Trees, Random Forsets, Boosting for Continuous Variable Prediction

David Dobor

Fitting Regression Trees

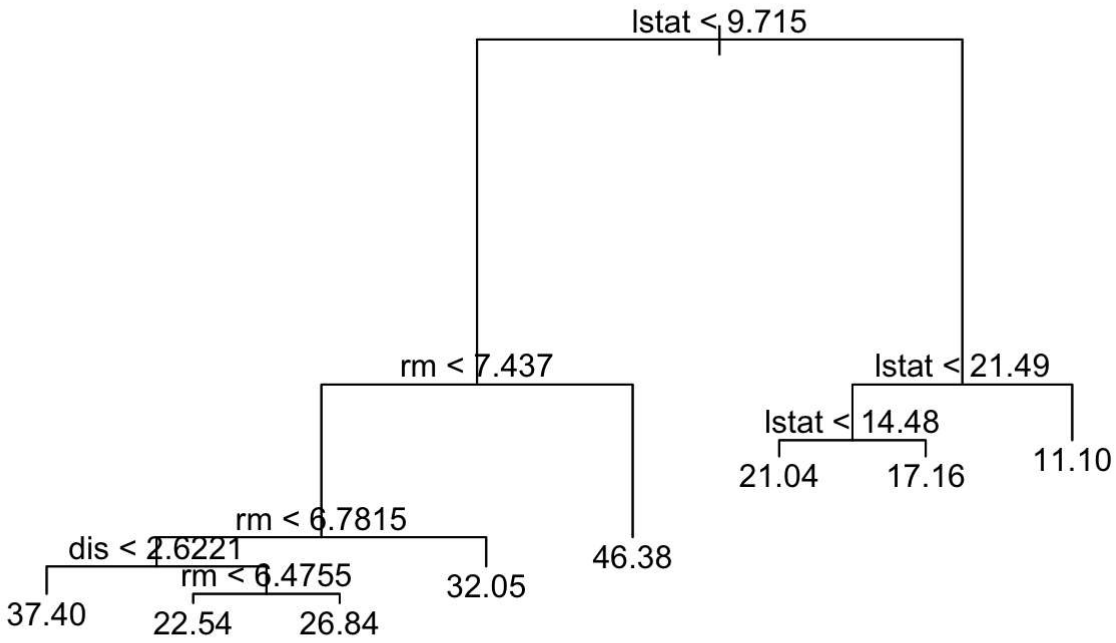
- We fit a regression tree to the `Boston Housing Data`, which is available at UCI machine learning repository (https://archive.ics.uci.edu/ml/datasets/Housing). The data is also available through the `MASS` package in `R` and has 14 features (columns) and 506 observations (rows).
- Variable to predict: `MEDV` (median value of owner-occupied homes in $1000s). Features include `CRIM` (per capita crime rate), `DIS` (distance to Boston employment centers), `RM` (average number of rooms per dwelling), `LSTAT` (percent of population with lower socio-economic status), among others.
- Split data 50-50 into training, test sets.

First, we fit the regression tree model on the training data only and plot the tree.

```
library(MASS)
set.seed(1)
library(tree)
set.seed(1)
train = sample(1:nrow(Boston), nrow(Boston)/2)
tree.boston=tree(medv~.,Boston,subset=train)
summary(tree.boston)
```

```
##
## Regression tree:
## tree(formula = medv ~ ., data = Boston, subset = train)
## Variables actually used in tree construction:
## [1] "lstat" "rm"    "dis"
## Number of terminal nodes:  8
## Residual mean deviance:  12.65 = 3099 / 245
## Distribution of residuals:
##      Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
## -14.10000  -2.04200  -0.05357   0.00000   1.96000  12.60000
```

```
plot(tree.boston)
text(tree.boston,pretty=0)
```



Some observations:

- The tree grown to full depth has 8 leaves and only three of the variables (`lstat` , `rm` and `dis`) have been used to construct this tree.
- The *deviance* reported here is simply the *sum of squared errors* for the tree.
- The `lstat` variable measures the percentage of individuals with lower socioeconomic status. The tree shows that higher values of `lstat` correspond to lower house values.
- The tree predicts a median house price of $46, 380 for larger homes (`rm >= 7.437`) in which residents have higher socio-economic status (`lstat < 9.715`).

