THE BIG-R BOOK

FROM DATA SCIENCE TO LEARNING MACHINES AND BIG DATA

— PART 02—

Dr. Philippe J.S. De Brouwer

last compiled: September 1, 2021
Version 0.1.1

(c) 2021 Philippe J.S. De Brouwer – distribution allowed by John Wiley & Sons, Inc.

THE BIG R-BOOK:
From Data Science to Big Data and Learning Machines

♡– PART 02: Starting with R and Elements of Statistics –♡

(c) 2021 by Philippe J.S. De Brouwer – distribution allowed by John Wiley & Sons, Inc.

These slides are to be used in with the book – for best experience, teachers will read the book *before* using the slides and students have access to the book and the code.

part 02: Starting with R and Elements of Statistics

chapter 4:

The Basics of R

Essentially, R is . . .

- a programming language built for statistical analysis, graphics representation and reporting;
- an interpreted computer language which allows branching, looping, modular programming as well as object and functional oriented programming features.

The Main Features of R

R offers its users...

- integration with the procedures written in the C, C++, .Net, Python, or FORTRAN languages for efficiency;
- zero purchase cost (available under the GNU General Public License), and pre-compiled binary versions are provided for various operating systems like Linux, Windows, and Mac;
- simplicity and effectiveness;
- a free and open environment;
- an effective data handling and storage facility;
- a suite of operators for calculations on arrays, lists, vectors, and matrices;
- a large, coherent, and integrated collection of tools for data analysis;
- graphical facilities for data analysis and display either directly at the computer or printing;
- a supportive on-line community;
- the ability for you to stand on the shoulders of giants (e.g. by using libraries).

R is arguably the most widely used statistics programming language and is used from universities to business applications, while it still gains rapidly in popularity.

© Dr. Philippe J.S. De Brouwer

part 02: Starting with R and Elements of Statistics

chapter 4: The Basics of R

section 1:

Getting Started with R

- You need a working installation of R on your computer.
- R is available for Mac, Linux, and Windows from `https://cran.r-project.org`
- To start R, open the command line and type `R` (followed by enter).
- You should then get the command line prompt of R. It is of course also possible to use a graphical interface such as RStudio (see `https://www.rstudio.com`).

Hint – Using R Online

It is also possible to use R online:

- `https://www.tutorialspoint.com/execute_r_online.php`
- `http://www.r-fiddle.org`

© Dr. Philippe J.S. De Brouwer

RStudio

For the user, who is not familiar with the command line, it is highly recommendable to use an IDE, such as RStudio (see `https://www.rstudio.com`). Later on

we will see that RStudio has some unique advantages over the R-console in store, that will convince even the most traditional command-line-users.

Whether you use standard R or MRAN, using RStudio will enhance your performance and help you to be more productive. Rstudio is an integrated development environment (IDE) for R and provides a console, editor with syntax-highlighting, a window to show plots and some workspace management.

Basic arithmetic

The basic operators work as one would expect. Simply type in the R terminal `2+3` followed by ENTER and R will immediately display the result.

```
# Addition:
2 + 3
## [1] 5

# Product:
2 * 3
## [1] 6

# Power:
2**3
## [1] 8

2^3
## [1] 8

# Logic:
2 < 3
## [1] TRUE

x <- c(1,3,4,3)
x.mean <- mean(x)
x.mean
## [1] 2.75
```

the `scan()` function

To create a variable `x` via an editor, type:

```
x <- scan()
```

This code will start an interface that invites you to type all values of the vector one by one. In order to get back to the command prompt: type enter without typing a number (ie. leave one empty to end).

Editing variables

To modify an existing variable, one can use the `edit()` function

```
edit(x)
```

Batch mode

1. create a file `test.R`
2. add the content `print("Hello World")`
3. run the command line `Rscript test.R`
4. now, open R and run the command `source("test.R")`
5. add in the file

```
my_function <- function(a,b)
{
  a +  b
}
```

6. now repeat step 4 and run `my_function(4,5)`

© Dr. Philippe J.S. De Brouwer

PART 02: STARTING WITH R AND ELEMENTS OF STATISTICS

CHAPTER 4: THE BASICS OF R

SECTION 2:

Variables

Valid Variables

In R, variables

- can contain letters as well as "_" (underscore) and "." (dot), and
- variables must start with a letter (that can be preceded with a dot).

For example, `my_var.1` and `my.Cvar` are valid variables, but `_myVar`, `my%var` and `1.var` are not acceptable.

Assignment

Assignment can be made left or right:

```
# x.1 is assigned the value 5:
x.1 <- 5

# The result of x.1 + 3 is stored in .x:
x.1 + 3 -> .x

# Show the result:
print(.x)
## [1] 8
```

Variable Management

```r
# List all variables:
ls()                        # hidden variable starts with dot
ls(all.names = TRUE)        # shows all

# Remove a variable:
rm(x.1)                     # removes the variable x.1
ls()                        # x.1 is not there any more
rm(list = ls())             # removes all variables
ls()
```

PART 02: STARTING WITH R AND ELEMENTS OF STATISTICS

CHAPTER 4: THE BASICS OF R

SECTION 3:

Data Types

© DR. PHILIPPE J.S. DE BROUWER

Base Data Types i

There is no need to declare variables explicitly and tell R what type the variable will be before using it. R will assign them a class whenever this is needed and even change the type when our code implies a change.

© Dr. Philippe J.S. De Brouwer

Base Data Types ii

```r
# Booleans can be TRUE or FALSE:
x <- TRUE
class(x)
## [1] "logical"

# Integers use the letter L (for Long integer):
x <- 5L
class(x)
## [1] "integer"

# Decimal numbers, are referred to as 'numeric':
x <- 5.135
class(x)
## [1] "numeric"

# Complex numbers use the letter i (without multiplication sign):
x <- 2.2 + 3.2i
class(x)
## [1] "complex"

# Strings are called 'character':
x <- "test"
class(x)
## [1] "character"
```

Dates

```
# The function as.Data coerces its argument to a date:
d <- as.Date(c("1852-05-12", "1914-11-5", "2015-05-01"))

# Dates will work as expected:
d_recent <- subset(d, d > as.Date("2005-01-01"))
print(d_recent)
## [1] "2015-05-01"
```

© Dr. Philippe J.S. De Brouwer

Simply put, vectors are lists of objects that are all of the same type. They can be the result of a calculation or be declared with the function `c()`. The following code generates two vectors of different types.

```
x <- c(2, 2.5, 4, 6)
y <- c("apple", "pear")
class(x)
## [1] "numeric"

class(y)
## [1] "character"
```

© Dr. Philippe J.S. De Brouwer

Accessing Elements of a Vector

```
# Create v as a vector of the numbes one to 5:
v <- c(1:5)

# Access elements via indexing:
v[2]
## [1] 2

v[c(1,5)]
## [1] 1 5

# Access via TRUE/FALSE:
v[c(TRUE,TRUE,FALSE,FALSE,TRUE)]
## [1] 1 2 5

# Access elements via names:
v <- c("pear" = "green", "banana" = "yellow", "coconut" = "brown")
v
##     pear   banana  coconut
##  "green" "yellow"  "brown"

v["banana"]
##   banana
## "yellow"

# Leave out certain elements:
v[c(-2,-3)]
##    pear
```

The standard behaviour for vector arithmetic in R is element per element. With “standard” we mean operators that do not appear between percentage signs (such as `%.%` for example).

```
# Define two vectors:
v1 <- c(1,2,3)
v2 <- c(4,5,6)

# Standard arithmetic:
v1 + v2
## [1] 5 7 9

v1 - v2
## [1] -3 -3 -3

v1 * v2
## [1]  4 10 18
```

Vector Recycling

```
# Define a short and long vector:
v1 <- c(1, 2, 3, 4, 5)
v2 <- c(1, 2)

# Note that R 'recycles' v2 to match the length of v1:
v1 + v2
## [1] 2 4 4 6 6
```

© Dr. Philippe J.S. De Brouwer

Vector Sorting

To sort a vector, we can use the function `sort()`.

```
# Example 1:
v1 <- c(1, -4, 2, 0, pi)
sort(v1)
## [1] -4.000000  0.000000  1.000000  2.000000  3.141593

# Example 2: To make sorting meaningful, all variables are coerced to
# the most complex type:
v1 <- c(1:3, 2 + 2i)
sort(v1)
## [1] 1+0i 2+0i 2+2i 3+0i

# Sorting is per increasing numerical or alphabetical order:
v3 <- c("January", "February", "March", "April")
sort(v3)
## [1] "April"    "February" "January"  "March"

# Sort order can be reversed:
sort(v3, decreasing = TRUE)
## [1] "March"    "January"  "February" "April"
```

© Dr. Philippe J.S. De Brouwer

Question #1 Temperature conversion

The time series `nottem` (from the package “datasets” that is usually loaded when R starts) contains the temperatures in Notthingham from 1920 to 1939 in Fahrenheit. Create a new object that contains a list of all temperatures in Celsius.

Hint – Addressing the object nottem

Note that `nottem` is a time series object and not a matrix. Its elements are addressed with `nottam[n]` where `n` is between `1` and `length(nottam)`. However, when printed it will look like a matrix with months in the columns and years in the rows. This is because the print-function will use functionality specific to the time series object.[a]
Remember that $T(C) = \frac{5}{9}(T(F) - 32)$.

[a] This behaviour is caused by the dispatcher-function implementation of an object-oriented programming model. To understand how this works and what it means, we refer to Section **??** “**??**” on page **??**.

A matrix is in two-dimensional data set where all elements are of the same type. The `matrix()` function offers a convenient way to define it:

```
# Create a matrix:
M <- matrix( c(1:6), nrow = 2, ncol = 3, byrow = TRUE)

# Show it on the screen:
print(M)
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6

M <- matrix( c(1:6), nrow = 2, ncol = 3, byrow = FALSE)
print(M)
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

© Dr. Philippe J.S. De Brouwer

Naming Rows and Columns

While in general naming rows and/or columns is more relevant for datasets than matrices it is possible to work with matrices to store data if it only contains one type of variable.

```
# Store the names in a vector:
row_names = c("row1", "row2", "row3", "row4")
col_names = c("col1", "col2", "col3")

# Create the matrix:
M <- matrix(c(10:21), nrow = 4, byrow = TRUE,
            dimnames = list(row_names, col_names))

# Display the result:
print(M)
##      col1 col2 col3
## row1   10   11   12
## row2   13   14   15
## row3   16   17   18
## row4   19   20   21
```

© Dr. Philippe J.S. De Brouwer

Accessing Data in a Matrix

```
M <- matrix(c(10:21), nrow = 4, byrow = TRUE)
M
##      [,1] [,2] [,3]
## [1,]   10   11   12
## [2,]   13   14   15
## [3,]   16   17   18
## [4,]   19   20   21

# Access one element:
M[1,2]
## [1] 11

# The second row:
M[2,]
## [1] 13 14 15

# The second column:
M[,2]
## [1] 11 14 17 20

# Row 1 and 3 only:
M[c(1, 3),]
##      [,1] [,2] [,3]
## [1,]   10   11   12
## [2,]   16   17   18

# Row 2 to 3 with column 3 to 1:
```

Matrix Arithmetic

Basic arithmetic on matrices works element by element:

```
M1 <- matrix(c(10:21), nrow = 4, byrow = TRUE)
M2 <- matrix(c(0:11),  nrow = 4, byrow = TRUE)
M1 + M2
##      [,1] [,2] [,3]
## [1,]   10   12   14
## [2,]   16   18   20
## [3,]   22   24   26
## [4,]   28   30   32

M1 * M2
##      [,1] [,2] [,3]
## [1,]    0   11   24
## [2,]   39   56   75
## [3,]   96  119  144
## [4,]  171  200  231

M1 / M2
##          [,1]      [,2]     [,3]
## [1,]      Inf 11.000000 6.000000
## [2,] 4.333333  3.500000 3.000000
## [3,] 2.666667  2.428571 2.250000
## [4,] 2.111111  2.000000 1.909091
```

Question #2 Dot product

Write a function for the dot-product for matrices. Add also some security checks. Finally, compare your results with the “%*%-operator.”

The dot-product is pre-defined via the `%*%` opeartor. Note that the function `t()` creates the transposed vector or matrix.

```
# Example of the dot-product:
a <- c(1:3)
a %*% a
##      [,1]
## [1,]   14

a %*% t(a)
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    2    4    6
## [3,]    3    6    9

t(a) %*% a
##      [,1]
## [1,]   14

# Define A:
A <- matrix(0:8, nrow = 3, byrow = TRUE)
```

Arrays can be created with the `array()` function; this function takes a “dim” attribute which defines the number of dimension. While arrays are similar to lists, they have to be of one class type (lists can consist of different class types).

In the example we create an array with two elements, which are both three by three matrices.

```
# Create an array:
a <- array(c('A','B'),dim = c(3,3,2))
print(a)
## , , 1
##
##      [,1] [,2] [,3]
## [1,] "A"  "B"  "A"
## [2,] "B"  "A"  "B"
## [3,] "A"  "B"  "A"
##
## , , 2
##
##      [,1] [,2] [,3]
## [1,] "B"  "A"  "B"
## [2,] "A"  "B"  "A"
## [3,] "B"  "A"  "B"

# Access one element:
a[2,2,2]
## [1] "B"

# Access one layer:
a[,,2]
##      [,1] [,2] [,3]
## [1,] "B"  "A"  "B"
## [2,] "A"  "B"  "A"
## [3,] "B"  "A"  "B"
```

Naming of Array Elements

```
# Create two vectors:
v1 <- c(1,1)
v2 <- c(10:13)
col.names <- c("col1","col2", "col3")
row.names <- c("R1","R2")
matrix.names <- c("Matrix1","Matrix2")

# Take these vectors as input to the array.
a <- array(c(v1,v2),dim = c(2,3,2),
     dimnames = list(row.names,col.names,
     matrix.names))
print(a)
## , , Matrix1
##
##    col1 col2 col3
## R1    1   10   12
## R2    1   11   13
##
## , , Matrix2
##
##    col1 col2 col3
## R1    1   10   12
## R2    1   11   13

# This allows to address the first row in Matrix 1 as follows:
a['R1',,'Matrix1']
## col1 col2 col3
##    1   10   12
```

Manipulating Arrays

```
M1 <- a[,,1]
M2 <- a[,,2]
M2
##    col1 col2 col3
## R1    1   10   12
## R2    1   11   13
```

An efficient way to apply the same function over each element of an array is via the function `apply()`: that functions is designed to do exactly that.

Function use for apply()

`apply(X, MARGIN, FUN, ...)` with:

1. X: an array, including a matrix.
2. MARGIN: a vector giving the subscripts which the function will be applied over. E.g., for a matrix '1' indicates rows, '2' indicates columns, 'c(1, 2)' indicates rows and columns. Where 'X' has named dimnames, it can be a character vector selecting dimension names.
3. FUN: the function to be applied: see 'Details'. In the case of functions like '+', 'backquoted or quoted

An example for apply()

```
x <- cbind(x1 = 3, x2 = c(4:1, 2:5))
dimnames(x)[[1]] <- letters[1:8]
apply(x, 2, mean, trim = .2)
## x1 x2
##  3  3

col.sums <- apply(x, 2, sum)
row.sums <- apply(x, 1, sum)
rbind(cbind(x, Rtot = row.sums),
      Ctot = c(col.sums, sum(col.sums)))
##      x1 x2 Rtot
## a     3  4    7
## b     3  3    6
## c     3  2    5
## d     3  1    4
## e     3  2    5
## f     3  3    6
## g     3  4    7
## h     3  5    8
## Ctot 24 24   48
```

Lists: definition

Definition 1 (List)

In R, lists are objects which are sets of elements that are not necessarily all of the same type. Lists can mix numbers, strings, vectors, matrices, functions, boolean variables, and even lists.

