MATH 680 Computation Intensive Statistics

July 11, 2018

Gradient Boosting

Contents

1 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 solving (**??**). 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 1: 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 $f(\mathbf{x}) = f_M(\mathbf{x})$

2 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.

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

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

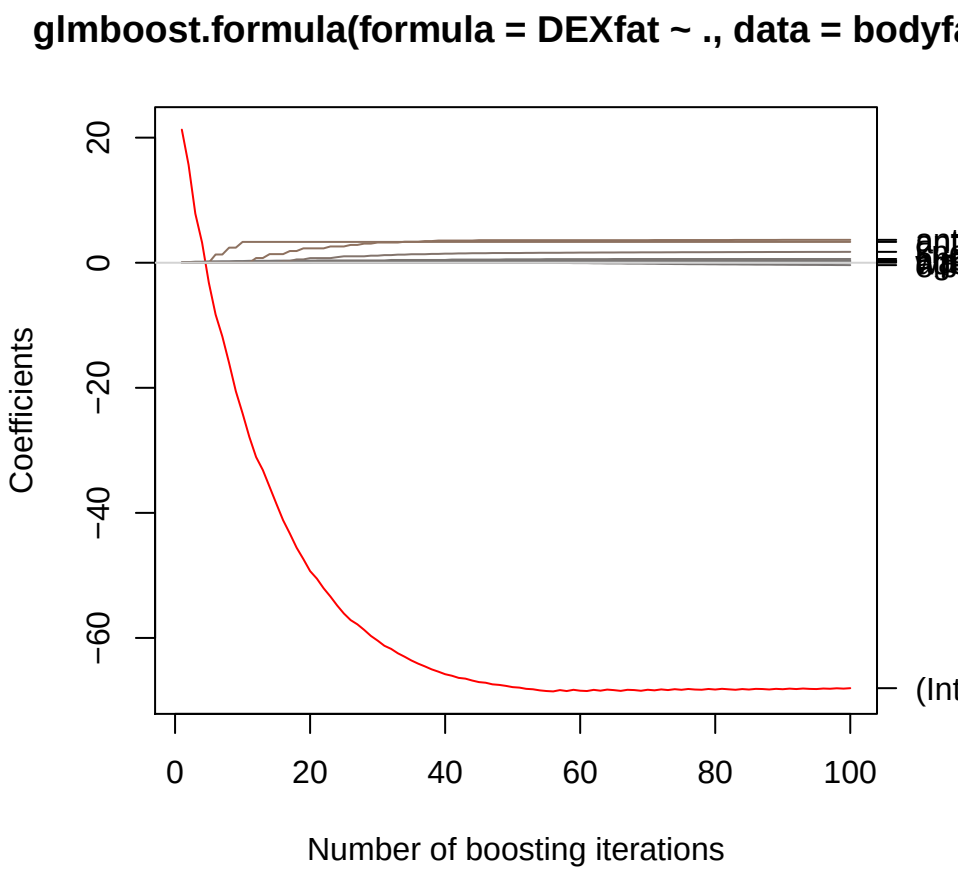
It is clear that this will result in a sparse estimate, at least if M is small. To see this, let us rewrite the update as follows:

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

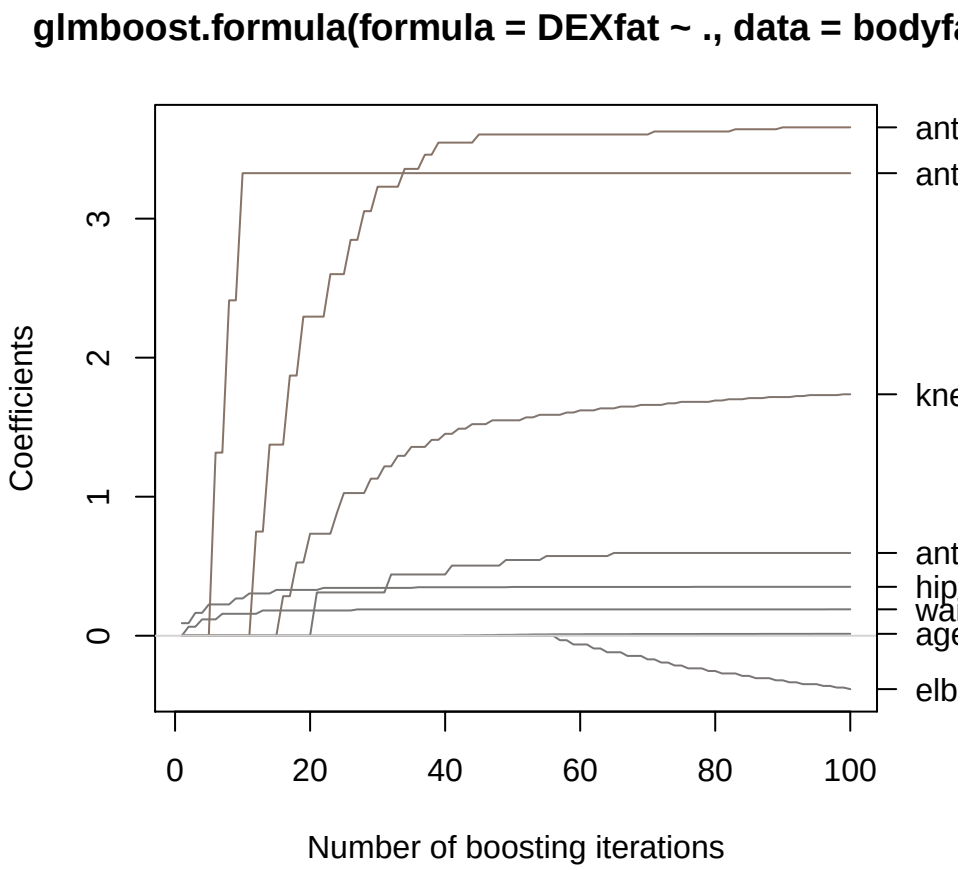
where the non-zero entry occurs in location $j(m)$. This is known as forward stagewise linear regression (Hastie et al. 2009, p608), which becomes equivalent to the LARS algorithm as $\nu \to 0$. Increasing the number of steps m in boosting is analogous to decreasing the regularization penalty λ. Now 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).

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

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

## Warning:  package 'mboost' was built under R version 3.4.4
## Loading required package:  methods
## Loading required package:  parallel
## Loading required package:  stabs
## This is mboost 2.9-0.  See 'package?mboost' and 'news(package  = "mboost")'
## for a complete list of changes.

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

predict(lm1, newdata = bodyfat[1,])

