Tree-based Methods

Tree-based methods partition the feature space into a set of rectangles and then fit a simple model (like a constant) in each one.

simple regions

This results in a simple model, useful for interpretation.

These simple tree model does not provide much predictive accuracy.

Combining a large number of trees can often result in dramatic improvements in prediction accuracy at the expense of interpretation.

bagging, random forests, boosting, etc. (stacking)

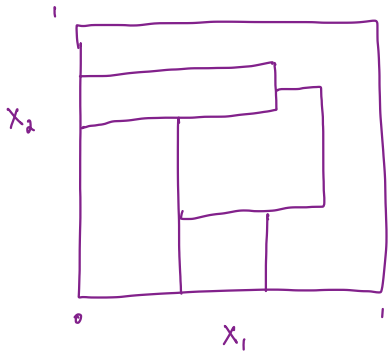
quantitative response y

categorical response.

Decision trees can be applied to both regression and classification problems. We will start with regression.

1 Decision Trees

Let's consider a regression problem with continuous response Y and inputs X_1 and X_2, each taking values in the unit interval.

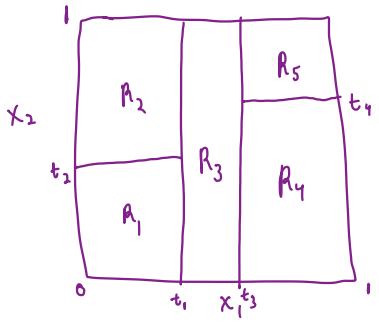


can partition w/ lines parallel to axes and model Y in each region as a constant.

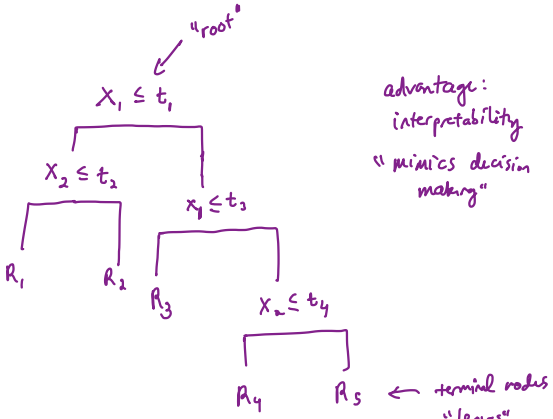
In each partition, we can model Y with a different constant. However, there is a problem:

Although each partition line has a simple description $X_1 = c$, resulting regions are hard to describe.

To simplify, we restrict attention to binary partitions.



$\Leftrightarrow$



advantage: interpretability "mimics decision making"

The result is a partition into five regions $R_1, \ldots, R_5$. The corresponding regression model predicts Y with a constant c_m in region R_m:

$$\hat{f}(\underline{x}) = \sum_{m=1}^{5} c_m \mathbb{I}\left((x_1, x_2) \in R_m\right)$$

1.1 Regression Trees

How should we grow a regression tree? Our data consists of p inputs for $i = 1, \ldots, n$. We need an automatic way to decide which variables to split on and where to split them.

Suppose we have a partition into M regions and we model the response as a constant in each region. We want a final model that "closely" fits our actual data.

If we use sum of squares to evaluate "closeness" and thus minimize as a criteria to choose our model,

$$\sum (y_i - \hat{f}(x_i))^2$$

The best $\hat{c}_m$ is the mean $\hat{c}_m = \frac{\sum y_i}{i : x_i \in R_m}$

Finding the best binary partition in terms of minimum sums of squares is generally computationally infeasible.

So we use a top-down, greedy approach called recursive binary splitting.

(1) Select the predictor and cutpoint s s.t. splitting the predictor space here leads to the greatest reduction in RSS.

Consider all possible half-planes $R_1(j,s) = \{X \mid X_j \leq s\}$ and $R_2(j,s) = \{X \mid X_j > s\}$. we seek j, s to minimize

$$\sum_{i : x_i \in R_1(j,s)} (y_i - \hat{c}_1)^2 + \sum_{i : x_i \in R_2(j,s)} (y_i - \hat{c}_2)^2$$

(2) Repeat process for next best combo of j, s, but instead of splitting the whole space, split $R_1(j,s)$ or $R_2(j,s)$ to minimize RSS.

