Tree Models

Sowmia Narayani Janakiraman & Lauren Kapraun

12/06/18

Document Setup

```r
knitr::opts_chunk$set(echo = TRUE)
library(readr)
library(rpart)
library(dplyr)
library(ggplot2)
install.packages("party")
install.packages("tree")
install.packages("rpart.plot")
library(party)
library("rpart.plot")
loan = read.csv(file = "loan3000.csv", header = TRUE, stringsAsFactors=FALSE)
data <- read_csv("cardiotocography.csv")
```

The Tree Model

Decision trees are a type of supervised learning algorithm that can be used in both regression and classification problems. It works for both categorical and continuous input and output variables. The tree model is a set of "if-then-else"" rules that can discover hidden patterns corresponding to complex interactions in data. The two main packages to fit tree models are `rpart` and `tree`.

To build your first decision trees, we will proceed as follow:

- Step 1: Import the data
- Step 2: Clean the dataset
- Step 3: Create a train/test set
- Step 4: Build the model
- Step 5: Make a prediction
- Step 6: Measure performance

Before you train your model, you need to perform two steps: Create a train and test set: You train the model on the train set and test the prediction on the test set (i.e. unseen data)

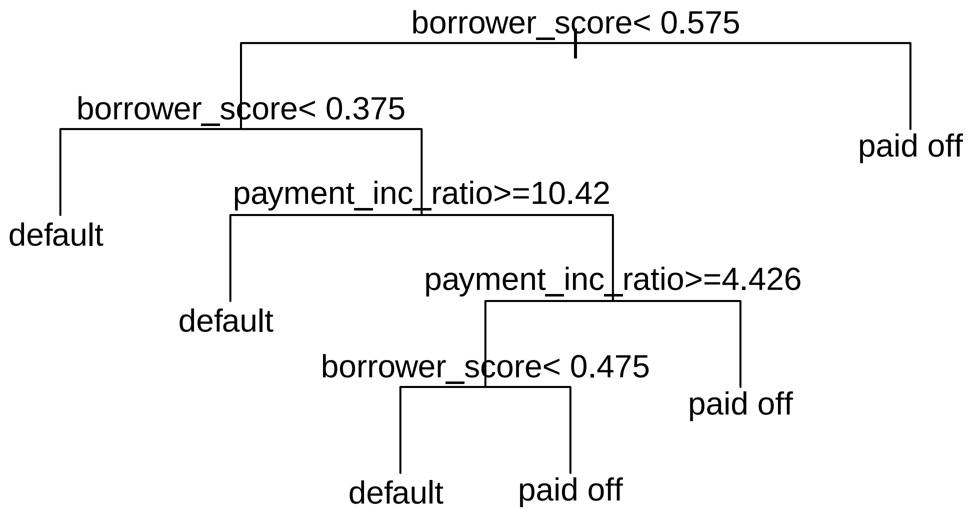
Display Tree

When you display the text tree, the depth of the tree is shown by the indent. The values in the parentheses correspond to the proportion of records that are paid off and default. The classification rules are determined by traveling through a hierarchical tree, starting at the root and moving **left if the node is true** and **right if the node is false**, until a leaf is reached.

```r
# Textbook Example
loan$outcome = ordered(loan$outcome, levels=c('paid off', 'default'))

# loan_tree = rpart(formula, data, control)
```

```r
loan_tree = rpart(outcome ~ borrower_score + payment_inc_ratio, data=loan,
                  control = rpart.control(cp=.005))

plot(loan_tree, uniform=TRUE, margin=.05)
loan_tree
text(loan_tree)
```

