1 Introduction

1.1 Statistical software

A comparison of statistical software

A personal opinion:
• One advantage of a “free” (as in free beer) software is that it is portable. When you move to a new job you can still use your old work.

• Some of the commercial packages include a lot of documentation which is very well structured and follows a consistent pattern. The documentation that comes with R is sometimes less well structured. However, a few books can make up for this very soon.

• Correctness is always a problem, but it seems to be a smaller problem with R. The following example illustrates:

A Wilcoxon rank-sum test in R

```r
wilcox.test(x ~ group)
```

```
Wilcoxon rank sum test

data:  x by group
W = 5, p-value = 0.04798
alternative hypothesis: true location shift is not equal to 0
```

A Wilcoxon rank-sum test in Stata

```
.ranksum x ,by(group)
```

```
Two-sample Wilcoxon rank-sum (Mann-Whitney) test

group | obs  rank sum  expected
-------------+----------------------------------
   1 | 5 20 32.5
   2 | 7 58 45.5
-------------+----------------------------------
combined | 12 78 78

unadjusted variance 37.92
```
adjustment for ties 0.00

 adjusted variance 37.92

Ho: \( x(\text{group==1}) = x(\text{group==2}) \)

\[ z = -2.030 \]

\[ \text{Prob } > |z| = 0.0424 \]

The two \( p \)-values from R and Stata differ. R uses the exact method, Stata uses an approximation.

1.2 Ways to interact with R

There are lots of possible front ends:

- Figure 1 shows RStudio.
- Figure 2 shows the RCommander.
- Figure 3 shows the command line.

1.3 The help system

From the command line we might have to say \texttt{help.start()} to start the interactive help system.

**Getting help for R**

\texttt{help.start()}

\texttt{RSiteSearch} searches R functions, package vignettes, and task views (regardless whether they are locally installed).

\texttt{RSiteSearch("probit")}

\texttt{help.search} searches the help system for a given topic:

\texttt{help.search("probit")}

This will give you a list of Vignettes, demos, and functions. Use \texttt{vignette} to show a specific vignette.
Figure 1: The RStudio Interface

```
# Load the diamonds dataset
library(ggplot2)
view(diamonds)
summary(diamonds)

average <- round(mean(diamonds$carat), 4)
clarity <- levels(diamonds$clarity)

# Plot carat vs. price
p <- ggplot(carat, price, 
data=diamonds, color=clarity, 
  xlab="carat", ylab="price", 
  main="Diamond Pricing")

# Show the dataset summary
summary(diamonds$price)
```
Figure 2: The RCommander Interface
R version 3.0.2 (2013-09-25) -- "Frisbee Sailing"
Copyright (C) 2013 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> 2+2
[1] 4
> |

Figure 3: The command-line interface
vignette("probit", package = "Zelig")

Use demo to play a demo.

library(Zelig)
demo(Zelig::probit)

To learn about the syntax of a specific function, use help or ?

? glm
help("glm")

# 2 Datatypes

## 2.1 Overview

To get a first idea, let us load one of the many builtin datasets, and have a look at the first three rows of this dataset:

**A data frame**

data(Wages, package = "Ecdat")

<table>
<thead>
<tr>
<th>exp</th>
<th>wks</th>
<th>bluecol</th>
<th>ind</th>
<th>south</th>
<th>smsa</th>
<th>married</th>
<th>sex</th>
<th>union</th>
<th>ed</th>
<th>black</th>
<th>lwage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>no</td>
<td>0</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>male</td>
<td>no</td>
<td>9</td>
<td>no</td>
<td>5.56</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>no</td>
<td>0</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>male</td>
<td>no</td>
<td>9</td>
<td>no</td>
<td>5.72</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>no</td>
<td>0</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>male</td>
<td>no</td>
<td>9</td>
<td>no</td>
<td>6.00</td>
</tr>
</tbody>
</table>

lm(lwage ~ ed, data = Wages)

Call:
lm(formula = lwage ~ ed, data = Wages)

Coefficients:
(Intercept)       ed
      5.8388    0.0652

library(lattice)

xyplot(lwage ~ ed, data = Wages, type = "smooth")
Some primitive datatypes:

- numbers: 3, 42, 3.141592…
- logicals: TRUE, FALSE
- characters: "Jena", "Bonn", …
- factors: "Jena", "Bonn", …
- formulas: \( y \sim x + z \)
- functions: \( \text{function(x) } x^2 \)

**Combining data**

We can combine data of the *same type*:

- Vector (several members of the same type): \((3, 42, 3.1415926\ldots)\)
- Matrix (rectangular):
  \[
  \begin{pmatrix}
  3 & 42 \\
  3.14 & 0
  \end{pmatrix}
  \]
- Array (multidimensional):
  \[
  \left( \begin{pmatrix}
  3 & 42 \\
  3.14 & 0
  \end{pmatrix}, \begin{pmatrix}
  12 & 16 \\
  21 & 11
  \end{pmatrix} \right)
  \]

We can also combine data of *different types*:

- List: \((3, \text{"Jena"}, \text{FALSE}, \ldots)\)
- Data frame (rectangular, different types per column)

<table>
<thead>
<tr>
<th></th>
<th>exp</th>
<th>wks</th>
<th>bluecol</th>
<th>ind</th>
<th>south</th>
<th>smsa</th>
<th>married</th>
<th>sex</th>
<th>union</th>
<th>ed</th>
<th>black</th>
<th>lwage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>32</td>
<td>no</td>
<td>0</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>male</td>
<td>no</td>
<td>9</td>
<td>no</td>
<td>5.56</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>43</td>
<td>no</td>
<td>0</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>male</td>
<td>no</td>
<td>9</td>
<td>no</td>
<td>5.72</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>40</td>
<td>no</td>
<td>0</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>male</td>
<td>no</td>
<td>9</td>
<td>no</td>
<td>6.00</td>
</tr>
</tbody>
</table>
2.2 Numbers

We see, R handles very different datatypes: numbers, characters, . . . . Let us start with something simple, with numbers. We can do usual arithmetic like this:

**Numeric: simple arithmetic**

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 + 3</td>
<td>[1] 5</td>
</tr>
<tr>
<td>2 - 3</td>
<td>[1] -1</td>
</tr>
<tr>
<td>2 * 3</td>
<td>[1] 6</td>
</tr>
<tr>
<td>2/3</td>
<td>[1] 0.6667</td>
</tr>
<tr>
<td>2^3</td>
<td>[1] 8</td>
</tr>
<tr>
<td>2^3 + 1/3</td>
<td>[1] 8.333</td>
</tr>
<tr>
<td>(2^3 + 1)/3</td>
<td>[1] 3</td>
</tr>
<tr>
<td>7 &lt; 11</td>
<td>[1] TRUE</td>
</tr>
<tr>
<td>7 == 11</td>
<td>[1] FALSE</td>
</tr>
</tbody>
</table>
The assignment operator: <-
We can assign the result of our calculations to variables with <-. Usually, assignments produce no output.

\[
x <- 2 + 2
\]

We have to repeat the name of the variable to which we assigned our result to see something:

\[
x
\]

[1] 4

Names of variables
Variable names contain a-z, A-Z, 0-9 and . and _ and start with a letter, e.g.

\[
x
groupContribution5
group.contribution
group.contribution
\]

In the example we have seen several operators. Here is a list:

Some operators

- Arithmetic operators: +, -, *, /, ^,...
- Assignment: <-, <<-,->, =
- Logical: ==, <=, <, >, >=
- and more ...(you can define you own operators)

2.3 Using functions

Functions and parameters

A function with one parameter:

\[
\text{log(100)}
\]

[1] 4.605

Calling a function with several named parameters:

\[
\text{log(x = 100, base = 10)}
\]

[1] 2
\begin{verbatim}
log(base = 10, x = 100)
\end{verbatim}

\textbf{[1] 2}

\textbf{Functions and parameters}

Calling a function with several \textbf{positional} parameters:

\begin{verbatim}
log(100, 10)
\end{verbatim}

\textbf{[1] 2}

Mixing \textbf{positional} and \textbf{named} parameters:

\begin{verbatim}
log(base = 10, 100)
\end{verbatim}

\textbf{[1] 2}

\textbf{The arguments of a function}

If we want to find out the arguments of a function:

\begin{verbatim}
args(log)
\end{verbatim}

\texttt{function (x, base = exp(1))}

\texttt{NULL}

We see that the parameter \texttt{base} has a \textit{default} value (\texttt{exp(1)}).
If \texttt{base} is not specified (e.g. we just say \texttt{log(100)}), \texttt{R} assumes \texttt{base=exp(1)}

\textbf{Getting help for a function} We can find out what parameters a function takes and how they are called and in which position they come with the \texttt{help} function.

\begin{verbatim}
help(log)
\end{verbatim}

\texttt{log} \hspace{1cm} \texttt{package:base} \hspace{1cm} \texttt{R Documentation}

\texttt{Logarithms and Exponentials}

Description:

‘\texttt{log}’ computes logarithms, by default natural logarithms, ‘\texttt{log10}’ computes common (i.e., base 10) logarithms, and ‘\texttt{log2}’ computes binary (i.e., base 2) logarithms. The general form ‘\texttt{log(x, base)}’ computes logarithms with base ‘\texttt{base}’.
'log1p(x)' computes log(1+x) accurately also for |x| << 1 (and less accurately when x is approximately -1).

'exp' computes the exponential function.

'expm1(x)' computes exp(x) - 1 accurately also for |x| << 1.

Usage:

log(x, base = exp(1))
logb(x, base = exp(1))
log10(x)
log2(x)
log1p(x)
exp(x)
expm1(x)

Arguments:

x: a numeric or complex vector.

base: a positive or complex number: the base with respect to which logarithms are computed. Defaults to e='exp(1)'.

Details:

References:

See Also:

Examples:

2.4 Vectors
The simplest datatype in R is a vector.

Vectors
Since we need sequences like the above very often, there is a special operator that helps us to create sequences:

```
21:30
```

"::" is nothing else but a shortcut for the function `seq`:

```
seq(21, 30)
```

Descending sequences are done like this:

```
30:21
[1]  30  29  28  27  26  25  24  23  22  21
```

The parameter `by` sets the step width of the sequence:

```
seq(21, 30, by = 2)
[1]  21  23  25  27  29
```

```
seq(21, 22, by = 0.2)
```

```
seq(21, 22, length = 6)
```

2.5 The recycling rule

What happens if we combine two vectors? If both have the same length, the operation is performed for each matching element:

Recycling (when vectors have different lengths)
\begin{verbatim}
x <- c(1, 2, 3)
y <- c(10, 10, 10)
x + y
[1] 11 12 13

What if the two vectors do not have the same length? Then the elements of the shorter vector are \textit{recycled}. This makes in particular sense if one vector has length one:
\end{verbatim}
\begin{verbatim}
x <- c(1, 2, 3)
y <- 10
x + y
[1] 11 12 13
10^(0:3)
[1] 1 10 100 1000
log(c(0.1, 1, 10, 100), base = 10)
[1] -1 0 1 2
\end{verbatim}

Here is an example where one vector has length 3 and the other length 2:
\begin{verbatim}
x <- c(1, 2, 3)
y <- c(10, 20)
x + y
Warning: longer object length is not a multiple of shorter object length
[1] 11 22 13
(1:2)^(0:3)
[1] 1 2 1 8
\end{verbatim}

\textbf{2.6 Arithmetic with vectors}

\textbf{Arithmetic with vectors}
\begin{verbatim}
x <- c(1, 2, 3)
y <- c(3, 4, 5)
x + y
[1] 4 6 8
\end{verbatim}
element-wise product:

```r
x * y
```

```
[1] 3 8 15
```

inner product:

```r
x %*% y
```

```
[,1]
[1,] 26
```

outer product:

```r
x %o% y
```

```
[,1] [,2] [,3]
[1,] 3 4 5
[2,] 6 8 10
[3,] 9 12 15
```

2.7 More on numbers

R distinguishes floating point numbers (double) and integer numbers (integer). The standard type for numbers is double.

We can find out the type of a variable with the help of the function typeof.

Floating point numbers

```r
x <- c(1, 2, 3)
typeof(x)
```

```
[1] "double"
```

double has about 16 significant decimal digits precision.

