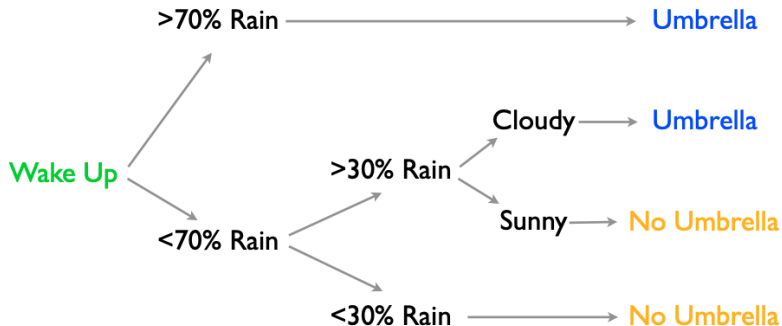


Tree models

James Scott

ECO 395M: Data Mining and Statistical Learning

Tree models



Trees involve simple mini-decisions that combine to make a choice or prediction.

Each decision is a *node*; the final choice or prediction is a *leaf node*.

Tree models

You can think of a tree as a form of regression model:

- ▶ inputs x : forecast, current conditions
- ▶ output y : need for an umbrella

Based on previous data, the the goal is to specify branches/choices that lead to good predictions in new scenarios.

In other words, you want to estimate a **Tree Model**. Instead of linear coefficients, we need to find 'decision nodes': binary splitting rules defined by the x features.

Tree models

Tree models come in two flavors.

- ▶ Classification tree: the leaf nodes are predicted class labels/probabilities for a categorical outcome. (**“The predicted probability of rain is 90%, so take an umbrella.”**)
- ▶ Regression tree: the leaf nodes specify $E(y \mid x)$ for a numerical outcome. (**“The predicted amount of rain is 2 cm, so take an umbrella.”**)

The basic idea is the same for both; just a few of the details change.

Tree models

The goal of tree modeling: specify the sequence of mini-decisions that get you to the leaves.

- ▶ How many?
- ▶ What decisions? (Which features, which values of the features?)
- ▶ What order?

The space of possible trees is *huge*.

We'll talk about fitting the tree later; for now, let's see some examples to build our intuition.

A classification tree

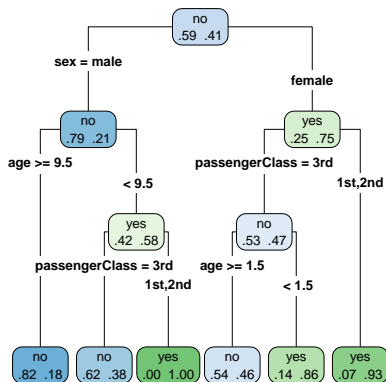
Classification trees are for categorical outcomes (with binary outcomes as a special case).

Let's see an example trained on data from the Titanic:

- ▶ x : a passenger's sex, age, class of travel
- ▶ y : whether the passenger survived the sinking of the ship

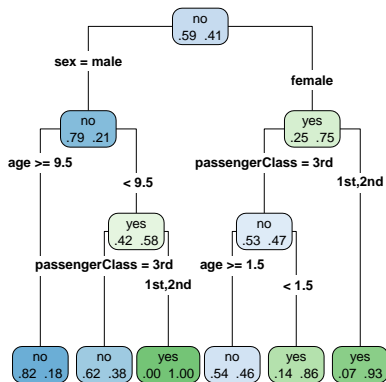
Goal: estimate $P(y \mid x)$

A classification tree



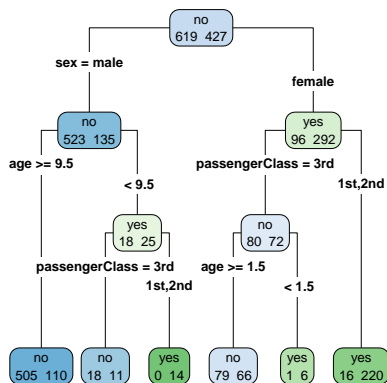
- ▶ Each split involves a yes/no question about a single variable.
- ▶ For numerical features (age), the yes/no question is whether x exceeds some threshold t .
- ▶ You might see/hear this called a “dendrogram,” which is just a fancy Latin word for “tree picture.”

A classification tree



- ▶ At each leaf node, we see fitted class probabilities. These come from the training data.
- ▶ To make a prediction, you “drop” your x down the tree, answering each yes/no question in turn.
- ▶ Notice the interactions! E.g. the questions we ask about age at later splits depend on which branch we’re on.

