Module 8: Tree-based Methods

TMA4268 Statistical Learning V2022

Stefanie Muff, Department of Mathematical Sciences, NTNU

March 7 and 10, 2022

Acknowledgements

- The course was originally developed by Mette Langaas (original material: https://github.com/mettelang/StatisticalLearningSpring2019). Mette did a fantastic job and I am very thankful that I was allowed to modify and use her material.
- Some of the figures and slides in this presentation are taken (or are inspired) from James et al. (2013).

Introduction

Learning material for this module

- James et al (2013): An Introduction to Statistical Learning. Chapter 8.
- All the material presented on these module slides.

What will you learn?

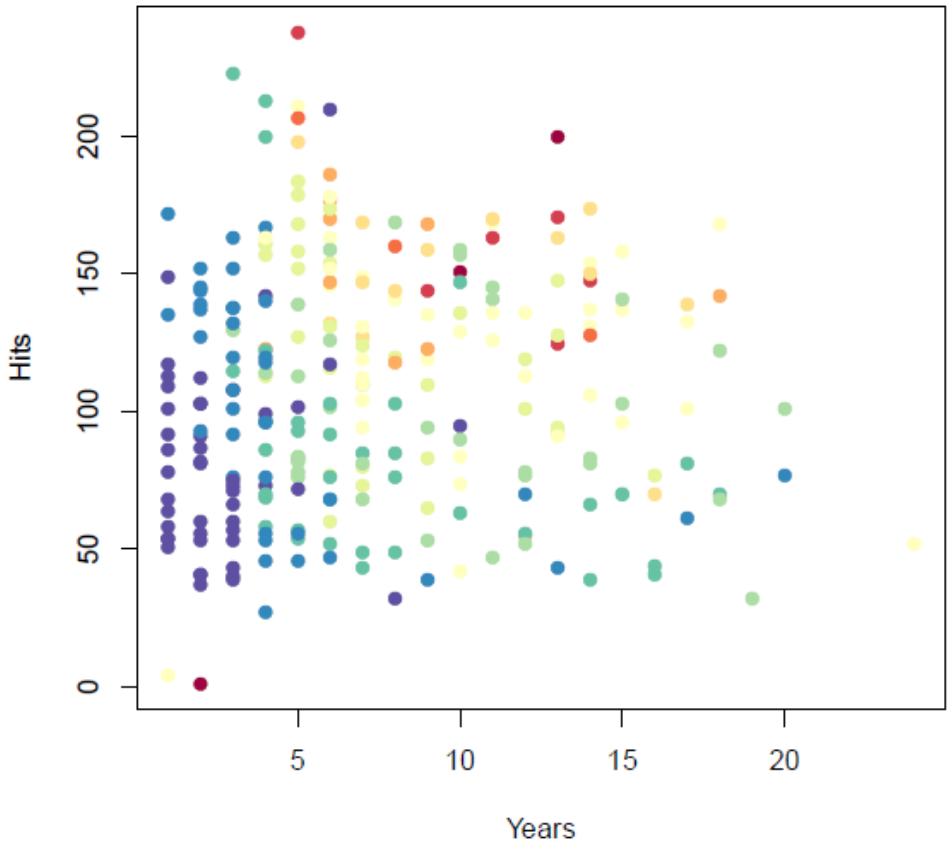
You will get to know

- Decision trees
  - Regression trees
  - Classification trees
- Pruning a tree
- Bagging
- Variable importance
- Random forests
- Boosting

and learn how to apply all that.

Example 1 (from chapter 8.1; `Hitters` data)

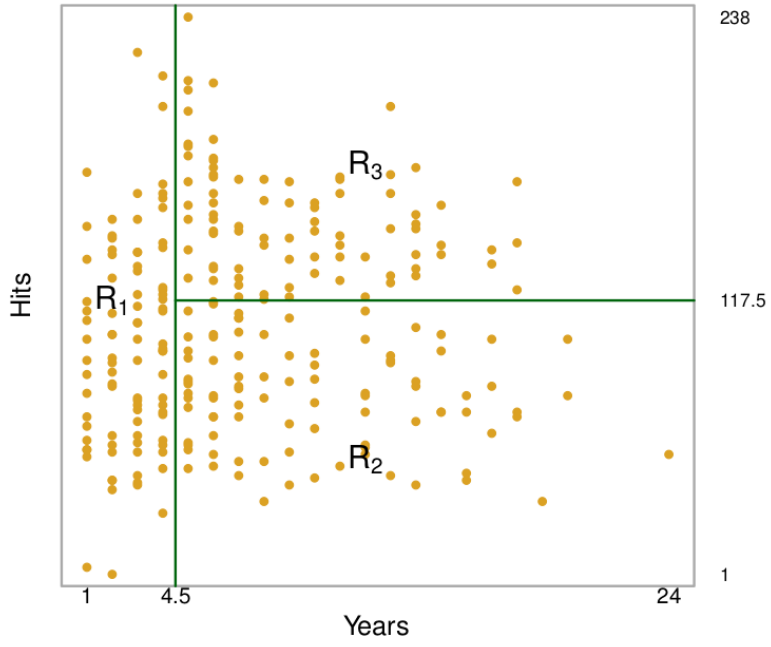
- Baseball players' salaries may depend on their experience (in years) and the number of hits.
- High salaries (yellow, red) vs low salaries (blue, green), salaries given on log-scale. How can these be stratified for prediction of the salary?



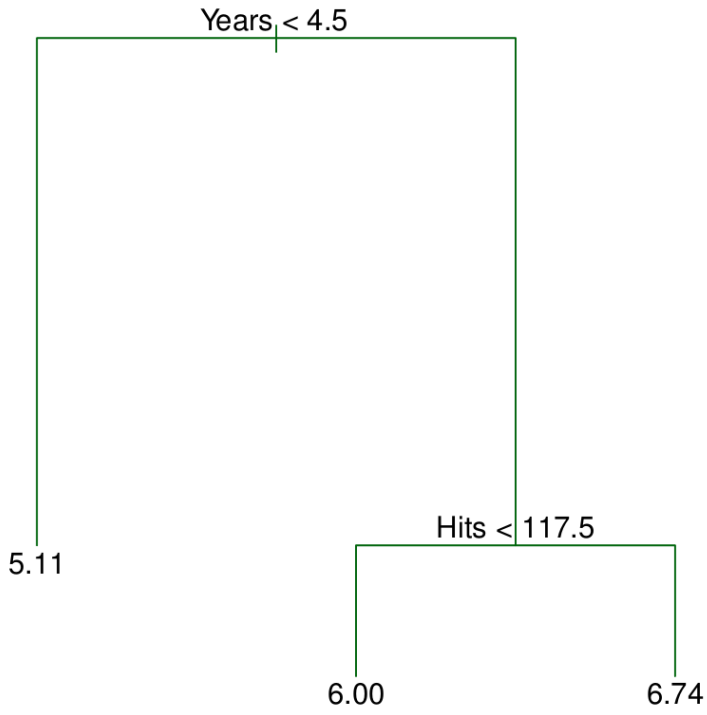
Main idea of tree-based methods

- Divide the area into rectangles with similar salaries.
- Idea: Derive a set of decision (splitting) rules for segmenting the predictor space into a number of finer and finer regions.
- All points in the same region will be given the same predictive value (the mean of all values in that square, or a majority vote).

Visualisaztion in two dimensions:



- CARTs (Classification and regression trees) are usually drawn upside down, where the top node is called the *root*.
- The series of splitting rules can be visualized with a *regression tree*.
- The following tree (which corresponds to the split in the previous slide) has three *leafs* (terminal nodes), and two *internal nodes*.



Interpretation

- Years is the most important factor in determining Salary, and players with less experience earn lower salaries than more experienced players.
- Given that a player is less experienced, the number of Hits that he made in the previous year seems to play little role in his Salary.
- But among players who have been in the major leagues for five or more years, the number of Hits made in the previous year does affect Salary, and players who made more Hits last year tend to have higher salaries.
- Compared to a regression model, it is easy to display, interpret and explain.

Regression tree (continous outcome)

Assume that we have a dataset consisting of n pairs (x_i, y_i), $i = 1, \ldots, n$, and each predictor is $x_i = (x_{i1}, x_{i2}, ..., x_{ip})$. The aim is to predict y_i.

Two steps:

1. Divide the predictor space into non-overlapping regions $R_1, R_2, \ldots, R_J$.
2. For every observation that falls into region R_j we make the same prediction - which is the mean of the responses for the training observations that fall into R_j.

But: How to divide the predictor space into non-overlapping regions $R_1, R_2, \ldots, R_J$?

We could try to minimize the RSS (residual sums of squares) on the training set given by

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

where $\hat{y}_{R_j}$ is the mean response for the training observations in region j. The mean $\hat{y}_{R_j}$ is also the predicted value for a new observations that falls into region j.

To do this we need to consider every partition of the predictor space, and compute the RSS for each partition.

But: An exhaustive search over possible splits is *computationally infeasible*! [1]

[1] In fact, constructing optimal binary decision trees is an NP-complete problem (Hyafil and Rivest, 1976).

Recursive binary splitting

- A *greedy* approach is taken (aka top-down) - called *recursive binary splitting*: Find a split that minimizes RSS at each step[2].
- Start at the top of the tree and divide the predictor space into two regions $R_1(j,s) = \{x \mid x_j < s\}$ and $R_2(j,s) = \{x \mid x_j \geq s\}$, by making a decision rule for one of the predictors $x_1, x_2, ..., x_p$.
- We thus need to find the (predictor) j and (splitting point) s that 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 $\hat{y}_{R_1}$ and $\hat{y}_{R_2}$ are the mean responses for the training observations in $R_1(j,s)$ and $R_2(j,s)$ respectively. This way we get the two first branches in our decision tree.

[2]This does not necessarily give the optimal global solution, but will give the best solution at each split, given what is done previously.

- We repeat the process to make branches further down in the tree.
- For every iteration we let each single split depend on *only one of the predictors*, giving us two new branches.
- This is done *successively* and in each step we choose the split that gives the *best split at that particular step*, *i.e.*, the split that gives the smallest RSS.
- However, this time, instead of splitting the entire predictor space, we split only *one of the previously identified regions*.
- Continue splitting the predictor space until we reach some *stopping criterion*. For example we stop when a region contains less than 10 observations or when the reduction in the RSS is smaller than a specified limit.

Q: Why is this algorithm called *greedy*?

Regression tree: ozone example

Consider the `ozone` data set from the `ElemStatLearn` library. The data set consists of 111 observations on the following variables:

- `ozone` : the concentration of ozone in ppb
- `radiation`: the solar radiation (langleys)
- `temperature` : the daily maximum temperature in degrees F
- `wind` : wind speed in mph

```
head(myozone)
```

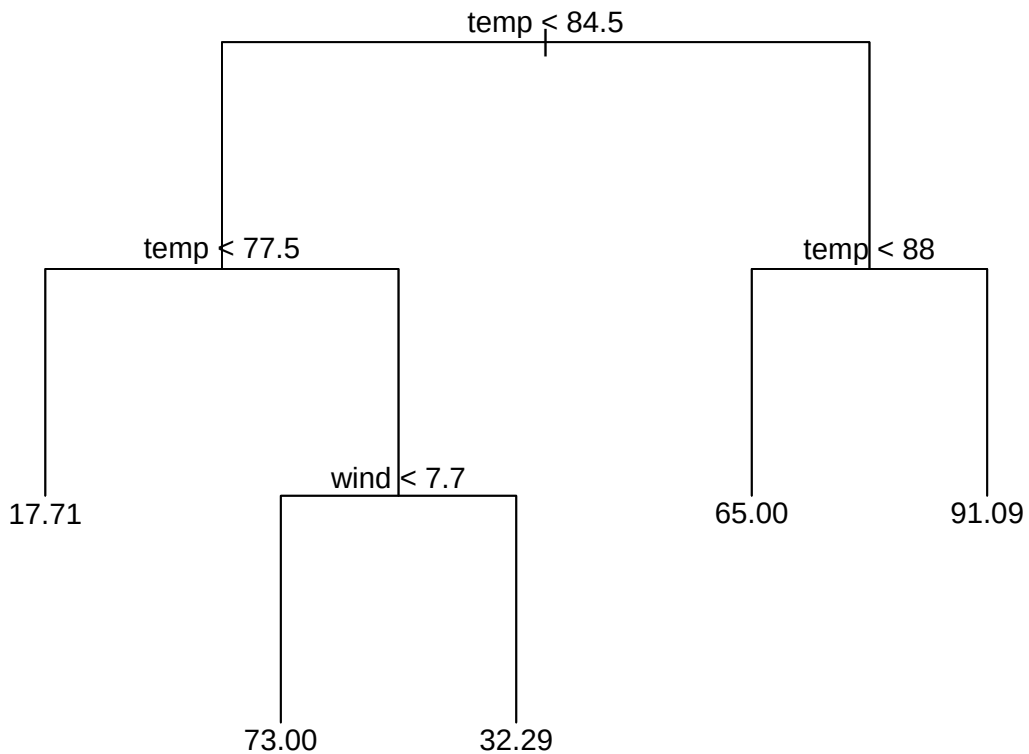
```
##   ozone radi temp wind
## 1    41  190   67  7.4
## 2    36  118   72  8.0
## 3    12  149   74 12.6
## 4    18  313   62 11.5
## 5    23  299   65  8.6
## 6    19   99   59 13.8
```