Largest exact integer number: 9,007,199,254,740,992

For numbers this large, R usually prints the first 7 significant digits:

```r
2^53
```

```
[1] 9.007e+15
```

If we insist, we can see more digits:
print($2^{53}$, digits = 16)

[1] 9007199254740992

**Floating point numbers**

However, adding small quantities to such a large number leads to a rounding error:

print($2^{53} + 1$, digits = 16)

[1] 9007199254740992

Only if we add a larger quantity (here 2), we get a visible increment.

print($2^{53} + 2$, digits = 16)

[1] 9007199254740994

**Floating point numbers**

Even if we accept a rounding error, the is a largest and a smallest number R can represent:

Largest positive real: $1.797693 \cdot 10^{308}$

$2^{1023.99999999}$

[1] 1.798e+308

$2^{1024}$

[1] Inf

Smallest positive real: $4.940656 \cdot 10^{-324}$

$2^{-1074}$

[1] 4.941e-324

$2^{-1075}$

[1] 0

**Integers**

We can *force* a number to be an integer:
```
x <- as.integer(c(1, 2, 3))
typeof(x)

[1] "integer"

Largest integer number: 2,147,483,647

as.integer(2147483647 + 1)

Warning: NAs introduced by coercion

[1] NA

We see that if an integer number becomes “too large” it becomes NA

Functions

- Arithmetic `log`, `exp`, `sin`, ...
- Aggregation:
  - `mean`, `sum`, `sd`, `var`, ...
  - `summary`
- Data:
  - `c`, `seq`:
    - `order(c(2,3,1))=c(3,1,2)`
    - `sort(c(2,3,1))=c(1,2,3)`
    - `sample(c(1,2,3))=c(2,1,3)`
    - `unique(c(1,1,2,2,3)=c(1,2,3)`
    - `which(c(1,1,2,2,3)==2)=c(3,4)`
    - `length(c(1,2,3))=3`
  - `typeof`, `class`, `as.numeric`, `is.numeric`, `as.integer`, ...
- Randomness
  - `runif`, `rnorm`, `rt`, `rf`, `rchi`, `rpoisson`, `rbinom`, ...

3 Not exactly numbers

3.1 Characters

Textual information is represented as characters:
Characters

```r
x <- c("07745", "Jena", "Kahlaische Straße", "10")
x
```

```
[1] "07745" "Jena" "Kahlaische Straße"
[4] "10"
```

Again, we can find out the type of a variable with `typeof`:

```r
typeof(x)
```

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

We can also force this variable to `numeric`:

```r
as.numeric(x)
```

```
Warning: NAs introduced by coercion
[1] 7745 NA NA 10
```

**Pasting strings together**

And we can `collapse` the different elements of this character vector to a single character string:

```r
paste(x, collapse = " ")
```

```
[1] "07745 Jena Kahlaische Straße 10"
```

**Functions**

- `as.numeric`, `as.character`, `is.character`, ...
- `paste`

**3.2 Factors**

Factors are a way to store characters more efficiently. They associate a number with a meaning.

**Factors**

Let us start with a character:
x <- c("07745", "Jena", "Kahlaische Straße", "10")
x

When we convert x to a factor, it looks almost the same:

y <- as.factor(x)
y
[1] 07745 Jena Kahlaische Straße 10
Levels: 07745 10 Jena Kahlaische Straße

Something new appears. x has now levels, i.e. R has stored the different values our character variable could take.

levels(y)
[1] "07745" "10" "Jena"
[4] "Kahlaische Straße"

Factors

y
[1] 07745 Jena Kahlaische Straße 10
Levels: 07745 10 Jena Kahlaische Straße

To store the original character values, it is now sufficient to only remember the number of the level.

as.numeric(y)
[1] 1 3 4 2
levels(y)[as.numeric(y)]

Factors are useful, when a variable takes only a limited number of values (e.g. male, female or Jena, Berlin, Bonn)

If something is not a factor, but we still want to find unique values, we can use unique:
unique(c(1, 2, 3, 1, 2, 3, 1, 2))

[1] 1 2 3

Functions

- as.factor
- levels
- reorder
- relevel

3.3 Logicals

We use logicals when a variable can be only TRUE or FALSE.

Logicals

c(TRUE, FALSE, TRUE, TRUE)

[1] TRUE FALSE TRUE TRUE

The operator ! inverts TRUE and FALSE.

!c(TRUE, FALSE, TRUE, TRUE)

[1] FALSE TRUE FALSE FALSE

Logicals can be coerced as numeric. TRUE becomes 1, FALSE becomes 0.

as.numeric(c(TRUE, FALSE, TRUE, TRUE))

[1] 1 0 1 1

Logicals are automagically coerced to numeric when we calculate with them. Thus, sum counts the number of TRUE values, mean calculates the relative fraction of TRUE.

sum(c(TRUE, FALSE, TRUE, TRUE))

[1] 3
mean(c(TRUE, FALSE, TRUE, TRUE))

[1] 0.75

Functions

• !, &, |, &&, ||, xor, ...
• ifelse( c(TRUE,FALSE,TRUE), c(1,2,3), c(10,20,30)) = c(1,20,3)
• sum, mean,...

3.4 Dates

Sometimes it might be sufficient to store dates as characters. However, when we want to calculate with dates, we should store values as Date. R tries to guess the format from the character string, but it is always safe to specify the format:

Dates

as.Date("24/Jul/2012", format = "/d/%b/%Y")

[1] "2012-07-24"

x <- as.Date("Aug/24/2012", format = "/b/%d/%Y")
x

[1] "2012-08-24"

We can use format to print a string in a different format:

format(x, format = "/a, /d.%m.%Y")

[1] "Fri, 24.08.2012"

format(x + 1, format = "/A, /d.%m.%Y")

[1] "Saturday, 25.08.2012"

Dates

We can now calculate the difference between two dates:
Table 1: Date formats

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%a</td>
<td>abbreviated weekday (e.g., Sun)</td>
</tr>
<tr>
<td>%A</td>
<td>full weekday (e.g., Sunday)</td>
</tr>
<tr>
<td>%b</td>
<td>abbreviated month (e.g., Jan)</td>
</tr>
<tr>
<td>%B</td>
<td>full month name (e.g., January)</td>
</tr>
<tr>
<td>%C</td>
<td>century (e.g., 20)</td>
</tr>
<tr>
<td>%d</td>
<td>day of month (e.g., 01)</td>
</tr>
<tr>
<td>%e</td>
<td>day of month, space padded; same as %d</td>
</tr>
<tr>
<td>%g</td>
<td>last two digits of year of ISO week number (see %G)</td>
</tr>
<tr>
<td>%G</td>
<td>year of ISO week number (see %V); normally useful only with %V</td>
</tr>
<tr>
<td>%h</td>
<td>same as %b</td>
</tr>
<tr>
<td>%j</td>
<td>day of year (001..366)</td>
</tr>
<tr>
<td>%m</td>
<td>month (01..12)</td>
</tr>
<tr>
<td>%u</td>
<td>day of week (1..7); 1 is Monday</td>
</tr>
<tr>
<td>%U</td>
<td>week number of year, with Sunday as first day of week (00..53)</td>
</tr>
<tr>
<td>%V</td>
<td>ISO week number, with Monday as first day of week (01..53)</td>
</tr>
<tr>
<td>%w</td>
<td>day of week (0..6); 0 is Sunday</td>
</tr>
<tr>
<td>%W</td>
<td>week number of year, with Monday as first day of week (00..53)</td>
</tr>
<tr>
<td>%x</td>
<td>locale’s date representation (e.g., 12/31/99)</td>
</tr>
<tr>
<td>%y</td>
<td>last two digits of year (00..99)</td>
</tr>
<tr>
<td>%Y</td>
<td>year</td>
</tr>
</tbody>
</table>

R uses the date formats shown in table 1.

Date formats

Date and time
If we want to store date and time together, we can use two datatypes:

- POSIXct: (signed) number of seconds since the beginning of 1970 (in the UTC timezone) as a numeric vector.
  This is easy to integrate into a data frame.
• POSIXlt: is a named list of vectors representing `sec, min, hour, mday, mon, year, wday, yday, isdst`. This is difficult to integrate into a data frame.

**Date and time**

```r
x <- as.POSIXct("Dec/24/2012 20:00:00", format = "%b/%d/%Y %H:%M:%S")
x
[1] "2012-12-24 20:00:00 CET"

format(x, format = "%a, %d.%m.%Y")

format(x, format = "%A, %e %B '%y, %l:%M %p")
[1] "Monday, 24 December '12, 8:00 PM"
```

**Date and time**

As above, we can calculate with date-time objects:

```r
y <- as.POSIXct("Aug/21/2011 07:30:00", format = "%b/%d/%Y %H:%M:%S")
x - y
Time difference of 491.6 days
as.numeric(x - y)
[1] 491.6

Calculations are done on the level of seconds:

x + 1000
[1] "2012-12-24 20:16:40 CET"
```

**Time formats**

R uses the date formats shown in table[2]
%H  hour (00..23)
%H  hour (01..12)
%k  hour, space padded ( 0..23); same as %H
%M  minute (00..59)
%p  either AM or PM
%P  like %p, but lower case
%r  locale's 12-hour clock time (e.g., 11:11:04 PM)
%R  24-hour hour and minute; same as %H:%M
%S  second (00..60)
%T  time; same as %H:%M:%S
%Z  alphabetic time zone abbreviation (e.g., EDT)

Table 2: Time formats

Functions

- `as.Date`
- `format`
- `as.POSIXct`
- `as.POSIXlt`

3.5 Special values

Next to regular “numbers”, R also knows “special values”:

Special values

- `NA`  missing
- `NULL`  not defined
- `NaN`  not a number (numeric)
- `Inf`  infinity (numeric)

`NA` and `Inf`:

```r
c(1, 1, 0)/c(1, 0, 0)
```

```
[1] 1 Inf NaN
```
Special values

NA and NULL:

\[
\text{length(c(1, 2, 3, NA, 5))}
\]

\[
[1] 5
\]

\[
\text{length(c(1, 2, 3, NULL, 5))}
\]

\[
[1] 4
\]

Use NA to denote *missings*, NULL to *delete*.

4 Working with a lot of data

4.1 Indexing and subsetting

Indexing

\[
x <- \text{c("07745", "Jena", "Kahlaische Straße", "10")}
\]

\[
x
\]

\[
[1] \"07745\" \"Jena\" \"Kahlaische Straße\"
\]

[4] "10"

We can now access a single element of \(x\):

\[
x[3]
\]

\[
[1] \"Kahlaische Straße\"
\]

This also works with numbers:

\[
x <- \text{21:30}
\]

\[
x
\]

\[
\]

\[
x[5]
\]

\[
[1] 25
\]

Indexing

We can also address a range of elements:
Such a range of elements need not be contiguous:

```r
x[c(1, 5, 9)]
```

Indexing with logicals

We can index with either a vector of indices (which can be smaller or larger than the original vector), or with a vector of logicals of the same size of the original vector.

```r
x <- c(3, 7, -5)
x[c(TRUE, FALSE, TRUE)]
```

Indexing with logicals is interesting when the logical is actually a condition:

```r
x > 0
```

Matrices of Numbers

The function `cbind` and `rbind` combine several vectors (of the same length) to form a matrix:

Matrices of numbers

Here is a $3 \times 2$ matrix. Note how the recycling rule works in the construction of this matrix.
Here is a $2 \times 3$ matrix. Note again how the recycling rule works in the construction of this matrix.

