

# Boosting and Random Forests

ACM

October 26th, 2021

In this example we'll use some data on the number of carseats sold at 400 different stores.

```
library(tidyverse)
library(printr)
library(randomForest)
library(tidyverse)
library(MASS)
library(gbm)
library(tree)
library(ISLR)
library(rpart)
```

## Random Forests

?Carseats

Sales of Child Car Seats

Description:

A simulated data set containing sales of child car seats at 400 different stores.

Usage:

Carseats

```
Carseats <- Carseats %>% mutate(High = factor(1*I(Sales > 8)))
head(Carseats)
```

Sales	CompPrice	Income	Advertising	Population	Price	ShelveLoc	Age	Education	Urban	US	High
9.50	138	73	11	276	120	Bad	42	17	Yes	Yes	1
11.22	111	48	16	260	83	Good	65	10	Yes	Yes	1
10.06	113	35	10	269	80	Medium	59	12	Yes	Yes	1
7.40	117	100	4	466	97	Medium	55	14	Yes	Yes	0
4.15	141	64	3	340	128	Bad	38	13	Yes	No	0

Sales	CompPrice	Income	Advertising	Population	Price	ShelveLoc	Age	Education	Urban	US	High
10.81	124	113	13	501	72	Bad	78	16	No	Yes	1

```
?randomForest
```

## Classification and Regression with Random Forest

### Description:

'randomForest' implements Breiman's random forest algorithm (based on Breiman and Cutler's original Fortran code) for classification and regression. It can also be used in unsupervised mode for assessing proximities among data points.

### Usage:

```
## S3 method for class 'formula'
randomForest(formula, data=NULL, ..., subset, na.action=na.fail)
## Default S3 method:
randomForest(x, y=NULL, xtest=NULL, ytest=NULL, ntree=500,
             mtry=if (!is.null(y) && !is.factor(y))
               max(floor(ncol(x)/3), 1) else floor(sqrt(ncol(x))),
             weights=NULL,
             replace=TRUE, classwt=NULL, cutoff, strata,
             sampsize = if (replace) nrow(x) else ceiling(.632*nrow(x)),
             nodesize = if (!is.null(y) && !is.factor(y)) 5 else 1,
             maxnodes = NULL,
             importance=FALSE, localImp=FALSE, nPerm=1,
             proximity, oob.prox=proximity,
             norm.votes=TRUE, do.trace=FALSE,
             keep.forest=!is.null(y) && is.null(xtest), corr.bias=FALSE,
             keep.inbag=FALSE, ...)
## S3 method for class 'randomForest'
print(x, ...)
```

### Arguments:

**data:** an optional data frame containing the variables in the model.  
By default the variables are taken from the environment which 'randomForest' is called from.

**subset:** an index vector indicating which rows should be used. (NOTE: If given, this argument must be named.)

**na.action:** A function to specify the action to be taken if NAs are found. (NOTE: If given, this argument must be named.)

**x, formula:** a data frame or a matrix of predictors, or a formula describing the model to be fitted (for the 'print' method, an 'randomForest' object).

**y:** A response vector. If a factor, classification is assumed,

otherwise regression is assumed. If omitted, 'randomForest' will run in unsupervised mode.

xtest: a data frame or matrix (like 'x') containing predictors for the test set.

ytest: response for the test set.

ntree: Number of trees to grow. This should not be set to too small a number, to ensure that every input row gets predicted at least a few times.

mtry: Number of variables randomly sampled as candidates at each split. Note that the default values are different for classification ( $\sqrt{p}$  where  $p$  is number of variables in 'x') and regression ( $p/3$ )

weights: A vector of length same as 'y' that are positive weights used only in sampling data to grow each tree (not used in any other calculation)

replace: Should sampling of cases be done with or without replacement?

classwt: Priors of the classes. Need not add up to one. Ignored for regression.

cutoff: (Classification only) A vector of length equal to number of classes. The 'winning' class for an observation is the one with the maximum ratio of proportion of votes to cutoff. Default is  $1/k$  where  $k$  is the number of classes (i.e., majority vote wins).

strata: A (factor) variable that is used for stratified sampling.

sampsize: Size(s) of sample to draw. For classification, if sampsize is a vector of the length the number of strata, then sampling is stratified by strata, and the elements of sampsize indicate the numbers to be drawn from the strata.

nodesize: Minimum size of terminal nodes. Setting this number larger causes smaller trees to be grown (and thus take less time). Note that the default values are different for classification (1) and regression (5).

maxnodes: Maximum number of terminal nodes trees in the forest can have. If not given, trees are grown to the maximum possible (subject to limits by 'nodesize'). If set larger than maximum possible, a warning is issued.

importance: Should importance of predictors be assessed?

localImp: Should casewise importance measure be computed? (Setting this to 'TRUE' will override 'importance'.)

nPerm: Number of times the OOB data are permuted per tree for assessing variable importance. Number larger than 1 gives slightly more stable estimate, but not very effective. Currently only implemented for regression.

proximity: Should proximity measure among the rows be calculated?

oob.prox: Should proximity be calculated only on ‘out-of-bag’ data?

norm.votes: If ‘TRUE’ (default), the final result of votes are expressed as fractions. If ‘FALSE’, raw vote counts are returned (useful for combining results from different runs). Ignored for regression.

do.trace: If set to ‘TRUE’, give a more verbose output as ‘randomForest’ is run. If set to some integer, then running output is printed for every ‘do.trace’ trees.

keep.forest: If set to ‘FALSE’, the forest will not be retained in the output object. If ‘xtest’ is given, defaults to ‘FALSE’.

corr.bias: perform bias correction for regression? Note: Experimental. Use at your own risk.

keep.inbag: Should an ‘n’ by ‘ntree’ matrix be returned that keeps track of which samples are ‘in-bag’ in which trees (but not how many times, if sampling with replacement)

...: optional parameters to be passed to the low level function ‘randomForest.default’.

```
set.seed(2)
train=sample(1:nrow(Carseats), 267)

num_vars <- 1:10
tst.loss <- NULL
oob.loss <- NULL
for(m in num_vars){
  set.seed(2)
  RF_Carseats=randomForest(High~.-Sales, data =
                           Carseats,subset=train,ntree=1000,mtry=m)
  yhat.oob <- RF_Carseats$predicted
  yhat.bag <- predict(RF_Carseats,newdata=Carseats[-train,])
  tst.loss <- c(tst.loss,mean(yhat.bag==Carseats$High[-train]))
  oob.loss <- c(oob.loss, mean(Carseats$High[train] == yhat.oob))
}
data.frame(num_vars,oob.loss,tst.loss)
```

num_vars	oob.loss	tst.loss
1	0.7677903	0.8421053
2	0.7827715	0.8270677
3	0.8014981	0.8421053

num_vars	oob.loss	tst.loss
4	0.7902622	0.8345865
5	0.7865169	0.8496241
6	0.7790262	0.8646617
7	0.7752809	0.8571429
8	0.7827715	0.8571429
9	0.7790262	0.8796992
10	0.7715356	0.8872180

```
M <- num_vars[which.min(oob.loss)]

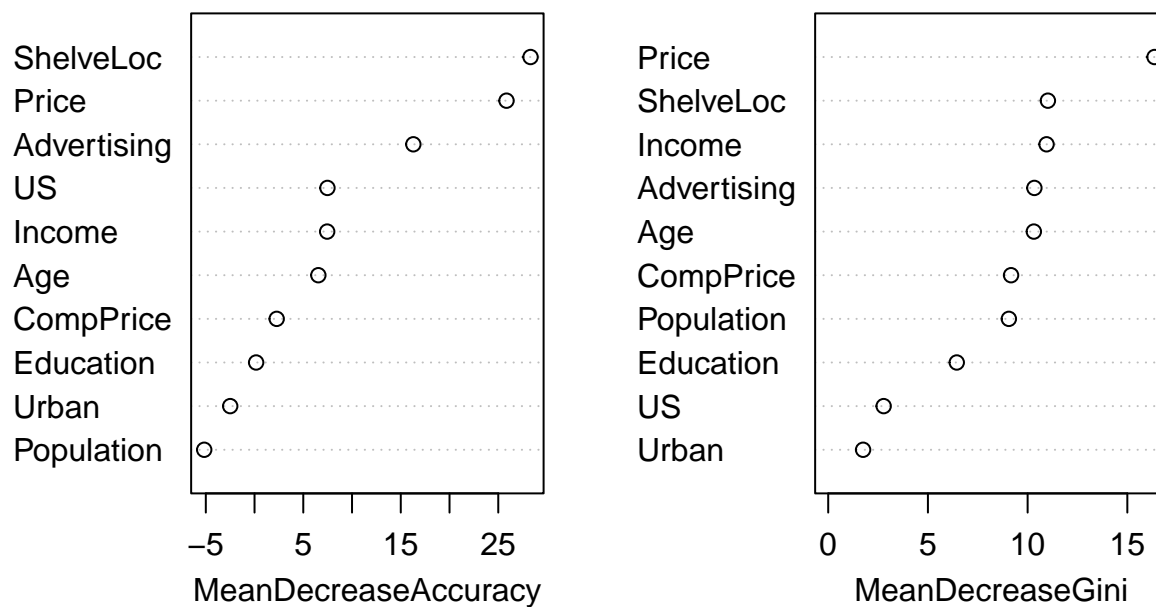
RF_Carseats=randomForest(High~.-Sales,data=Carseats,subset=train,
                           ntree = 1000, mtry=M, importance=TRUE)

importance(RF_Carseats)
```

	0	1	MeanDecreaseAccuracy	MeanDecreaseGini
CompPrice	1.257404	1.9562537	2.2717912	9.169253
Income	4.231956	7.2327258	7.4493482	10.949578
Advertising	10.318522	12.5515327	16.2886234	10.338101
Population	-2.987465	-4.5682403	-5.1635890	9.059153
Price	22.109692	19.6000217	25.8458926	16.364477
ShelveLoc	24.642655	22.3124208	28.3108198	11.015812
Age	4.872811	4.6954731	6.5438847	10.310997
Education	0.099139	0.1201581	0.1620056	6.446135
Urban	-1.514967	-1.8663463	-2.4970693	1.749225
US	4.976477	4.4587048	7.4759622	2.780967

```
varImpPlot(RF_Carseats)
```

## RF\_Carseats



## Fitting Regression Trees

Here we'll fit regression trees to the Boston housing dataset.

```
?Boston
```

Housing Values in Suburbs of Boston

Description:

The 'Boston' data frame has 506 rows and 14 columns.

