Statistical learning for data science: Handout 5

Stefan Zohren

5 May 2016

General Information

Course Content

This course is the second part of a course on "Statistical learning for data science". The topics we cover are:

- Day 1: Introduction, Review of bias-variance-tradeoff and cross-validation; Decision trees;
- Day 2: Bagging, boosting and random forests;
- Day 3: Support vector machines;
- Day 4: Neural networks.

The previous course "Statistical learning for data science: Basic Techniques using R" covered: Regression analysis: linear and polynomial, bias-variance-tradeoff; Classification: logistic regression, discriminant analysis; Regularisation (ridge regression and lasso) and cross-validation; Unsupervised learning methods: principle component analysis (PCA) and clustering (k-means and hierarchical). There is also a one-day topic course on "Data Wrangling using R".

Both the first part of the Statistical Learning course, as well as the Data Wrangling course require no prior knowledge in R. This second course, while reviewing some basic concepts of statistical learning from the first course, assumes basic familiarity with R.

The main text book for this course is:

- An Introduction to Statistical Learning by James, Witten, Hastie and Tibshirani (ISLR)

Occasionally we also refer to selected sections of the more advanced textbook:

- The Elements of Statistical Learning by Hastie, Tibshirani and Friedman (ESL)

For a comprehensive textbook on machine learning (going far beyond the content of this course), we recommend:

- Machine Learning: A Probabilistic Perspective, by Murphy

Statistical Software R: Installation and Packages

You can download R from The Comprehensive R Archive Network cran.r-project.org and RStudio from rstudio.com. Both are free software.

For many people RStudio is the preferred GUI for R. It has a very similar layout to Matlab. Alternatively, you can also use the R application which comes with the official R distribution. In this case, you might want to use it together with a text editor to copy over commands. You can also use R using the command line which will be necessary for example when using it on ARC. You can write R scripts in any editor; Emacs is a good command line editor with syntax highlighting for R. You can find a help document on how to use R on

the ARC cluster at Oxford here. Whenever using R on the command line you run it via `RScript file.R <optional arguments>`.

In many cases you want to install packages for advanced analysis. Standard functions are part of the `base` package which is automatically loaded when starting R. The quality of packages varies widely and you might want to investigate a bit before settling on a given package. A useful reference is CRAN and in particular CRAN Task View which has commented list of packages for different applications fields, i.e. Bayesian analysis.

Packages can be installed from the Package installer in the application or using the command

```
install.packages("package_name")
```

At the beginning of your session or script you then load the package using

```
library("package_name")
```

A review of bias-variance-tradeoff and cross-validation

The bias-variance-tradeoff

We start this session by revising the discussion on the bias-variance-tradeoff from the first lecture of the previous course. We repeat the same example as used previously. This section also serves as a refresher on R syntax, especially for those who have not attended the first course.

Let us revisit the toy example of data used in the first lecture of the previous course. We considered a model $y_i = f(x_i) + \epsilon_i$, where $f(x)$ is the true model and ϵ_i is the noise term of observation i = 1,...,n with n being the total number of observations. For concreteness we chose

$$f(x) = 2 + 0.2(x - 5)^2$$

and $\epsilon_i \sim \mathcal{N}(0, 1)$.

In `R` we can sample (simulate) n=10 observations, as follows.

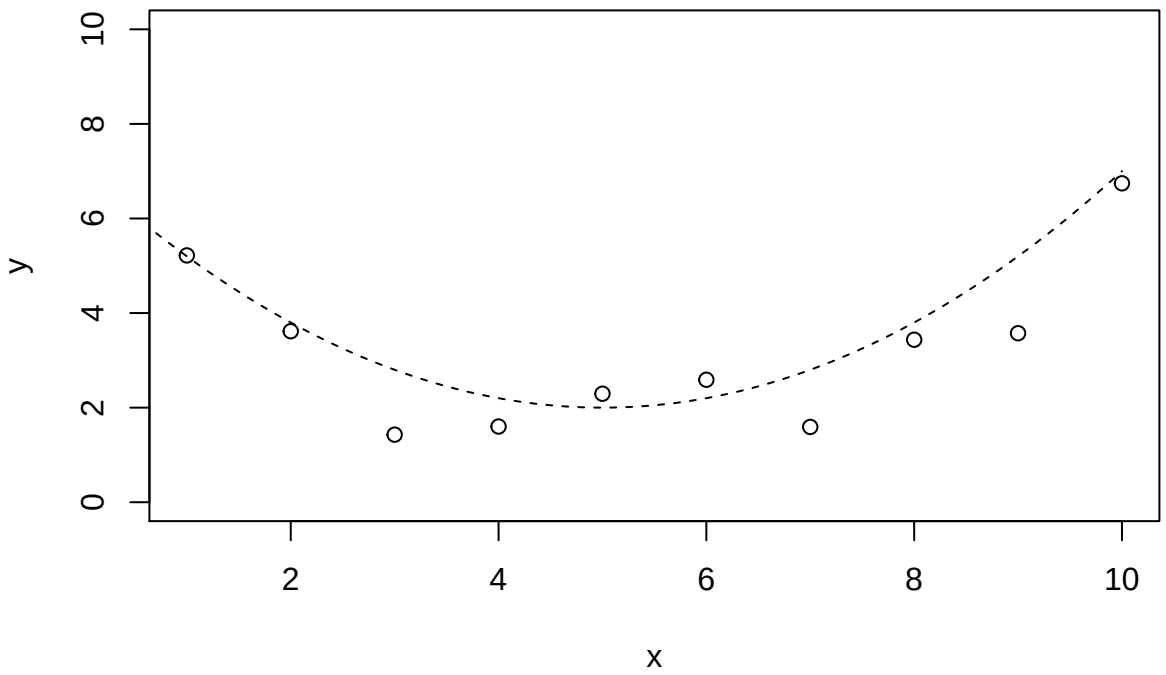
```
# Define sequence x = 1,2,...,10
x <- 1:10
# Define function f(x)
f <- function(x) { 2 + 0.2*(x-5)^2 }
# Set seed of random number generator:
set.seed(10)
# Compute the value of the function for each x and add noise term. This is vectorised:
y <- f(x)+rnorm(10)
```

To refresh some important `R` commands: `n:m` generates the sequence of integers from `n` to `m`. We define a function using `function(argument)`. There is no need for a `return` statement in the body of the function as `R` automatically returns the last command in the body of the function.