```r
cbind(c(1, 2), c(40, 20, 30))
```

Warning: number of rows of result is not a multiple of vector length (arg 1)

```
[,1] [,2]
[1,] 1 40
[2,] 2 20
[3,] 1 30
```

What happens if we add $x$ and $y$ which have not the same dimension?

### Matrices of numbers

```r
x <- cbind(c(1, 2), c(40, 20, 30))
```

Warning: number of rows of result is not a multiple of vector length (arg 1)

```r
y <- rbind(c(1, 2), c(10, 20, 30))
```

Warning: number of columns of result is not a multiple of vector length (arg 1)

```r
x + y
```

Error: non-conformable arrays

We see that the recycling rule allows us to work with vectors of different length, but not with matrices of a different dimension.

### Elements of a matrix

Here is the entire matrix again:
Now we extract only the first row:

```r
x[1,]
```

```
[1] 1 40
```

Here we extract only the second column:

```r
x[,2]
```

```
[1] 40 20 30
```

Here are the first two rows:

```r
x[c(1,3),]
```

```
[,1] [,2]
[1,] 1 40
[2,] 1 30
```

One way to access elements (or rows or columns) is with indices. Alternatively we can use logicals. Only those elements where the index is `TRUE` are used. The following gives us the first and the third row:

```r
x[c(TRUE,FALSE,TRUE),]
```

```
[,1] [,2]
[1,] 1 40
[2,] 1 30
```

This is a first step to extract a subset of a matrix on a certain condition. Here is the condition:

```r
x[,2] > 20
```

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

A subset
Of course, the same can be done for data sets:

### Elements of a data sets

<table>
<thead>
<tr>
<th>Wages</th>
</tr>
</thead>
<tbody>
<tr>
<td>exp</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>

[ reached getOption("max.print") -- omitted 4162 rows ]

<table>
<thead>
<tr>
<th>Wages[1, ]</th>
</tr>
</thead>
<tbody>
<tr>
<td>exp</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

| Elements of a data sets |

<table>
<thead>
<tr>
<th>Wages[, 1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1] 3 4 5 6 7 8 9 30 31 32 33 34 35 36 6 7 8 9 10 11 12 31 32 33 34</td>
</tr>
<tr>
<td>[26] 35 36 37 10 11 12 13 14 15 16 26 27 28 29 30</td>
</tr>
</tbody>
</table>

[ reached getOption("max.print") -- omitted 4125 entries ]

If the rows or columns of a matrix or a data set have names, we can also use the names (instead of the index):

### Columns of a data set by name

<table>
<thead>
<tr>
<th>Wages</th>
</tr>
</thead>
<tbody>
<tr>
<td>exp</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>

[ reached getOption("max.print") -- omitted 4162 rows ]
The columns of a data set can also be addressed with $\$: 

```
Wages$exp
```

In a similar way we can now specify columns and rows at the same time:

**Elements of a data set**

```
Wages[1, "exp"]
```

```
[1] 3
```

```
Wages$exp[1]
```

```
[1] 3
```

```
Wages[c(1:3), "exp"]
```

```
[1] 3 4 5
```

```
Wages[c(1:3, 15, 30:32), c("exp", "black", "lwage")]
```

<table>
<thead>
<tr>
<th>exp</th>
<th>black</th>
<th>lwage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>no</td>
<td>5.561</td>
</tr>
<tr>
<td>2</td>
<td>no</td>
<td>5.720</td>
</tr>
<tr>
<td>3</td>
<td>no</td>
<td>5.996</td>
</tr>
<tr>
<td>15</td>
<td>no</td>
<td>5.652</td>
</tr>
<tr>
<td>30</td>
<td>no</td>
<td>6.620</td>
</tr>
<tr>
<td>31</td>
<td>no</td>
<td>6.633</td>
</tr>
<tr>
<td>32</td>
<td>no</td>
<td>6.983</td>
</tr>
</tbody>
</table>
Subsetting a data set

Let us have another look at Wages:

```r
Wages[1:3, ]
```

<table>
<thead>
<tr>
<th>exp</th>
<th>wks</th>
<th>bluecol</th>
<th>ind</th>
<th>south</th>
<th>smsa</th>
<th>married</th>
<th>sex</th>
<th>union</th>
<th>ed</th>
<th>black</th>
<th>lwage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>32</td>
<td>no</td>
<td>0</td>
<td>yes</td>
<td>no</td>
<td>male</td>
<td>no</td>
<td>9</td>
<td>no</td>
<td>5.561</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>43</td>
<td>no</td>
<td>0</td>
<td>yes</td>
<td>no</td>
<td>male</td>
<td>no</td>
<td>9</td>
<td>no</td>
<td>5.720</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>40</td>
<td>no</td>
<td>0</td>
<td>yes</td>
<td>no</td>
<td>male</td>
<td>no</td>
<td>9</td>
<td>no</td>
<td>5.996</td>
</tr>
</tbody>
</table>

Here is a condition that exp (the work experience) must be 1:

```r
Wages$exp == 1
```

```
[1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[13] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[25] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[37] FALSE FALSE FALSE FALSE
[ reached getOption("max.print") -- omitted 4125 entries ]
```

Here is the subset of Wages where this condition is TRUE:

```r
Wages[Wages$exp == 1, ]
```

<table>
<thead>
<tr>
<th>exp</th>
<th>wks</th>
<th>bluecol</th>
<th>ind</th>
<th>south</th>
<th>smsa</th>
<th>married</th>
<th>sex</th>
<th>union</th>
<th>ed</th>
<th>black</th>
<th>lwage</th>
</tr>
</thead>
<tbody>
<tr>
<td>624</td>
<td>1</td>
<td>48</td>
<td>no</td>
<td>0</td>
<td>no</td>
<td>yes</td>
<td>male</td>
<td>no</td>
<td>12</td>
<td>no</td>
<td>6.215</td>
</tr>
<tr>
<td>1548</td>
<td>1</td>
<td>45</td>
<td>yes</td>
<td>0</td>
<td>no</td>
<td>yes</td>
<td>male</td>
<td>no</td>
<td>11</td>
<td>no</td>
<td>5.704</td>
</tr>
<tr>
<td>1919</td>
<td>1</td>
<td>30</td>
<td>no</td>
<td>0</td>
<td>no</td>
<td>yes</td>
<td>male</td>
<td>no</td>
<td>17</td>
<td>no</td>
<td>6.215</td>
</tr>
</tbody>
</table>

[ reached getOption("max.print") -- omitted 5 rows ]

Let us be more specific and only look at exp==1 and married="yes":

Subsets

```r
Wages[Wages$exp == 1 & Wages$married == "yes", ]
```

<table>
<thead>
<tr>
<th>exp</th>
<th>wks</th>
<th>bluecol</th>
<th>ind</th>
<th>south</th>
<th>smsa</th>
<th>married</th>
<th>sex</th>
<th>union</th>
<th>ed</th>
<th>black</th>
<th>lwage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1919</td>
<td>1</td>
<td>30</td>
<td>no</td>
<td>0</td>
<td>no</td>
<td>yes</td>
<td>male</td>
<td>no</td>
<td>17</td>
<td>no</td>
<td>6.215</td>
</tr>
<tr>
<td>2878</td>
<td>1</td>
<td>50</td>
<td>no</td>
<td>0</td>
<td>no</td>
<td>yes</td>
<td>male</td>
<td>no</td>
<td>16</td>
<td>no</td>
<td>6.372</td>
</tr>
<tr>
<td>4110</td>
<td>1</td>
<td>51</td>
<td>no</td>
<td>0</td>
<td>yes</td>
<td>no</td>
<td>male</td>
<td>no</td>
<td>16</td>
<td>no</td>
<td>5.438</td>
</tr>
</tbody>
</table>

This can be written in a simpler way with subset:
We can subset rows and columns at the same time:

**Subsets of rows and columns**

\[
\text{Wages[Wages$exp == 1 & Wages$married == "yes", \}, c("sex", "black", "lwage")]
\]

<table>
<thead>
<tr>
<th>sex</th>
<th>black</th>
<th>lwage</th>
</tr>
</thead>
<tbody>
<tr>
<td>male</td>
<td>no</td>
<td>6.215</td>
</tr>
<tr>
<td>male</td>
<td>no</td>
<td>6.372</td>
</tr>
<tr>
<td>male</td>
<td>no</td>
<td>5.438</td>
</tr>
</tbody>
</table>

\[
\text{subset(Wages, exp == 1 & married == "yes")[, c("sex", "black", "lwage")]}\\
\]

4.3 Sorting

In R we usually sort with indices. `order` gives us a vector of indices:

**Sorting with indices**

\[
\text{order(Wages$lwage)}
\]

```
[1]  3279  1163  1164  2551  3280  526  1277  3281  4145  1278  1450  3277  512  1058  1732  
[16]  2521  3004  4146  4147  4148  1275  547  162  1723  3278  4110  1059  1060  1733  548  
[31]  2522  3599  3600  1276  163  513  533  890  1165  1451  
[ reached getOption("max.print") -- omitted 4125 entries ]
```

We can use this vector of indices to order a vector:

\[
\text{Wages$lwage[order(Wages$lwage)]}
\]
We can use the same vector of indices to order the entire data frame:

```r
Wages[order(Wages$lwage), ]
```

```
exp  wks  bluecol  ind  south  smsa  married  sex  union  ed  black  lwage
3279  46   39       yes  0      yes  yes    no  female  no  9  yes  4.605
1163  10   20       yes  0      no  no     no  female  no 10  no  5.011
1164 11  30       yes  0      no  no     no  female  no 10  no  5.011
```

In the same way we can order according to several variables:

```r
Wages[order(Wages$exp, Wages$lwage), ]
```

```
exp  wks  bluecol  ind  south  smsa  married  sex  union  ed  black  lwage
4110  1    51      no  0      yes  no      yes  male  no 16  no  5.438
4159  1    52      no  0      no  yes     no  female  no 12  no  5.687
1548  1    45      yes  0      no  yes     no  male  no 11  no  5.704
```

### Functions

- `cbind`, `rbind`, ...
- `dim`
- `subset`
- `order`
- `table`

### 4.4 Lists

Lists

Lists = a collection of (potentially) different types.
Elements of a list can be accessed with \([ [ \ldots ] \])

\[
x[[1]] \\
[1] 3 \\
x[[2]] \\
[1] "abc"
\]

**Inspecting the structure of lists**

Lists can be inspected with `str`:

\[
\text{str}(x)
\]

List of 2  
$ : num 3 
$ : chr "abc"

**Lists with named elements**

Elements of a list can (but need not) have names. We relate names to values with `=`

\[
x <- \text{list}(\text{value} = 123, \text{name} = "abc") \\
x
\]

$\text{value}$  
[1] 123  

$\text{name}$  
[1] "abc"

When we inspect \(x\) with the command `str` we see that the elements have names.
Accessing elements of lists

Elements of a list can be accessed by their position or by their name, with [[ .. ]] or $:

\[
\text{x\[\[1\]\]} \\
[1] 123 \\
\text{x\[["value"]\]} \\
[1] 123 \\
\text{x$\text{name}} \\
[1] 123
\]

More complex lists

Elements of a list can be any data type. In particular they can themselves be lists.

```r
x <- list(x = 7, y = c("abc", "def"), z = cbind(1:2, 3:4))
str(x)
```

List of 3

$ x: num 7

$ y: chr [1:2] "abc" "def"

$ z: int [1:2, 1:2] 1 2 3 4

unlist tries to make a vector out of a list.