Let's fit a regression tree with **ozone** as our response variable and **temperature** and **wind** as predictors.

```
ozone.trainID = sample(1:111, 75)
ozone.train = myozone[ozone.trainID, ]
ozone.test = myozone[-ozone.trainID, ]
```

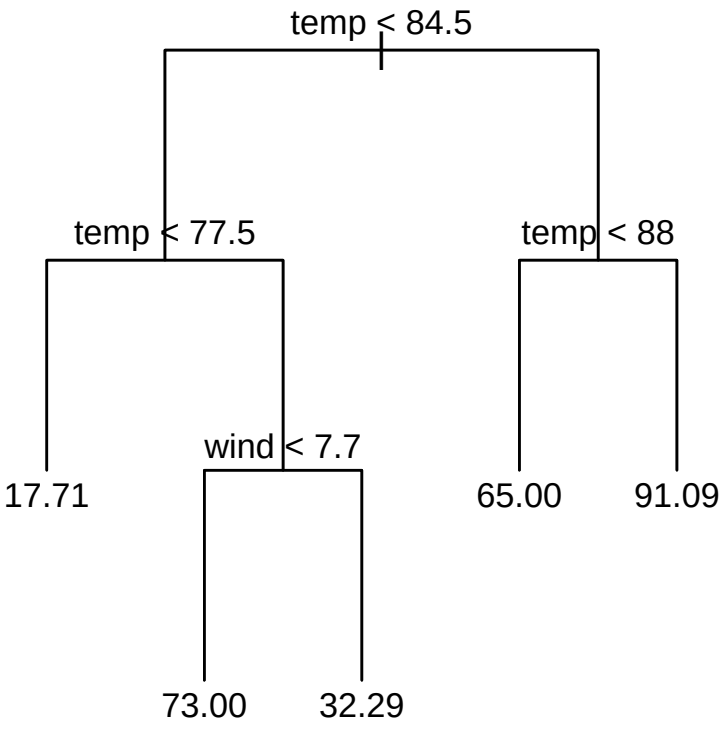
Use the default settings in the tree function:

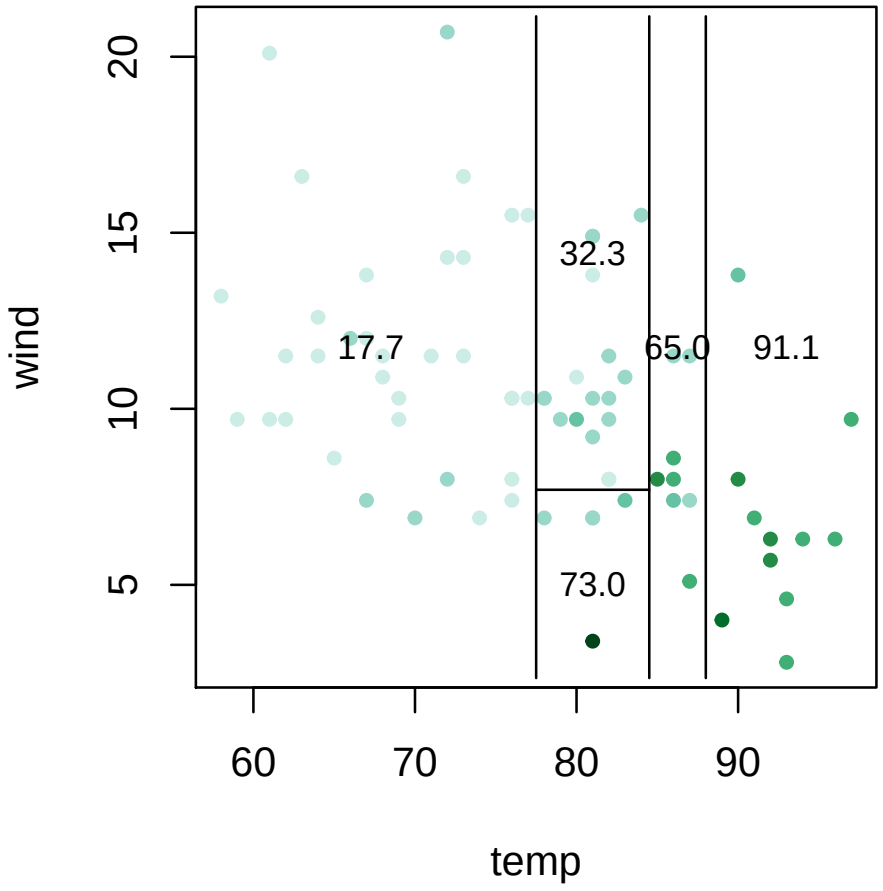
```
library(tree)
ozone.tree = tree(ozone ~ temp + wind, data = ozone.train)
```



- We see that `temperature` is the "most important" predictor for predicting the ozone concentration. Observe that we can split on the same variable several times.
- Focus on the regions R_j, $j = 1, \ldots, J$. What is J here?

Tree- vs region plot





Q:

- Explain the connection between the tree and the region plot.
- Advantages and disadvantages of letting each single split depend on only one of the predictors?
- Does our tree include interactions between variables?

R: function `tree` in library `tree`

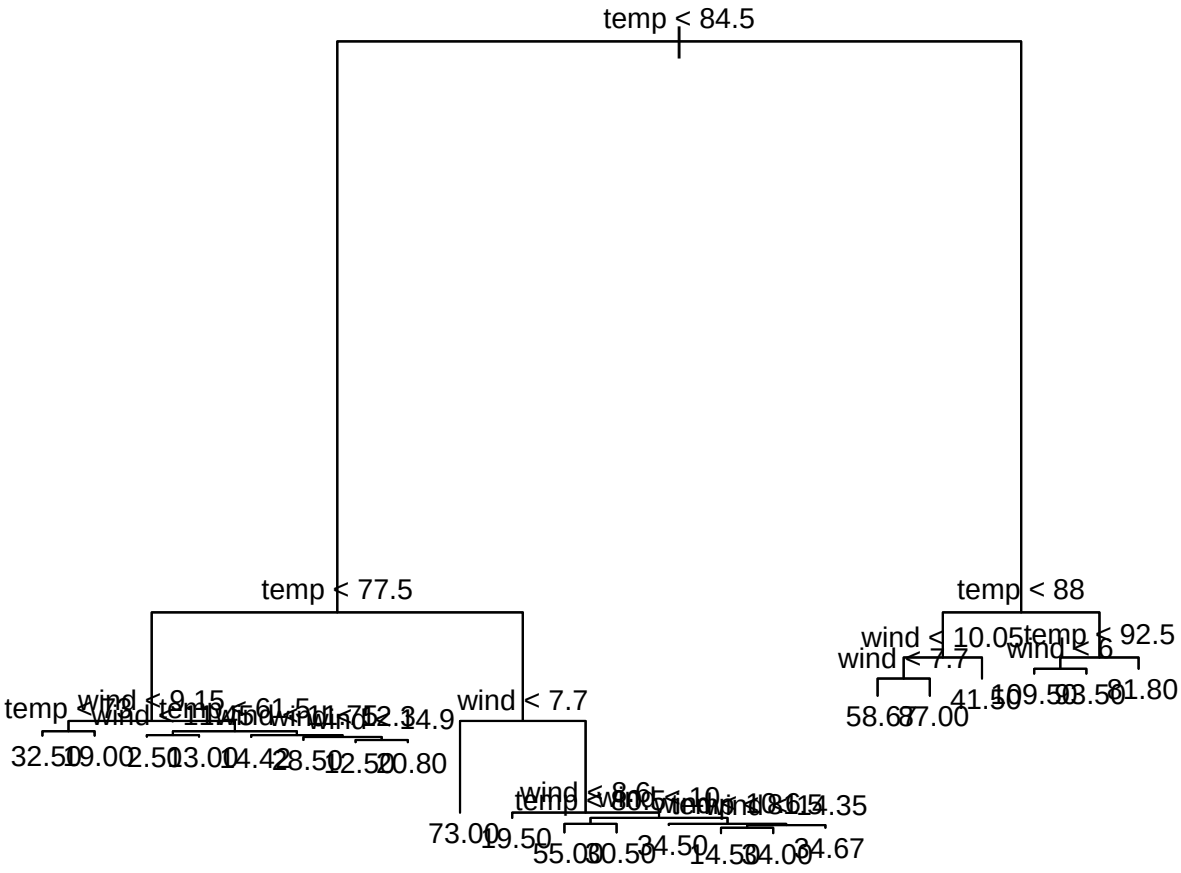
- R package `tree` by Brian D. Ripley (Ripley 2019).
- Note: The default choice for a function to minimize is the deviance, and for normal data (as we may assume for regression), the deviance is proportional to the RSS.
- A competing R function is `rpart`, explained in https://cran.r-project.org/web/packages/rpart/vignettes/longintro.pdf

Stopping criterion

- When building the tree, we can (in principle) split until each leaf corresponds to one data point.
- Usually we use a less stringent stopping criterion, like the minimal number of nodes per region and/or the minimum reduction in the RSS.
- For example, the default in `tree()`: is given by `mincut=5, minsize=10, mindev=0.01`, thus a minimal number of observations in a node is 5, the smallest nodes that are potentially split is 10, and the minimal reduction in deviance (RSS) equal to 0.01.
- We could make the ozone tree much deeper:

```
ozone.tree2 = tree(ozone ~ temp + wind, data = ozone.train, control = tree.control(75,
    mincut = 2, minsize = 4, mindev = 0.001))
```

With the above command, the tree would become very fine. Overfitting?



Tree performance

Test the predictive performance of our regression tree by using a training and a test set. But:

How do we know if our tree performs good, or if there would be trees with a better predictive performance?

Pruning

- If we have a data set with many predictors or choose a stringent stopping criterion, we may fit a (too) large tree. Thus, the number of observations from the training set that falls into some of the regions R_j may be small $\rightarrow$ *overfitting*?
- A smaller tree with fewer splits leads to fewer regions $R_1, \ldots, R_J$ with more observations. This might lead to lower variance and better interpretation at the cost of more bias.
- One possible alternative to the process described above is to grow the tree only so long as the decrease in the RSS due to each split exceeds some (high) threshold.
- This strategy will result in smaller trees, but is *too short-sighted*: a seemingly worthless split early on in the tree might be followed by a very good split — that is, a split that leads to a large reduction in RSS later on.

Cost complexity pruning

- Better idea: to grow a very large tree T_0, and then *prune* it back in order to obtain a *subtree*.
- *Cost complexity pruning* is used for this: We try to find a subtree $T \subset T_0$ that (for a given value of α) minimizes

$$C_\alpha(T) = Q(T) + \alpha|T|,$$

  where $Q(T)$ is our cost function, $|T|$ is the number of terminal nodes in tree T. The parameter α is then a parameter penalizing the number of terminal nodes, ensuring that the tree does not get too many branches.
- For regression trees (continous outcome) we choose

$$Q(T) = \sum_{m=1}^{|T|} \sum_{x_i \in R_m} (y_i - \hat{y}_{R_m})^2 \ .$$

- For $\alpha > 0$ we get a pruned tree (note: the same pruned tree within a small range of α).
- For $\alpha = 0$ we get T_0.
- As α increases we get shorter and shorter trees.

So, which value of α is best?

- We can use K-fold cross-validation to find out!
- Importantly, by increasing α, branches get pruned in a *hierarchical (nested) fashion*.

Please study this note from Bo Lindqvist in MA8701 in 2017 - Advanced topics in Statistical Learning and Inference for an example of how we perform cost complexity pruning in detail. Alternatively, this method, with proofs, are given in Ripley (1996), Section 7.2.

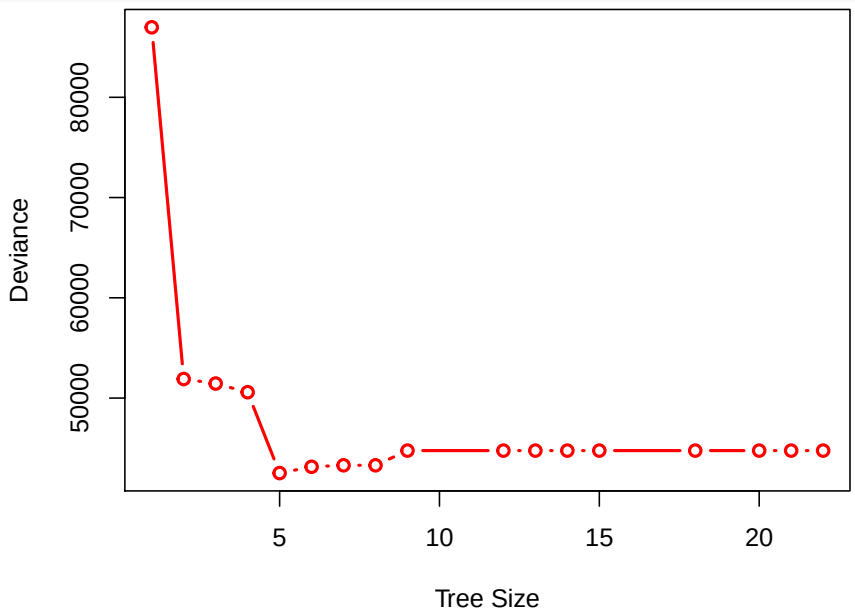
Building a regression tree: Algorithm 8.1

1. Use recursive binary splitting to grow a large tree on the training data, stopping only when each terminal node has fewer than some minimum number of observations.
2. Apply cost complexity pruning to the large tree in order to obtain a sequence of best subtrees, as a function of α.
3. Use K-fold cross-validation to choose α. That is, divide the training observations into k folds. For each $k = 1, \ldots, K$:
   - Repeat Steps 1 and 2 on all but the kth fold of the training data.
   - Evaluate the mean squared prediction error on the data in the left-out kth fold, as a function of α.
   - Average the results for each value of α, and pick α to minimize the average error.
4. Return the subtree from Step 2 that corresponds to the chosen value of α.

