CART – Classification and Regression Trees

Trees can be viewed as *basis expansions of simple functions*

$$f(x) = \sum_{m=1}^{M} c_m 1(x \in R_m)$$

with $R_1, \ldots, R_m \subseteq \mathbb{R}^p$ disjoint.

The *CART* algorithm is a heuristic, adaptive algorithm for *basis function selection.*

A recursive, binary partition (a tree) is given by a *list of splits*

$$\{(t_{01}), (t_{11}, t_{12}), (t_{21}, t_{22}, t_{23}, t_{24}), \ldots, (t_{n1}, \ldots, t_{n2^n})\}$$

and corresponding split variable indices

$$\{(i_{01}), (i_{11}, i_{12}), (i_{21}, i_{22}, i_{23}, i_{24}), \ldots, (i_{n1}, \ldots, i_{n2^n})\}$$
$$R_1 = (x_{i_{01}} < t_{01}) \cap (x_{i_{11}} < t_{11}) \cap \ldots \cap (x_{i_{n1}} < t_{n1})$$

and we can determine if $x \in R_1$ in n steps $\ll M = 2^n$.

All the remaining sets in the partition corresponding to the 2^n leafs are determined similarly and recursively. It is by far easier to draw a picture of the corresponding tree than to write down the precise mathematical recursion in terms of indices. A point is that the algorithmic complexity of determining which of the 2^n sets in the partition an x belongs to scales with n. Thus even for extremely large partitions we can very rapidly determine where a concrete x belongs.

In practice not all of the 2^n partitions need to be present. The tree can be "pruned" so that some of the leaf are not at depth n.

Figure 9.2 – Recursive Binary Partitions

The *recursive* partition of $[0, 1]^2$ above and the representation of the partition by a tree.

A binary tree of *depth* n can represent up to 2^n partitions/basis functions.

We can determine which R_j an x belongs to by n recursive yes/no questions.

Figure 9.2 – General Partitions

A general partition that can not be represented as binary splits.

With M sets in a general partition we would in general need of the order M yes/no questions to determine which of the sets an x belongs to.

Figure 9.2 – Recursive Binary Partitions

For a *fixed* partition $R_1, \ldots, R_M$ the least squares estimates are

$$\hat{c}_m = \bar{y}(R_m) = \frac{1}{N_m} \sum_{i: x_i \in R_m} y_i$$

$N_m = |\{i \mid x_i \in R_m\}$.

The recursive partion allows for rapid computation of the estimates and rapid predition of new observations.

Greedy Splitting Algorithm

With *squared error loss* and an unknown partition $R_1, \ldots, R_M$ we would seek to minimize

$$\sum_{i=1}^{N} (y_i - \bar{y}(R_{m(i)}))^2$$

over the possible binary, recursive partitions. But this is computationally difficult.

An optimal single split on a region R is determined by

$$\min_j \underbrace{\min_s \left(\sum_{i:x_i \in R(j,s)} (y_i - \bar{y}(R(j,s)))^2 + \sum_{i:x_i \in R(j,s)^c} (y_i - \bar{y}(R(j,s)^c))^2 \right)}_{\text{univariate optimization problem}}$$

with $R(j,s) = \{x \in R \mid x_j < s\}$ The *tree growing algorithm* recursively does single, optimal splits on each of the partitions obtained in the previous step.

Note that the complements above are taken within the region R, that is, $R(j,s)^c = R \backslash R(j,s)$.

Tree Pruning

The full binary tree, T_0, representing the partitions $R_1, \ldots, R_M$ with $M = 2^n$ may be too large. We *prune* it by snipping of leafs or subtrees.

For any subtree T of T_0 with $|T|$ leafs and partition $R_1(T), \ldots, R_{|T|}(T)$ the *cost-complexity* of T is

$$C_\alpha(T) = \sum_{i=1}^{N} (y_i - \bar{y}(R_{m(i)}(T)))^2 + \alpha|T|.$$

Theorem 1. *There is a finite set of subtrees* $T_0 \supseteq T_{\alpha_1} \supset T_{\alpha_2} \supset \ldots \supset T_{\alpha_r}$ *with* $0 \leq \alpha_1 < \alpha_2 < \ldots < \alpha_r$ *such that* T_{α_i} *minimizes* $C_\alpha(T)$ *for* $\alpha \in [\alpha_i, \alpha_{i+1})$

A proof of the theorem above can be found in Section 7.2 in *Pattern Recognition and Neural Networks* by Brian D. Ripley.

