Bagging and Boosting

Contents

1 Getting Set Up

1.1 Setting chunk options

```
knitr::opts_chunk$set(warning=FALSE, message=FALSE)
knitr::purl("tree-based-methods.Rmd")
```

```
##
##
## processing file: tree-based-methods.Rmd

## output file: tree-based-methods.R
```

1.2 Installing Packages

```
install.packages('tree')
library(tree)
library(ISLR)
install.packages('MASS')
library(MASS)
install.packages('randomForest')
library(randomForest)
```

2 Fitting Regression Trees

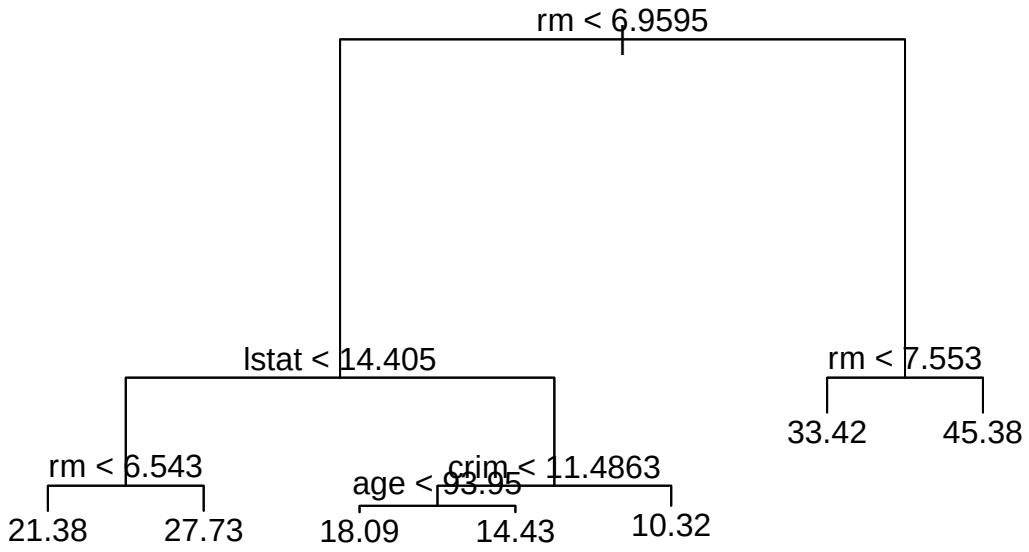
Here we fit a regression tree to the Boston data set. First, we create a training set, and fit the tree to the training data.

```
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] "rm"    "lstat" "crim"  "age"
## Number of terminal nodes:  7
## Residual mean deviance:  10.38 = 2555 / 246
## Distribution of residuals:
##     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
## -10.1800  -1.7770  -0.1775   0.0000   1.9230  16.5800
```

Notice that the output of summary() indicates that only three of the variables have been used in constructing the tree. In the context of a regression tree, the deviance is simply the sum of squared errors for the tree. We now plot the tree.

