Trees

Rob McCulloch

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
8. Variable Importance Measures
9. Trees, Random Forests, Boosting: The California Data
10. Classification Loss for Trees
11. More on Boosting, Gradient Boosting and XGBoost

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

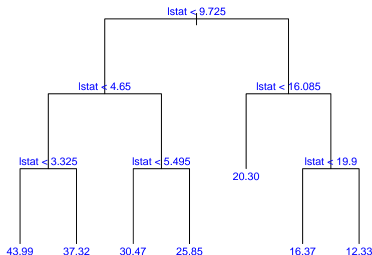
Let's look at a simple 1-dimensional example so that we can see what is going on.

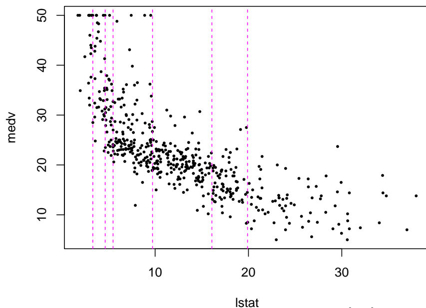
We'll use the Boston housing data and relate x=lstat to y=medval.

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.

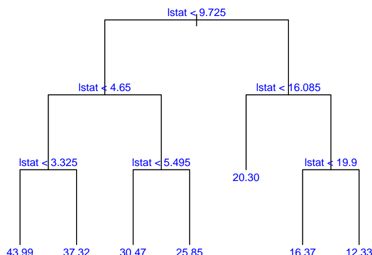
Each observation is sent down the tree until it hits a bottom node or *leaf* of the tree.

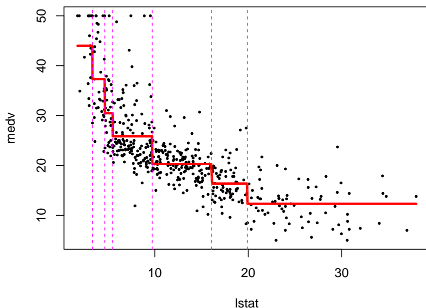




The set of bottom nodes gives us a partition of the predictor (x) space into disjoint regions. At right, the vertical lines display the partition. With just one x, this is just a set of intervals.

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).





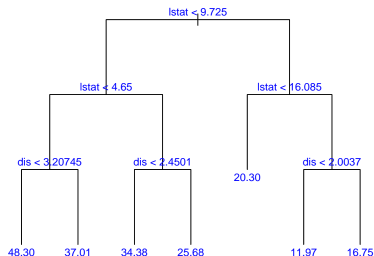
To predict, we just use our step function estimate of $f(x)$.

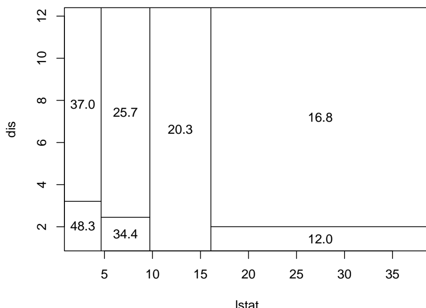
Equivalently, we drop x down the tree until it lands in a leaf and then predict the average of the y values for the training observations in the same leaf.

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.



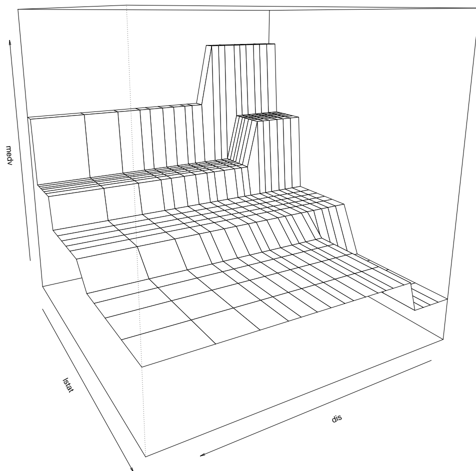


At right is the *partition* of the x space corresponding to the set of bottom nodes (leaves).

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.

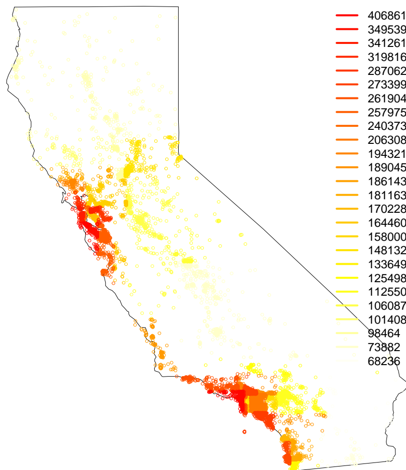
The effect of `dis` depends on `lstat`!!

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.

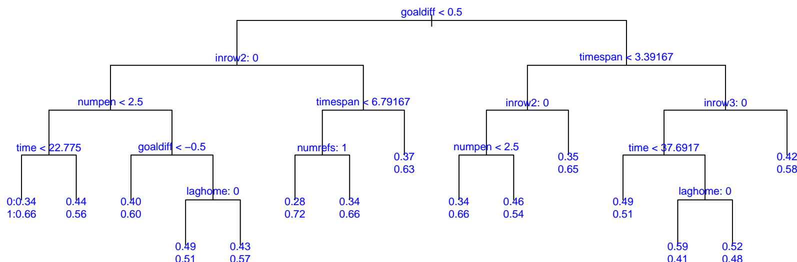
Table 5: Variable Descriptions

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.

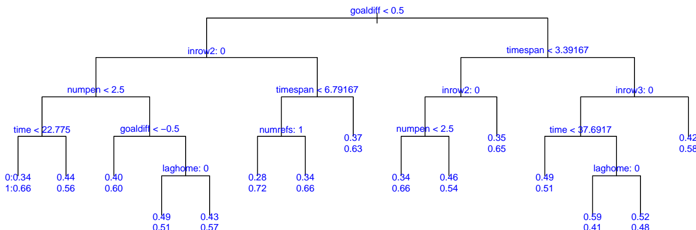
There is a lot of fit!!!

