Chapter 8: Tree-Based Methods

→ non parametric supervised.

We will introduce *tree-based* methods for regression (↗ quantitative response Y) and classification (↗ categorical response Y).

These involve segmenting the predictor space into a number of simple regions
↳ all values $X_1, \ldots, X_p$ can take.

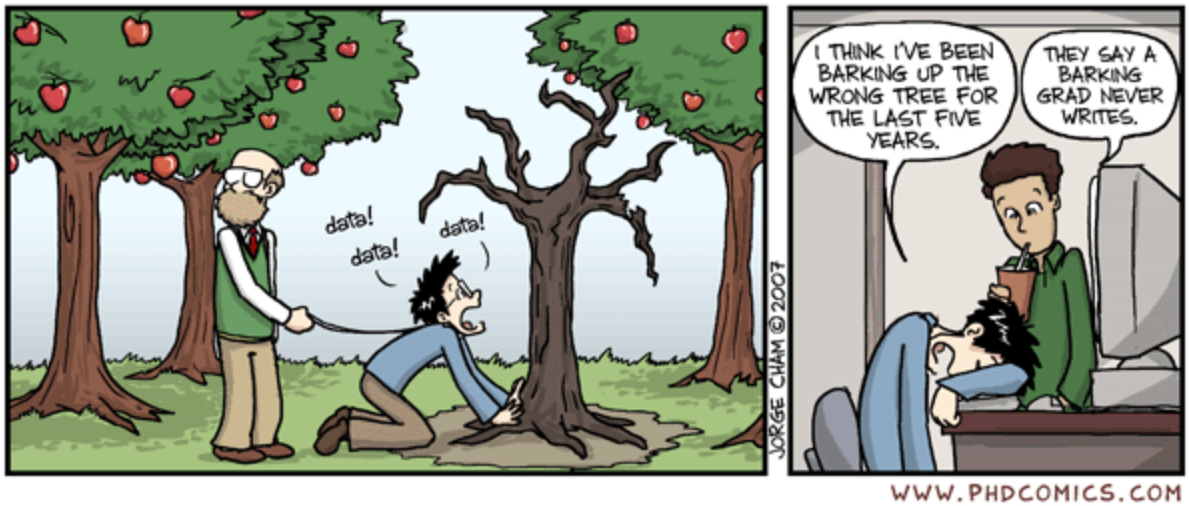
To make a prediction for an observation, we use the mean (↓ regression) or mode (↘ classification) of training observations in the region to which it belongs.

The set of splitting rules can be summarized in a tree ⇒ "decision trees".

- simple and useful for interpretation.
- not competitive w/ other supervised approaches (e.g. lasso) for prediction.

↗ boosting, bagging, random forests (later)

Combining a large number of trees can often result in dramatic improvements in prediction accuracy at the expense of interpretation.



Credit: http://phdcomics.com/comics.php?f=852

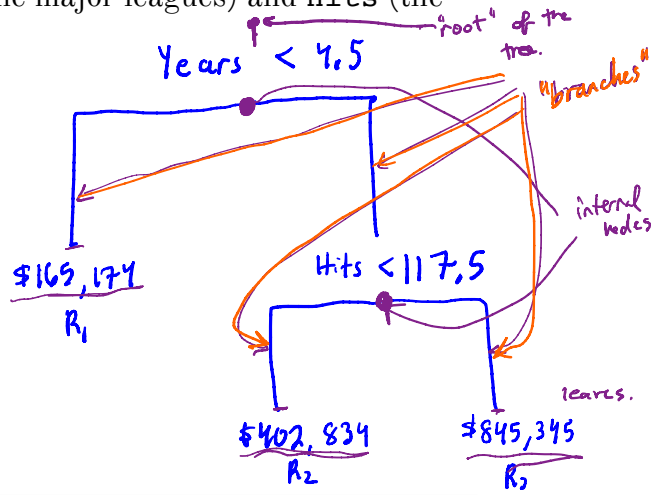
Decision trees can be applied to both regression and classification problems. We will start with regression.

