8 Tree-Based Models

Andrew Liang

11/22/2020

Notes

- Useful for interpretation, but typically not competitive with best supervised learning approaches in terms of prediction

Basics of Decision Trees

Regression Trees

Process of Building a Regression Tree:

1. Divide the predictor space (set of possible values of $X_1, X_2, \ldots, X_p$ into J distinct and non-overlapping regions $R_1, R_2, \ldots, R_J$
2. For every observation in theR_j region, we make the same prediction, which is simply the mean of the response values for the training observations in R_j

- In step 1, we construct $R_1, R_2, \ldots, R_J$ such that the predictor space is divided into high-dimensional rectangles, or *boxes*, for ease of interpretation
  - goal is to minimize RSS given by:

$$\sum_{j=1}^{J} \sum_{i \in R_j} (y_i - \hat{y}_{R_j})^2$$

where $\hat{y}_{R_j}$ is the mean response for the training observations within the jth box.

- Another way to look at it is that we consider all predictors $X_1, X_2, \ldots, X_p$ and all possible values of cutpoint s for each of the predictors, then choose the predictor and cutpoint such that the resulting tree has lowest RSS
  - for any j and s, we define pair of half-planes:

$$R_1(j,s) = \{X|X_j < s\} \quad and \quad R_2(j,s) = \{X|X_j \geq s\}$$

and we seek to value of j and s to minimize:

$$\sum_{i:x_i \in R_1(j,s)} (y_i - \hat{y}_{R_1})^2 + \sum_{i:x_i \in R_2(j,s)} (y_i - \hat{y}_{R_2})^2$$

where $\hat{y}_{R_1}$ is the mean response for training observations in $R_1(j,s)$ and $\hat{y}_{R_2}$ is mean response for train obs in $R_2(j,s)$

Once the regions $R_1, R_2, \ldots, R_J$ have been created, we can then predict response for a given test observation using mean of train obs in the region to which that test obs belongs

Tree Pruning

- Process above may likely overfit data as the resulting tree may be too complex
  - smaller tree with fewer splits can lead to lower variance and better interpretation at the cost of a little bias
- Better strategy may be to grow a very large tree T_0, and then *prune* it back to get a subtree
  - want to get a subtree that leads us to the lowest test error rate
  - however, using CV for every subtree may be infeasible, so we need to select a small set of subtrees
- can use *cost complexity pruning*, consider a sequence of trees indexed by a nonnegative tuning parameter α
  - for each value of α, there corresponds a subtree $T \subset T_0$ such that it minimizes:

$$\sum_{m=1}^{|T|} \sum_{i:x \in R_m} (y_i - \hat{y}_{R_m})^2 + \alpha |T|$$

where $|T|$ indicates the number of terminal nodes of tree T, and R_m is the region corresponding to the mth terminal node

- as the number of terminal nodes increases, there is a penalty α, so the above quantity will tend to be minimzied for a smaller subtree
  - select α using CV

Algorithm for building Regression Trees

1. Use recursive binary splitting to grow a large tree on training data, stopping when each terminal node has fewer than some minimum # of observations
2. Apply cost complexity pruning to large tree in order to obtain a sequence of best subtrees as a function of α
3. Use K-fold CV to choose α. Divide training observations into K fold. For each $k = 1, \ldots, K$:
   - repeat steps 1 and 2 on all but kth fold of training data
   - evaluate mean squared prediction error on data in left-out kth fold, as a function of α
   - average out the results for each value of α, and pick α to minimize error
4. Return the subtree from step 2 that corresponds to chosen value of α

Classification Trees

- very similar to regression tree, but predicts qualitative response instead
  - predict that each observation belongs to the *most commonly occuring class* or training observations in the region to which it belongs
- also interested in the *class proportions* among training observations that fall into that region
- also use recursive binary splitting to grow a classification tree, but instead of RSS, *classification error rate* is used as the criterion for making the binary splits
  - simply the fraction of training obs in that region that don't belong to the most common class:

$$E = 1 - \max_k(\hat{p}_{mk})$$

where $\hat{p}_{mk}$ is the proportion of training observations in th mth region that are from the kth class. In practice however, two other measures are preferable:

- The *Gini index*:

$$G = \sum_{k=1}^{K} \hat{p}_{mk}(1 - \hat{p}_{mk})$$

is a measure of total variance (of which we want to minimize) across the K classes. G takes on a small value if $\hat{p}_{mk}$ is close to zero or one. It is a measure of *node purity* - a small value indicates that a node contains predominantly observations from a single class

- The *Entropy*:

$$D = -\sum_{k=1}^{K} \hat{p}_{mk} log(\hat{p}_{mk})$$

similarly to the Gini, will take on a value near zero if all $\hat{p}_{mk}$ are near zero or one. Both measurements are quite similar numerically

- generally, classification error rate is preferable if prediction accuracy is the goal of the final pruned tree

Trees vs Linear Models

- if relationship between features and response is well approximated by a linear model, then a linear regression would outperform trees
- if there is a highly non-linear and complex relationship between features and response, then trees may outperform linear regression

Pros of Trees

- very easy to explain
- may closely mirror human decision-making more than regression and classification approaches in previous methods
- displayed graphiccaly and easily interpreted
- can handle qualitative predictors without the need to create dummy variables

Cons of Trees

- generally don't have same level of predictive accuracy as other regression and classification techniques
- tend to be very non-robust, small change in data can cause large change in the tree

Bagging, Random Forests, Boosting

Bagging

- in order to solve the high variance problem in trees, *bagging* can help reduce it through boostrapping procedures
  - recall in bootstrap, given n independent observations $Z_1, \ldots, Z_n$, each with variance σ^2
    - variance of mean $\overline{Z}$ is given by $\frac{\sigma^2}{n}$, thus has a lower variance
    - essentially we use repeated samples from our original training data to create B different training sets:

$$\hat{f}_{bag}(x) = \frac{1}{B}\sum_{b=1}^{B}\hat{f}^{*b}(x)$$

which is called bagging

- in the context of regression trees, we construct B different trees using B boostrapped training sets, then avg the resulting predictions
  - avg out all the trees reduces the variance
- in classification trees, for a given test observation, we can record the class predicted by each of the B tres, and take a *majority vote*, which is the overall prediction most commonly occurring among all the B predictions
- important to note that using a large B will not lead to overfitting
  - generally use $B = 100$ to achieve sufficient performance

