MATH 680 Computation Intensive Statistics

October 2, 2019

Regression Tree and Gradient Boosting

Contents

1 Tree-based method

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

1.1 Background

Example: regression problem

- Y: Continuous response.
- $X = (X_1, X_2)$: values in the unit interval.

Recursive binary partitions

1. Split the feature space into two regions R_1 and R_2:

$$\hat{Y}(x \in R_1) = ave(y_i | x_i \in R_1)$$

$$\hat{Y}(x \in R_2) = ave(y_i | x_i \in R_2)$$

choose the variable and splitting point to achieve the best fit.

2. Then R_1 and R_2 are split into two more regions. And this process continues until some stopping rule applied.

For example

- First split $X_1 = t_1$.
- $X_1 \leq t_1$ is split at $X_2 = t_2$.
- $X_1 > t_1$ is split at $X_1 = t_3$.
- $X_1 > t_3$ is split at $X_2 = t_4$.

There are five partitions $R_1, R_2, \ldots, R_5$. The corresponding regression model predicts Y with a constant c_m in regression R_m

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

The model can be represented by the binary tree.

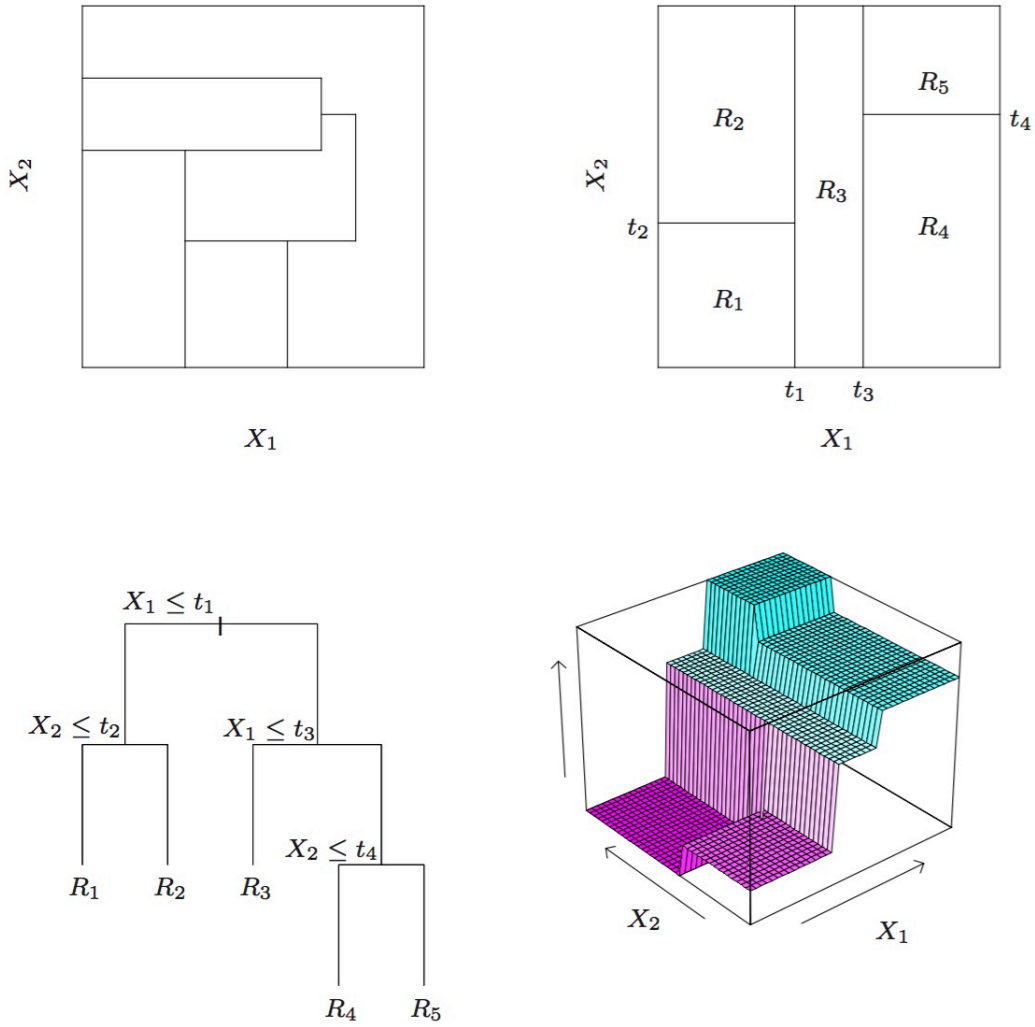


Figure 1: Top right panel shows a partition of a two-dimensional feature space by recursive binary splitting, applied to some fake data. Top left panel shows a general partition that cannot be obtained from recursive binary splitting. Bottom left panel shows the tree corresponding to the partition in the top right panel, and a perspective plot of the prediction surface appears in the bottom right panel.

1.2 Regression Trees

How to grow a regression tree?

Data: $\{(x_i, y_i)\}_{i=1}^N$ with $x_i = (x_{i1}, \ldots, x_{ip})$. We need to address three issues:

1. Constant fit
2. Splitting variable
3. Splitting point

(1) Constant fit. Suppose we have M regions $R_1, R_2, \ldots, R_M$. Model the response as a constant c_m in each region:

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

If we adopt the sum of squares

$$\sum (y_i - f(x_i))^2$$

as our minimization criterion. We see that the best $\hat{c}_m$ is just

$$\hat{c}_m = ave(y_i | x_i \in R_m)$$

(2) and (3) Splitting variable and splitting point. Starting with all of the data. Splitting variable j and splitting point s. Define the pair of half-planes

$$R_1(j, s) = \{x | x_j \leq s\}$$

$$R_2(j, s) = \{x | x_j > s\}$$

we solve

$$\min_{j,s} \left[\min_{c_1} \sum_{x_i \in R_1(j,s)} (y_i - c_1)^2 + \min_{c_2} \sum_{x_i \in R_2(j,s)} (y_i - c_2)^2 \right]$$

for any choice of j and s.

For fixed j and s, the inner minimization has solutions $\hat{c}_1 = ave(y_i | x_i \in R_1)$ and $\hat{c}_2 = ave(y_i | x_i \in R_2)$. For each j, the determination of s can be done very quickly:

Algorithm 1: Algorithm for finding the optimal (j, s) pair.

for $j = 1, \ldots, p$ **do**
  for $s = s_1, \ldots, s_K$ **do**
    Compute

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

  end
end
Scan through all pairs of (j, s) and get $\min_{j,s} v(s, j)$;

Having found the best split (j, s), we partition the data into the two resulting regions R_1 and R_2 and repeat the process on R_1 and R_2.

How large should we grow a tree?

- Too large – overfitting
- Too small – cannot capture the important structure.

Strategy: back-pruning

- Grow a large tree T_0.
- Then prune T_0 using cost-complexity pruning.

Cost-complexity pruning

- Define subtree $T \subset T_0$ as any tree obtained by pruning T_0.
- Define terminal region as R_m.
- Define $|T|$ the number of terminal nodes in T.
- $N_m = \#\{x_i \in R_m\}$.
- $\hat{c}_m = \frac{1}{N_m} \sum_{x_i \in R_m} y_i$.
- $Q_m(T) = \frac{1}{N_m} \sum_{x_i \in R_m} (y_i - \hat{c}_m)^2$

Define the **cost-complexity criterion** as

$$C_\alpha(T) = \sum_{m=1}^{|T|} N_m Q_m(T) + \alpha |T|.$$

Here α controls the trade-off between tree size and its goodness-of-fit to the data.

- Large α – small tree.
- Small α – large tree.
- $\alpha = 0$ – full tree.