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.

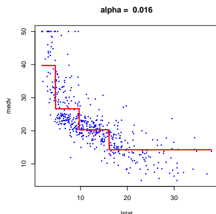
Boston Data, lstat and medv:

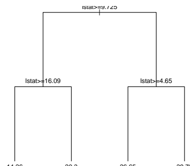
At right are three different tree fits we get from three different α values (using all the data).

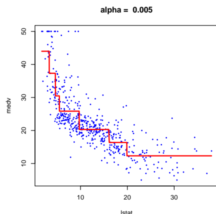
The smaller α is, the lower the penalty for complexity is, the bigger tree you get.

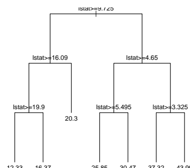
The top tree is a sub-tree of the middle tree, and the middle tree is a sub-tree of the bottom tree.

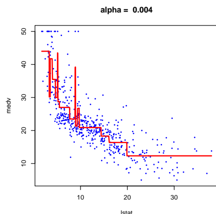
The middle α is the one suggested by CV.

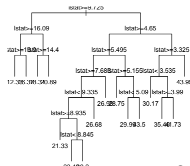








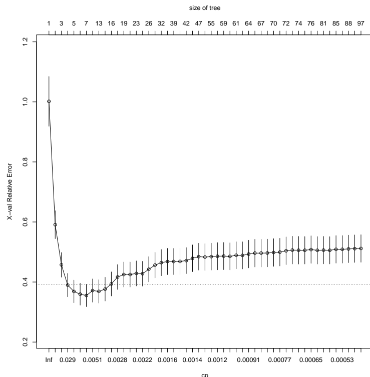




This is the CV plot giving by the R package rpart for y=medv x=lstat.

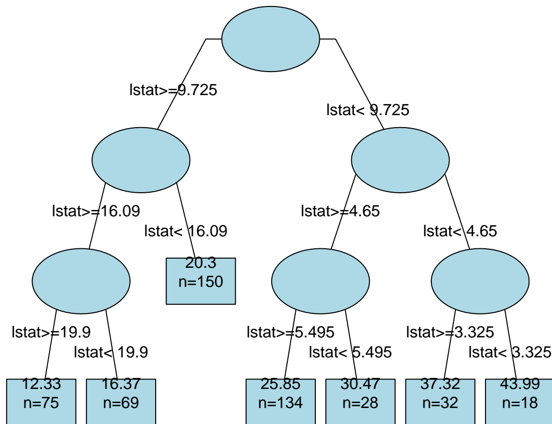
Tree sizes at top of plot, and (a transformation of) α (the "cost-complexity" parameter) on the bottom.

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.

Note:

The parameters that determine how big the big tree is could also matter.
Usually the default seems to work ok on this.

For example, rpart in R has the rpart.control data structure.
For example, you can specify a minimum number of observations you can have in a bottom node
or the minimun needed to split (or some other stuff).

```
minbucket=100
cat("minbucket: ",minbucket,"\n")

cntrl = rpart.control(minsplit = 2*minbucket,
                                    minbucket=minbucket)

## get big tree
temptree = rpart(R~.,data=ddf,control=cntrl)
szbt = length(unique(temptree$where))
cat("size of big tree: ",szbt,"\n")
```

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.

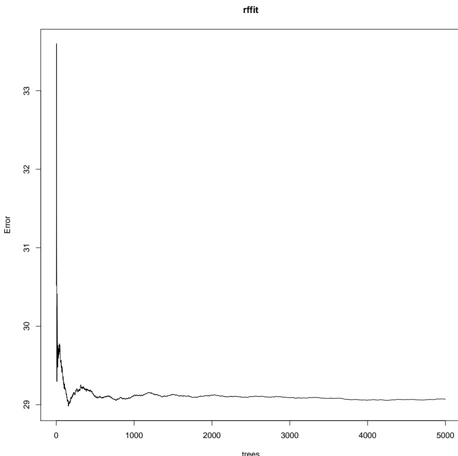
One can show that, on average, each bagged tree makes use of about $2/3$ of the observations.

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.

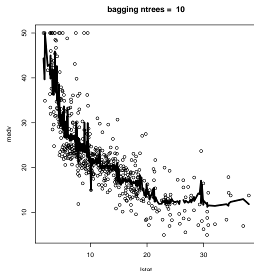


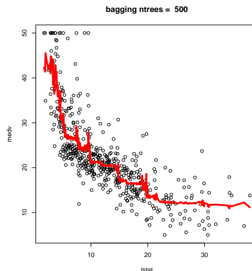
Bagging for Boston: y=medv, x=lstat.

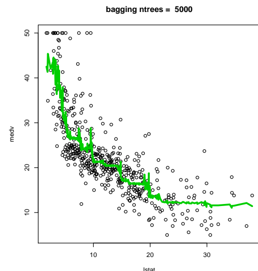
With 10 trees our fit is too jumbly.

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.

Gradient boosting fits the derivative of the loss.
For squared error loss, this is the residuals.

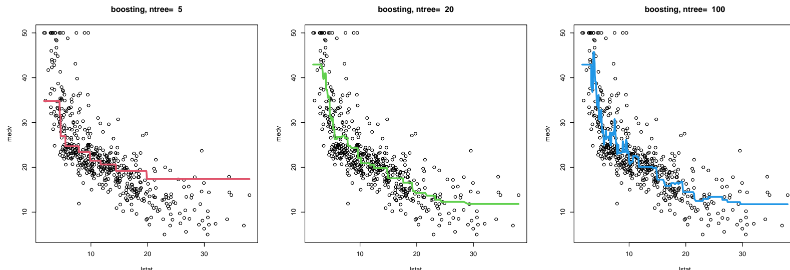
It is always the same basic idea:

Sequentially add in small fits of fit ("weak 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.