```
# Create a list with the list() function:
myList <- list("Approximation", pi, 3.14, c)

# Display the result:
print(myList)
## [[1]]
## [1] "Approximation"
##
## [[2]]
## [1] 3.141593
##
## [[3]]
## [1] 3.14
##
## [[4]]
## function (...)  .Primitive("c")
```

© Dr. Philippe J.S. De Brouwer

Naming Elements of Lists i

```
# Create the list:
L <- list("Approximation", pi, 3.14, c)

# Assign names to elements:
names(L) <- c("description", "exact", "approx","function")

# Show the result:
print(L)
## $description
## [1] "Approximation"
##
## $exact
## [1] 3.141593
##
## $approx
## [1] 3.14
##
## $`function`
## function (...)  .Primitive("c")
```

```
# Addressing elements of the named list:
print(paste("The difference is", L$exact - L$approx))
## [1] "The difference is 0.00159265358979299"

print(L[3])
## $approx
## [1] 3.14

print(L$approx)
## [1] 3.14

# However, "function" was a reserved word, so we need to use
# back-ticks in order to address the element:
a <- L$`function`(2,3,pi,5)  # to access the function c(...)
print(a)
## [1] 2.000000 3.000000 3.141593 5.000000
```

Lists of Lists Are Also Lists

```
# Start with a vector:
V1 <- c(1,2,3)

# Define two lists:
L2 <- list(V1, c(2:7))
L3 <- list(L2,V1)

# Show the results:
print(L3)
## [[1]]
## [[1]][[1]]
## [1] 1 2 3
##
## [[1]][[2]]
## [1] 2 3 4 5 6 7
##
##
## [[2]]
## [1] 1 2 3

print(L3[[1]][[2]][3])
## [1] 4
```

Add and Delete Elements of a List i

A numbered element can be added while skipping positions. In the following example the position 3 is left undefined (NULL).

```
# Define a simple list:
L <- list(1,2)

# Coerce the fourth position to 4:
L[4] <- 4  # position 3 is NULL

# Show the results:
L
## [[1]]
## [1] 1
##
## [[2]]
## [1] 2
##
## [[3]]
## NULL
##
## [[4]]
## [1] 4
```

Named elements are always added at the end of the list:

Add and Delete Elements of a List ii

```
L$pi_value <- pi
L
## [[1]]
## [1] 1
##
## [[2]]
## [1] 2
##
## [[3]]
## NULL
##
## [[4]]
## [1] 4
##
## $pi_value
## [1] 3.141593
```

Delete an element by assigning NULL to it. Notice that the elements get re-ordered. This means that if we address the elements of a list by their number, we need to recalculate the numbers. If we were addressing the elements of the list by name, nothing needs to be changed.

Add and Delete Elements of a List iii

```
L[1] <- NULL
L
## [[1]]
## [1] 2
##
## [[2]]
## NULL
##
## [[3]]
## [1] 4
##
## $pi_value
## [1] 3.141593
```

It is also possible to delete an element via the squared brackets.

```
L <- L[-2]
L
## [[1]]
## [1] 2
##
## [[2]]
## [1] 4
##
## $pi_value
## [1] 3.141593
```

Convert list to vectors

```r
# The list:
L  <- list(c(1:5), c(6:10))

# The vectors obtained from the list:
v1 <- unlist(L[1])
v2 <- unlist(L[2])

# Show the results:
v1
## [1] 1 2 3 4 5


v2
## [1]  6  7  8  9 10


v2-v1
## [1] 5 5 5 5 5
```

Factors

Factors are created using the `factor()` function.

```
# Create a vector containing all your observations:
feedback <- c('Good','Good','Bad','Average','Bad','Good')

# Create a factor object:
factor_feedback <- factor(feedback)

# Print the factor object:
print(factor_feedback)
## [1] Good    Good    Bad     Average Bad     Good
## Levels: Average Bad Good
```

Plotting factor objects i

```
# Plot the histogram -- note the default order is alphabetic
plot(factor_feedback)
```

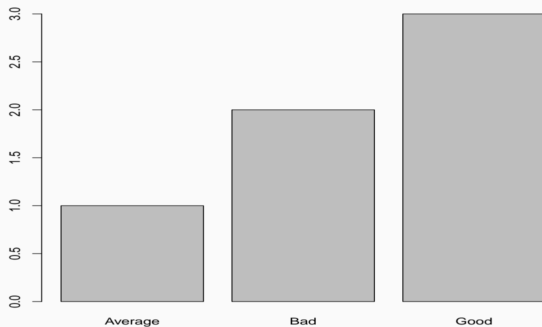


Figure 1: The plot-function will result in a bar-chart for a factor-object.

There are a few specific functions for the factor-object. For example, the function `nlevels()` returns the number of levels in the factor object.

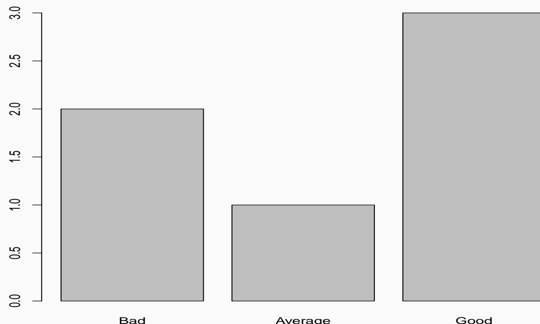
```
# The nlevels function returns the number of levels:
print(nlevels(factor_feedback))
## [1] 3
```

Ordering Factors

```
# Store the survey results:
feedback <- c('Good','Good','Bad','Average','Bad','Good')

# Define the factors while providing the levels in right order:
factor_feedback <- factor(feedback,
                          levels = c("Bad", "Average", "Good"))

# Display results:
plot(factor_feedback)
```



Generate Factors with the Function gl()

Function use for gl()

```
gl(n, k, length = n*k, labels = seq_len(n), ordered = FALSE)
```

with

- n: The number of levels
- k: The number of replications (for each level)
- length (optional): An integer giving the length of the result
- labels (optional): A vector with the labels
- ordered: A boolean variable indicating whether the results should be ordered.

```
gl(3,2,,c("bad","average","good"),TRUE)
## [1] bad     bad     average average good    good
## Levels: bad < average < good
```

© Dr. Philippe J.S. De Brouwer

Question #3

Use the dataset mtcars (from the library MASS) and explore the distribution of number of gears. Then explore the correlation between gears and transmission.

Question #4

Then focus on the transmission and create a factor-object with the words “automatic” and “manual” instead of the numbers 0 and 1.

Use the `?mtcars` to find out the exact definition of the data.

Question #5

Use the dataset mtcars (from the library MASS) and explore the distribution of the horsepower (hp). How would you proceed to make a factoring (e.g. Low, Medium, High) for this attribute? Hint: Use the function `cut()`.

Data frames are created using the `data.frame()` function.

```
# Create the data frame.
data_test <- data.frame(
   Name   = c("Piotr", "Pawel","Paula","Lisa","Laura"),
   Gender = c("Male", "Male","Female", "Female","Female"),
   Score  = c(78,88,92,89,84),
   Age    = c(42,38,26,30,35)
   )
print(data_test)
##    Name Gender Score Age
## 1 Piotr   Male    78  42
## 2 Pawel   Male    88  38
## 3 Paula Female    92  26
## 4  Lisa Female    89  30
## 5 Laura Female    84  35

# The standard plot function on a data-frame (Figure 4.3)
# is the same as using the pairs() function:
plot(data_test)
```

© Dr. Philippe J.S. De Brouwer

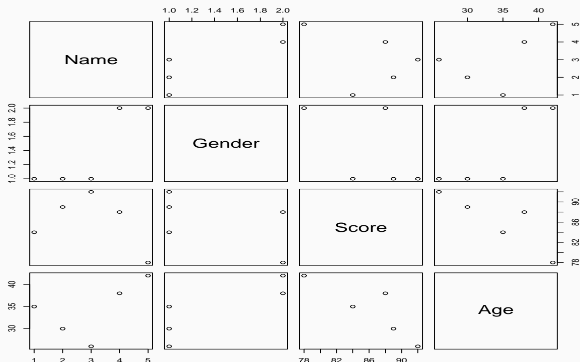


Figure 3: The standard plot for a data frame in R shows each column printed in function of each other. This is useful to see correlations or how generally the data is structured.

© Dr. Philippe J.S. De Brouwer

Get Information About Data Frames i

```
# Get the structure of the data frame:
str(data_test)
## 'data.frame': 5 obs. of  4 variables:
##  $ Name  : Factor w/ 5 levels "Laura","Lisa",..: 5 4 3 2 1
##  $ Gender: Factor w/ 2 levels "Female","Male": 2 2 1 1 1
##  $ Score : num  78 88 92 89 84
##  $ Age   : num  42 38 26 30 35

# Note that the names became factors.
```

```
# Get the summary of the data frame:
summary(data_test)
##     Name      Gender      Score           Age
##  Laura:1   Female:3   Min.   :78.0   Min.   :26.0
##  Lisa :1   Male  :2   1st Qu.:84.0   1st Qu.:30.0
##  Paula:1              Median :88.0   Median :35.0
##  Pawel:1              Mean   :86.2   Mean   :34.2
##  Piotr:1              3rd Qu.:89.0   3rd Qu.:38.0
##                       Max.   :92.0   Max.   :42.0
```

Get Information About Data Frames ii

```
# Get the first rows:
head(data_test)
##    Name Gender Score Age
## 1 Piotr   Male    78  42
## 2 Pawel   Male    88  38
## 3 Paula Female    92  26
## 4  Lisa Female    89  30
## 5 Laura Female    84  35
```

```
# Get the last rows:
tail(data_test)
##    Name Gender Score Age
## 1 Piotr   Male    78  42
## 2 Pawel   Male    88  38
## 3 Paula Female    92  26
## 4  Lisa Female    89  30
## 5 Laura Female    84  35
```

Get Information About Data Frames iii

```
# Extract the column 2 and 4 and keep all rows
data_test.1 <- data_test[,c(2,4)]
print(data_test.1)
##   Gender Age
## 1   Male  42
## 2   Male  38
## 3 Female  26
## 4 Female  30
## 5 Female  35
```

```
# Extract columns by name and keep only selected rows
data_test[c(2:4),c(2,4)]
##   Gender Age
## 2   Male  38
## 3 Female  26
## 4 Female  30
```

Editing Data in Data-frames

```r
de(x)                      # fails if x is not defined
de(x <- c(NA))             # works
x <- de(x <- c(NA))        # will also save the changes in x
data.entry(x)              # de is short for data.entry
x <- edit(x)               # use the standard editor (vi in *nix)
```

Add Columns to a Data-frame i

```
# Expand the data frame, simply define the additional column:
data_test$End_date <-  as.Date(c("2014-03-01", "2017-02-13",
                     "2014-10-10", "2015-05-10","2010-08-25"))
print(data_test)
##    Name Gender Score Age   End_date
## 1 Piotr   Male    80  42 2014-03-01
## 2 Pawel   Male    88  38 2017-02-13
## 3  <NA> Female    92  26 2014-10-10
## 4  Lisa Female    89  30 2015-05-10
## 5 Laura Female    84  35 2010-08-25
```

Add Columns to a Data-frame ii

```
# Or use the function cbind() to combine data frames along columns:
Start_date <- as.Date(c("2012-03-01", "2013-02-13",
                        "2012-10-10", "2011-05-10","2001-08-25"))

# Use this vector directly:
df <- cbind(data_test, Start_date)
print(df)
##    Name Gender Score Age   End_date Start_date
## 1 Piotr   Male    80  42 2014-03-01 2012-03-01
## 2 Pawel   Male    88  38 2017-02-13 2013-02-13
## 3  <NA> Female    92  26 2014-10-10 2012-10-10
## 4  Lisa Female    89  30 2015-05-10 2011-05-10
## 5 Laura Female    84  35 2010-08-25 2001-08-25

# or first convert to a data frame:
df <- data.frame("Start_date" = t(Start_date))
df <- cbind(data_test, Start_date)
print(df)
##    Name Gender Score Age   End_date Start_date
## 1 Piotr   Male    80  42 2014-03-01 2012-03-01
## 2 Pawel   Male    88  38 2017-02-13 2013-02-13
## 3  <NA> Female    92  26 2014-10-10 2012-10-10
## 4  Lisa Female    89  30 2015-05-10 2011-05-10
## 5 Laura Female    84  35 2010-08-25 2001-08-25
```

Adding Rows to a Data-frame

```
# To add a row, we need the rbind() function:
data_test.to.add <- data.frame(
   Name = c("Ricardo", "Anna"),
   Gender = c("Male", "Female"),
   Score = c(66,80),
   Age = c(70,36),
   End_date = as.Date(c("2016-05-05","2016-07-07"))
   )
data_test.new <- rbind(data_test,data_test.to.add)
print(data_test.new)
##      Name Gender Score Age   End_date
## 1   Piotr   Male    80  42 2014-03-01
## 2   Pawel   Male    88  38 2017-02-13
## 3    <NA> Female    92  26 2014-10-10
## 4    Lisa Female    89  30 2015-05-10
## 5   Laura Female    84  35 2010-08-25
## 6 Ricardo   Male    66  70 2016-05-05
## 7    Anna Female    80  36 2016-07-07
```

Merging data frames

Merging allows to extract the subset of two data-frames where a given set of columns match.

```
data_test.1 <- data.frame(
   Name = c("Piotr", "Pawel","Paula","Lisa","Laura"),
   Gender = c("Male", "Male","Female", "Female","Female"),
   Score = c(78,88,92,89,84),
   Age = c(42,38,26,30,35)
   )
data_test.2 <- data.frame(
   Name = c("Piotr", "Pawel","notPaula","notLisa","Laura"),
   Gender = c("Male", "Male","Female", "Female","Female"),
   Score = c(78,88,92,89,84),
   Age = c(42,38,26,30,135)
   )
data_test.merged <- merge(x=data_test.1,y=data_test.2,
                          by.x=c("Name","Age"),by.y=c("Name","Age"))

# Only records that match in name and age are in the merged table:
print(data_test.merged)
##    Name Age Gender.x Score.x Gender.y Score.y
## 1 Pawel  38     Male      88     Male      88
## 2 Piotr  42     Male      78     Male      78
```

© Dr. Philippe J.S. De Brouwer

R will allow the use of short-cuts, provided that they are unique. For example, in the data-frame `data_test` there is a column `Name`. There are no other columns whose name start with the letter “N”; hence. this one letter is enough to address this column.

```
# Use 'N' to refer to 'Name'
data_test$N
## [1] Piotr Pawel <NA>  Lisa  Laura
## Levels: Laura Lisa Paula Pawel Piotr
```

© Dr. Philippe J.S. De Brouwer

Rows and Column Names in Data Frames

```
# Get the rownames:
colnames(data_test)
## [1] "Name"      "Gender"    "Score"     "Age"       "End_date"

# Access the rownames:
rownames(data_test)
## [1] "1" "2" "3" "4" "5"

colnames(data_test)[2]
## [1] "Gender"

rownames(data_test)[3]
## [1] "3"

# Assign new names:
colnames(data_test)[1] <- "first_name"
rownames(data_test) <- LETTERS[1:nrow(data_test)]
print(data_test)
##   first_name Gender Score Age   End_date
## A      Piotr   Male    80  42 2014-03-01
## B      Pawel   Male    88  38 2017-02-13
## C       <NA> Female    92  26 2014-10-10
## D       Lisa Female    89  30 2015-05-10
## E      Laura Female    84  35 2010-08-25
```

Simple rules for strings

- strings must start and end with single or double quotes,
- a string ends when the same quotes are encountered the next time,
- until then it can contain the other type of quotes.

Example (Using strings)

```
a <- "Hello"
b <- "world"
paste(a, b, sep = ", ")
## [1] "Hello, world"

c <- "A 'valid' string"
```

Function use for format()

```
format(x, trim = FALSE, digits = NULL, nsmall = 0L,
       justify = c("left", "right", "centre", "none"),
       width = NULL, na.encode = TRUE, scientific = NA,
       big.mark   = "",   big.interval = 3L,
       small.mark = "", small.interval = 5L,
       decimal.mark = getOption("OutDec"),
       zero.print = NULL, drop0trailing = FALSE, ...)
```

- `x` is the vector input.
- `digits` is the total number of digits displayed.
- `nsmall` is the minimum number of digits to the right of the decimal point.
- `scientific` is set to TRUE to display scientific notation.
- `width` is the minimum width to be displayed by padding blanks in the beginning.
- `justify` is the display of the string to left, right or center.