Since the tree was grown to full depth, it may be too variable (i.e. has relatively high variance and low bias and may be overfitting the data).

We now **use 10-fold cross validation** (using `cv.tree()` function) in order to determine the optimal level of tree complexity. This will help us decide whether pruning the tree will improve performance.

`cv.tree()` function reports the number of terminal nodes of each tree considered (in variable `size` as shown in next output) as well as the corresponding error rate and the value of the cost-comlexity parameter [explain cost-complexity parameter].

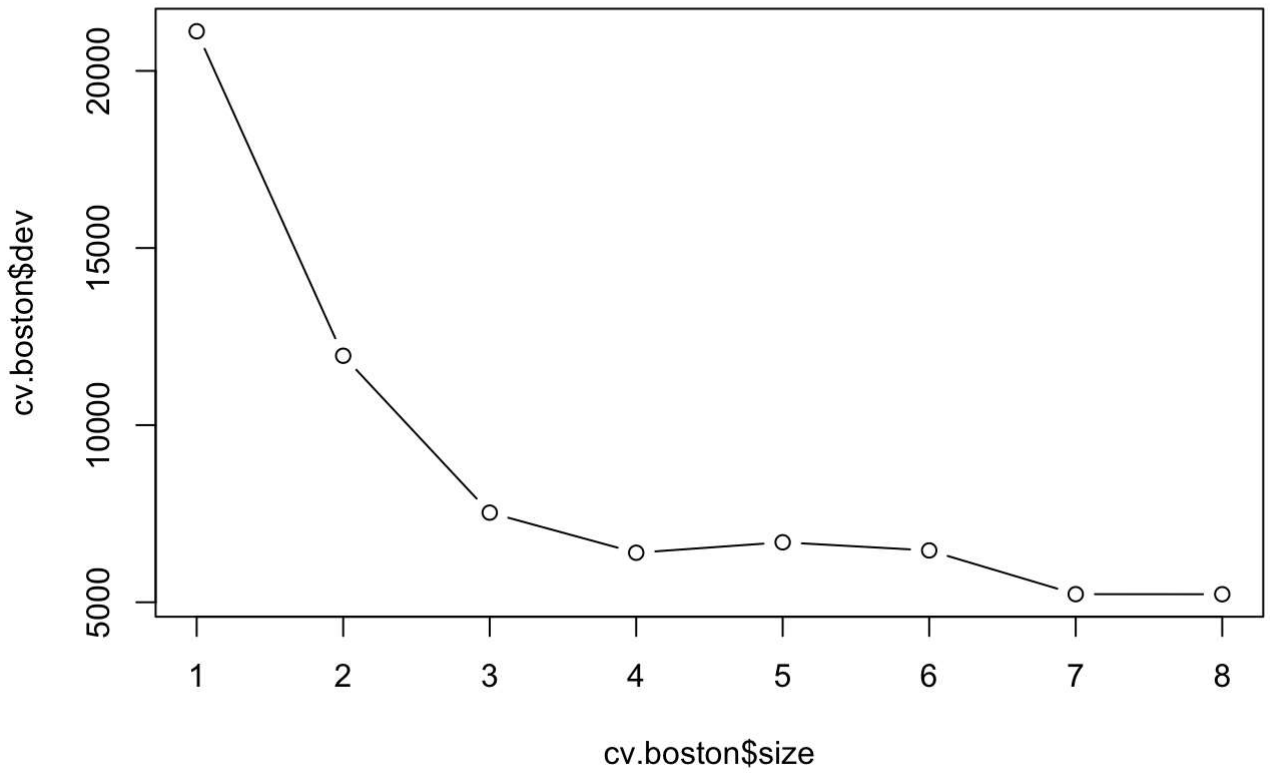
```
cv.boston=cv.tree(tree.boston); cv.boston
```

```
## $size
## [1] 8 7 6 5 4 3 2 1
##
## $dev
## [1]  5226.322  5228.360  6462.626  6692.615  6397.438  7529.846 11958.691
## [8] 21118.139
##
## $k
## [1]      -Inf  255.6581  451.9272  768.5087  818.8885 1559.1264 4276.5803
## [8] 9665.3582
##
## $method
## [1] "deviance"
##
## attr(,"class")
## [1] "prune"         "tree.sequence"
```

Note that `dev` in the above output corresponds to the cross-validation error. The lowest `dev` corresponds to the tree with 8 leaves.