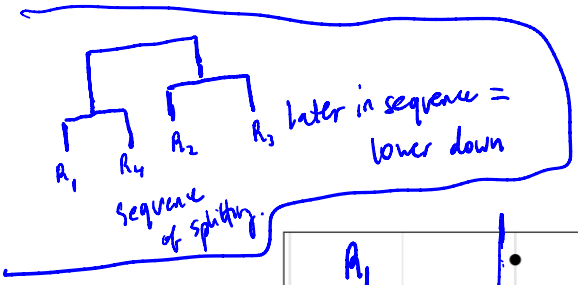
1 Regression Trees

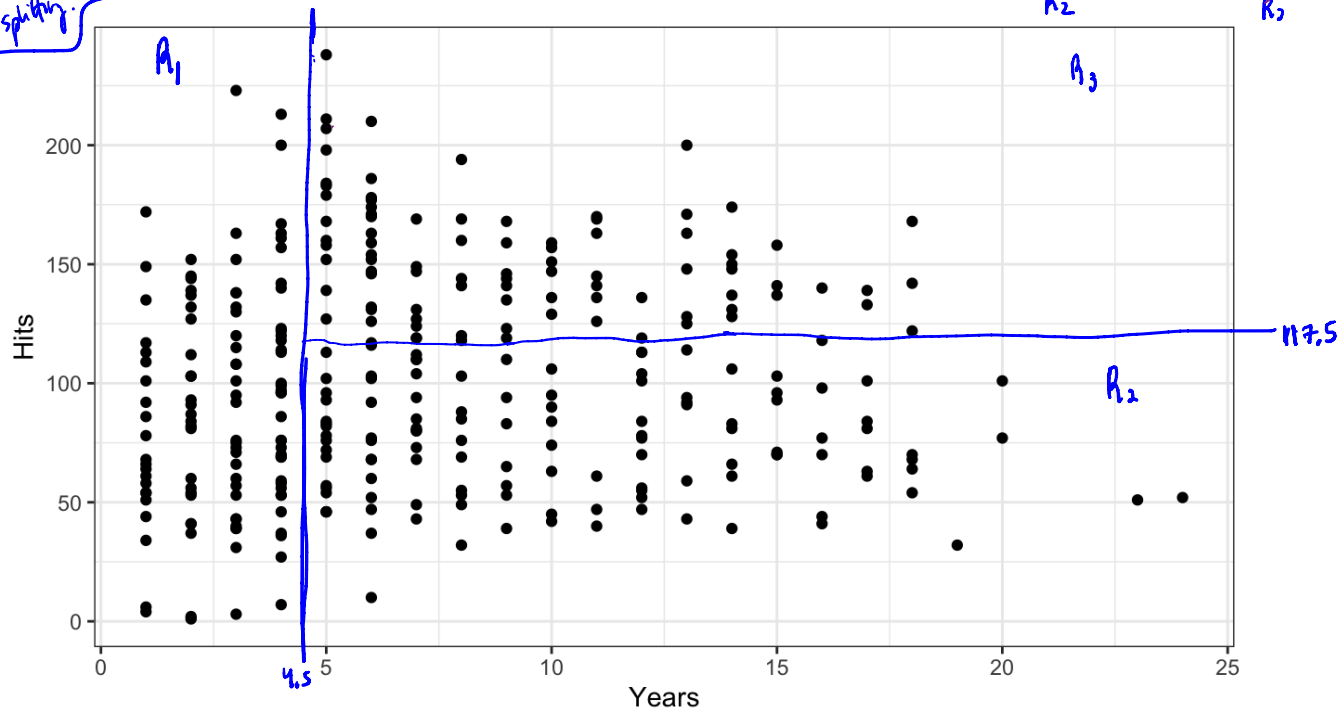
start with

Example: We want to predict baseball salaries using the `Hitters` data set based on `Years` (the number of years that a player has been in the major leagues) and `Hits` (the number of hits he made the previous year).

we can make a series of splitting rules (according to a tree fit to this data) to create regions and predict salary as the mean in each region.







The predicted salary for players is given by the mean response value for the players in that box. Overall, the tree segments the players into 3 regions of predictor space.

Terminology R_1, R_2, R_3 = "terminal nodes" or "leaves" of the tree.

points along tree where predictor space is split = "internal nodes"

segments of tree that connect nodes = "branches"

Interpretation Years is most important factor in determining salary.