For a fixed α, one can show that there is a unique smallest subtree $T_\alpha \subset T_0$ to minimize $C_\alpha(T)$. To find T_α, we successively collapse the internal nodes that produce the smallest per-node increase in $\sum_m N_m Q_m(T)$, i.e. given the current tree $T^{(k)}$, the collapse subtree $T^{(k+1)}$ must satisfies

$$T^{(k+1)} = \arg\min_{T \subset T^{(k)}} \Delta(T|T^{(k)})$$

where

$$\Delta(T|T^{(k)}) = \sum_m N_m Q_m(T) - \sum_m N_m Q_m(T^{(k)})$$

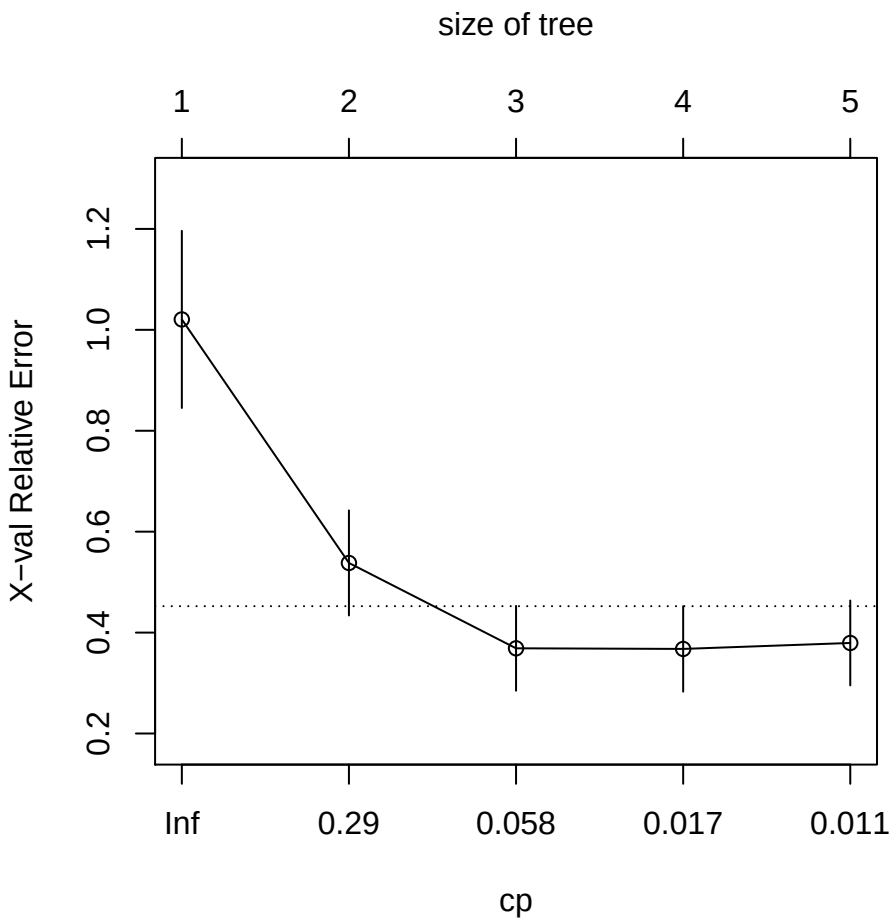
Those give a finite sequence of subtrees $T^{(K)} \subset, \ldots, \subset T^{(2)} \subset T^{(1)} \subset T_0$. One can show that this sequence must contains T_α.

Estimation of α is achieved by five-fold or ten-fold cross validation. We choose the value $\hat{\alpha}$ to minimize the cross-validation sum of squares. The final tree is $T_{\hat{\alpha}}$.

```r
###### TREES ######
# Regression Tree Example
library(rpart)

# grow tree
fit <- rpart(Mileage~Price + Country + Reliability + Type,
   method="anova", data=cu.summary)

printcp(fit) # display the results
```

```
##
## Regression tree:
## rpart(formula = Mileage ~ Price + Country + Reliability + Type,
##     data = cu.summary, method = "anova")
##
## Variables actually used in tree construction:
## [1] Price Type
##
## Root node error: 1354.6/60 = 22.576
##
## n=60 (57 observations deleted due to missingness)
##
##         CP nsplit rel error  xerror     xstd
## 1 0.622885      0   1.00000 1.02068 0.175559
## 2 0.132061      1   0.37711 0.53802 0.104113
## 3 0.025441      2   0.24505 0.36889 0.084087
## 4 0.011604      3   0.21961 0.36766 0.084717
## 5 0.010000      4   0.20801 0.37946 0.084098
```

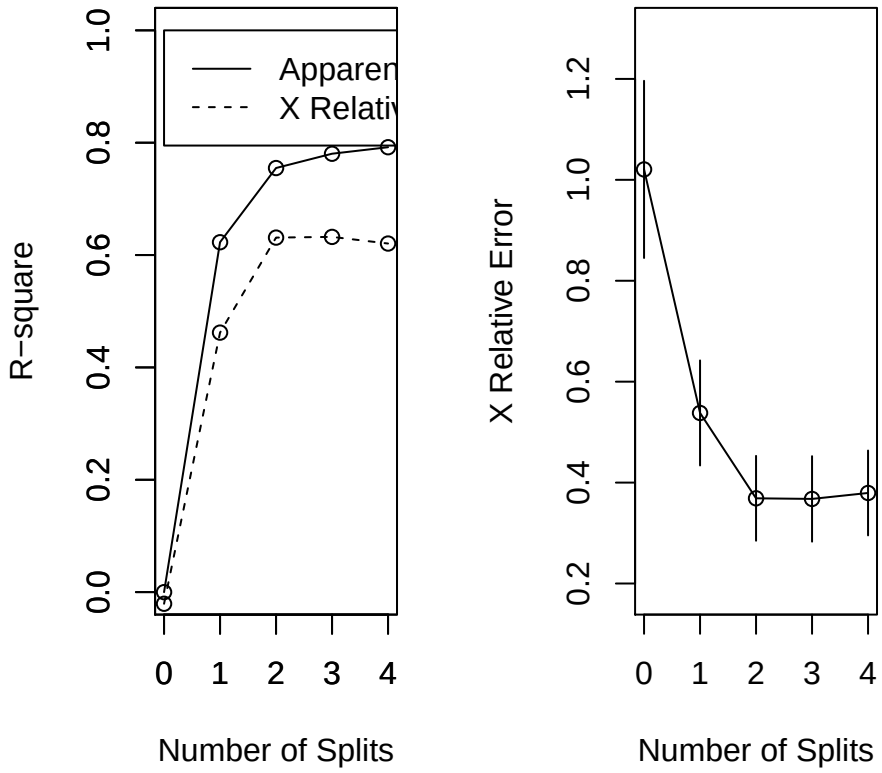
```r
plotcp(fit) # visualize cross-validation results
```