Pruning the ozone tree

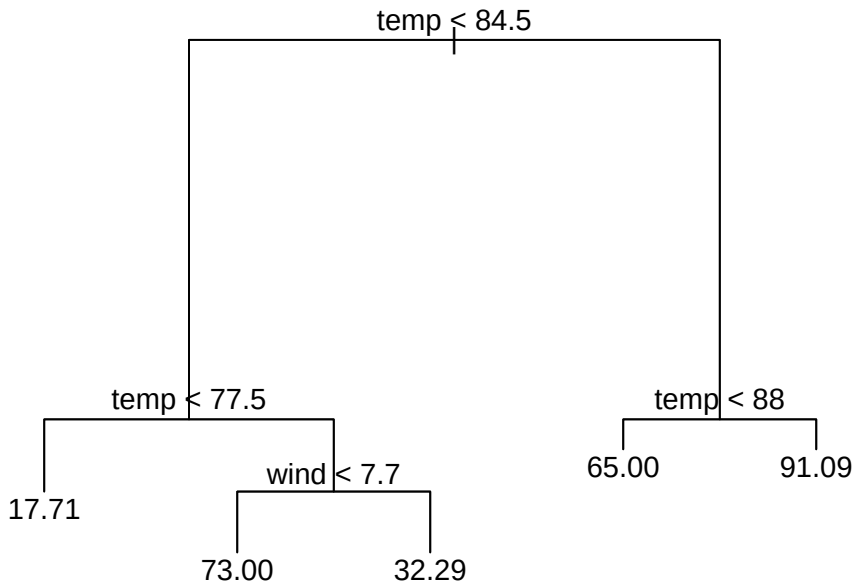
Let us start with the tree with many leaves, and then prune it with 5-fold CV. We can then plot the CV error as a function of tree size (instead of α – why?):

```
set.seed(563)
ozone.tree <- tree(ozone ~ temp + wind, data = ozone.train, control = tree.control(nrow(myo
    mincut = 2, minsize = 4, mindev = 0.001))
cv.ozone <- cv.tree(ozone.tree, K = 5)
plot(cv.ozone$dev ~ cv.ozone$size, type = "b", lwd = 2, col = "red",
    xlab = "Tree Size", ylab = "Deviance")
```



Interestingly, a tree with 5 leaves performs best - which corresponds to the original choice:

```
prune.ozone <- prune.tree(ozone.tree, best = 5)
plot(prune.ozone)
text(prune.ozone, pretty = 0)
```



Classification trees (binary or categorical outcome)

- Now allow for $K \geq 2$ number of classes for the response.
- Building a decision tree in this setting is similar to building a regression tree for a quantitative response, but there are two main differences: *the prediction* and *the splitting criterion*.

1) The prediction:

- In the regression case we use the mean value of the responses in R_j as a prediction for an observation that falls into region R_j.

- For the *classification case*, however, we have two possibilities:

  - **Majority vote**: Predict that the observation belongs to the most commonly occurring class of the training observations in R_j.

  - Estimate the **probability** that an observation x_i belongs to a class k, $\hat{p}_{jk}(x_i)$, given as the proportion of class k training observations in region R_j. Region j has N_j observations and with n_{jk} observations lying in class k:

$$\hat{p}_{jk} = \frac{1}{N_j} \sum_{i:x_i \in R_j} I(y_i = k) = \frac{n_{jk}}{N_j}.$$

2) The splitting criterion: We do not use RSS as a splitting criterion for a qualitative variable. Instead we can use some *measure of impurity* of the node. For leaf node j and class $k = 1, \ldots, K$:

- **Gini index**:

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

  which is small if all of the $\hat{p}_{jk}$'s are close to 0 or 1.

- **Cross entropy**:

$$D = -\sum_{k=1}^{K} \hat{p}_{jk} \log \hat{p}_{jk} \ .$$

  Since $0 \leq \hat{p}_{jk} \leq 1$, it follows that $0 \leq -\hat{p}_{jk} \log \hat{p}_{jk}$, with values near zero if $\hat{p}_{jk}$ is close to 0 or 1.

When making a split in our classification tree, we want to minimize the Gini index or the cross-entropy.

Why don't we just minimize the misclassification error

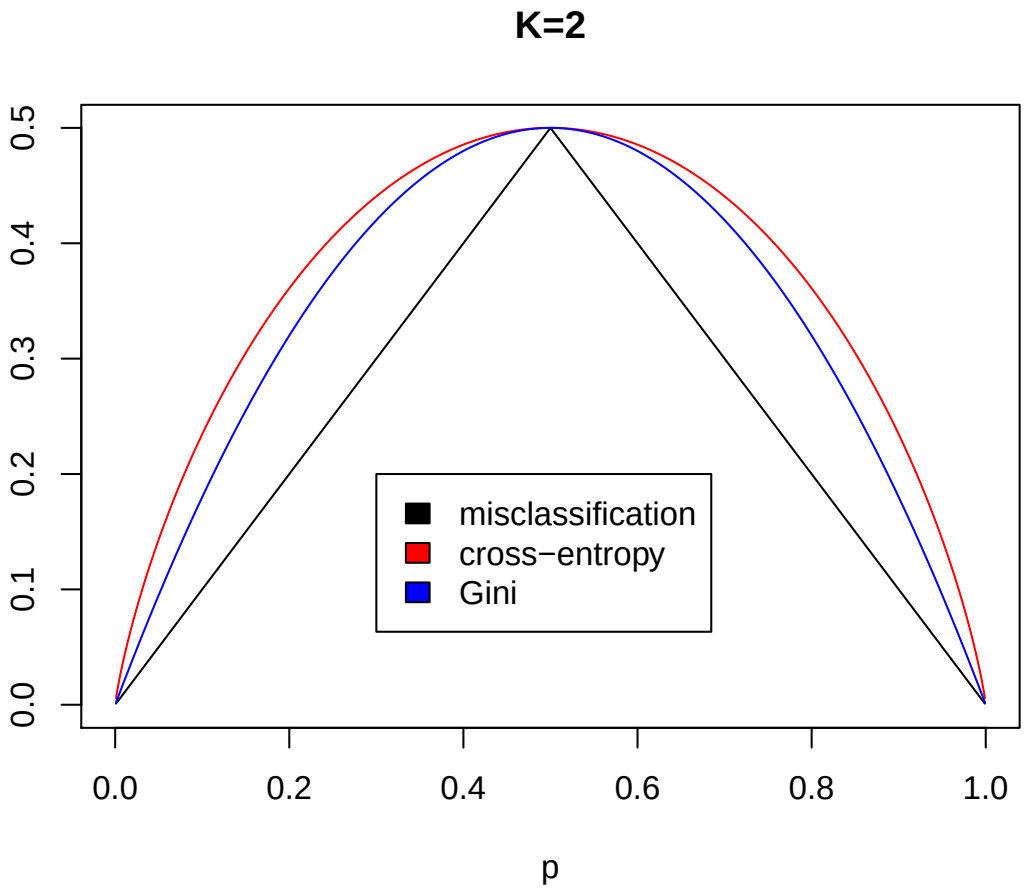
$$E = 1 - \max_k \hat{p}_{jk} \ ?$$

A:

- The Gini index and Entropy are *measure of impurity*. They are more sensitive to changes in the node probabilities.
- Example for two classes: Assume we have two classes with 400 nodes each, written as $(400, 400)$. Now we can choose between two splits:
  - Split 1: (100,300) and (300,100)
  - Split 2: (200,400) and (200,0).

Both splits result in 25% misclassification. Which of these splits is better? Probably the second one, because it produces a *pure node*.

- Moreover, The Gini index and Entropy are differentiable (preferred for numerical optimization!)



Example: Detection of Minor Head Injury

(Artificial data)

- Data from patients that enter hospital. The aim is to quickly assess whether a patient as a brain injury or not.
- Patients are investigated and (possibly) asked questions.
- *Our job*: To build a good model to predict quickly if someone has a brain injury. The method should be
  - **easy** to interpret for the medical personell that are not skilled in statistics, and
  - **fast**, such that the medical personell quickly can identify a patient that needs treatment.

The dataset includes data about 1321 patients and is a modified and smaller version of the (simulated) dataset `headInjury` from the `DAAG` library.

```
##    amnesia bskullf GCSdecr GCS.13 GCS.15 risk consc oskullf vomit brain.injury
## 3        0       0       0      0      0    0     0       0     0            0
## 9        0       0       0      0      0    1     0       0     0            0
## 11       0       0       0      0      0    0     0       0     0            0
## 12       1       0       0      0      0    0     0       0     0            0
## 14       0       0       0      0      0    0     0       0     0            0
## 16       0       0       0      0      0    0     0       0     0            0
##    age
## 3   44
## 9   67
## 11  62
## 12   1
## 14  55
## 16  63
```

Split with cross-entropy

We use the 850 training samples to get a tree using cross-entropy (deviance):

```
tree.HIClass = tree(brain.injury ~ ., data = headInjury2, subset = train,
    split = "deviance")
summary(tree.HIClass)
```

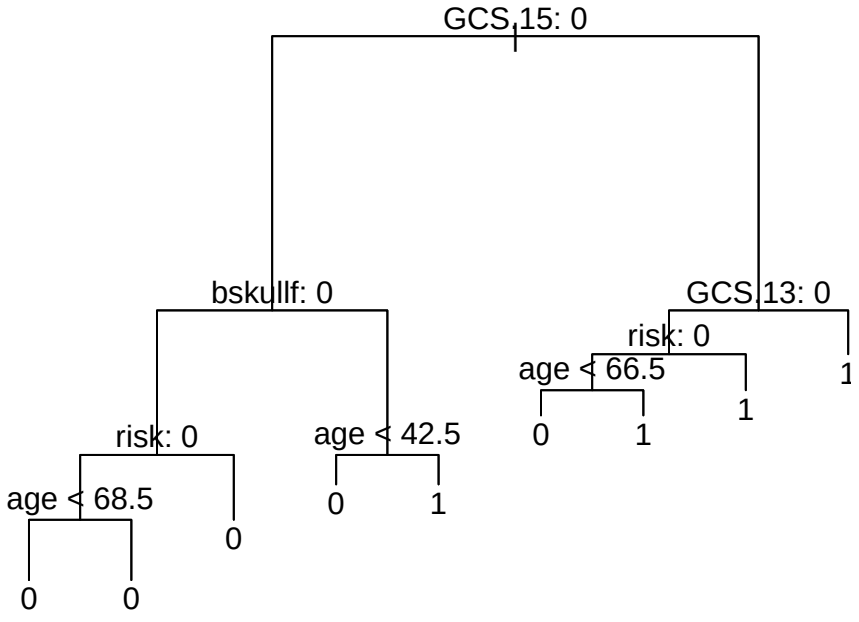
```
##
## Classification tree:
## tree(formula = brain.injury ~ ., data = headInjury2, subset = train,
##     split = "deviance")
## Variables actually used in tree construction:
## [1] "GCS.15"  "bskullf" "risk"    "age"     "GCS.13"
## Number of terminal nodes:  9
## Residual mean deviance:  0.66 = 555 / 841
## Misclassification error rate: 0.126 = 107 / 850
```

Remark: the deviance $(-2\log(L))$ is a scaled version of the cross entropy:

$$-2\sum_{k=1}^{K} n_{jk} \log \hat{p}_{jk} \ , \text{where} \quad \hat{p}_{jk} = \frac{n_{jk}}{N_j} \ ,$$

thus `split="deviance"` implies that we split according to the entropy criterion.

```
plot(tree.HIClass, type = "proportional")
text(tree.HIClass, pretty = 1)
```



- With `type="proportional"`, the length of branches are now proportional to the decrease in impurity.
- The classification tree has two terminal nodes with factor "0" originating from the same branch. Why do we get this "unnecessary" split?