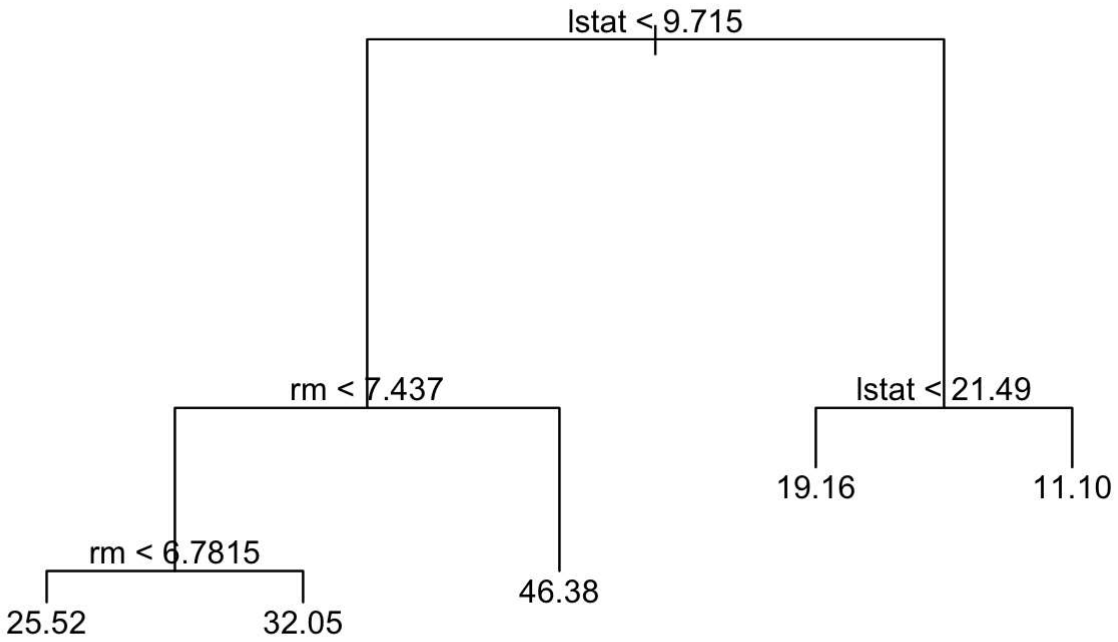
We can also see this in the following plot.

```
plot(cv.boston$size,cv.boston$dev,type='b')
```



Although the most complex tree is selected by cross-validation (the lowest error rate corresponds to the most complex tree with 8 leaves), if we wanted to prune the tree, we would do it as follows, using the `prune.tree()` function.

```
prune.boston=prune.tree(tree.boston,best=5)
#summary(prune.boston)
#cv.tree(tree.boston,,prune.tree)$dev
plot(prune.boston)
text(prune.boston,pretty=0)
```



But ultimately, we go with the cross-validation results and use the unpruned tree to make predictions on the test set.

```
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] 25.04559
```

So the test set `MSE` for the regression tree is 25.05, with its square root around 5.005, meaning that this model gives predictions that are within around $5, 005 of the true median home value.

Now we look to other techniques, like random forests and boosting, to see if better results can be obtained.

Random Forests and Bagging

Recalling that bagging is a special case of a random forest (with m = p [insert reference]), the `randomForest()` function can be used to perform both random forests and bagging.