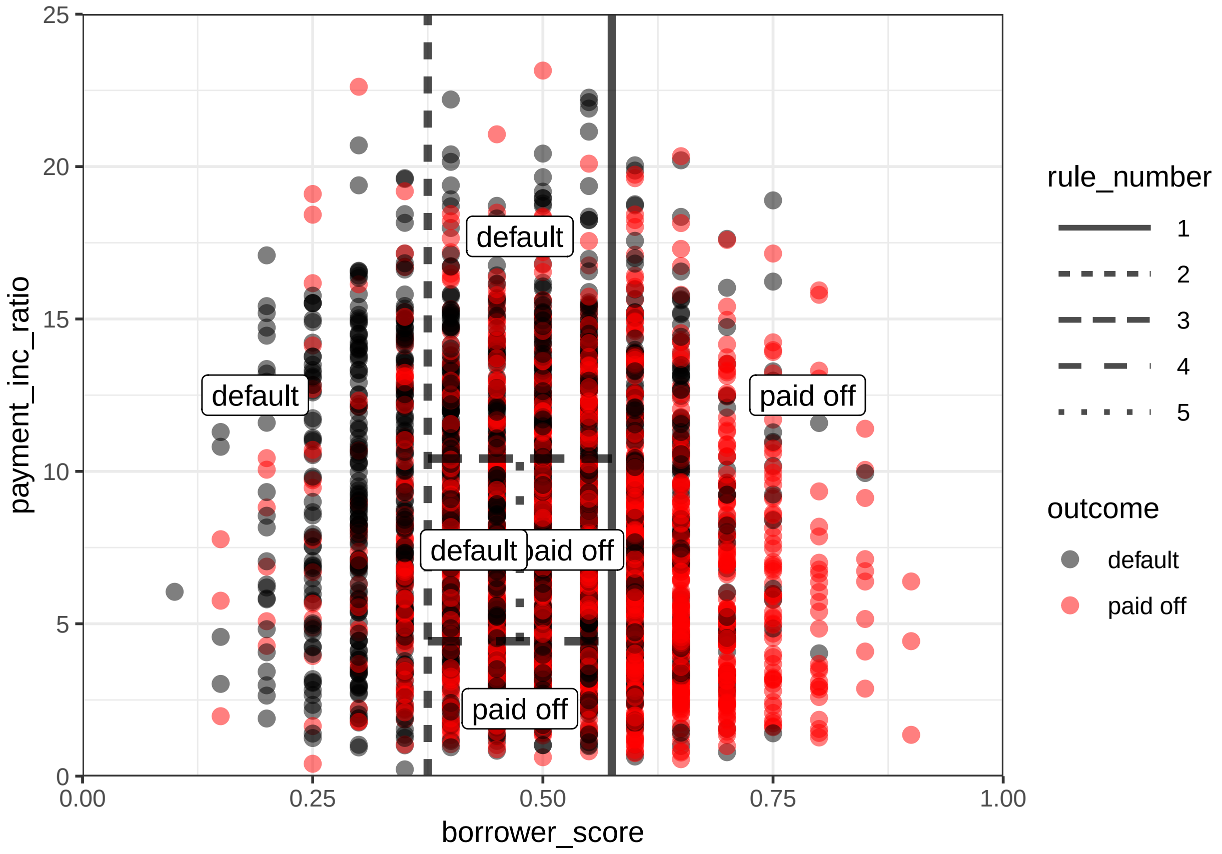
```
## n= 3000
##
## node), split, n, loss, yval, (yprob)
##       * denotes terminal node
##
##  1) root 3000 1445 paid off (0.4816667 0.5183333)
##    2) borrower_score< 0.575 2122  938 default (0.5579642 0.4420358)
##      4) borrower_score< 0.375 483  136 default (0.7184265 0.2815735) *
##      5) borrower_score>=0.375 1639  802 default (0.5106772 0.4893228)
##       10) payment_inc_ratio>=10.42265 482  192 default (0.6016598 0.3983402) *
##       11) payment_inc_ratio< 10.42265 1157  547 paid off (0.4727744 0.5272256)
##         22) payment_inc_ratio>=4.42601 823  408 paid off (0.4957473 0.5042527)
##           44) borrower_score< 0.475 405  187 default (0.5382716 0.4617284) *
##           45) borrower_score>=0.475 418  190 paid off (0.4545455 0.5454545) *
##         23) payment_inc_ratio< 4.42601 334  139 paid off (0.4161677 0.5838323) *
##    3) borrower_score>=0.575 878  261 paid off (0.2972665 0.7027335) *
```



The Recursive Partitioning Algorithm

Recursive Partitioning is an algorithm to construct a decision tree. The data is repeatedly partitioned using predictor values that do the best job of separating the data into relatively homogeneous partitions.

