supervised, nonparametric

Chapter 8: Tree-Based Methods

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

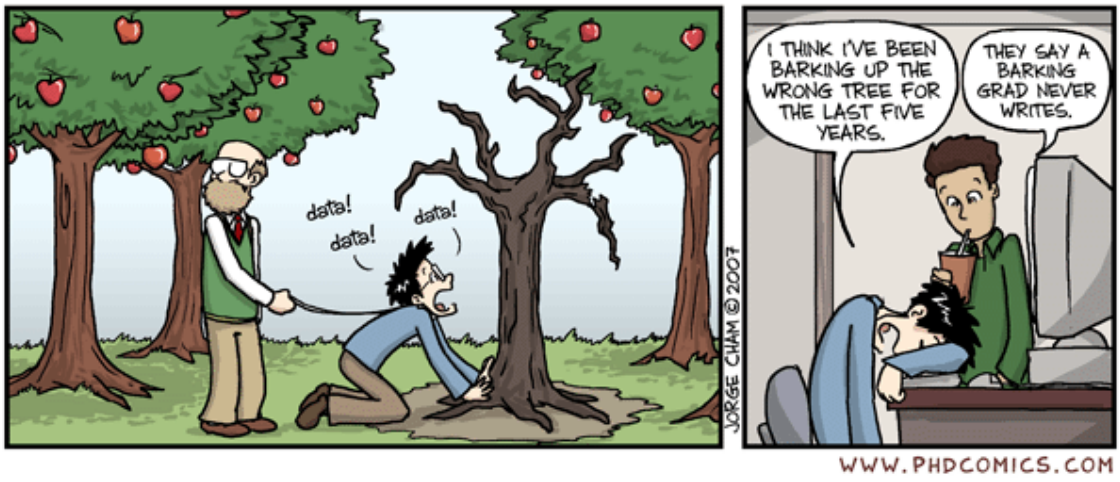
These involve segmenting the predictor space into a number of simple regions.

to make a prediction for an observation, we use mean or mode of training data 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 supervise approaches (e.g. lasso) for prediction.

Combining a large number of trees (bagging, random forests, boosting. later) can often result in dramatic improvements in prediction accuracy at the expense of interpretation.



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

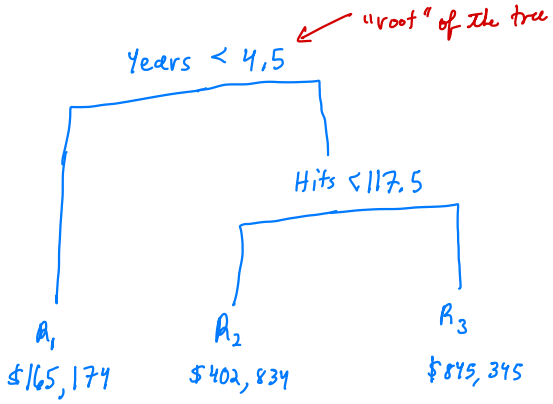
Decision trees can be applied to both regression and classification problems. We will start with regression.