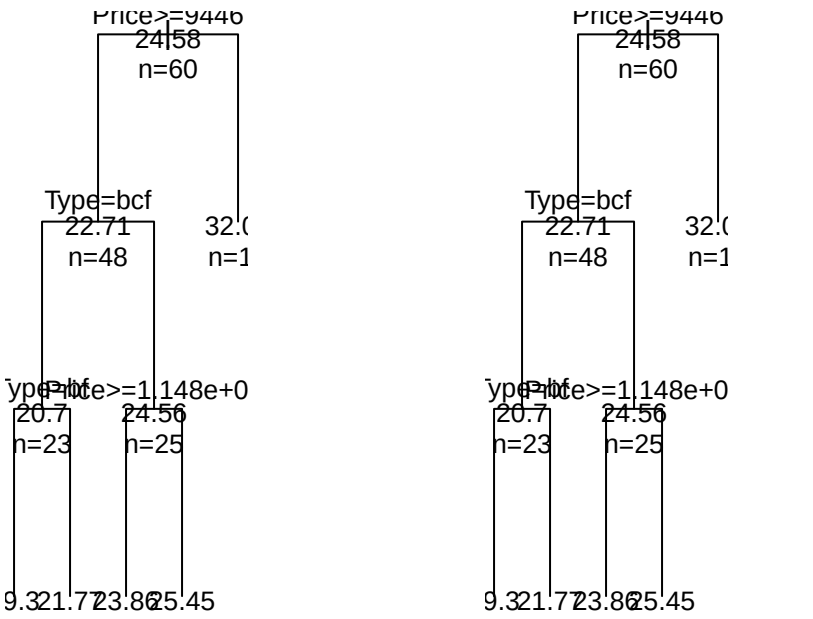

```
summary(fit) # detailed summary of splits

## Call:
## rpart(formula = Mileage ~ Price + Country + Reliability + Type,
##     data = cu.summary, method = "anova")
##   n=60 (57 observations deleted due to missingness)
##
##           CP nsplit rel error    xerror       xstd
## 1 0.62288527      0 1.0000000 1.0206789 0.17555873
## 2 0.13206061      1 0.3771147 0.5380196 0.10411251
## 3 0.02544094      2 0.2450541 0.3688866 0.08408695
## 4 0.01160389      3 0.2196132 0.3676566 0.08471729
## 5 0.01000000      4 0.2080093 0.3794646 0.08409785
##
## Variable importance
```

```
##   Price    Type Country
##      48      42      10
##
## Node number 1: 60 observations,    complexity param=0.6228853
##   mean=24.58333, MSE=22.57639
##   left son=2 (48 obs) right son=3 (12 obs)
##   Primary splits:
##       Price       < 9446.5  to the right, improve=0.6228853, (0 missing)
##       Type        splits as  LLLRLL,      improve=0.5044405, (0 missing)
##       Reliability splits as  LLLRR,       improve=0.1263005, (11 missing)
##       Country     splits as  --LRLRRRLL,  improve=0.1243525, (0 missing)
##   Surrogate splits:
##       Type    splits as  LLLRLL,     agree=0.950, adj=0.750, (0 split)
##       Country splits as  --LLLLRRLL, agree=0.833, adj=0.167, (0 split)
##
## Node number 2: 48 observations,    complexity param=0.1320606
##   mean=22.70833, MSE=8.498264
##   left son=4 (23 obs) right son=5 (25 obs)
##   Primary splits:
##       Type        splits as  RLLRRL,      improve=0.43853830, (0 missing)
##       Price       < 12154.5 to the right, improve=0.25748500, (0 missing)
##       Country     splits as  --RRLRL-LL,  improve=0.13345700, (0 missing)
##       Reliability splits as  LLLRR,       improve=0.01637086, (10 missing)
##   Surrogate splits:
##       Price   < 12215.5 to the right, agree=0.812, adj=0.609, (0 split)
##       Country splits as  --RRLRL-RL,  agree=0.646, adj=0.261, (0 split)
##
## Node number 3: 12 observations
##   mean=32.08333, MSE=8.576389
##
## Node number 4: 23 observations,    complexity param=0.02544094
##   mean=20.69565, MSE=2.907372
##   left son=8 (10 obs) right son=9 (13 obs)
##   Primary splits:
##       Type    splits as  -LR--L,      improve=0.515359600, (0 missing)
##       Price   < 14962   to the left,  improve=0.131259400, (0 missing)
##       Country splits as  ----L-R--R,  improve=0.007022107, (0 missing)
##   Surrogate splits:
##       Price < 13572   to the right, agree=0.609, adj=0.1, (0 split)
##
## Node number 5: 25 observations,    complexity param=0.01160389
##   mean=24.56, MSE=6.4864
##   left son=10 (14 obs) right son=11 (11 obs)
##   Primary splits:
##       Price       < 11484.5 to the right, improve=0.09693168, (0 missing)
##       Reliability splits as  LLRRR,       improve=0.07767167, (4 missing)
```

```
##       Type        splits as  L--RR-,      improve=0.04209834, (0 missing)
##       Country     splits as  --LRRR--LL,  improve=0.02201687, (0 missing)
##   Surrogate splits:
##       Country splits as  --LLLL--LR, agree=0.80, adj=0.545, (0 split)
##       Type    splits as  L--RL-,     agree=0.64, adj=0.182, (0 split)
##
## Node number 8: 10 observations
##   mean=19.3, MSE=2.21
##
## Node number 9: 13 observations
##   mean=21.76923, MSE=0.7928994
##
## Node number 10: 14 observations
##   mean=23.85714, MSE=7.693878
##
## Node number 11: 11 observations
##   mean=25.45455, MSE=3.520661

# create additional plots
par(mfrow=c(1,2)) # two plots on one page
rsq.rpart(fit) # visualize cross-validation results

##
## Regression tree:
## rpart(formula = Mileage ~ Price + Country + Reliability + Type,
##     data = cu.summary, method = "anova")
##
## Variables actually used in tree construction:
## [1] Price Type
##
## Root node error: 1354.6/60 = 22.576
##
## n=60 (57 observations deleted due to missingness)
##
##         CP nsplit rel error  xerror     xstd
## 1 0.622885      0   1.00000 1.02068 0.175559
## 2 0.132061      1   0.37711 0.53802 0.104113
## 3 0.025441      2   0.24505 0.36889 0.084087
## 4 0.011604      3   0.21961 0.36766 0.084717
## 5 0.010000      4   0.20801 0.37946 0.084098
```




```
# plot tree
plot(fit, uniform=TRUE,
        main="Regression Tree for Mileage ")
text(fit, use.n=TRUE, all=TRUE, cex=.8)

# prune the tree
# It turns out that pruning produces the same tree as the original.

pfit<- prune(fit, cp=0.01160389) # from cptable

# plot the pruned tree
plot(pfit, uniform=TRUE,
        main="Pruned Regression Tree for Mileage")
text(pfit, use.n=TRUE, all=TRUE, cex=.8)
```



1.3 Classification Trees

If the target is a classification outcome taking values $1, 2, \ldots, K$, the only changes needed in the tree algorithm pertain to the criteria for splitting nodes and pruning the tree. For regression we used the squared-error node impurity measure $Q_m(T) = \frac{1}{N_m} \sum_{x_i \in R_m} (y_i - \hat{c}_m)^2$, but this is not suitable for classification. In a node m, representing a region R_m with N_m observations, let

$$\hat{p}_{mk} = \frac{1}{N_m} \sum_{x_i \in R_m} I(y_i = k),$$

the proportion of class k observations in node m. We classify the observations in node m to class $k(m) = \arg\max_k \hat{p}_{mk}$, the majority class in node m. Different measures $Q_m(T)$ of node impurity include the following:

Misclassification error: $\frac{1}{N_m} \sum_{x_i \in R_m} I(y_i \neq k(m)) = 1 - \hat{p}_{mk(m)}$.

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

Cross-entropy or deviance: $-\sum_{k=1}^{K} \hat{p}_{mk} \log \hat{p}_{mk}$.

For two classes, if p is the proportion in the second class, these three measures are $1-\max(p, 1-p)$, $2p(1-p)$ and $-p \log p - (1-p)\log(1-p)$, respectively. They are shown in the figure below. All three are similar, but cross-entropy and the Gini index are differentiable, and hence more amenable to numerical optimization. We see that we need to weight the node impurity measures by the number Nm_L and Nm_R of observations in the two child nodes created by splitting node m.

In addition, cross-entropy and the Gini index are more sensitive to changes in the node probabilities than the misclassification rate. For example, in a two-class problem with 400 observations in each class (denote this by (400, 400)), suppose one split created nodes (300, 100) and (100, 300), while the other created nodes (200, 400) and (200, 0). Both splits produce a misclassification rate of 0.25, but the second split produces a pure node and is probably preferable. Both the Gini index and cross-entropy are lower for the second split. For this reason, either the Gini index or cross-entropy should be used when growing the tree. To guide cost-complexity pruning, any of the three measures can be used, but typically it is the misclassification rate.