```
a <- format(100000000,big.mark=" ",
                  nsmall=3,
                  width=20,
                  scientific=FALSE,
                  justify="r")
print(a)
## [1] "       100 000 000.000"
```

Further information – format()

More information about the format-function can be obtained via `?format` or `help(format)`.

Other string functions

- `nchar()`: returns the number of characters in a string
- `toupper()`: puts the string in uppercase
- `tolower()`: puts the string in lowercase
- `substring(x,first,last)`: returns a substring from x starting with the “first” and ending with the “last”
- `strsplit(x,split)`: split the elements of a vector into substrings according to matches of a substring “split.”
  there is also a family of search functions: `grep()`, `grepl()`, `regexpr()`, `gregexpr()`, and `regexec()` that supply powerful search and replace capabilities.
  `sub()` will replace the first of all matches and `gsub()` will replace all matches.

PART 02: STARTING WITH R AND ELEMENTS OF STATISTICS

CHAPTER 4: THE BASICS OF R

SECTION 4:

Operators

© DR. PHILIPPE J.S. DE BROUWER

Arithmetic operators act on each element of an object

```
v1 <- c(2,4,6,8)
v2 <- c(1,2,3,5)
v1 + v2     # addition
## [1]  3  6  9 13

v1 - v2     # subtraction
## [1] 1 2 3 3

v1 * v2     # multiplication
## [1]  2  8 18 40

v1 / v2     # division
## [1] 2.0 2.0 2.0 1.6

v1 %% v2    # remainder of division
## [1] 0 0 0 3

v1 %/% v2   # round(v1/v2 -0.5)
## [1] 2 2 2 1

v1 ^ v2     # v1 to the power of v2
## [1]     2    16   216 32768
```

Relational Operators compare vectors element by element

```
v1 <- c(8,6,3,2)
v2 <- c(1,2,3,5)
v1 > v2     # bigger than
## [1]  TRUE  TRUE FALSE FALSE

v1 < v2     # smaller than
## [1] FALSE FALSE FALSE  TRUE

v1 <= v2    # smaller or equal
## [1] FALSE FALSE  TRUE  TRUE

v1 >= v2    # bigger or equal
## [1]  TRUE  TRUE  TRUE FALSE

v1 == v2    # equal
## [1] FALSE FALSE  TRUE FALSE

v1 != v2    # not equal
## [1]  TRUE  TRUE FALSE  TRUE
```

© Dr. Philippe J.S. De Brouwer

Logical Operators i

Logical Operators combine vectors element by element. While logical operators can be applied directly on composite types, they must be able to act on numeric, logical or complex types in order to produce understandable results.

```
# The vectors:
v1 <- c(TRUE, TRUE, FALSE, FALSE)
v2 <- c(TRUE, FALSE, FALSE, TRUE)

# The basic logical operations:
v1 & v2     # and
## [1]  TRUE FALSE FALSE FALSE

v1 | v2     # or
## [1]  TRUE  TRUE FALSE  TRUE

!v1         # not
## [1] FALSE FALSE  TRUE  TRUE

v1 && v2    # and applied to the first element
## [1] TRUE

v1 || v2    # or applied to the first element
## [1] TRUE
```

Logical Operators ii

```r
# More aspects of logical values:
v1 <- c(TRUE, FALSE, TRUE, FALSE, 8, 6+3i, -2, 0, NA)

class(v1)  # v1 is a vector or complex numbers
## [1] "complex"

v2 <- c(TRUE)
as.logical(v1)  # coerce to logical (only 0 is FALSE)
## [1]  TRUE FALSE  TRUE FALSE  TRUE  TRUE  TRUE FALSE    NA

v1 & v2
## [1]  TRUE FALSE  TRUE FALSE  TRUE  TRUE  TRUE FALSE    NA

v1 | v2
## [1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
```

Assignment operators are left or right

```
# left assignment
x <- 3
x = 3
x <<- 3

# right assignment
3 -> x
3 ->> x

#chained assignment
x <- y <- 4
```

```
# create a list
x <- c(10:20)
x
##  [1] 10 11 12 13 14 15 16 17 18 19 20

# %in% can find an element in a vector
2  %in% x   # FALSE since 2 is not an element of x
## [1] FALSE

11 %in% x   # TRUE since 11 is in x
## [1] TRUE

x[x %in% c(12,13)] # selects elements from x
## [1] 12 13

x[2:4]   # selects the elements with index
## [1] 11 12 13

            # between 2 and 4
```

PART 02: STARTING WITH R AND ELEMENTS OF STATISTICS

CHAPTER 4: THE BASICS OF R

SECTION 5:

Flow Control Statements

The for-loop is useful to repeat a block of code a certain number of times. R will iterate a given variable through elements of a vector.

Function use for for()

```
for (value in vector) {
   statements
}
```

The for-loop will execute the statements for each value in the given vector.

Example (For loop)

```
x <- LETTERS[1:5]
for ( j in x) {
   print(j)
}
## [1] "A"
## [1] "B"
## [1] "C"
## [1] "D"
## [1] "E"
```

© Dr. Philippe J.S. De Brouwer

The repeat-loop will repeat the block of commands till it executes the `break` command.

Function use for repeat()

```
repeat {
   commands
   if(condition) {break}
}
```

Example (Repeat loop)

```
x <- c(1,2)
c <- 2
repeat {
   print(x+c)
   c <- c+1
   if(c > 4) {break}
}
## [1] 3 4
## [1] 4 5
## [1] 5 6
```

While

The while-loop is similar to the repeat-loop. However, the while-loop will first check the condition and then run the code to be repeated. So, this code might not be executed at all.

Function use for while()

```
while (test_expression) {
   statement
}
```

The statements are executed as long the test_expression is TRUE.

Example (While loop)

```
x <- c(1,2); c <- 2
while (c < 4) {
   print(x+c)
   c <- c + 1
}
## [1] 3 4
## [1] 4 5
```

Loop Control Statements

When the `break` statement is encountered inside a loop, that loop is immediately terminated and program control resumes at the next statement following the loop.

```
v <- c(1:5)
for (j in v) {
   if (j == 3) {
      print("--break--")
      break
   }
   print(j)
}
## [1] 1
## [1] 2
## [1] "--break--"
```

Loop control statements

The `next` statement will skip the remainder of the current iteration of a loop and starts next iteration of the loop.

```
v <- c(1:5)
for (j in v) {
   if (j == 3) {
      print("--skip--")
      next
   }
   print(j)
}
## [1] 1
## [1] 2
## [1] "--skip--"
## [1] 4
## [1] 5
```

part 02: Starting with R and Elements of Statistics

chapter 4: The Basics of R

section 6:

Functions

Built-in function

Right after starting, R some functions are available. We call these the “built-in functions.” Some examples are:

- `demo()`: shows some of the capabilities of R
- `q()`: quits R
- `data()`: shows the datasets available
- `help()`: shows help
- `ls()`: shows variables
- `c()`: creates a vector
- `seq()`: creates a sequence
- `mean()`: calculates the mean
- `max()`: returns the maximum
- `sum()`: returns the sum
- `paste()`: concatenates vector elements

© Dr. Philippe J.S. De Brouwer

Help with functions

```
help(c)   # shows help help with the function c
?c        # same result

apropos("cov") # fuzzy search for functions
```

© Dr. Philippe J.S. De Brouwer

User defined functions

Function use for function()

In R a user defined function (UDF) is created via the function `function()`.

```
function_name <- function(arg_1, arg_2, ...) {
   function_body
   return_value
}
```

Example (A bespoke function)

```
# c_surface
# Calculates the surface of a circle
# Arguments:
#   radius -- numeric, the radius of the circle
# Returns
#   the surface of the cicle
c_surface <- function(radius) {
  x <- radius ^ 2 * pi
  return (x)
  }

# Test the function:
c_surface(2)
## [1] 12.56637
```

Most probably you will work in a modern environment such as the IDE RStudio, which makes editing a text-file with code and running that code a breeze. However, there might be cases where one has only terminal access to R. In that case, the following functions might come in handy.

```
# Edit the function with vi:
fix(c_surface)

# Or us edit:
c_surface <- edit()
```

Function with a default argument

Example

The function `paste()` collates the arguments provided and returns one string that is a concatenation of all strings supplied, separated by a separator. This separator is supplied in the function via the argument `sep`. What is the default separator used in `paste()`?

Creating functions with a default value

Example (default value for function)

```
c_surface <- function(radius = 2) {
  radius ^ 2 * pi
  }
c_surface(1)
## [1] 3.141593

c_surface()
## [1] 12.56637
```

© Dr. Philippe J.S. De Brouwer

PART 02: STARTING WITH R AND ELEMENTS OF STATISTICS

CHAPTER 4: THE BASICS OF R

SECTION 7:

Packages

© DR. PHILIPPE J.S. DE BROUWER

Additional functions come in "packages." To use them one needs to install the package first with the function `install.packages()`; this will connect to a server, download the functions and prepare them for use. Once installed on our computer, they can be loaded in the active environment with the function `library()` or `require()`.

Example (loading the package DiagrammeR)

```
# Download the package (only once):
install.packages('DiagrammeR')

# Load it before we can use it (once per session):
library(DiagrammeR)
```

Examples of packages i

- **To load data**
  - `RODBC, RMySQL, RPostgresSQL, RSQLite`: read data from a database
  - `XLConnect, xlsx`: read and write Micorsoft Excel files (of course You can also just export your spreadsheets from Excel as csv-files)
  - `foreign`: to use eg SAS data
  - `Note`: among R's standard functionality is handling text files. Use the functions read.table or its more specific siblings such as read.csv() to read a CSV file and read.fwf() to read a fixed with table.
- **To manipulate data**
  - `dplyr`: creating subsets, summarizing, rearranging, and joining data sets
  - `tidyr`: changing the layout of your data sets
  - `stringr`: tools for regular expressions and character strings
  - `lubridate`: tools to facilitate working with dates and times
  - `reshape`: tools to present data differently (melt and cast)
- **To visualize data**
  - `ggplot2`: allows professional graphics (and has many extensions)
  - `ggvis`: to build interactive, web based graphics
  - `rgl`: Interactive 3D visualizations with R
  - `htmlwidgets`: build interactive (javascript based) visualizations. Packages that implement htmlwidgets include: leaflet (maps), dygraphs (time series), DT (tables), diagrammeR (diagrams), network3D (network graphs), threeJS (3D scatterplots and globes).
  - `googleVis`: use Google Chart tools to visualize data in R.

© Dr. Philippe J.S. De Brouwer

Examples of packages ii

- **To model data**
  - `car`: car's Anova function is popular for making type II and type III Anova tables
  - `mgcv`: Generalized Additive Models
  - `lme4/nlme`: Linear and Non-linear mixed effects models
  - `randomForest`: random forest methods from machine learning
  - `multcomp`: tools for multiple comparison testing
  - `vcd`: visualization tools and tests for categorical data
  - `glmnet`:Lasso and elastic-net regression methods with cross validation
  - `survival`: tools for survival analysis
  - `caret`: tools for training regression and classification models
- **To report results**
  - `shiny`: make interactive web-apps (e.g. explore data and share findings with non-programmers)
  - `R Markdown`: write R code in markdown report (when run render is run, R Markdown will replace the code with its results and then export your report as an HTML, pdf, or MS Word document, or a HTML or pdf slideshow. Hence, allows automated reporting. R Markdown is integrated into RStudio.
  - `knitr`: the same tool but for use in LaTeX (and can be used for other markup languages)
  - `xtable`: coverts R objects (such as data frames) and returns the latex or HTML code
- **For Spatial data**
  - `sp, maptools`: tools for loading and using spatial data including shapefiles.
  - `maps`: use map polygons for plots.
  - `ggmap`: use street maps from Google maps as a background in ggplots

Examples of packages iii

- **For Time Series and Financial data**
  - `zoo`: provides a format for saving time series objects
  - `xts`: tools for manipulating time series data sets
  - `quantmod`: tools for downloading financial data, plotting common charts, and doing technical analysis
- **To write high performance R code**
  - `Rcpp`: use C++ code from within R functions for fast speed
  - `data.table`: an alternative way to organize data sets for faster operations. Useful for big data.
  - `parallel`: parallel processing in R
- **To work with the web**
  - `XML`: read and create XML documents with R
  - `jsonlite`: read and create JSON data tables with R
  - `httr`: tools for working with http connections
- **To write your own R packages**
  - `devtools`: tools for turning your code into an R package
  - `testthat`: provides an easy way to write tests for your code
  - `roxygen2`: (like Oxygen for C++) turns inline code comments into documentation pages and builds a package namespace.

Useful functions for packages

Below we show some useful functions related to working with packages. Note that the output is supressed in this code. The reason is that it would be too lenghty and have limited relevance for the reader: best is to try it yourself.

```
# See the path where libraries are stored:
.libPaths()

# See the list of installed packages:
library()

# See the list of currently loaded packages:
search()
```

part 02: Starting with R and Elements of Statistics

chapter 4: The Basics of R

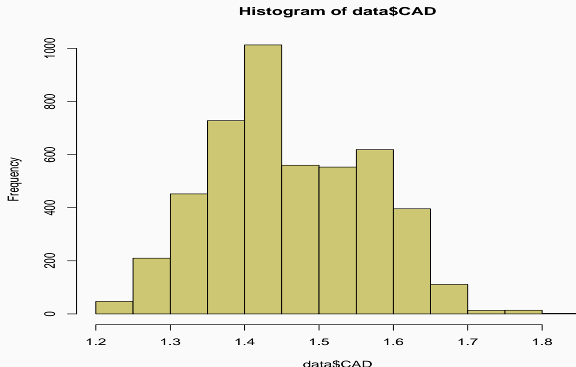
section 8:

Selected Data Interfaces

© Dr. Philippe J.S. De Brouwer

Import a CSV file

```
# To read a CSV-file it needs to be in the current directory
# or we need to supply the full path, or go first to the relevant folder.
setwd("./../../data/") # change working directory
data <- read.csv("eurofxref-hist.csv")
is.data.frame(data)
ncol(data)
nrow(data)
head(data)
hist(data$CAD, col = 'khaki3')
```



© Dr. Philippe J.S. De Brouwer

Finding data i

```
# get the maximum exchange rate
maxCAD <- max(data$CAD)
# use SQL-like selection
d0 <- subset(data, CAD == maxCAD)
d1 <- subset(data, CAD > maxCAD - 0.1)

d1[,1]
##  [1] 2008-12-30 2008-12-29 2008-12-18 1999-02-03 1999-01-29 1999-01-28
##  [7] 1999-01-27 1999-01-26 1999-01-25 1999-01-22 1999-01-21 1999-01-20
## [13] 1999-01-19 1999-01-18 1999-01-15 1999-01-14 1999-01-13 1999-01-12
## [19] 1999-01-11 1999-01-08 1999-01-07 1999-01-06 1999-01-05 1999-01-04
## 4718 Levels: 1999-01-04 1999-01-05 1999-01-06 1999-01-07 ... 2017-06-05
```

Finding data ii

```
d2 <- data.frame(d1$Date,d1$CAD)
d2
##       d1.Date d1.CAD
## 1  2008-12-30 1.7331
## 2  2008-12-29 1.7408
## 3  2008-12-18 1.7433
## 4  1999-02-03 1.7151
## 5  1999-01-29 1.7260
## 6  1999-01-28 1.7374
## 7  1999-01-27 1.7526
## 8  1999-01-26 1.7609
## 9  1999-01-25 1.7620
## 10 1999-01-22 1.7515
## 11 1999-01-21 1.7529
## 12 1999-01-20 1.7626
## 13 1999-01-19 1.7739
## 14 1999-01-18 1.7717
## 15 1999-01-15 1.7797
## 16 1999-01-14 1.7707
## 17 1999-01-13 1.8123
## 18 1999-01-12 1.7392
## 19 1999-01-11 1.7463
## 20 1999-01-08 1.7643
## 21 1999-01-07 1.7602
## 22 1999-01-06 1.7711
## 23 1999-01-05 1.7965
## 24 1999-01-04 1.8004
```

Writing to a CSV file i

```
write.csv(d2,"output.csv", row.names = FALSE)
new.d2 <- read.csv("output.csv")
print(new.d2)
##       d1.Date d1.CAD
## 1  2008-12-30 1.7331
## 2  2008-12-29 1.7408
## 3  2008-12-18 1.7433
## 4  1999-02-03 1.7151
## 5  1999-01-29 1.7260
## 6  1999-01-28 1.7374
## 7  1999-01-27 1.7526
## 8  1999-01-26 1.7609
## 9  1999-01-25 1.7620
## 10 1999-01-22 1.7515
## 11 1999-01-21 1.7529
## 12 1999-01-20 1.7626
## 13 1999-01-19 1.7739
## 14 1999-01-18 1.7717
## 15 1999-01-15 1.7797
## 16 1999-01-14 1.7707
## 17 1999-01-13 1.8123
## 18 1999-01-12 1.7392
## 19 1999-01-11 1.7463
## 20 1999-01-08 1.7643
## 21 1999-01-07 1.7602
## 22 1999-01-06 1.7711
## 23 1999-01-05 1.7965
## 24 1999-01-04 1.8004
```

© Dr. Philippe J.S. De Brouwer

Import of Excel files

```
# install the package xlsx if not yet done
if (!any(grepl("xlsx",installed.packages()))){
  install.packages("xlsx")}
library(xlsx)
data <- read.xlsx("input.xlsx", sheetIndex = 1)
```

R can connect to many popular database systems. For example, MySQL: as usual there is a package that will provide this functionality.

