

R- a tool for statistical analysis

Introduction course

Håkan Jonsson and Kaspar Meili

Department of Epidemiology and Global Health,
Faculty of Medicine,
Umeå University

June 26, 2020

Contents

1	Installation	1
1.1	R	1
1.2	RStudio	1
2	Introduction	3
3	The software R	5
4	RStudio	7
4.1	How script files work	8
4.2	Why script files?	8
5	R objects - part 1	11
5.1	Objects	11
5.2	Numbers	11
5.3	Vectors	12
5.3.1	Contruction of vectors - the <code>c()</code> function	12
5.3.2	Examples of numerical vectors and calculations	13
5.4	Type and class	13
5.4.1	Integer	14
5.4.2	Logical	14
5.4.3	Character	15
5.5	Accessing elements of a vector	15
5.6	Concatenating text elements with the paste function	16
5.7	Data frames	17
5.7.1	Accessing elements of a data frame	17
5.8	Tibbles	18

6	Package installation and help resources	19
6.1	Package installation	19
6.2	Help resources	19
6.2.1	?	19
6.2.2	Vignettes	19
6.2.3	Internet	20
7	Import and export of data sets	21
7.1	R data sets	21
7.2	External data	21
7.2.1	Working directory	21
7.2.2	CSV import	21
7.2.3	Excel	22
7.3	Import and export file formats from other statistical software	22
7.4	Import export using RStudio	22
7.5	R binary files	22
8	R objects - part 2	25
8.1	Factors	25
8.2	Lists	26
8.3	Functions	29
8.3.1	Installation of a package	30
8.3.2	Loading a package	30
8.3.3	Masking	31
9	Some useful functions	33
9.1	Functions in R	33
9.2	Some basic functions on numerical vectors	33
9.3	Generic functions	34
9.4	Some functions for investigation of data frames	34
9.5	Some more useful functions for data frames	35
9.6	Systematic generation of data	37
9.7	Sorting	38
9.8	Filtering subsets of data frames	40

<i>CONTENTS</i>	5
9.9 Missing data	41
9.9.1 Some functions when dealing with missing data	43
9.9.2 Arguments in functions to handle missing data	44
9.10 The function cut	45
9.11 The function apply	46
9.12 Extension of apply to tapply	47
9.13 The head and tail functions	47
10 Tables	49
10.1 Enveloping	50
10.1.1 Summary	50
10.1.2 Proportion	50
10.1.3 Tabulation of more variables	51
11 Basic graphics	53
11.1 Introduction to graphics with R	53
11.2 Histogram	53
11.3 Boxplot	54
11.4 Scatterplot	55
11.4.1 Graphical parameters	58
12 Simulation	59
12.1 Simulation - what is it?	59
12.2 Generate random samples	59
12.3 Generate observations with common distributions like normal, uniform etc.	61
12.4 Generating observations with other distributions	64
12.5 Bootstrapping	64
13 R markdown	67
13.1 Text formatting	68
13.2 Code chunks and plots	68
13.3 Tables	69
14 Programming	71
14.1 Branching	71

14.2	Loops	72
14.3	Functions	74
14.3.1	Anonymous functions	75
14.3.2	Return values	75
14.3.3	Variable scope	77
15	Statistical methods part 1	79
15.1	Statistical measures	79
15.1.1	Data set	79
15.1.2	Mean, median and variance	80
15.1.3	Correlation	81
15.2	Test and confidence intervals	82
15.2.1	T-test	83
15.2.2	Two-sample t-test	83
15.2.3	One-sided t-test	85
15.2.4	Welch's test	86
15.2.5	One-sample t-test	86
15.2.6	Paired t-test	87
15.3	Nonparametric tests	88
15.3.1	Wilcoxon rank sum test (Mann-Whitney test)	89
15.3.2	Wilcoxon signed rank test	89
15.4	Chi-2 test and Fishers exact test	89
15.4.1	Aggregated data	89
15.4.2	Individual data	91
15.4.3	Fishers exact test	92
15.4.4	Tests in n by k tables	93
15.4.5	Count data in one dimension	93
15.5	Kolmogorov-Smirnov test	94
15.6	Q-Q plot	94
16	ggplot2	97
16.1	ggplot2	97
16.2	Plotting distribution of single variables	97

16.3 Relationship of variables	101
16.4 Scales and legends	104
16.4.1 Continous scales	104
16.4.2 Discrete scales	106
16.4.3 Legends	107
17 Functions in tidyverse	109
17.1 The packages dplyr and tidyr - the core of tidyverse	109
17.1.1 Data set	109
17.2 Single table verbs	111
17.2.1 <code>select()</code>	111
17.2.2 <code>filter()</code>	113
17.2.3 <code>slice()</code>	114
17.2.4 <code>arrange()</code>	114
17.2.5 <code>rename()</code>	115
17.2.6 <code>mutate()</code>	115
17.2.7 <code>count()</code>	116
17.2.8 <code>base::summary()</code>	117
17.3 Pipes	117
17.4 Manipulating factors with tidyverse tools	118
17.4.1 Coercing numeric and character variables to a factor	119
17.4.2 Changing the codes of a factor	120
17.4.3 Re-ordering factor levels	120
17.4.4 Re-coding (renaming) categories (levels) in factors	121
17.4.5 Converting numerics to factors using the <code>cut_</code> functions	121
17.5 <code>group_by()</code>	123
17.5.1 Summarising groups	123
17.6 <code>left_join()</code> : Merging two data frames	125
17.7 Tidyr functions	127
17.7.1 Functions for converting between long and wide format	127
17.7.2 Example - the dataset Subliminal	128
18 Statistical methods part 2	131

18.1	The formula object	131
18.2	Linear regression	131
18.2.1	Simple linear regression	131
18.2.2	Multiple linear regression	134
18.2.3	Including factors in the regression model	135
18.3	Analysis of variance - ANOVA	138
18.3.1	One-way anova	138
18.3.2	Likelihood ratio test	139
18.3.3	Two-way anova	140
18.3.4	Interaction	140
18.4	Generalized linear models	141
18.4.1	Logistic regression	142
18.4.2	Confidence intervals	143
18.5	Survival analysis	144
18.5.1	Survival data	144
18.5.2	The survival object	144
18.5.3	Survival curves - Kaplan Meier	145
18.5.4	Comparing Survival curves - log rank test	145
18.6	Cox models	147
18.6.1	Model check	148
18.6.2	Stratification	149
19	Graphics with ggplot2 - part 2	151
19.1	Overplotting	151
19.2	Multiple plots	152
19.2.1	Using different data sets in the same plot	152
19.2.2	Using aesthetics (aes argument) to make multiple plots	153
19.3	Using facets	158
19.3.1	Using facets and other grouping together with converting data into long format	166
20	Matrix calculations	169
20.1	Matrix definition	169
20.2	Calculations	170

20.3 Equation system and matrix inverse	172
20.4 Arrays	173

Chapter 1

Installation

1.1 R

R is the underlying software that does all the calculations. R can be downloaded and installed from the R project website: <https://www.r-project.org/>. Or from the Swedish mirror here (which actually is in Umeå): Download R. Generally no adjustments are needed in the Windows installation program; always clicking on "next" during the installation process should work fine.

1.2 RStudio

While R is doing all the work, it is not very convenient to work with. This we work with RStudio during these course. It is a program to that makes it much easier to work with R. Go to rstudio.com. Find the download link, download and install the free RStudio Desktop version.

Chapter 2

Introduction

This document is designed to guide the students through the R course "R- a tool for statistical analysis" given at the University of Umeå. It is not a comprehensive introduction to R but focuses on introducing basic concepts. Examples are mostly of epidemiological nature, but the knowledge of how to use R, especially the broader context of scripting and data analysis, is not subject specific and hence students from all disciplines should be readily able to adapt the concepts to their subjects.

R is based on another programming language, S, and was first released in 1995. Nowadays it is a very popular choice for performing statistical evaluations and data analysis. One of the main advantages of R is, that a lot of extensions ("packages" in R terminology) have been developed for it that greatly enhance the functionality. R is also free and open source, meaning that everybody can look at the code of R itself, can modify it, and can adopt it for other operating systems and architectures.

RStudio is an integrated development environment for R. R itself basically just runs in a console. RStudio provides a user-friendly graphical interface for writing R code, running it, looking at plots data, or documentation, and much more.

R comes with some basic functionality inbuilt; commonly called 'base R'. However, a lot of people are working with R, and there is almost always multiple ways to do something in R. A lot of packages have been published that provide new functionality or that aim to replace the base R functionality with a more refined approach. One noteworthy example here is the tidyverse (<https://www.tidyverse.org/>), a whole collection of packages that aim to make working with R more tidy.

Another advantage of R is related to reproducibility, an important topic in research. R makes it relatively easy to bundle R code, results, and documentation. For example, one could write R code and text in one file and that later is converted into a pdf report that contains the results and figures of a data analysis. One solution to achieve this is called RMarkdown.

During this course we will cover the basics of working with R, handling data with R, how to program in R, how to visualize data, how to calculate statistics, and how to work with RMarkdown.

Chapter 3

The software R

R in its core really is a program that works with instructions, also called commands, in text form. R can receive commands in two ways: The first is to read a whole file with the extension .R. Each line instructs R to do something. In the second variant, the user types a command and then presses enter.

That means that core R is not a program with a sophisticated graphical user interface, as for example Microsoft Word is. Instead, R can be accessed and used by only writing text, without using the mouse.

The following demonstration shows this way of working with R. Later, we will use a program called RStudio that is much more convenient to work with because it features among others a sophisticated graphical user interface.

Note that if you install R, a program that is called *Rgui* will also be installed. This is the standard user interface for R, but we will work instead with RStudio.

The goal of this demonstration is to point out that R at its core is not dependent on windows or menus, but only on text.

- Windows users: → Go to the directory where you installed R, and open the bin directory, this is usually like C:\R\R-3.6.3\bin (the version number is likely to vary). Then doubleclick on R.exe
- Mac users: → Look for R using either the dock, the launchpad, finder, or the spotlight search, then start it.

A terminal will open. Write the following text and press enter (See Figure 3.1):

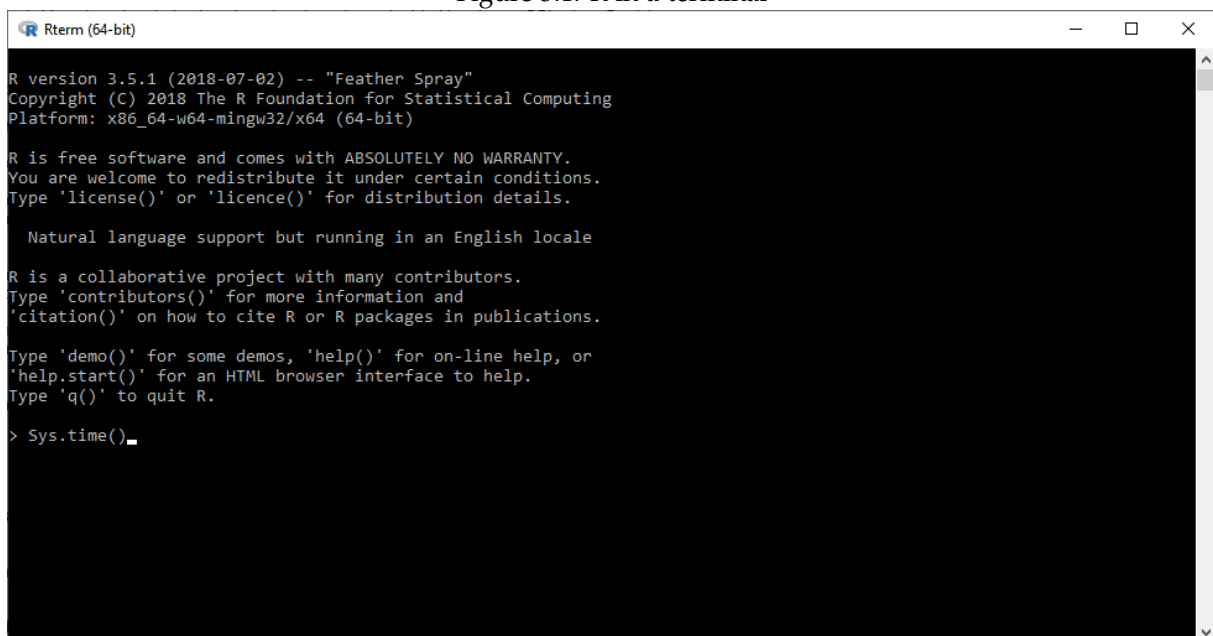
```
Sys.time()  
## [1] "2020-06-07 14:35:48 CEST"
```

The current system time will appear. Congratulations! You just run your first command using R.

Own experimentation

Run play around with the `demo()` function, for example try to execute `demo("graphics")`.

Figure 3.1: R in a terminal



```
Rterm (64-bit)

R version 3.5.1 (2018-07-02) -- "Feather Spray"
Copyright (C) 2018 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

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

  Natural language support but running in an English locale

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

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

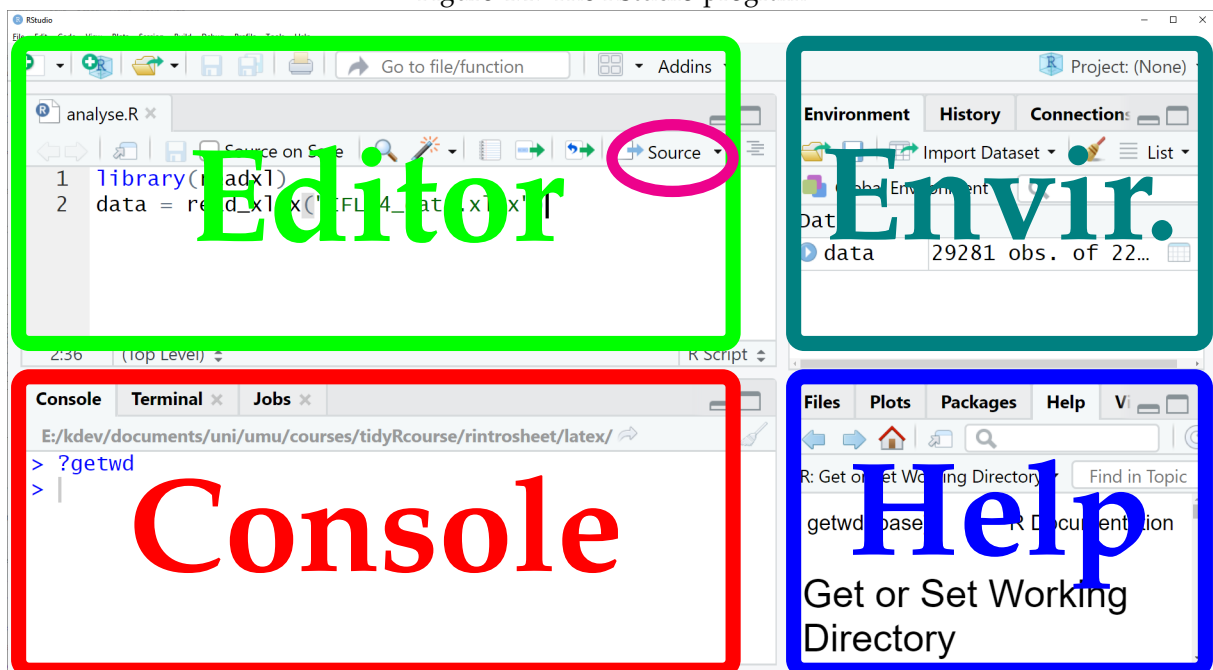
> Sys.time()_
```


Chapter 4

RStudio

Now we switch to RStudio and will see why it is much more convenient. Windows users can open RStudio using the start menu or the search box. Mac users should use the dock, launchpad, finder or spotlight.

