Lecture 3

Bagging and Boosting

Mladen Kolar (mkolar@chicagobooth.edu)

Classification trees are popular since they are highly **interpretable**. They are also model-free (don't assume an underlying distribution for the data).

But they don't generally give a **prediction accuracy** competitive with that of other classifiers (logistic regression, k-nearest neighbors, etc.) The reason: trees have somewhat inherently **high variance**. The separations made by splits are enforced at all lower levels of the tree, which means that if the data is perturbed slightly, the new tree can have a considerably different sequence of splits, leading to a different classification rule.

Now we'll learn of two ways to control the variance, or **stabilize** the predictions made by trees. Of course these solutions aren't perfect: in doing so, we can greatly improve prediction accuracy but we suffer in terms of interpretability.

Joint distribution and Bayes classifier

Suppose that we observe training data (x_i, y_i), $i = 1, \dots, n$, which represents n independent draws from some unknown probability distribution $\mathcal{F}$. E.g., this could be classification data, with $y_i \in \{1, \dots, K\}$ being the class label and $x_i \in \mathbb{R}^p$ the associated feature vector.

Note that $\mathcal{F}$ describes the joint distribution of X and Y:

$$P_{\mathcal{F}}\{(X, Y) = (x, y)\}.$$

If we knew $\mathcal{F}$, then the best thing to do would be to simply classify according to the Bayes classifier:

$$f(x) = \arg\max_{j=1,\dots,K} P_{\mathcal{F}}\{(X, Y) = (x, j)\}.$$

The bootstrap

A key idea in modern statistics is the bootstrap.

Suppose that we observe training data (x_i, y_i), $i = 1, \ldots, n$, which represents n independent draws from some unknown probability distribution $\mathcal{F}$.

The bootstrap is a fundamental resampling tool in statistics. The basic idea underlying the bootstrap is that we can estimate the true $\mathcal{F}$ by the so-called empirical distribution $\hat{\mathcal{F}}$.

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

The bootstrap

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

A bootstrap sample of size m from the training data is (x_i^*, y_i^*), $i = 1, \ldots, m$, where each (x_i^*, y_i^*) are drawn from uniformly at random from $(x_1, y_1), \ldots, (x_n, y_n)$, with replacement.

This corresponds exactly to m independent draws from $\hat{\mathcal{F}}$. Hence it approximates what we would see if we could sample more data from the true $\mathcal{F}$. We often consider $m = n$, which is like sampling an entirely new training set.

Note: not all of the training points are represented in a bootstrap sample, and some are represented more than once. When $m = n$, about 36.8% of points are left out, for large n.

Bagging

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 **complex** model to each bootstrap sample. his will give us B models
- Combine the results from each of the B models to get an overall prediction.

Let us see an example using kNN.

Bagging

Given a training data (x_i, y_i), $i = 1, \ldots n$, bagging[2] averages the predictions from classification trees over a collection of boostrap samples. For $b = 1, \ldots B$ (e.g., $B = 100$), we draw n boostrap samples (x_i^{*b}, y_i^{*b}), $i = 1, \ldots n$, and we fit a classification tree $\hat{f}^{\text{tree},b}$ on this sampled data set. Then at the end, to classify an input $x \in \mathbb{R}^p$, we simply take the most commonly predicted class:

$$\hat{f}^{\text{bag}}(x) = \underset{k=1,\ldots K}{\text{argmax}} \sum_{b=1}^{B} 1\{\hat{f}^{\text{tree},b}(x) = k\}$$

This is just choosing the class with the most votes. Two options:

- Simple strategy: grow fairly large trees on each sampled data set, with no pruning
- More involved strategy: prune back each tree as we do with CART, but use the original training data (x_i, y_i), $i = 1, \ldots n$ as the validation set, instead of performing cross-validation

[2]Breiman (1996), "Bagging Predictors"

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.

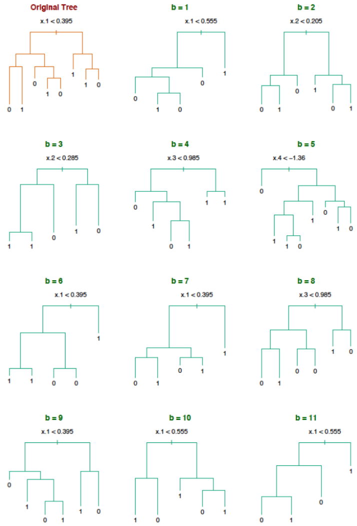
For categorical y, it is not quite so obvious how you want to combine the results from the different trees. Often people let the trees vote: given x

get a prediction from each tree and the category that gets the most votes (out of B ballots) is the prediction.

