Advanced Data Analysis with R

Lecture notes with selected answers to problems

Florian Meinfelder

Release date: `2017-07-15`

1 Overview

1.1 Why Data Analysis with R?

- R has become an increasingly popular tool for analyzing various kinds of data (due to its flexibility and vast availability of implemented methods)
- This appears strange at first sight, given that R looks very raw (some might say ‘ugly’) compared to commercial packages
- Usability has increased tremendously with the advent of powerful editors and IDE’s, as well as books on R (truck loads of books) and other documentation
- (But one might argue that books were written and powerful editors developed, *because* of R’s increasing popularity)
- So…what is the difference (benefit?) of R compared to the commercial software packages?

1.2 Reasons for R’s popularity

- For a start it’s free…
- Open source and free accessability has lead to a large contributing community
- New methods are much quicker available in R than in commercial packages
- R is very adaptable – you can do almost anything with it!

2 R’s Object Orientation

2.1 Different Objects in R

If you have worked with SPSS or Stata so far, you are probably used to rectangular data structures (such as .sav and .dta files, Excel sheets or ASCII files). All commands in those statistics compilers are interpreted as manipulations of such a rectangular data set: adding a new variable a = 1 (via `(e)gen` in Stata or `compute` in SPSS) will get you a vector consisting of n ’1’s which is appended to the existing work file. Macros in SPSS and do-Files in Stata give you some additional options for generating scalars or vectors smaller than the total number of observations, etc. But elaborated programming with SPSS and Stata (or SAS or M-plus,or…) is kind of nerve-wrecking if your task lies beyond standard functions which are already implemented.

If you have to create your own code and statistical algorithms, R has a wider range of possible objects for programming tasks, quite similar to genuine programming languages such as C, C++, Java, Python, Ruby …

- Scalars
- Vectors

- Matrices
- Arrays
- Data frames
- Lists

R is *object-oriented*!

2.1.1 Scalars and Vectors

A vector:

$$\mathbf{y} = (1, 3, 1, 2, 3.5, 1)'$$

And this is how we assign it in R:

```
y <- c(1,3,1,2,3.5,1);y
```

```
## [1] 1.0 3.0 1.0 2.0 3.5 1.0
```

Accessing an element, say y_3, via brackets:

```
y[3]
```

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

Note: `<-` is the assignment statement `=` works as well – but please do not use it…)

... or the elements 2 to 4:

```
y[2:4]
```

```
## [1] 3 1 2
```

```
2:4
```

```
## [1] 2 3 4
```

$\ldots$ or y_2, y_4, y_6:

```
y[c(2,4,6)]
```

```
## [1] 3 2 1
```

```
y[seq(0,6,2)]
```

```
## [1] 3 2 1
```

```
seq(0,6,2)
```

```
## [1] 0 2 4 6
```

We can name each element:

```
names(y) <- c("Well", "the", "bird","is", "the", "word")
y
```

```
## Well  the bird   is  the word
##  1.0  3.0  1.0  2.0  3.5  1.0
```

… and access it by indexing with the assigned name:

```
y["bird"]
```

```
## bird
##    1
```

```
y[c("Well","the","is")]
```

```
## Well  the   is
##    1    3    2
```

Exclude elements:

```
y[-3]
```

```
## Well  the   is  the word
##  1.0  3.0  2.0  3.5  1.0
```

```
y[-(3:6)]
```

```
## Well  the
##    1    3
```

```
y[-c(2,5)]
```

```
## Well bird   is word
##    1    1    2    1
```

```
#but
#y[-"bird"] and
#y[-3:6] won't work
```

```
# EXERCISE 1.1: Create a vector that represents the integers from 20 to 11.
# Subsequently, Reorder this vector such that it contains the integers from 11 to 2.
```

Consider the following two vectors

$$\mathbf{x} = (1, 2, 0, 0.2, 2.5, 1)';\ \mathbf{y} = (1, 3, 1, 2, 3.5, 1)'$$

We assign them (again):

```
x <- c(1,2,0,0.2,2.5,1);x
```

```
## [1] 1.0 2.0 0.0 0.2 2.5 1.0
```

```
y <- c(1,3,1,2,3.5,1);y
```

```
## [1] 1.0 3.0 1.0 2.0 3.5 1.0
```

Add and subtract:

```
x+y
```

```
## [1] 2.0 5.0 1.0 2.2 6.0 2.0
```

```
x-y
```

```
## [1]  0.0 -1.0 -1.0 -1.8 -1.0  0.0
```

Multiply elementwise:

```
2*x
```

```
## [1] 2.0 4.0 0.0 0.4 5.0 2.0
```

```
x*y
```

```
## [1] 1.00 6.00 0.00 0.40 8.75 1.00
```

Calculate the inner/dot product $\mathbf{x}'\mathbf{y}$:

```
x%*%y
```

```
##       [,1]
## [1,] 17.15
```

Or calculate the outer/dyadic product $\mathbf{x} \otimes \mathbf{y} = \mathbf{x}\mathbf{y}'$:

```
outer(x,y)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]  1.0  3.0  1.0  2.0 3.50  1.0
## [2,]  2.0  6.0  2.0  4.0 7.00  2.0
## [3,]  0.0  0.0  0.0  0.0 0.00  0.0
## [4,]  0.2  0.6  0.2  0.4 0.70  0.2
## [5,]  2.5  7.5  2.5  5.0 8.75  2.5
## [6,]  1.0  3.0  1.0  2.0 3.50  1.0
```

```
x%*%t(y)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]  1.0  3.0  1.0  2.0 3.50  1.0
## [2,]  2.0  6.0  2.0  4.0 7.00  2.0
## [3,]  0.0  0.0  0.0  0.0 0.00  0.0
## [4,]  0.2  0.6  0.2  0.4 0.70  0.2
## [5,]  2.5  7.5  2.5  5.0 8.75  2.5
## [6,]  1.0  3.0  1.0  2.0 3.50  1.0
```

2.1.2 Matrices

Now let's consider a matrix

$$\mathbf{X} = \begin{bmatrix} 2 & 1 \\ 3.4 & 0.3 \end{bmatrix}$$

And this is how we assign it:

```
X <- matrix(c(2,3.4,1,0.3),2);X
```

```
##      [,1] [,2]
## [1,]  2.0  1.0
## [2,]  3.4  0.3
```

Extract the second row ($\mathbf{x}_{2\cdot}$):

```
X[2, ]
```

```
## [1] 3.4 0.3
```

Extract the first column ($\mathbf{x}_{\cdot 1}$):

```
X[,1]
```

```
## [1] 2.0 3.4
```

Extract the last element (x_{11}) and change it to 1:

```
X[2,2]
```

```
## [1] 0.3
```

```
X[2,2] <- 1
```

Extract the diagonal elements:

```
diag(X)
```

```
## [1] 2 1
```

Change row- and column names:

```
rownames(X) <- c("Apples", "Pears")
colnames(X) <- c("Pie", "Tart")
```

Access via these names:

```
X[,"Pie"]
```

```
## Apples  Pears
##    2.0    3.4
```

```
X[,c("Pie","Tart")]
```

```
##        Pie Tart
## Apples 2.0    1
## Pears  3.4    1
```

```
X["Apples",]
```

```
##  Pie Tart
##    2    1
```

```
X["Apples","Pie"]
```

```
## [1] 2
```

Exclude (a) row(s):

```
X[-1,]
```

```
##  Pie Tart
##  3.4  1.0
```

```
X[-c(1,2),]
```

```
##      Pie Tart
```

Exclude (a) column(s):

```
X[,-2]
```

```
## Apples  Pears
##    2.0    3.4
```

```
X[,-c(2,1)]
```

```
##
## Apples
## Pears
```

Exclude rows and columns (not single elements!):

```
X[-1,-2]
```

```
## [1] 3.4
```

```
X[-2,-2]
```

```
## [1] 2
```

```
X[-2,2]
```

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

Transpose of a matrix:

```
t(X)
```

```
##      Apples Pears
## Pie       2   3.4
## Tart      1   1.0
```

Multiply elementwise and as matrix product:

```
X*X
```

```
##          Pie Tart
## Apples  4.00    1
## Pears  11.56    1
```

```
X%*%X
```

```
##         Pie Tart
## Apples  7.4  3.0
## Pears  10.2  4.4
```

Invert the matrix:

```
solve(X)
```

```
##          Apples      Pears
## Pie  -0.7142857  0.7142857
## Tart  2.4285714 -1.4285714
```

```
#EXERCISE 2.1: Assume you have a design matrix $X$ and a response variable vector y. The OLS estimator for beta is
given by $\hat{\beta}=(X^tX)^{-1}X^ty$. Produce the corresponding _R_ syntax.
```

2.1.3 Arrays

Arrays have the purpose of storing higher dimensional matrices An assignment example for a $2 \times 2 \times 3$-array looks like this:

```
arr1 <- array(1:12,dim=c(2,2,3));arr1
```

```
## , , 1
##
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
##
## , , 2
##
##      [,1] [,2]
## [1,]    5    7
## [2,]    6    8
##
## , , 3
##
##      [,1] [,2]
## [1,]    9   11
## [2,]   10   12
```

Extract $every$ first row of all matrices stored:

```
arr1[1,,]
```

```
##      [,1] [,2] [,3]
## [1,]    1    5    9
## [2,]    3    7   11
```

Extract $every$ second column of all matrices stored:

```
arr1[,2,]
```

```
##      [,1] [,2] [,3]
## [1,]    3    7   11
## [2,]    4    8   12
```

Extract the second matrix in the array:

```
arr1[,,2]
```

```
##      [,1] [,2]
## [1,]    5    7
## [2,]    6    8
```

Extract the first row in the third matrix and the second column in the first matrix:

```
arr1[1,,3]
```

```
## [1]  9 11
```

```
arr1[,2,1]
```

```
## [1] 3 4
```

Extract the first element in (every) matrix:

```
arr1[1,1,]
```

```
## [1] 1 5 9
```

Extract the first element in the third matrix:

```
arr1[1,1,3]
```

```
## [1] 9
```

2.1.4 Data frames

Data frames are the most common format for storing rectangular data. We can load sample data sets (shipped with R) with data(), load already stored data or create our own data frame.

```
DAT1 <- data.frame(Var1 = 1:5, Var2=c(3,2,1,2,2.4), Var3=c("M","F", "F", "F","M"))
DAT1
```

```
##   Var1 Var2 Var3
## 1    1  3.0    M
## 2    2  2.0    F
## 3    3  1.0    F
## 4    4  2.0    F
## 5    5  2.4    M
```

Extract the third column and the second row (the way we already have done before):

```
DAT1[,3]
```

```
## [1] M F F F M
## Levels: F M
```

```
DAT1[2,]
```

```
##   Var1 Var2 Var3
## 2    2    2    F
```

```
DAT1[,"Var3"]
```

```
## [1] M F F F M
## Levels: F M
```

But there is more: We can access the columns via `[[]]` and `$` :

```
DAT1[[3]]
```

```
## [1] M F F F M
## Levels: F M
```

```
DAT1$Var3
```

```
## [1] M F F F M
## Levels: F M
```

2.1.5 Lists

Lists are able to store every object you have already met, i.e. scalars, vectors, matrices, arrays, data frames and, yes, lists.

```
aList <- list(Vector1 = x, Vector2 = y, ANumber=3, ALetter="a",
  AMatrix=X, Datas=DAT1, AList = list(One=1, Two=2))
aList
```

```
## $Vector1
## [1] 1.0 2.0 0.0 0.2 2.5 1.0
##
## $Vector2
## [1] 1.0 3.0 1.0 2.0 3.5 1.0
##
## $ANumber
## [1] 3
##
## $ALetter
## [1] "a"
##
## $AMatrix
##        Pie Tart
## Apples 2.0    1
## Pears  3.4    1
##
## $Datas
##   Var1 Var2 Var3
## 1    1  3.0    M
## 2    2  2.0    F
## 3    3  1.0    F
## 4    4  2.0    F
## 5    5  2.4    M
##
## $AList
## $AList$One
## [1] 1
##
## $AList$Two
## [1] 2
```

Just like data frames (because a data frame is a special kind of list), we can access the list elements via `[[]]` and `$`:

```
aList$Vector1
```

```
## [1] 1.0 2.0 0.0 0.2 2.5 1.0
```

```
aList[[1]]
```

```
## [1] 1.0 2.0 0.0 0.2 2.5 1.0
```

```
aList$Vector1[2:4]
```

```
## [1] 2.0 0.0 0.2
```

```
aList$Datas$Var1
```

```
## [1] 1 2 3 4 5
```

```
aList$Datas[[1]]
```

```
## [1] 1 2 3 4 5
```

```
aList[[6]][[1]]
```

```
## [1] 1 2 3 4 5
```

```
aList[[6]][,1]
```

```
## [1] 1 2 3 4 5
```

2.2 Logical Operators

`TRUE` and `FALSE` are used to evaluate an expression. The operators that are used for this purpose are `<` (less than), `<=` (less equal), `>` (greater), `>=` (greater equal), `==` (equal) and `!=` (unequal).

```
5 < 6
```

```
## [1] TRUE
```

```
(a <- 3)
```

```
## [1] 3
```

```
a == 3
```

```
## [1] TRUE
```

```
a != 3
```

```
## [1] FALSE
```

We can also evaluate a non-atomic object elementwise:

```
y
```

```
## [1] 1.0 3.0 1.0 2.0 3.5 1.0
```

```
y < 2
```

```
## [1]  TRUE FALSE  TRUE FALSE FALSE  TRUE
```

…and two objects (ideally of identical dimensions)

```
x
```

```
## [1] 1.0 2.0 0.0 0.2 2.5 1.0
```

```
y != x
```

```
## [1] FALSE  TRUE  TRUE  TRUE  TRUE FALSE
```

Element positions which are evaluated as `TRUE` can be used for indexing.

```
x[y > 1]
```

```
## [1] 2.0 0.2 2.5
```

In this example, elements of `x` are selected if the corresponding element of `y` is greater than 1.

The `!` character negates logical expressions.

```
x[!(y > 1)]
```

```
## [1] 1 0 1
```

Now we extract those elements of `x` , where the corresponding element of `y` is *NOT* greater 1.

```
#EXERCISE 2.2: Take the `DAT1`data frame, and create a subset of the male observations (i.e. rows, where `var3`equ
als "M")
```

2.3 Classes and Modes

We have seen a lot of numbers already, and some characters as well, but there is something missing: measurement scales. Note that you can assign matrices and vectors either with numbers, characters (strings), or logicals (TRUE/FALSE).

```
(a <- c(2,2,1,42))
```

```
## [1]  2  2  1 42
```

```
(b <- c("a", "d", "e","M"))
```

```
## [1] "a" "d" "e" "M"
```

Now Let us combine them to create another vector and a matrix:

```
(combivec <- c(a,b))
```

```
## [1] "2"  "2"  "1"  "42" "a"  "d"  "e"  "M"
```

```
(combiMat <- rbind(a,b))
```

```
##   [,1] [,2] [,3] [,4]
## a "2"  "2"  "1"  "42"
## b "a"  "d"  "e"  "M"
```

The quotation marks indicate that something is different. We can check this using `mode()` and `class()` :

```
mode(a); class(a)
```

```
## [1] "numeric"
```

```
## [1] "numeric"
```

```
mode(b); class(b)
```

```
## [1] "character"
```

```
## [1] "character"
```

```
mode(combivec); class(combivec)
```

```
## [1] "character"
```

```
## [1] "character"
```

```
mode(combiMat); class(combiMat)
```

```
## [1] "character"
```

```
## [1] "matrix"
```

- `mode` specifies how the data is stored internally (numeric, character, logical)
- `class` specifies what object type is used.

Mode and class are not redundant. As you might have noticed at `combiMat` : The `mode` is `character` , the `class` is `matrix` . Therefore, the mode of the first column of our data frame `DAT1` is numeric and the mode of the last column is …

```
mode(DAT1[[1]])
```

```
## [1] "numeric"
```

```
mode(DAT1[[3]])
```

```
## [1] "numeric"
```

… not a character, because the third column is a factor variable.

```
class(DAT1[,3])
```

```
## [1] "factor"
```

The internal representation of factor variables (ordered variables and integer variables as well), even though they are labeled, is still numeric. But what about *list* and *data frame*? They seem to mix these variables – they have their own mode and class. A data frame’s class is `data.frame` , but its mode - as mentioned above - is in fact `list` . Well, and a list is a list.

```
mode(DAT1)
```

```
## [1] "list"
```

```
class(DAT1)
```

```
## [1] "data.frame"
```

```
mode(aList)
```

```
## [1] "list"
```

```
class(aList)
```

```
## [1] "list"
```

Note that the the output of a linear regression analysis in *R* is also a `list`.

```
data(anscombe)
model_ansc <- lm(y1 ~ x1, data=anscombe)
mode(model_ansc)
```

```
## [1] "list"
```

```
class(model_ansc)
```

```
## [1] "lm"
```

Therefore, you can access these output elements using `$` and `[[]]`

```
names(model_ansc)
```

```
##  [1] "coefficients"  "residuals"     "effects"       "rank"         
##  [5] "fitted.values" "assign"        "qr"            "df.residual"  
##  [9] "xlevels"       "call"          "terms"         "model"
```

```
model_ansc$coefficients
```

```
## (Intercept)          x1 
##   3.0000909   0.5000909
```

```
model_ansc[[1]]
```

```
## (Intercept)          x1
##   3.0000909   0.5000909
```

2.4 Conversion

Item We have learned about scales of measurement in R, but how do we construct them? Let us start by defining a vector:

```
y <- c(1,2,3)
class(y)
```

```
## [1] "numeric"
```

Suppose we want a factor variable: We can use `as.factor()` converting already existing data or `factor()` . The latter is useful as well if you want to create a factor variable immediately.

```
(y1 <- as.factor(y))
```

```
## [1] 1 2 3
## Levels: 1 2 3
```

```
class(y1)
```

```
## [1] "factor"
```

```
(y2 <- factor(c(1,2,1,1,1,2), labels=c("M", "W")))
```

```
## [1] M W M M M W
## Levels: M W
```

```
class(y2)
```

```
## [1] "factor"
```

You can also change a variable within a data frame

```
class(DAT1[,2])
```

```
## [1] "numeric"
```

```
DAT1[,2] <- as.factor(DAT1[,2])
class(DAT1[,2])
```

```
## [1] "factor"
```

The procedure for ordered variables, integer variables (whole numbers) and metric-scale variables (numeric) is quite similar.

```
(y3 <- as.ordered(y))
```

```
## [1] 1 2 3
## Levels: 1 < 2 < 3
```

```
class(y3)
```

```
## [1] "ordered" "factor"
```

```
(y4 <- as.integer(y))
```

```
## [1] 1 2 3
```

```
class(y4)
```

```
## [1] "integer"
```

```
(y5 <- as.numeric(y4))
```

```
## [1] 1 2 3
```

```
class(y5)
```

```
## [1] "numeric"
```

Tip: `ordered()` works as an alternative to `as.ordered()`, whereas `numeric()` and `integer()` do not:

```
(y6 <- as.ordered(y))
```

```
## [1] 1 2 3
## Levels: 1 < 2 < 3
```

```
class(y6)
```

```
## [1] "ordered" "factor"
```

```
(y7 <- as.integer(y))
```

```
## [1] 1 2 3
```

```
class(y7)
```

```
## [1] "integer"
```

```
(y8 <- as.numeric(y4))
```

```
## [1] 1 2 3
```

```
class(y8)
```

```
## [1] "numeric"
```

`numeric()` and `integer()` are just needed for creating zero-filled vectors (pre-allocation of memory):

```
(y7 <- integer(10))
```

```
##  [1] 0 0 0 0 0 0 0 0 0 0
```

```
#Alternative
(y7 <- vector(mode="integer", length=10))
```

```
##  [1] 0 0 0 0 0 0 0 0 0 0
```

```
(y8 <- numeric(10))
```

```
##  [1] 0 0 0 0 0 0 0 0 0 0
```

```
#Alternative
(y8 <- vector(mode="numeric", length=10))
```

```
##  [1] 0 0 0 0 0 0 0 0 0 0
```

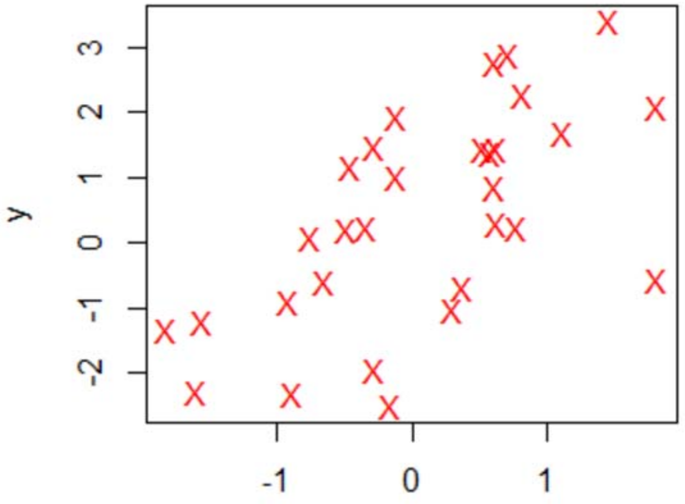
```
#EXERCISE 2.3: Identify the class and mode of the first variable (Var1) of the data frame `DAT1`. Convert this variable _within_ `DAT1` into a factor and check the class again.
```

3 Data Visualization

3.1 Using the plot device

General information on *R* graphics devices: Standard *R* plots are rather “sober”, but there are parameters to pimp them (and tons of packages, such as `lattice` , `ggplot2` ,…) *R* plots are embedded in `<nameOfDevice>(Parameters) ... dev.off()`

```
set.seed((12345))
x <- rnorm(30)
y <- x+rnorm(30)
windows()
plot(x,y, pch="X", col="red")
```



```
dev.off()
```

```
## png
##   2
```

3.2 High-level and low-level graphics functions

Some graphics function automatically initiate a new plot, e.g.

- `plot()` : generic plot method
- `barplot()` : requires a factor or table as input
- `pie()` : creates a pie chart
- `hist()` : requires a numeric (or integer) vector or data.frame element

We refer to those graphics functions as ‘high-level’ functions.

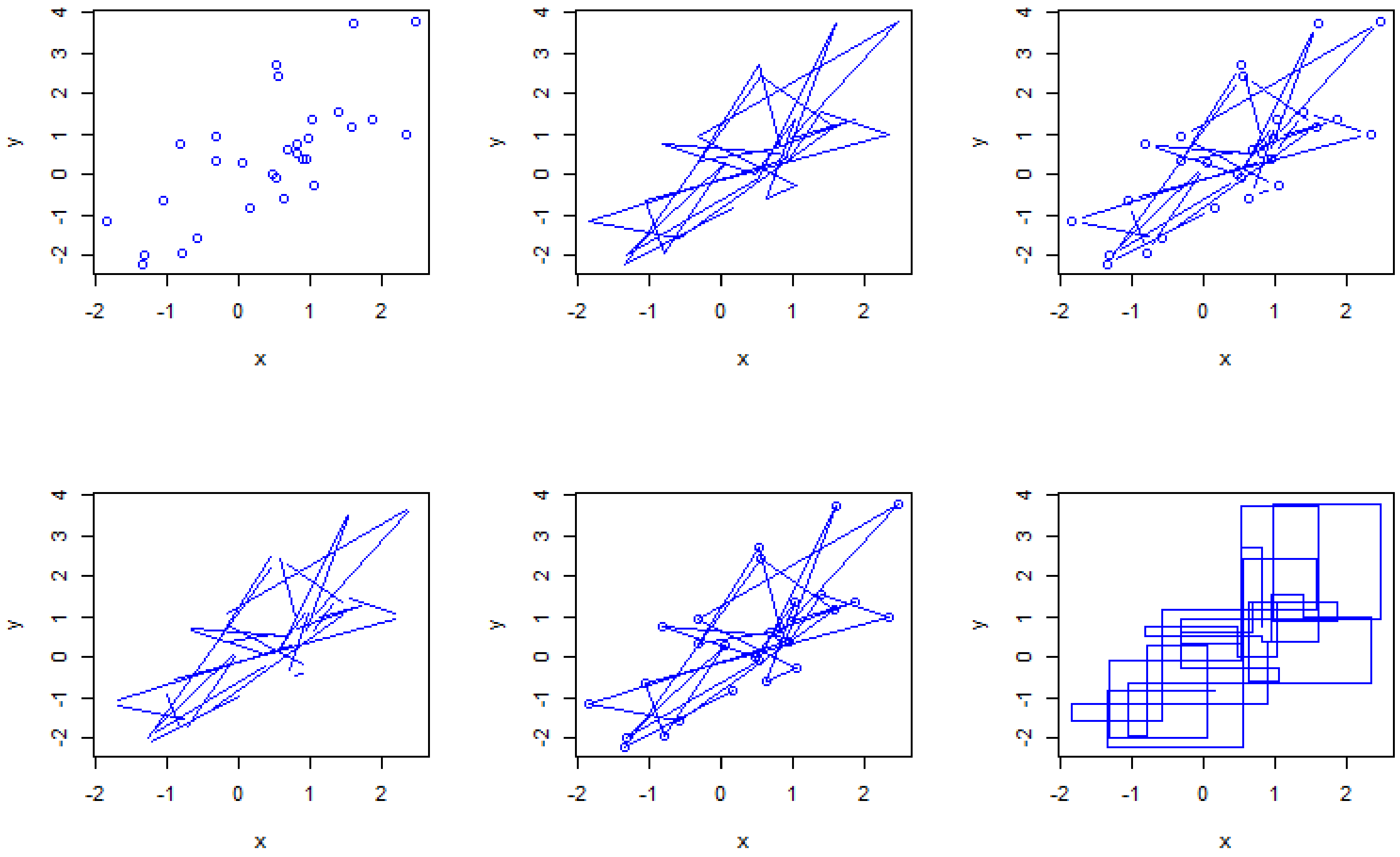
On the other hand, ‘low-level’ graphics functions are ‘supplements’ to ‘high-level’ functions, and can not initiate a plot, e.g.

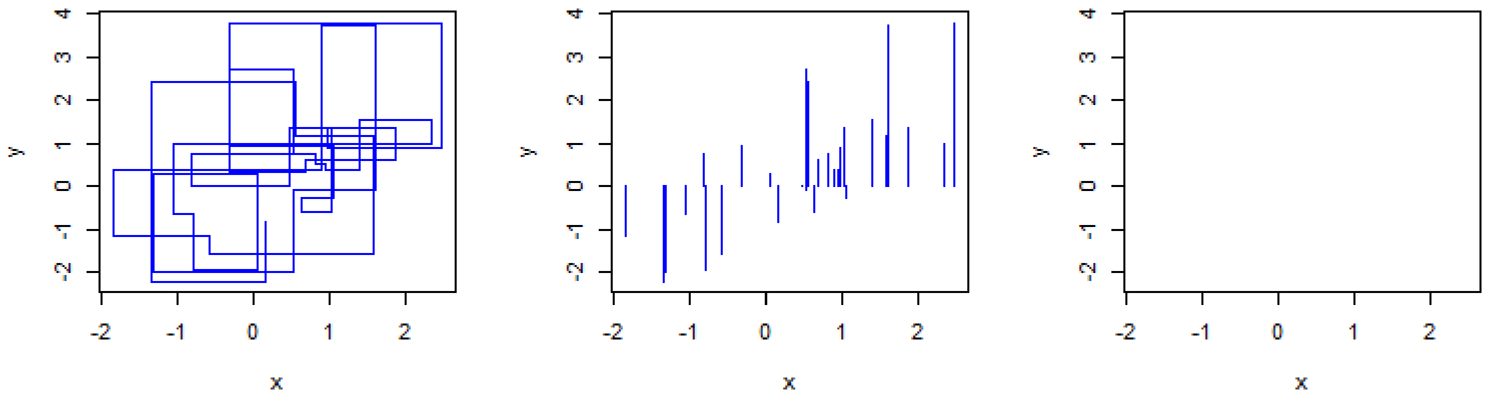
- `points()` : adding points based on an x (and optional y) vector
- `lines()` : see above but joining the points
- `abline()` : can be either a scalar for a horizontal or vertical line, a vector with elements for intercept (a) and slope (b), or an lm-object
- `polygon()` : creates polygons based on x and y coordinates
- `arrows()` : create an arrow based on x and y coordinates
- `legend()` : add a legend to a plot (position via x and y coordinates or via prespecified parameters such as `"topright"`)
- `text()` : free text in plot based on x and y coordinates
- `mtext()` : margin text on either side of the plot

3.3 plot() and par() define a lot of settings

`plot.default()` has many parameter options… `type = "p"` or `"l"` or `"b"` or `"c"` or `"o"` or `"s"` or `"S"` or `"h"` or `"n"`

```
x <- rnorm(30)
y <- x+rnorm(30)
typ <- c("p", "l", "b", "c", "o", "s", "S", "h", "n")
layout(matrix(1:9,nrow=3,byrow=TRUE))
for (i in typ) {
plot(x, y, col="blue", type=i)
}
```





```
# EXERCISE 3.1: Create a data.frame 'someData' with n=1000 rows,
# consisting of a single variable 'PartyVote',and assign "Republican" with
# p=0.48 and "Democrat" with p=0.52 (hint: look up the help of the sample
# function).
# Create a horizontal bar plot with colors "blue" for the Republicans and
# "red" for the Democrats and add a legend in the bottom right corner
```

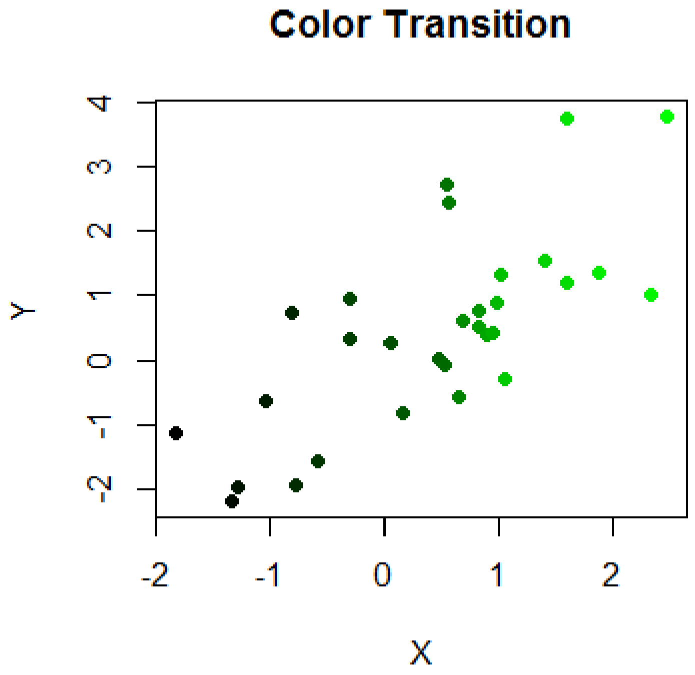
3.4 Color Schemes

R provides several ways to specify colors for your graph. 657 colors can be chosen by specifying the *name* of the color (e.g. “red” or “honeydew”). The complete palette can be found by typing ‘colors()’ into the console.