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

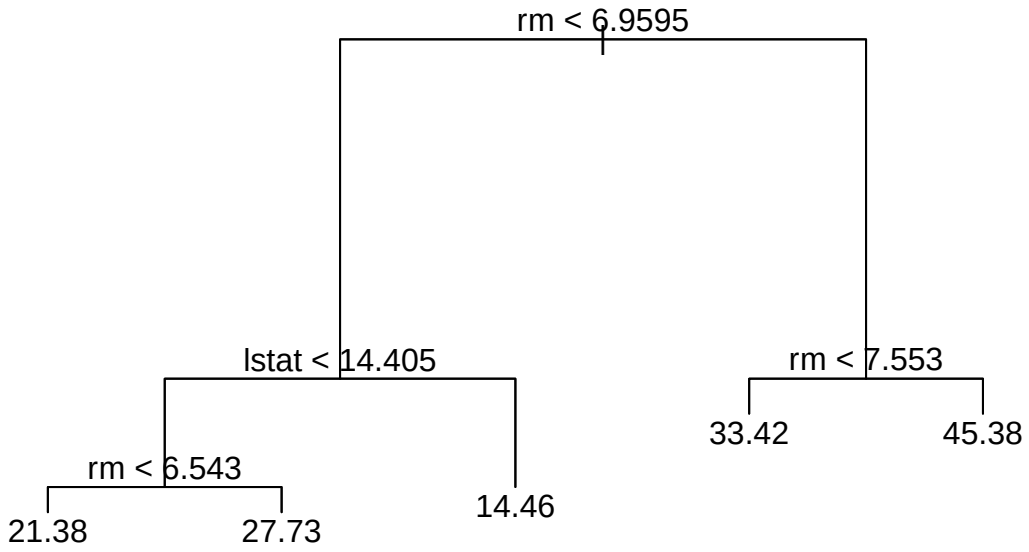


Now we use the cv.tree() function to see whether pruning the tree will improve performance.

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

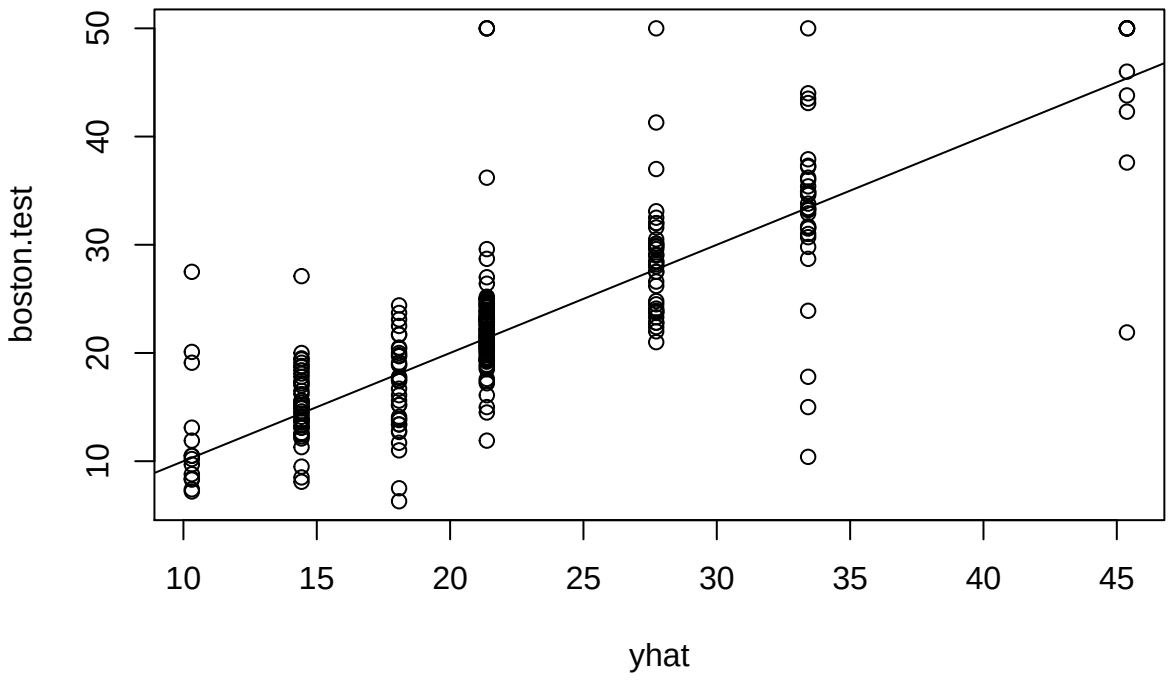
In this case, the most complex tree is selected by cross-validation. However, if we wish to prune the tree, we could do so as follows, using the prune.tree() function:

```
prune.boston = prune.tree(tree.boston, best=5)
plot(prune.boston)
text(prune.boston, pretty =0)
```