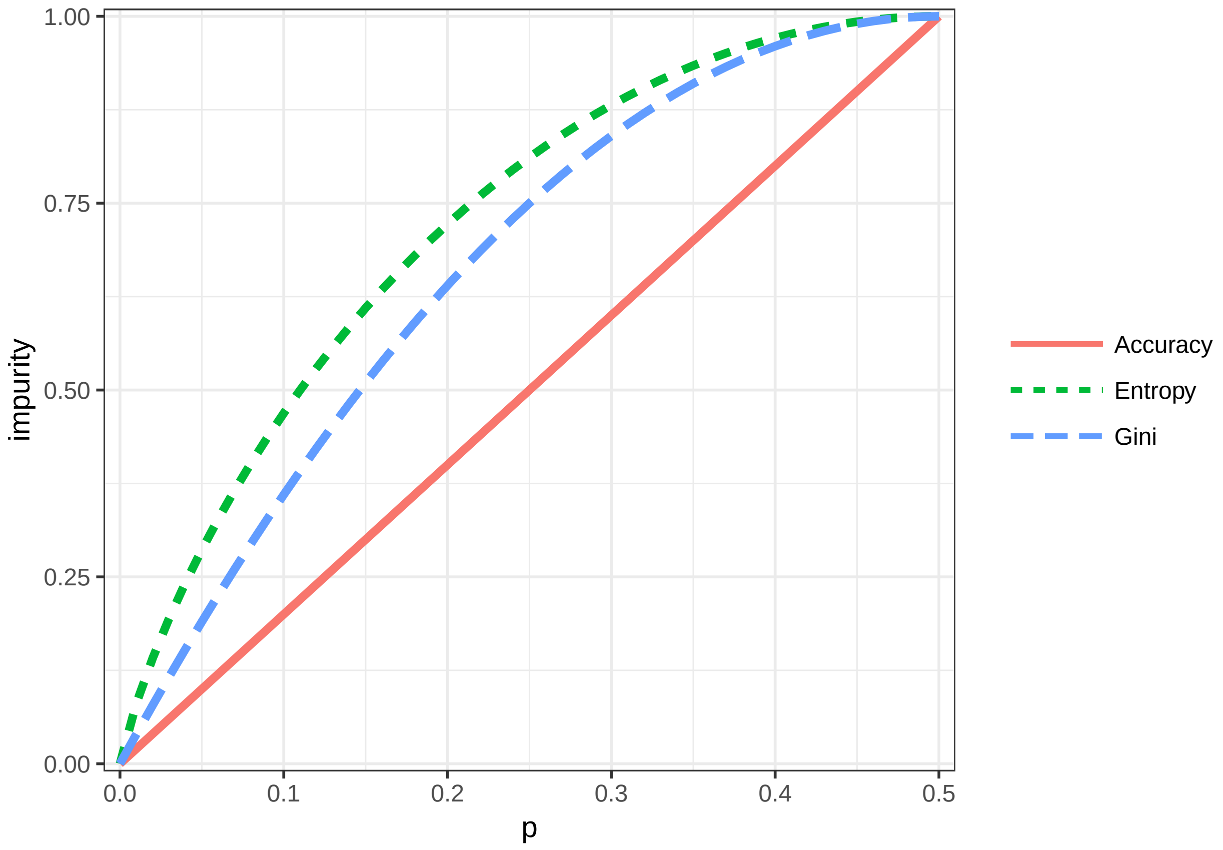
```r
# Textbook Example
r_tree = data_frame(x1 = c(0.575, 0.375, 0.375, 0.375, 0.475),
                    x2 = c(0.575, 0.375, 0.575, 0.575, 0.475),
                    y1 = c(0, 0, 10.42, 4.426, 4.426),
                    y2 = c(25, 25, 10.42, 4.426, 10.42),
                    rule_number = factor(c(1, 2, 3, 4, 5)))
r_tree = as.data.frame(r_tree)
labs = data.frame(x=c(.575 + (1-.575)/2, .375/2, (.375 + .575)/2, (.375 + .575)/2,
                     (.475 + .575)/2, (.375 + .475)/2 ),
                  y=c(12.5, 12.5, 10.42 + (25-10.42)/2, 4.426/2, 4.426 + (10.42-4.426)/2,
                      4.426 + (10.42-4.426)/2 ),
                  decision = factor(c('paid off', 'default', 'default',
                                      'paid off', 'paid off', 'default')))

ggplot(data=loan, aes(x=borrower_score, y=payment_inc_ratio)) +
  geom_point( aes(color=outcome), size=2.5, alpha=.5) +
  scale_color_manual(values=c('black', 'red')) +
  geom_segment(data=r_tree, aes(x=x1, y=y1, xend=x2, yend=y2, linetype=rule_number),
               size=1.5, alpha=.7) +
  guides(colour = guide_legend(override.aes = list(size=2.5)),
         linetype = guide_legend(keywidth=3, override.aes = list(size=1))) +
  scale_x_continuous(expand=c(0,0), limits=c(0, 1)) +
  scale_y_continuous(expand=c(0,0), limits=c(0, 25)) +
  geom_label(data=labs, aes(x=x, y=y, label=decision)) +
  theme_bw()
```