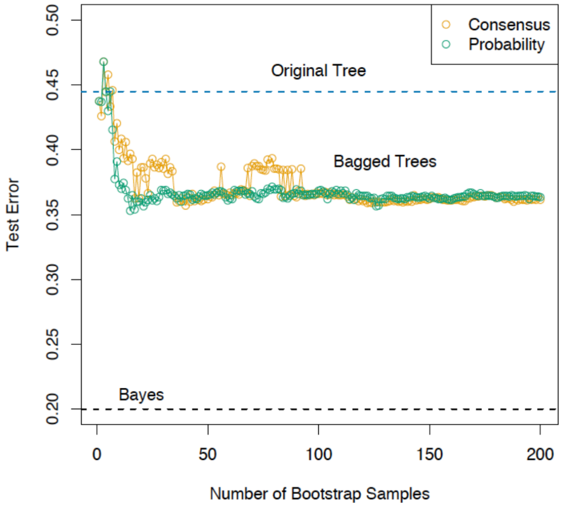
Alternatively, you could average the $\hat{p}$ from each tree. Most software seems to follow the vote plan.

Example: bagging

$n = 30$ training data points, $p = 5$ features, and $\mathrm{K} = 2$ classes. No pruning used in growing trees:

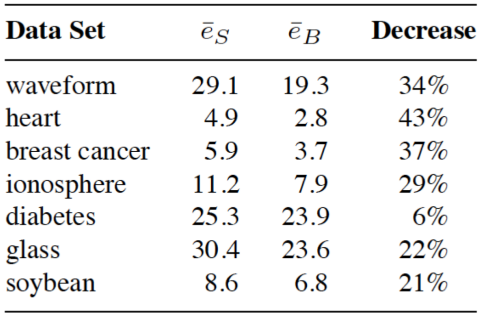


Bagging helps decrease the misclassification rate of the classifier (evaluated on a large independent test set). Look at the orange curve:



Example: Breiman's bagging

Example from the original Breiman paper on bagging: comparing the misclassification error of the CART tree (pruning performed by cross-validation) and of the bagging classifier (with $B = 50$):



Data Set	$\bar{e}_S$	$\bar{e}_B$	**Decrease**
waveform	29.1	19.3	34%
heart	4.9	2.8	43%
breast cancer	5.9	3.7	37%
ionosphere	11.2	7.9	29%
diabetes	25.3	23.9	6%
glass	30.4	23.6	22%
soybean	8.6	6.8	21%

Voting probabilities are not estimated class probabilities

Suppose that we wanted estimated class probabilities out of our bagging procedure. What about using, for each $k = 1, \dots K$:

$$\hat{p}_k^{\text{bag}}(x) = \frac{1}{B} \sum_{b=1}^{B} 1\{\hat{f}^{\text{tree},b}(x) = k\}$$

I.e., the proportion of votes that were for class k?

This is generally not a good estimate. Simple example: suppose that the true probability of class 1 given x is 0.75. Suppose also that each of the bagged classifiers $\hat{f}^{\text{tree},b}(x)$ correctly predicts the class to be 1. Then $\hat{p}_1^{\text{bag}}(x) = 1$, which is wrong

What's nice about trees is that each tree already gives us a set of predicted class probabilities at x: $\hat{p}_k^{\text{tree},b}(x)$, $k = 1, \dots K$. These are simply the proportion of points in the appropriate region that are in each class

Alternative form of bagging

This suggests an alternative method for bagging. Now given an input $x \in \mathbb{R}^p$, instead of simply taking the prediction $\hat{f}^{\mathrm{tree},b}(x)$ from each tree, we go further and look at its predicted class probabilities $\hat{p}_k^{\mathrm{tree},b}(x)$, $k = 1, \dots K$. We then define the bagging estimates of class probabilities:

$$\hat{p}_k^{\mathrm{bag}}(x) = \frac{1}{B} \sum_{b=1}^{B} \hat{p}_k^{\mathrm{tree},b}(x) \quad k = 1, \dots K$$