Note that `set.seed` initialises the seed of the (pseudo) random number generator. `rnorm(10)` generates a vector of 10 (pseudo) random numbers from the Gaussian distribution with mean zero and variance 1. See `?rnorm` for more information.

Data can be plotted using the `plot` function. Additional information can be added to the plot using e.g. `points` and `lines`:

```r
# Plot the points:
plot(y ~ x, ylim=c(0,10))
# Define a fine grid for plotting a 'smooth' line:
xGrid <- seq(0,10,by=0.1)
# Add the line to the plot:
lines(xGrid,f(xGrid) , lty=2)
```

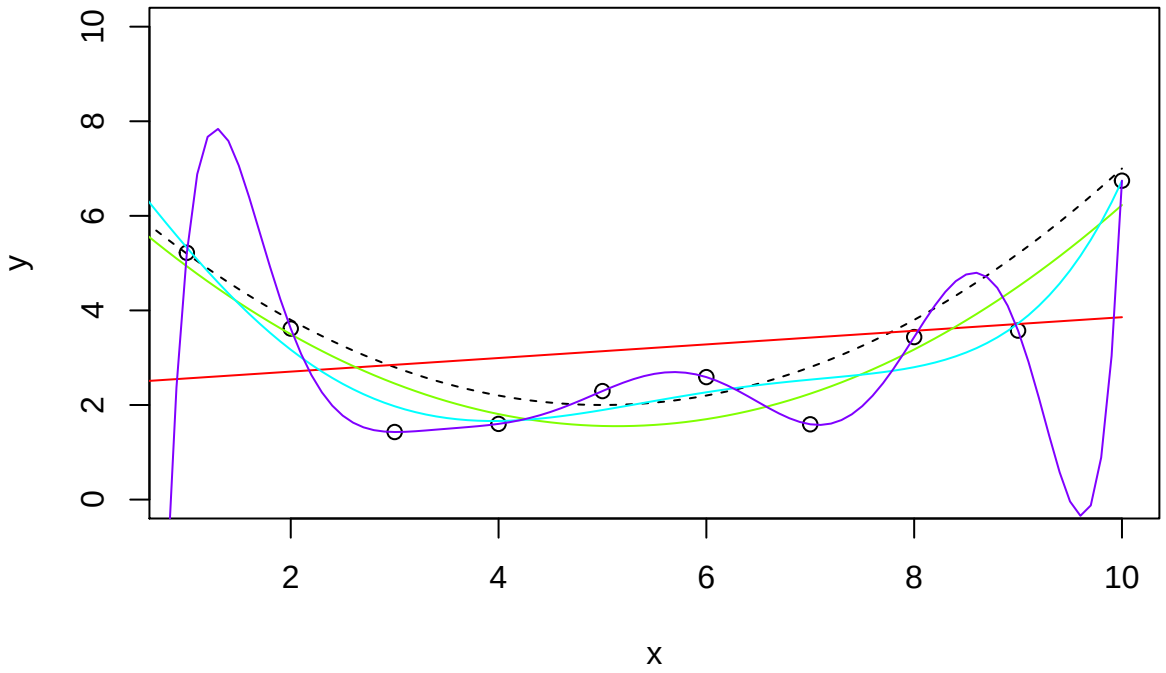


(`lty=2` is the second line type, which is dashed, see `?lines` or `?plot` for more information.)

To illustrate the problem of **overfitting** we performed a little experiment in a previous lecture, where we fitted polynomials

$$y = a + a_1 x + a_2 x^2 + ... + a_p x^p$$

of increasing degree p to the data. In particular, the following plot shows the fitted polynomials for $p = 1, 2, 5, 10$:



Exercise: (i) Execute the above commands to generate the data for the toy model, as well as to plot the data points together with the underlying true model function. (ii) Recall that you can fit a linear model using the command `lm` (see `?lm` for help). Reproduce the linear fit of the data and add the line to your figure. (iii) If you attended the first part of the statistical learning for data science course you should be able to reproduce the entire figure.

Let us denote the fitted response variable by $\hat{\mathbf{y}}$. A natural measure for the error in regression models is the so-called **root mean squared error (RMSE)** defined as:

$$\text{RMSE} = \left(\frac{1}{n} \sum_{i=1}^{n} \epsilon_i^2 \right)^{\frac{1}{2}} = \sqrt{\text{mean}((\hat{\mathbf{y}} - \mathbf{y})^2)}$$

Intuitively, we obtain the error by summing the squared distances of the vertical lines connecting each data point with the fitted line and taking the square root, e.g. for the linear model (p=1) this would be all the vertical lines connecting the data points with the red line.

When inspecting the above figure we see that when increasing the order of the polynomial p the RMSE becomes smaller. In fact, in the case of p=10 (violet curve) all data points lie on the fitted curve and thus the RMSE = 0, we say that the model has low **bias**. The higher the p the more **flexible** we say the model is. In general it can be shown that the (training) RMSE always decreases as we increase the flexibility.

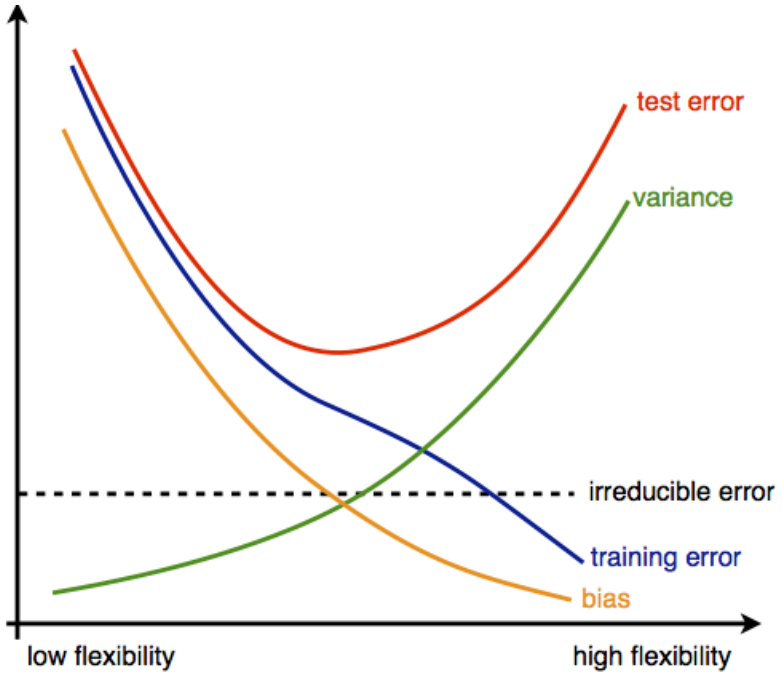
The above shows a problem in using the RMSE as we naively did. By inspecting the above figures by eye, we would agree that the models with p=2,5 are much better fits to the data than the model with p=10, however, the RMSE is smallest for the latter. The problem with the p=10 model becomes apparent if we were to collect new data (which corresponds to repeating the above experiment with a new set of random variables). In this case the old fit with p=10 would have a very large RMSE. If we were to re-fit the model the estimated curve for p=10 would look very different. We say that the model has high **variance**.