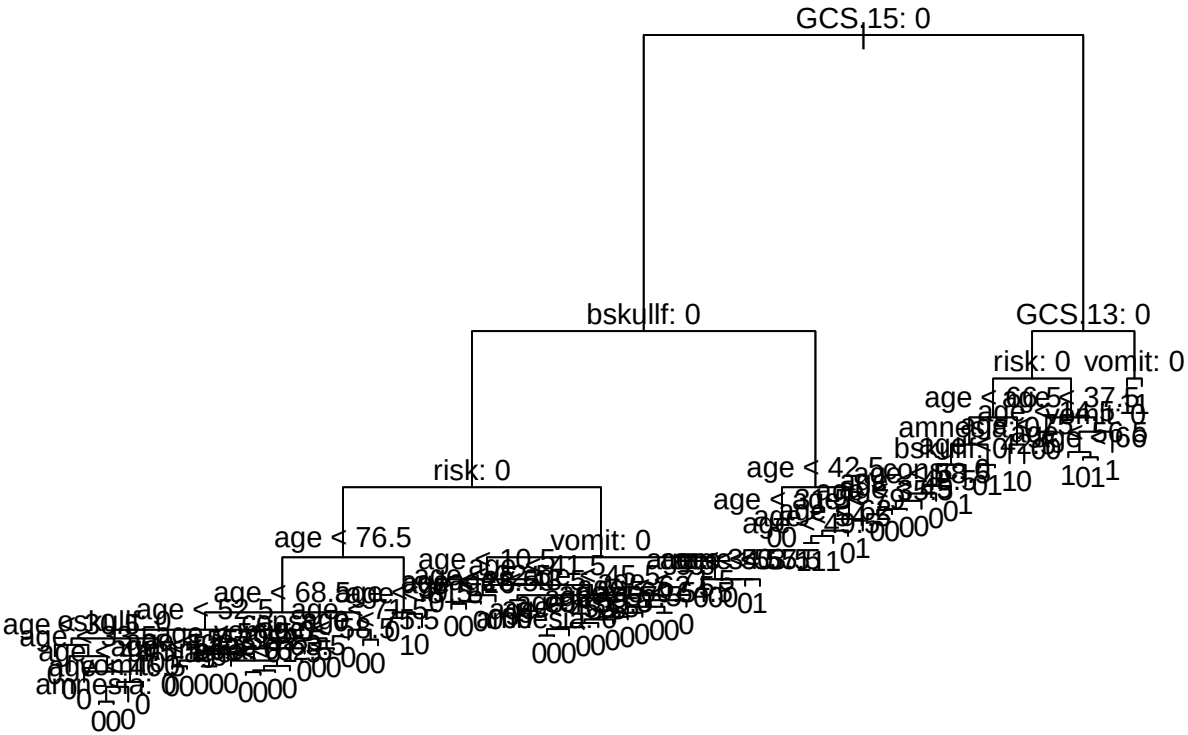
Split with Gini index

The same analysis with the Gini index:

```
tree.HIClassG = tree(brain.injury ~ ., headInjury2, subset = train, split = "gini")
summary(tree.HIClassG)
```

```
##
## Classification tree:
## tree(formula = brain.injury ~ ., data = headInjury2, subset = train,
##     split = "gini")
## Variables actually used in tree construction:
## [1] "GCS.15"  "bskullf" "risk"    "age"     "oskullf" "vomit"   "amnesia"
## [8] "consc"   "GCS.13"
## Number of terminal nodes:  75
## Residual mean deviance:  0.52 = 403 / 775
## Misclassification error rate: 0.109 = 93 / 850
```

```
plot(tree.HIClassG, type = "proportional")
text(tree.HIClassG, pretty = 1)
```



This is a very bushy tree!

Checking predictions

We also use the classification tree to predict the status of the patients in the test set.

With the deviance:

```
library(caret)
tree.pred = predict(tree.HIClass, headInjury2[test, ], type = "class")
(confMat <- confusionMatrix(tree.pred, reference = headInjury2[test,
    ]$brain.injury)$table)
```

```
##           Reference
## Prediction   0   1
##          0 356  42
##          1  24  49
```

```
1 - sum(diag(confMat))/sum(confMat[1:2, 1:2])
```

```
## [1] 0.140127
```

With the Gini-index:

```
tree.predG = predict(tree.HIClassG, headInjury2[test, ], type = "class")
(confMatG <- confusionMatrix(tree.predG, reference = headInjury2[test,
    ]$brain.injury)$table)
```

```
##           Reference
## Prediction   0   1
##          0 354  47
##          1  26  44
```

```
1 - sum(diag(confMatG))/sum(confMatG[1:2, 1:2])
```

```
## [1] 0.154989
```

Group discussion

Study the confusion matrices on the previous slides:

- Which algorithm makes more errors?
- Is one type of mistakes more severe than the other?
- Discuss if/how it is possible to change the algorithm in order to decrease the number of severe mistakes.

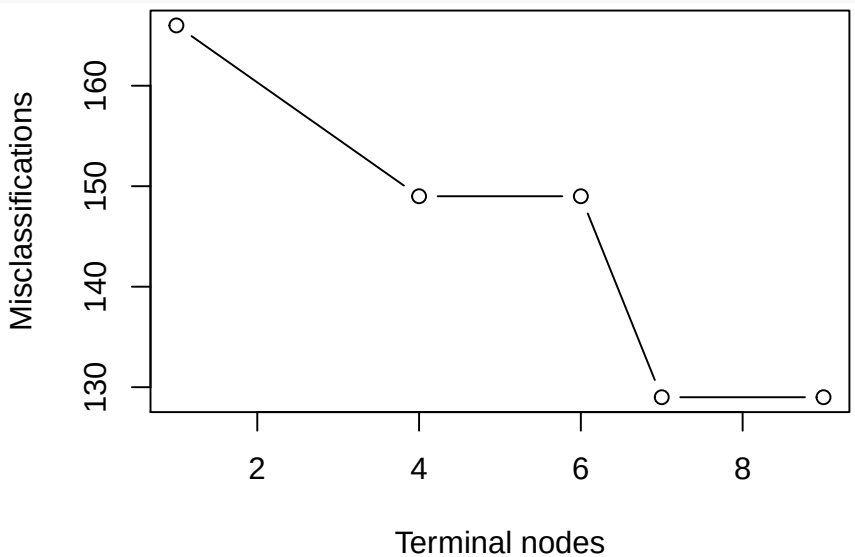
Building a classification tree: Algorithm 8.1

- Like for regression trees, we would like to find classification trees that are good at prediction.
- Algorithm 8.1 can be used with misclassification, the Gini index, or cross-entropy instead of the RSS as quality measure.

Finding an optimal classification tree

We prune the classification tree that was built with the deviance criterion. We use the misclassification error to do so[3]:

```
set.seed(1)
cv.head = cv.tree(tree.HIClass, FUN = prune.misclass)
plot(cv.head$size, cv.head$dev, type = "b", xlab = "Terminal nodes",
    ylab = "Misclassifications")
```



[3]While trees should be built using the deviance or the Gini index, pruning is typically done with the misclassification criterion when the aim is predictive accuracy.

The function `cv.tree` automatically does 10-fold cross-validation. `dev` is here the number of misclassifications.

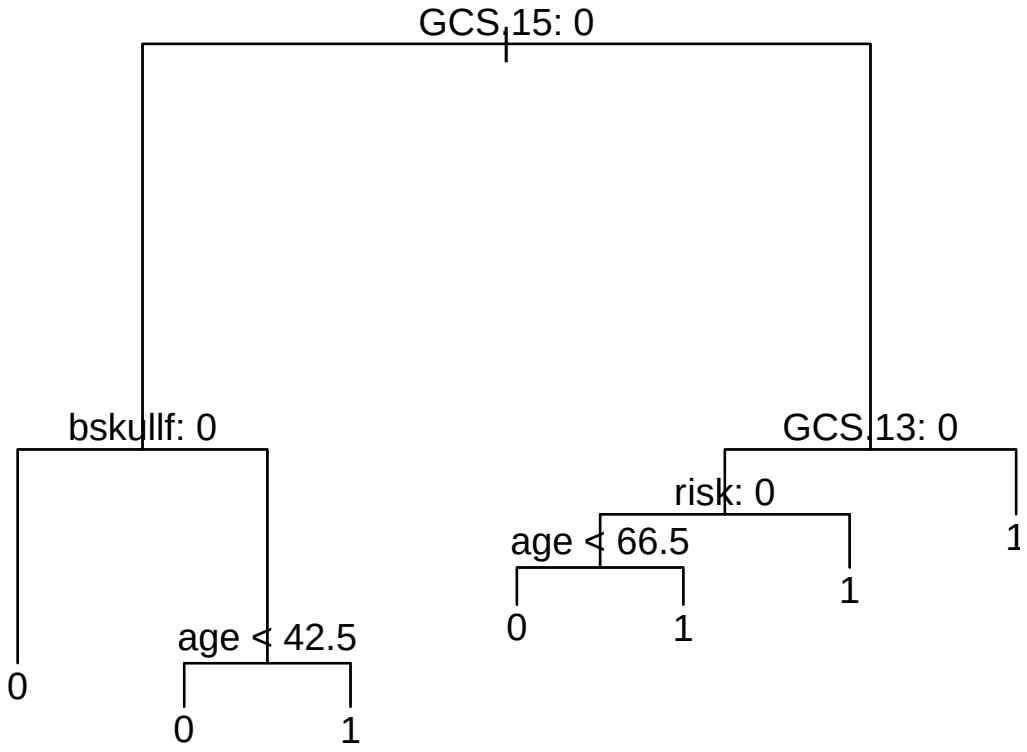
```
print(cv.head)
```

```
## $size
## [1] 9 7 6 4 1
##
## $dev
## [1] 129 129 149 149 166
##
## $k
## [1] -Inf  0.0  6.0  6.5 11.0
##
## $method
## [1] "misclass"
##
## attr(,"class")
## [1] "prune"         "tree.sequence"
```

- We have done cross-validation on our training set of 850 observations. According to the plot, the number of misclassifications is as low for 7, 8 or 9 nodes, so we choose 7 terminal nodes as the smallest model (lowest variance).
- We prune the classification tree according to this value:

```
prune.HIClass = prune.misclass(tree.HIClass, best = 7)
```

```
plot(prune.HIClass)
text(prune.HIClass, pretty = 1)
```



→ No unnecessary splits left, and we have a simple and interpretable decision tree.

How is the predictive performance of the model affected?

```
tree.pred.prune <- predict(prune.HIClass, headInjury2[test, ], type = "class")
(confMat <- confusionMatrix(tree.pred.prune, headInjury2[test, ]$brain.injury)$table)
```

```
##            Reference
## Prediction   0   1
##          0 356  42
##          1  24  49
```

```
1 - (sum(diag(confMat))/sum(confMat[1:2, 1:2]))
```

```
## [1] 0.140127
```

→ We see that the misclassification rate is as small as before, thus the pruned tree is as good as the original tree for the test data.

→ If you want, you can apply the same pruning procedure to the very bushy Gini-grown tree.

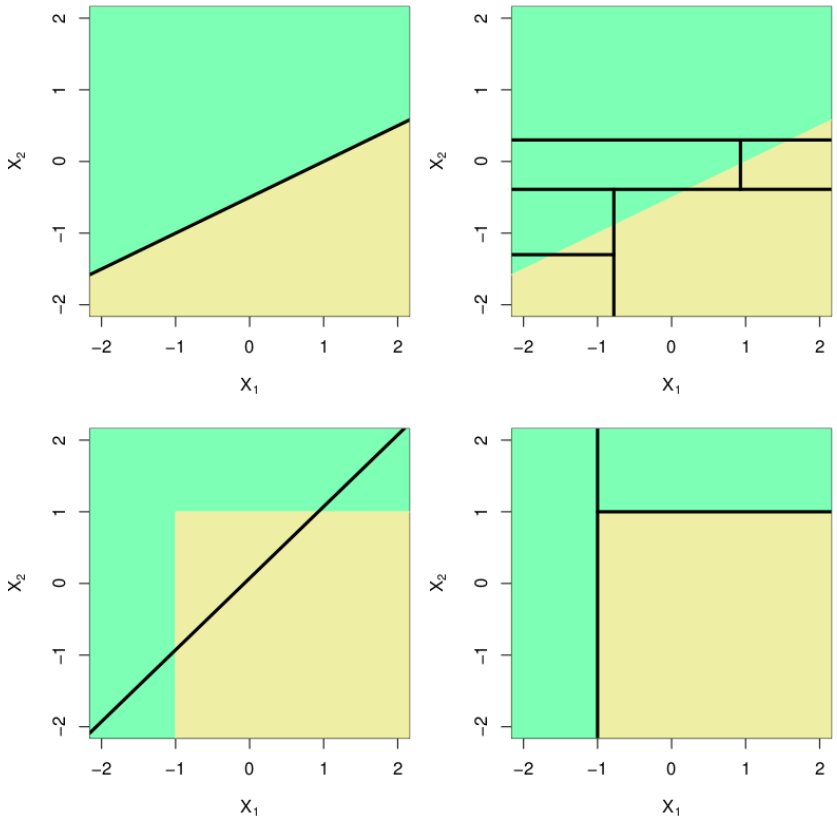
Questions:

Discuss the *bias-variance tradeoff of a regression tree* when increasing/decreasing the number of terminal nodes, i.e:

- What happens to the bias?
- What happens to the variance of a prediction if we reduce the tree size?

