, K =
plot(last_data$x, last_data$y, col = cols4[sing20], xlab = 'X', ylab = 'Y', main = 'Single Linkage, K =
```

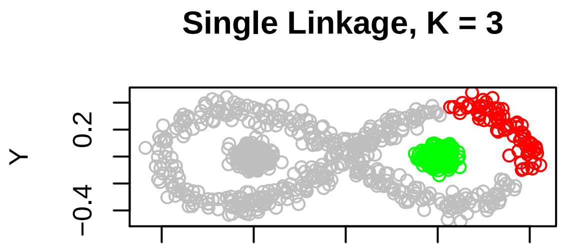


b)

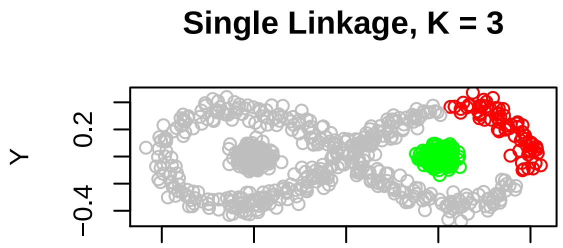
From the output below, it looks like the threshold for single linkage clustering with K = 3 to perform perfectly on the data is a spread of .041. If sllightly larger values are used, there's some separation in the top right of the plot that is actually greater than the separation between the left bubble inside the figure eight and the boundary of the closest rung. Because single linkage clustering always groups the closest clusters based on the minimum distance between any two points, this small difference was just enough for the algorithm to call that patch in the top right of the plot the third cluster, rather than the left bubble inside the figure eight. As mentioned, setting the spread to .041 in the data-generating process seems to remedy this.

```r
par(mfrow = c(2,2))
for (i in rev(seq(.041, .043, .001))) {
  set.seed(2019)
  dat <- make.eight(1000, spread = i, makeplot = F)
  dat <- dat[, -3]

  hc.single <- hclust(dist(dat), method = 'single')

  sing3 <- cutree(hc.single, k = 3)

  cols <- c('gray', 'red', 'green')

  text <- as.character(i)

  plot(dat$x, dat$y, col = cols[sing3], xlab = c('Spread: ', text), ylab = 'Y', main = 'Single Linkage,
}
```

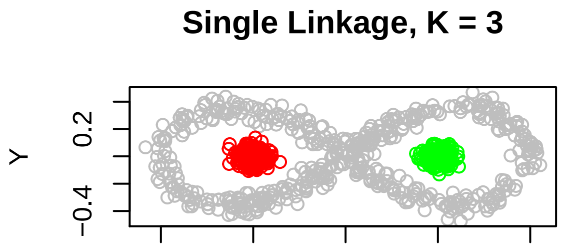
```
## Warning in dist(dat): NAs introduced by coercion

## Warning in dist(dat): NAs introduced by coercion

## Warning in dist(dat): NAs introduced by coercion
```



Spread:
0.043



Spread:
0.042



Spread:
0.041