ML Elective Project - (Mostly) decision trees

William Swain, MD

2025-03-15

In this exercise, we will apply various machine learning methods to a dataset that I have previously used for a research project. As a forewarning, given the limitations of this dataset none of this is going to yield any meaningful conclusions and the purpose of this is for learning. For this project, I used the book Machine Learning with R which is linked here (https://www.amazon.com/Machine-Learning-cleansing-modeling-tidyverse/dp/1801071322).

We previously published a paper about predictors of sustained ventricular tachycardia in patients with wild-type transthyretin cardiac amyloidosis (attr-wt CA). The paper is linked here (https://pubmed.ncbi.nlm.nih.gov/39506467/). To summarize, we looked at patients with attr-wt CA and that had a holter monitor that was completed as a part of routine clinical care. We found that the presence of non-sustained ventricular tachycardia (NSVT) and the percentage of ventricular ectopy (%VE) are variables on holter monitoring that may help predict correlate with having sustained ventricular tachycardia in the future.

Somewhat surprising was that we found that %VE also correlated with mortality, and this was across a variety of cutoffs for %VE. We ended up making Figure 5 with a cutoff of %VE = 0.5% as that was the median of the dataset.

Regardless, we will use this data and different machine learning techniques to analyze it and see what we can learn.

We start by cleaning up the memory, and loading some packages and the data.

```
rm(list=ls(all=T))
options(stringAsFactors = F)
options("scipen" = 100, "digits" = 4)
library(ggpubr)
library(descr)
library(survival)
library(survminer)
library(tidyverse)
library(knitr)
setwd("/Users/wills/Documents - PC/R projects/AI/AI elective project/")
ansvt <- read.csv("Copy of NSVT_WS_no_identifiers.csv", na.strings = c("", "NA"))
```

Background

The paper has a whole table with univariate cox regression predictors, which I will omit here but we did report two (very small) multivariate cox regression models, as shown below. These are models for the presence of NSVT and %VE, and adjusted for age and New York Heart Association (NYHA) class 3 or 4

heart failure. Since the presence of NSVT and %VE are not necessarily mutually exclusive, we reported them in separate.

```
# For NSVT
multivar_a <- coxph(Surv(time = ansvt[,61], event = ansvt[,54]) ~ ansvt$Age +
                  ansvt$NSVT + nyha34)
summary(multivar_a)
```

```
## Call:
## coxph(formula = Surv(time = ansvt[, 61], event = ansvt[, 54]) ~
##     ansvt$Age + ansvt$NSVT + nyha34)
##
##   n= 217, number of events= 11
##
##               coef exp(coef) se(coef)     z Pr(>|z|)
## ansvt$Age  -0.0143    0.9858   0.0317 -0.45    0.652
## ansvt$NSVT  2.3726   10.7252   1.0773  2.20    0.028 *
## nyha34      1.7876    5.9750   0.7864  2.27    0.023 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
##            exp(coef) exp(-coef) lower .95 upper .95
## ansvt$Age      0.986     1.0144     0.926      1.05
## ansvt$NSVT    10.725     0.0932     1.298     88.60
## nyha34         5.975     0.1674     1.279     27.91
##
## Concordance= 0.867  (se = 0.069 )
## Likelihood ratio test= 12.6  on 3 df,   p=0.006
## Wald test            = 8.48  on 3 df,   p=0.04
## Score (logrank) test = 11.4  on 3 df,   p=0.01
```

```
# For VE
multivar_b <- coxph(Surv(time = ansvt[,61], event = ansvt[,54]) ~ ansvt$Age +
                  ansvt$VE.. + nyha34)
summary(multivar_b)
```

```
## Call:
## coxph(formula = Surv(time = ansvt[, 61], event = ansvt[, 54]) ~
##     ansvt$Age + ansvt$VE.. + nyha34)
##
##   n= 217, number of events= 11
##
##                coef exp(coef) se(coef)     z Pr(>|z|)
## ansvt$Age  -0.00672   0.99330  0.03243 -0.21   0.8357
## ansvt$VE..  0.10551   1.11128  0.03439  3.07   0.0022 **
## nyha34      1.18200   3.26088  0.73183  1.62   0.1063
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
##            exp(coef) exp(-coef) lower .95 upper .95
## ansvt$Age      0.993      1.007     0.932      1.06
## ansvt$VE..     1.111      0.900     1.039      1.19
## nyha34         3.261      0.307     0.777     13.69
##
## Concordance= 0.916  (se = 0.025 )
## Likelihood ratio test= 10.8  on 3 df,   p=0.01
## Wald test            = 16.2  on 3 df,   p=0.001
## Score (logrank) test = 23.8  on 3 df,   p=0.00003
```

Both models have a good concordance - 0.87 and 0.92, respectively. Although, these models are via cox regression (modeling time to event and not just the presence of an event), they are reasonable enough benchmarks to aim for in machine learning practice.

Cleaning data and initial trials for modeling

We'll start with what I think is a reasonable starting point of variables from the big dataset. We will also change some of them to factors.

```
newvars <- ansvt %>%
  select(c("Age", "Sex", "EF", "IVST",
           "PWT", "LAVI", "BB", "CCB", "AA", "Hs.cTnT",
           "NYHA", "NT.proBNP","Mayo.stage",
           "Syncope","NSVT.1","Runs.per.monitoring.period",
           "Longest","VE..","Atrial.fibrillation",
           "VT_VF_WS", "Mortality")
         ) %>%
  mutate(across(c(Sex, BB, CCB, AA,
                  Syncope, NSVT.1,
                  Atrial.fibrillation,
                  VT_VF_WS, Mortality),
                as.factor)
  )
colSums(is.na(newvars))
```

```
##                        Age                        Sex
##                          0                          0
##                         EF                       IVST
##                          0                          3
##                        PWT                       LAVI
##                          3                          7
##                         BB                        CCB
##                          0                          0
##                         AA                    Hs.cTnT
##                          0                         12
##                       NYHA                  NT.proBNP
##                          5                          1
##                 Mayo.stage                    Syncope
##                         13                          0
##                     NSVT.1 Runs.per.monitoring.period
##                          0                          0
##                    Longest                       VE..
##                          0                          0
##        Atrial.fibrillation                   VT_VF_WS
##                          0                          0
##                  Mortality
##                          0
```

Note that because some data was missing, we'll impute the median of these variables. Having NAs will not help us down the road. While Cox regression modeling, what we used in the paper, can ignore NAs some (not all) of the machine learning techniques we'll use cannot use NA values.

We'll impute the median of these columns for all of the missing NA values.

```
 newvars <- newvars %>%
mutate(
  IVST = if_else(is.na(IVST), median(IVST, na.rm = TRUE), IVST),
  PWT = if_else(is.na(PWT), median(PWT, na.rm = TRUE), PWT),
  LAVI = if_else(is.na(LAVI), median(LAVI, na.rm = TRUE), LAVI),
  Hs.cTnT = if_else(is.na(Hs.cTnT), median(Hs.cTnT, na.rm = TRUE), Hs.cTnT),
  NYHA = if_else(is.na(NYHA), median(NYHA, na.rm = TRUE), NYHA),
  NT.proBNP = if_else(is.na(NT.proBNP), median(NT.proBNP, na.rm = TRUE), NT.proBNP),
  Mayo.stage = if_else(is.na(Mayo.stage), median(Mayo.stage, na.rm = TRUE), Mayo.stage)
)
```

To show the limitations of our dataset, I want to show what happens when we try to make a decision tree looking for sustained ventricular arrhythmia as an outcome. I will skip the code for now, but show the result below:

```
CrossTable(test_data$VT_VF_WS, p,
             prop.chisq = FALSE, prop.c = FALSE, prop.r = FALSE,
             dnn = c('actual mortality', 'predicted mortality'))
```

```
##    Cell Contents
## |-------------------------|
## |                       N |
## |         N / Table Total |
## |-------------------------|
##
## ================================
##                     predicted mortality
## actual mortality        0   Total
## --------------------------------
## 0                      50      50
##                     0.926
## --------------------------------
## 1                       4       4
##                     0.074
## --------------------------------
## Total                  54      54
## ================================
```

The table shows that there are no predictions as “1” for the outcome (sustained ventricular arrhythmia). Unfortunately, given that only 11 patients had a sustained ventricular arrhythmia (and only 4 in the test dataset) almost all machine learning models will predict that this outcome won’t happen. I.E. if 5% of patients had the event, by predicting the event will never happen the models will be 95% accurate. This would be nearly impossible to beat with such a small dataset.

We’re going to abandon looking at predictors of sustained ventricular arrhythmias and focus on mortality instead.