Usage:

Boston

```
set.seed(1)
train = sample(1:nrow(Boston), nrow(Boston)/2)

tree.boston <- rpart(medv~., data = Boston[train,], method="anova",
                     control = list(cp = 0,xval=10, minsplit = 0))
printcp(tree.boston)
```

Regression tree:

```
rpart(formula = medv ~ ., data = Boston[train, ], method = "anova",
      control = list(cp = 0, xval = 10, minsplit = 0))
```

Variables actually used in tree construction:

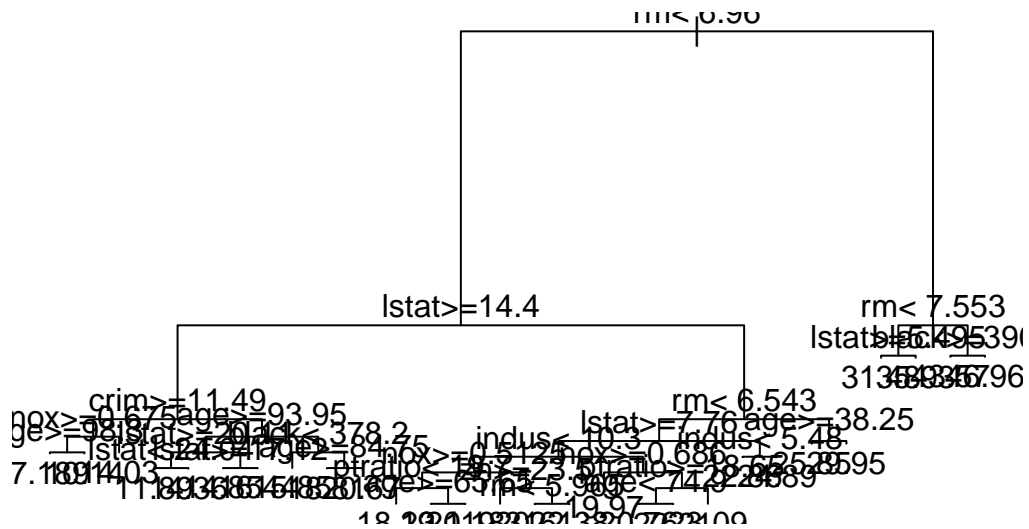
```
[1] age      black  crim   indus  lstat  nox      ptratio rm      zn
```

Root node error: 19448/253 = 76.869

n= 253

	CP	nsplit	rel error	xerror	xstd
1	0.55145333	0	1.000000	1.00246	0.126271
2	0.17610053	1	0.448547	0.48456	0.042763
3	0.05689532	2	0.272446	0.30320	0.030609
4	0.04093613	3	0.215551	0.27604	0.029893
5	0.03276814	4	0.174615	0.23875	0.028484
6	0.01048773	5	0.141847	0.21692	0.027290
7	0.00985230	6	0.131359	0.21760	0.027319
8	0.00817667	7	0.121507	0.21208	0.027800
9	0.00588100	8	0.113330	0.19448	0.025474
10	0.00512910	9	0.107449	0.18930	0.025913
11	0.00401073	10	0.102320	0.19029	0.026063
12	0.00224451	11	0.098309	0.19649	0.028806
13	0.00220749	12	0.096065	0.20285	0.028590
14	0.00219850	13	0.093857	0.20084	0.028143
15	0.00202124	14	0.091659	0.20094	0.028155
16	0.00197575	15	0.089637	0.20092	0.028214
17	0.00172550	16	0.087662	0.19970	0.028200
18	0.00112897	17	0.085936	0.19861	0.028171
19	0.00109924	18	0.084807	0.19841	0.028167
20	0.00101740	19	0.083708	0.19775	0.028115
21	0.00096903	20	0.082690	0.20045	0.028145
22	0.00082276	21	0.081721	0.19985	0.028141
23	0.00074870	22	0.080899	0.20096	0.028145
24	0.00055812	23	0.080150	0.20248	0.028171
25	0.00053719	24	0.079592	0.20440	0.028255
26	0.00049705	25	0.079055	0.20435	0.028256
27	0.00023769	26	0.078558	0.20507	0.028271
28	0.00000000	27	0.078320	0.20424	0.028223

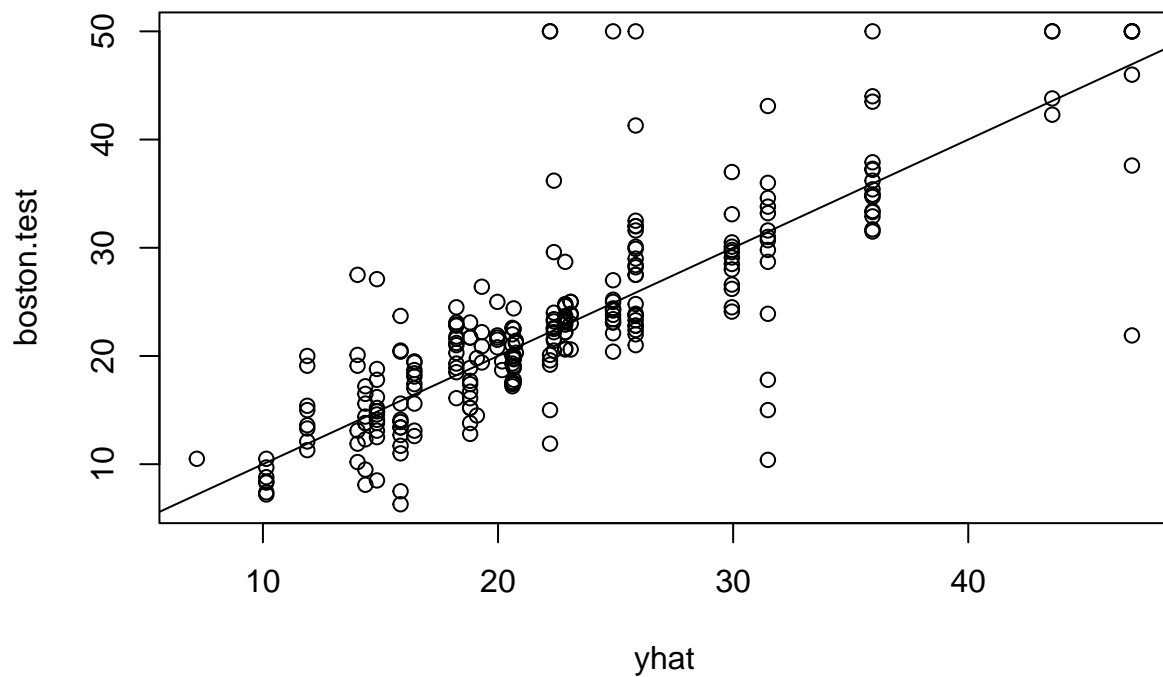
```
plot(tree.boston)
text(tree.boston,pretty=1)
```



```
tree.boston.cp <- as.data.frame(tree.boston$cptable)
cp <- tree.boston.cp$CP[which.min(tree.boston.cp$error)][1]
prune.boston=prune(tree.boston,cp=cp)

yhat=predict(tree.boston,newdata=Boston[-train,])
boston.test=Boston[-train,"medv"]
plot(yhat,boston.test)
abline(0,1)
```





```
mean((yhat-boston.test)^2)
```

```
[1] 31.70559
```

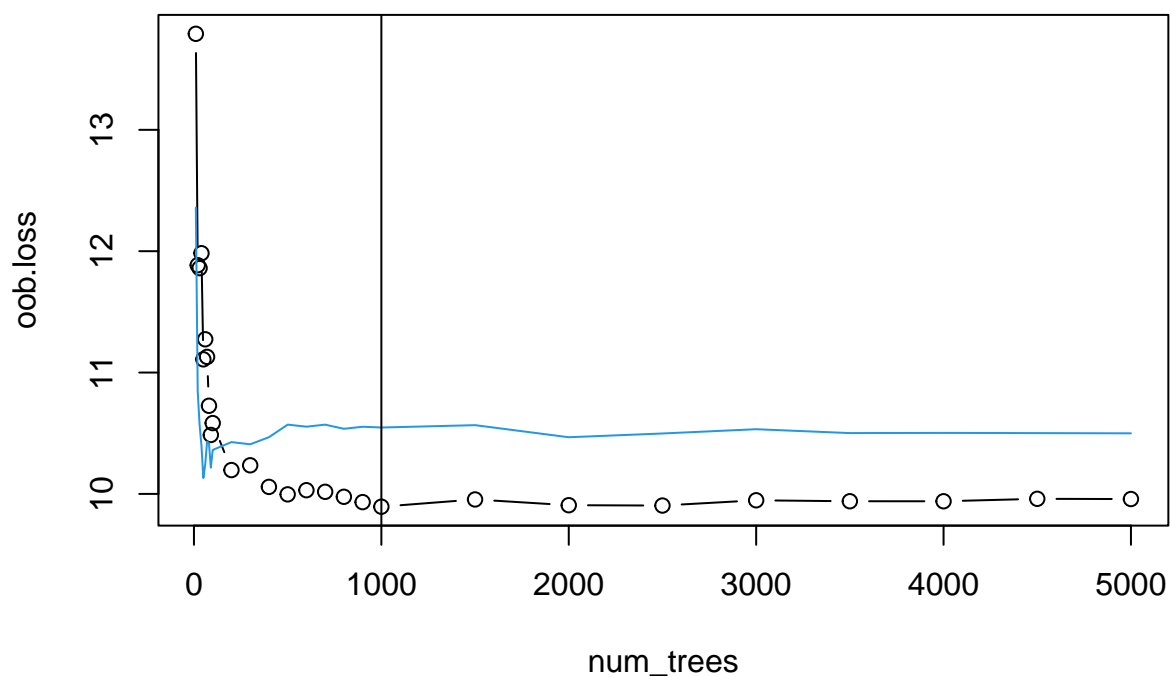
```
num_vars <- 1:10
tst.loss <- NULL
oob.loss <- NULL
for(m in num_vars){
  set.seed(1)
  bag.boston=randomForest(medv~.,data=Boston, subset = train,
                           mtry = m, ntree = 1000)
  yhat.oob <- bag.boston$predicted
  yhat.bag = predict(bag.boston,newdata=Boston[-train,])
  tst.loss <- c(tst.loss,mean((yhat.bag-Boston$medv[-train])^2))
  oob.loss <- c(oob.loss,mean((yhat.oob-Boston$medv[train])^2))
}
data.frame(num_vars,oob.loss,tst.loss)
```

num_vars	oob.loss	tst.loss
1	20.230660	25.84646
2	12.910341	19.57372
3	11.091572	18.58931
4	10.126007	18.72469

num_vars	oob.loss	tst.loss
5	10.000576	19.03545
6	9.914823	19.37815
7	9.894657	20.23230
8	10.400673	20.77523
9	10.353134	21.37730
10	10.475958	22.16682

```
M <- num_vars[ which.min(oob.loss)]

num_trees <- c(seq(10,90,10), seq(100,1000,100), seq(1500, 5000, 500))
tst.loss <- NULL
oob.loss <- NULL
for(m in num_trees){
  set.seed(1)
  bag.boston=randomForest(medv~.,data=Boston,subset=train,mtry=7,ntree=m)
  yhat.oob <- bag.boston$predicted
  yhat.bag = predict(bag.boston,newdata=Boston[-train,])
  tst.loss <- c(tst.loss,mean((yhat.bag-Boston$medv[-train])^2))
  oob.loss <- c(oob.loss,mean((yhat.oob-Boston$medv[train])^2))
}
plot(num_trees,oob.loss,type = "b")
lines(num_trees, tst.loss*mean(oob.loss)/mean(tst.loss)), col = 4)
abline(v = num_trees[which.min(oob.loss)])
```



```
T <- num_trees[ which.min(oob.loss)]

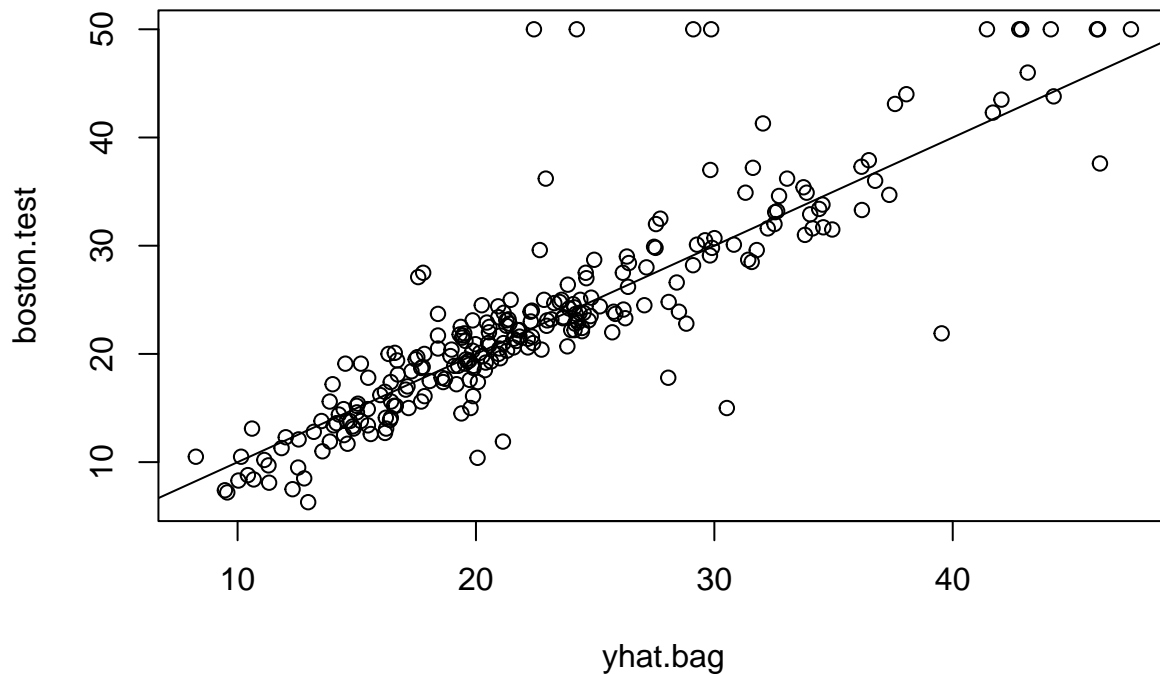
set.seed(1)
bag.boston=randomForest(medv~.,data=Boston,subset=train,
                        mtry=M, ntree=T, importance=TRUE)
bag.boston
```