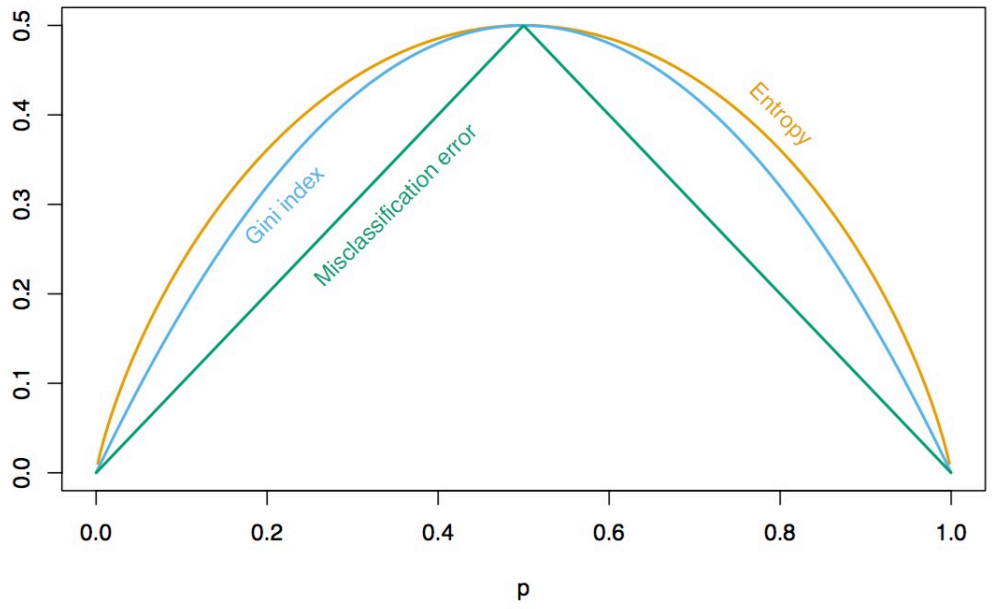


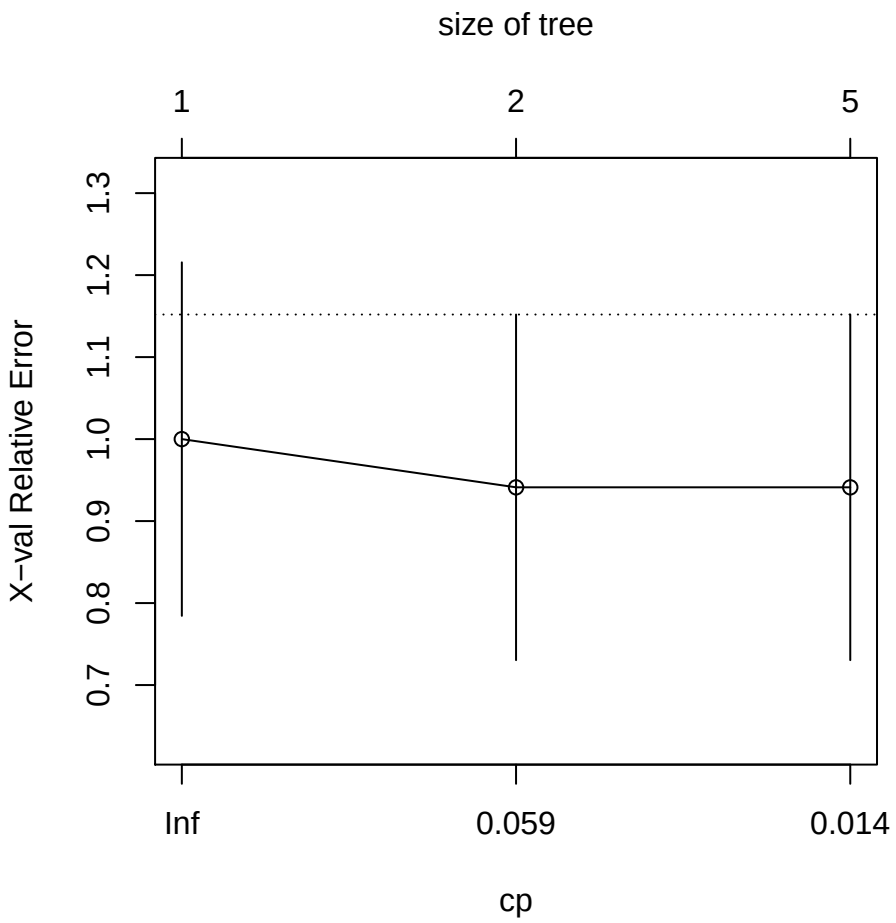
Figure 2: Node impurity measures for two-class classification, as a function of the proportion p in class 2. Cross-entropy has been scaled to pass through (0.5, 0.5).

```
# Classification Tree with rpart
library(rpart)

# grow tree
```

```r
fit <- rpart(Kyphosis ~ Age + Number + Start,
        method="class", data=kyphosis)

printcp(fit) # display the results
```

```
##
## Classification tree:
## rpart(formula = Kyphosis ~ Age + Number + Start, data = kyphosis,
##     method = "class")
##
## Variables actually used in tree construction:
## [1] Age   Start
##
## Root node error: 17/81 = 0.20988
##
## n= 81
##
##         CP nsplit rel error  xerror    xstd
## 1 0.176471      0   1.00000 1.00000 0.21559
## 2 0.019608      1   0.82353 0.94118 0.21078
## 3 0.010000      4   0.76471 0.94118 0.21078
```

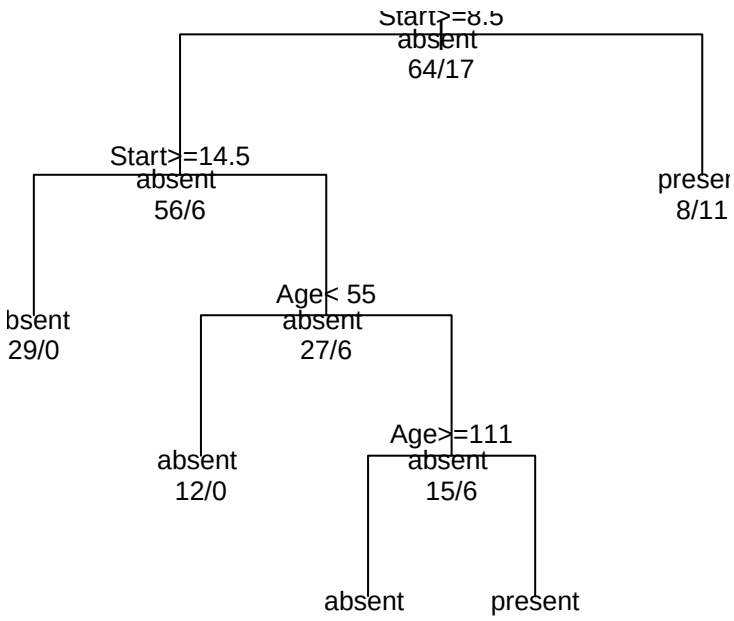
```r
plotcp(fit) # visualize cross-validation results
```




```
summary(fit) # detailed summary of splits

## Call:
## rpart(formula = Kyphosis ~ Age + Number + Start, data = kyphosis,
##     method = "class")
##   n= 81
##
##           CP nsplit rel error    xerror      xstd
## 1 0.17647059      0 1.0000000 1.0000000 0.2155872
## 2 0.01960784      1 0.8235294 0.9411765 0.2107780
## 3 0.01000000      4 0.7647059 0.9411765 0.2107780
##
## Variable importance
##  Start    Age Number
##     64     24     12
```

```
##
## Node number 1: 81 observations,    complexity param=0.1764706
##   predicted class=absent   expected loss=0.2098765  P(node) =1
##     class counts:    64    17
##    probabilities: 0.790 0.210
##   left son=2 (62 obs) right son=3 (19 obs)
##   Primary splits:
##       Start  < 8.5  to the right, improve=6.762330, (0 missing)
##       Number < 5.5  to the left,  improve=2.866795, (0 missing)
##       Age    < 39.5 to the left,  improve=2.250212, (0 missing)
##   Surrogate splits:
##       Number < 6.5  to the left,  agree=0.802, adj=0.158, (0 split)
##
## Node number 2: 62 observations,    complexity param=0.01960784
##   predicted class=absent   expected loss=0.09677419  P(node) =0.7654321
##     class counts:    56     6
##    probabilities: 0.903 0.097
##   left son=4 (29 obs) right son=5 (33 obs)
##   Primary splits:
##       Start  < 14.5 to the right, improve=1.0205280, (0 missing)
##       Age    < 55   to the left,  improve=0.6848635, (0 missing)
##       Number < 4.5  to the left,  improve=0.2975332, (0 missing)
##   Surrogate splits:
##       Number < 3.5  to the left,  agree=0.645, adj=0.241, (0 split)
##       Age    < 16   to the left,  agree=0.597, adj=0.138, (0 split)
##
## Node number 3: 19 observations
##   predicted class=present  expected loss=0.4210526  P(node) =0.2345679
##     class counts:     8    11
##    probabilities: 0.421 0.579
##
## Node number 4: 29 observations
##   predicted class=absent   expected loss=0  P(node) =0.3580247
##     class counts:    29     0
##    probabilities: 1.000 0.000
##
## Node number 5: 33 observations,    complexity param=0.01960784
##   predicted class=absent   expected loss=0.1818182  P(node) =0.4074074
##     class counts:    27     6
##    probabilities: 0.818 0.182
##   left son=10 (12 obs) right son=11 (21 obs)
##   Primary splits:
##       Age    < 55   to the left,  improve=1.2467530, (0 missing)
##       Start  < 12.5 to the right, improve=0.2887701, (0 missing)
##       Number < 3.5  to the right, improve=0.1753247, (0 missing)
##   Surrogate splits:
```

```
##       Start  < 9.5  to the left,  agree=0.758, adj=0.333, (0 split)
##       Number < 5.5  to the right, agree=0.697, adj=0.167, (0 split)
##
## Node number 10: 12 observations
##   predicted class=absent   expected loss=0  P(node) =0.1481481
##     class counts:    12     0
##    probabilities: 1.000 0.000
##
## Node number 11: 21 observations,    complexity param=0.01960784
##   predicted class=absent   expected loss=0.2857143  P(node) =0.2592593
##     class counts:    15     6
##    probabilities: 0.714 0.286
##   left son=22 (14 obs) right son=23 (7 obs)
##   Primary splits:
##       Age    < 111  to the right, improve=1.71428600, (0 missing)
##       Start  < 12.5 to the right, improve=0.79365080, (0 missing)
##       Number < 3.5  to the right, improve=0.07142857, (0 missing)
##
## Node number 22: 14 observations
##   predicted class=absent   expected loss=0.1428571  P(node) =0.1728395
##     class counts:    12     2
##    probabilities: 0.857 0.143
##
## Node number 23: 7 observations
##   predicted class=present  expected loss=0.4285714  P(node) =0.08641975
##     class counts:     3     4
##    probabilities: 0.429 0.571

# plot tree
plot(fit, uniform=TRUE, main="Classification Tree for Kyphosis")
text(fit, use.n=TRUE, all=TRUE, cex=.8)
```



1.4 Other Issues

Instability of Trees. One major problem with trees is their high variance. Often a small change in the data can result in a very different series of splits, making interpretation somewhat precarious. The major reason for this instability is the hierarchical nature of the process: the effect of an error in the top split is propagated down to all of the splits below it. One can alleviate this to some degree by trying to use a more stable split criterion, but the inherent instability is not removed. It is the price to be paid for estimating a simple, tree-based structure from the data.