Measuring Homogeneity or Impurity

```
# Textbook Example

# Two common measures of impurity are the Gini Impurity & Entropy of Information.
x = 0:50/100
```

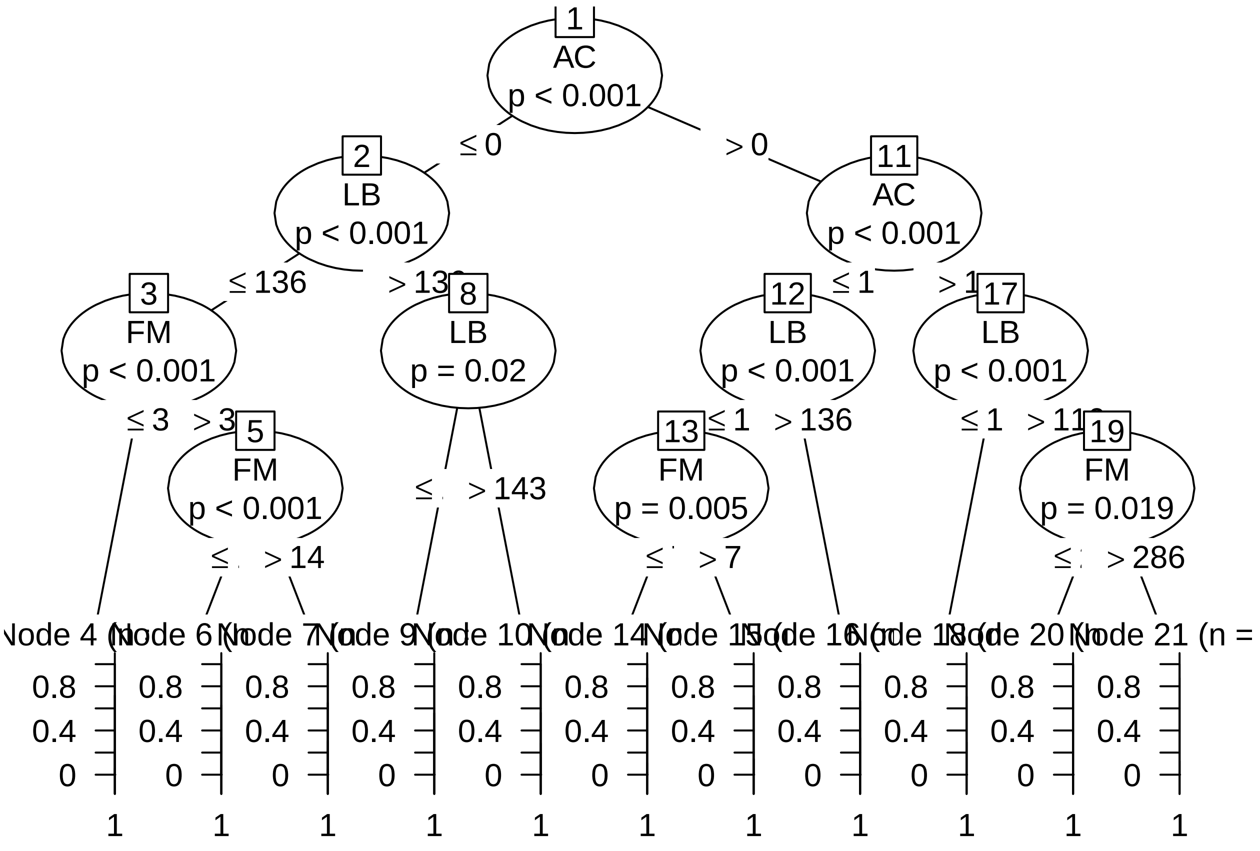
```r
# The Gini Impurity for a set of records A is: I(A) = p(1 -- p)
gini = function(x){
  return(x * (1-x))
}

# The Entropy measure is given by: I(A) = -p log2(p) - (1 - p) log2(1 - p)
info = function(x){
  info = ifelse(x==0, 0, -x * log2(x) - (1-x) * log2(1-x))
  return(info)
}

impure = data.frame(p = rep(x, 3),
                     impurity = c(2*x, gini(x)/gini(.5)*info(.5), info(x)),
                     type = rep(c('Accuracy', 'Gini', 'Entropy'), rep(51,3))
                    )

ggplot(data=impure, aes(x=p, y=impurity, linetype=type, color=type)) +
  geom_line(size=1.5) +
  guides(linetype = guide_legend(keywidth=3, override.aes = list(size=1))) +
  scale_x_continuous(expand=c(0,0.01)) +
  scale_y_continuous(expand=c(0,0.01)) +
  theme_bw() +
  theme(legend.title=element_blank())
```