A proper way to measure the error of a model would thus be to use a new data set when evaluating the RMSE. We call this new data set the **test data**, while the data set we use to train the model on is called the **training data**. Instead of using the RMSE evaluated on the training set, i.e. the **training error** $\text{RMSE}_{\text{train}}$ and we should use the RMSE evaluated on the test set, also called the **test error** $\text{RMSE}_{\text{test}}$.

We have already seen that $\text{RMSE}_{\text{train}}$ will always decrease when we increase the flexibility (increase p). However, the above example shows that $\text{RMSE}_{\text{test}}$ first decreases when we move up in flexibility from the

linear (p=1) to the quadratic (p=2) model, but then increases once we move to higher p.

The general behaviour can be sketched as follows:



The training error always decreases as we increase the flexibility of the model. However, the test error first decreases and then increases. More precisely, the test error is a combination of three terms

$$\text{RMSE}_{test} = \text{Variance} + \text{Bias}^2 + \text{irreducible error}$$

For low flexibility the bias term dominates, while for large flexibility the variance term dominates. However, there exists an optimal level of flexibility which minimises the test error. We would like to find this level by training the model on the test data while varying the flexibility and evaluating the RMSE on the test data.

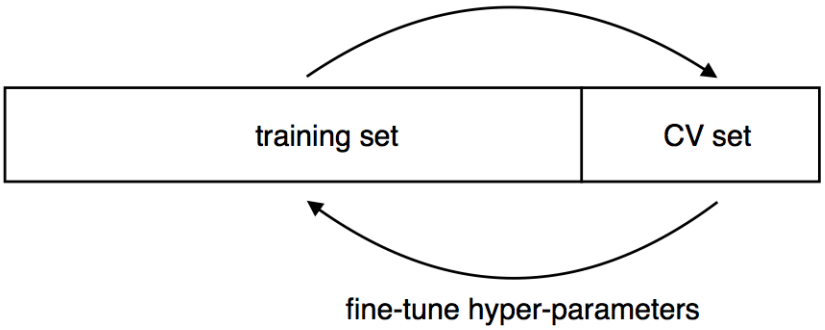
The above relation is a key concept in statistical learning and is often also referred to as the **bias-variance-tradeoff**.

Training, cross-validation and test sets

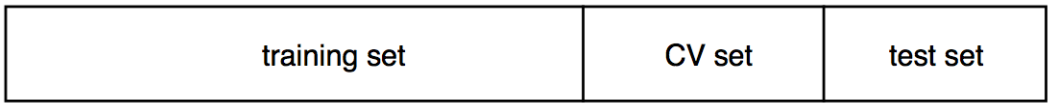
In the previous section we saw the need to have both a training set as well as a test set. In practice, we would not obtain new data to test our model, but rather hold-out a part of the data. The test data set is therefore often also called the **hold-out set**. Popular splits are 70/30 % or 80/20 % for training and hold-out set.

When discussing regularisation in the previous module, we have seen how one can introduce a single parameter, such as the shrinkage, which can be used to tune the flexibility of the model. Such a parameter is also called a **hyper-parameter** of the model. In fitting the model for a range of values of the hyper-parameter and then evaluating the error on the hold-out set we can **fine-tune** the flexibility to the optimal level corresponding to the minimum of the test error (red curve in above figure). This process is also called **cross-validation (CV)** and the error is also often referred to as the **cross-validation error**.

The process of cross-validation is illustrated in the following figure:



In simple models, such as lasso, there might be only one hyper-parameter, however, in more complex models, such as neural networks, there could be many hyper-parameter and fine-tuning those parameters might involve an extensive grid search over many values for those parameters. One problem in doing this is that each time we look at the hold-out set to tune the parameters, we are effectively training on it. In this case, the error evaluated on the hold-out is under-estimating the true test error. To circumvent this and to obtain an unbiased estimate for the test error, it is thus advisable to hold-out yet another part of the training data to do cross-validation on and to leave the test set untouched during this process. The situation is then as depicted in the following figure:



After having fine-tuned the hyper-parameters on the CV set, we can re-train the model with the optimal hyper-paramters on the combined training and CV set.

Let us now see how to do this in R. We start by recalling how to import data, in this case the California Housing data already used in the previous module:

```
HousingData <- read.csv("California-Housing.txt")
```

Make sure that the data file is in your working directory. To change your working directory in RStudio go to "Session" and then "Set Working Directory".

It is always a good idea to inspect the data using `head()` or `str()`. In RStudio you can also open a spreadsheet editor using `View()`.

```
head(HousingData)
```

```
##   MedianHouseValue MedianIncome HousingMedianAge AverageNoRooms
## 1            4.526       8.3252               41       6.984127
## 2            3.585       8.3014               21       6.238137
## 3            3.521       7.2574               52       8.288136
## 4            3.413       5.6431               52       5.817352
## 5            3.422       3.8462               52       6.281853
## 6            2.697       4.0368               52       4.761658
##   AverageNoBedrooms Population AverageOccupancy Latitude Longitude
## 1         1.0238095        322         2.555556    37.88   -122.23
## 2         0.9718805       2401         2.109842    37.86   -122.22
## 3         1.0734463        496         2.802260    37.85   -122.24
## 4         1.0730594        558         2.547945    37.85   -122.25
## 5         1.0810811        565         2.181467    37.85   -122.25
## 6         1.1036269        413         2.139896    37.85   -122.25
```

We now split the data into a training and a test set using a 80/20% split:

```
set.seed(10)
n <- nrow(HousingData)
trainIndices <- sample(1:n, n*0.8)
HousingData.train <- HousingData[trainIndices,]
HousingData.test <- HousingData[-trainIndices,]
```

We can now split off a cross-validation set from the training set using another split:

```
n <- nrow(HousingData.train)
trainIndices <- sample(1:n, n*0.8)
HousingData.train.tr <- HousingData[trainIndices,]
HousingData.train.cv <- HousingData[-trainIndices,]
```

Note that there are several models in R which do CV automatically for you. In those cases we can pass `HousingData.train` directly to the function which will split it off further.

