Classification and Regression Trees

Clay Ford

Spring 2016

In this workshop

- What are and how to read classification and regression trees
- How to create classification and regression trees in R
- Overview of how trees are constructed
- How to interpret output from building trees
- How to improve prediction accuracy of trees
- Trees versus linear models / Strengths and Weaknesses

What are Classification and Regression Trees?

They are basically *decision trees* that make predictions based on binary splits.

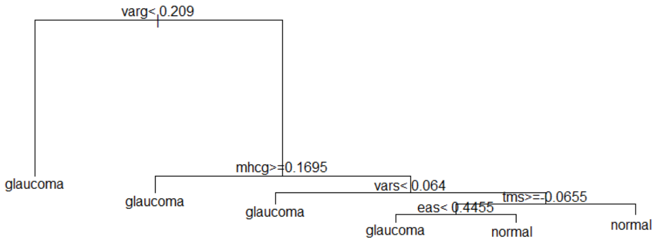
Classification trees predict a classification (categorical response).

Regression trees predict means or medians (continuous response).

They are non-parametric in nature in that they don't make distributional assumptions about the response or prediction errors.

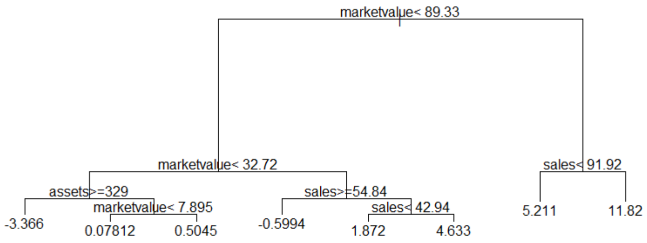
Example of Classification Tree

Using laser scanner data to predict whether a patient's eye suffers from glaucoma or not.



Example of Regression Tree

Explaining profit of a company based on assets, sales and market value.



How to read the trees

Start at the top of the tree, called the *root node*:

- go left if condition is true
- otherwise go right

Repeat process at subsequent *nodes* until reaching a *terminal node*, or *leaf*. The value in the leaf is the predicted value.

Hence classification and regression trees tend to be easier to interpret than traditional linear modeling output.

However, that does not mean they always perform better than linear models.

Using R to build trees

There are several packages available for building decision trees in R. The package we'll use in this workshop is `rpart`, which is short for "Recursive Partitioning".

The main function is `rpart`. The basic steps to fitting and plotting a tree are:

1. `fit <- rpart(response ~ x1 + x2 + x3..., data=DF)` where DF is your data frame and x1, x2, ... are your predictors.
2. `plot(fit)`
3. `text(fit)`

If `response` is a factor, then a classification tree is built. Otherwise a regression tree is built.

Plotting tree branches

Calling `plot` on an rpart object produces the tree without labels. Additional arguments of note include

- `uniform`: is vertical spacing between nodes uniform? The default is FALSE.
- `branch`: number between 0 and 1 that controls the shape of branches. 1 draws square-shouldered branches, 0 draws V-shaped branches, and values in between draw a combination. Default is 1.
- `compress`: should routine attempt a more compact arrangement of tree? Default is FALSE.
- `margin`: extra fraction of white space to leave around the borders of the tree. Default is 0.

Adding text to tree branches

Calling `text` on an rpart object adds text labels to tree branches. Additional arguments of note include

- `use.n`: show number of observations in leaves? Default is FALSE.
- `all`: label all nodes with additional information? Default is FALSE.
- `cex`: character expansion, expressed as a fraction of 1 (normal size). Greater than 1, bigger text; less than 1, smaller text.

Let's go to R!