The practical consequence of the theorem above is that tree algorithms find this sequence of pruned trees in the estimation process and then the choice of α can be determined by validation or cross-validation. We do not need to consider other choices of α than those corresponding to the precomputed pruned trees.

Node Impurities and Classification Trees

Define the *node impurity* as the average loss for the node R

$$Q(R) = \frac{1}{N(R)} \sum_{i:x_i \in R} (y_i - \bar{y}(R))^2$$

The greedy split of R is found by

$$\min_j \min_s \left(N(R(j,s))Q(R(j,s)) + N(R(j,s)^c)Q(R(j,s)^c) \right)$$

with $R(j,s) = \{x \in R \mid x_j < s\}$ and we have

$$C_\alpha(T) = \sum_{m=1}^{|T|} N(R_m(T))Q(R_m(T)) + \alpha|T|.$$

If Y takes K discrete values we focus on the *node estimate* for $R_m(T)$ in tree T as being

$$\hat{p}_m(T)(k) = \frac{1}{N_m} \sum_{i:x_i \in R_m(T)} 1(y_i = k)$$

Node Impurities and Classification Trees

The loss functions for classification enter in the specification of the *node impurities* used for splitting an cost-complexity computations.

Examples of

- *0-1 loss* gives *misclassification error* impurity:

$$Q(R_m(T)) = 1 - \max\{\hat{p}(R_m(T))(1), \ldots, \hat{p}(R_m(T))(K)\}$$

- *likelihood loss* gives *entropy* impurity:

$$Q(R_m(T)) = -\sum_{k=1}^{K} \hat{p}(R_m(T))(k) \log \hat{p}(R_m(T))(k)$$

- The *Gini index* impurity:

$$Q(R_m(T)) = \sum_{k=1}^{K} \hat{p}(R_m(T))(k)(1 - \hat{p}(R_m(T))(k))$$

Note that it is certainly not always the case that all K different cases are present in a single set R_m, hence some of the estimated conditional probabilities are 0 and we have to remember that $0 \log 0 = 0$ yields a continuous extension of $p \log p$ in 0.

The *Gini index* can be thought of in the following way: With the 0-1 loss ...

Figure 9.3 – Node Impurities

Spam Example

Spam Example

Spensitivity and Specificity

The *sensitivity* is the probability of predicting 1 given that the true value is 1 (predict a case given that there is a case).

$$\begin{aligned} \text{sensitivity} &= \Pr(f(X) = 1 | Y = 1) \\ &= \frac{\Pr(Y = 1, f(X) = 1)}{\Pr(Y = 1, f(X) = 1) + \Pr(Y = 1, f(X) = 0)} \end{aligned}$$

The *specificity* is the probability of predicting 0 given that the true value is 0 (predict that there is no case given that there is no case).

$$\begin{aligned} \text{specificity} &= \Pr(f(X) = 0 | Y = 0) \\ &= \frac{\Pr(Y = 0, f(X) = 0)}{\Pr(Y = 0, f(X) = 0) + \Pr(Y = 0, f(X) = 1)} \end{aligned}$$

ROC curves

The *reciever operating characteristic* or ROC curve.

Ensembles of Weak Predictors

A *weak predictor* is a predictor that performs only a little better than random guessing.

With an ensemble or collection of weak predictors $\hat{f}_1, \ldots, \hat{f}_B$ we seek to combine their predictions, e.g. as

$$\hat{f}^B = \frac{1}{B} \sum_{b=1}^{B} \hat{f}_b$$

hoping to improve performance.

Bootstrap aggregation or *Bagging* is an example where the ensemble of preditors are obtained by estimation of the predictor on bootstrapped data sets.

Bagging is treated in Section 8.7 in the book. Note the "weak predictor" is often taken to mean "simple predictor" where again simple refers to a predictor with few parameters, which has low variance and besides from special cases considerable bias. There is, however, nothing that prevent us from considering ensembles of weak predictors where "weak" refers to high variance as opposed to high bias.