Since we have 217 patients, we will use 163 patients as a training sample (about 75%) and 54 patients for testing. I should also note that we will use all of the variables in the newvars dataset, as denoted by ~. If we were selecting other variables, the formula would look like: Output ~ var_A + var_B + ...

Lastly, we are going to cheat and use sustained VT_VF as a predictor variable for mortality. This variable was collected in followup so this is not a “baseline” variable. Trust me when I say that excluding it as a variable will make this exercise much more difficult.

We set a list of 163 random numbers as a test dataset (75% of the dataset), and make datasets for testing and training.

```
# set pseudorandom number generator seed
set.seed(900)
train_sample <- sample(217,163)
# test and train datsets
  train_data <- newvars[train_sample,] # 163 items long
  test_data <- newvars[-train_sample,] # 54 items long
table(train_data$Mortality)
```

```
##
##   0   1
## 131  32
```

```
table(test_data$Mortality)
```

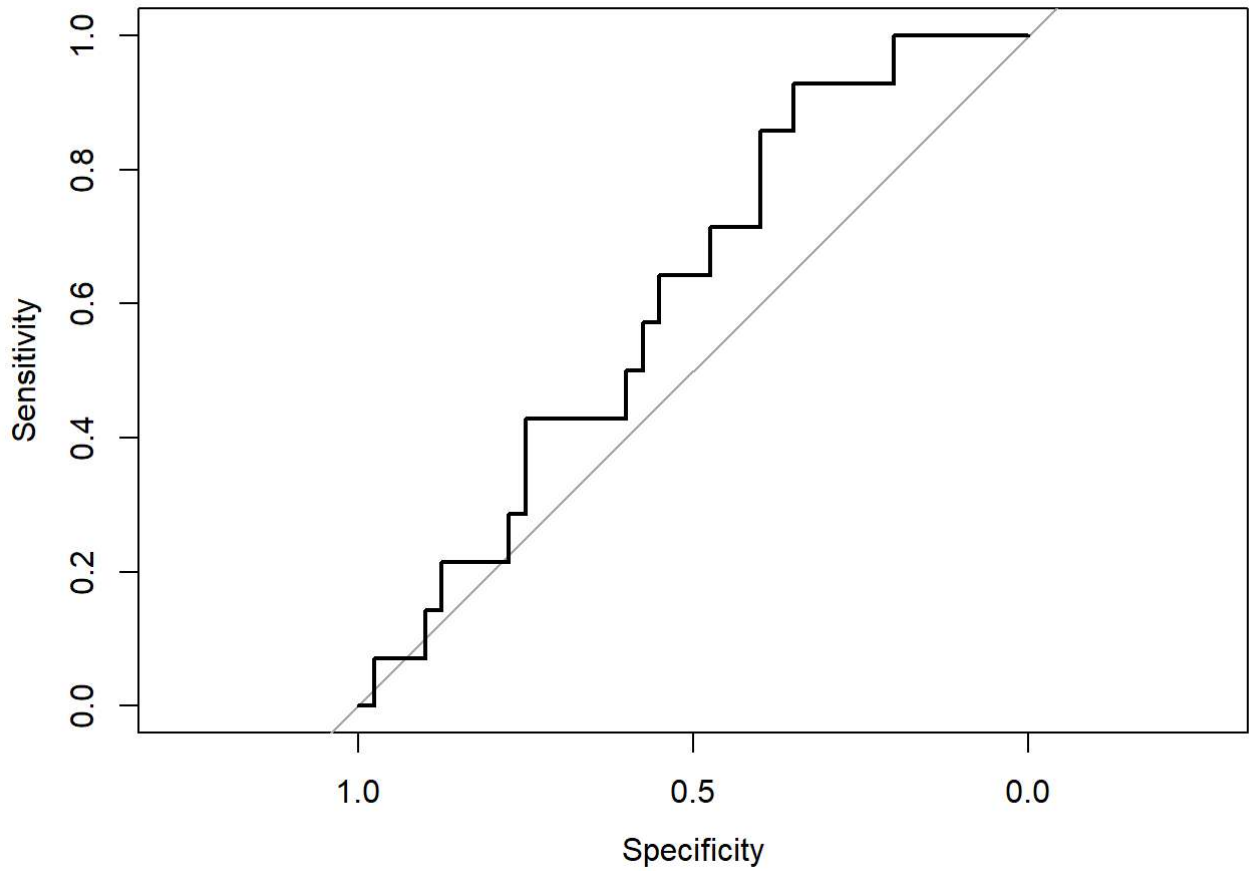
```
##
##  0  1
## 40 14
```

Note that in our training data, 32 patients died (20%) and in our test data, 14 died (26%). Therefore, if we predicted in our test data that everyone died, we would be 74% correct. We are barely going to beat this from here on out.

Just to show a baseline, lets look at how badly a logistic regression model works on our data.

It should be noted that we're not really going to be able to beat this. This is the nature of our dataset, and this is a good word of caution. Machine learning methods work well for big datasets with complex relationships. The simpler the dataset is, the more "basic" and "traditional" statistical methods will win out.

```
# logistic regression
set.seed(100)
simple_model <- glm(Mortality ~., family = binomial, data = train_data)
p <- predict(object = simple_model, newdata = test_data, type = "response")
library(pROC)
roc_curve_logistic_regression <- roc(response = test_data$Mortality, predictor = p)
plot(roc_curve_logistic_regression)
```




```
auc(roc_curve_logistic_regression)
```

```
## Area under the curve: 0.612
```

Our area under the curve is just greater than 60% which is barely better than random chance alone! Now, there's probably ways we can improve the logistic model and find variables to throw out, but we will move onward.