##       47
## 39.31548

## 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
```

```r
#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
```




```r
## 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)))
```



2.1 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 2: 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

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

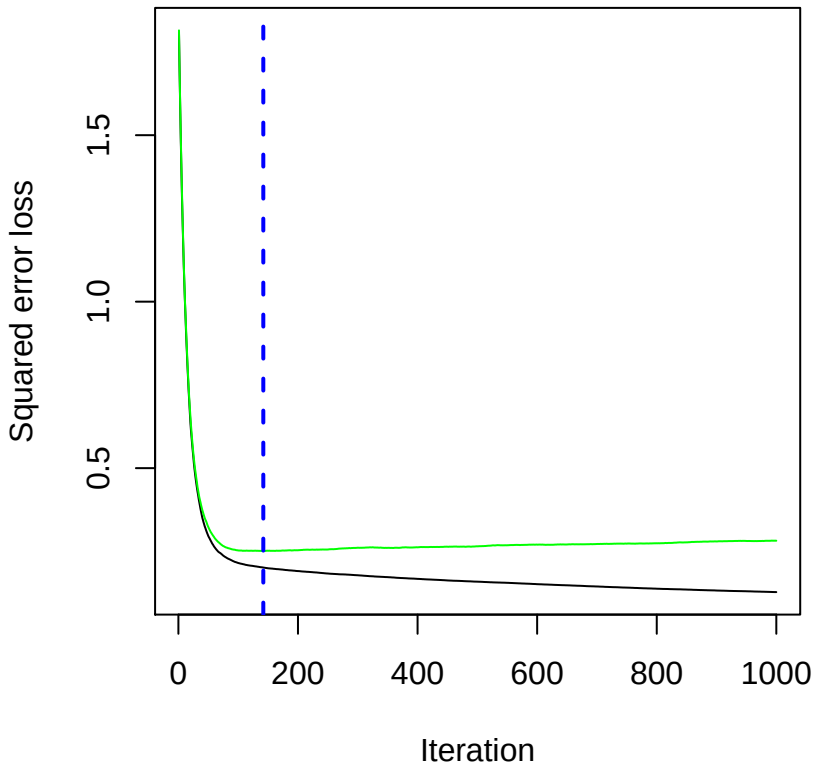
   (b) Fit the negative gradient vector $g_{1m}, \ldots, g_{Nm}$ to $\mathbf{x}_1, \ldots, \mathbf{x}_N$ by an L-terminal node regression tree, giving us terminal regions R_{jm}, $j = 1, 2, \ldots, J_m$.
   (c) For $j = 1, 2, \ldots, J_m$ compute

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

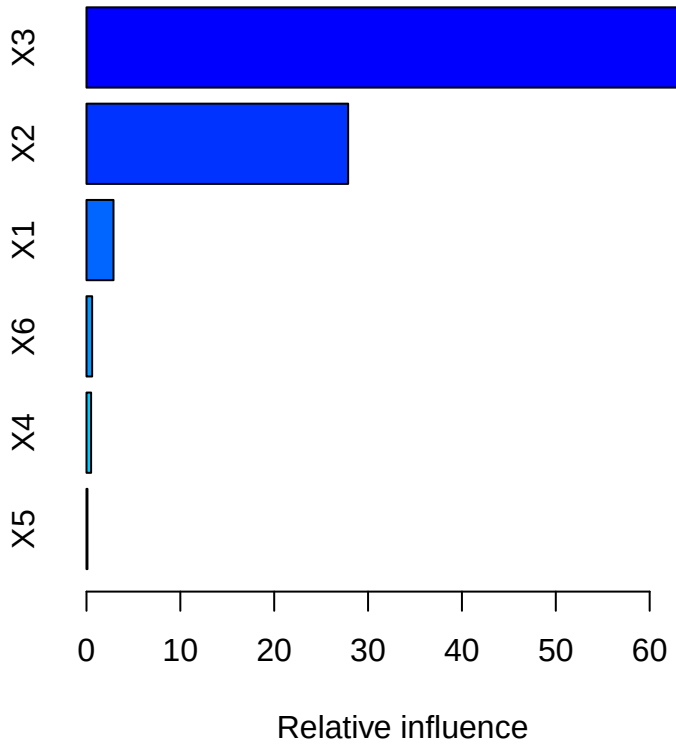
   (d) Update $f_m(\mathbf{x}) = f_{m-1}(\mathbf{x}) + \nu \sum_{j=1}^{J_m} \gamma_{jm} I(x \in R_{jm})$.
3. Report $\hat{f}(\mathbf{x}) = f_M(\mathbf{x})$ as the final estimate.

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

## Loading required package:  survival
## Loading required package:  lattice
## Loading required package:  splines
## Loaded gbm 2.1.3

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")
```