Out-of-Bag Error Estimation

- very straightforward way of estimating test error of a bagged model, without the need of CV
- on avg, each bagged tree makes use around two-thirds of observations (sampling with replacement of training set)
  - remaining one-third of observations not used are the *Out-of-Bag* (OOB) observations
  - can predict response for ith observation using each of the trees in which that observation was OOB
    - yields around $B/3$ predictions for ith observation
  - with B sufficiently large, OOB error is essentially the same as LOOCV
    - convenient when perform CV would be computationally infeasible

Variable Importance Measures

- bagging improves prediction accuracy at the expense of interpretability
  - can obtain summary of important predictors using RSS or Gini index
    - for regression, record the total amount the RSS is decreased due to splits over a given predictor
    - for classification, sum the total amount that the Gini index is decreased by splits over a given predictor

Random Forests

- improves over bagging through docorrelation of the trees
- similar to bagging, we build a number of trees based on bootstrapping, but now we choose a *random sample of m predictors* as split candidates from the full set of p predictors
  - split is only allowed to use one of those m predictors
    - a fresh sample of m predictors chosen at each split
  - typically choose $m \approx \sqrt{p}$
  - thus at each split, the algorithm is not even allowed to consider a majority of the available predictors
    - helps *decorrelate* trees, to prevent the domination of one strong predictor on all the trees
  - when $m = p$, then the process is just bagging, hence bagging is a special case of random forest

Boosting

- boosting works similarly to bagging, but now each tree is grown *sequentially*
  - uses info from previous tree
  - doesn't use bootstrap, instead tree is fit on a modified version of original data

Algorithm for Boosting

1. Set $\hat{f}(x) = 0$ and $r_i = y_i$ for all i in the training set
2. For $b = 1, 2, \ldots, B$, repeat:

- Fit a tree $\hat{f}^b$ with d splits ($d + 1$ terminal nodes) to training data (X, r)
- Update $\hat{f}$ by adding shrunken version of new tree:

$$\hat{f}(x) \leftarrow \hat{f}(x) + \lambda \hat{f}^b(x)$$

- Update the residuals:

$$r_i \leftarrow r_i + \lambda \hat{f}^b(x)$$

3. Output the boosted model:

$$\hat{f}(x) = \sum_{b=1}^{B} \lambda \hat{f}^b(x)$$

- the idea is to slowly improve $\hat{f}$ in areas where it doesn't perform well
- shrinkage λ slows the process down, allowing more and different shaped trees to attack residuals

Boosting has three tuning parameters:

1. Number of trees B. Unlike bagging and random forests, boosting can overfit if B is too large, hence we use CV to choose B

2. shrinkage parameter λ, a small positive number and controls rate at which boosting learns (typical values are 0.01 or 0.001). Very small λ can require large B to achieve good performance
3. Number of splits d in each tree. Controls complexity of boosted ensemble, and often $d = 1$ works well, where each tree is a *stump* of a single split. When $d = 1$, boosted ensemble is fitting an additive model, since each term involves only one variable. Generally, d is the *interaction depth*

- because growth of a tree depends on previous trees, smaller trees are typically sufficient
  - smaller trees can aid interpretability

Applied

Fitting Classification Trees

```r
library(tree)
library(ISLR)
attach(Carseats)
High <- ifelse(Sales <= 8, "No","Yes") #recode into binary variable
Carseats <- data.frame(Carseats, High)
```

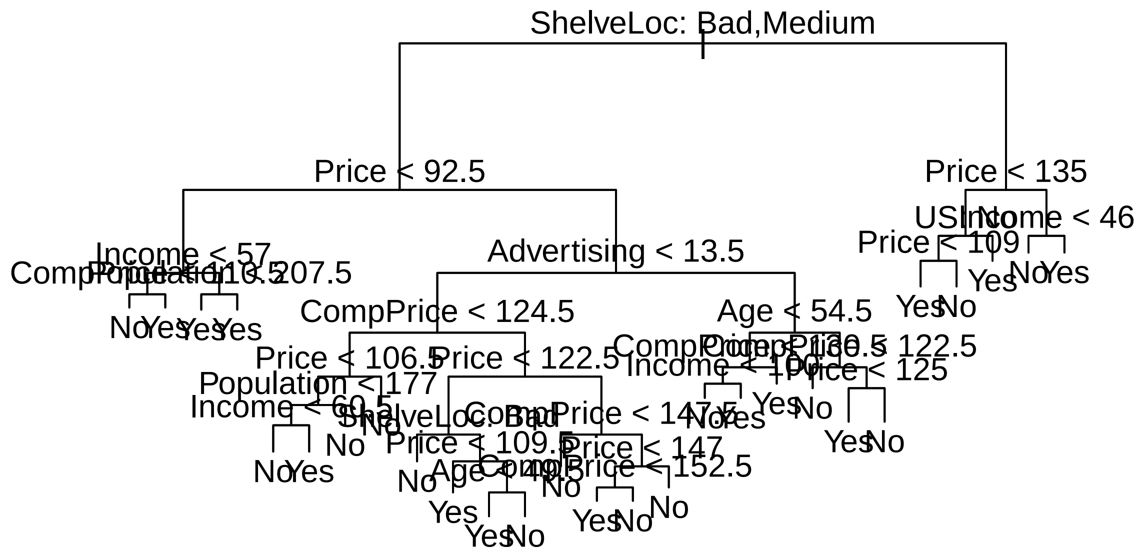
```r
# fit classification tree to predict High using all variables but sales
tree.carseats <- tree(High~. -Sales, Carseats)
summary(tree.carseats)
```

```
##
## Classification tree:
## tree(formula = High ~ . - Sales, data = Carseats)
## Variables actually used in tree construction:
## [1] "ShelveLoc"   "Price"       "Income"      "CompPrice"   "Population"
## [6] "Advertising" "Age"         "US"
## Number of terminal nodes:  27
## Residual mean deviance:  0.4575 = 170.7 / 373
## Misclassification error rate: 0.09 = 36 / 400
```

summary function lists variables that are used as internal nodes in tree, the number of terminal nodes, and training error rate

Now we plot the tree:

```r
plot(tree.carseats)
text(tree.carseats, pretty = 0)
```



