Lecture 8: Trees & Ensembles

STAT 1361/2360: Statistical Learning and Data Science

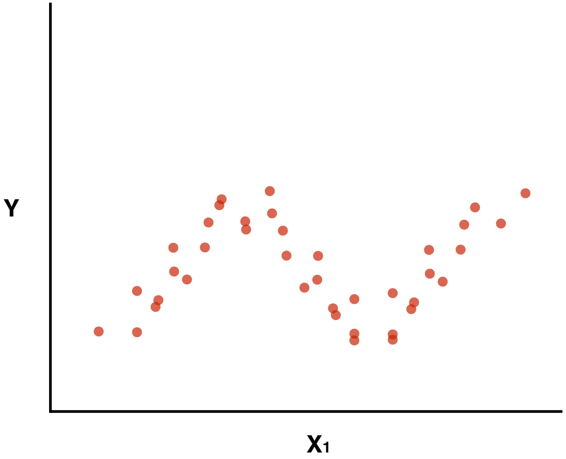
University of Pittsburgh
Prof. Lucas Mentch

Overview

- In Chapter 7 we covered a few more flexible models: Splines, GAMs, and Local Linear Regression
- In Chapter 8, we'll cover tree-based models and their ensemble extensions (bagging, boosting, and random forests)
- We'll start with exactly the same motivation that we used for splines and see that we need to make only one important modification ...

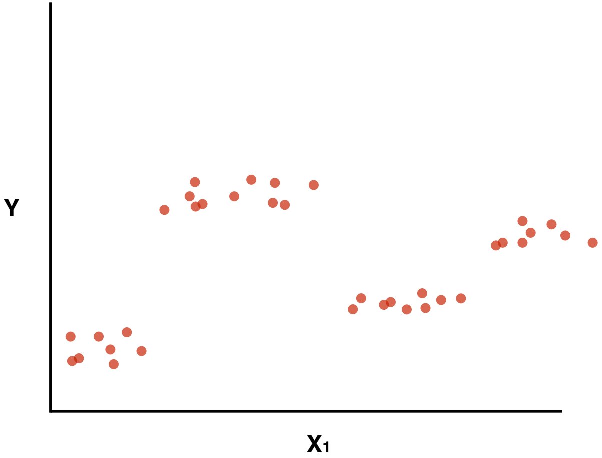
Motivating Examples

Consider the following data examples:



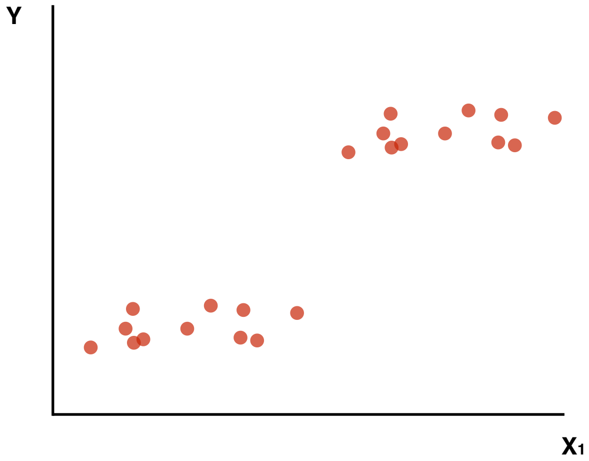
Motivating Examples

Consider the following data examples:



Motivating Examples

Consider the following data examples:



Fitting a Spline

We said in the last lecture that ideally we'd want to fit a different model in different regions. If I wanted to fit a spline, what are all the things I need to decide?

Fitting a Spline

We said in the last lecture that ideally we'd want to fit a different model in different regions. If I wanted to fit a spline, what are all the things I need to decide?

1. **The number of regions**
2. **The locations of those regions**
3. **The kind of models I want to fit in each region**

Fitting a Spline

What's the downside to this?

Fitting a Spline

What's the downside to this?

In practice, we often have a lot of observations and predictors – hard to determine where to put those regions and how many there should be.

Fitting a Spline

What's the downside to this?

In practice, we often have a lot of observations and predictors – hard to determine where to put those regions and how many there should be.

Idea: The number of regions and where they are will depend on the flexibility of the models chosen. If I fix that, could I use the data to determine where and how many of those models to fit?

Fitting a Spline

This is the big idea behind trees: we'll use **constant** models and rely on the data to determine where to put them, how many there should be, and what values those constants should take.

Fitting a Spline

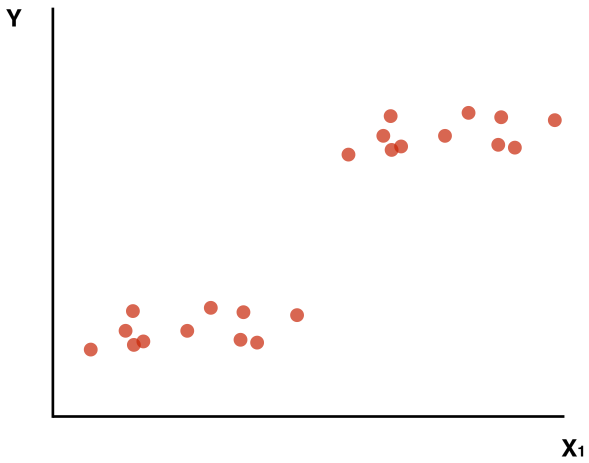
This is the big idea behind trees: we'll use **constant** models and rely on the data to determine where to put them, how many there should be, and what values those constants should take.

How should we do this?

- Let's start with a simple example where we have one predictor X_1 and a continuous response Y. We'll also start by looking at a situation where the underlying relationship actually looks like it should be piecewise constant:

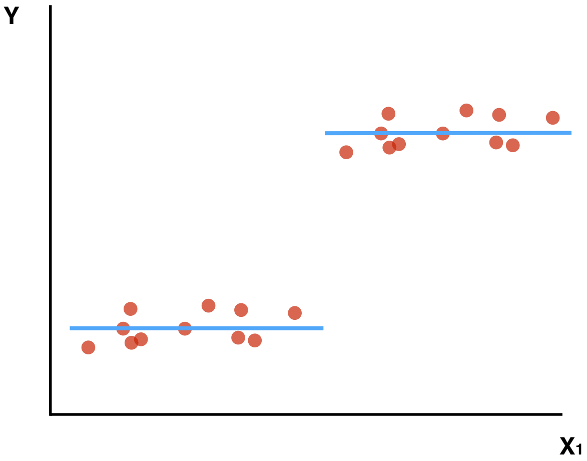
Simple First Example

Where should the constants go in this model?



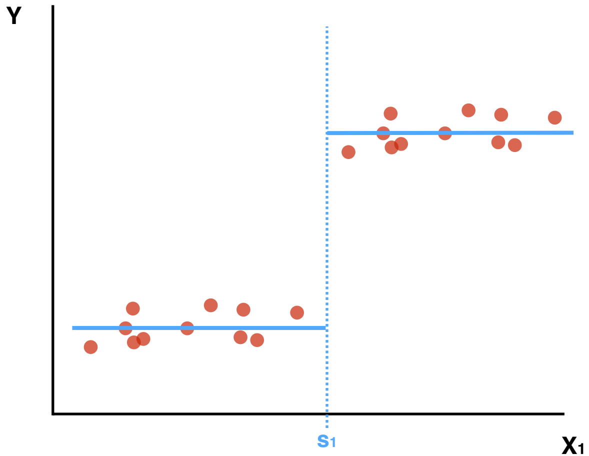
Simple First Example

Here, right? Why?



Simple First Example

Rephrasing, why should we “split" at s_1?



Tree Basics

Because that's what produces the best model! (i.e. It fits the data best).

Tree Basics

Because that's what produces the best model! (i.e. It fits the data best).

More specifically, given that we're fitting constants, the “best" constant will always be at the mean of the observations in that region. How do we decide the regions?

Tree Basics

Because that's what produces the best model! (i.e. It fits the data best).

More specifically, given that we're fitting constants, the "best" constant will always be at the mean of the observations in that region. How do we decide the regions?

$\Longrightarrow$ **We can simply slide the split point s along the X_1-axis, see how well the resulting piecewise constants fit the data at each potential split point, and choose the split point that best fits the data.**

Tree Basics

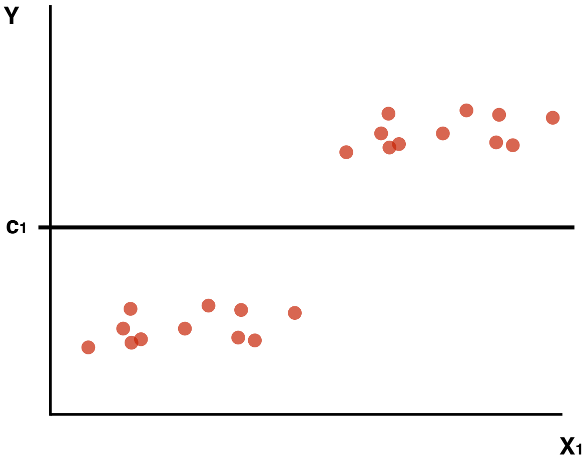
- Trees begin with a single constant at the overall mean value of the response (stump)
- To go from the model with just one constant to a model with two constants, we need to decide where to split
- Each potential split point s_i induces two regions (left and right).
- The potential new model fits a constant in both regions at the mean of the response values in those regions
- This gives us an MSE for both models (left and right), so

$$MSE_s = MSE_{s,\text{Left}} + MSE_{s,\text{Right}}$$

- Given $n-1$ potential split points, $s_1, s_2, \ldots, s_{n-1}$, we choose the one with the lowest resulting MSE, MSE_{s_i}

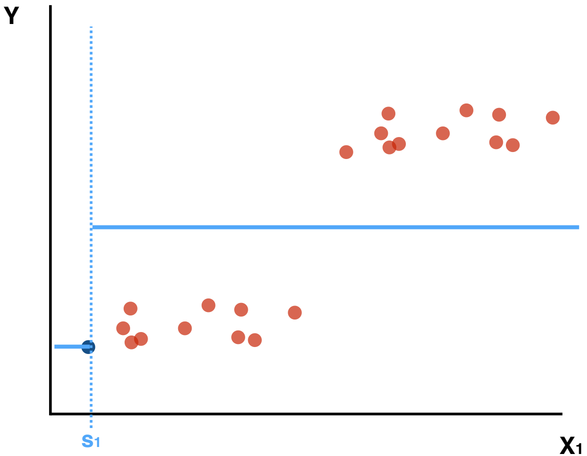
In Pictures ...