But in total, there exist $256^3 = 16,777,216$ different colors! Any of them can be accessed via the ‘rgb()’ function (referring to “red”, “green”, “blue”):

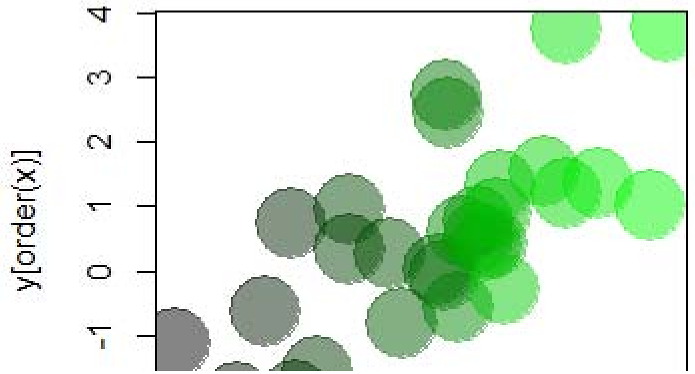
```
greenPerc <- round(seq(0, 255, length.out = 30))
plot(x[order(x)], y[order(x)], xlab="X", ylab="Y", main="Color Transition",
     col=rgb(rep(0, 30), greenPerc, rep(0, 30), maxColorValue = 255), pch=19)
```

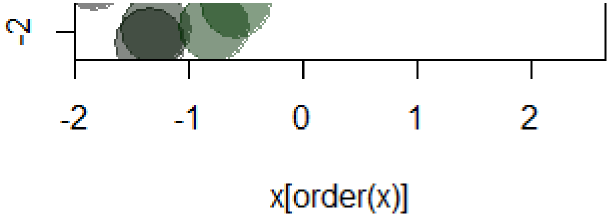
Color Transition



What if the symbols overlap? There is a way to make the symbols partially transparent which is referred to as “alpha blending”. The alpha channel serves as the fourth argument of the ‘rgb()’ function.

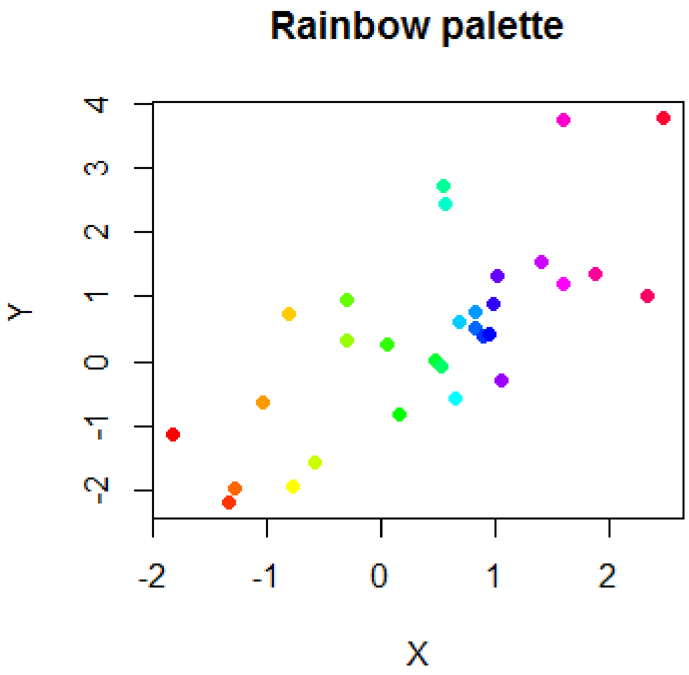
```
plot(x[order(x)], y[order(x)], col=rgb(rep(0,30), greenPerc, rep(0, 30), alpha=123,
                                        maxColorValue = 255), pch=19, cex=5)
```





Alternatively to specifiying color transitions manually, you can work with “palettes”, such as ‘rainbow’ or ‘terrain.colors’.

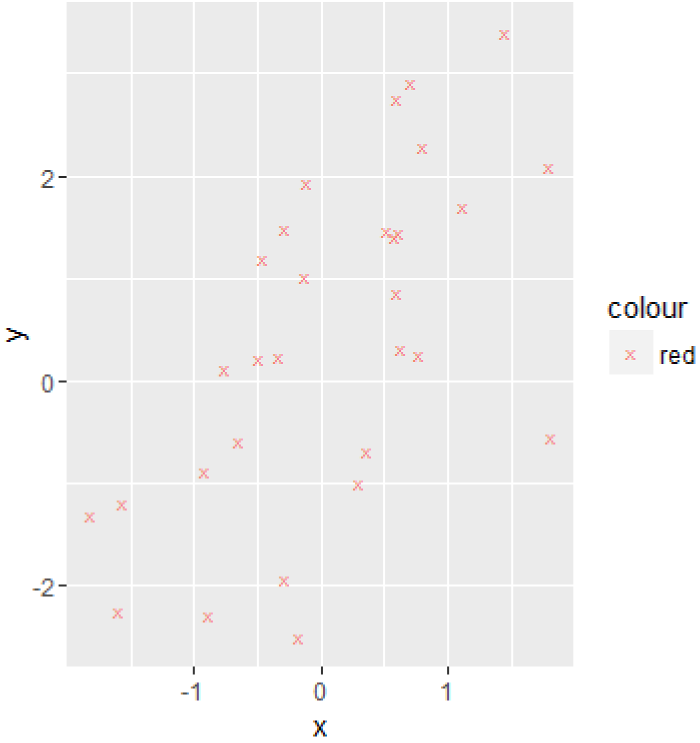
```
plot(x[order(x)], y[order(x)], xlab="X", ylab="Y", main="Rainbow palette",
     col=rainbow(30), pch=19)
```



3.5 The ggplot2 Package

ggplot2 by Hadley Wickham has become one of the most popular packages in *R*. One reason might be that the vanilla options create nicer looking plots than the standard functions:

```
if (!require("ggplot2")) install.packages("ggplot2")
library(ggplot2)
set.seed((12345))
x <- rnorm(30)
y <- x+rnorm(30)
D <- data.frame(x,y)
windows()
ggplot(D, aes(x=x, y=y, col="red")) + geom_point(shape="X")
```

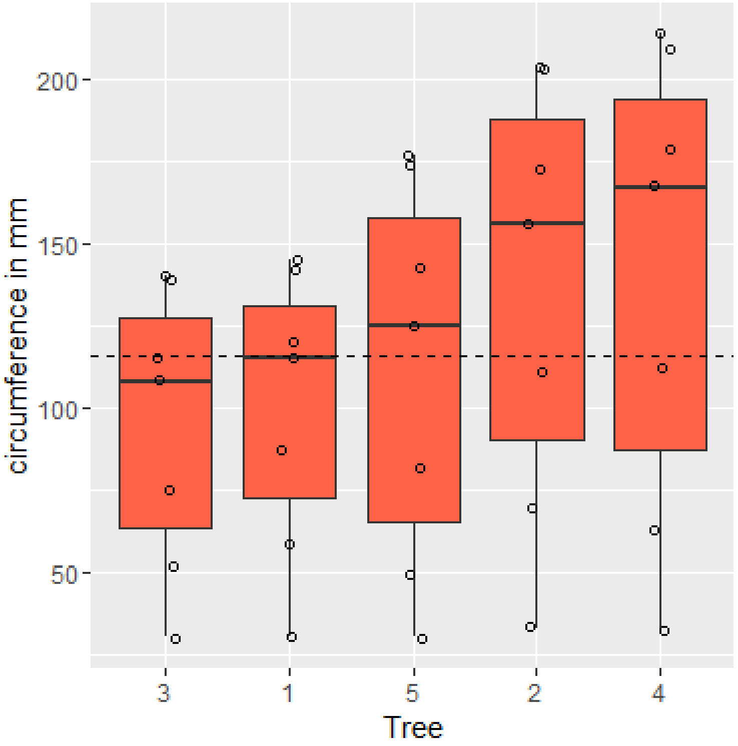


```
dev.off()
```

```
## png
##   2
```

The function `ggplot()` creates some sort of ‘canvas’ for different layers (called ‘geom_…’) of a painting.

```
data(Orange)
p <- ggplot(Orange,
            aes(x=Tree, y=circumference))
p + geom_boxplot(varwidth=TRUE, fill="tomato") +
  labs(x="Tree", y="circumference in mm") +
  geom_hline(aes(yintercept=mean(circumference)), linetype="dashed") +
  geom_jitter(position=position_jitter(width=0.2), shape = 1)
```



The sub function `aes()` defines the so-called ‘aesthetic mappings’ where as a minimum the involved variables have to be assigned.

ggplot2 does not distinguish between ‘high-level’ and ‘low-level’ and allows for a lot more flexibility than the basic graphics functions.

```
# EXERCISE 3.2: Create again a ggplot object from the 'Orange' data set using the
# variable 'age'. Create then a geom with the (Gaussian) kernel density (try
# 'geom_density' :-) ) of 'age'. Optional: Choose a fancy 'colour' from the
# colors() function.
```

3.6 Visualizing Categorical Data

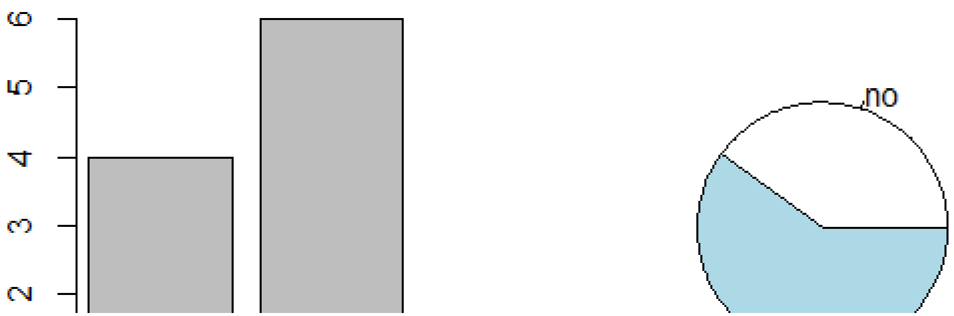
Categorical data (in *R*-speak: the object class is `factor`):

- pie charts and bar charts

```
fac <- factor(c(rep("no",4), rep("yes",6)))
fac
```

```
##  [1] no  no  no  no  yes yes yes yes yes yes
## Levels: no yes
```

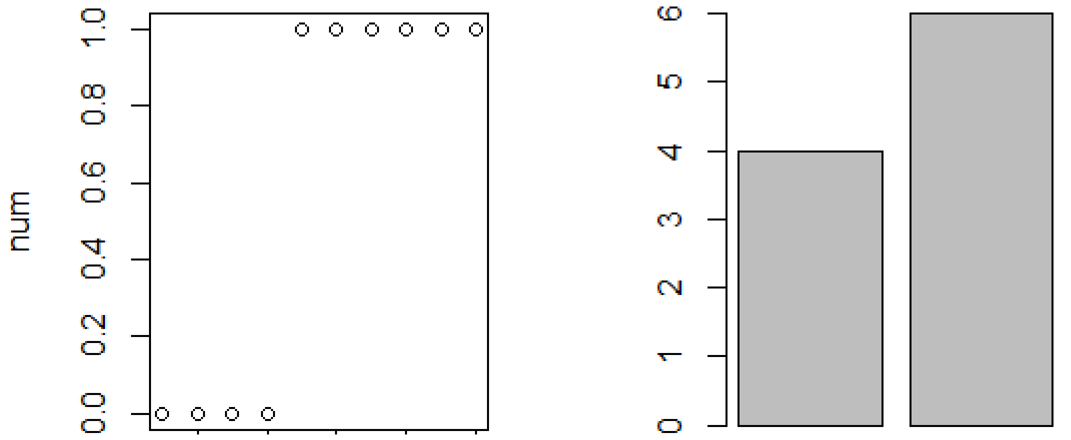
```
layout(matrix(c(1,2),nrow=1))
plot(fac)
pie(table(fac))
```

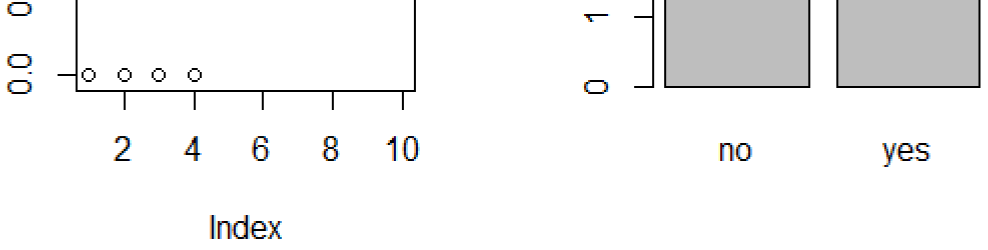


```
num <- c(rep(0,4),rep(1,6))
names(num) <- c(rep("no",4), rep("yes",6))
num
```

```
##   no   no   no   no yes yes yes yes yes yes
##    0    0    0    0   1   1   1   1   1   1
```

```
layout(matrix(c(1,2),nrow=1))
plot(num)
barplot(table(num),names.arg=unique(names(num)))
```



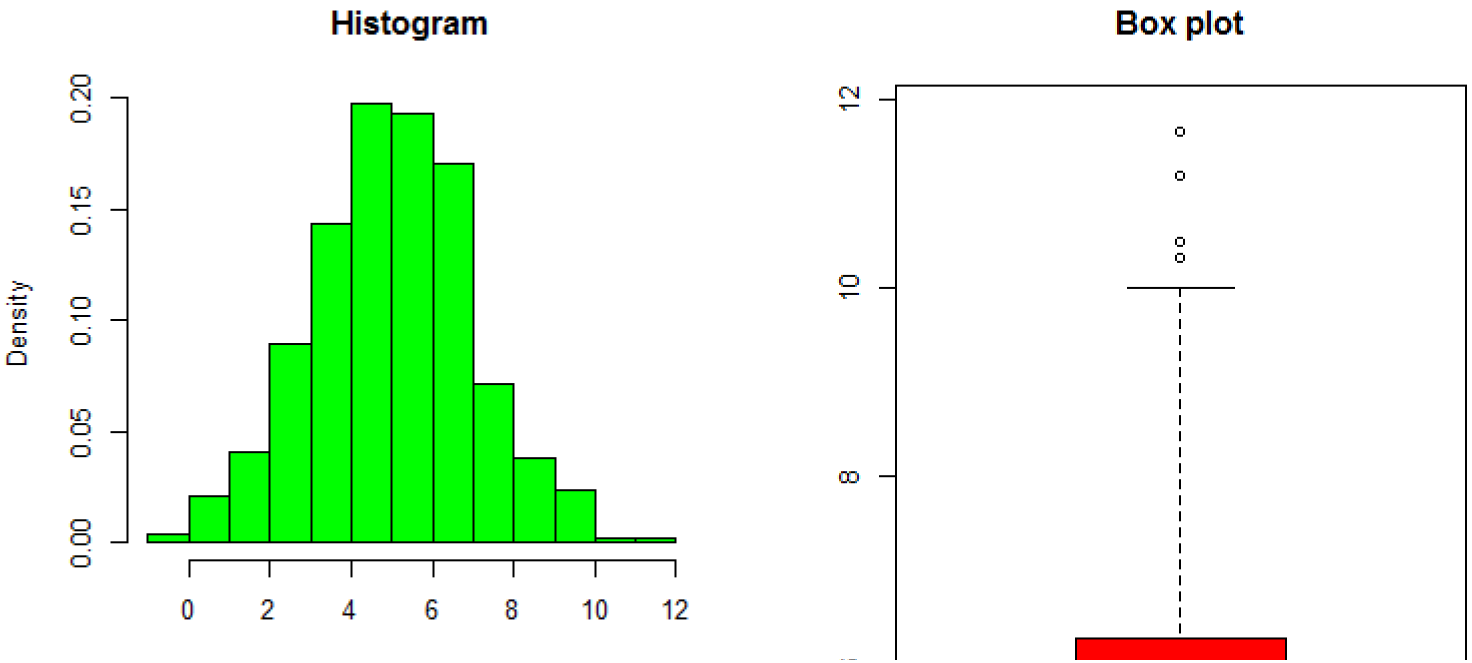


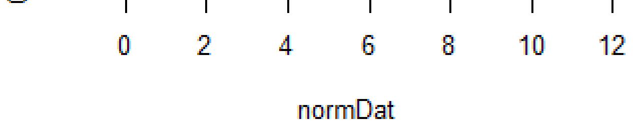
3.7 Visualizing Metric-Scale Data

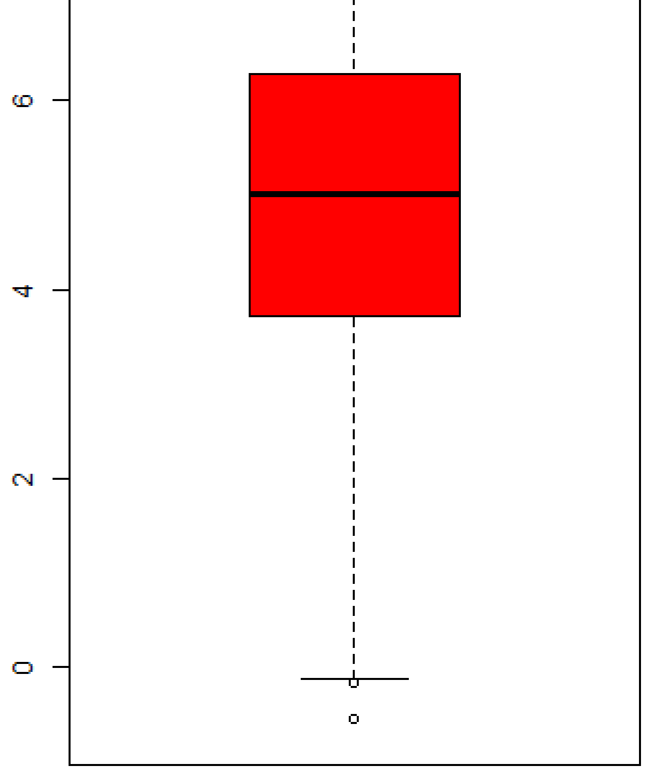
Metric-scale data (in *R*-speak: the object class is `integer` if discrete or `numeric` if continuous):

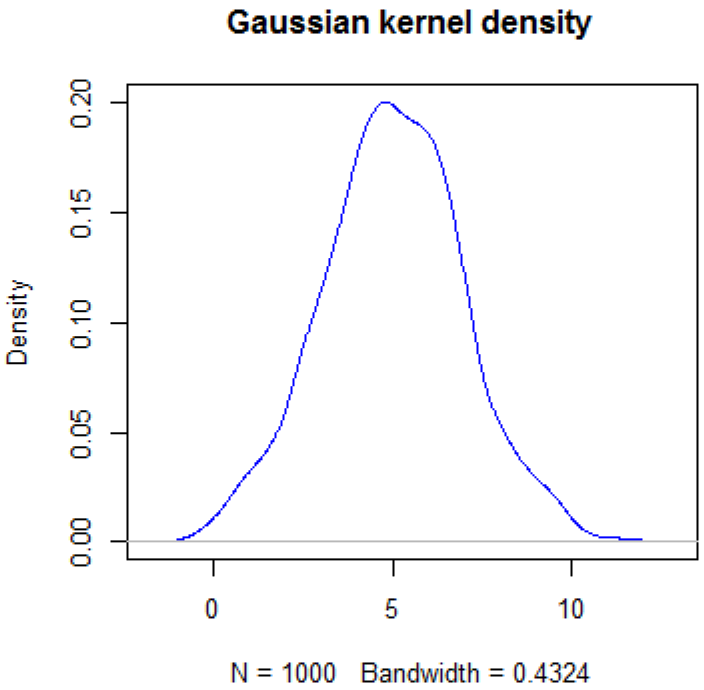
- histograms and kernel density plots

```
normDat <- rnorm(1000, 5, 2)
layout(matrix(c(1,2,3,3),nrow=2))
hist(normDat, freq=FALSE, col="green", main="Histogram")
plot(density(normDat), col="blue", main="Gaussian kernel density")
boxplot(normDat, col="red", main="Box plot")
```









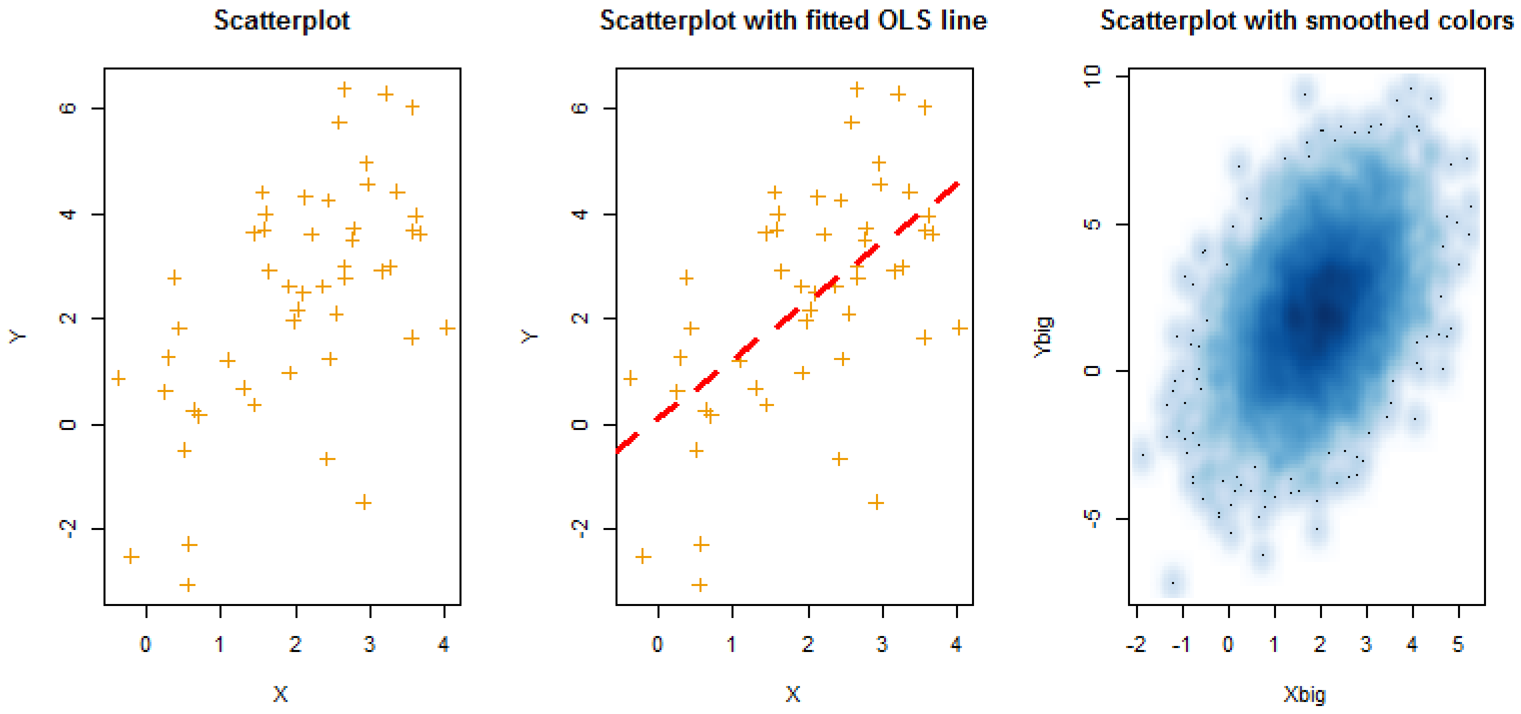
3.8 Bivariate Data

3.8.1 Continuous Data

X and Y both continuous:

- scatterplot

```
X <- rnorm(50, 2, 1)
Y <- X + rnorm(50, 0, 2)
Xbig <- rnorm(5000, 2, 1)
Ybig <- Xbig + rnorm(5000, 0, 2)
layout(matrix(c(1, 2, 3),nrow=1))
plot(X, Y, col = "orange2", pch = 3, main = "Scatterplot")
plot(X, Y, col = "orange2", pch = 3,
  main = "Scatterplot with fitted OLS line")
abline(lm(Y~X), col = "red", lwd = 3, lty = 2)
smoothScatter(Xbig, Ybig, main = "Scatterplot with smoothed colors")
```

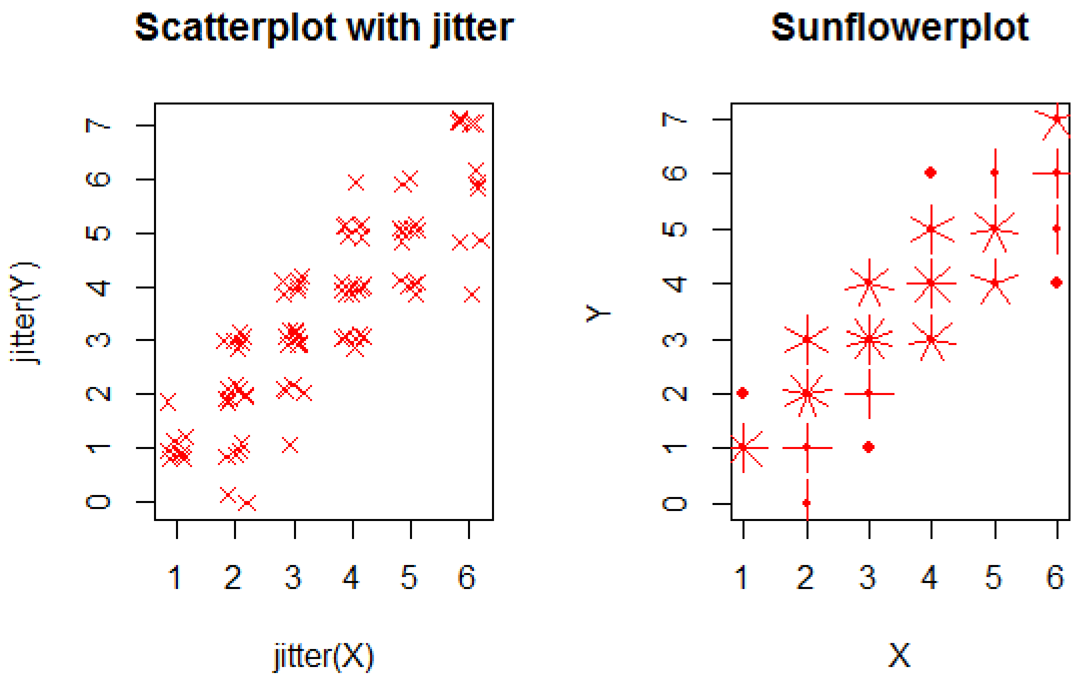


3.8.2 Discrete Data

X and Y both discrete:

- scatterplot with jitter
- sunflowerplot

```
X <- sample(1:6, 100, TRUE)
Y <- X + round(rnorm(100))
layout(matrix(c(1,2),nrow=1))
plot(jitter(X), jitter(Y), col="red",pch=4,
  main="Scatterplot with jitter")
sunflowerplot(X, Y, col="red",main="Sunflowerplot")
```



3.9 Visualization of Three Dimensions

X, Y and Z:

- three-dimensional scatterplot (using the rgl package)

```
X <- rnorm(100, 2, 1)
Y <- X + rnorm(100, 0, 2)
Z <- X + Y + rnorm(100, 0, 1)
library(rgl)
plot3d(X,Y,Z, col="magenta", type="s")
grid3d(side=c("y","z"))
```

Find much more inspiration at: http://www.r-graph-gallery.com/ (http://www.r-graph-gallery.com/)!

4 Loading Data into R

4.1 Loading internal Data

R internal data can either be accessed by loading the package that provides the data first, followed by the function call `data()`

```
library(MASS)
data(cats)
```

or by loading the data set directly without loading the whole package

```
data(prof.salary, package="MBESS")
```

Note that the second option does not give access to the help file (only obvious from running the extracted *R* code) which is why the first option is preferable.

```
# EXERCISE 4.1: Load the data set 'AMSsurvey' from the car package.
# How many men are in the data set?
```

Any previously stored *R* object (or bundle of *R* objects) can be loaded using `load('<path>/<filename>.RData')`.