The Curse of Dimensionality

The curse of dimensionality is that as the dimension of our problem (e.g number of features) goes up, "our data gets sparse".

Suppose X_i are iid uniform in (0,1), $i = 1, 2, \ldots n$.

On average, how many observations are in an interval of length h?

$h \times n$.

Suppose X_{ij} are all uniform, iid, $i = 1, 2, \ldots n$, $j = 1, 2, \ldots, p$.

On average, how many observation are such that X_{ij} are in an interval of length h?

That is, how may X_i in R^p are in a cube with each dimension of length h?

$h^p \times n$.

Boosting defeats the curse of dimensionality by just looking at small trees!!!

But, the trees have to be big enough to capture the level of interaction in the features.

8. 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.

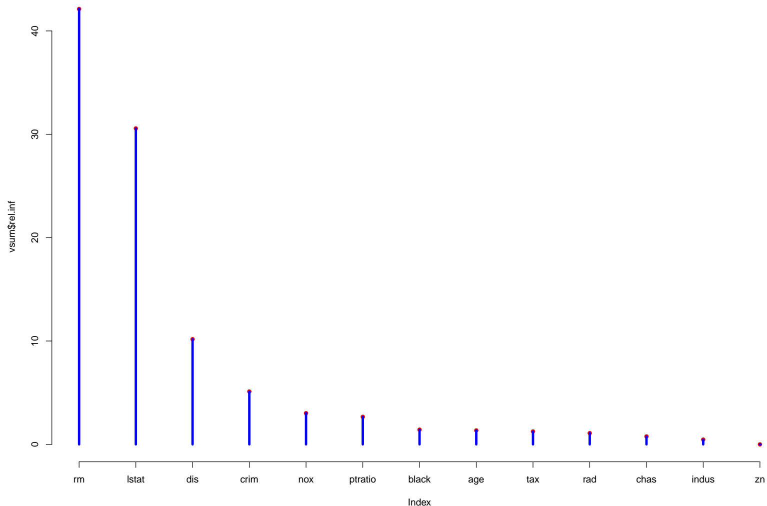
In particular, we are often interested in which variables in x are really the "important" ones.

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.

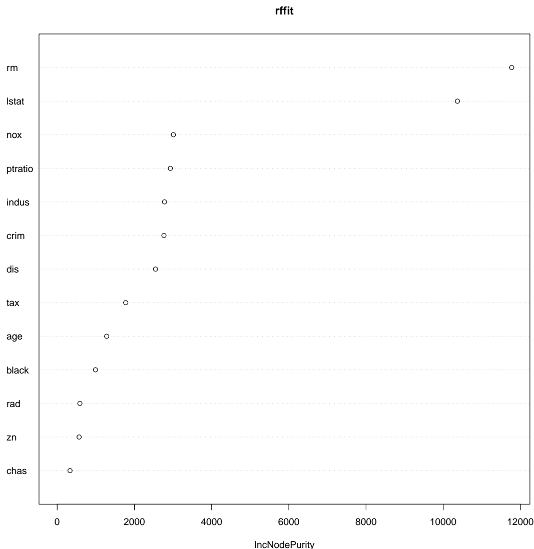
For a single tree we are done.

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

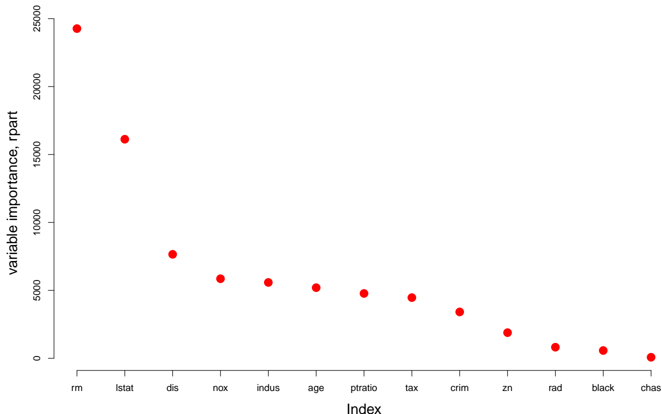
Here is the variable importance for the Boston data with all the variables obtained from a Boosting fit.



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



Here is the variable importance for the Boston data with all the variables obtained from a single tree fit (using rpart).



Variable importance from XGBoost, diabetes data.

```
xgbmod = XGBRegressor(booster='gbtree',objective='reg:squarederror',
   max_depth=2, learning_rate=0.1, n_estimators=100, random_state=2, n_jobs=-1)
xgbmod.fit(X,y)
fimp = pd.DataFrame({'nms':xvnms,'imp':xgbmod.feature_importances_})
print(fimp)
```

```
   nms       imp
0  age  0.031797
1  sex  0.040149
2  bmi  0.243854
3  map  0.084954
4   tc  0.038039
5  ldl  0.028799
6  hdl  0.054243
7  tch  0.088452
8  ltg  0.335191
9  glu  0.054523
```

9. Trees, Random Forests, Boosting: The California Data

Let's try all this stuff 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.

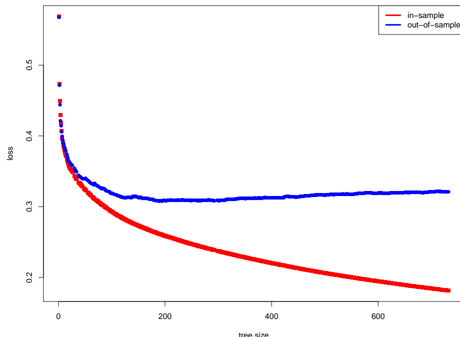
Test: 5,160 observations.

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.
- $\lambda = .2$ 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.