Figure 4.1: The Rstudio program



Text editor

The *editor* pane displays the content of the *.R* file. This is the place to put all the code and commands that you want to keep so that you can run the same analysis again in the future. To run all code in the file, click on **Source** on the top right of the editor pane.

Console

The *Console* tab is good to try out single pieces of codes that you are not sure will work or that you just want to play with. Make sure the **Console** tab is selected. Write the code that you want to run and press enter. Pressing the **Arrow-up** key on the keyboard while the cursor is blinking in the console will bring back the last line of code that was run in the console.

Environment

The *Environment* tab in the top right pane contains an overview of all the objects in R. Available R objects will appear there. Ensure the **Environment** tab is selected.

Help

The *Help* tab displays help information. For example writing `?getwd` in the console and pressing **Enter** will show information on setting working directories. The help command can be used with any inbuilt function. Ensure the **Help** tab is selected.

4.1 How script files work

When a script is started by clicking on **Source** in RStudio, R reads the text inside of it from top to bottom. If a line contains text, it instructs R to do certain things. These are instructions. Every instruction is followed by a line break. So that is really all what a script file contains, a series of commands in textual form that are executed from top to bottom of the file.

4.2 Why script files?

As mentioned in the boxes above and in the R Basics chapter, there is two ways of entering R code in Rstudio: The console or executing a file with R code in it. Even if it may be tempting to just use the console, try to always put keep the essential R code in a file so that the analysis can be reproduced later. Actually, it may the likely primary beneficent is the author of the code because it often is difficult to exactly remember how to preform an analysis after some time.

A particular problematic scenario would be if a scientist performed an analysis and published the results in a paper. This scientist does only uses the R console for analysing data, without storing the commands in a file. Somebody else that reads the paper two years later thinks there is problem with the results and tries to contact the scientist. Even though there is a basic description of the procedure in the paper, the scientist does not arrive at the published results, no matter how hard she tries. Is there really a problem with the original analysis? Or does the scientist do something wrong while trying to reproduce the results?

Other benefits of keeping R code in script files:

- Reusable code
- If the data changes, it is easy to run the script again
- It is possible to publish the code and let others reproduce and verify the results

As such, R code files should be self-sufficient, that means they should run without any further

issues after opening them in Rstudio, potentially adjusting the R working directory, and clicking on **Source**.

They should also be well commented so that others or yourself can easily understand what the code is supposed to do.

Chapter 5

R objects - part 1

Håkan Jonsson < hakan.jonsson@umu.se >

5.1 Objects

R is said to be object oriented. This means that everything in R is an object. That is not so strange when we think about a single number or a vector. It may however become more diffuse when an object can include other objects or when all results from a statistical calculation or even a plot is an object. We start from the beginning with numbers and vectors of numbers.

5.2 Numbers

R can e.g. be used as a calculator (some examples below). We can also give names to the objects. Upper case matters.

```
4+13
[1] 17

21-4
[1] 17

21-(-4)
[1] 25

5*7
[1] 35

-3/2
[1] -1.5

5^2 # 5 squared
[1] 25

2^4 # ^ is the power function
```

```

[1] 16
9^0.5 # the square root

[1] 3
sqrt(9) # the specific function for square root

[1] 3

      # (more about functions later)

2^(-2) # 1/(2*2)

[1] 0.25

x<-5
x

[1] 5

Important.Number<-14
Important.Number

[1] 14

x*Important.Number

[1] 70

Important.Number/x

[1] 2.8

x<-2      # it is easier to change x
1/x+1/x^2+1/x^3+1/x^4+1/x^5

[1] 0.96875

```

5.3 Vectors

Vectors play an important role in data analysis because the data in each of our variables *is* a vector. A vector is a list of data which are all of the same type e.g. numbers, character strings or factors. Data frames are the most common data representation when we do statistical analyses or other calculations. We can use vectors to construct dataframes or we can extract vectors from dataframes.

5.3.1 Construction of vectors - the `c()` function

The function `c(some comma-separated list of object arguments)` combines the arguments into a single new object (if possible!). It is one of **the most widely used of all R functions** and is used very often when you want to use a group of numbers or characters as an argument in some other function. Its murky origins in Unix, mean that it is also sometimes referred to as *concatenate*.

If you use `c()` to combine a list of arguments which have different classes, R tries to figure out how the user might want to combine the arguments with a common type. (Remember our standard *atomic vector* must have all elements of the same class.)

Here are some examples:

5.3.2 Examples of numerical vectors and calculations

Now we use vectors of numbers.

```
x<-c(2,4,6)
x

[1] 2 4 6

2*x

[1] 4 8 12

y<-x*x
y

[1] 4 16 36

x^2

[1] 4 16 36

z<-(x+y)/2
z

[1] 3 10 21

z<-c(z,x,3,14)
z

[1] 3 10 21 2 4 6 3 14

xi<-5:12 # to generate a sequence
xi

[1] 5 6 7 8 9 10 11 12
```

5.4 Type and class

R objects have both **type** and **class**. The objects of numbers and vectors above have type=double and class=numeric. Most often this is not necessary to have in mind. However the function **str** is useful to assess the structure of an object as well as the functions **typeof** and **class**.

```
x<-c(2,4,6)
typeof(x) # real numbers

[1] "double"

class(x)

[1] "numeric"

str(x) # describe the object

num [1:3] 2 4 6
```

5.4.1 Integer

Can be described as numerical without decimals

```
x<-c(2,4,6)
y<-as.integer(x)
x

[1] 2 4 6

y

[1] 2 4 6

typeof(x)

[1] "double"

class(x)

[1] "numeric"

typeof(y) #even if you cannot see the difference x and y have different type

[1] "integer"

class(y)

[1] "integer"
```

5.4.2 Logical

The elements of the vector are either TRUE or FALSE.

```
z1<-c(TRUE,FALSE,TRUE)
z1

[1] TRUE FALSE TRUE

typeof(z1) # this is an important type in R

[1] "logical"

class(z1)

[1] "logical"

z2<-c(T,F,T) # the same as z1
z2

[1] TRUE FALSE TRUE

y<-z1*1 # converting logical to a s.k. dummy variable
y

[1] 1 0 1

5<7

[1] TRUE
```



```
x<-c(5<7,5>7,2==2)    # "==" means equal to
x

[1] TRUE FALSE TRUE

# some more examples (!= means not equal to)
x<-c(5>5, 5>=5,3!=5,4!=4)
x

[1] FALSE TRUE TRUE FALSE

!x

[1] TRUE FALSE FALSE TRUE
```

5.4.3 Character

The elements are text strings.

```
x<-"b"
x

[1] "b"

y<-"R-object"
y

[1] "R-object"

x<-c(x,y,"c")
x

[1] "b"          "R-object" "c"

typeof(x)

[1] "character"
```

Own experimentation

Try creating numeric vectors and do calculations. Also create vectors of other type like character and logical.

5.5 Accessing elements of a vector

Elements of a vector are accessed by giving the row(s) within brackets [] or using a logical vector of the same length.

```
x<-c(2,4,7,13,5,8)
x

[1] 2 4 7 13 5 8

y<-x[3]
y
```

```

[1] 7

z<-x[c(1,4,6)]
z

[1] 2 13 8

y2<-x[c(1,1,4,4,4,6,6,6,6)]
y2

[1] 2 2 13 13 13 8 8 8 8

z2<-x[-c(2,3,5)] # minus sign means exclusion of the elements
z2

[1] 2 13 8

x2<-x[c(T,F,F,T,F,T)] # you can also use a vector (of the same length) of logics
x2

[1] 2 13 8

```

5.6 Concatenating text elements with the paste function

It is sometimes useful to concatenate text and numerical elements to a string.

```

paste("part one","part two") # one string

[1] "part one part two"

paste("part one","part two",sep="") # sep=" " is the default

[1] "part onepart two"

paste("part one","part two",sep=" | ")

[1] "part one | part two"

x<-c(3.19,"Method")
x

[1] "3.19" "Method"

paste("The result from the calculaion of",x[2], "=",x[1])

[1] "The result from the calculaion of Method = 3.19"

```

Own experimentation

Try to create vectors and access its elements. You may also try the paste function.

5.7 Data frames

A data frame is a list of variables. However, **list** is a more general object which will be presented later. Each variable is a vector of equal length. The elements of the list are the columns of the data frame. The types of these vectors (the data frame columns) are not required to be the same, though, of course, within the columns each element must be of the same type.

Data frames can access columns using `df$colName` notation and they can also use matrix-style notation. These are both demonstrated below.

Where possible, it is recommended that you use the `df$colName` notation because it is easier to follow in scripts and also if column order is changed, the named notation still works. As you might expect, a data frame can be constructed from a group of equal length vectors.

```
people <- c("Lena", "Solveig", "Anna", "Hans", "Erik" )
byear <- c(1982, 1976, 1949, 2001, 1967)
score <- c( 22, 43, 87, 45, 60)
salary <- c(40, 47, 31, 32, 45)
df <- data.frame(Name = people, BirthYear = byear, Score = score, Salary = salary)
df
```

	Name	BirthYear	Score	Salary
1	Lena	1982	22	40
2	Solveig	1976	43	47
3	Anna	1949	87	31
4	Hans	2001	45	32
5	Erik	1967	60	45

```
df$Sex<-c("woman","woman","woman","man","man") # we can add a variable
df
```

	Name	BirthYear	Score	Salary	Sex
1	Lena	1982	22	40	woman
2	Solveig	1976	43	47	woman
3	Anna	1949	87	31	woman
4	Hans	2001	45	32	man
5	Erik	1967	60	45	man

NOTE that this is a little confusing to look at, because the variables (vectors) have been laid out across the page (rows) and in the dataframe they are placed in columns.

5.7.1 Accessing elements of a data frame

Here are the common ways of accessing elements of the dataframe

```
df$BirthYear[3] # List form - 3rd row of column "BirthYear"
[1] 1949

df$byear        # does not work
NULL

df[2,3]         # Matrix form - 2nd row of 3rd column
[1] 43

df[,3]          # Row number omitted, assume ALL rows i.e. this is equivalent to df$score
```

```
[1] 22 43 87 45 60

df[4:5,1:3]      # Selects the subset - rows 4 and 5, columns 1 to 3

  Name BirthYear Score
4 Hans      2001    45
5 Erik      1967    60
```

Own experimentation

Create a data frame (don't make it complicated, use simple elements like numbers and short text strings) and try to access specific rows or columns from it.

5.8 Tibbles

Tibbles are the tidyverse equivalent (almost) of a dataframe, but they avoid some of the problems - like if you name a large dataframe in the Console, you will fill the pane with numbers. A tibble will only print the first ten row and as many columns as will fit on the screen. The printout shows the variable type, too. If you really want to see all your variables, you can use `print` and set the width to `Inf`.

The datasets available in the core release of R are generally `data.frames`. Dataframes can easily be converted to tibbles and vice versa. Finally a warning - occasionally older R functions won't work with tibbles so you may need to convert your tibble back to a `data.frame`.

```
library(tibble)
df <- as_tibble(df)
class(df)

[1] "tbl_df"      "tbl"        "data.frame"

df <- as.data.frame(df)
class(df)

[1] "data.frame"
```

Chapter 6

Package installation and help resources

6.1 Package installation

Packages in R are modular, small pieces of software that can be loaded on demand. They extend the functionality of R, and there is thousands of packages that greatly enhance R's capabilities. To install a package: In the RStudio menu click on **Tools->Install package**, write '*packagename*' into the empty field and click **Install**. To load a package: Write `library('packagename')` at the beginning of the script.

6.2 Help resources

6.2.1 ?

R and its packages come in general with easily accessible, high quality documentation and help texts on how to use functions. The easiest way to access the help text for a function is to type `?` followed by the name of the function. For example, the help text for the *sum* function:

```
?sum
```

Now, the help text will appear in the help tab on the right hand side (Figure 4.1). The text is structured into the following sections:

Description: Short oral description of what the functions does. Usage: The function and all its arguments in paranthesis. Arguments: A list of the arguments, each described in short. Details: A detailed description of what the function does and the exact behaviour. Value: What the function will return. References: Literature references that describe the methods used in the function. This is mainly relevant for complex statistical functions, such as those that run regressions. See Also: Other functions that are similar and maybe useful. Examples: Some examples on how the function can be used.

The `?` function is easiest place to check on how to use a specific function.

6.2.2 Vignettes

Vignettes are more detailed documents that describe the functionality in packages. Type `browseVignettes()` to get open a list of all installed vignettes.

```
browseVignettes()
```

Vignettes can give a good idea of how the functionality, that is provided by packages, is supposed to be used. For a specific purpose, they describe what functions to use when and in what order.

6.2.3 Internet

The internet is the single most useful source for information on how to work with R. Google can be used to search for specific question or error messages, and chances are that results from Stackoverflow appear prominent in Google's results. Stackoverflow is a website where users asks questions and other users reply to them. Often someone already posted a solution on Stackoverflow for the same problem that might occur for you.

THE CRAN website (<https://cran.r-project.org/>) is the place from where all official R packages are distributed. Useful there may be the *Task Views* that categorize packages according to subjects, and give a short description what a package does.

Some of the bigger packages my have their own website, for example <https://www.tidyverse.org>. A lot of packages nowadays also have a github repository with helpful resource.

Chapter 7

Import and export of data sets

7.1 R data sets

To get a list of all available datasets provided by R, type `data()`. This data objects can be used for your own calculations and examples.

7.2 External data

Here we cover the basic for reading into R and exporting data from R. The most common and basic way to store tabular data in text format are *.csv* files. Each line in a *.csv* file represents a row of data, and single cells are separated by commas. Hence the name csv, comma separated values.

7.2.1 Working directory

At all time, R internally points to a certain directory, called the *working directory*. If one wants to access a file in the filesystem from R, relative Paths start from the working directory. It is convenient to put scripts with R code (ending in *.R*), and data files in the same directory and use that as the working directory. If you don't know what relative paths are and are interested, check out this short video <https://www.youtube.com/watch?v=ephId3mYu9o>.

7.2.2 CSV import

To read a *.csv* file in R use the `read.csv()` function. Here we read the *BackPain.csv* file in the data folder. Make sure that you set R's working directory to the same location as the *BackPain.csv* file. This can be done in RStudio with the Menu: Session *rightarrow* Set Working Directory *rightarrow* To source File Location. The following code reads the *BackPain.csv* and stores the data in the *backpain_data* object.

```
backpain_data <- read.csv("../data/BackPain.csv")
```

Saving tabular data in a *.csv* file works similarly. To save the same data with another filename:

```
write.csv(backpain_data, "../data/tmp_DifferentName.csv")
```

7.2.3 Excel

Its possible to directly work with excel files, but this requires to load an extension, called a *package*. One established package that can read and write xlsx file is called *openxlsx*. After installation, excel files can be accessed. Saving tabular data in a .csv file works similarly. The following code shows how to read and save the same data with another filename using openxlsx:

```
library(openxlsx)

# read first sheet
packpain_data <- read.xlsx("../data/BackPain.xlsx", sheet = 1)
# write again
write.xlsx(x = packpain_data, file = "../data/tmp_DifferentName.xlsx")
```

7.3 Import and export file formats from other statistical software

Its possible to import datafiles from other statistical softwrae packages. One good package to use is *haven*, it may need to be installed. This packages supports for example the *Stata* and *SPSS* formats:

```
library(haven)
aids <- read_dta("../data/Aids.dta") # read stata aids data
write_dta(aids, "../data/tmp_Aids.dta") # write stata

norsjo <- read_sav("../data/norsjo86.sav") # read spss save file
write_sav(norsjo, "../data/tmp_norsjoj86") # write spss save
```

7.4 Import export using RStudio

R Studio provides a way to automatically generate the code for importing data from various file formats via **File->Import Dataset**. One can conveniently use a graphical file browser to go to select the file and import the data directly into the working environment. It works by pasting the code to do that into the console and executing it.