We start with the overall constant model (stump)



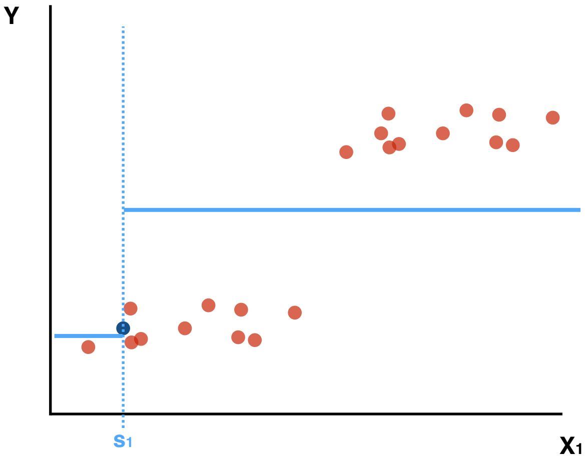
In Pictures ...

Consider the first possible split and recording resulting error ...



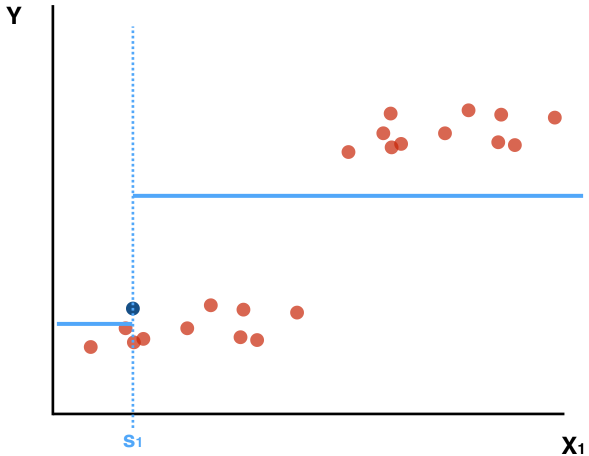
In Pictures ...

Consider the next possible split and recording resulting error ...



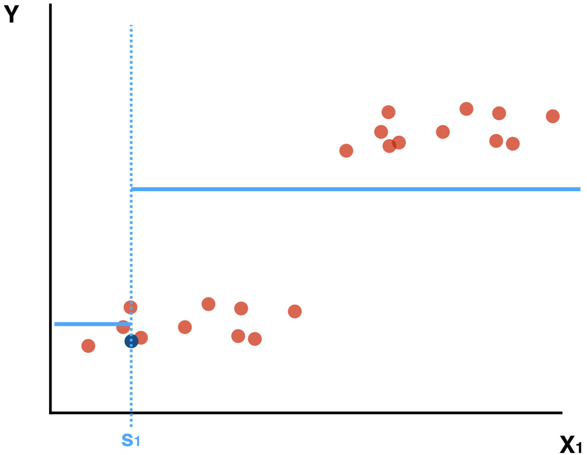
In Pictures ...

And repeat for the next possible split ...



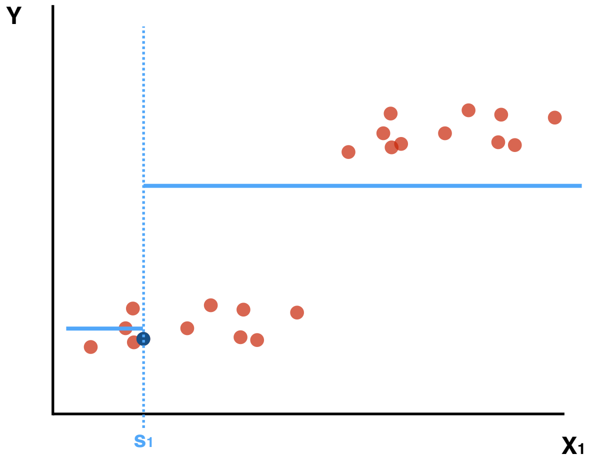
In Pictures ...

And the next possible split ...



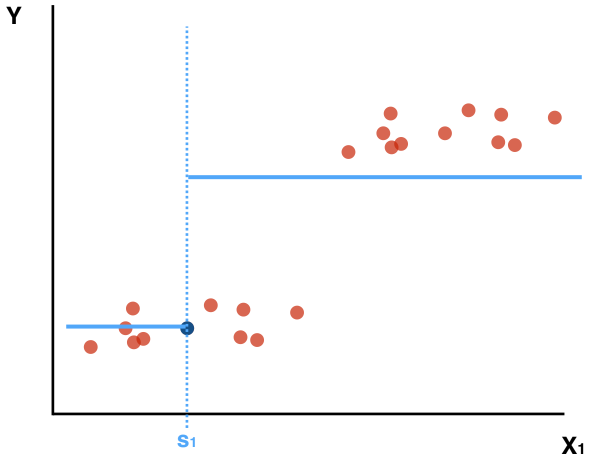
In Pictures ...

And the next possible split ...



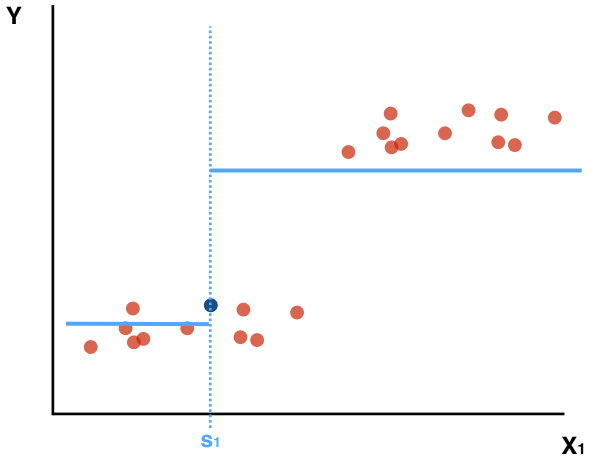
In Pictures ...

And the next possible split ...



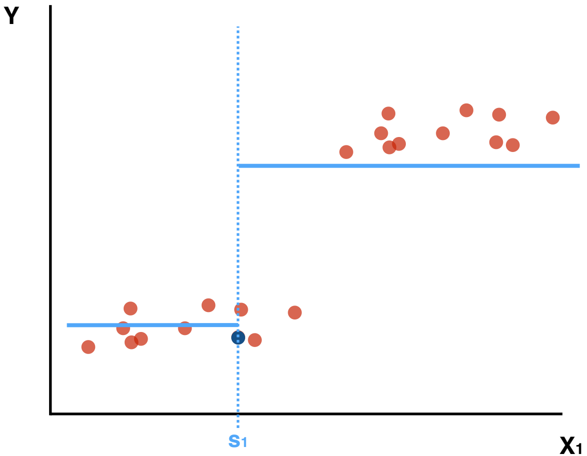
In Pictures ...

And the next possible split ...



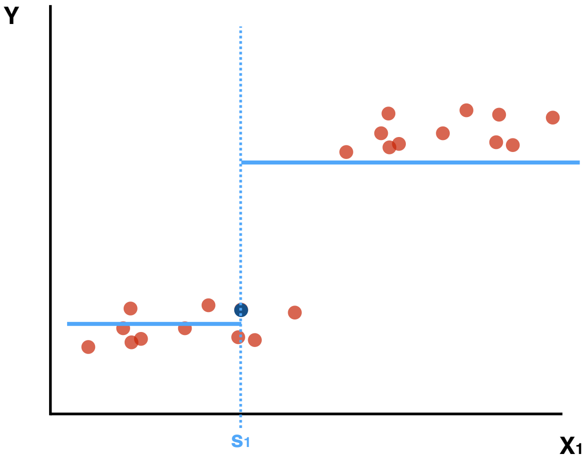
In Pictures ...

And the next possible split ...



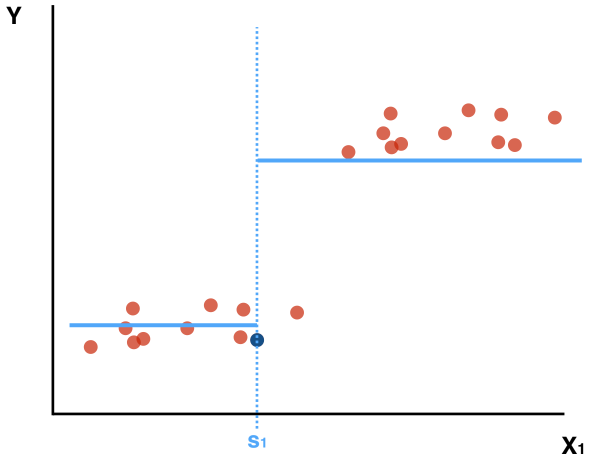
In Pictures ...

And the next possible split ...



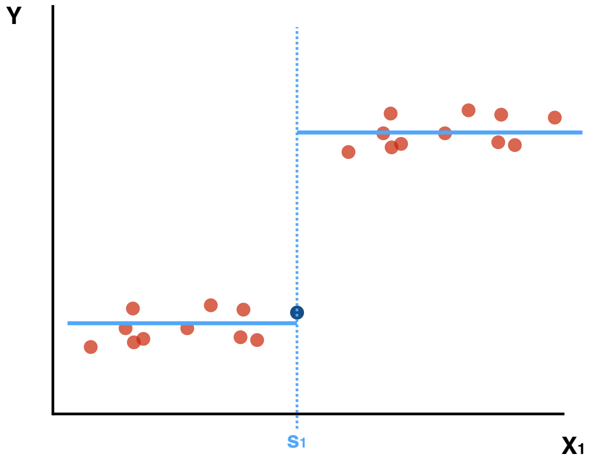
In Pictures ...

And the next possible split ...



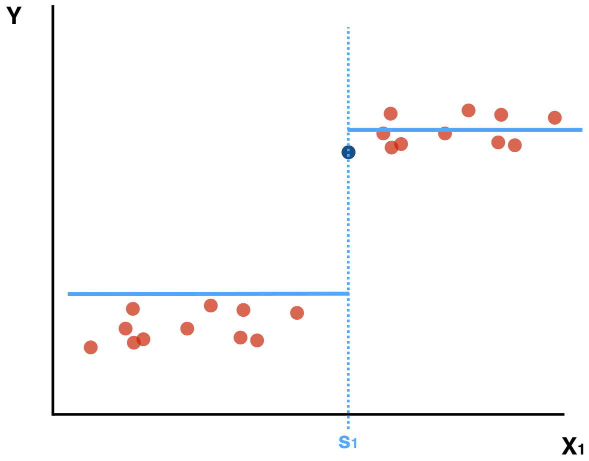
In Pictures ...