The function `dim()` returns the numbers of rows and columns of the imported data set.

```
library(MASS)
data(survey)
## checking the dimensions
dim(survey)
```

```
## [1] 237  12
```

```
## or
nrow(survey)
```

```
## [1] 237
```

```
ncol(survey)
```

```
## [1] 12
```

You might want to look at the variable names and some raw data units as well: `head()` and `tail()` display the firts (last) k (default: k=6) cases.

```
## First three rows (default is n=6 rows):
head(survey, n=3)
```

```
##      Sex Wr.Hnd NW.Hnd W.Hnd   Fold Pulse    Clap Exer Smoke Height
## 1 Female   18.5   18.0 Right R on L    92    Left Some Never  173.0
## 2   Male   19.5   20.5  Left R on L   104    Left None Regul  177.8
## 3   Male   18.0   13.3 Right L on R    87 Neither None Occas     NA
##         M.I    Age
## 1   Metric 18.250
## 2 Imperial 17.583
## 3     <NA> 16.917
```

```
## tail() will give you the last lines
# (again default: n=6)
tail(survey, n=4)
```

```
##        Sex Wr.Hnd NW.Hnd W.Hnd   Fold Pulse  Clap Exer Smoke Height    M.I
## 234 Female   18.5   18.0 Right L on R    88 Right Some Never  160.0 Metric
## 235 Female   17.5   16.5 Right R on L    NA Right Some Never  170.0 Metric
## 236   Male   21.0   21.5 Right R on L    90 Right Some Never  183.0 Metric
## 237 Female   17.6   17.3 Right R on L    85 Right Freq Never  168.5 Metric
##        Age
## 234 16.917
## 235 18.583
## 236 17.167
## 237 17.750
```

```
## variable / column names
names(survey)
```

```
##  [1] "Sex"    "Wr.Hnd" "NW.Hnd" "W.Hnd"  "Fold"   "Pulse"  "Clap"  
##  [8] "Exer"   "Smoke"  "Height" "M.I"    "Age"
```

If variable names appear correctly, `str()` will give us information how the data was coded in terms of internal classes. These classes tell us something about the levels of measurement, i.e. nominal (=> factor), ordinal (=> ordered factor) and metric scale (=> integer or numeric).

```
str(survey)
```

```
## 'data.frame':    237 obs. of  12 variables:
##  $ Sex   : Factor w/ 2 levels "Female","Male": 1 2 2 2 2 1 2 1 2 2 ...
##  $ Wr.Hnd: num  18.5 19.5 18 18.8 20 18 17.7 17 20 18.5 ...
##  $ NW.Hnd: num  18 20.5 13.3 18.9 20 17.7 17.7 17.3 19.5 18.5 ...
##  $ W.Hnd : Factor w/ 2 levels "Left","Right": 2 1 2 2 2 2 2 2 2 2 ...
##  $ Fold  : Factor w/ 3 levels "L on R","Neither",..: 3 3 1 3 2 1 1 3 3 3 ...
##  $ Pulse : int  92 104 87 NA 35 64 83 74 72 90 ...
##  $ Clap  : Factor w/ 3 levels "Left","Neither",..: 1 1 2 2 3 3 3 3 3 3 ...
##  $ Exer  : Factor w/ 3 levels "Freq","None",..: 3 2 2 2 3 3 1 1 3 3 ...
##  $ Smoke : Factor w/ 4 levels "Heavy","Never",..: 2 4 3 2 2 2 2 2 2 2 ...
##  $ Height: num  173 178 NA 160 165 ...
##  $ M.I   : Factor w/ 2 levels "Imperial","Metric": 2 1 NA 2 2 1 1 2 2 2 ...
##  $ Age   : num  18.2 17.6 16.9 20.3 23.7 ...
```

The table printed by *R* will give you an overview of:

- the type of the object (in our case: `data.frame`)
- its dimensions
- the variable names (after `$`)
- the coding / levels of measurement
- the levels, if the variables are categorial
- and the first 10 values of every variable

4.2 Importing External Data

Functions and packages for importing data:

- `read.table()` for spreadsheet-type data (ASCII, csv,…), `scan()' and 'readLines()` to import data as a vector
- for data or content in general from websites: `url()` and `file()` (for connections in general) as well as various packages like `RCurl`, `rvest` (currently only on github), `scrapeR`, and many others. Note that functions like `read.table()` support `http:\\` file paths as well.
- `foreign` and `haven` for SPSS, SAS and Stata
- `gdata` (needs perl to be installed), `XLconnect` (Java-based) for Excel files (There are more packages around to import from and export to Excel: `xlsReadWrite`,…and you can always save an Excel sheet in csv-format and use the `read.csv()` -function (a wrapper for `read.table()`)
- Relational database structures: `RODBC` (only for Windows OS – makes use of the open data base connectivity interface), `RMySQL`, `RSQLite`, and `ROracle` (guess what these might be for…)
- The package `XML` for reading from and writing to xml-type documents.

A comprehensive overview can be found at http://cran.r-project.org/doc/manuals/r-release/R-data.pdf (http://cran.r-project.org/doc/manuals/r-release/R-data.pdf).

4.3 How to preserve meta data information in R

An example with SPSS data:

The Problem: R has no ‘meta data window’ (with variable and value label overview)

The Solution: assign the meta data to R objects for later use (e.g. for plot descriptions)

```
library(foreign)
setwd("C:/Uni/Daten")
SPSS2R <- read.spss("cereal.sav", to.data.frame=TRUE,
                    use.value.labels=FALSE)
(varnames <- names(SPSS2R))
```

```
## [1] "agecat"  "gender"  "marital" "active"  "bfast"
```

```
(varlabels <- attributes(SPSS2R)$variable.labels)
```

```
##                 agecat                gender               marital
##         "Age category"              "Gender"      "Marital status"
##                 active                 bfast
##            "Lifestyle" "Preferred breakfast"
```

```
vallabels <- sapply(sapply(SPSS2R, function(x) attributes(x)$value.labels),names)
head(vallabels, n=3)
```

```
## $agecat
## [1] "Over 60"  "46-60"    "31-45"    "Under 31"
##
## $gender
## [1] "Female" "Male"
##
## $marital
## [1] "Married"   "Unmarried"
```

4.4 Saving and loading R objects

R has its own proprietary data format with file ending '.RData'. Using the functions `save()` or `save.image()` you can either save a list of objects or save *every* object which is currently in the Global Environment. The function `load()` allows to restore these objects, even if they had been removed from the Global Environment (or the *R* session was interrupted). This is useful if you want to make sure an important result is stored or if some objects are very large and not needed at the moment. Note that neither saving nor loading requires an assignment!

```
manyRows <- 50000
manyCols <- 1000
MyLargeMatrix <- matrix(nrow=manyRows, ncol=manyCols,rnorm(manyRows*manyCols))
object.size(MyLargeMatrix)
```

```
## 400000200 bytes
```

```
save("manyRows", "manyCols","MyLargeMatrix", file = "LargeObject.RData")
rm("MyLargeMatrix", "manyRows", "manyCols")
objects()
```

```
##  [1] "a"           "aList"       "anscombe"    "arr1"        "b"
##  [6] "cats"        "combiMat"    "combivec"    "D"           "DAT1"
## [11] "fac"         "greenPerc"   "i"           "model_ansc"  "normDat"
## [16] "num"         "Orange"      "p"           "prof.salary" "SPSS2R"
## [21] "survey"      "typ"         "vallabels"   "varlabels"   "varnames"
## [26] "x"           "X"           "Xbig"        "y"           "Y"
## [31] "y1"          "y2"          "y3"          "y4"          "y5"
## [36] "y6"          "y7"          "y8"          "Ybig"        "Z"
```

```
load("LargeObject.RData")
objects()
```

```
##  [1] "a"             "aList"         "anscombe"      "arr1"
##  [5] "b"             "cats"          "combiMat"      "combivec"
##  [9] "D"             "DAT1"          "fac"           "greenPerc"
## [13] "i"             "manyCols"      "manyRows"      "model_ansc"
## [17] "MyLargeMatrix" "normDat"       "num"           "Orange"
## [21] "p"             "prof.salary"   "SPSS2R"        "survey"
## [25] "typ"           "vallabels"     "varlabels"     "varnames"
## [29] "x"             "X"             "Xbig"          "y"
## [33] "Y"             "y1"            "y2"            "y3"
## [37] "y4"            "y5"            "y6"            "y7"
## [41] "y8"            "Ybig"          "Z"
```

```
rm("MyLargeMatrix")
```

5 Data aggregation

5.1 The `apply`-family

- `apply()` …applies a function to rows or columns to ‘rectangular’ objects (or – if the object is an array – to rows, columns and other margins as well)
- `lapply()` applies a function to list objects and returns a list
- `sapply()` applies a function to list object and returns whatever it thinks the user would like to get
- `tapply()` applies a function to a vector that is segmented by another vector
- And there are even more:

```
apropos("apply")
```

```
##  [1] ".mapply"    "apply"      "dendrapply" "eapply"     "gapply"
##  [6] "kernapply"  "lapply"     "mapply"     "rapply"     "sapply"
## [11] "tapply"     "Tree_apply" "vapply"
```

`apply()` and its cousins help to avoid copy&paste code, and even loops. apply allows to aggregate over rows (set parameter `MARGIN = 2`), columns (set parameter `MARGIN = 1`), or even higher dimensions (e.g. within arrays).

Example: A table for each variable

```
library(car)
apply(AMSsurvey, 2, table)
```

```
## $type
## 
## I(Pr) I(Pu)    II   III    IV    Va 
##     4     4     4     4     4     4 
## 
## $sex
## 
## Female   Male 
##     12     12 
## 
## $citizen
## 
## Non-US     US 
##     12     12 
## 
## $count
## 
##  12  14  20  25  28  29  32  34  35  39  47  50  53  54  71  79  87  89 
##   1   1   1   1   1   1   1   1   1   1   2   1   1   1   1   1   1   1 
##  96 105 122 130 132 
##   1   1   1   1   1 
## 
## $count11
## 
##  17  21  22  26  27  28  30  32  40  42  53  55  56  61  63  71  82  89 
##   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1 
## 115 116 136 148 153 161 
##   1   1   1   1   1   1
```

Example: The somewhat obscure `rapply()` `rapply` works recursive, and is useful to bypass list structures.

```
listA <- list(A1=matrix(1:4,nrow=2),A2=matrix(5:8,nrow=2))
listB <- list(B1=matrix(9:12,nrow=2),B2=matrix(13:16,nrow=2))
listAll <- list(listA, listB)
sapply(listAll, mean)
```

```
## [1] NA NA
```

```
rapply(listAll, mean)
```

```
##   A1   A2   B1   B2
##  2.5  6.5 10.5 14.5
```

```
#EXERCISE 5.1: Load the data set 'cats' from the MASS package.
#Calculate the average heart weight and the average body weight for cats weighing at least 3 kg.
```

5.2 Subgroup Functions

5.2.1 Subsetting the data

The `subset()` function extracts a subsample of the data set based on a logical argument. Of course, you can get the same result via (a bit elaborated) indexing. Example: Suppose we want to extract the female PhDs only from the AMSsurvey data and keep the `type` and the `count` variable.

```
subset(AMSsurvey, sex == "Female", select=c(type, count))
```

```
##      type count
## 2  I(Pu)    35
## 4  I(Pr)    20
## 6      II    47
## 8     III    32
## 10     IV    54
## 12     Va    14
## 14 I(Pu)    29
## 16 I(Pr)    25
## 18     II    50
## 20    III    39
## 22     IV   105
## 24     Va    12
```

5.2.2 Assigning a function to subsets

The `aggregate()` function takes it one step further by assigning a function to subgroups. This function is equivalent to its namesake in SPSS. Suppose we now want to calculate the number of PhDs from the AMSsurvey data averaged over gender and citizenship.

```
aggregate(count~sex+citizen, data=AMSsurvey, mean)
```

```
##      sex citizen    count
## 1 Female  Non-US 43.33333
## 2   Male  Non-US 83.50000
## 3 Female      US 33.66667
## 4   Male      US 77.83333
```

The first parameter is a formula object (assignment via `~`), i.e. in this example the variable `count` is conditioned on `sex` *and* `citizen`, and the mean is calculated within the subcells defined by these two variables.

```
#EXERCISE 5.2: Install the 'VIM' package, and load it into the library.
#Load the 'sleep' data. Calculate the following:
#* the mean of the variable 'Span' (beware of missing values!)
#* the mean of the first three variables
#* the mean of the variable 'BodyWgt' conditioned on 'Exp' and 'Danger'
# (assign the result to a new object)
```

6 Linear Regression

Linear models are typically fitted using Ordinary Least Squares (OLS).

6.1 Diagnostics before estimating the model

Looking at bivariate correlations and scatterplots is a good start to pre-select potential covariates, if you choose a exploratory approach to define your model:

```
data(Salaries, package="car")
names(Salaries)
```

```
## [1] "rank"          "discipline"    "yrs.since.phd" "yrs.service"
## [5] "sex"           "salary"
```

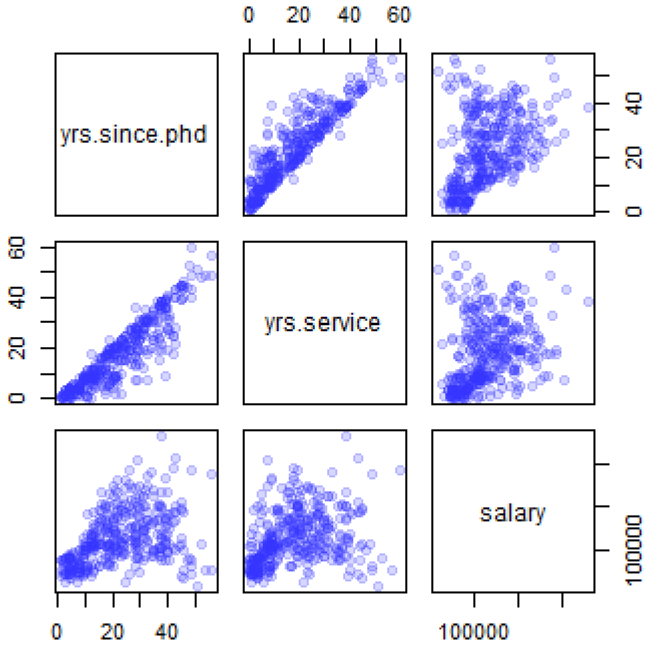
```
sapply(Salaries, class)
```

```
##          rank    discipline yrs.since.phd   yrs.service           sex
##      "factor"      "factor"     "integer"     "integer"      "factor"
##        salary
##     "integer"
```

```
nonFac <- sapply(Salaries, class) != "factor"
cor(Salaries[, nonFac], use="pairwise.complete.obs")
```

```
##               yrs.since.phd yrs.service    salary
## yrs.since.phd     1.0000000   0.9096491 0.4192311
## yrs.service       0.9096491   1.0000000 0.3347447
## salary            0.4192311   0.3347447 1.0000000
```

```
pairs(Salaries[, nonFac], col=rgb(0.2,0.2,1,0.2),pch=19)
```



The `vcd` package offers many options to plot associations among categorical variables (in this example we check for dependencies among covariates to prevent multicollinearity and to account for interactions).

```
if (!require("vcd")) install.packages("vcd")
```

```
## Loading required package: vcd
## Loading required package: grid
```

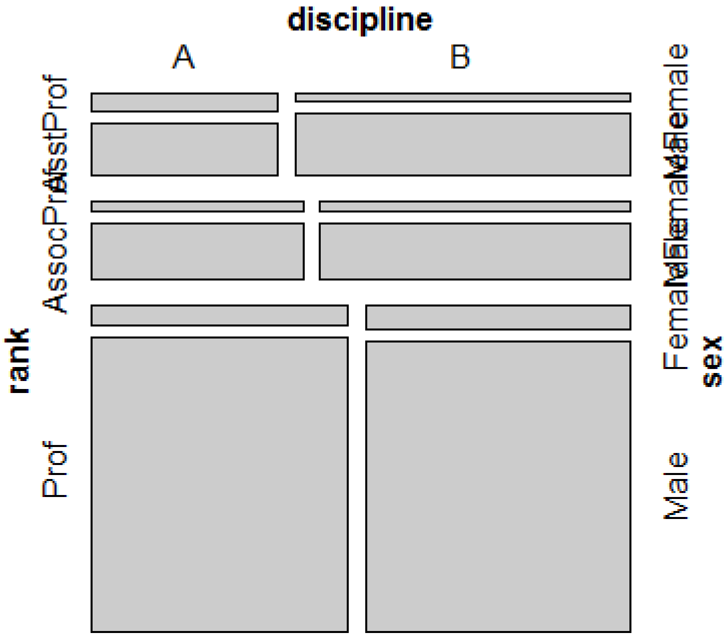
```
library(vcd)
table(Salaries[, !nonFac])
```

```
## , , sex = Female
## 
##             discipline
## rank          A   B
##   AsstProf    6   5
##   AssocProf   4   6
##   Prof        8  10
## 
## , , sex = Male
## 
##             discipline
## rank          A   B
##   AsstProf   18  38
##   AssocProf  22  32
##   Prof      123 125
```

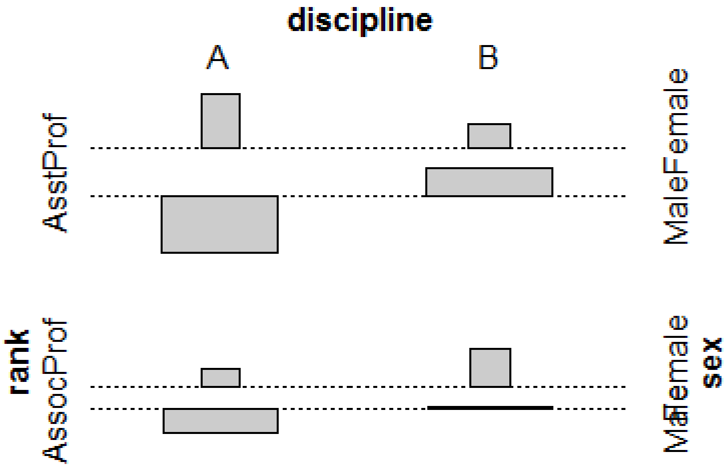
```
structable(table(Salaries[, !nonFac]))
```

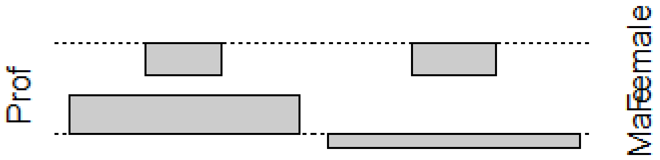
```
##                   discipline   A   B
## rank      sex                       
## AsstProf  Female               6   5
##           Male                18  38
## AssocProf Female               4   6
##           Male                22  32
## Prof      Female               8  10
##           Male               123 125
```

```
mosaic(table(Salaries[, !nonFac]), col="blue")
```



```
assoc(table(Salaries[, !nonFac]), col="blue")
```





6.2 The `lm()` object

- `lm()` estimates a linear model
- The most basic specification is `lm(formula, data)`
- `data` can be either a data.frame or a matrix
- `formula` needs a bit more explanation…

6.2.1 The `formula` object

`formula` typically looks something like this: `y~x1+x2`

- the equal sign is replaced by ´~` (why?)
- as default the intercept is included (omit using `0 + ...` or `... - 1`)
    - `-` is used to omit specified terms in general
- `:` is used for (factor) interactions
- `*` is used for factor crossings (including the marginal effect)
- `%in\%` indicates a nested term
- `~ .` indicates R that you want all variables in the data set to be included
- arithmetic expression such as `log()` are allowed
- to use the power of a variable or the sum/product/ratio of two variables you need the 'as.is'-function `I()` (e.g. `(x1*x2)` translates to `x1+x2+x1:x2`, whereas `I(x1*x2)` yields the product of x_1 and x_2)

6.2.2 Some further parameters

- `data`: Typically a data frame or – if the `subset` argument of `lm()` is specified – selected rows of a data frame
- `na.action` specifies how to deal with missing values (the default being listwise deletion)
- setting `x` and `y` to `TRUE` provides the design matrix and the response variable as elements of the output object

6.3 OLS regression using the lm()-function – Introductory example

```
MyModel <- lm(salary~sex+yrs.service, data=Salaries)
MyModel
```

```
##
## Call:
## lm(formula = salary ~ sex + yrs.service, data = Salaries)
##
## Coefficients:
## (Intercept)      sexMale  yrs.service
##     92356.9       9071.8        747.6
```

```
summary(MyModel)
```

```
##
## Call:
## lm(formula = salary ~ sex + yrs.service, data = Salaries)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -81757 -20614  -3376  16779 101707
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  92356.9     4740.2  19.484  < 2e-16 ***
## sexMale       9071.8     4861.6   1.866   0.0628 .
## yrs.service    747.6      111.4   6.711 6.74e-11 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 28490 on 394 degrees of freedom
## Multiple R-squared:  0.1198, Adjusted R-squared:  0.1154
## F-statistic: 26.82 on 2 and 394 DF,  p-value: 1.201e-11
```

```
MyModel2 <- lm(salary~sex*yrs.service, data=Salaries)
MyModel2
```

```
##
## Call:
## lm(formula = salary ~ sex * yrs.service, data = Salaries)
##
## Coefficients:
##         (Intercept)              sexMale          yrs.service
##             82068.5              20128.6               1637.3
## sexMale:yrs.service
##              -931.7
```

```
summary(MyModel2)
```

```
## 
## Call:
## lm(formula = salary ~ sex * yrs.service, data = Salaries)
## 
## Residuals:
##    Min     1Q Median     3Q    Max 
## -80381 -20258  -3727  16353 102536 
## 
## Coefficients:
##                      Estimate Std. Error t value Pr(>|t|)    
## (Intercept)           82068.5     7568.7  10.843  < 2e-16 ***
## sexMale               20128.6     7991.1   2.519  0.01217 *  
## yrs.service            1637.3      523.0   3.130  0.00188 ** 
## sexMale:yrs.service    -931.7      535.2  -1.741  0.08251 .  
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
## 
## Residual standard error: 28420 on 393 degrees of freedom
## Multiple R-squared:  0.1266, Adjusted R-squared:  0.1199 
## F-statistic: 18.98 on 3 and 393 DF,  p-value: 1.622e-11
```

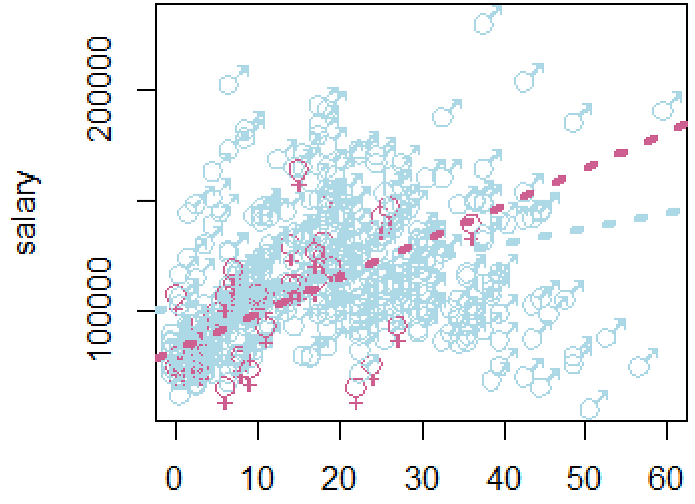
```
#EXERCISE 6.1: Interpret the regression coefficients of `MyModel1` and `MyModel2`.
#Can you roughly predict, after how many years of service (according to `MyModel2`) female salaries will #overtake
their male counterparts? Plot the regression lines for `MyModel2` both for males and females #into one single plot
! (hint: beware -- very difficult exercise! `abline()` can handle lm-objects, but #you need two different models)

modelF <- lm(salary~yrs.service, data=Salaries,
             subset = which(Salaries$sex == "Female"))
modelF$coefficients
```

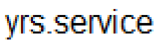
```
## (Intercept) yrs.service 
##    82068.51     1637.30
```

```
modelM <- lm(salary~yrs.service, data=Salaries,
             subset = which(Salaries$sex == "Male"))
coef(modelM)
```

```
## (Intercept) yrs.service
## 102197.1345    705.5634
```

```
attach(Salaries)

plot(yrs.service, salary, type="n")
text(yrs.service, salary, labels=ifelse(sex == "Female",
                                        "\\VE", "\\MA"),
     family="HersheySerif", cex=1.5,
     col=ifelse(sex == "Female",
                "hotpink3", "lightblue"))
abline(modelF, lwd=4, lty=3, col="hotpink3")
abline(modelM, lwd=4, lty=3, col="lightblue")
```





```
detach(Salaries)
```

6.4 Namespaces of the lm-object and its summary

```
names(MyModel)
```

```
##  [1] "coefficients"  "residuals"     "effects"       "rank"
##  [5] "fitted.values" "assign"        "qr"            "df.residual"
##  [9] "contrasts"     "xlevels"       "call"          "terms"
## [13] "model"
```

```
names(summary(MyModel))
```

```
##  [1] "call"          "terms"         "residuals"     "coefficients"
##  [5] "aliased"       "sigma"         "df"            "r.squared"
##  [9] "adj.r.squared" "fstatistic"    "cov.unscaled"
```

6.4.1 Coefficients

There are (at least) three ways to access the vector of the parameter estimates

```
rbind(MyModel$coefficients,
summary(MyModel)$coefficients[,1],
coef(MyModel))
```

```
##      (Intercept) sexMale yrs.service
## [1,]    92356.95  9071.8    747.6121
## [2,]    92356.95  9071.8    747.6121
## [3,]    92356.95  9071.8    747.6121
```

The ‘coefficients’ element in the summary is a matrix containing the estimates, SE’s, t- and p-values:

```
summary(MyModel)$coefficients
```

```
##                Estimate Std. Error   t value     Pr(>|t|)
## (Intercept) 92356.9467   4740.188 19.483816 1.072205e-59
## sexMale      9071.8000   4861.644  1.865994 6.278500e-02
## yrs.service   747.6121    111.396  6.711301 6.735837e-11
```

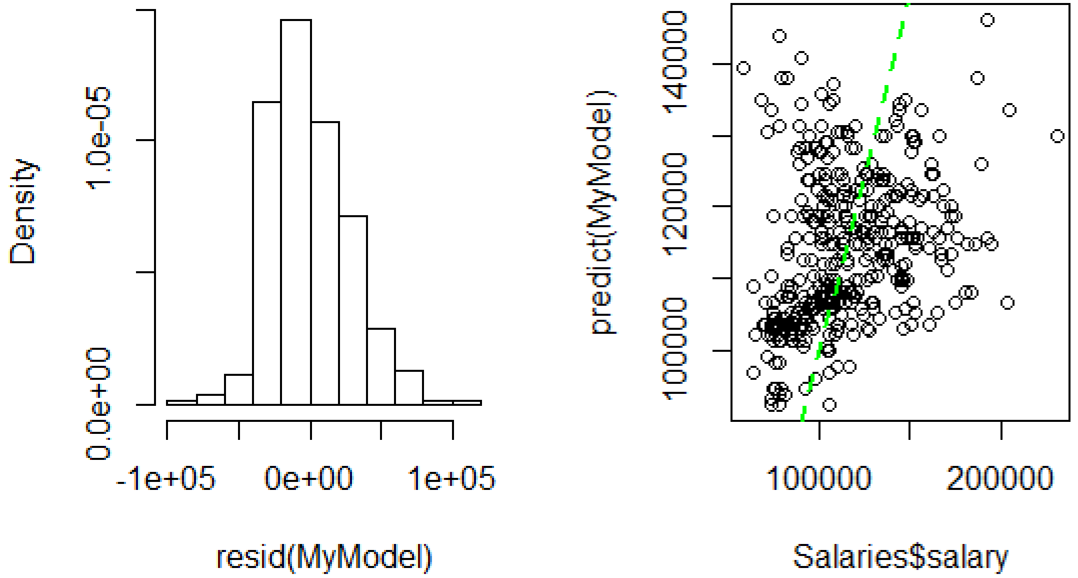
Digression: How does R ‘know’, what the output should look like?

- Every object belongs to a class, and you can define methods for these classes using the logic `<method>.<class>` (for S3 classes).
- `print.lm` defined the output when we called the lm-object ‘MyModel’ – which is implicit for `print(MyModel).
- Usually, `summary.<class>` is a method, too, but `summary.lm` is also a class!
- And, of course, there is a method `plot.lm` …

6.5 Plotting lm-objects

6.5.1 Fitted vs. residual values (manually)

```
## extract and plot the fitted values residuals
layout(matrix(c(1,2), nrow=1))
hist(resid(MyModel), freq=FALSE, main=expression(hat(u)))
plot(Salaries$salary,predict(MyModel))
abline(a=0,b=1, lwd=2, lty=2, col="green")
```



6.5.2 Cook's Distance and Leverage

Cook's distance D_j for observation j determines the influence of observation j on the estimated regression coefficients (of the all-observation model). It is defined as

$$D_j = \frac{\sum_{i=1}^{n-1} \left(\hat{y}_{i(-j)} - \hat{y}_i\right)^2}{p \cdot MSE},$$

where $\hat{y}_{i(-j)}$ is the predicted value of y_i from a regression without observation y_j, $\hat{y}_i$ is the predicted value of y_i from the regression model estimation with all observations, p is the number of parameters in the model and MSE is the *Mean Squared Error* $n^{-1} \sum_{i=1}^{n} \left(\hat{y}_i - y_i\right)^2$.

A slightly different approach that does not take the dependent variable Y into consideration is the concept of *leverages*. If we re-write the predicted values $\hat{y} = X\hat{\beta}$ as Hy, where $H = X(X^T X)^{-1} X$, we can define the main diagonal elements of H as leverages h_{ii}, i.e.