↳ given that a player has less experience (< 4.5 years), # hits plays very little role in his salary

↳ among player who had been in the league 5+ years, # hits does affect salary: ↑ hits ↑ salary.

probably an oversimplification but it is easy to interpret & has nice graphical representation.

↗ quantitative response Y.

We now discuss the process of building a regression tree. There are two steps:

1. Divide predictor space (↗ set of possible values for $X_1, \ldots, X_p$)
   into J distinct and non-overlapping regions $R_1, \ldots, R_J$

2. Predict
   For every observation that falls into the region R_j we make the same prediction
   = mean of response Y for every training observation in R_j.

How do we construct the regions $R_1, \ldots, R_J$? How to divide the predictor space?
regions could have any shape, but that's too hard (to do & to interpret).
⇒ divide predictor space into high dimensional rectangles ("boxes").

The goal is to find boxes $R_1, \ldots, R_J$ that minimize the RSS. $= \sum_{j=1}^{J} \sum_{i \in R_j} (y_i - \hat{y}_{R_j})^2$ where
$\hat{y}_{R_j}$ = mean response of training data in R_jth box.

Unfortunately it is computationally infeasible to consider every possible partition.

⇒ take top-down, greedy approach called recursive binary splitting.

The approach is *top-down* because
We start at top of the tree (where all observations belong to a single region) and successively split the predictor space.

each split is indicated via two new branches down the tree.

The approach is *greedy* because
at each step in the building process, the best split is made at that particular step.
↳ not looking ahead to make a split that will lead to a better tree later.

In order to perform recursive binary splitting,

① Select the predictor and cutpoint s s.t. splitting the predictor space into regions $\{X \mid X_j < s\}$ and $\{X \mid X_j \geq s\}$ leads to greatest possible reduction in RSS.

↑ region of predictor space where X_j takes values $< s$.

↳ consider all possible $X_1, \dots, X_p$ and cutpoints s (based on training data), then choose predictor & cutpoint that result in lowest RSS.

i.e. consider all possible half planes $R_1(j,s) = \{X \mid X_j < s\}$ and $R_2(j,s) = \{X \mid X_j \geq s\}$

we seek j, s that minimize

$$\sum_{i \in x_i \in R_1(j,s)} (y_i - \hat{y}_{R_1})^2 + \sum_{i \in x_i \in R_2(j,s)} (y_i - \hat{y}_{R_2})^2$$

← finding j, s can be quickly done when p not too large.

② Repeat process, looking for next best j, s combo but instead of splitting entire space, split $R_1(j,s)$ or $R_2(j,s)$ to minimize RSS.

③ Continue until stopping criteria is met, i.e. no region contains more than 5 obs.

④ predict using mean of training observations in the region to which test observation falls.

The process described above may produce good predictions on the training set, but is likely to overfit the data.

because the resulting tree may be too complex.

A smaller tree, with less splits (→ less regions $R_1, \dots, R_J$) might lead to lower variance and better interpretation at the cost of a little bias.

Idea: only split the tree if it resulted in "large enough" drop in RSS.

↑

bad idea because a seemingly "worthless" split early in the tree might be followed by a good split later (large drop in RSS).

A strategy is to grow a very large tree T_0 and then *prune* it back to obtain a *subtree*.

better idea

How to prune the tree?

goal: select a subtree that leads to lowest test error rate. → could use CV to estimate the error for every possible subtree (large # of potential subtrees). This is expensive!

solution: "cost complexity pruning" aka "weakest link pruning".

Consider a sequence of trees indexed by a nonnegative tuning parameter α

For each value of α, $\exists$ a corresponding subtree $T \subset T_0$ s.t. penalizing complexity in the tree

$$\sum_{m=1}^{|T|} \sum_{x_i \in R_m} (y_i - \hat{y}_{R_m})^2 + \alpha |T| \text{ is as small as possible.}$$

($|T|$ = # of terminal nodes in tree T)

R_m = mth terminal node region

$\hat{y}_{R_m}$ = predicted response in R_m

when $\alpha = 0$, $T = T_0$

α controls trade off between subtree complexity & fit to training data ↗ $\alpha \uparrow \Rightarrow$ pay for having many terminal nodes $\Rightarrow$ smaller subtree.

Select α via CV, then use full dataset & chosen α to get subtree T.