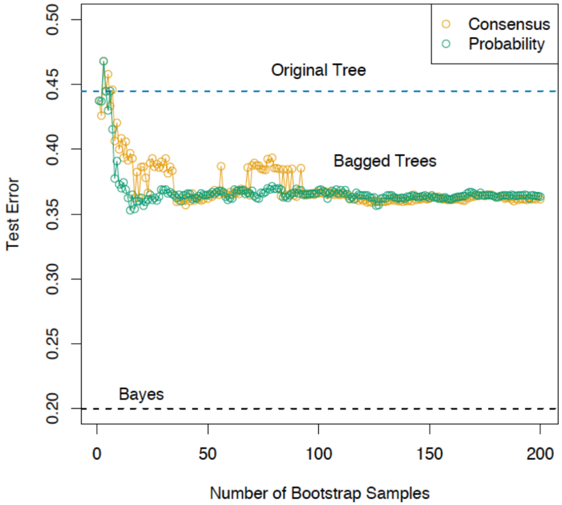
The final bagged classifier just chooses the class with the highest probability:

$$\hat{f}^{\mathrm{bag}}(x) = \underset{k=1,\dots K}{\operatorname{argmax}}\ \hat{p}_k^{\mathrm{bag}}(x)$$

This form of bagging is preferred if it is desired to get estimates of the class probabilities. Also, it can sometimes help the overall prediction accuracy

Example: alternative form of bagging

Previous example revisited: the alternative form of bagging produces misclassification errors shown in green



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.

Why is bagging working?

Now, we consider classification. Here is a simplified setup with $K = 2$ classes to help understand the basic phenomenon.

Suppose that for a given input x, we have B independent classifiers $\hat{f}^b(x)$, $b = 1, \ldots, B$, and each classifier has a misclassification rate of 0.4. That is, suppose that the true class at x is 1, so

$$P(\hat{f}^b(x) = 2) = 0.4.$$

Now we form the bagged classifier:

$$\hat{f}^{\mathrm{bag}}(x) = \arg\max_{k=1,2} \sum_{b=1}^{B} 1\{\hat{f}^b(x) = k\}.$$

Let $B_2 = \sum_{b=1}^{B} 1\{\hat{f}^b(x) = k\}$ be the number of votes for class 2.

Notice that $B_2 = \sum_{b=1}^{B} 1\{\hat{f}^b(x) = k\}$

$$B_2 \sim \mathrm{Binom}(B, 0.4).$$

Therefore the **misclassification rate** of the bagged classifier is

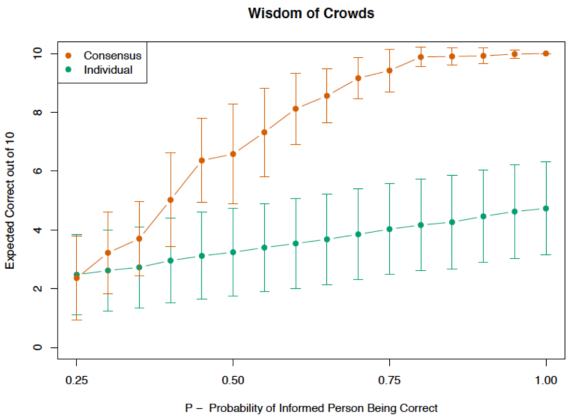
$$P(\hat{f}^{\mathrm{bag}}(x) = 2) = P(B_2 \geq B/2).$$

As $B_2 \sim \mathrm{Binom}(B, 0.4)$, this $\to 0$ as $B \to \infty$. In other words, the bagged classifier has perfect predictive accuracy as the number of sampled data sets $B \to \infty$.

So why did the prediction error seem to level off in our examples? Of course, the caveat here is **independence**. The classifiers that we use in practice, $\hat{f}^{\mathrm{tree},b}$, are clearly not independent, because they are fit on very similar data sets (bootstrap samples from the same training set).

Wisdom of crowds

The **wisdom of crowds** is a concept popularized outside of statistics to describe the same phenomenon. It is the idea that the collection of knowledge of an independent group of people can exceed the knowledge of any one person individually. Interesting example (from ESL page 287):



When will bagging fail?

Now suppose that we consider the same simplified setup as before (independent classifiers), but each classifier has a misclassification rate:

$$P(\hat{f}^b(x) = 2) = 0.6.$$

Then by the same arguments, the bagged classifier has a misclassification rate of

$$P(\hat{f}^{\mathrm{bag}}(x) = 2) = P(B_2 \geq B/2).$$

As $B_2 \sim \mathrm{Binom}(B, 0.6)$, this $\to 1$ as $B \to \infty$. In other words, the bagged classifier is perfectly inaccurate as the number of sampled data sets $B \to \infty$.