In keeping with the cross-validation results, we 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] 35.28688
```

3 Bagging and Random Forests

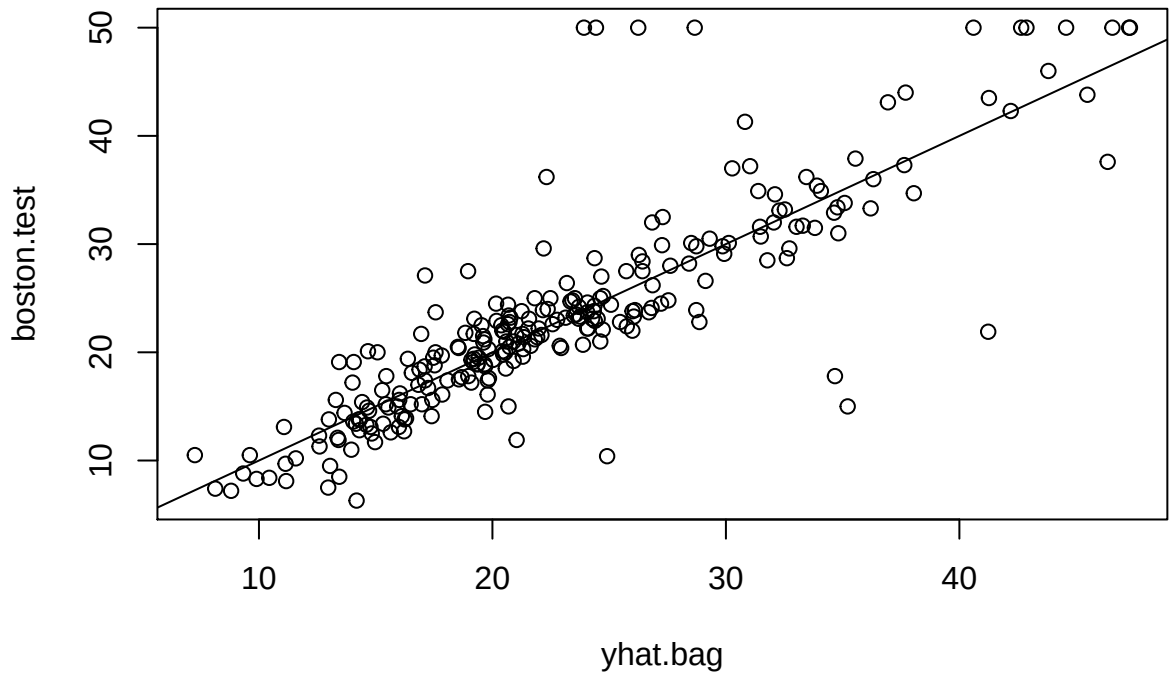
Recall that bagging is simply a special case of a random forest with m = p. Therefore, the randomForest() function can be used to perform both random forests and bagging.

```
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: 11.39601
##                     % Var explained: 85.17
```

The argument mtry=13 indicates that all 13 predictors should be considered for each split of the tree—in other words, that bagging should be done. How well does this bagged model perform 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] 23.59273
```

We could change 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] 23.66716
```

Growing a random forest proceeds in exactly the same way, except that we use a smaller value of the mtry argument. By default, randomForest() uses p/3 variables when building a random forest of regression trees, and $\sqrt{p}$ variables when building a random forest of classification trees. Here we use mtry = 6.