A classification tree



- ▶ It's easier to see where the fitted probabilities come from if we show the number of observations per class in each leaf node.
- ▶ Let's reason through two quick examples.
 - ▶ $x = \{\text{male, 3rd, 5 years}\}$
 - ▶ $x = \{\text{female, 1st, 25 years}\}$.

A regression tree

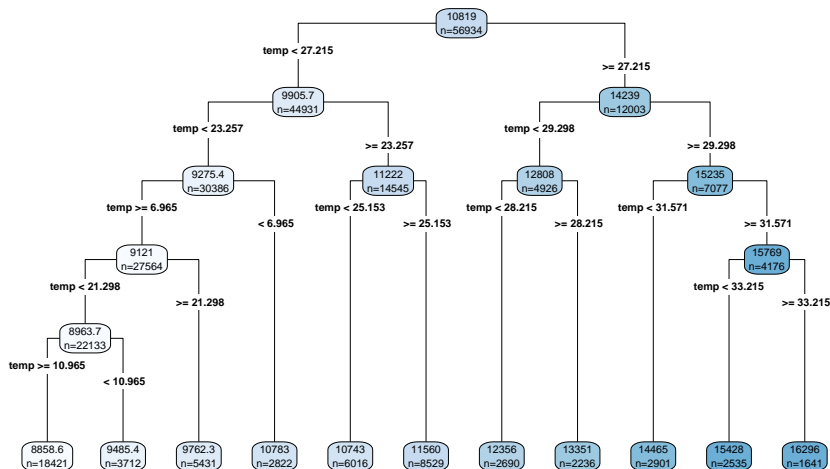
Regression trees are for numerical (as opposed to categorical) outcomes.

Let's see one on a familiar data set:

- ▶ y = peak power consumption in the ERCOT coast region
- ▶ x = temperature at Houston's Hobby airport in degrees C (so all splits are of the form $\text{temp} < t$ for some threshold t).

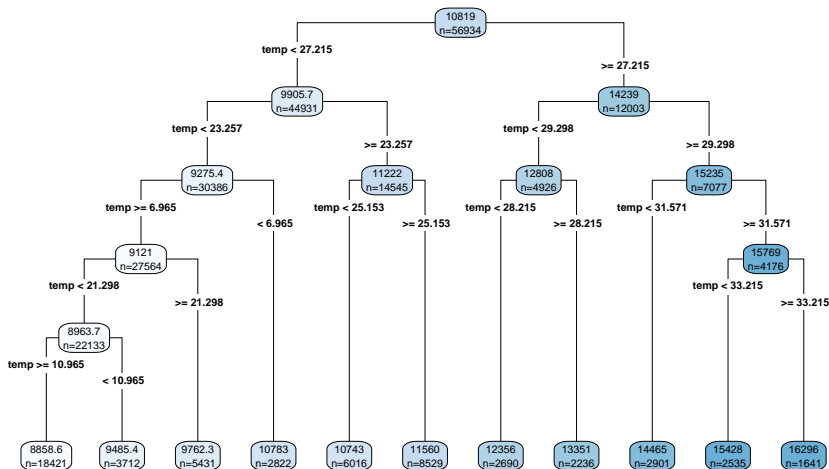
Goal: estimate $E(y \mid x)$

A regression tree



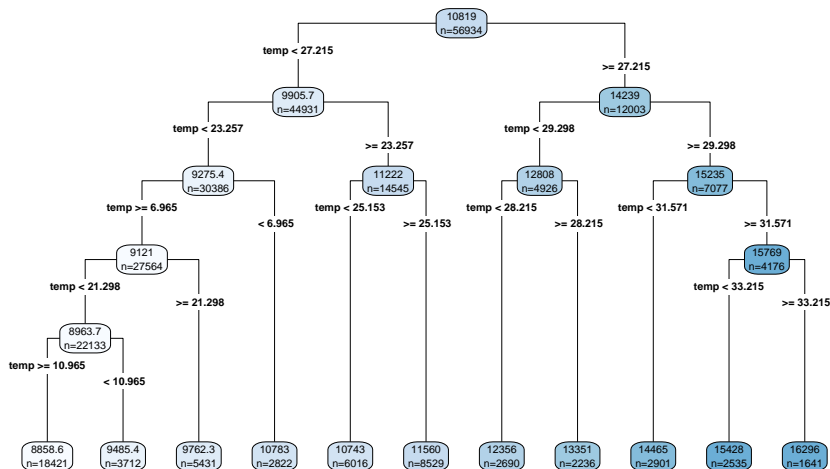
Now the leaf nodes show $E(y \mid x)$, estimated by the average response (y) for the x 's that “land” in that leaf.