Splitting

Splits are formed on a particular variable and in a particular location. For each split, two determinations are made: the predictor variable used for the split, called the splitting variable, and the set of values for the predictor variable (which are split between the left child node and the right child node), called the split point. The split is based on a particular criterion, for example, Gini (for classification) or sums of squares (for regression) from the entire data set. The leaf node, also called a terminal node, contains a small subset of the observations. Splitting continues until a leaf node is constructed.

```r
data$NSPF = factor(data$NSP)
data <- data[, c(24, 1:3)]
set.seed(1234)

#Splitting data into training data and testing data
train <- sample(2, nrow(data), replace = TRUE, prob = c(0.7, 0.3))
trainData <- data[train==1,]
testData <- data[train==2,]

#Tree model using training data
treeMod <- NSPF ~ (.)
data_ctree <- ctree(treeMod, trainData)
plot(data_ctree)
```



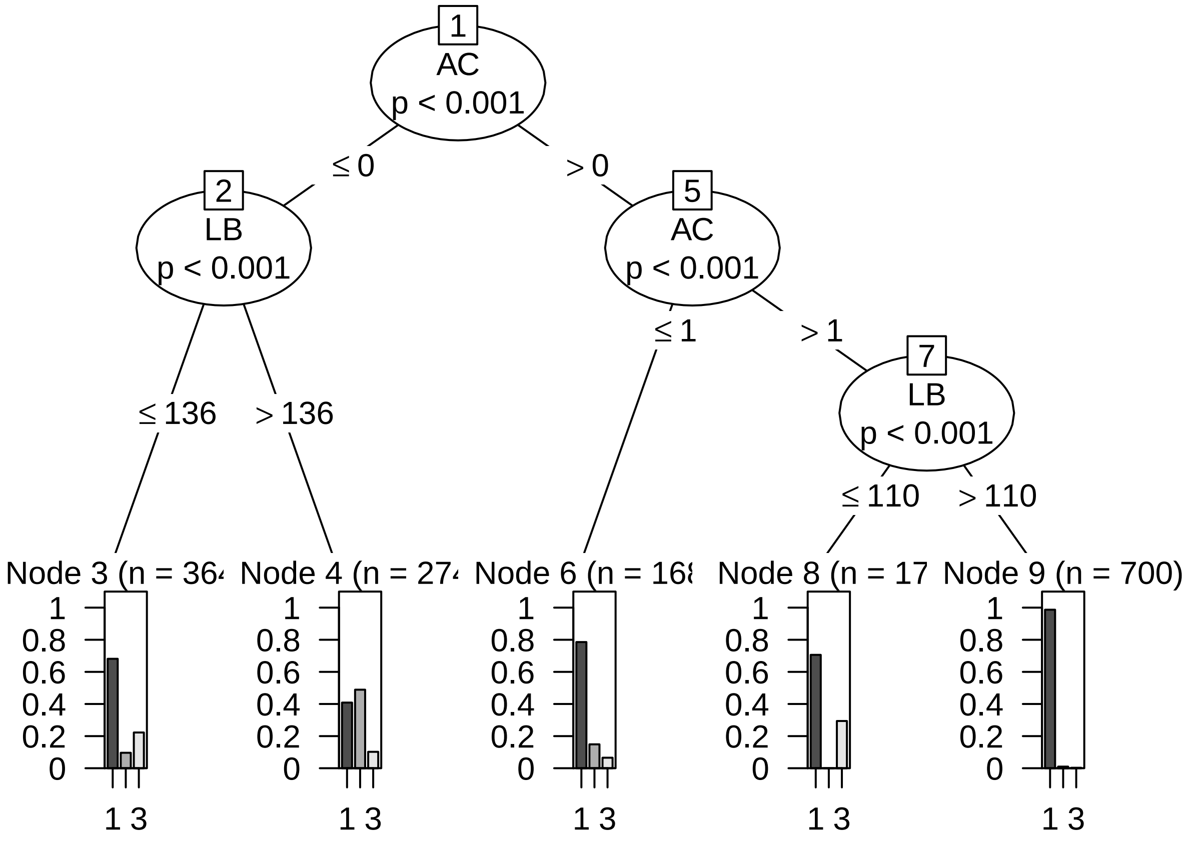
Stopping the Tree from Growing

A simple method of reducing tree size is **prune** back the terminal and smaller branches of the tree. A common technique is to prune the tree back to the point where the error on the holdout data is minimized.

Pruning plays a role in the process of cross-validation to determine how far to grow trees that are used in ensemble methods.

The shortening of branches of the tree. Pruning is the process of reducing the size of the tree by turning some branch nodes into leaf nodes and removing the leaf nodes under the original branch. Pruning is useful because classification trees may fit the training data well, but may do a poor job of classifying new values. Lower branches may be strongly affected by outliers. Pruning enables you to find the next largest tree and minimize the problem. A simpler tree often avoids over-fitting.

```
pruned_ctree <- ctree(treeMod, trainData,
                      controls = ctree_control(mincriterion = 0.99, minsplit = 500))
plot(pruned_ctree)
```



To Check How the Pruned Model is Doing

```
ptab<-table(predict(pruned_ctree), trainData$NSPF)
print(ptab)
```

```
##
##         1    2    3
##   1  1083   67   99
##   2   112  134   28
##   3     0    0    0
```

```
1-sum(diag(ptab))/sum(ptab)
```

```
## [1] 0.2009192
```