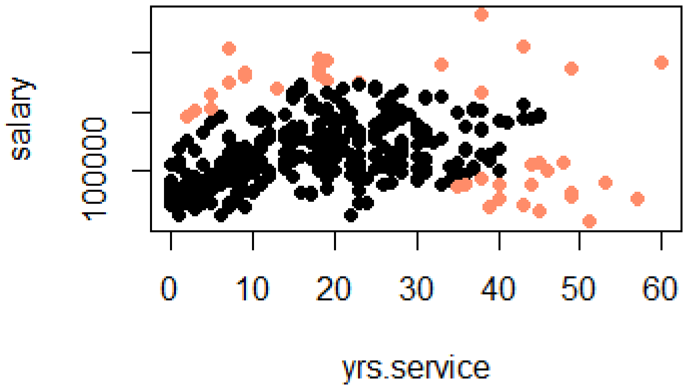
$$H = \begin{pmatrix} h_{11} & \cdots & h_{1n} \\ \vdots & \ddots & \vdots \\ h_{n1} & \cdots & h_{nn} \end{pmatrix}.$$

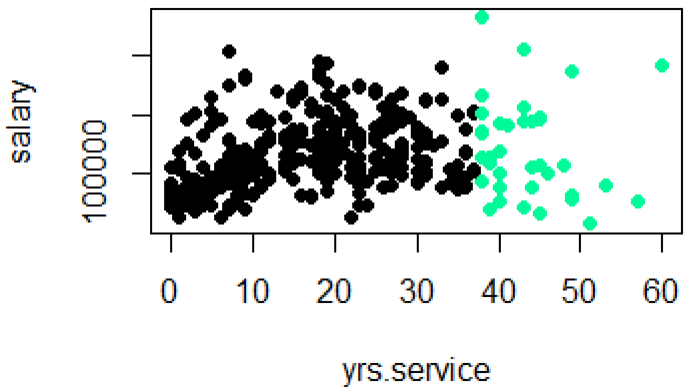
The basic idea is best visualized in a data situation with only one covariate:

```
MyMiniModel <- lm(salary~yrs.service, data=Salaries)
cook <- cooks.distance(MyMiniModel)
summary(cook)
```

```
##      Min.   1st Qu.    Median      Mean   3rd Qu.      Max. 
## 0.0000000 0.0002077 0.0009202 0.0031750 0.0026970 0.0818400
```

```
h <- lm.influence(MyMiniModel)$hat
attach(Salaries)
layout(1:2)
plot(yrs.service,salary, pch=19,
     col=ifelse(cook > quantile(cook,0.9),"salmon1","black"))
plot(yrs.service,salary, pch=19,
     col=ifelse(h > quantile(h,0.9),"mediumspringgreen","black"))
```

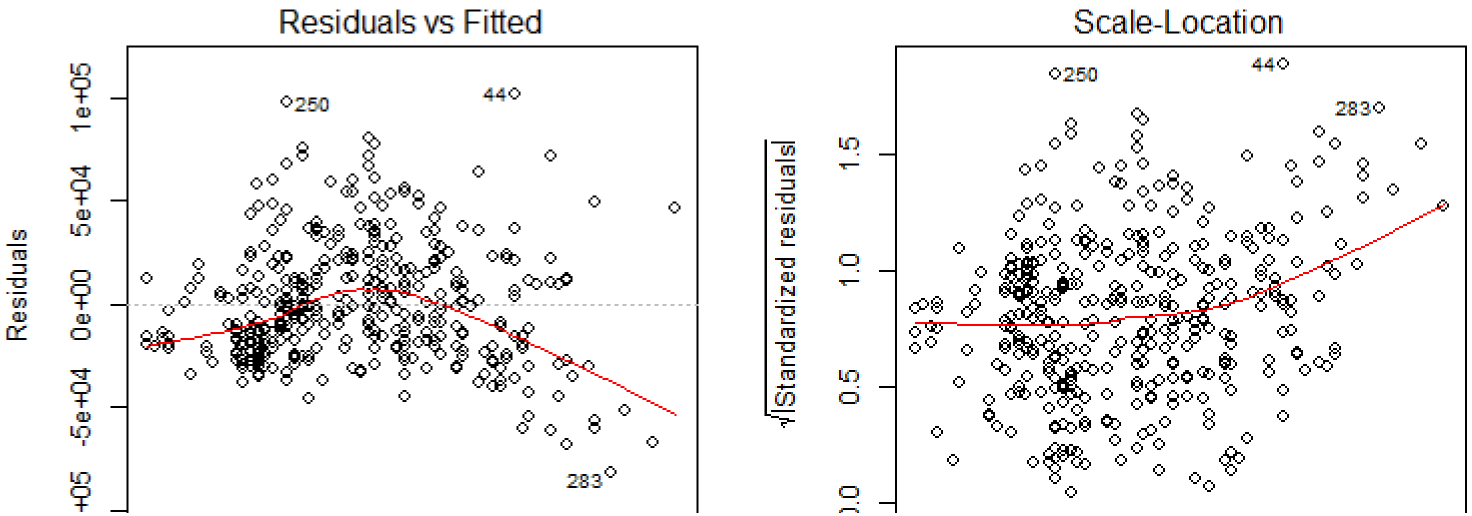


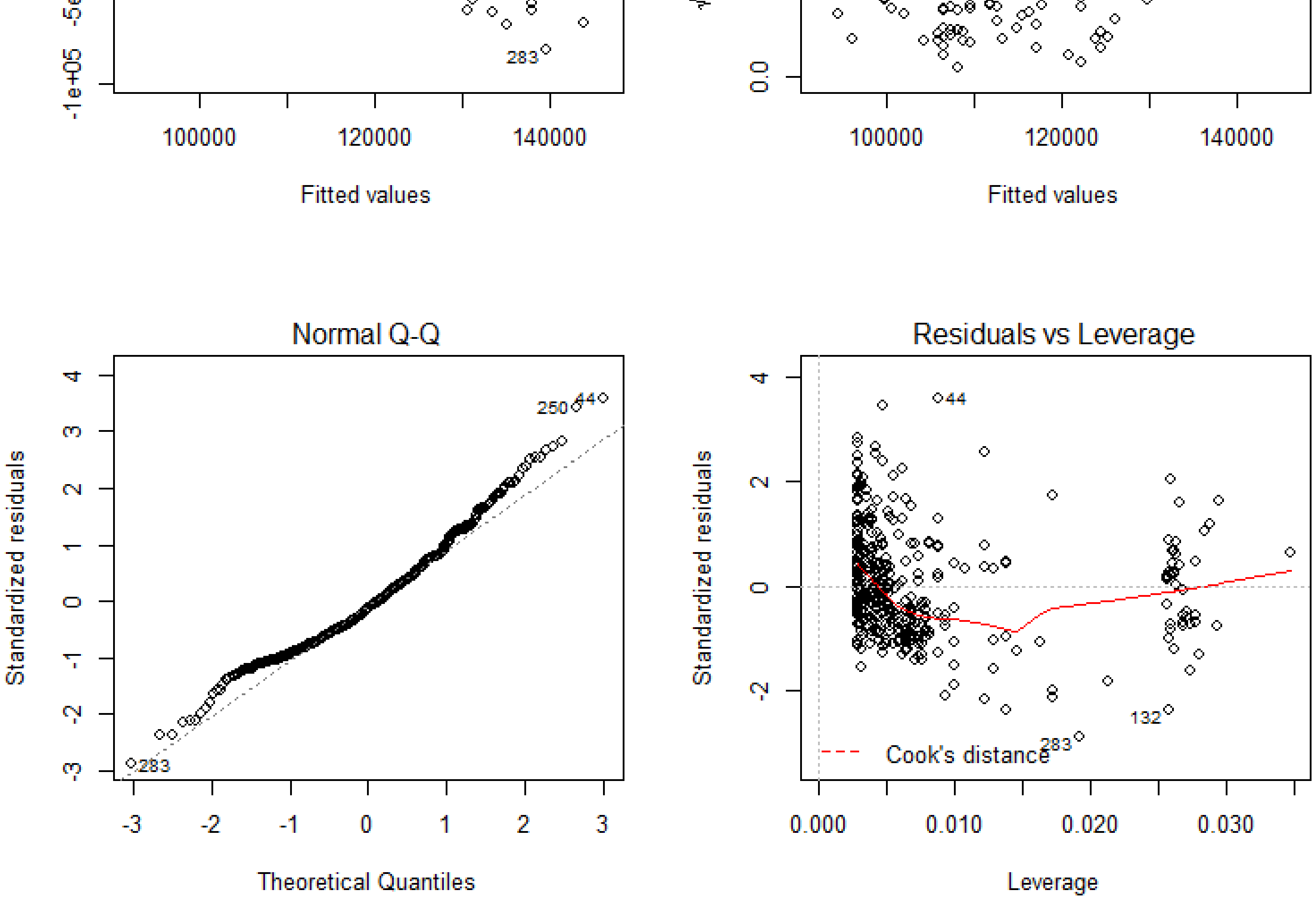


```
detach(Salaries)
```

6.5.3 Using the `plot.lm` methods

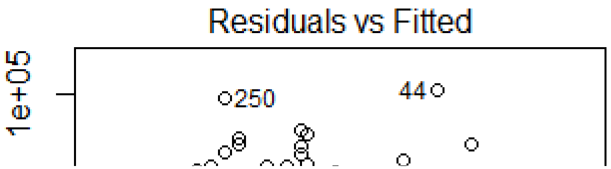
```
layout(matrix(1:4, nrow=2))
plot(MyModel)
```

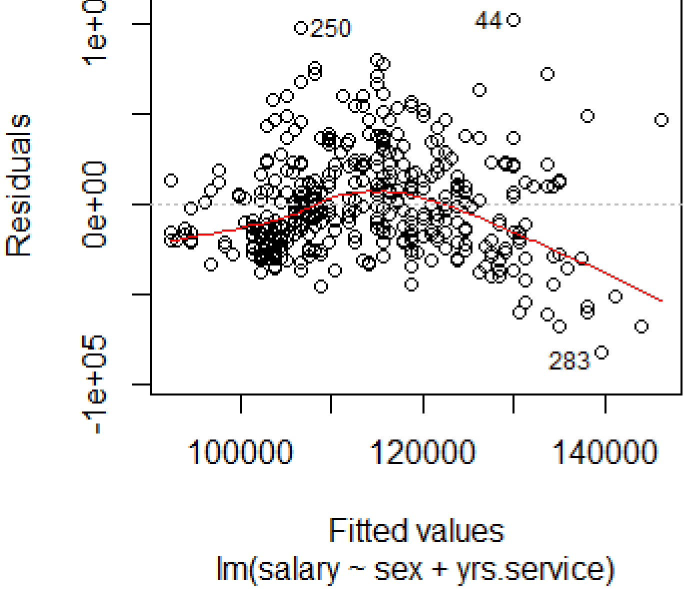




6.5.3.1 Fitted residuals plot

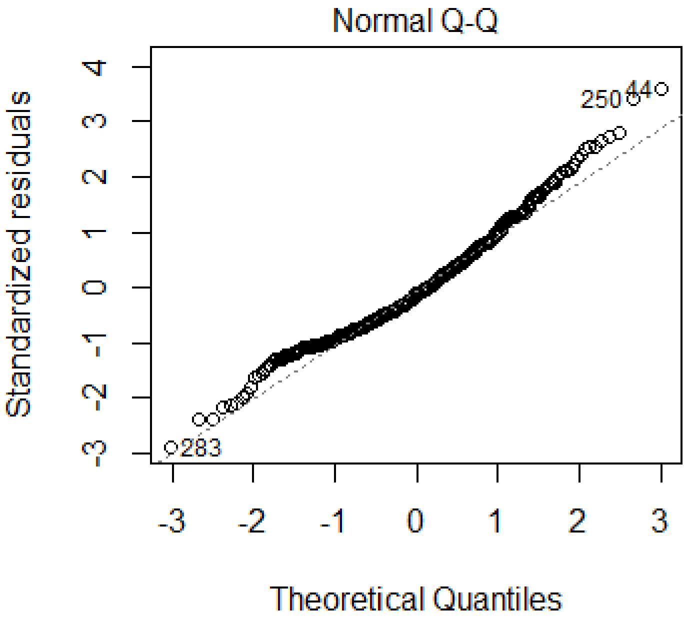
```
plot(MyModel, which=1)
```

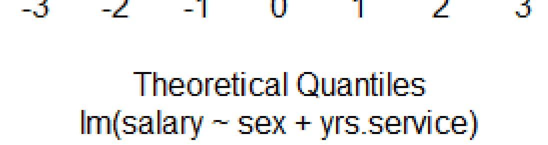




6.5.3.2 Q-Q plot

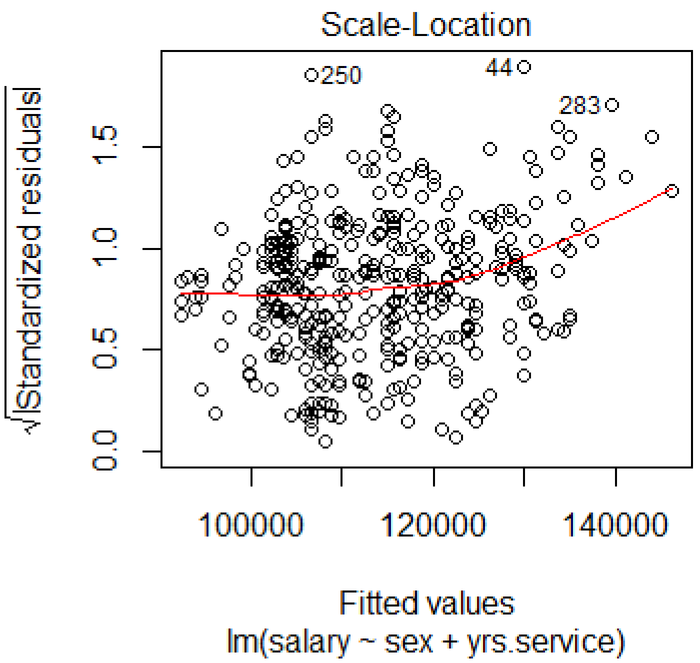
```
plot(MyModel, which=2)
```





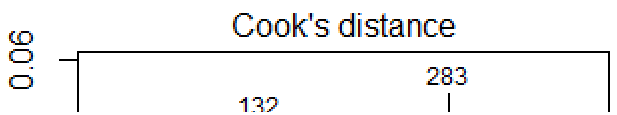
6.5.3.3 S-L plot

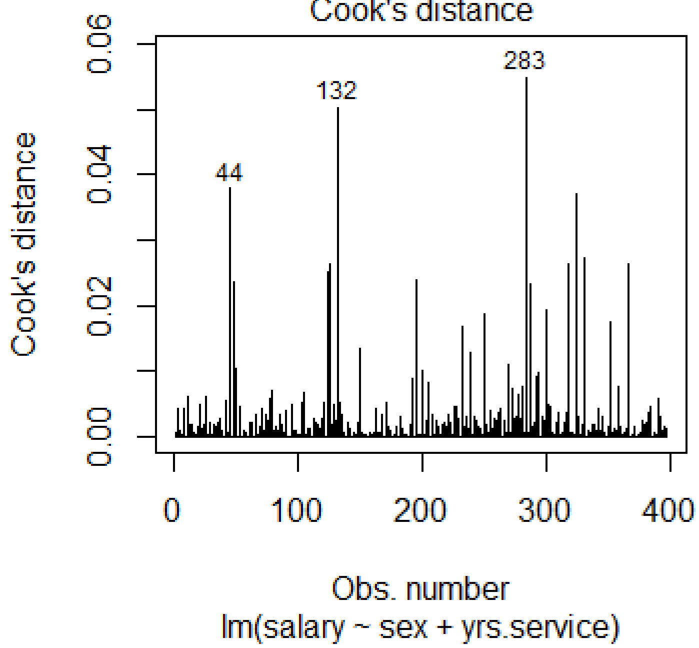
```
plot(MyModel, which=3)
```



6.5.3.4 Cook's distance plot

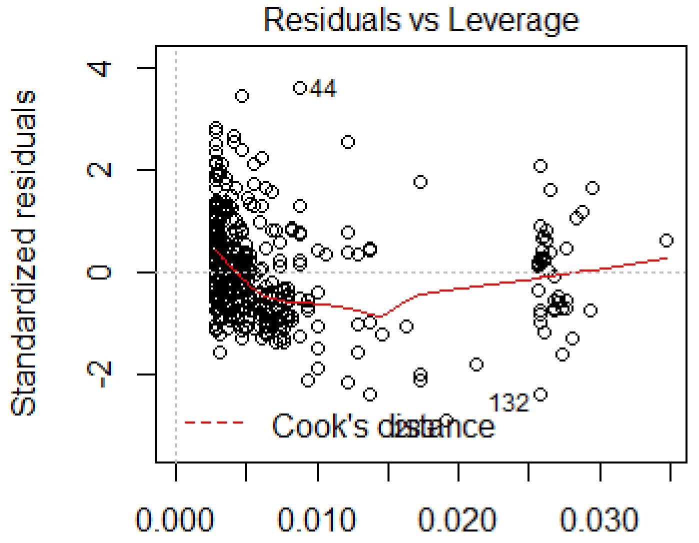
```
plot(MyModel, which=4)
```





6.5.3.5 Residual-Leverage plot

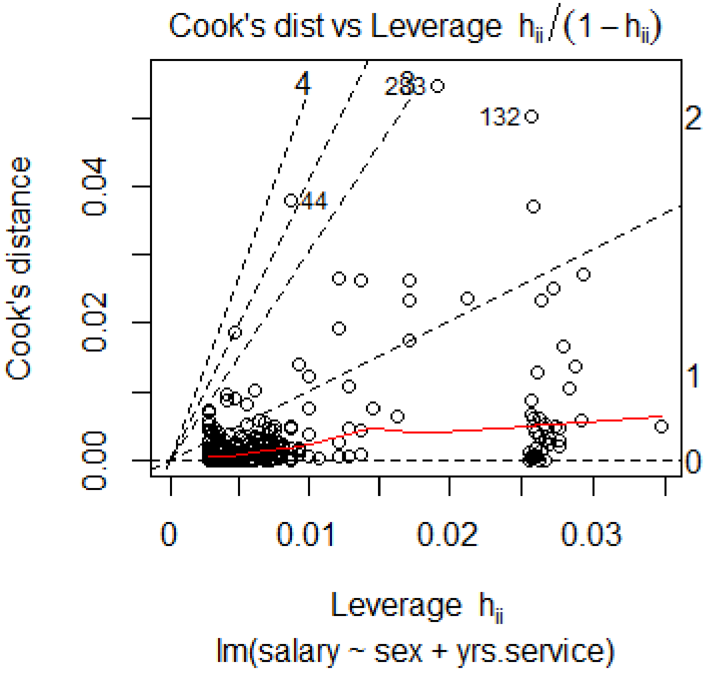
```
plot(MyModel, which=5)
```



Leverage
lm(salary ~ sex + yrs.service)

6.5.3.6 Cook's distances against leverage/(1-leverage)

```
plot(MyModel, which=6)
```



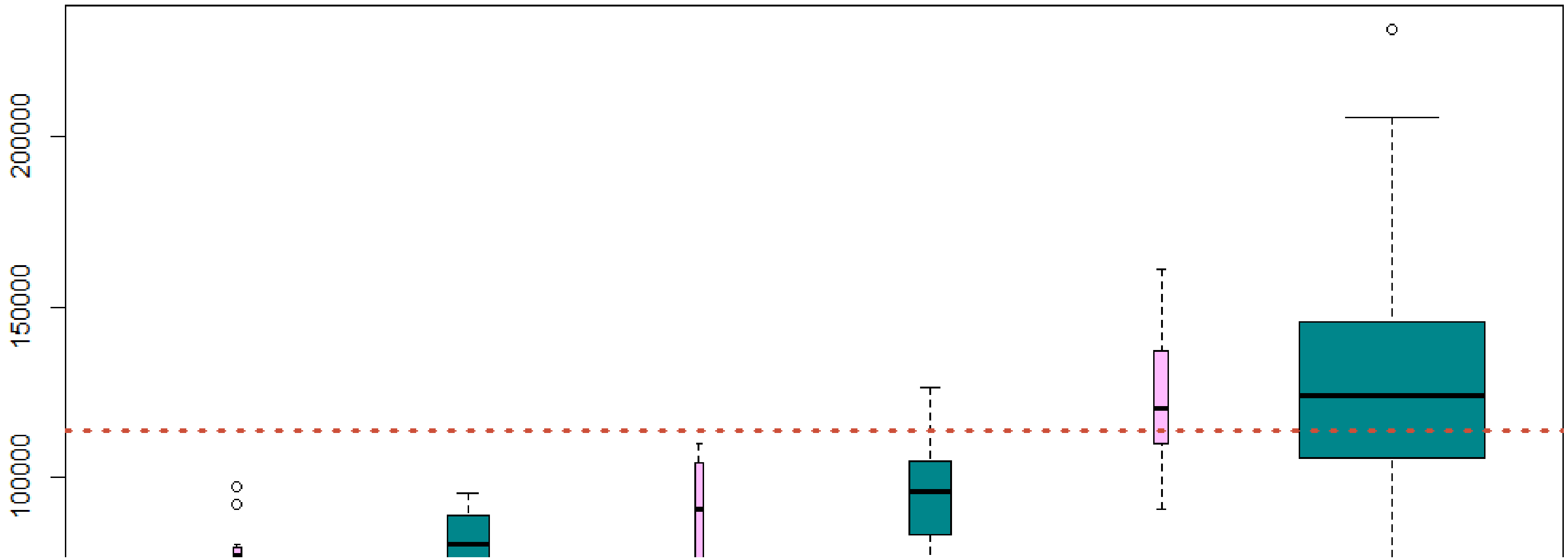
6.6 ANalysis of VAriance (ANOVA)

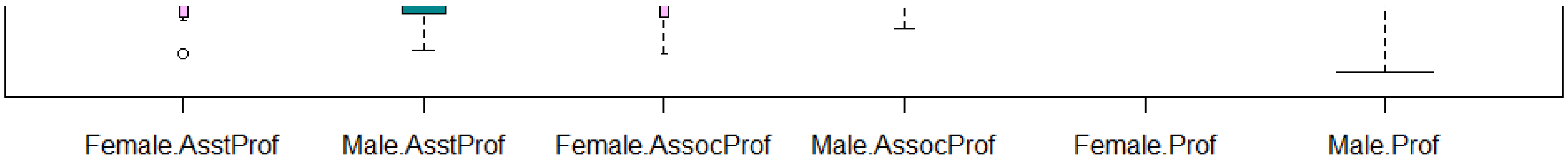
ANOVA is a different way to look at data. It can be conducted conditioning on the categories of one covariate (one-way anova) or several covariates. Typically an *F*-test is performed which tests against a 'null' model that does not condition on the categories of the covariates. The following example describes an (artifically created) one-way ANOVA model:

```
AnovaMod <- lm(salary~sex:rank, data=Salaries)
anova(AnovaMod)
```

```
## Analysis of Variance Table
##
## Response: salary
##              Df     Sum Sq    Mean Sq F value    Pr(>F)
## sex:rank    5 1.4412e+11 2.8823e+10  51.417 < 2.2e-16 ***
## Residuals 391 2.1918e+11 5.6057e+08
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
boxplot(salary~sex:rank, data=Salaries, width=as.numeric(table(Salaries$sex,Salaries$rank)), col=c("plum1","turquo
ise4"))
abline(h=mean(Salaries$salary),lty=3, lwd=3, col="tomato3")
```





Alternatively, one can specify the `boxplot()` argument `varwidth=TRUE` to get width proportional to the square of the number of observations per group.

However, in case you have more than one factor it makes sense to investigate main effects and interactions separately, rather than combining them into one covariate. The corresponding ANOVA model for any two covariates is specified by

$$y_{ijk} = \mu + \gamma_i + \beta_j + (\gamma\beta)_{ij} + \epsilon_{ijk},$$

where μ denotes the overall mean, γ_i the parameter for the i-th category of the first covariate, β_j the parameter for the j-th category of the second covariate, $(\gamma\beta)_{ij}$ their interaction, and y_{ijk} denotes the k-th observation in cell ij.

```
anova(lm(salary~sex*rank, data=Salaries))
```

```
## Analysis of Variance Table
##
## Response: salary
##            Df    Sum Sq   Mean Sq  F value    Pr(>F)
## sex         1 6.9800e+09 6.9800e+09  12.4515 0.0004673 ***
## rank        2 1.3709e+11 6.8546e+10 122.2787 < 2.2e-16 ***
## sex:rank    2 4.3603e+07 2.1802e+07   0.0389 0.9618588
## Residuals 391 2.1918e+11 5.6057e+08
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Note that it is possible to investigate the effect on several outcome variables, and, consequently, this multivariate analysis of variance is referred to as *MANOVA*.

6.7 Other useful functions for lm-objects

```
## structure of original data
head(Salaries, n=3)
```

```
##       rank discipline yrs.since.phd yrs.service  sex salary
## 1     Prof          B            19          18 Male 139750
## 2     Prof          B            20          16 Male 173200
## 3 AsstProf          B             4           3 Male  79750
```

```
## new hypothetical data
(newCovar <- data.frame(rank=sample(Salaries$rank,2),
                        discipline=sample(Salaries$discipline,2),
                        yrs.since.phd=c(20, 10),yrs.service=c(18,5),
                        sex=rep('Male',2)))
```

```
##   rank discipline yrs.since.phd yrs.service  sex
## 1 Prof          B            20          18 Male
## 2 Prof          B            10           5 Male
```

```
## predicted salary for new covariates
predict(MyModel, newdata=newCovar)
```

```
##        1        2
## 114885.8 105166.8
```

```
## number of observations used in 'MyModel' --
## not necessarily identical with nrow(Salaries) (why?)
nobs(MyModel)
```

```
## [1] 397
## CI's for the parameter estimates
confint(MyModel)
```

```
##                   2.5 %     97.5 %
## (Intercept) 83037.7223 101676.171
## sexMale      -486.2084  18629.808
## yrs.service   528.6072    966.617
```

```
## Covariance matrix
vcov(MyModel)
```

```
##             (Intercept)     sexMale yrs.service
## (Intercept)  22469380.8 -19847103.1  -143499.76
## sexMale     -19847103.1  23635586.3   -83260.40
## yrs.service   -143499.8    -83260.4    12409.07
```

```
## Akaike Information Criterion
AIC(MyModel)
```

```
## [1] 9275.883
```

```
## Schwartz/Bayesian Information Criterion
BIC(MyModel)
```

```
## [1] 9291.819
```

```
## Loglikelihood
logLik(MyModel)
```

```
## 'log Lik.' -4633.942 (df=4)
```

6.8 Model modification and variable selection

lm oblects can be easily expanded using the `update()` function

```
update(MyModel, .~.+rank)
```

```
## 
## Call:
## lm(formula = salary ~ sex + yrs.service + rank, data = Salaries)
## 
## Coefficients:
##   (Intercept)        sexMale    yrs.service  rankAssocProf       rankProf  
##       76612.8         5468.7         -171.8        14702.9        48980.2  
```

Based on an information criterion (AIC or BIC), variable selection can be performed – either `forward` (by defining a `scope` , i.e. the maximum model), `backward` , or `both` .

```
step(MyModel, scope=~rank+sex+yrs.service+yrs.since.phd,direction="forward")
```

```
## Start:  AIC=8147.25
## salary ~ sex + yrs.service
## 
##                  Df  Sum of Sq        RSS    AIC
## + rank            2 1.0177e+11 2.1799e+11 7999.1
## + yrs.since.phd  1 2.7346e+10 2.9242e+11 8113.8
## <none>                        3.1977e+11 8147.2
## 
## Step:  AIC=7999.15
## salary ~ sex + yrs.service + rank
## 
##                  Df Sum of Sq        RSS    AIC
## <none>                       2.1799e+11 7999.1
## + yrs.since.phd  1 639233285 2.1735e+11 8000.0
```

```
##
## Call:
## lm(formula = salary ~ sex + yrs.service + rank, data = Salaries)
##
## Coefficients:
##   (Intercept)        sexMale    yrs.service  rankAssocProf       rankProf
##       76612.8         5468.7         -171.8        14702.9        48980.2
```

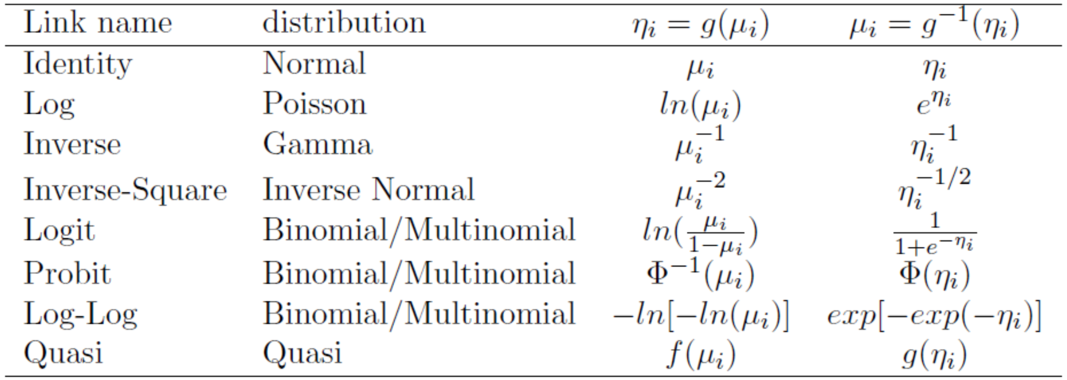
7 Generalized Linear Modeling

A Generalized Linear Model (GLM) is typically fitted using Maximum Likelihood (ML) techniques, and often iterative procedures are needed to fit the model to the data (e.g. Fisher Scoring for the binomial logit/ probit model in *R*). Since ML estimators are approximately normally distributed, the output for these models features a *z* value instead of a *t* value!

7.1 the `glm()` object

- `glm()` estimates a generalized linear model
- the most basic specification is `glm(formula, data, family)`
- `family` specifies the error distribution and link function (the default for `family` is ‘gaussian’, and the corresponding link is the identity link – Déja vù?)
- most of the functions applicable to lm-objects (`coef()` , `predict()` , `AIC()` ,…) work for glm-objects as well

7.2 Link functions



Link name	distribution	$\eta_i = g(\mu_i)$	$\mu_i = g^{-1}(\eta_i)$
Identity	Normal	μ_i	η_i
Log	Poisson	$ln(\mu_i)$	e^{η_i}
Inverse	Gamma	μ_i^{-1}	η_i^{-1}
Inverse-Square	Inverse Normal	μ_i^{-2}	$\eta_i^{-1/2}$
Logit	Binomial/Multinomial	$ln(\frac{\mu_i}{1-\mu_i})$	$\frac{1}{1+e^{-\eta_i}}$
Probit	Binomial/Multinomial	$\Phi^{-1}(\mu_i)$	$\Phi(\eta_i)$
Log-Log	Binomial/Multinomial	$-ln[-ln(\mu_i)]$	$exp[-exp(-\eta_i)]$
Quasi	Quasi	$f(\mu_i)$	$g(\eta_i)$

Overview of link functions

```
#EXERCISE 7.1: Reestimate the model from `MyModel` using the `glm()` function.
#What differences can be found in the summary output?
```

7.3 Introductory example: binomial logit model

```
  data(snails, package = "MASS")
snails$snailsDied <- snails$Deaths > 0
MyGLM <- glm(snailsDied ~ Species + Exposure + Rel.Hum + Temp, data=snails,
             family = binomial(link = "logit"))
MyGLM
```

```
##
## Call:  glm(formula = snailsDied ~ Species + Exposure + Rel.Hum + Temp,
##     family = binomial(link = "logit"), data = snails)
##
## Coefficients:
## (Intercept)     SpeciesB     Exposure      Rel.Hum         Temp
##     -7.5005       3.0893       5.2697      -0.1852       0.4616
##
## Degrees of Freedom: 95 Total (i.e. Null);  91 Residual
## Null Deviance:       131
## Residual Deviance: 29.93     AIC: 39.93
```

The `glm.summary` object yields the more useful information.

