Ensembles of Trees

Rob McCulloch and Rodney Sparapani

1. Trees
2. Regression Trees
3. Classification Trees
4. Trees: A Summary
5. Fitting Trees: the Bias Variance Trade Off Again
6. Bagging and Random Forests
7. Boosting Trees

1. Trees

Tree based methods are a major player in statistics/machine-learning.

Good:

- flexible fitters, capture non-linearity and interactions.
  without having to choose a set of transformations !!!!
- do not have to think about scale of x variables.
- handles categorical and numeric y and x very nicely.
- fast.
- interpretable (when small).

Bad:

Not the best in out-of-sample predictive performance
(*but not bad!!*).

But,

If we **bag** or **boost** trees, we can get the best off-the-shelf prediction available.

Bagging and Boosting are *ensemble methods* that combine the fit from many (hundreds, thousands) of tree models to get an overall predictor.

> *".. it is rather amazing that an ensemble of trees leads to the state of the art in black-box predictors !*

Bradley Efron and Trevor Hastie, Computer Age Statistical Inference, chapter 17, 2016.

2. Regression Trees

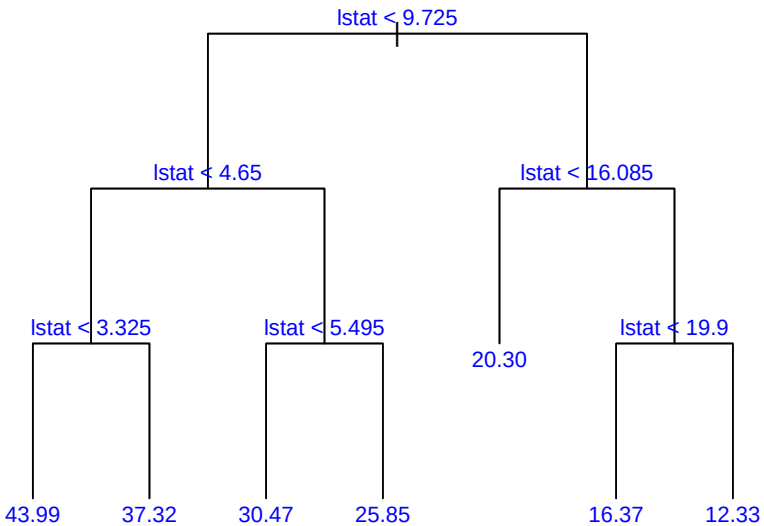
Let's look at a simple 1-dimensional example so that we can see what is going on.

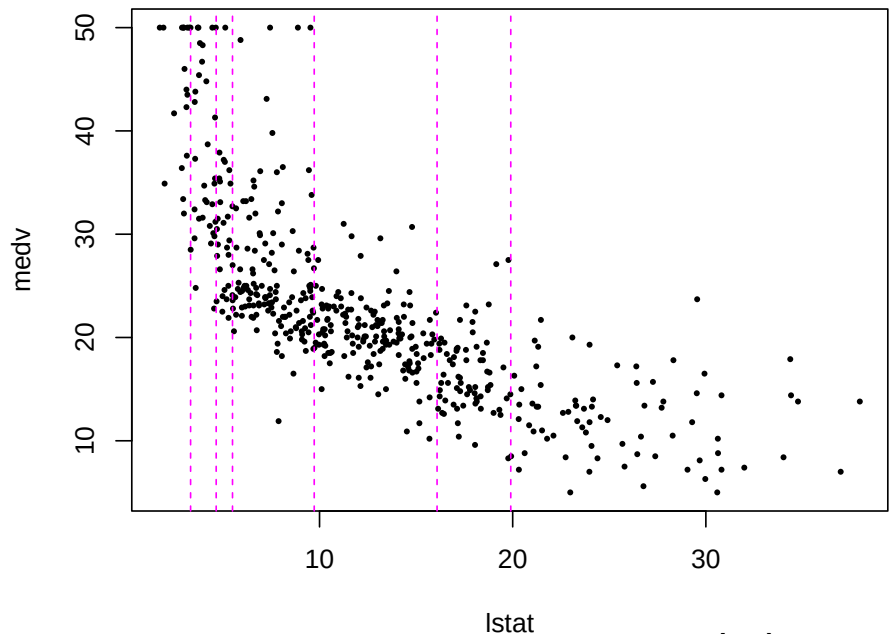
We'll use the Boston housing data and relate x=lstat to y=medval.

At left is the *tree* fit to the data.

At each *interior node* there is a decision rule of the form $\{x < c\}$. If $x < c$ you go left, otherwise you go right.

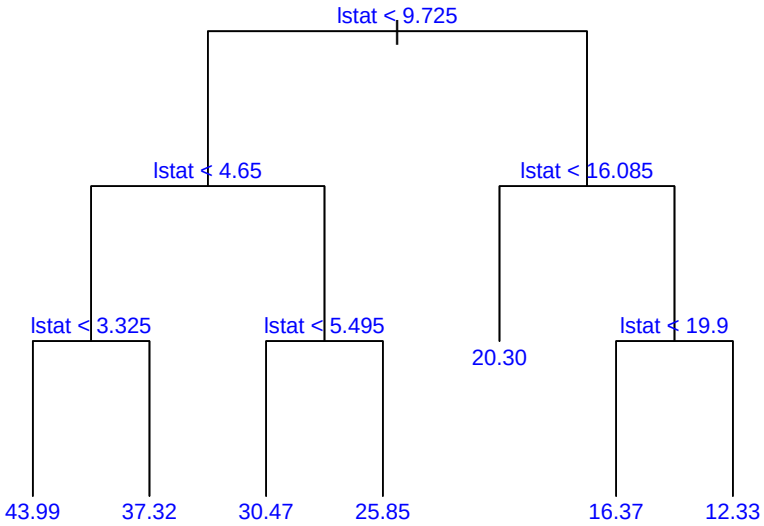
Each observation is sent down the tree until it hits a bottom node or *leaf* of the tree.

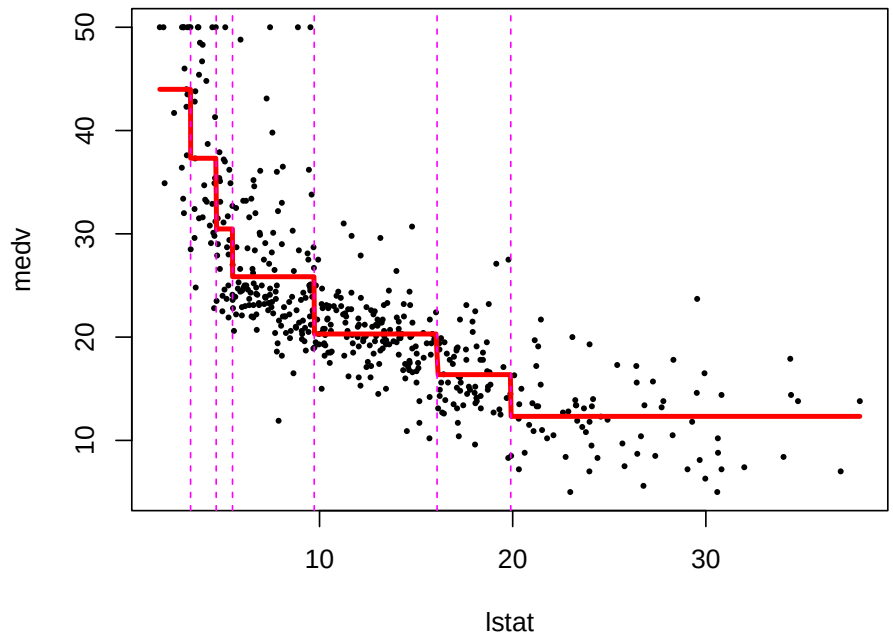




The set of bottom nodes gives us a partition of the predictor (x) space into disjoint regions. At right, the vertical lines display the partition. With just one x, this is just a set of intervals.