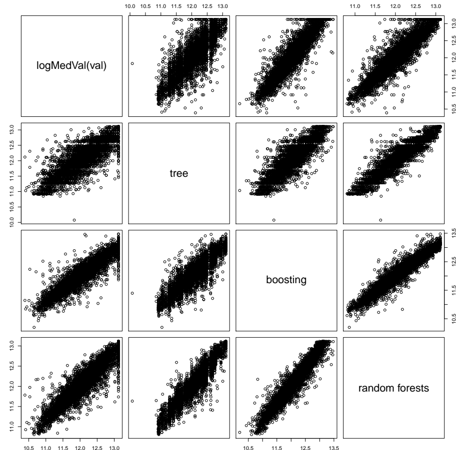
	mtry	ntree	olrf	ilrf
1	9	100	0.241	0.255
2	3	100	0.236	0.250
3	9	500	0.241	0.253
4	3	500	0.233	0.245

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).

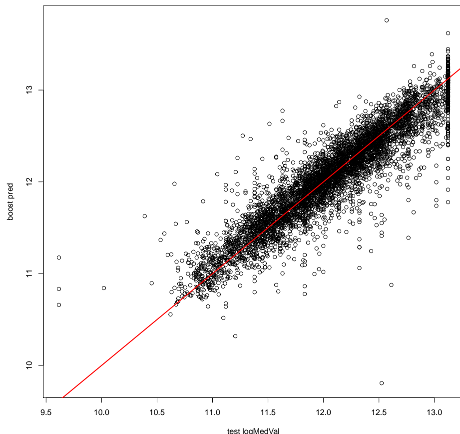


Test Set Performance, Boosting

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

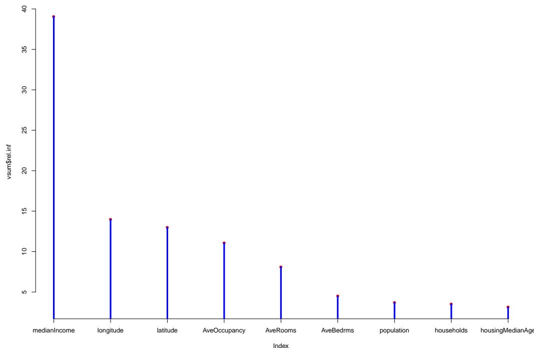
The RMSE on test data is .231.

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 13.963321
3          latitude 12.988301
4      AveOccupancy 11.055079
5          AveRooms  8.093967
6         AveBedrms  4.480044
7        population  3.708594
8        households  3.520058
9 housingMedianAge  3.125583
```

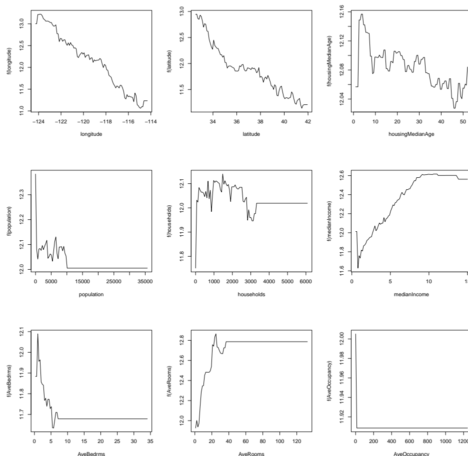


`medianIncome` is by far the most important variable.
After that, it is location - *makes sense*.

The boosting package also generated plot which are supposed to show the plot of x_i vs. y for each individual x_i by averaging 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!!



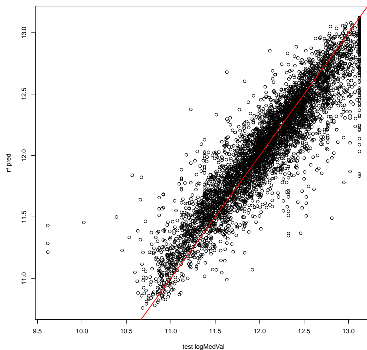
Test Set Performance, Random Forests

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!!

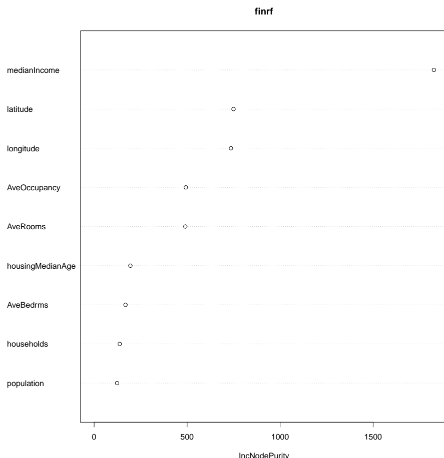
The RMSE is .23, so our train-validation results hold up.



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.



In R:

```
#--------------------------------------------------
#load libraries
library(randomForest)
library(gbm) #boosting

#--------------------------------------------------
#read in California Housing Data
ca = read.csv("calhouse.csv")

#--------------------------------------------------
#train, val , test
set.seed(14) #Dave Keon was captain of the Toronto Maple Leafs!!!
n=nrow(ca)
n1=floor(n/2)
n2=floor(n/4)
n3=n-n1-n2
ii = sample(1:n,n)
catrain=ca[ii[1:n1],]
caval = ca[ii[n1+1:n2],]
catest = ca[ii[n1+n2+1:n3],]
```

```
#-----------------------------------------------------
#fit using random forests (on train, predict on val)
#mtry is the number of variables to try
rffit = randomForest(logMedVal~.,data=catrain,mtry=3,ntree=500)
rfvalpred = predict(rffit,newdata=caval)

#-----------------------------------------------------
#fit using boosting
boostfit = gbm(logMedVal~.,data=catrain,distribution="gaussian",
              interaction.depth=4,n.trees=5000,shrinkage=.2)
boostvalpred=predict(boostfit,newdata=caval,n.trees=5000)

#-----------------------------------------------------
#plot (out-of-sample) fits
pairs(cbind(caval$logMedVal,rfvalpred,boostvalpred))
print(cor(cbind(caval$logMedVal,rfvalpred,boostvalpred)))
```

Let's combine the train and validation data set and refit using boosting.
Then we'll get our out-of-sample rmse from the test data.

