Decision Trees and Random Forests

Shane T. Mueller shanem@mtu.edu

February 24, 2017

Decision Trees, Forests, and Nearest-Neighbors classifiers

The classic statistical decision theory on which LDA and QDA are based are highly model-based. We assume the features are fit by some model, we fit that model, and use inferences from that model to make a decision. Using the model means we make assumptions, and if those assumptions are correct, we can have a lot of success. Not all classifiers make such strong assumptions, and we will cover several today.

Decision trees

These methods assume that the different predictors are independent and combine together to form a an overall likelihood of one class over another. However, this may not be true. Many times, we might want to make a classification rule based on a few simple measures. The notion is that you may have several measures, and by asking a few decisions about individual dimensions, end up with a classification decision. For example, such trees are frequently used in medical contexts when doing triage or first-response:

Decision 1: is the person conscious Yes: Move to another person No: Use Decision 2

Decision 2: Is the person breathing? Yes: Move to decision 3 No: Move to decision 4

Decision 3: Are they bleeding? Yes: etc. No: etc. Decision 4: Does the person have a pulse? No: Begin CPR Yes: etc.

Notice that if we were trying to determine an action via LDA, we'd create a single composite score based on all the questions and make a decision at the end. Tree-based decision tools can be useful in many cases:

- When we want simple decision rules that can be applied by novices or people under time stress.
- When the structure of our classes are dependent or nested, or somehow hierarchical. Many natural categories have a hierarchical structure, so that the way you split one variable may depend on the value of another. For example, if you want to know if someone is 'tall', you first might determine their gender, and then use a different cutoff for each gender.
- When many of our observables are binary states. To classify lawmakers we can look at their voting patterns–which are always binary. We may be able to identify just a couple issues we care about that will tell us everything we need to know.

To make a decision tree, we essentially have to use heuristic processes to determine the best order and cutoffs to use in making decisions, while identifying our error rates. Even with a single feature, we can make a decision tree that correctly classifies all the elements in a set (as long as all feature values are unique). So we also need to understand how many rules to use, in order to minimize over-fitting.

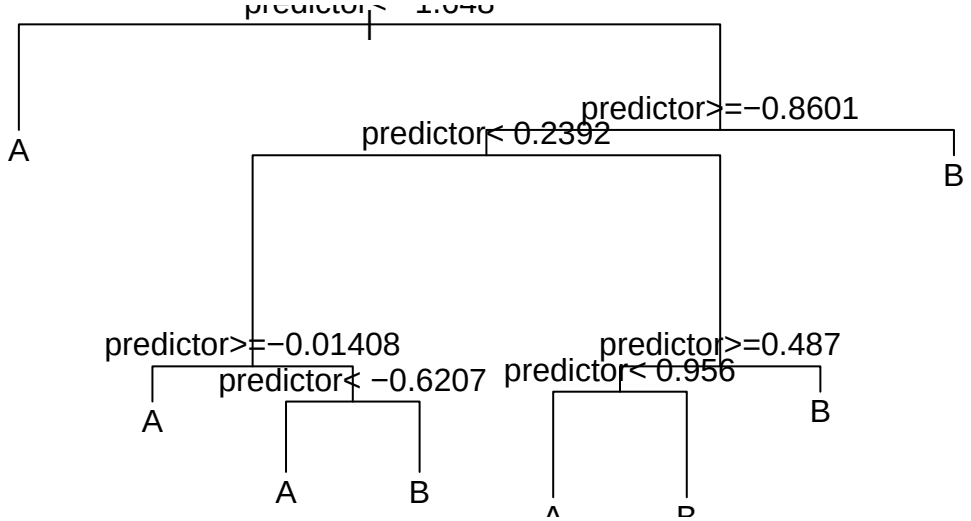
There are a number of software packages available for classification trees. One commonly-used package in R is called rpart.