Own experimentation

What are the advantages uf using the R Studio menu, what are the disadvantages? If you publish an R script as part of a scientific article, Would you chose to include the code for importing the data as part of the script?

7.5 R binary files

Sometimes it is convenient to store R objects directly in binary formmat, without using one of the text-based formats such as *csv*. In binary means that the file format can not be editet with text editors and is harder to read for humans. Instead, the data is stored in a format that is related to how the data looks internally in the computer memory.

R provides several ways of doing that. One way is using `save()`. This function stores the specified objects in an *.RData* file.


```

data <- cars
var1 <- 42

# objects
save(data, var1, file = "../data/tmp_current_environment.RData")

# specify list of object character names
save(list = c("data", "var1"), file = "../data/tmp_current_environment.RData")

# directly specify the objects
save(data, var1, file = "../data/tmp_current_environment.RData")

```

The counterpart to `save()` is `load()`. It can be used to load the `.RData` files back into R. Note that all the objects in such a workspace image file will automatically be added as objects to the current working environment.

```
load(file = "../data/tmp_current_environment.RData")
```

Another way to load and save data is using the `(saveRDS())` and `readRDS()` function. They operate on single objects, meaning that they save and read single R objects:

```

# save and load single objects
saveRDS(mtcars, file = "../data/tmp_data.rds")
readRDS(file = "../data/tmp_data.rds")

##           mpg  cyl  disp  hp drat   wt  qsec vs  am  gear carb
## Mazda RX4      21.0    6 160.0 110 3.90 2.620 16.46 0   1    4    4
## Mazda RX4 Wag  21.0    6 160.0 110 3.90 2.875 17.02 0   1    4    4
## Datsun 710     22.8    4 108.0  93 3.85 2.320 18.61 1   1    4    1
## Hornet 4 Drive  21.4    6 258.0 110 3.08 3.215 19.44 1   0    3    1
## Hornet Sportabout 18.7    8 360.0 175 3.15 3.440 17.02 0   0    3    2
## Valiant        18.1    6 225.0 105 2.76 3.460 20.22 1   0    3    1
## Duster 360     14.3    8 360.0 245 3.21 3.570 15.84 0   0    3    4
## Merc 240D      24.4    4 146.7  62 3.69 3.190 20.00 1   0    4    2
## Merc 230       22.8    4 140.8  95 3.92 3.150 22.90 1   0    4    2
## Merc 280       19.2    6 167.6 123 3.92 3.440 18.30 1   0    4    4
## Merc 280C      17.8    6 167.6 123 3.92 3.440 18.90 1   0    4    4
## Merc 450SE     16.4    8 275.8 180 3.07 4.070 17.40 0   0    3    3
## Merc 450SL     17.3    8 275.8 180 3.07 3.730 17.60 0   0    3    3
## Merc 450SLC    15.2    8 275.8 180 3.07 3.780 18.00 0   0    3    3
## Cadillac Fleetwood 10.4    8 472.0 205 2.93 5.250 17.98 0   0    3    4
## Lincoln Continental 10.4    8 460.0 215 3.00 5.424 17.82 0   0    3    4
## Chrysler Imperial 14.7    8 440.0 230 3.23 5.345 17.42 0   0    3    4
## Fiat 128       32.4    4  78.7  66 4.08 2.200 19.47 1   1    4    1
## Honda Civic    30.4    4  75.7  52 4.93 1.615 18.52 1   1    4    2
## Toyota Corolla 33.9    4  71.1  65 4.22 1.835 19.90 1   1    4    1
## Toyota Corona  21.5    4 120.1  97 3.70 2.465 20.01 1   0    3    1
## Dodge Challenger 15.5    8 318.0 150 2.76 3.520 16.87 0   0    3    2
## AMC Javelin    15.2    8 304.0 150 3.15 3.435 17.30 0   0    3    2
## Camaro Z28     13.3    8 350.0 245 3.73 3.840 15.41 0   0    3    4
## Pontiac Firebird 19.2    8 400.0 175 3.08 3.845 17.05 0   0    3    2
## Fiat X1-9      27.3    4  79.0  66 4.08 1.935 18.90 1   1    4    1
## Porsche 914-2  26.0    4 120.3  91 4.43 2.140 16.70 0   1    5    2
## Lotus Europa   30.4    4  95.1 113 3.77 1.513 16.90 1   1    5    2
## Ford Pantera L 15.8    8 351.0 264 4.22 3.170 14.50 0   1    5    4
## Ferrari Dino   19.7    6 145.0 175 3.62 2.770 15.50 0   1    5    6
## Maserati Bora   15.0    8 301.0 335 3.54 3.570 14.60 0   1    5    8
## Volvo 142E     21.4    4 121.0 109 4.11 2.780 18.60 1   1    4    2

```

This way offers some advantage over `save()` and `load()`. Associating object explicitly to filenames makes scripts easier to understand, write, and enhances reproducibility. When saving and loading whole working environments, it may not be clear what data has been loaded with the working environment and what data has been loaded prior to that using other means.

Chapter 8

R objects - part 2

Håkan Jonsson < hakan.jonsson@umu.se >

8.1 Factors

Factors are "vectors" of categorical data. In practice categorical usually means that there are more than one observation of each category. However, since a R vector is defined so that all elements are of the same type and this is a more general type so **a factor is not a vector**. A factor also includes more information than a vector. Most important are the levels. It can also include labels. There is a function **is.factor** for checking if the object is a factor or not. It returns TRUE or FALSE.

```
x<-c(1,2,3,4,1,3)
x

[1] 1 2 3 4 1 3

str(x)

num [1:6] 1 2 3 4 1 3

y<-as.factor(x)
str(y)

Factor w/ 4 levels "1","2","3","4": 1 2 3 4 1 3

is.vector(x)

[1] TRUE

is.vector(y)

[1] FALSE

x<-factor(c("f","b","c","f","c","b","b")) # levels are stored in alphabetical order
x

[1] f b c f c b b
Levels: b c f

x<-factor(c(1,7.1,"a","Low",14,1,"a","high","low",14,1,"Low"))
x
```

```
[1] 1      7.1 a      Low 14    1      a      high low 14    1      Low
Levels: 1 14 7.1 a high low Low

levels(x)      # if we only want the levels

[1] "1"      "14"     "7.1"    "a"      "high"   "low"    "Low"

x[c(2,4)]      # all levels are printed

[1] 7.1 Low
Levels: 1 14 7.1 a high low Low
```

The levels of factors are by default listed alphabetically (numbers are included and come before letters). Sometimes the first level is used as a reference. We can change it.

```
y<-relevel(x,ref="Low")
y

[1] 1      7.1 a      Low 14    1      a      high low 14    1      Low
Levels: Low 1 14 7.1 a high low
```

Own experimentation

Create one or two own factors. Choose only few different values which are repeated several times.

8.2 Lists

A list is a general object which include e.g. data frame. It is a common result object from many functions.

```
x1<-data.frame(measurement=c(12.7,6.4,5.7,14,2.5),
               gender=factor(c("male","male","female","male","female")))
x2<-c(2.5,14)
x3<-5
x4<- "This is the minimum and maximum values"

x1

  measurement gender
1         12.7   male
2          6.4   male
3          5.7 female
4         14.0   male
5          2.5 female

str(x1)

'data.frame': 5 obs. of 2 variables:
 $ measurement: num 12.7 6.4 5.7 14 2.5
 $ gender      : Factor w/ 2 levels "female","male": 2 2 1 2 1

is.list(x1)

[1] TRUE
```

```

y<-list(what=x4,min.max=x2,N=x3)
y

$what
[1] "This is the minimum and maximum values"

$min.max
[1] 2.5 14.0

$N
[1] 5

str(y)

List of 3
 $ what : chr "This is the minimum and maximum values"
 $ min.max: num [1:2] 2.5 14
 $ N      : num 5

y$N
[1] 5

y[[2]]
[1] 2.5 14.0

z<-list(oldlist=y,data=x1)
str(z)

List of 2
 $ oldlist:List of 3
   ..$ what : chr "This is the minimum and maximum values"
   ..$ min.max: num [1:2] 2.5 14
   ..$ N      : num 5
 $ data : 'data.frame': 5 obs. of 2 variables:
   ..$ measurement: num [1:5] 12.7 6.4 5.7 14 2.5
   ..$ gender : Factor w/ 2 levels "female","male": 2 2 1 2 1

z

$oldlist
$oldlist$what
[1] "This is the minimum and maximum values"

$oldlist$min.max
[1] 2.5 14.0

$oldlist$N
[1] 5

$data
 measurement gender
1          12.7   male
2           6.4   male
3           5.7 female
4          14.0   male
5           2.5 female

z2<-c(y,data=list(x1))
str(z2)

```

```

List of 4
 $ what   : chr "This is the minimum and maximum values"
 $ min.max: num [1:2] 2.5 14
 $ N      : num 5
 $ data   :'data.frame': 5 obs. of  2 variables:
  ..$ measurement: num [1:5] 12.7 6.4 5.7 14 2.5
  ..$ gender      : Factor w/ 2 levels "female","male": 2 2 1 2 1

z
$oldlist
$oldlist$what
[1] "This is the minimum and maximum values"

$oldlist$min.max
[1] 2.5 14.0

$oldlist$N
[1] 5

$data
  measurement gender
1         12.7   male
2          6.4   male
3          5.7 female
4         14.0   male
5          2.5 female

z$oldlist
$what
[1] "This is the minimum and maximum values"

$min.max
[1] 2.5 14.0

$N
[1] 5

z[[1]] # same as z$oldlist
$what
[1] "This is the minimum and maximum values"

$min.max
[1] 2.5 14.0

$N
[1] 5

z$oldlist[2]
$min.max
[1] 2.5 14.0

z[[1]][2]
$min.max
[1] 2.5 14.0

z[[2]][,2]
[1] male  male  female male  female
Levels: female male

```

Own experimentation

Play around a little by creating lists. Try to include different elements like; vectors, text string, number or even data frames.

8.3 Functions

We have seen that we can generate objects like vectors, data frames and lists. Other important objects are functions which are not used for storage of data but for e.g calculations or plots. Even if we can create our own functions, which we will look at later, most common is to use the large numbers of functions available in R. Functions (and sometimes data sets) are stored in **packages**. Some of these are available per default in core R.

We have already seen some examples of functions. Some more examples of functions are shown below.

```
x<-c(1,4,9,25,10,2)
length(x)

[1] 6

sqrt(x)

[1] 1.000000 2.000000 3.000000 5.000000 3.162278 1.414214

round(sqrt(x),2)

[1] 1.00 2.00 3.00 5.00 3.16 1.41

sort(x)

[1] 1 2 4 9 10 25

log(x)

[1] 0.0000000 1.3862944 2.1972246 3.2188758 2.3025851 0.6931472

sample(x,3)

[1] 25 2 9

sample(x,10) # does not work because default is without replacement

Error in sample.int(length(x), size, replace, prob): cannot take a sample larger than the
population when 'replace = FALSE'

sample(x,10,replace=TRUE) # but this works

[1] 1 4 4 10 10 10 10 9 4 2
```

We have already seen that functions may be generic, i.e. different function depending on the class of arguments. Later on we will learn how to make own functions.

Own experimentation

Try the presented functions using own data. There are also alternatives to the function **round** if you want e.g. the number 2.6 to be 2 or 2.4 to be 3. Can you find them by looking at help for round?

8.3.1 Installation of a package

To install a package the easiest way is to use one of the menus in RStudio [Tools->Install Packages] above the **Edit** pane or [Packages->Install] in the **Help** pane.

8.3.2 Loading a package

If a package is not available per default it has to be installed. This means that the files are downloaded to the computer. However, to make them accessible in R you need to use the **library** function to **load** the package. The installation only needs to be made once but the loading has to be made for every new R session.

The installed packages can be found in RStudio in the **Help** pane. Click on the package name and you will get a description.

```
options()$defaultPackages

[1] "datasets" "utils" "grDevices" "graphics" "stats" "methods"

search()

[1] ".GlobalEnv" "package:boot" "package:survival" "package:readxl"
[5] "package:Epi" "package:janitor" "package:data.table" "package:forcats"
[9] "package:stringr" "package:purrr" "package:readr" "package:tidyr"
[13] "package:tibble" "package:tidyverse" "package:labelled" "package:gridExtra"
[17] "package:dplyr" "package:haven" "package:openxlsx" "package:ggplot2"
[21] "package:cowplot" "package:knitr" "package:htmlTable" "tools:rstudio"
[25] "package:stats" "package:graphics" "package:grDevices" "package:utils"
[29] "package:datasets" "package:methods" "Autoloads" "package:base"

library(Epi)
search()

[1] ".GlobalEnv" "package:boot" "package:survival" "package:readxl"
[5] "package:Epi" "package:janitor" "package:data.table" "package:forcats"
[9] "package:stringr" "package:purrr" "package:readr" "package:tidyr"
[13] "package:tibble" "package:tidyverse" "package:labelled" "package:gridExtra"
[17] "package:dplyr" "package:haven" "package:openxlsx" "package:ggplot2"
[21] "package:cowplot" "package:knitr" "package:htmlTable" "tools:rstudio"
[25] "package:stats" "package:graphics" "package:grDevices" "package:utils"
[29] "package:datasets" "package:methods" "Autoloads" "package:base"

detach("package:Epi")
search()

[1] ".GlobalEnv" "package:boot" "package:survival" "package:readxl"
[5] "package:janitor" "package:data.table" "package:forcats" "package:stringr"
[9] "package:purrr" "package:readr" "package:tidyr" "package:tibble"
[13] "package:tidyverse" "package:labelled" "package:gridExtra" "package:dplyr"
[17] "package:haven" "package:openxlsx" "package:ggplot2" "package:cowplot"
[21] "package:knitr" "package:htmlTable" "tools:rstudio" "package:stats"
[25] "package:graphics" "package:grDevices" "package:utils" "package:datasets"
[29] "package:methods" "Autoloads" "package:base"
```


8.3.3 Masking

Because R is developed by an open source community, it is not uncommon that multiple packages may use the same name for a function or dataset. If you load packages that use the same name for an object, R will warn that certain object(s) have been "masked".

If we look at help for the function **search** we can see that it belongs to the **base** package. If there is another package that masks the function **search** we can still use the function.

```
# ?search is used to see the help
```

```
base::search()
```

[1] ".GlobalEnv"	"package:Epi"	"package:boot"	"package:survival"
[5] "package:readxl"	"package:janitor"	"package:data.table"	"package:forcats"
[9] "package:stringr"	"package:purrr"	"package:readr"	"package:tidyr"
[13] "package:tibble"	"package:tidyverse"	"package:labelled"	"package:gridExtra"
[17] "package:dplyr"	"package:haven"	"package:openxlsx"	"package:ggplot2"
[21] "package:cowplot"	"package:knitr"	"package:htmlTable"	"tools:rstudio"
[25] "package:stats"	"package:graphics"	"package:grDevices"	"package:utils"
[29] "package:datasets"	"package:methods"	"Autoloads"	"package:base"

Chapter 9

Some useful functions

Håkan Jonsson < hakan.jonsson@umu.se >

9.1 Functions in R

Functions are always followed by brackets () and within these they usually have a number of arguments. Some arguments may have a default value and usually only one or two arguments are needed. So mostly you don't have to care about the optional arguments. You can however find them using the help function e.g. `?sum` which gives information on the arguments, the value returned from the function as well as examples and references. This help exists for all objects.