And the next possible split ...



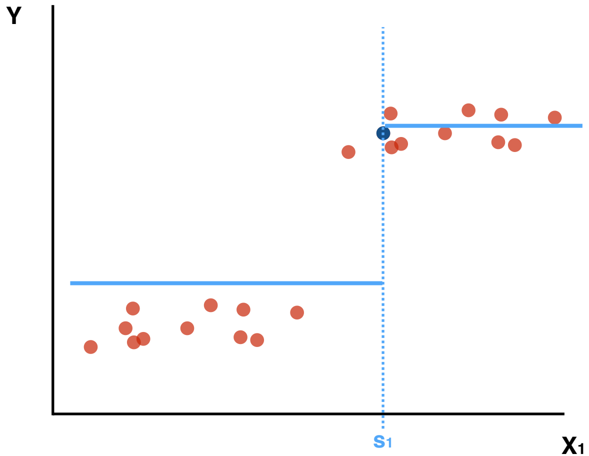
In Pictures ...

And the next possible split ...



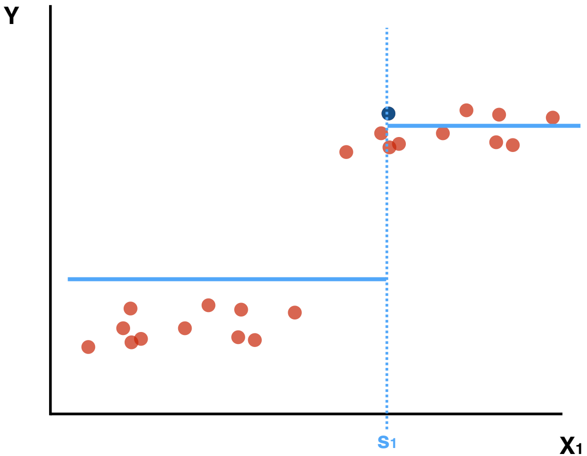
In Pictures ...

And the next possible split ...



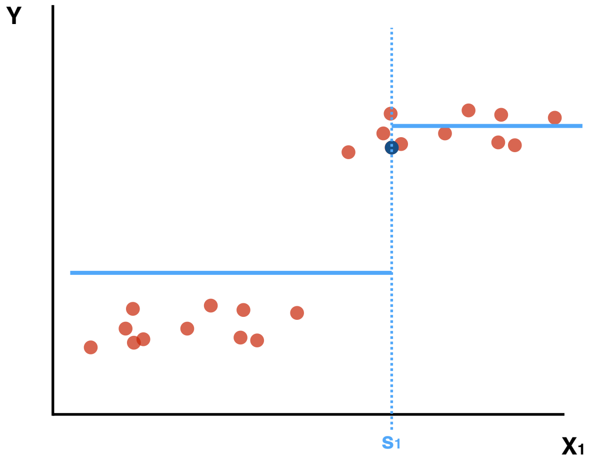
In Pictures ...

And the next possible split ...



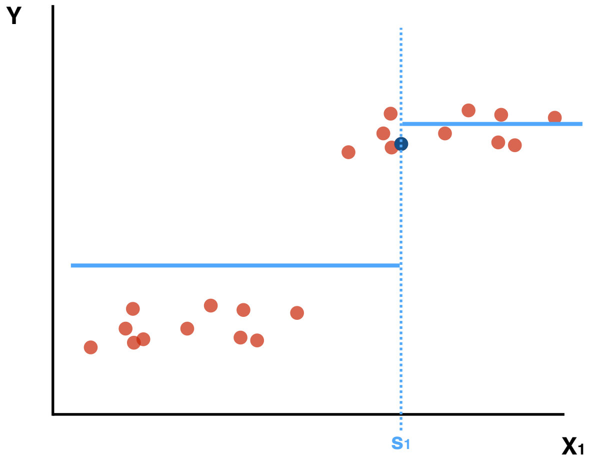
In Pictures ...

And the next possible split ...



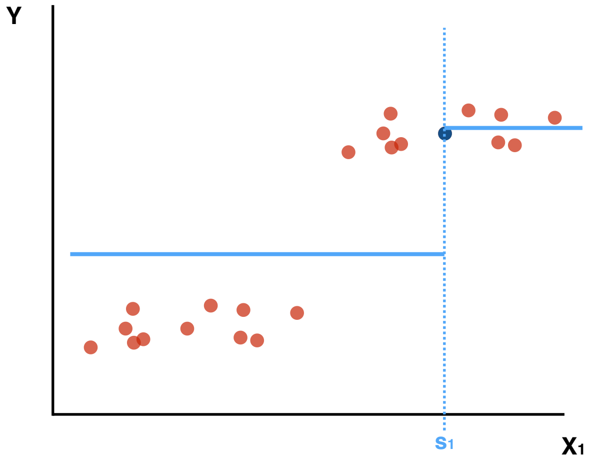
In Pictures ...

And the next possible split ...



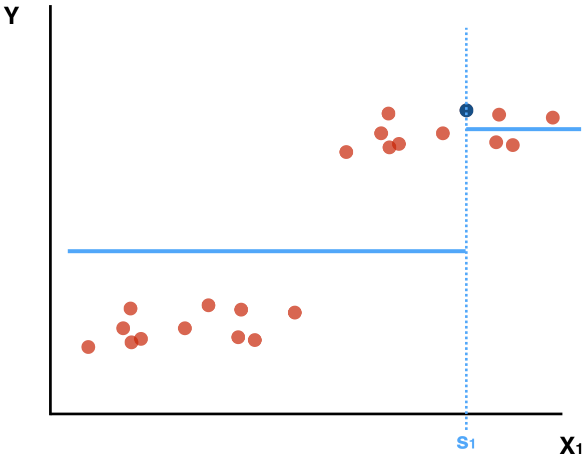
In Pictures ...

And the next possible split ...



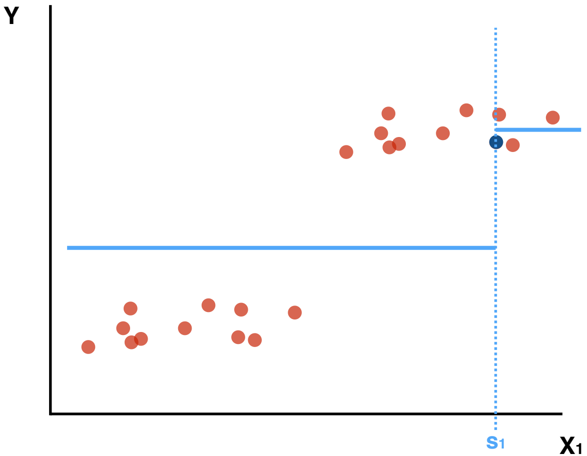
In Pictures ...

And the next possible split ...



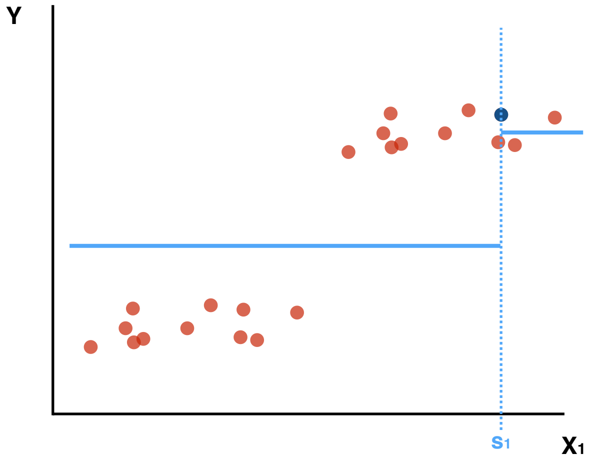
In Pictures ...

And the next possible split ...



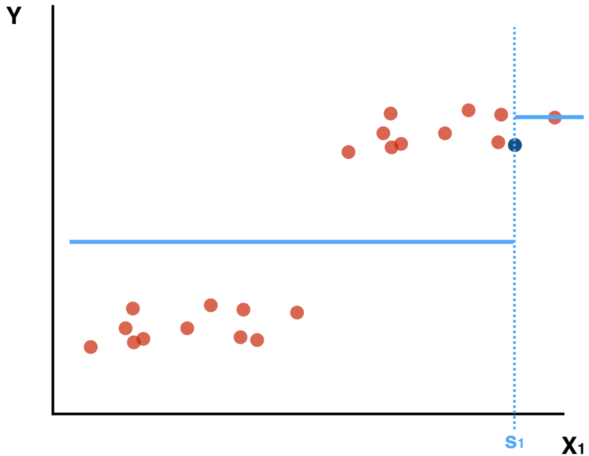
In Pictures ...

And the next possible split ...



In Pictures ...

And the next possible split ...



Tree: First Split

- Note that since our models only depend on which observations fall into each region, we need only consider splits at the data points we have
- For a given split location s, the resulting regions are defined by

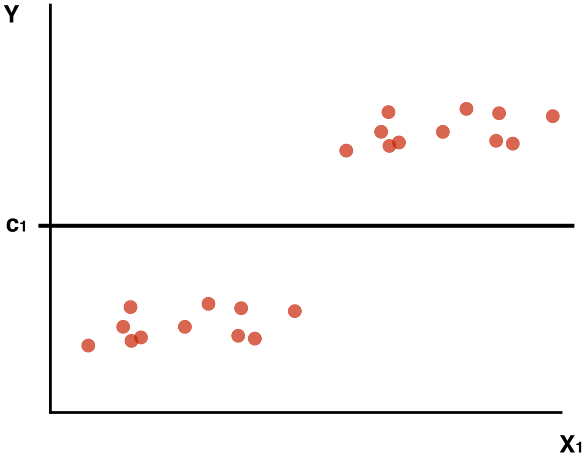
$$\begin{aligned} \text{Left:} \quad & \{(x_1, y) : x_1 \leq s\} \\ \text{Right:} \quad & \{(x_1, y) : x_1 > s\} \end{aligned}$$

  (This is by default; other options possible)

- Once we find the resulting MSE from every potential new model, we make the split corresponding to the minimum MSE

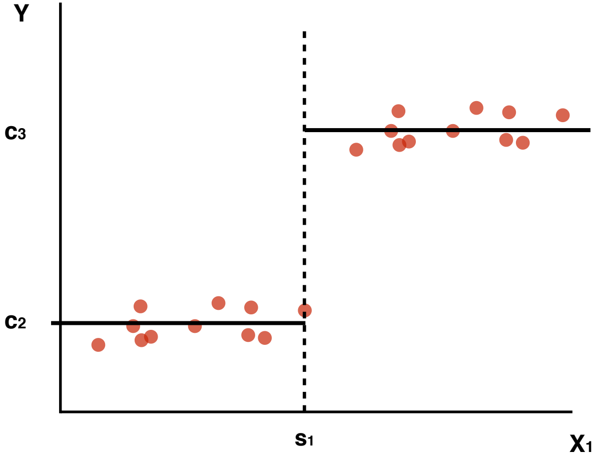
In Pictures ...