```
if(!any(grepl("xls", installed.packages()))){
  install.packages("RMySQL")}
library(RMySQL)
```

Connecting to the Database

Note: more details in the part dealing with databases.

```
# The connection is stored in an R object, myConnection, and
# it needs the database name (db_name), username and password
myConnection = dbConnect(MySQL(),
   user     = 'root',
   password = 'xxx',
   dbname   = 'db_name',
   host     = 'localhost')

# e.g. list the tables available in this database.
dbListTables(myConnection)
```

© Dr. Philippe J.S. De Brouwer

Fetching Data Drom a Database

```
# Prepare the query for the database
result <- dbSendQuery(myConnection,
  "SELECT * from tbl_students WHERE age > 33")

# fetch() will get us the results, it takes a parameter n, which
# is the number of desired records.
# Fetch all the records(with n = -1) and store it in a data frame.
data <- fetch(result, n = -1)
```

Update Queries

The `dbSendQuery()` function can be used to send any query, including UPDATE, INSERT, CREATE TABLE and DROP TABLE queries so we can push results back to the database.

```
sSQL = ""
sSQL[1] <- "UPDATE tbl_students
            SET score = 'A' WHERE raw_score > 90;"
sSQL[2] <- "INSERT INTO tbl_students
            (name, class, score, raw_score)
    VALUES ('Robert', 'Grade 0', 88,NULL);"
sSQL[3] <- "DROP TABLE IF EXISTS tbl_students;"
for (k in c(1:3)){
  dbSendQuery(myConnection, sSQL[k])
  }
```

Create Tables from R Data-frames

It is most useful to be able to write back complete data-frames to a data-base.

```
dbWriteTable(myConnection, "tbl_name",
             data_frame_name[, ], overwrite = TRUE)
```

part 02: Starting with R and Elements of Statistics

chapter 5:

Lexical Scoping and Environments

© Dr. Philippe J.S. De Brouwer

part 02: Starting with R and Elements of Statistics

chapter 5: Lexical Scoping and Environments

section 1:

Environments in R

© Dr. Philippe J.S. De Brouwer

The Global Environment i

The top-level environment is the R command prompt. This is the “global environment” and known as `R_GlobalEnv` and can be accessed as `.GlobalEnv`.

```
environment()  # get the environment
## <environment: R_GlobalEnv>

rm(list = ls())  # clear the environment
ls()           # list all objects
## character(0)

a <- "a"
f <- function (x) print(x)
ls()           # note that x is not part of.GlobalEnv
## [1] "a" "f"
```

© Dr. Philippe J.S. De Brouwer

part 02: Starting with R and Elements of Statistics

chapter 5: Lexical Scoping and Environments

section 2:

Lexical Scoping in R

R has dynamic scoping

```
# f
# Demonstrates the scope of variables
f <- function() {
  a <- pi     # define local variable
  print(a)    # print the local variable
  print(b)    # b is not in the scope of the function
}

# Define two variables a and b
a <- 1
b <- 2

# Run the function and note that it knows both a and b.
# For b it cannot find a local definition and hence
# uses the definition of the higher level of scope.
f()
## [1] 3.141593
## [1] 2

# f() did not change the value of a in the environment that called f():
print(a)
## [1] 1
```

The variable `b` is not defined within the function, but the function can access it. This means that at the moment we use `b` in the function, R will first try to find it in the environment of the function. Since this fails, R will check the superior environment. R will repeat this process till it finds the variable. Of course, once it cannot find the

part 02: Starting with R and Elements of Statistics

chapter 6:

The Implementation of OO

© Dr. Philippe J.S. De Brouwer

part 02: Starting with R and Elements of Statistics

chapter 6: The Implementation of OO

section 1:

Base Types

© Dr. Philippe J.S. De Brouwer

The basic object provided by R are called “base types”. Examples are:

- character,
- double,
- complex,
- vectors,
- lists,
- arrays,
- matrices,
- functions,
- etc.

These are objects that can readily be used when R is started.

part 02: Starting with R and Elements of Statistics

chapter 6: The Implementation of OO

section 2:

S3 Objects

Many of the most simple objects in R are S3 objects. Examples are:

- data frames,
- models,
- time series,
- predictions about models,
- etc.

S3 ojbects have the class-attribute set so that dispatcher functions can identify the appropriate method that should be executed. For example a data frame has the class attribute set to `data.frame` and hence the method `plot()` will dispatch execution to `plot.data.frame()`.
All we need to do to create an S3 object is to set its class attribute:

© Dr. Philippe J.S. De Brouwer

S3 Objects ii

```r
# Define a function to check if something is S3:
is.S3 <- function(x){is.object(x) & !isS4(x)}

# A string is a base type and not an non-S3 ojbect:
x <- 'x'
is.S3(x)
## [1] FALSE

# A string with a class attribute is a valid S3 object:
class(x) <- 'myclass'
is.S3(x)
## [1] TRUE
```

© Dr. Philippe J.S. De Brouwer

Even for this simple example, it is possible to build specific methods.

```
# Define the method print for the class myclass:
print.myclass <- function(x) {print(paste0('Hello, ', x, '.'))}

# Test it, first with the existing myclass object:
print(x)
## [1] "Hello, x."

# Test it on a character base type (string):
print('x')
## [1] "x"
```

Above, we created a function `is.S3()` to test if an something is an S3 object. It is not really necessary to create such function by ourselves. We can leverage the library `pryr`, which provides a function `otype()` that returns the type of object.

```
library(pryr)
otype(M)
## [1] "base"

otype(df)
## [1] "S3"

otype(df$X1)            # a vector is not S3
## [1] "base"

df$fac <-factor(df$X4)
otype(df$fac)           # a factor is S3
## [1] "S3"
```

© Dr. Philippe J.S. De Brouwer

part 02: Starting with R and Elements of Statistics

chapter 6: The Implementation of OO

section 3:

S4 Objects

S4 Objects i

The S4 system is very similar to the S3 system, but it adds a certain obligatory formalism. For example, it is necessary to define the class before using it. This adds some lines of code but the payoff is increased clarity.

In S4

1. classes have formal definitions that describe their data fields and inheritance structures (parent classes);
2. method dispatch is more flexible and can be based on multiple arguments to a generic function, not just one; and
3. there is a special operator, `@`, for extracting fields from an S4 object.

All the S4 related code is stored in the `methods` package.

Creating S4 class definitions

S4 objects are created with the function `setClass()`.

```
# Create the object type Acc to hold bank-accounts:
setClass("Acc",
  representation(holder       = "character",
                 branch       = "character",
                 opening_date = "Date"))

# Create the object type Bnk (bank):
setClass("Bnk",
  representation(name = "character", phone = "numeric"))

# Define current account as a child of Acc:
setClass("CurrAcc",
  representation(interest_rate = "numeric",
                 balance  = "numeric"),
  contains = "Acc")

# Define investment account as a child of Acc
setClass("InvAcc",
  representation(custodian = "Bnk"), contains = "Acc")
```

At this point, the *classes* `Bnk` and `Acc` exist and we can create a first instance for both.

```
# Create an instance of Bnk:
my_cust_bank <- new("Bnk",
                    name = "HSBC",
                    phone = 123456789)

# Create an instance of Acc:
my_acc <- new("Acc",
              holder       = "Philippe",
              branch       = "BXL12",
              opening_date = as.Date("2018-10-02"))
```

Now, we have two S4 objects and we can use them in our code as necessary. For example, we can change the phone number.

```
# Check if it is really an S4 object:
isS4(my_cust_bank)
## [1] TRUE

# Change the phone number and check:
my_cust_bank@phone = 987654321  # change the phone number
print(my_cust_bank@phone)       # check if it changed
## [1] 987654321
```

Note – Compare addressing slots in S4 and S3

In order to access slots of an S4 object, we use `@`, not `$`:

© Dr. Philippe J.S. De Brouwer

```
my_curr_acc <- new("CurrAcc",
                   holder        = "Philippe",
                   interest_rate = 0.01,
                   balance       = 0,
                   branch        = "LDN12",
                   opening_date  = as.Date("2018-12-01"))

# Note that the following does not work and is bound to fail:
also_an_account <- new("CurrAcc",
                        holder        = "Philippe",
                        interest_rate = 0.01,
                        balance       = 0,
                        Acc           = my_acc)

## Error in initialize(value, ...):  invalid name for slot of class "CurrAcc":  Acc
```

The object `my_curr_acc` is now ready to be used. For example, we can change the balance.

```
my_curr_acc@balance <- 500
```

© Dr. Philippe J.S. De Brouwer

Validation of input for S4 objects

```
# Note the mistake in the following code:
my_curr_acc <- new("CurrAcc",
                   holder        = "Philippe",
                   interest_rate = 0.01,
                   balance       = "0",  # Here is the mistake!
                   branch        = "LDN12",
                   opening_date  = as.Date("2018-12-01"))

## Error in validObject(.Object):  invalid class "CurrAcc" object:  invalid object for slot "balance" in class "CurrAcc":
got class "character", should be or extend class "numeric"
```

If you omit a slot, R coerces that slot to the default value.

```
x_account <- new("CurrAcc",
                   holder        = "Philippe",
                   interest_rate = 0.01,
                   #no balance provided
                   branch        = "LDN12",
                   opening_date  = as.Date("2018-12-01"))

# Show what R did with it:
x_account@balance
## numeric(0)
```

© Dr. Philippe J.S. De Brouwer

Creating S4 Generics

```
# setGeneric needs a function, so we need to create it first.

# credit
# Dispatcher function to credit the ledger of an object of
# type 'account'.
# Arguments:
#    x -- account object
#    y -- numeric -- the amount to be credited
credit <- function(x,y){useMethod()}

# transform our function credit() to a generic one:
setGeneric("credit")
## [1] "credit"

# Add the credit function to the object CurrAcc
setMethod("credit",
   c("CurrAcc"),
   function (x, y) {
     new_bal <- x@balance + y
     new_bal
     }
   )

# Test the function:
my_curr_acc@balance
## [1] 500

my_curr_acc@balance <- credit(my_curr_acc, 100)
```

PART 02: STARTING WITH R AND ELEMENTS OF STATISTICS

CHAPTER 6: THE IMPLEMENTATION OF OO

SECTION 4:

The Reference Class, refclass, RC or R5 Model

Create an RC object

```r
# Definition of RC object currentAccount
currAccount <- setRefClass("currentAccount",
                  fields = list(interest_rate = "numeric",
                                balance       = "numeric"),
                  contains = c("account"),
                  methods = list(
                     credit = function(amnt) {
                           balance <<- balance + amnt
                           },
                     debet =  function(amnt) {
                           if (amnt <= balance) {
                              balance <<- balance - amnt
                              } else {
                              stop("Not rich enough!")
                              }
                        }
                     )
                  )
# note how the class reports on itself:
currAccount
## Generator for class "currentAccount":
##
## Class fields:
##
## Name:     ref_number        holder        branch  opening_date
## Class:       numeric     character     character          Date
##
## Name:   account_type interest_rate       balance
## Class:     character       numeric       numeric
```

© Dr. Philippe J.S. De Brouwer

We can now create accounts and use the methods supplied.

```
ph_acc <- currAccount$new(ref_number    = 321654987,
                          holder        = "Philippe",
                          branch        = "LDN05",
                          opening_date  = as.Date(Sys.Date()),
                          account_type  = "current",
                          interest_rate = 0.01,
                          balance       = 0
                          )
```

Now, we can start using the money and withdrawing money.

```
ph_acc$balance     # after creating balance is 0:
## [1] 0

ph_acc$debet(200)  # impossible (not enough balance)

## Error in ph_acc$debet(200):  Not rich enough!

ph_acc$credit(200) # add 200 to the acount
ph_acc$balance     # the money arrived in our account
## [1] 200

ph_acc$debet(100)  # this is possible
ph_acc$balance     # the money is indeed gone
## [1] 100
```

© Dr. Philippe J.S. De Brouwer

PART 02: STARTING WITH R AND ELEMENTS OF STATISTICS

CHAPTER 6: THE IMPLEMENTATION OF OO

SECTION 5:

Conclusions about the OO Implementation

Practical summary of the OO system in R

The OO system that R provides is unlike what other OO languages provide. In the first place, it offers not only a method-dispatching system but also has a message-passing system. Secondly, it is of great importance that it is possible to use R without even knowing that the OO system exists. In fact, for most of the following chapters in this book, it is enough to know that the generic-function implementation of the OO logic exists.

part 02: Starting with R and Elements of Statistics

chapter 7:

Tidy R with the Tidyverse

part 02: Starting with R and Elements of Statistics

chapter 7: Tidy R with the Tidyverse

section 1:

The Philosophy of the Tidyverse

© Dr. Philippe J.S. De Brouwer

The philosophy endoresed by the creators of the tidyverse

The developers of `tidyverse` promote[1]:

- Use existing and common data structures. So all the packages in the tidyverse will share a common S3 class types; this means that in general functions will accept data frames (or `tibbles`). More low-level functions will work with the base R vector types.
- Reuse data structures in your code. The idea here is that there is a better option than always over-writing a variable or create a new one in every line: pass on the output of one line to the next with a "pipe": `%>%`. To be accepted in the tidyverse, the functions in a package need to be able to use this pipe.[2]
- Keep functions concise and clear. For example, do not mix side-effects and transformations, function names should be verbs where ever possible (unless they become too generic or meaningless of course), and keep functions short (they do only one thing, but do it well).
- Embrace R as a functional programming language. This means that reflexes that you might have from say C++, C#, python, PHP, etc., will have to be mended. This means for example that it is best to use immutable objects and copy-on-modify semantics and avoid using the refclass model.
  Use where possible the generic functions provided by S3 and S4. Avoid writing loops (such as `repeat` and `for` but use the apply family of functions (or refer to the package `purrr`).
- Keep code clean and readable for humans. For example, prefer meaningful but long variable names over short but meaningless ones, be considerate towards people using auto-complete in RStudio (so add an id in the first and not last letters of a function name), etc.

part 02: Starting with R and Elements of Statistics

chapter 7: Tidy R with the Tidyverse

section 2:

Packages in the Tidyverse

© Dr. Philippe J.S. De Brouwer

The Core Tidyverse

Loading the tidyverse will report on which packages are included:

```
# we assume that you installed the package before:
# install.packages("tidyverse")
# so load it:
library(tidyverse)
```

```
## - Attaching packages ------------------------- tidyverse 1.3.1 -

## v ggplot2 3.3.3     v purrr   0.3.4
## v tibble  3.1.1     v dplyr   1.0.5
## v tidyr   1.1.3     v stringr 1.4.0
## v readr   1.4.0     v forcats 0.5.1