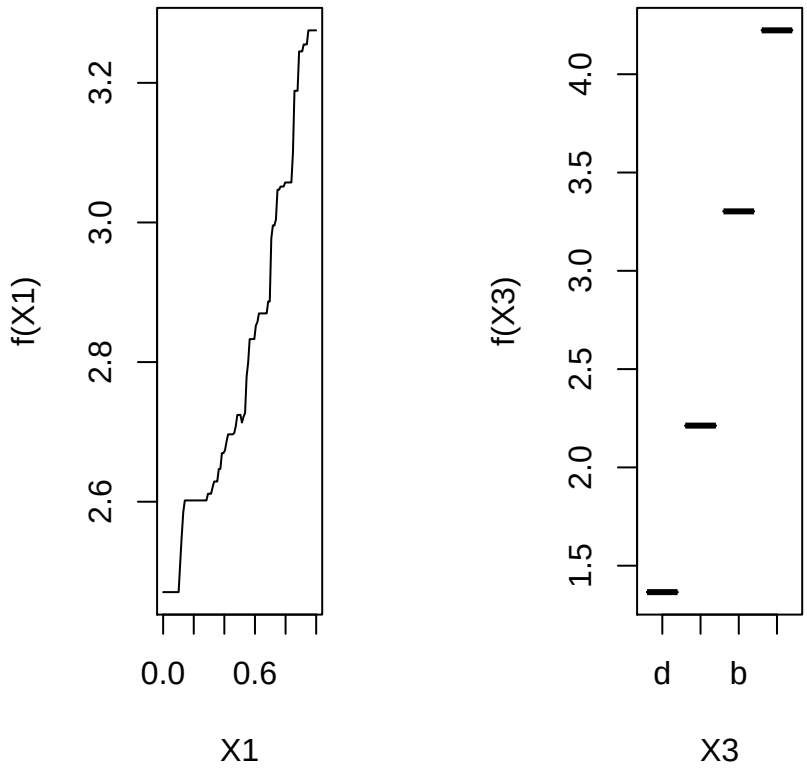

```
print(best.iter)

## [1] 142

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



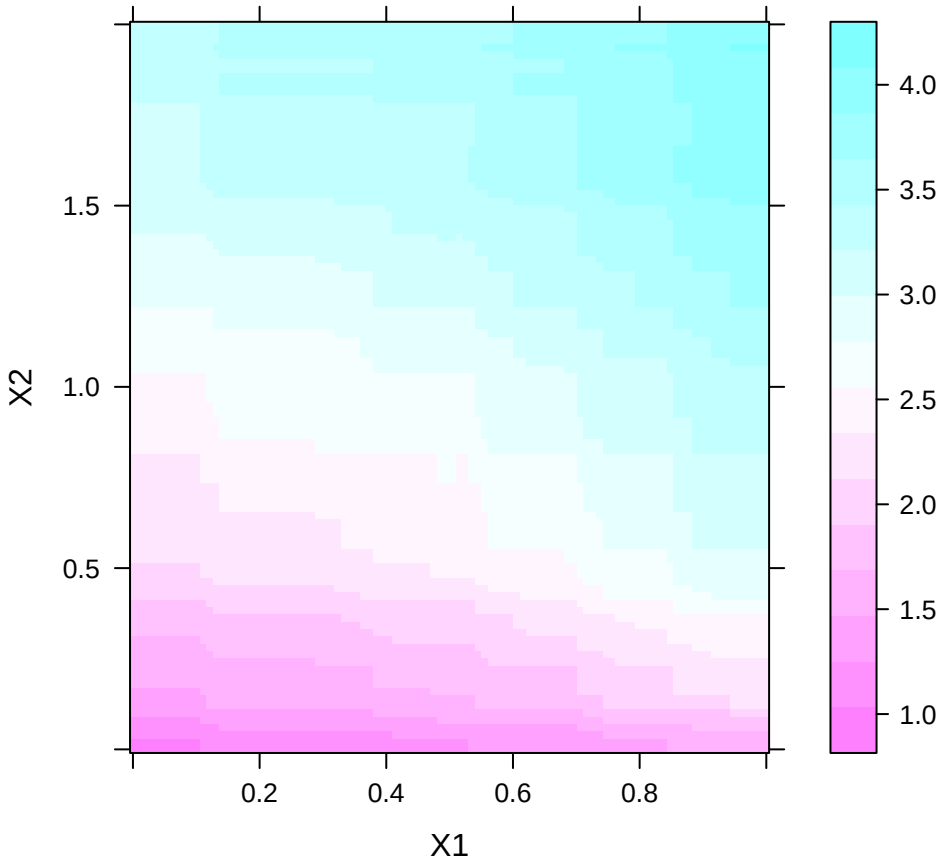

```
##    var    rel.inf
## X3  X3 68.0478126
## X2  X2 27.8765069
## X1  X1  2.8769985
## X6  X6  0.6033120
## X4  X4  0.4927230
## X5  X5  0.1026471

# 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      519.73854
## 1        1   0.813844783        2         3           4       90.88892
## 2       -1  -0.094570432       -1        -1          -1        0.00000
```

```
## 3       -1  -0.030685277       -1        -1          -1        0.00000
## 4       -1  -0.055910317       -1        -1          -1        0.00000
## 5        1   1.021839116        6         7           8       69.82533
## 6       -1   0.020997253       -1        -1          -1        0.00000
## 7       -1   0.072139322       -1        -1          -1        0.00000
## 8       -1   0.046280972       -1        -1          -1        0.00000
## 9       -1  -0.001340168       -1        -1          -1        0.00000
##   Weight   Prediction
## 0    500 -0.001340168
## 1    233 -0.055910317
## 2     92 -0.094570432
## 3    141 -0.030685277
## 4    233 -0.055910317
## 5    267  0.046280972
## 6    135  0.020997253
## 7    132  0.072139322
## 8    267  0.046280972
## 9    500 -0.001340168