Original model with no splits (one constant):



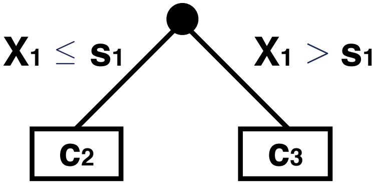
In Pictures ...

New model with one split (two constants):



In Pictures ...

Now we can start to see why these kinds of models are referred to as trees:



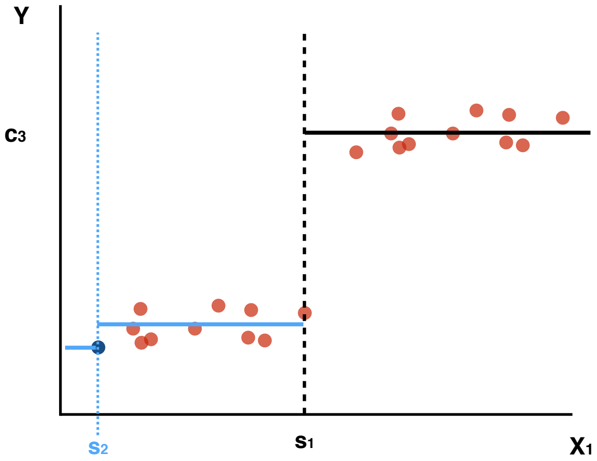
Tree: Future Splits

Now we have a model consisting of two regions and a constant model in each region. Moving forward, we simply repeat the exact **same process in each region**.

- Consider all possible splits in the first region. Let s_{R_1} be the split that reduces MSE the most in Region 1.
- Consider all possible splits in the second region. Let s_{R_2} be the split that reduces MSE the most in Region 2.
- Choose the region and split that reduces the MSE more between s_{R_1} and s_{R_2}

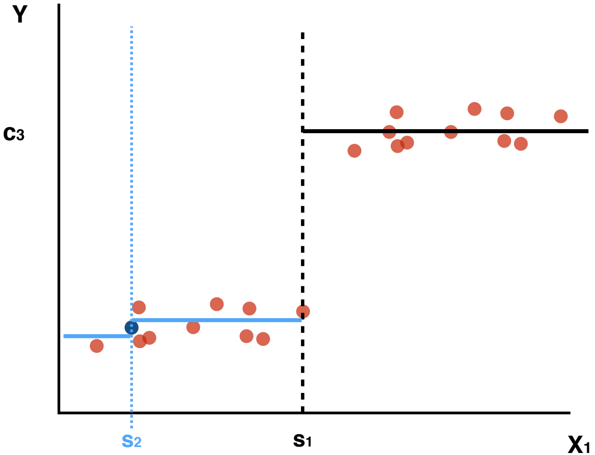
In Pictures ...

Let's start with the region on the left:



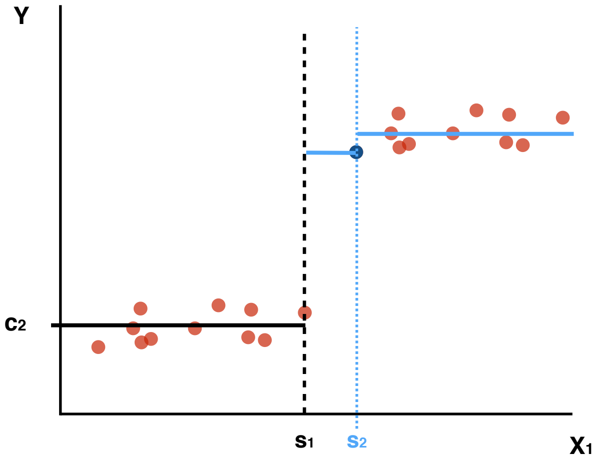
In Pictures ...

Try every possible split, record the best ...



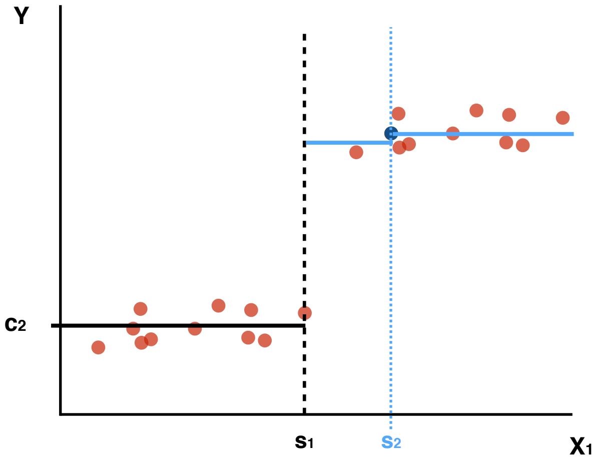
In Pictures ...

Then go to the region on the right ...



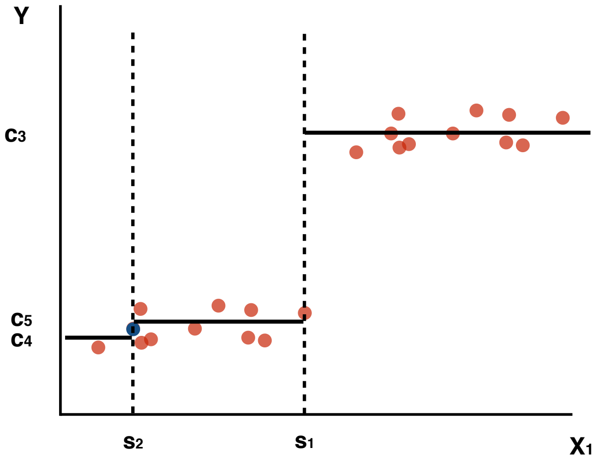
In Pictures ...

Try every possible split in this region, and again record the best



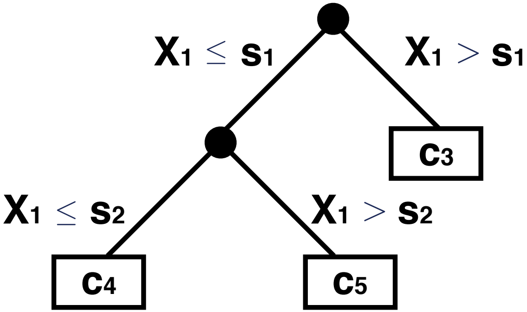
In Pictures ...

Let's suppose the best was this split point in the left region



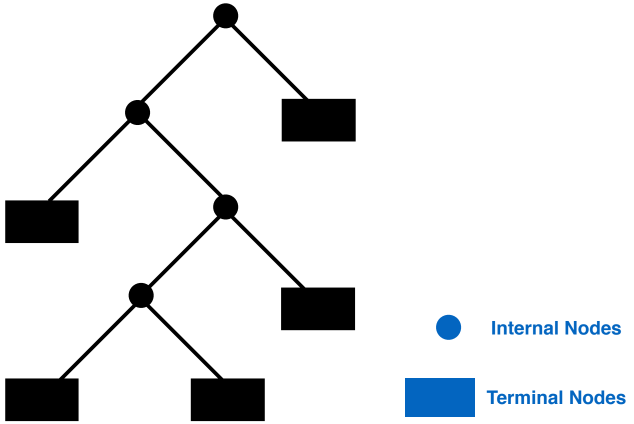
In Pictures ...

And now our tree model looks like this:



In Pictures ...

In general, we can continue in this fashion and get models of this form:



Trees: General Procedure

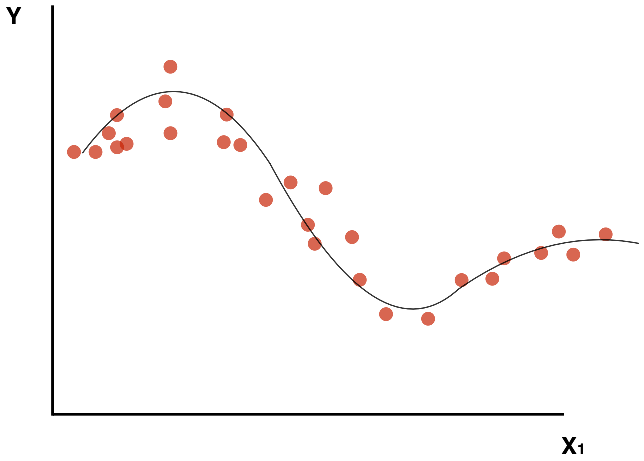
The big idea with tree-based models is that we split the data into several regions, fit a simple model (constant) in each region, and make predictions by finding the corresponding terminal node.

More precisely, say we divide the space into M regions $A_1, \ldots, A_M$ and in each region, we take the average of the points in that region to get constants $c_1, \ldots, c_M$. Our prediction at some point x takes the form

$$\hat{f}(x) = \sum_{i=1}^{M} c_i \mathbf{1}\{x \in A_i\}$$

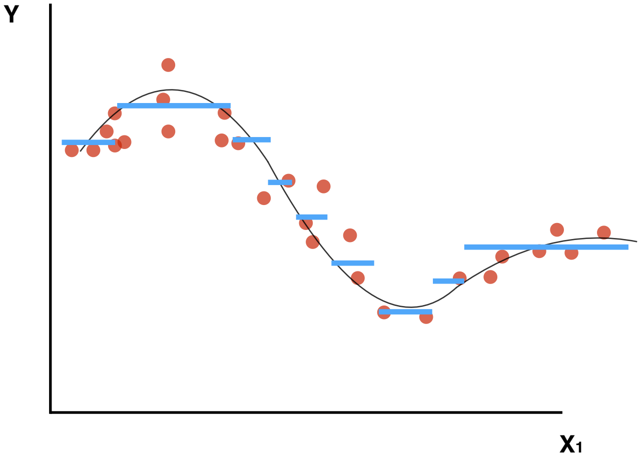
In Pictures ...