How trees are constructed

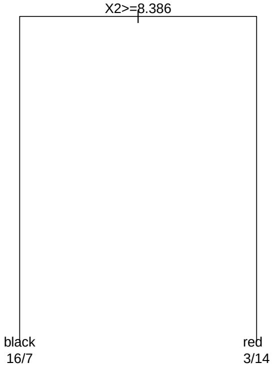
The basic process:

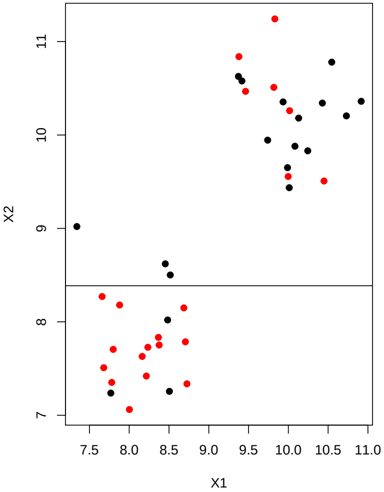
1. Examine all possible splits for all possible covariates and choose the split which leads to 2 groups that are “purer” than the current group. That is, take the split that produces the largest improvement in *purity*.
2. Repeat step 1 for the 2 new groups. Hence the name *recursive partitioning*.
3. Repeat steps 1 and 2 for all subsequent groups until some stopping criterion is fulfilled.

In the `rpart` package, a node must have at least 20 observations for it to attempt a split. This is a setting that can be modified.

A 2-dimensional example of classification tree

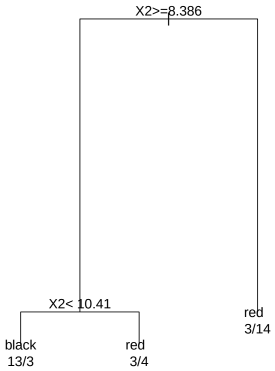
Predict black or red classification based on X1 and X2 values. The prediction is based on majority in terminal node.

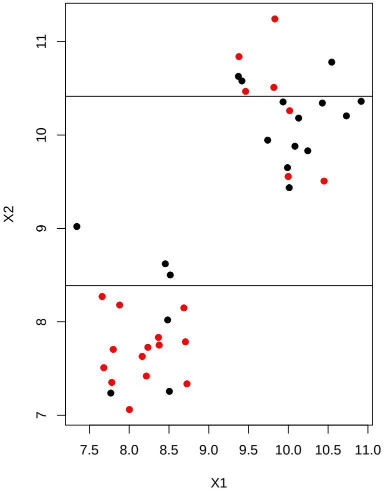




A 2-dimensional example of classification tree

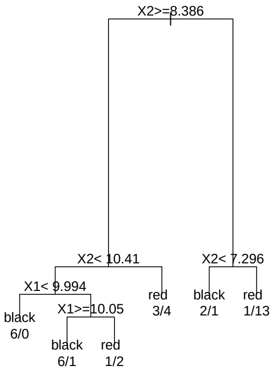
Predict black or red classification based on X1 and X2 values. The prediction is based on majority in terminal node.

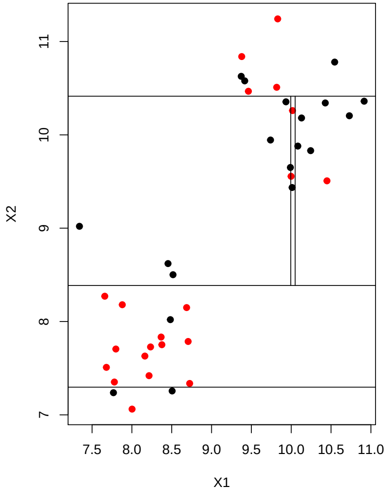




A 2-dimensional example of classification tree

Predict black or red classification based on X1 and X2 values. The prediction is based on majority in terminal node.