## - Conflicts ---------------------------- tidyverse_conflicts() -
## x purrr::compose() masks pryr::compose()
## x dplyr::filter()  masks stats::filter()
## x dplyr::lag()     masks stats::lag()
## x purrr::partial() masks pryr::partial()
```

So, loading the library `tidyverse`, loads actually a series of other packages. The collection of these packages are called “core-tidyverse.”

- `tidyr` provides a set of functions that help you get to tidy up data and make adhering to the rules of tidy data easier.
  The idea of tidy data is really simple: it is data where every variable has its own column, and every column is a variable.
- `dplyr` provides a grammar of data manipulation, providing a consistent set of verbs that solve the most common data manipulation challenges.
- `ggplot2` is a system to create graphics with a philosophy: it adheres to a "Grammar of Graphics" and is able to create really stunning results at a reasonable price (it is a notch more abstract to use than the core-R functionality).
- `readr` expands R's standard[3] functionality to read in rectangular[4] data.
  It is more robust, knows more data types and is faster than the core-R functionality.
- `purrr` is mentioned in the section about the OO model in R
  It is a rather complete and consistent set of tools for working with functions and vectors. Using `purrr` it should be possible to replace most loops with call to `purr` functions that will work faster.
- `tibble` is a new take on the data frame of core-R. It provides a new base type: `tibbles`.
  Tibbles are in essence data frames, that do a little less (so there is less clutter on the screen and less unexpected things happen), but rather give more feedback (show what went wrong instead of assuming that you have read all manuals and remember everything).

© Dr. Philippe J.S. De Brouwer

- `stringr` expands the standard functions to work with strings and provides a nice coherent set of functions that all start with `str_`.
  The package is built on top of `stringi`, which uses the ICU library that is written in C, so it is fast too.
- `forcats` provides tools to address common problems when working with categorical variables[5].

part 02: Starting with R and Elements of Statistics

chapter 7: Tidy R with the Tidyverse

section 3:

Working with the Tidyverse

Creating a tibble

```
library(tidyverse)
x <- seq(from = 0, to = 2 * pi, length.out = 100)
tb <- tibble(x, sin(x), cos(x), cos(x) + sin(x))
```

Using a tibble i

```
# Note how concise and relevant the output is:
print(tb)
## # A tibble: 100 x 4
##         x `sin(x)` `cos(x)` `cos(x) + sin(x)`
##     <dbl>    <dbl>    <dbl>             <dbl>
##  1 0        0         1                  1
##  2 0.0635   0.0634    0.998              1.06
##  3 0.127    0.127     0.992              1.12
##  4 0.190    0.189     0.982              1.17
##  5 0.254    0.251     0.968              1.22
##  6 0.317    0.312     0.950              1.26
##  7 0.381    0.372     0.928              1.30
##  8 0.444    0.430     0.903              1.33
##  9 0.508    0.486     0.874              1.36
## 10 0.571    0.541     0.841              1.38
## # ... with 90 more rows

# This does the same as for a data-frame:
plot(tb)
```

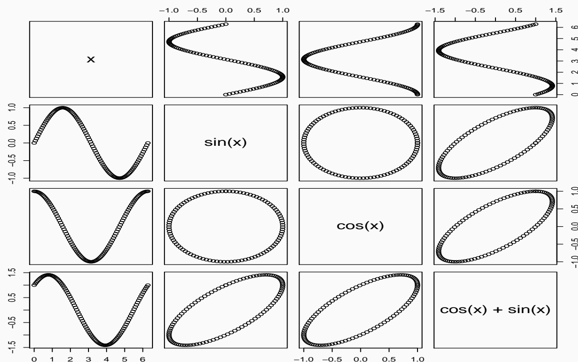


Figure 6: A tibble plots itself like a data-frame.

```
# Actually a tibble will still behave as a data frame:
is.data.frame(tb)
## [1] TRUE
```

To start, consider a simple example:

```
t <- tibble("x" = runif(10))
t <- within(t, y <- 2 * x + 4 + rnorm(10, mean = 0, sd = 0.5))
```

This can also be written with the piping operator from `magrittr`:

```
t <- tibble("x" = runif(10))  %>%
     within(y <- 2 * x + 4 + rnorm(10, mean = 0,sd = 0.5))
```

What R does behind the scenes, is feeding the output left of the pipe operator as main input right of the pipe operator. This means that the following are equivalent:

```
# 1. pipe:
a %>% f()
# 2. pipe with shortened function:
a %>% f
# 3. is equivalent with:
f(a)
```

© Dr. Philippe J.S. De Brouwer

The dollar pipe

```r
# The Tidyverse only makes the %>% pipe available. So, to use the
# special pipes, we need to load magrittr
library(magrittr)
lm2 <- tibble("x" = runif(10))                                    %>%
       within(y <- 2 * x + 4 + rnorm(10, mean=0,sd=0.5)) %$%
       lm(y ~ x)
summary(lm2)
##
## Call:
## lm(formula = y ~ x)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -0.6101 -0.3534 -0.1390  0.2685  0.8798
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)   4.0770     0.3109  13.115 1.09e-06 ***
## x             2.2068     0.5308   4.158  0.00317 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.5171 on 8 degrees of freedom
## Multiple R-squared:  0.6836,Adjusted R-squared:  0.6441
## F-statistic: 17.29 on 1 and 8 DF,  p-value: 0.003174
```

© Dr. Philippe J.S. De Brouwer

The T-pipe passes the previous output on i

```
library(magrittr)
t <- tibble("x" = runif(100))                              %>%
     within(y <- 2 * x + 4 + rnorm(10, mean=0, sd=0.5)) %T>%
     plot(col="red")   # The function plot does not return anything
                       # so we used the %T>% pipe. Hence the result of
                       # within() is passed to t.

lm3 <-   t                  %$%
         lm(y ~ x)          %T>% # pass on the linear model for assignment
         summary            %T>% # further pass on the linear model
         coefficients

tcoef <- lm3 %>% coefficients  # we anyhow need the coefficients

# Add the model (the solid line) to the previous plot:
abline(a = tcoef[1], b=tcoef[2], col="blue", lwd = 3)
```

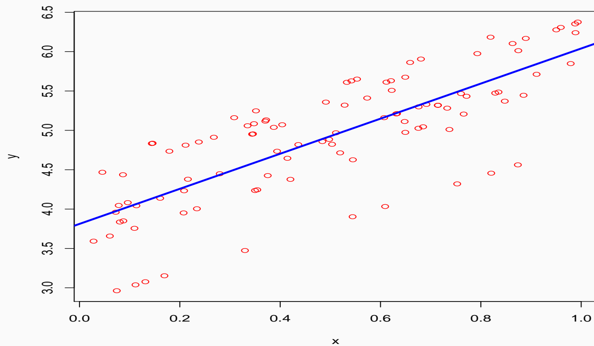


Figure 7: A linear model fit on generated data to illustrate the piping command.

© Dr. Philippe J.S. De Brouwer

part 02: Starting with R and Elements of Statistics

chapter 8:

Elements of Descriptive Statistics

© Dr. Philippe J.S. De Brouwer

PART 02: STARTING WITH R AND ELEMENTS OF STATISTICS

CHAPTER 8: ELEMENTS OF DESCRIPTIVE STATISTICS

SECTION 1:

Measures of Central Tendency

© DR. PHILIPPE J.S. DE BROUWER

Definition 2 (Arithmetic mean)

$$\bar{x} = \sum_{n=1}^{N} P(x).x \qquad \text{(for discrete distributions)}$$

$$= \int_{-\infty}^{+\infty} x.f(x)\,\mathrm{d}x \qquad \text{(for continuous distributions)}$$

The unbiased estimator of the mean for K observations x_k is:

$$E[\bar{x}] = \frac{1}{K}\sum_{k=1}^{K} x_k$$

© Dr. Philippe J.S. De Brouwer

The Mean in R

```
# The mean of a vector:
x <- c(1,2,3,4,5,60)
mean(x)
## [1] 12.5

# Missing values will block the override the result:
x <- c(1,2,3,4,5,60,NA)
mean(x)
## [1] NA

# Missing values can be ignored with na.rm = TRUE:
mean(x, na.rm = TRUE)
## [1] 12.5

# This works also for a matrix:
M <- matrix(c(1,2,3,4,5,60), nrow=3)
mean(M)
## [1] 12.5
```

Definition 3 (f-mean)

$$\bar{x} = f^{-1}\left(\frac{1}{n}.\sum_{k=1}^{K} f(x_k)\right)$$

Popular choices for `f()` are:

- $f(x) = x$: arithmetic mean,
- $f(x) = \frac{1}{x}$: harmonic mean,
- $f(x) = x^m$: power mean,
- $f(x) = \ln x$: geometric mean, so $\bar{x} = \left(\prod_{k=1}^{K} x_k\right)^{\frac{1}{K}}$

© Dr. Philippe J.S. De Brouwer

One particular generalized mean is the power mean or Hölder mean. It is defined for a set of K positive numbers x_k by

$$\bar{x}(m) = \left(\frac{1}{n} \cdot \sum_{k=1}^{K} x_k^m \right)^{\frac{1}{m}}$$

by choosing particular values for m one can get the quadratic, arithmetic, geometric and harmonic means.

- $m \to \infty$: maximum of x_k
- $m = 2$: quadratic mean
- $m = 1$: arithmetic mean
- $m \to 0$: geometric mean
- $m = 1$: harmonic mean
- $m \to -\infty$: minimum of x_k

© Dr. Philippe J.S. De Brouwer

The median is the middle-value so that 50% of the observations are lower and 50% are higher.

```
x <- c(1:5,5e10,NA)
x
## [1] 1e+00 2e+00 3e+00 4e+00 5e+00 5e+10    NA

median(x)               # no meaningful result with NAs
## [1] NA

median(x,na.rm = TRUE)  # ignore the NA
## [1] 3.5

# Note how the median is not impacted by the outlier,
# but the outlier dominates the mean:
mean(x, na.rm = TRUE)
## [1] 8333333336
```

© Dr. Philippe J.S. De Brouwer

The mode

The mode is the value that has highest probability to occur. For a series of observations, this should be the one that occurs most often. Note that the mode is also defined for variables that have no order-relation (even labels such as “green,” “yellow,” etc. have a mode, but not a mean or median — without further abstraction or a numerical representation).[6]

In R, the function `mode()` or `storage.mode()` returns a character string describing how a variable is stored. In fact, R does not have a standard function to calculate mode, so let us create our own:

```
# my_mode
# Finds the first mode (only one)
# Arguments:
#   v -- numeric vector or factor
# Returns:
#   the first mode
my_mode <- function(v) {
   uniqv <- unique(v)
   tabv  <- tabulate(match(v, uniqv))
   uniqv[which.max(tabv)]
   }

# now test this function
x <- c(1,2,3,3,4,5,60,NA)
my_mode(x)
## [1] 3

x1 <- c("relevant", "N/A", "undesired", "great", "N/A",
        "undesired", "great", "great")
```

PART 02: STARTING WITH R AND ELEMENTS OF STATISTICS

CHAPTER 8: ELEMENTS OF DESCRIPTIVE STATISTICS

SECTION 2:

Measures of Variation or Spread

© DR. PHILIPPE J.S. DE BROUWER

Standard Deviation

Definition 4 (Standard deviation)

$$\mathrm{SD}(X) = \sqrt{\mathrm{VAR}(X)}$$

The estimator for standard deviation is:

$$\widehat{\mathrm{SD}(X)} = \sqrt{\frac{1}{N-1}\sum_{n=1}^{N}(X_n - \bar{X})^2}$$

```
t <- rnorm(100, mean=0, sd=20)
var(t)
## [1] 248.2647

sd(t)
## [1] 15.75642

sqrt(var(t))
## [1] 15.75642

sqrt(sum((t - mean(t))^2)/(length(t) - 1))
## [1] 15.75642
```

Median absolute deviation

Definition 5 (mad)

$$\text{mad}(X) = \frac{1}{1.4826}\text{median}\left(|X - \text{median}(X)|\right)$$

```
mad(t)
## [1] 14.54922

mad(t,constant=1)
## [1] 9.813314
```

PART 02: STARTING WITH R AND ELEMENTS OF STATISTICS

CHAPTER 8: ELEMENTS OF DESCRIPTIVE STATISTICS

SECTION 3:

Measures of Covariation

© DR. PHILIPPE J.S. DE BROUWER

Definition 6 (Pearson Correlation Coefficient)

$$\begin{aligned}\rho_{XY} &= \frac{\mathrm{covar}(X,Y)}{\sigma_X \sigma_Y} \\ &= \frac{(X - E[X])(Y - E[Y])}{\sqrt{(X - E[X])(Y - E[Y])}} \\ &=: \mathrm{covar}(x,y)\end{aligned}$$

```
cor(mtcars$hp,mtcars$wt)
## [1] 0.6587479
```

© Dr. Philippe J.S. De Brouwer

The Spearman correlation

The Spearman correlation is the correlation applied to the ranks of the data. It is one if an increase in the variable X is always accompanied with an increase in variable Y.

```
cor(rank(df$x), rank(df$x_sq))
## [1] 0

cor(rank(df$x), rank(df$x_abs))
## [1] 0

cor(rank(df$x), rank(df$x_exp))
## [1] 1
```

The Spearman correlation checks for a relationship that can be more general than only linear. It will be one if X increases when Y increases.

Question #6

Consider the vectors

1. $x = c(1, 2, 33, 44)$ and $y = c(22, 23, 100, 200)$,
2. $x = c(1 : 10)$ and $y = 2 * x$,
3. $x = c(1 : 10)$ and $y = exp(x)$,

Plot y in function of x. What is their Pearson correlation? What is their Spearman correlation? How do you understand that?

© Dr. Philippe J.S. De Brouwer

Function use for chisq.test()

```
chisq.test(data)
```

where `data` is the data in form of a table containing the count value of the variables

© Dr. Philippe J.S. De Brouwer

An example for chisq.test()

```
# we use the dataset mtcars from MASS
df <- data.frame(mtcars$cyl,mtcars$am)
chisq.test(df)
##
##  Pearson's Chi-squared test
##
## data:  df
## X-squared = 25.077, df = 31, p-value = 0.7643
```

The chi-square test reports a p-value. This p-value is the probability that the correlations is actually insignificant. It appears that in practice a correlation lower than 5% can be considered as insignificant. In this example, the p-value is higher than 0.05, so there is no significant correlation.

part 02: Starting with R and Elements of Statistics

chapter 8: Elements of Descriptive Statistics

section 4:

Distributions

© Dr. Philippe J.S. De Brouwer

Distribution functions in R

- `d:` The pdf (probability density function)
- `p:` The cdf (cumulative probability density function)
- `q:` The quantile function
- `r:` The random number generator.

© Dr. Philippe J.S. De Brouwer

The Normal Distribution in R

R has four built-in functions to work with the normal distribution. They are described below.

- `dnorm(x, mean, sd)`: The height of the probability distribution
- `pnorm(x, mean, sd)`: The cumulative distribution function (the probability of the observation to be lower than x)
- `qnorm(p, mean, sd)`: Gives a number whose cumulative value matches the given probability value p
- `rnorm(n, mean, sd)`: Generates normally distributed variables,

with

- x: A vector of numbers
- p: A vector of probabilities
- n: The number of observations(sample size)
- *mean*: The mean value of the sample data (default is zero)
- *sd*: The standard deviation (default is 1).

```
obs <- rnorm(600,10,3)
hist(obs,col="khaki3",freq=FALSE)
x <- seq(from=0,to=20,by=0.001)
lines(x, dnorm(x,10,3),col="blue",lwd = 4)
```

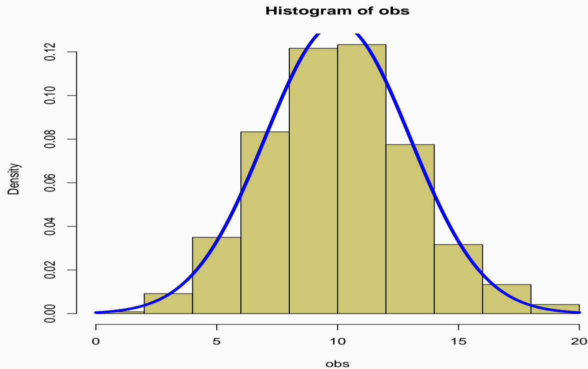


Figure 8: A comparison between a set of random numbers drawn from the normal distribution (khaki) and the theoretical shape of the normal distribution in blue.

In this simple illustration, we will compare the returns of the index S&P500 to the Normal distribution. The output of the code below is in Figure 9 on slide 171.