Typing in the tree object prints out the split criterions, # of observations in the branch, deviance, the overall prediction for that branch, and the fraction of observations in that branch that take on values of Yes and No respectively.

```
tree.carseats
```

```
## node), split, n, deviance, yval, (yprob)
##       * denotes terminal node
##
##   1) root 400 541.500 No ( 0.59000 0.41000 )
##     2) ShelveLoc: Bad,Medium 315 390.600 No ( 0.68889 0.31111 )
##       4) Price < 92.5 46  56.530 Yes ( 0.30435 0.69565 )
##         8) Income < 57 10  12.220 No ( 0.70000 0.30000 )
##          16) CompPrice < 110.5 5   0.000 No ( 1.00000 0.00000 ) *
##          17) CompPrice > 110.5 5   6.730 Yes ( 0.40000 0.60000 ) *
##         9) Income > 57 36  35.470 Yes ( 0.19444 0.80556 )
##          18) Population < 207.5 16  21.170 Yes ( 0.37500 0.62500 ) *
##          19) Population > 207.5 20   7.941 Yes ( 0.05000 0.95000 ) *
##       5) Price > 92.5 269 299.800 No ( 0.75465 0.24535 )
##        10) Advertising < 13.5 224 213.200 No ( 0.81696 0.18304 )
##          20) CompPrice < 124.5 96  44.890 No ( 0.93750 0.06250 )
##            40) Price < 106.5 38  33.150 No ( 0.84211 0.15789 )
##              80) Population < 177 12  16.300 No ( 0.58333 0.41667 )
##               160) Income < 60.5 6   0.000 No ( 1.00000 0.00000 ) *
##               161) Income > 60.5 6   5.407 Yes ( 0.16667 0.83333 ) *
##              81) Population > 177 26   8.477 No ( 0.96154 0.03846 ) *
##            41) Price > 106.5 58   0.000 No ( 1.00000 0.00000 ) *
##           21) CompPrice > 124.5 128 150.200 No ( 0.72656 0.27344 )
##             42) Price < 122.5 51  70.680 Yes ( 0.49020 0.50980 )
##               84) ShelveLoc: Bad 11   6.702 No ( 0.90909 0.09091 ) *
##               85) ShelveLoc: Medium 40  52.930 Yes ( 0.37500 0.62500 )
##                170) Price < 109.5 16   7.481 Yes ( 0.06250 0.93750 ) *
##                171) Price > 109.5 24  32.600 No ( 0.58333 0.41667 )
##                  342) Age < 49.5 13  16.050 Yes ( 0.30769 0.69231 ) *
##                  343) Age > 49.5 11   6.702 No ( 0.90909 0.09091 ) *
##             43) Price > 122.5 77  55.540 No ( 0.88312 0.11688 )
##               86) CompPrice < 147.5 58  17.400 No ( 0.96552 0.03448 ) *
##               87) CompPrice > 147.5 19  25.010 No ( 0.63158 0.36842 )
##                174) Price < 147 12  16.300 Yes ( 0.41667 0.58333 )
##                  348) CompPrice < 152.5 7   5.742 Yes ( 0.14286 0.85714 ) *
##                  349) CompPrice > 152.5 5   5.004 No ( 0.80000 0.20000 ) *
##                175) Price > 147 7   0.000 No ( 1.00000 0.00000 ) *
##         11) Advertising > 13.5 45  61.830 Yes ( 0.44444 0.55556 )
##           22) Age < 54.5 25  25.020 Yes ( 0.20000 0.80000 )
##             44) CompPrice < 130.5 14  18.250 Yes ( 0.35714 0.64286 )
##               88) Income < 100 9  12.370 No ( 0.55556 0.44444 ) *
##               89) Income > 100 5   0.000 Yes ( 0.00000 1.00000 ) *
##             45) CompPrice > 130.5 11   0.000 Yes ( 0.00000 1.00000 ) *
##           23) Age > 54.5 20  22.490 No ( 0.75000 0.25000 )
##             46) CompPrice < 122.5 10   0.000 No ( 1.00000 0.00000 ) *
##             47) CompPrice > 122.5 10  13.860 No ( 0.50000 0.50000 )
##               94) Price < 125 5   0.000 Yes ( 0.00000 1.00000 ) *
##               95) Price > 125 5   0.000 No ( 1.00000 0.00000 ) *
##      3) ShelveLoc: Good 85  90.330 Yes ( 0.22353 0.77647 )
##        6) Price < 135 68  49.260 Yes ( 0.11765 0.88235 )
##         12) US: No 17  22.070 Yes ( 0.35294 0.64706 )
##           24) Price < 109 8   0.000 Yes ( 0.00000 1.00000 ) *
##           25) Price > 109 9  11.460 No ( 0.66667 0.33333 ) *
##         13) US: Yes 51  16.880 Yes ( 0.03922 0.96078 ) *
##        7) Price > 135 17  22.070 No ( 0.64706 0.35294 )
##         14) Income < 46 6   0.000 No ( 1.00000 0.00000 ) *
##         15) Income > 46 11  15.160 Yes ( 0.45455 0.54545 ) *
```

Now we split train/test to evaluate test error:

```
set.seed(1)
train <- sample(1:nrow(Carseats),200) # 50/50 split
Carseats.test <- Carseats[-train,]
High.test <- High[-train]
tree.carseats <- tree(High~. -Sales, Carseats,subset = train)
tree.pred <- predict(tree.carseats,Carseats.test,type = "class")
table(tree.pred,High.test)
```

```
##           High.test
## tree.pred No Yes
##       No  84  37
##       Yes 35  44
```

correct predictions:

```r
(84+44)/200
```

```
## [1] 0.64
```

So the prediction accuracy is around 64%

Next we consider if pruning helps our accuracy:

```r
set.seed(1)
cv.carseats <- cv.tree(tree.carseats,FUN = prune.misclass) # pruning our tree
names(cv.carseats)
```

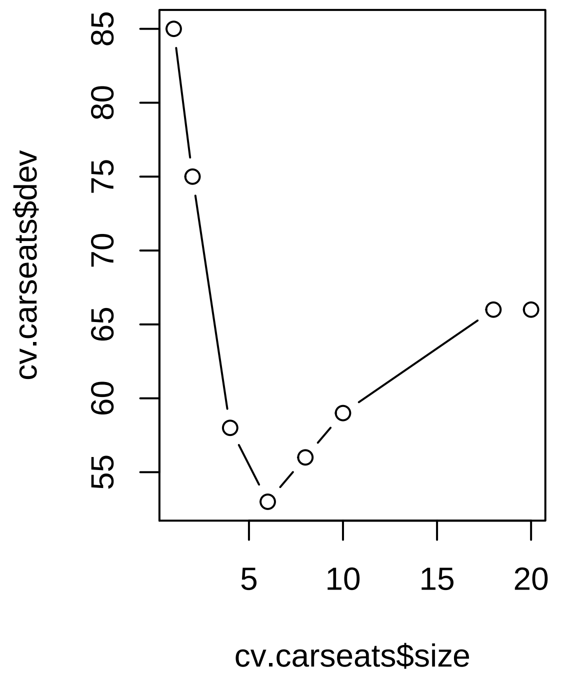
```
## [1] "size"   "dev"    "k"      "method"
```