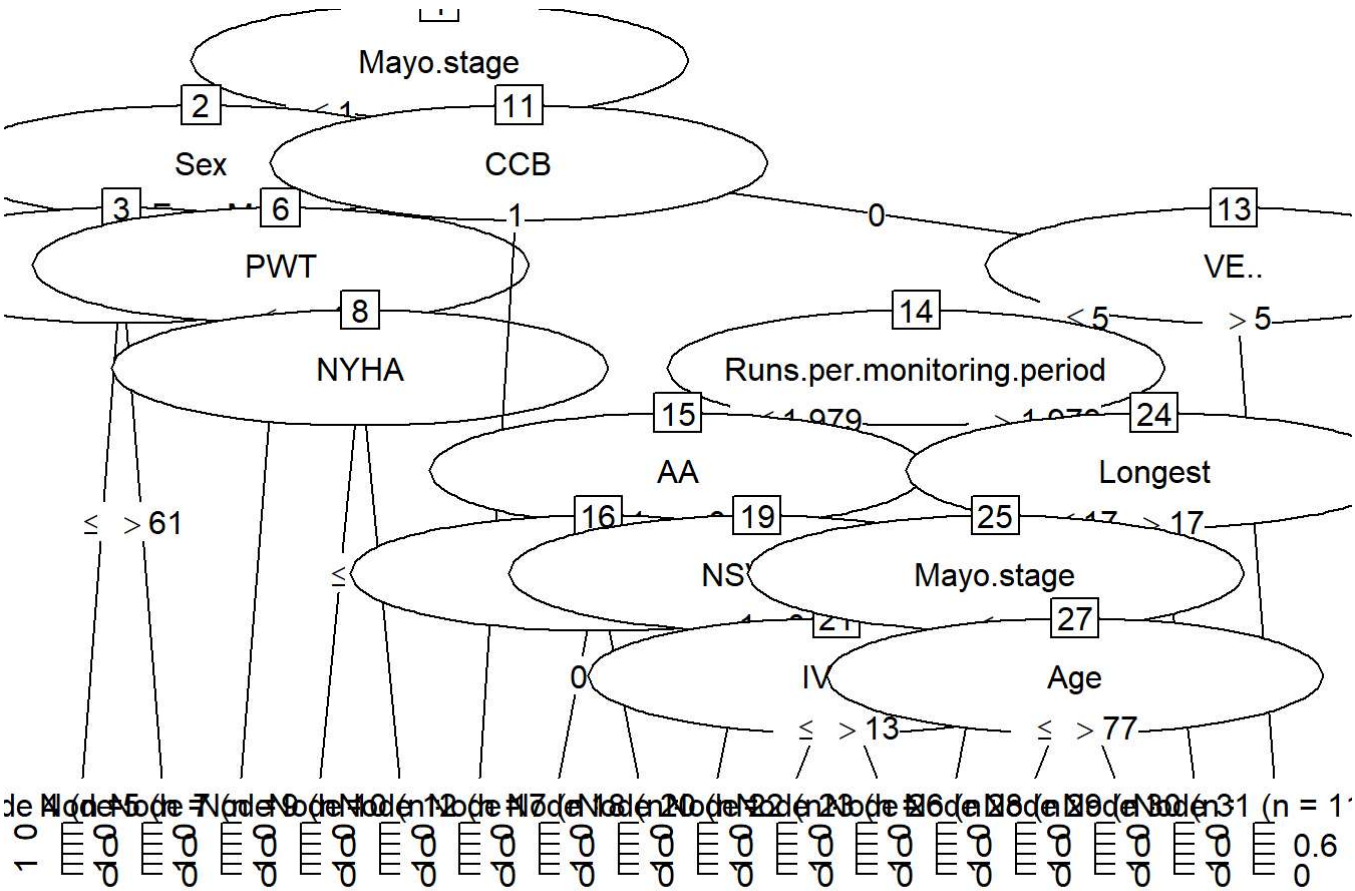
Decision trees

We'll use the c5.0() function to make decision trees. This will attempt to find an "algorithm" based on input we give to predict mortality. Decision trees divide the dataset into smaller and smaller subsets until there is only one class left, and the results is a "tree" model that shows how to get to a prediction

```
# Make the model
library(gmodels)
set.seed(50)
amyloidmod <- C5.0(as.factor(Mortality) ~., data = train_data)
amyloidmod
```

```
##
## Call:
## C5.0.formula(formula = as.factor(Mortality) ~ ., data = train_data)
##
## Classification Tree
## Number of samples: 163
## Number of predictors: 20
##
## Tree size: 16
##
## Non-standard options: attempt to group attributes
```

```
plot(amyloidmod)
```



You can see with the small screen that our tree model is complicated. Our tree is 16 decisions deep and this is tough to view on the screen. It starts with the mayo stage (which is based on troponin and BNP values), and next looks at sex, etc.

```
p <- predict(object = amyloidmod, newdata = test_data)
CrossTable(test_data$Mortality, p,
             prop.chisq = FALSE, prop.c = FALSE, prop.r = FALSE,
             dnn = c('actual mortality', 'predicted mortality'))
```

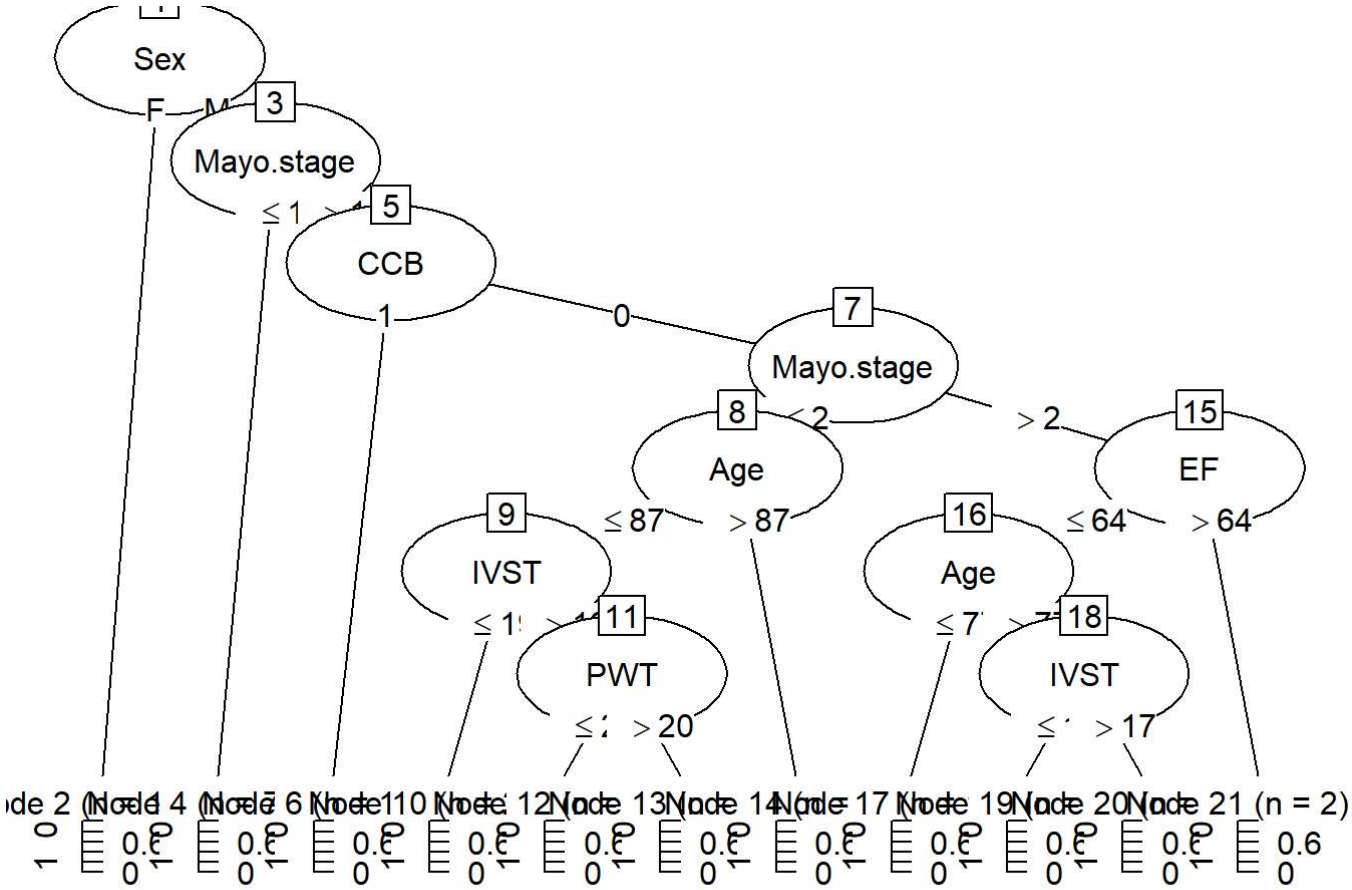
```
##
##
##    Cell Contents
## |-------------------------|
## |                       N |
## |         N / Table Total |
## |-------------------------|
##
##
## Total Observations in Table:  54
##
##
##                  | predicted mortality
## actual mortality |         0 |         1 | Row Total |
## -----------------|-----------|-----------|-----------|
##                0 |        34 |         6 |        40 |
##                  |     0.630 |     0.111 |           |
## -----------------|-----------|-----------|-----------|
##                1 |        12 |         2 |        14 |
##                  |     0.222 |     0.037 |           |
## -----------------|-----------|-----------|-----------|
##     Column Total |        46 |         8 |        54 |
## -----------------|-----------|-----------|-----------|
##
##
```

Our model predicts the correct outcome 67% of the time. This is worse than just guessing that everyone would die!

Alot of this is limited by our small amount of data. Again, the point here is to learn and not to make novel observations.

Just to show an example of a tree that can actually fit on this screen, here is a model with much less variables.

```
# Make the model
set.seed(50)
amyloidmod <- C5.0(as.factor(Mortality) ~ Mayo.stage + Sex + EF + CCB + PWT + NYHA + Age
+ IVST + VT_VF_WS, data = train_data)
plot(amyloidmod)
```



Boosting

Next, we'll use boosting to improve our decision tree. Boosting is a method that makes several weak decision trees sequentially, improving over each iteration. It starts by making a decision tree, and notes what data points were misclassified. These misclassified points are given a higher weight of importance on the next round, and another tree is made. Once all trees are done, they are combined to weighted vote. Boosting will make performance be often better and certainly no worse than the best model of the ensemble.

We'll boost 10 times - so the maximum number of trees that will be made is 10 here.

```
# boost 10x
set.seed(400)
amyloidmod_10x <- C5.0(as.factor(Mortality) ~., data = train_data,trials = 10)
amyloidmod_10x
```

```
##
## Call:
## C5.0.formula(formula = as.factor(Mortality) ~ ., data = train_data, trials = 10)
##
## Classification Tree
## Number of samples: 163
## Number of predictors: 20
##
## Number of boosting iterations: 10
## Average tree size: 13
##
## Non-standard options: attempt to group attributes
```

```
amyloidpredict_10x <- predict(object = amyloidmod_10x, newdata = test_data)  # makes a ve
ctor with probabilities
amyloidpredict_10x
```

```
##  [1] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0 0
## [39] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
## Levels: 0 1
```

```
CrossTable(test_data$Mortality, amyloidpredict_10x,
            prop.chisq = FALSE, prop.c = FALSE, prop.r = FALSE,
            dnn = c('actual mortality', 'predicted mortality'))
```

```
##
##
##    Cell Contents
## |-------------------------|
## |                       N |
## |         N / Table Total |
## |-------------------------|
##
##
## Total Observations in Table:  54
##
##
##                  | predicted mortality
## actual mortality |         0 |         1 | Row Total |
## -----------------|-----------|-----------|-----------|
##                0 |        39 |         1 |        40 |
##                  |     0.722 |     0.019 |           |
## -----------------|-----------|-----------|-----------|
##                1 |        12 |         2 |        14 |
##                  |     0.222 |     0.037 |           |
## -----------------|-----------|-----------|-----------|
##     Column Total |        51 |         3 |        54 |
## -----------------|-----------|-----------|-----------|
##
##
```

Now, our model only incorrectly predicts 13 wrong. In otherwords, it is 76% accurate overall for our test set and that is actually better than predicting that everyone will die! Our first “positive” result. It is worth acknoledging that our model only predicted 3 deaths, and missed 12 people that in actually, died. So this wouldn’t be good enough for the “real world.”