Measures of impurity - Classification Trees

For classification trees, the usual measure of impurity is the *Gini index*:

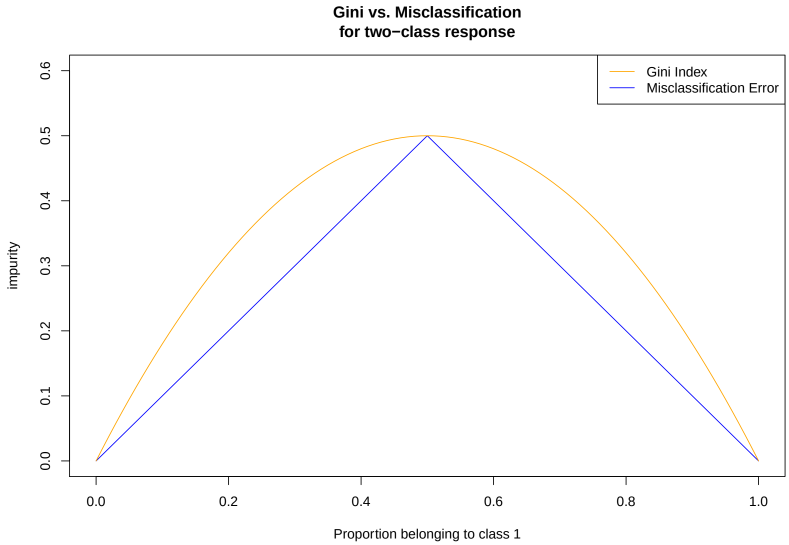
$$G = \sum_{k=1}^{K} \hat{p}_{mk}(1 - \hat{p}_{mk})$$

where K is the number of classes and $\hat{p}_{mk}$ is the proportion of observations in region m that are from class k.

This is basically a measure of misclassification.

Why use the Gini index?

Simply put, it's more *sensitive* than misclassification rate:



Gini index vs misclassification rate example

Let's say we have a two-class response with 400 observations in each class (400, 400).
Suppose when growing a tree we have two splits to consider:

1. nodes with (300, 100) and (100, 300)
2. nodes with (200, 400) and (200, 0)

Both splits produce a misclassification rate of $200/800 = 0.25$.

But the second split produces a pure node and is probably preferable. The Gini index captures this:
$(300/400) \times (100/400) + (300/400) \times (100/400) = 0.375$
$(200/600) \times (400/600) + (200/200) \times (0/200) = 0.222$

The second split has nodes with lower *impurity*.

Measuring improvement in purity - Classification Trees

A tree split in `rpart` is selected based on the greatest improvement in purity.

The improvement in purity is calculated as follows:

$$\Delta I = n[p(A)I(A) - p(A_L)I(A_L) - p(A_R)I(A_R)]$$

where. . .

- $p(A)$ is the proportion of observations in the current node
- $p(A_L)$ and $p(A_R)$ are the proportion of observations sent to the left and right child nodes, respectively
- $I(A)$, $I(A_L)$ and $I(A_R)$ are the measures of impurity in each node (ie, the Gini Index)
- n is the total number of observations.

Measuring improvement in purity - example cont'd

Let's say we have a two-class response with 400 observations in each class (400, 400).

Suppose when growing a tree from the root node we have two splits to consider:

1. nodes with (300, 100) and (100, 300)
2. nodes with (200, 400) and (200, 0)

The second has the largest improvement in purity:

$$800[(1)(0.5)(0.5) - (0.5)(300/400)(100/400) - (0.5)(300/400)(100/400)] = 50$$

$$800[(1)(0.5)(0.5) - (0.75)(200/600)(400/600) - (0.25)(200/200)(0/200)] = 66.66667$$

Measures of impurity - Regression Trees

For regression trees, the usual measure of purity is the Residual Sum of Squares (RSS). We select a split such that the RSS is minimized:

$$RSS = \sum_{j=1}^{J} \sum_{i \in R_j} (y_i - \hat{y}_{R_j})^2$$

where J is the number of regions, R_j are the regions of predictors, and $\hat{y}_{R_j}$ is the mean response for the *Jth* region.

Measuring improvement in purity - Regression Trees

The improvement in purity as displayed in `rpart` is calculated as follows:

$$\Delta I = 1 - (SS_{right} + SS_{left})/SS_{parent}$$

where SS is the respective sum of squares for the parent and children nodes: $\sum(y_i - \bar{y})^2$.