9.2 Some basic functions on numerical vectors

We start by looking at **sum**, **mean** and **max**. It is obvious from the names what the functions do. There are also related functions like **min**, **sd**, **var**, **cumsum**, **range** which we don't exemplify here. They can be found in the help or simply by google e.g. "standard deviation r".

```
x<-c(1,3,5,7,7,5)
x

[1] 1 3 5 7 7 5

sum(x)

[1] 28

mean(x)

[1] 4.666667

max(x)

[1] 7
```

Own experimentation

First generate another vector than **x** above. Then try the functions above on the new vector. Also try the **cumsum** and **min** functions. Are there any important arguments for any of these functions? Try to find the functions for standard deviation (S) and variance (V). Calculate S and V for a vector e.g. **x** above and compare the result of the two measures by using either **sqrt** or \wedge . $S = \sqrt{V}$ What happens if you forget the () when you use a function?

9.3 Generic functions

In R there are some generic functions where the **class** of the object determines what the function will do. The function **summary()** is an example of a such generic function. It also works in a similar way on data frames. The result from **summary** include the functions **min**, **max**, **median**, **mean** but is also include the function **quantile**. The result of quantile is not a single number but a vector, by default it is the limits when the data is partitioned in 4 equal parts according to the values. It is not obvious how to calculate it, therefore there are several methods to chose among. Examples of the use of e.g. **summary** will follow later.

```
people <- c('Lena', 'Solveig', 'Anna', 'Hans', 'Erik' )
byear <- c(1982, 1976, 1949, 2001, 1967)
score <- c( 22, 43, 87, 45, 60)
salary <- c(40, 47, 31, 32, 45)
df <- data.frame(Name = people, BirthYear = byear, Score = score, Salary = salary)
df
```

	Name	BirthYear	Score	Salary
1	Lena	1982	22	40
2	Solveig	1976	43	47
3	Anna	1949	87	31
4	Hans	2001	45	32
5	Erik	1967	60	45

```
summary(df)
```

	Name	BirthYear	Score	Salary
Anna	:1	Min. :1949	Min. :22.0	Min. :31
Erik	:1	1st Qu.:1967	1st Qu.:43.0	1st Qu.:32
Hans	:1	Median :1976	Median :45.0	Median :40
Lena	:1	Mean :1975	Mean :51.4	Mean :39
Solveig	:1	3rd Qu.:1982	3rd Qu.:60.0	3rd Qu.:45
		Max. :2001	Max. :87.0	Max. :47

9.4 Some functions for investigation of data frames

Here are a few of core R's functions to investigate dataframe structure. Summary is useful but it is already demonstrated above.

```
names(df)

[1] "Name"      "BirthYear" "Score"      "Salary"

names(df)[3]
```

```
[1] "Score"

ncol(df)

[1] 4

nrow(df) # length() for a vector

[1] 5

dim(df)

[1] 5 4

str(df)

'data.frame': 5 obs. of 4 variables:
 $ Name      : Factor w/ 5 levels "Anna","Erik",...: 4 5 1 3 2
 $ BirthYear: num  1982 1976 1949 2001 1967
 $ Score     : num  22 43 87 45 60
 $ Salary    : num  40 47 31 32 45

str(people)

chr [1:5] "Lena" "Solveig" "Anna" "Hans" "Erik"

class(df)

[1] "data.frame"
```

Notice that the character vector 'people' has become a factor when incorporated into a dataframe.

Own experimentation

Check if the result of **summary** is correct by using the functions **min**, **max**, **mean**, **median** and **quantile** on each vector e.g. `mean(df$Salary)`. The function **dim** results in two numbers. What kind of object is it?

9.5 Some more useful functions for data frames

It is sometimes useful to add a data frame to the rows or columns of another data frame. Observe that for **cbind** the number of rows must agree and for **rbind** the number of columns and column names must agree.

```
a<-c(37,35,62,45,70)
h<-c(170,162,166,174,192)
df2<-data.frame(Age=a,Heigt=h)
df2
```

	Age	Heigt
1	37	170
2	35	162
3	62	166
4	45	174
5	70	192

```
dfw<-cbind(df,df2) # add the data frames by column
dfw
```

	Name	BirthYear	Score	Salary	Age	Heigt
1	Lena	1982	22	40	37	170
2	Solveig	1976	43	47	35	162
3	Anna	1949	87	31	62	166
4	Hans	2001	45	32	45	174
5	Erik	1967	60	45	70	192

```
dfw2<-dfw[c(2,4),]
dfw2
```

	Name	BirthYear	Score	Salary	Age	Heigt
2	Solveig	1976	43	47	35	162
4	Hans	2001	45	32	45	174

```
df.dupl<-rbind(dfw,dfw2) # add the data frames by row
df.dupl
```

	Name	BirthYear	Score	Salary	Age	Heigt
1	Lena	1982	22	40	37	170
2	Solveig	1976	43	47	35	162
3	Anna	1949	87	31	62	166
4	Hans	2001	45	32	45	174
5	Erik	1967	60	45	70	192
21	Solveig	1976	43	47	35	162
41	Hans	2001	45	32	45	174

Later on we will be looking at the tidyverse package dplyr. There are functions **bind_rows** and **bind_cols** similar to the **rbind** and **cbind** above. Another function in dplyr is shown below. It is useful for removing duplicated rows.

```
df.dupl
```

	Name	BirthYear	Score	Salary	Age	Heigt
1	Lena	1982	22	40	37	170
2	Solveig	1976	43	47	35	162
3	Anna	1949	87	31	62	166
4	Hans	2001	45	32	45	174
5	Erik	1967	60	45	70	192
21	Solveig	1976	43	47	35	162
41	Hans	2001	45	32	45	174

```
library(dplyr)
distinct(df.dupl)
```

	Name	BirthYear	Score	Salary	Age	Heigt
1	Lena	1982	22	40	37	170
2	Solveig	1976	43	47	35	162
3	Anna	1949	87	31	62	166
4	Hans	2001	45	32	45	174
5	Erik	1967	60	45	70	192

Own experimentation

Try the functions **cbind** and **rbind**. You may use **df** in the example above and create sub-data.frames that you add to the original **df**.

9.6 Systematic generation of data

We here show two functions **rep** and **seq** useful for generation of data. They can also be combined in several ways.

```
rep(2,6)
[1] 2 2 2 2 2 2

rep(c(1,5,7),times=3)
[1] 1 5 7 1 5 7 1 5 7

rep(c(1,5,7),each=3)
[1] 1 1 1 5 5 5 7 7 7

rep(c(1,5,7),times=2,each=3)
[1] 1 1 1 5 5 5 7 7 7 1 1 1 5 5 5 7 7 7

1:5
[1] 1 2 3 4 5

seq(from=1,to=5)
[1] 1 2 3 4 5

seq(1,5,by=1)
[1] 1 2 3 4 5

seq(1,5,by=0.5)
[1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0

seq(1,8,by=3)
[1] 1 4 7

seq(4,by=3,length.out=6)
[1] 4 7 10 13 16 19

#
#rep is general. Also characters and strings can be used
rep(c(412,"c","May is a nice month"),times=2)

[1] "412" "c" "May is a nice month"
[4] "412" "c" "May is a nice month"
```

Own experimentation

Try out the functions in different ways. Suggestion: try to use **rep** to generate the vectors below.

```
[1] "vector 1"
[1] 8 9 8 9 8 9 8 9 8 9
[1] "vector 2"
[1] 8 8 8 9 9 9 8 8 8 9 9 9
[1] "vector 3"
[1] 8 8 9 9 8 8 9 9 8 8 9 9
[1] "vector 4"
[1] "2"      "monkey" "Lemon"  "2"      "monkey" "Lemon"
[1] "vector 5"
[1] "2"      "2"      "monkey" "monkey" "Lemon"  "Lemon"
[1] "vector 6"
[1] "2"      "2"      "monkey" "monkey" "Lemon"  "Lemon"  "2"      "2"      "monkey"
[10] "monkey" "Lemon"  "Lemon"
[1] "vector 7"
[1] "2"      "2"      "monkey" "monkey" "Lemon"  "Lemon"  "2"      "2"      "monkey"
```

Now try to use **seq** to generate the vectors below.

```
[1] "vector 1"
[1] 11 12 13 14
[1] "vector 2"
[1] 7 10 13 16 19
[1] "vector 3; the odd numbers up to 39"
[1] 1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39
```

9.7 Sorting

Let us first generate a data frame. The function **sort** sorts a vector by default ascending but it can be changed. To sort a data frame due to one column (vector) the easiest way is to give the ordered row numbers as a vector (see the example). The function **order** gives the row numbers of the original vector sorted. This can be generalised to sort data frames due to more than one column.

```
df<-data.frame(var1=c(1,3,5,3,5,7),var2=rep(c(2,5,3)),var3=rep(c("A","O"),3),
  var4=rep(c("a",4,"Lars")),var5=seq(10,7,length.out=6))
```

```
df
```

```
  var1 var2 var3 var4 var5
1    1    2    A    a 10.0
2    3    5    O    4  9.4
3    5    3    A  Lars  8.8
4    3    2    O    a  8.2
5    5    5    A    4  7.6
6    7    3    O  Lars  7.0
```

```
sort(df$var2)                                # Sorting
```

```
[1] 2 2 3 3 5 5
```

```
sort(df$var2,decreasing=T)                  #change to decreasing order
```

```
[1] 5 5 3 3 2 2
```



```
df
  var1 var2 var3 var4 var5
1    1    2    A    a 10.0
2    3    5    0    4  9.4
3    5    3    A Lars  8.8
4    3    2    0    a  8.2
5    5    5    A    4  7.6
6    7    3    0 Lars  7.0

df[c(1,4,3,6,2,5),] # manual sorting by var2

  var1 var2 var3 var4 var5
1    1    2    A    a 10.0
4    3    2    0    a  8.2
3    5    3    A Lars  8.8
6    7    3    0 Lars  7.0
2    3    5    0    4  9.4
5    5    5    A    4  7.6

df[c(1,4,3,6),] # combined sorting and selection of the the first three rows

  var1 var2 var3 var4 var5
1    1    2    A    a 10.0
4    3    2    0    a  8.2
3    5    3    A Lars  8.8
6    7    3    0 Lars  7.0

df[c(1,4,3,6,F,F),] # is actually the same

  var1 var2 var3 var4 var5
1    1    2    A    a 10.0
4    3    2    0    a  8.2
3    5    3    A Lars  8.8
6    7    3    0 Lars  7.0

df$var2
[1] 2 5 3 2 5 3

order(df$var2) # Sorting using positions(row numbers) of var2.
[1] 1 4 3 6 2 5

# It is the same vector as we used for sorting above.
df[order(df$var2),] # so therefore this will also be the dataframe sorted by var2

  var1 var2 var3 var4 var5
1    1    2    A    a 10.0
4    3    2    0    a  8.2
3    5    3    A Lars  8.8
6    7    3    0 Lars  7.0
2    3    5    0    4  9.4
5    5    5    A    4  7.6

df[order(df$var2,decreasing=T),] # same but decreasing

  var1 var2 var3 var4 var5
2    3    5    0    4  9.4
5    5    5    A    4  7.6
3    5    3    A Lars  8.8
6    7    3    0 Lars  7.0
1    1    2    A    a 10.0
4    3    2    0    a  8.2
```

```
df[order(df$var3),] # we can also sort on characters or factors
```

```
  var1 var2 var3 var4 var5
1     1     2    A    a 10.0
3     5     3    A Lars  8.8
5     5     5    A    4  7.6
2     3     5    0    4  9.4
4     3     2    0    a  8.2
6     7     3    0 Lars  7.0
```

```
df[order(df$var3,df$var2),] # sorted by var3 and then var2
```

```
  var1 var2 var3 var4 var5
1     1     2    A    a 10.0
3     5     3    A Lars  8.8
5     5     5    A    4  7.6
4     3     2    0    a  8.2
6     7     3    0 Lars  7.0
2     3     5    0    4  9.4
```

Own experimentation

Suggestion: for example you can try to sort the data frame above by var1, by var5, by var5 and var1, by var1(increasing) and var2(decreasing). You can also sort it in the order 1,6,2,5,3,4 of the rows in df.

9.8 Filtering subsets of data frames

By giving a logical expression in the row we can get a subset of a vector or data frame. However, we have to watch out for missing data which can give an incorrect result. There is a simpler method for this, the **subset** function. As an example, assume we want to select females below the age of 40 with a height over 175.

```
df<-data.frame(sex=factor(c("male","male","female","female","female","female")),
               age=c(25,42,38,39,44,24),height=c(172,184,177,152,171,180))
df
```

```
  sex age height
1 male  25   172
2 male  42   184
3 female 38   177
4 female 39   152
5 female 44   171
6 female 24   180
```

```
df[df$sex=="female" & df$age<40 & df$height>175,]
```

```
  sex age height
3 female 38   177
6 female 24   180
```

```
dataframe.with.a.long.name<-df
dataframe.with.a.long.name[dataframe.with.a.long.name$sex=="female"
                             & dataframe.with.a.long.name$age<40
                             & dataframe.with.a.long.name$height>175,]
```

```

      sex age height
3 female  38    177
6 female  24    180

subset(dataframe.with.a.long.name,sex=="female" & age<40 & height>175)

      sex age height
3 female  38    177
6 female  24    180

```

Let us have a look at the dataframe in the above exmple. The selection of female (`df$sex=="female"`) actually gives a logic vector. Which means that we select the rows with result equal to TRUE. We can see that the only rows whith all three conditions TRUE are the 3rd and 6th row which were selected in the former example.

```

df<-data.frame(sex=factor(c("male","male","female","female","female","female")),
               age=c(25,42,38,39,44,24),height=c(172,184,177,152,171,180))
df

      sex age height
1  male  25    172
2  male  42    184
3 female  38    177
4 female  39    152
5 female  44    171
6 female  24    180

v1<-df$sex=="female"
v2<-df$age<40
v3<-df$height>175
data.frame(v1,v2,v3)

   v1   v2   v3
1 FALSE TRUE FALSE
2 FALSE FALSE  TRUE
3  TRUE  TRUE  TRUE
4  TRUE  TRUE FALSE
5  TRUE FALSE FALSE
6  TRUE  TRUE  TRUE

vall<-v1 & v2 & v3
vall

[1] FALSE FALSE  TRUE FALSE FALSE  TRUE

df[vall,]

      sex age height
3 female  38    177
6 female  24    180

```

9.9 Missing data

Now what happens if there is a missing in any of the variables sex, age and height? A missing appears as a **NA** (not available). We need to be somewhat careful when handling missing values. There are some functions to help with this.

```

df.missing<-data.frame(sex=factor(c("male","male",NA,"female","female","female")),
                       age=c(25,42,38,39,NA,24),height=c(172,NA,177,152,NA,180))
df.missing
      sex age height
1  male  25   172
2  male  42    NA
3  <NA> 38   177
4 female 39   152
5 female NA    NA
6 female 24   180

# we now filter the data and get
df.missing[df.missing$sex=="female" & df.missing$age<40 & df.missing$height>175,]
      sex age height
NA    <NA>  NA    NA
NA.1  <NA>  NA    NA
6  female 24   180

df.missing[6,] # this is probably what we actually wanted
      sex age height
6 female 24   180

df.missing$height
[1] 172  NA 177 152  NA 180

df.missing$sex # missing is not seen as a factor level
[1] male  male  <NA>  female female female
Levels: female male

# WARNING in the following example NA is interpreted as a text category
factor(c("male","male","NA","female","female","female"))
[1] male  male  NA    female female female
Levels: female male NA

v1<-df.missing$sex=="female" # check the three conditions
v2<-df.missing$age<40
v3<-df.missing$height>175
vall<-v1 & v2 & v3
data.frame(sex=v1,age=v2,height=v3,total=vall) # vall = all conditions fulfilled
      sex  age height total
1 FALSE TRUE  FALSE FALSE
2 FALSE FALSE    NA FALSE
3  NA TRUE  TRUE  NA
4 TRUE TRUE  FALSE FALSE
5 TRUE  NA    NA  NA
6 TRUE TRUE  TRUE  TRUE

df.missing[vall,] # the same as we got above
      sex age height
NA    <NA>  NA    NA
NA.1  <NA>  NA    NA
6  female 24   180

# also rows where the logical expression cannot be evaluated are included
nrow(df.missing[vall,])
[1] 3