Exercise: Run the above commands and inspect the size of each data set.

The right way of doing cross-validation

As we have seen above, each time we peak at the test data and adapt our model, we are effectively 'learning' from it. In the ideal case we should only look at the test data once at the end of our analysis when evaluating the test error.

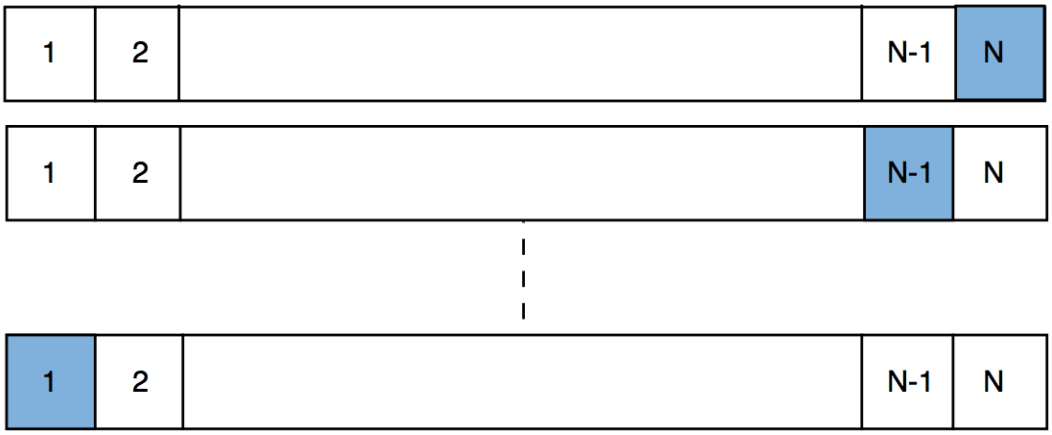
This might seem simple, but it is actually a very common mistake to not do this properly. We refer to the chapter "The Wrong and Right Way to Do Cross-validation" (Sec. 7.10.2) of Elements for Statistical Learning for more details.

Here we note a few very common mistakes:

- When starting our analysis with unsupervised learning, we might not split the data as we cannot evaluate any test error anyhow. However, if we do for example a principle component analysis (PCA) on the entire data set and then use the principle components as regressors in supervised models, then we are doing cross-validation wrongly, even if we split up the data for the supervised learning part. Instead we should have split up the data before doing the PCA.
- Several models, such neural networks, require us to normalise the data before training the model, i.e. transforming each column of the data set to have mean 0 and variance 1. A very common pitfall is to normalise the data first and then split the data set into training and test sets. Instead we should split the data first, evaluate the mean and variance on the training data and then rescale the hold-out set with respect to the mean and variance determined on the training set.

N-fold cross-validation

So far we have split the data in a training set and a cross-validation set using a single split, say for example a 80/20% split. Another way one could do cross-validation is to first split the data (after having hold out a test set) into N sets of equal size. In the case of N=5, each set would contain 20% of the data. What we could then do is to train the model on the set obtained when combining sets 1,2,...,N-1 and calculating the CV error on set N. Next we train the model on sets 1,2,..., N-2, N combined and evaluate the CV error on set N-1 etc. until in the final set we train the model on the combined set from 2,...,N and evaluate the CV error on set 1. This is illustrated in the following figure:



In following the above procedure we obtain a total of N different estimates for the CV error. We combine them by simply calculating the mean. Apart from calculating the mean we could also calculate the variance, since the test error is now a random number. When tuning the flexibility of our model we generally follow the so-called **one-standard-deviation-rule**, where we choose the least flexible model, where the test error is within one standard deviation of the smallest test error.

Exercise: Split the Housing Data into a training set `HousingData.train` and test set `HousingData.test`. (i) Split the training data further into a smaller training set `HousingData.train.tr` and a CV set `HousingData.train.cv`. Train a lasso model (using `glmnet()` from the package with the same name) for predicting the `MedianHouseValue` on the smaller training. Evaluate the CV error on the CV set. Use the CV error for a range of values for the shrinkage to fine-tune the model. Having obtained the optimal value of the shrinkage, retrain the model on the combined training and CV set. Finally, evaluate the test error on the test set.

Exercise: Repeat the previous exercise using 10-fold cross-validation. There is a build-in function `cv.glmnet()` which you can use directly on `HousingData.train` to do N-fold cross-validation.

Exercise (Optional): Write your own N-fold CV validation procedure using a for-loop. Repeat the previous exercise using 10-fold cross-validation implemented with your own procedure.

In this section we introduced cross-validation and in particular N-fold cross-validation. In principle N-fold cross-validation has better statistics than using a single split of the same size, e.g. 5-fold cross-validation vs a single 80/20% split. However, N-fold cross-validation also requires one to train the model N times which can be too computational intensive to be feasible. In other cases, there might be little gain in doing N-fold cross-validation in which case one might prefer a single split for simplicity.