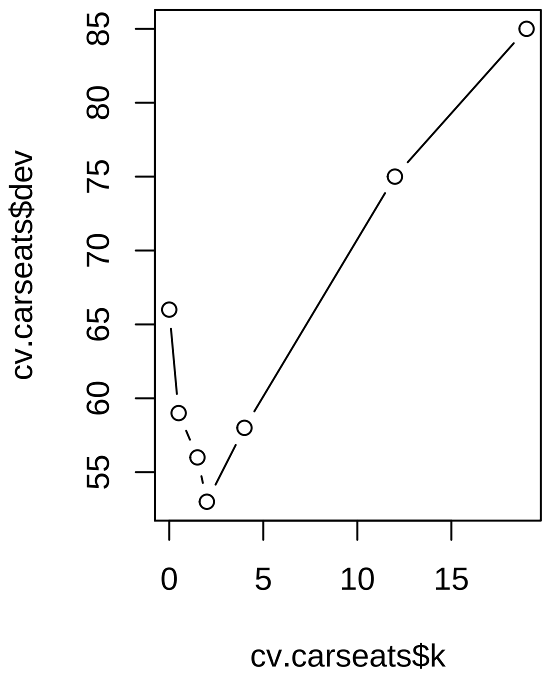
```r
cv.carseats
```

```
## $size
## [1] 20 18 10  8  6  4  2  1
##
## $dev
## [1] 66 66 59 56 53 58 75 85
##
## $k
## [1] -Inf  0.0  0.5  1.5  2.0  4.0 12.0 19.0
##
## $method
## [1] "misclass"
##
## attr(,"class")
## [1] "prune"         "tree.sequence"
```

dev is the CV error rate, and thus it looks like the tree with 6 terminal nodes results in lowest error rate, with 53 CV-errors. k corresponds to the cost-complexity parameter used, in which this case was α.

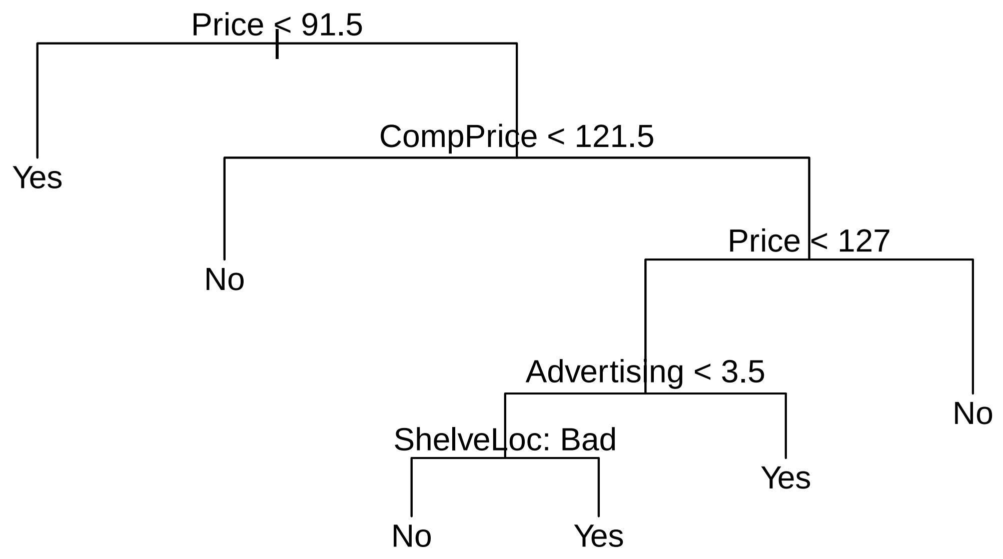
```r
par(mfrow=c(1,2))
plot(cv.carseats$size,cv.carseats$dev, type = "b")
plot(cv.carseats$k,cv.carseats$dev,type = "b")
```





Now apply prune.misclass() to prune tree to obrain the 6 node tree:

```r
prune.carseats <- prune.misclass(tree.carseats, best=6)
plot(prune.carseats)
text(prune.carseats, pretty = 0)
```



Now lets test it on the test data:

```
tree.pred <- predict(prune.carseats,Carseats.test, type = "class")
table(tree.pred,High.test)
```

```
##          High.test
## tree.pred No  Yes
##       No  86  32
##       Yes 33  49
```

```
(86+49)/200
```

```
## [1] 0.675
```

Seems like we have improved on the original tree by about 4%. Not only is it more accurate, but much more interpretable as well.

Fitting Regression Trees

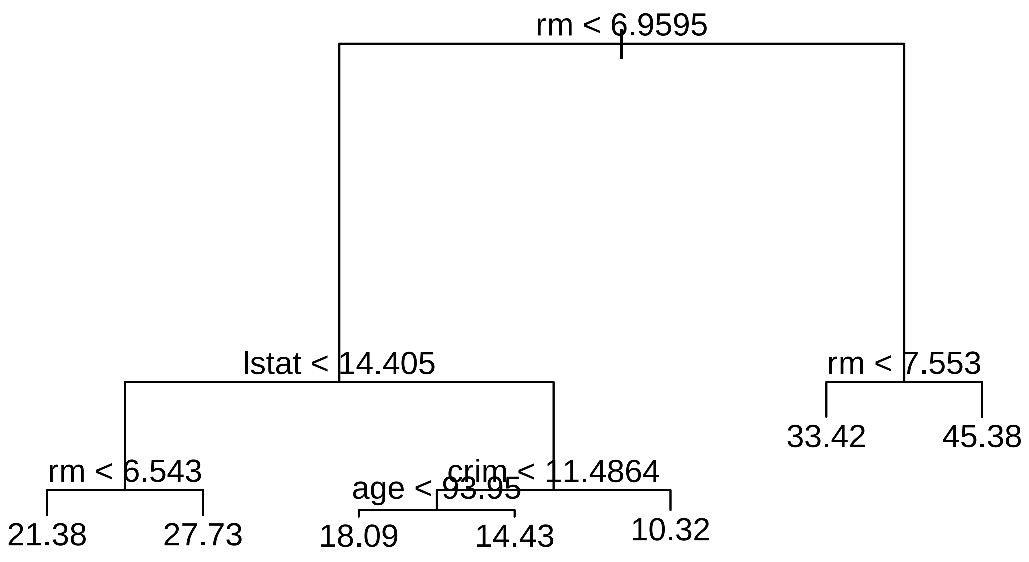
Let's create a train/test split on the Boston data set

```
library(MASS)
set.seed(1)
train <- sample(1:nrow(Boston), nrow(Boston)/2) # 50/50 split
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
```