In general, we might have data that looks like this ...



In Pictures ...

... and a tree-based model fits something like this:



Trees: Lingering Issues

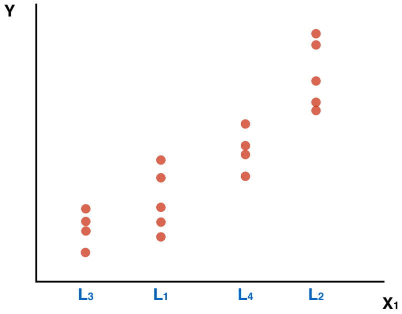
Note that there are still many issues we haven't addressed:

- What if we have categorical predictors?
- What do we do when we have more than 1 predictor?
- What if we have a categorical response?
- When do we stop splitting?

Trees: Categorical Predictors

What if a predictor is categorical?

- If it's ordinal, we don't really need to change anything
- If not ordinal, we can re-order by within-level mean response:



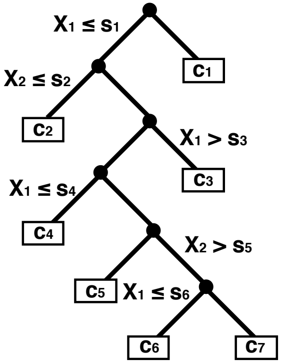
Trees: Multiple Predictors

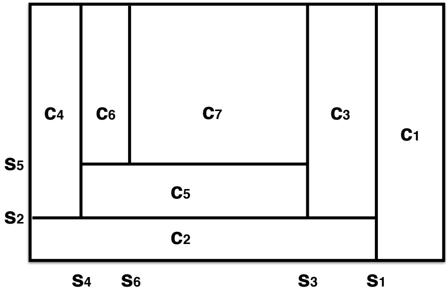
What if we have multiple predictors? This is actually relatively straightforward:

- Suppose that we're within a given region A
- With one predictor, we considered only the best place to split in A along X_1
- With multiple predictors, we repeat that process for each predictor $X_1, \ldots, X_p$, find the best place to split in each direction, then choose from amongst the best of these
  - This is known as a **greedy** approach – we're always looking for the next best place to split and update the model

In Pictures ...

It's important to understand the correspondence between images:





Trees: Categorical Response

What if we have a categorical response?

- If it's ordinal (or binary, $\{0, 1\}$), we still have the option of splitting via MSE and averaging
- If not ordered, then averaging and fitting with constants doesn't make sense. Instead, we take a *majority vote* in each region / terminal node (same as kNN)
- Trees fit via averaging (with constants) are **regression trees**; Trees fit via majority vote are **classification trees**

Trees: Categorical Response

This takes care of how to fit the model in each region, but how do we make splits? Two options:

1. **Gini Index:**

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

   where $\hat{p}_{m,k}$ is the proportion of observations in the m^{th} region from the k^{th} class

   $\Longrightarrow$ smaller values mean most responses in region are from same class (node purity)

2. **Cross-Entropy:**

$$D = -\sum_{k=1}^{K} \hat{p}_{m,k} \log(\hat{p}_{m,k})$$

   Same intuition, small value if all $\hat{p}_{m,k}$ are close to 0 or 1

Trees: Stopping Criteria

How many constants/regions should there be? When should we stop growing the tree?

- Same old model complexity issue we've faced with nearly every model we've talked about: under- vs overfitting; bias vs. variance (deep trees = low bias, high variance)
- Trees with only one observation per terminal node are said to be "fully grown"
- How have we dealt with this issue for other models?

  Cross-validation! We can do the same here, build trees to different depths and examine CV error.

Trees: Pruning

Final tree models can be obtained in a different way than with other models

- Instead of limiting how deep a tree can be grown, grow the trees to full-depth and then cut off the splits that are the least helpful. This process is called **pruning**.
- Most common way to prune is via a *cost-complexity* tuning parameter α
    - Same idea as we've seen before: larger $\alpha \implies$ stronger penalty $\implies$ encourages shallower trees

Trees vs Linear Models

- Linear models are of the form $f(X) = \beta_0 + \sum_{i=1}^{p} \beta_i X_i$
- Tree models are of the form $f(X) = \sum_{i=1}^{M} c_i \mathbf{1}\{X \in A_i\}$
  - Tree models are inherently simpler, but as $n \longrightarrow \infty$, more constants can be added giving a more precise fit and the **ability to adapt** to a wide variety of true underlying regression problems
- Trees avoid the need to create explicit dummy variables for categorical predictors
- Trees can still provide a model fit when $p > n$

Trees vs kNN

Trees should remind you a lot of k-nearest neighbors. What's the key difference?

Trees vs kNN

Trees should remind you a lot of k-nearest neighbors. What's the key difference?

- Both are *local* methods (ultimately use only data that's close by)

Trees vs kNN

Trees should remind you a lot of k-nearest neighbors. What's the key difference?

- Both are *local* methods (ultimately use only data that's close by)
- Both can handle either regression (averaging) or classification (majority vote)

Trees vs kNN

Trees should remind you a lot of k-nearest neighbors. What's the key difference?

- Both are *local* methods (ultimately use only data that's close by)
- Both can handle either regression (averaging) or classification (majority vote)
- Both have a tuning parameter, k (kNN) or α (trees), to control how many local observations to use

Trees vs kNN

Trees should remind you a lot of k-nearest neighbors. What's the key difference?

- Both are *local* methods (ultimately use only data that's close by)
- Both can handle either regression (averaging) or classification (majority vote)
- Both have a tuning parameter, k (kNN) or α (trees), to control how many local observations to use
- **But ... Trees use the response Y to determine which local points to use, kNN does not. Terminal nodes in trees needn't contain the same number of points. In some sense, can think of trees as "supervised" kNN.**

Trees Wrap-up

Final thought ... why constants? Couldn't we fit something a little more complex (e.g. linear models) in each region?

Trees Wrap-up

Final thought ... why constants? Couldn't we fit something a little more complex (e.g. linear models) in each region?

- Sure! But this makes the bias-variance trade-off issue more complicated. Fitting more complex models makes it very easy to overfit.

- What did we say about sticking with cubic splines in the last chapter?

  If the model was more complex, we could just fit more of them as long as we have enough data. Exact same argument can be made for fitting constants in trees.

Tree Ensembles

Tree Ensembles

- Individual classification or regression trees can sometimes fit well and have the advantage of being interpretable when relatively shallow
- In practice though, especially when predictive accuracy is the primary goal, *ensembles* of decision trees are very common. We'll look at three popular ensemble extensions of decision trees:
    1. Bagging
    2. Random Forests
    3. Boosting

Bagging

Bagging

- Individual trees, even when relatively shallow, tend to have high variance
  - Changes to original data can result in substantial changes to the model (tree)

Bagging

- Individual trees, even when relatively shallow, tend to have high variance
  - Changes to original data can result in substantial changes to the model (tree)
- Instead of building a single tree, suppose that to predict at some point $\boldsymbol{x}$, we built several (B) trees, made predictions at $\boldsymbol{x}$ using each, and took our final prediction as the average (majority vote) across all:

$$\tilde{T}(\boldsymbol{x}) = \frac{1}{B}\sum_{i=1}^{B} T_i(\boldsymbol{x})$$

where $T_i(\boldsymbol{x})$ denotes the prediction at $\boldsymbol{x}$ from the i^{th} tree

Bagging

- But to build different trees, we need different datasets. How can we obtain a collection of similar (but non-identical) datasets?

Bagging

- But to build different trees, we need different datasets. How can we obtain a collection of similar (but non-identical) datasets?

  Bootstrapping! In fact, we've seen this idea before:

Bagging

From Lecture 6 ...

Given bootstrap estimates $\hat{\theta}_1^*, \ldots, \hat{\theta}_B^*$, define a new estimate

$$\tilde{\theta}^* = \frac{1}{B}\sum_{i=1}^{B}\hat{\theta}_i^*$$

- This results in a more stable estimate of θ – the variance of the sampling distribution of $\tilde{\theta}^*$ is smaller than that of the original $\hat{\theta}_0$. (Think if we got a new original dataset and repeated the entire procedure)
- In general, we refer to $\tilde{\theta}^*$ as the *bagged* estimate of θ (**b**ootstrap **agg**regat**ed**), or sometimes just the *bootstrap* estimate of θ

Bagging

Bagging (**b**ootstrap **agg**regat**ing**) is a general procedure useful for obtaining estimates/predictions with a lower variance (more stable) when individual models may have high variance.

In the context of trees specifically, the bagging procedure is:

1. Take B bootstrap samples of original data
2. With each bootstrap sample, create a tree model to obtain $T_1, \ldots, T_B$
3. To make a prediction at $\boldsymbol{x}$, average the predictions from the individual trees
$$\tilde{T}(\boldsymbol{x}) = \frac{1}{B} \sum_{i=1}^{B} T_i(\boldsymbol{x})$$

Bagging

- Predictions from bagging usually a bit more accurate than from individual trees, though often not a huge improvement
- In practice, 30-50 bootstrap samples (trees) are all that's required before things stabilize
- Typically, trees are built to full-depth (no pruning) – why?

Bagging

- Predictions from bagging usually a bit more accurate than from individual trees, though often not a huge improvement
- In practice, 30-50 bootstrap samples (trees) are all that's required before things stabilize
- Typically, trees are built to full-depth (no pruning) – why?

  Each tree likely to overfit, but in different directions; overfitting should average out.

Random Forests

Random Forests

[2] Random Forests:

- The random forest procedure is similar to bagging – draw bootstrap samples, build trees with each, and average over predictions
- Key difference: With bagging, we assume all trees are built in the same fashion. With random forests, each tree is assumed to contain some additional randomness (denoted here by ω):

$$\tilde{T}_{RF}(\boldsymbol{x}) = \frac{1}{B} \sum_{i=1}^{B} T_{\omega,i}(\boldsymbol{x})$$

Random Forests

Where does this randomness come from?