Decision trees

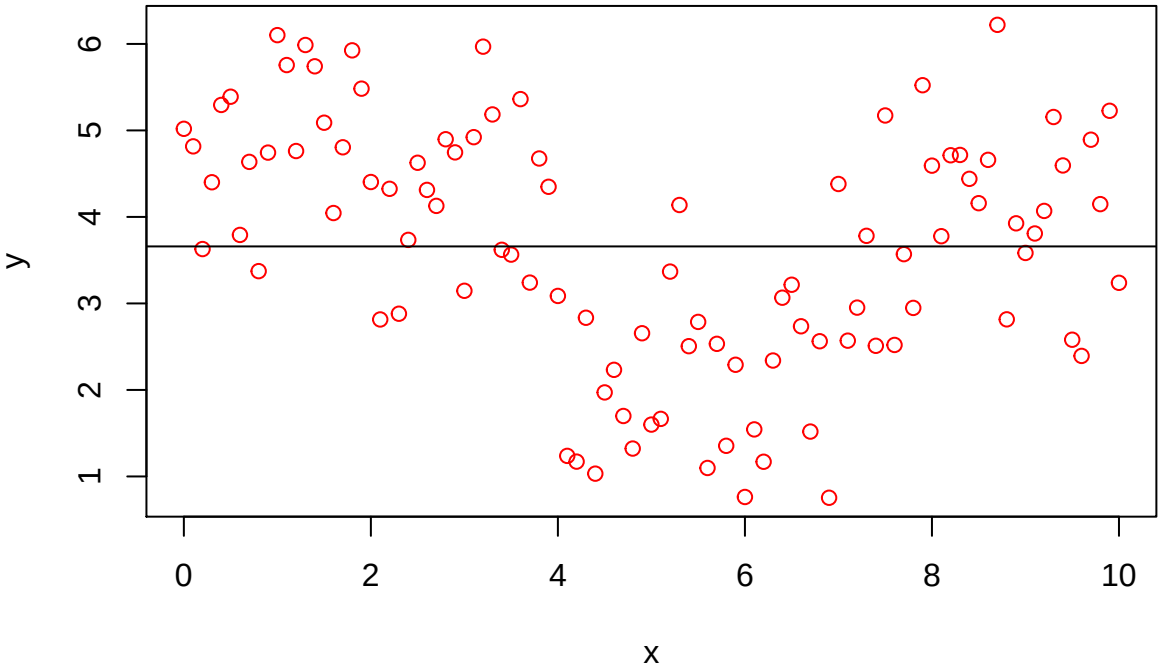
In this section we introduce decision trees which are covered in Chapter 8 of An Introduction to Statistical Learning. Sometimes decision trees also go under the name CART (classification and regression trees).

So far most of the learning algorithms we discussed during the first module "Statistical learning for data science: Basic Techniques using R" were **linear models** and variations thereof. The focus on this second module "Statistical learning for data science: Advanced Techniques using R" will be on **non-linear models**. Decision trees are our first example of non-linear models. A general difficulty in training non-linear models is the fact that the corresponding optimisation problem is often highly non-convex which makes it difficult to solve.

Regression trees

Let us again start with a toy example of simulated data. In the following, we give the code used to generate the simulated data. For those relatively new to `R` it is instructive to try to understand each command. The plot of the data points looks as follows:

```
set.seed(10)
x <- seq(0,10,by=0.1)
y <- c(rnorm(40,mean=5),rnorm(30,mean=2),rnorm(31,mean=4))
plot(y~x,col='red')
abline(h=mean(y))
```



Imagine we would like to fit a very simple model to the data which just consists of an intercept (i.e. a constant):

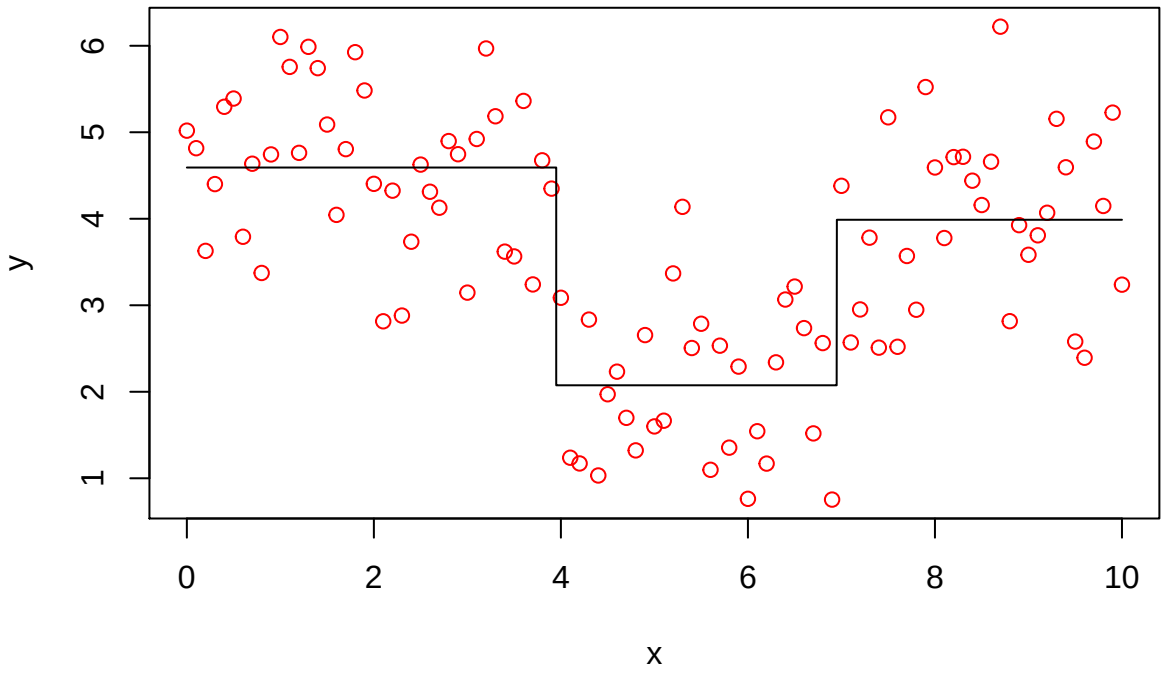
$$y_i = f(x_i) + \epsilon_i, \quad f(x) = a$$

It is intuitively clear that the least squared estimator $\hat{a}$ for a (or equivalent the maximum likelihood estimator) is given by the mean

$$\hat{a} = \text{mean}(\mathbf{y})$$

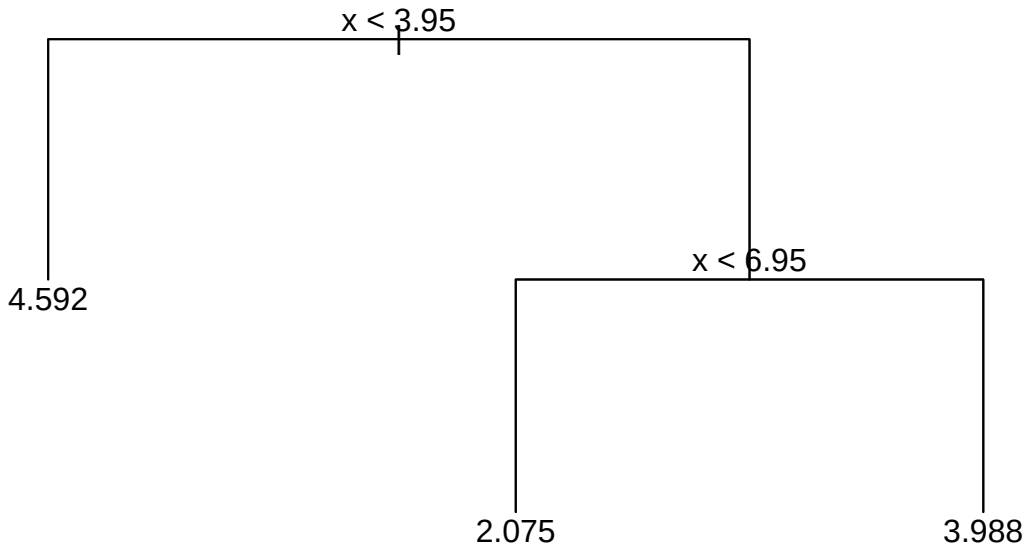
This can also be easily proven, e.g. using the mathematical formalism from the appendix of Handout 1 from the first module. The function $f(x) = \text{mean}(\mathbf{y})$ is shown in the figure above (black horizontal line).