Algorithm for building a regression tree:

(1) Use recursive binary splitting to grow a large tree on training data, stopping only when each terminal node has fewer than some min. # of observations.

(2) Use cost complexity pruning to get a sequence of best trees as a function of α.

(3) Use k-fold CV to choose α

Divide training data into K folds, for each $k = 1, \ldots, K$

(a) Repeat (1) and (2) on all data but k^{th} fold.

(b) evaluate the predicted MSE on k^{th} fold as a function of α.

Average results for each value of α and pick α that minimizes CV error.

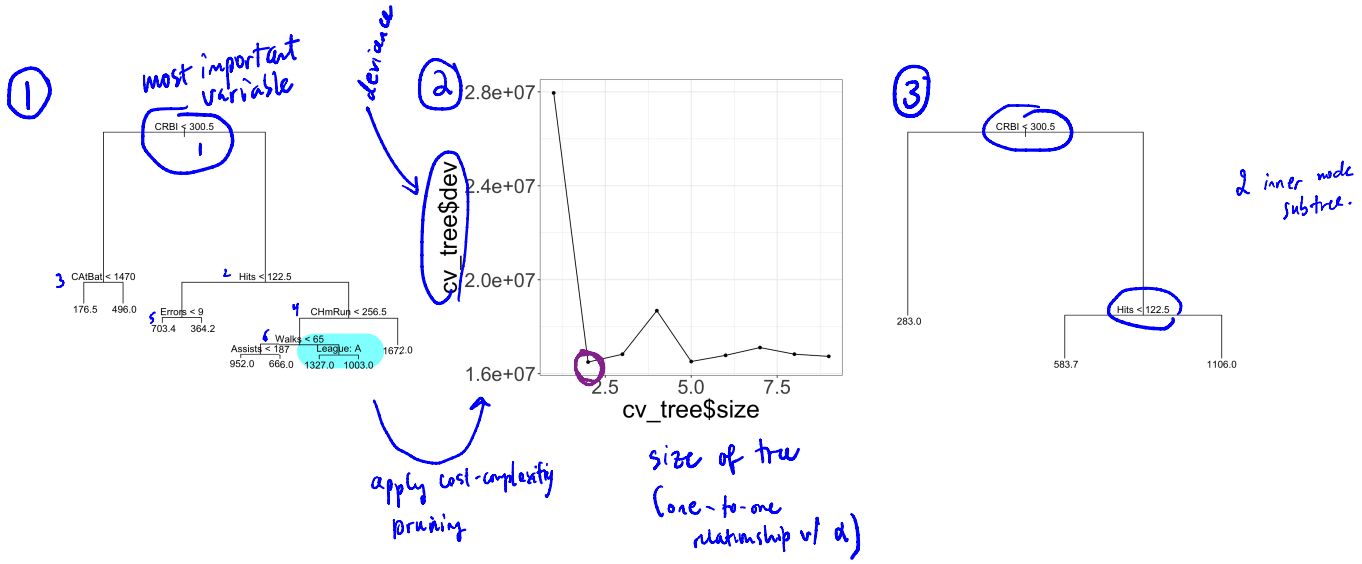
(4) Return the subtree from (2) that corresponds to chosen α from (3).

Example: Fit regression tree to subset of Hitters using 9 features. predicting salary.

(1) is the large tree

(2) CV error to estimate test MSE as a function of α

(3) subtree selected.



2 Classification Trees

A *classification tree* is very similar to a regression tree, except that it is used to predict a categorical response.

Recall for regression tree, the predicted response for an observation is given by the mean response of the training observations that belong to the same terminal node.

For a classification tree, we predict that each observation belongs to the *most commonly occurring class* of training observation in the region to which it belongs.

the mode

We are also often interested in the class prediction proportions that fall into each terminal node.
↳ this can give us some idea of how reliable the prediction is:

e.g. terminal node — training data in terminal node is 100% Class 1 vs. 55% Class 1, 45% Class 2 — "node purity"

both terminal nodes will predict as "Class 1"

The task of growing a classification tree is quite similar to the task of growing a regression tree.

Use recursive binary splitting to grow a classification tree.

But RSS cannot be used as criterion for splitting.