Trees versus linear models

- What if we have x_1 and x_2 and the true class boundary (two classes) is linear in x_1, x_2 space. How can we do that with our binary recursive splits?
- What about a rectangular boundary?



From trees to forests

Advantages (+)

- Trees automatically select variables.
- Tree-growing algorithms scale well to large n, growing a tree greedily.
- Trees can handle mixed features (continouos, categorical) seamlessly, and can deal with missing data.
- Small trees are easy to interpret and explain to people.
- Some believe that decision trees mirror human decision making.
- Trees can be displayed graphically.

Disadvantages (-)

- Large trees are not easy to interpret.
- Trees do not generally have good prediction performance (high variance).
- Trees are not very robust, a small change in the data may cause a large change in the final estimated tree.

What is next?

Several of the above listed disadvantages can be addressed by

- **Bagging**: grow many trees (from bootstrapped data) and average - to get rid of the non-robustness and high variance by averaging.
- **Random forests**: inject more randomness (and even less variance) by just allowing a random selection of predictors to be used for the splits at each node.
- **Boosting**: make one tree, then another based on the residuals from the previous, repeat. The final predictor is a weighted sum of these trees.

In addition:

- **Variable importance** - to see which variables make a difference (now that we have many trees).

Leo Breiman - the inventor of CART, bagging and random forests

See Wikipedia entry: https://en.wikipedia.org/wiki/Leo_Breiman

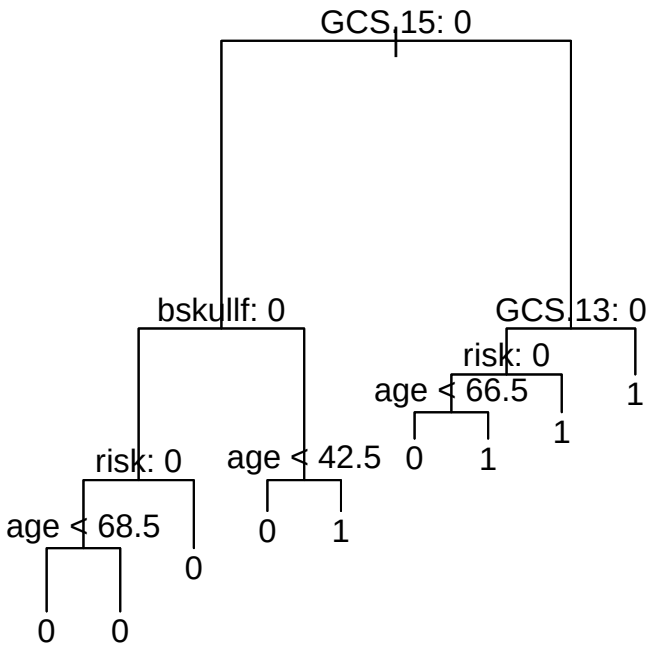
> *Breiman's work helped to bridge the gap between statistics and computer science, particularly in the field of machine learning. His most important contributions were his work on classification and regression trees and ensembles of trees fit to bootstrap samples. Bootstrap aggregation was given the name bagging by Breiman. Another of Breiman's ensemble approaches is the random forest.*

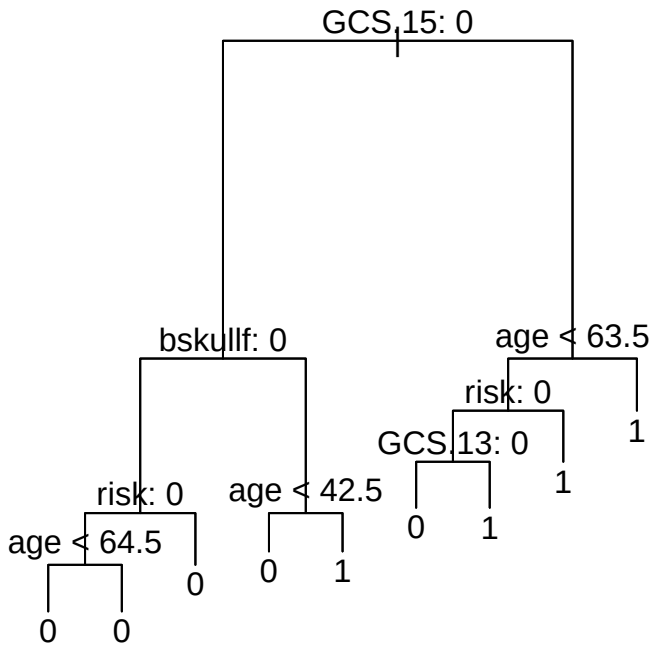
Bagging

- Decision trees often suffer from high variance. By this we mean that the trees are sensitive to small changes in the predictors: If we change the observation set, we may get a very different tree.
- Another way to understand "high variance" is that, if we split our training data into two parts and fit a tree on each, we might get rather different decision trees.
- To reduce the variance of decision trees we can apply *bootstrap aggregating* (*bagging*), invented by Leo Breiman in 1996 (Breiman 1996).

High variance in decision trees – Illustration

- Let's draw a new training set (`train2`) and use the deviance criterion to grow the tree.
- The two classification trees contructed from 850 different random subsets each:





- We get two rather different trees.

Recall: Variance for independent datasets

- Assume we have B *i.i.d.* observations of a random variable X each with the same mean and with variance σ^2. We calculate the mean $\bar{X} = \frac{1}{B}\sum_{b=1}^{B} X_b$. The variance of the mean is

$$\mathrm{Var}(\bar{X}) = \mathrm{Var}\Big(\frac{1}{B}\sum_{b=1}^{B} X_b\Big) = \frac{1}{B^2}\sum_{b=1}^{B}\mathrm{Var}(X_b) = \frac{\sigma^2}{B}.$$

  By averaging we get reduced variance. This is the basic idea!

- For decision trees, if we have B training sets, we could estimate $\hat{f}_1(x), \hat{f}_2(x), \ldots, \hat{f}_B(x)$ and average them as

$$\hat{f}_{avg}(x) = \frac{1}{B}\sum_{b=1}^{B}\hat{f}_b(x)\ .$$

However, we do not have many independent data set - so we use *bootstrapping* to construct B data sets.

Bagging = Bootstrap aggregating

- Bootsrapping: Draw *with replacement* n observations from our sample - and that is our first *bootstrap sample*.
- We repeat this B times and get B bootstrap samples - that we use as our B data sets.
- For each bootstrap sample $b = 1, \ldots, B$ we construct a decision tree, $\hat{f}^{*b}(x)$.
- For a *regression tree*, we take the average of all of the predictions and use this as the final result:

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

- For a *classification tree* we record the predicted class (for a given observation x) for each of the B trees and use the most occurring classification (*majority vote*) as the final prediction.

Out-of-bag error estimation

- Recall (module 5) that the probability that an observation is in the bootstrap sample is approximately $1 - e^{-1}$=0.63 ($\approx 2/3$).
- When an observation is left out of the bootstrap sample it is not used to build the tree, and we can use this observation as a part of a "test set" to measure the predictive performance and error of the fitted model, $\widehat{f}^{*b}(x)$.
- For B bootstrap samples, observation i will be outside the bootstrap sample in approximately $B/3$ of the fitted trees. Obtain a single prediction by averaging (regression problem) or taking the majority vote/mean probability (classification problem) of the $B/3$ predictions.

Terminology:

- The observations left out are referred to as the *out-of-bag* (OOB) observations.
- The measured error of the $B/3$ predictions is called the *out-of-bag error*.

Example

We can do bagging by using the function `randomForest()` in the `randomForest` library.

```
library(randomForest)
set.seed(1)
r.brain.bag <- randomForest(brain.injury ~ ., data = headInjury2, subset = train,
    mtry = 10, ntree = 500, importance = TRUE)
r.brain.bag$confusion
```

```
##     0  1 class.error
## 0 637 54   0.0781476
## 1  76 83   0.4779874
```

```
1 - sum(diag(r.brain.bag$confusion))/sum(r.brain.bag$confusion[1:2, 1:2])
```

```
## [1] 0.152941
```

The variable `mtry=10` because we want to consider all 10 predictors in each split of the tree. The variable `ntree=500` because we want to average over 500 trees.

Predictive performance of the bagged tree on unseen test data:

```
yhat.bag = predict(r.brain.bag, newdata = headInjury2[test, ])
misclass.bag = table(yhat.bag, headInjury2[test, ]$brain.injury)
print(misclass.bag)
```

```
##
## yhat.bag   0   1
##        0 346  39
##        1  34  52
```

```
1 - sum(diag(misclass.bag))/(sum(misclass.bag))
```

```
## [1] 0.154989
```

Note: The misclassification rate has increased slightly for the bagged tree (as compared to our previous full and pruned tree). **Typically, we would expect an improvement!**

Variable importance plots

- Drawback of bagging: It becomes difficult to interpret the results. Instead of having one tree, the resulting model consists of many trees (it is an *ensemble method*).
- *Variable importance plots* show *the relative importance of the predictors:* the predictors are sorted according to their importance, such that the top variables have a higher importance than the bottom variables.
- There are in general two types of variable importance plots:
  - variable importance based on decrease in node impurity.
  - variable importance based on randomization.

Variable importance based on node impurity

Importance relates to *total decrease in the node impurity, over splits for a predictor.*

Regression trees:

- The importance of each predictor is calculated using the RSS.
- The algorithm records the total amount that the RSS is decreased due to splits for each predictor (there may be many splits for one predictor for each tree).
- This decrease in RSS is then averaged over the B trees.

Classification trees:

- The importance of each predictor is calculated using the Gini index.
- The importance is the mean decrease (over all B trees) in the Gini index by splits of a predictor.

R-hint: `varImpPlot()` function (or `importance`) in `randomForest` with `type=2`.

Variable importance based on randomization

Variable importance based on randomization is calculated using the OOB sample.

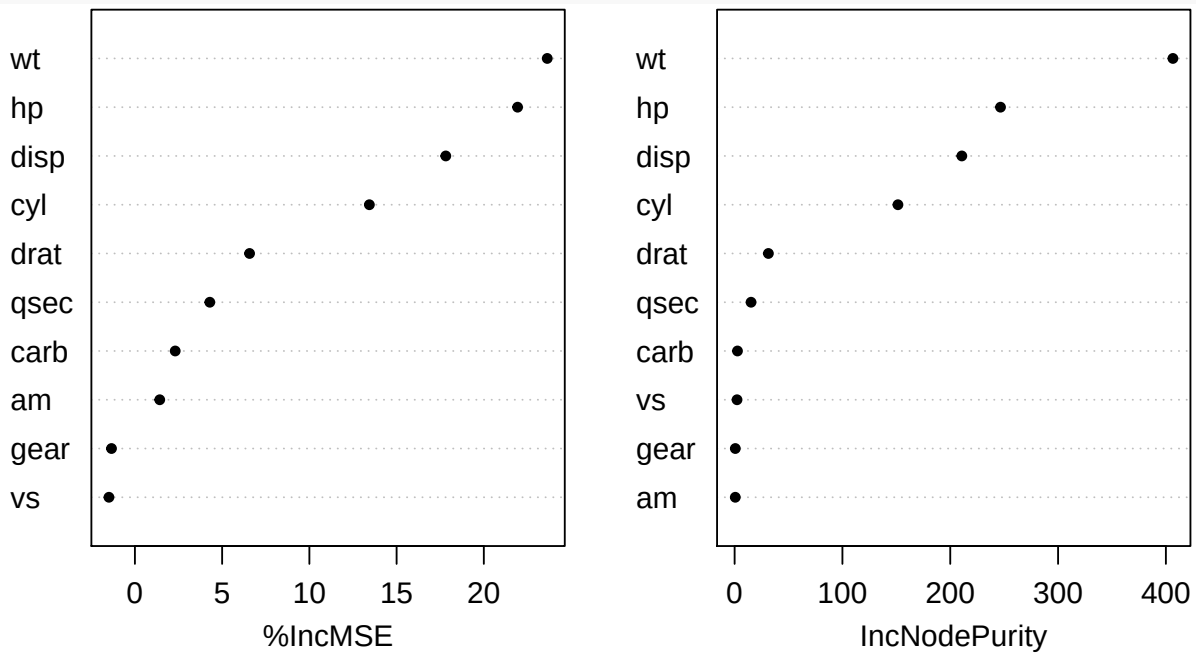
- Computations are carried out for one bootstrap sample at a time.
- Each time a tree is grown the OOB sample is used to test the predictive power of the tree.
- Then for one predictor at a time, repeat the following:
  - permute the OOB observations for the jth variable x_j and calculate the new OOB error.
  - If x_j is important, permuting its observations will decrease the predictive performance.
- The difference between the two is averaged over all trees (and normalized by the standard deviation of the differences).

R-hint: `varImpPlot()` (or `importance()`) function with `type=1`.

Example 1: Auto data (regression tree)

The data relates fuel consumption (`mpg`) to 10 aspects of automobile desing and performance.