'Let's start with a simple tree made to classify elements on which we have only one measure:

```
library(rpart)
library(DAAG)
```

```
## Loading required package: lattice
```

```
classes <- sample(c("A","B"),100,replace=T)
predictor <- rnorm(100)
r1 <- rpart(classes~predictor,method="class")
plot(r1)
text(r1)
```

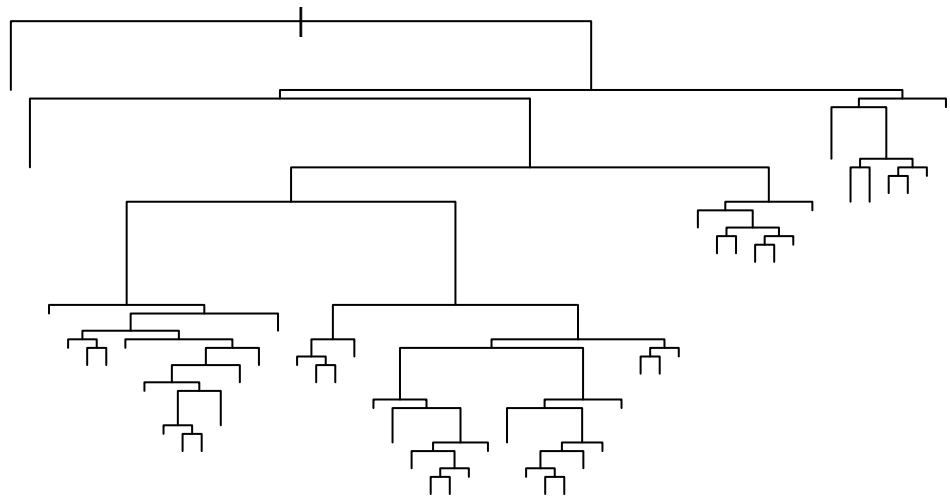



```
confusion(classes,predict(r1,type="class"))
```

```
## Overall accuracy = 0.67
##
## Confusion matrix
##       Predicted (cv)
## Actual  [,1]  [,2]
##   [1,] 0.562 0.438
##   [2,] 0.231 0.769
```

Notice that even though the predictor had no true relationship to the categories, we were able to get 71% accuracy. We could in fact do better.

```
r2 <- rpart(classes~predictor,method="class",control=c(minsplit=1,minbucket=1,
                                                       cp=-1))
plot(r2)
```



```
confusion(classes,predict(r2,type="class"))
```

```
## Overall accuracy = 1
##
## Confusion matrix
##       Predicted (cv)
## Actual [,1] [,2]
##   [1,]    1    0
##   [2,]    0    1
```

Now, we have completely predicted every item by successively dividing the line. There were three parameters we adjusted to do this. First, we set the minsplit and minbucket to 1–this determines the smallest size of the leaf nodes. Then, we set the cp argument to be negative (it defaults to .01). cp is a complexity parameter, that is a criterion for how much better each model must be before splitting a leaf node. A negative value will mean that any additional node is better.

How will this work for a real data set? Let's re-load the engineering data set:

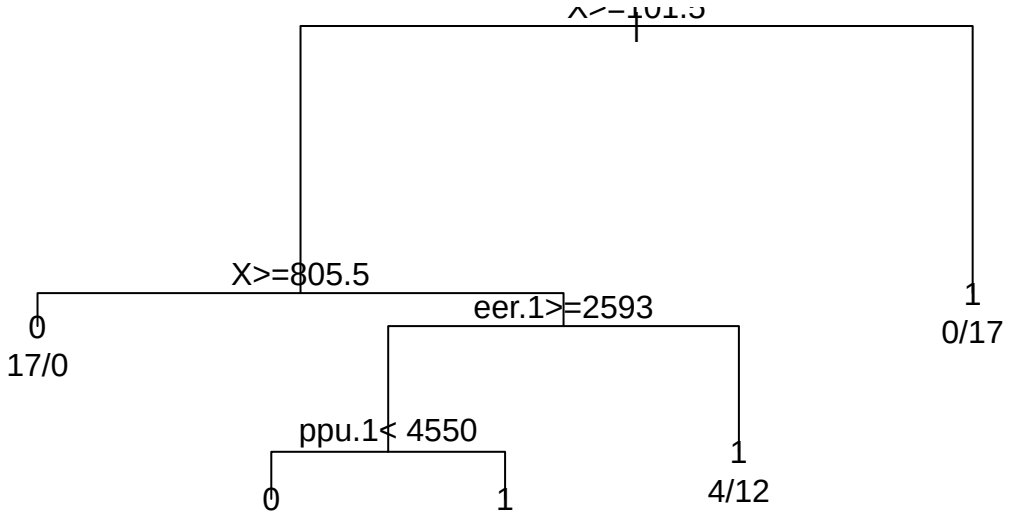
```
joint <- read.csv("eng-joint.csv")

##This is the partitioning tree:

r1 <- rpart(eng~.,data=joint,method="class")
r1
```

```
## n= 76
##
## node), split, n, loss, yval, (yprob)
##       * denotes terminal node
##
##  1) root 76 38 0 (0.5000000 0.5000000)
##    2) X>=101.5 59 21 0 (0.6440678 0.3559322)
##      4) X>=805.5 17  0 0 (1.0000000 0.0000000) *
##      5) X< 805.5 42 21 0 (0.5000000 0.5000000)
##       10) eer.1>=2592.575 26  9 0 (0.6538462 0.3461538)
##         20) ppu.1< 4549.837 19  4 0 (0.7894737 0.2105263) *
##         21) ppu.1>=4549.837 7  2 1 (0.2857143 0.7142857) *
##       11) eer.1< 2592.575 16  4 1 (0.2500000 0.7500000) *
##    3) X< 101.5 17  0 1 (0.0000000 1.0000000) *
```

```
plot(r1)
text(r1,use.n=TRUE)
```




```
confusion(joint$eng,predict(r1,type="class"))
```

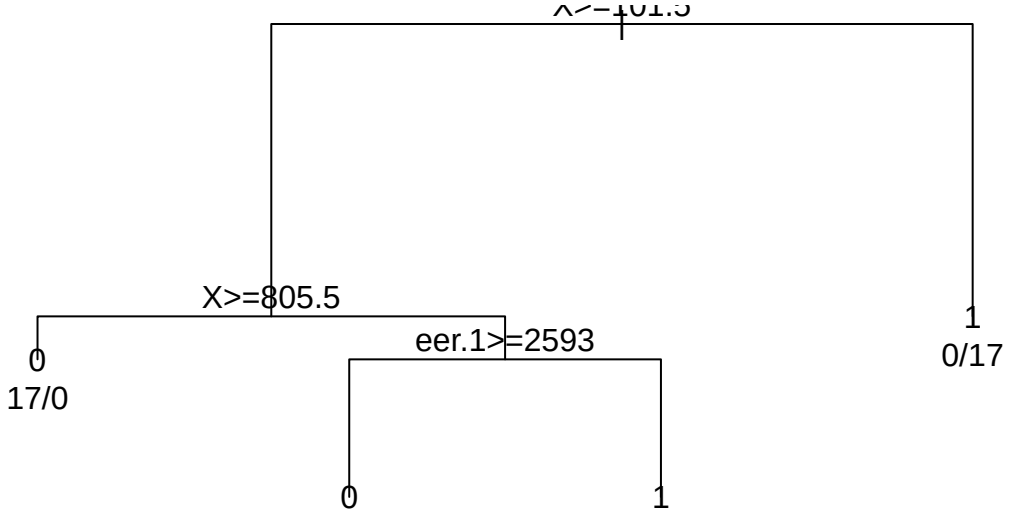
```
## Overall accuracy = 0.868
##
## Confusion matrix
##       Predicted (cv)
## Actual  [,1]  [,2]
##   [1,] 0.842 0.158
##   [2,] 0.105 0.895
```