Imagine now that instead of fitting a constant function to the entire range for x we are allowed to fit different constant functions to different intervals on the x-axis. Here is an example using three different intervals:



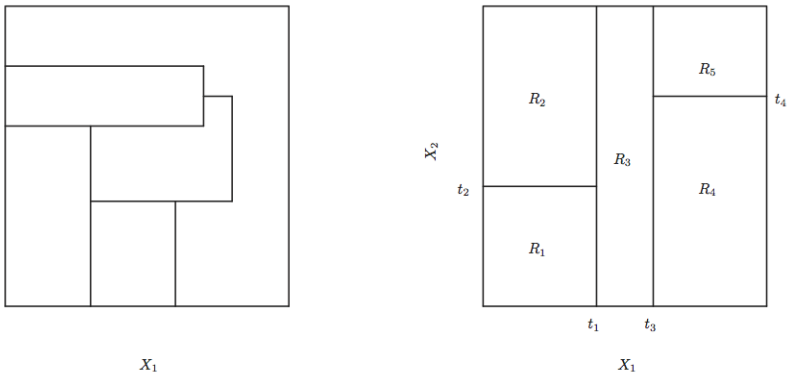
If you look back at how the data was simulated, namely, normal random variables with mean 5 for x=0,0.1,...,3.9, normal random variables with mean 2 for x=4,4.1, ..., 6.9 and normal random variables with mean 4 for x=7,7.1, ..., 10, then you see that this fit is actually very good. In principle, we can approximate any non-linear function by making the intervals of our piecewise constant function small enough.

The above fit is in fact a result from fitting a **regression tree** to the above data! All one does in a regression tree is to look for the split in the predictor space which leads to the largest decrease in RMSE when fitting a constant function in each interval. This is done in a **greedy approach**, meaning that once one has found a split, one keeps it and looks for the next best split etc. If one only performs binary splits, i.e. splits of intervals in two parts, and performs each split in a greedy fashion, then the sequence of splits can be mapped onto a **binary decision tree**. This is shown below:



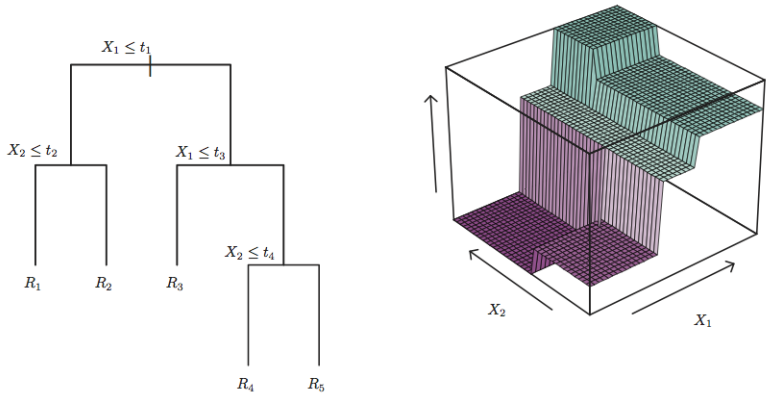
This is the decision tree corresponding to the previous plot. It is read as follows: Starting from the top we check whether our observation has $x < 3.95$ or $x \geq 3.95$. If $x < 3.95$, we follow the left branch which leads to a terminal node at which we predict $y = 4.591$. If $x \geq 3.95$, we follow the right branch, where we encounter another decision node. If furthermore $x < 6.95$, we predict $y = 2.075$, whereas if $x \geq 6.95$, we predict $y = 3.988$. The resulting piecewise constant function is the black line shown in the figure above.

The above construction can easily be generalised to higher-dimensional models. Consider for example the case where we have two predictors x_1 and x_2. The idea is most easily explained using the following two figures taken from ISLR:



We now consider partitions of the two dimensional predictor space, similar to the one-dimensional case, where we partitioned the x-axis in different intervals. It is important to notice that using binary splits applied in a greedy manner, we cannot obtain partition the predictor space as shown in the left part of the figure above, but rather only partitions of the type as shown in the right part are possible.

The following figure shows the decision tree corresponding to the splits shown in the right part of the above figure. It also shows an illustration of the corresponding piecewise constant function.



It is clear that the bigger the tree, the more flexible the model is. Thus, controlling the size of the tree is a type of regularisation, which is usually referred to as **pruning the tree**. This can be done by introducing a cost function which besides the residual sum of squares (RSS) has a penalty term proportional to the size of the tree (number of vertices):

$$\text{cost} = \text{RSS} + \alpha \, |\text{tree-size}|$$

To train regression trees in R, we can use the `tree` package

```
library(tree)
```