```
summary(MyGLM)
```

```
##
## Call:
## glm(formula = snailsDied ~ Species + Exposure + Rel.Hum + Temp,
##     family = binomial(link = "logit"), data = snails)
##
## Deviance Residuals:
##      Min        1Q    Median        3Q       Max
## -1.50340  -0.09650   0.00584   0.10311   2.58020
##
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept) -7.50048    6.14050  -1.221  0.22191
## SpeciesB     3.08925    1.24070   2.490  0.01278 *
## Exposure     5.26966    1.36976   3.847  0.00012 ***
## Rel.Hum     -0.18518    0.09362  -1.978  0.04792 *
## Temp         0.46165    0.16574   2.785  0.00535 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 131.035  on 95  degrees of freedom
## Residual deviance:  29.927  on 91  degrees of freedom
## AIC: 39.927
##
## Number of Fisher Scoring iterations: 8
```

7.4 Parameter interpretation for the binomial logit

- Since $\mu = \frac{exp\{X\beta\}}{1+exp\{X\beta\}}$, the estimated coefficients are the logarithmic odds
- Therefore, calculating $exp\{\hat{\beta}\}$ gives us the odds

```
exp(coef(MyGLM))
```

```
##  (Intercept)     SpeciesB     Exposure      Rel.Hum         Temp
## 5.528190e-04 2.196064e+01 1.943496e+02 8.309534e-01 1.586685e+00
```

For instance, the odds of some snails dying if the temperature is raised by 1 degree Celsius, change by roughly 60/40 ($1.587e+00 \approx 1.5$).

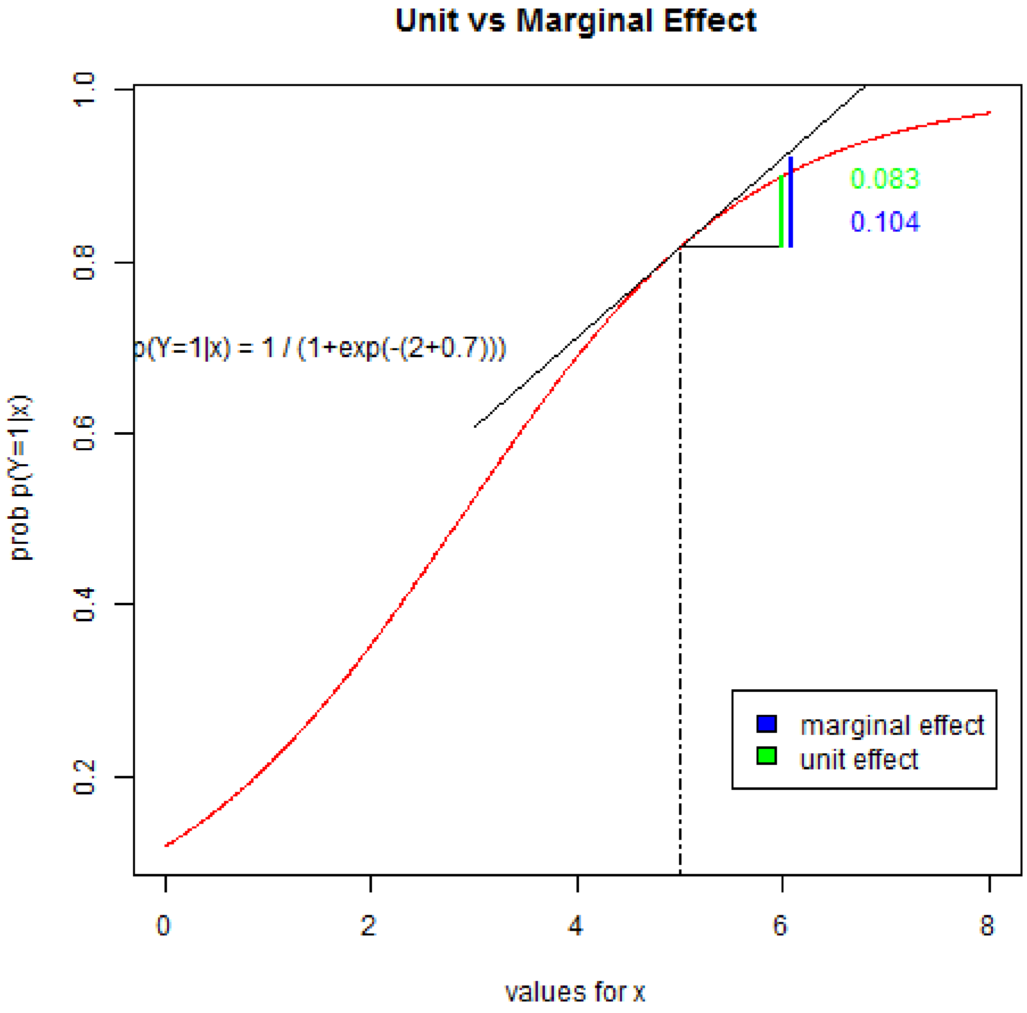
```
exp(confint(MyGLM,parm="Temp"))
```

```
## Waiting for profiling to be done...
```

```
##    2.5 %   97.5 %
## 1.204246 2.352041
```

Since “1” (odds of 50/50) is not included in the CI, the effect is significant at $\alpha = 0.05$.

7.4.1 Unit vs marginal effect



- marginal effect at $x = 5$:

$$\frac{\delta p(Y=1|x=5)}{\delta x} = \frac{e^{-\alpha+\beta x}\beta}{(1+e^{-(\alpha+\beta x)})^2} = \frac{\hat{p}(Y=1|x=5)}{\hat{p}(Y=0|x=5)\beta}$$

* unit effect at $x = 5$:

$$p(Y=1|x=6) - p(Y=1|x=5)$$

```
#EXERCISE 7.2: Load the Aids2 data set from the MASS package and estimate a binomial logit model of #`status` using `age`, `sex` and `diag` as predictors. Interpret the parameter estimate for `sex`.
```

7.4.2 Loglikelihood and Deviance

```
### Loglikelihood
(lnLG <- as.numeric(logLik(MyGLM)))
```

```
## [1] -14.96327
```

```
### number of parameters
kG <- MyGLM$rank
### Deviance (-2*lnL)
MyGLM$deviance
```

```
## [1] 29.92653
```

```
### Loglikelihood of the 0-model
MyGLM$null.deviance
```

```
## [1] 131.0353
```

```
### -2*lnLG_0 = MyGLM$null.deviance
### -->
(lnLG_0 <- MyGLM$null.deviance/-2)
```

```
## [1] -65.51765
```

There are four different Ways to obtain the AIC:

```
### a)
summary(MyGLM)$aic
```

```
## [1] 39.92653
```

```
### b)
AIC(MyGLM)
```

```
## [1] 39.92653
```

```
###c)
as.numeric(-2*lnLG+2*MyGLM$rank)
```

```
## [1] 39.92653
```

```
### d)
MyGLM$deviance+2*kG
```

```
## [1] 39.92653
```

7.4.3 Goodness of Fit

Since GLMs can not be fitted using OLS in a meaningful way, there are several suggestions around for ‘pseudo’ R^2s:

```
n <- nobs(MyGLM)
LG_0 <- exp(lnLG_0)
LG <- exp(lnLG)
### Cox&Snells
(r2CS <- 1-(LG_0/LG)^(2/n))
```

```
## [1] 0.6511859
```

```
### Cragg&Uher (Nagelkerke)
(r2NK <- (1-(LG_0/LG)^(2/n))/(1-LG_0^(2/n)))
```

```
## [1] 0.8745366
```

```
### adjusted McFadden
(r2McF <- 1-(lnLG-kG)/lnLG_0)
```

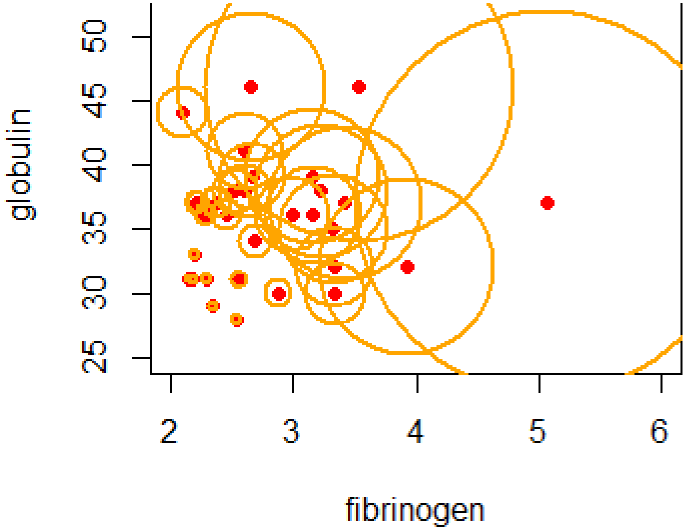
```
## [1] 0.6952994
```

Alternatively, you can use the function `pr2()` from the package `pscl` .

7.5 Final binomial logit example

7.5.1 Plotting two covariates *plus* the response predictor

```
  data(plasma, package="HSAUR")
plasma_glm_2 <- glm(ESR ~ fibrinogen + globulin,
                    data = plasma, family = binomial())
prob <- predict(plasma_glm_2, type = "response")
plot(globulin ~ fibrinogen, data = plasma, xlim = c(2, 6),
     ylim = c(25, 55), pch = 19, col="red")
symbols(plasma$fibrinogen, plasma$globulin, circles = prob, fg="orange",lwd=2,
        add = TRUE)
```



Source: *Everitt, Hothorn (2006): A Handbook of Statistical Analysis Using R.*

7.6 The multinomial logit model

Multinomial logit or probit models are fitted for nominal-scale response variables with $k > 2$ categories. In general, the objective is to calculate the probability for category $j = 1, \dots, k$ as a function of some covariates X. In the multinomial model case we get

$$P(Y_i = j|X) = \frac{e^{x_i^T \beta_j}}{\sum_{l=1}^{k-1} e^{x_i^T \beta_l}}$$

Multinomial models in R can be fitted using the function `multinom()` from the `nnet` package, where the iterative parameter estimation is conducted based on a neural network algorithm.

```
library(nnet)
```

```
##
## Attaching package: 'nnet'
## The following object is masked from 'package:mgcv':
##
##     multinom
```

```
example(multinom)
```

```
##
## multnm> options(contrasts = c("contr.treatment", "contr.poly"))
##
## multnm> library(MASS)
##
## multnm> example(birthwt)
##
## brthwt> bwt <- with(birthwt, {
## brthwt+ race <- factor(race, labels = c("white", "black", "other"))
## brthwt+ ptd <- factor(ptl > 0)
## brthwt+ ftv <- factor(ftv)
## brthwt+ levels(ftv)[-(1:2)] <- "2+"
## brthwt+ data.frame(low = factor(low), age, lwt, race, smoke = (smoke > 0),
## brthwt+            ptd, ht = (ht > 0), ui = (ui > 0), ftv)
## brthwt+ })
##
## brthwt> options(contrasts = c("contr.treatment", "contr.poly"))
##
## brthwt> glm(low ~ ., binomial, bwt)
##
## Call:  glm(formula = low ~ ., family = binomial, data = bwt)
##
## Coefficients:
## (Intercept)          age          lwt    raceblack    raceother
##     0.82302     -0.03723     -0.01565      1.19241      0.74068
##   smokeTRUE      ptdTRUE       htTRUE       uiTRUE         ftv1
##     0.75553      1.34376      1.91317      0.68020     -0.43638
##       ftv2+
##     0.17901
##
## Degrees of Freedom: 188 Total (i.e. Null);  178 Residual
## Null Deviance:       234.7
## Residual Deviance: 195.5     AIC: 217.5
##
## multnm> (bwt.mu <- multinom(low ~ ., bwt))
## # weights:  12 (11 variable)
## initial  value 131.004817
## iter  10 value 98.029803
## final  value 97.737759
## converged
## Call:
## multinom(formula = low ~ ., data = bwt)
##
## Coefficients:
## (Intercept)         age         lwt   raceblack   raceother   smokeTRUE
##  0.82320102 -0.03723828 -0.01565359  1.19240391  0.74065606  0.75550487
##     ptdTRUE      htTRUE      uiTRUE        ftv1       ftv2+
##  1.34375901  1.91320116  0.68020207 -0.43638470  0.17900392
##
## Residual Deviance: 195.4755
## AIC: 217.4755
```

8 Modeling Nonlinear Data

- Nonlinear least squares (`nls` -function)
- Local-area regression (`locfit` -package, `loess` -function)
- Splines (`spline` -function, `smooth.spline` -function, `locfit` -package)
- Generalized additive models (`mgcv` and `locfit` -package)

8.1 Nonlinear least squares estimation

- accessible via `nls()`
- Useful if the (vague) functional form of the model is known to the analyst
- Needs some theoretical justification of the model
- Starting values for the parameters have to be specified

Example 1 *(taken from Crawley, 2013)*:

Consider the following model that captures the relationship for deers between age (X) and jaw bone length (Y):

$$y = \frac{\alpha x}{1 + \beta x}$$

where α and β are unknown parameters to be estimated. This is a so-called Michaelis-Menten model, and clearly there exists no transformation that changes the above form into a linear model. However, using `nls()` still allows us to fit such models using iteratively weighted least squares.

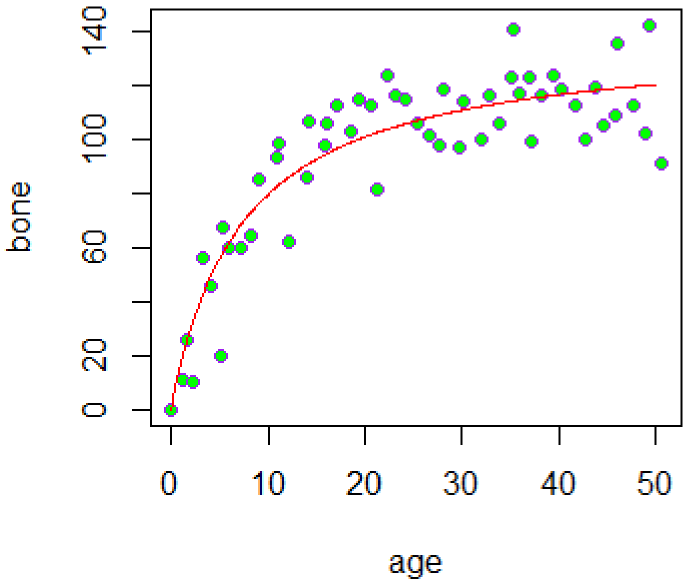
```
deer <- read.table("c:/Uni/Daten/Crawley/jaws.txt", header=TRUE)
attach(deer)
names(deer)
```

```
## [1] "age"  "bone"
```

```
plot(age, bone, pch=21, col="purple", bg="green")
# specify starting values for a and b
model <- nls(bone ~ a*age/(1+b*age), start=list(a=8, b=0.08))
summary(model)
```

```
##
## Formula: bone ~ a * age/(1 + b * age)
##
## Parameters:
##   Estimate Std. Error t value Pr(>|t|)
## a 18.72539    2.52587   7.413 1.09e-09 ***
## b  0.13596    0.02339   5.814 3.79e-07 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 13.77 on 52 degrees of freedom
##
## Number of iterations to convergence: 7
## Achieved convergence tolerance: 1.553e-06
```

```
av <- seq(0, 50, 0.1)
bv <- predict(model, list(age=av))
lines(av, bv, col="red", lwd=1.5)
```



We can also calculate an R^2 for nonlinear regression models:

```
sse <- summary(model)$sigma^2*summary(model)$df[2]
# estimate null model (intercept only)
null <- lm(bone~1)
summary(null)
```

```
## 
## Call:
## lm(formula = bone ~ 1)
## 
## Residuals:
##     Min      1Q  Median      3Q     Max 
## -93.979  -8.844   9.872  21.775  48.021 
## 
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)    
## (Intercept)   93.979      4.541    20.7   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
## 
## Residual standard error: 33.37 on 53 degrees of freedom
```

```
sst <- summary(null)$sigma^2*summary(null)$df[2]
# explained variation in percent
100*(sst-sse)/sst
```

```
## [1] 83.29987
```

```
detach(deer)
```

Example 2 *(provided by Falko Tesch / Professur für Bevölkerungswissenschaften)*:

Objective of the analysis is to model for ‘centenarians’ (people 100+ yrs) over time. Proposed model:

$$\mu_i = \frac{50,000}{1 + e^{\beta_2 + \beta_3 time_i}}$$

with $D(time) = \{0, \ldots, 6\}$.

The mode is fitted separately for women and men.

```
pop.f <- c(263.2,525.43,1042.89,2088.72,3786.86,5730.27,11077.1)
pop.m <- c(38.6,72.05,130.27,250.15,435.41,559.28,1628.57)
year <- c(1950,1960,1970,1980,1990,2000,2010)
year2 <- c(1950,1960,1970,1980,1990,2000,2010,2020,2030,2040,2050)
time <- 0:6
pop.modf <- nls(pop.f~50000/(1+exp(beta2+beta3*time)),
                start=list(beta2=4,beta3=-0.7),
                trace=T,algorithm="default")
```

```
## 530222090 :   4.0 -0.7
## 12932661 :   4.2553229 -0.5259445
## 762441.3 :   4.9425736 -0.6100616
## 513912.2 :   5.1776965 -0.6490725
## 512291.4 :   5.2004105 -0.6530907
## 512287.4 :   5.2015400 -0.6532945
## 512287.4 :   5.201592 -0.653304
```

```
pop.modm <- nls(pop.m~50000/(1+exp(beta2+beta3*time)),
                start=list(beta2=4,beta3=-0.7),
                trace=T,algorithm="default")
```

```
## 1182352490 :   4.0 -0.7
## 71197891 :   4.0945928 -0.3847772
## 6776612 :   5.1057416 -0.3992528
## 537046.4 :   6.1726790 -0.4842845
## 87837.39 :   7.1740597 -0.6219118
## 59273.19 :   7.7934934 -0.7260452
## 57212.42 :   8.0038015 -0.7626365
## 57101.31 :   8.0552956 -0.7716225
## 57096 :   8.0666836 -0.7736132
## 57095.75 :   8.0691436 -0.7740435
## 57095.74 :   8.069672 -0.774136
## 57095.74 :   8.0697857 -0.7741558
## 57095.74 :   8.0698101 -0.7741601
```

Results for women:

```
summary(pop.modf)
```

```
## 
## Formula: pop.f ~ 50000/(1 + exp(beta2 + beta3 * time))
## 
## Parameters:
##       Estimate Std. Error t value Pr(>|t|)    
## beta2  5.20159    0.18689   27.83 1.12e-06 ***
## beta3 -0.65330    0.03398  -19.23 7.02e-06 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
## 
## Residual standard error: 320.1 on 5 degrees of freedom
## 
## Number of iterations to convergence: 6 
## Achieved convergence tolerance: 5.814e-06
```

```
coef(pop.modf)
```

```
##     beta2     beta3 
##  5.201592 -0.653304
```

Results for men:

```
summary(pop.modm)
```

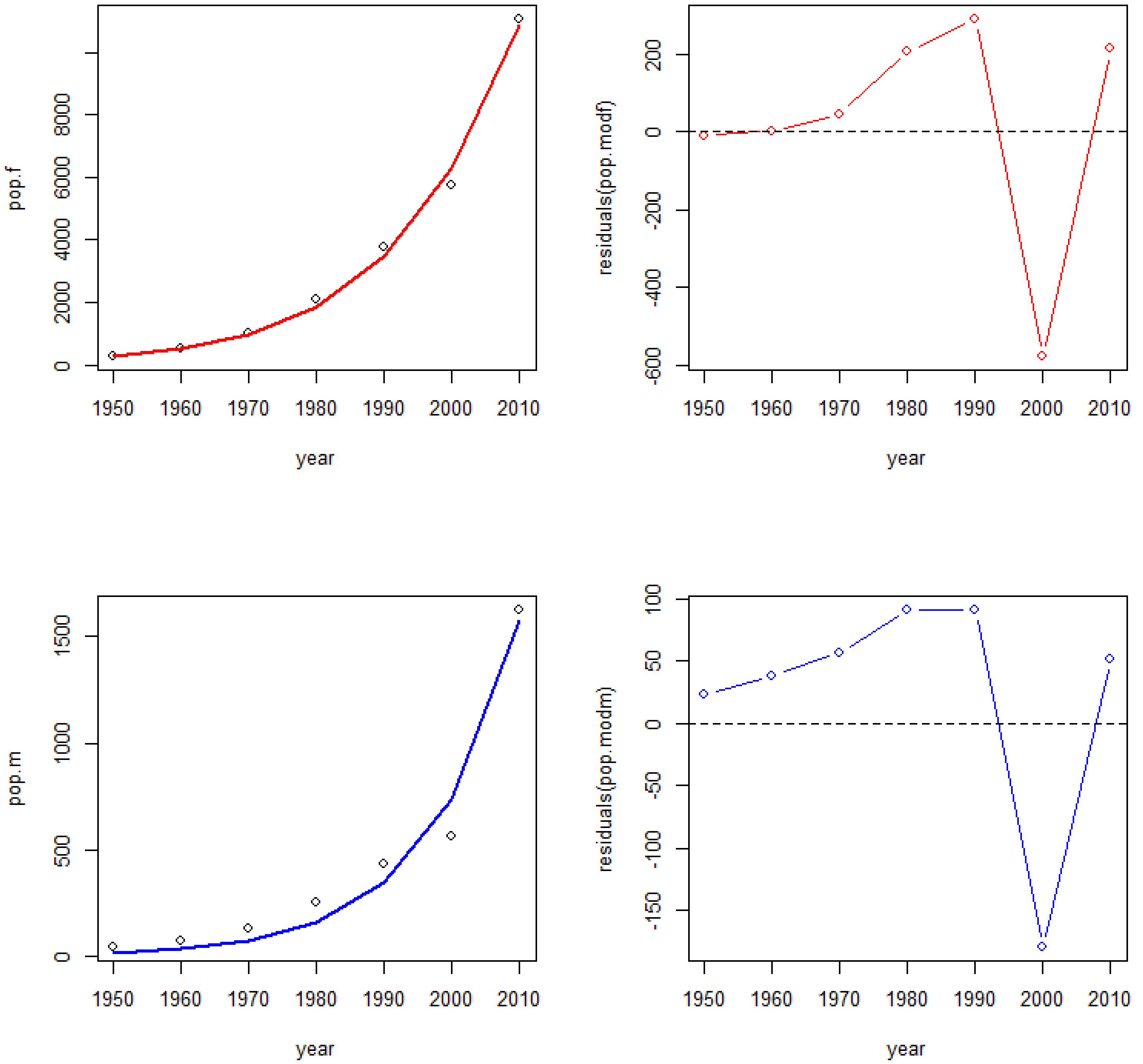
```
##
## Formula: pop.m ~ 50000/(1 + exp(beta2 + beta3 * time))
##
## Parameters:
##       Estimate Std. Error t value Pr(>|t|)
## beta2   8.0698     0.5862  13.767 3.63e-05 ***
## beta3  -0.7742     0.1021  -7.585 0.000632 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 106.9 on 5 degrees of freedom
##
## Number of iterations to convergence: 12
## Achieved convergence tolerance: 4.048e-06
```

```
coef(pop.modm)
```

```
##      beta2      beta3
##  8.0698101 -0.7741601
```

visualization of results:

```
layout(matrix(1:4,nrow=2))
plot(year,pop.f)
lines(year,fitted.values(pop.modf),lwd=2,col="red")
plot(year,pop.m)
lines(year,fitted.values(pop.modm),lwd=2, col="blue")
plot(year,residuals(pop.modf),type='b',col="Red")
abline(h=0,lty=2)
plot(year,residuals(pop.modm),type='b', col="blue")
abline(h=0,lty=2)
```


pop.f
0
2000
4000
6000
8000
1950
1960
1970
1980
1990
2000
2010
year
residuals(pop.modf)
200
0
-200
-400
-600
1950
1960
1970
1980
1990
2000
2010
year
pop.m
0
500
1000
1500
1950
1960
1970
1980
1990
2000
2010
year
residuals(pop.modm)
100
50
0
-50
-100
-150
1950
1960
1970
1980
1990
2000
2010
year

```
### EXERCISE 8.1 (inspired by the appendix of John Fox' car book):
# a) Load the 'USPop' data set from the car package.
# Run a nonlinear regression of the following form:
# population ~ theta1/(1 + exp(-(theta2 + theta3*year)) using self start and
# theta1 = 400, theta2 = -49 and theta3 = 0.025 as starting values.
# b) plot year against population, using 0 and 500 as limits on the y-axis, and
# 1790 and 2100 as limits on the x-axis.
# c) Add the regression line of the nls-model to the plot, and extend the line for
# the decades 2010 to 2100 (in 10-year steps)
```

8.2 Local-area regression

As we have seen, bivariate associations can be highly non-linear, and not always a theoretical model presents itself. But maybe a model can be *locally* linear?

8.2.1 LOESS in a nutshell

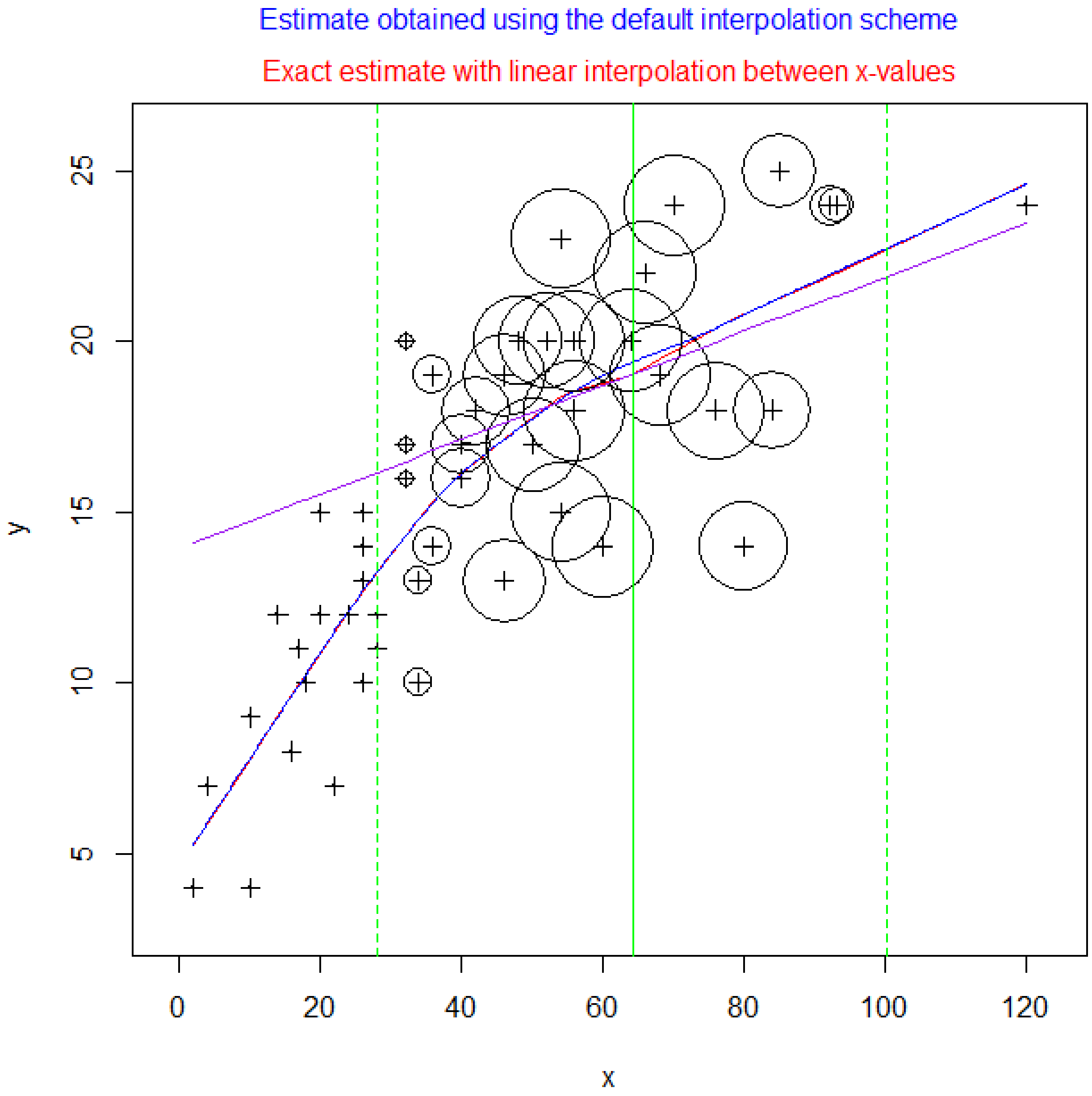
- polynomials are fitted locally
- predict each observation separately using observations in the vicinity
- decrease the influence (weight) of more distant observations
- ‘non-parametric’ approach
- each observation is predicted by several values

The model for local regression

$$Y_i = \mu(x_i) + \epsilon_i$$

fits a polynomial model within a moving interval. Locally weighted least squares are fitted for every Y_i using

$$\sum_{i=1}^{n} w_i(Y_i - (\beta_0 + \beta_1(x_i - x)))^2$$



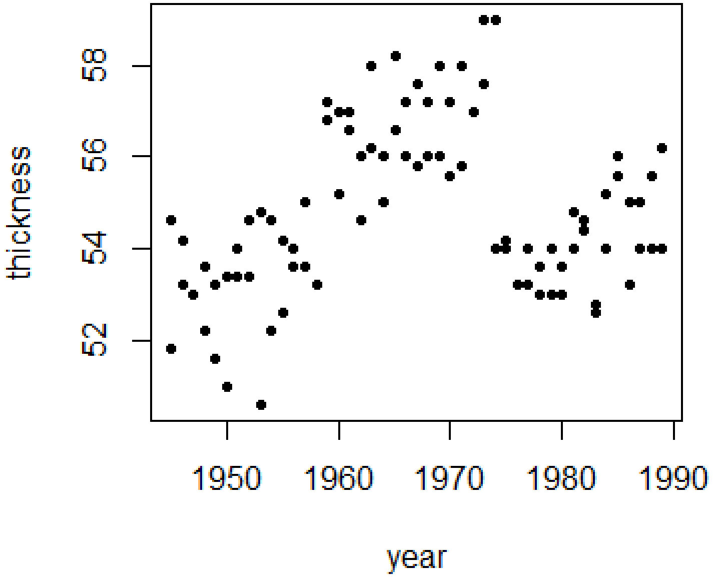
Visualization of the loess principle (loess.demo example from TeachingDemos package)

8.2.2 Introductory Example

Thickness of penny coins (as indicator of a society's economic welfare)

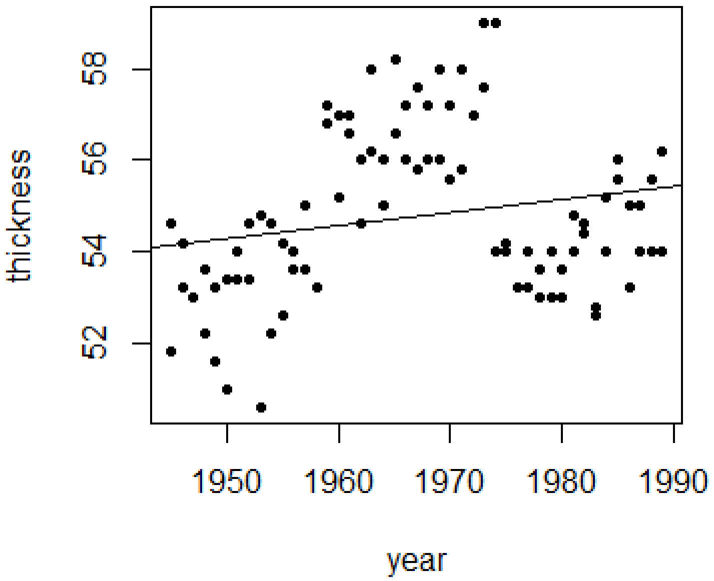
We start out by looking at the scatterplot…