With the default full partitioning, we get 73% accuracy.. But the decision tree is fairly complicated, and we might want something simpler. Let's only allow it to go to a depth of 2

```
r2 <- rpart(eng~.,data=joint,method="class",control=c(maxdepth=3))
r2
```

```
## n= 76
##
## node), split, n, loss, yval, (yprob)
##       * denotes terminal node
##
##  1) root 76 38 0 (0.5000000 0.5000000)
##    2) X>=101.5 59 21 0 (0.6440678 0.3559322)
##      4) X>=805.5 17  0 0 (1.0000000 0.0000000) *
##      5) X< 805.5 42 21 0 (0.5000000 0.5000000)
##       10) eer.1>=2592.575 26  9 0 (0.6538462 0.3461538) *
##       11) eer.1< 2592.575 16  4 1 (0.2500000 0.7500000) *
##    3) X< 101.5 17  0 1 (0.0000000 1.0000000) *
```

```
plot(r2)
text(r2,use.n=T)
```




```
predict(r2)
```

```
##            0         1
## 1  0.0000000 1.0000000
## 2  0.0000000 1.0000000
## 3  0.0000000 1.0000000
## 4  0.0000000 1.0000000
## 5  0.0000000 1.0000000
## 6  0.0000000 1.0000000
## 7  0.0000000 1.0000000
## 8  0.0000000 1.0000000
## 9  0.0000000 1.0000000
## 10 0.0000000 1.0000000
## 11 0.0000000 1.0000000
## 12 0.0000000 1.0000000
## 13 0.0000000 1.0000000
## 14 0.0000000 1.0000000
## 15 0.0000000 1.0000000
## 16 0.0000000 1.0000000
## 17 0.0000000 1.0000000
## 18 0.6538462 0.3461538
## 19 0.6538462 0.3461538
## 20 0.2500000 0.7500000
## 21 0.6538462 0.3461538
## 22 0.6538462 0.3461538
## 23 0.2500000 0.7500000
## 24 0.6538462 0.3461538
## 25 0.6538462 0.3461538
## 26 0.6538462 0.3461538
## 27 0.2500000 0.7500000
## 28 0.2500000 0.7500000
## 29 0.2500000 0.7500000
## 30 0.6538462 0.3461538
## 31 0.6538462 0.3461538
## 32 0.2500000 0.7500000
## 33 0.6538462 0.3461538
## 34 0.6538462 0.3461538
## 35 0.6538462 0.3461538
```

```
## 36 0.2500000 0.7500000
## 37 0.2500000 0.7500000
## 38 0.2500000 0.7500000
## 39 0.2500000 0.7500000
## 40 0.2500000 0.7500000
## 41 0.2500000 0.7500000
## 42 0.6538462 0.3461538
## 43 0.6538462 0.3461538
## 44 0.2500000 0.7500000
## 45 0.6538462 0.3461538
## 46 0.6538462 0.3461538
## 47 0.2500000 0.7500000
## 48 0.6538462 0.3461538
## 49 0.6538462 0.3461538
## 50 0.6538462 0.3461538
## 51 0.2500000 0.7500000
## 52 0.6538462 0.3461538
## 53 0.6538462 0.3461538
## 54 0.6538462 0.3461538
## 55 0.6538462 0.3461538
## 56 0.6538462 0.3461538
## 57 0.2500000 0.7500000
## 58 0.6538462 0.3461538
## 59 0.6538462 0.3461538
## 60 1.0000000 0.0000000
## 61 1.0000000 0.0000000
## 62 1.0000000 0.0000000
## 63 1.0000000 0.0000000
## 64 1.0000000 0.0000000
## 65 1.0000000 0.0000000
## 66 1.0000000 0.0000000
## 67 1.0000000 0.0000000
## 68 1.0000000 0.0000000
## 69 1.0000000 0.0000000
## 70 1.0000000 0.0000000
## 71 1.0000000 0.0000000
## 72 1.0000000 0.0000000
## 73 1.0000000 0.0000000
## 74 1.0000000 0.0000000
## 75 1.0000000 0.0000000
## 76 1.0000000 0.0000000
```

```
confusion(joint$eng,predict(r2,type="class"))
```

```
## Overall accuracy = 0.829
##
## Confusion matrix
##       Predicted (cv)
## Actual  [,1]  [,2]
##   [1,] 0.895 0.105
##   [2,] 0.237 0.763
```

```
post(r2,filename="RPART.pdf")
```

Here, we are down to 68% 'correct' classifications.

