TEXAS McCombs

MASTER OF SCIENCE IN BUSINESS ANALYTICS

Fitting tree-based models

Outline

Fitting trees and the bias-variance tradeoff

Bagging and random forests

Boosting

Variable importance

Bringing it all together: The California housing data

Fitting trees: The bias-variance tradeoff again

How do we fit trees to data??

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

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

To fit a tree, we try to minimize:

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

where,

- $L(T, y)$ is our loss in fitting data y with tree T.
- $|T|$ is the number of bottom nodes in tree T.

For numeric y our loss is usually sum of squared errors, for categorical y we can use the deviance or the miss-classification rate.

Note: α is analogous to the lasso λ !!!

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 rule to find the one that gives the biggest decrease in loss (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.

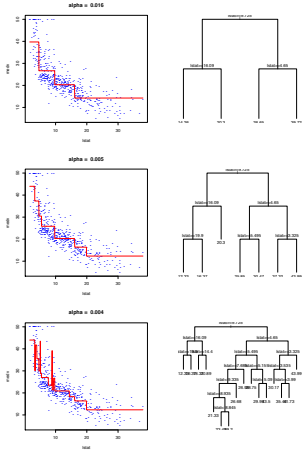
Boston data, lstat and medv:

On the 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.

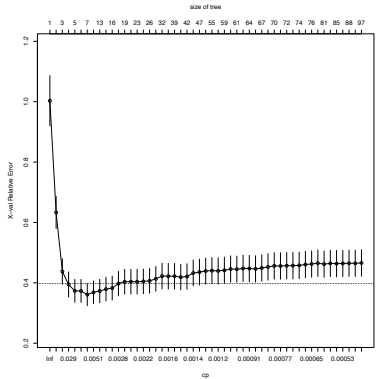
The top tree is a sub-tree of the middle tree, and the middle tree is a sub-tree of the bottom tree.

The middle α is the one suggested by CV.

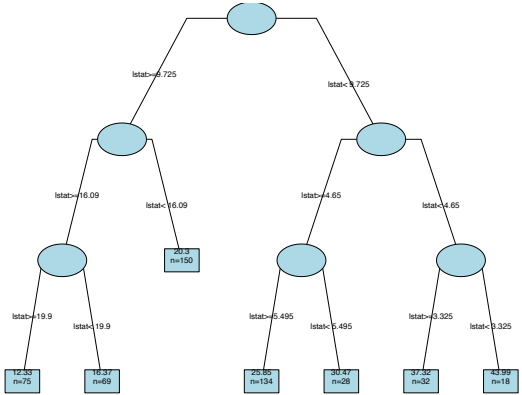


This is the CV plot giving by the R package rpart for y=medv x=lstat. Tree sizes at top of plot, and (a transformation of) α (the "cost-complexity" parameter) on the bottom.

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



Here is the best CV tree as plotted by rpart.



Bagging and random forests

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*.

To **B***ootsrap* **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.

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 data scientists 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**.

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.

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.

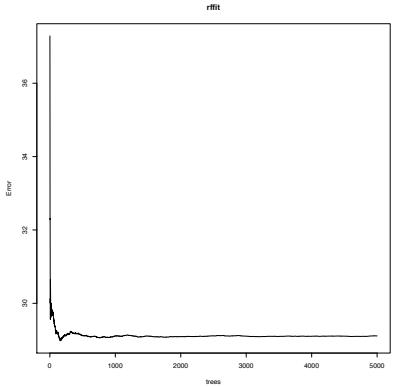
One can show that, on average, each bagged tree makes use of about 2/3 of the observations.

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.

This suggests you just need a couple of hundred trees.