```

Let us for a moment only focus on the selection on sex.

```
is.na(df.missing$sex)  # check if missing

[1] FALSE FALSE  TRUE FALSE FALSE FALSE

!is.na(df.missing$sex) # not missing

[1]  TRUE  TRUE FALSE  TRUE  TRUE  TRUE

df.missing[df.missing$sex=="female" ,]

      sex age height
NA    <NA>  NA     NA
4 female  39    152
5 female  NA     NA
6 female  24    180

df.missing[df.missing$sex=="female" & !is.na(df.missing$sex) ,] # this is better,

      sex age height
4 female  39    152
5 female  NA     NA
6 female  24    180

# To make a correct selection to avoid problems with missing data
df.missing[df.missing$sex=="female" & !is.na(df.missing$sex) & df.missing$age<40 &
           !is.na(df.missing$age)& df.missing$height>175& !is.na(df.missing$height),]

      sex age height
6 female  24    180
```

9.9.1 Some functions when dealing with missing data

The functions **subset** and **na.omit** may be useful.

```
df.missing

      sex age height
1  male  25    172
2  male  42     NA
3  <NA>  38    177
4 female  39    152
5 female  NA     NA
6 female  24    180

# This is a function that make things easier
# Here is a difference - missing values are taken as false
subset(df.missing,sex=="female" & age<40 & height>175)

      sex age height
6 female  24    180

# You may want to remove all missing, so called removing listwise
df.clean<-na.omit(df.missing)
df.clean

      sex age height
```

```

1   male  25   172
4  female 39   152
6  female 24   180

df.clean[df.clean$sex=="female" & df.clean$age<40 & df.clean$height>175,]

      sex age height
6  female  24   180

```

9.9.2 Arguments in functions to handle missing data

The **na.rm** is an argument you can find in many functions. Another alternative is **use** (used in `var()` and `cor()`). It is a good idea to check the help.

```

#
x<-c(1,3,5,NA,7,5,14,NA,6)
x

[1]  1  3  5 NA  7  5 14 NA  6

sum(x) #does not work

[1] NA

sum(x,na.rm=T) # this works

[1] 41

mean(x)

[1] NA

mean(x,na.rm=T)

[1] 5.857143

set.seed(221)
df<-data.frame(size=sample(c("big","small"),size=20,replace=T),time=sample(c("before","middle","after"),size=20,replace=T))
dfm<-df
dfm[c(6,16),1]<-NA
dfm[c(4),2]<-NA
dfm          # dfm has 3 missing

      size  time
1  small middle
2  small before
3    big  after
4    big  <NA>
5    big  after
6  <NA> middle
7    big before
8  small middle
9  small middle
10   big middle
11   big before
12   big  after
13 small  after
14 small  after

```

```

15  big before
16  <NA> after
17  big middle
18  small middle
19  small middle
20  big middle

table(df$time,df$size)

      big small
after   4     2
before  3     1
middle  5     5

table(dfm$time,dfm$size) # we cannot see in the table that we have missing data

      big small
after   3     2
before  3     1
middle  3     5

table(dfm$time,dfm$size,useNA = "ifany")

      big small <NA>
after   3     2    1
before  3     1    0
middle  3     5    1
<NA>    1     0    0

xtabs(~time+size,data=dfm,addNA = T)

      size
time   big small <NA>
after   3     2    1
before  3     1    0
middle  3     5    1
<NA>    1     0    0

```

9.10 The function cut

The function cut is used to split a numerical continuous variable into categories.

```

x<-seq(0,10)
x

[1] 0 1 2 3 4 5 6 7 8 9 10

gr<-cut(x,breaks=c(0,2,5,Inf))

gr

[1] <NA> (0,2] (0,2] (2,5] (2,5] (2,5] (5,Inf] (5,Inf] (5,Inf] (5,Inf]
[11] (5,Inf]
Levels: (0,2] (2,5] (5,Inf]

class(gr)

[1] "factor"

```

```
summary(gr)      # similar as table() for factors

(0,2]  (2,5] (5,Inf]  NA's
    2      3      5      1

gr.lab<-cut(x,breaks=c(0,2,5,Inf),labels=c("small","medium","large"))
summary(gr.lab)

small medium large  NA's
    2      3      5      1

table(gr.lab,useNA="ifany") # alt for table

gr.lab
small medium large <NA>
    2      3      5      1

xtabs(~gr.lab,addNA=T)      # alt for xtabs

gr.lab
small medium large <NA>
    2      3      5      1
```

Observe e.g. (2,5] means that the intervals are open to the left (2 is not included but 5 is). This is possible to change. See help for info (cut(right))

Own experimentation

Try to change the cut-off points and the values in the original vector x.
Why is there one missing (NA) in the variable gr above? Can you make so all values from 0 to 2 belong to a category?

9.11 The function apply

Apply is a function for calculations made by rows or by columns in a data frame. Calculations by row is not always meaningful.

```
x<-data.frame(bmi.gr1=c(21.6,24.3,29.2,26.1,23.6),bmi.gr2=c(26.7,28.3,25.2,28.1,27.1),bmi.gr3=c(24.3,23.2,24.8,23.5,25.0))
x
  bmi.gr1 bmi.gr2 bmi.gr3
1   21.6   26.7   24.3
2   24.3   28.3   23.2
3   29.2   25.2   24.8
4   26.1   28.1   23.5
5   23.6   27.1   25.0

apply(x,1,mean)      # mean by row
[1] 24.20000 25.26667 26.40000 25.90000 25.23333

apply(x,2,mean)      # mean by column
bmi.gr1 bmi.gr2 bmi.gr3
  24.96   27.08   24.16

apply(x,2,sd)
```



```

bmi.gr1  bmi.gr2  bmi.gr3
2.8658332 1.2457929 0.7893035

apply(x,2,sort)  # the function does not need to produce one numeric result

bmi.gr1 bmi.gr2 bmi.gr3
[1,]    21.6    25.2    23.2
[2,]    23.6    26.7    23.5
[3,]    24.3    27.1    24.3
[4,]    26.1    28.1    24.8
[5,]    29.2    28.3    25.0

```

9.12 Extension of apply to tapply

```

library(haven)
norsjo86 <- read_sav("../data/norsjo86.sav")
norsjo86

# A tibble: 260 x 10
   agegrp health sex height weight sbp dbp kolester smoker bmi
   <dbl+lbl> <dbl+lbl> <dbl+lb> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl+lbl> <dbl>
1 60 [60 year~ 0 [good] 2 [Woma~ 157 61 110 70 6.7 0 [Non-s~ 24.7
2 60 [60 year~ 1 [not quite ~ 2 [Woma~ 157 97 150 100 6.6 0 [Non-s~ 39.4
3 60 [60 year~ 0 [good] 1 [Man] 170 74 136 96 8.2 0 [Non-s~ 25.6
4 60 [60 year~ 0 [good] 2 [Woma~ 163 66 156 76 7.5 0 [Non-s~ 24.8
5 60 [60 year~ 0 [good] 2 [Woma~ 166 66 110 70 10.2 0 [Non-s~ 24.0
6 60 [60 year~ 0 [good] 2 [Woma~ 168 61 130 78 7.3 0 [Non-s~ 21.6
7 60 [60 year~ 1 [not quite ~ 2 [Woma~ 159 67 122 74 5.2 0 [Non-s~ 26.5
8 60 [60 year~ 0 [good] 1 [Man] 172 62 142 88 7.4 0 [Non-s~ 21.0
9 60 [60 year~ 1 [not quite ~ 2 [Woma~ 153 68 150 100 7.2 0 [Non-s~ 29.0
10 60 [60 year~ 1 [not quite ~ 1 [Man] 179 87 133 85 7.8 0 [Non-s~ 27.2
# ... with 250 more rows

tapply(norsjo86$height, INDEX=list(factor(norsjo86$sex)), FUN=max, na.rm=T)

 1 2
191 178

tapply(norsjo86$sbp, INDEX=list(factor(norsjo86$sex), factor(norsjo86$agegrp)), FUN=mean, na.rm=T)

 30 40 50 60
1 119.0000 121.4839 127.4444 134.1212
2 109.6667 117.0556 132.1143 133.5938

```

9.13 The head and tail functions

If you have a data frame with many rows it is not practical to print all of it. With these functions you can look at the beginning or the end of the data frame.

```

x<-data.frame(x=seq(1,10,length.out=4000),y=seq(-628,10,length.out=4000),
              z=seq(0,180,length.out=4000))
nrow(x)

```

```
[1] 4000

head(x)

      x      y      z
1 1.000000 -628.0000 0.00000000
2 1.002251 -627.8405 0.04501125
3 1.004501 -627.6809 0.09002251
4 1.006752 -627.5214 0.13503376
5 1.009002 -627.3618 0.18004501
6 1.011253 -627.2023 0.22505626

tail(x,10)

      x      y      z
3991 9.979745 8.564141 179.5949
3992 9.981995 8.723681 179.6399
3993 9.984246 8.883221 179.6849
3994 9.986497 9.042761 179.7299
3995 9.988747 9.202301 179.7749
3996 9.990998 9.361840 179.8200
3997 9.993248 9.521380 179.8650
3998 9.995499 9.680920 179.9100
3999 9.997749 9.840460 179.9550
4000 10.000000 10.000000 180.0000
```

Take a look at the object `x` in this example - not very practical to print. There may even be a maximum number of rows for the output.

Chapter 10

Tables

Tabulation is a useful tool to describe the distribution of categorical data. R provides several possibilities to create tables. We use the data in *BackPain.csv* to demonstrate the concepts.

`table()` is the basic function for tabulation. It takes vectors as arguments to create tables, corresponding to the variables to be tabulated.

`xtabs()` is a convenient function for 1 and 2 way tabulations. It takes two arguments, the first starts with `'~'` followed by the names of columns to be tabulated separated by `'+'`, the second argument is the data that contains the columns. We focus on `xtabs()` as it often is more convenient than `tables`, but the two are largely similar.

First we load the *BackPain* dataset. Consider the following examples:

```
d <- read.csv("../data/BackPain.csv")
```

```
table(d$sex)

##
## Female    Male
##  18456   15666
```

This is equivalent to

```
xtabs(~sex, d)

## sex
## Female    Male
##  18456   15666
```

Cross tabulation of two variables in with `xtabs()` can be done like this:

```
xtabs(~sex+asthma, d) # two way with sex and asthma

##           asthma
## sex           no    yes
## Female    415 16968  1073
## Male      338 14279  1049
```

You may wonder what the `~` means. This is called a *formula* and is a special way in R to specify variables. You will later encounter it in a statistical context as well. For now, it is enough to know that

some functions like `xtabs()` work with formulas and that you use them to specify the variables. For `xtabs()`, the variables to be tabulated are specified after the `~` sign and are separated by a `+`. Normally, these are the columns of a dataframe that is provided as the *data* argument of `xtabs()`.

Own experimentation

Perform the same crosstabulation of sex and asthma using the `table()` function.

10.1 Enveloping

10.1.1 Summary

`xtabs()` results can be inserted into the `summary()` function to give chi square statistics. Note how the empty string `""` is also counted (the leftmost column). We can exclude values using the `exclude` argument in `xtabs()`:

```
summary(xtabs(~sex+asthma+age, d, exclude = "")) # calculate chi square statistics

## Call: xtabs(formula = ~sex + asthma + age, data = d, exclude = "")
## Number of cases in table: 33369
## Number of factors: 3
## Test for independence of all factors:
##  Chisq = 369.6, df = 178, p-value = 1.649e-15
##  Chi-squared approximation may be incorrect
```

Conveniently, the `summary()` function also computes a chi-square test of independence for the provided variables. The null hypothesis in that case is that the variables are independent. Because of the high *Chisq* the null hypothesis may be rejected. You will later learn much more on how to calculate test statistics with R.

10.1.2 Proportion

It can also be inserted into the `prop.table()` function to display proportions. Because in this example the numbers are that large, we use the `round` function with 2 decimal places to round the proportions:

```
# report proportions instead of frequencies
round(prop.table(xtabs(~sex+asthma, d, exclude = "")), 2)

##           asthma
## sex          no  yes
##  Female 0.51 0.03
##   Male  0.43 0.03

# gives percentages
round(prop.table(xtabs(~sex+asthma, d, exclude = "")), 2)*100

##           asthma
## sex          no  yes
##  Female 51   3
##   Male  43   3
```

To get row and column totals, we can use `margin.table()` function:

```
addmargins(xtabs(~sex+asthma, d, exclude = "")) # add both row and column totals
```

```
##           asthma
## sex          no   yes   Sum
## Female 16968 1073 18041
## Male   14279 1049 15328
## Sum    31247 2122 33369
```

Own experimentation

Use the `prop.table` function to create a table of percentages of physical and country, rounded to two places. How could you check that the values sum up to 100%?

10.1.3 Tabulation of more variables

2-way or crass tabulation is the most straight-forward way of tabulation, given it has the same dimensionality as the output (paper and computerscreens). However, its possibly to tabulate according to an arbitrary number of variables. Tables with more than two dimensions need to be collapsed to two dimensions.

`xtabs` can create 3-way contingency table:

```
xtabs(~sex+asthma+diabetes, d)
```

```
## , , diabetes =
##
##           asthma
## sex          no   yes
## Female    317   14    1
## Male     249   18    0
##
## , , diabetes = no
##
##           asthma
## sex          no   yes
## Female    91 15500  932
## Male     82 13313  959
##
## , , diabetes = yes
##
##           asthma
## sex          no   yes
## Female     7 1454  140
## Male       7  948   90
```

However, the output is not nicely formatted, or put otherwise, the collapsing is not very sophisticated. Feeding `xtabs()` to `fable()` can solve this:

```
fable(xtabs(~sex+asthma+wealthQ,d))
```

```
##           wealthQ      Q1 poorest  Q2  Q3  Q4 Q5 richest
## sex  asthma
## Female                2      111  91  68  72          71
##      no                67     3342 3340 3350 3443        3426
##      yes                7      227  240  242  185          172
```

```
## Male                2         76  67  62  55         76
##      no              51        2497 2727 2708 3052        3244
##      yes             3         241  230  215  202         158
```

It's also possible to directly use a formula and `ftable()`. `ftable()` expects variables on the left-hand side of the formula also, they indicate along what variables the table columns will be constructed. The right-hand side variables in the formula indicates the variables to contain the row categories:

```
ftable(sex+asthma~wealthQ+diabetes,d)
```

```
##              sex      Female      Male
##              asthma      no  yes      no  yes
## wealthQ      diabetes
##              0    0    1    1    0    0
##              no      2   64    6    1   46    2
##              yes     0    3    0    0    5    1
## Q1 poorest
##              92    1    0   65    5    0
##              no     19 3148  206  11 2420 227
##              yes     0  193   21    0   72   14
## Q2
##              72    1    0   47    2    0
##              no     17 3105  221  17 2601 217
##              yes     2  234   19    3  124   13
## Q3
##              50    5    0   41    7    0
##              no     17 3028  201  19 2534 191
##              yes     1  317   41    2  167   24
## Q4
##              51    4    0   39    2    0
##              no     19 3106  160  15 2797 179
##              yes     2  333   25    1  253   23
## Q5 richest
##              52    3    0   56    2    0
##              no     17 3049  138  19 2915 143
##              yes     2  374   34    1  327   15
```

Own experimentation

Try to use the `kable()` function in the *knitr* package to format 3 way tables generated using `xtabs()`. Unfortunately, `kable()` is not compatible with `ftable()`.