Call:

```
randomForest(formula = medv ~ ., data = Boston, mtry = M, ntree = T,      importance = TRUE, subset =
              Type of random forest: regression
              Number of trees: 1000
No. of variables tried at each split: 7
```

```
      Mean of squared residuals: 10.12321
      % Var explained: 86.83
```

```
yhat.bag = predict(bag.boston,newdata=Boston[-train,])
plot(yhat.bag, boston.test)
abline(0,1)
```



```
mean((yhat.bag-Boston$medv[-train])^2)
```

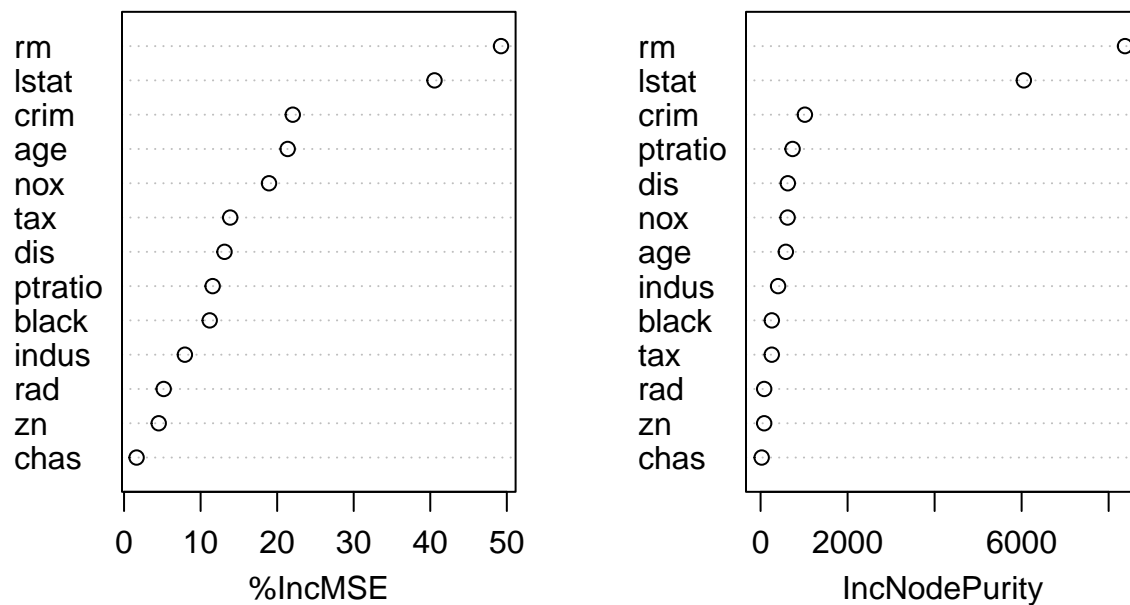
```
[1] 20.11327
```

```
importance(bag.boston)
```

	%IncMSE	IncNodePurity
crim	22.039957	1018.10793
zn	4.528856	82.69857
indus	7.945936	402.16913
chas	1.636318	25.55607
nox	18.934733	620.85250
rm	49.241099	8376.85386
age	21.373243	580.36449
dis	13.122556	626.30432
rad	5.175081	83.24623
tax	13.866610	256.99981
ptratio	11.573443	735.82927
black	11.187081	258.47575
lstat	40.551755	6051.13833

```
varImpPlot(bag.boston)
```

bag.boston



Now let's take a look at some other RF's with different tuning parameters

```
bag.boston=randomForest(medv~.,data=Boston,subset=train,mtry=13,ntree=1000)
yhat.bag = predict(bag.boston,newdata=Boston[-train,])
mean((yhat.bag-Boston$medv[-train])^2)
```

```
## [1] 23.48635
```

## Boosting

```
?gbm
```

Generalized Boosted Regression Modeling (GBM)

Description:

Fits generalized boosted regression models. For technical details, see the vignette: `'utils::browseVignettes("gbm")'`.

Usage:

```
gbm(  
  formula = formula(data),  
  distribution = "bernoulli",  
  data = list(),  
  weights,  
  var.monotone = NULL,  
  n.trees = 100,  
  interaction.depth = 1,  
  n.minobsinnode = 10,  
  shrinkage = 0.1,  
  bag.fraction = 0.5,  
  train.fraction = 1,  
  cv.folds = 0,  
  keep.data = TRUE,  
  verbose = FALSE,  
  class.stratify.cv = NULL,  
  n.cores = NULL  
)
```

Arguments:

**formula:** A symbolic description of the model to be fit. The formula may include an offset term (e.g. `y~offset(n)+x`). If `'keep.data = FALSE'` in the initial call to `'gbm'` then it is the user's responsibility to resupply the offset to `'gbm.more'`.

**distribution:** Either a character string specifying the name of the distribution to use or a list with a component `'name'` specifying the distribution and any additional parameters needed. If not specified, `'gbm'` will try to guess: if the response has only 2 unique values, `bernoulli` is assumed; otherwise, if the response is a factor, `multinomial` is assumed; otherwise, if the response has class `'Surv'`, `coxph` is assumed; otherwise, `gaussian` is assumed.

Currently available options are 'gaussian' (squared error), 'laplace' (absolute loss), 'tdist' (t-distribution loss), 'bernoulli' (logistic regression for 0-1 outcomes), 'huberized' (huberized hinge loss for 0-1 outcomes), 'adaboost' (the AdaBoost exponential loss for 0-1 outcomes), 'poisson' (count outcomes), 'coxph' (right censored observations), 'quantile', or 'pairwise' (ranking measure using the LambdaMart algorithm).

If quantile regression is specified, 'distribution' must be a list of the form 'list(name = "quantile", alpha = 0.25)' where 'alpha' is the quantile to estimate. The current version's quantile regression method does not handle non-constant weights and will stop.

If 'tdist' is specified, the default degrees of freedom is 4 and this can be controlled by specifying 'distribution = list(name = "tdist", df = DF)' where 'DF' is your chosen degrees of freedom.

If "pairwise" regression is specified, 'distribution' must be a list of the form 'list(name="pairwise",group=...,metric=...,max.rank=...)' ('metric' and 'max.rank' are optional, see below). 'group' is a character vector with the column names of 'data' that jointly indicate the group an instance belongs to (typically a query in Information Retrieval applications). For training, only pairs of instances from the same group and with different target labels can be considered. 'metric' is the IR measure to use, one of

list("conc") Fraction of concordant pairs; for binary labels, this is equivalent to the Area under the ROC Curve

: Fraction of concordant pairs; for binary labels, this is equivalent to the Area under the ROC Curve

list("mrr") Mean reciprocal rank of the highest-ranked positive instance

: Mean reciprocal rank of the highest-ranked positive instance

list("map") Mean average precision, a generalization of 'mrr' to multiple positive instances

: Mean average precision, a generalization of 'mrr' to multiple positive instances

list("ndcg:") Normalized discounted cumulative gain. The score is the weighted sum (DCG) of the user-supplied target values, weighted by  $\log(\text{rank}+1)$ , and normalized to the maximum achievable value. This is the default if the user did not specify a metric.

'ndcg' and 'conc' allow arbitrary target values, while binary targets 0,1 are expected for 'map' and 'mrr'. For 'ndcg' and 'mrr', a cut-off can be chosen using a positive integer parameter 'max.rank'. If left unspecified, all ranks are taken into account.

Note that splitting of instances into training and validation sets follows group boundaries and therefore only approximates the specified 'train.fraction' ratio (the same applies to cross-validation folds). Internally, queries are randomly shuffled before training, to avoid bias.

Weights can be used in conjunction with pairwise metrics, however it is assumed that they are constant for instances from the same group.

For details and background on the algorithm, see e.g. Burges (2010).

**data:** an optional data frame containing the variables in the model. By default the variables are taken from 'environment(formula)', typically the environment from which 'gbm' is called. If 'keep.data=TRUE' in the initial call to 'gbm' then 'gbm' stores a copy with the object. If 'keep.data=FALSE' then subsequent calls to 'gbm.more' must resupply the same dataset. It becomes the user's responsibility to resupply the same data at this point.

**weights:** an optional vector of weights to be used in the fitting process. Must be positive but do not need to be normalized. If 'keep.data=FALSE' in the initial call to 'gbm' then it is the user's responsibility to resupply the weights to 'gbm.more'.

**var.monotone:** an optional vector, the same length as the number of predictors, indicating which variables have a monotone increasing (+1), decreasing (-1), or arbitrary (0) relationship with the outcome.

**n.trees:** Integer specifying the total number of trees to fit. This is equivalent to the number of iterations and the number of basis functions in the additive expansion. Default is 100.

**interaction.depth:** Integer specifying the maximum depth of each tree (i.e., the highest level of variable interactions allowed). A value of 1 implies an additive model, a value of 2 implies a model with up to 2-way interactions, etc. Default is 1.

**n.minobsinnode:** Integer specifying the minimum number of observations in the terminal nodes of the trees. Note that this is the actual number of observations, not the total weight.

**shrinkage:** a shrinkage parameter applied to each tree in the expansion.

Also known as the learning rate or step-size reduction; 0.001 to 0.1 usually work, but a smaller learning rate typically requires more trees. Default is 0.1.

`bag.fraction`: the fraction of the training set observations randomly selected to propose the next tree in the expansion. This introduces randomness into the model fit. If `'bag.fraction' < 1` then running the same model twice will result in similar but different fits. `'gbm'` uses the R random number generator so `'set.seed'` can ensure that the model can be reconstructed. Preferably, the user can save the returned `'gbm.object'` using `'save'`. Default is 0.5.

`train.fraction`: The first `'train.fraction * nrow(data)'` observations are used to fit the `'gbm'` and the remainder are used for computing out-of-sample estimates of the loss function.

`cv.folds`: Number of cross-validation folds to perform. If `'cv.folds' > 1` then `'gbm'`, in addition to the usual fit, will perform a cross-validation, calculate an estimate of generalization error returned in `'cv.error'`.