A regression tree



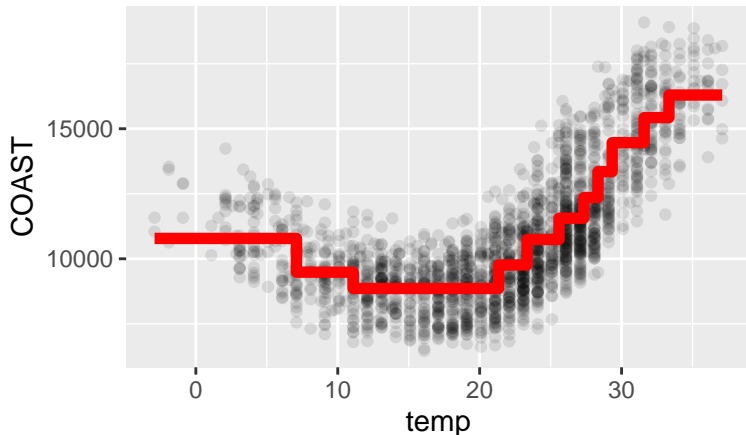
What does $f(x) = E(y | x)$ look like as a function of x ?

A regression tree



Hint: the tree partitions the x space into disjoint regions. Within each region, $E(y \mid x)$ is constant.

A regression tree



This is the fitted regression function $f(x) \approx E(y \mid x)$. It's a **step function** (always the case with tree models).

A regression tree (two x's)

What if we consider the same problem, but with an additional feature?

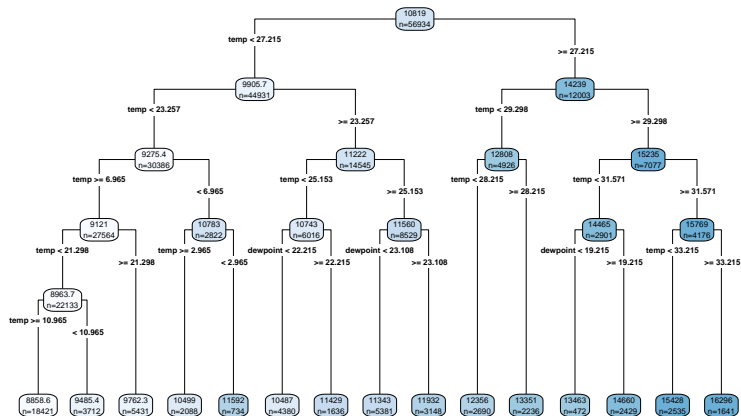
- ▶ x_1 = temperature at Houston's Hobby airport in degrees C
- ▶ x_2 = dewpoint at Houston's Hobby airport in degrees C
- ▶ y = peak power consumption in the ERCOT coast region

The linear model equivalent would look like

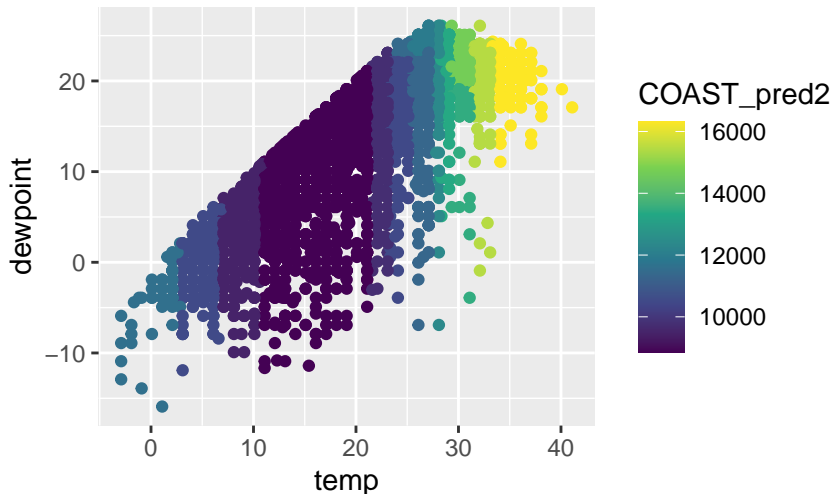
$$E(y \mid x) = \beta_0 + \beta_1 x_1 + \beta_2 x_2$$

What does the tree look like?