- Recall that trees are traditionally built in a *greedy* fashion: find best split point for each variable, choose the best from amongst these
- With random forests, at each node we select (uniformly at random) a subset of $d < p$ predictors and choose the best split *only* from amongst those d
- The number of available predictors d is often referred to as $mtry$ and is a **tuning parameter** (though very robust); by default, $mtry = \sqrt{p}$ for classification and $mtry = p/3$ for regression
  - **Note:** $mtry = p$ reduces to bagging

Random Forests

- As with bagging, trees in a random forest are typically fully-grown
- Random forests should contain many more trees (≈ 500) than bagging (≈ 30). Why?

Random Forests

- As with bagging, trees in a random forest are typically fully-grown
- Random forests should contain many more trees (≈ 500) than bagging (≈ 30). Why?

  More randomness $\implies$ need more samples to stabilize predictions

Random Forests

- As with bagging, trees in a random forest are typically fully-grown
- Random forests should contain many more trees (≈ 500) than bagging (≈ 30). Why?

  More randomness $\implies$ need more samples to stabilize predictions
- Random forest predictions are often *substantially* more accurate than individual trees or even bagging.
  - Fernandez-Delgado et al. (2014), "Do we need 100's of classifiers ..." (JMLR): Random forests performed best on average compared with >100 classifiers on >100 datasets
  - Random forests still considered amongst the best "off-the-shelf" learning methods

Random Forests

Why do random forests work so well?

- Does it make sense that random forests would outperform bagging?

Random Forests

Why do random forests work so well?

- Does it make sense that random forests would outperform bagging?

  It shouldn't – bagging is always choosing the "best" overall split.

Random Forests

Why do random forests work so well?

- Does it make sense that random forests would outperform bagging?

  It shouldn't – bagging is always choosing the "best" overall split.

- Original intuition: With bagging, trees are built in same way with bootstrap samples $\Longrightarrow$ trees and predictions highly correlated
  - The randomness in forests helps to reduce this correlation – reduce the variance at the averaging step
  - We'll return to this topic in much more detail in a future lecture

Boosting

Boosting

[3] Boosting:

- With both bagging and RFs, the trees are built in an independent fashion – results of one sample/tree don't affect any others. This is not the case with boosting.
- Boosting Idea Overview:
  1. Fit model and get residuals
  2. Fit new model to residuals and add to existing model
  3. Continue until no pattern remains

Boosting

More formally with (shallow) tree models:

1. Fit (shallow) tree model $\hat{f}_1$ to original data
2. Record *shrunken* residuals $r_1(\boldsymbol{x}) = y - \lambda \hat{f}_1(\boldsymbol{x})$
3. Fit new tree model $\hat{f}_2$ to residuals (i.e. treating r_1 as the response)
4. Update predictions: $\hat{f}(\boldsymbol{x}) = \lambda \hat{f}_1(\boldsymbol{x}) + \lambda \hat{f}_2(\boldsymbol{x})$
5. Record new residuals $r_2(\boldsymbol{x}) = r_1(\boldsymbol{x}) - \lambda \hat{f}_2(\boldsymbol{x})$
6. Repeat steps 3-5 a total of B times, creating B trees

Boosting

Big idea with boosting is that we want to learn *a little bit* of information each iteration

- $\lambda > 0$ is a tuning parameter called the *learning rate*, **BUT,** works in conjunction with the size (depth) of trees chosen, as well as the *number* of trees B
  - For given tree depth, λ and B need tuned together – **CROSS-VALIDATION**
  - Unlike bagging and RF, the number of trees B is very important. B too large $\implies$ overfitting (finding patterns in residuals that aren't real)
- Tree depth can be chosen relative to how many interactions you want to include in the model
- Boosting can rival RFs in terms of predictive accuracy, but requires very careful tuning

Prediction vs. Inference

Prediction vs. Inference

- Given enough data, tree-based models – random forests and boosting in particular – can generate extremely accurate predictions
- While individual (shallow) trees are somewhat interpretable, there are not (well-established) inference methods for things like confidence intervals and hypothesis tests for tree-based methods
- Some *ad hoc* tools do exist – we'll look at a popular method for examining variable importance

OoB Variable Importance

Out-of-bag (OoB) Variable Importance:

- With bagging and random forests, each tree is built with a bootstrap sample of original data
- For each bootstrap sample, there is a very high probability that some observations in the original dataset will not be included – these are the **out-of-bag** (OoB) samples for that bootstrap sample
- **Big Idea:** record MSE on those OoB samples, then, to measure importance of some predictor X_i, permute values of X_i in those samples, recompute MSE and see how much predictions change /error increases
  - **Intuition:** If changing the value of X_i doesn't change predictions much, X_i shouldn't be that important

OoB Variable Importance

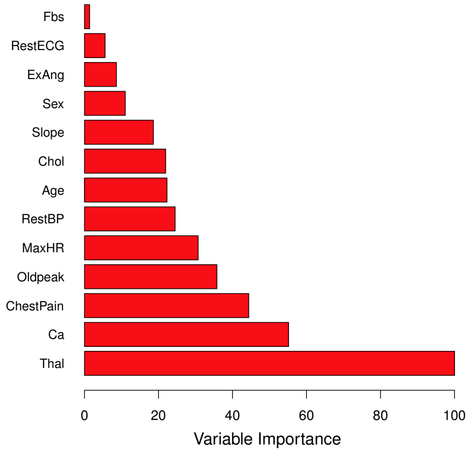
More formally, for each observation $z = (x_1, \ldots, x_p, y)$ in the original data, to test the importance of X_1

1. Find all trees for which z is an OoB sample
2. Use these trees to predict at z and record squared error
3. Permute X_1 in the OoB samples to get $z* = (x_1^*, x_2, \ldots, x_p, y)$
4. Re-predict and record new error

Repeat for each observation to obtain OoB Error (MSE) and permuted OoB Error (MSE*)

- Repeat for each variable $X_1, \ldots, X_p$; variables with the largest difference between MSE and MSE* are deemed most important

Motivating Examples



ISLR Fig. 8.9: OoB Variable importance example (computed using drop in Gini index instead of MSE)

OoB Variable Importance

This is a reasonably efficient method, but several major issues with it have since been identified:

- Ongoing problem with current literature – term “importance" isn’t all that well-defined or agreed upon
- Note that here we build model – predict – permute – re-predict; we don’t build a new model with the permuted data like we talked about with permutation tests
- This leads to very strange behavior when predictors are correlated – including both in the model makes each appear *more* important rather than less, as you’d expect (think about what happens in a linear model). More on this in the homework.

Why do RFs work?

Explaining RF Success

Random forests have enjoyed tremendous success in recent decades as a general purpose, “off-the-shelf" (`mtry` parameter often not tuned) prediction tool:

- Ecology
- Image recognition
- Bioinformatics
- A recent large-scale empirical study comparing 100’s of classifiers on 100’s of datasets (entire UCI repository) found:
    - RF ranked 1^{st} overall
    - Of the top 5 performing classifiers, 3 were some variant of RFs

Explaining RF Success

Despite all of this progress there has been shocking little work on actually explaining the underlying mechanisms at work in RFs that might explain their success.

From recent review paper by Scornet and Biau (2016):

- Research investigating the properties of random forest tuning parameters is “unfortunately rare"
- “present results are insufficient to explain in full generality the remarkable behavior of random forests."

Explaining RF Success

There have been numerous studies experimenting with various tuning parameter setups and strategies in recent years, though results are largely limited to high-level heuristics, e.g.

- Building more trees helps stabilize predictions
- Tuning parameters (e.g. tree depth, `mtry`) can have a significant impact on accuracy

Let's begin by looking at the original (and until recently, still most popular) argument for random forests given in the original paper by Breiman

Existing Explanations

Breiman (2001): The additional randomness in RFs serves to de-correlate trees, thereby reducing the variance of the ensemble

- Breiman showed that for classification, the generalization error for RFs is bounded above by a function of two components: the “strength" (accuracy) of the individual trees and the correlation between them
  - Randomness trades off accuracy at the tree level for reduced correlation (akin to bias-variance tradeoff)
- Updated discussion given in Hastie et al. (2009) (ESL book)
  - More randomness (smaller `mtry`) increases variance at tree level, but by much smaller magnitude than decrease in variance seen at the ensemble level

Existing Explanations

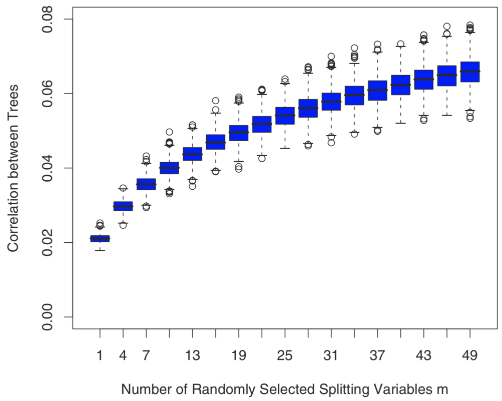


Figure: *Elements of Statistical Learning, Figure 15.9.* Between-tree correlation vs `mtry`.

Existing Explanations

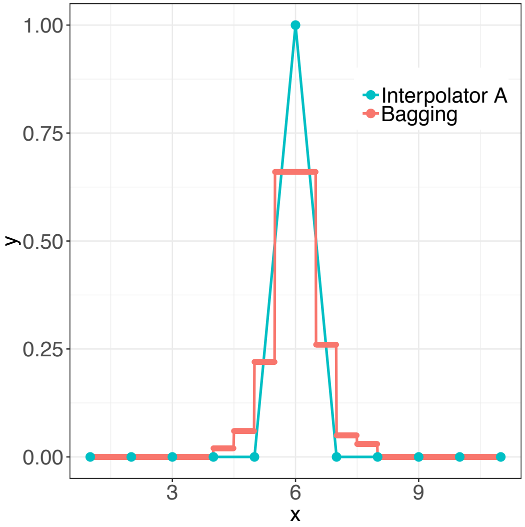
- Nothing “wrong" with this idea, but feels incomplete.
  - Lots of ways to reduce correlation, few if any of which lead to performance improvements as drastic as often seen with random forests
- Better seen as motivation for why RFs *might* work rather than an explanation for why they *do* work
  - Breiman himself doesn’t seem to believe this is the whole story in the original RF paper

Existing Explanations

Elsewhere in the literature, some have recently proposed that RFs work because of “interpolation" – they argue that RFs fit the training data in most cases (for classification) and that’s good because real world data isn’t noisy

What happens when we treat this as a regression problem and construct a random forest (in this case just bagging since we’ll use $p = 1$)?