```r
unlist(x)
```

<table>
<thead>
<tr>
<th>x</th>
<th>y1</th>
<th>y2</th>
<th>z1</th>
<th>z2</th>
<th>z3</th>
<th>z4</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;7&quot;</td>
<td>&quot;abc&quot;</td>
<td>&quot;def&quot;</td>
<td>&quot;1&quot;</td>
<td>&quot;2&quot;</td>
<td>&quot;3&quot;</td>
<td>&quot;4&quot;</td>
</tr>
</tbody>
</table>

Functions

• str
• [[
• $
• \text{unlist}

4.5 Summary datatypes

Summary data types

• Vector (1 dimension)
• Matrix (2 dimensions)
• Array (n dimensions)
• Data frame (2 dimensions, different types per column)
• List (not rectangular, different types per entry)

5 Reading and writing data

5.1 Using built in data sets
data makes a data set from a package accessible.

Using built in data sets

data(Wages, package = "Ecdat")

Now we can find out the names of the variables, the dimension, and statistics of some variables

names(Wages)

[1] "exp" "wks" "bluecol" "ind" "south" "smsa" "married"
[8] "sex" "union" "ed" "black" "lwage"

dim(Wages)

[1] 4165 12

mean(Wages$exp)

[1] 19.85

\text{str} gives a brief overview of the types and the first few values of each variable in the dataset.
Using built in data sets

```r
str(Wages)

'data.frame': 4165 obs. of 12 variables:
$ exp  : int 3 4 5 6 7 8 9 30 31 32 ...  
$ wks  : int 32 43 40 39 42 35 32 34 27 33 ...  
$ bluecol: Factor w/ 2 levels "no","yes": 1 1 1 1 1 1 2 2 2 2 ...  
$ ind  : int 0 0 0 0 1 1 1 0 0 1 ...  
$ south: Factor w/ 2 levels "no","yes": 2 2 2 2 2 2 2 1 1 1 ...  
$ smsa : Factor w/ 2 levels "no","yes": 1 1 1 1 1 1 1 1 1 1 ...  
$ married: Factor w/ 2 levels "no","yes": 2 2 2 2 2 2 2 2 2 2 ...  
$ sex  : Factor w/ 2 levels "female","male": 2 2 2 2 2 2 2 2 2 2 ...  
$ union: Factor w/ 2 levels "no","yes": 1 1 1 1 1 1 1 1 1 1 ...  
$ ed   : int 9 9 9 9 9 9 9 9 9 9 ...  
$ black: Factor w/ 2 levels "no","yes": 1 1 1 1 1 1 1 1 1 1 ...  
$ lwage: num 5.56 5.72 6 6 6.06 ...  

summary gives some basic statistics:

```r
summary(Wages)

```r
table(Wages)

5.2 Reading and writing csv files

Writing csv files

Let us first export a data frame as csv:

```r
write.csv(Wages, file = "wages.csv", row.names = FALSE)
```

The file wages.csv looks like this:
"exp","wks","bluecol","ind","south","smsa","married","sex","union","ed","black","lwage"
3,32,"no","0","yes","no","yes","male","no","9","no",5.56068
4,43,"no","0","yes","no","yes","male","no","9","no",5.72031
5,40,"no","0","yes","no","yes","male","no","9","no",5.99645
6,39,"no","0","yes","no","yes","male","no","9","no",5.99645
7,42,"no","1","yes","no","yes","male","no","9","no",6.06146
8,35,"no","1","yes","no","yes","male","no","9","no",6.17379
9,32,"no","1","yes","no","yes","male","no","9","no",6.24417

Reading csv files
Now let us read the file back into a variable:

```r
w <- read.csv("wages.csv")
w
```

```
 exp wks bluecol ind south smsa married sex union ed black lwage
1  3  32  no  0   yes no   yes  male no  9  no 5.561
2  4  43  no  0   yes no   yes  male no  9  no 5.720
3  5  40  no  0   yes no   yes  male no  9  no 5.996
[ reached getOption("max.print") -- omitted 4162 rows ]
```

5.3 Reading z-Tree Output

Reading z-Tree Output

```r
source("http://www.kirchkamp.de/lab/zTree.R")
```

```
Loading required package: plyr
```

The function

```r
zTreeTables(...vector of filenames...[,vector of tables])
```
reads zTree .xls files and returns a list of tables. Here we use list.files to find
all files that match the typical z-Tree pattern. If we ever get more experiments our
command will find them and use them.

```r
setwd("rawdata/Trust/")
(files <- list.files(pattern = "[0-9]{6}_[0-9]{4}.xls", recursive = TRUE))
[1] "090722_0601.xls" "090722_0602.xls" "090722_0603.xls" "090722_0604.xls"

trustGS <- zTreeTables(files)
```
The structure of the z-Tree object

```
str(trustGS)
```

List of 2

$ globals: 'data.frame': 24 obs. of 5 variables:
  ..$ Date : chr [1:24] "090722_0601" "090722_0601" "090722_0601" "090722_0601" ...
  ..$ Treatment : num [1:24] 1 1 1 1 1 1 1 1 1 1 ...
  ..$ Period : num [1:24] 1 2 3 4 5 6 1 2 3 4 ...
  ..$ NumPeriods : num [1:24] 6 6 6 6 6 6 6 6 6 6 ...
  ..$ RepeatTreatment: num [1:24] 0 0 0 0 0 0 0 0 0 0 ...

$ subjects: 'data.frame': 432 obs. of 14 variables:
  ..$ Date : chr [1:432] "090722_0601" "090722_0601" "090722_0601" "090722_0601" ...
  ..$ Treatment: num [1:432] 1 1 1 1 1 1 1 1 1 1 ...
  ..$ Period: num [1:432] 1 1 1 1 1 1 1 1 1 1 ...
  ..$ Subject : num [1:432] 1 2 3 4 5 6 7 8 9 10 ...
  ..$ Pos : num [1:432] 2 2 1 2 1 1 2 1 2 2 ...
  ..$ Group : num [1:432] 1 4 5 2 4 3 5 2 9 7 ...
  ..$ Offer: num [1:432] 0 0 0.495 0 0.558 ...
  ..$ Receive: num [1:432] 1.53 1.67 0 2.53 0 ...
  ..$ Return: num [1:432] 0.586 1.132 0 1.471 0 ...

40
Using the z-Tree object
As long as we need only a single table, we can access, e.g. the subjects table with $subjects.
If we need, e.g. the globals table together with the subjects table, we can merge them:

```r
x <- with(trustGS, merge(globals, subjects))
str(x)
```

5.4 Reading and writing R-Files

Reading and writing R-Files
If we want to save one or more R objects in a file, we use save

```r
save(trustGS, zTreeTables, file = "120722_060x.Rdata")
```
To retrieve them, we use load
Advantages:

- Rdata is very compact, files are small
- All attributes are saved together with the data
- We can save functions together with data

5.5 Reading and writing Stata files

Stata files

Let us first create a Stata file:

```r
library(foreign)
write.dta(Wages, file = "Wages.dta")
```

Now we do the following in Stata:

```stata
> pwd
> use Wages
> sum
> table ed
> save Wages2
```

Reading back from Stata

```r
read.dta(file = "Wages2.dta")
```

```r
Wa2 <- read.dta(file = "Wages2.dta")
summary(Wa2$ed)
```

```r
summary(Wages$ed)
```
The two data frames (Wa2 and Wages) seem to be the same (as they should).

### 5.6 Other formats

**foreign**

The `foreign` package contains functions to read and (often) write the following:

<table>
<thead>
<tr>
<th></th>
<th>read</th>
<th>write</th>
</tr>
</thead>
<tbody>
<tr>
<td>S3</td>
<td>r</td>
<td>-</td>
</tr>
<tr>
<td>SAS</td>
<td>r</td>
<td>-</td>
</tr>
<tr>
<td>ARFF</td>
<td>r</td>
<td>w</td>
</tr>
<tr>
<td>DBF</td>
<td>r</td>
<td>w</td>
</tr>
<tr>
<td>Stata Files</td>
<td>r</td>
<td>w</td>
</tr>
<tr>
<td>Epi</td>
<td>r</td>
<td>-</td>
</tr>
<tr>
<td>Minitab</td>
<td>r</td>
<td>-</td>
</tr>
<tr>
<td>Octave</td>
<td>r</td>
<td>-</td>
</tr>
<tr>
<td>SPSS</td>
<td>r</td>
<td>w</td>
</tr>
<tr>
<td>Systat</td>
<td>r</td>
<td>-</td>
</tr>
</tbody>
</table>

**gnumeric**

`read.gnumeric.sheet` from the `gnumeric` package reads the following:
Applix Data Interchange Format *.as
Gnumeric XML *.dif *gnumeric
GNU Oleo *.oleo *.html, *.htm
HTML
Linear and integer program expression *.mps
Lotus 123 *.wk1, *.wks, *.123
MS Excel *.xls, *.xlsx
MultiPlan *.slk
Open Document Format *.sxc, *.ods
Plan Perfect Format *.pln
Quattro Pro *.wb1, *.wb2, *.wb3
xspread *.sc
Xbase *.dbf

Reading from the clipboard
We can also read data from the clipboard (e.g. select some data in a spreadsheet and use this in R. However, this workflow will make it impossible to document what exactly was on the clipboard. It is always better to really import the spreadsheet

```
read.table("clipboard")
read.table("clipboard", header = TRUE)
```

Functions

- `data`, `names`, `rownames`, `colnames`
- `read.csv`, `write.csv`, `read.table`, `write.table`
- `read.dta`, `write.dta`, `zTreeTables`
- `save`, `save.image`, `load`
- `setwd`, `getwd`

5.7 Which data frame is actually used?
When use names of variables, R searches this name in different contexts.

Which data frame is actually used?

- `Wages$exp`
- `(....,data=Wages,...)`
- `attach(Wages),..., detach(Wages)`

44
• with(Wages,...)
• within(Wages,...)

**Which data frame is actually used**

• We can always explicitly prefix our variables with the name of the data frame. However, this can be a bit clumsy and is not necessary if we want to use always the same data frame.

```r
lm(Wages$lwage ~ Wages$exp + Wages$sex + Wages$ed)
```

• Many commands have a parameter data which can be used to specify a data frame.

```r
lm(lwage ~ exp + sex + ed, data=Wages)
```

• A data frame can also be brought into the current context with attach. So, after attach(Wages) we can simply say exp and this variable will first be searched in the dataset Wages. Note: Changing exp (or any other variable) does only change in the local context, not in the original data frame.

```r
attach(Wages)
lm(lwage ~ exp + sex + ed)
```

```r
detach(Wages)
```

• We can surround a command with with(≪data frame≫,≪command≫). Then ≪command≫ is executed within the context of the data frame.

```r
with(Wages, lm(lwage ~ exp + sex + ed))
```

• within(…) solves a different problem: It performs (similar to with(…)) all calculations within the context of a given data frame, and then returns this data frame with all changes that have been calculated and with all new variables that have been generated.

```r
Wages2 <- within(Wages, {lwage <- NULL;
female <- sex=="female";
  wage[female] <- wage[female]*2;
})
```

... creates a copy of Wages with two more added (wage and female), one removed (lwage), and the wage of female workers multiplied by 2.
Functions

- `as.data.frame`, `is.data.frame`
- `with`
- `within`
- `attach`, `detach`

6 Regressions

6.1 Regression objects

A simple regression

The function `lm` estimates OLS models. We use the *formula* notation to specify the estimated model.

```r
lm(lwage ~ exp + sex + ed, data = Wages)
```

Call:
`lm(formula = lwage ~ exp + sex + ed, data = Wages)`

Coefficients:
```
(Intercept) exp sexmale ed
5.0876 0.0118 0.4358 0.0753
```

We can write the result of a regression into a variable:

```r
est <- lm(lwage ~ exp + sex + ed, data = Wages)
```

This variable has not the class `lm` (linear model).

```r
class(est)
```

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

Regression objects - structure

Actually, the regression object is a long list. The elements of this list are the estimated coefficients, the estimated residuals, etc.

```r
str(est)
```
6.2 What to do with a regression object?

Regression objects
We can do different things with such a variable (a regression object):

- `str(est)`
- `coef(est)`
- `logLik(est)`
- `AIC(est)`
- `summary(est)`
- `plot(est)`
- ...

Regression objects - summary
summary produces an overview with some statistics of the regression object.
**summary(est)**

Call:
`lm(formula = lwage ~ exp + sex + ed, data = Wages)`

Residuals:

<table>
<thead>
<tr>
<th></th>
<th>Min</th>
<th>1Q</th>
<th>Median</th>
<th>3Q</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-2.1883</td>
<td>-0.2370</td>
<td>0.0042</td>
<td>0.2521</td>
<td>1.9558</td>
</tr>
</tbody>
</table>

Coefficients:

|         | Estimate | Std. Error | t value | Pr(>|t|) |
|--------|----------|------------|---------|---------|
| (Intercept) | 5.087573 | 0.035488 | 143.4 | <2e-16 *** |
| exp      | 0.011828 | 0.000548 | 21.6  | <2e-16 *** |
| sexmale  | 0.435815 | 0.018538 | 23.5  | <2e-16 *** |
| ed       | 0.075295 | 0.002145 | 35.1  | <2e-16 *** |

---

Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.376 on 4161 degrees of freedom
Multiple R-squared: 0.335, Adjusted R-squared: 0.335
F-statistic: 699 on 3 and 4161 DF, p-value: <2e-16

**Regression and heteroscedasticity**

The core of the above table can be shown with `coeftest`.