In regression trees, deviance is the sum of squared errors for the tree

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



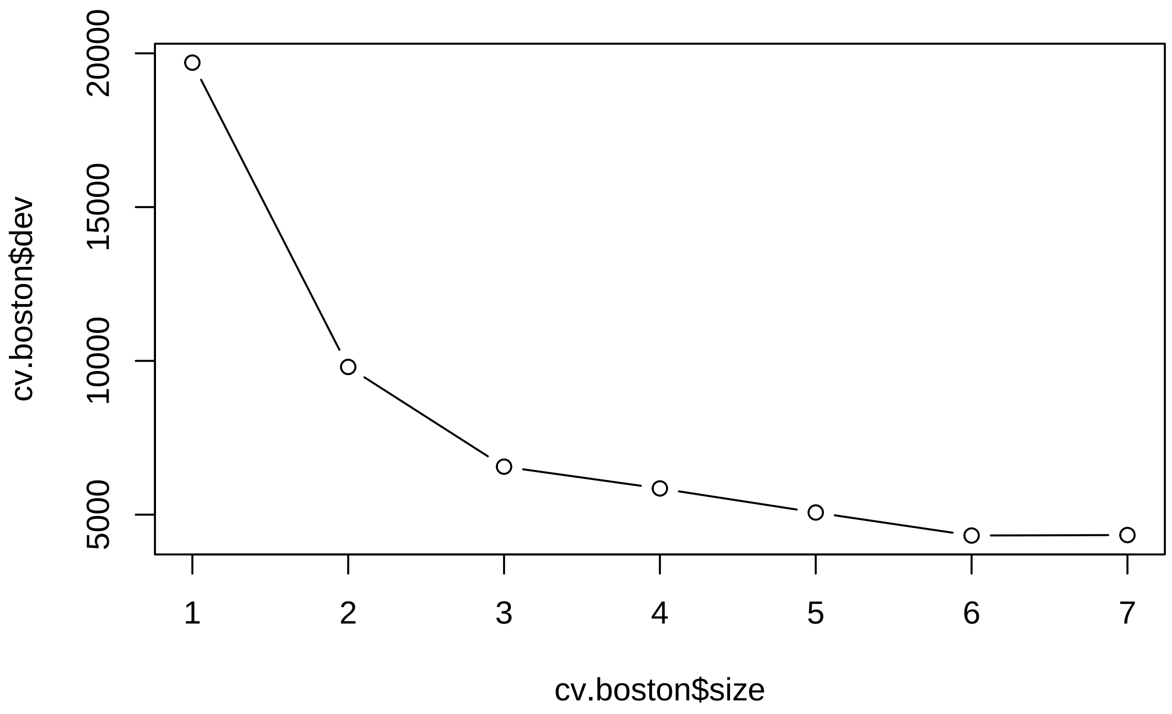
It seems that the most important variables are rm(avg number of rooms per dwelling), and lstat(% of lower status of population).

Let's prune the tree:

```
set.seed(1)
cv.boston <- cv.tree(tree.boston)
cv.boston
```

```
## $size
## [1] 7 6 5 4 3 2 1
##
## $dev
## [1]  4336.868  4321.549  5070.107  5852.631  6560.984  9802.545 19697.191
##
## $k
## [1]       -Inf   203.9641   637.2707   796.1207  1106.4931  3424.7810 10724.5951
##
## $method
## [1] "deviance"
##
## attr(,"class")
## [1] "prune"         "tree.sequence"
```

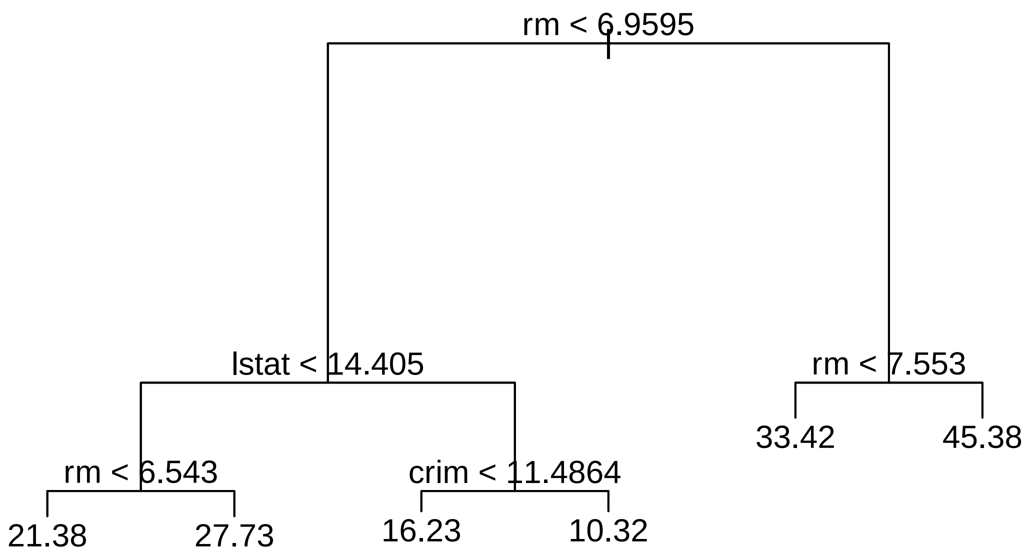
```
plot(cv.boston$size,cv.boston$dev, type="b")
```



Using CV, seems like the one one with the best performance is the one with 6 terminal nodes