Within each region (interval) we compute the average of the y values for the subset of training data in the region. This gives us the step function which is our $\hat{f}$. The $\bar{y}$ values are also printed at the bottom nodes (left plot).





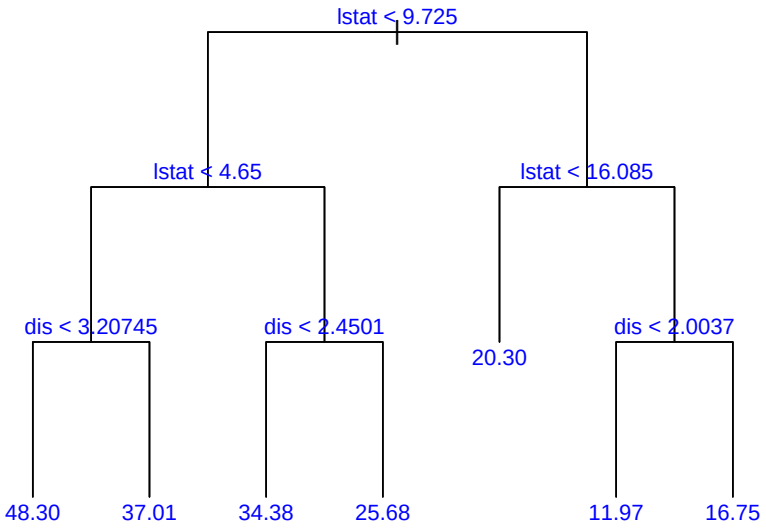
To predict, we just use our step function estimate of $f(x)$.

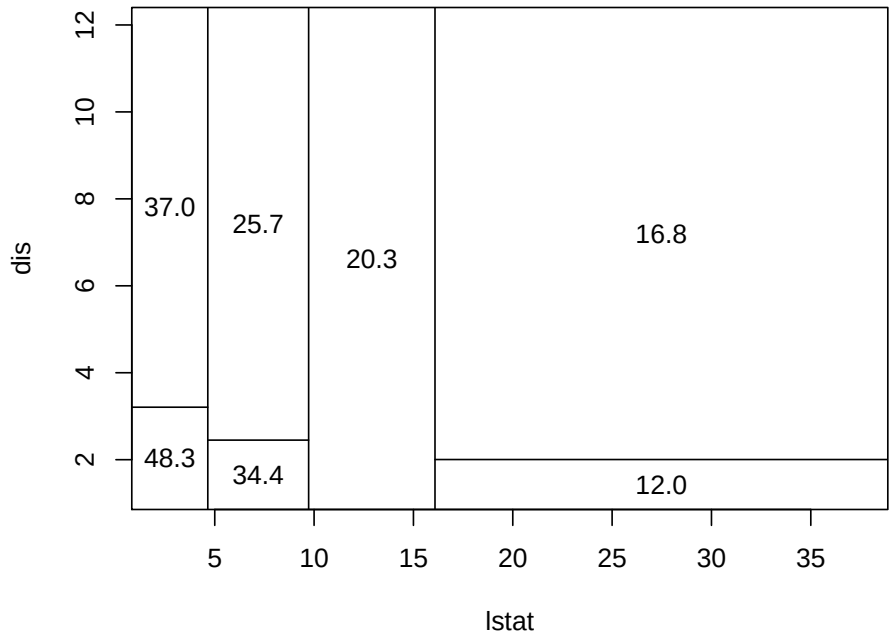
Equivalently, we drop x down the tree until it lands in a leaf and then predict the average of the y values for the training observations in the same leaf.

A Tree with Two Explanatory Variables

Here is a tree with $x = (x_1, x_2) =$ (lstat,dis) and y=medv.

Now the decision rules can use either of the two x's.





At right is the *partition* of the x space corresponding to the set of bottom nodes (leaves).

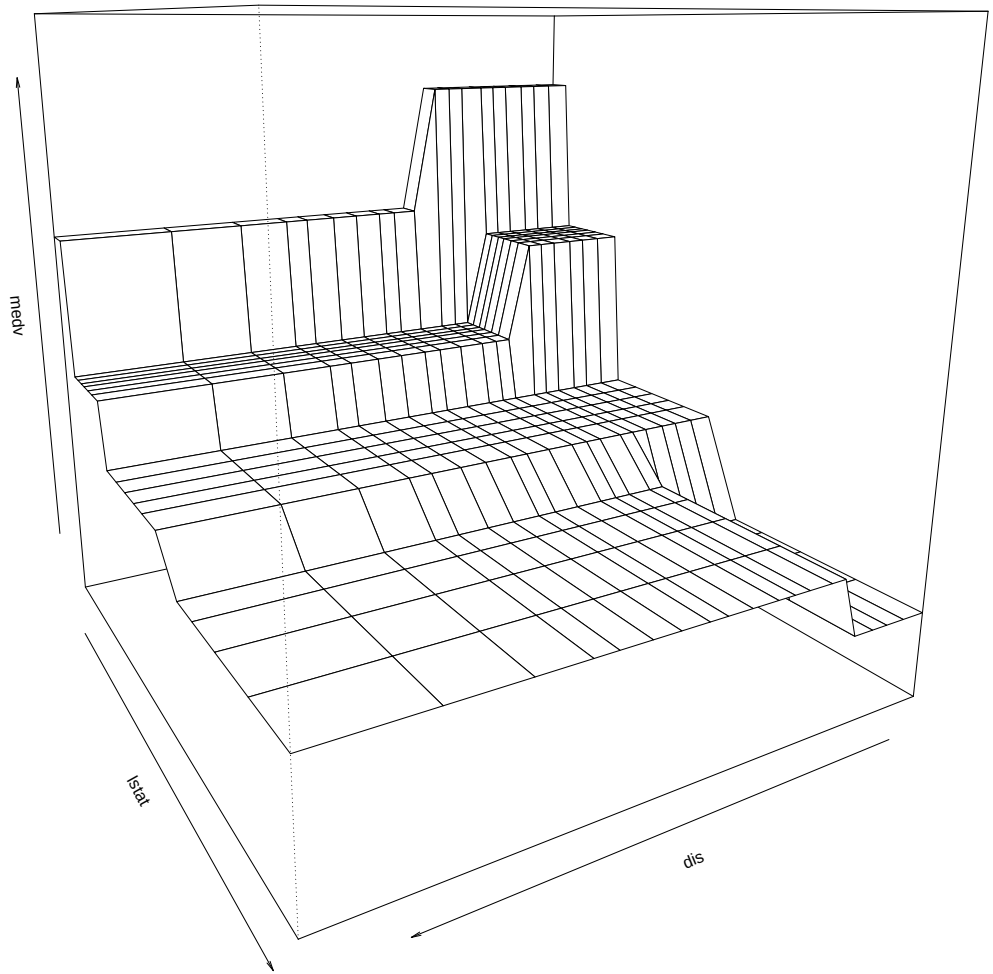
The average y for training observations assigned to a region is printed in each region and at the bottom nodes.