The summary of tree construction

- A summary of the tree construction process can be viewed with the `summary` function. Beware, it produces a lot of output!
- It displays the *primary* split for each node as well as up to four "runner-up" splits (the splits that came closest to the chosen split in terms of improving purity).
- It also displays five *surrogate* splits. These are splits for observations that are missing data for the primary split.
- Any observation missing the primary split variable is classified using the first surrogate split, or if missing that, the second surrogate, and so on.

Let's go to R.

Pruning trees

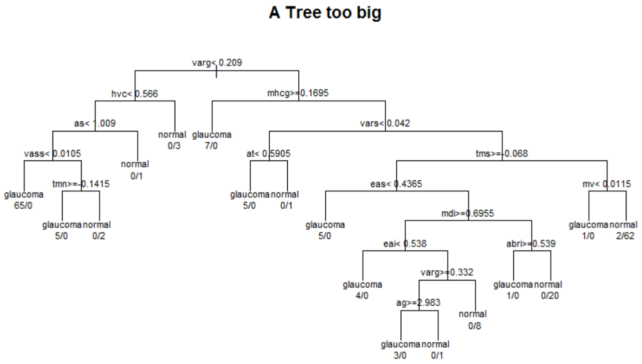
Once we grow a tree, we usually need to *prune* it. That is, we need to trim off the bottom branches.

A tree that is too large can overfit the data and not perform well for data not used in growing the tree.

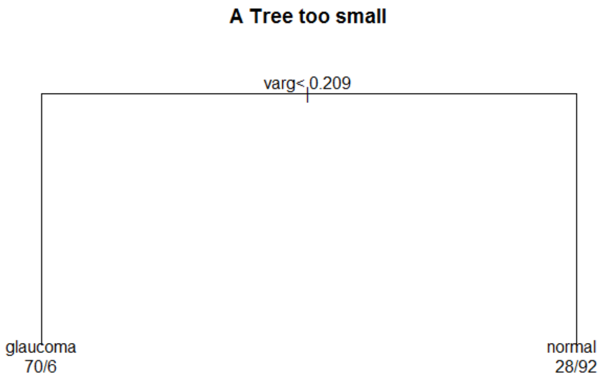
On the other hand, a tree that is too small will not include important predictors and also not perform well.

Therefore pruning a tree can pose a challenge.

A Large Tree - overfits data



A Small Tree - underfits data



How pruning works

We build a large tree, T, and then recursively snip off the least important splits.

Importance is captured by the cost-complexity measure:

$$R_\alpha(T') = R(T') + \alpha|T'|$$

where α is the *complexity parameter* ranging from 0 to ∞, $R(T')$ is the *risk* of the subtree T', and $|T'|$ is the number of terminal nodes (leaves) in the subtree. Thus $R_\alpha(T')$ is the risk of a subtree for a given α.

For any specified α, cost-complexity pruning determines the subtree T' that minimizes $R_\alpha(T')$ over all subtrees of T, the original large tree.

The complexity parameter, α

α measures the *cost* of adding another variable to the tree. In other words, there's a penalty to pay for adding more branches to a tree. α is the price you pay to add more branches.

$$R_\alpha(T') = R(T') + \alpha|T'|$$

If α is small, the penalty is small and the tree will be large. The minimal value is achieved by having more terminal nodes (leaves).

If α is large, the penalty is large and the tree will be small. The minimal value is achieved by having fewer terminal nodes (leaves).

Selecting the optimal α

`rpart` use K-fold cross validation to find the optimal α. That is, it divides our data into K folds (or sets), and then for each K:

1. Hold out set K and grow a large tree with the other folds
2. Obtain a sequence of best subtrees as a function of α
3. Evaluate the error for the left out set K (misclassification rate for classification trees; mean squared prediction error for regression trees)
4. Average the error results for each α