A regression tree (two x's)

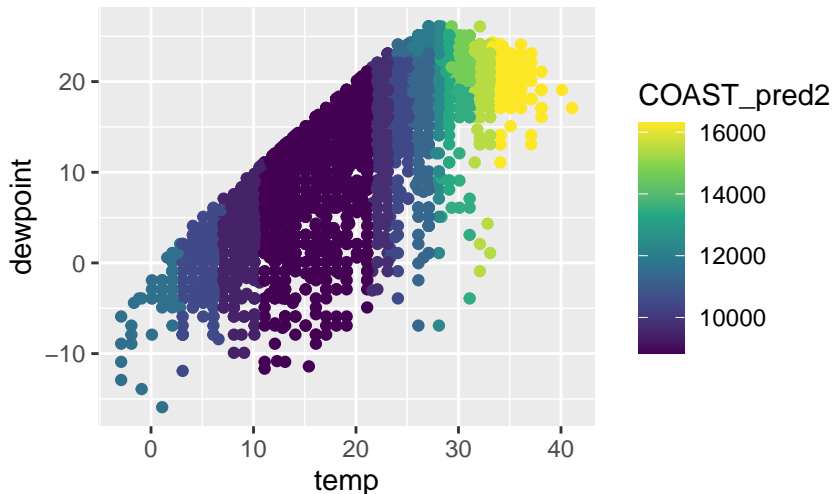


A regression tree (two x's)



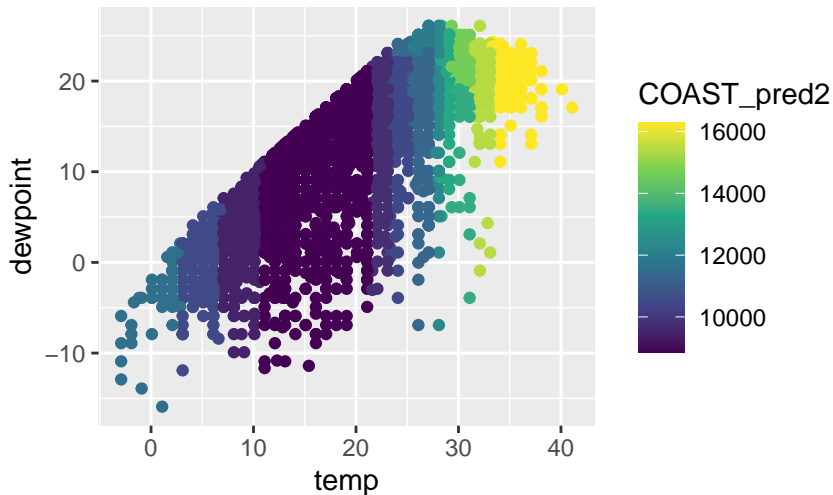
The tree partitions (x_1, x_2) space into rectangles. Within each rectangle, the fitted value $E(y \mid x_1, x_2)$ is constant.

A regression tree (two x's)



The resulting fit is a step function in the 2D plane. The regions of constant color show the steps.

A regression tree (two x's)



Key point: **notice the interaction!** Do you see it?

Trees: interactions

Trees provide automatic interaction detection (AID). For example:

- ▶ Effect of age on survival different for male and female passengers.
- ▶ Effect of dewpoint on power consumption is different at low vs. high temperatures.

AID was an original motivation for building decision trees. (Older algorithms have it in their name: CHAID, US-AID, ...)

This is pretty powerful technology:

- ▶ automatic adaptation to nonlinearity and interaction, without having to specify it in advance! (compare with `lm...`)
- ▶ Moreover, nonconstant variance is no problem.

Trees: a summary

- ▶ Trees use recursive binary splits to partition the feature space.
- ▶ Each binary split is a rule that sends x left or right.
- ▶ For numeric x , the decision rule is of the form if $x < c$.
- ▶ For categorical x , 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 x down the tree until it lands in a leaf node.
 - ▶ For numeric y , we predict using that leaf's average y value from the training data.
 - ▶ For categorical y , predict using that leaf's category proportions from the training data.

Tree models: pros and cons

Pros:

- ▶ Flexible fitters that can automatically detect nonlinearities and interactions.
- ▶ Invariant to transformations of the x variables.
- ▶ Handles categorical and numeric variables easily.
- ▶ Fast to fit.
- ▶ Interpretable (when small).

Tree models: pros and cons

Cons:

- ▶ Inherently non-smooth (step functions).
- ▶ Don't scale to very large feature sets.
 - ▶ Trees can bog down with hundreds or thousands of features.
 - ▶ But can often fix this with dimension-reduction techniques.
- ▶ Not the best at out-of-sample performance.
 - ▶ Generally must be deep to predict well (and thus less interpretable).
 - ▶ But still not bad at prediction!
 - ▶ And by *ensembling*, or averaging multiple trees, we can get excellent off-the-shelf predictions.

Fitting trees

As usual, we'll maximize data log-likelihood (minimize deviance)—here by fiddling with the tree's decision nodes.

- ▶ How many?
- ▶ What order?
- ▶ What feature and how to split that feature?