This is the regression function given by the tree.

It is a step function which can seem dumb, but it delivers non-linearity *and* interactions in a simple way and works with a lot of variables.

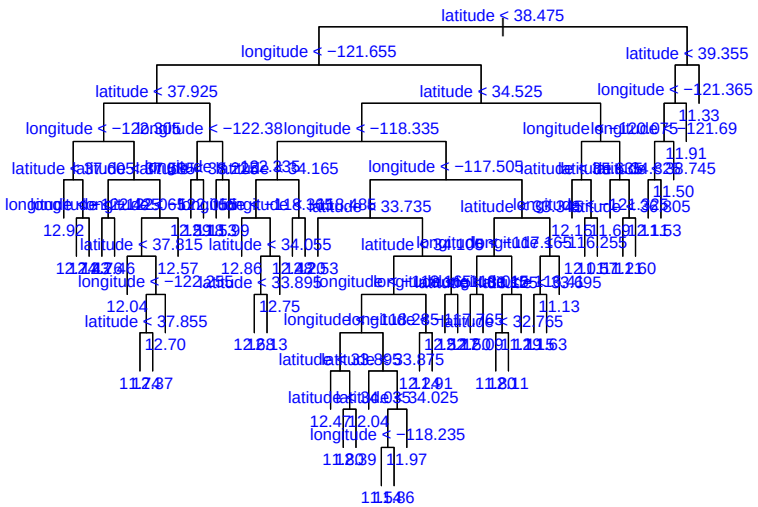
Notice the interaction.

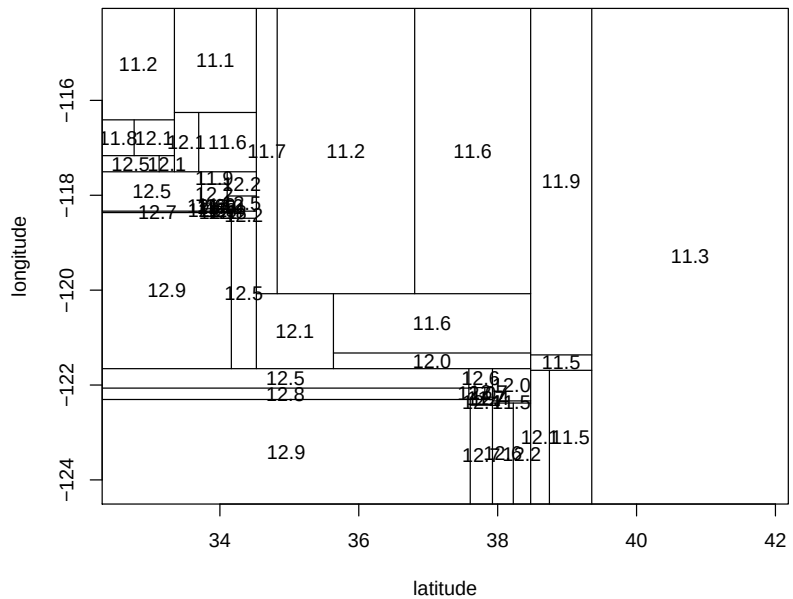
The effect of `dis` depends on `lstat`!!



The California Housing Data

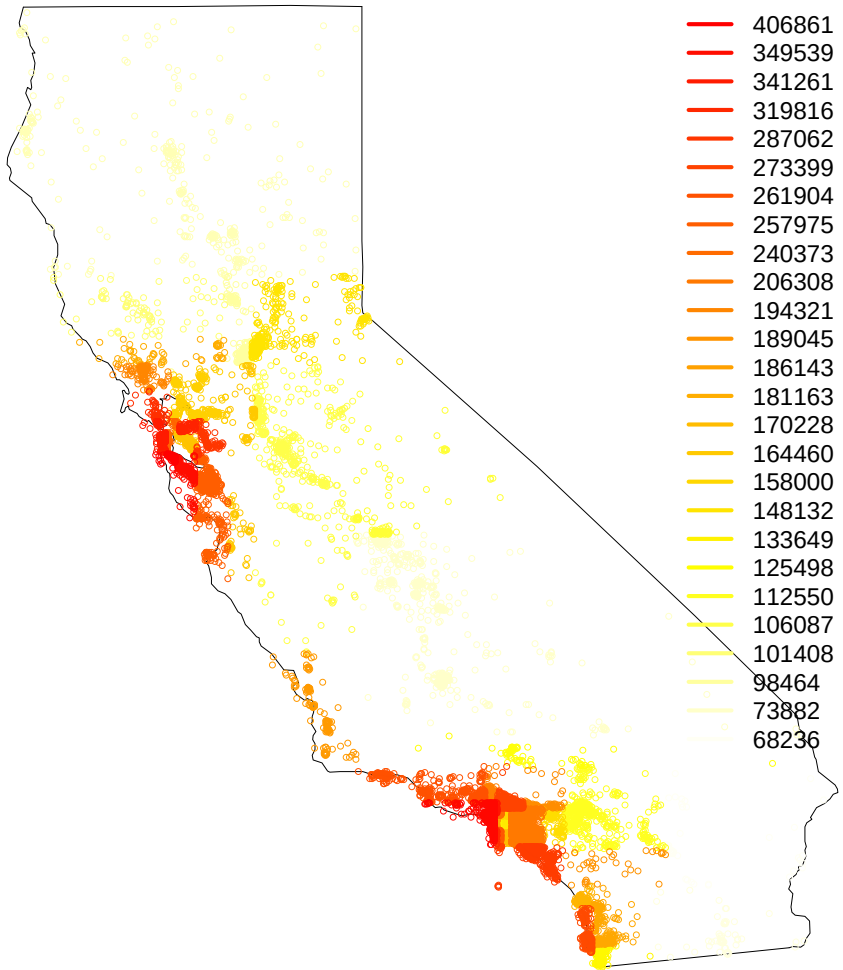
Here is a tree with 50 bottom nodes fit to the California Housing data using only `longitude` and `latitude`.





Don't extrapolate into the ocean!

Here is a view of the fit using the map of the state.
(units are dollars, the logMedVal was exponentiated for the labels).



3. Classification Trees

Let's do a tree for a classification problem.

We'll use the hockey penalty data.

The response is 1 if the current penalty is *not* on the same team as the previous penalty and 0 otherwise.

x is a bunch of stuff about the game situation
(the score ...).

The x values refer to the team that had the previous penalty.
For example, `goaldiff`$=1$ means the team that had the previous penalty is ahead by one goal.

Our response is binary and some of our predictors are categorical as well.

