# STA414/2104: Week 10

## Trees, Bagging, and Boosting

Alex Stringer

March 13th, 2018

## This week

This week we will talk about tree-based methods.

We will describe decision trees, and then look at two algorithms for improving learning methods that have become very popular, in the context of decision trees:

- **Bagging**: fit a bunch of models on bootstrapped samples from the training data, then average them. Modifications of this leads to *Random Forests*
- **Boosting**: fit a sequence of weak models that learn from the mistakes of their predecessors. Modification of this leads to *Gradient Boosted Trees*

## Suggested Reading

Material in these slides was adapted from the following sections of *Elements of Statistical Learning*, as referenced in the course syllabus. You should review these sections for a more in-depth view of the material.

- 8.7, “Bagging”
- 9.2, “Tree-Based Methods” and 9.6, “Missing Data”
- All of chapter 10, “Boosting and Additive Trees”, although we focus more on sections 10.1 - 10.4
- All of chapter 15, “Random Forests”

## Decision Regions

Suppose we have a training set $(y_i, \mathbf{x}_i), i = 1 \ldots n$ where the features $\mathbf{x}_i \in \mathbb{R}^p$.

Roughly speaking, we can view all of the learning procedures we have talked about in this course as aiming to find a function $\hat{y}(\mathbf{x})$ that partitions the feature space $\mathbb{R}^p$ into disjoint regions.

For a new $\mathbf{x}$, we look at what region it falls into, and assign the corresponding prediction.

Of course, usually $\hat{y}(\mathbf{x})$ is a continuous function of $\mathbf{x}$, so we don't *think* of the predictions as being "regions", really.

But do we even need a continuous function? Is $y$ *that* sensitive to changes in $\mathbf{x}$?

## Decision Trees

Rather than specifying a parametric model for $y$, optimizing for parameters, and then using the resulting prediction function, we can attempt to directly partition the feature space into disjoint regions $R_m, m = 1, 2, \ldots, M \in \mathbb{N}$.

We would then predict $y$ for a new $\mathbf{x}$ by looking at what region $\mathbf{x}$ falls in to, and taking the average $y$ for that region (this is still optimal under squared-error loss for regression and entropy-loss for classification, so it is not completely arbitrary).

## Decision Trees

Mathematically, we would predict according to

$$\hat{y}(\mathbf{x}) = \sum_{m=1}^{M} \alpha_m I(\mathbf{x} \in R_m)$$

Decision trees, then, are linear basis function models.

How can we find these regions $R_m$?

## Decision Trees

Our task is to find features $\{j : j = 1 \dots p\}$ and corresponding split-points $s \in \mathbb{R}$ for each, to separate $\mathbb{R}^p$ into disjoint regions

$$R_1 \dots R_m$$

for some $m \in \mathbb{R}$, and to do this to minimize the appropriate loss function,

$$L(j, s) = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 \text{ (regression)}$$

$$L(j, s) = -\sum_{i=1}^{n} \sum_{k=1}^{K} \hat{y}_{ik} \log \hat{p}_{ik} \text{ (classification)}$$

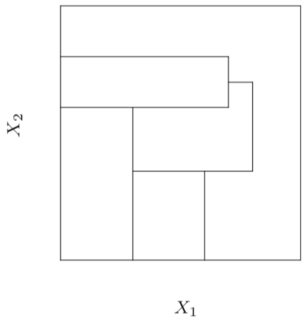
Doing this globally is computationally infeasible.

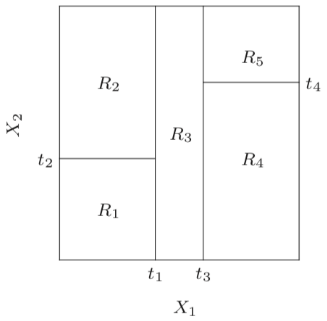
## Decision Trees

A **decision tree** solves this problem using a greedy recursive partitioning algorithm, as follows:

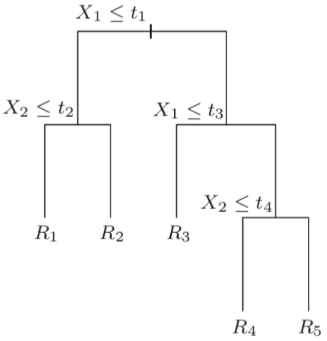
1. Start with the null prediction, $\hat{y}_i = \bar{y}$. Evaluate the loss function.
2. Choose the feature and the split point that provide the largest reduction in the loss function
    1. For each feature, do a grid search over split points at all unique values in the training set
3. Split the data according to this feature, obtaining two new regions
4. Recursively apply the above steps to data within each of these regions

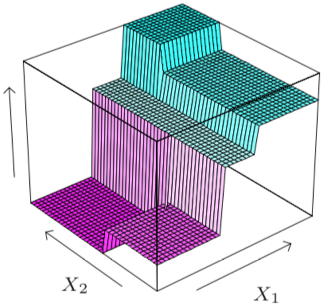
## Illustration (ESL, page 306)





## Illustration (ESL, page 306)





## Stopping

How to decide when to stop growing the tree?