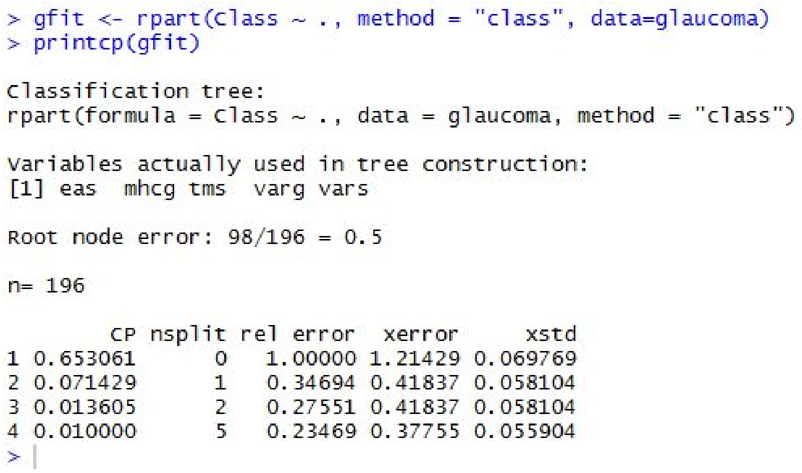
`rpart` does this with a default of $K = 10$. When finished we select α with the lowest error rate and use that to prune the tree.

Pruning results in `rpart`

The cross-validation results can be viewed with the `printcp` function. It prints 5 columns of numbers:

- `CP`: the complexity parameter α scaled by the number of misclassifications in a model with no splits.
- `nsplit`: number of splits in tree
- `rel error`: relative error (on data used to build tree)
- `xerror`: cross-validated error rate (on held out data)
- `xstd`: cross-validated standard error

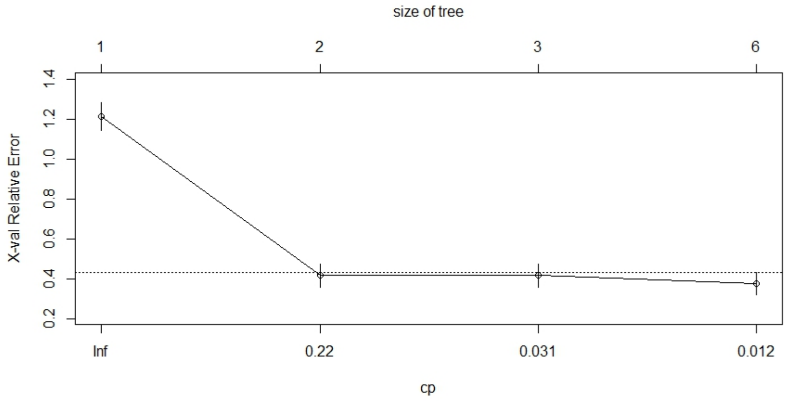
printcp example



```
> gfit <- rpart(Class ~ ., method = "class", data=glaucoma)
> printcp(gfit)

Classification tree:
rpart(formula = Class ~ ., data = glaucoma, method = "class")

Variables actually used in tree construction:
[1] eas  mhcg tms  varg vars

Root node error: 98/196 = 0.5

n= 196

        CP nsplit rel error  xerror     xstd
1 0.653061      0   1.00000 1.21429 0.069769
2 0.071429      1   0.34694 0.41837 0.058104
3 0.013605      2   0.27551 0.41837 0.058104
4 0.010000      5   0.23469 0.37755 0.055904
> 
```

plotcp example

Use the `plotcp` function to plot `xerror` vs. `CP`. Example:
`plotcp(gfit)`



Selecting a CP and pruning a tree

Recommendation: Pick the CP within one standard error of the minimum.

To prune a tree, use the `prune` function on the rpart object with a `cp` argument specifying the complexity parameter.

For example, to prune a tree using complexity parameter of 0.031:

```
gfit.prune <- prune(gfit, cp=0.031)
```

Let's go to R.

Making predictions

Use the `predict` function to run data through your tree and make predictions.

For classification trees there are three types of predictions you can make:

- `predict(tree, type="prob")`: predicted probabilities (default)
- `predict(tree, type="class")`: predicted class using level label
- `predict(tree, type="vector")`: predicted class using level number

For regression trees, calling `predict(tree)` returns predicted means.

Making predictions with data not used to build tree

To make predictions for data not used to build tree, use the `newdata=` argument.

The new data must be a data frame and include all variables used in building the tree *even if the tree only includes a subset of the variables*.