Lack of Smoothness. Another limitation of trees is the lack of smoothness of the prediction surface. In classification with 0/1 loss, this does not hurt much, since bias in estimation of the class probabilities has a limited effect. However, this can degrade performance in the regression setting, where we would normally expect the underlying function to be smooth.

Difficulty in Capturing Additive Structure. Another problem with trees is their difficulty in modeling additive structure. In regression, suppose, for example, that $Y = c_1 I(X_1 < t_1) + c_2 I(X_2 < t_2) + \varepsilon$ where ε is zero-mean noise. Then a binary tree might make its first split on X_1 near t_1. At the next level down it would have to split both nodes on X_2 at t_2 in order to capture the additive structure. This might happen with sufficient data, but the model is given no special encouragement to find such structure. If there were ten rather than two additive effects, it would take many fortuitous splits to recreate the structure, and the data analyst would be hard pressed to recognize it in the estimated tree. The "blame" here can again be attributed to the binary tree structure, which has both advantages and drawbacks.

2 Gradient Boosting

Predictive learning problem

- A random "output" or "response" variable y
- A set of random "input" or "explanatory" variables $\mathbf{x} = (x_1, \ldots, x_p)$.

Theoretically, if the joint distribution of $(y, \mathbf{x})$ is known, then we can obtain

$$f^*(\mathbf{x}) = \arg\min_f E_{y,\mathbf{x}} L(y, f(\mathbf{x})) = \arg\min_f E_{\mathbf{x}}[E_y(L(y, f(\mathbf{x}))|\mathbf{x}].$$

Given $\{y_i, \mathbf{x}_i\}_{i=1}^N$ of know $(y, \mathbf{x})$-values, the goal is to obtain an estimate $\hat{f}(\mathbf{x})$, as the approximation of $f^*(\mathbf{x})$

$$\hat{f}(x) = \arg\min_f \frac{1}{N} \sum_{i=1}^{N} L(y_i, f(\mathbf{x}_i)). \tag{1}$$

Let $L(f) = \frac{1}{N} \sum_{i=1}^N L(y_i, f(\mathbf{x}_i))$, solving (1) is equivalent to solving

$$\hat{f} = \arg\min_f L(f),$$

where $f = (f(\mathbf{x}_1), f(\mathbf{x}_2), \ldots, f(\mathbf{x}_N))$ are the "parameters". We will solve this stagewise, using gradient descent. At step m, let g_m be the negative gradient of $L(f)$ evaluated at $f = f_{m-1}$:

$$g_{im} = -\left[\frac{\partial L(f)}{\partial f}\right]_{f=f_{m-1}} = -\left[\frac{\partial \left(\frac{1}{N}\sum_{i=1}^N L(y_i, f(\mathbf{x}_i))\right)}{\partial f(\mathbf{x}_i)}\right]_{f=f_{m-1}} \tag{2}$$

We then make the update

$$f_m = f_{m-1} + \rho_m g_m$$

where ρ_m is the step length, chosen by

$$\rho_m = \arg\min_\rho L(f_{m-1} + \rho g_m).$$

This is called functional gradient descent. In its current form, this is not much use, since the gradient (2) is defined only at the training data points $\mathbf{x}_i$, so we **can not** learn a function that can **generalize**. The ultimate goal is to generalize f_m to new data not represented in the training set.

However, we can modify the algorithm by fitting a weak learner to approximate the negative gradient signal. That is, we use this update

$$\gamma_m = \arg\min_{\gamma} \sum_{i=1}^{N} (g_{im} - \phi(\mathbf{x}_i; \gamma))^2 \tag{3}$$

Note: When $L(f) = (y-f)^2$, solving (3) is equivalent to fit the residuals. Since $g = -\frac{\partial L(f)}{\partial f} = 2(y-f)$, negative gradient g is just residual r.

The overall algorithm is summarized below. (We have omitted the line search step, which is not strictly necessary, as argued in (Buhlmann and Hothorn 2007).

Algorithm 2: Gradient boosting

Initialize $f_0(\mathbf{x}) = \arg\min_{\gamma} \sum_{i=1}^{N} L(y_i, \phi(\mathbf{x}_i; \gamma))$;

for $m = 1, \ldots, M$ **do**

Compute the gradient residual using $g_{im} = -\left[\frac{\partial\left(\frac{1}{N}\sum_{i=1}^{N}\Phi(y_i, f(\mathbf{x}_i))\right)}{\partial f(\mathbf{x}_i)}\right]_{f(\mathbf{x}_i)=f_{m-1}(\mathbf{x}_i)}$;

Use the weak learner to compute γ_m which minimizes $\sum_{i=1}^{N}(g_{im} - \phi(\mathbf{x}_i; \gamma))^2$;

Update $f_m(\mathbf{x}) = f_{m-1}(\mathbf{x}) + \nu\phi(\mathbf{x}; \gamma_m)$;

end

Return $f(\mathbf{x}) = f_M(\mathbf{x})$

3 Sparse Boosting

Suppose we use as our weak learner the following algorithm: search over all possible variables $j = 1, \ldots, p$, and pick the one $j(m)$ that best predicts the negative gradient.

$$j(m) = \arg\min_{j} \left[\min_{\beta_{jm}} \sum_{i=1}^{N} (g_{im} - \beta_{jm} x_{ij})^2\right]$$

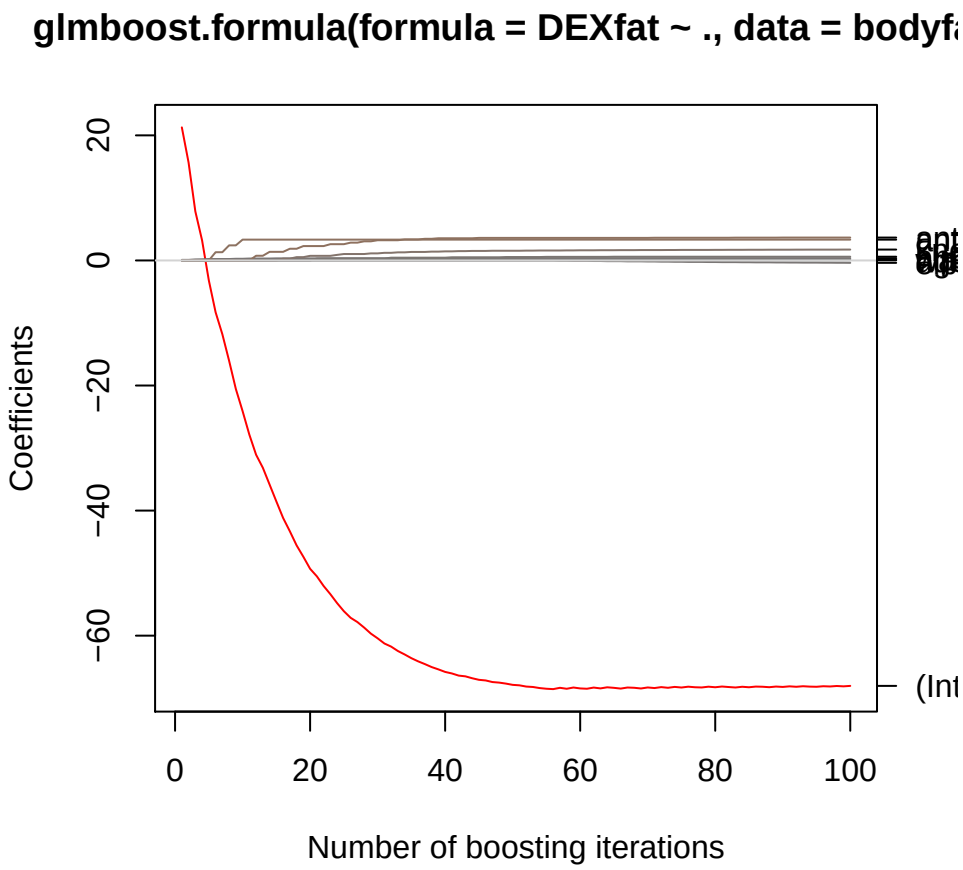
$$\phi_m(\mathbf{x}) = \hat{\beta}_{j(m)} \mathbf{x}_{j(m)}$$