```
#--------------------------------------------------
catrainval = rbind(catrain,caval) #stacks the two data frames
#--------------------------------------------------
#refit boosting
boostfit2 = gbm(logMedVal~.,data=catrainval,distribution="gaussian",
              interaction.depth=4,n.trees=5000,shrinkage=.2)
boosttestpred=predict(boostfit2,newdata=catest,n.trees=5000)
#--------------------------------------------------
#plot test y vs test predictions
plot(catest$logMedVal,boosttestpred)
abline(0,1,col="red",lwd=2)
#--------------------------------------------------
rmse = sqrt(mean((catest$logMedVal-boosttestpred)^2))
cat("rmse on test for boosting: ",rmse,"\n")
#--------------------------------------------------
#variable importance from boosting
summary(boostfit2)
```

```
#--------------------------------------------------
#refit random forests on train-val
rffit2 = randomForest(logMedVal~.,data=catrainval,mtry=3,ntree=500)
rftestpred = predict(rffit2,newdata=catest)
#--------------------------------------------------
rmse = sqrt(mean((catest$logMedVal-rftestpred)^2))
cat("rmse on test for random forests: ",rmse,"\n")
#--------------------------------------------------
#variable importance from Random Forests
varImpPlot(rffit2)
```

10. Classification Loss for Trees

To fit trees we need to pick our loss.

As usual, for numeric y, the usual loss is mean squared error, or, equivalently, RMSE.

For classification it is a little more tricky choosing the loss.

The default loss is deviance, but there are a couple other loss measures used in the tree literature.

Recall, deviance loss

For data (x_i, y_i) (train) or (test)

Total loss is

$$\sum L(y_i, x_i) = \sum -2 \log(P(Y = y_i \mid x_i))$$

Example:

A tiny data set with 6 observations and model fits from model 1 ($P1(Y = y \mid x)$) and model 2 ($P2(Y = y \mid x)$).

```
     x y P1(Y=1|x) P1(Y=0|x)     dev1
[1,] 1 0       0.1       0.9 0.210721
[2,] 2 0       0.1       0.9 0.210721
[3,] 3 0       0.1       0.9 0.210721
[4,] 4 1       0.9       0.1 0.210721
[5,] 5 0       0.9       0.1 4.605170
[6,] 6 1       0.9       0.1 0.210721

     x y P2(Y=1|x) P2(Y=0|x)     dev2
[1,] 1 0       0.5       0.5 1.386294
[2,] 2 0       0.5       0.5 1.386294
[3,] 3 0       0.5       0.5 1.386294
[4,] 4 1       0.5       0.5 1.386294
[5,] 5 0       0.5       0.5 1.386294
[6,] 6 1       0.5       0.5 1.386294
```

Note: -2*log(.5) = 1.386294, -2*log(.1) = 4.60517, -2*log(.9) = 0.210721

Deviance under Model 2: 6*1.386294 = 8.317764

Deviance under Model 1: 5*0.210721 + 4.605170 = 5.658775

What happens if we fit a tree to this data set in R?

```
xydf = data.frame(x=1:6,y=as.factor(c(0,0,0,1,0,1)))
temp = tree(y~x,xydf,control=tree.control(6,mincut=3,minsize=6))
print(temp)
node), split, n, deviance, yval, (yprob)
      * denotes terminal node

1) root 6 7.638 0 ( 0.6667 0.3333 )
  2) x < 3.5 3 0.000 0 ( 1.0000 0.0000 ) *
  3) x > 3.5 3 3.819 1 ( 0.3333 0.6667 ) *
```

The deviance from the left child is
$3*(-2*\log(1)) = 0$.

The deviance from the right child is
$1*(-2*\log(1/3)) + 2*(-2*\log(2/3)) = 3.819085$

The left child is "pure" so there is no loss.

If we print the R summary of the tree we get:

```
> print(summary(temp))

Classification tree:
tree(formula = y ~ x, data = xydf, control = tree.control(6,
    mincut = 3, minsize = 6))
Number of terminal nodes:  2
Residual mean deviance:  0.9548 = 3.819 / 4
Misclassification error rate: 0.1667 = 1 / 6
```

We get the (in sample) missclassification rate and deviance as summaries.

The "average deviance" is obtained by dividing by (n − number of bottom nodes) for reasons we skip.

Notes:

While the deviance is not terribly interpretable, it gets used a fair amount in statistics.
We have seen that it is related to the Likelihood.

For binary classification problems another obvious loss is
$|y - P(Y = 1 \mid x)|$ where y is $0/1$.

Node Purity:

When using decision trees for classification, loss measures are often expressed in terms of the concept of *node purity*.

Consider the deviance for a single bottom node.
Let $p_k = (\%y = k)$ out of the observations in the node.
Then

$$
\begin{aligned}
\text{deviance} &= \\
&= -2 \sum_{\text{y in node}} log(p_y) \\
&= -2 \sum_k n_k \, log(p_k) \\
&= -2n \sum_k p_k \, log(p_k) \\
&= 2n \, H(p)
\end{aligned}
$$

$H(p) = -\sum p_k log(p_k)$ is the *information* or *Shannon entopy* of the discrete distribution given by $p = (p_1, p_2, \ldots, p_k)$.

The entropy is a very famous measure of how the level of uncertainty associated with a distribution. For example $I(p)$ is maximized at $p_k = 1/k$ and minimized when one of the probabilities is 1 (0log(0)=0).

High entropy means the outcome is unpredictable, and low entropy means the outcome is more predictable.

We say that a node is "purer" when the outcome is more predictable.

Measures of node purity that are also used:

missclassification: $1 - max_k(p_k)$.
Gini index: $\sum_{k \neq k'} p_k \, p_{k'}$.
Entropy: $-\sum p_k log(p_k)$.

There are various ways to motivate the Gini. For example, it is related to the probability of getting two different outcomes from two draws.

These are all measures of “node impurity” so we would want any of these to be small.

Note that when fitting trees it is often recommended to train with gini or entropy even if your eventual out-of-sample criterion will be miss-classification.

The `tree` package in `R` gives the choice of “deviance” or “gini” with default of deviance.

11. More on Boosting, Gradient Boosting and XGBoost

There a variety of extentions and modifications to the basic boosting algorithm.
In this section we mention of few of them.

This will give us a few more options in fitting boosting beyond the basic d, B, and λ.