`keep.data`: a logical variable indicating whether to keep the data and an index of the data stored with the object. Keeping the data and index makes subsequent calls to `'gbm.more'` faster at the cost of storing an extra copy of the dataset.

`verbose`: Logical indicating whether or not to print out progress and performance indicators (`'TRUE'`). If this option is left unspecified for `'gbm.more'`, then it uses `'verbose'` from `'object'`. Default is `'FALSE'`.

`class.stratify.cv`: Logical indicating whether or not the cross-validation should be stratified by class. Defaults to `'TRUE'` for `'distribution = "multinomial"'` and is only implemented for `'"multinomial"'` and `'"bernoulli"'`. The purpose of stratifying the cross-validation is to help avoiding situations in which training sets do not contain all classes.

`n.cores`: The number of CPU cores to use. The cross-validation loop will attempt to send different CV folds off to different cores. If `'n.cores'` is not specified by the user, it is guessed using the `'detectCores'` function in the `'parallel'` package. Note that the documentation for `'detectCores'` makes clear that it is not failsafe and could return a spurious number of available cores.

`?gbm.object`

Generalized Boosted Regression Model Object

Description:



These are objects representing fitted 'gbm's.

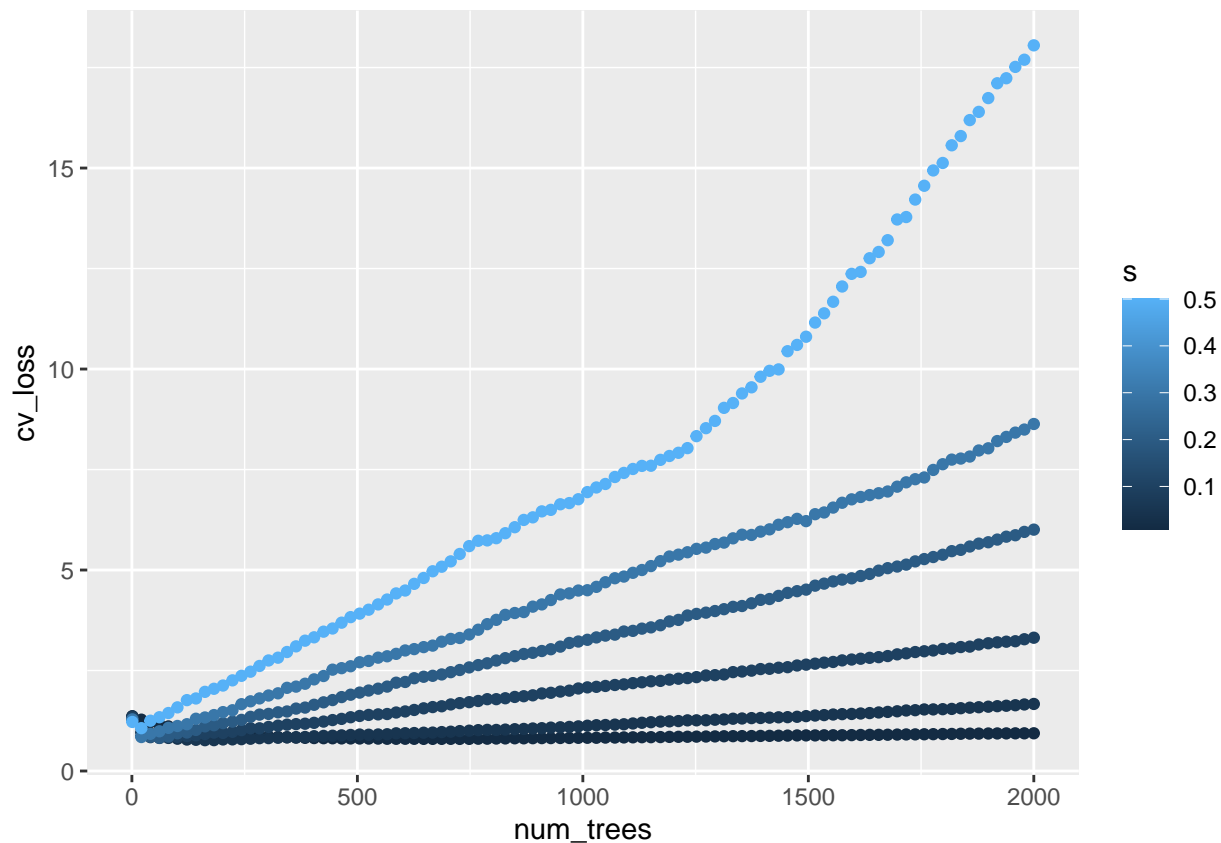
```
set.seed(2)
train=sample(1:nrow(Carseats), 267)
set.seed(1)

Carseats <- Carseats %>% mutate(High = 1*I(Sales > 8) )
shrink_vec <- c(0.01,0.05,0.1,0.2,0.3,0.5)
num_trees <- floor(seq(1,2000,length.out = 100))
cv.loss <- NULL
for(s in shrink_vec){
  boost.carseats=gbm(High~.-Sales,data=Carseats[train,],
                     distribution = "bernoulli",
                     n.trees = 5000,
                     interaction.depth = 4,
                     shrinkage = s,
                     cv.folds = 10)

  tst.loss <- NULL
  for(m in num_trees){
    yhat.bag = predict(boost.carseats, newdata=Carseats[-train,],
                       n.trees = m, type = "response")
    tst.loss <- c(tst.loss,mean((yhat.bag-Carseats$High[-train])^2))
  }
  t_cv.loss <- boost.carseats$cv.error[num_trees]
  cv.loss <- rbind(cv.loss,
                  cbind(s, num_trees, t_cv.loss, tst.loss)
                  )
}

cv.loss <- data.frame(cv.loss)
names(cv.loss) <- c("s", "num_trees", "cv_loss", "tst_loss")

ggplot(data = cv.loss, aes(num_trees, cv_loss, col = s)) +
  geom_point()
```

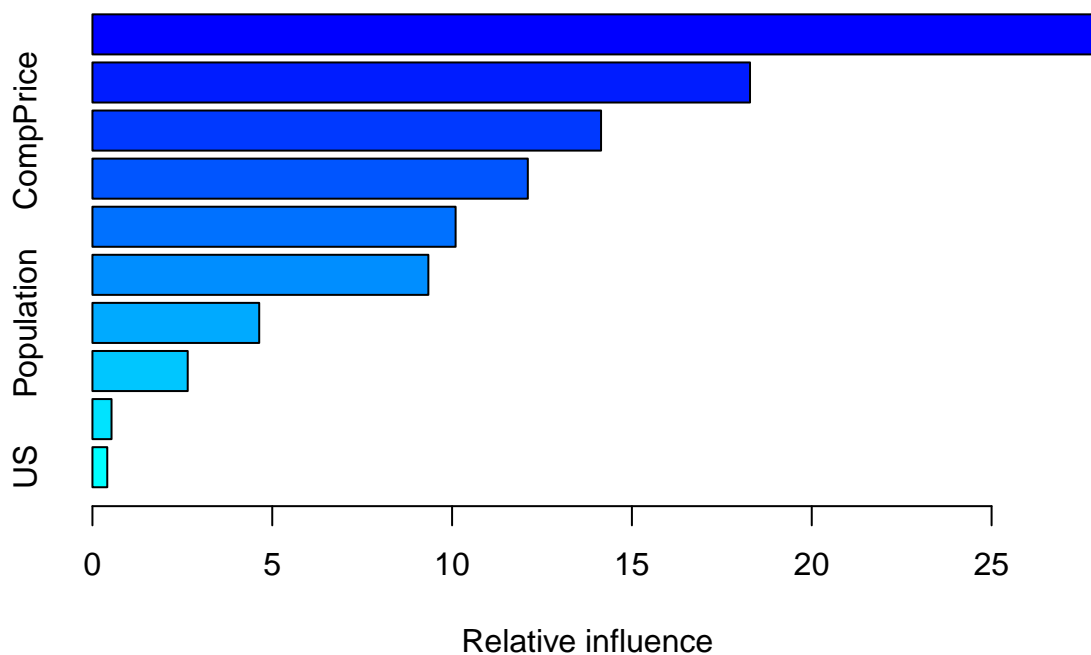


```
cv.loss[c(which.min(cv.loss$cv_loss),which.min(cv.loss$tst_loss)),]
```

	s	num_trees	cv_loss	tst_loss
109	0.05	162	0.7676945	0.0795636
113	0.05	243	0.7928936	0.0723198

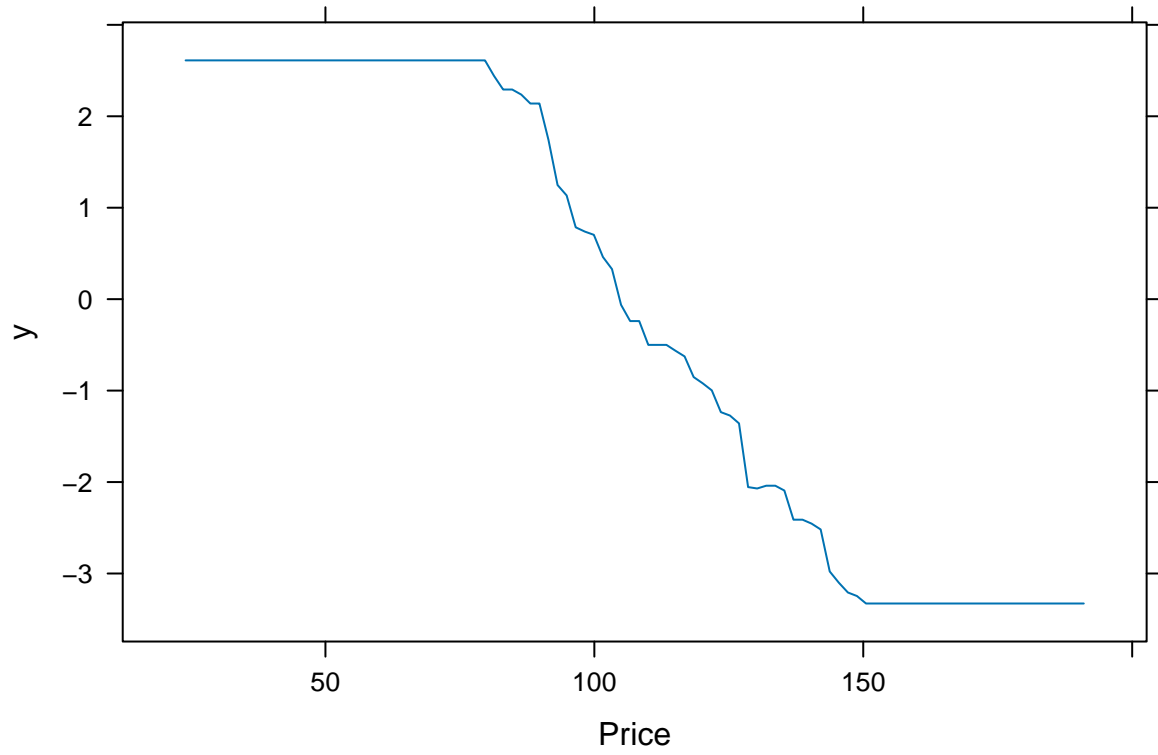
```
M <- cv.loss$num_trees[which.min(cv.loss$cv_loss)]
s <- cv.loss$s[which.min(cv.loss$cv_loss)]

boost.carseats=gbm(High ~.-Sales, data = Carseats[train,],
  distribution = "bernoulli", n.trees=M,
  shrinkage = s,
  interaction.depth=4)
summary(boost.carseats)
```