(3) Continue until stopping criteria is met (i.e. no region contains more than 5 observations).

The process described above may produce good predictions on the training set, but is likely to overfit the data.

$y = f(x) + \varepsilon$

tree may be too complex, fitting "noise" rather than $f(x)$.

A smaller tree, with less splits might lead to lower variance and better interpretation at the cost of a little bias.

(Bad) idea: only split if results in "large enough" reduction in RSS

↳ seemingly worthless splits might be followed by good splits.

(Better idea)

A strategy is to grow a very large tree T_0 and then *prune* it back to obtain a *subtree*.

"cost complexity pruning"

consider a sequence of trees indexed a nonnegative (tuning) parameter α.

For each α, $\exists$ a corresponding subtree $T \subset T_0$ st.

$$\sum_{m=1}^{|T|} \sum_{x_i \in R_m} (y_i - \hat{c}_m)^2 + \alpha |T| \text{ is minimized.}$$

$|T|$: # terminal nodes.

α controls trade-off btw tree complexity & closeness of fit.

when $\alpha = 0$, $T = T_0$

$\alpha \uparrow \Rightarrow \uparrow$ price to "pay" for having many terminal nodes $\Rightarrow$ smaller tree.

Choose α via CV.

1.2 Classification Trees

If the target is a classification outcome $\tilde{y}$ taking values $1, 2, \ldots, K$, the only changes needed in the tree algorithm are the criteria for splitting, pruning, and c_m.

c_m:

Let $\hat{p}_{mk} = \frac{1}{n_m} \sum_{i: x_i \in R_m} I(y_i = k)$ = prop. of class k in R_m.

Then we classify obs. in node m to class $k(m) = \underset{k}{\operatorname{argmax}}\ \hat{p}_{mk}$. (majority class).

Node impurity (Splitting):

what we minimize to choose j, s:

misclassification error: $\frac{1}{n_m} \sum_{i \in R_m} I(y_i \neq k(m)) = 1 - \hat{p}_{mk(m)}$

result in purer terminal nodes

$\Rightarrow$ use for splitting.

- Gini Index: $\sum_{k \neq k'} \hat{p}_{mk} \hat{p}_{mk'} = \sum_{k=1}^{K} \hat{p}_{mk}(1 - \hat{p}_{mk})$.
- Deviance: $-\sum_{k=1}^{K} \hat{p}_{mk} \log \hat{p}_{mk}$

Pruning:

Can use any of above 3.

If prediction is goal of tree, typically use misclassification error.

2 Bagging

Decision trees suffer from *high variance*.

low bias.

Bootstrap aggregation or *bagging* is a general-purpose procedure for reducing the variance of a statistical learning method, particularly useful for trees.

Recall for a set of indep. r.v.'s $Z_1, \dots, Z_n$ each w/ variance σ^2

$$\text{Var}(\bar{Z}) = \frac{\sigma^2}{n}$$

So a natural way to reduce the variance is to take many training sets from the population, build a separate prediction model using each training set, and average the resulting predictions.

i.e. take B training data sets

Calculate $\hat{f}^1(x), \dots, \hat{f}^B(x)$

Obtain low-variance model:

$$\hat{f}_{AVG}(x) = \frac{1}{B} \sum_{b=1}^{B} \hat{f}^b(x).$$

Of course, this is not practical because we generally do not have access to multiple training sets.

get B bootstrapped data sets,

Fit our model on b^{th} bootstrapped data set to get $\hat{f}^{*(b)}(x)$ and average:

$$\hat{f}_{bag}(x) = \frac{1}{B} \sum_{i=1}^{B} \hat{f}^{*(b)}(x)$$

While bagging can improve predictions for many regression methods, it's particularly useful for decision trees.

can get high variance, low bias decision trees!

These trees are grown deep and not pruned.

can use many trees, won't lead to overfitting.

How can bagging be extended to a classification problem? (averaging no longer an option)

most common: majority vote of classes predicted by each tree in ensemble.

(usually better)
average probability estimates.