Two common loss functions:

- ▶ Regression deviance: $\sum_{i=1}^n (y_i - \hat{y}_i(x_i))^2$
- ▶ Classification deviance: $-\sum_{i=1}^n \log \hat{p}_{y_i}(x_i)$

Instead of being based on $x \cdot \beta$, predicted \hat{y} and \hat{p} are functions of x passed through the tree's decision nodes, just like we've seen.

Fitting trees

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 algorithm for fitting trees to training data: **grow big, prune back**.
- ▶ This algorithm is *greedy* and *recursive*. Let's unpack those terms.

Grow big

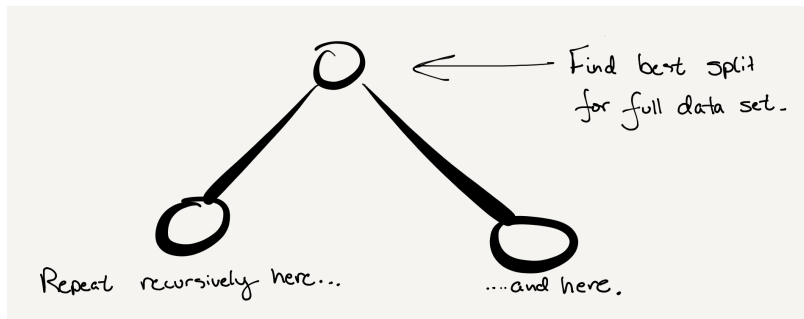
Use a greedy, recursive forward search to build a big tree, starting with all data in a single node (one leaf).

For each leaf node, *get greedy*:

1. Search over all possible splitting rules to find the single split that gives the biggest decrease in loss (increase in fit). This can be done very quickly.
2. Using this optimal rule, split this “parent” into two “children”.

Then *repeat recursively*, treating each child as a new parent. You typically stop splitting and growing when the size of the leaf node hits some minimum threshold (e.g., 10 obs. per leaf).

Grow big



The key word is *recursive*—like how trees grow in the real world!

Prune back

Given a current tree with D leaf nodes:

1. Examine every pair of “sibling” leaf nodes (i.e. leaf nodes of the tree having the same parent) and check the increase in loss (decrease in fit) from “pruning” that split.
2. Prune the “least useful” split, i.e. the prune that yields the smallest increase in loss (decrease in fit).

Repeat recursively on the newly pruned tree having $D-1$ leaf nodes. Stop when you've pruned all the way down to a single node.

Let's see this process on the board.

Prune back

Prune back

Why grow, then prune? To **generate a sequence of trees** of sizes $D, D - 1, D - 2, \dots, 2, 1$, each tree locally optimal for its size.

A good analogy is the lasso solution path:

- ▶ The big trees fit best, but have lots of splits and fewer data points in each leaf node. (Lower bias, higher variance).
- ▶ The small trees fit less well, but are simpler and have more data points in each leaf node. (Higher bias, lower variance.)

Grow-then-prune yields candidate trees that (hopefully) span the bias-variance trade-off. We can then use cross-validation to choose.

CART

This basic fitting algorithm is called CART, for “classification and regression trees.”

CART is also sometimes called “recursive partitioning” and this is reflected in the R syntax:

```
load.tree = rpart(COAST~temp + dewpoint, data=load,  
                  control = rpart.control(cp = 0.002, minsplit=30))
```

`control` gives the “stopping points” for controlling tree growth. You’ll often want to change these from their defaults. Here we split a node only if:

- ▶ it has at least 30 observations. . .
- ▶ AND if the split improves the deviance by a factor of 0.002 (0.2%).

CART

So to recap, CART:

- ▶ Grow the tree greedily and recursively to make deviance as small as possible.
- ▶ Stop growing when you hit your minimum size or complexity stopping points.
- ▶ Prune back from there to generate candidate trees.
- ▶ Choose by cross validation (min or 1SE).

Let's go see some examples in `tree_examples.R`.

CART: what we learned

From our examples, we learned a few things:

1. As tree complexity increases, CV error generally goes down quickly, levels off, and goes back up really s-l-o-w-l-y.
2. A sensible way to pick a tree is to use the “1SE rule”:
 - ▶ Choose the smallest tree whose cross-validated error is within one standard error of the minimum.
 - ▶ This gives a simpler model whose performance is not discernibly different from the best performer.
3. Trees that perform well tend to be pretty deep. (This is often true even on simple problems—but still worth trying, you might be surprised.)

CART: what we learned

This last point—that good trees tend to be deep trees—is especially concerning.