Chapter 11

Basic graphics

11.1 Introduction to graphics with R

There are a lot of nice functions for graphics in R. In this chapter we use some of the most common functions in base R. There are more advanced possibilities in the `ggplot2` package which we will see later on. However, also the plot functions in this chapter have a lot of options. It may be good to know about both worlds - sometimes you need quick graphics to illustrate your data and other times you need a nice layout or more complex illustration and then it is worth spending somewhat more time on how to do it.

Håkan Jonsson < hakan.jonsson@umu.se >

11.2 Histogram

Let us import the data file `norsjo86` and do a histogram for body height.

agegrp:	Age group	(30, 40, 50 ,60 years)
health:	Health status	(0=good, 1=not quite good/bad)
sex:	Sex	(1=man, 2=woman)
height:	Body height	(cm)
weight:	Body weight	(kg)
sbp:	Systolic blood pressure	
dbp:	Diastolic blood pressure	
kolester:	Cholesterol	
smoker:	Smoking status	(0=non-smoker, 1=smoker)
bmi:	Body mass index	(kg/m^2)

```
# import spss file norsjo86
library(haven)
nors <- read_sav("../data/norsjo86.sav")
nors

# A tibble: 260 x 10
   agegrp      health      sex height weight  sbp  dbp kolester  smoker  bmi
   <dbl+lbl> <dbl+lbl> <dbl+lbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl+lbl> <dbl>
1 60 [60 year~ 0 [good]      2 [Woma~ 157    61  110    70    6.7 0 [Non-s~ 24.7
2 60 [60 year~ 1 [not quite ~ 2 [Woma~ 157    97  150   100    6.6 0 [Non-s~ 39.4
```

```

3 60 [60 year~ 0 [good]      1 [Man]      170      74 136 96      8.2 0 [Non-s~ 25.6
4 60 [60 year~ 0 [good]      2 [Woma~     163      66 156 76      7.5 0 [Non-s~ 24.8
5 60 [60 year~ 0 [good]      2 [Woma~     166      66 110 70     10.2 0 [Non-s~ 24.0
6 60 [60 year~ 0 [good]      2 [Woma~     168      61 130 78      7.3 0 [Non-s~ 21.6
7 60 [60 year~ 1 [not quite ~ 2 [Woma~     159      67 122 74      5.2 0 [Non-s~ 26.5
8 60 [60 year~ 0 [good]      1 [Man]      172      62 142 88      7.4 0 [Non-s~ 21.0
9 60 [60 year~ 1 [not quite ~ 2 [Woma~     153      68 150 100     7.2 0 [Non-s~ 29.0
10 60 [60 year~ 1 [not quite ~ 1 [Man]      179      87 133 85      7.8 0 [Non-s~ 27.2
# ... with 250 more rows

```

```

par(mfrow=c(2,2))
hist(nors$height)
# hist(nors$height)$breaks
# str(hist(nors$height))
range(nors$height,na.rm=T)

```

```
[1] 145 191
```

```

hist(nors$height,breaks=seq(140,200,by=2))
hist(nors$height,breaks=seq(140,200,by=10),col=3,density=30)

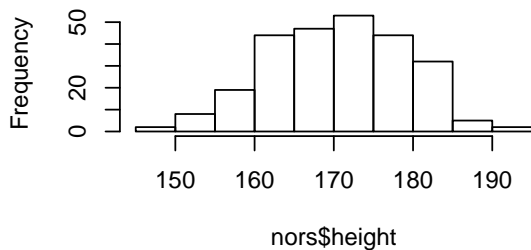
```

```

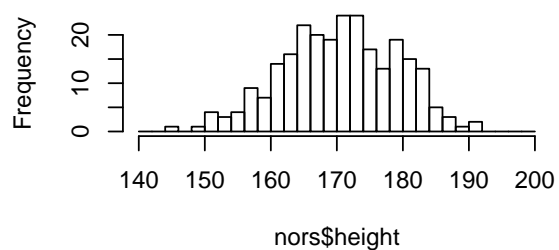
hist(nors$height,xlim=c(100,200),freq=F,xlab="Body height",main="Histogram") # total area equal to 1

```

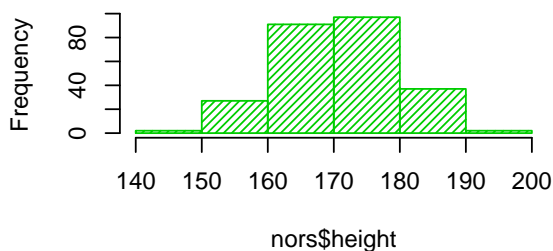
Histogram of nors\$height



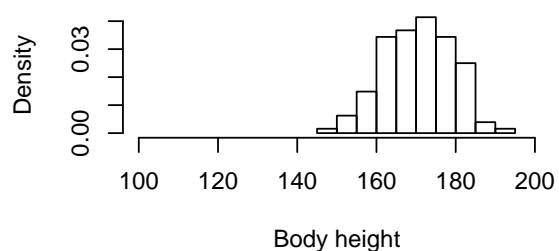
Histogram of nors\$height



Histogram of nors\$height



Histogram



11.3 Boxplot

Now let us look at boxplots for cholesterol by sex and age group.

```

par(mfrow=c(2,2))
nors$sex<-factor(nors$sex,levels=c("1","2"),labels=c("man","woman")) # make a factor

```



```

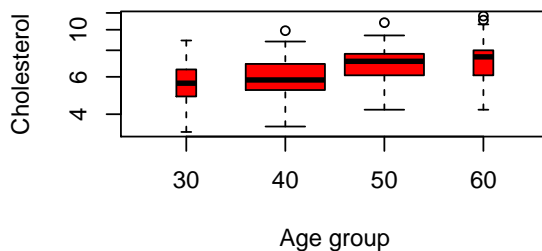
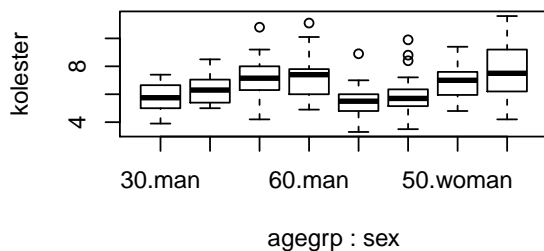
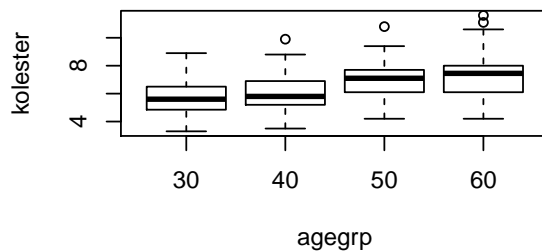
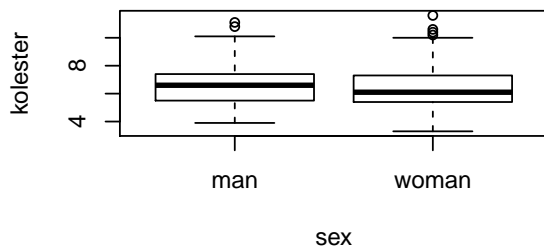
boxplot(kolester~sex,data=nors)

boxplot(kolester~agegrp,data=nors)

boxplot(kolester~agegrp+sex,data=nors) # combining two categorical variables

# Example of some extra options
boxplot(kolester~agegrp,data=nors,col=2,ylab="Cholesterol",
        xlab="Age group",width=c(0.1,0.4,0.4,0.1),log="y")

```



11.4 Scatterplot

Scatterplots are very useful for illustrating relationships between two continuous variables. They can also be used to illustrate trends e.g. over time. There are alternatives how to give the x and y variable but it is usually more convenient to use the formula object alternative.

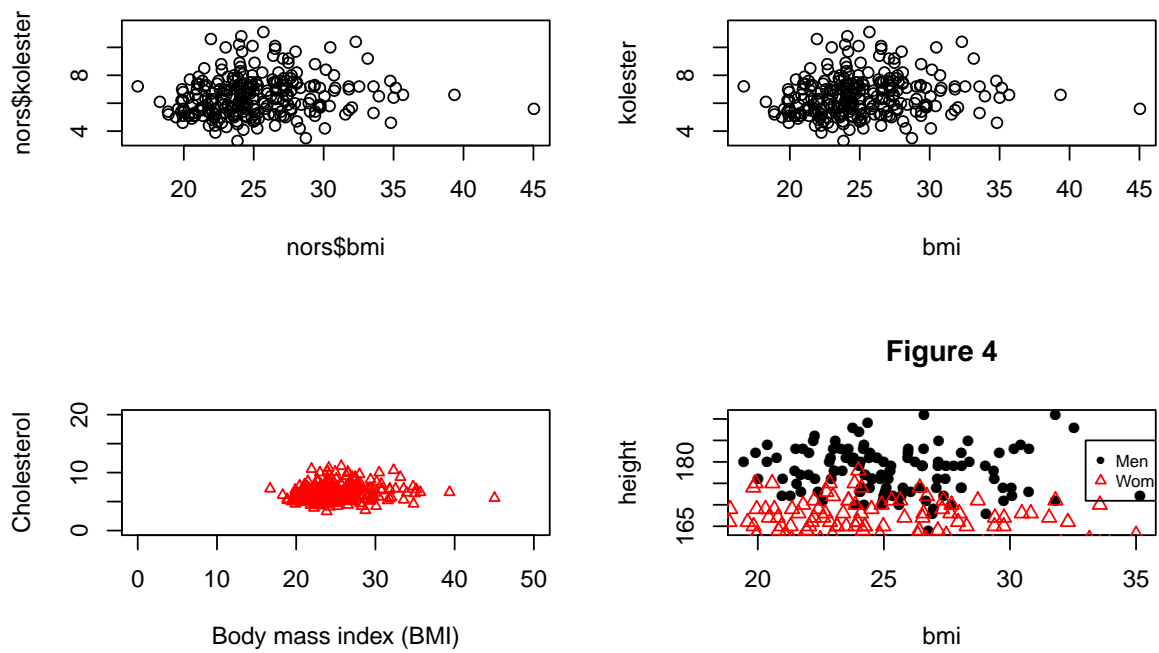
```

# two similar alternatives with different symbols
par(mfrow=c(2,2))
plot(nors$bmi,nors$kolester)
plot(kolester~bmi,data=nors) # using formula object

# Example of some extra options
plot(kolester~bmi,data=nors,col=2,pch=2,cex=0.7,ylab="Cholesterol",
     xlab="Body mass index (BMI)",ylim=c(0,20),xlim=c(0,50))

# two groups in the same plot
plot(height~bmi,data=subset(nors,sex=="man"),pch=16,main="Figure 4")
points(height~bmi,data=subset(nors,sex=="woman"),pch=2,col=2)
legend(x=33,y=185,legend=c("Men","Women"),col=c(1,2),pch=c(16,2),cex=0.7)

```

**Figure 4**

We can also use scatterplot to do plots based on mathematical functions.

```
x<-seq(-20,20,by=0.001)

y1<-10*x^2-100
plot(x,y1,type="l",ylim=c(-5000,5000),ylab="y")

y2<-5+-x-3*x^2+x^3
lines(x,y2,col=2,lty=2,lwd=2)

y3<-sin(x)*1000
lines(x,y3,col=3,lwd=3)

abline(h=0,lty=2) # adding a horizontal line
```

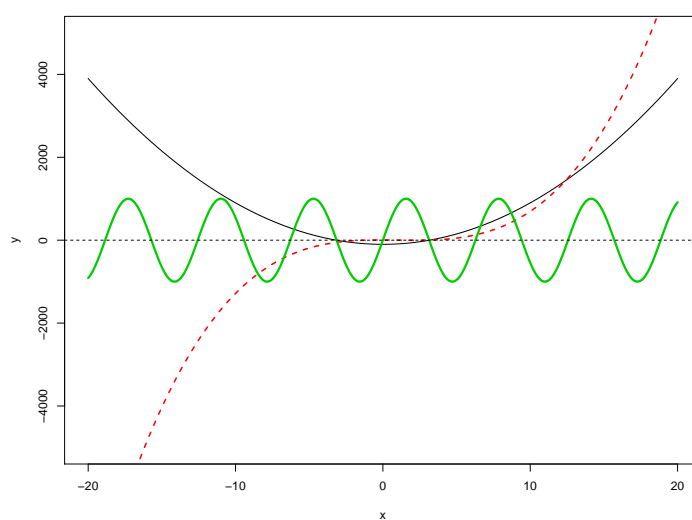


Figure 11.1: Plot of mathematical functions

Own experimentation

The plot (Figure 4) does not show the height for all women - why? Also the legend need to be adjusted. Can you fix it? Try to plot some own mathematical functions.

11.4.1 Graphical parameters

You have seen that for many functions there are a varying number of arguments, some important and some more optional with default values we may not be aware of. Plot functions usually have a large number of arguments. The argument parameters can be set globally by `par()` but usually they are set in the respective plot function. However you can find a list of these options in the help for `par` under Graphical Parameters.

Chapter 12

Simulation

Håkan Jonsson < hakan.jonsson@umu.se >

12.1 Simulation - what is it?

Simulation is a way of generating pseudo random numbers based on a numeric algorithm. If the algorithm is good enough we will in practice not see a difference compared to real randomness. Pseudo random numbers are most often generated as independent random observations from a uniform distribution between 0 and 1. Based on this we can generate random numbers from any distribution.

We have earlier used the function **sample** to get a subset of a data set. It was actually a random sample meaning that each observation have the same probability to be chosen. It can also be used to do simulation.

12.2 Generate random samples

What about throwing a die. We use `replace=T` because the throws are independent of each other.

```
die<-c(1:6)
die

[1] 1 2 3 4 5 6

sample(die,1) # one throw

[1] 4

sample(die,1) # another throw

[1] 1

y<-sample(die,size=15,replace=T) # result of 15throws
y

[1] 2 4 4 6 4 5 5 1 6 2 5 1 6 2 4
```

Now we have a chance to calculate the probability of events. What is the probability to get a sum larger than or equal 12 with 3 throws. We can as above get the result of 3 throws and calculate the sum.