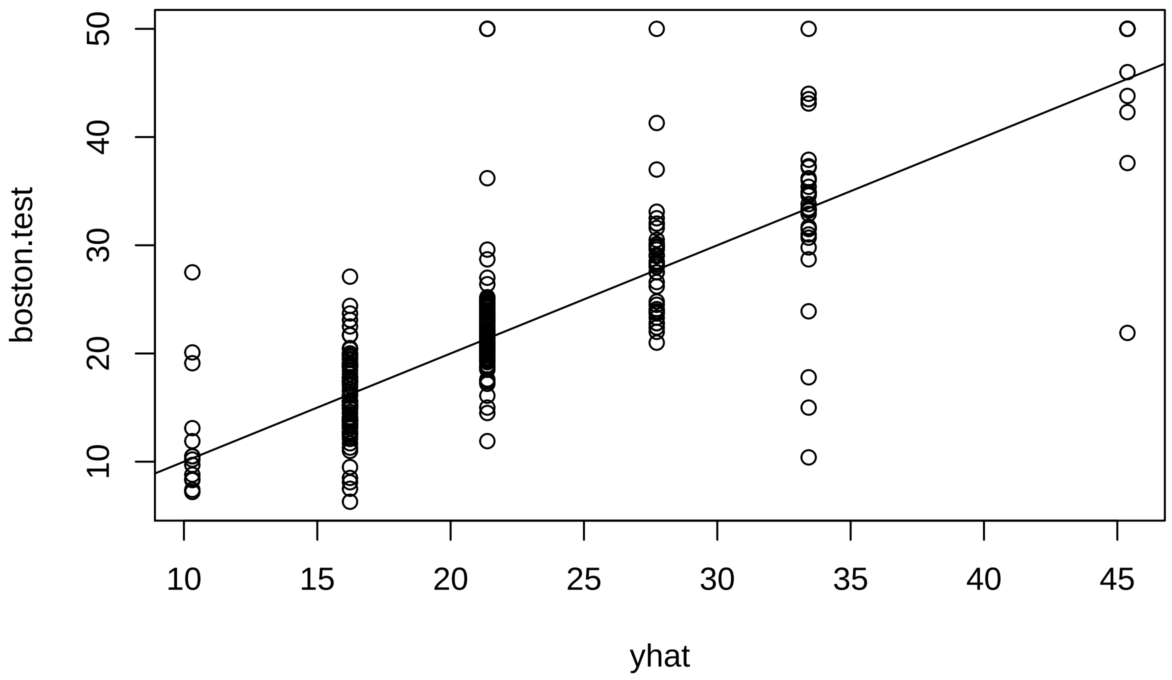
Now let's prune the tree:

```
prune.boston <- prune.tree(tree.boston,best = 6)
plot(prune.boston)
text(prune.boston,pretty=0)
```



Let's now make predictions on test set:

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



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

```
## [1] 35.16439
```

So the test set MSE is around 35.

Baggin and Random Forests

```r
library(randomForest)
```

```
## randomForest 4.6-14
```

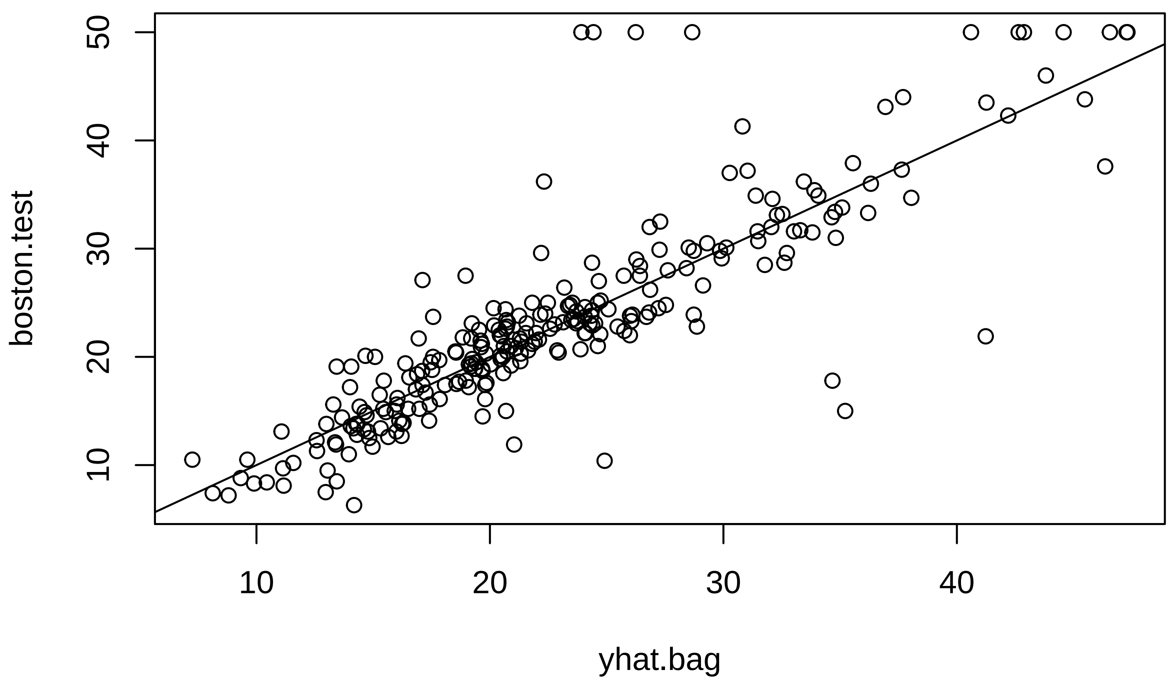
```
## Type rfNews() to see new features/changes/bug fixes.
```

```r
set.seed(1)

# Bagging
bag.boston <- randomForest(medv~.,
                           data = Boston,
                           subset=train,
                           mtry = 13,
                           importance = T) # we set mtry = 13 since we are using all predictors

bag.boston
```

```
##
## Call:
##  randomForest(formula = medv ~ ., data = Boston, mtry = 13, importance = T,      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
```

Let's see how well bagging performs on 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
```

MSE is around 23.6, which is significantly better than the optimally-pruned tree from before

Let's try using random forest with m = 6:

```r
set.seed(1)
rf.boston <- randomForest(medv~.,
                          data = Boston,
                          subset = train,
                          mtry = 6,
                          importance = T)
yhat.rf <- predict(rf.boston, newdata = Boston[-train,])
mean((yhat.rf-boston.test)^2)
```