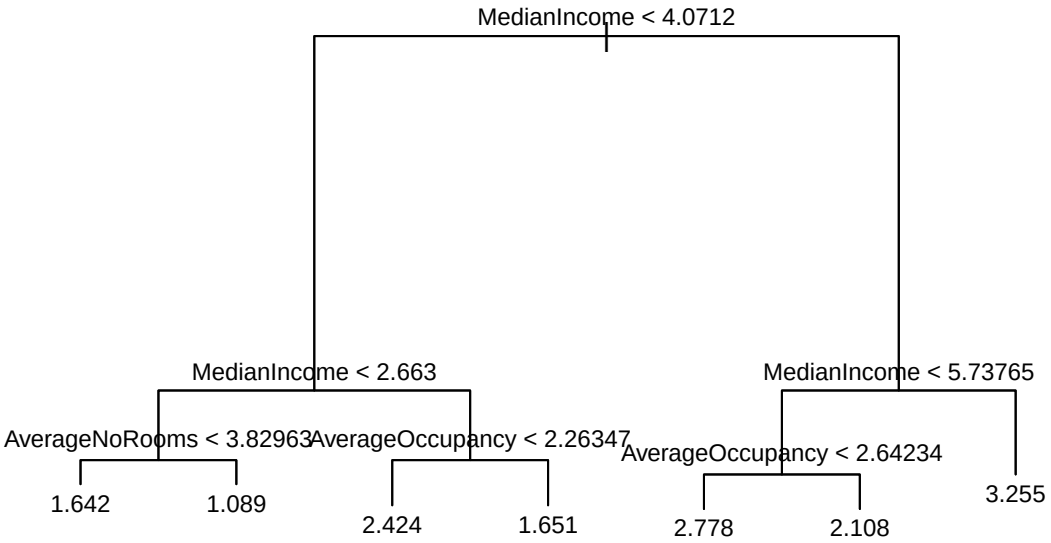
Training is done using the function `tree()` (see `?tree` for help):

```
reg.tree <- tree(MedianHouseValue ~. , data = HousingData.train)
summary(reg.tree)
```

```
##
## Regression tree:
## tree(formula = MedianHouseValue ~ ., data = HousingData.train)
## Variables actually used in tree construction:
## [1] "MedianIncome"     "AverageNoRooms"   "AverageOccupancy"
## Number of terminal nodes:  7
## Residual mean deviance:  0.4997 = 7850 / 15710
## Distribution of residuals:
##     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
## -2.58000 -0.47450 -0.09655  0.00000  0.36250  3.80000
```

The function uses the standard `R` formulas, where `MedianHouseValue ~.` means `MedianHouseValue` as a function of all predictors. We can make a plot of the decision tree:

```
plot(reg.tree)
text(reg.tree,cex=0.75)
```



To calculate the test error we make predictions on the test set using our trained regression tree and then calculate the mean squared error:

```
pred = predict(reg.tree, newdata = HousingData.test)
mean((HousingData.test$MedianHouseValue-pred)^2)
```

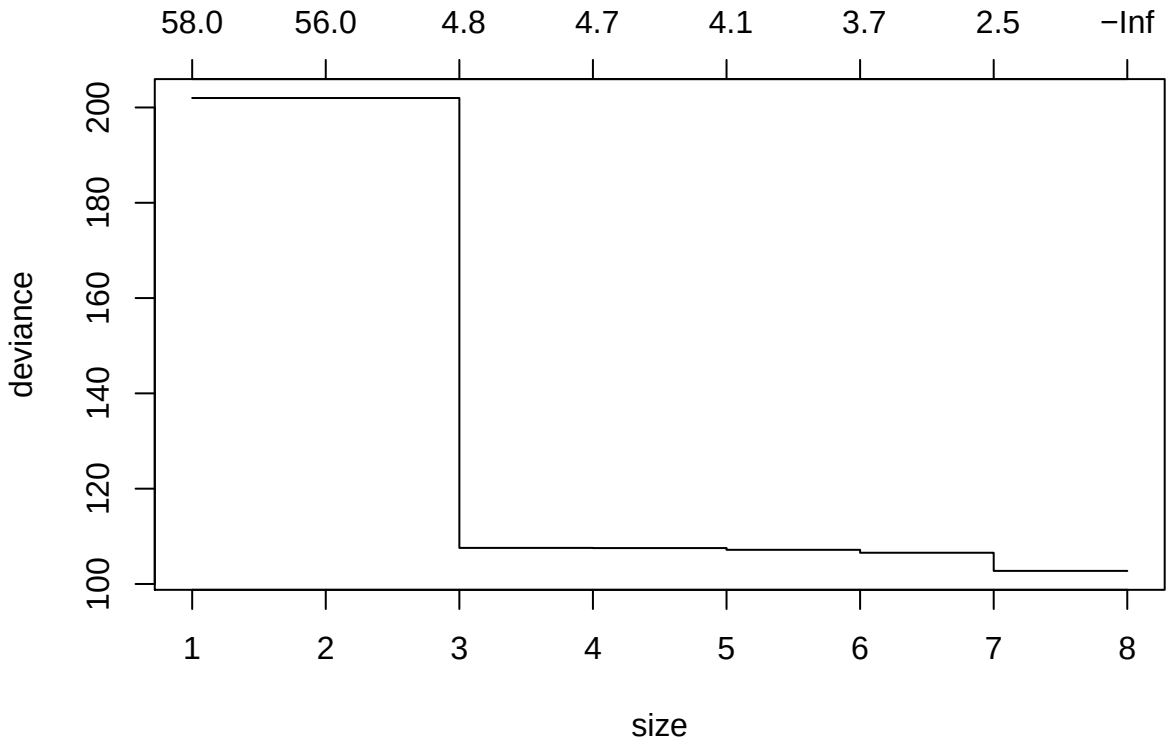
```
## [1] 0.5040067
```

The result is of the same order but slightly worse then the test error of our lasso model, which we trained in the third lecture of the first module.

Exercise: Go in detail through the above commands, executing them on your machine and inspecting the different outcomes.

The `tree` package as a build-in N-fold cross-validation functionality using the function `cv.tree()`. Once you determined the optimal size of the tree using cross-validation, you can prune the tree using the `prune.tree` function. Note that the `cv.tree()` uses deviance as a measure for the quality of fit, as opposed to RMSE or MSE, but there is no problem in using deviance to determine the optimal size of the tree.

Exercise: Take the toy model of simulated data and fit a regression tree. Perform 10-fold cross-validation and determine the optimal size of the tree. The cross-validation should result in a plot, similar to the following plot for deviance as a function of size of the tree:



Exercise: Perform a similar analysis as in the previous exercise for the regression tree trained on the Housing Data. Having determined the optimal tree size, calculate the test error on the test set.

Classification trees

A nice aspect of decision trees is that we can easily also deal with categorial variables both in the predictors as well as in the response variable. If the response variable is a categorial variable we call the model a **classification tree**. The basic principle remains the same. As opposed to predicting a constant continuous value for the response variable in a certain partition of the predictor space, we now predict a constant value of the label. The loss function is then changed from RSS to a measure suitable for a categorial variable, such as so-called cross-entropy or Gini-index.