Because of the recursive nature of the algorithm, this is a task in nested model comparison.

BIC and AIC aren't usually used (how to penalize complexity?).

Typically the following tuning parameters are considered:

- Minimum # of observations per terminal node (i.e. per region $R_m$)
- Maximum number of terminal nodes ($m$)
- Maximum number of splits (recursions)
- "Complexity parameter" $c$- the minimum reduction in loss required for a split to be accepted

## Stopping

These 4 parameters are all related.

Typically in practice, one cross-validates for $c$ while setting reasoable limits on the other parameters.

Be careful though- the effect of $c$ on the tree is highly discrete. You have to zero in on the best values, and always examine the resulting trees.

You could, technically, optimize for $c$ directly, but this is hard because of this discrete effect. Even though $c$ takes values in the real numbers, the loss as a function of $c$ is discrete.

## Overfitting

Decision trees find very complex higher-order interactions in data.

This is because every split is made conditional on the previous splits.

If you split $X_1 < s_1$ then $X_2 < s_2$, your resulting model is

$$\begin{aligned}\hat{y}(x_1, x_2) =& \alpha_1 I(X_1 < s_1) I(X_2 < s_2) \\ &+ \alpha_2 I(X_1 > s_1) I(X_2 < s_2) \\ &+ \alpha_3 I(X_1 < s_1) I(X_2 > s_2) \\ &+ \alpha_4 I(X_1 > s_1) I(X_2 > s_2)\end{aligned}$$

When the number of features and splits is large, the algorithm mines deeper and deeper structure.

## Overfitting

. . . but it mines deeper and deeper into the *training* data, which is not necessarily a good thing.

Cross validation and test-set evaluation are **imperative** when fitting decision trees.

It is worth making this abundently clear. In a regression model, we predict using a conditional mean, which is a **globally** optimal prediction function. Hence regression models don't overfit that badly.

Decision trees' optimality is entirely **local**, which can give great predictions if there actually is a lot of replicable local structure in both the training and test data. But overfitting becomes a major issue.

## Missing Data

Decision trees are one of the few machine learning algorithms that adapt well to missing data.

They do this is two common ways:

- Include "missing" as a possible split point
- Use surrogate variables

The first is pretty self-explanatory. The second is pretty cool.

## Surrogate Variables

This idea exploits correlation between the features to replace missing features with non-missing ones.

Suppose $X_1$ is the most predictive feature at a given split of the tree, based on all the training observations for which $X_1$ is not missing. Then $X_1$ will be chosen.

But there may have been another variable $X_2$ which was almost as good as $X_1$ (and if so, is probably highly correlated with $X_1$).

Using $X_2$ as a *surrogate* for $X_1$ means that in training cases where $X_1$ is missing, $X_2$ is used as the splitting variable.

## Practical Application: Feature Discretization

Decision trees are pure "data mining"- there is no statistical/probabilistic model behind them.

It is perfectly acceptable to use decision trees in the feature preprocessing stage. One clever use is for *feature discretization*.

Suppose you have a particular feature $x$ (univariate). You plot $x$ vs the target (continuous or categorical), and see that there is a relationship between the two, but it's nonlinear.

We have seen that you can do a basis function expansion, essentially replacing $x$ by $h_1(x), h_2(x), \ldots$ for some chosen number of chosen basis functions.

## Practical Application: Feature Discretization

If you are trying to account for a univariate non-linear relationship between $x$ and $y$, one choice for the basis functions is

$$h_m(x) = I(k_{m-1} < x < k_m) = \begin{cases} 1 \text{ if } k_{m-1} < x < k_m \\ 0 \text{ else} \end{cases}$$

This represents a discretization of the continuous feature $x$ into ranges with *knots* $k_0, \ldots, k_m$.

This is also sometimes called *variable binning*.

## Practical Application: Feature Discretization

How to choose the number and location of the knots? Fit a single-variable decision tree!

Why not? At each split, there is only one candidate feature; the best split point is determined, and then the process is repeated recursively until one of the stopping criteria is reached.

This works well in practice, and it is simple to validate the quality of your basis function expansion, as each basis function only depends on one feature.

## Variance

One major problem with decision trees that is not easily overcome without modifying the algoritm is that these predictions have very high variance when compared to other approaches.

This is because of the local nature of the algorithm: if an error is made at one split, then this error will propagate down the tree for all subsequent splits.

Any test observation then will be being predicted using a noisy prediction function, which causes predictions to have high variance.