**coeftest(est)**

t test of coefficients:

|         | Estimate | Std. Error | t value | Pr(>|t|) |
|--------|----------|------------|---------|---------|
| (Intercept) | 5.087573 | 0.035488 | 143.4  | <2e-16 *** |
| exp      | 0.011828 | 0.000548 | 21.6   | <2e-16 *** |
| sexmale  | 0.435815 | 0.018538 | 23.5   | <2e-16 *** |
| ed       | 0.075295 | 0.002145 | 35.1   | <2e-16 *** |

---

Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

**Regression and heteroscedasticity**

As a default R assumes homoscedasticity. For the heteroscedastic case we specify `vcov=hccm`.  

49
coeftest(est, vcov = hccm)

t test of coefficients:

|                | Estimate | Std. Error | t value | Pr(>|t|) |
|----------------|----------|------------|---------|----------|
| (Intercept)    | 5.087573 | 0.037168   | 136.9   | <2e-16   *** |
| exp            | 0.011828 | 0.000618   | 19.1    | <2e-16   *** |
| sexmale        | 0.435815 | 0.017543   | 24.8    | <2e-16   *** |
| ed             | 0.075295 | 0.002277   | 33.1    | <2e-16   *** |

---

Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Regression objects - confidence intervals

Different from Stata, confidence intervals are not included in the summary. We calculate them separately.

confint(est)

<table>
<thead>
<tr>
<th></th>
<th>2.5 %</th>
<th>97.5 %</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Intercept)</td>
<td>5.01800</td>
<td>5.1571</td>
</tr>
<tr>
<td>exp</td>
<td>0.01075</td>
<td>0.0129</td>
</tr>
<tr>
<td>sexmale</td>
<td>0.39947</td>
<td>0.4722</td>
</tr>
<tr>
<td>ed</td>
<td>0.07109</td>
<td>0.0795</td>
</tr>
</tbody>
</table>

However, if we need the results side by side, we can do this:

cbind(coeftest(est), confint(est))

|                | Estimate | Std. Error | t value | Pr(>|t|) | 2.5 % | 97.5 % |
|----------------|----------|------------|---------|----------|-------|--------|
| (Intercept)    | 5.087573 | 0.0354883  | 143.36  | 0.000e+00| 5.01800| 5.1571 |
| exp            | 0.01183  | 0.0005476  | 21.60   | 3.691e-98| 0.01075| 0.0129 |
| sexmale        | 0.43582  | 0.0185376  | 23.51   | 7.340e-115| 0.39947| 0.4722 |
| ed             | 0.07530  | 0.0021448  | 35.11   | 1.022e-236| 0.07109| 0.0795 |

Regression objects - coefficients and other statistics

Here are some extractor-functions that extract statistics from the regression object.

ccoef(est)

<table>
<thead>
<tr>
<th></th>
<th>(Intercept)</th>
<th>exp</th>
<th>sexmale</th>
<th>ed</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5.08757</td>
<td>0.01183</td>
<td>0.43582</td>
<td>0.07530</td>
</tr>
</tbody>
</table>
\textbf{logLik(est)}

'\text{log Lik.}' -1839 (df=5)

\textbf{AIC(est)}

[1] 3688

\textbf{Regression objects - plots}

If we simply \texttt{plot} a regression object we get four diagnostic plots.

\texttt{plot(est)}

\textbf{Regression objects - lines}

We can also draw the regression as a line:

\texttt{est2 <- lm(lwage ~ exp, data = w)}
\texttt{plot(lwage ~ exp, data = w)}
\texttt{abline(est2, col = "red", lwd = 3)
6.3 Hypothesis tests

Regression objects - hypothesis tests

```r
library(car)

The function `linearHypothesis` from the package car allows us to write linear restrictions in a simple way.

```r
linearHypothesis(est, "6*exp=ed")
```

Linear hypothesis test

Hypothesis:
6 exp - ed = 0

Model 1: restricted model
Model 2: lwage ~ exp + sex + ed

<table>
<thead>
<tr>
<th>Res.Df</th>
<th>RSS</th>
<th>Df</th>
<th>Sum of Sq</th>
<th>F</th>
<th>Pr(&gt;F)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.27</td>
<td>17</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1.27</td>
<td>16</td>
<td>0.00238</td>
<td>0.03</td>
<td>0.86</td>
</tr>
</tbody>
</table>

Again, this can also be done for the heteroscedastic case.

```r
linearHypothesis(est, "6*exp=ed", vcov = hccm)
```

Linear hypothesis test

Hypothesis:
6 exp - ed = 0

Model 1: restricted model
Model 2: lwage ~ exp + sex + ed

Note: Coefficient covariance matrix supplied.

<table>
<thead>
<tr>
<th>Res.Df</th>
<th>Df</th>
<th>F</th>
<th>Pr(&gt;F)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>17</td>
<td>0.03</td>
<td>0.87</td>
</tr>
</tbody>
</table>

Regression objects - hypothesis tests
A Breusch-Pagan test against heteroskedasticity.

bptest(est)

studentized Breusch-Pagan test

data: est
BP = 1.539, df = 3, p-value = 0.6734

6.4 Regression with dummies

Dummies
Dummy variables will be automagically coerced into integers. R will also create a variable name the shows the meaning (here blackyes).

summary(lm(lwage ~ exp + black, data = Wages))

Call:
lm(formula = lwage ~ exp + black, data = Wages)

Residuals:
Min 1Q Median 3Q Max
-1.9825 -0.2872 0.0218 0.2737 1.9512

Coefficients:
  Estimate Std. Error t value Pr(> |t|)
(Intercept) 6.520275 0.014237 458.0 <2e-16 ***
exp 0.009147 0.000625 14.6 <2e-16 ***
blackyes -0.353412 0.026474 -13.3 <2e-16 ***
Interactions

Interactions can be specified as just the product (*) of several terms. By default R will add all possible interactions to the model.

```
summary(lm(lwage ~ exp * black, data = Wages))
```

```
Call:
lm(formula = lwage ~ exp * black, data = Wages)

Residuals:
     Min       1Q   Median       3Q      Max
-1.89170 -0.28600  0.02225  0.27450  1.95590

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   6.51398   0.01469  443.50  < 2e-16 ***
exp           0.00947   0.00065   14.53  < 2e-16 ***
blackyes     -0.26782   0.05591  -4.79 1.7e-06 ***
exp:blackyes -0.00402   0.00231  -1.74    0.08  .
```

Residual standard error: 0.442 on 4161 degrees of freedom
Multiple R-squared: 0.0831, Adjusted R-squared: 0.0826
F-statistic: 189 on 2 and 4162 DF, p-value: <2e-16

Specific interactions

Here we include only some interactions with the operator :

```
summary(lm(lwage ~ exp + black + sex + black:sex:exp, data = Wages))
```

```
Call:
lm(formula = lwage ~ exp + black + sex + black:sex:exp, data = Wages)
```

---
Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.442 on 4162 degrees of freedom
Multiple R-squared: 0.0831, Adjusted R-squared: 0.0826
F-statistic: 189 on 2 and 4162 DF, p-value: <2e-16

Interactions

Interactions can be specified as just the product (*) of several terms. By default R will add all possible interactions to the model.

```
summary(lm(lwage ~ exp * black, data = Wages))
```

```
Call:
lm(formula = lwage ~ exp * black, data = Wages)

Residuals:
     Min       1Q   Median       3Q      Max
-1.89170 -0.28600  0.02225  0.27450  1.95590

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   6.51398   0.01469  443.50  < 2e-16 ***
exp           0.00947   0.00065   14.53  < 2e-16 ***
blackyes     -0.26782   0.05591  -4.79 1.7e-06 ***
exp:blackyes -0.00402   0.00231  -1.74    0.08  .
```

Residual standard error: 0.442 on 4161 degrees of freedom
Multiple R-squared: 0.0831, Adjusted R-squared: 0.0826
F-statistic: 189 on 2 and 4162 DF, p-value: <2e-16

Specific interactions

Here we include only some interactions with the operator :

```
summary(lm(lwage ~ exp + black + sex + black:sex:exp, data = Wages))
```

```
Call:
lm(formula = lwage ~ exp + black + sex + black:sex:exp, data = Wages)
```
Factors in the formula

Factors with more than only two values will be included in the model as one dummy for each value of the factor.

```r
summary(lm(lwage ~ as.factor(ed) + black, data = Wages))
```
as.factor(ed)9  0.4463  0.1163  3.84 0.00013 ***
as.factor(ed)10 0.4780  0.1149  4.16 3.3e-05 ***
as.factor(ed)11 0.5347  0.1154  4.64 3.7e-06 ***
[ reached getOption("max.print") -- omitted 7 rows ]
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.418 on 4150 degrees of freedom
Multiple R-squared: 0.184, Adjusted R-squared: 0.181
F-statistic: 66.9 on 14 and 4150 DF, p-value: <2e-16

**Remove the regression constant with** \(-1\)

To estimate the model without a constant we add \(-1\) to the formula.

```
summary(lm(lwage ~ ed + black - 1, data = Wages))
```

Call:
`lm(formula = lwage ~ ed + black - 1, data = Wages)`

Residuals:
```
Min     1Q   Median     3Q    Max
-1.9367 -0.2716  0.0106  0.2747  1.8126
```

Coefficients:
```
            Estimate Std. Error t value Pr(>|t|)
ed          0.06234   0.00235   26.6 <2e-16 ***
blackno     5.89412   0.03110  189.5 <2e-16 ***
blackyes    5.63662   0.03648  154.5 <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Residual standard error: 0.419 on 4162 degrees of freedom
Multiple R-squared: 0.996, Adjusted R-squared: 0.996
F-statistic: 3.53e+05 on 3 and 4162 DF, p-value: <2e-16

**Using update**

The same could have been achieved with `update`:

```
est <- lm(lwage ~ ed + black, data = Wages)
update(est, . ~ . - 1)
```
Call:
```r
lm(formula = lwage ~ ed + black - 1, data = Wages)
```

Coefficients:
```
            ed  blackno  blackyes
 0.0623  5.8941  5.6366
```

We can as well add variables (here we add `exp`):
```
update(est, . ~ . + exp)
```

Call:
```r
lm(formula = lwage ~ ed + black + exp, data = Wages)
```

Coefficients:
```
            (Intercept)            ed  blackyes            exp
  5.4900    0.0735   -0.2656    0.0131
```

### 6.5 Nonlinear models

A log-log model

Transformations of variables can be done in the formula:
```
summary(lm(lwage ~ ed + log(ed) + black, data = Wages))
```

Call:
```r
lm(formula = lwage ~ ed + log(ed) + black, data = Wages)
```

Residuals:
```
     Min  1Q Median  3Q Max
-1.9602 -0.2738  0.0032  0.2815  1.8261
```

Coefficients:
```
            Estimate  Std. Error   t value Pr(>|t|)
(Intercept)  6.6454     0.2145    30.98  < 2e-16 ***
ed          0.1067     0.0127     8.37  < 2e-16 ***
log(ed)  -0.5228     0.1477    -3.54   0.00041 ***
blackyes  -0.2548     0.0252   -10.10  < 2e-16 ***
```

---

Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
Residual standard error: 0.419 on 4161 degrees of freedom
Multiple R-squared: 0.178, Adjusted R-squared: 0.178
F-statistic: 301 on 3 and 4161 DF, p-value: <2e-16

A logit model
Here is an example for logistic regression:
The Boston HDMA Data Set:

- dir: debt to income ratio
- hir: housing expenses to income ratio
- self: self employed
- deny: mortgage application denied

```r
data(Hdma, package = "Ecdat")
summary(Hdma[, c("dir", "hir", "self", "deny")])
```

<table>
<thead>
<tr>
<th></th>
<th>dir</th>
<th>hir</th>
<th>self</th>
<th>deny</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min.</td>
<td>0.000</td>
<td>0.000</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>1st Qu.</td>
<td>0.280</td>
<td>0.214</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Median</td>
<td>0.330</td>
<td>0.260</td>
<td>NA's</td>
<td>1</td>
</tr>
<tr>
<td>Mean</td>
<td>0.331</td>
<td>0.255</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3rd Qu.</td>
<td>0.370</td>
<td>0.299</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Max.</td>
<td>3.000</td>
<td>3.000</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A logit model
The estimation is done with glm. The parameter family specifies the “family” of the model (binomial, gaussian, Gamma, inverse.gaussian, poisson, quasi, quasibinomial, quasipoisson). Hence, glm covers a range of estimation problems.

```r
glm(deny ~ dir + hir + self, family = binomial, data = Hdma)
```