Cross validation

Another technique for improving our results, cross validation, splits the data into k number of “folds.” For example, if there are 10 folds, it will split the data into 10 subsets, train the model on k-1 (in this case 9) subsets, and test on the remaining subset. it does this k=10 times over with each fold being used as the test set once. It then aggregares the metrics to get an average performance score.

We’ll use the caret package, which is great for hyperparameter tuning. Essentially, using caret, we can make adjust many parameters and train models on all of them, finding the best outcome.

Below, we set k=10 folds, and run over different numbers of trials. We’ll keep Winnow FALSE (winnow assigns initial weights to input features), but train over different numbers of boosting trials. So in this example, we combine cross validation (based on k or number = 10) and boosting (the different numbers in trials). This code will make 8 models that we run.

```
library(caret)
ctrl <- trainControl(method = "cv", number = 10, selectionFunction = "oneSE")
grid <- expand.grid(model = "tree",
                    trials = c(1, 5, 10, 15, 20, 25, 30, 35),
                    winnow = FALSE)
grid
```

```
##   model trials winnow
## 1  tree      1  FALSE
## 2  tree      5  FALSE
## 3  tree     10  FALSE
## 4  tree     15  FALSE
## 5  tree     20  FALSE
## 6  tree     25  FALSE
## 7  tree     30  FALSE
## 8  tree     35  FALSE
```

Our grid is set up with the different models we will train, and we make all 8 models at once.

```
set.seed(300)
model <- train(Mortality ~., data = train_data, method = "C5.0",
                metric = "Kappa",
                trControl = ctrl,
                tuneGrid = grid)
model
```

```
## C5.0
##
## 163 samples
##  20 predictor
##   2 classes: '0', '1'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 147, 147, 146, 147, 146, 147, ...
## Resampling results across tuning parameters:
##
##   trials  Accuracy  Kappa
##    1      0.7239    -0.05558
##    5      0.7540     0.01791
##   10      0.7544    -0.06805
##   15      0.7482    -0.08027
##   20      0.7540    -0.04162
##   25      0.7603    -0.03619
##   30      0.7544    -0.07601
##   35      0.7548    -0.07242
##
## Tuning parameter 'model' was held constant at a value of tree
## Tuning
##  parameter 'winnow' was held constant at a value of FALSE
## Kappa was used to select the optimal model using  the one SE rule.
## The final values used for the model were trials = 5, model = tree and winnow
##  = FALSE.
```

```
predictions <- predict(model, newdata = test_data)
table(Predicted = predictions, actual = test_data$Mortality)
```

```
##          actual
## Predicted  0  1
##         0 38 11
##         1  2  3
```

The best model is listed after training - it seems the optimal number of trials is 5, and it determines this using kappa. kappa is a measure of the level of agreement between classifiers while accounting for agreement by change. kappa <0 is worse than chance, 0-0.2 is slight agreement, 0.2-0.4 fair, 0.4-0.6 moderate, 0.6-0.8 substantial, and 0.8-1.0 almost perfect. The maximum is 1.

Our model shows we get 41/54 or 76% correct on our best model. With a low number of kappa this is barely better than chance. As stated previously, this is from dataset limitations. Onward.

Random Forests

Random forest models use bootstrapping - where a dataset is sampled over and over with replacement and then a model is trained and then evaluated on data it didn't see in the bootstrap sample. In addition, Random Forest models make decision trees that only let the algorithm choose from a randomly selected subset of features each time it attempts to split at a branch. Once the "forest" is made from the different "trees", it votes on the best one to make the final prediction.

Since each tree is made from random features, no two trees are the same and this limits the model choosing "low hanging fruit" each time that it splits and makes a decision.

```
library(randomForest)
set.seed(300)
rf <- randomForest(Mortality ~., data = newvars)
rf
```

```
##
## Call:
##  randomForest(formula = Mortality ~ ., data = newvars)
##                Type of random forest: classification
##                      Number of trees: 500
## No. of variables tried at each split: 4
##
##         OOB estimate of  error rate: 22.58%
## Confusion matrix:
##     0 1 class.error
## 0 167 4     0.02339
## 1  45 1     0.97826
```

We see here, that the random forest model (which is trained on the entire dataset) predicts 77.4% of the dataset correctly. This output made 500 trees, and tried 4 variables to split. This error rate of 22.6% should be relatively valid if tested on new data, and reflects "out-of-bag" error. In summary, any example not selected for a tree's bootstrap sample can be used to test the model on unseen data. So overall, since each tree is using bootstrapped samples, each tree will not use all of the data. After 500 trees are made, the model can use data not seen by individual trees, and create an error rate by using unseen examples with those trees.

```
library(vcd)
rf$confusion
```

```
##     0 1 class.error
## 0 167 4     0.02339
## 1  45 1     0.97826
```

```
dim(rf$confusion)
```

```
## [1] 2 3
```

```
Kappa(rf$confusion[1:2, 1:2])
```

```
##               value    ASE       z Pr(>|z|)
## Unweighted -0.00245 0.0362 -0.0677    0.946
## Weighted   -0.00245 0.0362 -0.0677    0.946
```

We note the kappa statistic is still very poor. There were 46 total deaths in the entire dataset (for a survival rate of 79%) so by predicting that nobody dies, we would be 79% accurate. Our accuracy was below this, hence the low kappa score.

Gradient Boosting

For our last example of decision trees, we'll use the XGBoost algorithm, a powerful algorithm that uses gradient boosting.

Say that a model makes an initial prediction, and an error rate is calculated. The model learns what trees are contributing the most to the error rate, and optimizes those spots. Overall, we are trying to minimize the error loss function. Generally, this is called gradient descent, and is a key method of how neural networks learn over time - by taking derivatives of a loss function and minimizing that loss function until the error rate is small.

Gradient boosting is the analogous term for gradient descent for decision trees. The algorithm finds areas where trees perform poorly, and optimizes these spots to minimize a loss function in order to boost the model over time.

We'll skip some of the basic gradient boosting methods and go directly to the XGBoost algorithm.

We first start by converting our data into a sparse matrix. For this dataset, the sparse matrix is not that different from normal. A sparse matrix basically is a matrix that contains alot of zeros. Think of a word document that is converted into a matrix where the rows are the word position, and the words are the columns. If you have 2000 words used in combination to make a document, there will be 2000 columns with a 1 representing the position of the word in the document. This is how large language models can store information.

```
library(xgboost)
library(Matrix)
newvarssparse <- sparse.model.matrix(~. -Mortality, data = newvars)
print(newvarssparse[1:5, 1:15])
```

```
## 5 x 15 sparse Matrix of class "dgCMatrix"
```

```
##
## 1 1 60 1 32 21 19 47 1 . 1 0.042 2 1905 2 .
## 2 1 79 1 45 19 12 41 1 1 1 0.139 3 6061 3 .
## 3 1 71 1 58 17 17 32 1 1 . 0.042 2 1261 2 .
## 4 1 68 1 50 24 14 36 . 1 1 0.026 3 1717 1 .
## 5 1 85 1 59 14 13 48 1 . 1 0.010 1 1356 1 .
```

```
newvarssparse <- newvarssparse[,-1] # remove intercepts
```

Our own sparse matrix is much simpler than this, because most of our variables are already encoded to be 1 or zeros.

We make testing and training data out of our sparse matrix, and set vectors of the labels for the outcomes.

```
set.seed(12345)
train_sample <- sample(217,163)
train_data_sparse <- newvarssparse[train_sample,] # 163 items long
test_data_sparse <- newvarssparse[-train_sample,] # 54 items long

test_data_labels <- ifelse(newvars[-train_sample, c("Mortality")] == 1, 1, 0)
train_data_labels <- ifelse(newvars[train_sample, c("Mortality")] == 1, 1, 0)
```

For XGBoost, we set a list of parameters. Below are the standard parameters (we will play with these in a minute).