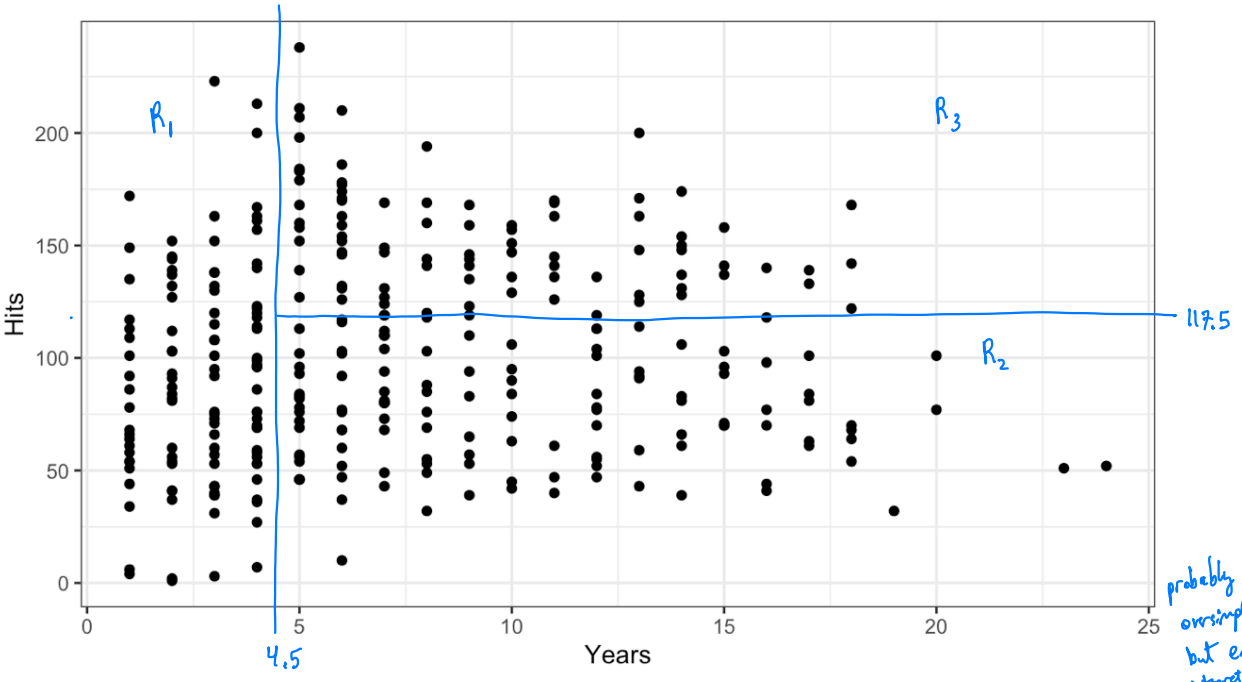
1 Regression Trees

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 to create regions and predict salary as the mean in each region.





probably an oversimplification but easy to interpret & nice visual graphical representation

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 the tree where predictor space is split = internal nodes

segments of tree that connect nodes = branches

interpretation: Years is the most important factor in determining salary

↳ given that a player has less experience, # hits in previous year plays little role in his salary.

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

↗ quantitative Y

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

↗ set of possible values for $X_1, \ldots, X_p$.

1. Divide the predictor space

   into J distinct and non-overlapping regions $R_1, \ldots, R_J$

2. Predict

   For every observation that falls into region R_j we make the same prediction,

   the mean of response Y for training values 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 is too hard (to do + to interpret)

$\Rightarrow$ divide predictor space into high dimensional rectangles (or boxes)

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

↙ mean response of training data in R_j

Unfortunately it is computationally infeasible to solve.

(b/c consider every possible partition)

$\Rightarrow$ take top-down, greedy approach called recursive binary splitting.

The approach is *top-down* because

We start at the 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 of the tree 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,

region of predictor space where X_j takes values less than s.

(1) Select the predictor and cut point s s.t. splitting the predictor space into regions $\{X: X_j < s\}$ and $\{X: X_j \geq s\}$ leads to greatest possible reduction in RSS.

↳ we consider all possible $X_1, \dots, X_p$ and all possible cut points

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 and s that minimise

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

this can be quickly done when p (or unique values of x_j) not too large.

(2) Repeat process, looking 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 observations)

(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 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 tree if it results in a large enough drop in RSS.

bad idea because a seemingly worthless split early in tree might be followed by a good split!

Better idea:

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

How to prune the tree?

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

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

Consider a sequence of trees indexed by a non negative tuning parameter α.

For each value of α, $\exists$ a corresponding subtree $T \subset T_0$ s.t.

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

($|T|$: # terminal nodes of the tree.)

R_m = m^{th} terminal node region

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

α control trade off between subtree's complexity & fit to training data → when $\alpha = 0$, $T = T_0$; $\alpha \uparrow$ ⇒ price to pay for having many terminal nodes ⇒ smaller tree

Select α via CV, then use full data set & chosen α to get subtree.

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 minimum # of observations.

(2) Apply cost complexity pruning to large tree 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) & (2) on all but k^{th} fold.

(b) evaluate MSPE on data in k^{th} fold as a function of α

Average results for each value of α and pick α to minimize CV error.

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

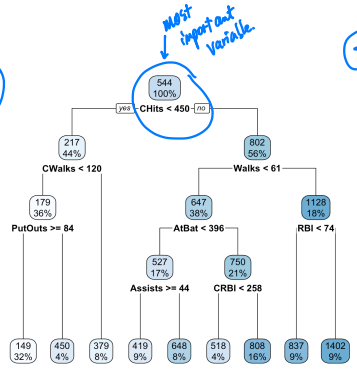
Example: Fit regression tree to Hitters using 9 features ~ 50% test, 50% split.

(1) is the large tree

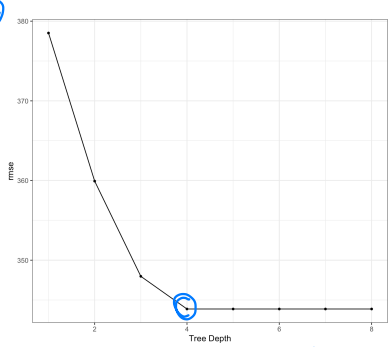
(2) CV error to estimate test MSE as a function of α.