## Bagging

How do we reduce variance? Average!

A major advancement in overcoming this problem was the idea of fitting a bunch of trees to bootstrap resamples of the data, then averaging their predictions (continuous or soft classifications).

In particular, we have the following algorithm, called **Bagging** (**B**ootstrapped **Agg**regat**ing**).

## Bagging

*Algorithm: Bagging*.

1. Randomly sample $B$ training sets of size $N$, $(y_b, \mathbf{x}_b), b = 1 \ldots B$ **with replacement** from the original training set
2. Fit the model separately to each of these bootstrapped samples, obtaining prediction functions $f_b(\mathbf{x}), b = 1 \ldots B$
3. Average the predictions from these models to obtained the bagged prediction function,

$$f_{bag}(\mathbf{x}) = \frac{1}{B} \sum_{b=1}^{B} f_b(\mathbf{x})$$

## Bagging

For regression, one can view bagging as obtaining a sample from the distribution of the model predictions, and then taking your final prediction to be the mean of this distribution.

When will this be any different? The benefits of bagging are not apparent when using models that are globally optimal, like linear regression.

Trees are locally optimal, and different training sets could cause different splits to happen high up in the tree, changing all of the subsequent splits, and hence producing different trees for different bootstrapped training sets.

## Classifications

**Important**: when bagging *classification* models, you should average the *soft* probabilities, and then harden the result if you need a hard prediction.

If all you care about is getting hard predictions, a common strategy is the majority vote: harden the individual model predictions, then predict the class that got the most "votes" from the individual classifiers.

This will often give the same results as above.

But the proportion of models that predict $y_k = 1$ is not a good estimator of the *probability* that $y_k = 1$ (why?)

## The Formula On This Slide Is Not Correct

Bagging reduces the variance of the predictions, because

$$Var(f_{bag}(\mathbf{x})) = \frac{1}{B} Var(f_b(\mathbf{x})) \to 0 \text{ as } B \to \infty$$

so you can always lower the variance of your bagging estimator by increasing the number of bootstrapping samples.

. . . or can you? Is that formula correct?

## Correlation

Applying that variance formula assumes that predictions from the individual classifiers are *independent*. Since they were built on different combinations of the same training data, this doesn't seem reasonable

If the trees have positive pairwise correlation $\rho$ and each have common variance $\sigma^2$, then

$$Var(f_{bag}(\mathbf{x})) = \sigma^2\rho + \frac{1-\rho}{B}\sigma^2$$

This makes sense- if the trees are all perfectly correlated ($\rho = 1$), then bagging does nothing. The less correlated the better.

## Decorrelating

We need to decorrelate the trees somehow. What is causing the correlation?

- The same features are candidates for splits in each model, so the training sets have to be very different (which happens with low probablity) in order to produce different trees
- Training observations are shared between trees

## Random Forests

Solving the first problem yields a **Random Forest**.

Specifically, the bagging algorithm is modified as follows:

- At each split in each tree, randomly select $m < p$ features to use as candidates, and ignore the others.

This has the effect of greatly increasing the chances of changing a high-level split in a tree, which changes the whole rest of the tree.

The result is trees that are much less correlated.

## Tuning Parameters

The tuning parameters for random forests are the same as for a regular tree, with the addition of $m$, the number of features to consider at each split.

This has a substantial effect on the performance of these models.

Larger $m$ produced trees that are more correlated; smaller $m$ produces worse performing trees.

As usual, choose this via CV.

## OOB Samples

Modern Random Forest algorithms also make use of the notion of Out-Of-Bag samples:

- For each training observation, compute its final prediction by averaging the predictions only from models in which it was not used for training.

Because each model is built on a bootstrapped sample, none of them use the full dataset.

Random Forests do a kind of "automatic" cross-validation. The model can be fit in a sequence (over $b$), and training can be terminated when the OOB-error stabilizes.

## Overfitting

A common notion is that random forests can't possibly overfit.

What *is* true is that increasing $B$, the number of trees, can't cause the model to overfit.

However, when the number of variables is large, but many of them are noisy/unpredictive, the probability of considering a truly predictive variable at any split is low.

Each tree will overfit in this case, and therefore so will the forest.

## Overfitting

It is worth mentioning that even if there is a very large number of noisy, useless features present in the data, all that is required is that there be at least *some* truly predictive features, and a reasonable $m$.

All that is needed to ensure the individual trees aren't fitting to noise is to ensure that the chances of picking *at least one* predictive feature at each split are reasonably high.

Even though $m$ variables are considered at each split, only one is actually chosen.

## Computationally Independent Trees

Random forests combine the predictions of many indepedently (*computationally* independent, not *statistically* independent) prediction functions.

They are trivially parallelizable, in the sense that the algorithm requires no modification in order to be implemented in parallel.