2.1 Out-of-Bag Error

There is a very straightforward way to estimate the test error of a bagged model.

2.2 Interpretation

"Importance plot"

↳ total reduction in RSS (or Gini) due to splits for a given predictor.

3 Random Forests

Random forests provide an improvement over bagged trees by a small tweak that decorrelates the trees.

As with bagged trees, we build a number of decision trees on bootstrapped training samples.

But when we build the trees, a random sample of m predictors is chosen as split candidates from the full set of predictors.

fresh sample of predictors taken at each split.

typically $m \approx \sqrt{p}$

In other words, in building a random forest, at each split in the tree, the algorithm is not allowed to consider a majority of the predictors.

The main difference between bagging and random forests is the choice of predictor subset size m. if $m=p$ => random forests = bagging.

4 Boosting

The basic idea of *boosting* is to take a simple (and poorly performing form of) predictor and by sequentially modifying/perturbing it and re-weighting (or modifying) the training data set, to creep toward an effective predictor.

"slow" learning. $L_{01}(\hat{y}, y) = \sum_{i=1}^{n} \mathbb{I}(y_i \neq \hat{y}_i)$.

Consider a 2-class 0-1 loss classification problem. We'll suppose that output y takes values in $\mathcal{G} = \{-1, 1\}$. The AdaBoost.M1 algorithm is built on some base classifier form.

f, this can be almost any classifier
works but w/ low variance
most people use w/ a tree w/ 2 terminal nodes "stubs"

Algorithm (AdaBoost.M1) $(\underline{x}_i, y_i)$ $i = 1, \ldots, n$

1. Initialize the weights on the training data.

$$w_{i1} = \frac{1}{n}, \; i = 1, \ldots, n$$

2. Fit a $\mathcal{G}$-valued predictor/classifier $\hat{f}_1$ (like a stub) to the training data to optimize the 0-1 loss.

Let $\overline{err}_1 = \frac{1}{n} \sum_{i=1}^{n} \mathbb{I}(y_i \neq \hat{f}_1(\underline{x}_i))$ $\quad \sum_{i=1}^{n} \mathbb{I}(y_i \neq \hat{y}_i)$.

$$\alpha_1 = \ln\left(\frac{1 - \overline{err}_1}{\overline{err}_1}\right)$$

3. Set new weights on the training data.

$$w_{i2} = \frac{1}{n} \exp\left(\alpha_1 \mathbb{I}(y_i \neq \hat{f}_1(\underline{x}_i))\right) \quad i = 1, \ldots, n$$

↑ this upweights misclassified observations by a factor of $\frac{1 - \overline{err}_1}{\overline{err}_1}$

4. For $m = 2, \ldots, M$,

a. Fit a $\mathcal{G}$-valued classifier $\hat{f}_m$ to the training data to minimize $\sum_{i=1}^{n} w_{im} \mathbb{I}(y_i \neq \hat{f}_m(\underline{x}_i))$.

b. Let $\overline{err}_m = \frac{1}{\sum_{i=1}^{n} w_{im}} \sum_{i=1}^{n} w_{im} \mathbb{I}(y_i \neq \hat{f}_m(\underline{x}_i))$

c. Set $\alpha_m = \ln\left(\frac{1 - \overline{err}_m}{\overline{err}_m}\right)$

d. Update weights as

$$w_{i(m+1)} = w_{im} \exp\left(\alpha_m \mathbb{I}(y_i \neq \hat{f}_m(\underline{x}_i))\right) \quad i = 1, \ldots, n$$

AdaBoost can be adapted for regression problems w/ different loss function (which leads to different of weights, errors, etc).

5. Output an updated classifier based on "weighted voting".

$$\hat{f}(\underline{x}) = \text{sign}\left(\sum_{m=1}^{M} \alpha_m \hat{f}_m(\underline{x})\right)$$

classifiers w/ small $\overline{err}_m$ get big positive weights in the voting.

This works well!