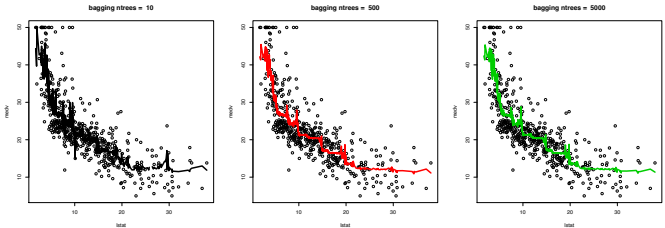


Bagging for Boston: y=medv, x=lstat.

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 multiple trees (average over) so we no longer have a simple step function!!



Boosting trees

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

The idea however, is totally different!

In Boosting we:

- Fit the data with a single tree.
- Crush the fit so that it does not work very well.
- Look at the part of y not captured by the crushed tree and fit a new tree to what is "left over".
- Crush the new tree. Your new fit is the sum of the two trees.
- Repeat the above steps iteratively. At each iteration you fit "what is left over" with a tree, crush the tree, and then add the new crushed tree into the fit.
- Your final fit is the sum of many trees.

This one is actually made clearer by the mathematical notation.
This is Algorithm 8.2 in the book (*ISL*).

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 make each new tree a *weak leaner* in that is only does a little more fitting.

Have to choose:

- B, number of interations (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.

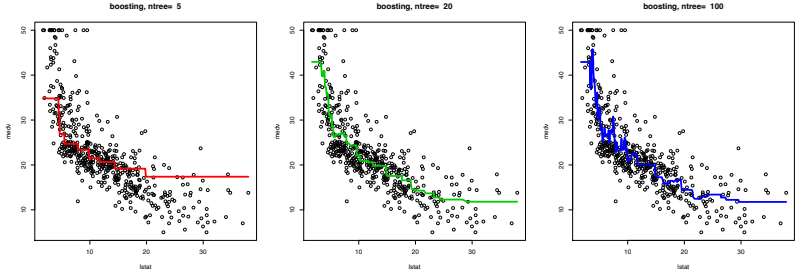
Also you can't just add up the fit.

But, it is the same idea:

- fit.
- crush fit.
- fit what is left over.
- aggregate crushed fits.

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.

Variable importance measures

The ensemble methods Random Forests and Boosting can give dramatically better fits than simple trees. Out-of-sample, they can work amazingly well. They are a breakthrough in statistical science.

However, they are certainly not interpretable!!

You cannot look at hundreds or thousands of trees.

Nonetheless, by computing summary measures, you can get some sense of how the trees work.

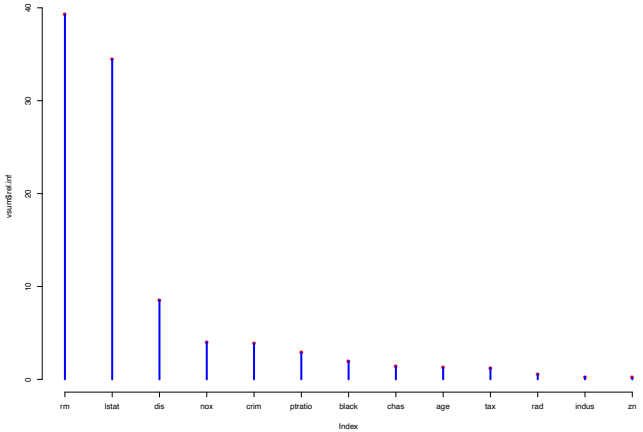
In particular, we are often interested in which variables in x are really the “important” ones.

What we do is look at the splits (decision rules) in a tree and pick out the ones that use a particular variable. Then we can add up the reduction in loss (eg residual sum of squares) due to the splits using the variable.