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

##   SplitVar SplitCodePred LeftNode RightNode MissingNode ErrorReduction
## 0        1  1.5610180844        1         2           9      0.7566877
## 1       -1 -0.0012374534       -1        -1          -1      0.0000000
## 2        5  1.7743009989        3         4           8      1.1068601
## 3       -1  0.0073564969       -1        -1          -1      0.0000000
## 4        5  2.4917323182        5         6           7      1.2222829
## 5       -1 -0.0094688201       -1        -1          -1      0.0000000
## 6       -1  0.0093083365       -1        -1          -1      0.0000000
## 7       -1 -0.0032097679       -1        -1          -1      0.0000000
## 8       -1  0.0035052415       -1        -1          -1      0.0000000
## 9       -1 -0.0002225167       -1        -1          -1      0.0000000
##   Weight    Prediction
## 0    500 -0.0002225167
## 1    393 -0.0012374534
## 2    107  0.0035052415
## 3     68  0.0073564969
## 4     39 -0.0032097679
## 5     26 -0.0094688201
## 6     13  0.0093083365
## 7     39 -0.0032097679
## 8    107  0.0035052415
## 9    500 -0.0002225167

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

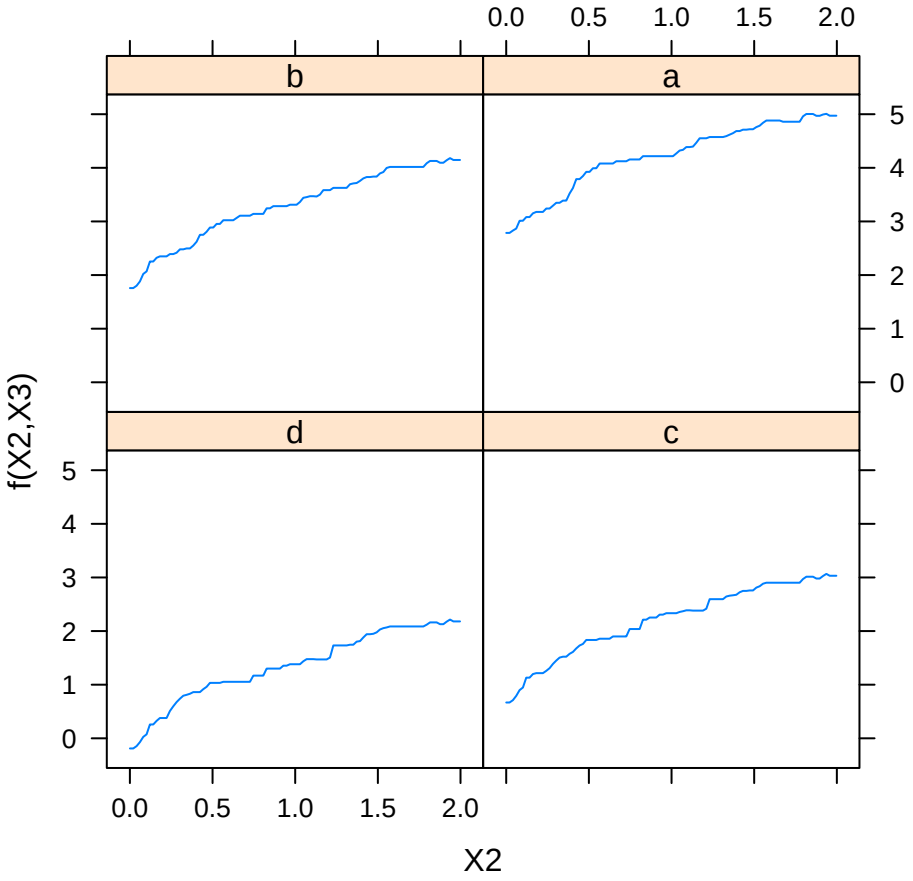
```
X2 <- 2*runif(N)
X3 <- ordered(sample(letters[1:4],N,replace=TRUE))
X4 <- factor(sample(letters[1:6],N,replace=TRUE))
X5 <- factor(sample(letters[1:3],N,replace=TRUE))
X6 <- 3*runif(N)
mu <- c(-1,0,1,2)[as.numeric(X3)]

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] 4871.355

# 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