```
library(MASS)
hist(SP500,col="khaki3",freq=FALSE,border="khaki3")
x <- seq(from=-5,to=5,by=0.001)
lines(x, dnorm(x,mean(SP500),sd(SP500)),col="blue",lwd=2)
```

© Dr. Philippe J.S. De Brouwer

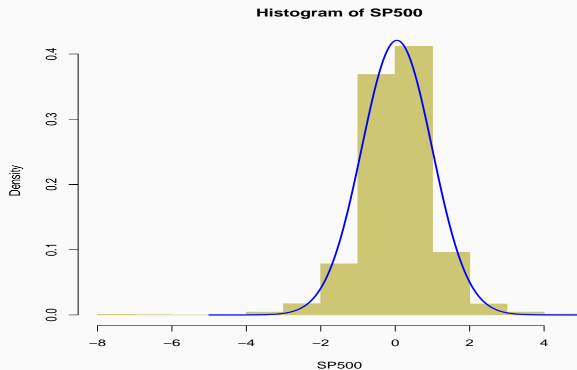


Figure 9: The same plot for the returns of the SP500 index seems acceptable, though there are outliers (where the normal distribution converges fast to zero).

© Dr. Philippe J.S. De Brouwer

```r
library(MASS)
qqnorm(SP500,col="red"); qqline(SP500,col="blue")
```

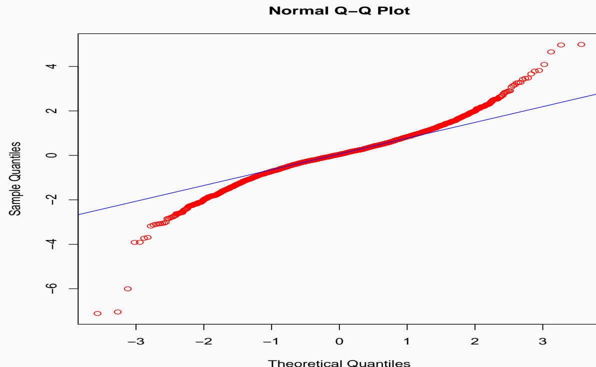


Figure 10: A Q-Q plot is a good way to judge if a set of observations is normally distributed or not.

The Binomial Distribution in R

As for all distributions, R has four in-built functions to generate binomial distribution:

- `dbinom(x, size, prob)`: The density function
- `pbinom(x, size, prob)`: The cumulative probability of an event
- `qbinom(p, size, prob)`: Gives a number whose cumulative value matches a given probability value
- `rbinom(n, size, prob)`: Generates random variables following the binomial distribution.

Following parameters are used:

- *x*: A vector of numbers
- *p*: A vector of probabilities
- *n*: The number of observations
- *size*: The number of trials
- *prob* The probability of success of each trial

The example below illustrates the biniomial distribution and generates the plot in Figure 11 on slide 175.

```
# Probability of getting 5 or less heads from 10 tosses of
# a coin.
pbinom(5,10,0.5)
## [1] 0.6230469

# visualize this for one to 10 numbers of tosses
x <- 1:10
y <- pbinom(x,10,0.5)
plot(x,y,type="b",col="blue", lwd = 3,
    xlab="Number of tails",
    ylab="prob of maxium x tails",
    main="Ten tosses of a coin")
```

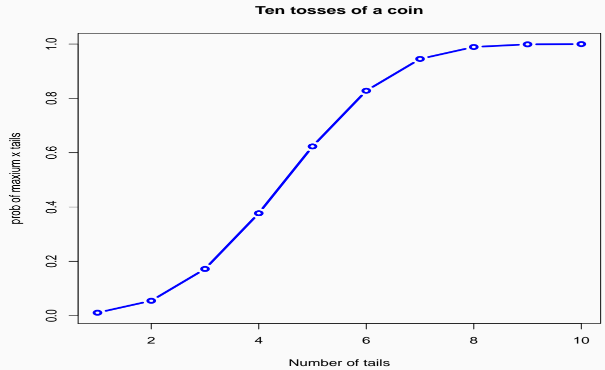


Figure 11: The probability to get maximum x tails when flipping a fair coin, illustrated with the binomial distribution.

```r
# How many heads should we at least expect (with a probability
# of 0.25) when a coin is tossed 10 times.
qbinom(0.25,10,1/2)
## [1] 4
```

Generate random variables

```
# Find 20 random numbers of tails from and event of 10 tosses
# of a coin
rbinom(20,10,.5)
##  [1] 5 7 2 6 7 4 6 7 3 2 5 9 5 9 5 5 5 5 5 6
```

PART 02: STARTING WITH R AND ELEMENTS OF STATISTICS

CHAPTER 8: ELEMENTS OF DESCRIPTIVE STATISTICS

SECTION 5:

Creating an Overview of Data Characteristics

© DR. PHILIPPE J.S. DE BROUWER

part 02: Starting with R and Elements of Statistics

chapter 9:

Visualisation Methods

© Dr. Philippe J.S. De Brouwer

part 02: Starting with R and Elements of Statistics

chapter 9: Visualisation Methods

section 1:

Scatterplots

Function use for plot() – for a scatterplot

`plot(x, y, main, xlab, ylab, xlim, ylim, axes, ...)` with

- `x:` the data set for the horizontal axis
- `y:` the data set for the vertical axis
- `main:` the tile of the graph
- `xlab:` the title of the *x*-axis
- `ylab:` the title of the *x*-axis
- `xlim:` the range of values on the *x*-axis
- `ylim:` the range of values on the *y*-axis
- `pch:` the display symbol
- `axes:` indicates whether both axes should be drawn on the plot.

© Dr. Philippe J.S. De Brouwer

Some pch arguments

Figure 12: Some plot characters. Most other characters will just plot themselves.

```r
# Import the data:
library(MASS)

# To make this example more interesting, we convert mpg to l/100km

# mpg2l
# Converts miles per gallon into litres per 100 km
# Arguments:
#    mpg -- numeric -- fuel consumption in MPG
# Returns:
#    Numeric -- fuel consumption in litres per 100 km
mpg2l <- function(mpg = 0) {
100 * 3.785411784 / 1.609344 / mpg}

mtcars$l <- mpg2l(mtcars$mpg)
plot(x = mtcars$hp,y = mtcars$l, xlab = "Horse Power",
 ylab = "L per 100km", main = "Horse Power vs Milage",
 pch = 22, col="red", bg="yellow")
```

Scatterplot example ii

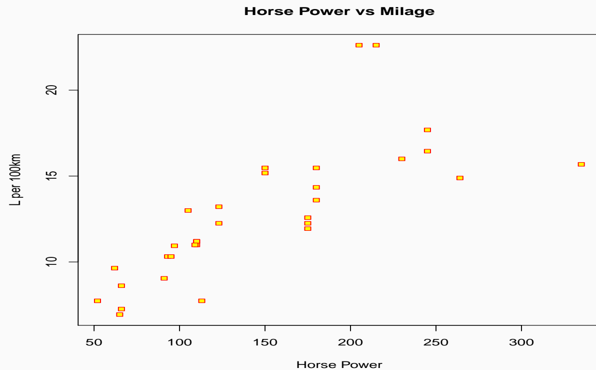


Figure 13: A scatter-plot needs an x and a y variable.

part 02: Starting with R and Elements of Statistics

chapter 9: Visualisation Methods

section 2:

Line Graphs

Function use for plot() – for line plots

`plot(x, type , main, xlab, ylab, xlim, ylim, axes, sub, asp ...)` with

- `x`: the data set for the horizontal axis
- `y`: the data set for the vertical axis (optional)
- `type`: indicates the type of plot to be made:
  - "p" for *p*oints,
  - "l" for *l*ines,
  - "b" for *b*oth,
  - "c" for the lines part alone of "b",
  - "o" for both '*o*verplotted',
  - "h" for '*h*istogram' like (or 'high-density') vertical lines,
  - "s" for stair *s*teps,
  - "S" for other *s*teps, see 'Details' in the documentation,
  - "n" for no plotting.
- `main`: the tile of the graph
- `xlab`: the title of the *x*-axis
- `ylab`: the title of the *x*-axis
- `xlim`: the range of values on the *x*-axis
- `ylim`: the range of values on the *y*-axis
- `axes`: indicates whether both axes should be drawn on the plot.
- `sub`: the sub-title

© Dr. Philippe J.S. De Brouwer

A line-plot example i

```
# Prepare the data:
years <- c(2000,2001,2002,2003,2004,2005)
sales <- c(2000,2101,3002,2803,3500,3450)
plot(x = years,y = sales, type = 'b',
     xlab = "Years", ylab = "Sales in USD",
     main = "The evolution of our sales")
points(2004,3500,col="red",pch=16) # highlight one point
text(2004,3400,"top sales")        # annotate the highlight
```

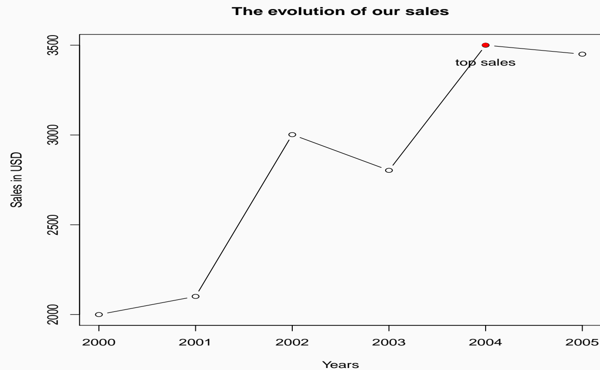


Figure 14: A line plot of the type b, with one dot highlighted.

PART 02: STARTING WITH R AND ELEMENTS OF STATISTICS

CHAPTER 9: VISUALISATION METHODS

SECTION 3:

Pie Charts

Function use for pie()

```
pie(x, labels = names(x), edges = 200, radius = 0.8, clockwise = FALSE, init.angle =
if(clockwise) 90 else 0, density = NULL, angle = 45, col = NULL, border = NULL, lty
= NULL, main = NULL, ...)
```

where the most important parameters are

- `x:` a vector of non-negative numerical quantities. The values in 'x' are displayed as the areas of pie slices
- `labels:` strings with names for the slices
- `radius:` the radius of the circle of the chart (value between âĹŠ1 and +1)
- `main:` indicates the title of the chart
- `col:` the colour palette
- `clockwise:` a logical value indicating if the slices are drawn clockwise or anti clockwise

© Dr. Philippe J.S. De Brouwer

Pie chart example i

```
x <- c(10, 20, 12)                  # Create data for the graph
labels <- c("good", "average", "bad")
pie(x,labels)                       # Show in the R Graphics screen
```

© Dr. Philippe J.S. De Brouwer

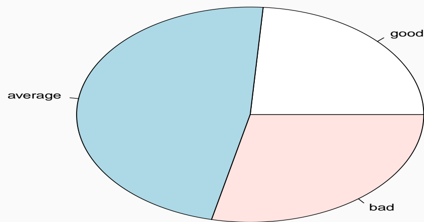


Figure 15: A pie-chart in R.

part 02: Starting with R and Elements of Statistics

chapter 9: Visualisation Methods

section 4:

Bar Charts

© Dr. Philippe J.S. De Brouwer

The function barplot()

Function use for barplot()

```
barplot(height, width=1, xlab=NULL, ylab=NULL, main=NULL, names.arg=NuLL, col=NULL,
...)
```

Some parameters:

- `height:` is the vector or matrix containing numeric values used in chart
- `xlab:` the label for the *x*-axis
- `ylab:` is the label for *y*-axis
- `main:` is the title of the chart
- `names.arg:` is a vector of names of each bar
- `col:` is used to give colors to the bars in the graph.

An example for barplot() i

```
sales <- c(100,200,150,50,125)
regions <- c("France", "Poland", "UK", "Spain", "Belgium")
barplot(sales, width=1,
      xlab="Regions", ylab="Sales in EUR",
        main="Sales 2016", names.arg=regions,
        border="blue", col="brown")
```

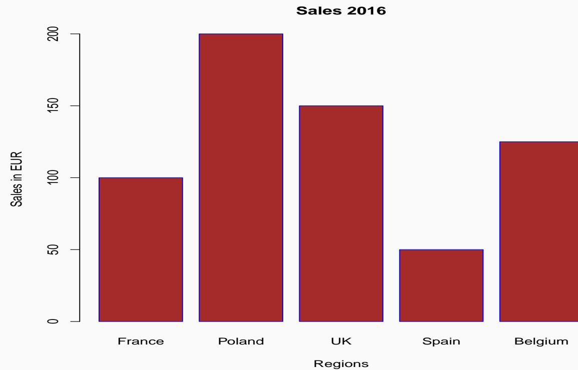


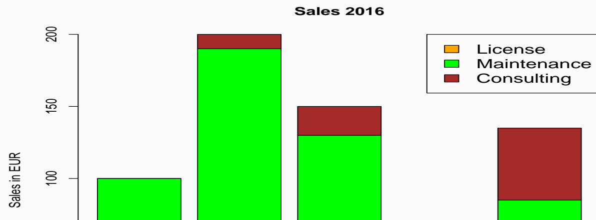
Figure 16: A standard bar-chart based on a vector.

```
# Create the input vectors:
colours <- c("orange","green","brown")
regions <- c("Mar","Apr","May","Jun","Jul")
product <- c("License","Maintenance","Consulting")

# Create the matrix of the values.
values <- matrix(c(20,80,0,50,140,10,50,80,20,10,30,
     10,25,60,50), nrow = 3, ncol = 5, byrow = FALSE)

# Create the bar chart:
barplot(values, main = "Sales 2016",
 names.arg = regions, xlab = "Region",
 ylab = "Sales in EUR", col = colours)

# Add the legend to the chart:
legend("topright", product, cex = 1.3, fill = colours)
```



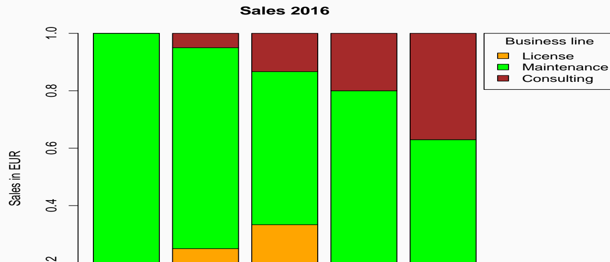
Barplots With Total of 100 Procent

```
# We reuse the matrix 'values' from previous example.

# Add extra space to right of plot area by changeing clipping to figure:
par(mar = c(5, 4, 4, 8) + 0.1, # default margin was c(5, 4, 4, 2) + 0.1
    xpd = TRUE)      # TRUE to restrict all plotting to the plot region

# Create the plot with all totals coerced to 1.0 with prop.table():
barplot(prop.table(values, 2), main = "Sales 2016",
   names.arg = regions, xlab = "Region",
   ylab = "Sales in EUR", col = colours)

# Add the legend, but move it to the right with inset:
legend("topright", product, cex = 1.0, inset=c(-0.3,0), fill = colours,
       title="Business line")
```



© Dr. Philippe J.S. De Brouwer

part 02: Starting with R and Elements of Statistics

chapter 9: Visualisation Methods

section 5:

Boxplots

© Dr. Philippe J.S. De Brouwer

Function use for boxplot()

`boxplot(formula, data = NULL,notch = FALSE, varwidth = FALSE, names, main = NULL, ...` with: Following is the description of the parameters used âĹŠ

- `formula:` a vector or a formula.
- `data:` the data frame.
- `notch:` a logical value (set to TRUE to draw a notch)
- `varwidth:` a logical value (set to true to draw width of the box proportionate to the sample size)
- `names:` the group labels which will be printed under each boxplot.
- `main:` the title to the graph.
- `range:` this number determines how far the plot whiskers can reach. If it is positive, then the whiskers extend to the most extreme data point which is no more than "range" times the interquartile range from the box. If range is set to 0, then the whiskers extend to the data extremes.

© Dr. Philippe J.S. De Brouwer

A boxplot example i

To illustrate boxplots, we can consider the dataset `ships` (from the library “MASS”), and use the following code to generate Figure 19 on slide 202:

```
library(MASS)
boxplot(mpg ~ cyl,data=mtcars,col="khaki3",
        main="MPG by number of cylinders")
```

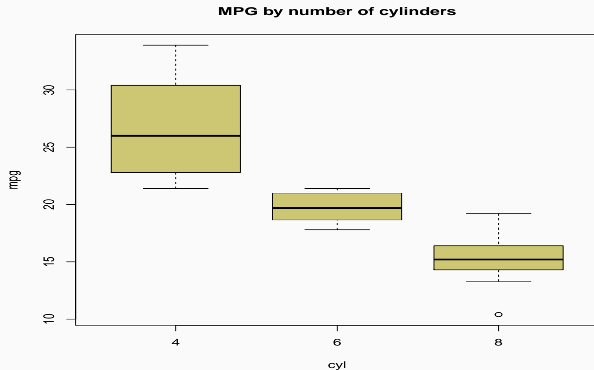


Figure 19: Boxplots show information about the central tendency (median) as well as the spread of the data.

PART 02: STARTING WITH R AND ELEMENTS OF STATISTICS

CHAPTER 9: VISUALISATION METHODS

SECTION 6:

Violin Plots

© Dr. Philippe J.S. De Brouwer

part 02: Starting with R and Elements of Statistics

chapter 9: Visualisation Methods

section 7:

Histograms

The function hist()

Function use for hist()

`hist(x, breaks = "Sturges", freq = NULL, probability = !freq, include.lowest = TRUE, right = TRUE, density = NULL, angle = 45, col = NULL, border = NULL, main = paste("Histogram of" , deparse(substitute(x))), xlim = range(breaks), ylim = NULL, xlab = deparse(substitute(x)), ylab, axes = TRUE, plot = TRUE, labels = FALSE, nclass = NULL, warn.unused = TRUE, ...)` with the most important parameters:

- `x:` the vector containing numeric values to be used in the histogram
- `main:` the title of the chart
- `col:` the color of the bars
- `border:` the border color of each bar
- `xlab:` the title of the *x*-axis
- `xlim:` the range of values on the *x*-axis
- `ylim:` the range of values on the *y*-axis
- `breaks:` one of
  - a vector giving the breakpoints between histogram cells,
  - a function to compute the vector of breakpoints,
  - a single number giving the number of cells for the histogram,
  - a character string naming an algorithm to compute the number of cells,
  - a function to compute the number of cells

Histogram example i

```r
library(MASS)
incidents <- ships$incidents
# figure 1: with a rug and fixed breaks
hist(incidents,
   col=c("red","orange","yellow","green","blue","purple"))
rug(jitter(incidents))  # add the tick-marks
```

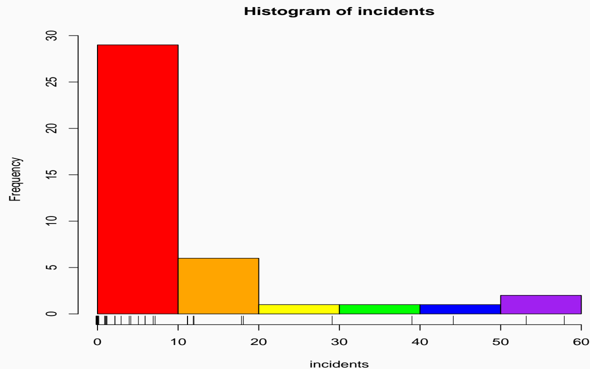
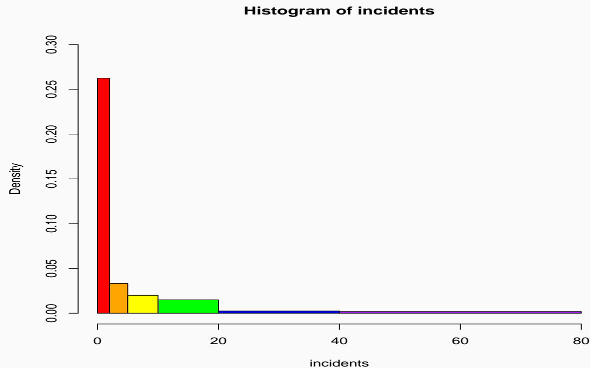


Figure 20: A histogram in R is produced by the hist() function.

Histogram example iii

```
# figure 2: user-defined breaks for the buckets
hist(incidents,
  col=c("red","orange","yellow","green","blue","purple"),
  ylim=c(0,0.3), breaks=c(0,2,5,10,20,40,80),freq=FALSE)
```



Histogram example iv

Figure 21: In this histogram, the breaks are changed, and the y-axes is now calibrated as a probability. Note that leaving freq=TRUE would give the wrong impression that there are more observations in the wider brackets.

PART 02: Starting with R and Elements of Statistics

CHAPTER 9: Visualisation Methods

SECTION 8:

Plotting Functions

© Dr. Philippe J.S. De Brouwer

While the function plot() allows to draw functions, there is a specific function `curve()` that draws functions. The following code illustrate this function by creating Figure 22 on slide 212 and makes also clear how to add mathematical expressions to the plot:

```
fn1 <- function(x) sqrt(1-(abs(x)-1)^2)
fn2 <- function(x) -3*sqrt(1-sqrt(abs(x))/sqrt(2))
curve(fn1,-2,2,ylim=c(-3,1),col="red",lwd = 4,
      ylab = expression(sqrt(1-(abs(x)-1)^2) +++ fn_2))
curve(fn2,-2,2,add=TRUE,lw=4,col="red")
text(0,-1,expression(sqrt(1-(abs(x)-1)^2)))
text(0,-1.25,"++++")
text(0,-1.5,expression(-3*sqrt(1-sqrt(abs(x))/sqrt(2))))
```

© Dr. Philippe J.S. De Brouwer

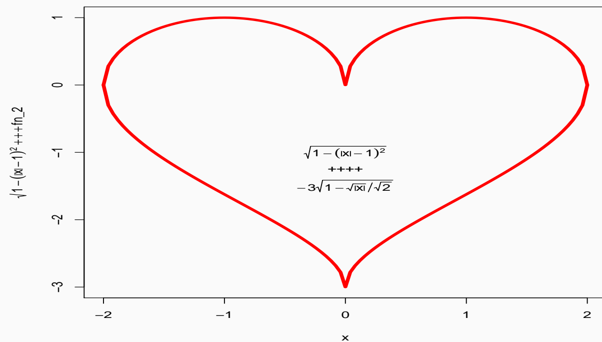


Figure 22: Two line plots plotted by the function curve().

PART 02: STARTING WITH R AND ELEMENTS OF STATISTICS

CHAPTER 9: VISUALISATION METHODS

SECTION 9:

Maps and Contour Plots

© DR. PHILIPPE J.S. DE BROUWER

part 02: Starting with R and Elements of Statistics

chapter 9: Visualisation Methods

section 10:

Heat-maps

© Dr. Philippe J.S. De Brouwer

```
d = as.matrix(mtcars, scale = "none")
heatmap(d)
```

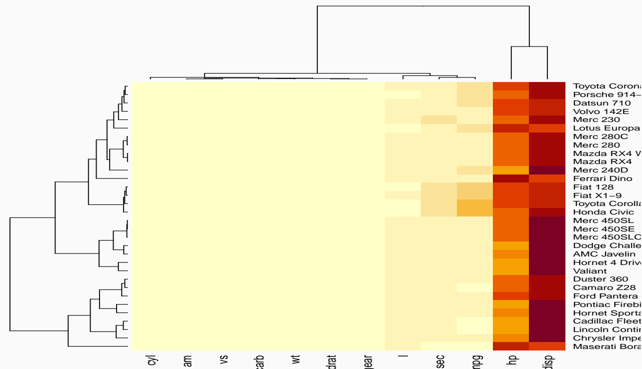


Figure 23: Heatmap for the "mtcars" data.

```
heatmap(d,scale="column")
```

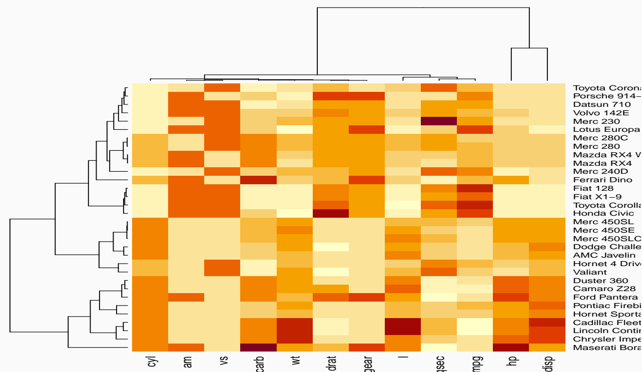


Figure 24: Heatmap for the “mtcars” data with all columns rescaled

PART 02: STARTING WITH R AND ELEMENTS OF STATISTICS

CHAPTER 9: VISUALISATION METHODS

SECTION 11:

Text Mining

© DR. PHILIPPE J.S. DE BROUWER

```
# If neccesary first download the packages:
install.packages("tm")           # text mining
install.packages("SnowballC")    # text stemming
install.packages("RColorBrewer") # colour palettes
install.packages("wordcloud")    # word-cloud generator
```

Then, we load the packages:

```
# Then load the packages:
library("tm")
library("SnowballC")
library("RColorBrewer")
library("wordcloud")
```

© Dr. Philippe J.S. De Brouwer

Step 1: Importing the Text

```
# In this example we use a text version of this very book.
# You will need to use your own text file in the line below:
t <- readLines("../../data/r-book.txt")

# Then create a corpus of text
doc <- Corpus(VectorSource(t))
```

```
# The file has still a lot of special characters
# e.g. the following replaces '\', '#', and '|' with space:
toSpace <- content_transformer(function (x , pattern )
                               gsub(pattern, " ", x))
doc <- tm_map(doc, toSpace, "\\\\")
doc <- tm_map(doc, toSpace, "#")
doc <- tm_map(doc, toSpace, "\\|")
# Note that the backslash needs to be escaped in R
```

Step 3: Build a Term-document Matrix

```
dtm <- TermDocumentMatrix(doc)
m   <- as.matrix(dtm)
v   <- sort(rowSums(m),decreasing=TRUE)
d   <- data.frame(word = names(v),freq=v)
head(d, 10)
##                word freq
## function function  378
## data         data  240
## use           use  190
## model       model  180
## example   example  153
## code         code  142
## package   package  125
## company   company  116
## method     method  103
## market     market   96
```

Step 4: Generate the Word-cloud i

Finally, we can generate the word-could and produce Figure 25 on slide 223.

```
set.seed(1879)
wordcloud(words = d$word, freq = d$freq, min.freq = 10,
          max.words=200, random.order=FALSE, rot.per=0.35,
          colors=brewer.pal(8, "Dark2"))
```

© Dr. Philippe J.S. De Brouwer

Step 4: Generate the Word-cloud ii

```
findFreqTerms(dtm, lowfreq = 150)
## [1] "example"  "model"    "function" "data"     "use"
```

One can analyze the association between frequent terms (i.e., terms which correlate) using findAssocs() function. function `findAssocs()` of the same package can identifey the functions

© Dr. Philippe J.S. De Brouwer

```
# e.g. for the word "function"
findAssocs(dtm, terms = "function", corlimit = 0.15)
## $`function`
##                recycled                     list
##                    0.27                     0.25
##               arguments                 argument
##                    0.24                     0.22
##                sequence                  strings
##                    0.22                     0.22
##                  second               dispatcher
##                    0.21                     0.21
##                 calling            functionality
##                    0.21                     0.20
##                     via                     code
##                    0.19                     0.19
##                  unique                    calls
##                    0.19                     0.19
##                 element             clusterapply
##                    0.18                     0.18
##                    work                   allows
##                    0.17                     0.17
##                     run                  cluster
##                    0.17                     0.17
##                   apply                  wrapped
##                    0.17                     0.16
##                 shorter                 dispatch
##                    0.16                     0.16
##                  except                     note
##                    0.16                     0.15
```

part 02: Starting with R and Elements of Statistics

chapter 9: Visualisation Methods

section 12:

Colours in R

© Dr. Philippe J.S. De Brouwer

R has 657 coulours built-in and ready to use

This list of colours can be used to search for a colour whose name contains a certain string.

```
# find colour numbers that contain the word 'khaki'
grep("khaki",colours())
## [1]  83 382 383 384 385 386

# find the names of those colours
colors()[grep("khaki",colours())]
## [1] "darkkhaki" "khaki"     "khaki1"    "khaki2"    "khaki3"
## [6] "khaki4"
```

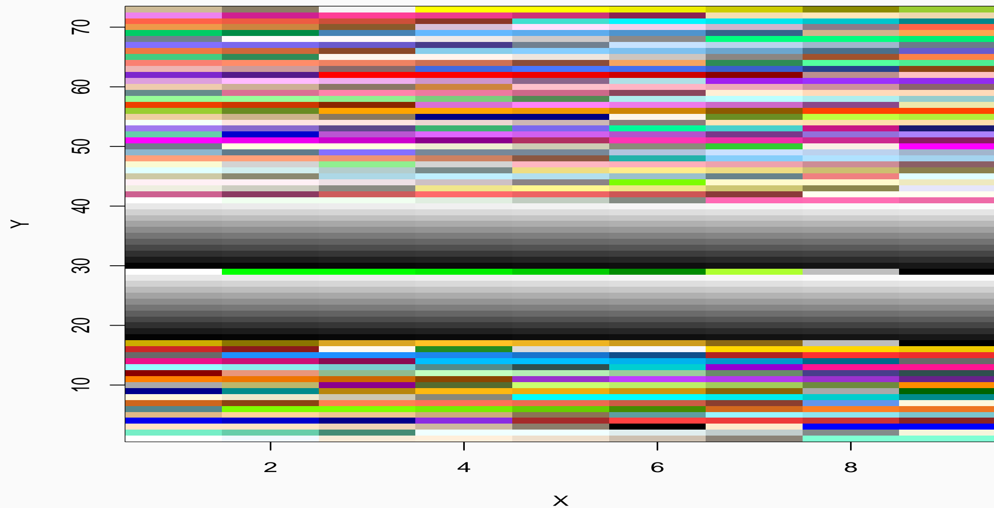
R allows also to define colours in different ways: named colours, RGB colours, hexadecimal colours, and it also allows to convert the one to the other.

```
# extract the rgb value of a named colour
col2rgb("khaki3")
##       [,1]
## red    205
## green  198
## blue   115
```

Visualising all the built-in colours i

```
N   <- length(colours())  # this is 657
df  <- data.frame(matrix(1:N, nrow=73, byrow = TRUE))
image(1:(ncol(df)), 1:(nrow(df)), as.matrix(t(df)),
      col = colours(),
      xlab = "X", ylab = "Y")
```

© Dr. Philippe J.S. De Brouwer



© Dr. Philippe J.S. De Brouwer

part 02: Starting with R and Elements of Statistics

chapter 10:

Time Series Analysis

© Dr. Philippe J.S. De Brouwer

part 02: Starting with R and Elements of Statistics

chapter 10: Time Series Analysis

section 1:

Time Series in R

© Dr. Philippe J.S. De Brouwer

The time series object is created by using the `ts()` function.

Function use for ts()

```
ts(data = NA, start = 1, end = numeric(), frequency = 1,
       deltat = 1, ts.eps = getOption("ts.eps"), class = ,
       names = )
```

with

- `data:` A vector or matrix containing the values used in the time series.
- `start:` The start time for the first observation in time series.
- `end:` The end time for the last observation in time series.
- `frequency` The number of observations per unit time.
  - frequency = 12: pegs the data points for every month of a year.
  - frequency = 4: pegs the data points for every quarter of a year.
  - frequency = 6: pegs the data points for every 10 minutes of an hour.
  - frequency = 24×6: pegs the data points for every 10 minutes of a day.

Except the parameter "data" all other parameters are optional. To check if an object is a time series, we can use the function `is.ts()`, and `as.ts(x)` will coerce the variable `x` into a time series object.

© Dr. Philippe J.S. De Brouwer

Time series example

```r
library(MASS)
# The SP500 is available as a numeric vector:
str(SP500)
##  num [1:2780] -0.259 -0.865 -0.98 0.45 -1.186 ...
```

Now, we convert it to a time series object with the function `ts()`:

```r
# Convert it to a time series object.
SP500_ts <- ts(SP500,start = c(1990,1),frequency = 260)
```

```
plot(SP500_ts)
```

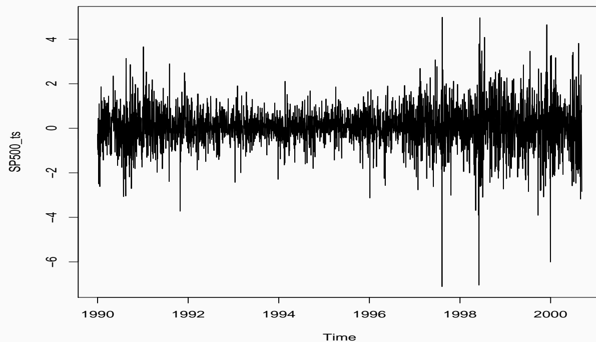


Figure 27: The standard plot for a time series object for the returns of the SP500 index in the 1990s.

Multiple time series in one object

```
val = c(339.97)
for (k in 2:length(SP500)){
  val[k] = val[k-1] * (SP500[k-1] / 100 + 1)
}
# Convert both series to a matrix:
M <- matrix(c(SP500,val),nrow=length(SP500))