Objective uses binary:logistic for binary classification, but other methods like multi:softprob can be used for categorical variables

Max_depth is between zero and infinity, and determines the maximum depth of any tree

eta is between zero and 1 and affects the learning rate.

gamma is between zero and 1, and determines whether the algorithm continues splitting

colsample_bytree is between zero and 1, and defines the % of features that will be selected at random by each tree

min_child_weight is between zero and infinity, and is the minimum examples needed to split

subsample is betweeen zero and 1, and defines the % of randomly selected examples for each iteration

We set our parameters, and then make a model with 100 rounds of boosting iterations and print every 10.

```
params.xgb <- list(objective = "binary:logistic",
                   max_depth = 6,
                   eta = 0.3,
                   gamma = 0,
                   colsample_bytree = 1,
                   min_child_Weight =1,
                   subsample = 1)
set.seed(12345)
xgbm <- xgboost(params.xgb, data = train_data_sparse,
                label = train_data_labels,
                nrounds = 100,
                verbose = 1,
                print_every_n = 10,
                eval_metric = "logloss")
```

```
## [1]  train-logloss:0.526764
## [11] train-logloss:0.082912
## [21] train-logloss:0.017406
## [31] train-logloss:0.004305
## [41] train-logloss:0.001009
## [51] train-logloss:0.000296
## [61] train-logloss:0.000306
## [71] train-logloss:0.000307
## [81] train-logloss:0.000307
## [91] train-logloss:0.000307
## [100]    train-logloss:0.000307
```

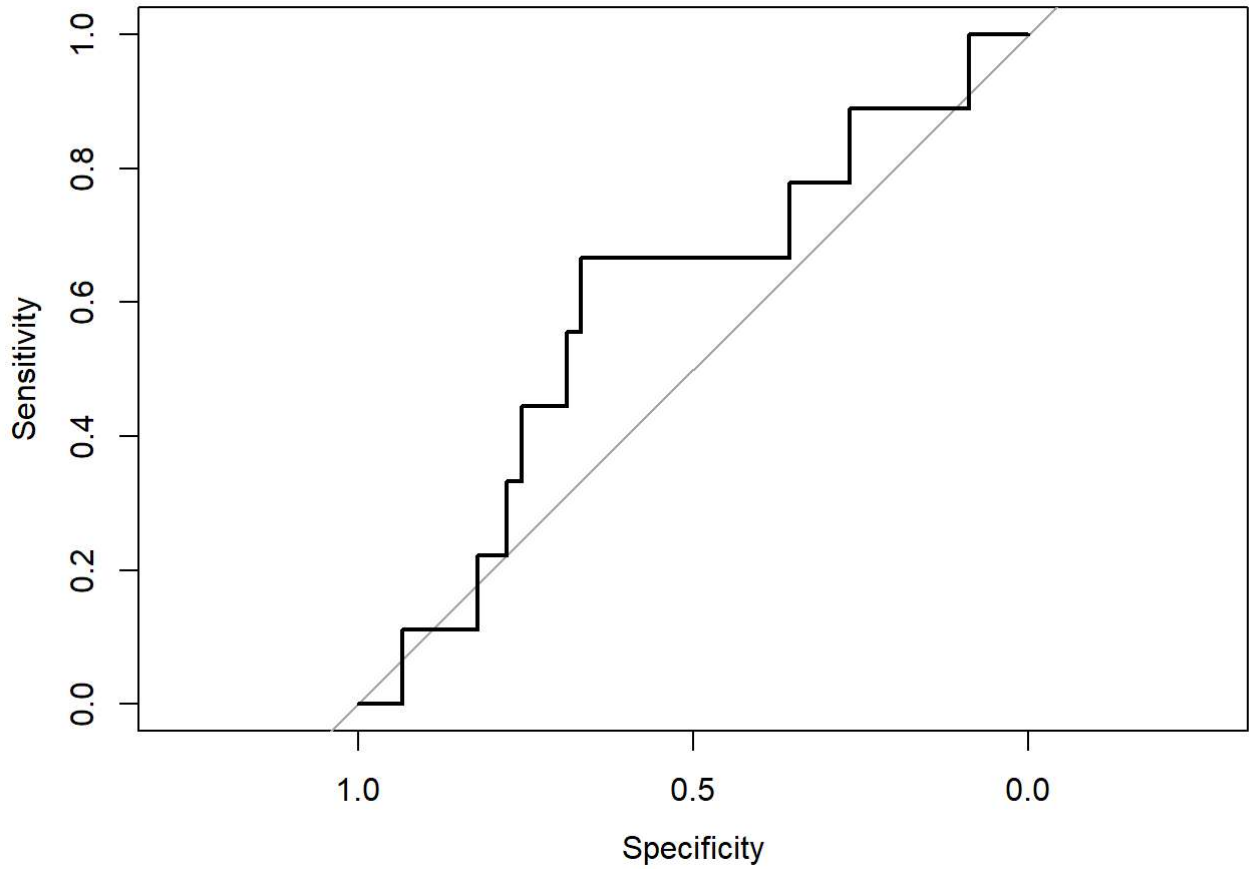
We make our predictions, and these are outputted as a probability.

Below that, we show a ROC curve given our results which is still uh, not great (but better than before) but we accept the result given our dataset limitations.

```
set.seed(300)
prob_mortality <- predict(xgbm, test_data_sparse)
prob_mortality
```

```
##  [1] -0.069858 -0.042983 -0.053300  0.223265  0.106916  0.024175  0.900237
##  [8]  0.766703  0.361150  0.116899 -0.020109  0.275024  0.798817  0.101861
## [15]  0.235173  0.146476  0.815063  0.765462  0.702506  0.203588  0.009228
## [22]  0.004961 -0.043898  0.155254  0.344028  0.751216  0.081789  0.901183
## [29]  0.047373  0.879758  0.136746  0.501567  0.822145  0.514440  0.491150
## [36]  0.469379  0.621188  0.107921  0.191584  0.047363 -0.053561  0.932258
## [43]  0.719102  0.864993  0.085603  0.611049  0.008374  0.488562 -0.110359
## [50]  0.579186  0.189343  0.613892  0.127769  0.453972
```

```
library(pROC)
roc_curve_xgb <- roc(response = test_data_labels, predictor = prob_mortality)
plot(roc_curve_xgb)
```




```
auc(roc_curve_xgb)
```

```
## Area under the curve: 0.595
```

Lastly, to make a kappa statistic we use the probabilities to say that anything with a 50% chance of mortality or greater is 1, and anything less is zero.

```
predict_mortality <- ifelse(prob_mortality >= 0.5, 1, 0)
table(predict_mortality, test_data_labels)
```

```
##                  test_data_labels
## predict_mortality  0  1
##                 0 31  4
##                 1 14  5
```

```
library(vcd)
kappa(table(predict_mortality, test_data_labels))
```

```
##       Estimate Std.Err     2.5% 97.5% P-value
## kappa   0.1692  0.1285 -0.08255 0.421  0.1877
```

We get a kappa of 0.17 which is still not awesome but better than anything we have done before!

Note that the accuracy of our test dataset is 36/54 or 67%. Which is really not better than we've done.

Lastly, we'll use the caret package to play around with some of the hyperparameters of the model to see if we can beat 67%. We make a grid with different hyperparameters of the parameter list for training using the caret package like we've done before.

```
 grid_xgb <- expand.grid(
    eta = c(0.3, 0.4),
    max_depth = c(1,2,3),
    colsample_bytree = c(0.6, 0.8),
    subsample = c(0.5, 0.75, 1.00),
    nrounds = c(50, 100, 150),
    gamma = c(0,1),
    min_child_weight = 1
  )
library(caret)
ctrl <- trainControl(method = "cv",
                        number = 10,
                        selectionFunction = "best")
```

Now, when we train the model using our grid, we make many different models with combinations of all of the inputs of hyperparameters we used.

We used 2 variables for eta, 3 for max_depth, 2 for colsample_bytree, 3 for subsample, 3 for nrounds, 2 for gamma. so we will make 216 different combinations of hyperparameters and be able to find the best model out of these.

```
set.seed(300)
m_xgb <- train(Mortality ~., data = newvars, method = "xgbTree",
                 trControl = ctrl, tuneGrid = grid_xgb,
                 metric = "Kappa", verbosity = 0)
```

What we now have is 216 different combinations, and we can select the best one

```
m_xgb$bestTune
```

```
##     nrounds max_depth eta gamma colsample_bytree min_child_weight subsample
## 205      50         3 0.4     1              0.6                1         1
```

Our best model used 50 rounds, depth of 3, eta of 0.4, gamma of 1, colsample_bytree of 0.6.

```
max(m_xgb$results["Kappa"])
```

```
## [1] 0.2117
```

The best kappa that we get is 0.21. We will happily accept this!