Call: glm(formula = deny ~ dir + hir + self, family = binomial, data = Hdma)

Coefficients:
(Intercept) dir hir selfyes
-4.017 6.082 -0.466 0.346

Degrees of Freedom: 2379 Total (i.e. Null); 2376 Residual
(1 observation deleted due to missingness)
Null Deviance: 1740
Residual Deviance: 1660 AIC: 1660
The logit model object

Similar to `lm`, also `glm` creates an object that we can write into a variable and that we can summarise.

```r
est <- glm(deny ~ dir + hir + self, family = binomial, data = Hdma)
summary(est)
```

Call:
`glm(formula = deny ~ dir + hir + self, family = binomial, data = Hdma)`

Deviance Residuals:

<table>
<thead>
<tr>
<th>Min</th>
<th>1Q</th>
<th>Median</th>
<th>3Q</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>-2.755</td>
<td>-0.525</td>
<td>-0.463</td>
<td>-0.388</td>
<td>2.761</td>
</tr>
</tbody>
</table>

Coefficients:

| Estimate | Std. Error | z value | Pr(>|z|) |
|----------|------------|---------|----------|
| (Intercept) -4.017 | 0.281 | -14.29 | < 2e-16 *** |
| dir 6.082 | 0.916 | 6.64 | 3.1e-11 *** |
| hir -0.466 | 1.039 | -0.45 | 0.654 |
| selfyes 0.346 | 0.188 | 1.84 | 0.066 . |

---

Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for binomial family taken to be 1)

Null deviance: 1744.2 on 2379 degrees of freedom
Residual deviance: 1656.6 on 2376 degrees of freedom
(1 observation deleted due to missingness)
AIC: 1665

Number of Fisher Scoring iterations: 5

The logit model object and heteroscedasticity

To see estimated standard errors for the heteroscedastic case we use `vcov=vcovHC` from the `sandwich` package.

```r
library(sandwich)
coeftest(est, vcov = vcovHC)
```

z test of coefficients:
## Linear hypotheses

Linear hypotheses can be specified as in `lm`.

```r
linearHypothesis(est, "dir=hir")
```

Linear hypothesis test

Hypothesis:
\[
\text{dir} - \text{hir} = 0
\]

Model 1: restricted model
Model 2: deny ~ dir + hir + self

<table>
<thead>
<tr>
<th>Res.Df</th>
<th>Df</th>
<th>Chisq</th>
<th>Pr(&gt;Chisq)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2377</td>
<td>1</td>
<td>14</td>
<td>0.00018 ***</td>
</tr>
</tbody>
</table>

---

Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

### Linear hypotheses and heteroscedasticity

```r
linearHypothesis(est, "dir=hir", vcov = vcovHC)
```

Linear hypothesis test

Hypothesis:
\[
\text{dir} - \text{hir} = 0
\]

Model 1: restricted model
Model 2: deny ~ dir + hir + self

Note: Coefficient covariance matrix supplied.

<table>
<thead>
<tr>
<th>Res.Df</th>
<th>Df</th>
<th>Chisq</th>
<th>Pr(&gt;Chisq)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
Maximum likelihood manually

Although R comes with a wide range of ML estimators, we can, for special estimators, do our own likelihood maximisation. Here is the logistic model as an example:

\[
\ln L = \sum_i \left( y_i \ln F(x_i \beta) + (1 - y_i) \ln (1 - F(x_i \beta)) \right)
\]

The logistic regression above can also be done “manually” as follows.

```r
logli <- function(beta) {
  n <- length(x)
  -(sum(y * (plogis(theta, log.p = TRUE))) +
    sum((1 - y) * plogis(theta, lower.tail = FALSE, log.p = TRUE)))
}

x <- 1:8
y <- c(0, 1, 0, 0, 1, 1, 0, 1)
optim(c(0, 0), logli)

$par
[1] -1.3759 0.3057

$value
[1] 5.102

$counts
function gradient
   63   NA

$convergence
[1] 0

$message
NULL

glm obtains the same result:
**Count data**

Here is the poisson model with doctors visits:

```r
data(DoctorAUS, package = "Ecdat")
summary(glm(doctorco ~ sex + age + I(age^2) + income, family = poisson, data = DoctorAUS))
```

Call:
```
glm(formula = doctorco ~ sex + age + I(age^2) + income, family = poisson, data = DoctorAUS)
```

Deviance Residuals:
```
Min 1Q Median 3Q Max
-1.020 -0.815 -0.672 -0.597 6.313
```

Coefficients:
```
             Estimate Std. Error z value Pr(>|z|)
(Intercept) -1.9777     0.1841  -10.74  < 2e-16 ***
   sex      0.2146     0.0559    3.84  0.00012 ***
   age      2.8565     0.9893    2.89  0.00388 **
I(age^2)   -1.7900     1.0847   -1.65  0.09889 .
   income  -0.3207     0.0837   -3.83  0.00013 ***
```

Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for poisson family taken to be 1)

Null deviance: 5634.8  on 5189 degrees of freedom
Residual deviance: 5432.2  on 5185 degrees of freedom
AIC: 7774
Count data with overdispersion

Now we do the same exercise with negative binomial

```r
library(MASS)
summary(glm.nb(doctorco ~ sex + age + I(age^2) + income, data = DoctorAUS))
```

Call:
`glm.nb(formula = doctorco ~ sex + age + I(age^2) + income, data = DoctorAUS, init.theta = 0.4281145934, link = log)`

Deviance Residuals:
```
Min 1Q Median 3Q Max
-0.827 -0.706 -0.601 -0.543 3.573
```

Coefficients:
```
            Estimate Std. Error z value Pr(>|z|)
(Intercept) -2.0252 0.2399  -8.44  < 2e-16 ***
  sex        0.2446 0.0726   3.37  0.00076 ***
  age        2.9328 1.3251   2.21  0.02687 *
I(age^2)    -1.8549 1.4656  -1.27  0.20566
income     -0.2985 0.1076  -2.77  0.00554 **
```

---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for Negative Binomial(0.4281) family taken to be 1)

Null deviance: 3110.1 on 5189 degrees of freedom
Residual deviance: 2989.9 on 5185 degrees of freedom
AIC: 7067

Number of Fisher Scoring iterations: 1

Theta: 0.4281
Std. Err.: 0.0313

2 x log-likelihood: -7054.9350

Ordered probit
This is an ordered probit model:

```r
data(Mathlevel, package = "Ecdat")
summary(polr(mathlevel ~ sat + sex + mathcourse + major, data = Mathlevel, method = "probit"))

Re-fitting to get Hessian

Call:
polr(formula = mathlevel ~ sat + sex + mathcourse + major, data = Mathlevel,
     method = "probit")

Coefficients:
             Value Std. Error t value
sat    0.00526 0.000811  6.48
sexfemale  0.21143 0.093070  2.27
mathcourse  0.56334 0.077317  7.29
majoreco  -0.11311 0.121649  -0.93
majoross  -0.28797 0.144411  -1.99
majors  0.44182 0.135447   3.26
majorhum  -0.45335 0.198786  -2.28

Intercepts:
             Value Std. Error t value
170|171a  3.420 0.508   6.738
171a|172a  3.687 0.509   7.240
172a|171b  3.743 0.510   7.345
171b|172b  4.910 0.521   9.431
172b|221a  5.184 0.523   9.908
221a|221b  5.423 0.525  10.322

Residual Deviance: 1792.06
AIC: 1818.06

Multinomial

This is an example for a multinomial model:

```r
data(Mode, package = "Ecdat")
summary(Mode)
```

choice cost.car cost.carpool cost.bus cost.rail
car :218 Min. :0.41 Min. :0.129 Min. :1.01 Min. :1.27
carpool: 32 1st Qu.:3.70 1st Qu.:0.952 1st Qu.:1.78 1st Qu.:1.95
bus : 81 Median :4.88 Median :1.666 Median :2.03 Median :2.20
rail :122 Mean :4.87 Mean :1.686 Mean :2.04 Mean :2.21

64
Multinomial

```
multinom(formula = choice ~ cost.car + cost.carpool + cost.bus + cost.rail + time.car + time.carpool +
    time.bus + time.rail, data = Mode)
```

Coefficients:

<table>
<thead>
<tr>
<th>(Intercept)</th>
<th>cost.car</th>
<th>cost.carpool</th>
<th>cost.bus</th>
<th>cost.rail</th>
<th>time.car</th>
<th>time.carpool</th>
<th>time.bus</th>
<th>time.rail</th>
</tr>
</thead>
<tbody>
<tr>
<td>carpool</td>
<td>-4.106</td>
<td>0.6361</td>
<td>-0.4473</td>
<td>0.04506</td>
<td>-0.5501</td>
<td>0.12367</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bus</td>
<td>-4.789</td>
<td>0.8461</td>
<td>0.2163</td>
<td>0.01013</td>
<td>-0.5277</td>
<td>0.02285</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rail</td>
<td>-4.300</td>
<td>0.8901</td>
<td>0.2058</td>
<td>0.56601</td>
<td>-1.2753</td>
<td>0.03639</td>
<td></td>
<td></td>
</tr>
<tr>
<td>time.carpool</td>
<td>-0.06923</td>
<td>0.007442</td>
<td>-0.02773</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>time.bus</td>
<td>0.09462</td>
<td>-0.107751</td>
<td>-0.006393</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>time.rail</td>
<td>0.07297</td>
<td>-0.018132</td>
<td>-0.075283</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Residual Deviance: 684.1
AIC: 738.1

Interval regression

In this example for an interval regression the dependent variable tobacco is censored. Many observations are zero. We can imagine that the underlying latent variable could be negative in these cases.
data(Tobacco, package = "Ecdat")

summary(Tobacco[, c("stobacco", "nkids", "nkids2", "lnx")])

<table>
<thead>
<tr>
<th></th>
<th>Min.</th>
<th>1st Qu.</th>
<th>Median</th>
<th>Mean</th>
<th>3rd Qu.</th>
<th>Max.</th>
</tr>
</thead>
<tbody>
<tr>
<td>stobacco</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0122</td>
<td>0.0138</td>
<td>0.1928</td>
</tr>
<tr>
<td>nkids</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.565</td>
<td>1.0000</td>
<td>5.0000</td>
</tr>
<tr>
<td>nkids2</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0448</td>
<td>0.0000</td>
<td>2.0000</td>
</tr>
<tr>
<td>lnx</td>
<td>11.8</td>
<td>13.4</td>
<td>13.8</td>
<td>13.7</td>
<td>14.1</td>
<td>15.3</td>
</tr>
</tbody>
</table>

- **stobacco**: budget share of tobacco
- **nkids**: number of kids of more than two years old
- **nkids2**: number of kids of less than two years old
- **lnx**: log of total expenditures

**Interval regression**

For each observation we define an interval. The right boundary of the interval is **stobacco**. The left boundary is the same value for the uncensored observations. For the censored observations the left boundary is − which is coded as `NA`.

library(survival)
Tobacco <- within(Tobacco, stobaccoMin <- ifelse(stobacco == 0, NA, stobacco))

summary(survreg(Surv(stobaccoMin, stobacco, type = "interval2") ~ nkids + nkids2 + lnx, dist = "gaussian", data = Tobacco))

Call:
survreg(formula = Surv(stobaccoMin, stobacco, type = "interval2") ~ nkids + nkids2 + lnx, data = Tobacco, dist = "gaussian")

Value Std. Error z     p
(Intercept) 0.24887 0.03291 7.56 3.95e-14
nkids 0.00642 0.00120 5.35 8.79e-08
nkids2 -0.00671 0.00542 -1.24 2.15e-01
lnx -0.01958 0.00242 -8.09 5.75e-16
Log(scale) -3.01205 0.02474 -121.76 0.00e+00