Combining Weak Predictors

Recall that

$$V(\hat{f}^B(x)) = \frac{1}{B^2}\sum_{b=1}^{B} V(\hat{f}_b(x)) + \frac{1}{B^2}\sum_{b\neq b'} \mathrm{cov}(\hat{f}_b(x), \hat{f}_b'(x))$$

hence bagging can be improved if the preditors can be "de-correlated".

Random Forests (Chapter 15) is a modification of bagging for trees where the "bagged trees" are de-correlated.

The problem of ensemble learning is broken down into the

- The selection of *base learners.*
- The combination of the *base learners.*

Trees as Ensemble Learners

The *basis expansion techniques* can be seen as ensemble learning (with or without regularization) where we have specified the base learners a priori.

For trees we build and combine sequentially and recursively the *simplest* base learners; the stumps or single splits.

Are there general ways to search *the space of learners and combinations of simple learners*?

Stagewise Additive Modeling

With $b(\cdot, \gamma)$ for $\gamma \in \Gamma$ a parameterized family of *basis functions* we can seek expansions of the form

$$\sum_{m=1}^{M} \beta_m b(x, \gamma_m)$$

With fixed γ_m this is standard, with unrestricted γ_m this is in general very difficult numerically.

Suggetion: Evolve the expansions in *stages* where (β_m, γ_m) are estimated in step m *and then fixed forever.*

Boosting

With any loss function L the *Forward Stagewise Additive Model* is estimated by the algorithm:

1. Set $m = 1$ and initialize with $\hat{f}_0(x) = 0$.
2. Compute
$$(\hat{\beta}_m, \hat{\gamma}_m) = \operatorname*{argmin}_{\beta,\gamma} \sum_{i=1}^{N} L(y_i, \hat{f}_{m-1}(x_i) + \beta b(x_i, \gamma))$$
3. Set $f_m = f_{m-1} + \hat{\beta}_m b(\cdot, \hat{\gamma}_m)$, $m = m + 1$ and return to 2.

Note that with squared error loss

$$L(y_i, \hat{f}_{m-1}(x_i) + \beta b(x_i, \gamma)) = ((y_i - \hat{f}_{m-1}(x_i)) - \beta b(x_i, \gamma))^2$$

every estimation step is a *reestimation on the residuals.*

Base Classifiers

With $Y \in \{-1, 1\}$ and any classifier $G(x) \in \{-1, 1\}$ the *misclassification error* is

$$\operatorname{err}(G) = \frac{1}{N} \sum_{i=1}^{N} 1(y_i \neq G(x_i)) = \frac{1}{2N} \sum_{i=1}^{N} (1 - y_i G(x_i))$$

With $\mathcal{G}$ a *class of classifiers* the (unweigted) optimal classifier is

$$\hat{G} = \operatorname*{argmin}_{G \in \mathcal{G}} \operatorname{err}(G)$$

With $w_1, \ldots, w_N \geq 0$ the weighted optimal classifier is

$$\hat{G} = \operatorname*{argmin}_{G \in \mathcal{G}} \sum_{i=1}^{N} w_i 1(y_i \neq G(x_i))$$

A simple class of classifiers is the class of stumps – trees with only two leafs. A stump is given simply by a pair (i, t) of the splitting variable and the split point, and if we use the misclassification node impurity the optimization over (i, t) is precisely the optimization we carry out when we make each greedy step in the algorithm for estimation of trees.

Surrogate Loss Functions

Most important property of the surrogate loss functions is that they are convexifications of the 0-1-loss.