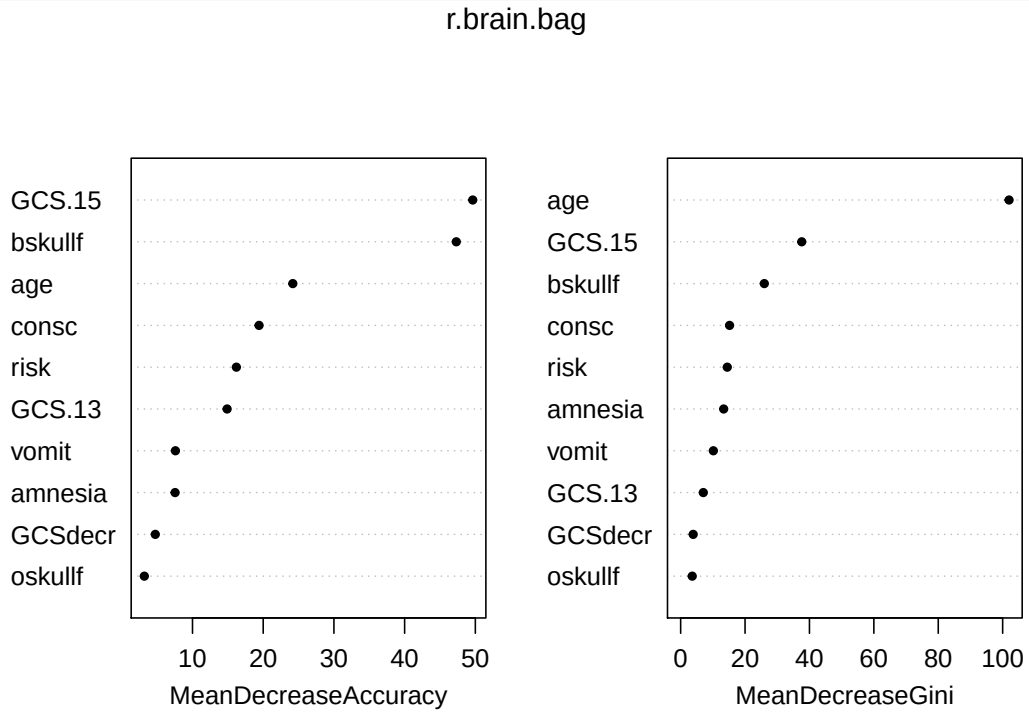
```
data(mtcars)
mtcars.rf <- randomForest(mpg ~ ., data = mtcars, ntree = 1000, keep.forest = FALSE,
    mtry = 10, importance = TRUE)
varImpPlot(mtcars.rf, pch = 20, main = "")
```



(Randomization: left, node purity: right)

Example 2: Head injury (classification tree)

```
varImpPlot(r.brain.bag, pch = 20)
```



(Randomization: left, node purity: right)

Random Forests

- If there is a strong predictor in the dataset, the decision trees produced by each of the bootstrap samples in the bagging algorithm becomes very similar: Most of the trees will use the same strong predictor in the top split.

- We have seen this for our example trees for the minor head injury example, the predictor `GCS.15` was chosen in the top split every time. This is probably the case for a large amount of the bagged trees as well.

- Not optimal, as we get B trees that are highly correlated. $\rightarrow$ No large reduction in variance by averaging $\widehat{f}^{*b}(x)$ when the correlation between the trees is high.

The effect of correlation on the variance of the mean

- The variance of the average of B observations of *i.i.d.* random variables X, each with variance σ^2 is $\frac{\sigma^2}{B}$.
- But if we have B *i.i.d.* observations of a random variable X with a positive correlation ρ such that $\mathrm{Cov}(X_i, X_j) = \rho\sigma^2, \quad i \neq j$, then

$$\mathrm{Var}(\bar{X}) = \mathrm{Var}\Big(\frac{1}{B}\sum_{i=1}^{B} X_i\Big) =$$

Check: $\rho = 0$ and $\rho = 1$?

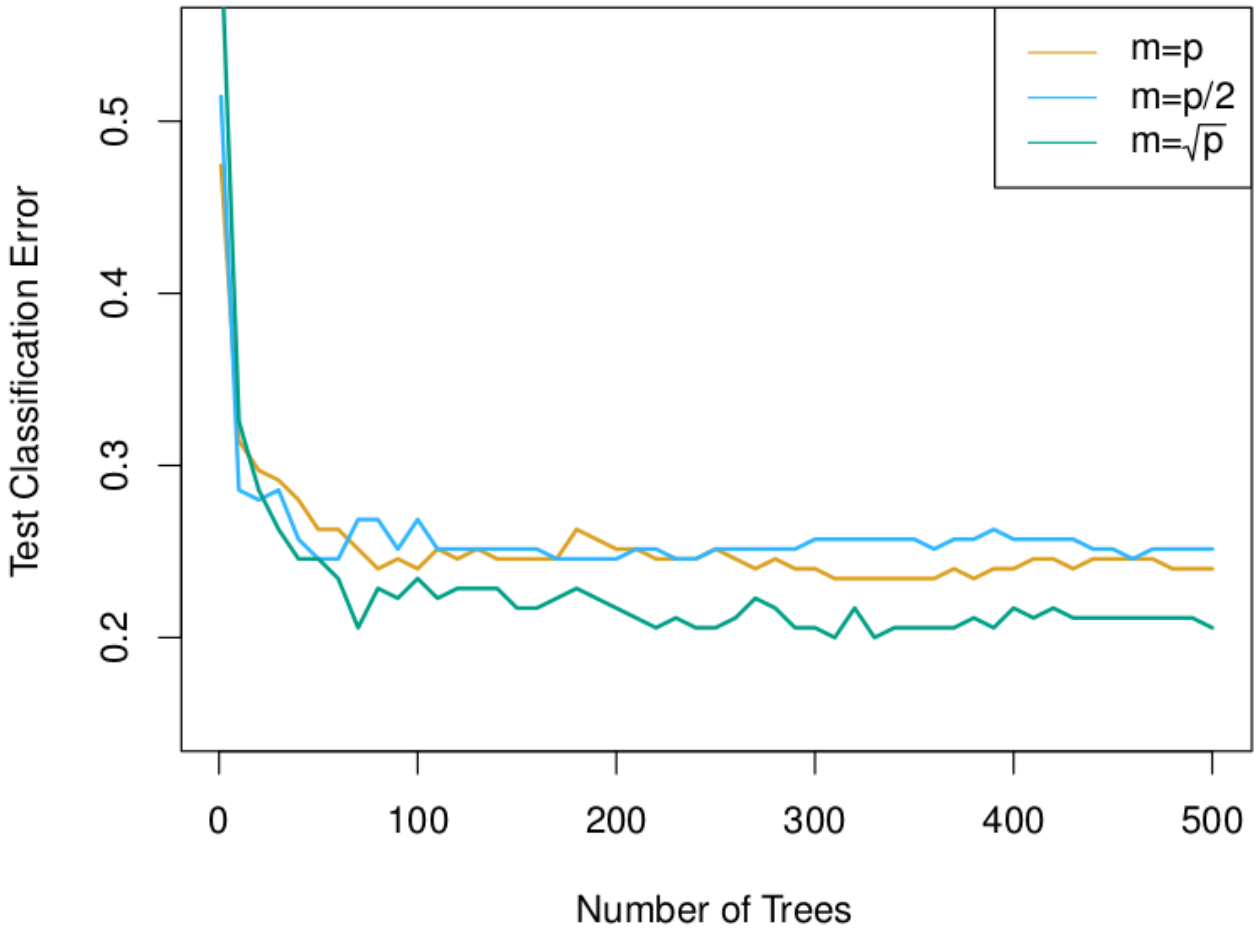
- *Random forests* provide an improvement over bagged trees by a small tweak that *decorrelates the trees*. This reduces the variance when we average the trees.
- As in bagging, we build a number of decision trees on bootstrapped training samples.
- But each time a split in a tree is considered, a *random selection* of m predictors is chosen as split candidates from the full set of p predictors. The split is allowed to use only one of those m predictors.
- A fresh selection of m predictors is taken at each split, typically
  - $m \approx \sqrt{p}$ (classification),
  - $m = p/3$ (regression).

The general idea is that for very correlated predictors m is chosen to be small.

How many trees?

- The number of trees, B, is not a tuning parameter, and the best is to choose it large enough (as large as "necessary"). An increase in B will not lead to overfitting.
- Increasing B will not change the goodness of fit measure.
- To find out which number B is sufficient, we do *not* need to run cross-validation, but can again use the OOB error (works best for bagging and random forests).

ISLR Figure 8.10, gene expression data set with 15 classes and 500 predictors:



Example

We decorrelate the brain injury classification trees by using the `randomForest()` function again, but this time we set `mtry=3` (instead for `mtry=10`). This means that the algorithm only considers three of the predictors in each split. We choose 3 because we have 10 predictors in total and $\sqrt{10} \approx 3$.

```
set.seed(1)

r.brain.rf = randomForest(brain.injury ~ ., data = headInjury2, subset = train,
    mtry = 3, ntree = 500, importance = TRUE)
```

We check the predictive performance as before, using the test set:

```
yhat.rf = predict(r.brain.rf, newdata = headInjury2[test, ])

misclass.rf = table(yhat.rf, headInjury2[test, ]$brain.injury)
print(misclass.rf)
```

```
##
## yhat.rf   0   1
##       0 363  47
##       1  17  44
```

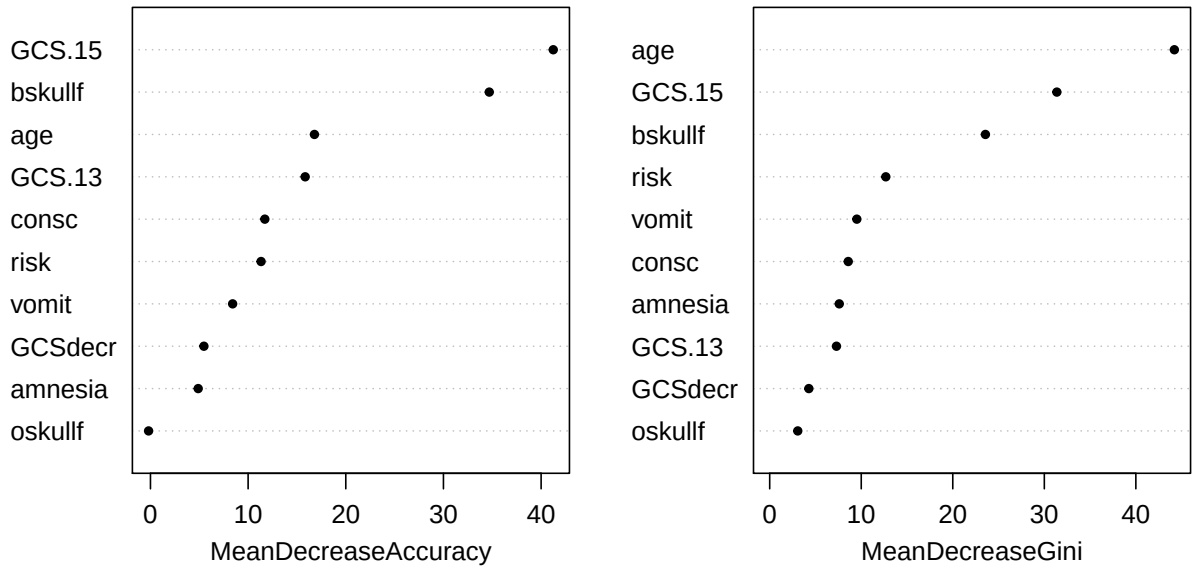
```
1 - sum(diag(misclass.rf))/(sum(misclass.rf))
```

```
## [1] 0.1358811
```

The misclassification rate is slightly decreased compared to the bagged tree (and to the pruned tree).

By using the `varImpPlot()` function we can study the importance of each predictor.

```
varImpPlot(r.brain.rf, pch = 20, main = "")
```



As expected `GCS.15` (GCS=15 at 2h) is a strong predictor along with `bskullf` (basal skull facture) and `age`. This means that most of the trees will have these predictors in the top split.

Boosting

Boosting is an alternative approach for improving the predictions resulting from a decision tree. We will only consider the description of boosting regression trees (and not classification trees) in this course.

In boosting the trees are grown *sequentially* so that each tree is grown using information from the previous tree.

- First build a decision tree with d splits (and $d+1$ terminal notes).
- Next, improve the model in areas where the model didn't perform well. This is done by fitting a decision tree to the *residuals of the model*. This procedure is called *learning slowly*.
- The first decision tree is then updated based on the residual tree, but with a weight.
- The procedure is repeated until some stopping criterion is reached. Each of the trees can be very small, with just a few terminal nodes (or just one split).

Algorithm 8.2: Boosting for regression trees

1. Set $\hat{f}(x) = 0$ and $r_i = y_i$ for all i in the training set.
2. For $b = 1, 2, ..., B$, repeat:
   a) Fit a tree $\hat{f}^b$ with d splits ($d + 1$ terminal nodes) to the training data.
   b) Update $\hat{f}$ by adding in a shrunken version of the new tree:

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

   c) Update the residuals,

$$r_i \leftarrow r_i - \lambda \hat{f}^b(x_i).$$

3. The boosted model is $\hat{f}(x) = \sum_{b=1}^{B} \lambda \hat{f}^b(x)$.

Boosting has three tuning parameters which need to be set (B, λ, d), and can be found using cross-validation.