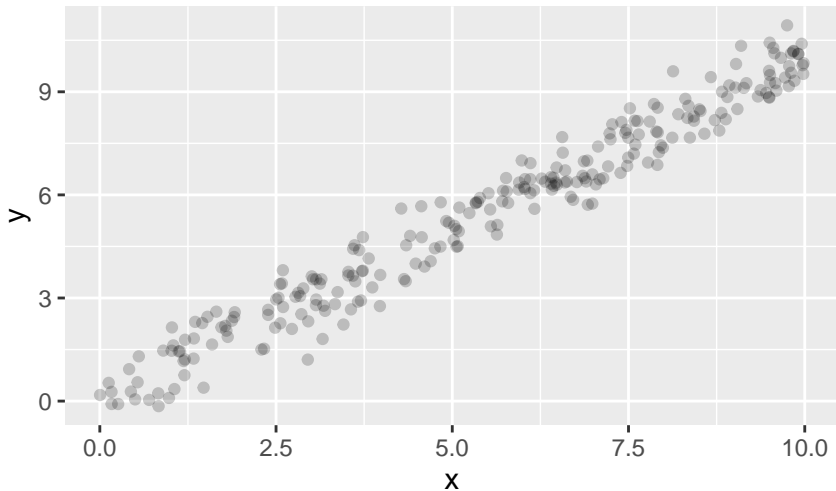
In deep trees, at least some (and perhaps most) of the leaf nodes have very few observations in them.

- ▶ This deep structure makes trees inherently prone to overfitting.
- ▶ They tend to find mini-decisions that memorize random noise, in addition to the underlying signal.

Deep trees also ruin interpretability:

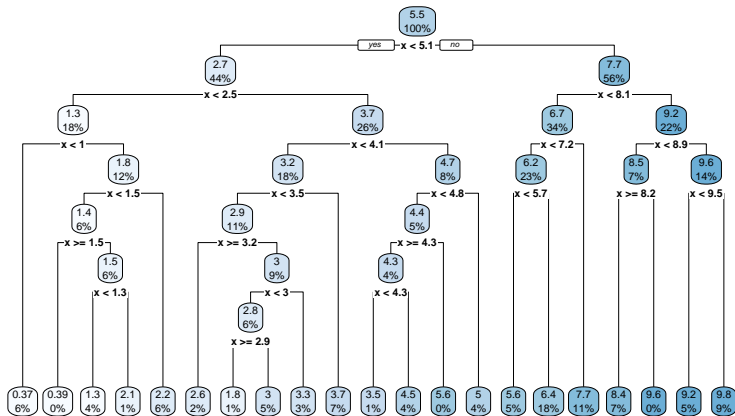
- ▶ a small tree with a handful of mini-decisions can be interpreted by a person. . .
- ▶ but probably not hundreds or thousands of mini-decisions in a deep tree.

A toy example



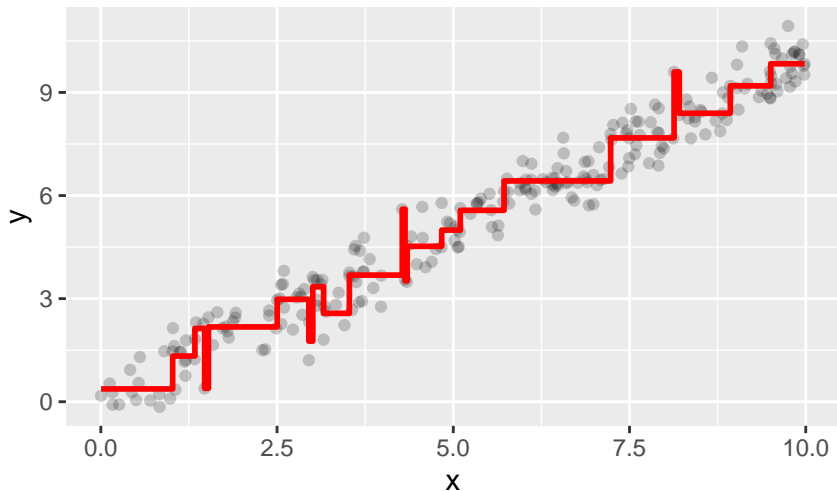
True model: $y = x + e, e \sim N(0, 1)$

A toy example



A single tree fit using the 1SE rule.

A toy example



A single tree fit using the 1SE rule.

Bagging: bootstrap aggregating

Bagging—or Bootstrap Aggregating:

- ▶ involves averaging the predictions from multiple trees.
- ▶ is a way to *reduce estimation variance without adding bias*, thereby preventing trees from overfitting quite so badly.
- ▶ doesn't address interpretability (but hey, deep trees are hard to interpret anyway!)