Looking deeper

If we look at the summary of the tree object, it gives us a lot of details about goodness of fit and decsiion points.

```
summary(r2)
```

```
## Call:
## rpart(formula = eng ~ ., data = joint, method = "class", control = c(maxdepth = 3))
##   n= 76
##
##          CP nsplit rel error    xerror      xstd
## 1 0.4473684      0 1.0000000 1.3684211 0.1066392
## 2 0.1052632      1 0.5526316 0.7894737 0.1121371
## 3 0.0100000      3 0.3421053 0.5789474 0.1040442
##
## Variable importance
##     X eer.1  ep.1 ppu.1 ppr.1 eeu.1    ep
##    56    13    11     7     7     4     2
##
## Node number 1: 76 observations,    complexity param=0.4473684
##   predicted class=0  expected loss=0.5  P(node) =1
##     class counts:    38    38
##    probabilities: 0.500 0.500
##   left son=2 (59 obs) right son=3 (17 obs)
##   Primary splits:
##       X     < 101.5     to the right, improve=10.949150, (0 missing)
##       ep.1  < 2049.256  to the left,  improve= 2.514706, (0 missing)
##       eeu.1 < 2431.336  to the left,  improve= 2.326531, (0 missing)
##       ppu.1 < 4579.384  to the left,  improve= 2.072727, (0 missing)
##       eer.1 < 4687.518  to the left,  improve= 1.966874, (0 missing)
##   Surrogate splits:
##       ep.1  < 5045.675  to the left,  agree=0.803, adj=0.118, (0 split)
##       ppr.1 < 5662.658  to the left,  agree=0.789, adj=0.059, (0 split)
##
## Node number 2: 59 observations,    complexity param=0.1052632
##   predicted class=0  expected loss=0.3559322  P(node) =0.7763158
##     class counts:    38    21
##    probabilities: 0.644 0.356
##   left son=4 (17 obs) right son=5 (42 obs)
##   Primary splits:
##       X     < 805.5     to the right, improve=6.050847, (0 missing)
##       ppu   < 0.8333333 to the left,  improve=1.704816, (0 missing)
##       eer.1 < 2742.365  to the right, improve=1.496824, (0 missing)
##       eeu.1 < 2431.336  to the left,  improve=1.446983, (0 missing)
##       ep    < 0.7916667 to the left,  improve=1.437402, (0 missing)
##   Surrogate splits:
##       eer.1 < 1471.599  to the left,  agree=0.746, adj=0.118, (0 split)
##       ppu.1 < 1532.392  to the left,  agree=0.746, adj=0.118, (0 split)
##       eeu.1 < 1666.909  to the left,  agree=0.729, adj=0.059, (0 split)
##       ep.1  < 1831.915  to the left,  agree=0.729, adj=0.059, (0 split)
##       ppr.1 < 1525.105  to the left,  agree=0.729, adj=0.059, (0 split)
##
## Node number 3: 17 observations
##   predicted class=1  expected loss=0  P(node) =0.2236842
```

```
##     class counts:     0    17
##    probabilities: 0.000 1.000
##
## Node number 4: 17 observations
##   predicted class=0  expected loss=0  P(node) =0.2236842
##     class counts:    17     0
##    probabilities: 1.000 0.000
##
## Node number 5: 42 observations,    complexity param=0.1052632
##   predicted class=0  expected loss=0.5  P(node) =0.5526316
##     class counts:    21    21
##    probabilities: 0.500 0.500
##   left son=10 (26 obs) right son=11 (16 obs)
##   Primary splits:
##       eer.1 < 2592.575  to the right, improve=3.230769, (0 missing)
##       ep    < 0.7916667 to the left,  improve=2.100000, (0 missing)
##       eeu   < 0.5       to the left,  improve=1.817308, (0 missing)
##       ppr.1 < 2957.047  to the right, improve=1.817308, (0 missing)
##       X     < 352.5     to the left,  improve=1.767677, (0 missing)
##   Surrogate splits:
##       ep.1  < 2513.859  to the right, agree=0.810, adj=0.500, (0 split)
##       ppu.1 < 3230.144  to the right, agree=0.786, adj=0.437, (0 split)
##       ppr.1 < 2205.929  to the right, agree=0.738, adj=0.313, (0 split)
##       eeu.1 < 2795.064  to the right, agree=0.714, adj=0.250, (0 split)
##       ep    < 0.875     to the left,  agree=0.690, adj=0.187, (0 split)
##
## Node number 10: 26 observations
##   predicted class=0  expected loss=0.3461538  P(node) =0.3421053
##     class counts:    17     9
##    probabilities: 0.654 0.346
##
## Node number 11: 16 observations
##   predicted class=1  expected loss=0.25  P(node) =0.2105263
##     class counts:     4    12
##    probabilities: 0.250 0.750
```