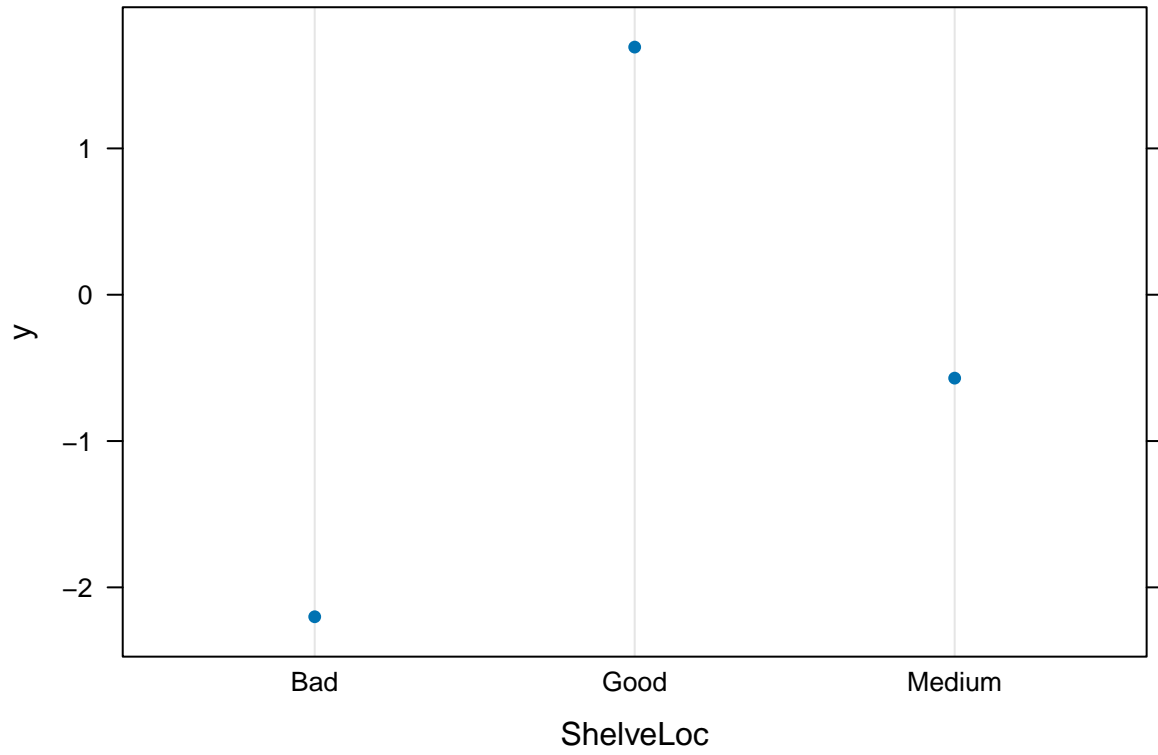


	var	rel.inf
Price	Price	27.8023189
ShelveLoc	ShelveLoc	18.2836487
CompPrice	CompPrice	14.1416004
Advertising	Advertising	12.1054488
Age	Age	10.0981940
Income	Income	9.3399932
Population	Population	4.6370510
Education	Education	2.6491189
Urban	Urban	0.5306436
US	US	0.4119826

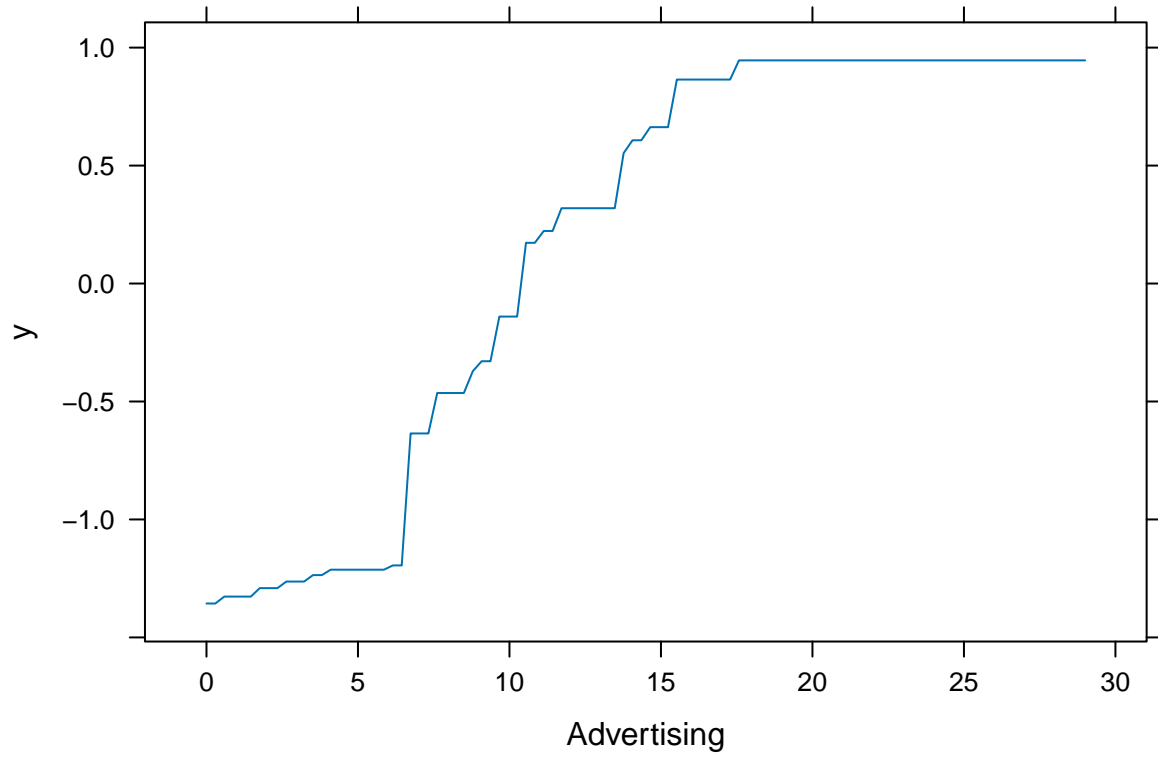
```
par(mfrow=c(2,2))
plot(boost.carseats,i="Price")
```



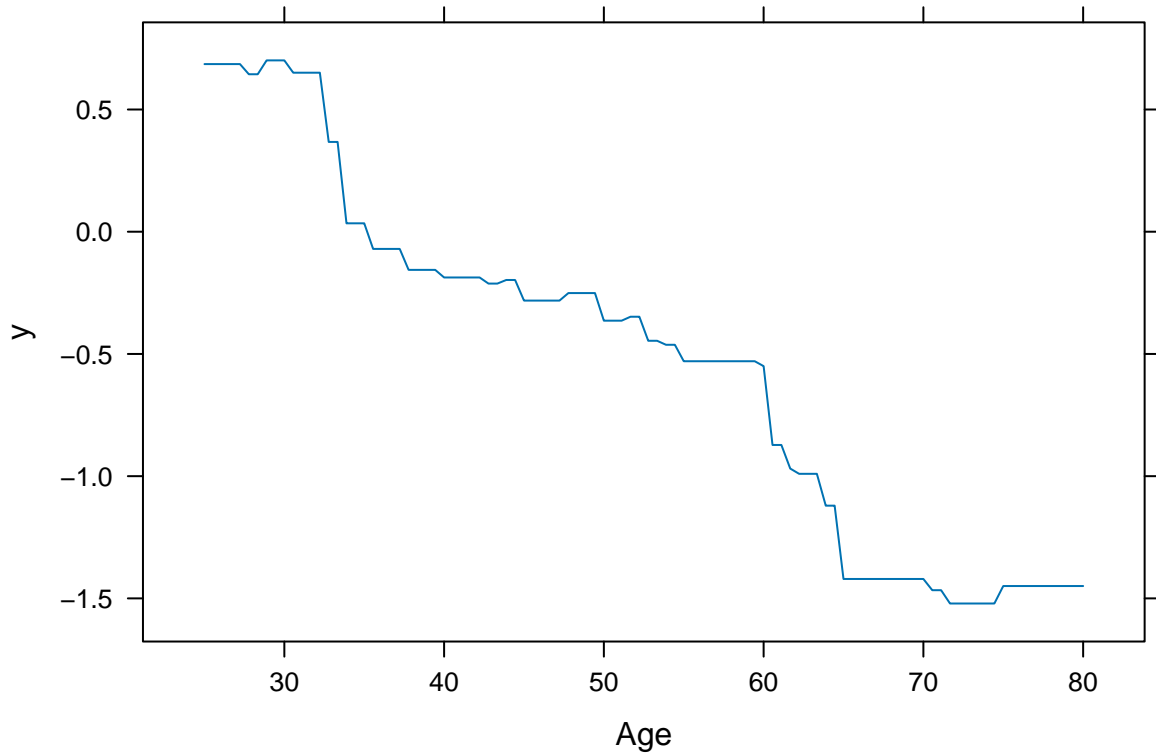
```
plot(boost.carseats,i="ShelveLoc")
```



```
plot(boost.carseats,i="Advertising")
```



```
plot(boost.carseats,i="Age")
```

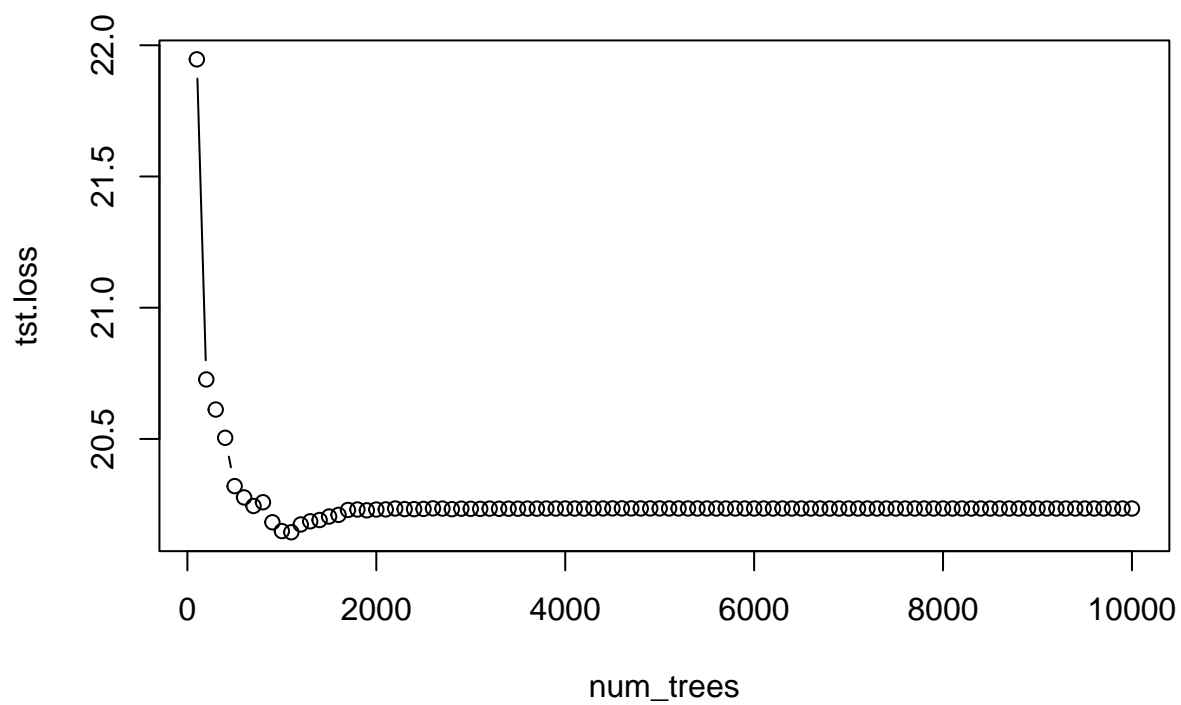


```
yhat.boost=predict( boost.carseats, newdata=Carseats[-train,],
                     n.trees = M, type = "response")
mean((yhat.boost-Carseats$High[-train])^2)
```

```
## [1] 0.0758269
```

```
set.seed(1)
train = sample(1:nrow(Boston), nrow(Boston)/2)
set.seed(1)

num_trees <- floor(seq(100,10000,length.out = 100))
tst.loss <- NULL
boost.boston=gbm( medv~., data=Boston[train,], distribution="gaussian",
                  n.trees=max(num_trees), interaction.depth=5)
for(m in num_trees){
  yhat.bag = predict(boost.boston,newdata=Boston[-train,],n.trees = m,
                     type = "response")
  tst.loss <- c(tst.loss,mean((yhat.bag-Boston$medv[-train])^2))
}
plot(num_trees,tst.loss,type = "b")
```