But that is just one observation. We need a large sample but here we use an example of 5 observations. To calculate the probability more exact we can e.g. set $n=10000$.

```
set.seed(1001)
# small example for illustration. Each row is three throws with the die.
n<-5
throws<-sample(die,size=3*n,replace=T)
throws

[1] 3 3 6 6 4 4 4 4 4 5 5 6 3 2 2

y<-data.frame(matrix(throws,ncol=3))
y

  X1 X2 X3
1  3  4  5
2  3  4  6
3  6  4  3
4  6  4  2
5  4  5  2

ys3<-y[,1]+y[,2]+y[,3]
ys3

[1] 12 13 13 12 11

ys3b<-apply(y,MARGIN=1,FUN=sum) # this may be a better alternative
ys3b

[1] 12 13 13 12 11

# repetition of sam commands but large size calculation (n)
n<-10000
y<-matrix(sample(die,size=3*n,replace=T),ncol=3)
str(y)

int [1:10000, 1:3] 4 3 4 1 2 6 6 3 5 6 ...

ys3b<-apply(y,MARGIN=1,FUN=sum)
str(ys3b)

int [1:10000] 16 13 12 7 10 13 13 8 9 15 ...

# the proportion of TRUE is the probability we wanted to calculate
table(ys3b>=12)/n

FALSE TRUE
0.6213 0.3787

# or similarly
prop.table(table(ys3b>=12))

FALSE TRUE
0.6213 0.3787
```

If we repeat the same simulation the result will not be the same as the first time although the simulation is made similarly. To ensure we can get the same result every time we can use **set.seed**

```
sample(die,size=10,replace=T)

[1] 5 1 1 3 6 4 6 2 2 5
```

```
sample(die,size=10,replace=T)

[1] 1 5 3 6 3 6 1 4 4 6

set.seed(1001)
sample(die,size=10,replace=T)

[1] 3 3 6 6 4 4 4 4 4 5

set.seed(1001)
sample(die,size=10,replace=T)

[1] 3 3 6 6 4 4 4 4 4 5

sample(die,size=10,replace=T)

[1] 5 6 3 2 2 4 3 4 1 2
```

Now let us look at the statistician's favorite; the urn with balls in different colors. Assume an urn with 10 black, 5 white and 7 red balls. Let us simulate a randomly chosen sample of 5 balls. This is an example of sampling without replacement.

```
set.seed(1001)
urn<-c(rep("black",10),rep("white",5),rep("red",7))
urn

[1] "black" "black" "black" "black" "black" "black" "black" "black" "black" "black"
[11] "white" "white" "white" "white" "white" "red" "red" "red" "red" "red"
[21] "red" "red"

sample(urn,size=5,replace=F) # Observe replace=F

[1] "black" "white" "red" "black" "red"

sample(urn,size=5,replace=F) # another sample

[1] "white" "black" "white" "black" "white"
```

12.3 Generate observations with common distributions like normal, uniform etc.

We now take a look at the simulation functions for the most common distributions; uniform, normal and binomial. Observe that very similar functions can be used for calculation of the density, distribution and quantile function depending on the first letter. The randomisation function names all start with r. Uniformly distributed values is generated by **runif**, **rnorm** and **rbinom** respectively.

```
set.seed(1001)
x<-runif(10,min=0,max=1) # uniform distribution
x

[1] 0.985688783 0.412628483 0.429539246 0.419172236 0.426506559 0.887797565 0.006096034
[8] 0.081215761 0.288657362 0.765342145

x<-runif(1000,min=-2,max=2)

y<-rnorm(10) # per default standard normal distribution
y
```

```

[1] -1.22661462  0.91854625  0.02702411 -2.12719151 -1.60081510  0.74968130  0.72191446
[8] -0.33823015  0.91359284  0.58779494

y1<-rnorm(1000)
summary(y1)

      Min.   1st Qu.   Median     Mean   3rd Qu.     Max.
-2.99286 -0.62887  0.02054  0.02637  0.67440  3.31052

sd(y1)

[1] 0.9758944

y2<-rnorm(1000,mean=2,sd=0.1)
summary(y2)

      Min. 1st Qu.  Median     Mean 3rd Qu.     Max.
 1.670   1.927   1.992   1.994   2.061   2.260

sd(y2)

[1] 0.1001423

# illustration of the number of times we get "six" when throwing a die 5 times.
# The experiment is repeated 1000 times. The probability of each throw=1/6
z<-rbinom(n=1000,size=5,p=1/6)
table(z)

z
 0   1   2   3   4
401 408 161  29   1

# the estimated probabilities of the number of getting "6" in five throws
prop.table(table(z))

z
 0   1   2   3   4
0.401 0.408 0.161 0.029 0.001

```



```
par(mfrow=c(3,1))  
hist(x,main="Uniform(2,2)")  
hist(y1,main="Normal(0,1) i.e. standard normal")  
hist(y2,main="Normal(2,0.1)")
```

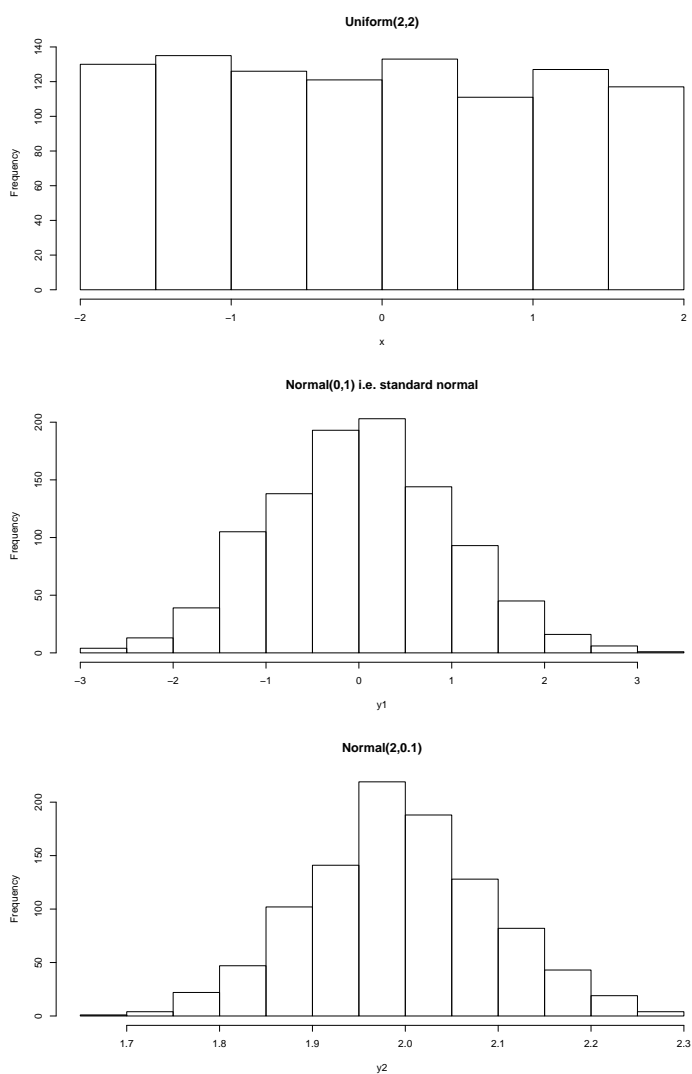


Figure 12.1: Histograms

12.4 Generating observations with other distributions

Random numbers can be generated for a large number of distributions (see `help` for distribution). Similarly functions for the density/mass function, cumulative distribution function, quantile function can be found with corresponding names starting with `d`, `p` or `q`, respectively.

12.5 Bootstrapping

Bootstrapping is a method where random sub-samples are drawn from the original data (with replacement) and estimations are repeated a number of times. This enables information about the random variation of the results and calculation of for example confidence intervals is possible. We may do this using simulation and some programming but there is a function for this **boot::boot** which makes it more convenient. However, some preparations are necessary. The method is illustrated by an example using the **mtcars** dataset. The linear regression model $mpg = a + b * wt$ is estimated and we want to calculate a confidence interval for R^2 . The variables are described below. However, the variable `disp` is not used in this example.

<code>mpg</code>	Miles/(US) gallon
<code>wt</code>	Weight (1000 lbs)
<code>disp</code>	Displacement (cu.in.)

```
head(mtcars)
```

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Mazda RX4	21.0	6	160	110	3.90	2.620	16.46	0	1	4	4
Mazda RX4 Wag	21.0	6	160	110	3.90	2.875	17.02	0	1	4	4
Datsun 710	22.8	4	108	93	3.85	2.320	18.61	1	1	4	1
Hornet 4 Drive	21.4	6	258	110	3.08	3.215	19.44	1	0	3	1
Hornet Sportabout	18.7	8	360	175	3.15	3.440	17.02	0	0	3	2
Valiant	18.1	6	225	105	2.76	3.460	20.22	1	0	3	1

```
fit0<-lm(mpg~wt,data=mtcars)
summary(fit0)
```

Call:

```
lm(formula = mpg ~ wt, data = mtcars)
```

Residuals:

Min	1Q	Median	3Q	Max
-4.5432	-2.3647	-0.1252	1.4096	6.8727

Coefficients:

	Estimate	Std. Error	t value	Pr(> t)
(Intercept)	37.2851	1.8776	19.858	< 2e-16 ***
wt	-5.3445	0.5591	-9.559	1.29e-10 ***

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

Residual standard error: 3.046 on 30 degrees of freedom

Multiple R-squared: 0.7528, Adjusted R-squared: 0.7446

F-statistic: 91.38 on 1 and 30 DF, p-value: 1.294e-10

```
set.seed(1001)
```

```
library(boot)
```

```

# This is a function to estimate the model and as result give the R-Squared
rsq <- function(formula, data, indices) {
  d <- data[indices,] # indices allows boot to select a specific sample
  fit <- lm(formula, data=d)
  return(summary(fit)$r.square)
}

# Example of the function rsq where all observations in the original data is used
# Compare with R-squared in the result (fit0) above
rsq(mpg~wt,data=mtcars,1:32)

[1] 0.7528328

# bootstrapping with 1000 replications
results <- boot(data=mtcars, statistic=rsq,
                R=1000, formula=mpg~wt)

results

ORDINARY NONPARAMETRIC BOOTSTRAP

Call:
boot(data = mtcars, statistic = rsq, R = 1000, formula = mpg ~
      wt)

Bootstrap Statistics :
      original      bias    std. error
t1* 0.7528328 0.006813838 0.05781024

results$t0 # this is the result from the original sample, compare R-squared

[1] 0.7528328

# this is the simulated 1000 estimates of R-squared
length(results$t)

[1] 1000

mean(results$t)-results$t0 # compare the bias in the printed result

[1] 0.006813838

sd(results$t) # compare the standard deviation in the printed result

[1] 0.05781024

# there are 5 possible types, the default is "all"
boot.ci(results, type="perc")

BOOTSTRAP CONFIDENCE INTERVAL CALCULATIONS
Based on 1000 bootstrap replicates

CALL :
boot.ci(boot.out = results, type = "perc")

Intervals :
Level      Percentile
95%      ( 0.6268,  0.8525 )
Calculations and Intervals on Original Scale

```

```
# compare with quantiles, it seems very similar
```

```
quantile(results$t,c(0,0.025,0.975,1))
```

0%	2.5%	97.5%	100%
0.4901092	0.6323692	0.8521974	0.9048545

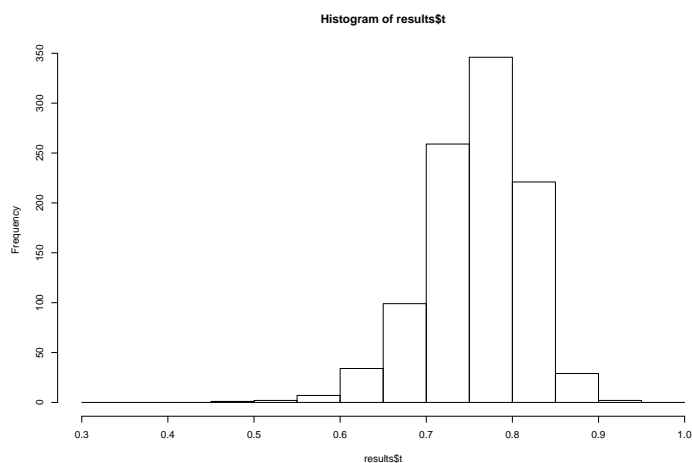


Figure 12.2: Histogram of estimated R-square

Chapter 13

R markdown

One of the interesting features around R is the ability to combine the data analysis and the writing-up of results into one single document. In practice this means mixing R code that will be executed with content such as text or images in a file. While this file is processed, the contained R code is executed and the results inserted into the final rendered document content (See 13)

This has several potential advantages that tie nicely into good accepted good research practices:

- Connection of result reporting and analysis: The explicit connection strengthens truthful reporting of results, reproducibility of research, and transparency.
- Reduced effort: Results do not need to be updated in the report seperatly
- Avalability: Sharing the raw unprocessed file shares both the content and the contained code (ideally the underlying data is shared additionally as well)

An easy way do achieve this in R is using R Markdown with R Studio. R Markdown files end in ".Rmd". To create a basic R Markdown file in R Studio, go to File->New File->R Markdown.

An R Markdown file starts with the YAML header that denotes the title and the output format for the document. Three underscore characters on one line indicate the beginning and the end of the YAML header, which here defines a title and the output format:

```
---  
title: "Example document"  
output: "html_document"  
---
```

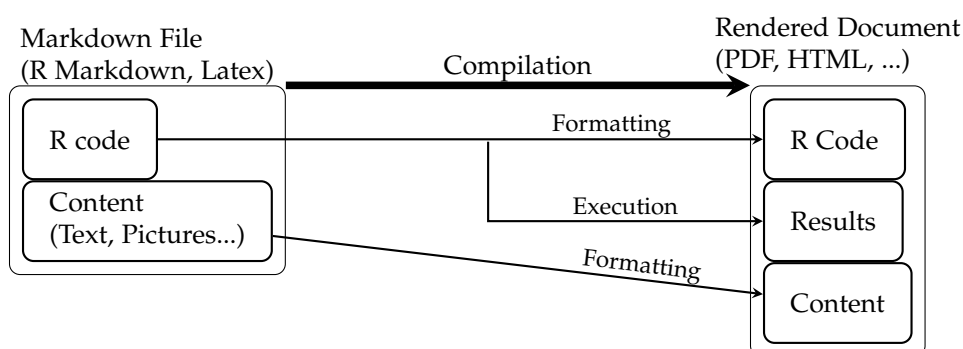


Figure 13.1: Schmeatic of embedding R in Markdown documents

After the YAML header follows the actual content. R markdown files contain two main components in addition to the YAML header: Text to be displayed in the document and R code chunks.

13.1 Text formatting

Unlike in visual text processors like Microsoft Word, the R Markdown files do not look like the final document when during the editing. The R Markdown files themselves are simple text files. In addition to the actual content, they contain instructions how the content should be formatted in the final document. To get this final document, R markdown files need to be processed, or "compiled". The R package that does this processing is called *knitr*. In R studio, an R markdown document can be compiled by clicking on *Preview* in the toolbar. Alternatively, another output format such as PDF can be selected by clicking on the small downwards arrow to the right of *Preview*.

Text written in an R Markdown file will be put as is in the final document, subject to adjusting line breaks or similar.

R Markdown also provides basic text formatting capabilities. Some special characters combined with the text to be formatted instruct the document compiler to apply the respective formatting. For example, two spaces followed by a linebreak will produce a new paragraph. **text** puts the text in italic, ***text*** in bold.

R markdown has many more formatting commands to provide all basic formatting capabilities such as insertion of images, citations, lists, and more. See the R Markdown cheatsheet for a list. The R Markdown cheatsheet can also be found in the RStudio **Help** menu.

13.2 Code chunks and plots

R code chunks contain R code that is executed when the document is compiled. R code chunks start with ````{r}` and end with `````

Consider this example:

```
```{r Scatterplot1, fig.align='right', fig.cap='Scatterplot 1'}
plot(cars)
```
```

In the curly `{}` brackets, after "r", follows the name of the chunk (here `Scatterplot1`). Then follow one or several options. After the curly `{}` brackets follows the R code. In the example a simple scatterplot of the in R per default available `cars` data.frame. More complex `ggplot2` plots or similar can be included in R markdown documents in the same fashion.

For a list of useful code chunk options check the R markdown cheatsheet. Comprehensive lists can be found on the internet, for example at <https://yihui.org/knitr/options/>. Other important chunk options include `echo=FALSE` to disable printing the R code in the document, `results='hide'` to only run the code but not print the results, and `eval=FALSE` to not actually run the R code (the code itself is still printed, depending on the *echo* option)

There is also the option to include R code in R markdown documents in a shorter fashion, using the inline syntax. The inline syntax starts with ``r`, followed by the R code, and ends with a single ```. For example, ``r Sys.time()`` includes the output of `Sys.time()` in the text, without paragraph formatting and code output as with code chunks:

Current date and time: ``r Sys.time()`` → Current date and time: 2020-06-26 13:19:38

13.3 Tables

Putting tables in R markdown documents is made easy by the *kable* package:

```
```{r table1, results='asis'}  
knitr::kable(head(cars))
```
```

The chunk option `results='asis'` is necessary because the `kable` function returns html that can be directly 'asis' integrated in a html document, and does not need to be processed further by knitr. The format that `kable` returns is determined automatically, in our case html if we did not select another output format instead.