```
gbm(
      weights,
      var.monotone = NULL,
      n.trees = 100,
      interaction.depth = 1,
      n.minobsinnode = 10,
      shrinkage = 0.1,
      bag.fraction = 0.5,
      train.fraction = 1,
      cv.folds = 0,
      keep.data = TRUE,
      verbose = FALSE,
      class.stratify.cv = NULL,
      n.cores = NULL
    )
```

Randomization

Rather than using all the data to fit the new tree at each boosting iteration, we can randomly pick a subset of the observations for training.

This can capture some of the effects of bagging, and speed up computation.

e.g in `R`: `gbm::gbm` we have:

```
bag.fraction: the fraction of the training set observations randomly
          selected to propose the next tree in the expansion. This
          introduces randomnesses into the model fit. If 'bag.fraction'
          < 1 then running the same model twice will result in similar
          but different fits. 'gbm' uses the R random number generator
          so 'set.seed' can ensure that the model can be reconstructed.
          Preferably, the user can save the returned 'gbm.object' using
          'save'. Default is 0.5.
```

Gradient Boosting

How can we do boosting for binary y??

How do we generalize of boosting modeling approach??

Regression reviewed

Let's review linear regression in way we can generalize it.

We can think of our model as

$$Y_i \sim N(\mu_i, \sigma^2), \;\; \mu_i = x_i' \beta.$$

We have an associated (- log likelihood)

$$\frac{1}{2\sigma^2} (y_i - \mu_i)^2.$$

Since our focus is on the mean, we can drop σ and let

$$L(y_i, \mu_i) = \frac{1}{2} (y_i - \mu_i)^2.$$

Logit reviewed

We can think of our model as:

$$Y_i \sim \text{Bernoulli}(p_i), \;\; \log(\frac{p_i}{1-p_i}) = \theta_i = x_i' \beta,$$

with an associated loss:

$$L(y_i, \theta_i) = -\log(P(y_i|\theta_i))$$

General setup

We can write a generalized modeling approach as:

$$Y_i \sim p(Y_i|\alpha_i), \;\; \alpha_i = f(x_i)$$

with loss,

$$L(y_i, \alpha_i).$$

To boost with trees, we will have

$$\alpha_i \approx \hat{f}(x_i) = \sum_{b=1}^{B} \lambda \, \hat{f}^b(x_i).$$

where each $\hat{f}^b$ is a simple regression tree with d splits.

General Boosting Algorithm with Trees

(i) Set $\hat{f}(x) = 0$.

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

- Solve

$$\underset{\hat{f}^b}{\text{minimize}} \sum_{i=1}^{n} L(y_i, \hat{f}(x_i) + \hat{f}^b(x_i))$$

  where $\hat{f}^b$ is a tree with d splits.

- Update $\hat{f}$ by adding in a shrunken version of the new tree: $\hat{f}(x) \leftarrow \hat{f}(x) + \lambda \hat{f}^b(x)$.

(iii) Output the boosted model:

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

Efron and Hastie:

This algorithm is easier to state than to implement.

In the squared error loss case we have

$$L(y_i, \hat{f}(x_i) + \hat{f}^b(x_i)) = .5(y_i - (\hat{f}(x_i) + \hat{f}^b(x_i))^2 = .5(r_i - \hat{f}^b(x_i))^2$$

so we get $\hat{f}^b$ by fitting a regression tree to the residuals.

We will solve this by making the general case look like the squared error case by generalizing the notion of a residual.

Generalized residuals

In the linear mean (normal) case we have:

$$L(y_i, \mu_i) = \frac{1}{2}(y_i - \mu_i)^2.$$

$$-\frac{\partial L(y_i, \mu_i)}{\partial \mu_i} = (y_i - \mu_i).$$

We lower our loss, by moving μ_i towards y_i.

In general we let,

$$r_i = -\frac{\partial L(y_i, \alpha_i)}{\partial \alpha_i}$$

Gradient Boosting Algorithm with Trees

(i) Set $\hat{f}(x) = 0$.

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

- compute $r_i = -\frac{\partial L(y_i, \alpha_i)}{\partial \alpha_i}$, evaluated at $\alpha_i = \hat{f}(x_i)$, $i = 1, 2, \ldots, n$.
- Fit a tree $\hat{f}^b$ with d splits ($d + 1$ terminal nodes) to the training data (X, r) using squared error loss.
- Update $\hat{f}$ by adding in a shrunken version of the new tree: $\hat{f}(x) \leftarrow \hat{f}(x) + \lambda \hat{f}^b(x)$.

(iii) Output the boosted model:

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

At each boosting iteration, we move $\hat{f}$ in the direction that will lower the loss on the training data, subject to our move being a simple tree.

Note:

You can boost with other weak learners besids small trees, but in practice boosting is almost always done with trees.

Truly remarkable.

XGBoost

We will just sketch this version of boosting.

We start by reformulating the “fit resids then crush” approach to one the looks like regularization.

At each boosting iteration, instead of solving,

$$\underset{\hat{f}^b}{\text{minimize}} \sum_{i=1}^{n} L(y_i, \hat{f}(x_i) + \hat{f}^b(x_i))$$

and then crushing, we solve

$$\underset{\hat{f}^b}{\text{minimize}} \sum_{i=1}^{n} L(y_i, \hat{f}(x_i) + \hat{f}^b(x_i)) + \Omega(\hat{f}^b)$$

where Ω measures the complexity of $\hat{f}^b$.

Remember, $\hat{f}^b$ corresponds to a regression tree T^b.

A large complexity penalty will give us a simple T^b.

Let $w(T)$ be the values in the leaf nodes of the tree.

$$\Omega(\hat{f}^b) = \gamma \, |T^b| + \frac{1}{2} \, ||w(T^b)||^2.$$

At each boosting iteration we optimize to get $\hat{f}^b$ and our final function is just

$$\hat{f}(x) = \sum_{i=1}^{n} \eta \, \hat{f}^b(x)$$

So, η is the xgboost greek name for what we have called λ.
It is called the *learning rate* in that is controls how big a move we make in a certain direction ($\hat{f}^b(x)$) but the space we are moving in is function space rather in a parameter space as in our usual gradient descent.

The XGBoost algorithm involve an approximate method for solving this problem based on a quadratic expansion of $L(y, \alpha)$ in α.

Then there are a lot of hard core computational details that make the algorithm effective in big data high dimensional problems.

Note:

There are very popular versions of xgboost in both python and R.

Things people like about xgboost:

- based on gradient boosting but can handle big data.
- has our basic B and λ parameters. λ is called the "learning rate" based on an analogy with gradient descent.
- has parameters to L1 or L2 shrink the leaf node parameters.
- can add in random forest type ideas by sampling the data and/or the features when building the new tree.
- several parameters control tree size, max depth , max number of bottom nodes, min number of observations in a node ...

For "tabular data" XGBoost is very successful !!!

See also LightGBM (big data) Catboost (categorical features).

Diabetes Data

Let's run try the `xgboost` module in python.

This implements a variety of boosting approaches, in particular, extreme gradient boosting.

There is also an `xgboost` package in R.

Here are the imports we use.