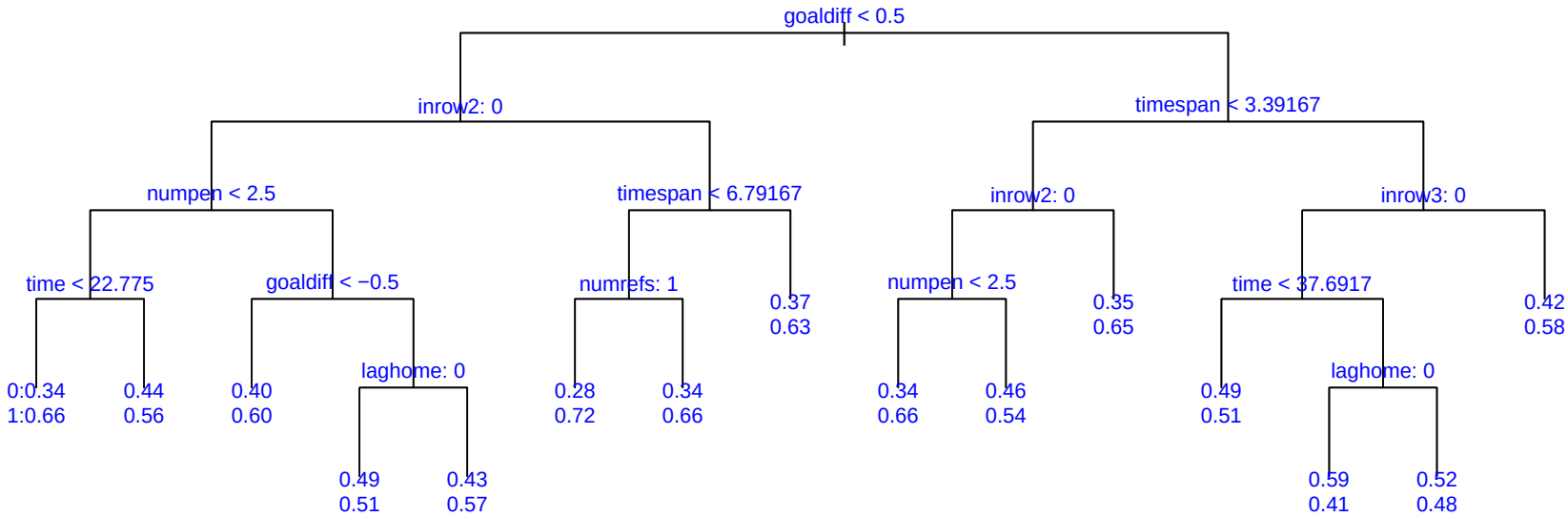
Table 5: Variable Descriptions

Variable	Description	Mean	Min	Max
Dependent variable				
`revcall`	1 if current penalty and last penalty are on different teams	0.589	0	1
Indicator-Variable Covariates				
`ppgoal`	1 if last penalty resulted in a power-play goal	0.157	0	1
`home`	1 if last penalty was called on the home team	0.483	0	1
`inrow2`	1 if last two penalties called on the same team	0.354	0	1
`inrow3`	1 if last three penalties called on the same team	0.107	0	1
`inrow4`	1 if last four penalties called on the same team	0.027	0	1
`tworef`	1 if game is officiated by two referees	0.414	0	1
Categorical-variable covariate				
`season`	Season that game is played (e.g., 1995 for 95-6 season)		1995	2001
Other covariates				
`timeingame`	Time in the game (in minutes)	31.44	0.43	59.98
`dayofseason`	Number of days since season began	95.95	1	201
`numpen`	Number of penalties called so far (in the game)	5.76	2	21
`timebetpens`	Time (in minutes) since the last penalty call	5.96	0.02	55.13
`goaldiff`	Goals for last penalized team minus goals for opponent	-0.02	-10	10
`gf1`	Goals/game scored by the last team penalized	2.78	1.84	4.40
`ga1`	Goals/game allowed by the last team penalized	2.75	1.98	4.44
`pf1`	Penalties/game committed by the last team penalized	6.01	4.11	8.37
`pa1`	Penalties/game by opponents of the last team penalized	5.97	4.33	8.25
`gf2`	Goals/game scored by other team (not just penalized)	2.78	1.84	4.40
`ga2`	Goals/game allowed by other team	2.78	1.98	4.44
`pf2`	Penalties/game committed by other team	5.96	4.11	8.37
`pa2`	Penalties/game by opponents of other team	5.98	4.33	8.25

$n \approx 60,000.$

Here is the tree.

`goaldiff` $< .5$ means the last penalized team is not winning.
Do you want to give them a another penalty ???



- Each bottom node gives the fraction of training data in the two outcome categories. Think of it as $\hat{p}$ for the kind of x associated with that bottom node.

- The form of the decision rule can't be $x < c$ for categorical variables.
  We pick a subset of the levels to go left. `inrow2:0` means all the observations with `inrow2` in the category labeled 0 go left.

There is a lot of fit!!!

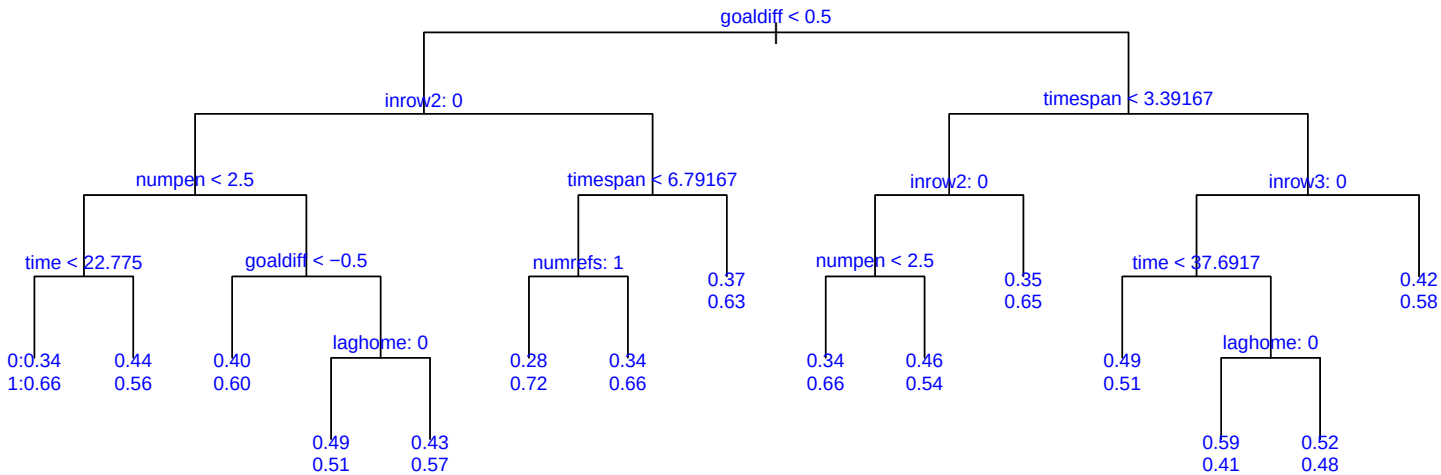
Suppose "you" got the last penalty.

if:

- if you are not winning
- you had the last two penalties
- it has not been long since the last call
- and there is only 1 referee

then:

there is a 72% chance the next call will be on the other team.