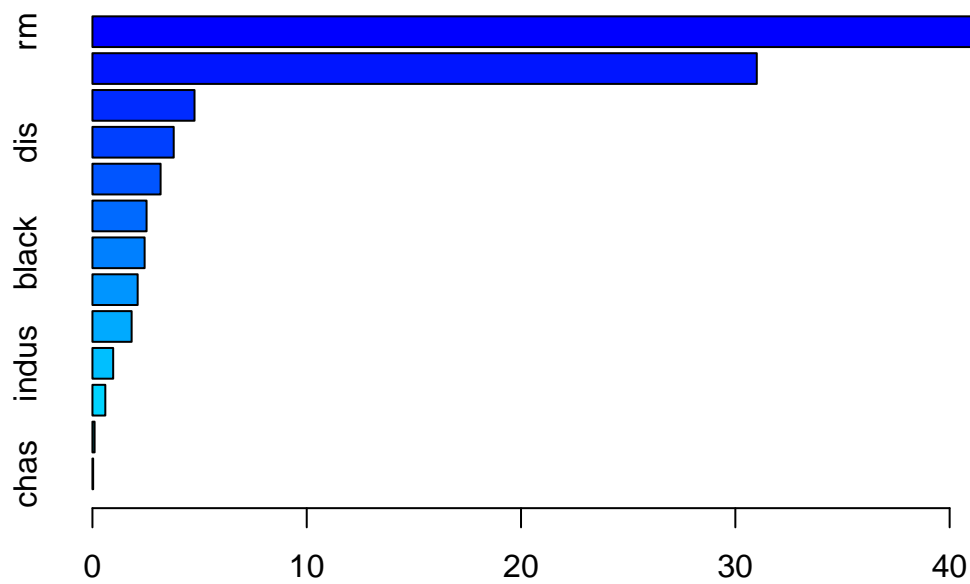


```
M <- num_trees[which.min(tst.loss)]  
M
```

```
## [1] 1100
```

```
boost.boston=gbm(medv~., data=Boston[train,], distribution="gaussian",  
                  n.trees=M, interaction.depth=5)  
summary(boost.boston)
```

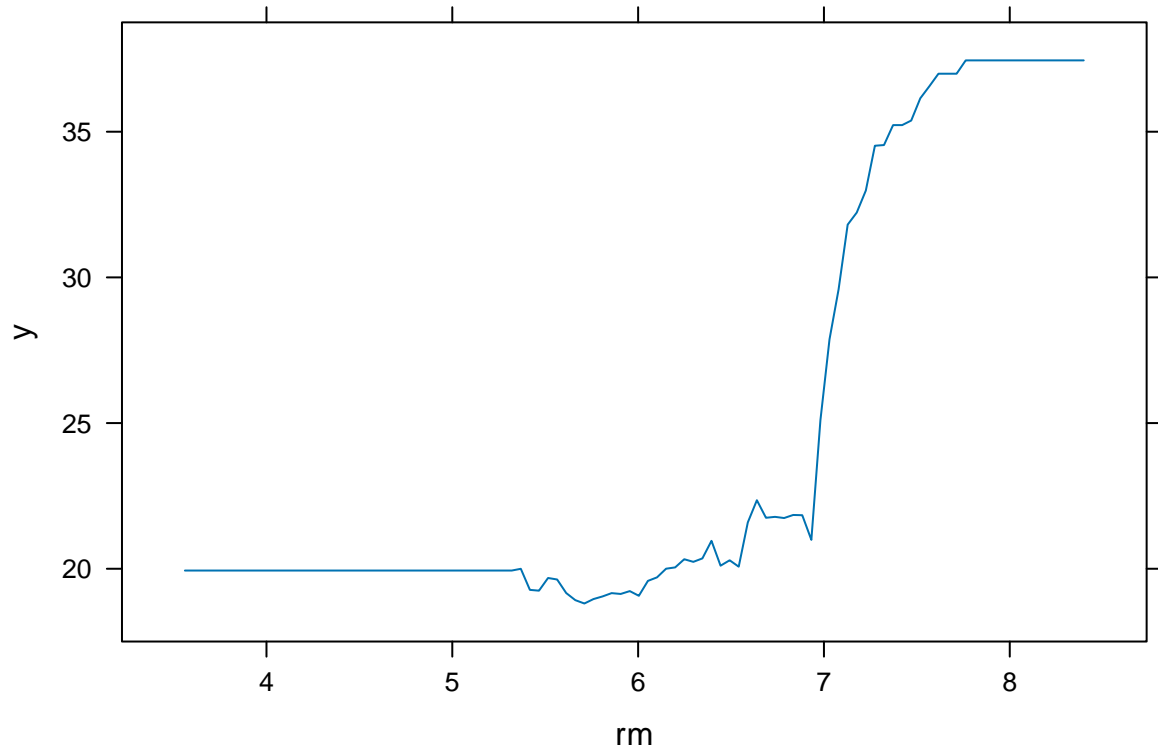




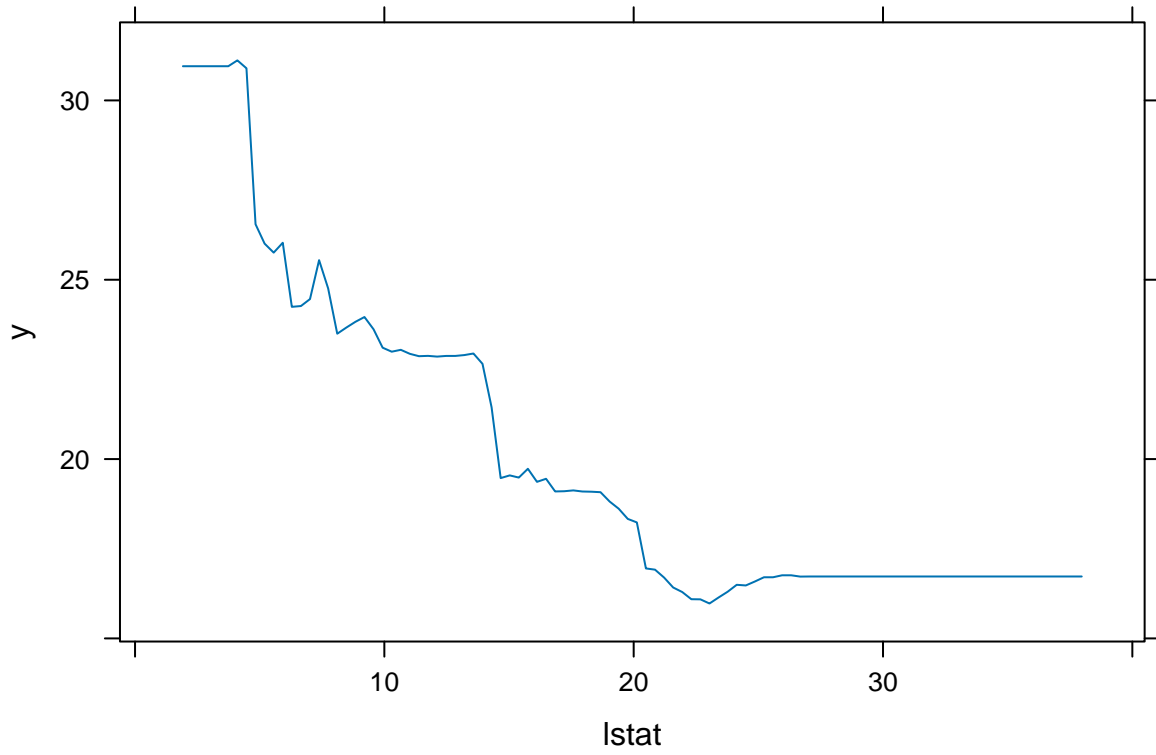
Relative influence

	var	rel.inf
rm	rm	46.6601217
lstat	lstat	31.0014372
crim	crim	4.7628832
dis	dis	3.7922972
nox	nox	3.1805425
age	age	2.5260908
black	black	2.4348971
ptratio	ptratio	2.1110133
tax	tax	1.8287830
indus	indus	0.9675263
rad	rad	0.6007242
zn	zn	0.1004767
chas	chas	0.0332068

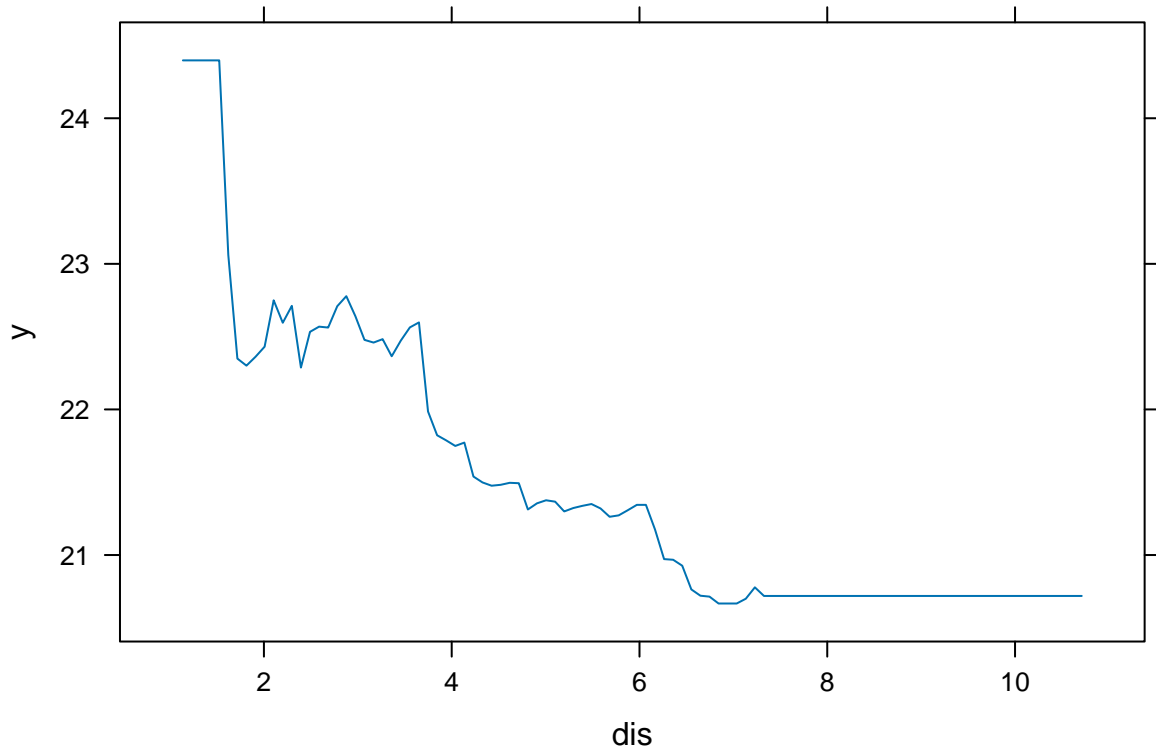
```
par(mfrow=c(2,2))
plot(boost.boston,i="rm")
```