For classification problems, could we do better by using the misclassifications from previous trees to inform future trees?

This is the idea behind **boosting**.

## Boosting

**Boosting** (ESL, chapter 10) is another technique for combining the predictions from many models in order to make better predictions overall.

While **bagging** builds many *strong* classifiers on different training sets, **boosting** builds a *sequence of weak classifiers*.

The whole training set is used to build each classifier in the sequence. However, observations are **weighted** according to whether the previous classifier classified them correctly or not.

In this way, later classifiers in the sequence focus more heavily on observations that are harder to classify correctly.

## Boosting

Specifically, suppose we label our training targets $y_i$ as $y_i \in \{-1, 1\}$ (instead of $0$ and $1$).

If the boosting procedure fits a sequence of models

$$f_1(\mathbf{x}), \ldots, f_M(\mathbf{x})$$

then the final classifications are computed as

$$f(\mathbf{x}) = \text{sign}\left(\sum_{m=1}^{M} \alpha_m f_m(\mathbf{x})\right)$$

where the $\alpha_m$ are weights that must be computed by the boosting procedure.

## It's a Basis Function Model

In this way, boosting can also be seen as a type of LBFM, where the basis functions are individual classifiers.

We don't find the $\alpha_m$ by least squares- they are defined by the boosting procedure.

It can be shown (ESL, section 10.4) that for the Adaboost algorithm we are about to discuss, the $\alpha_m$ are actually chosen to minimize a particular loss function.

## Adaboost

*Algorithm: Adaboost*:

```
1. Initialize weights $w_i = 1/n, i = 1 \dots n$
2. For $m = 1 \dots M$:
    1. Fit a decision tree classifier $f_m(\mathbf{x})$ to the weighted training data,
       where each $\mathbf{x}_i \to w_i \mathbf{x}_i$.
    2. Compute
       $$\epsilon_m = \frac{\sum_{i=1}^n w_i I(y_i \neq f_m(\mathbf{x}_i))}{\sum_{i=1}^n w_i}$$
    3. Compute
       $$\alpha_m = \log\left(\frac{1 - \epsilon_m}{\epsilon_m}\right)$$
    4. Set $w_i \leftarrow w_i \times \exp\left(\alpha_m \times I(y_i \neq f_m(\mathbf{x}_i))\right)$
```

## What it's Doing

Adaboost is quite clever. We defined our boosting classifer as

$$f(\mathbf{x}) = \text{sign}\left(\sum_{m=1}^{M} \alpha_m f_m(\mathbf{x})\right)$$

Imagine optimizing for this globally. We would need to find both the $f_m$ and the $\alpha_m$, which would amount to solving

$$(\alpha_m, f_m(\mathbf{x}))_{m=1}^{M} = \text{argmin} \sum_{i=1}^{n} L\left(y_i, \sum_{m=1}^{M} \alpha_m f_m(\mathbf{x})\right)$$

for some loss function $L$.

## Forward Stagewise Additive Modelling

Solving this simultaneously over $M$ is too hard. In many cases, though solving it for *each* $m$ will be easy.

This leads to the idea of **F**orward **S**tagewise **A**dditive **M**odelling (**FSAM**):

1. Initialize $f_0(\mathbf{x}) = 0$
2. For $m = 1$ to $M$:
    1. Compute
    $$(\alpha_m, f_m) = \text{argmin} \sum_{i=1}^{n} L(y_i, f_{m-1}(\mathbf{x}) + \alpha f(\mathbf{x}))$$
    2. Set
    $$f_m(\mathbf{x}) = f_{m-1}(\mathbf{x}) + \alpha_m f_m(\mathbf{x})$$

## Forward Stagewise Additive Modelling

It can be shown that

- The resulting algorithm approximates a solution to the full optimization problem stated above
- Adaboost is equivalent to the above algorthm, in the sense that there exists a sensible loss function for which the solution to the FSAM optimization problem for each $m$ are the $\alpha_m$ from Adaboost, and a decision tree classifier fit to the training data, weighted using the Adaboost weights

## Boosting in Practice

In practice, boosting works extremely well, producing classifiers that work well “out of the box”, and not tending to overfit.

You can subsample the training set at each iteration to make a “stochastic” version, with the usual tradeoff between stability and overfitting.

$M$ is a tuning parameter- but unlike the number of trees in a random forest, the stagewise nature of boosting makes it easy to choose $M$, by noticing that the model with $M + 1$ iterations first produces a model identical to what would have been produced with $M$ iterations, then just goes for one more iteration.

So in practice, just run the boosting algorithm until the validation error stabilizes.

## Boosting in Practice

One still needs to tune the trees as normal. Note a few differences:

- The size of the trees will typically need to be very small- one or two splits. Remember, boosting fits a series of *weak* classifiers