Bagging: bootstrap aggregating

Bagging—or Bootstrap Aggregating:

- ▶ involves averaging the predictions from multiple trees.
- ▶ is a way to *reduce estimation variance without adding bias*, thereby preventing trees from overfitting quite so badly.
- ▶ doesn't address interpretability (but hey, deep trees are hard to interpret anyway!)

Let's see *how* bagging works, before we consider *why* it works:

- ▶ **Bootstrap**: resample the data with replacement B times, to get B “jittered” versions of your original data set (each size n).
- ▶ **Fit**: for each bootstrapped data set, fit a deep tree by CART.
- ▶ **Aggregate**: when you want to predict y for some x , average the predictions from this “forest” of B trees.

Remember our basic intuition for why averaging works in the most basic problem of all: estimating a mean.

$$y_i = \mu + \epsilon_i$$

Why on earth would this work?

It's kind of the same in bagging, where we think $y_i = f(x) + e_i$:

- ▶ We take bootstrapped samples $b = 1, \dots, B$ and build lots of big trees to give an estimate $\hat{f}^{(b)}(x)$.
- ▶ We estimate $f(x)$ as the average

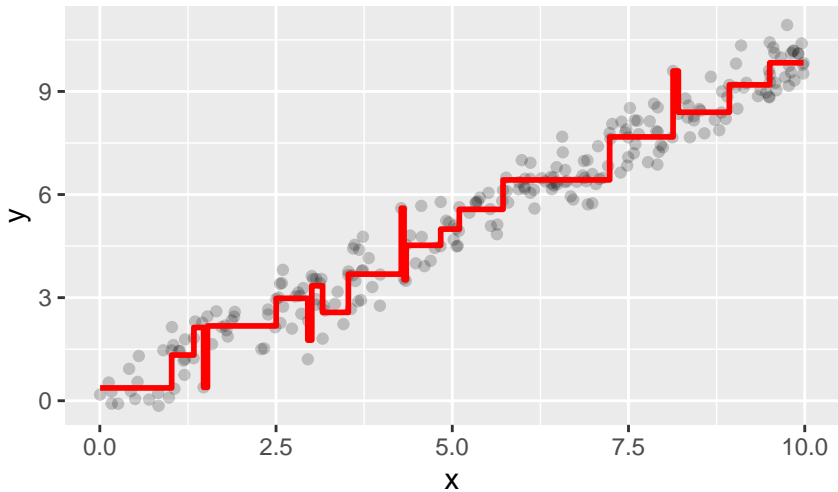
$$f(x) = \frac{1}{B} \sum_b \hat{f}^{(b)}(x)$$

- ▶ Wiggles in $f(x)$ that are real captured in most or all of the bootstrapped estimates.
- ▶ Wiggles that are over-fit to a few data points, “by chance,” are idiosyncratic to only a few estimates, and get “washed out” in the averaging.¹

Leo Breiman. **Brilliant.**

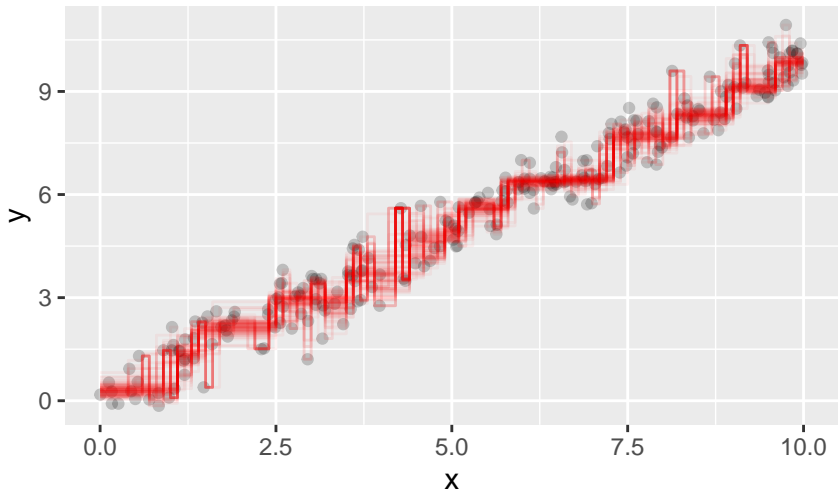
¹The math is a bit more complicated than on the previous slide, because the bootstrapped estimates are correlated with each other. But they're not perfectly correlated, which is the source of variance reduction.

Let's see it work.