Existing Explanations



Issues with Existing Explanations

When we treat this as a regression problem, the RF is not interpolating (even in this most extreme construction) and in fact looks to be doing something nearly opposite – trying to “smooth out" the influence of the outlying point (foreshadowing ...)

So what’s being overlooked here? All previous explanations – including from Breiman – have started by assuming RFs are much better than bagging.

Ingredients for a good explanation

A good explanation for RF success should ...

1. Be specific enough to determine (at least roughly) when (in what settings) RFs should be expected to perform well relative to other methods
   - Largely ignored in previous work, claiming that RFs simply “do" work well
2. Identify an intuitive role for the randomness (`mtry` parameter)
   - In most previous work, same arguments would apply to bagging
3. Either extend to other kinds of methods (base-learners) or provide intuition into why this is a tree-based phenomenon

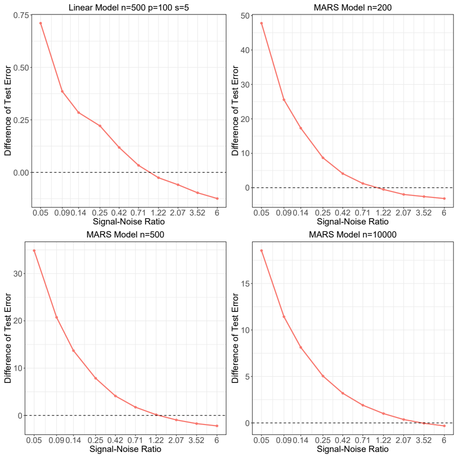
Relative Performance of RFs

So let's just see if RFs really are "just better" than bagging. Let's look at some simulations:

- Linear model with $n = 500$, $p = 100$, and $s = 5$ (s is the number of true/signal variables)
- Nonlinear (MARS) model, with $p = s = 5$ with $n = 200, 500$ or 10000
- Consider SNRs ranging from 0.05 to 6 (equally spaced on the log scale)
- Test error of random forests ($\texttt{mtry} = 0.33$) and bagging ($\texttt{mtry} = 1$) calculated on separate test set with sample size $n_t = 1000$ is calculated and differences averaged over $N = 500$ simulations.

Relative Performance of RFs

Error(Bagg) - Error(RF) vs SNR. Positive values indicate better performance by RFs:



Relative Performance of RFs

In each case we see a clear pattern: as the SNR goes up, the advantage offered by RFs dies out.

How about on real-world data? Here we take 10 datasets intended for regression from the UCI Machine Learning Repository to compare performance.

- Since we don't know the true SNRs, we inject additional random noise $\epsilon \sim N(0, \sigma^2)$ into the response
- σ^2 chosen as a proportion α of the sample variance of the response for $\alpha = 0, 0.01, 0.05, 0.1, 0.25, 0.5$

Relative Performance of RFs

Dataset	p	n
Abalone Age [abalone]	8	4177
Bike Sharing [bike]	11	731
Bioston Housing [boston]	13	506
Concrete Compressive Strength [concrete]	8	1030
CPU Performance [cpu]	7	209
Conventional and Social Movie [csm]	10	187
Facebook Metrics [fb]	7	499
Parkinsons Telemonitoring [parkinsons]	20	5875
Servo System [servo]	4	167
Solar Flare [solar]	10	1066

Table: Summary of UCI data utilized.

Relative Performance of RFs

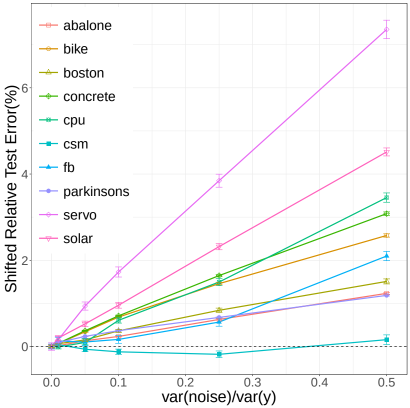


Figure: RTE vs amount of additional noise on real data. Results averaged over 500 repetitions.

Relative Performance of RFs

Note that in comparing RFs to bagging, we're really just comparing RFs with different values of `mtry`. Suppose we reverse the direction of the problem and estimate the optimal value of `mtry` at various SNRs ...

- Here again we consider the same MARS and linear model setups as above with the same sampling, covariate, and noise settings (here $p = 20$, $s = 10$ for linear model)
- For both models, we consider $n = 50$ and $n = 500$
- Optimal `mtry` determined on independent test sets of same size; results averaged over 500 repetitions at each SNR level for each setting

Relative Performance of RFs

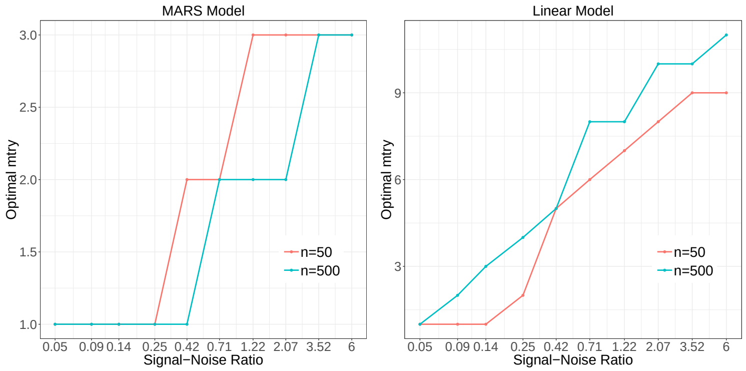


Figure: Optimal `mtry` vs SNR as measured by lowest average test error across 500 replicates.

Degrees of Freedom for RFs

So what's going on here to cause this?

Remember the Hastie, Tibshirani, and Tibshirani (2017) study comparing best subset selection (BSS), forward selection (FS), lasso, and relaxed lasso? There the authors observe similar patterns of relative performance and attribute this to differences in degrees of freedom (dof).

Let's investigate this possibility ...

Degrees of Freedom for RFs

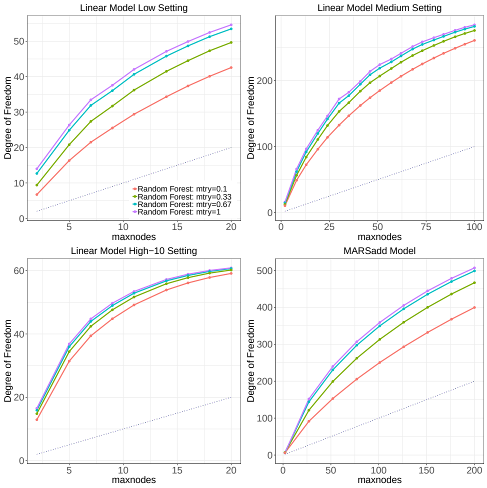


Figure: Estimated dof of RFs with different values of *mtry*.

Extensions to RandFS

Results thus far are really interesting and helpful, but perhaps not shocking: more randomness $\Longrightarrow$ lower variance / less overfitting $\Longrightarrow$ improved performance at low SNRs.

But ... there's nothing tree-specific about this. Trees are just sequentially partitioning the feature space into response-homogeneous regions ... can think of this as "building up" a model in the same fashion as forward selection.

So, if we were to create ensemble-ized versions of classical forward selection – analogues to the classic tree-based bagging and random forest procedures – would we see the same pattern? How would they compare to classical procedures (FS, lasso, relaxed lasso)?

Extensions to RandFS

Bagging and random forests analogues for forward selection:

- **Bagged Forward Selection (BaggFS)**
  - Draw B bootstrap samples
  - Perform forward selection on each to depth of d
  - Average across the B models
- **Randomized Forward Selection (RandFS)**
  - Draw B bootstrap samples
  - Perform forward selection on each to depth of d, but at each step, randomly choose $\texttt{mtry}\times p$ features as candidates for selection. Such candidates are selected uniformly at random without replacement.
  - Average across the B models

Extensions to RandFS

- We consider linear models (constructed as before) in the following settings from HTT (2017)

	n	p	s
Low setting	100	10	5
Medium setting	500	100	5
High-5 setting	50	1000	5
High-10 setting	100	1000	10

- SNR equally spaced from 0.05 to 6 on log scale
- Tuning done on external dataset
- For RandFS, we consider `mtry` = 0.33 (default RF), `mtry` = 1 (BaggFS), and tuned `mtry` on 10 values from 0.1, ..., 1

Extensions to RandFS

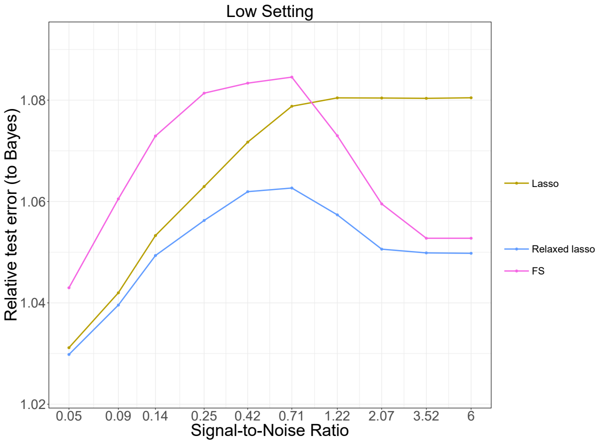


Figure: Performance Comparisons in low setting.

Extensions to RandFS

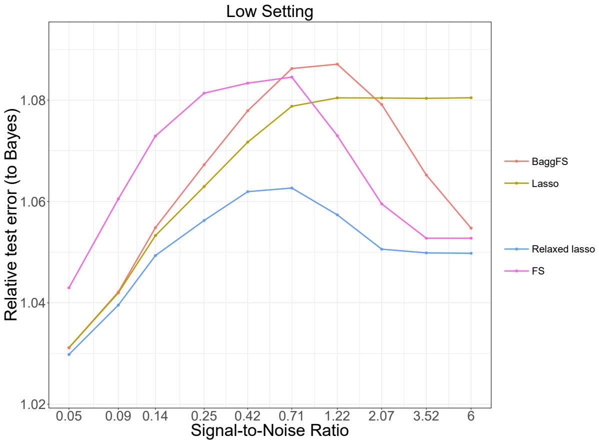


Figure: Performance Comparisons in low setting.