We don't have an accuracy score, because we used the whole dataset for this model. We could find the accuracy if we applied other data.

Regression trees

We'll change gears here to a regression tree. A regression tree is a decision tree that is designed for predicting continuous variables. It will still work in this case, as it will output a continuous number (probability) and then we can set any output that is >50% to be a prediction for mortality. So while its not ideal to use here, it is still a technique that I wanted to include.

```
library(rpart) # rpart - recursive partitioning - from CART team
set.seed(1234)
amyloid.part <- rpart(Mortality ~., data = train_data)
amyloid.part
```

```
## n= 163
##
## node), split, n, loss, yval, (yprob)
##       * denotes terminal node
##
##  1) root 163 32 0 (0.80368 0.19632)
##    2) NT.proBNP< 1889 72  5 0 (0.93056 0.06944) *
##    3) NT.proBNP>=1889 91 27 0 (0.70330 0.29670)
##      6) NT.proBNP>=2769 53 11 0 (0.79245 0.20755) *
##      7) NT.proBNP< 2769 38 16 0 (0.57895 0.42105)
##       14) LAVI< 49.5 26  8 0 (0.69231 0.30769) *
##       15) LAVI>=49.5 12  4 1 (0.33333 0.66667) *
```

```
summary(amyloid.part)
```

```
## Call:
## rpart(formula = Mortality ~ ., data = train_data)
##   n= 163
##
##        CP nsplit rel error xerror   xstd
## 1 0.04167      0     1.000  1.000 0.1585
## 2 0.01000      3     0.875  1.344 0.1758
##
## Variable importance
##                  NT.proBNP                 Mayo.stage
##                         35                         16
##                       LAVI                    Hs.cTnT
##                         13                         10
##                         EF                        PWT
##                          9                          7
##                       NYHA                       IVST
##                          5                          3
##                        Age Runs.per.monitoring.period
##                          2                          1
##
## Node number 1: 163 observations,    complexity param=0.04167
##   predicted class=0  expected loss=0.1963  P(node) =1
##     class counts:   131    32
##    probabilities: 0.804 0.196
##   left son=2 (72 obs) right son=3 (91 obs)
##   Primary splits:
##       NT.proBNP  < 1888   to the left,  improve=4.152, (0 missing)
##       VE..       < 0.28   to the left,  improve=1.983, (0 missing)
##       Age        < 86.5   to the left,  improve=1.652, (0 missing)
##       LAVI       < 52.5   to the left,  improve=1.248, (0 missing)
##       Mayo.stage < 1.5    to the left,  improve=1.102, (0 missing)
##   Surrogate splits:
##       Mayo.stage < 1.5    to the left,  agree=0.773, adj=0.486, (0 split)
##       Hs.cTnT    < 0.0335 to the left,  agree=0.724, adj=0.375, (0 split)
##       EF         < 52.5   to the right, agree=0.669, adj=0.250, (0 split)
##       PWT        < 14.5   to the left,  agree=0.650, adj=0.208, (0 split)
##       NYHA       < 1.5    to the left,  agree=0.644, adj=0.194, (0 split)
##
## Node number 2: 72 observations
##   predicted class=0  expected loss=0.06944  P(node) =0.4417
##     class counts:    67     5
##    probabilities: 0.931 0.069
##
## Node number 3: 91 observations,    complexity param=0.04167
##   predicted class=0  expected loss=0.2967  P(node) =0.5583
##     class counts:    64    27
##    probabilities: 0.703 0.297
##   left son=6 (53 obs) right son=7 (38 obs)
```

```
##   Primary splits:
##       NT.proBNP < 2769   to the right, improve=2.018, (0 missing)
##       VE..      < 5.5    to the right, improve=1.978, (0 missing)
##       Age       < 69.5   to the left,  improve=1.546, (0 missing)
##       Hs.cTnT   < 0.0165 to the right, improve=1.145, (0 missing)
##       IVST      < 13.5   to the right, improve=1.143, (0 missing)
##   Surrogate splits:
##       Mayo.stage                < 2.5    to the right, agree=0.758, adj=0.421, (0 spl
it)
##       Runs.per.monitoring.period < 6.965  to the left,  agree=0.615, adj=0.079, (0 spl
it)
##       IVST                      < 13.5   to the right, agree=0.604, adj=0.053, (0 spl
it)
##       PWT                       < 18.5   to the left,  agree=0.604, adj=0.053, (0 spl
it)
##       LAVI                      < 38.5   to the right, agree=0.604, adj=0.053, (0 spl
it)
##
## Node number 6: 53 observations
##   predicted class=0  expected loss=0.2075  P(node) =0.3252
##     class counts:    42    11
##    probabilities: 0.792 0.208
##
## Node number 7: 38 observations,    complexity param=0.04167
##   predicted class=0  expected loss=0.4211  P(node) =0.2331
##     class counts:    22    16
##    probabilities: 0.579 0.421
##   left son=14 (26 obs) right son=15 (12 obs)
##   Primary splits:
##       LAVI < 49.5   to the left,  improve=2.1160, (0 missing)
##       IVST < 13.5   to the right, improve=1.4760, (0 missing)
##       EF   < 41     to the left,  improve=1.0260, (0 missing)
##       Age  < 72     to the left,  improve=0.9324, (0 missing)
##       VE.. < 0.22   to the left,  improve=0.6812, (0 missing)
##   Surrogate splits:
##       EF      < 30.5   to the right, agree=0.763, adj=0.250, (0 split)
##       Age     < 91.5   to the left,  agree=0.737, adj=0.167, (0 split)
##       IVST    < 12.5   to the right, agree=0.737, adj=0.167, (0 split)
##       PWT     < 19.5   to the left,  agree=0.737, adj=0.167, (0 split)
##       Hs.cTnT < 0.0125 to the right, agree=0.711, adj=0.083, (0 split)
##
## Node number 14: 26 observations
##   predicted class=0  expected loss=0.3077  P(node) =0.1595
##     class counts:    18     8
##    probabilities: 0.692 0.308
##
## Node number 15: 12 observations
##   predicted class=1  expected loss=0.3333  P(node) =0.07362
```

```
##     class counts:     4     8
##    probabilities: 0.333 0.667
```

We get a lot of text here.

The first output, the name of our model (amyloid.part) shows in text the splits of our tree.

The second output, summary(amyloid.part) gives a more comprhensive summary of the model. for example, it weights continuous variables like NT.proBNP and %VE as more important unlike the simple decision treees before. This model likes continuous variables as it is doing regression at each step.

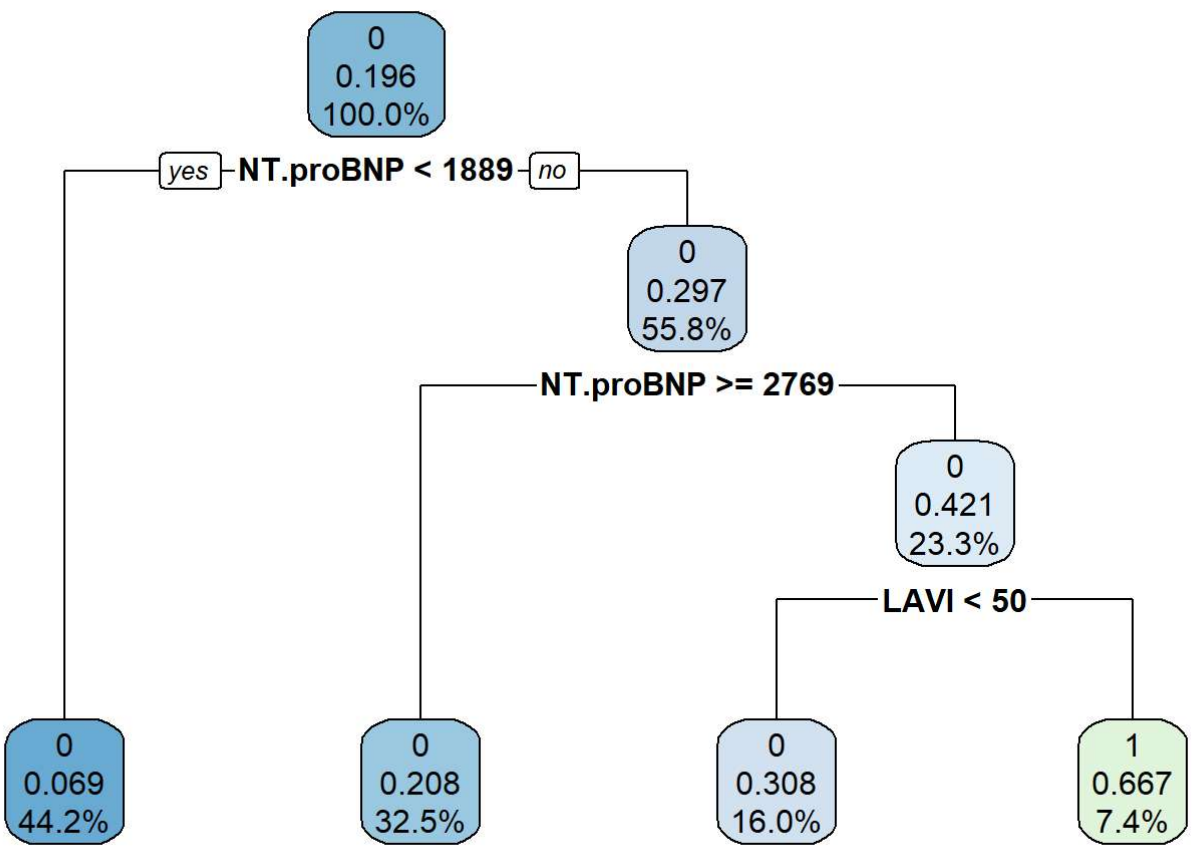
At the first split, it puts 124 training patients into class zero based on the majority of that node. the expected loss at the first split is 0.239, meaning that 23.9% of instances are misclassified at this node. AS it grows, it will get better at classifying patients as 1 (mortality) or zero (no mortality).