Scale= 0.0492

Gaussian distribution
Loglik(model)= 711.8  Loglik(intercept only)= 673.6
7 Functions

7.1 Writing functions

Writing functions
So far assign numbers to variables:

```r
x <- 2
class(x)
```

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

Functions are just a datatype. As such they are just assigned to a variable:

```r
x <- function(z) {
  z^2
}
class(x)
```

```
[1] "function"
```

The value of functions

```r
x
```

```r
function(z) {
  z^2
}
```

Of course, we can simply use the function:

```r
x(10)
```

```
[1] 100
```

7.2 More on parameters

Positional and named parameters
\begin{verbatim}
z <- function(a, b = 3) {
    a/b
}

We can call z with named parameters:

z(a = 10, b = 2)
[1] 5

With named parameters the order does not matter:

z(b = 2, a = 10)
[1] 5

If names are omitted, the order matters:

z(10, 2)
[1] 5

Parameters with a default (b=3) can be omitted.

z(10)
[1] 3.333

The \ldots argument to functions

\begin{verbatim}
z <- function(data, ...) {
    est <- lm(lwage ~ exp + sex + ed, data = data)
    linearHypothesis(est, "6*exp=ed", ...)[["Pr(>F)"]][2]
}
\end{verbatim}

z(Wages)
[1] 0.2176

z(Wages, vcov = hccm)
[1] 0.2459

The \ldots argument can be used to pass on a variable number of arguments to another function. This is also very useful for \texttt{plot}.
\end{verbatim}
7.3 Return values

Return values of functions

Functions always return the result of the last statement:

```r
z <- function(x) {
  q <- x + 1
  r <- x - 1
  r^2
}
z(10)
[1] 81
```

Return values of functions

If we can to return more than one value, then we can use lists:

```r
z <- function(x) {
  list(x = x, y = x^2, z = x^3)
}
result <- z(10)
str(result)
```

```
List of 3
$ x: num 10
$ y: num 100
$ z: num 1000
```

```r
result$z
[1] 1000
```

Most higher level statistical functions return lists. This makes it easy to access their results.

Scope of variables

Assignments to variables only change these variables within a function.

```r
q <- 7
z <- function() {
  q <- 3
  q
}
z()
```

69
If we really want to change variables outside the scope of a function, we have to use `<<-`.

**Scope of variables**

```r
q <- 7
z <- function() {
  q <<- 3
  q
}
z()
```

```
[1] 3
```

`z()` is now a function with a *side effect*, it changes the value of `q`. Most of the time this is confusing and should be avoided.

### 7.4 Debugging functions

**Debugging**

The following will not work:

```r
bad <- function(x) {
  lm(x ~ y, data = doesnotexist)
}
bad()
```

```
Error: object 'doesnotexist' not found
```

**Debugging**

```r
options(error = recover)
bad()
```

```
```
Error in inherits(x, "data.frame") : object "doesnotexist" not found

Enter a frame number, or 0 to exit

1: bad()
2: lm(x ~ y, data = doesnotexist)
3: eval(mf, parent.frame())
4: eval(expr, envir, enclos)
5: model.frame(formula = x ~ y, data = doesnotexist, drop.unused.levels = TRUE)
6: model.frame.default(formula = x ~ y, data = doesnotexist, drop.unused.levels = TRUE)
7: is.data.frame(data)
8: inherits(x, "data.frame")

Selection:

We can now choose a frame to inspect (here we choose frame 2):

Debugging

Selection: 2
Called from: eval(expr, envir, enclos)
Browse[1]> ls()
[1] "cl" "contrasts" "data" "formula" "m" "method"
[9] "na.action" "offset" "qr" "ret.x" "ret.y" "singular.ok"
[17] "x" "y"
Browse[1]> Q

Functions

• options(error=recover), options(error=stop)
• debug, debugonce, undebug, trace

8 Control and repetition

8.1 if and else

if and else

x <- 7
if (x > 5) print("x is larger than five") else print("x is small")

[1] "x is larger than five"
ifelse does something different:

```r
x <- 31:40
x

[1] 31 32 33 34 35 36 37 38 39 40

ifelse(x > 35, x + 10, x - 10)

```

8.2 for, while, repeat

for, while, repeat

```r
for (i in 1:3) print(i^2)

[1] 1
[1] 4
[1] 9

i <- 1
while (i < 4) {
  print(i^2)
  i <- i + 1
}

[1] 1
[1] 4
[1] 9
```

for, while, repeat

```r
i <- 1
repeat {
  print(i^2)
  i <- i + 1
  if (i > 3)
    break
    next
}

[1] 1
[1] 4
[1] 9
```
We should avoid for, while, and repeat. They are slow and clumsy.

### 8.3 sapply and apply

**sapply**

sapply applies a function to a vector (and returns a vector).

```r
sapply(1:3, function(x) x^2)
[1] 1 4 9
```

```r
sapply(1:3, log)
[1] 0.0000 0.6931 1.0986
```

```r
sapply(1:3, log, base = 3)
[1] 0.0000 0.6309 1.0000
```

```r
sapply(1:3, function(x) c(x = x, y = x^2, r = 1/x))
  [,1] [,2] [,3]
  x   1  2.0 3.0000
  y   1  4.0 9.0000
  r   1  0.5 0.3333
```

**apply**

apply applies a function (here `sum`) along a dimension of an array (here 1 and 2) and returns an array of lower dimensionality.

```r
(x <- rbind(c(1, 2, 3), c(4, 5, 6)))
  [,1] [,2] [,3]
[1,]  1  2  3
[2,]  4  5  6
```

```r
apply(x, 1, sum)
[1] 6 15
```

```r
apply(x, 2, sum)
[1] 5 7 9
```
8.4 Wide and long arrays

Wide and long regular arrays:

*wide* form of a regular array:

\[
\begin{pmatrix}
  a & b & c \\
  d & e & f
\end{pmatrix}
\]

*long* form of the same regular array:

<table>
<thead>
<tr>
<th>row</th>
<th>column</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>a</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>b</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>c</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>d</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>e</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>f</td>
</tr>
</tbody>
</table>

Wide and long ragged arrays:

*wide* form of a ragged array:

\[
\begin{pmatrix}
  a & b \\
  d & f
\end{pmatrix}
\]

*long* form of the same ragged array:

<table>
<thead>
<tr>
<th>row</th>
<th>column</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>a</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>b</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>d</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>f</td>
</tr>
</tbody>
</table>

We apply functions along the dimensions of a regular or ragged array in *long* form with `aggregate` and `by`.

8.5 aggregate and by
aggregate

\texttt{aggregate}(Wages, \texttt{list(ed = Wages$ed)}, \texttt{mean})

\begin{tabular}{cccccccccccc}
\text{ed} & \text{exp} & \text{wks} & \text{bluecol} & \text{ind} & \text{south} & \text{smsa} & \text{married} & \text{sex} & \text{union} & \text{ed} & \text{black} & \text{l wage} \\
4 & 22.00 & 46.00 & 0.64 & & & & & & & 4.00 & 6.04 \\
5 & 37.67 & 45.71 & 0.33 & & & & & & & 5.00 & 6.44 \\
6 & 31.00 & 45.86 & 0.93 & & & & & & & 6.00 & 6.42 \\
7 & 23.91 & 47.10 & 0.73 & & & & & & & 7.00 & 6.27 \\
8 & 27.84 & 45.35 & 0.64 & & & & & & & 8.00 & 6.41 \\
9 & 22.26 & 46.67 & 0.50 & & & & & & & 9.00 & 6.47 \\
10 & 24.30 & 46.50 & 0.53 & & & & & & & 10.00 & 6.50 \\
11 & 22.83 & 46.43 & 0.48 & & & & & & & 11.00 & 6.52 \\
12 & 19.49 & 47.11 & 0.44 & & & & & & & 12.00 & 6.60 \\
13 & 15.38 & 48.05 & 0.30 & & & & & & & 13.00 & 6.59 \\
14 & 17.29 & 47.62 & 0.27 & & & & & & & 14.00 & 6.73 \\
15 & 20.83 & 48.10 & 0.57 & & & & & & & 15.00 & 6.88 \\
16 & 17.03 & 47.10 & 0.27 & & & & & & & 16.00 & 6.89 \\
17 & 18.85 & 45.29 & 0.23 & & & & & & & 17.00 & 6.98 \\
\end{tabular}

by

\texttt{by}(Wages, \texttt{list(ed = Wages$ed)}, \texttt{function(x) coef(lm(lwage ~ exp, data = x))})

ed: 4

(Intercept) \hspace{1cm} \text{exp}

5.49710 \hspace{1cm} 0.02459

-------------------------------------------------------------
ed: 5
(Intercept)  exp
  7.78186  -0.03573
------------------------------------------------------------
ed: 6
(Intercept)  exp
  6.299444  0.004007
------------------------------------------------------------
ed: 7
(Intercept)  exp
  5.85985  0.01705
------------------------------------------------------------
ed: 8
(Intercept)  exp
  5.64363  0.02745
------------------------------------------------------------
ed: 9
(Intercept)  exp
  6.380162  0.004161
------------------------------------------------------------
ed: 10
(Intercept)  exp
  6.342956  0.006462
------------------------------------------------------------
ed: 11
(Intercept)  exp
  5.99853  0.02282
------------------------------------------------------------
ed: 12
(Intercept)  exp
  6.40168  0.01036
------------------------------------------------------------
ed: 13
(Intercept)  exp
  6.37998  0.01383
------------------------------------------------------------
ed: 14
(Intercept)  exp
  6.48491  0.01445
------------------------------------------------------------
ed: 15
(Intercept)  exp
  6.8984666  -0.0007323
x <- by(Wages, list(ed = Wages$ed), function(x) coef(lm(lwage ~ exp, data = x)))

sapply(x, c)

        4        5        6        7        8        9       10       11
(Intercept) 5.49710  7.78186  6.299444  5.85985  5.64363  6.380162  6.342956  5.99853
exp     0.02459 -0.03573  0.004007  0.01705  0.02745  0.004161  0.006462  0.02282

        12       13       14       15       16       17
exp     0.01036  0.01383  0.01445 -0.0007323  0.0193  0.01257

t(sapply(x, c))

(Intercept)   exp
   4   5.497 0.0245893
   5   7.782 -0.0357275
   6   6.299 0.0040072
   7   5.860 0.0170517
   8   5.644 0.0274480
   9   6.380 0.0041614
  10   6.343 0.0064618
  11   5.999 0.0228244
  12   6.402 0.0103584
  13   6.380 0.0138323
  14   6.485 0.0144524
  15   6.898 -0.0007323
  16   6.564 0.0193044
  17   6.747 0.0125693
9 Organising data

9.1 Merge + Append

merge as append

```r
W1 <- subset(Wages, exp == 1)
W2 <- subset(Wages, exp == 2)
merge(W1, W2, all = TRUE)
```

```
exp  wks  bluecol  ind  south  smsa  married  sex  union  ed  black  lwage
1   1    30      no  no     no    yes     yes  male  no  17 no 6.215
2   1    35     yes  1   yes   no     no    male  no  14 no 5.704
3   1    45     yes  0   yes   no     no    male  no  11 no 5.704
[ reached getOption("max.print") -- omitted 23 rows ]
```

```
merge(X <- as.data.frame(list(ed = c(9, 11, 12), type = c("A", "B", "C"))))
```

```
ed  type
1   9   A
2  11   B
3  12   C
```

```
merge(Wages, X)[, c("ed", "exp", "wks", "type")]
```

```
ed  exp  wks  type
1   9   19  45   A
2   9   22  50   A
3   9   31  49   A
4   9   45  51   A
5   9   13  49   A
6   9   33  48   A
7   9   16  52   A
8   9   28  47   A
9   9   35  52   A
10  9    6  50   A
[ reached getOption("max.print") -- omitted 1859 rows ]
```
merge(Wages, X, all = TRUE)[, c("ed", "exp", "wks", "type")]