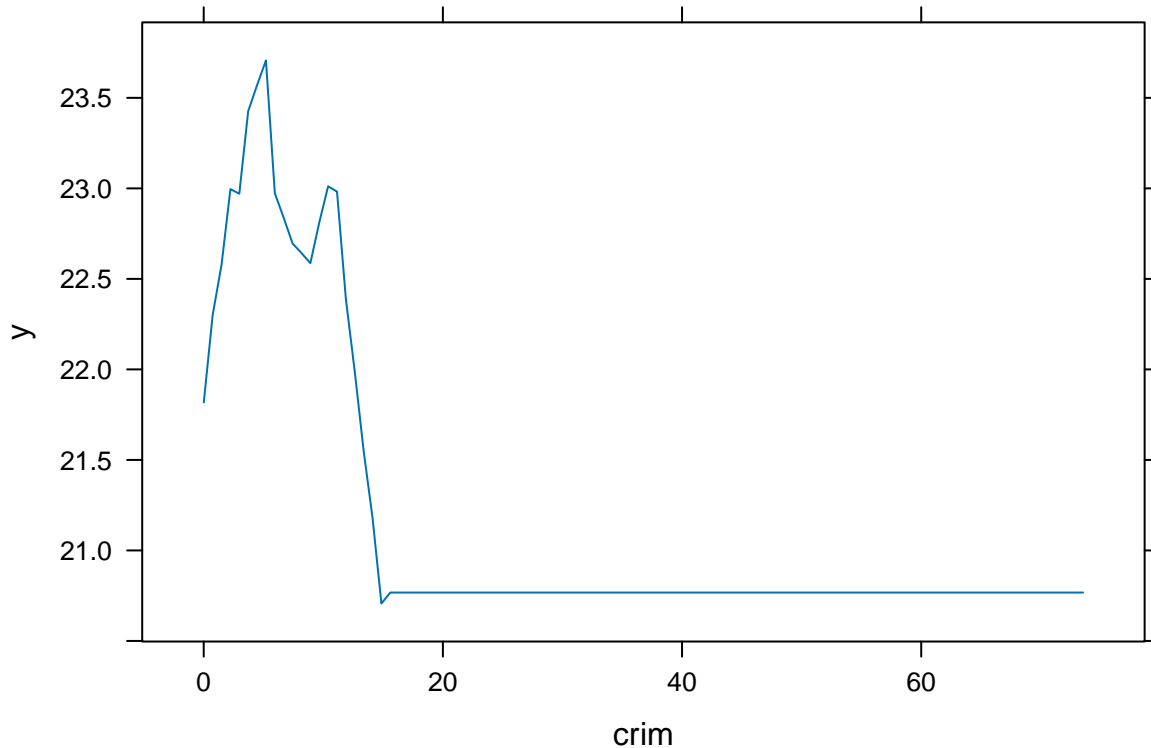
```
plot(boost.boston,i="lstat")
```



```
plot(boost.boston,i="dis")
```



```
plot(boost.boston,i="crim")
```



```
yhat.boost=predict(boost.boston,newdata=Boston[-train,], n.trees = M,
                    type = "response")
mean((yhat.boost-boston.test)^2)
```

```
## [1] 19.19356
```

```
boost.boston=gbm(medv~.,data=Boston[train,], distribution="gaussian",
                 n.trees=M, interaction.depth=5, shrinkage=0.2, verbose=F)
yhat.boost=predict(boost.boston,newdata=Boston[-train,],n.trees=M,type = "response")
mean((yhat.boost-boston.test)^2)
```

```
## [1] 19.69013
```

```
set.seed(2)
train=sample(1:nrow(Boston), 267)
set.seed(1)

shrink_vec <- c(0.01,0.05,0.1,0.2,0.3,0.5)
num_trees <- floor(seq(100,10000,length.out = 100))
cv.loss <- NULL
for(s in shrink_vec){
  boost.boston=gbm(medv~.,data=Boston[train,],
                  distribution="gaussian",
                  n.trees = max(num_trees),
```

```

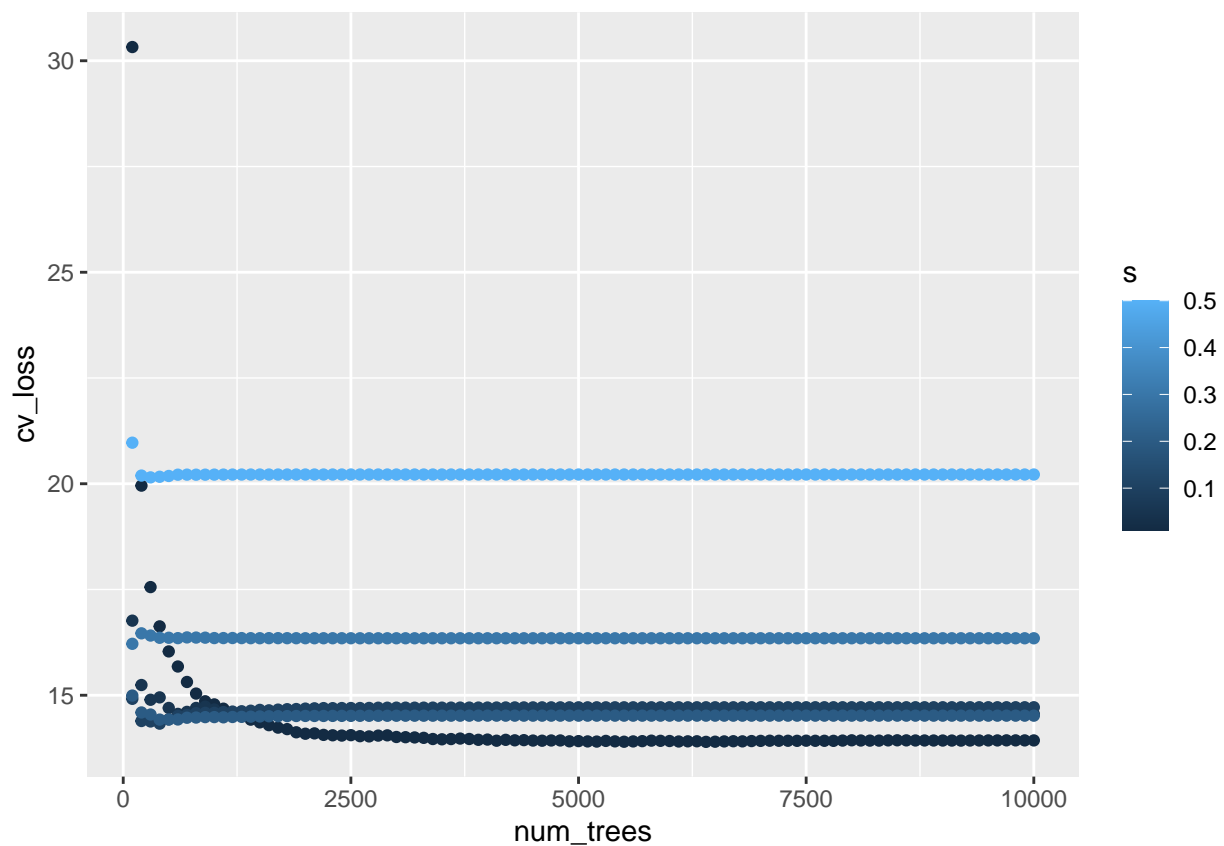
        interaction.depth = 5,
        shrinkage = s,
        cv.folds = 10)

tst.loss <- NULL
for(m in num_trees){
  yhat.bag = predict(boost.boston,newdata=Boston[-train,],
                      n.trees = m,
                      type = "response")
  tst.loss <- c(tst.loss,mean((yhat.bag-Boston$medv[-train])^2))
}
t_cv.loss <- boost.boston$cv.error[num_trees]
cv.loss <- rbind(cv.loss,
                 cbind(s, num_trees, t_cv.loss, tst.loss)
                )
}

cv.loss <- data.frame(cv.loss)
names(cv.loss) <- c("s", "num_trees","cv_loss", "tst_loss")

ggplot(data = cv.loss, aes(num_trees, cv_loss, col = s)) +
  geom_point()

```



```

cv.loss[c(which.min(cv.loss$cv_loss),which.min(cv.loss$tst_loss)),]

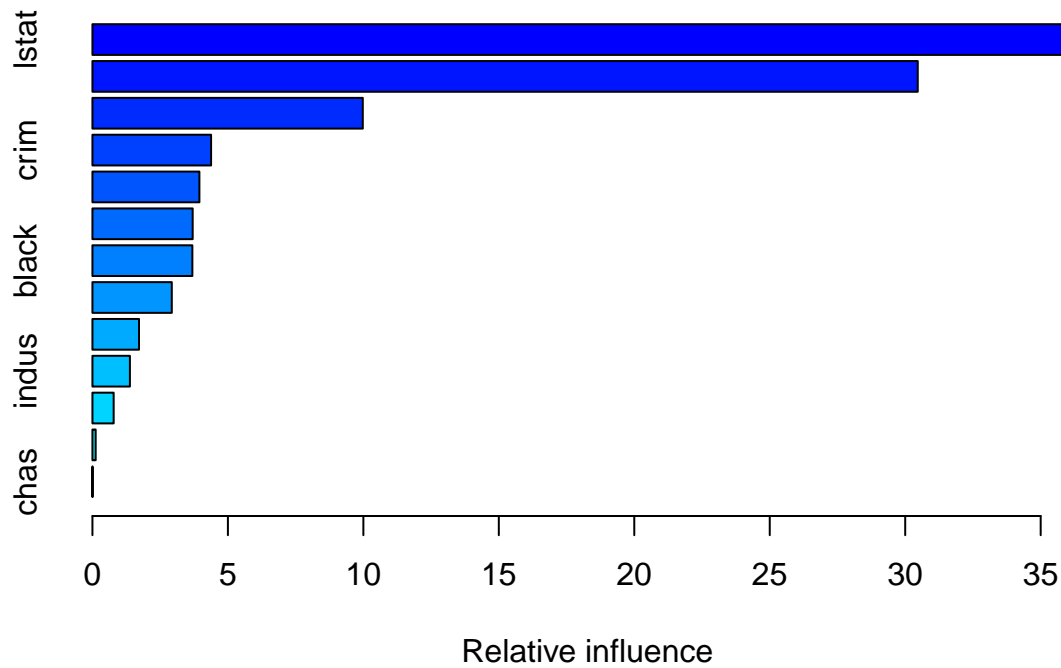
```

	s	num_trees	cv_loss	tst_loss
64	0.01	6400	13.89766	9.116855
23	0.01	2300	14.05345	8.980604

```
M <- cv.loss$num_trees[which.min(cv.loss$cv_loss)]
s <- cv.loss$s[which.min(cv.loss$cv_loss)]
```

```
boost.boston=gbm(medv~., data=Boston[train,],
  distribution="gaussian",
  n.trees=M,
  shrinkage = s,
  interaction.depth=5)
```