Below is a plot to show this

```
library(rpart.plot)
```

```
## Warning: package 'rpart.plot' was built under R version 4.4.2
```

```
rpart.plot(amyloid.part, digits = 3)
```



Next, we make predictions based on our model and store this in p.amyloid.part.

We use an ifelse() function to state that anything with a probability of mortality >50% counts as a 1, and anything below it does not predict mortality

```
set.seed(300)
p.amyloid.part <- predict(amyloid.part, test_data)
predicted_classes <- ifelse(p.amyloid.part[,2] >= 0.5, 1, 0)
table(predicted = predicted_classes, Actual = test_data$Mortality)
```

```
##          Actual
## predicted  0  1
##         0 38 14
##         1  2  0
```

Here, we get an accuracy of 38/52 = 73% on our testing dataset. This is pretty decent especially considering how a regression tree - made for estimating continuous variables - isn't really what we should be using.

Boruta

The next section is relatively short, and we'll look at the Boruta algorithm. This algorithm is designed to show the most relevant variables for making models. Although we ignored it for the prior exercises, it has utility and is worth showing.

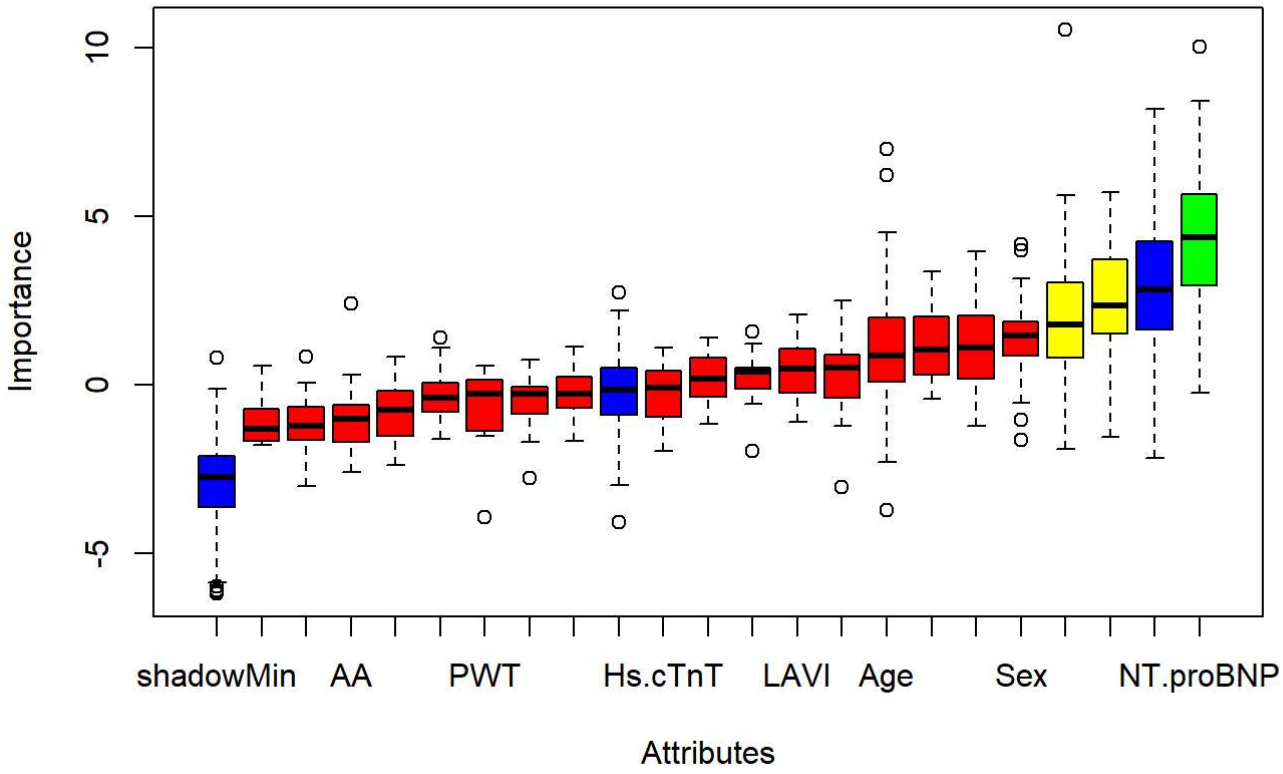
Boruta makes "shadow" features by inputting random values of the dataset, and then compares the actual variables in the dataset to the "shadow variables". If variables outperform the shadow variables, they are deemed to be important.

So, if we had an extensive dataset, it may have been worth starting with this algorithm. Because we used all the features in our dataset it doesn't matter as much, but I still wanted to include it here.

```
library(Boruta)
amyloid_boruta <- Boruta(Mortality ~., data = train_data, doTrace =1)
amyloid_boruta
```

```
## Boruta performed 99 iterations in 4.145 secs.
##  1 attributes confirmed important: NT.proBNP;
##  17 attributes confirmed unimportant: AA, Age, Atrial.fibrillation, BB,
## CCB and 12 more;
##  2 tentative attributes left: NYHA, VE..;
```

```
plot(amyloid_boruta)
```



As shown by the output and the plot, it picks Mayo.stage, NT.ProBNP, and VT_VT_WS as the most important predictors. This makes sense - BNP being high in cardiac amyloidosis is not a good prognostic sign, and the mayo staging was invented as a way to prognosticate wild type amyloidosis. Having sustained ventricular arrhythmias is also a reasonable predictor of mortality.

Neural nets

We'll move onto an example using neural nets. Because the neural nets won't accept factor data, we'll have to do alot of converting. We'll start over with our newvars dataset.

First, we will make a normalize() function to center each variable.

```
library(neuralnet)
library(knitr)
set.seed(12345)
normalize <- function(x) {
    return((x - min(x)) / (max(x) - min(x)))
}
str(newvars)
```

```
## 'data.frame':    217 obs. of  21 variables:
##  $ Age                      : int  60 79 71 68 85 82 77 71 79 73 ...
##  $ Sex                      : Factor w/ 2 levels "F","M": 2 2 2 2 2 2 2 2 2 2 ...
##  $ EF                       : int  32 45 58 50 59 71 36 65 48 56 ...
##  $ IVST                     : num  21 19 17 24 14 17 12 19 16 11 ...
##  $ PWT                      : num  19 12 17 14 13 14 11 17 14 13 ...
##  $ LAVI                     : num  47 41 32 36 48 67 44 46 44 57 ...
##  $ BB                       : Factor w/ 2 levels "0","1": 2 2 2 1 2 1 1 2 2 2 ...
##  $ CCB                      : Factor w/ 2 levels "0","1": 1 2 2 2 1 1 1 1 1 1 ...
##  $ AA                       : Factor w/ 2 levels "0","1": 2 2 1 2 2 2 1 1 1 1 ...
##  $ Hs.cTnT                  : num  0.042 0.139 0.042 0.026 0.01 0.07 0.01 0.06 0.045
0.074 ...
##  $ NYHA                     : num  2 3 2 3 1 2 3 3 3 2 ...
##  $ NT.proBNP                : num  1905 6061 1261 1717 1356 ...
##  $ Mayo.stage               : num  2 3 2 1 1 2 2 2 1 2 ...
##  $ Syncope                  : Factor w/ 2 levels "0","1": 1 1 1 1 1 1 1 1 2 1 ...
##  $ NSVT.1                   : Factor w/ 2 levels "0","1": 2 1 1 1 2 2 1 1 1 2 ...
##  $ Runs.per.monitoring.period: num  18.6 0 0 0 758.2 ...
##  $ Longest                  : int  13 0 0 0 10 6 0 0 0 6 ...
##  $ VE..                     : num  1.1 0.01 0.3 0.01 26 3 4 0.1 15 2 ...
##  $ Atrial.fibrillation      : Factor w/ 2 levels "0","1": 2 1 2 2 1 2 2 2 2 2 ...
##  $ VT_VF_WS                 : Factor w/ 2 levels "0","1": 1 1 1 1 2 1 1 1 1 1 ...
##  $ Mortality                : Factor w/ 2 levels "0","1": 2 1 1 1 1 1 2 1 1 1 ...
```