It is clear that this will result in a sparse estimate, at least if M is small. To see this, let us rewrite the update as follows:

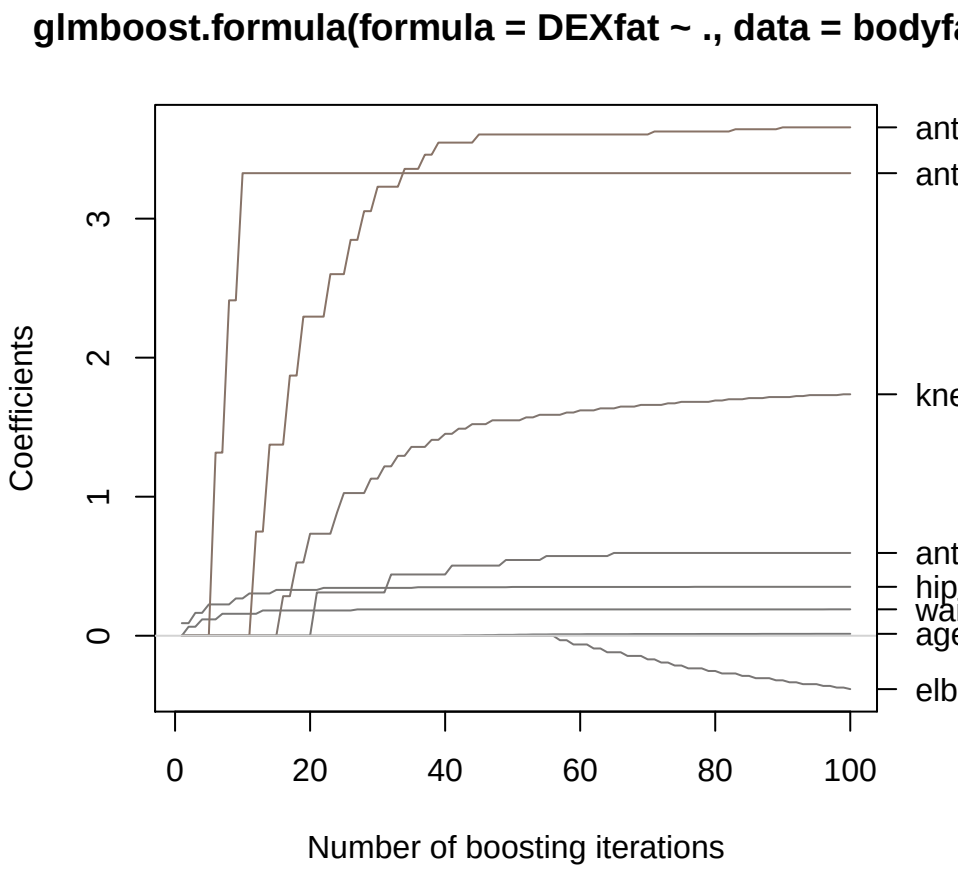
$$\beta_m = \beta_{m-1} + \nu(0, \ldots, 0, \hat{\beta}_{j(m)}, 0, \ldots, 0)$$

where the non-zero entry occurs in location $j(m)$. This is known as forward stagewise linear regression (Hastie et al. 2009, p608), which becomes equivalent to the LARS algorithm as $\nu \to 0$. Increasing the number of steps m in boosting is analogous to decreasing the regularization penalty λ. Now consider a weak learner that is similar to the above, except it uses a smoothing spline instead of linear regression when mapping from $\mathbf{x}_j$ to the residual. The result is a sparse generalized additive model (see Section 16.3).

```r
###### SPARSE BOOSTING ######

# library for sparse boosting
library("mboost")   ## load package

data("bodyfat", package = "TH.data")  ## load data

## Reproduce formula of Garcia et al., 2005
lm1 <- lm(DEXfat ~ hipcirc + kneebreadth + anthro3a, data = bodyfat)
coef(lm1)

## (Intercept)     hipcirc kneebreadth    anthro3a
## -75.2347840   0.5115264   1.9019904   8.9096375

## Estimate same model by glmboost
glm1 <- glmboost(DEXfat ~ hipcirc + kneebreadth + anthro3a, data = bodyfat)
coef(glm1, off2int=TRUE) ## off2int adds the offset to the intercept

## (Intercept)     hipcirc kneebreadth    anthro3a
## -75.2073365   0.5114861   1.9005386   8.9071301

#Note that in this case we used the default settings in control and the default
#family Gaussian() leading to
#boosting with the L2 loss.

#We now want to consider all available variables as potential predictors. One
#way is to simply specify "."
#on the right side of the formula:

glm2 <- glmboost(DEXfat ~ ., data = bodyfat)

#A plot of the coefficient paths, similar to the ones commonly known from the
#LARS algorithm (Efron et al.
#2004), can be easily produced by using plot() on the glmboost object


plot(glm2, off2int = TRUE) ## default plot, offset added to intercept
```




```
## now change ylim to the range of the coefficients without intercept (zoom-in)
preds <- names(bodyfat[, names(bodyfat) != "DEXfat"])
plot(glm2, ylim = range(coef(glm2, which = preds)))
```



3.1 Gradient Tree Boosting

In the Tree Boosting case, induce a tree $T(x; \Theta_m)$ at the mth iteration whose predictions are as close as possible to the negative gradient. Using squared error to measure closeness, this leads us to

$$\tilde{\Theta}_m = \arg\min_{\Theta} \sum_{i=1}^{N} (g_{im} - T(\mathbf{x}_i; \Theta))^2$$

Given the regions R_{jm}, finding the optimal constants γ_{jm} in each region is straightforward:

$$\hat{\gamma}_{jm} = \arg\min_{\gamma_{jm}} \sum_{\mathbf{x}_i \in R_{jm}} L(y_i, f_{m-1}(\mathbf{x}_i) + \gamma_{jm}).$$

Advantages of gradient tree boosting:

- Model structure is learned from data and not predetermined, avoiding an explicit model specification.

- Naturally incorporate complex and higher order interactions.
- Produce high predictive performance.
- Handle any type of data without the need for transformation.
- Insensitive to outliers and missing values.

Algorithm 3: Gradient Tree Boosting Algorithm.

---

1. Initialize $f_0(\mathbf{x}) = \arg\min_{\gamma} \sum_{i=1}^{N} L(y_i, \gamma)$.
2. For $m = 1, \ldots, M$:
   (a) For $i = 1, 2, \ldots, N$ compute

$$g_{im} = -\left[\frac{\partial\left(\frac{1}{N}\sum_{i=1}^{N} L(y_i, f(\mathbf{x}_i))\right)}{\partial f(\mathbf{x}_i)}\right]_{f=f_{m-1}}$$