Tuning parameters

- **Number of trees** B. Could be chosen using cross-validation. A too small value of B would imply that much information is unused (remember that boosting is a slow learner), whereas a too large value of B may lead to overfitting.
- **Shrinkage parameter** λ. Controls the rate at which boosting learns. λ scales the new information from the b-th tree, when added to the existing tree $\widehat{f}$. Choosing a small value for λ ensures that the algorithm learns slowly, but will require a larger B. Typical values of λ is 0.1 or 0.01.
- **Interaction depth** d: The number of splits in each tree. This parameter controls the complexity of the boosted tree ensemble (the level of interaction between variables that we may estimate). By choosing $d = 1$ a tree stump will be fitted at each step and this gives an additive model.

Short quizz poll

www.menti.com, Code 62 76 78 5

Example: Boston data set

(ISLR book, Sections 8.3.2 to 8.3.4.)

Remember the data set: The aim is to predict the median value of owner-occupied homes (in 1000$)

We first run through trees, bagging and random forests - before arriving at boosting.

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

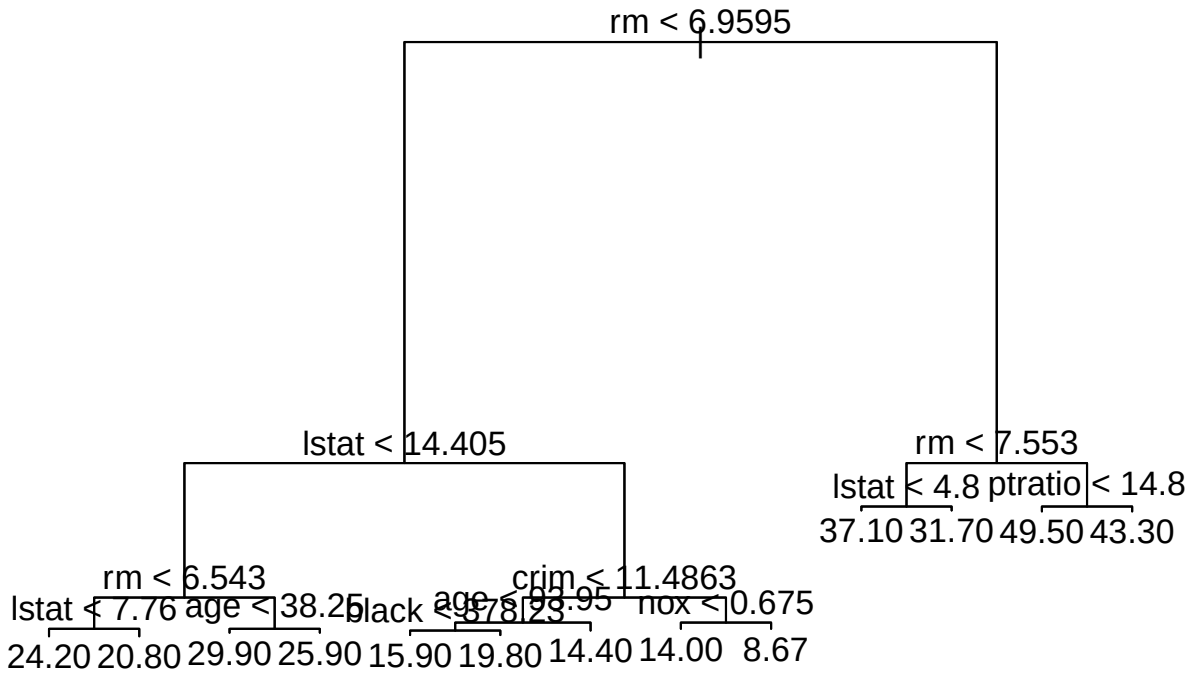
```
##      crim zn indus chas   nox    rm  age    dis rad tax ptratio  black lstat
## 1 0.00632 18  2.31    0 0.538 6.575 65.2 4.0900   1 296    15.3 396.90  4.98
## 2 0.02731  0  7.07    0 0.469 6.421 78.9 4.9671   2 242    17.8 396.90  9.14
## 3 0.02729  0  7.07    0 0.469 7.185 61.1 4.9671   2 242    17.8 392.83  4.03
## 4 0.03237  0  2.18    0 0.458 6.998 45.8 6.0622   3 222    18.7 394.63  2.94
## 5 0.06905  0  2.18    0 0.458 7.147 54.2 6.0622   3 222    18.7 396.90  5.33
## 6 0.02985  0  2.18    0 0.458 6.430 58.7 6.0622   3 222    18.7 394.12  5.21
##   medv
## 1 24.0
## 2 21.6
## 3 34.7
## 4 33.4
## 5 36.2
## 6 28.7
```

Regression tree

```
tree.boston = tree(medv ~ ., Boston, subset = train, control = tree.control(nrow(Boston),
    mindev = 0.005))
summary(tree.boston)
```

```
##
## Regression tree:
## tree(formula = medv ~ ., data = Boston, subset = train, control = tree.control(nrow(Bost
##     mindev = 0.005))
## Variables actually used in tree construction:
## [1] "rm"      "lstat"   "age"     "crim"    "black"   "nox"     "ptratio"
## Number of terminal nodes:  13
## Residual mean deviance:  7.35 = 1760 / 240
## Distribution of residuals:
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  -8.140  -1.440  -0.162   0.000   1.460  12.900
```

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

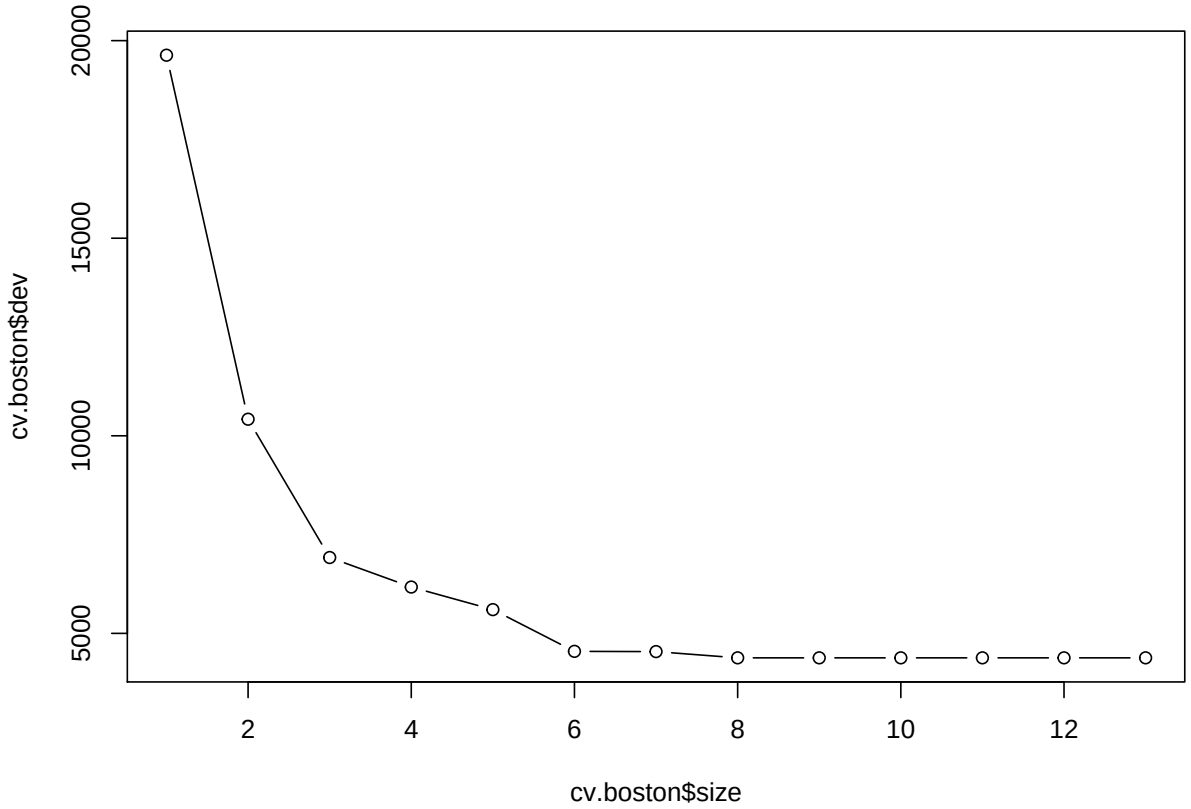


Remember:

- The `tree()` function has a built-in default stopping criterion.
- You can change this with the `control` option, for example by setting `control = tree.control(mincut = 2, minsize = 4, mindev = 0.001)`. Here we used `mindev=0.005`.

Need to prune?

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

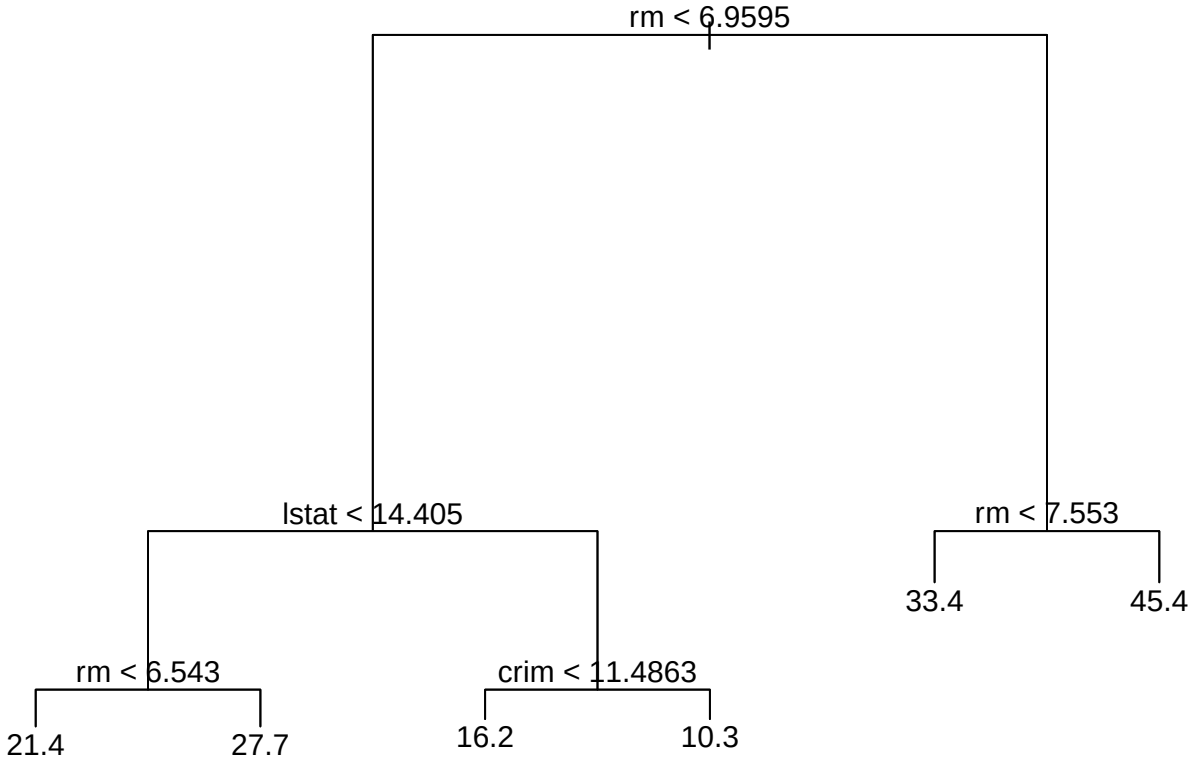


It looks like a tree with 6 leaves would work well.