4.1 Why might this work?

For g an arbitrary function of $\boldsymbol{x}$, consider a classifier built using g as a voting function, e.g. $f(\boldsymbol{x}) = \text{sign}(g(\boldsymbol{x}))$, ignoring the possibility that $g(\boldsymbol{x}) = 0$. Then

$$\mathbb{I}(y \neq \hat{y}) = \mathbb{I}(yg(\boldsymbol{x}) < 0).$$

Using the following fact,

$$\mathbb{I}(u < 0) \leq \exp(-u) \; \forall u,$$

provided $P(g(\boldsymbol{X}) = 0) = 0$, the 0-1 loss error rate for $f(\boldsymbol{x})$ is

$$\text{E}[\mathbb{I}(Y \neq \hat{Y})] = \text{E}[\mathbb{I}(Yg(\boldsymbol{X}) < 0)] \leq \text{E}[\exp(-Yg(\boldsymbol{X})].$$

In other words, the error rate is bounded above by expected exponential loss. AdaBoost works by **providing a voting function that produces a small value of this bound.**

To see this, we need to identify for each $\boldsymbol{u}$ a value a that optimizes $\text{E}\left[\exp(-aY)|\boldsymbol{X} = \boldsymbol{u}\right]$, where

$$\text{E}\left[\exp(-aY)|\boldsymbol{X} = \boldsymbol{u}\right] = \exp(-a)P\left[Y = 1|\boldsymbol{X} = \boldsymbol{u}\right] + \exp(a)P\left[Y = -1|\boldsymbol{X} = \boldsymbol{u}\right].$$

An optimal a is easily seen to be half the log odds ratio, i.e. the g optimizing the upper bound is

$$g\left(\boldsymbol{u}\right) = \frac{1}{2}\ln\left(\frac{P\left[yY = 1|\boldsymbol{X} = \boldsymbol{u}\right]}{P\left[Y = -1|\boldsymbol{X} = \boldsymbol{u}\right]}\right).$$

Now consider "base classifiers" $h_\ell\left(\boldsymbol{x}, \boldsymbol{\gamma}_\ell\right)$ taking values in $\mathcal{G} = \{-1, 1\}$ with parameters $\boldsymbol{\gamma}_l$ and functions built from them of the form

$$g_m\left(\boldsymbol{x}\right) = \sum_{l=1}^{m} \beta_\ell h_\ell\left(\boldsymbol{x}, \boldsymbol{\gamma}_\ell\right).$$

for training-data-dependent β_l and $\boldsymbol{\gamma}_l$.

Then, $g_m\left(\boldsymbol{x}\right) = g_{m-1}\left(\boldsymbol{x}\right) + \beta_m h_m\left(\boldsymbol{x}, \boldsymbol{\gamma}_m\right)$. Thus, successive g's are perturbations of the previous ones.

How can we define the perturbations to produce small values of the upper bound of our error ($\mathrm{E}[\exp(-Yg(\boldsymbol{X})]$)?

Well, we don't have a complete probability model for $(\boldsymbol{X}, Y)$ (if we did, we would be done). So, let's optmize an empirical version of this bound.

$$\begin{aligned} E_m &= \sum_{i=1}^{n} \exp(-y_i g_m(\boldsymbol{x}_i)) \qquad \text{(Now based on tr} \\ &= \sum_{i=1}^{n} \exp(-y_i g_{m-1}(\boldsymbol{x}_i) - y_i \beta_m h_m(\boldsymbol{x}_i, \boldsymbol{\gamma}_m)) \\ &= \sum_{i=1}^{n} \exp(-y_i g_{m-1}(\boldsymbol{x}_i)) \exp(-y_i \beta_m h_m(\boldsymbol{x}, \boldsymbol{\gamma}_m)), \end{aligned}$$

and let's call $v_{im} = \exp(-y_i g_{m-1}(\boldsymbol{x}_i))$.

We will consider optimal choice of $\boldsymbol{\gamma}_m$ and $\beta_m > 0$ for purposes of making g_m the best possible perturbation of g_{m-1} in terms of minimizing E_m.