Whilst there is another game situation where the chance the next call is on the other team is only 41%.

4. Trees: A Summary

Trees:

- Trees use recursive binary splits to partition the predictor space.
- Each binary split consists of a decision rule which sends x left or right.
- For numeric x_i, the decision rule is of the form if $x_i < c$.
- For categorical x_i, the rule lists the set of categories sent left.
- The set of bottom nodes (or leaves) give a partition of the x space.
- To predict, we drop an out-of-sample x down the tree until it lands in a bottom node.
- For numeric y, we predict the average y value for the training data that ended up in the bottom node.
- For categorical y we use the category proportions for the training data that ended up in the bottom node.

Good:

- Handles categorical/numeric x and y nicely.
- Don't have to think about the scale of x's !!!
- Computationally fast ("scales").
- Small trees are interpretable.
- Variable selection.

Bad:

- Step function is crude, does not give the best predictive performance.
- Hard to assess uncertainly.
- Big trees are not interpretable.

5. Tree Models and the Bias Variance Trade Off

How do we fit trees to data??

The key idea is that a complex tree is simply a big tree.

We usually measure the complexity of the tree by the number of bottom nodes.

To fit a tree, we choose a tree to minimize (on the training data):

$$C(T, y) = L(T, y) + \alpha |T|$$

where,

- $L(T, y)$ is our loss in fitting data y with tree T.
  We want good fit on the training data $\Rightarrow$ want L small.
- $|T|$ is the number of bottom nodes in tree T.
  But, we don't want a complex model that fits *too well*
  $\Rightarrow$ we want $|T|$ small.

For numeric y our loss is usually sum of squared errors, for categorical y we can use the deviance or some other measure of classification fit.

$$C(T, y) = L(T, y) + \alpha |T|$$

α big:

The *penalty* for having a big tree is large.
When we do our minimization, we will get a smaller tree with a bigger L on the training data.

α small:

We do not mind having a big tree.
We will get a smaller L (better fit) on the training data.

α is analogous to k in k-NN !!!!!
α is analogous to λ in the lasso !!!!!

α is called the *complexity-cost penalty parameter*.

How do we do the minimization ??!!

Now we have a problem.

While trees are simple in some sense, once we view them as variables in an optimization they are large and complex.

A key to tree modeling is the success of the following heuristic algorithm for fitting trees to training data.

(I. Grow Big)

Use a greedy, recursive forward search to build a big tree.

(i)

Start with the tree that is a single node.

(ii)

At each bottom node, search over all possible decision rules to find the one that gives the biggest decrease in L (increase in fit).

(iii)

Grow a big tree, stopping (for example) when each bottom node has 5 observations in it.

(II. Prune Back)

(i)

Recursively, prune back the big tree from step (I).

(ii)

Give a current pruned tree, examine every pair of bottom nodes (having the same parent node) and consider eliminating the pair.

Prune the pair the gives the biggest decrease in our criterion C.

This is give us a sequence of subtrees of our initial big tree.

(iii)

For a given α, choose the subtree of the big tree that has the smallest C.

So,

Give training data and α we get a tree.

How do we choose α ??

As usual, we can leave out a validation data set and choose the α the performs best on the validation data, or use k-fold cross validation.

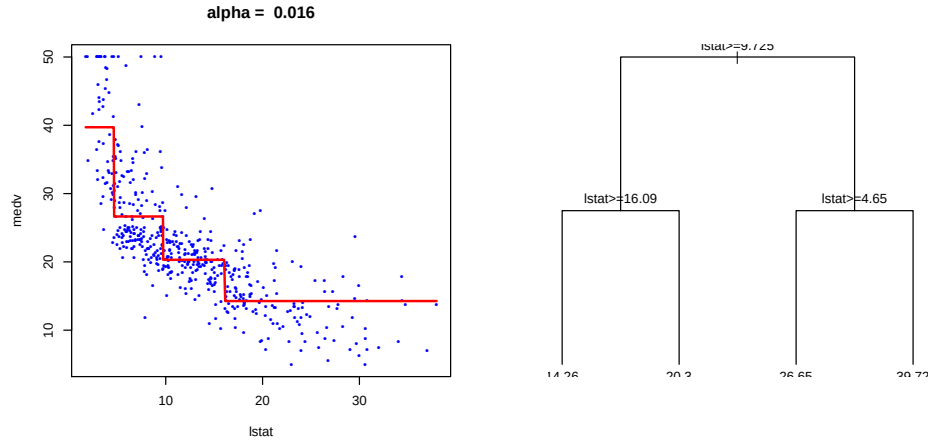
Boston Data, lstat and medv:

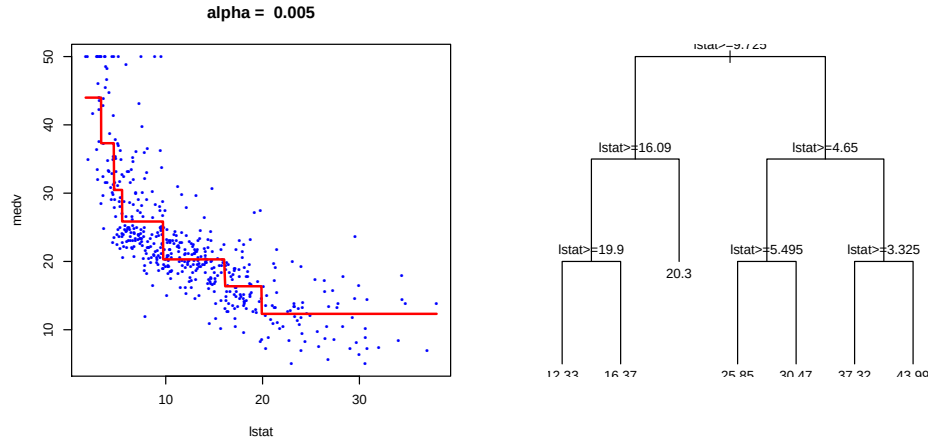
At right are three different tree fits we get from three different α values (using all the data).

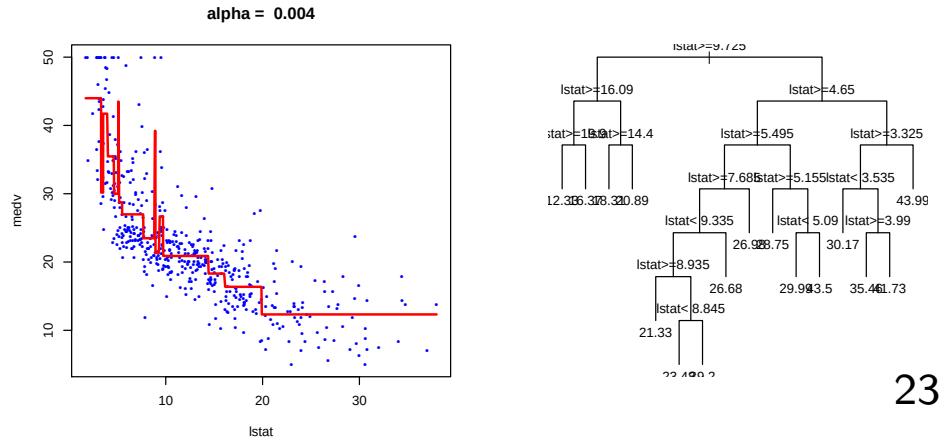
The smaller α is, the lower the penalty for complexity is, the bigger tree you get.