Bagging first:

```
library(randomForest)
```

```
## randomForest 4.6-12
## Type rfNews() to see new features/changes/bug fixes.
```

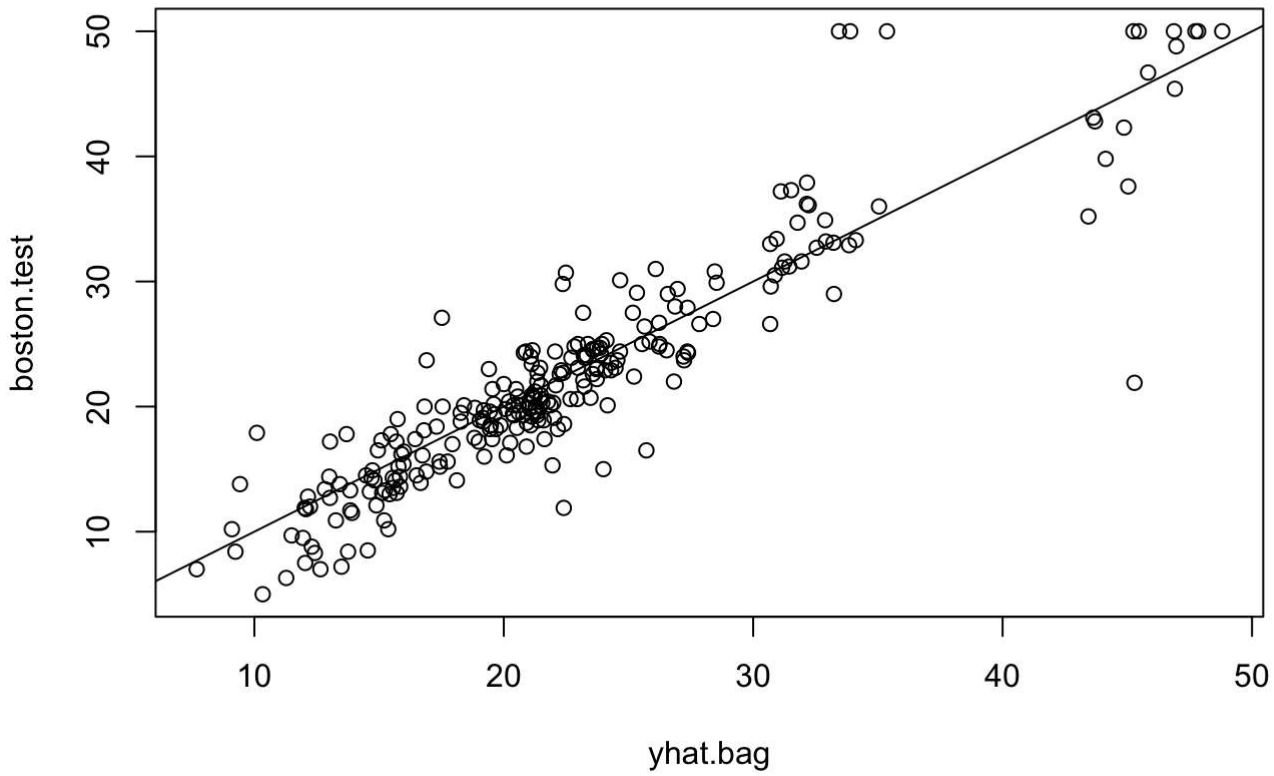
```
set.seed(1)
bag.boston=randomForest(medv~.,data=Boston,subset=train,mtry=13,importance=TRUE)
bag.boston
```

```
##
## Call:
##  randomForest(formula = medv ~ ., data = Boston, mtry = 13, importance = TRUE,      subset = train)
##                Type of random forest: regression
##                      Number of trees: 500
## No. of variables tried at each split: 13
##
##           Mean of squared residuals: 11.02509
##                     % Var explained: 86.65
```

The argument `mtry=13` indicates that all 13 predictors should be considered for each split of the tree. This means that *bagging* should be done.

How good is the performance of bagging on the test set?

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




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

```
## [1] 13.47349
```

This is quite an improvement over trees - almost half the `MSE` obtained with the optimally-pruned single tree regression.

But we can experiment further by changeing the number of trees grown by `randomForest()` using the `ntree` argument:

```
bag.boston=randomForest(medv~.,data=Boston,subset=train,mtry=13,ntree=25)
yhat.bag = predict(bag.boston,newdata=Boston[-train,])
mean((yhat.bag-boston.test)^2)
```

```
## [1] 13.43068
```

We can use a different `mtry` argument:

```
rf.boston=randomForest(medv~.,data=Boston,subset=train,mtry=6,importance=TRUE)
yhat.rf = predict(rf.boston,newdata=Boston[-train,])
mean((yhat.rf-boston.test)^2)
```

```
## [1] 11.20996
```

Now the `MSE` is down to 11.21. Thus random forests are better than bagging in thie example.