```
data(penny)
plot(penny,pch=20)
```



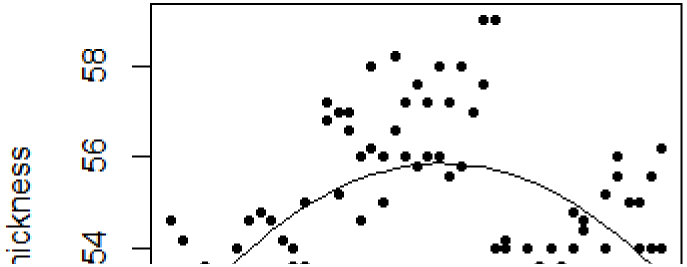
…and find out that linear regression is not a wise option.

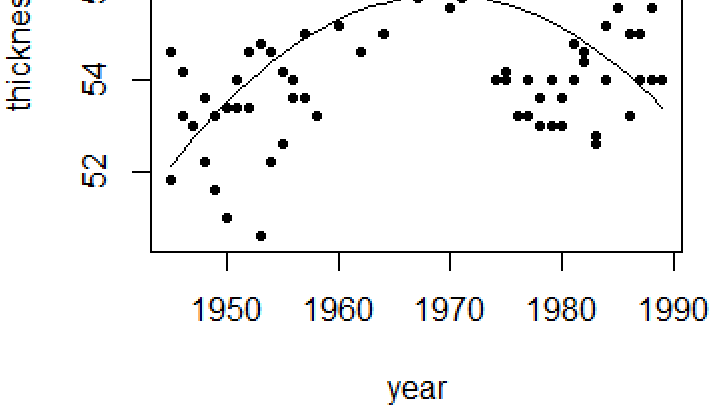
```
rmod1 <- lm(thickness~year, data=penny)
plot(penny,pch=20)
abline(rmod1)
```



Common suggestion: ‘Add a quadratic term!’

```
rmod2 <- lm(thickness~year+I(year^2), data=penny)
plot(penny,pch=20)
lines(penny$year,predict(rmod2))
```

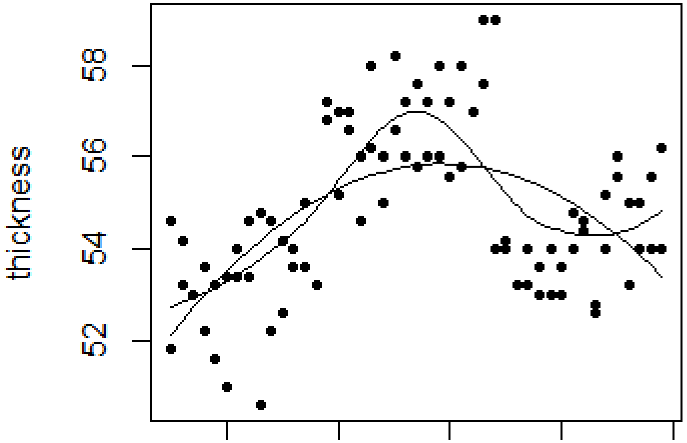


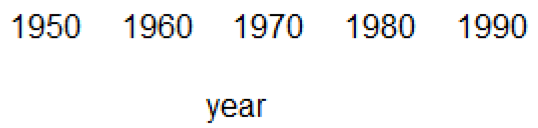


But does this really reflect the data-generating process?

- Splines and local-area regression are data-driven alternatives
- Drawback of polynomials: The model can ‘run away’ at the margins of the data space

```
library(locfit)
lmod <- locfit(thickness~year, data=penny)
plot(penny,pch=20)
lines(penny$year,predict(rmod2))
lines(lmod)
```





The local-area regression seems to fit the data better!

The function `locfit()` from the same-named package fits locally weighted least squares, where the weight function defaults to

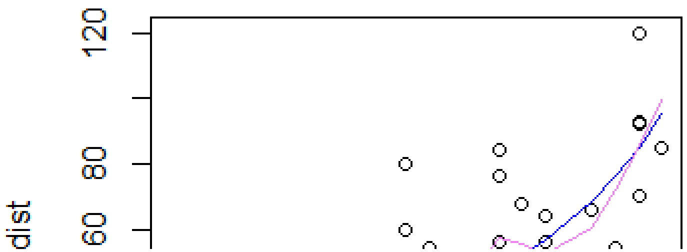
$$w_i = \begin{cases} \left(1 - \left|\frac{x-x_i}{d(x)}\right|^3\right)^3 & \text{for } \left|\frac{x-x_i}{d(x)}\right| < 1 \\ 0 & \text{otherwise} \end{cases},$$

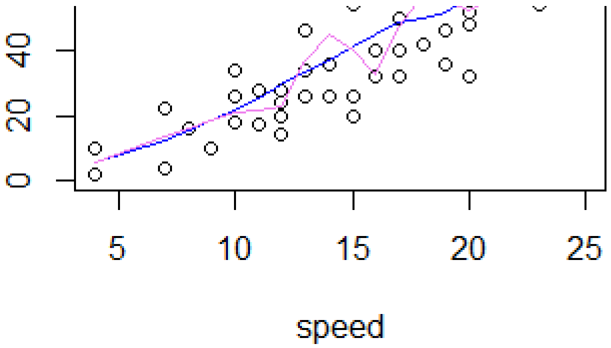
where x is the covariate value of some response Y to be smoothed, x_i is a covariate value of a nearest neighbour (i.e. falls within a pre-defined range), and $d(x)$ is the distance along the abscissa from x to the ‘furthest nearest neighbour’.

`loess()` -function in R (less general than `locfit()` and a bit outdated):

- takes up to four regressors (typically interactions)
- ‘vicinity’ defined by parameter `span`

```
attach(cars)
lomod <- loess(dist ~ speed)
lomod2 <- loess(dist ~ speed, span=0.4)
plot(speed, dist)
lines(predict(lomod)~speed, col="blue")
lines(predict(lomod2)~speed, col="violet")
```





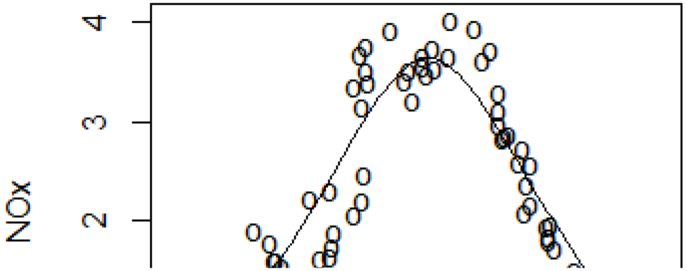
The `span` -parameter defines the proportion of observations to be taken into consideration (i.e. values > 1 would use all observations, and we would get a ‘regular’ polynomial regression).

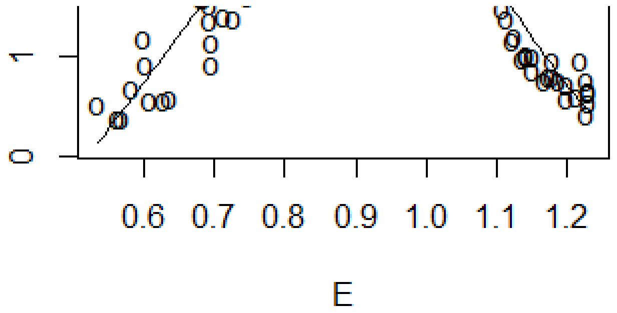
`locfit()` -function in R

- `lp()` -function allows to use several regressors
- local polynomial estimator
- ‘old’ set of functions developed by Bell Labs

`locfit()` -fit

```
library(locfit)
data(ethanol, package="locfit")
fit <- locfit(NOx ~ E, data=ethanol)
plot(fit, get.data=TRUE)
```





`locfit()` and `loess()` are for fitting data - not modeling

- no interpreation of parameters possible

```
# loess output
summary(lomod)
```

```
## Call:
## loess(formula = dist ~ speed)
##
## Number of Observations: 50
## Equivalent Number of Parameters: 4.78
## Residual Standard Error: 15.29
## Trace of smoother matrix: 5.24  (exact)
##
## Control settings:
##   span     :  0.75
##   degree   :  2
##   family   :  gaussian
##   surface  :  interpolate      cell = 0.2
##   normalize:  TRUE
##  parametric:  FALSE
## drop.square:  FALSE
```

```
#locfit output
summary(fit)
```

```
## Estimation type: Local Regression
##
## Call:
## locfit(formula = NOx ~ E, data = ethanol)
##
## Number of data points:  88
## Independent variables:  E
## Evaluation structure: Rectangular Tree
## Number of evaluation points:  5
## Degree of fit:  2
## Fitted Degrees of Freedom:  4.762
```

```
#EXERCISE 8.2:
# a) Load the 'Highway1' data set from the car package. Run a linear
# regression and a local area regression of 'ADT' on 'len'.
# b) Plot the results from the local area regression and add the regression line
# from the linear regression.
# c) Make predictions for len=50 with both models and comment on the results.
# d) Re-estimate the local area regression using a first-order polynomial
# (hint: You need the function lp())
```

8.3 Regression and Smoothing Splines

8.3.1 Introduction to regression splines

- Basic idea (akin to local-area regression): define knots over the data space, where (flexible) regression lines are connected (linear, cubic, natural,…) → no breaks – Y should be a function of x!
- notational concept:

$\beta_0 + \beta_1 x$, if $x \leq c$

$\beta_0 + \beta_1 x + \beta_2 (x - c)$, if $x > c$

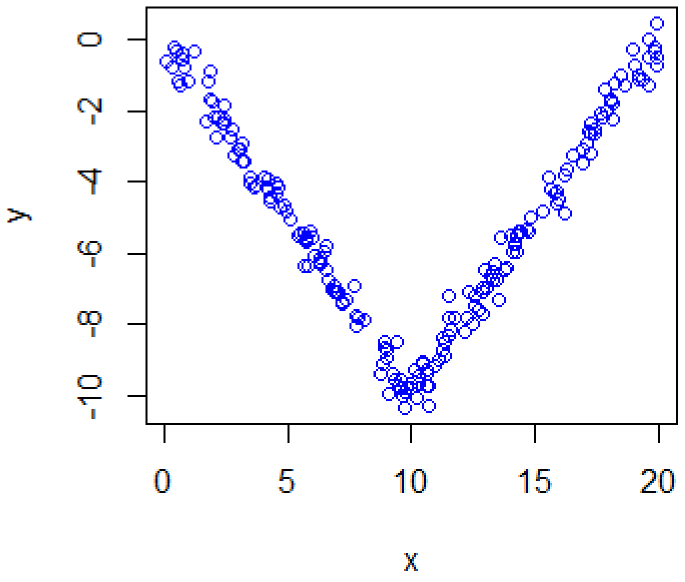
This leads to

$\beta_0 + \beta_1 x + \beta_2 x_+$, with $x_+ = 0$, if $x \leq c$ and $x_+ = x$, if $x > c$. * $\Rightarrow$ splines allow for parameter interpretation

Basic example:

- Generate non-linear data

```
x1 <- runif(100,0,10)
x2 <- runif(100,10,20)
y1 <- -x1+rnorm(100,0,0.4)
y2 <- -20+x2+rnorm(100,0,0.4)
dat <- data.frame(x=c(x1,x2),y=c(y1,y2))
plot(dat, col="blue")
```



Fitting a linear regression would, however, lead to ‘sharp edges’ at the turning point, where the function is no longer continuously differentiable. Cubic splines add third order polynomials to ensure curvature, and continuity at the cut-off point (which is referred to as *knot*) is ensured using the corresponding truncated power basis function to makes ends meet:

$$(x-\xi)^3_+ = \begin{cases} (x-\xi)^3 & \text{if } x > \xi \\ 0 & \text{otherwise.} \end{cases}$$

This can obviously be extended to *K* knots easily, but too many knots might lead to overfitting the model which is why typically cross validation is employed to find a good compromise between model flexibility and model fit.

Because splines tend to ‘lash out’ at the data space margins like a whip, *natural splines* are often used to add boundary constraints which have the effect of a linearization at these boundaries.

8.3.2 Introduction to smoothing splines

Smoothing splines do not require a pre-specified number of knots, because they follow a different approach (which is even closer to the local area regression idea): The objective is now to find a function that minimizes $RSS = \sum_{i=1}^n (y_i - g(x_i))^2$. Technically, a function that would fit the data perfectly yields a value of zero, but solutions might exist which yield better forecasts for *new* observations! Adding a so-called tuning parameter λ leads to

$$\sum_{i=1}^n (y_i - g(x_i))^2 + \lambda \int g^{''}(t)^2 dt.$$

The second term penalizes the flexibility in $g(\cdot)$. Since the second derivatives measure the change of slope, the parameter λ governs the ‘wiggliness’ of the spline (with $\lambda = 0$ being a perfectly smooth fit to the observed data). Leave-one-out cross validation is frequently used to determine a suitable value for λ.

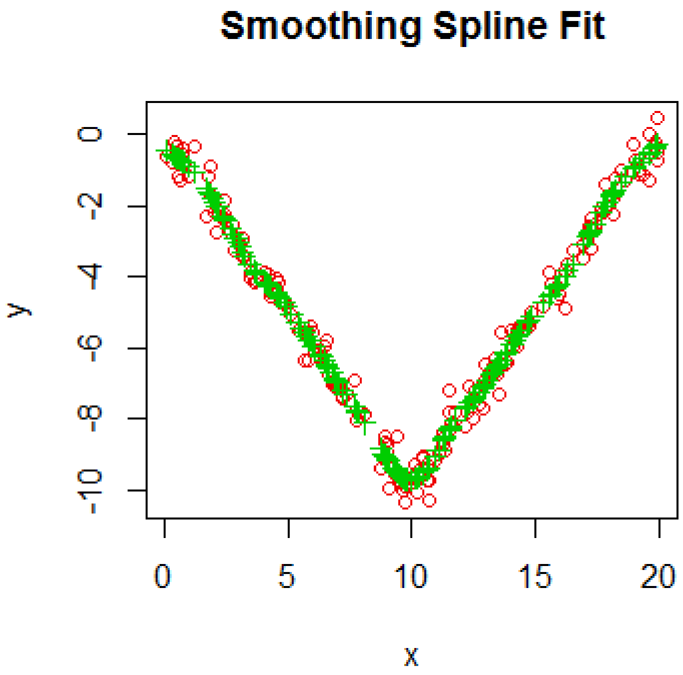
8.3.3 `spline()` / `splinefun()` / `smooth.spline()` : Fitting univariate cubic and smoothing splines in R

- only one regressor possible

```
smod <- smooth.spline(dat$x,dat$y)
names(smod)
```

```
##  [1] "x"        "y"        "w"        "yin"      "data"     "lev"
##  [7] "cv.crit"  "pen.crit" "crit"     "df"       "spar"     "lambda"
## [13] "iparms"   "fit"      "call"
```

```
plot(dat, col="red2",main="Smoothing Spline Fit")
points(smod,pch=3, col="green3")
```



8.4 Generalized Additive Models

Two approaches to dealing with multivariate predictors:

- MARS (Multivariate adaptive regression splines – not dealt with in the following)
  - Package `mda`
  - `mars()` -function from `mda` package
- GAM (Generalized Additive Models)
  - Packages `gam` and `mgcv`
  - `gam()` -function from `mgcv` package most popular choice
- more than on regressor possible

- hard to determine the border, where more smoothness is gained at expense of overfitting

Generalized additive models capture multivariate nonlinear associations without requiring the user to specify a parametric functional form for the model.

The formula object logic of `gam()` from the `mgcv` package allows for a combination of parametric and non-parametric inputs:

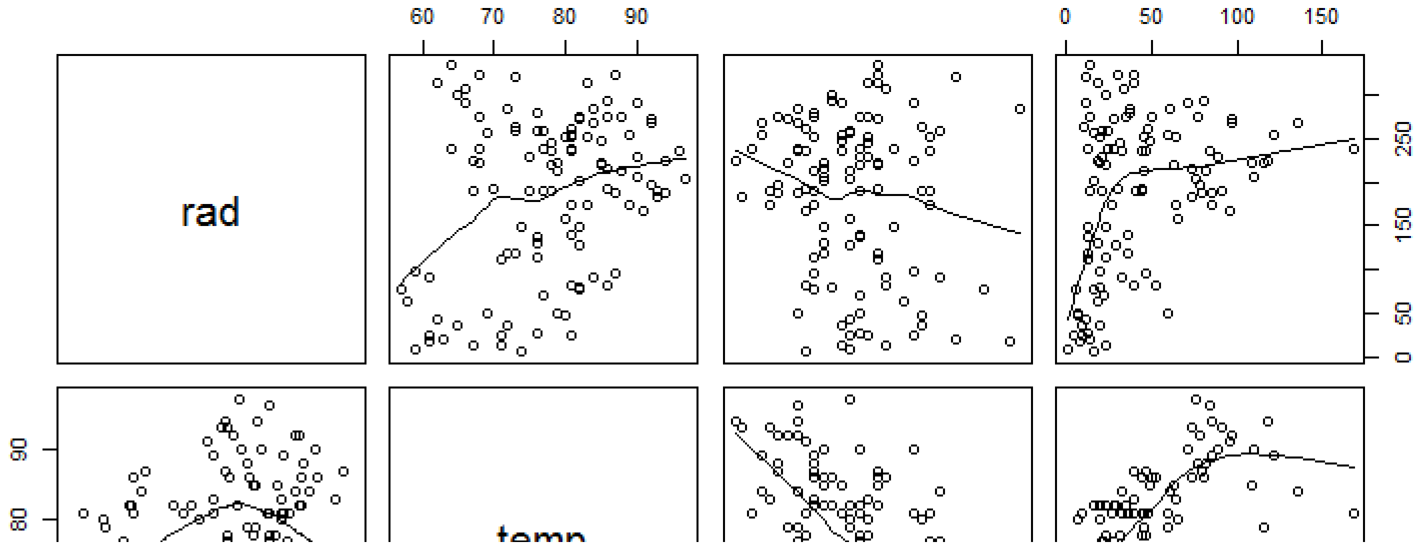
- `y ~ s(x1) + s(x2)` : Both covariates are non-parametrically smoothed
- `y ~ s(x1) + x2` : Only `x1` is parametrically smoothed – `x2` is parametrically estimated
- `y ~ s(x1) + s(x1,x2)` : implies an isotrophic smooth (e.g. if `x2` is nested in `x1`)

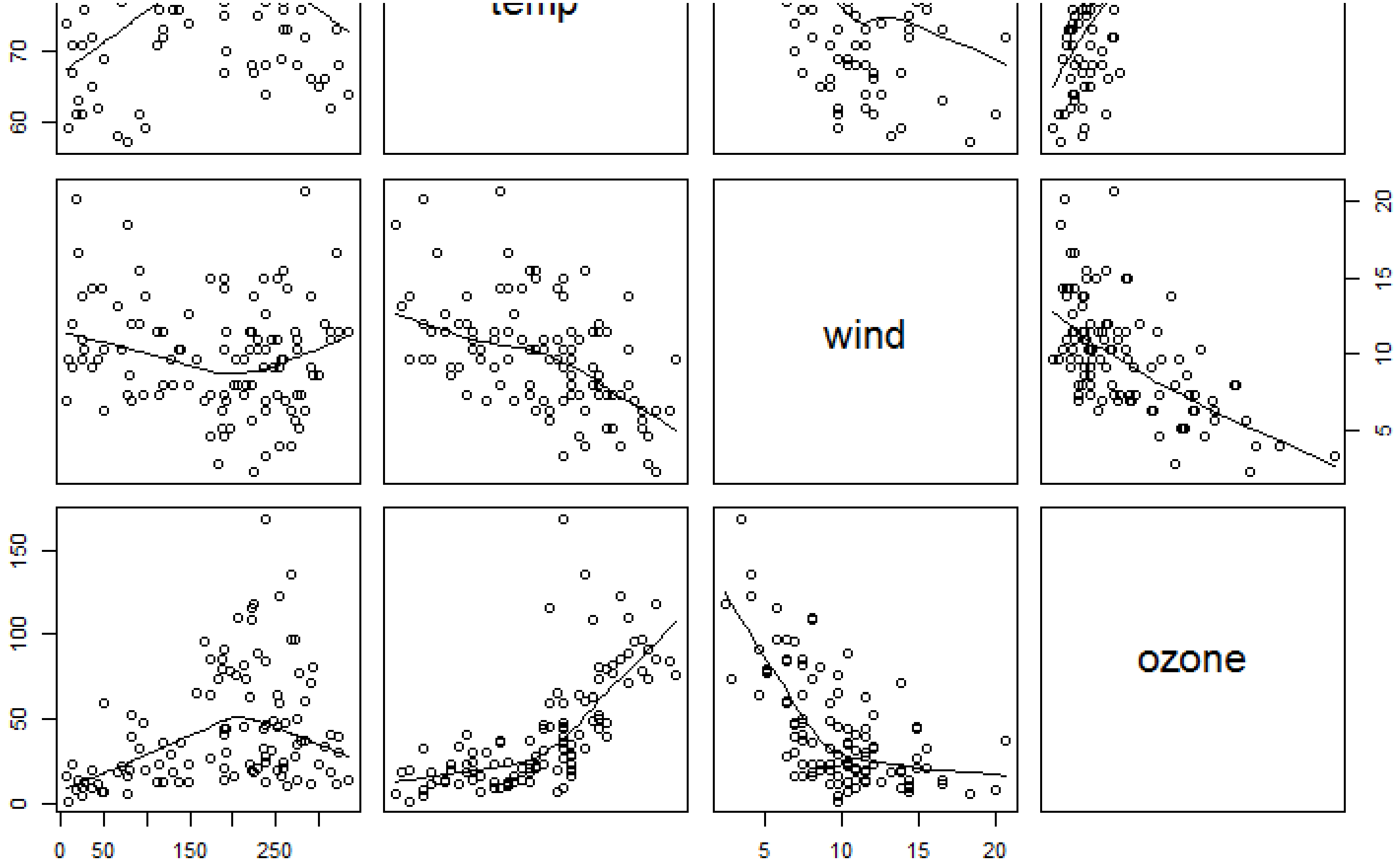
Example (from Crawley, 2013): Ozone concentration

```
ozone.data <- read.table("c:/Uni/Daten/Crawley/ozone.data.txt", header=TRUE)
attach(ozone.data)
names(ozone.data)
```

```
## [1] "rad"   "temp"  "wind"  "ozone"
```

```
pairs(ozone.data, panel=function(x, y) {points(x,y); lines(lowess(x,y))})
```





```
library(mgcv)
gam1 <- gam(ozone~s(rad)+s(temp)+s(wind))
summary(gam1)
```

```
##
## Family: gaussian
## Link function: identity
##
## Formula:
## ozone ~ s(rad) + s(temp) + s(wind)
##
## Parametric coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)    42.10       1.66   25.36   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Approximate significance of smooth terms:
##           edf Ref.df      F  p-value
## s(rad)  2.763  3.451  3.964   0.0085 **
## s(temp) 3.841  4.762 11.612 8.19e-09 ***
## s(wind) 2.918  3.666 13.770 1.39e-08 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## R-sq.(adj) =  0.724   Deviance explained = 74.8%
## GCV =    338  Scale est. = 305.96    n = 111
```

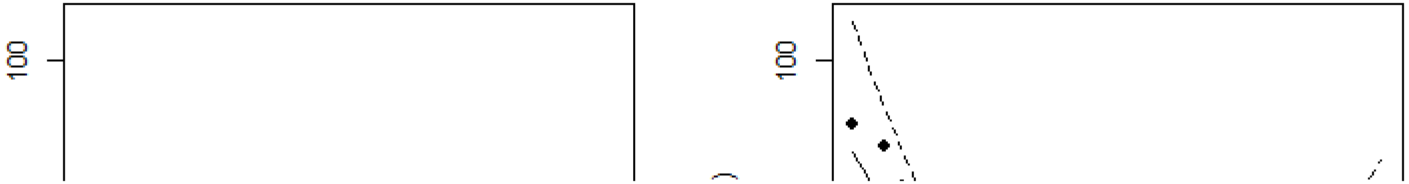
```
gam2 <- gam(ozone~s(temp)+s(wind))
anova(gam1, gam2, test="F")
```

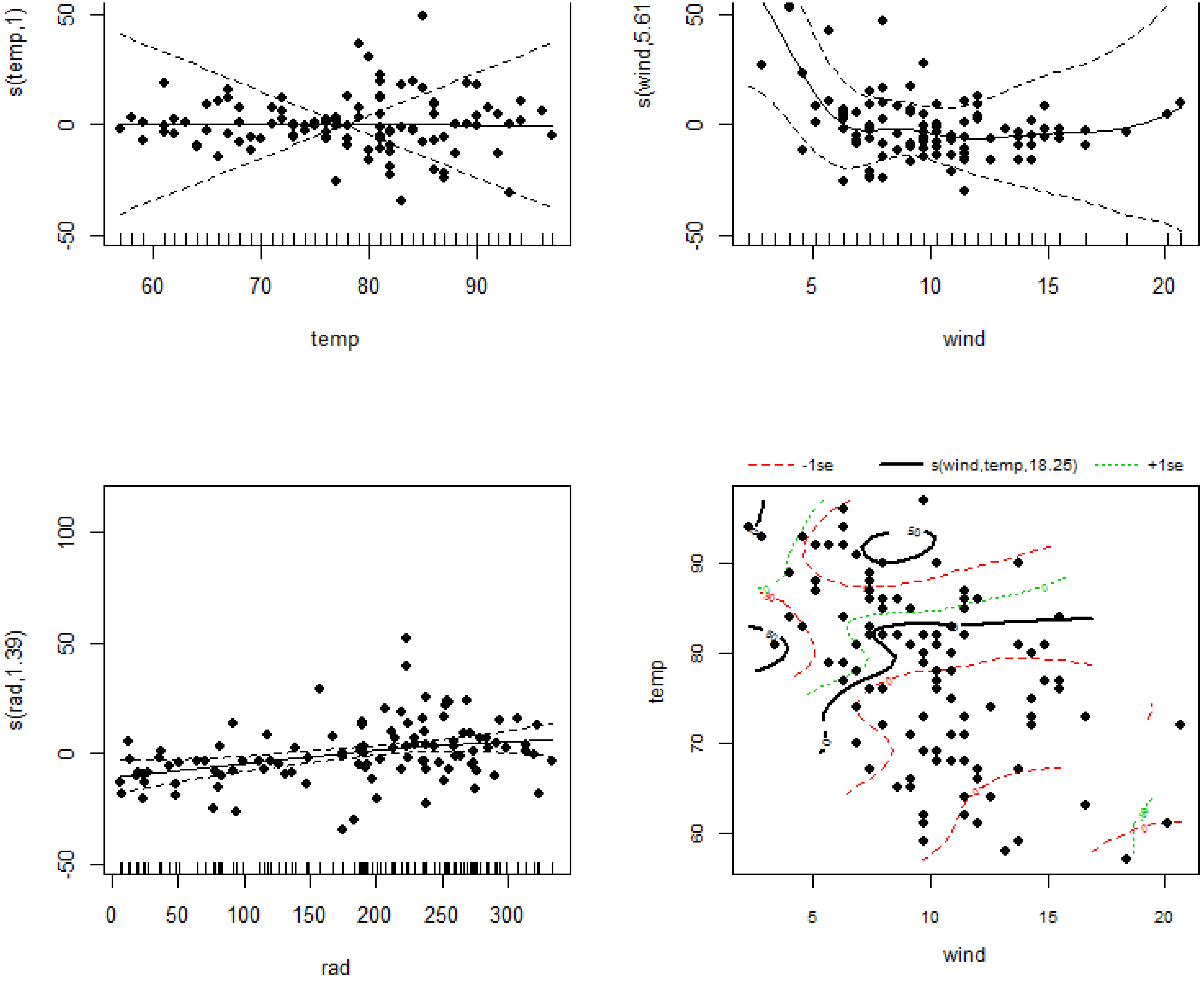
```
## Analysis of Deviance Table
##
## Model 1: ozone ~ s(rad) + s(temp) + s(wind)
## Model 2: ozone ~ s(temp) + s(wind)
##   Resid. Df Resid. Dev      Df Deviance      F Pr(>F)
## 1     98.12      30742
## 2    101.10      34885 -2.9757  -4142.2 4.5496 0.0051 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
gam3 <- gam(ozone~s(temp)+s(wind)+s(rad)+s(wind, temp))
summary(gam3)
```