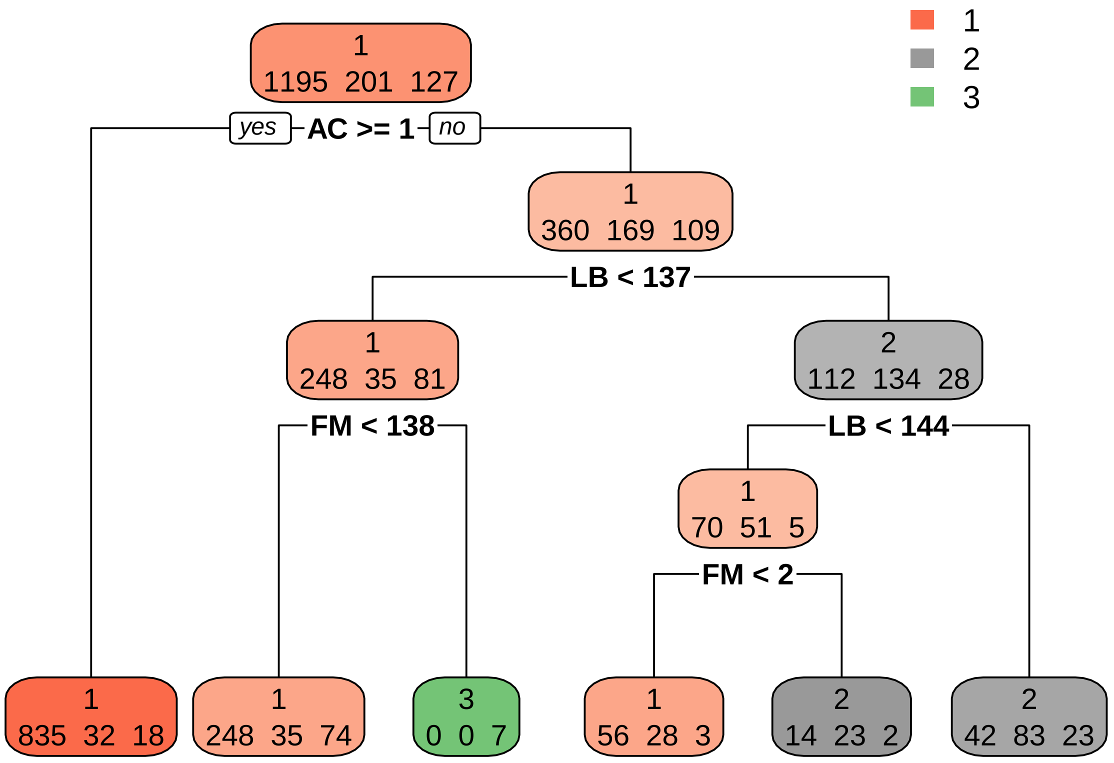
```r
ptestPred <- predict(pruned_ctree, newdata = testData)
ptab1 <- table(ptestPred, testData$NSPF)
print(ptab1)
```

```
##
## ptestPred   1   2   3
##         1 424  29  41
##         2  36  65   8
##         3   0   0   0
```

```r
1-sum(diag(ptab1))/sum(ptab1)
```

```
## [1] 0.1890547
```

```r
rtree <- rpart(NSPF ~ LB+AC+FM, trainData)
rpart.plot(rtree, extra = 1)
```



```r
rtreePred <- predict(rtree, trainData, type = 'vector')
rtab<-table(rtreePred, trainData$NSPF)
print(rtab)
```

```
##
## rtreePred    1    2    3
##         1 1139   95   95
##         2   56  106   25
##         3    0    0    7
```

```r
1-sum(diag(rtab))/sum(rtab)
```

```
## [1] 0.1779383
```

```r
rtreePred1 <- predict(rtree, testData, type = 'vector')
rtab1<-table(rtreePred1, testData$NSPF)
print(rtab1)
```

```
##
## rtreePred1   1   2   3
##          1 441  41  43
##          2  19  52   5
##          3   0   1   1
```

```r
1-sum(diag(rtab1))/sum(rtab1)
```

```
## [1] 0.1807629
```