Again, the independence assumption doesn't hold with trees, but the **take-away message** is clear: bagging a good classifier can improve predictive accuracy, but bagging a bad one can seriously degrade predictive accuracy.

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.

Disadvantages

It is important to discuss some disadvantages of bagging:

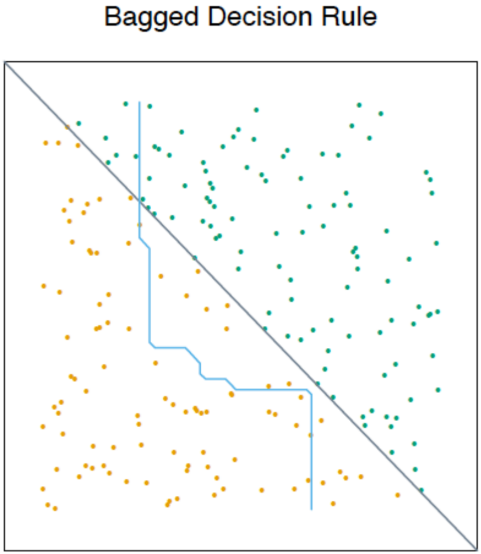
- Loss of interpretability: the final bagged classifier is not a tree, and so we forfeit the clear interpretative ability of a classification tree
- Computational complexity: we are essentially multiplying the work of growing a single tree by B (especially if we are using the more involved implementation that prunes and validates on the original training data)

You can think of bagging as extending the space of models. We go from fitting a single tree to a large group of trees. Note that the final prediction rule cannot always be represented by a single tree.

Sometimes, this enlargement of the model space isn't enough, and we would benefit from an even greater enlargement.

Example: limited model space

Bagging cannot really represent a diagonal decision rule



Recap: bagging

We learned **bagging**, a technique in which we draw many bootstrap-sampled data sets from the original training data, train on each sampled data set individually, and then aggregate predictions at the end. We applied bagging to classification trees.

There were two strategies for aggregate the predictions: taking the class with the majority vote (the **consensus** strategy), and averaging the estimated class probabilities and then voting (the **probability** strategy). The former does not give good estimated class probabilities; the latter does and can sometimes even improve prediction accuracy.

Bagging works if the base classifier is **not bad** (in terms of its prediction error) to begin with. Bagging bad classifiers can degrade their performance even further. Bagging still can't represent some basic decision rules.

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 $m = \sqrt{p}$

Notes:

- Bagging is Random Forests with $m = p$.
- There is no explicit regularization parameter as in the lasso and single tree prediction.

OOB Error Estimation

OOB is “Out of Bag”.

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

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

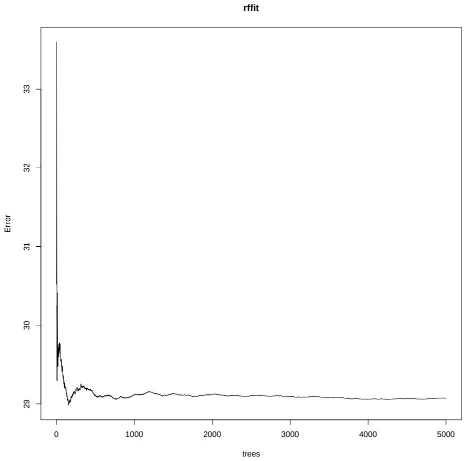
One can show that, on average, each bagged tree makes use of about 2/3 of the observations.

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

Bagging for Boston: y=medv, x=lstat

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

Suggests you just need a couple of hundred trees.

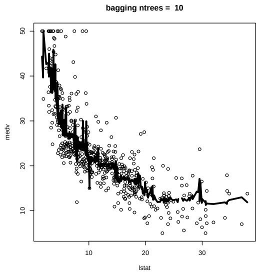


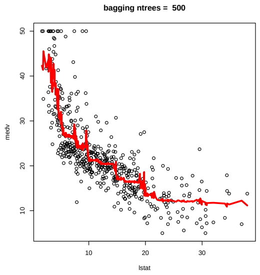
Bagging for Boston: y=medv, x=lstat.

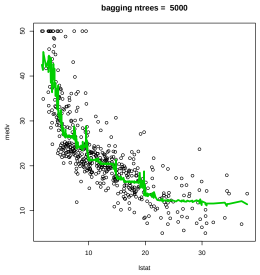
With 10 trees our fit is too jumbly.