AdaBoost – Classification with Exponential Loss

With *exponential loss* $L(y, f(x)) = \exp(-yf(x))$ and with $w_i^{(m)} = \exp(-y_i f_{m-1}(x_i))$

$$\begin{aligned}\sum_{i=1}^N L(y_i, \hat{f}_{m-1}(x_i) + \beta G(x_i)) &= \sum_{i=1}^N w_i^{(m)} \exp(-y_i \beta G(x_i)) \\ &= (e^{\beta} - e^{-\beta}) \sum_{i=1}^N w_i^{(m)} 1(y_i \neq G(x_i)) + e^{-\beta} \sum_{i=1}^N w_i^{(m)}\end{aligned}$$

The minimizer is $\hat{G}_m = \text{argmin}_{G \in \mathcal{G}} \sum_{i=1}^N w_i^{(m)} 1(y_i \neq G(x_i))$,

$$\hat{\beta}_m = \frac{1}{2} \log \frac{1 - \text{err}_m}{\text{err}_m}$$

The updated *weights* in step $m+1$ are

$$\begin{aligned}w_i^{(m+1)} &= w_i^{(m)} \exp(-y_i \hat{\beta}_m \hat{G}_m(x_i)) \\ &= w_i^{(m)} \exp(2\hat{\beta}_m 1(y_i \neq \hat{G}_m(x_i))) \exp(-\hat{\beta}_m).\end{aligned}$$

The weights in the AdaBoost algorithm evolve over time multiplying larger weights in the m'th step on those pairs (x_i, y_i) that are misclassified by the classifier in the m'th step.

Figure 10.1 – Schematic AdaBoost

$$G(x) = \sum_{m=1}^M \alpha_m G_m(x)$$

1. Initialize with weights $w_i = 1/N$ and set $m = 1$ and fix M.
2. Fit a classifier G_m using weights w_i.
3. Recompute weights as

   $$w_i \leftarrow w_i \exp(\alpha_m 1(y_i \neq G_m(x_i)))$$

   where $\alpha_m = \log((1 - \text{err}_m)/\text{err}_m)$ and

   $$\text{err}_m = \frac{1}{\sum_{i=1}^N w_i} \sum_{i=1}^N w_i 1(y_i \neq G(x_i)).$$

4. Stop if $m = M$ or set $m \to m + 1$ and return to 2

Figure 10.2 and 10.3

Boosting using stumps only can outperform even large trees in terms of test error (simulation).

Even when the misclassification error is 0 on the training data it can pay to continue the boosting and the exponential loss will continue to decrease.

More Boosting

The computational problem in boosting is minimization of

$$\sum_{i=1}^{N} L(y_i, \hat{f}_{m-1}(x_i) + \beta b(x_i, \gamma)).$$

For classification with *exponential loss* this simplifies to *weighted* optimal classification.

For regression and *squared error loss* this is re-estimation based on the residuals.
With the notation

$$L(\mathbf{f}) = \sum_{i=1}^{N} L(y_i, f_i)$$

for $\mathbf{f} = (f_1, \ldots, f_N) \in \mathbb{R}^N$ we aim at finding *steps* $\mathbf{h}_1, \ldots, \mathbf{h}_M$ and with $\mathbf{h}_0$ the initial guess an approximate minimizer of the form

$$\mathbf{f}_M = \sum_{m=0}^{M} \mathbf{h}_m.$$

Gradient Boosting

The gradient of $L : \mathbb{R}^N \to \mathbb{R}$ is

$$\nabla L(\mathbf{f}) = (\partial_z L(y_1, f_1), \ldots, \partial_z L(y_N, f_N))^T$$

Gradient descent algorithms suggest steps from $\mathbf{f}_m$ in the direction of $-\nabla L(\mathbf{f}_m)$;

$$\mathbf{h}_m = -\rho_m \nabla L(\mathbf{f}_m).$$

Problem: $-\rho_m \nabla L(\mathbf{f}_m)$ is most likely *not* obtainable as a prediction within the class of *base learners* – it is *not* of the form $\beta(b(x_1, \gamma), \ldots, b(x_N, \gamma))^T$.

Solution: Fit a *base learner* $\hat{\mathbf{h}}_m$ to $-\nabla L(\mathbf{f}_m)$ and compute by iteration the expansion

$$\hat{f}_M = \sum_{m=0}^{M} \rho_m \hat{h}_m.$$

This is *gradient boosting* as implemented in the `mboost` library.

More information on gradient boosting and the `mboost` R-package can be found in: Peter Bühlmann and Torsten Hothorn. *Boosting Algorithms: Regularization, Prediction and Model Fitting.* Statistical Science 2007, Vol. 22, No. 4, 477-505. There is also an interesting discussion following the paper.