```
##
## Family: gaussian
## Link function: identity
##
## Formula:
## ozone ~ s(temp) + s(wind) + s(rad) + s(wind, temp)
##
## Parametric coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)   42.099      1.361   30.92   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Approximate significance of smooth terms:
##                 edf Ref.df     F p-value
## s(temp)       1.000  1.000 0.000  0.9892
## s(wind)       5.613  6.482 2.492  0.0244 *
## s(rad)        1.389  1.667 5.799  0.0126 *
## s(wind,temp) 18.246 27.000 2.805 8.5e-08 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## R-sq.(adj) =  0.814   Deviance explained = 85.9%
## GCV = 272.66  Scale est. = 205.72    n = 111
```

```
layout(matrix(1:4,nrow=2, byrow=TRUE))
plot(gam3, residuals=TRUE, pch=16)
```





```
detach(ozone.data)
```

```
#EXERCISE 8.3:
# a) Load the 'car90' data set from the rpart package.
# b) Estimate a GAM of HP on a joint (default) smoother for 'Disp2' and 'Eng.Rev'.
# c) Estimate another GAM with the same smoother, and add smoothed terms for each
# variable separately, using a regression spline as smooth function.
# d) Comment on the explanatory power of the two models.
```

9 Cluster Analysis

- k-means: `kmeans()` , `kmeansruns()`
- Hierarchical clustering: `hclust()` , `agnes()` , `diana()`
- Only for classification – not prediction

9.1 k-means clustering

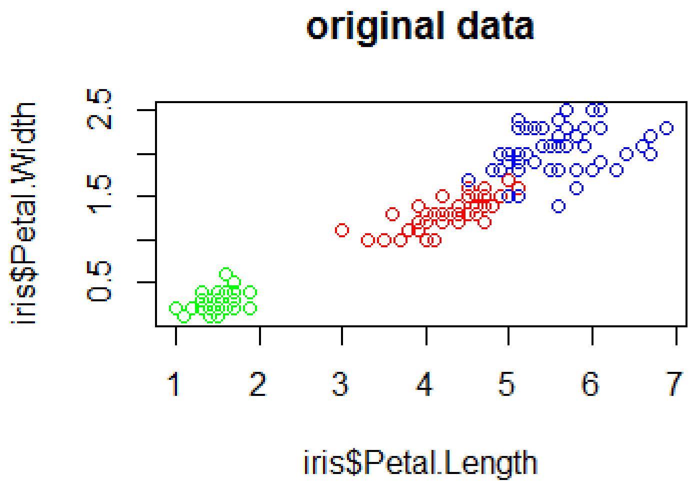
- Identifies for each of n observations its nearest out of a pre-specified k means
- computationally demanding, but heuristic techniques improve performance

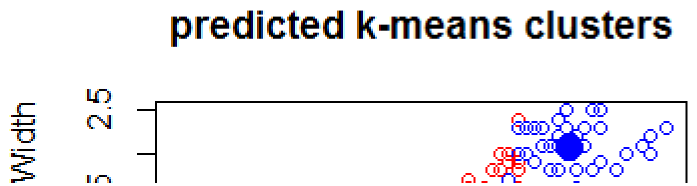
Example: Fisher’s famous iris data set (see http://en.wikipedia.org/wiki/Iris_flower_data_set (http://en.wikipedia.org/wiki/Iris_flower_data_set))

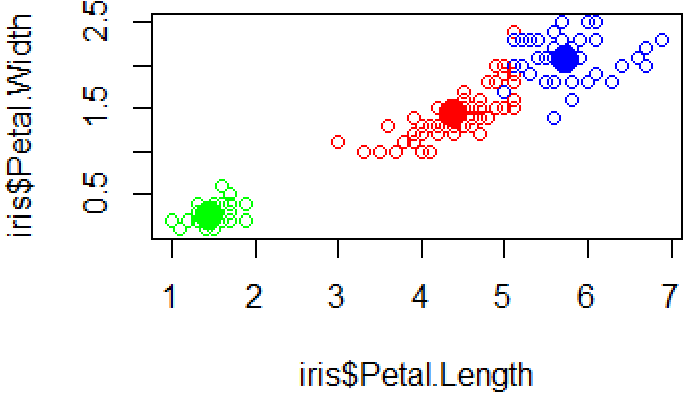
```
str(iris)
```

```
## 'data.frame':    150 obs. of  5 variables:
##  $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
##  $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
##  $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
##  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
##  $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
```

```
set.seed(1234)
km <- kmeans(iris[,-5], centers=3)
col1 <- col2 <- character(nrow(iris))
col1[iris$Species == levels(iris$Species)[1]] <- "green"
col1[iris$Species == levels(iris$Species)[2]] <- "red"
col1[iris$Species == levels(iris$Species)[3]] <- "blue"
# predicted cluster
col2[km$cluster == 1] <- "green"
col2[km$cluster == 2] <- "red"
col2[km$cluster == 3] <- "blue"
layout(1:2)
plot(iris$Petal.Length, iris$Petal.Width, col=col1, pch=21, main="original data")
plot(iris$Petal.Length, iris$Petal.Width, col=col2, pch=21, main="predicted k-means clusters")
points(km$centers[, c("Petal.Length", "Petal.Width")], pch=20, cex=3, col=c("green", "red","blue"))
```







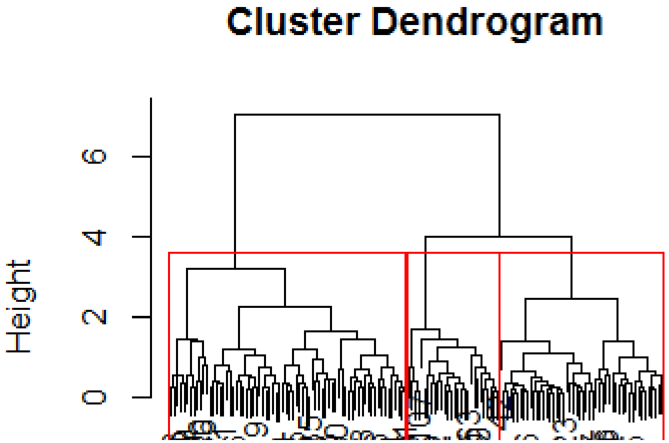
The k-means algorithm works fast (not relevant for 150 irises, but still…), but needs a pre-specified number of clusters.

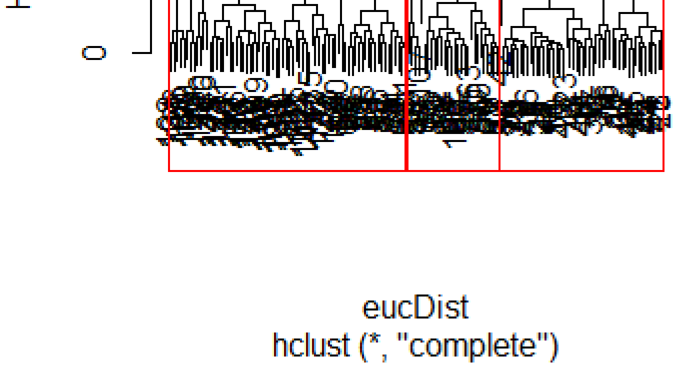
```
# Exercise 9.1: Create a cross table for the 'hit-rate' of the k-means algorithm.
```

9.2 Hierarchical Cluster Analysis

Hierarchical clustering allows to choose different cut-off points.

```
eucDist <- dist(iris[,-5])
hc <- hclust(eucDist)
plot(hc)
rect.hclust(hc, k=3)
```





The function `cutree()` allows us to cut the tree at a specified height (`h`) or to specify the number of groups (`k`).

```
table(iris[,5], cutree(hc, k=3))
```

```
##
##               1  2  3
##   setosa     50  0  0
##   versicolor  0 23 27
##   virginica   0 49  1
```

`hclust()` can take very long, because it uses ‘complete linkage’ as default method, where cluster distance between two clusters is the maximum distance between individual components. Faster solution: `rpuHclust` from the `rpud` -package.

10 Decision Trees

10.1 Overview

- `ctree()` from the `party` package provides conditional inference trees, recursive partitioning for continuous, censored, ordered, nominal and multivariate response variables in a conditional inference framework
- model-based recursive partitioning can also be done with the `party` package using the function `mob()`
- `rpart()` from the same-named package performs recursive partitioning and regression trees
- New hotness: C5.0 Decision Trees (package `C50`) not covered (yet) in this course

The following packages provide functions for sophisticated classification with R:

- Decision trees: `party` , `rpart` , `tree`

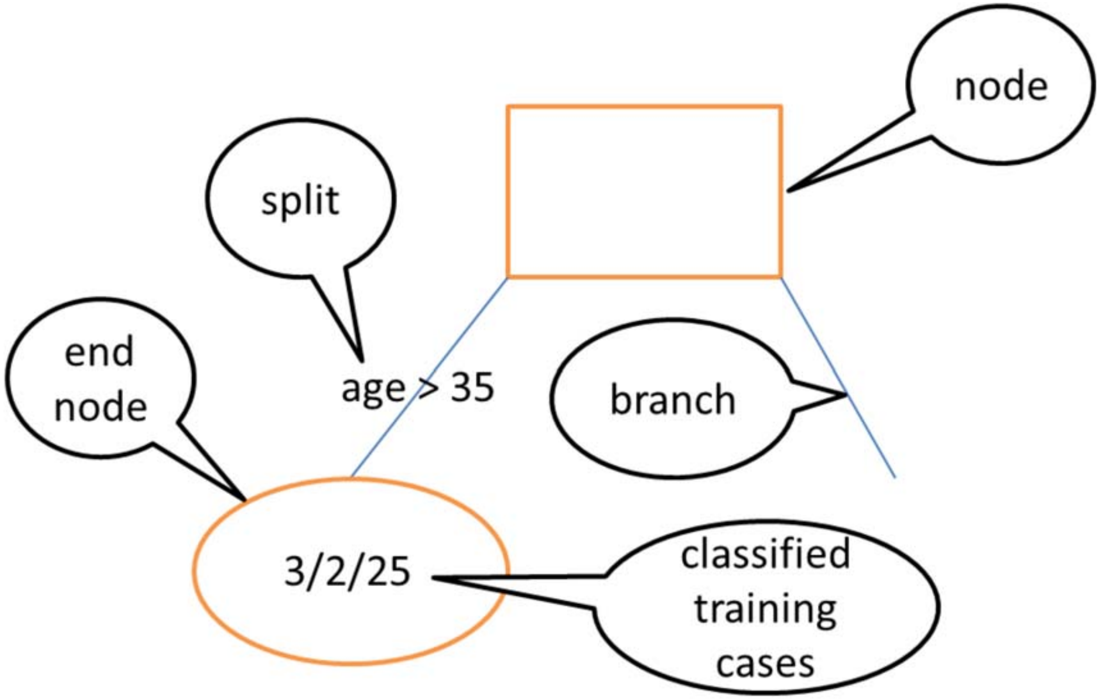
- Ensemble Methods: `party` , `randomForest`
- Support Vector Machines: `e1071` , `kernlab`
- Neural networks: `nnet` , `neuralnet` , `RSNNS`

SVN’s and Neural Networks will not be covered in the following.

- Objective: choose a threshold of an explanatory variable such that some split criterion is minimized (occasionally maximized), e.g. the *deviance*.
- fitted using binary recursive partitioning
- $\Rightarrow$ successive splits into subgroups until break-off criterion: +1. subgroup size $< n1$ (minimum size to attempt further split) +2. improvement of break-off criterion too low (and/or size of one of the created subgroups will be $< n2$)
- Relevant packages: `tree` with function `tree` (oldie but goldie), `rpart` with function `rpart()` and `party` with function `ctree()` .
- General note on tree models in R: the `rpart()` function in the `rpart` -package uses ANOVA with two-level factors and yields more balanced results than `tree()` from the `tree` -package.

10.2 Some Fundamentals

- Developed to analyze large data sets ($\rightarrow$ ‘Data Mining’)
- Group of methods (e.g. CART) which classify a given data set using Boolean arguments: AND, OR, NOT in combination with $=$, $\neq$, $<$, $\leq$, $>$, $\geq$.
- *Tree models*, because the *trunk* (original sample) is split into several *branches*, subbranches (homogenous subsamples), and *leafs* (terminal nodes)



10.2.1 Classification trees

- If the response is categorical, the end node is displayed as $n(y_1)/n(y_2)/\dots/n(y_k)$, and we use the term ‘classification tree’
- Predictions for Y are made by simply taking the mode of all c values in a particular end node

10.2.2 Regression trees

- If the response is metric-scale, the end node is a predicted value, and we use the term ‘regression tree’ instead of ‘classification tree’ for categorical variables
- Predictions for Y are made by simply taking the mean of all c values in a particular end node

10.2.3 Pruning

- Aims to avoid ‘overfitting’ (just like defining break-off criterions)
- Data are split into *training*, *validation*, and *test* group
- Several pruning variants:

    - reduced error pruning
    - minimal cost complexity pruning
    - rule post pruning
- Problem: Loss of information by defining two additional groups (*validation* and *test*) not used in the model-building process

10.3 The tree-package

- `tree()` not as ‘modern’ and up-to-date as `rpart` or `party` functions
- default break-off: $mincut = 5, minsize = 10, mindev = 0.01$
    - `mincut` The minimum number of observations to include in either child node
    - `minsize` The smallest allowed node size
    - `mindev` The within-node deviance must be at least this times that of the root node for the node to be split
- handles missing data in the covariates by ‘dropping them down the tree’

The following example is partially taken from Crawley (2013)

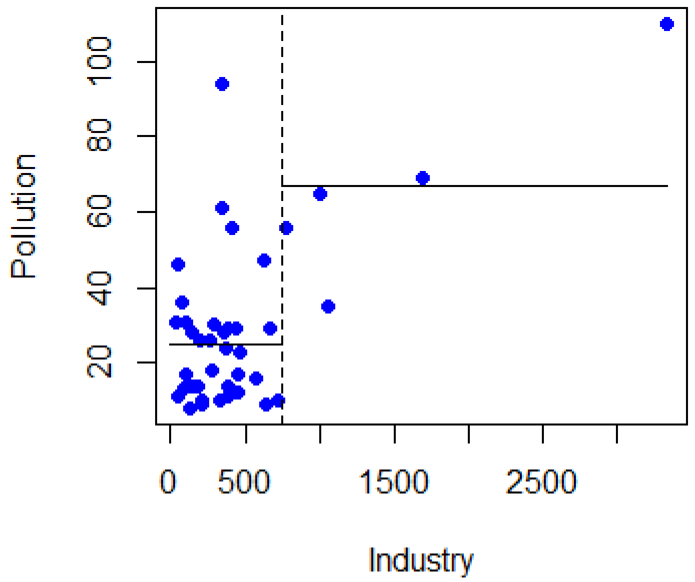
Task: split the data by ‘Industry’ with the objective to minimize the deviance of ‘Pollution’ within the new subsamples

The following syntax demonstrates how `tree()` arrived at the split decision at $Industry < 748$.

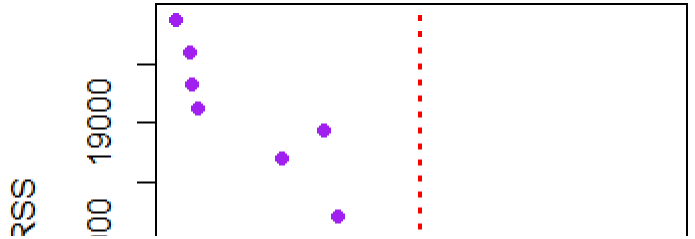
```
library(tree)
Pollute <- read.table("C:/Uni/Daten/Crawley/Pollute.txt", header=TRUE)
attach(Pollute)
tapply(Pollution, Industry < 748, mean)
```

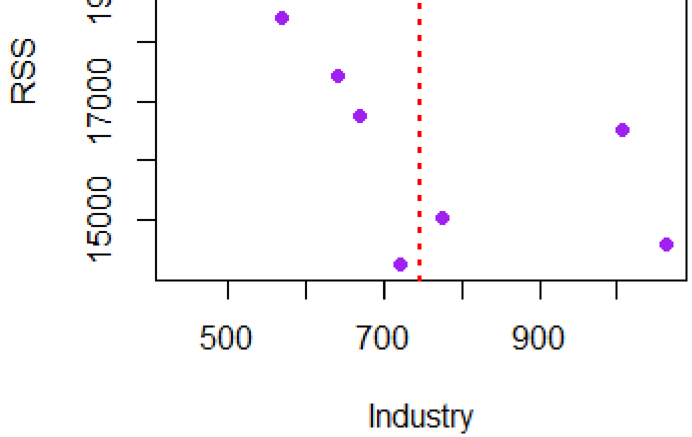
```
##    FALSE     TRUE
## 67.00000 24.91667
```

```
plot(Industry, Pollution, col="blue", pch=16)
abline(v=748, lty=2)
lines(c(0,748), c(24.92, 24.92))
lines(c(748, max(Industry)), c(67, 67))
```



```
indval <- sort(unique(Industry))[28:39]
res <- numeric(length(indval))
for (i in seq(along=indval)) {
  res[i] <- sum(aov(Pollution ~ Industry <= indval[i])$residuals^2)
}
plot(indval, res, pch=20, cex=1.5, col="purple", xlab="Industry", ylab="RSS")
abline(v=748, col="red", lwd=2, lty=3)
```



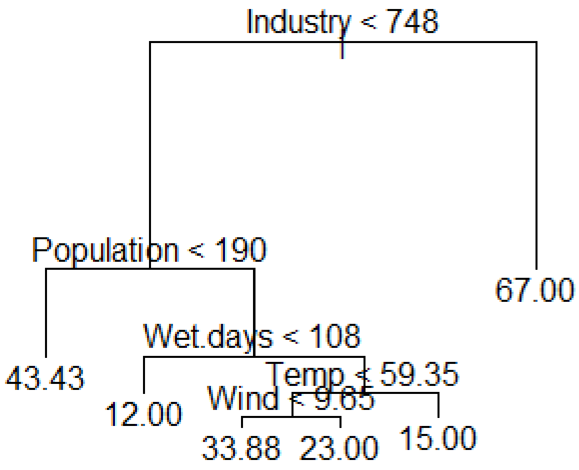


```
detach(Pollute)

tree(Pollution~Industry, data=Pollute)
```

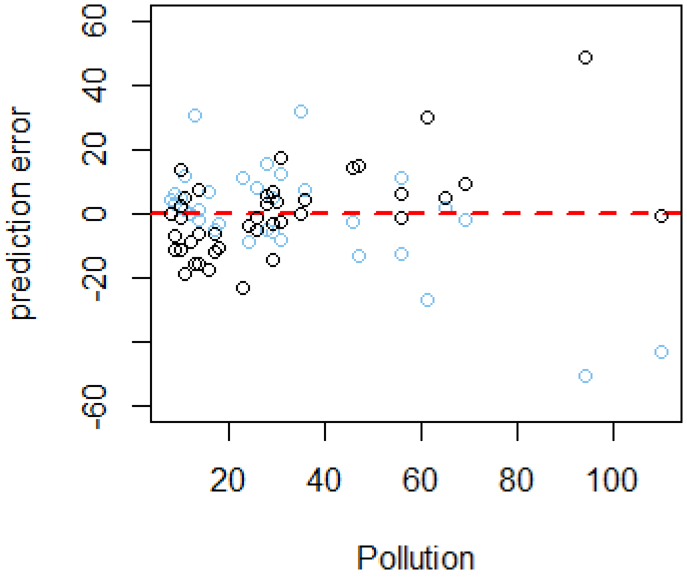
```
## node), split, n, deviance, yval
##       * denotes terminal node
##
##  1) root 41 22040.0 30.05
##    2) Industry < 748 36 11260.0 24.92
##      4) Industry < 340 19  2187.0 20.42
##        8) Industry < 100 6   920.0 28.00 *
##        9) Industry > 100 13   762.9 16.92 *
##      5) Industry > 340 17  8261.0 29.94
##       10) Industry < 380 5  3619.0 47.20 *
##       11) Industry > 380 12  2532.0 22.75 *
##    3) Industry > 748 5  3002.0 67.00 *
```

Estimating and plotting the full model for Pollution

```
model <- tree(Pollute)
plot(model)
text(model)
```

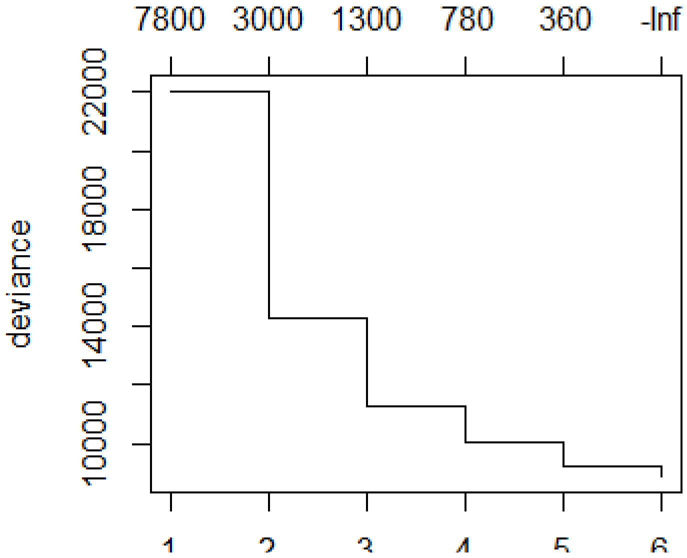


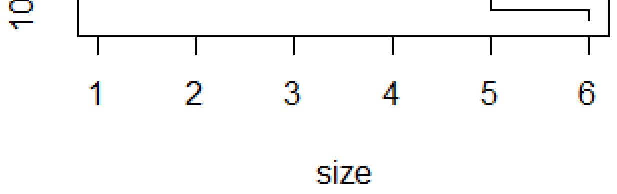
```
# EXERCISE 10.1:
# a) Plot Pollution against its prediction error from the tree model (use [-60;60] as bounds for the y axis),
# and add a straight line at y=0 to the plot.
# b) For comparison reasons add the residuals from a linear model of Pollution on all other variables
# to the plot.

plot(Pollute$Pollution, predict(model)-Pollute$Pollution, pch=21, col="skyblue2",
     xlab="Pollution", ylab="prediction error", ylim=c(-60, 60))
abline(h=0, col="red", lwd=2, lty=2)
linmod <- lm(Pollution~., data=Pollute)
points(Pollute$Pollution, linmod$residuals)
```



`prune.tree()` performs reduced error and minimal cost complexity pruning (default).

```
prmodel <- prune.tree(model)
plot(prmodel)
```

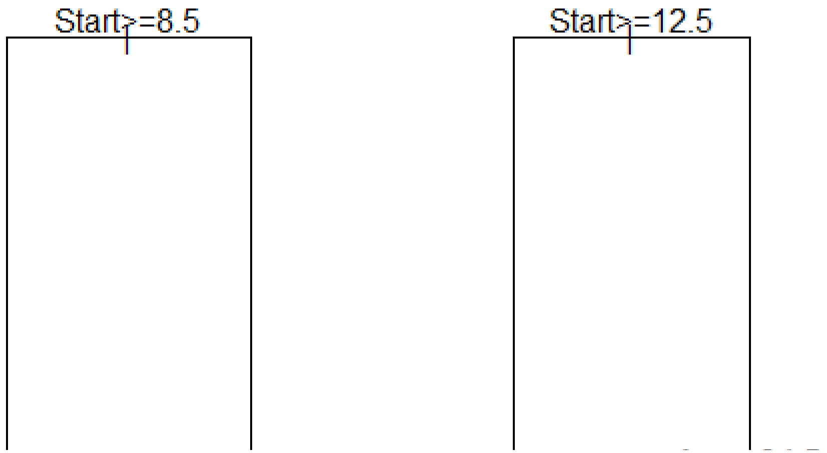


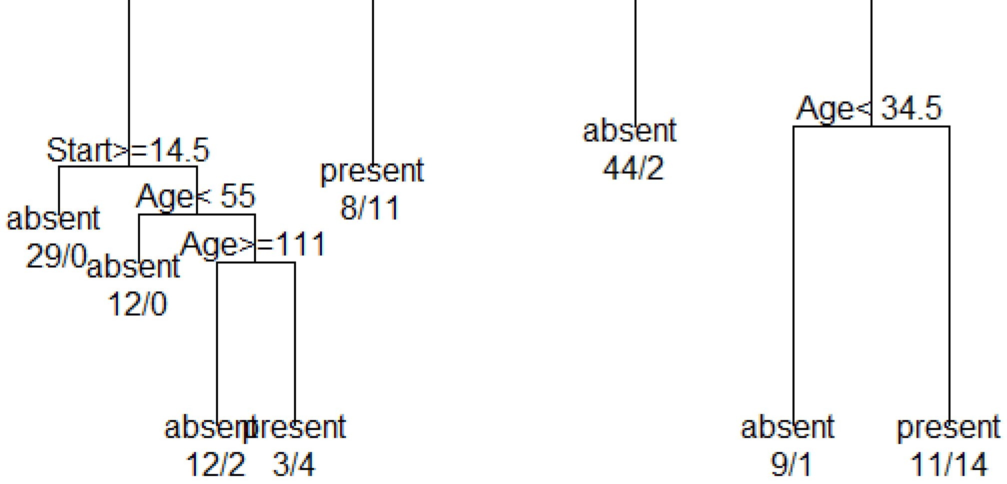


10.4 The rpart-package

Example: Presence of a spinal deformation after surgery for children

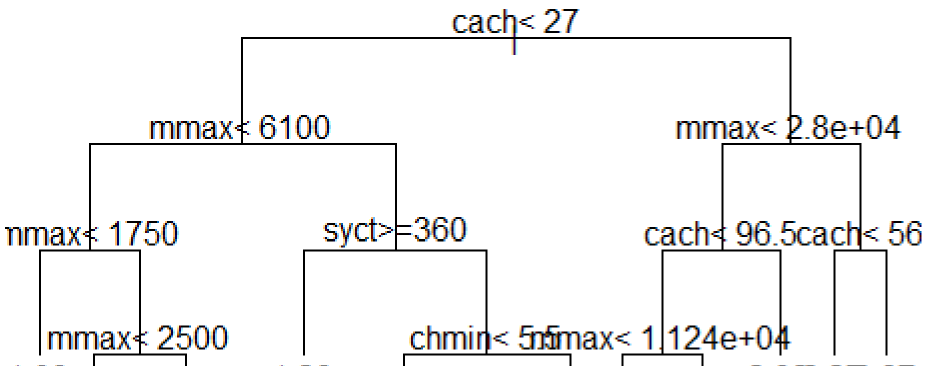
```
library(rpart)
fit <- rpart(Kyphosis ~ Age + Number + Start, data = kyphosis)
fit2 <- rpart(Kyphosis ~ Age + Number + Start, data = kyphosis,
              parms = list(prior = c(.65,.35), split = "information"))
fit3 <- rpart(Kyphosis ~ Age + Number + Start, data = kyphosis,
              control = rpart.control(cp = 0.05))
par(mfrow = c(1,2), xpd = NA) # otherwise on some devices the text is clipped
plot(fit)
text(fit, use.n = TRUE)
plot(fit2)
text(fit2, use.n = TRUE)
```

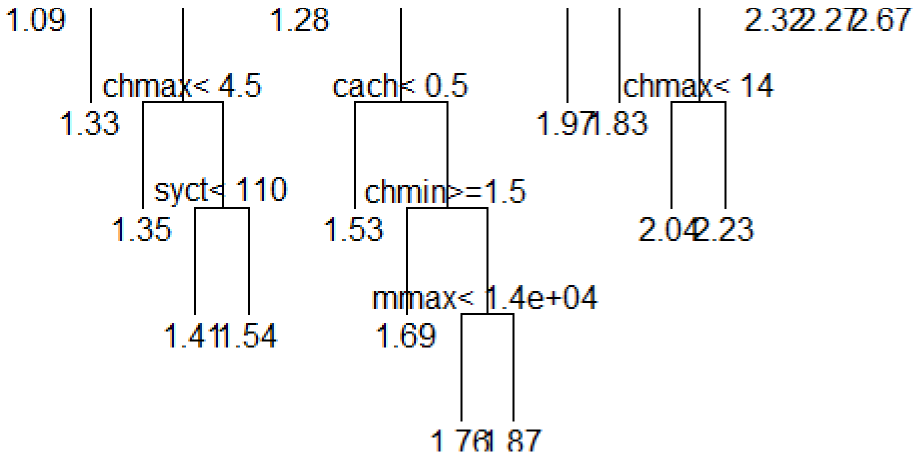




Another example from the MASS-book by Venables and Ripley (2004):