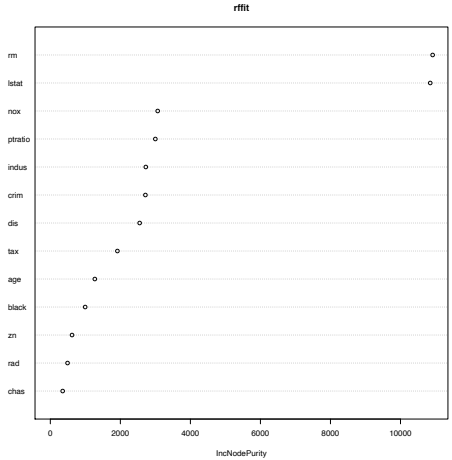
For a single tree we are done.

For bagging we can average the effect of a variable over the B trees and for Boosting we can sum the effects.

Here is the variable importance for the Boston data with all the variables obtained from a Boosting fit.



Here is the variable importance for the Boston data with all the variables obtained from a Random Forests fit.



Trees, random forests, boosting: The California data

Let's try all of these methods on the California Housing data.

That is, we'll try trees, Random Forests, and Boosting.

How will they do?

We'll do a simple three set approach since we have a fairly large data set.

We randomly divide the data into three sets:

Train: 10,320 observations.

Validation: 5,160 observations.

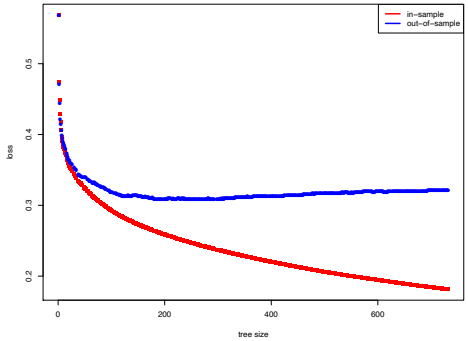
Test: 5,160 observations.

We,

- Try various approaches using the training data to fit and see how well we do out-of-sample on the validation data set.
- After we pick an approach we like, we fit using the combined train+validation and then predict on the test to get a final out-of-sample measure of performance.

Trees:

- Fit big tree on train.
- For many cp=α, prune tree, giving trees of various sizes.
- Get in-sample loss on train.
- Get out-of-sample loss on validation.



The loss is RMSE.

We get the smallest out-of-sample loss (.307) at a tree size of 194.

Boosting:

Let's try:

- maximum depths of 4 or 10.
- 1,000 or 5,000 trees.
- λ = .2 or .001.

`olb`:
out-of-sample loss

`ilb`:
in-sample loss.

min loss of .231 is quite a bit better than trees!

	tdepth	ntree	lam	olb	ilb
1	4	1000	0.001	0.414	0.416
2	10	1000	0.001	0.378	0.380
3	4	5000	0.001	0.279	0.282
4	10	5000	0.001	0.252	0.250
5	4	1000	0.200	0.232	0.164
6	10	1000	0.200	0.233	0.098
7	4	5000	0.200	0.231	0.081
8	10	5000	0.200	0.233	0.014

Random Forests:

Let's try:

- m equal 3 and 9 (Bagging).
- 100 or 500 trees.

`olrf` is the out-of-sample loss and `ilrf` is the in-sample loss.

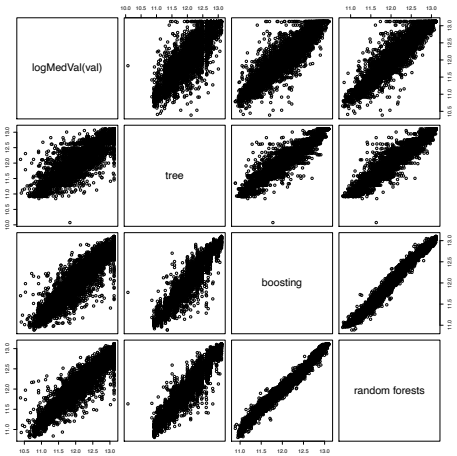
	mtry	ntree	olrf	ilrf
1	9	100	0.241	0.255
2	3	100	0.236	0.250
3	9	500	0.241	0.253
4	3	500	0.233	0.245

Minimum loss is comparable to boosting.