<table>
<thead>
<tr>
<th>ed</th>
<th>exp</th>
<th>wks</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>28</td>
<td>&lt;NA&gt;</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>29</td>
<td>&lt;NA&gt;</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>30</td>
<td>&lt;NA&gt;</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>31</td>
<td>&lt;NA&gt;</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>32</td>
<td>&lt;NA&gt;</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>33</td>
<td>&lt;NA&gt;</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>34</td>
<td>&lt;NA&gt;</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>10</td>
<td>&lt;NA&gt;</td>
</tr>
<tr>
<td>9</td>
<td>4</td>
<td>12</td>
<td>&lt;NA&gt;</td>
</tr>
<tr>
<td>10</td>
<td>4</td>
<td>13</td>
<td>&lt;NA&gt;</td>
</tr>
</tbody>
</table>

[ reached getOption("max.print") -- omitted 4155 rows ]

9.2 Strings and Renaming

Let us, for the sake of the example, assume that we want to combine several variables (of different types) into one character variable (sometimes we do this to create a factor, think of combining the Date and Subject in a z-Tree file to create a unique subject-id).

sprintf: From numbers to strings

```
exp wks bluecol ind south smsa married sex union ed black lwage
3  32 no 0 yes no yes male no 9 no 5.56
4  43 no 0 yes no yes male no 9 no 5.72
5  40 no 0 yes no yes male no 9 no 6.00
6  39 no 0 yes no yes male no 9 no 6.00
7  42 no 1 yes no yes male no 9 no 6.06
8  35 no 1 yes no yes male no 9 no 6.17
9  32 no 1 yes no yes male no 9 no 6.24
30 34 yes 0 no no yes male no 11 no 6.16
31 27 yes 0 no no yes male no 11 no 6.21
32 33 yes 1 no no yes male yes 11 no 6.26
33 30 yes 1 no no yes male no 11 no 6.54
34 30 yes 1 no yes no yes male no 11 no 6.70
35 37 yes 1 no yes no yes male no 11 no 6.79
36 30 yes 1 no yes no yes male no 11 no 6.82
```

sprintf: From numbers to strings

```
with(Wages, sprintf("%s-%d-%.2f", sex, ed, lwage))
```

```
[1] "male-9-5.56"  "male-9-5.72"  "male-9-6.00"  "male-9-6.00"
```
sprintf: From numbers to strings

\%
s string
\%10s string of width 10
\%d integer
\%8d integer of width 8
\%f floating point number
\%10.2f floating point number of width 10 with 2 digits precision

Manipulating strings

• Strings as data which we want to manipulate.
• Names of variables which we want to change in a systematic way.

W <- Wages
names(W)

[1] "exp" "wks" "bluecol" "ind" "south" "smsa" "married"
[8] "sex" "union" "ed" "black" "lwage"

names(W)[3]

[1] "bluecol"

names(W)[3] <- "Worker"

names(W)

[1] "exp" "wks" "Worker" "ind" "south" "smsa" "married"
[8] "sex" "union" "ed" "black" "lwage"
Manipulating strings

- Can we change the name in a safer way?

```r
names(W)[names(W) == "Worker"] <- "bluecollar"
names(W)

[1] "exp"  "wks"  "bluecollar"  "ind"  "south"
[6] "smsa"  "married"  "sex"  "union"  "ed"
[11] "black"  "lwage"
```

Changing names/strings in a systematic way

Assume that we have the following names (of variables):

```r
```

How do we drop all the variables that start with *Time*?

`grep` finds strings that match a pattern:

```r
grep("Time", n)
```

```
[1] 5 6 7
```

If we want to see what we have found:

```r
grep("Time", n, value = TRUE)
```

```
```

We can also invert the selection (we want to keep the variables who do not match):

```r
grep("Time", n, invert = TRUE)
```

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

```r
n[grep("Time", n, invert = TRUE)]
```

```
[1] "Date"  "Subject"  "Period"  "Invest"
```
Even safer would be to say the following:

```r
grep("Time", n, invert = TRUE)
```

This would make sure that only strings that start with `Time` match (and not `SomeTime`).

**Changing names/strings in a systematic way**

```r
n <- c("Date", "Subject", "Period", "A_invest", "B_invest", "C_invest", "other")
```

How can we translate `A_invest` into `Inv_A`?

```r
sub("(.*)_invest", "Inv_\1", n)
```

The pattern `(.*)` follows the syntax of “regular expressions” (see `help(regexp)`).

**Regular expressions**

- `a-z0-9` matches itself
- `.` matches anything
- `*` matches whatever was before zero or more times (hence, `.*` matches anything)
- `+` matches whatever was before one or more times
- `?` matches whatever was before at most once.
- `[3-5]` matches 3, or 4, or 5
- `[\^3-5]` matches everything, except 3, or 4, or 5
- `^` matches the start of the string (e.g. `^abc` matches `abc` at the start of the string)
- `$` matches the end of the string
- `(\(...))` the match of each group of parentheses can be used as \1, \2, \3,...(the letter \ has to be escaped in R, like \\).

For more details, see `help(regexp)`.
Strings and Formulas in Regressions

Sometimes we want to use many variables in a regression. We can extract them with `grep`, paste them together with `paste`, and make them a formula with `as.formula`. Let us assume that we want, for the sake of the example, use all variables that start with the letter `b`:

```r
(varnames <- grep("^b", names(Wages), value = TRUE))
[1] "bluecol" "black"

(myForm <- as.formula(paste("lwage ~", paste(varnames, collapse = " + "))))
lwage ~ bluecol + black

lm(myForm, data = Wages)
```

Coefficients:

<table>
<thead>
<tr>
<th></th>
<th>bluecolyes</th>
<th>blackyes</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Intercept)</td>
<td>6.841</td>
<td>-0.292</td>
</tr>
</tbody>
</table>

9.3 Reshape

Reshaping data

How can we reshape a data from from wide to long form and vice versa?

```r
x <- as.data.frame(cbind(id = c(1:3), invest.1 = c(100:102), invest.2 = c(200:202)))
x

id  invest.1  invest.2
1 1 100 200
2 2 101 201
3 3 102 202

reshape(x, idvar = "id", varying = list(2:3), direction = "long")
```

id  time  invest.1
1.1 1 1 100
2.1 2 1 101
3.1 3 1 102
and if we do not want to list the wide variables as numbers we can use `grep`

```r
grep("\^invest", names(x))
```

```r
[1]  2  3
```

```r
(longx <- reshape(x, idvar = "id", varying = grep("\^invest", names(x)), direction = "long"))
```

```r
 id time invest
1.1 1 1 100
2.1 2 1 101
3.1 3 1 102
1.2 1 2 200
2.2 2 2 201
3.2 3 2 202
```

**Reshape back to wide**

```r
reshape(longx, v.names = "invest", idvar = "id", timevar = "time", direction = "wide")
```

```r
 id invest.1 invest.2
1.1 1 100 200
2.1 2 101 201
3.1 3 102 202
```

if the long format misses some ids:

```r
longx[-3, ]
```

```r
 id time invest
1.1 1 1 100
2.1 2 1 101
1.2 1 2 200
2.2 2 2 201
3.2 3 2 202
```

```r
reshape(longx[-3, ], v.names = "invest", idvar = "id", timevar = "time", direction = "wide")
```

```r
 id invest.1 invest.2
1.1 1 100 200
2.1 2 101 201
3.2 3 NA 202
```

84
10 Plotting

10.1 The standard plot interface

R has different plotting routines for a large variety of objects. A $n \times 2$ matrix, e.g., is plotted as a scatterplot:

A simple plot

```r
plot(x2)
```
Adding elements to a plot

We can add arbitrary elements to a plot:

```r
code
plot(x2)
abline(h = 0)
title(main = "The impact of education on income")
est <- lm(exp ~ Intercept, data = x2)
abline(est, col = "green", lwd = 3)
text(x2, labels = rownames(x2), pos = 4, col = "blue")
```

Adding confidence bands

Confidence bands and polynomial approximations can be calculated with `predict`:

```r
code
par(mar = c(4, 4, 3, 0.1))
plot(x2)
title(main = "The impact of education on income")
est.pred <- as.data.frame(predict(est, interval = "confidence"))
est.pred$Intercept <- x2$Intercept
attach(est.pred)
lines(Intercept, upr, lty = "dotted")
lines(Intercept, lwr, lty = "dotted")
detach(est.pred)
est.pred4 <- predict(lm(exp ~ poly(Intercept, degree = 4), data = x2))
lines(x2$Intercept, est.pred4, lty = "dashed")
```
Adding smoothness

Using only the original data did not produce an entirely satisfactory result. To get a more smooth curve, we produce a new dataset with a finer resolution:

```r
plot(x2)
title(main = "The impact of education on income")
x4 <- list(Intercept = seq(5, 8, 0.02))
est.pred4 <- predict(lm(exp ~ poly(Intercept, degree = 4), data = x2), newdata = x4)
lines(x4$Intercept, est.pred4)
```
Several polynomials

If we want to look at more than a single polynomial, we can try a loop:

```r
plot(x2)
for (i in c(1:4)) {
  est.pred4 <- predict(lm(exp ~ poly(Intercept, degree = i), data = x2), newdata = x4)
  lines(x4$Intercept, est.pred4, lty = i)
}
legend("bottomleft", legend = sprintf("%d-deg.", 1:4), lty = 1:4)
```

Kernel density plot

Here is another way to organise the data with the help of a kernel density plot:

```r
library(ks)
H.scv <- Hscv(x2)
kdeObj <- kde(x2, H = H.scv)
plot(kdeObj)
points(x2, col = "red")
lines(x4$Intercept, est.pred4)
```
A boxplot

R can also draw boxplots:

```r
boxplot(lwage ~ ed, data = Wages)
title(main = "Wages and Education", xlab = "Education/years", ylab = "log(Wage)")
est <- lm(lwage ~ poly(ed, degree = 2), data = Wages)
lines(1:14, predict(est, newdata = list(ed = c(4:17))), col = "blue", lwd = 3)
```

Survival analysis
Here is a survival analysis. A Surv() object is just a special kind of dependent variable.

```r
library(survival)
library(MASS)
plot(log(time) ~ pair, data = gehan)
```

Survival analysis

```r
qq <- Surv(gehan$time, gehan$cens)
gehan.surv <- survfit(Surv(time, cens) ~ treat, data = gehan, conf.type = "log-log")
plot(gehan.surv, col = 3:2, log = TRUE)
Warning: 1 y value <= 0 omitted from logarithmic plot
```
### Survival analysis

```r
summary(gehan.surv)

Call: survfit(formula = Surv(time, cens) ~ treat, data = gehan, conf.type = "log-log")

treat=6-MP
time n.risk n.event survival std.err lower 95% CI upper 95% CI
6  21     3 0.857  0.0764  0.620    0.952
7  17     1 0.807  0.0869  0.563    0.923
10  15     1 0.753  0.0963  0.503    0.889
13  12     1 0.690  0.1068  0.432    0.849
16  11     1 0.627  0.1141  0.368    0.805
[ reached getOption("max.print") -- omitted 2 rows ]

treat=control
time n.risk n.event survival std.err lower 95% CI upper 95% CI
1  21     2 0.9048 0.0641  0.67005   0.975
2  19     2 0.8095 0.0857  0.56891   0.924
3  17     1 0.7619 0.0929  0.51939   0.893
4  16     2 0.6667 0.1029  0.42535   0.825
5  14     2 0.5714 0.1080  0.33798   0.749
[ reached getOption("max.print") -- omitted 7 rows ]
```

```r
plot(gehan.surv, conf.int = TRUE, col = 3:2, log = TRUE)

Warning: 1 y value <= 0 omitted from logarithmic plot

lines(gehan.surv, col = 3:2, lwd = 3, cex = 2)
title(xlab = "time of remission (weeks)", ylab = "survival")
legend(25, 0.1, c("control", "6-MP"), col = 2:3, lwd = 2)
```

Survival analysis

\texttt{survdiff(Surv(time, cens) ~ treat, data = gehan)}

Call:
survdiff(formula = Surv(time, cens) ~ treat, data = gehan)

\begin{verbatim}
    N  Observed  Expected  (O-E)^2/E  (O-E)^2/V
treat=6-MP  21     9     19.3     5.46     16.8
    treat=control  21  21     10.7     9.77     16.8
\end{verbatim}

\texttt{Chisq} = 16.8 on 1 degrees of freedom, p = 4.17e-05