```
## [1] 19.62021
```

we can see that the test MSE has decreased down to 19.6, which is an improvement over bagging

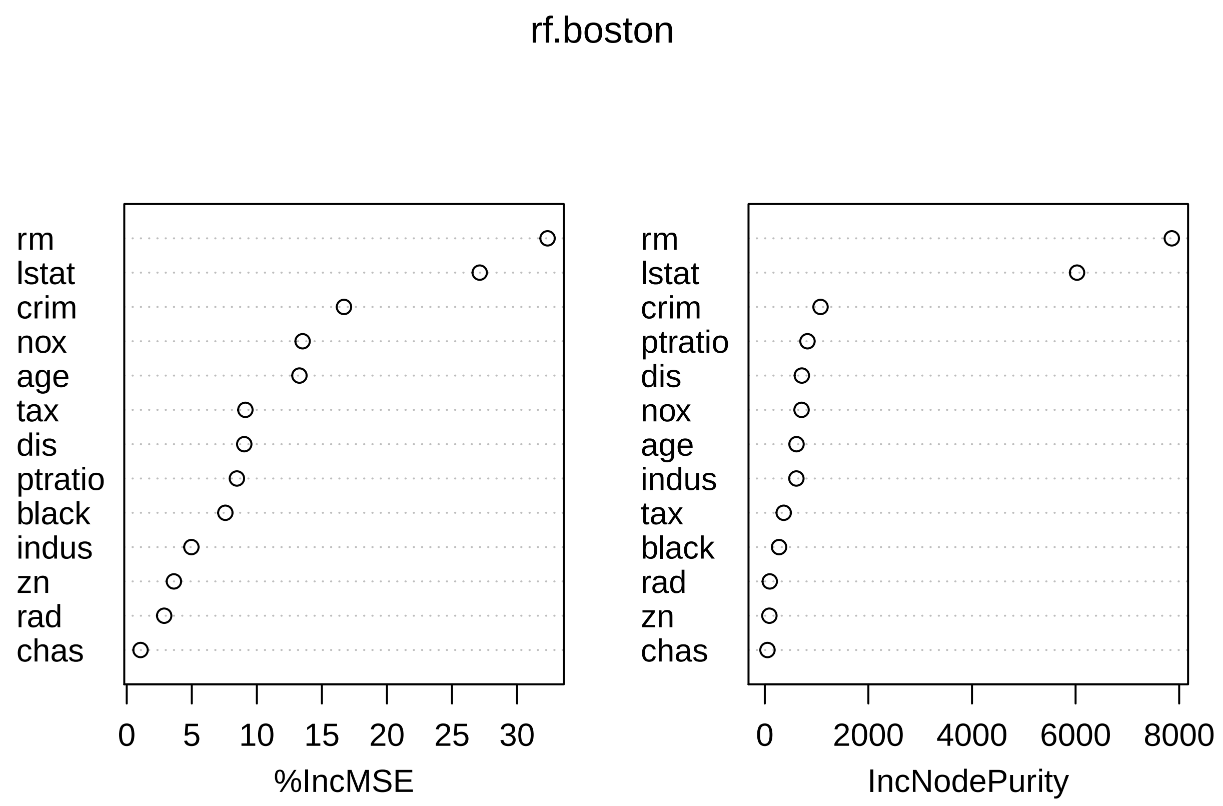
We can see the importance of each variable using the importance() function:

```r
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
```

The left column %IncMSE is based on the mean decrease of accuracy in predictions on the OOB samples when given variable is excluded from the model. IncNodePurity is a measure of total decrease in node purity that results from splits over that variable, averaged over all trees. We can plot these below:

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



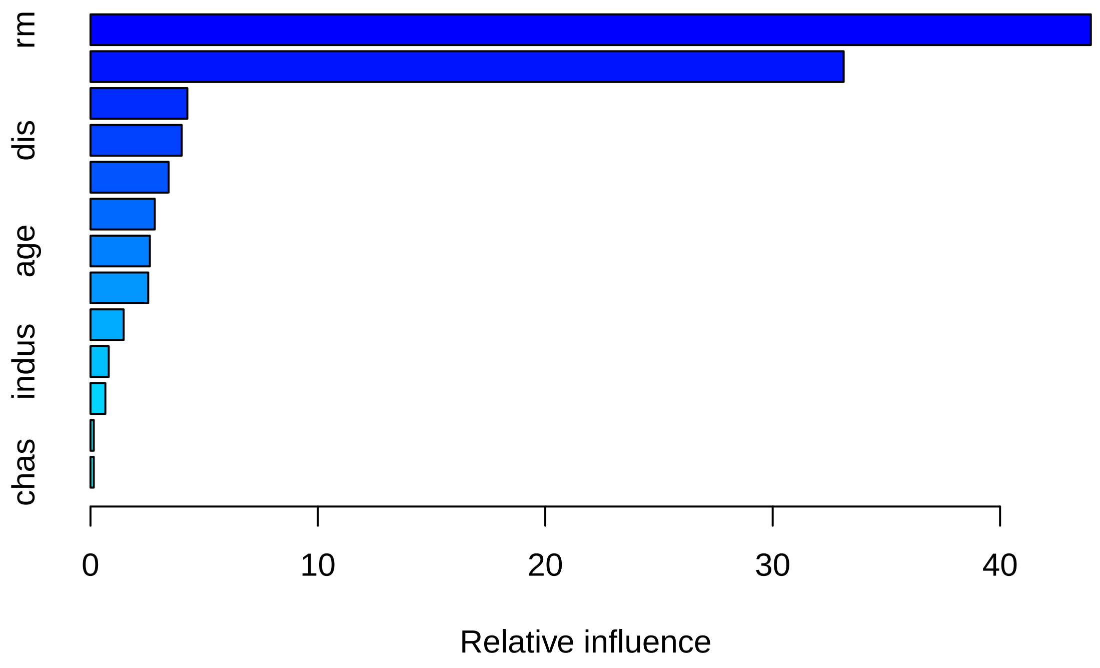
We can see that rm and lstat are by far the two most important variables

Boosting