1. Choice of $\boldsymbol{\gamma}_m$:

$$\begin{aligned} E_m &= \sum_{\substack{i \text{ with} \\ h_m(\boldsymbol{x}_i, \boldsymbol{\gamma}_m) = y_i}} v_{im} \exp(-\beta_m) + \sum_{\substack{i \text{ with} \\ h_m(\boldsymbol{x}_i, \boldsymbol{\gamma}_m) \neq y_i}} v_{im} \exp(\beta_m) \\ &= (\exp(\beta_m) - \exp(-\beta_m)) \sum_{i=1}^{n} v_{im} I\left[h_m(\boldsymbol{x}_i, \boldsymbol{\gamma}_m) \neq y_i\right] + \exp(-\beta_m) \sum_{i=1}^{n} v_{im} \end{aligned}$$

   Independentof β_m we need $\boldsymbol{\gamma}_m$ to minimize the v_{im}-weighted error rate of $h_m(\boldsymbol{x}, \boldsymbol{\gamma}_m)$. Call the optimized version $h_m(\boldsymbol{x})$. **This is the same as step 4a. in AdaBoost.m1.**

2. Choice of β_m:

$$\begin{aligned} E_m &= \exp(-\beta_m) \left(\sum_{\substack{i \text{ with} \\ h_m(\boldsymbol{x}_i, \boldsymbol{\gamma}_m) = y_i}} v_{im} + \sum_{\substack{i \text{ with} \\ h_m(\boldsymbol{x}_i, \boldsymbol{\gamma}_m) \neq y_i}} v_{im} \exp(2\beta_m) \right) \\ &= \exp(-\beta_m) \left(\sum_{i=1}^{n} v_{im} + \sum_{i=1}^{n} v_{im} \left(\exp(2\beta_m) - 1\right) I\left[h_m(\boldsymbol{x}_i) \neq y_i\right] \right) \end{aligned}$$

and minimization of E_m is equivalent to minimization of

$$\exp(-\beta_m)\left(1+(\exp(2\beta_m)-1)\frac{\sum_{i=1}^N v_{im}I\left[h_m\left(\boldsymbol{x}_i\right)\neq y_i\right]}{\sum_{i=1}^N v_{im}}\right).$$

Let

$$\overline{\text{err}}_m^{h_m}=\frac{\sum_{i=1}^n v_{im}I\left[h_m\left(\boldsymbol{x}_i\right)\neq y_i\right]}{\sum_{i=1}^n v_{im}},$$

then a bit of calculus shows that the optimizing β_m is

$$\beta_m=\frac{1}{2}\ln\left(\frac{1-\overline{\text{err}}_m^{h_m}}{\overline{\text{err}}_m^{h_m}}\right).$$

Notice this coefficient is **exactly $\frac{\alpha_m}{2}$ from step 4b. and 4c. in AdaBoost.m1 (and the $\frac{1}{2}$ is irrelevant for the sign).

3. Updating weights v_{im}:

   Note that

$$\begin{aligned}
v_{i(m+1)} &= \exp(-y_i g_m\left(\boldsymbol{x}_i\right)) \\
&= \exp(-y_i\left(g_{m-1}\left(\boldsymbol{x}_i\right)+\beta_m h_m\left(\boldsymbol{x}_i\right)\right)) \\
&= v_{im}\exp(-y_i\beta_m h_m\left(\boldsymbol{x}_i\right)) \\
&= v_{im}\exp(\beta_m\left(2I\left[h_m\left(\boldsymbol{x}_i\right)\neq y_i\right]-1\right)) \\
&= v_{im}\exp(2\beta_m I\left[h_m\left(\boldsymbol{x}_i\right)\neq y_i\right])\exp(-\beta_m).
\end{aligned}$$

   Since $\exp(-\beta_m)$ is constant across i, it is irrelevant to weighting, and since the prescription for β_m produces half what AdaBoost prescribes in 4b. for α_m, the weights used in the choice of β_{m+1} and $h_{m+1}\left(\boldsymbol{x},\boldsymbol{\gamma}_{m+1}\right)$ are exactly as in AdaBoost. Since g_1 corresponds to the first AdaBoost step, g_M is $1/2$ of the AdaBoost voting function and the g_m's generate the same classifier as the AdaBoost algorithm.

So, in conclusion, we have found g_M (a positive multiple of the AdaBoost voting function) which optimizes an empirical version of $\mathrm{E}\exp(-Yg(\boldsymbol{X}))$, the upper bound on our error rate!