Extensions to RandFS

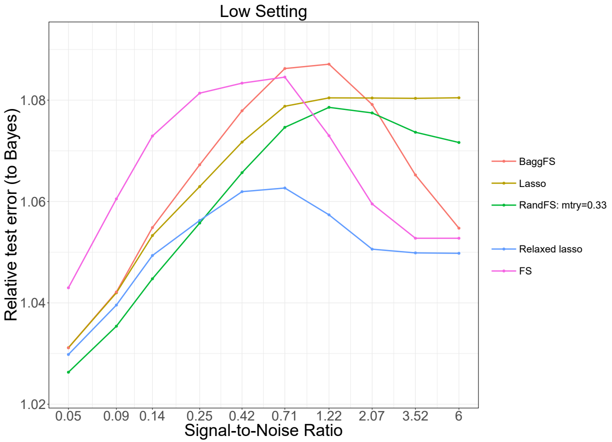


Figure: Performance Comparisons in low setting.

Extensions to RandFS

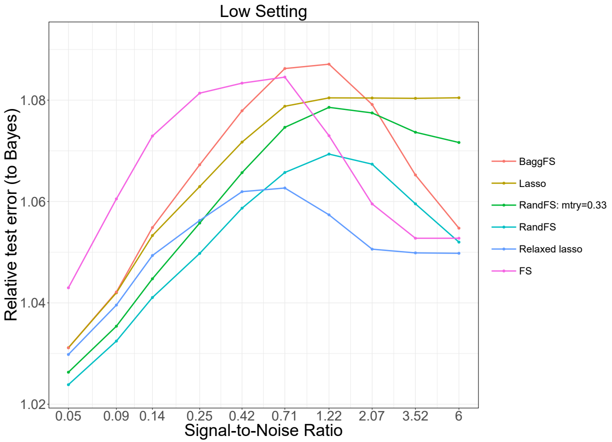


Figure: Performance Comparisons in low setting.

Extensions to RandFS

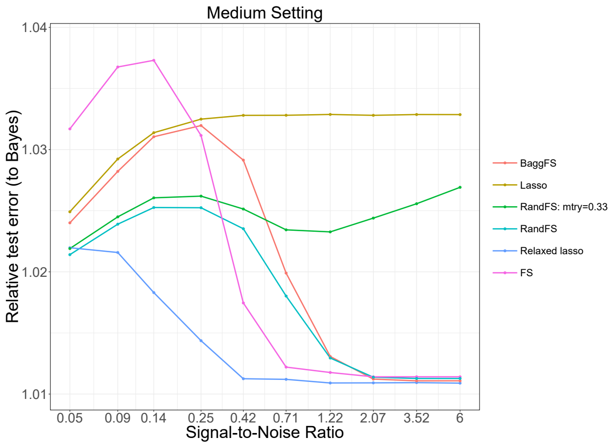


Figure: Performance Comparisons in medium setting.

Extensions to RandFS

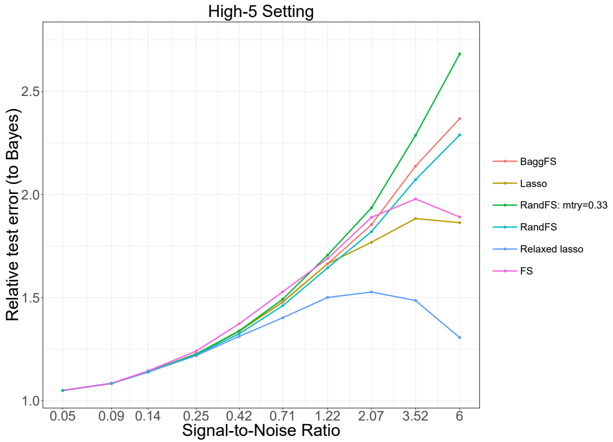


Figure: Performance Comparisons in high-5 setting.

Extensions to RandFS

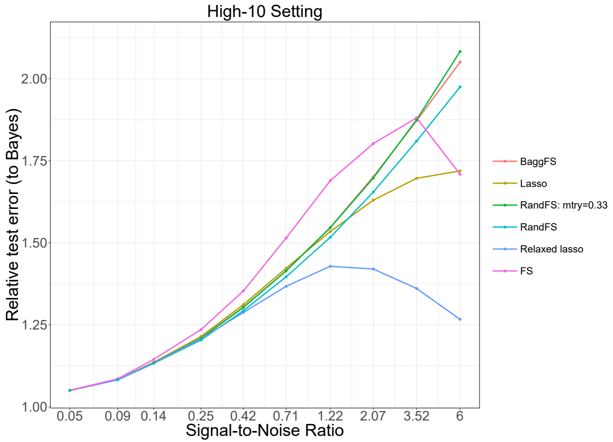
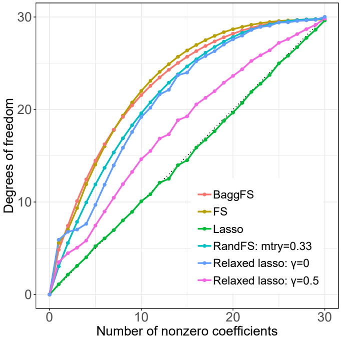
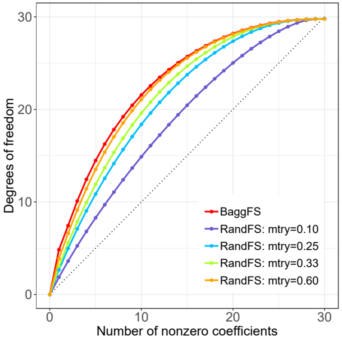


Figure: Performance Comparisons in high-10 setting.

Degrees of Freedom for RandFS

Dof estimates for RandFS follow the general pattern you would expect. Here we use a linear model with $n = 70, p = 30, s = 5$ and SNR = 0.7. Results are averaged over 500 repetitions.





Randomization as Regularization?

- The previous results show that RandFS can perform on par with explicit regularization procedures like lasso and relaxed lasso, at least in some settings
- RandFS seems to be doing a sort of *implicit* regularization:
  - For each of the B models, features are either selected or not
  - For each feature X_k, it's selection proportion α_k depends on the original data (and true relative importance), bootstrap samples, depth d to which models are grown, and `mtry`
  - Coefficient estimates are effectively shrunk by amount proportional to that selection proportion
- Same general idea likely going with random forests, though a bit more obscure

Conclusions

- Strong empirical evidence that relative improvement with RFs is a direct function of the SNR
- Reason to think that `mtry` serves much the same regularization role as, e.g. λ in lasso
- RFs not “just better" than bagging; `mtry` parameter really ought to be tuned (though we saw good performance even with `mtry` = 0.33 fixed)
- **Big question still remaining ...** Why do RFs *seem* to just always work better on real-world data?
  - ▶ If you believe any of this, the obvious answer would seem to be that real-world data really is pretty noisy

RFs and Regularization

But that's not all ...

- We argued that the randomization (`mtry` parameter) in RFs acted as a regularizer
- Effect of each feature X_k is effectively shrunk by an amount proportional to its selection proportion α_k
- **Key Point:** Column subsampling (`mtry`) is only one way to affect those selection proportions – if we hold `mtry` fixed and add more features, shouldn't we intensify the effect?
- If that's really what is "making RFs work", then bagging with extra features should improve predictions in a similar fashion to RFs

Augmented Bagging

Suppose we create an augmented dataset $\mathscr{D}_n^* = \{\boldsymbol{Z}_1^*, \ldots, \boldsymbol{Z}_n^*\}$ where each $\boldsymbol{Z}_i^* = (\boldsymbol{X}_i, \boldsymbol{N}_i, Y_i)$ and $\boldsymbol{N}_i = (N_{1,i}, \ldots, N_{q,i})$ is an additional set of noise features. The original bagging procedure is then performed on $\mathscr{D}_n^*$ so that these predictions from *Augmented Bagging* (AugBagg) take the form

$$\hat{y}_{\text{AugBagg}} = \frac{1}{B} \sum_{b=1}^{B} T((\boldsymbol{x}, \boldsymbol{n}); \Theta_b, \mathscr{D}_n^*). \tag{1}$$

where $\boldsymbol{n}$ can be filled in with random draws from the additional noise features.

- We assume only that $\boldsymbol{N}$ is conditionally independent of Y given $\boldsymbol{X}$.

Simulations on AugBagg

Consider the same general linear model setup as before:

- Set $n = 100$, $p = s = 5$.
- q additional i.i.d. noise features sampled from $\mathcal{N}(0, 1)$ independent of $\boldsymbol{X}$ are then added with q ranging from 1 to 250.
- The noise term ϵ was sampled from $\mathcal{N}(0, \sigma_\epsilon^2)$ with σ_ϵ^2 chosen to satisfy a particular SNR
- SNR = 0.01, 0.05, 0.09, or 0.14.
- Test error is calculated on an independent, randomly generated test set containing 1000 observations and averaged over 500 repetitions.

Simulations on AugBagg

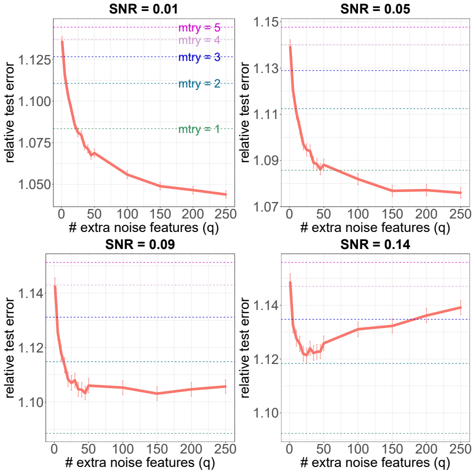


Figure: Performance of Augmented Bagging

Conclusions

- This is a somewhat scary finding ... we're explicitly suggesting that adding “bad" features into a regression model can help improve predictive accuracy ... ON A TEST SET
- Big implications for nonparametric kinds of hypothesis tests for variable importance that work by measuring model improvement when certain features are included
- Research along these lines just now emerging, but sure to be an interesting topic in the years to come