The top tree is a sub-tree of the middle tree, and the middle tree is a sub-tree of the bottom tree.

The middle α is the one suggested by CV.



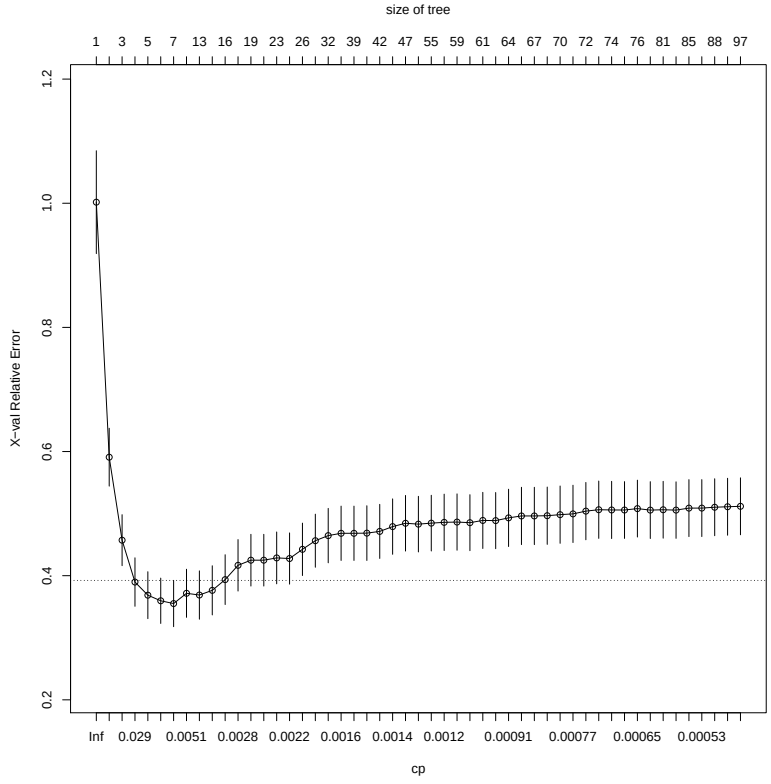




This is the CV plot giving by the R package rpart for y=medv x=lstat.

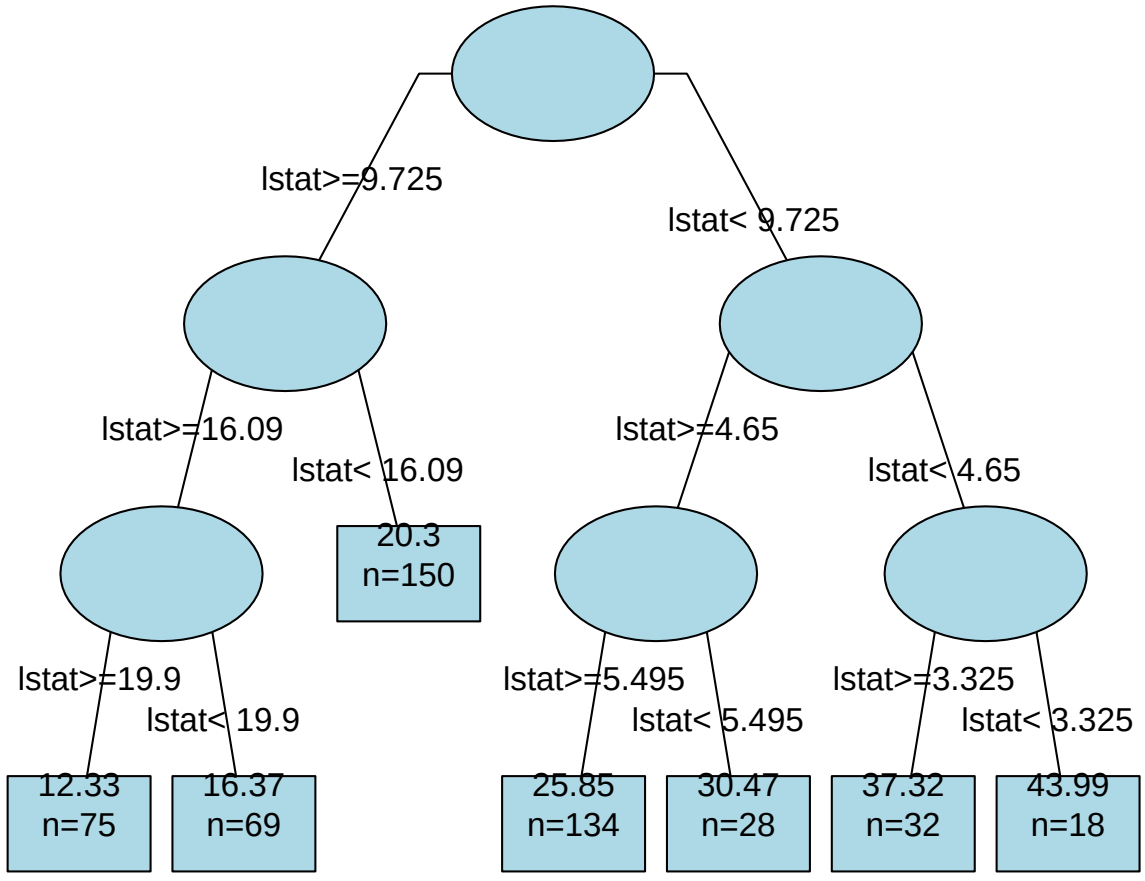
Tree sizes at top of plot, and (a transformation of) α (the "cost-complexity" parameter) on the bottom.

The error is relative to the error obtained with a single node (fit is $y = \bar{y}$, $\alpha = \infty$).



Caution: the bottom axis is a transformation of α.

Here is the best CV tree as plotted by rpart.



6. Bagging and Random Forests

A key idea in modern statistics is the *bootstrap:*

Treat the sample as if it were the population and then take iid draws.

That is, you sample *with replacement* so that you can get the same original sample value more than once in a *bootstrap sample*.

We can use the bootstrap to make trees *much* better predictors !!!!

To **B***ootstrap* **Ag***gregate (Bag) we:*

- Take B bootstrap samples from the training data, each of the same size as the training data.
- Fit a *large* tree to each bootstrap sample (we know how to do this fast!). This will give us B trees.
- Combine the results from each of the B trees to get an overall prediction.