Pruning

So we are pruning to a 6-node tree here:

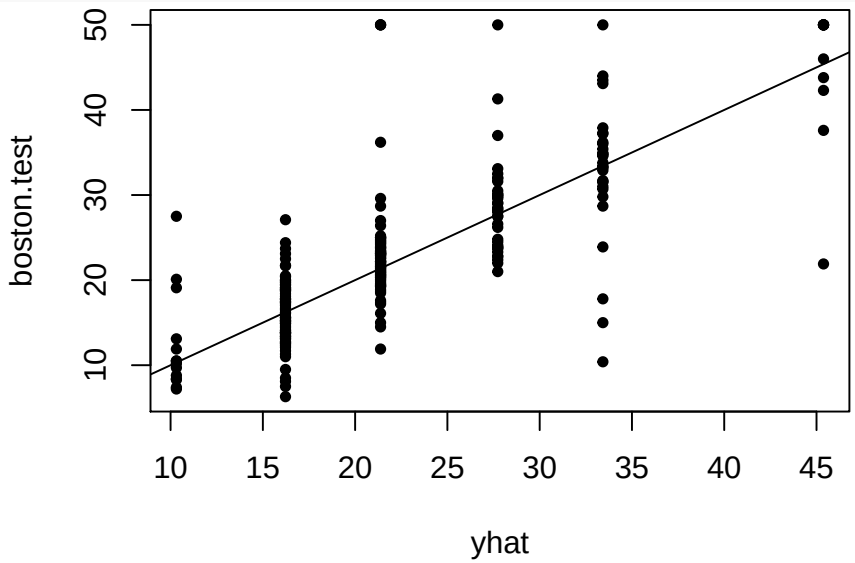
```
prune.boston = prune.tree(tree.boston, best = 6)
plot(prune.boston)
text(prune.boston, pretty = 0)
```



Test error for full tree

We calculate the test error for the pruned tree:

```
yhat = predict(prune.boston, newdata = Boston[-train, ])
boston.test = Boston[-train, "medv"]
plot(yhat, boston.test, pch = 20)
abline(0, 1)
```




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

```
## [1] 35.1644
```

Bagging

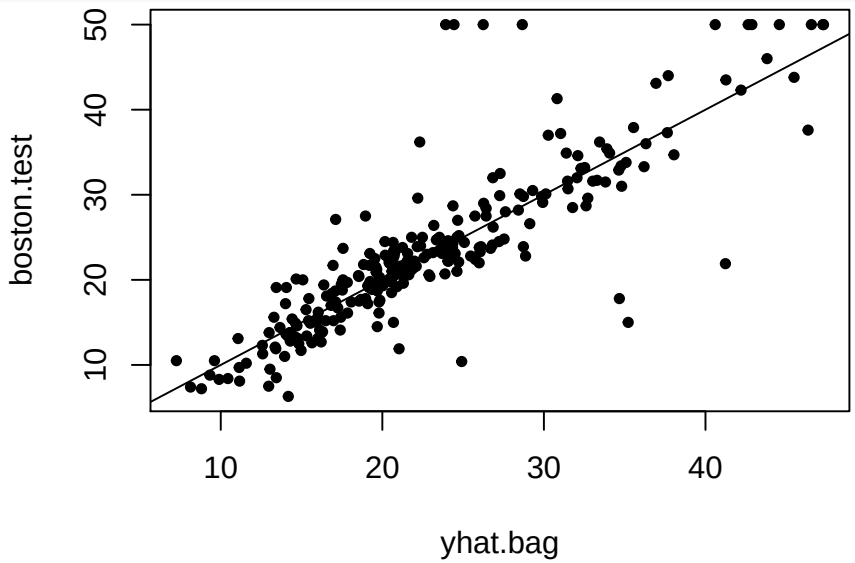
Remember: For bagging you can use the `randomForest()` function, but include all variables (here `mtry=13`).

```
library(randomForest)
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,      subs
##                Type of random forest: regression
##                      Number of trees: 500
## No. of variables tried at each split: 13
##
##           Mean of squared residuals: 11.396
##                     % Var explained: 85.17
```

Test error for bagged tree

```
yhat.bag = predict(bag.boston, newdata = Boston[-train, ])
plot(yhat.bag, boston.test, pch = 20)
abline(0, 1)
```




```
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] 23.6672
```

Random forest

Let's go from bagging to a random forest[4], using 6 randomly selected predictors for each tree:

```
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] 19.6202
```

It's interesting to see how the prediction error further decreased with respect to simple bagging.

[4] n.b., why are we now speaking of a forest and no longer of a tree?

Variable importance

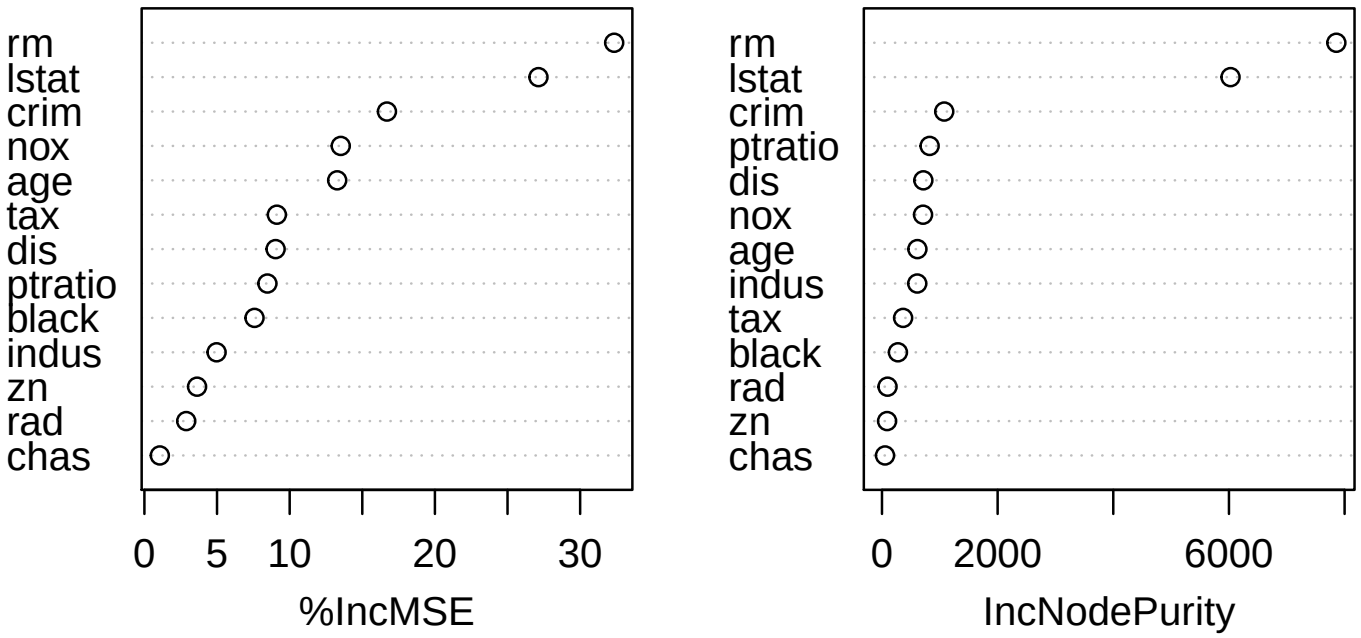
```
importance(rf.boston)
```

```
##           %IncMSE IncNodePurity
## crim    16.69702     1076.0879
## zn       3.62578       88.3534
## indus    4.96862      609.5336
## chas     1.06143       52.2179
## nox     13.51818      709.8734
## rm      32.34330     7857.6545
## age     13.27250      612.2142
## dis      9.03248      714.9467
## rad      2.87843       95.8060
## tax      9.11880      364.9248
## ptratio  8.46706      823.9334
## black    7.57948      275.6227
## lstat   27.12982     6027.6374
```

Interpretation?

And the variable importance plots

```
varImpPlot(rf.boston, main = "")
```



To understand what this means, please check again the meaning of the variables by typing `?Boston`.

Boosting

And finally, we are boosing the Boston trees! We boost with 5000 trees and allow the interaction depth (number of splits per tree) to be of degree 4:

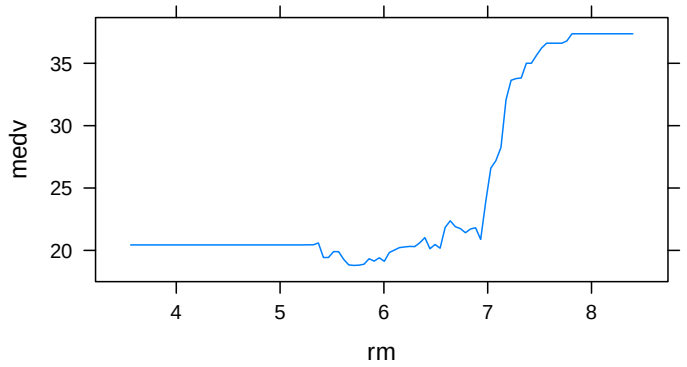
```
library(gbm)
set.seed(1)
boost.boston = gbm(medv ~ ., data = Boston[train, ], distribution = "gaussian",
    n.trees = 5000, interaction.depth = 4)
summary(boost.boston, plotit = FALSE)
```

```
##             var   rel.inf
## rm           rm 43.991933
## lstat     lstat 33.121694
## crim       crim  4.260417
## dis         dis  4.011109
## nox         nox  3.435302
## black     black  2.826755
## age         age  2.611394
## ptratio ptratio  2.540303
## tax         tax  1.456565
## indus     indus  0.800874
## rad         rad  0.654640
## zn           zn  0.144615
## chas       chas  0.144399
```

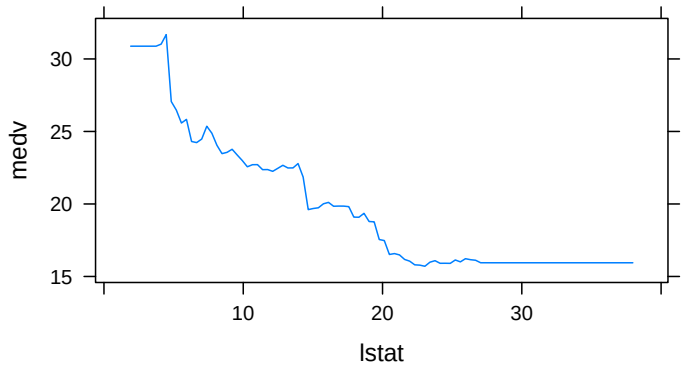
Partial dependency plots - integrating out other variables

`rm` (number of rooms) and `lstat` (% of lower status population) are the most important predictors. Partial dependency plots show the effect of individual predictors, integrated over the other predictors see Hastie, Tibshirani, and Friedman (2009), Section 10.13.2.

```
plot(boost.boston, i = "rm", ylab = "medv")
```




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



Prediction on test set

- Calculate the MSE on the test set, first for the model with $\lambda = 0.001$ (default), then with $\lambda = 0.2$.
- We could have done cross-validation to find the best λ over a grid, but it seems not to make a big difference.

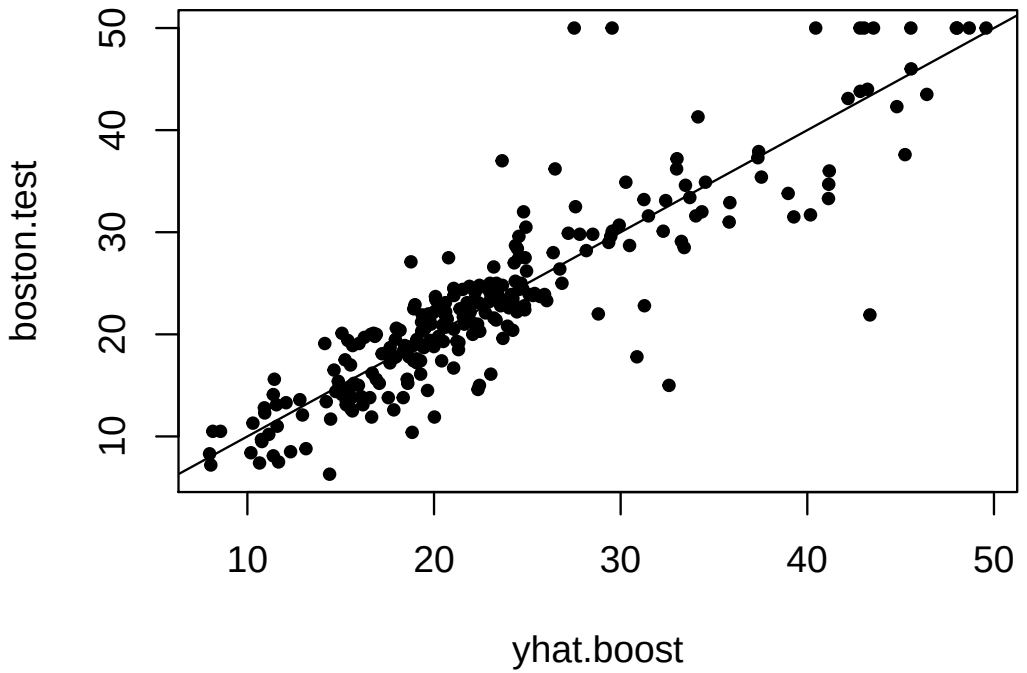
```
yhat.boost = predict(boost.boston, newdata = Boston[-train, ], n.trees = 5000)
mean((yhat.boost - boston.test)^2)
```

```
## [1] 18.8471
```

```
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] 18.3345
```

```
plot(yhat.boost, boston.test, pch = 20)
abline(0, 1)
```



Further reading

- Videoes on YouTube by the authors of ISL, Chapter 8, and corresponding slides.
- Solutions to exercises in the book, chapter 8

References

Breiman, Leo. 1996. "Bagging Predictors." *Machine Learning* 24: 123–40.

Hastie, Trevor, Robert Tibshirani, and Jerome Friedman. 2009. *The Elements of Statistical Learning*. Vol. 2. Springer series in statistics New York.

James, G., D. Witten, T. Hastie, and R. Tibshirani. 2013. *An Introduction to Statistical Learning with Applications in R*. New York: Springer.

Ripley, Brian. 2019. *Tree: Classification and Regression Trees*. https://CRAN.R-project.org/package=tree.

Ripley, Brian D. 1996. *Pattern Recognicion and Neural Networks*. Cambridge University Press.