Training and test data

If our tree is intended for prediction purposes, it's a good idea to build the tree using only some of the data (say half), and evaluate its predictive performance using the held out data (the other half).

Training data - data used to build the tree
Test data - data used to evaluate the performance of the tree

The training and test data must be selected randomly.

Let's go to R.

Improving prediction accuracy of trees

Classification and Regression trees tend to suffer from *high variance*. That is they tend to produce different sized trees using different samples from the same population.

As we'll demonstrate in the R script, pruning with cross validation does not always result in pruning the same size tree.

Two approaches to reducing this variability are *bagging* and *random forests*.

Bagging

"Boostrap aggregation", or *Bagging*, means resampling our data B times, building B trees, and then averaging all the predictions.

"Resampling" means sampling *with replacement* from our original data until we have a data set the same size as our original data.

Two keys:

1. We do not prune our trees. We grow them deep and use them to make predictions.
2. We make predictions using observations that were not used to grow the tree. These are called *out-of-bag* (OOB) predictions.

For regression trees, the average is simply the mean of the B predicted means. For classification trees, the prediction is the most commonly occurring prediction (ie, *majority vote*)

Random Forests

Random forests are similar to bagging, except *we only consider a random sample of m predictors at each split*.

A fresh sample of predictors is taken at each split.

We usually use $m \approx \sqrt{p}$, that is the number of predictors at each step is about equal to the square root of the total number of predictors. When $m = p$, we have bagging.

Sampling predictors at each split prevents strong predictors from dominating the tops of trees and gives moderately-strong predictors a chance to perform. Think of this as allowing some of the "runner-ups" in the summary results to have a turn as a primary split.

Implementation of bagging and random forests

The `randomForest` package allows you to easily implement bagging and random forests. The syntax is very similar to `rpart`.

Bagging:

```
bag.tree <- randomForest(response ~ ., data=DF,
mtry=p)
```

Random forest:

```
rf.tree <- randomForest(response ~ ., data=DF)
```

The `mtry=` argument specifies how many predictors to sample at each split. Recall $m = p$ is bagging. The default for classification is $\sqrt{p}$ while the default for regression is $p/3$.

The drawback of bagging and random forests

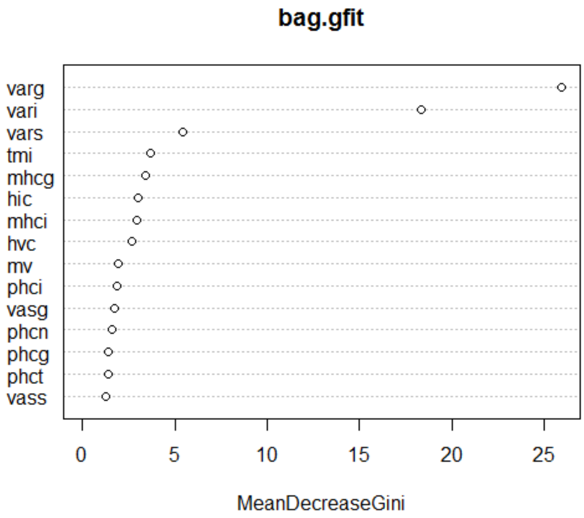
While these two approaches reduce variability and improve performance, they sacrifice the ease of interpretability.

Instead of one tree we can easily interpret, we now have many trees (usually 500) that we are using as an *ensemble*.

One way around this is to create a plot of *variable importance*.

This plot shows the most important predictors based on the total *decrease in node impurity* that results from splits over the variable.

Example of variable importance plot



Let's go to R.