What is a bootstrap a sample?

```
> set.seed(34) # Auston Matthews
>
> n = 20
> x = rnorm(n)
> print(x)
 [1] -0.138889971  1.199812897 -0.747722402 -0.575248177 -0.263581513
 [6] -0.455492149  0.670620044 -0.849014621  1.066804504 -0.007460534
[11] -0.402880091  0.719107939 -0.180058654  1.046190759  0.401254928
[16]  1.356390044  0.019226639 -0.469417841 -1.842661894 -0.279740938
>
> xs = sample(x,size=n,replace=TRUE)
> print(xs)
 [1]  1.1998129  1.1998129 -0.2797409  1.0461908 -0.7477224  1.0461908
 [7]  1.1998129 -0.2797409  1.0461908 -1.8426619  1.3563900  0.6706200
[13] -0.5752482 -0.4028801  0.6706200 -0.8490146  1.0668045 -0.4554921
[19] -0.4694178 -0.7477224
> print(length(unique(xs)))
[1] 13
```

We act like our sample x is the population, and then we can take as many bootstrap samples from x as we like by sampling with replacement.

For numeric y we can combine the results easily by making our overall prediction the average of the predictions from each of the B trees.

For categorical y, it is not quite so obvious how you want to combine the results from the different trees.
Often people let the trees vote: given x get a prediction from each tree and the category that gets the most votes (out of B ballots) is the prediction.

Alternatively, you could average the $\hat{p}$ from each tree. Most software seems to follow the vote plan.

Why on earth would this work??!!

Remember our basic intuition about *averaging*, for

$$y_i = \mu + \epsilon_i,$$

we think of μ as the signal and ϵ_i as the noise part of each observation.

When we average the y_i to get $\bar{y}$, the signal, μ, is in each draw, so it does not wash away, but the ϵ_i wash out.

For us, the *signal* is the part of y we can guess from knowing x!!

Bagging works the same way.

We randomize our data and then build a lot of big (and hence noisy!) trees.

The relationships which are real get captured in a lot of the trees and hence do not wash out when we average.

Stuff that happens "by chance" is idiosyncratic to one (or a few) trees and washes out in the average.

Brilliant. **Leo Brieman**.

Bagging and the Bias-Variance Tradeoff

A nice way to think about bagging is in terms of the *Bias-Variance tradeoff*.

A big tree is a complex model which gives us high variance and low bias.

What does low bias mean? We can find a good $\hat{f}$ on average, where *average* means average over data sets you might get from the population or process generating the data.

So, if we could average the results from many data sets, we could reduce the variance, and get the good average $\hat{f}$!!

But we only get one data set !!!!

We get many data sets by bootstrap sampling from our observations and then average the results !!!

Note:

You need B big enough to get the averaging to work, but it does not seem to hurt if you make B bigger than that.

The cost of having very large B is in computational time.

We can build trees fast, but if you start building thousands of really big trees on large data sets, it can end taking a while.

Random Forests:

Random Forests starts from Bagging and adds another kind of randomization.

Rather than searching over all the x_i in x when we do our greedy build of the big trees, we randomly sample a subset of m variables to search over each time we make a split.

This makes the big trees "move around more" so that we explore a rich set of trees, *but the important variables will still shine through!!*.

Have to choose:

- B: number of Bootstrap samples (hundreds, thousands).
- m: number of variables to sample.

  A common choice is $m = \sqrt{p}$,
  where p is the dimension of x.

Note:

Bagging is Random Forests with $m = p$.

Note:

There is no explicit regularization parameter as in the lasso and single tree prediction.

OOB Error Estimation:

OOB is "Out of Bag".

For a bootstrap sample, the observations chosen are "in the bag" and the rest are out.

There is a very nice way to estimate the out-of-sample error rate when bagging.

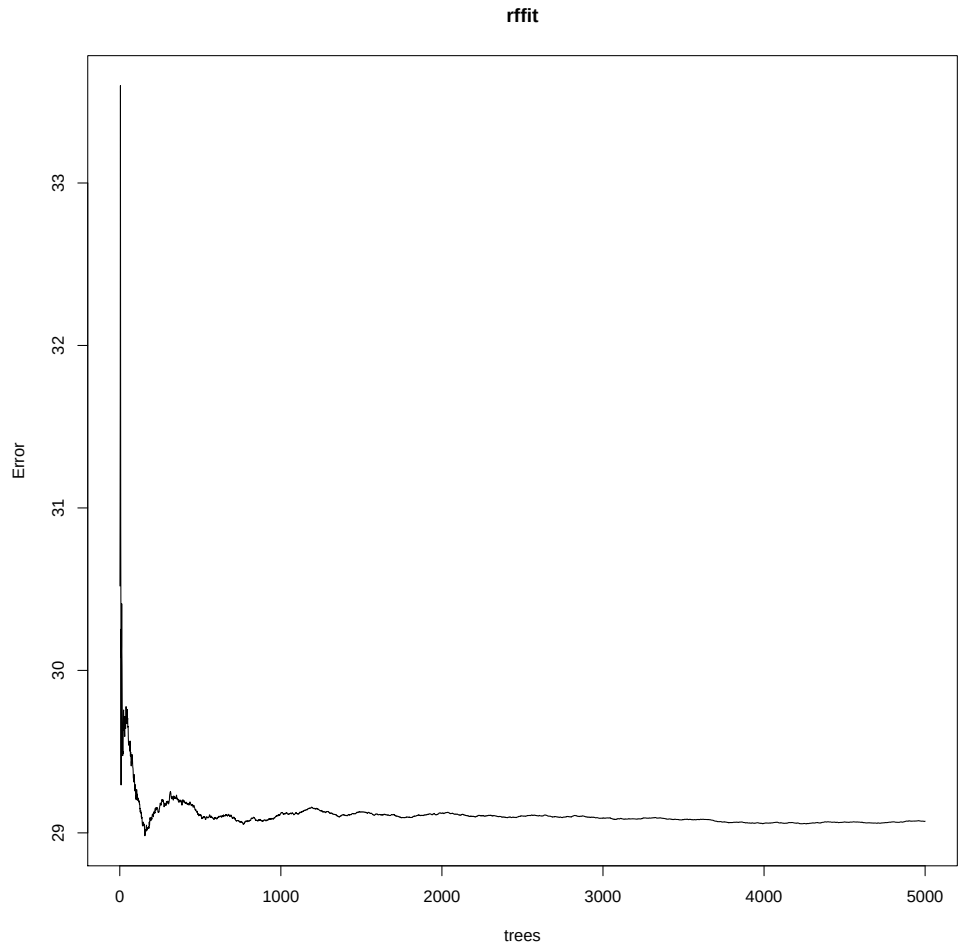
One can show that, on average, each bagged tree makes use of about 2/3 of the observations.

By carefully keeping track of which bagged trees use which observations you can get out-of-sample predictions.

Bagging for Boston: y=medv, x=lstat.

Here is the error estimation as a function of the number of trees based on OOB.

Suggests you just need a couple of hundred trees.