Instead, natural alternative is classification error rate.
= fraction of training observations that do not belong to the most common class.
$= 1 - \max_k \{\hat{p}_{mk}\}$

$\hat{p}_{mk}$: proportion of training observations in the m^{th} region from k^{th} class.

It turns out that classification error is not sensitive enough to use for growing the tree.

preferred measures: (as splitting criteria)

more sensitive to node purity than classification error rate.

① Gini index $G = \sum_{k=1}^{K} \hat{p}_{mk}(1 - \hat{p}_{mk})$ measure of total variance across K classes.
↳ will take small values if all $\hat{p}_{mk}$'s are close to zero or one ⇒ good measure of node purity, ↓ G ⇒ nodes contain primarily obs. from 1 class.

② Entropy $D = -\sum_{k=1}^{K} \hat{p}_{mk} \log \hat{p}_{mk}$
↳ will take small values if $\hat{p}_{mk}$ close to 0 or 1 ⇒ ↓D when nodes are more pure.

Gini and entropy are actually quite similar.

When building a classification tree, either the Gini index or the entropy are typically used to evaluate the quality of a particular split.

Any of 3 methods (classification error, Gini, or entropy) can also be used for pruning (minimize: measure of variance in data + $\alpha|T|$)

But if prediction accuracy of final pruned tree is the goal, classification error rate should be used.

Note: neither Gini nor entropy work well w/ unbalanced classes in data.

There are other options out there to split on.

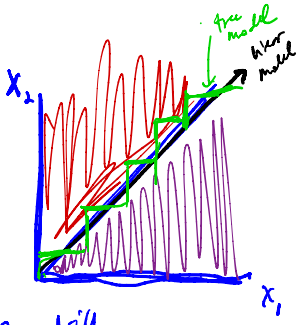
3 Trees vs. Linear Models

Regression and classification trees have a very different feel from the more classical approaches for regression and classification.

eg. linear regression: $f(X) = \beta_0 + \sum_{j=1}^{p} X_j \beta_j$

regression tree: $f(X) = \sum_{m=1}^{M} c_m \mathbb{I}(X \in R_m)$.

where $R_1, \ldots, R_M$ are partitions of the feature space.



Which method is better? It depends on the problem.

- If the true relationship between features and response is approximately linear, will out perform a regression tree.
- If highly nonlinear and complex relationships, decision trees may be better.

Also, trees may be preferred because of interpretation and visualization.

3.1 Advantages and Disadvantages of Trees

Advantages

- easy to explain, even easier than linear regression.
- some people think decision trees more closely mirror human decision making.
- can be displayed graphically, easy to interpret for non expert (especially if small).
- can handle categorical predictors without creating dummy variables.

Disadvantages

- do not have same level of predictive performance as other methods we've seen.
- Not robust: small changes in data can have large changes in estimated tree (high variability)

↓

We can aggregate many trees to try and improve this! (Next).

4 Bagging

Decision trees suffer from *high variance*.

Bootstrap aggregation or *bagging* is a general-purpose procedure for reducing the variance of a statistical learning method, particularly useful for trees.

So a natural way to reduce the variance is to take many training sets from the population, build a separate prediction model using each training set, and average the resulting predictions.

Of course, this is not practical because we generally do not have access to multiple training sets.

While bagging can improve predictions for many regression methods, it's particularly useful for decision trees.

These trees are grown deep and not pruned.

How can bagging be extended to a classification problem?

4.1 Out-of-Bag Error

There is a very straightforward way to estimate the test error of a bagged model, without the need to perform cross-validation.

4.2 Interpretation

5 Random Forests

Random forests provide an improvement over bagged trees by a small tweak that decorrelates the trees.

As with bagged trees, we build a number of decision trees on bootstrapped training samples.

In other words, in building a random forest, at each split in the tree, the algorithm is not allowed to consider a majority of the predictors.

The main difference between bagging and random forests is the chouce of predictor subset size m.

6 Boosting

Boosting is another approach for improving the prediction results from a decision tree.

While bagging involves creating multiple copies of the original training data set using the bootstrap and fitting a separate decision tree on each copy,

Boosting does not involve boostrap sampling, instead each tree is fit on a modified version of the original data set.

Boosting has three tuning parameters:

1.

2.

3.