```
import pandas as pd
import numpy as np

from sklearn import datasets
from sklearn.model_selection import  cross_val_score
from sklearn.model_selection import  validation_curve

from xgboost import XGBRegressor

import matplotlib.pyplot as plt
import seaborn as sns; sns.set()
```

There is a corresponding `XGBClassifier` module.

Get the diabetes data:

```
#ddf = pd.read_csv("diabetes.csv") # from Rob data sets webpage
ddf = pd.read_csv("https://bitbucket.org/remcc/rob-data-sets/downloads/diabetes.csv") # from Rob data sets
xvnms = ddf.columns.values[1:11]
yX = ddf.to_numpy()
y = yX[:,0]
X = yX[:,1:11]
```

```
In [3]: print("X is:")
   ...: print(X.shape)
   ...: print("y is:")
   ...: print(y.shape)
   ...:
X is:
(442, 10)
y is:
(442,)
```

- B: `n_estimators`
- d: `max_depth`
- λ: `learning_rate`

There are many other tuning parameters !!

```
Init signature: XGBRegressor(*, objective='reg:squarederror', **kwargs)
Docstring:
Implementation of the scikit-learn API for XGBoost regression.


Parameters
----------

    n_estimators : int
        Number of gradient boosted trees.  Equivalent to number of boosting
        rounds.

    max_depth : int
        Maximum tree depth for base learners.
    learning_rate : float
        Boosting learning rate (xgb's "eta")
```

Recall what the depth is:

```
Depth 0:    A        (just root)

Depth 1:    A        (root splits once)
           / \
          B   C

Depth 2:    A        (root splits twice)
           / \
          B   C
         / \
        D   E
```

So, with depth 2, we could have a split on x1 at A and a split on x2 at B giving an iteraction.

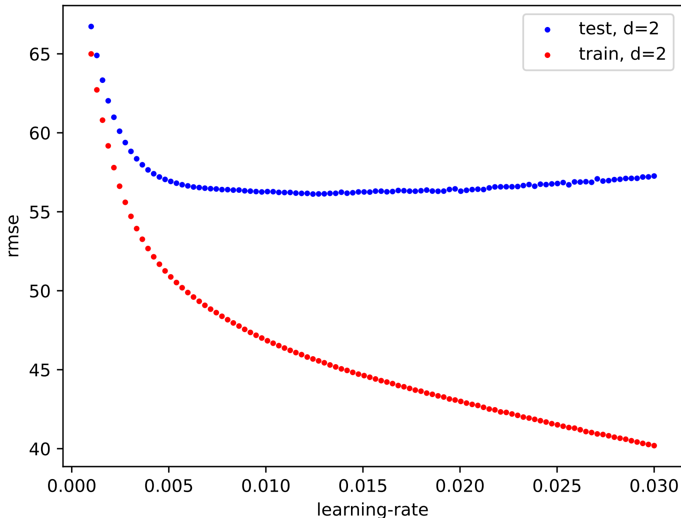
To compare with our previous results where we used polynomial terms, let's use $d = 2$, just two decision rules in our trees. This allows for 2-way interaction.

```
## d=2
xgbmod = XGBRegressor(booster='gbtree',objective='reg:squarederror',
   max_depth=2, learning_rate=0.1, n_estimators=500, random_state=2, n_jobs=-1)

# do cv at every value of lr in lrv (learning rate vector)
lrv = np.linspace(start=.001,stop=.03,num=100)
trainS, testS = validation_curve(xgbmod,X,y,'learning_rate',
  lrv,cv=10,scoring='neg_mean_squared_error')