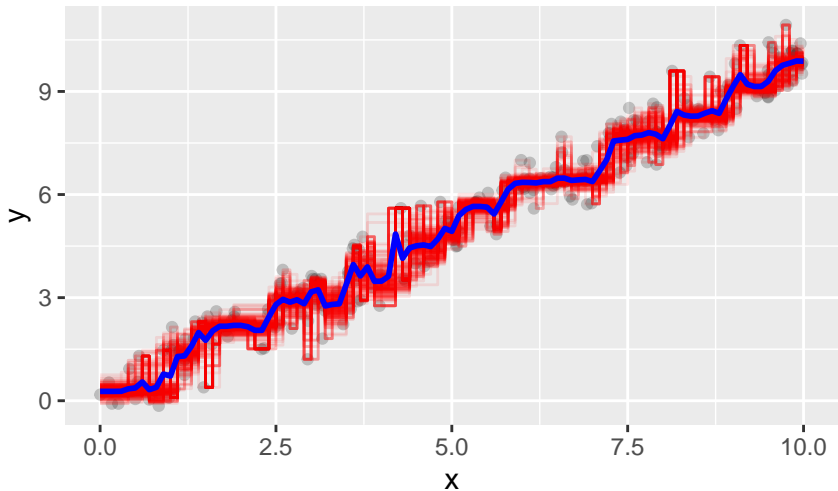
Original fit.

Let's see it work.



Fits from 100 bootstrapped samples.

Let's see it work.



Average of fits from 100 bootstrapped samples.

Bagging: a summary

Fit trees to B bootstrapped samples of the original data.

- ▶ For numerical y , average the predictions.
- ▶ For categorical y , let each tree vote, or average the trees' predicted class probabilities.

You need B large enough to enjoy the effect of averaging.

- ▶ 100 is a decent starting point, 500 is better if your machine has the flops and memory.
- ▶ It doesn't seem to hurt if you make B even larger.
- ▶ The only real cost of a large B is computational time.

Bagging: a summary

Bagging “works” because it usually gives you a smoother (lower-variance) fit than a single tree. This is quite a general phenomenon:

- ▶ we can often improve a high-variance nonparametric regression model by bagging it. This is the simplest form of “ensembling” or “model averaging”—super useful idea.
- ▶ the corollary is that a stable, low-variance estimator (e.g. a linear model) usually won’t be improved by bagging.
- ▶ trees are ideal candidates for bagging because they’re high-variance, flexible fitters that are also quite fast to fit.

The downside:

- ▶ when we bag a model, any simple structure is lost.
- ▶ this is because a bagged tree is no longer a tree, but a forest!

Random forests

A “random forest” starts from bagging...

- ▶ We still take B bootstrapped samples of the original data and fit a tree to each one.
- ▶ We still average the predictions of the B different trees.

But it adds **more randomness**. Within each bootstrapped sample:

- ▶ We don't search over **all** the features in x when we do our greedy build of a big tree.
- ▶ Instead, we randomly choose a subset of $m < p$ features to use in building that tree.

This does two things:

- ▶ it simplifies each tree, reducing its variance
- ▶ it diversifies the B trees, decorrelating their predictions

Random forests

Why would using fewer features in each tree actually help!?

An analogy: evolutionary co-adaptation.

- ▶ Co-adaptation occurs when two or more traits/genes undergo adaptation together as a group.
- ▶ Co-adapted traits involve mutually adjusted changes. Classic example: flowering plants and pollinating insects.

A common way that trees overfit is by learning heavily “co-adapted” sets of features, i.e. deep interactions that explain noise.

- ▶ Example: “No Democratic presidential incumbent without military service has ever beaten a challenger whose last name is worth more in Scrabble.”
- ▶ By forcing each tree to rely on only a few features, we force it to learn robust, generalizable relationships.

Random forests

In random forests, you must choose:

- ▶ B: number of bootstrapped samples. Use hundreds, or thousands if possible!
- ▶ m: number of features to sample within each bootstrapped sample. A common choice is $m \approx \sqrt{p}$.

Some notes:

- ▶ bagging is just random forests with $m = p$, but you'll typically see better performance with $m < p$.
- ▶ there is no explicit regularization parameter, as in the lasso and single-tree models.
- ▶ random forests might be the most popular “off the shelf” nonparametric regression technique. They're effective, fast, and require little or no tuning via CV (the default settings perform well).

“Out-of-bag” error estimation

With random forests, there's a nice built-in way to estimate the generalization error of the model.

- ▶ In each bootstrap sample (“bag”), some of your original observations are “in the bag” (math says: about $2/3$, on average).
- ▶ The rest are “out of bag.”

By carefully keeping track of which trees use which observations, you can get “out-of-bag” predictions.

- ▶ This is a decent way to estimate out-of-sample performance.
- ▶ We typically use this to reassure ourselves we've used enough trees in the forest.

Random forests: example

Let's go to `random_forest_example.R` to see random forests in action.

Boosting