Note that we have alot of factors in our data, and only Sex is as a character (all other factors are numeric and are usable-enough)

We'll make dummy variables for sex_male and sex_female (aka one-hot encoding).

we remove the old Sex variable, and convert all factors to numbers and then re-select our variables

```
NNdata <- newvars %>%
  mutate(
    sex_male = ifelse(Sex == "M", 1, 0),
    sex_female = ifelse(Sex == "F", 1, 0)
         ) %>%
  select (-Sex) %>%
  mutate(
    across(where(is.factor), ~ as.numeric(as.character(.)))
    ) %>%
  select(c("Age", "EF", "IVST", "PWT", "LAVI",
           "Hs.cTnT", "NYHA", "Mayo.stage", "Runs.per.monitoring.period",
           "Longest", "VE..", "Mortality", "sex_male", "sex_female",
           "BB", "CCB", "AA", "Syncope", "NSVT.1", "Atrial.fibrillation",
           "VT_VF_WS")) %>%
  mutate(across(as.numeric()))
```

We then apply our normalization function. Note that we remove column 12 of NNdata, as that is our column for mortality.

We then add back mortality, as we don't want to normalize that one.

```
## 'data.frame':    217 obs. of  21 variables:
##  $ Age                      : num  0.32 0.7 0.54 0.48 0.82 0.76 0.66 0.54 0.7 0.58
...
##  $ EF                       : num  0.264 0.509 0.755 0.604 0.774 ...
##  $ IVST                     : num  0.667 0.556 0.444 0.833 0.278 ...
##  $ PWT                      : num  0.786 0.286 0.643 0.429 0.357 ...
##  $ LAVI                     : num  0.291 0.221 0.116 0.163 0.302 ...
##  $ Hs.cTnT                  : num  0.0516 0.2081 0.0516 0.0258 0 ...
##  $ NYHA                     : num  0.333 0.667 0.333 0.667 0 ...
##  $ Mayo.stage               : num  0.5 1 0.5 0 0 0.5 0.5 0.5 0 0.5 ...
##  $ Runs.per.monitoring.period: num  0.0124 0 0 0 0.5035 ...
##  $ Longest                  : num  0.542 0 0 0 0.417 ...
##  $ VE..                     : num  0.034375 0.000312 0.009375 0.000312 0.8125 ...
##  $ sex_male                 : num  1 1 1 1 1 1 1 1 1 1 ...
##  $ sex_female               : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ BB                       : num  1 1 1 0 1 0 0 1 1 1 ...
##  $ CCB                      : num  0 1 1 1 0 0 0 0 0 0 ...
##  $ AA                       : num  1 1 0 1 1 1 0 0 0 0 ...
##  $ Syncope                  : num  0 0 0 0 0 0 0 0 1 0 ...
##  $ NSVT.1                   : num  1 0 0 0 1 1 0 0 0 1 ...
##  $ Atrial.fibrillation      : num  1 0 1 1 0 1 1 1 1 1 ...
##  $ VT_VF_WS                 : num  0 0 0 0 1 0 0 0 0 0 ...
##  $ Mortality                : Factor w/ 2 levels "0","1": 2 1 1 1 1 1 2 1 1 1 ...
```

The result is a dataset that we can work with. Like previously, we make training and testing sets.

```
set.seed(900)
train_sample <- sample(217,163)
train_data_NN <- NNdata_normalized[train_sample,] # 163 items long
test_data_NN <- NNdata_normalized[-train_sample,] # 54 items long
```

We'll now make the neural net.

We use a logistic activation function given that we want a binary outcome.

We will try 4 layers of hidden nodes. I experimented with these and came up with them. Its tough to not overfit or underfit but for this intro exercise this is okay.

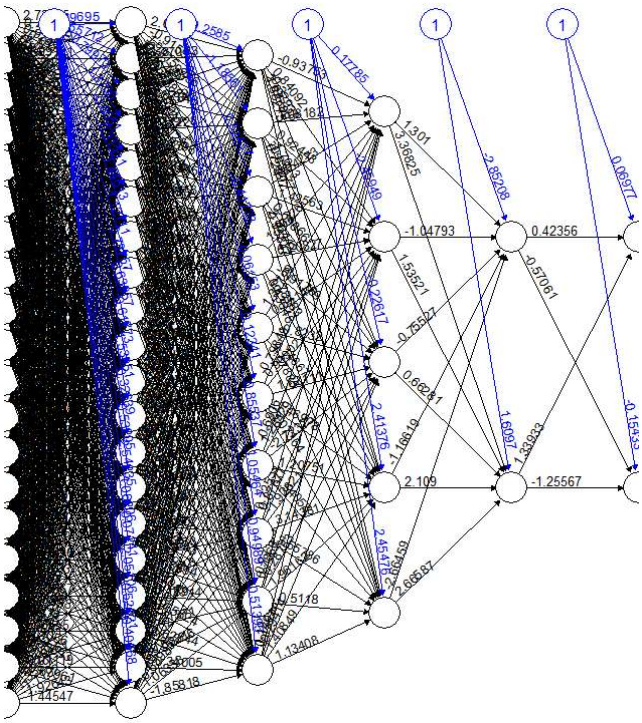
Note that since plotting a neural object opens it up in a new window, to get it to show in the markdown file we save it as a picture.

```
set.seed(900)
amy_nn <- neuralnet(
  Mortality ~.,
  data = train_data_NN,
  hidden = c(20,10,5,2),
  act.fct = "logistic",
  linear.output = FALSE
)
```

We'll show the plot but beware - it is implicated because we made so many hidden layers.

The numbers inbetween nodes show the strength and direction of the relationship between neurons.

```
knitr::include_graphics("amyloid_nn.png")
```



Lastly, we'll get an accuracy score.

Because we added back the mortality variable, this is now in column 21 and we remove it prior to predictions.

Once getting our predictions, anything with a probability >50% is bucketed into Mortality, like we have done previously.

```
set.seed(1234)
predictions <- compute(amy_nn, test_data_NN[,-21])
predicted_probabilities <- predictions$net.result
predicted_classes <- ifelse(predicted_probabilities >=0.5, 1, 0)
confusion_matrix <- table(Predicted = predicted_classes[,2], Actual = test_data_NN$Mortal
ity)
confusion_matrix
```

```
##          Actual
## Predicted  0  1
##         0 34  7
##         1  6  7
```

```
accuracy <- sum(diag(confusion_matrix)) / sum(confusion_matrix)
print(paste("Accuracy:", accuracy))
```

```
## [1] "Accuracy: 0.759259259259259"
```

We get an accuracy of 76% which is about as good as we've done all day.

I think the learning point from this is that, for our dataset which is not very large, tree based methods are actually better, and honestly the logistic regression model was pretty good and had an AUC that was better than some of our tree based methods. For alot of classification tasks, this holds true and neural nets tend to be best for unstructured data (words, pictures, etc.). So this wasn't really a dataset that we needed to use machine learning for.

It is really important to point out that in our test dataset, 40/54 patients lived. Thus, if we just predicted that everyone would live we would get 74% accuracy.

while this was an educational run-through of machine learning methods, it doesn't really move the needle much for making new discoveries for our small dataset. And while we maybe could have improved that with refining of variables and methods, ultimately this model is not going to be used in the future. Hopefully, with a large enough dataset, these methods could be applied for future studies or research projects

I hope that you learned during this exercise as I have.