```
library(MASS)
data(cpus)
cpus.rp <- rpart(log10(perf)~., cpus[, 2:8], cp=1e-3)
plot(cpus.rp, uniform=TRUE)
text(cpus.rp, digits=3)
```

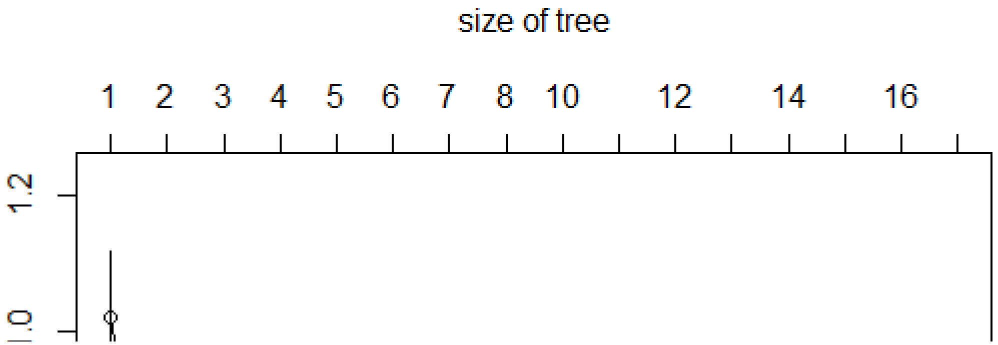


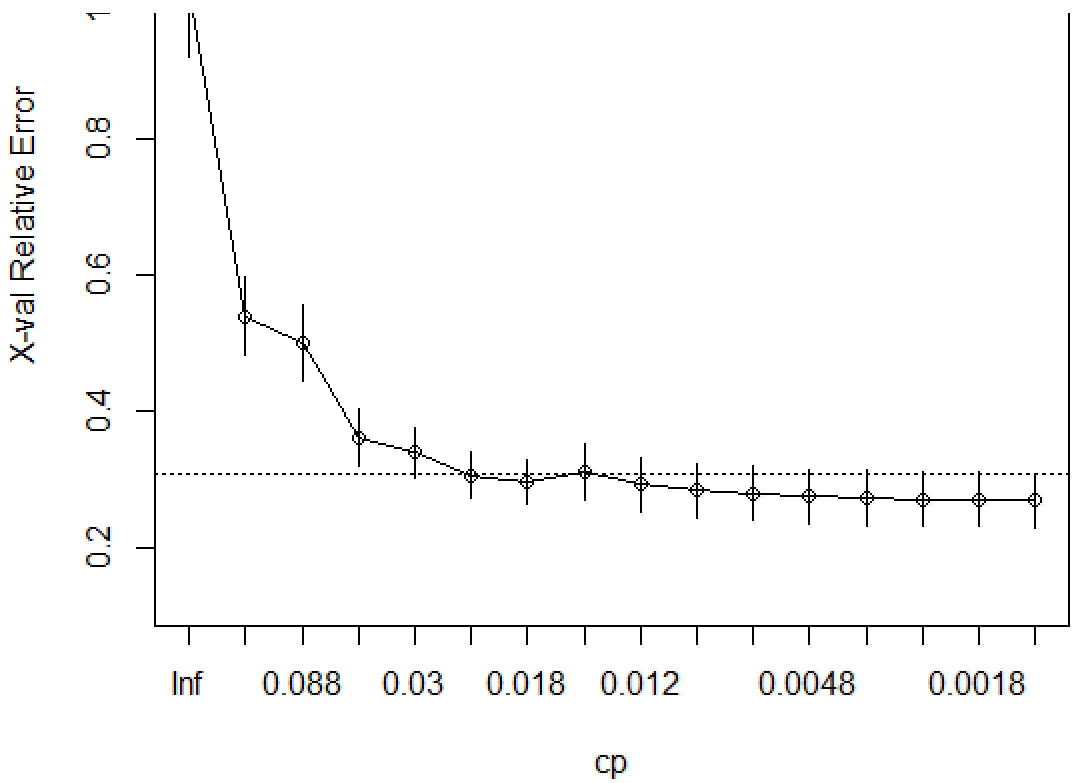


```
print(cpus.rp, cp=0.01)
```

```
## n= 209
##
## node), split, n, deviance, yval
##       * denotes terminal node
##
##  1) root 209 43.1155400 1.753333
##    2) cach< 27 143 11.7908500 1.524647
##      4) mmax< 6100 78  3.8937440 1.374824
##        8) mmax< 1750 12  0.7842516 1.088732 *
##        9) mmax>=1750 66  1.9487330 1.426840 *
##      5) mmax>=6100 65  4.0452030 1.704434
##       10) syct>=360 7  0.1290809 1.279749 *
##       11) syct< 360 58  2.5012470 1.755690
##         22) chmin< 5.5 46  1.2262290 1.698613 *
##         23) chmin>=5.5 12  0.5507131 1.974483 *
##    3) cach>=27 66  7.6426350 2.248821
##      6) mmax< 28000 41  2.3414170 2.061986
##       12) cach< 96.5 34  1.5919510 2.008124
##         24) mmax< 11240 14  0.4246237 1.826635 *
##         25) mmax>=11240 20  0.3834013 2.135166 *
##       13) cach>=96.5 7  0.1717302 2.323601 *
##      7) mmax>=28000 25  1.5228630 2.555230
##       14) cach< 56 7  0.0692943 2.268365 *
##       15) cach>=56 18  0.6535127 2.666788 *
```

```
plotcp(cpus.rp)
```





```
printcp(cpus.rp)
```

```
##
## Regression tree:
## rpart(formula = log10(perf) ~ ., data = cpus[, 2:8], cp = 0.001)
##
## Variables actually used in tree construction:
## [1] cach  chmax chmin mmax  syct
##
## Root node error: 43.116/209 = 0.20629
##
## n= 209
##
##           CP nsplit rel error  xerror     xstd
## 1  0.5492697      0   1.00000 1.01888 0.098105
## 2  0.0893390      1   0.45073 0.53855 0.057430
## 3  0.0876332      2   0.36139 0.49853 0.056447
## 4  0.0328159      3   0.27376 0.36017 0.041239
## 5  0.0269220      4   0.24094 0.33852 0.036175
## 6  0.0185561      5   0.21402 0.30489 0.033818
## 7  0.0167992      6   0.19546 0.29612 0.032708
## 8  0.0157908      7   0.17866 0.31023 0.041125
## 9  0.0094604      9   0.14708 0.29130 0.040153
## 10 0.0054766     10   0.13762 0.28293 0.040156
## 11 0.0052307     11   0.13215 0.27881 0.039949
## 12 0.0043985     12   0.12692 0.27378 0.039936
## 13 0.0022883     13   0.12252 0.27193 0.040049
## 14 0.0022704     14   0.12023 0.27017 0.039999
## 15 0.0014131     15   0.11796 0.26977 0.039997
## 16 0.0010000     16   0.11655 0.26855 0.039994
```

10.5 The party-package

Example: Fisher’s iris data set revisited

```
# iris data
str(iris)
```

```
## 'data.frame':    150 obs. of  5 variables:
##  $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
##  $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
##  $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
##  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
##  $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
```

```
set.seed(1234)
ind <- sample(2, nrow(iris), replace=T, prob=c(0.7, 0.3))
iris.train <- iris[ind==1, ]
iris.test <- iris[ind==2, ]
library(party)
```

```
## Loading required package: mvtnorm
```

```
## Loading required package: modeltools
```

```
## Loading required package: stats4
```

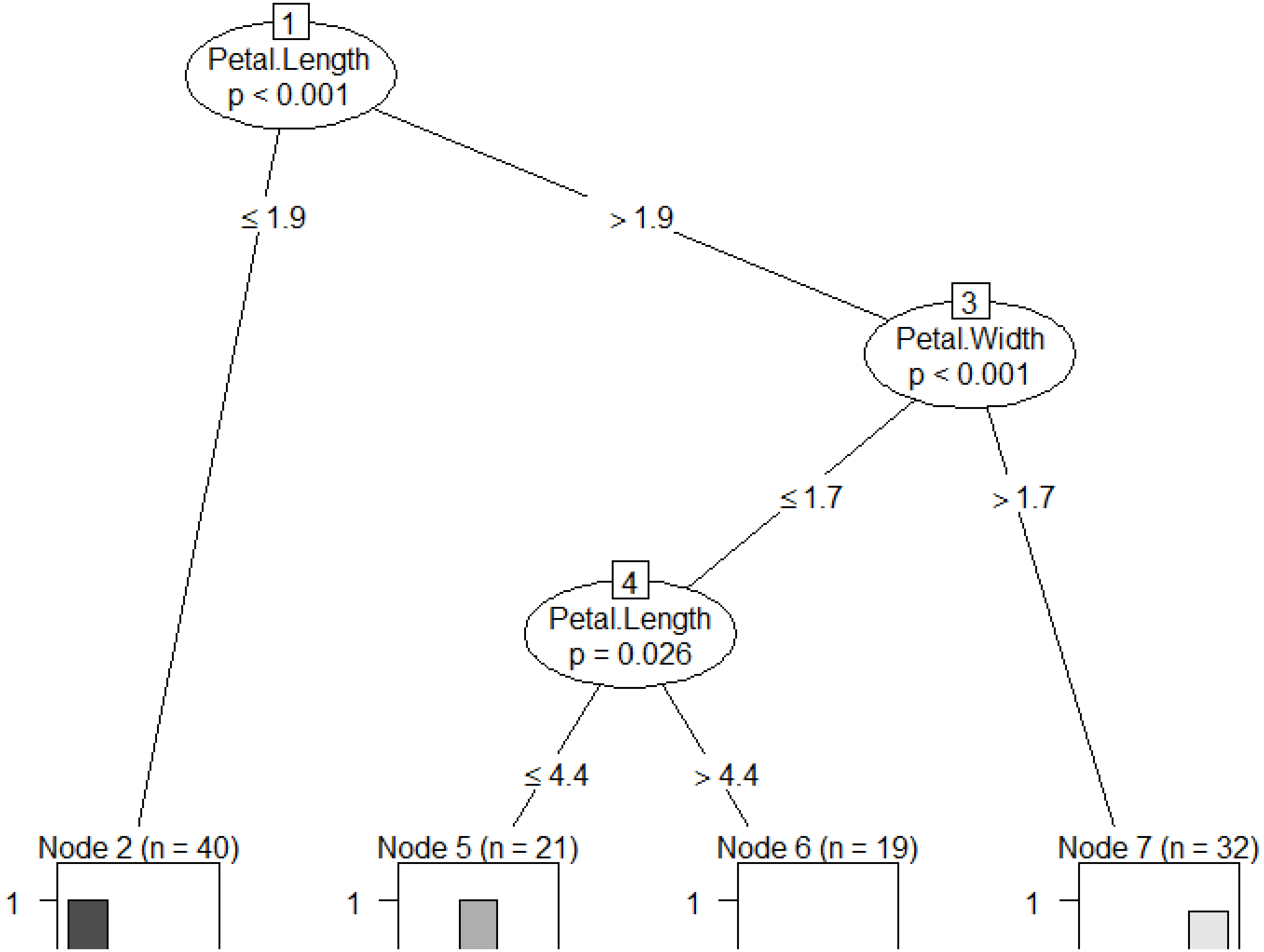
```
## Loading required package: strucchange
```

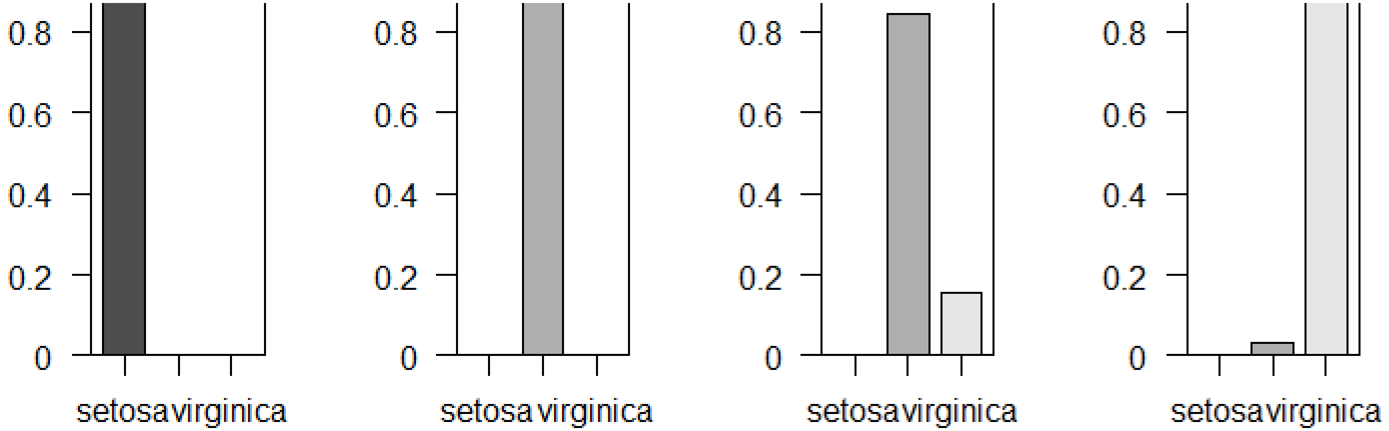
```
## Loading required package: zoo
```

```
##
## Attaching package: 'zoo'
```

```
## The following objects are masked from 'package:base':
##
##     as.Date, as.Date.numeric
## Loading required package: sandwich
```

```
iris.formula <- Species ~ Sepal.Length + Sepal.Width +
Petal.Length + Petal.Width
iris.ctree <- ctree(iris.formula, data=iris.train)
plot(iris.ctree)
```





```
#EXERCISE 10.2: Load the iris data set.
#Take the first 100 observations of the data to get splits (min end node size: 9).
#Apply split criterions to the remaining 50 observations of the data set.
#Compare the classification success rate for the first 100 observations (training sample) with the bottom 50 (test
sample).
#(Hint: use predict() for classification)
```

10.6 Limitations of tree-based algorithms

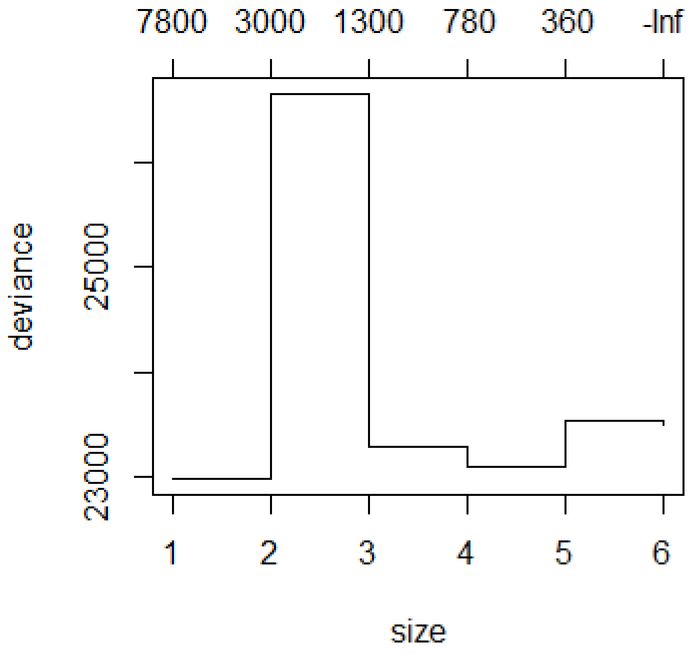
- no functions between covariates and response
- most algorithms work with one-step lookahead
- no sensible way of handling missing values
- danger of overfitting (especially if end node size is small)

10.7 Cross-Validation

- objective: prevent overfitting
- take subsample (training sample) for building the model and apply…
- …to the remainder of the sample (test sample)
- CV is alternative to *pruning* (and sometimes used in comnbination)

`cv.tree()` from the `tree` package performs K-fold (default: K=10) cross-validation to identify the deviance (for regression trees) or number of wrong classifications (classification trees) as a function of the cost complexity parameter k.

```
cvmod <- cv.tree(model, ,prune.tree)
plot(cvmod)
```



Unfortunately, `cv.tree()` apparently offers no option for extracting the cross-validated predictions.

The `rpart()` function, however, automatically provides (10-fold) cross-validation (see `rpart.control()` for details), and the function `xpred.rpart()` extracts those predictions as matrix.

```
weatherdat <- read.table("C:/Uni/Daten/Crawley/oxford weather.txt", header=TRUE)

summary(weatherdat$meantemp)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
##  -3.000   5.850   9.550   9.987  14.550  21.100
```

```
weatherdat$above10 <- weatherdat$meantemp > 10

wmod1 <- rpart(above10 ~ year + month + tmax + tmin + raindays + rain, data=weatherdat)
xvalWmod1 <- xpred.rpart(wmod1)
head(xvalWmod1)
```

```
##   0.94796113 0.20340994 0.03800104  0.01768310
## 1  0.4797093 0.01882353 0.01882353 0.001203369
## 2  0.4724409 0.01746217 0.01746217 0.001189061
## 3  0.4733333 0.02204176 0.02204176 0.002378121
## 4  0.4787879 0.02227433 0.02227433 0.000000000
## 5  0.4763636 0.96737767 0.99354839 0.993548387
## 6  0.4787879 0.96737767 0.96737767 1.000000000
```

10.8 Diagnostics and Performance Evaluation

Receiver Operationg Characteristic (ROC) curves are a widespread diagnostic to evaluate the performance of a (binary) classification algorithms. The basic idea is a line-connected scatterplot of the false-positive rate (FP/N) vs. the true-positive rate (TP/T). Sometimes *Accuracy* (ACC), i.e. `(sum of true positives + sum of true negatives)/Total number` is plotted as well.

The package `ROCR` offers many functions for graphical diagnostics, but the vanilla mode is:

- get predictions from your model and (binary) label of the true values and feed both as arguments to function `prediction()`
- The function `performance()` allows the specification of the diagnostics, and generates an output object of class `performance` which can be plotted

```
iris$versicolor <- iris$Species == "versicolor"
rpmodel <- rpart(versicolor ~ Sepal.Length + Sepal.Width + Petal.Length + Petal.Width, data = iris)

table(predict(rpmodel), iris$versicolor)
```

```
##
##                       FALSE TRUE
##   0                      50    0
##   0.0217391304347826    45    1
##   0.5                    4    4
##   0.978260869565217      1   45
```

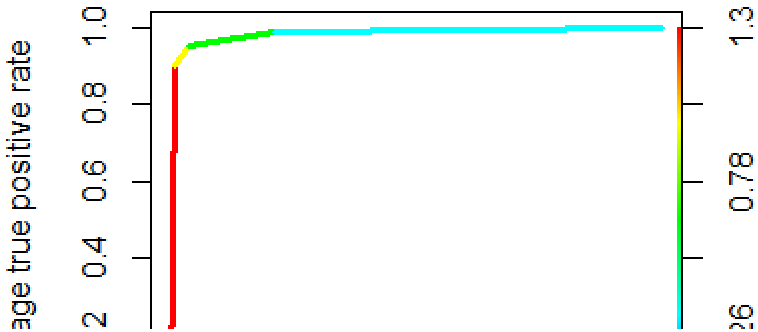
```
library(ROCR)
```

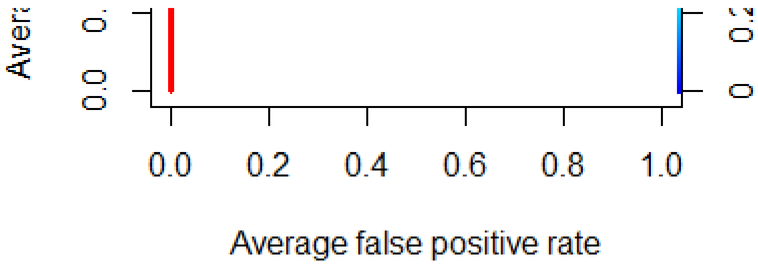
```
## Loading required package: gplots
```

```
##
## Attaching package: 'gplots'
```

```
## The following object is masked from 'package:stats':
##
##     lowess
```

```
test <- prediction(predict(rpmodel), iris$versicolor)
plot(performance(test, measure="tpr", x.measure="fpr"),avg= "threshold", colorize=T, lwd= 3)
```





```
#EXERCISE 10.3: Use the cross-validation predictions from 'xvalWmod1'
#and plot the accuracy, and the corresponding standard errors for each decentile (from 0.1 to 0.9).
```

11 Ensemble Methods

- `cforest()` from the `party` package
- `randomForest()` from the eponymous package
- `importance()` (`randomForest` package) and `varimp()` (`party` package) to determine variable importance

11.1 Bagging

- Bootstrap Aggregating (Bagging), (Breiman 1994)

- Bagging is a so-called *ensemble* method (usage of multiple models). Other ensembles comprise Boosting and Bayesian Model Averaging.

Bagging algorithm:

1. draw non-parametric bootstrap sample N_{BB} (training sample) from the original sample N [side note: $E(n(N_{BB})) = (1-1/e)N$]
2. apply classification/regression trees to bootstrap sample
3. (use remaining cases for cross-validation $\rightarrow$ ‘out-of-bag error’)
4. repeat steps (1) and (2) J times
5. final prediction is maximum (classification tree) or mean (regression tree) of combined J models

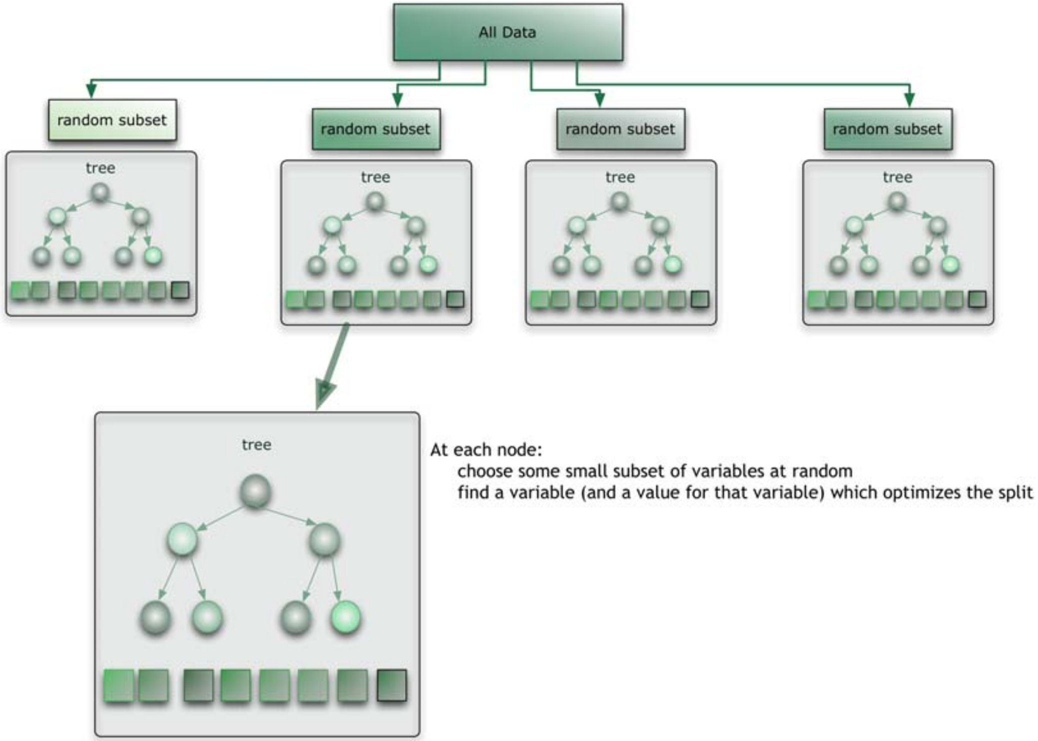
11.2 Random Forest

- Random Forest (Breiman 2001)

Suppose you are using a matrix $\mathbf{X} = [X_1, \ldots, X_K]$ as covariates for your classification model.

Random Forest algorithm:

1. apply bagging step to data using the reduced subset of predictors
2. draw (without replacement) $k << K$ variables from $\mathbf{X}$ (reasoning: intercollinearity of the covariates) for each node of the tree
3. repeat steps (1) and (2) J times



At each node:
choose some small subset of variables at random
find a variable (and a value for that variable) which optimizes the split

source: https://citizennet.com/blog/2012/11/10/random-forests-ensembles-and-performance-metrics/ (https://citizennet.com/blog/2012/11/10/random-forests-ensembles-and-performance-metrics/)

In the following we will focus on the `cforest()` function from the `party` package.

The *UC Irvine Machine Learning Repository* (http://archive.ics.uci.edu/ml/ (http://archive.ics.uci.edu/ml/)) provides many data sets that can be used for classification and other statistical analysis.

In order to apply ROC curves, we will recode the response – wine quality – into binary form again.

```
library(party)
Wine <- read.csv("http://archive.ics.uci.edu/ml/machine-learning-databases/wine-quality/winequality-white.csv", he
ader=TRUE,
                 sep=";")
Wine$good <- Wine$quality > 6
table(Wine$good)
```

```
## 
## FALSE  TRUE 
##  3838  1060
```

```
head(Wine)
```

```
##   fixed.acidity volatile.acidity citric.acid residual.sugar chlorides
## 1           7.0             0.27        0.36           20.7     0.045
## 2           6.3             0.30        0.34            1.6     0.049
## 3           8.1             0.28        0.40            6.9     0.050
## 4           7.2             0.23        0.32            8.5     0.058
## 5           7.2             0.23        0.32            8.5     0.058
## 6           8.1             0.28        0.40            6.9     0.050
##   free.sulfur.dioxide total.sulfur.dioxide density   pH sulphates alcohol
## 1                  45                  170  1.0010 3.00      0.45     8.8
## 2                  14                  132  0.9940 3.30      0.49     9.5
## 3                  30                   97  0.9951 3.26      0.44    10.1
## 4                  47                  186  0.9956 3.19      0.40     9.9
## 5                  47                  186  0.9956 3.19      0.40     9.9
## 6                  30                   97  0.9951 3.26      0.44    10.1
##   quality  good
## 1       6 FALSE
## 2       6 FALSE
## 3       6 FALSE
## 4       6 FALSE
## 5       6 FALSE
## 6       6 FALSE
```

```
Wine2 <- Wine[,names(Wine) != "quality"]
set.seed(201507)
train <- sample(nrow(Wine2), round(0.9*nrow(Wine2)))
test <- setdiff(1:nrow(Wine2), train)
```

After creating a training and a test set, we compare the performance of the default `ctree()` with the Random Forest (RF) function `cforest()` .

Note that the default for the number of chosen predictors is 5 for `cforest()` , and is – unlike to `randomForest()` , not flexibly chosen in relation to the number of predictors.

```
myTree <- ctree(good~., data=Wine2, subset=train)
predTree <- predict(myTree, newdata=Wine2[test,])
head(as.numeric(predTree))
```

```
## [1] 0.10801394 0.01515152 0.08219178 0.15891473 0.15891473 0.33333333
```

```
system.time({
  myForest <- cforest(good~., data=Wine2, subset=train)
})
```

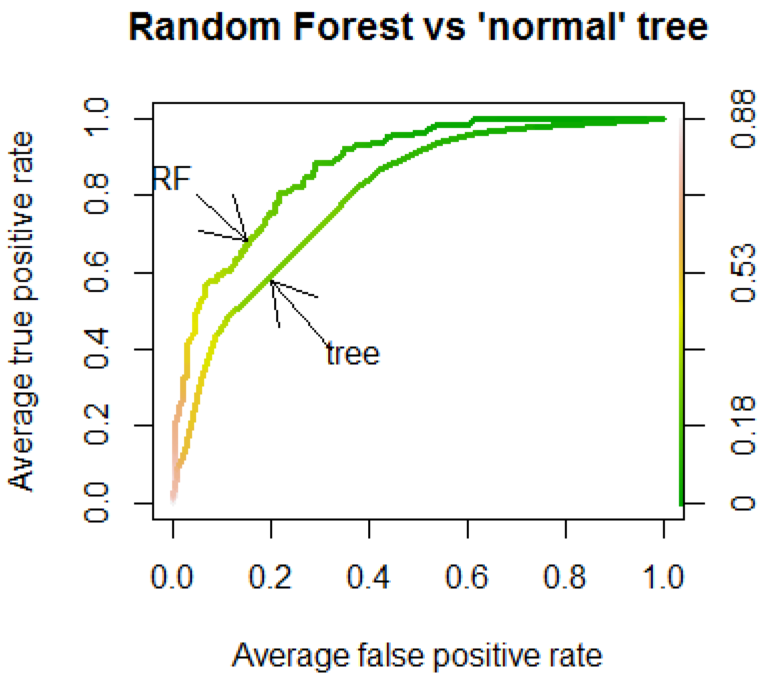
```
##    user  system elapsed
##   31.89    0.06   32.00
```

```
predRF <- treeresponse(myForest, newdata=Wine2[test,])
predRFraw <- sapply(predRF, function(x) x[[1]])
head(predRFraw)
```

```
##          55          64          65          69          75          82
## 0.048079246 0.001430099 0.037613344 0.229752066 0.229752066 0.074645972
```

The results are finally compared using ROC curves.

```
inputRF <- prediction(predRFraw, Wine2[test, "good"])
inputTR <- prediction(as.numeric(predTree), Wine2[test, "good"])
plot(performance(inputRF, measure="tpr", x.measure="fpr"),avg= "threshold", colorize=T, colorize.palette=terrain.c
olors(256),lwd= 3, main = "Random Forest vs 'normal' tree")
plot(performance(inputTR, measure="tpr", x.measure="fpr"),avg= "threshold", colorize=T, colorize.palette=terrain.c
olors(256),lwd= 3, add=TRUE)
text(x=0, y=0.85, labels="RF")
arrows(x0=0.05,x1=0.15, y0=0.8, y1=0.68)
text(x=0.37, y=0.4, labels="tree")
arrows(x0=0.32,x1=0.2, y0=0.4, y1=0.58)
```



```
#EXERCISE 11.1: Re-run the Random Forest with the choice for k, the
#number of selected predictors per tree, being the rounded squareroot
#of the total number of predictors.
#Create a lift curve for the previous RF result and this update
#(within one plot).
```