(3) ~~subtree selected~~ smaller subtree.

(1)

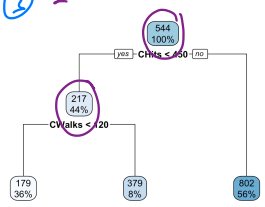


(2)



size (depth) is a function of α

(3) 2 node subtree.



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 the regression, the predicted response for an observation is given by the mean response of the training observations that belong to some 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.

mode We are 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 ✓ 100% class 1 vs. 55% class 1, 45% class 2

both 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 a 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 obs. in m^{th} region from k^{th} class.

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

preferred measures:

① Gini index $G = \sum_{k=1}^{K} \hat{p}_{mk}(1-\hat{p}_{mk})$ measure of total variance across all K classes.
↳ takes small values if all $\hat{p}_{mk}$'s are close to 0 or 1 ⇒ measure of node purity. ↓G ⇒ nodes will contain primarily 1 class

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

more sensitive to node purity than classification error rate.

Gini and entropy are actually quite similar numerically.

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

Note: neither Gini or entropy work well w/ unbalanced class data.
There are other options to split on.

Any of 3 methods can be used for pruning.

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

3 Trees vs. Linear Models

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

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

regression tree: $f(X) = \sum_{m=1}^{M} c_m \mathbb{I}(X \in R_m)$, where $R_1, \ldots, R_M$ partitions the feature space.

Which method is better? It depends on the problem.

- If the relationship between features and response is approximately linear, then a linear model will outperform a regression tree.
- If highly non-linear and complex relationship then trees may be better.

Also trees maybe prefered because of interpretation or 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 need to create dummy variables.

Disadvantages

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

  ↓

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

4 Bagging

Decision trees suffer from *high variance*.

i.e. If split data in half randomly, fit a decision tree to both halves, results could be quite different vs. low variance will yield similar results if applied to distinct data sets (from same population).

↳ linear regression is low variance if $n \gg p$.

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

Recall: For a given set of n independent observations $Z_1, \dots, Z_n$ each w/ variance σ^2

$$\mathrm{Var}(\bar{Z}) = \mathrm{Var}\left(\frac{1}{n}\sum_{i=1}^{n} Z_i\right) = \frac{1}{n^2}\mathrm{Var}\left(\sum_{i=1}^{n} Z_i\right) \overset{\text{indep}}{=} \frac{1}{n^2}\sum_{i=1}^{n}\mathrm{Var}\,Z_i$$
$$= \frac{1}{n^2}\sum_{i=1}^{n}\sigma^2 = \frac{\sigma^2}{n}$$

i.e. averaging a set of indep. observations reduces variance.

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.

i.e. take B training sets,
calculate $\hat{f}^1(x), \hat{f}^2(x), \dots, \hat{f}^B(x)$
obtain low variance statistical learning model

$$\hat{f}_{\text{AVG}}(x) = \frac{1}{B}\sum_{b=1}^{B}\hat{f}^b(x).$$

Of course, this is not practical because we generally do not have access to multiple training sets. Collecting training data can be expensive!

Instead we will take repeated samples (w/ replacement) from the training data set.
(These are called "bootstrapped" training data sets b/c we are bootstrapping samples from population using only one training dataset, i.e. "pulling ourselves up by our bootstraps").
↳ assumes empirical dsn in sample is similar to population dsn, i.e. we have a representative sample

Then we could train our method on b^{th} bootstrapped training data set to get $\hat{f}^{*b}(x)$ and average

$$\hat{f}_{\text{bag}}(x) = \frac{1}{B}\sum_{b=1}^{B}\hat{f}^{*b}(x).$$

↑
called "bootstrap aggregation" = "bagging"

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

To apply bagging to regression trees, ① construct B regression trees using B bootstrapped data sets
② average resulting predictions.

These trees are grown deep and not pruned.

⇒ each tree has low bias + high variance.

averaging trees reduces variance by combing hundreds or thousands of trees!
↳ won't lead to overfitting, but can be slow.

How can bagging be extended to a classification problem? (averaging no longer an option)

For a given test observation, record the class predicted by each of the trees and take a majority vote: overall prediction is the class that occurs most often.

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.