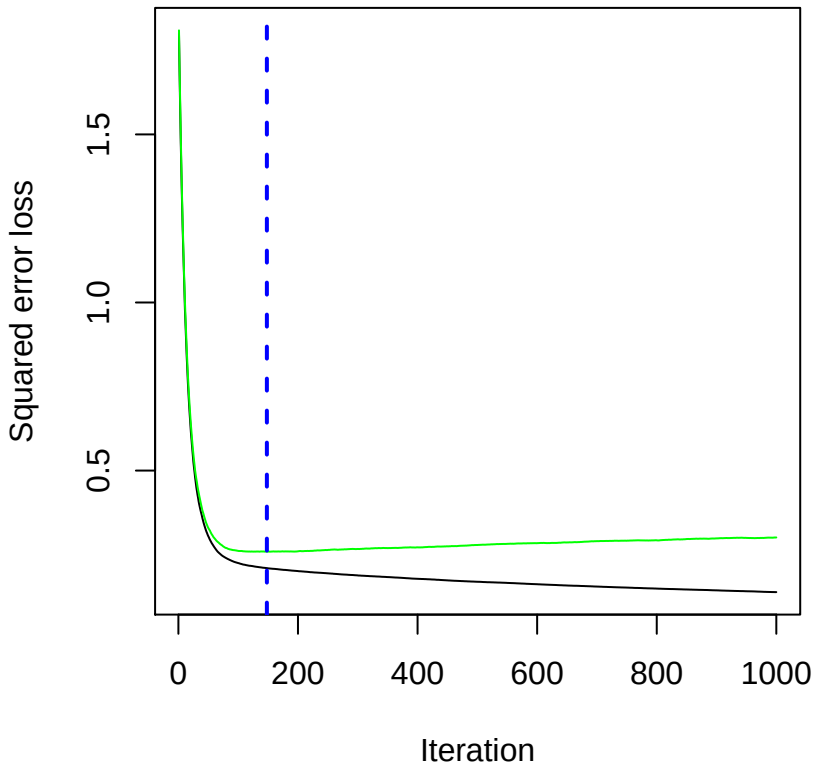
   (b) Fit the negative gradient vector $g_{1m}, \ldots, g_{Nm}$ to $\mathbf{x}_1, \ldots, \mathbf{x}_N$ by an L-terminal node regression tree, giving us terminal regions R_{jm}, $j = 1, 2, \ldots, J_m$.
   (c) For $j = 1, 2, \ldots, J_m$ compute

$$\gamma_{jm} = \arg\min_{\gamma} \frac{1}{N} \sum_{x_i \in R_{jm}} L(y_i, f_{m-1}(\mathbf{x}_i) + \gamma).$$

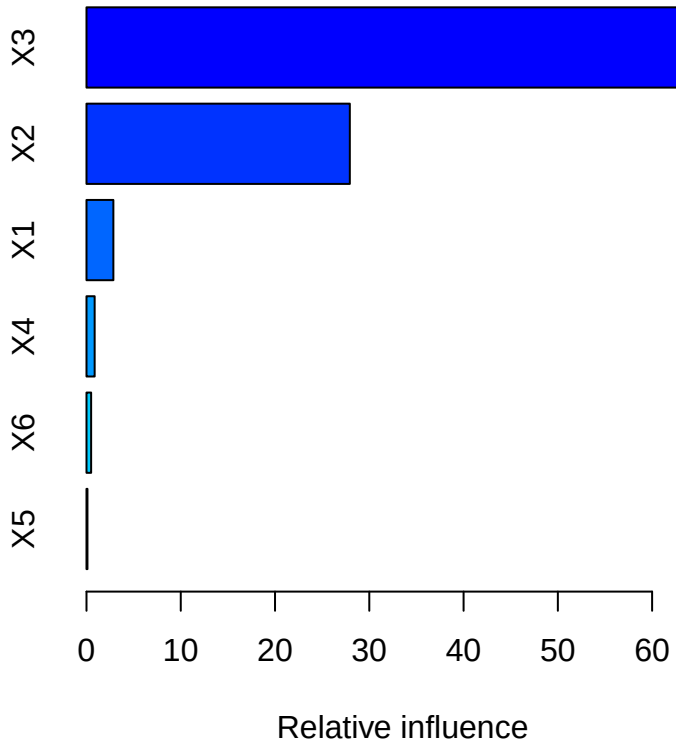
   (d) Update $f_m(\mathbf{x}) = f_{m-1}(\mathbf{x}) + \nu \sum_{j=1}^{J_m} \gamma_{jm} I(x \in R_{jm})$.
3. Report $\hat{f}(\mathbf{x}) = f_M(\mathbf{x})$ as the final estimate.

```
###### GRADIENT TREE BOOSTING ######
library("gbm")

N <- 1000
X1 <- runif(N)
X2 <- 2*runif(N)
X3 <- ordered(sample(letters[1:4],N,replace=TRUE),levels=letters[4:1])
X4 <- factor(sample(letters[1:6],N,replace=TRUE))
X5 <- factor(sample(letters[1:3],N,replace=TRUE))
X6 <- 3*runif(N)
mu <- c(-1,0,1,2)[as.numeric(X3)]

SNR <- 10 # signal-to-noise ratio
```

```
Y <- X1**1.5 + 2 * (X2**.5) + mu
sigma <- sqrt(var(Y)/SNR)
Y <- Y + rnorm(N,0,sigma)

# introduce some missing values
X1[sample(1:N,size=500)] <- NA
X4[sample(1:N,size=300)] <- NA

data <- data.frame(Y=Y,X1=X1,X2=X2,X3=X3,X4=X4,X5=X5,X6=X6)

# fit initial model
gbm1 <-
gbm(Y~X1+X2+X3+X4+X5+X6,          # formula
    data=data,                    # dataset
    var.monotone=c(0,0,0,0,0,0), # -1: monotone decrease,
                                  # +1: monotone increase,
                                  #  0: no monotone restrictions
    distribution="gaussian",      # see the help for other choices
    n.trees=1000,                 # number of trees
    shrinkage=0.05,               # shrinkage or learning rate,
                                  # 0.001 to 0.1 usually work
    interaction.depth=3,          # 1: additive model, 2: two-way interactions, etc.
    bag.fraction = 0.5,           # subsampling fraction, 0.5 is probably best
    train.fraction = 1,           # fraction of data for training,
                                  # first train.fraction*N used for training
    n.minobsinnode = 10,          # minimum total weight needed in each node
    cv.folds = 3,                 # do 3-fold cross-validation
    keep.data=TRUE,               # keep a copy of the dataset with the object
    verbose=FALSE,                # don't print out progress
    n.cores=1)                    # use only a single core (detecting #cores is
                                  # error-prone, so avoided here)


# check performance using 5-fold cross-validation
best.iter <- gbm.perf(gbm1,method="cv", plot.it="TRUE")
```



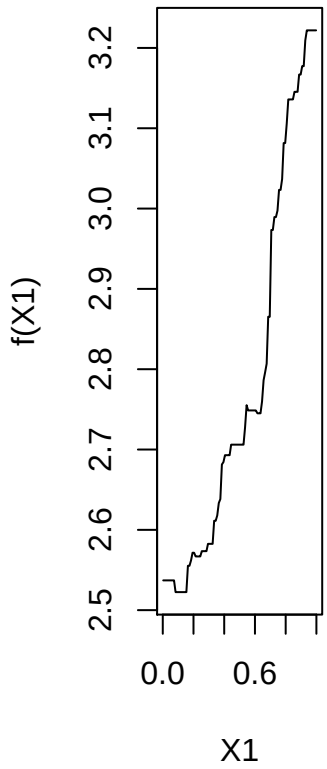

```
print(best.iter)

## [1] 118

# plot the performance # plot variable influence
summary(gbm1,n.trees=best.iter) # based on the estimated best number of trees
```

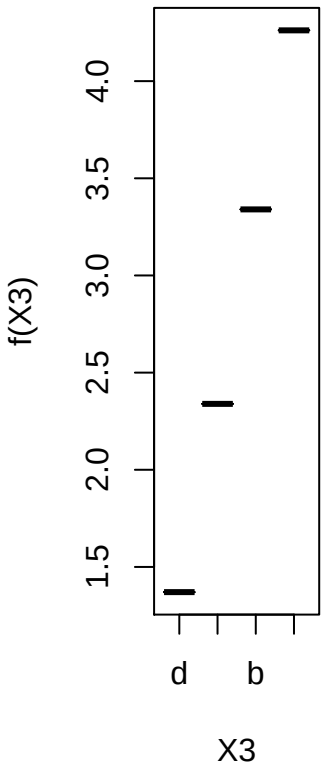



```
##    var    rel.inf
## X3  X3 68.3370556
## X2  X2 27.9675689
## X1  X1  2.9428360
## X4  X4  0.5364789
## X6  X6  0.2160606
## X5  X5  0.0000000

# compactly print the first and last trees for curiosity
print(pretty.gbm.tree(gbm1,1))

##   SplitVar SplitCodePred LeftNode RightNode MissingNode ErrorReduction
## 0        2   1.500000000        1         5           9      533.98185
## 1        1   0.604380587        2         3           4      103.78633
## 2       -1  -0.096073876       -1        -1          -1        0.00000
```