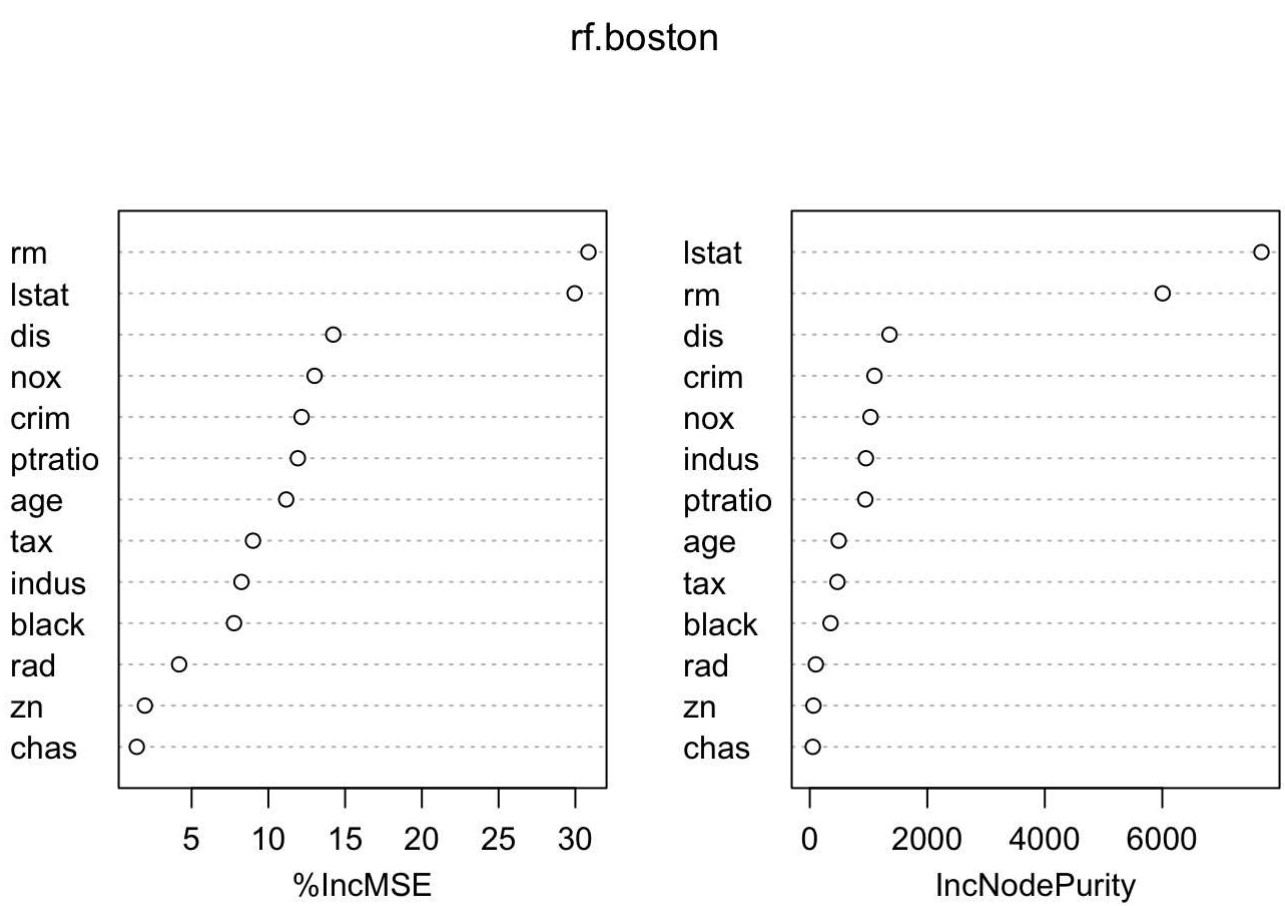
Using the `importance()` function we can view the importance of each variable.