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

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







Boosting

Boosting

Boosting[1] is similar to bagging in that we combine the results of several classification trees. However, boosting does something fundamentally different, and can work a lot better

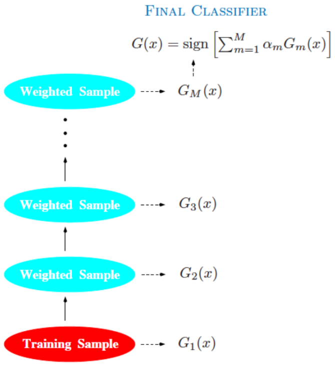
As usual, we start with training data (x_i, y_i), $i = 1, \ldots n$. We'll assume that $y_i \in \{-1, 1\}$ and $x_i \in \mathbb{R}^p$. Hence we suppose that a classification tree (fit, e.g., to the training data) will return a prediction of the form $\hat{f}^{\text{tree}}(x) \in \{-1, 1\}$ for an input $x \in \mathbb{R}^p$

In boosting we combine a weighted sum of B different tree classifiers,

$$\hat{f}^{\text{boost}}(x) = \text{sign}\left(\sum_{b=1}^{B} \alpha_b \, \hat{f}^{\text{tree},b}(x) \right)$$

[1] Freund and Schapire (1995), "A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting". Similar ideas were around earlier

One of the key differences between boosting and bagging is how the individual classifiers $\hat{f}^{\text{tree},b}$ are fit. Unlike in bagging, in boosting we fit the tree to the entire training set, but adaptively weight the observations to encourage better predictions for points that were previously misclassified



Classification trees with observation weights

Now suppose that we are are additionally given observation weights w_i, $i = 1, \ldots n$. Each $w_i \geq 0$, and a higher value means that we place a higher importance in correctly classifying this observation

The CART algorithm can be easily adapted to use the weights. All that changes is that our sums become weighted sums. I.e., we now use the weighted proportion of points of class k in region R_m:

$$\hat{p}_k(R_m) = \frac{\sum_{x_i \in R_m} w_i 1\{y_i = k\}}{\sum_{x_i \in R_m} w_i}$$

As before, we let

$$c_m = \underset{k=1,\ldots K}{\operatorname{argmax}} \ \hat{p}_k(R_m)$$

and hence $1 - \hat{p}_{c_m}(R_m)$ is the weighted misclassification error

The basic boosting algorithm (AdaBoost)

Given training data (x_i, y_i), $i = 1, \ldots n$, the basic boosting method is called AdaBoost, and can be described as:

- Initialize the weights by $w_i = 1/n$ for each i
- For $b = 1, \ldots B$:
  1. Fit a classification tree $\hat{f}^{\text{tree},b}$ to the training data with weights $w_1, \ldots w_n$
  2. Compute the weighted misclassification error
  $$e_b = \frac{\sum_{i=1}^n w_i 1\{y_i \neq \hat{f}^{\text{tree},b}(x_i)\}}{\sum_{i=1}^n w_i}$$
  3. Let $\alpha_b = \log\{(1 - e_b)/e_b\}$
  4. Update the weights as
  $$w_i \leftarrow w_i \cdot \exp(\alpha_b 1\{y_i \neq \hat{f}^{\text{tree},b}(x_i)\})$$
  for each i
- Return $\hat{f}^{\text{boost}}(x) = \text{sign}\Big(\sum_{b=1}^B \alpha_b \hat{f}^{\text{tree},b}(x) \Big)$

Example: boosting stumps

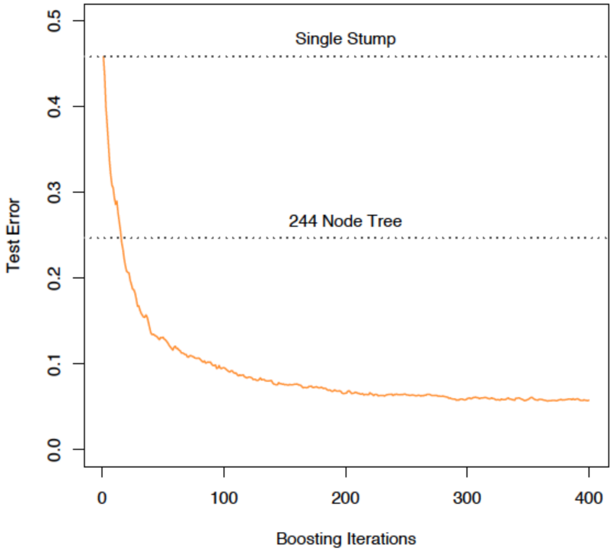
Example (from ESL page 339): here $n = 1000$ points were drawn from the model

$$Y_i = \begin{cases} 1 & \text{if } \sum_{j=1}^{10} X_{ij}^2 > \chi^2_{10}(0.5) \\ -1 & \text{otherwise} \end{cases}, \quad \text{for } i = 1, \dots n$$

where each $X_{ij} \sim N(0,1)$ independently, $j = 1, \dots 10$

A stump classifier was computed: this is just a classification tree with one split (two leaves). This has a bad misclassification rate on independent test data, of about 45.8%. (Why is this not surprising?)