```
set.seed(1)
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] 19.62021
```

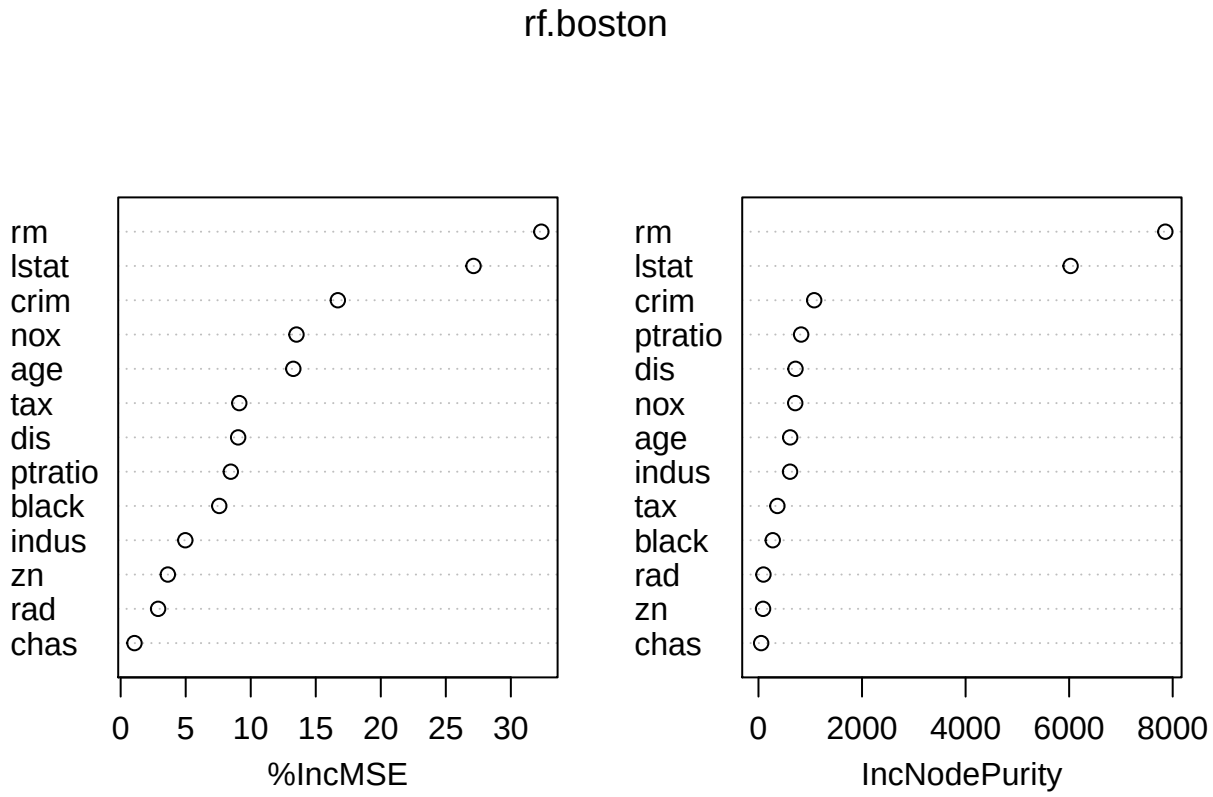
Using the importance() function, we can view the importance of each variable.

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