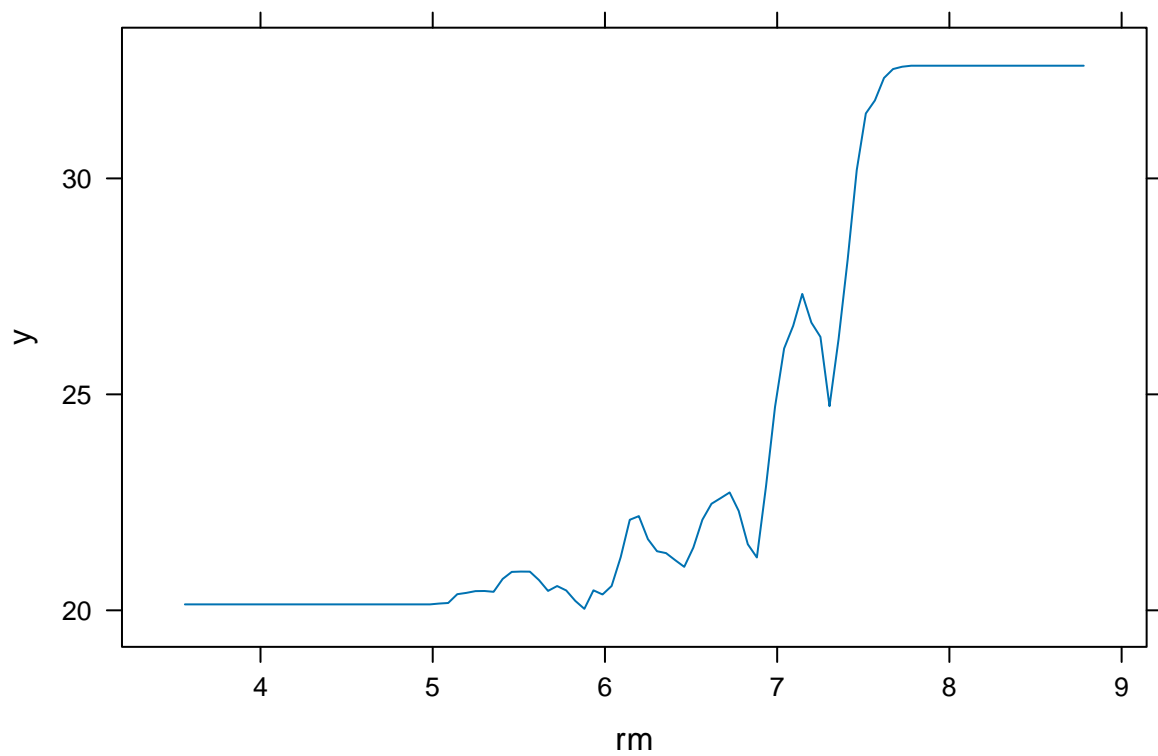
```
summary(boost.boston)
```



	var	rel.inf
lstat	lstat	36.9077964
rm	rm	30.4624775
dis	dis	9.9783433
crim	crim	4.3794315
nox	nox	3.9480219
age	age	3.6998334

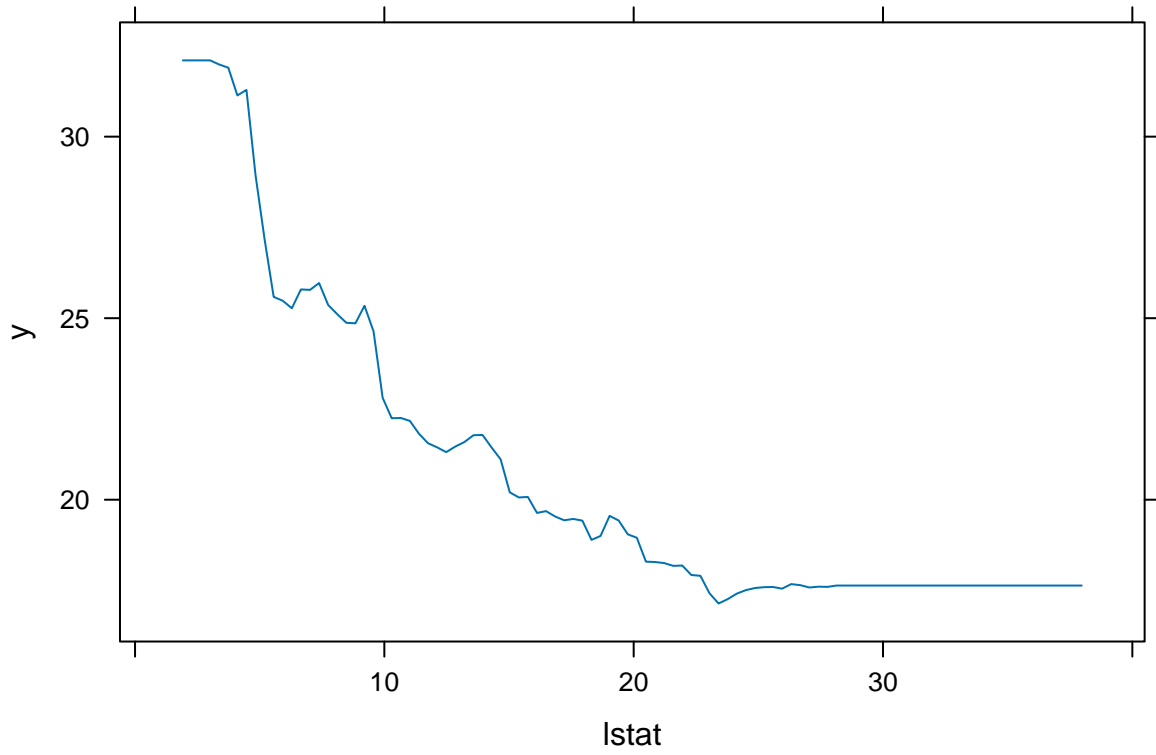
	var	rel.inf
black	black	3.6877496
ptratio	ptratio	2.9296837
tax	tax	1.7184586
indus	indus	1.3830188
rad	rad	0.7834752
zn	zn	0.1188686
chas	chas	0.0028415

```
par(mfrow=c(2,2))
plot(boost.boston,i="rm")
```

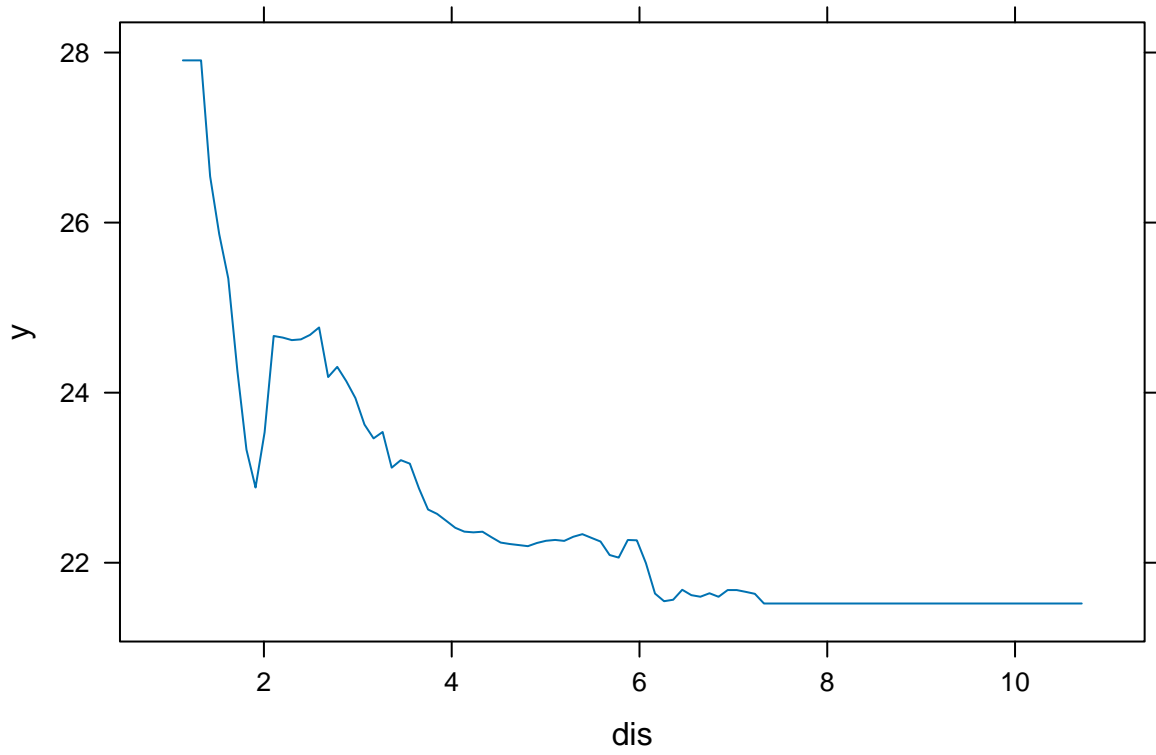


```
plot(boost.boston,i="lstat")
```

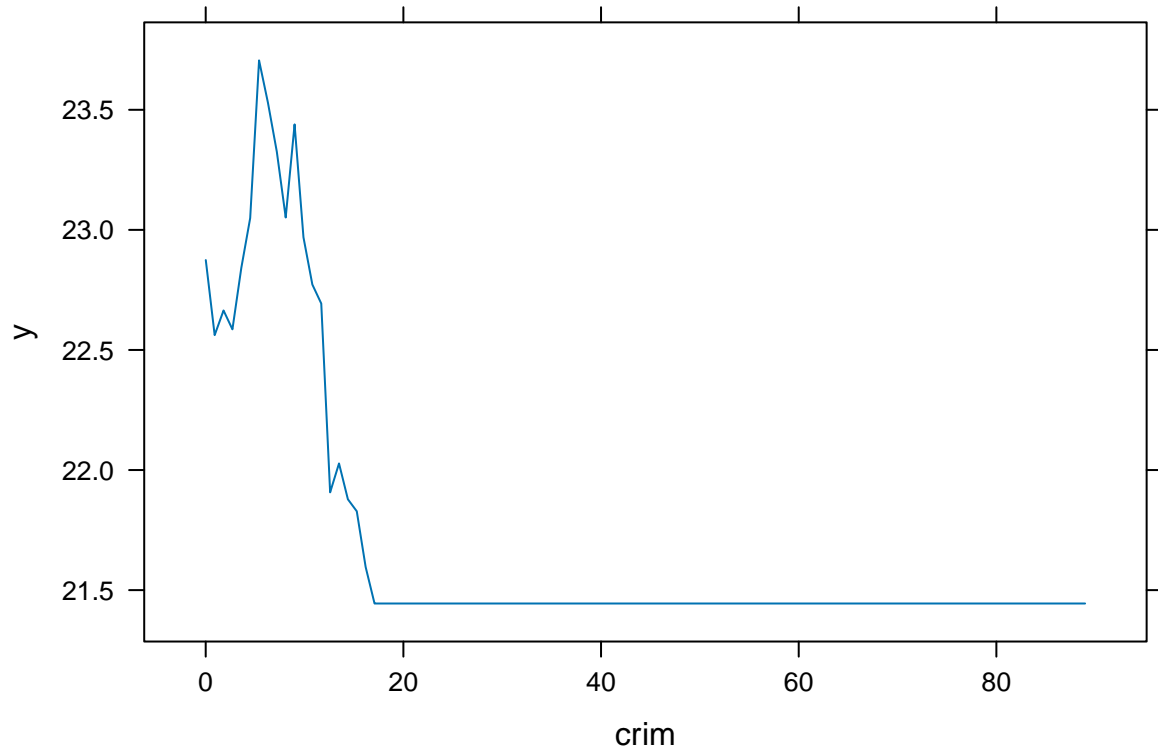




```
plot(boost.boston,i="dis")
```



```
plot(boost.boston,i="crim")
```



```
yhat.boost=predict(boost.boston,newdata=Boston[-train,], n.trees = M,
                    type = "response")
mean((yhat.boost-boston.test)^2)
```

```
## Warning in yhat.boost - boston.test: longer object length is not a multiple of
## shorter object length
```

```
## [1] 107.816
```