This model seems to fit a lot better than our earlier LDA models, which suggest that it is probably overfitting. Cross-validation can be done within the control parameter:

```
r3 <- rpart(eng~.,data=joint,method="class",control=c(maxdepth=3,xval=10))
r3
```

```
## n= 76
##
## node), split, n, loss, yval, (yprob)
##       * denotes terminal node
##
##  1) root 76 38 0 (0.5000000 0.5000000)
##    2) X>=101.5 59 21 0 (0.6440678 0.3559322)
##      4) X>=805.5 17  0 0 (1.0000000 0.0000000) *
##      5) X< 805.5 42 21 0 (0.5000000 0.5000000)
##       10) eer.1>=2592.575 26  9 0 (0.6538462 0.3461538) *
##       11) eer.1< 2592.575 16  4 1 (0.2500000 0.7500000) *
##    3) X< 101.5 17  0 1 (0.0000000 1.0000000) *
```

```
confusion(joint$eng,predict(r1,type="class"))
```

```
## Overall accuracy = 0.868
##
## Confusion matrix
##       Predicted (cv)
## Actual  [,1]  [,2]
##   [1,] 0.842 0.158
##   [2,] 0.105 0.895
```

```
confusion(joint$eng,predict(r3,type="class"))
```

```
## Overall accuracy = 0.829
##
## Confusion matrix
##       Predicted (cv)
## Actual  [,1]  [,2]
##   [1,] 0.895 0.105
##   [2,] 0.237 0.763
```

Accuracy goes down from 75% to 68%. This 68% is very close to what we achieved in the simple lda models.

Clearly, for partitioning trees, we have to be carefule about overfitting, because we can always easily get the perfect classification.

Random Forests

One advantage of partitioning trees is that they are fast and easy to make. Recently, researchers have been interested in combining the results of many small (and often not-very-good) classifiers to make one better one. This often is described as 'boosting', but there are a number of ways to achieve this. Doing this in a particular way with decision trees is referred to as a 'random forest' (see Breiman and Cutler).

Random forests can be used for both regression and classification (trees can be used in either way as well). A random forest works as follows:

- Build N trees (where N may be hundreds), where each tree is built from a random subset of features/variables For each tree:

1. Pick a subset of the variables.
2. Build a tree

Then, to classify any item, have each tree determine its best guess, and then take the most frequent outcome.

```
library(randomForest)
```

```
## randomForest 4.6-12
```

```
## Type rfNews() to see new features/changes/bug fixes.
```

```
joint$eng <- as.factor(joint$eng)
rf <- randomForest(eng~.,data=joint,importance=TRUE,proximity=TRUE)
rf
```

```
##
## Call:
##  randomForest(formula = eng ~ ., data = joint, importance = TRUE,      proximity = TRUE)
##                Type of random forest: classification
##                      Number of trees: 500
```

```
## No. of variables tried at each split: 3
##
##         OOB estimate of  error rate: 32.89%
## Confusion matrix:
##    0  1 class.error
## 0 26 12   0.3157895
## 1 13 25   0.3421053
```

```
confusion(joint$eng,predict(rf))
```

```
## Overall accuracy = 0.671
##
## Confusion matrix
##       Predicted (cv)
## Actual     0     1
##      0 0.684 0.316
##      1 0.342 0.658
```

```
phone <- read.csv("data_study1.csv")
phone.rf <- randomForest(Smartphone~.,data=phone)
phone.rf
```

```
##
## Call:
##  randomForest(formula = Smartphone ~ ., data = phone)
##                Type of random forest: classification
##                      Number of trees: 500
## No. of variables tried at each split: 3
##
##         OOB estimate of  error rate: 34.03%
## Confusion matrix:
##         Android iPhone class.error
## Android     110    109   0.4977169
## iPhone       71    239   0.2290323
```