```
##           %IncMSE IncNodePurity
## crim    16.697017    1076.08786
## zn       3.625784      88.35342
## indus    4.968621     609.53356
## chas     1.061432      52.21793
## nox     13.518179     709.87339
## rm      32.343305    7857.65451
## age     13.272498     612.21424
## dis      9.032477     714.94674
## rad      2.878434      95.80598
## tax      9.118801     364.92479
## ptratio  8.467062     823.93341
## black    7.579482     275.62272
## lstat   27.129817    6027.63740
```

Two measures of variable importance are reported. The former is based upon the mean decrease of accuracy in predictions on the out of bag samples when a given variable is excluded from the model. The latter is a measure of the total decrease in node impurity that results from splits over that variable, averaged over all trees (this was plotted in Figure 8.9). In the case of regression trees, the node impurity is measured by the training RSS, and for classification trees by the deviance. Plots of these importance measures can be produced using the varImpPlot() function.

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



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

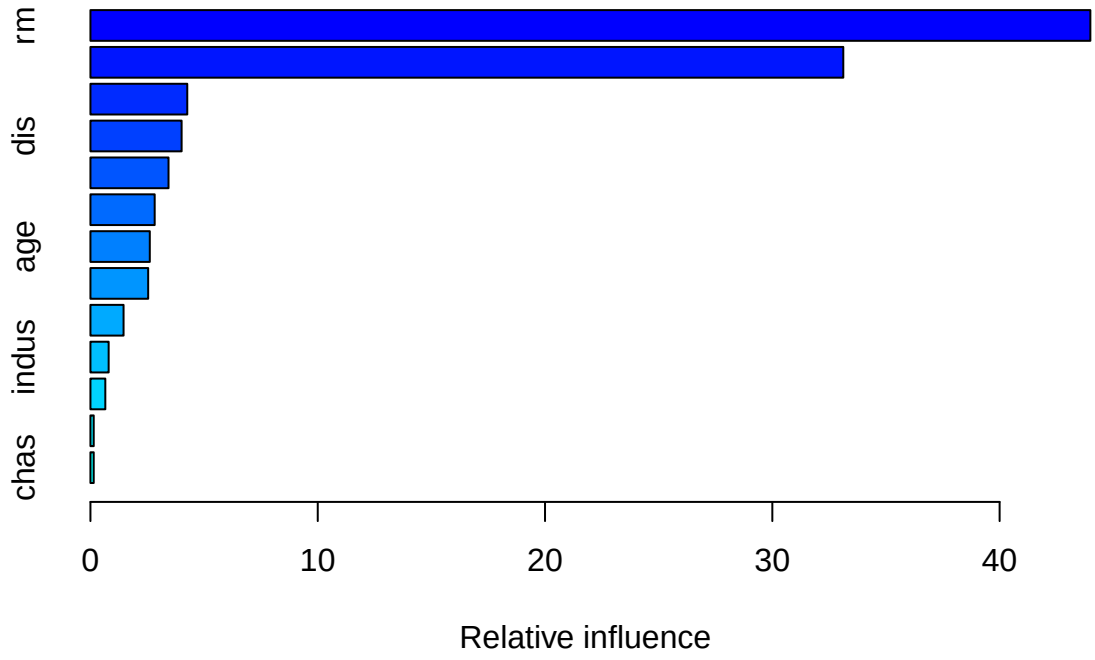
4 Boosting

Here we use the gbm package, and within it the gbm() function, to fit boosted regression trees to the Boston data set. We run gbm() with the option distribution="gaussian" since this is a regression problem; if it were a binary classification problem, we would use distribution="bernoulli". The argument n.trees=5000 indicates that we want 5000 trees, and the option interaction.depth=4 limits the depth of each tree.