```
library(gbm)
```

```
## Loaded gbm 2.1.8
```

```
set.seed(1)

boost.boston <- gbm(medv~.,
                    data = Boston[train,],
                    distribution = "gaussian",
                    n.trees = 5000,
                    interaction.depth = 4) # would use dist = bernoulli if it were classification

summary(boost.boston)
```

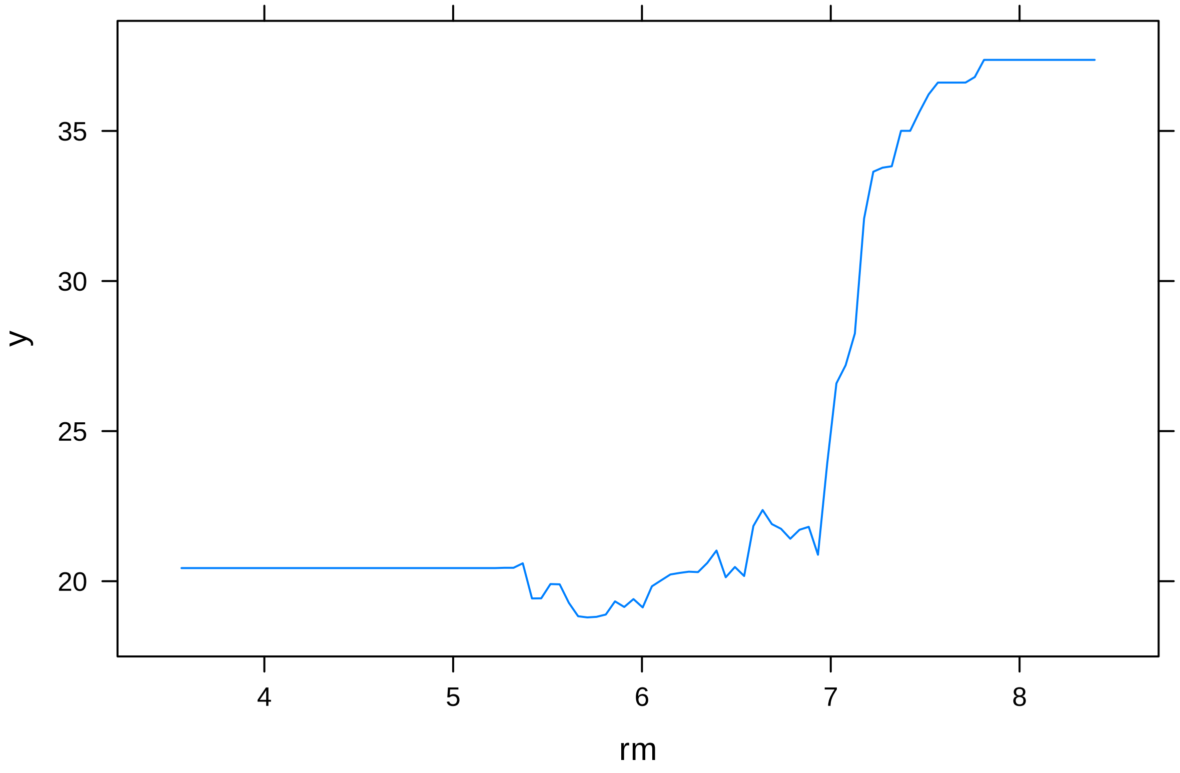


```
##             var    rel.inf
## rm           rm 43.9919329
## lstat     lstat 33.1216941
## crim       crim  4.2604167
## dis         dis  4.0111090
## nox         nox  3.4353017
## black     black  2.8267554
## age         age  2.6113938
## ptratio ptratio  2.5403035
## tax         tax  1.4565654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

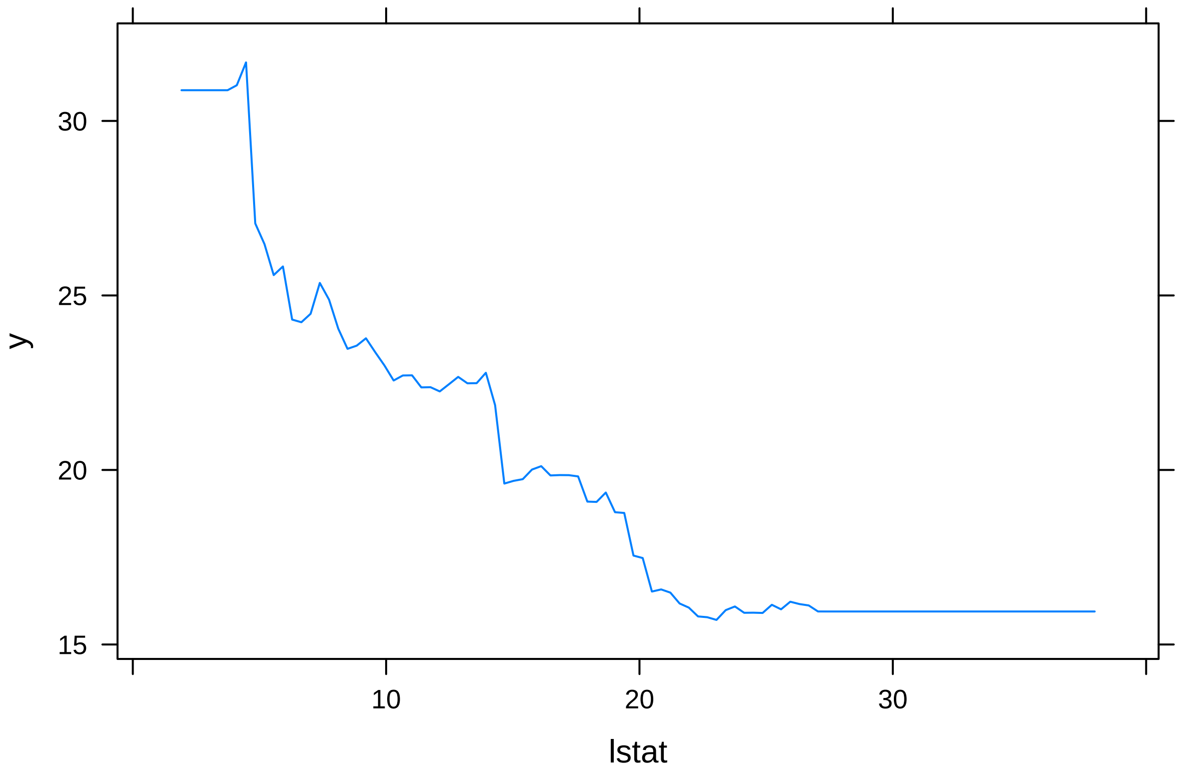
Again, we can see that rm and lstat are by far the most important variables

We can produce a *partial dependence plot* for these two variables, which illustrate the marginal effect of the variables on the response after *integrating out* the other variables:

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



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



We can see that house prices are increasing with rm and decreasing with lstat

Finally, we can use the boosted model to fit the test data:

```
boost.boston <- gbm(medv~.,
                    data = Boston[train,],
                    distribution = "gaussian",
                    n.trees = 5000,
                    interaction.depth = 4,
                    verbose = F,
                    )

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

```
## [1] 18.18255
```

The shrinkage default for gbm() is 0.1. We can see an incremental improvement of using boosting over random forests (improvement by about 1%).