Key: trees are repeatedly fit to bootstrapped subsets of observations.
⇒ on average each tree uses $\approx 2/3$ ← has to do w/ prob of being selected in the bootstrap. of the data to fit the tree.

i.e. $\approx \frac{1}{3}$ of observations are NOT used to fit the tree (out-of-bag OOB observations)

idea: We could predict the response for the i^{th} observation using all the trees in which that observation was OOB.
This will lead to $\approx B/3$ predictions for i^{th} observation.
Then average (or majority vote) of those predictions to get a single OOB prediction for i^{th} observation.

We can then get one OOB prediction for each training observation to get OOB MSE (or OOB classification error) which is an estimate of test error!

This is valid because we only ever use predictions from trees that didn't use those data points in fitting!

4.2 Interpretation

Bagging typically results in improved accuracy in predictions over a single tree.
But it can be difficult to interpret the resulting model!

↳ one of the biggest advantages of decision trees !!

↳ no longer possible to represent the procedure using a single tree!

⇒ no longer clear which variables are the most important to predict the response

bagging improves prediction at the expense of interpretability.

What can we do?

We can obtain an overall summary of the importance of each predictor using RSS (or Gini index)

- record total amount RSS (or Gini) is decreased due to splits over a given predictor averaged over B trees.

= a large value indicates an important predictor.

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.

But when building trees, a random sample of m predictors is chosen as split candidates from the full set of predictors.

↳ the split is allowed only to use one of those predictors

↳ fresh sample of predictors taken at each split.

↳ typically $m \approx \sqrt{p}$

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. Why?

Suppose there is one strong predictor in the data set and a number of moderately strong predictors.

In the collection of trees, most or all will use the strong predictor in the top split.

⇒ all of the bagged trees will look quite similar.

⇒ predictions will be highly correlated.

And averaging (bagging) highly correlated values does not lead to much variance reduction!

$$\text{Var}\left(\frac{X_1+X_2}{2}\right) = \frac{1}{4}\left[\text{Var}X_1 + \text{Var}X_2 + 2\text{cov}(X_1, X_2)\right]$$

Random forest overcomes this by forcing each split to consider a subset of predictors.

⇒ on average $\frac{(p-m)}{p}$ of the splits won't even consider the strong predictor ⇒ one of the other predictors will have a chance!

The main difference between bagging and random forests is the choice of predictor subset size m. If $m = p$ ⇒ random forest = bagging.

Using small m will typically help when we have a lot of correlated predictors.

As with bagging, we will not have overfitting w/ large B

And we can examine variable importance in the same way.

6 Boosting

Very popular
see AdaBoost, XGBoost

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

Again, boosting is a general idea and we could boost other statistical learning model, we will talk about trees

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

Boosting grows the trees sequentially using information from previously grown trees.

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

Regression:

idea – boosting approach learns slowly to avoid overfitting.

> Given the current model, we fit a decision tree to residuals from the model and add decision tree to the fitted function to update.

> each tree is very small (just a few terminal nodes) => slowly improve $\hat{f}$ in areas where it does not perform well.

Algorithm

① Set $\hat{f}(x) = 0$ and $r_i = y_i$ $\forall i$ in training set.

② $b = 1, \ldots, B$ repeat

(a) Fit a tree $\hat{f}^b$ w/ d splits ($d+1$ terminal nodes) to training data $(\underline{x}, r)$

(b) Update $\hat{f}$ by adding a shrunken version of the new tree

$$\hat{f}(x) = \hat{f}(x) + \lambda \hat{f}^b(x).$$

↖ helps not to learn too fast (avoid overfitting).

(c) Update the residuals

$$r_i = r_i - \lambda \hat{f}^b(x_i).$$

③ Output the boosted model

$$\hat{f}(x) = \sum_{b=1}^{B} \lambda \hat{f}^b(x).$$

Boosting classification trees similar in idea, but more complex in details.

Boosting has three tuning parameters:

1. B - the # of trees
   Unlike bagging and RF, boosting can overfit with large B.
   Use CV to select B.

2. λ - shrinkage parameter (small positive #).
   this controls the rate at which boosting learns.
   Typical values: $\lambda = 0.01$ or $\lambda = 0.001$
   Very small λ can require large B to achieve good performance
   depends on problem.

   use CV to select (λ, B).

3. d - # splits in each tree
   Controls the complexity of the whole model
   Often $d=1$ works well ("stumps").
   ↳ the boosted ensemble is additive. => "ada boost"

   Generally, d is the interaction depth and controls the interaction order of the boosted model
   since d splits => at most d variables.

One of the coolest things about boosting is that not only does it work well, but it fits nicely into a statistical framework called "Decision Theory", meaning we have some guarantees on its behavior!