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

The summary() function produces a relative influence plot and also outputs the relative influence statistics.

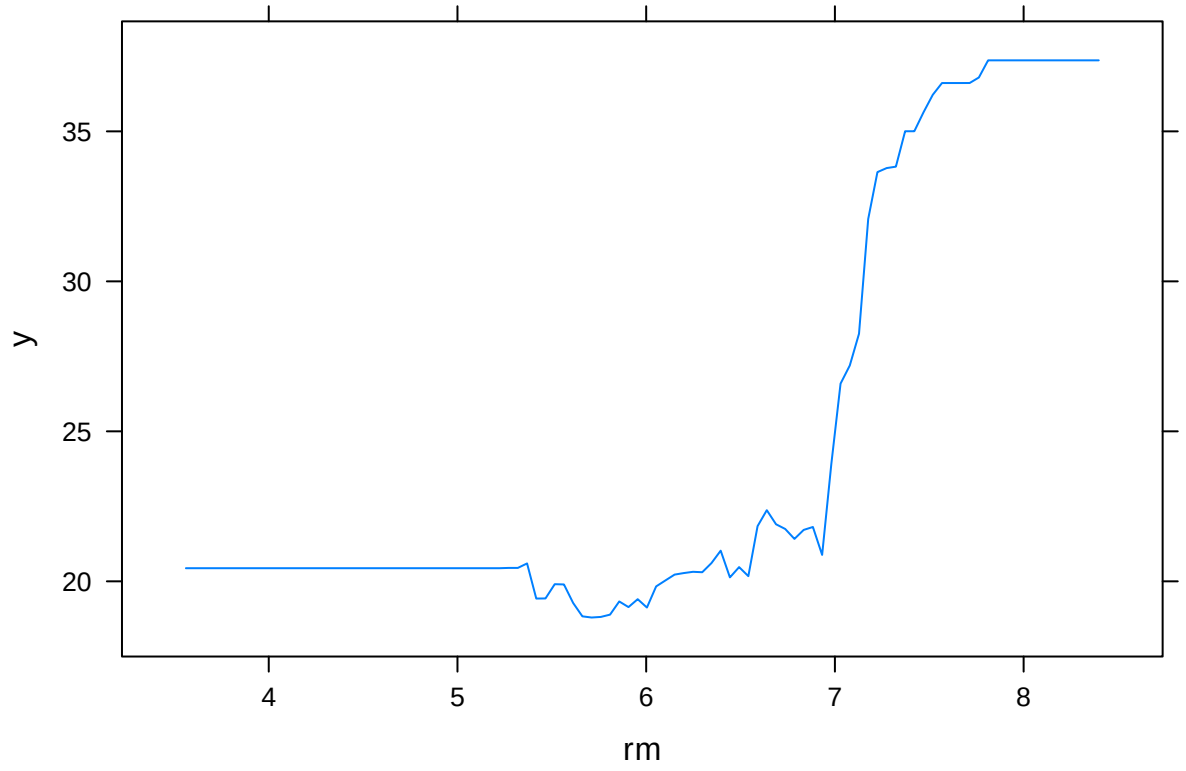
```
summary(boost.boston)
```



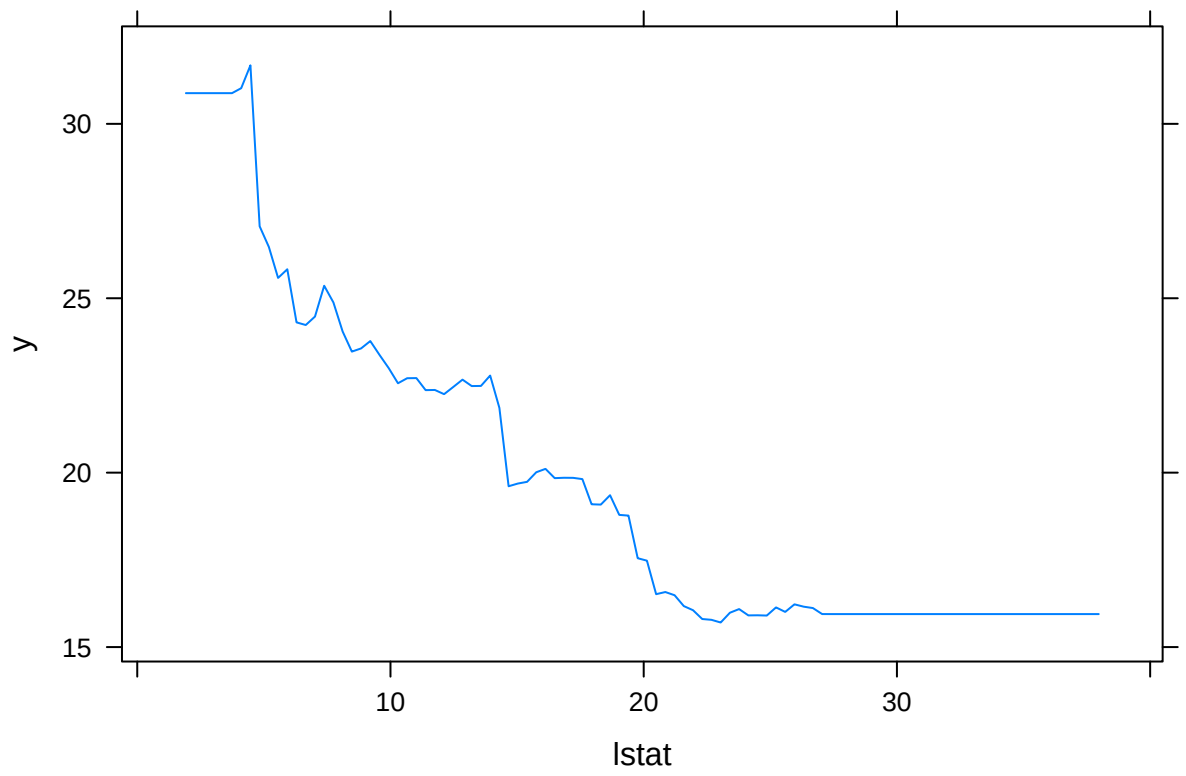

```
##             var    rel.inf
## rm           rm 43.9919329
## lstat     lstat 33.1216941
## crim       crim  4.2604167
## dis         dis  4.0111090
## nox         nox  3.4353017
## black     black  2.8267554
## age         age  2.6113938
## ptratio ptratio  2.5403035
## tax         tax  1.4565654
## indus     indus  0.8008740
## rad         rad  0.6546400
## zn           zn  0.1446149
## chas       chas  0.1443986
```

We see that lstat and rm are by far the most important variables. We can also produce partial dependence plots for these two variables. These plots illustrate the marginal effect of the selected variables on the response after integrating out the other variables. In this case, as we might expect, median house prices are increasing with rm and decreasing with lstat.

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




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



We now use the boosted model to predict medv on the test set:

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

```
## [1] 18.84709
```

If we want to, we can perform boosting with a different value of the shrinkage parameter . The default value is 0.001, but this is easily modified. Here we take = 0.2.

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

```
## [1] 18.33455
```