# transform neg_mean_squared_error to rmse
trrmse = np.sqrt(-trainS.mean(axis=1))
termse = np.sqrt(-testS.mean(axis=1))

#plot in and out of sample rmse
plt.scatter(lrv,termse,c='blue',s=5)
plt.scatter(lrv,trrmse,c='red',s=5)
plt.xlabel("learning-rate"); plt.ylabel("rmse")
plt.legend(['test, d=2','train, d=2'])
plt.savefig("xgb_d2_train-test.pdf")
```

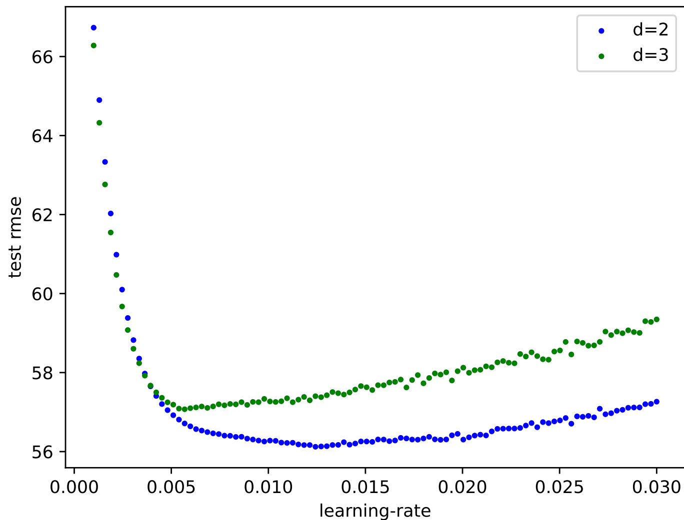


Recall that we got a minimum rmse of about 55 with the lasso and second order polynomials.
So, our boosting results are very similar.

Now let's up d to 3 and see what happens.

Note that *this is very interesting !!!!!!!!!!!!!!!!!!!!!*

In our previous analysis using the LASSO, we had to assume 2-way interaction.



$d = 3$ looks worse, *but I did not parameter tuning !!*

In R:

Here is code for doing logit, rf, and boosting with a binary categorical y using glm, randomForest, and gbm.

We the training td1.csv for train and td2.csv for test.

```
trainDf = read.csv("td1.csv")
trainDf$purchase = as.factor(trainDf$purchase)
testDf = read.csv("td2.csv")
testDf$purchase = as.factor(testDf$purchase)
names(trainDf)[1]="y"
names(testDf)[1]="y"

phatL = list() #store the test phat for the different methods here

###fit logit
lgfit = glm(y~.,trainDf,family=binomial)
print(summary(lgfit))
#predict using logit
phat = predict(lgfit,testDf,type="response")

phatL$logit = matrix(phat,ncol=1) #logit phat
```

```
##settings for randomForest
p=ncol(trainDf)-1
mtryv = c(p,sqrt(p))
ntreev = c(500,1000)
setrf = expand.grid(mtryv,ntreev)
colnames(setrf)=c("mtry","ntree")
phatL$rf = matrix(0.0,nrow(testDf),nrow(setrf))

###fit rf
library(randomForest)
for(i in 1:nrow(setrf)) {
   cat("on randomForest fit ",i,"\n")
   print(setrf[i,])

   #fit and predict
   frf = randomForest(y~.,data=trainDf,mtry=setrf[i,1],ntree=setrf[i,2])
   phat = predict(frf,newdata=testDf,type="prob")[,2]

   phatL$rf[,i]=phat
}
```

```
##settings for boosting
idv = c(2,4); ntv = c(1000,5000); shv = c(.1,.01)
setboost = expand.grid(idv,ntv,shv)
colnames(setboost) = c("tdepth","ntree","shrink")
phatL$boost = matrix(0.0,nrow(testDf),nrow(setboost))

trainDfB = trainDf; trainDfB$y = as.numeric(trainDfB$y)-1
testDfB = testDf; testDfB$y = as.numeric(testDfB$y)-1

##fit boosting
library(gbm)
tm1 = system.time({ #get the time, will use this later
for(i in 1:nrow(setboost)) {
   cat("on boosting fit ",i,"\n")
   print(setboost[i,])

   ##fit and predict
   fboost = gbm(y~.,data=trainDfB,distribution="bernoulli",
              n.trees=setboost[i,2],interaction.depth=setboost[i,1],
                         shrinkage=setboost[i,3])
   phat = predict(fboost,newdata=testDfB,n.trees=setboost[i,2],type="response")

   phatL$boost[,i] = phat
}
})
```

Let's look at the lift for the boosting fits.

The `caret` package has nice functions for the lift, ROC, and AUC. The list is so simple, we can just use a little function in "mlfuns.R".

```
source("mlfuns.R")

#have to store all the phats in a list.
boostL = list()
for(i in 1:ncol(phatL$boost)) boostL[[i]]= phatL$boost[,i]

#get the lift
par(mfrow=c(2,1))
plot(1:8,1:8)
for(i in 1:8) abline(v=i,col=i,lwd=2)
temp = liftfL(testDf$y,boostL)
```

Let's try the loop over boosting settings using doParallel, the simple R library for doing things in parallel.

```
library(doParallel)
cl <- makeCluster(4)
registerDoParallel(cl)

#how many workers?
cat("number of workers is: ",getDoParWorkers(),"\n")

tm2 = system.time({
   boostres = foreach(i=1:nrow(setboost), .combine=cbind) %dopar% {
      library(gbm)
      fboost = gbm(y~.,data=trainDfB,distribution="bernoulli",
              n.trees=setboost[i,2],interaction.depth=setboost[i,1],
                         shrinkage=setboost[i,3])
      phat = predict(fboost,newdata=testDfB,
                               n.trees=setboost[i,2],type="response")
      return(phat)
  }
})

stopCluster(cl)
```

```
##let's check get similar results
par(mfrow=c(4,2))
for(i in 1:8) {
  plot(boostres[,i],phatL$boost[,i])
  abline(0,1,col="red",lwd=2)
}

##let's compare the times
cat("serial time is: ",tm1[3],"\n")
cat("parallel time is: ",tm2[3],"\n")
```

The parallel version is quite a bit faster for almost no work.

For the big value of shrink, it does not look like two boosting runs are giving us the same result. The gbm package actually does a random sample of training data at each interation and this may explain it.

```
library(pROC)

par(mfrow=c(1,2))
temp=liftf(testDf$y,phatL$logit[,1])

rocCurve = roc(response = testDf$y,predictor=phatL$logit[,1])
plot(rocCurve)

cat("auc, logit: ", auc(rocCurve),"\n")
rocCurve = roc(response = testDf$y,predictor=phatL$boost[,5])
cat("auc, boost 5: ", auc(rocCurve),"\n")
rocCurve = roc(response = testDf$y,predictor=phatL$boost[,4])
cat("auc, boost 4: ", auc(rocCurve),"\n")
```