Trees versus linear models

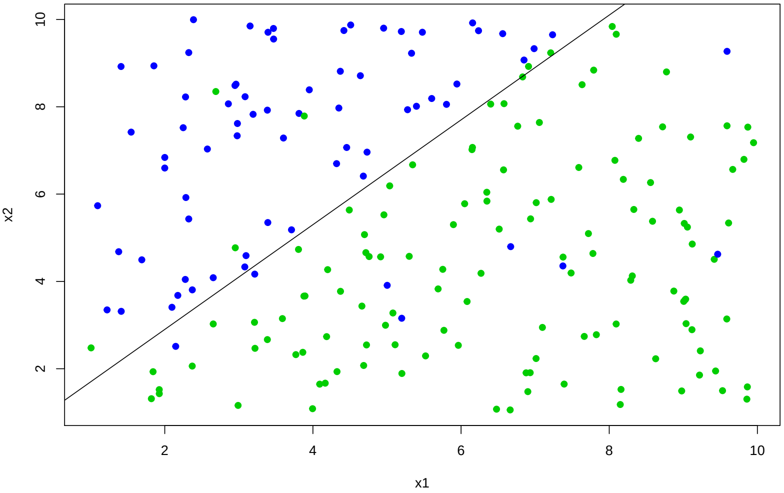
When it comes to classification problems, if the true decision boundary is linear, then a linear model will likely outperform a tree-based model.

On the other hand, if true decision boundary is non-linear, then a tree-model may outperform a linear model.

Another consideration is interpretability. A decision tree is easier to read and understand than linear model output. Thus we may be willing to sacrifice small gains in performance for an easier to understand model.

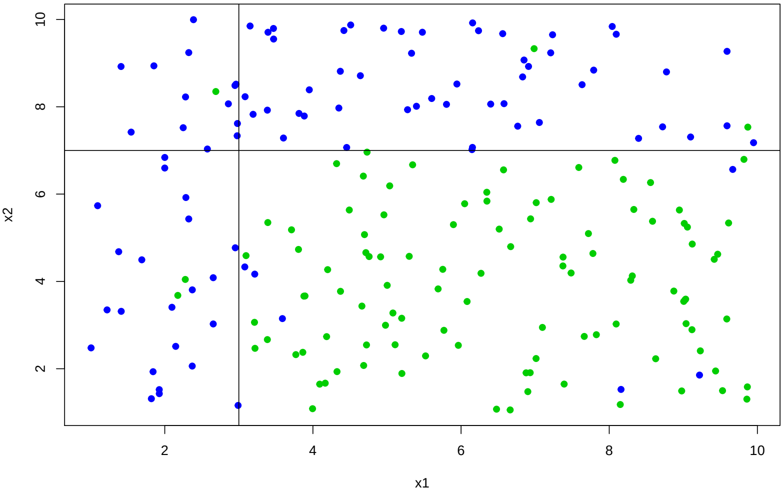
Linear decision boundary - 2D classification

A linear model will likely outperform a tree model in this situation.



Non-linear decision boundary - 2D classification

A tree model will likely outperform a linear model in this situation.



Strengths of trees

- Handles missing values without omitting complete observations
- Can capture non-additive behavior; automatically includes interactions
- Handles both regression and classification
- Results are easy to understand

Weaknesses of trees

- Tree may not be optimal
- Larger trees can make poor intuitive sense
- Continuous predictor variables are dichotomized
- May obscure relationships that are obvious from linear modeling output

References

Everitt, B. and Hothorn, T. (2006) *A Handbook of Statistical Analyses using R*. Chapman & Hall/CRC, Boca Raton. (Ch. 8)

Hastie, T. et al. (2009) *The Elements of Statistical Learning*. Springer, New York. (pages 308 - 310)

James, G. et al. (2013) *An Introduction to Statistical Learning*. Springer, New York. (Ch. 8)

Maindonald, J. and Braun, W. J. (2010) *Data Analysis and Graphics Using R*. Cambridge, UK. (Ch. 11)

Therneau, T. and Atkinson, E. (2014) *An Introduction to Recursive Partitioning Using the RPART Routines*. Included with `rpart` package.

StatLab

Thanks for coming today!

For help and advice with your data analysis, contact the StatLab to set up an appointment: `statlab@virginia.edu`

Sign up for more workshops or see past workshops:
`http://data.library.virginia.edu/training/`

Register for the Research Data Services newsletter to stay up-to-date on RDS events and resources:
`http://data.library.virginia.edu/newsletters/`