Case study: Heart data

In this section we analysis a real-life data set. A similar analysis is also covered in Sec. 8.1.2 of the text book. Thus if you cannot finish it on time you can revise it at home. If you do finish early and want to practice a bit more, you could train a classification tree on the Arcene data set which we used in Lecture 3 of the first module.

The data has 303 observations. The response variable is `AHD`, heart disease, yes or no. There are 13 predictors include factors such as `Sex`, `Age` etc.

We start by importing the data

```
heart <- read.csv("Heart.csv")
```

Have a look at the data using `head()`, `str()` and `View()` (in RStudio).. You should see that there is some pre-processing to be done:

```
heart$X <- NULL
heart$AHD <- factor(heart$AHD)
heart$Sex <- factor(heart$Sex)
heart$Thal <- factor(heart$Thal)
heart$ChestPain <- factor(heart$ChestPain)
```

Exercise: Make sure that you understand the above commands.

We now split the data and training and test data:

```
set.seed(10)
n <- nrow(heart)
trainIndices <- sample(1:n, n*0.8)
heart.train <- heart[trainIndices,]
heart.test <- heart[-trainIndices,]
```

Before getting started let us have a look at the balance of labels

```
sum(heart.train$AHD=='Yes')/nrow(heart.train)
```

```
## [1] 0.4710744
```

```
sum(heart.test$AHD=='Yes')/nrow(heart.test)
```

```
## [1] 0.4098361
```

We see that both labels are rather balanced. It is important to check this, as for unbalanced data sets it is possible to have apparently good misclassification rates without actually having trained a good model.

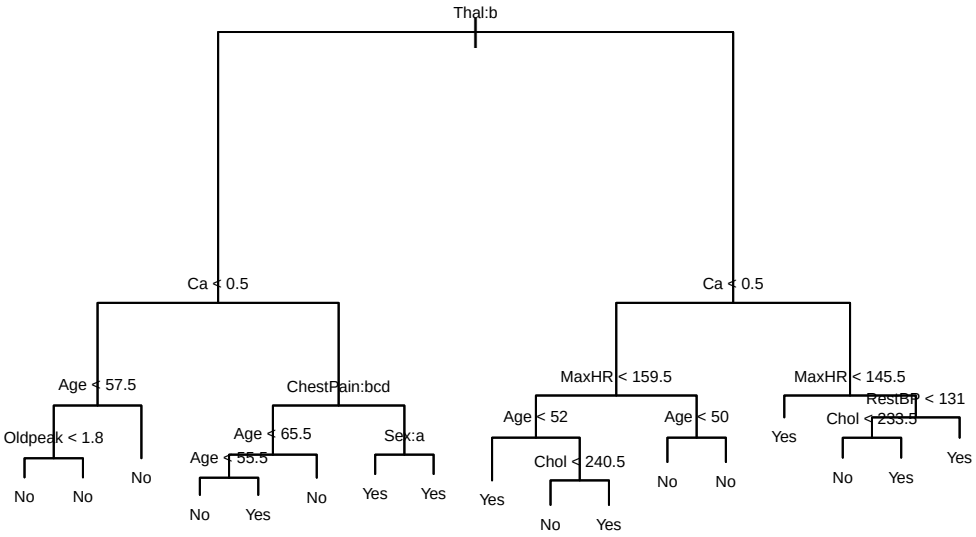
You can now fit a classification tree as follows:

```
heart.tree <- tree(AHD ~. ,data = heart.train)
summary(heart.tree)
```

```
##
## Classification tree:
## tree(formula = AHD ~ ., data = heart.train)
## Variables actually used in tree construction:
## [1] "Thal"      "Ca"        "Age"       "Oldpeak"   "ChestPain" "Sex"
## [7] "MaxHR"     "Chol"      "RestBP"
## Number of terminal nodes:  17
## Residual mean deviance:  0.442 = 97.69 / 221
## Misclassification error rate: 0.1008 = 24 / 238
```

and plot the tree:

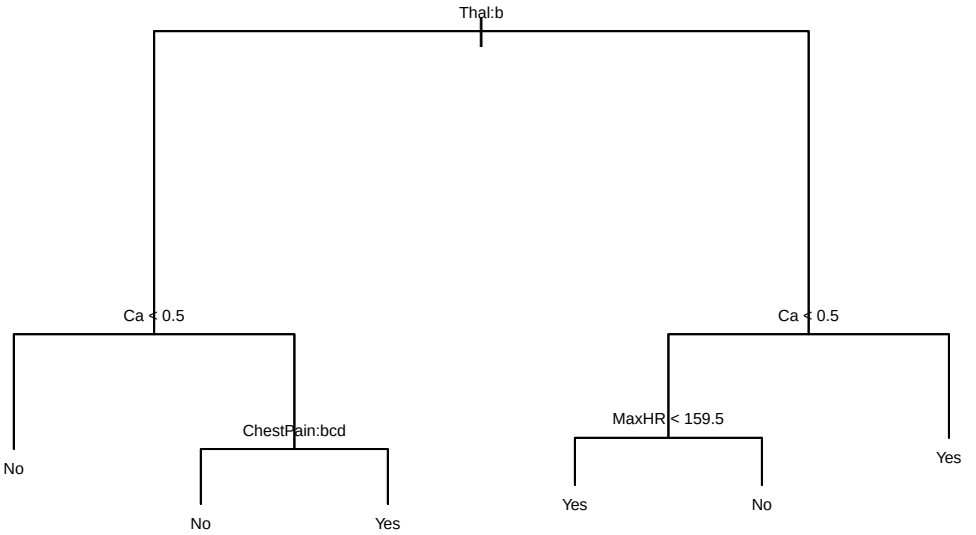
```
plot(heart.tree)
text(heart.tree,cex=0.5)
```



Exercise: Your takes is to properly train a classification tree on the heart data.

- Perform 10-fold cross-validation to obtain the optimal size of the classification tree and prune the tree (Hint: use `cv.tree` and `prune.misclass`)
- Make a plot of the decision tree of optimal size
- Evaluate the test error (misclassification rate) on the test data

Depending on your pruning and loss function, your final tree might be:



The misclassification rate evaluated on the test data set should be around:

```
## [1] 0.2622951
```

Final remarks

Classification and regression trees are simple non-linear models which are easy to train and particularly easy to interpret. In the next lecture we see how these models can be made much more powerful when combining them with ensemble techniques leasing to bagging, boosting and random forests. The previous models are some of the most popular models in data mining.