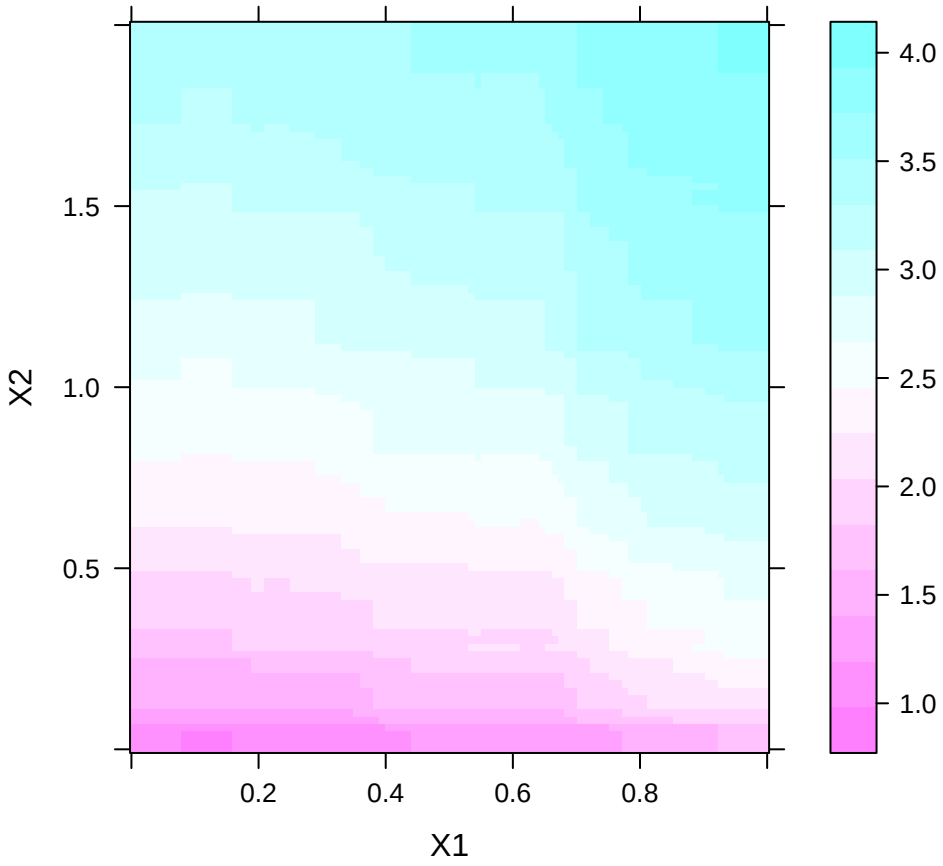
```
## 3       -1  -0.027170167       -1        -1          -1        0.00000
## 4       -1  -0.049584626       -1        -1          -1        0.00000
## 5        1   1.077410759        6         7           8       54.93304
## 6       -1   0.030274088       -1        -1          -1        0.00000
## 7       -1   0.077056587       -1        -1          -1        0.00000
## 8       -1   0.053758530       -1        -1          -1        0.00000
## 9       -1   0.002293638       -1        -1          -1        0.00000
##   Weight   Prediction
## 0    500  0.002293638
## 1    249 -0.049584626
## 2     81 -0.096073876
## 3    168 -0.027170167
## 4    249 -0.049584626
## 5    251  0.053758530
## 6    125  0.030274088
## 7    126  0.077056587
## 8    251  0.053758530
## 9    500  0.002293638

print(pretty.gbm.tree(gbm1,gbm1$n.trees))

##   SplitVar SplitCodePred LeftNode RightNode MissingNode ErrorReduction
## 0        1  8.611149e-02        1         2           9      0.3830277
## 1       -1 -7.046352e-03       -1        -1          -1      0.0000000
## 2        3  6.430000e+02        3         7           8      0.3233829
## 3        0  8.219938e-02        4         5           6      0.4939331
## 4       -1  1.022043e-02       -1        -1          -1      0.0000000
## 5       -1 -8.585792e-04       -1        -1          -1      0.0000000
## 6       -1 -1.182638e-03       -1        -1          -1      0.0000000
## 7       -1  3.945599e-03       -1        -1          -1      0.0000000
## 8       -1 -7.498607e-04       -1        -1          -1      0.0000000
## 9       -1 -5.957140e-04       -1        -1          -1      0.0000000
##   Weight    Prediction
## 0    500 -0.0005957140
## 1     22 -0.0070463523
## 2    478 -0.0002988227
## 3    303 -0.0006736813
## 4     10  0.0102204278
## 5    124 -0.0008585792
## 6    169 -0.0011826385
## 7     41  0.0039455987
## 8    134 -0.0007498607
## 9    500 -0.0005957140

# make some new data
N <- 1000
X1 <- runif(N)
```

```
X2 <- 2*runif(N)
X3 <- ordered(sample(letters[1:4],N,replace=TRUE))
X4 <- factor(sample(letters[1:6],N,replace=TRUE))
X5 <- factor(sample(letters[1:3],N,replace=TRUE))
X6 <- 3*runif(N)
mu <- c(-1,0,1,2)[as.numeric(X3)]

Y <- X1**1.5 + 2 * (X2**.5) + mu + rnorm(N,0,sigma)

data2 <- data.frame(Y=Y,X1=X1,X2=X2,X3=X3,X4=X4,X5=X5,X6=X6)

# predict on the new data using "best" number of trees
# f.predict generally will be on the canonical scale (logit,log,etc.)
f.predict <- predict(gbm1,data2,best.iter)

# least squares error
print(sum((data2$Y-f.predict)^2))

## [1] 4771.406

# create marginal plots
# plot variable X1,X3 after "best" iterations
par(mfrow=c(1,2))
plot(gbm1,1,best.iter)
plot(gbm1,3,best.iter)
```

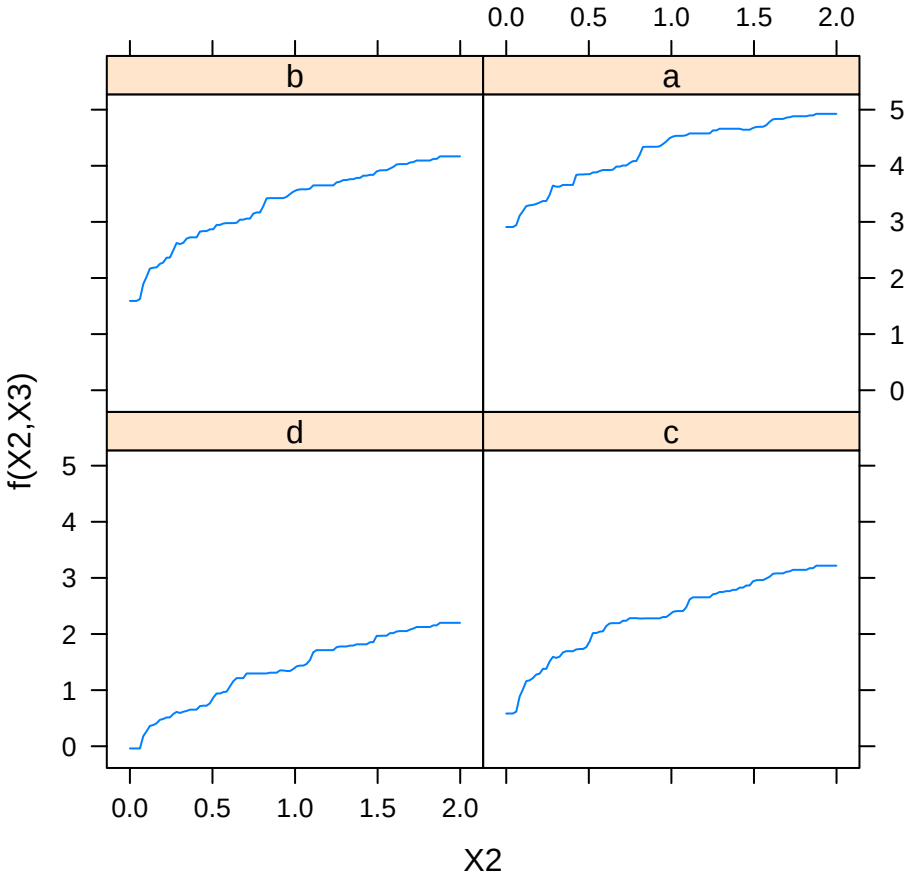





```
# contour plot of variables 1 and 2 after "best" iterations
plot(gbm1,1:2,best.iter)
```




```
# lattice plot of variables 2 and 3
plot(gbm1,2:3,best.iter)
```


b
a
d
c
f(X2,X3)
X2
0.0
0.5
1.0
1.5
2.0
0
1
2
3
4
5