# Convert the matrix to a time series object:
SP <- ts(M, start=c(1990,1),frequency=260)
colnames(SP) <- c("Daily Return in Pct","Value")
```

```
plot(SP, type = "l", main = "SP500 in the 1990s")
```

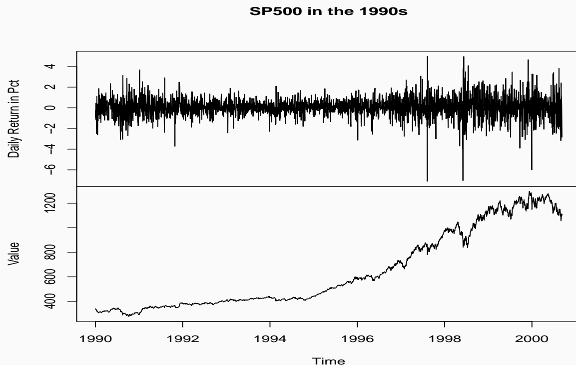


Figure 28: The standard plot functionality of time series will keep the *z*-axis for both variables the same (even use one common axis).

part 02: Starting with R and Elements of Statistics

chapter 10: Time Series Analysis

section 2:

Forecasting

© Dr. Philippe J.S. De Brouwer

In absence of a clear and simple trend (such as a linear or exponential trend) the moving average is a versatile tool. it is a non-parametric model that simply "forecasts" the near future based on the average observations of the near past.

Example (– GDP data)

When it comes to macro economical data, the World Bank is a class apart. It's website `https://data.worldbank.org` has thousands of indicators that can be downloaded and analysed. Their data catalogue is here: `https://datacatalog.worldbank.org`. We have downloaded the GDP data of Poland and stored it in a csv-file on our hard-disk. In the example that we use to explain the concepts in the following sections, we will use that data.

To start, we load in the data stored in a csv file on our local hard-drive, and plot the data in Figure 29 on slide 239

```
g <- read.csv('../../data/gdp/gdp_pol_sel.csv') # get the data
attach(g) # the names of the data are now always available
plot(year, GDP.per.capitia.in.current.USD, type='b',
     lwd = 3, xlab = 'Year', ylab = 'Polish GDP per Capita in USD')
```

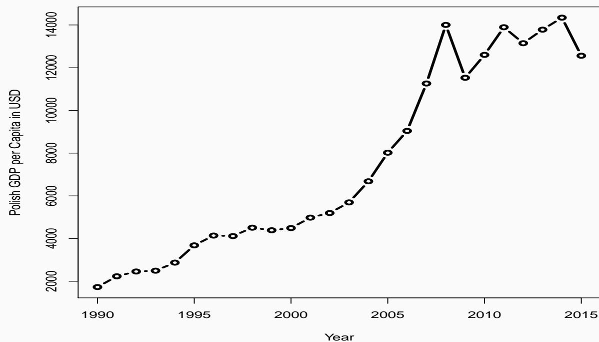


Figure 29: A first plot to show the data before we start. This will allow us to select a suitable method for forecasting.

```
require(forecast)
# make the forecast with the moving average (ma)
g.data <- ts(g$GDP.per.capitia.in.current.USD,start=c(1990))
g.movav = forecast(ma(g.data, order=3), h=5)
```

```
# show the result:
plot(g.movav,col="blue",lw=4,
     main="Forecast of GDP per capita of Poland",
     ylab="Income in current USD")
lines(year,GDP.per.capitia.in.current.USD,col="red",type='b')
```

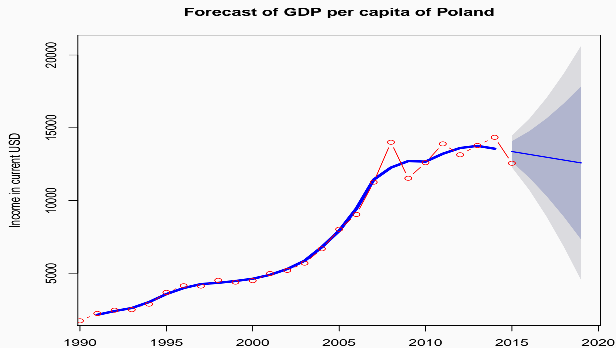


Figure 30: A forecast based on moving average.

© Dr. Philippe J.S. De Brouwer

Testing the Accuracy of Forecasts – Backtesting

```
# Testing accuracy of the model by sampling:
g.ts.tst <- ts(g.data[1:20],start=c(1990))
g.movav.tst <- forecast(ma(g.ts.tst,order=3),h=5)
accuracy(g.movav.tst, g.data[22:26])
##                       ME      RMSE       MAE        MPE      MAPE
## Training set    32.0006  342.1641  217.8447  0.7619824  3.229795
## Test set     -1206.5948 1925.5738 1527.0227 -9.2929075 11.599237
##                   MASE        ACF1
## Training set 0.3659014 -0.06250102
## Test set     2.5648536          NA
```

```
plot(g.movav.tst,col="blue",lw=4,
     main="Forecast of GDP per capita of Poland",
     ylab="Income in current USD")
lines(year, GDP.per.capitia.in.current.USD, col="red",type='b')
```

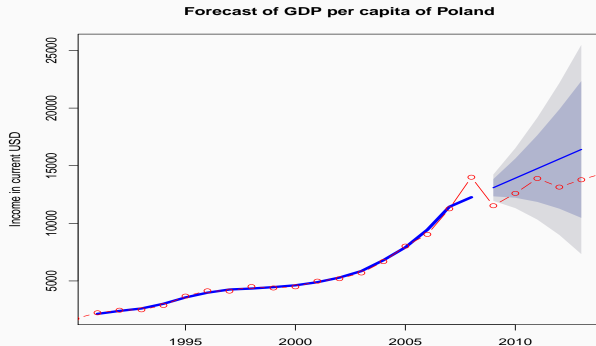


Figure 31: A backtest for our forecast.

Testing the accuracy of forecasts – backtesting

In the forecast package, there is an automated forecasting function that will run through possible models and select the most appropriate model give the data. This could be an auto regressive model of the first order (AR(1)), an ARIMA model (autoregressive integrated moving average model) with the right values for p, d, and q, or even something else that is more appropriate. The following code uses those functions to plot a forecast in figure Figure **??** on slide **??**.

```
train = ts(g.data[1:20],start=c(1990))
test  = ts(g.data[21:26],start=c(2010))
arma_fit <- auto.arima(train)
arma_forecast <- forecast(arma_fit, h = 6)
arma_fit_accuracy <- accuracy(arma_forecast, test)
arma_fit; arma_forecast; arma_fit_accuracy
## Series: train
## ARIMA(0,1,0) with drift
##
## Coefficients:
##          drift
##       515.5991
## s.e.  231.4786
##
## sigma^2 estimated as 1074618:  log likelihood=-158.38
## AIC=320.75   AICc=321.5   BIC=322.64
##      Point Forecast    Lo 80    Hi 80     Lo 95    Hi 95
## 2010       12043.19 10714.69 13371.70 10011.419 14074.97
## 2011       12558.79 10680.00 14437.58  9685.431 15432.15
## 2012       13074.39 10773.35 15375.43  9555.257 16593.52
## 2013       13589.99 10932.98 16247.00  9526.444 17653.54
## 2014       14105.59 11134.96 17076.22  9562.406 18648.77
## 2015       14621.19 11367.03 17875.35  9644.381 19598.00
```

```
plot(arma_forecast, col="blue",lw = 4,
     main = "Forecast of GDP per capita of Poland",
     ylab = "income in current USD")
lines(year,GDP.per.capitia.in.current.USD, col = "red", type = 'b')
```

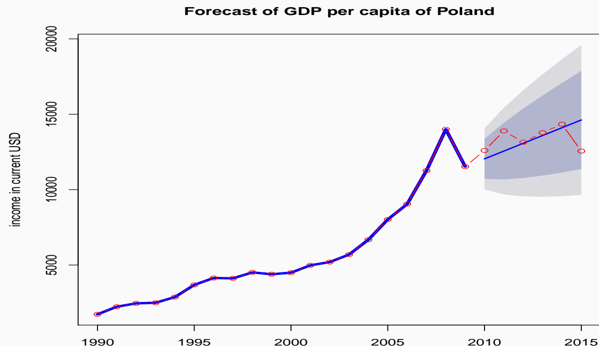


Figure 32: Optimal moving average forecast.

© Dr. Philippe J.S. De Brouwer

Simple Exponential smoothing

Exponential smoothing assigns higher weights to the most recent observations (the weight will decrease exponentially for older observations). The effect will be that a new dramatic event has a much faster impact, and that the "memory of it" will decrease exponentially.

The package `forecast` provides the function `ses` to execute this as follows:

```
g.exp <- ses(g.data,5,initial="simple")
g.exp  # simple exponential smoothing uses the last value as
##      Point Forecast     Lo 80    Hi 80     Lo 95    Hi 95
## 2016       12558.87 11144.352 13973.39 10395.550 14722.19
## 2017       12558.87 10558.438 14559.30  9499.473 15618.27
## 2018       12558.87 10108.851 15008.89  8811.889 16305.85
## 2019       12558.87  9729.832 15387.91  8232.229 16885.51
## 2020       12558.87  9395.909 15721.83  7721.538 17396.20

       # the forecast and finds confidence intervals around it
```

```
plot(g.exp,col="blue",lw=4,
     main="Forecast of GDP per capita of Poland",
     ylab="income in current USD")
lines(year,GDP.per.capitia.in.current.USD,col="red",type='b')
```

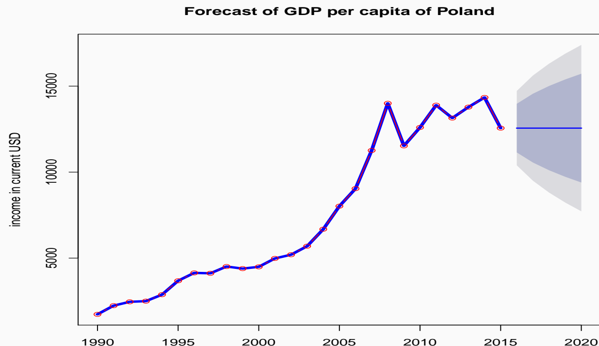


Figure 33: Forecasting with an exponentially smoothed moving average.

© Dr. Philippe J.S. De Brouwer

Holt Exponential smoothing

```
g.exp <- holt(g.data,5,initial="simple")
g.exp  # Holt exponential smoothing
##      Point Forecast    Lo 80    Hi 80    Lo 95    Hi 95
## 2016       13445.71 12144.40 14747.02 11455.53 15435.89
## 2017       13950.04 12257.07 15643.01 11360.86 16539.22
## 2018       14454.37 12444.67 16464.07 11380.80 17527.94
## 2019       14958.70 12675.80 17241.60 11467.31 18450.10
## 2020       15463.04 12936.30 17989.77 11598.73 19327.34
```

```
plot(g.exp,col="blue",lw=4,
     main="Forecast of GDP per capita of Poland",
     ylab="income in current USD")
lines(year,GDP.per.capitia.in.current.USD,col="red",type='b')
```

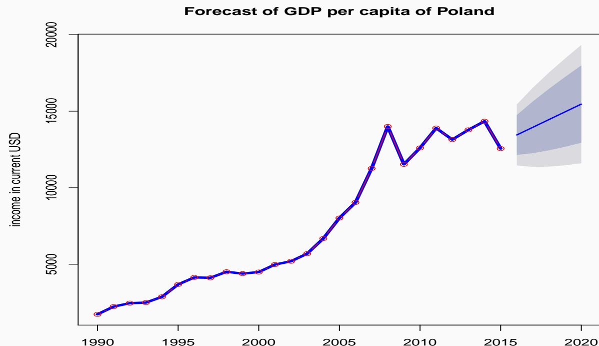


Figure 34: Holt exponentially smoothed moving average.

© Dr. Philippe J.S. De Brouwer

```
# we use the data nottem
# Average Monthly Temperatures at Nottingham, 1920-1939
nottem.stl = stl(nottem, s.window="periodic")
plot(nottem.stl)
```

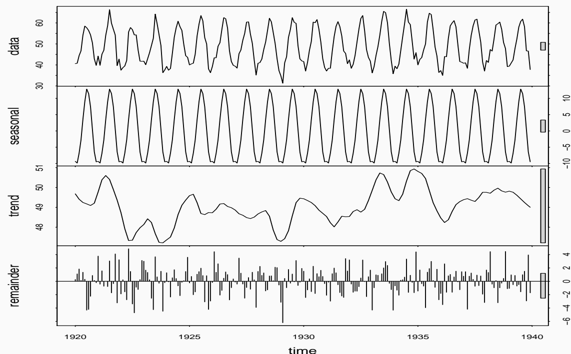


Figure 35: Using the stl-function to decompose data in a seasonal part and a trend.

```
# Simple exponential: models level
fit <- HoltWinters(g.data, beta=FALSE, gamma=FALSE)

# Double exponential: models level and trend
fit <- HoltWinters(g.data, gamma=FALSE)

# Triple exponential: models level, trend, and seasonal
# components. This fails on the example, as there is no
# seasonal trend:
#fit <- HoltWinters(g.data)

# Predictive accuracy
library(forecast)
accuracy(forecast(fit,5))
##                     ME     RMSE      MAE       MPE     MAPE      MASE
## Training set -69.84485 1051.488 711.7743 -2.775476 9.016881 0.8422587
##                     ACF1
## Training set 0.008888197
```

© Dr. Philippe J.S. De Brouwer

```
# predict next 5 future values
forecast(fit, 5)
##      Point Forecast    Lo 80    Hi 80    Lo 95    Hi 95
## 2016       13457.63 12084.15 14831.11 11357.07 15558.18
## 2017       13961.96 12179.69 15744.23 11236.21 16687.71
## 2018       14466.29 12352.87 16579.71 11234.10 17698.49
## 2019       14970.62 12571.33 17369.91 11301.23 18640.02
## 2020       15474.95 12820.41 18129.50 11415.17 19534.74
```

```
plot(forecast(fit, 5),col="blue",lw=4,
     main="Forecast of GDP per capita of Poland",
     ylab="income in current USD")
lines(year,GDP.per.capitia.in.current.USD,col="red",type='b')
```

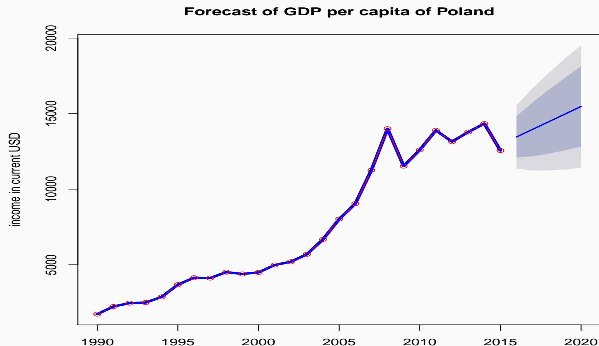


Figure 36: The Holt-Winters model fits an exponential trend. Here we plot the double exponential model.

Question #7

Use the moving average method on the temperatures in Nottingham (nottam). Does it work? Which model would work better?

© Dr. Philippe J.S. De Brouwer

part 02: Starting with R and Elements of Statistics

chapter 11:

Further Reading

© Dr. Philippe J.S. De Brouwer

Further information – CRAN

The website of the R is `https://cran.r-project.org` and it carries also plenty of documentation. Here is a selection:

- `/doc/manuals/R-intro.html:` another take on what we have covered in this part.
- `/doc/manuals/R-FAQ.html` A faq with very specific information that never will be included in a book like these but that might be relevant if you have happen to have certain backgrounds (such as S for example)
- `/doc/manuals/R-lang.html` describes in much more detail the internal workings of the language.
- `/doc/manuals/R-data.html:` covers the data-import and data-export functions in great detail. Best to read after next part if you happen to be on the data-side of the development cycle.
- `/doc/manuals/R-exts.html` will help you to create your very own packages. This makes a lot of sense, even if your company want to keep them propriety to the company. Capturing all the logic of your specifics in a package will save enormous amounts of time and money. Just imagine that you write once how to import data from your corporate mainframe or central data-warehouse. Hundreds of programmers and thousands of modellers can use the same package. Model validators only need to review this module once, etc.