minsplit & minbucket VS. cp

Avoid splitting a partition if a resulting subpartition is too small, or if a terminal leaf is too small. In `rpart`, these constraints are controlled separately by the parameters `minsplit` and `minbucket`, respectively with defaults 20 and 7.

```r
# Textbook Example
# defaults are 20 and 7
minsplit = 1500 # Set minsplit to any large number

loan_tree2 = rpart(outcome ~ borrower_score + payment_inc_ratio, data=loan,
                   control = rpart.control(minsplit))

loan_tree3 = rpart(outcome ~ borrower_score + payment_inc_ratio, data=loan,
                   control = rpart.control(minbucket = round(minsplit/3)))

loan_tree # Display original text tree
```

```
## n= 3000
##
## node), split, n, loss, yval, (yprob)
##       * denotes terminal node
##
##  1) root 3000 1445 paid off (0.4816667 0.5183333)
##    2) borrower_score< 0.575 2122  938 default (0.5579642 0.4420358)
##      4) borrower_score< 0.375 483  136 default (0.7184265 0.2815735) *
##      5) borrower_score>=0.375 1639  802 default (0.5106772 0.4893228)
##       10) payment_inc_ratio>=10.42265 482  192 default (0.6016598 0.3983402) *
##       11) payment_inc_ratio< 10.42265 1157  547 paid off (0.4727744 0.5272256)
##         22) payment_inc_ratio>=4.42601 823  408 paid off (0.4957473 0.5042527)
##           44) borrower_score< 0.475 405  187 default (0.5382716 0.4617284) *
##           45) borrower_score>=0.475 418  190 paid off (0.4545455 0.5454545) *
##         23) payment_inc_ratio< 4.42601 334  139 paid off (0.4161677 0.5838323) *
##    3) borrower_score>=0.575 878  261 paid off (0.2972665 0.7027335) *
```

```r
loan_tree2 # Display text tree with minsplit
```

```
## n= 3000
##
## node), split, n, loss, yval, (yprob)
##       * denotes terminal node
```

```
##
## 1) root 3000 1445 paid off (0.4816667 0.5183333)
##   2) borrower_score< 0.575 2122  938 default (0.5579642 0.4420358)
##     4) borrower_score< 0.425 837  287 default (0.6571087 0.3428913) *
##     5) borrower_score>=0.425 1285  634 paid off (0.4933852 0.5066148) *
##   3) borrower_score>=0.575 878  261 paid off (0.2972665 0.7027335) *
```

```
loan_tree3 # Display text tree with minbucket
```

```
## n= 3000
##
## node), split, n, loss, yval, (yprob)
##       * denotes terminal node
##
## 1) root 3000 1445 paid off (0.4816667 0.5183333)
##   2) borrower_score< 0.575 2122  938 default (0.5579642 0.4420358)
##     4) borrower_score< 0.425 837  287 default (0.6571087 0.3428913) *
##     5) borrower_score>=0.425 1285  634 paid off (0.4933852 0.5066148) *
##   3) borrower_score>=0.575 878  261 paid off (0.2972665 0.7027335) *
```

cp

Don't split a partition if the new partition does not "significantly" reduce the impurity. In `rpart`, this is controlled by the **complexity parameter** `cp`, which is a measure of how complex a tree is – the more complex, the greater the value of `cp`. In practice, `cp` is used to limit tree growth by attaching a penalty to additional complexity (splits) in a tree.

In `rpart` you can use the argument `cptable` to produce a table of the `cp values` and their associated cross-validation error (`exerror` in R).

```
# Textbook Example
# printcp(data, digits = getOption("digits") - 2)
printcp(loan_tree, digits = getOption("digits") - 2)
```

```
##
## Classification tree:
## rpart(formula = outcome ~ borrower_score + payment_inc_ratio,
##     data = loan, control = rpart.control(cp = 0.005))
##
## Variables actually used in tree construction:
## [1] borrower_score    payment_inc_ratio
##
## Root node error: 1445/3000 = 0.48167
##
## n= 3000
##
##         CP nsplit rel error  xerror     xstd
## 1 0.170242      0   1.00000 1.00000 0.018940
## 2 0.021799      1   0.82976 0.82976 0.018567
## 3 0.010727      3   0.78616 0.82284 0.018541
## 4 0.005000      5   0.76471 0.80692 0.018477
```

Predicting a Continuous Value

Also called **regression**, predicting a continuous value with a tree follows the same logic and procedure, except that impurity is measured by squared deviations from the mean (squared errors) in each subpartition, and predictive performance is judged by the square root of the mean squared error (RMSE) in each partition.