Let's compare the predictions on the Validation data with the best performing of each of the three methods.

It does look like Boosting and Random Forests are a lot better than a single tree.

The fits from Boosting and Random Forests are not too different (this is not always the case).

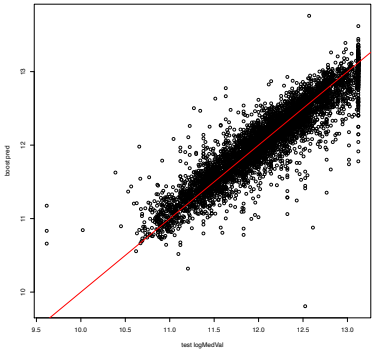


Test Set Performance, Boosting

Let's fit Boosting using depth=4, 5,000 trees, and shrinkage λ=.2 on the combined train and validation data sets.

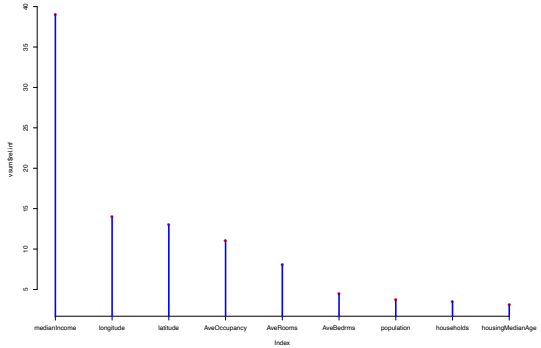
The RMSE on test data is .231.

This is consistent with what we had before from the train-validation data.



Boosting gives us a measure of variable importance:

	var	rel.inf
1	medianIncome	39.065051
2	longitude	13.965980
3	latitude	12.985643
4	AveOccupancy	11.055079
5	AveRooms	8.093967
6	AveBedrms	4.480044
7	population	3.708594
8	households	3.520058
9	housingMedianAge	3.125583

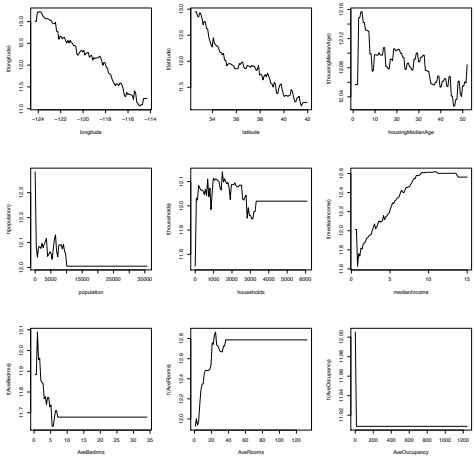


`medianIncome` is by far the most important variable.
After that, it is location - *makes sense*.

The boosting package also generated plot which are supposed to show the plot of x_i vs. y for each individual x_i by averageing out the other x's.

This is supposed to be a plot of x_i vs. y=logMedVal for each $i = 1, 2, \ldots, 9$.

It is not clear this works, or should work, when there are interactions!!

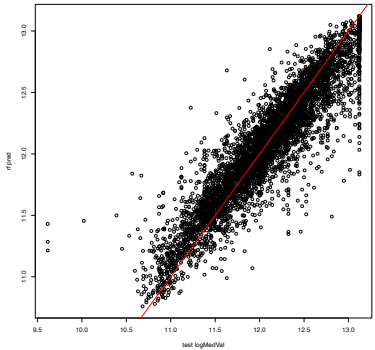


Test Set Performance, Random Forests

Let's fit Random Forests using m=3 and 500 trees on the combined train and validation data sets. Let's see how the predictions compare to the test values.

Not too bad!!

The RMSE is .23,
so our train-validation results hold up.



Random Forests: Variable importance

Random Forests give a measure of variable importance. It just adds up how much the loss decreases every time a variable is used in a split.

Not suprisingly, medianIncome is by far the most important variable.