On the other hand, boosting stumps achieves an excellent error rate of about 5.8% after $B = 400$ iterations. This is much better than, e.g., a single large tree (error rate 24.7%)


Single Stump
244 Node Tree
Test Error
0.0
0.1
0.2
0.3
0.4
0.5
0
100
200
300
400
Boosting Iterations

Boosting for regression

Algorithm:

1. Set $\hat{f}(x) = 0$ and $r_i = y_i$ for all i in the training set.
2. For $b = 1, 2, \ldots, B$, repeat:
   (a) Fit a tree $\hat{f}^b$ with d splits ($d+1$ terminal nodes) to the training data (X, r).
   (b) Update $\hat{f}$ by adding in a shrunken version of the new tree:
   $$\hat{f}(x) \leftarrow \hat{f}(x) + \lambda \hat{f}^b(x).$$
   (c) Update the residuals,
   $$r_i \leftarrow r_i - \lambda \hat{f}^b(x_i).$$
3. Output the boosted model,
   $$\hat{f}(x) = \sum_{b=1}^{B} \lambda \hat{f}^b(x).$$

Notes

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 "shrinkage" parameter. It makes each new tree is a weak learner in that it only does a little more fitting.

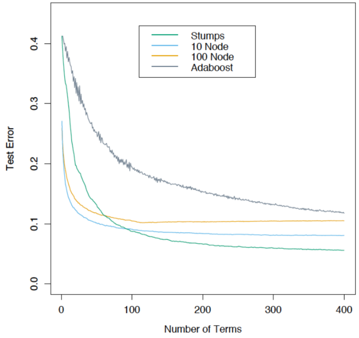
Observe that boosting for categorical and numeric variables works in a same way. However, it is easier to define what is left over for regression.

Choosing the size of the trees

How big should the trees be used in the boosting procedure?

Remember that in bagging we grew large trees, and then either stopped (no pruning) or pruned back using the original training data as a validation set. In boosting, actually, the best methods if to grow small trees with no pruning

In ESL 10.11, it is argued that right size actually depends on the level of the iteractions between the predictor variables. Generally trees with 2–8 leaves work well; rarely more than 10 leaves are needed



Disadvantages

As with bagging, a major downside is that we lose the simple interpretability of classification trees. The final classifier is a weighted sum of trees, which cannot necessarily be represented by a single tree.

Computation is also somewhat more difficult, though quite straightforward using packages in R. Further, because we are growing small trees, each step can be done relatively quickly (compared to say, bagging).

To deal with the first downside, a measure of variable importance has been developed for boosting (and it can be applied to bagging, also).

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.

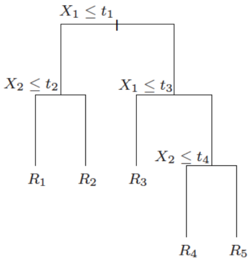
In particular, we are often interested in which variables in x are really the "important" ones.

Variable importance for trees

For a single decision tree $\hat{f}^{\text{tree}}$, we define the squared importance for variable j as

$$\text{Imp}_j^2(\hat{f}^{\text{tree}}) = \sum_{k=1}^{m} \hat{d}_k \cdot 1\{\text{split at node } k \text{ is on variable } j\}$$

where m is the number of internal nodes (non-leaves), and $\hat{d}_k$ is the improvement in training misclassification error from making the kth split



Variable importance for boosting

For boosting, we define the squared importance for a variable j by simplying averaging the squared importance over all of the fitted trees:

$$\mathrm{Imp}_j^2(\hat{f}^{\mathrm{boost}}) = \frac{1}{B}\sum_{b=1}^{B}\mathrm{Imp}_j^2(\hat{f}^{\mathrm{tree},b})$$

(To get the importance, we just take the squared root.) We also usually set the largest importance to 100, and scale all of the other variable importances accordingly, which are then called relative importances

This averaging stabilizes the variable importances, which means that they tend to be much more accurate for boosting than they do for any single tree