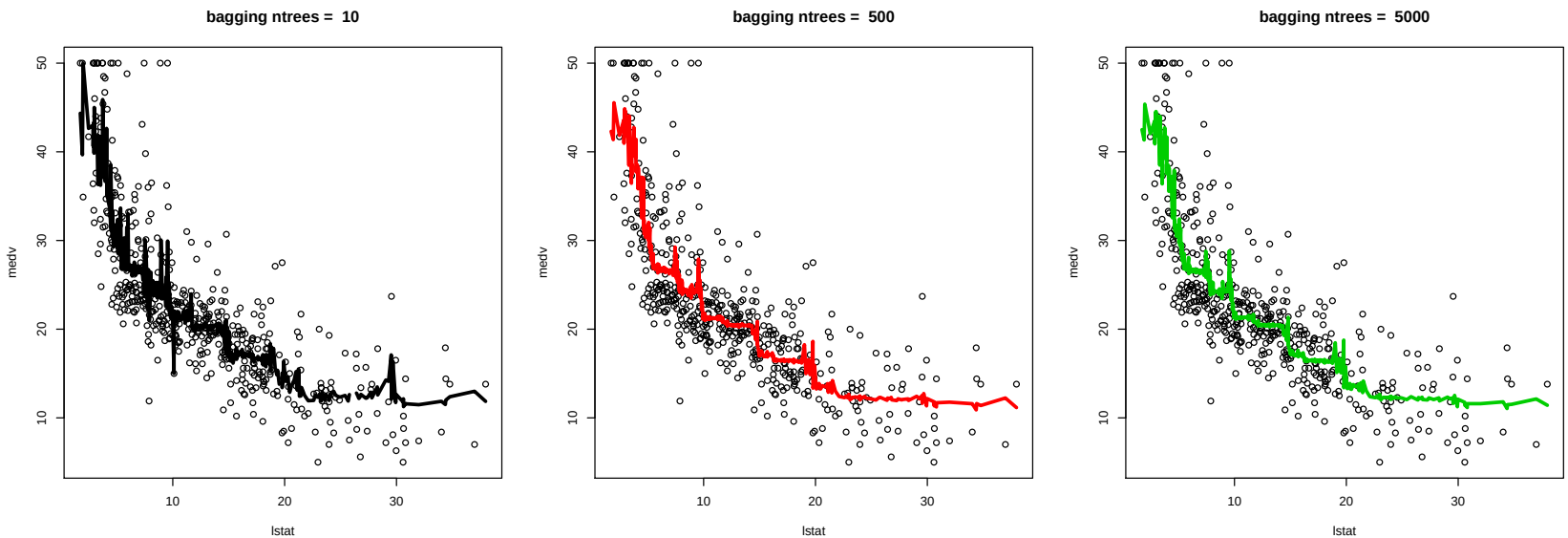


Bagging for Boston: y=medv, x=lstat.

With 10 trees our fit is too jumbly.

With 1,000 and 5,000 trees the fit is not bad and very similar.

Note that although our method is based on trees, we no longer have a simple step function!!



7. Boosting Trees

Like Random Forests, boosting is an *ensemble method* is that the overall fit it produced from many trees.

The idea however, is totally different!!

In Boosting we sequentially *add in* functions corresponding to *simple* trees to our overall fit, where each tree added in improves things "a bit".

This one is actually made clearer by the mathematical notation.

For Numeric y:

(i) Set $\hat{f}(x) = 0$. $r_i = y_i$ for all i in the training set.

(ii) for $b = 1, 2, \ldots B$, repeat:

- Fit a tree $\hat{f}^b$ with d splits ($d + 1$ terminal nodes) to the training data (X, r).
- Update $\hat{f}$ by adding in a shrunken version of the new tree: $\hat{f}(x) \leftarrow \hat{f}(x) + \lambda \hat{f}^b(x)$.
- Update the residuls: $r_i \leftarrow r_i - \lambda \hat{f}^b(x)$.

(iii) Output the boosted model:

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

Note:

λ is the "crushing" or "shrinkage" parameter.

It makes each new tree a *weak learner* in that is only does a little more fitting.

Have to choose:

- B, number of iterations (the number of trees in the sum) (hundreds, thousands).
- d, the size of each new tree.
- λ, the crush factor.

Note:

Boosting for categorical y works in an analogous manner but it is more messy how you define "the part left over", you can't just use residuals.

Also you can't just add up the fit.

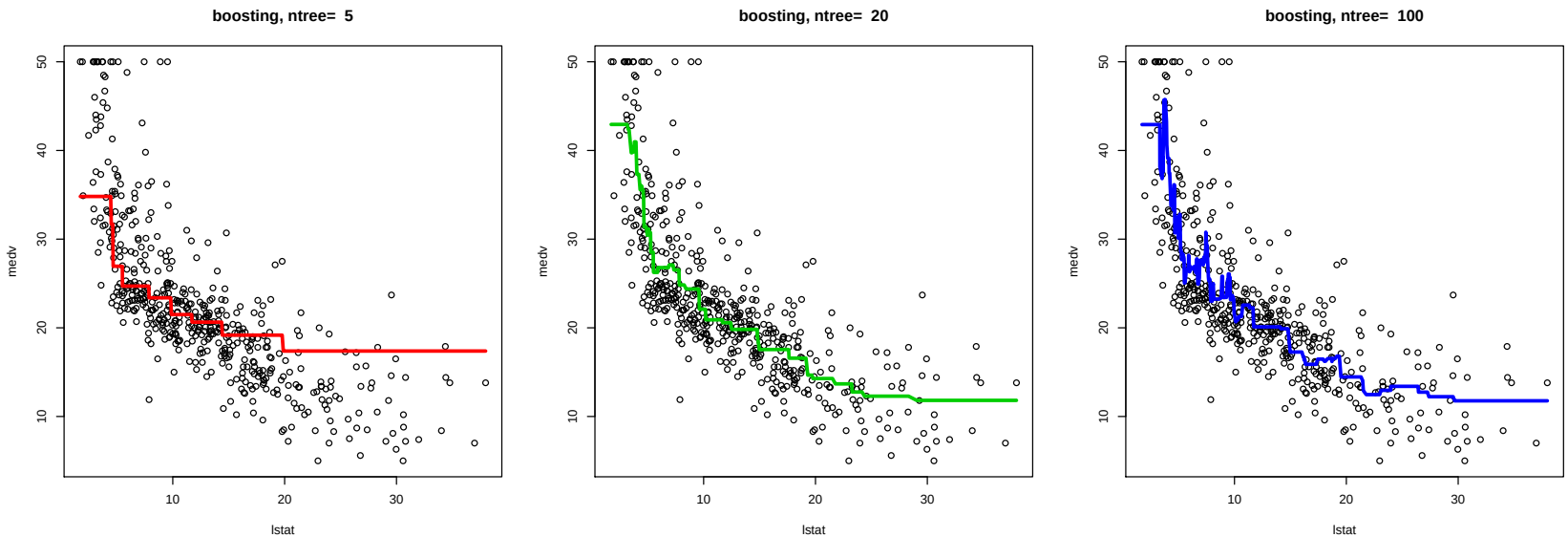
But, it is the same idea:

Sequentially add in small fits of fit ("weaker learners") focusing on the errors of the current fit (e.g observations where the residuals are large).

Efron and Hastie: ".. each tree is trying to amend errors made by the ensemble of previously grown trees."

Boosting for Boston: y=medv, x=lstat:

Here are some boosting fits where we vary the number of trees, but fix the depth at 2 (suitable with 1 x) and shrinkage = λ at .2.



Again, this ensemble method gets away from the crude step function given by a single tree.