[explain variable importance measures]

```
importance(rf.boston)
```

```
##           %IncMSE IncNodePurity
## crim    12.167433    1100.48678
## zn       1.956719      62.54669
## indus    8.245823     954.97538
## chas     1.429269      50.94527
## nox     13.010349    1032.95449
## rm      30.854978    6006.71143
## age     11.161356     491.91157
## dis     14.227309    1356.79634
## rad      4.186869     102.83437
## tax      8.997186     470.72817
## ptratio 11.922732     944.95930
## black    7.765048     352.90710
## lstat   29.955127    7688.11053
```

```
varImpPlot(rf.boston)
```



The results indicate that across all trees considered in the random forest, `lstat` (the wealth level) and `rm` (house size) are by far the two most important variables.

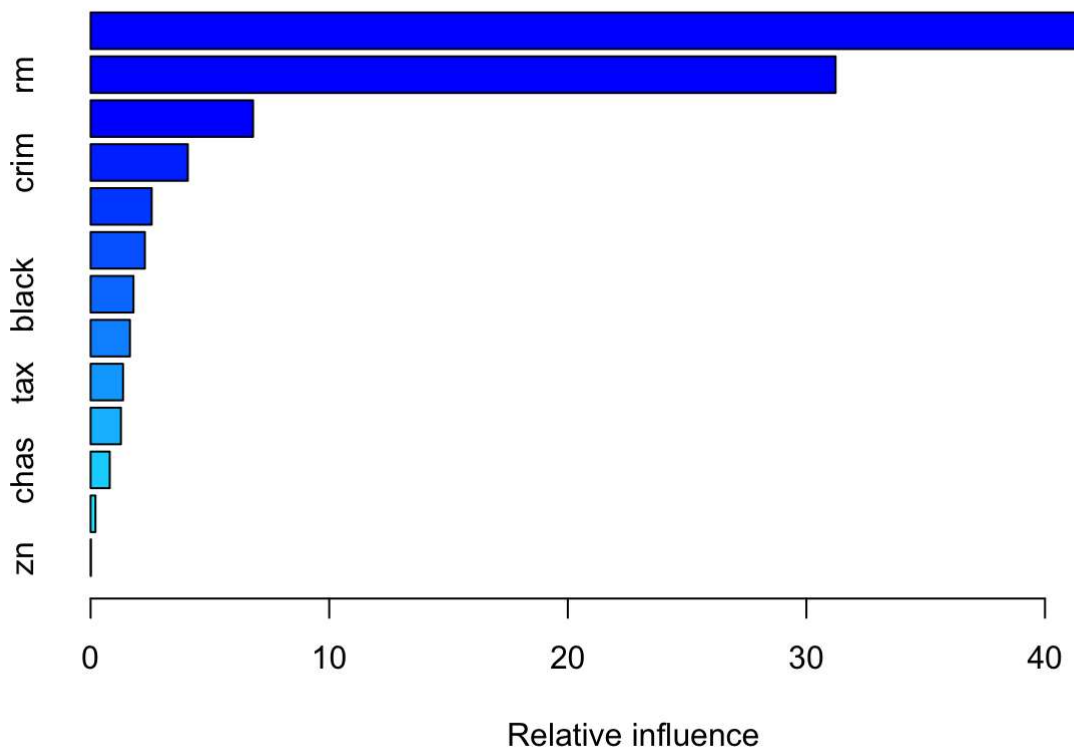
Now we try yet another method:

Boosting

```
library(gbm)
```

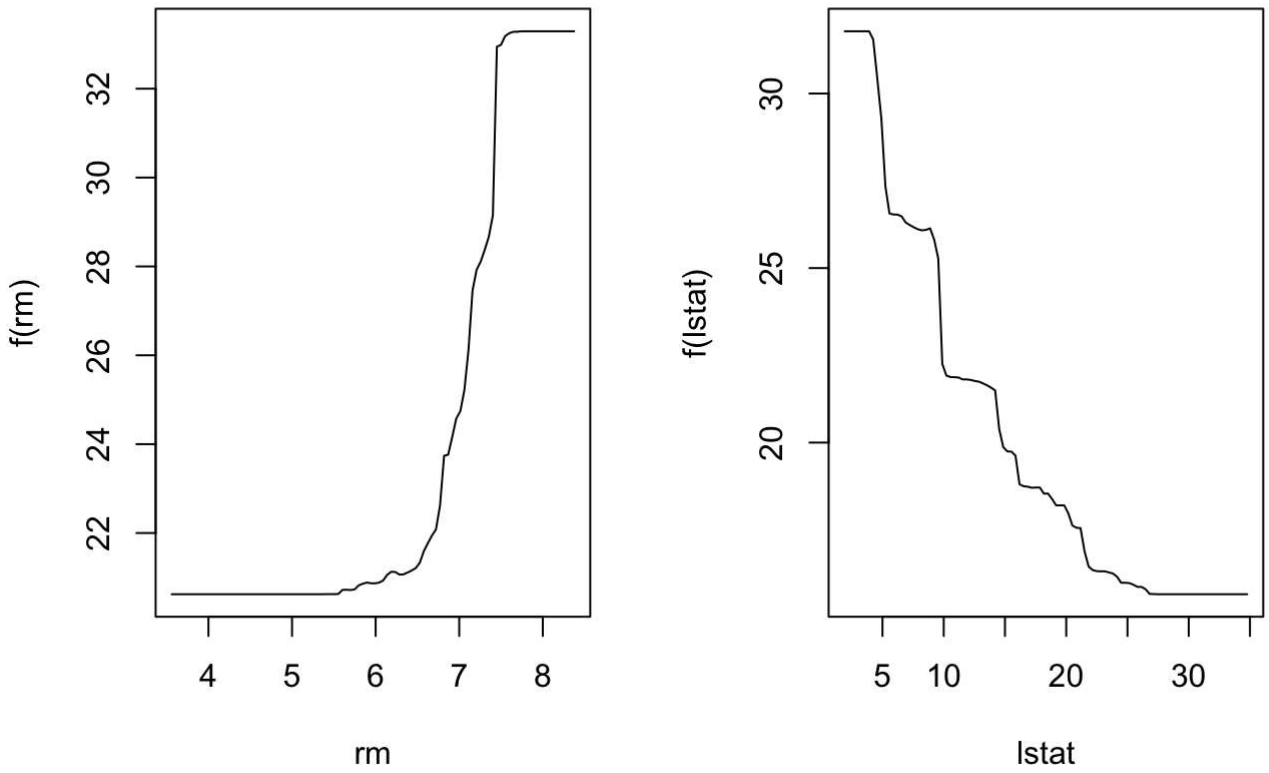
```
## Loading required package: survival
## Loading required package: lattice
## Loading required package: splines
## Loading required package: parallel
## Loaded gbm 2.1.1
```

```
set.seed(1)
boost.boston=gbm(medv~.,data=Boston[train,],distribution="gaussian",n.trees=5000,interaction.depth=4)
summary(boost.boston)
```



```
##             var    rel.inf
## lstat     lstat 45.9627334
## rm           rm 31.2238187
## dis         dis  6.8087398
## crim       crim  4.0743784
## nox         nox  2.5605001
## ptratio ptratio  2.2748652
## black     black  1.7971159
## age         age  1.6488532
## tax         tax  1.3595005
## indus     indus  1.2705924
## chas       chas  0.8014323
## rad         rad  0.2026619
## zn           zn  0.0148083
```

```
par(mfrow=c(1,2))
plot(boost.boston,i="rm")
plot(boost.boston,i="lstat")
```




```
yhat.boost=predict(boost.boston,newdata=Boston[-train,],n.trees=5000)
mean((yhat.boost-boston.test)^2)
```

```
## [1] 11.84434
```

```
boost.boston=gbm(medv~.,data=Boston[train,],distribution="gaussian",n.trees=5000,interaction.depth=4,shrinkage=0.2,verbose=F
)
yhat.boost=predict(boost.boston,newdata=Boston[-train,],n.trees=5000)
mean((yhat.boost-boston.test)^2)
```

```
## [1] 11.51109
```