Note that in the example the `kable` function is called using the double colon syntax, this way the knitr package does not have to be loaded.

Chapter 14

Programming

Even if R's and our focus is on statistical matters, R can be used for programming. Programming in this context means to perform general tasks on digitalized data. Strictly, R is a scripting language because it is "interpreted": Instead of directly giving instructions to the computer processor, another program reads the script code that we write and translates it into instructions that are sent to the processor.

R provides all the common programming constructs needed to control the flow of a program execution: As you now, R scripts are executed beginning with the first line, then the next line, until the end of the file. This is not very flexible, but the program execution can be controlled to follow any desired pattern.

Mainly three constructs are used to that purpose: *branching*, *loops*, and *functions*.

14.1 Branching

Branching, or conditional statements, means that a condition is checked before a certain section of code is run. These conditions need to result in either TRUE or FALSE. You can imagine a river that splits into two where the water flows down the one branch if the condition is TRUE and the other branch if its FALSE.

The branching is done putting `if else` statements in the code. `if` is followed by the condition in brackets, then by curly brackets that surround the code that should be executed depending on whether the condition is true.

```
x <- sample(1:3, 1)
print(x)

## [1] 1

if(x == 1){
  print("x is 1")
}

## [1] "x is 1"
```

If statements can optionally be followed by `else if` or `else` statements. `else if` statements work similarly to `if` statements. `else` statements are executed none of the preceding `if` or `else if` conditions result in TRUE:

```

if(x == 1){
  print("condition 1 is true, x is 1")
} else if(x == 2) {
  print("condition 2 is true, x is 2")
} else{
  print("Condition 1 and 2 are not true, x must be 3")
}

## [1] "condition 1 is true, x is 1"

```

Note that `else` and `else if` statements need to be on the same line as the closing curly bracket of the previous code block.

Branching is very important in programming because it allows programs to be flexible. For example, an R can check if a package that should be loaded is already installed, electric cars can play sounds if the battery level becomes low, social media platforms can check if uploaded pictures are appropriate, and countless more. Of course, all these examples involve much more than simple *if - else* checks.

Own experimentation

What happens if you run?

```

if(log(-1) > 0){
  print("?")
}

```

Try to fix the problem by using `is.na()`.

14.2 Loops

Loops are another important mean to control the program execution flow. They allow to blocks of code to be repeated. Similar to *ifs* they need to be tied to a condition because one wants to avoid loops to run forever. The two most common loop constructs are `for` and `while` loops.

`while` loops repeats a block of code until the associated condition evaluates to `FALSE`. They are useful to in situations where the exact number of repetitions is not known beforehand:

```

x <- 10
while(x >= 5){
  print(x)
  x <- sample(1:10, 1)
}

## [1] 10
## [1] 5
## [1] 6

```

This loop could run for a very long time if one is unlucky, but this scenario is unlikely.

`for` loops in R traverse over the elements of a vector. In the background, R checks after every repetition if the end of the vector is already reached:

```

for(i in 1:10){
  print(i)
}

```

```

}

## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
## [1] 10

```

This example prints 1 to 10 to the console.

In general, loops in R should be avoided as much as possible. Instead, the provided functions that use vectors as inputs should be used if available because it is usually much faster and requires much less code. Also consider using the `apply()` family of functions instead of loops.

However, sometimes you may feel more comfortable with loops than with alternatives. Loops are as performant as `apply()` functions and can make code easier to understand and write in some situations.

To demonstrate the differences in performance between the loop variants and vectorized functions, we write a little program to calculate the time it takes to calculate the sum of a large vector.

```

v = sample(100, 1000000, replace = T)

startTime <- Sys.time() # save current time point
i <- 1 # i stands for index
cs <- 0 # cs: current sum, start with 0

# to get the total sum of v, we retrieve each element in the loop
# and add it to the current sum
while(i < length(v)){ # run until i >= length of v

  cs <- cs+v[i] # add the number at index i
  i <- i +1 # increase the index to progress
}
endTime <- Sys.time() # save the current time point

print(paste0("Sum is: ", cs)) # print the current sum (use paste0 to )

## [1] "Sum is: 50494155"

print(paste0("Time for while loop: ", round(endTime - startTime,3), " seconds"))

## [1] "Time for while loop: 0.289 seconds"

startTime <- Sys.time()
cs <- 0 # set cs to 0 again
for(i in v){ # in the for loop, we automatically get the next element i in v
  cs <- cs+i # add i to current sum
}
endTime <- Sys.time()
print(paste0("Sum is: ", cs))

## [1] "Sum is: 50494192"

print(paste0("Time for for loop: ", round(endTime - startTime,3), " seconds"))

```

```
## [1] "Time for for loop: 0.034 seconds"

startTime <- Sys.time()
cs <- sum(v)
endTime <- Sys.time()
print(paste0("Sum is: ", cs))

## [1] "Sum is: 50494192"

print(paste0("Time for for sum: ", round(endTime - startTime,3), " seconds"))

## [1] "Time for for sum: 0.003 seconds"
```

Two other type of loops are the `dowhile()` loop and the `repeat` loop. These are not commonly used and they will not be explained here.

Own experimentation

Use the help to read about `next` and `break` (run `?next`) and what they do in a loop. What do you think are advantages and disadvantages of using `break` and `next` statements inside loops?

14.3 Functions

By now, you already have encountered a large number of functions in R, and probably got a good idea of why they are useful. They allow to encapsulate specific functionality and to make it reusable in varying contexts. Of course, R also allows you to define your own functions.

The defining feature of functions are their brackets. Functions take zero or more arguments in round brackets that follow the function name. These arguments allow to pass data from where the function is called to the code inside the function. This means that the code inside the functions sees the value of the objects that were given to the function when calling it.

Functions are stored in objects in R. In their definition, the object name is followed by the equal sign, the special keyword `function` and then the a comma-separated list of arguments in round brackets. The code of the function follows after that enclosed by curly brackets. This section is called the function body:

```
myFunction <- function(arg1, arg2){
  print("Hello, this is my function. The weighted sum of arguments is: ")
  print(arg1*2+arg2)
}
```

To execute the function, write the name, followed by the comma-separated argument list in brackets:

```
number <- 4
myFunction(4,number)

## [1] "Hello, this is my function. The weighted sum of arguments is: "
## [1] 12
```

Now the `myFunction` sees `arg1` as 4 and `arg2` as 12.

These arguments are just normal objects and can be used inside the function as such. However, modifying them won't alter the value of the original object that was passed to the function. The function just operates on a copy of the original object:

```
number <- 4
myFunction2 <- function(number){
  number <- 5 # last used value will be returned (v)
}
print(myFunction2())

## [1] 5

number # what is number?
## [1] 4
```

Note that 5 and 4 were printed, and not two times 5.

14.3.1 Anonymous functions

In R, functions are objects comparable to objects that just hold values. Because of that, it is straightforward to use functions in a similar way as objects. Specifically, functions are only executed when followed by () brackets, and otherwise treated similarly to objects:

```
afunction <- function(number){
  print("Hello")
}
bfunction <- afunction # use function as object
bfunction() # execute functions with ()

## [1] "Hello"
```

This allows for a lot of flexibility, because one can pass functions to other functions or one can directly specify function bodies without function names, so called *anonymous* functions:

```
cfunction <- function(x){
  print(paste0("Value of x is: ", x))
}
sapply(1:3, cfunction) # pass function as an argument

## [1] "Value of x is: 1"
## [1] "Value of x is: 2"
## [1] "Value of x is: 3"
## [1] "Value of x is: 1" "Value of x is: 2" "Value of x is: 3"

sapply(1:3, function(x){ x }) # use anonymous function

## [1] 1 2 3
```

14.3.2 Return values

Functions can return values that can be assigned to objects. In R, it is sufficient to just write the name of the value or object that should be returned:

```
myRetFunction <- function(){
  astring <- "blue"
  astring # return astring
}
color <- myRetFunction()
color
## [1] "blue"
```

This function without arguments returns the character sequence "blue".

In general, the return value of a function is the result of the last executed expression. This is way the above example works. It is possible to explicitly return values by using the `return()` statement, which accepts the return value as an argument:

```
lazySum <- function(x){
  # demonstrate explicit returns using return ()
  if(weekdays(Sys.time()) == "Sunday")
  {
    return("I dont work on Sundays")
  }
  else
  {
    return(sum(x))
  }

  "I don't know"
}
```

Explicit `return()` statements are useful if not the whole function body should be executed.

In R, it is possible to refer indicate the arguments of functions by their name when calling the function:

```
myFunction(arg2 = 2, arg1 = 1) # indicate arguments by name
## [1] "Hello, this is my function. The weighted sum of arguments is: "
## [1] 4
```

If no names are provided, the arguments values or objects are assigned to the arguments according to the order that they are listed. Even if not always necessary, indicating which argument is meant by using the '=' syntax may make an R script much easier to understand.

It is also possible to provide default values for function arguments when defining functions. This makes it optional to specify the value for this argument. Default values should preferable used for the last arguments in the function definition. This way, it is possible to leave out the argument with the default value when calling the function, even when not specifying all other arguments by name:

```
# provide a default argument
myDefArgFunction <- function(arg1, arg2, arg3=2){
  print(sum(arg1+arg2+arg3))
}
myDefArgFunction(2, 3) # only needs two arguments
## [1] 7
```

14.3.3 Variable scope

The objects created in R functions are local to those function, meaning that they cannot be used outside:

```
f <- function(){
  obj <- "object"
}
f()
print(obj)

## Error in print(obj): object 'obj' not found
# we get an error!
```

Objects that are available where a function is called from are also accessible inside of the the function:

```
x <- 10
f <- function(){
  print(x)
}
f()

## [1] 10
```

Such objects from parent contexts are also called *global* objects. They can not, however, be altered using `<-` or `=`, `x` will not change to 77 in the following example:

```
x <- 10
f <- function(){
  x <- 77
}
f()
x

## [1] 10
```

It is possible to do so using the `<-` operator, but this is not recommended in most circumstances. The reason for is that the code inside a function should in general only be related through the arguments and the return value to whatever other code calling it. Changing global objects from within a function violates this principle and can it harder to understand the code or find problems i case of unexpected behavior.

Two other type of loops are the `dowhile()` loop and the `repeat` loop. These are not commonly used and they will not be explained here.

Own experimentation

Write a function `testNA` that takes two arguments: The first is another function, and the second a numerical value. The function should return `TRUE` if the function that is passed as the first arguments returns `NA` for the numerical value in the second argument. Otherwise it should return `FALSE`.

Test your function with `log(-1)` and `log(1)`.

Chapter 15

Statistical methods part 1

The course is not actually a statistical course and we are not going through the theory of the statistical methods that will be demonstrated. However, some explanations of what the methods do will be given. So even if you are not familiar with the methods you should still be able to follow and learn how to use R for these statistical analyses.

Håkan Jonsson < hakan.jonsson@umu.se >

15.1 Statistical measures

15.1.1 Data set

Let us first import the data set norsjo86.

| | | |
|-----------|--------------------------|--------------------------------|
| agegrp: | Age group | (30, 40, 50 ,60 years) |
| health: | Health status | (0=good, 1=not quite good/bad) |
| sex: | Sex | (1=man, 2=woman) |
| height: | Body height | (cm) |
| weight: | Body weight | (kg) |
| sbp: | Systolic blood pressure | |
| dbp: | Diastolic blood pressure | |
| kolester: | Cholesterol | |
| smoker: | Smoking status | (0=non-smoker, 1=smoker) |
| bmi: | Body mass index | (kg/m^2) |

```
# import spss file norsjo86
library(haven)
norsjo86 <- read_sav("../data/norsjo86.sav") # this becomes a tibble
norsjo86<-as.data.frame(norsjo86)
head(norsjo86)
```

| | agegrp | health | sex | height | weight | sbp | dbp | kolester | smoker | bmi |
|---|--------|--------|-----|--------|--------|-----|-----|----------|--------|----------|
| 1 | 60 | 0 | 2 | 157 | 61 | 110 | 70 | 6.7 | 0 | 24.74745 |
| 2 | 60 | 1 | 2 | 157 | 97 | 150 | 100 | 6.6 | 0 | 39.35251 |
| 3 | 60 | 0 | 1 | 170 | 74 | 136 | 96 | 8.2 | 0 | 25.60554 |
| 4 | 60 | 0 | 2 | 163 | 66 | 156 | 76 | 7.5 | 0 | 24.84098 |
| 5 | 60 | 0 | 2 | 166 | 66 | 110 | 70 | 10.2 | 0 | 23.95123 |
| 6 | 60 | 0 | 2 | 168 | 61 | 130 | 78 | 7.3 | 0 | 21.61281 |

15.1.2 Mean, median and variance

We start by calculating the most common statistical measures. The formulas and common notations are as follows: mean $\bar{x} = \frac{\sum_{i=1}^n x_i}{n}$, variance $s^2 = \frac{\sum_{i=1}^n (x_i - \bar{x})^2}{n-1}$, standard deviation $s = \sqrt{s^2}$, correlation $\rho = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{s_x s_y}$ where x_i and y_i are the sample observations and n the sample size.

There are often missing in data. Arguments like **na.rm** or **use** has to be used in some of the functions to ensure the result to be calculated. For some of these measures we can use **summary**

```
x<-norsjo86$height
summary(x)

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.     NA's 
145.0   165.0   171.0   170.9   178.0   191.0         4 

mean(x) # what happens here?

[1] NA

mean(x,na.rm=T) # na.rm=T helps

[1] 170.9141

ownmean<-function(x){mean(x,na.rm=T)} # we can define an own function that removes NA
ownmean(x)

[1] 170.9141

median(x,na.rm=T)

[1] 171

range(x,na.rm=T)

[1] 145 191

quantile(x,na.rm=T) # as default it gives the same result as summary

 0%  25%  50%  75% 100%
145  165  171  178  191
```

The **quantile()** function gives as default the same result as **summary()**. It also give both results in **range()**. While mean and median gives information of the location of data the latter gives information on the variation in data . The most common measure of variation is however standard deviation which is the square root of the variance. It gives rise to some confusion because they actually measure the same thing. Standard deviation can approximately be interpreted as the average absolute deviation from the mean.

```
var(x,na.rm=T) # variance

[1] 75.85141

# alternative calculation of variance
xnm<-x[!is.na(x)] # first remove the missing observations
v<-sum((xnm-mean(xnm))^2)/(length(xnm)-1)
v

[1] 75.85141

sd(x,na.rm=T)
```

```
[1] 8.709271
sqrt(v)                # alternative calculation
[1] 8.709271
```

15.1.3 Correlation

Correlation is a measure of relationship with values in $[-1, 1]$ and is always calculated between two variables. The most common correlation is Pearson's correlation (ρ) which measures the linear relationship.

Correlation can be calculated for many variables at a time but always pairwise. The same function is used independent of the number of variables. Observe the argument `na.rm=T` does not work with the correlation function.

Pearson's correlation

```
cor(norsjo86$sbp, norsjo86$dbp)    # correlation
[1] NA
cor(norsjo86$sbp, norsjo86$dbp, use="complete.obs")
[1] 0.7582536
cor(norsjo86[,c("height", "weight", "sbp", "dbp")], use="complete.obs")

      height    weight      sbp      dbp
height 1.00000000 0.5049999 0.01716543 0.03305511
weight 0.50499987 1.0000000 0.26780039 0.40531170
sbp     0.01716543 0.2678004 1.00000000 0.75608662
dbp     0.03305511 0.4053117 0.75608662 1.00000000

# or similarly
cor(norsjo86[,4:7], use="complete.obs")

      height    weight      sbp      dbp
height 1.00000000 0.5049999 0.01716543 0.03305511
weight 0.50499987 1.0000000 0.26780039 0.40531170
sbp     0.01716543 0.2678004 1.00000000 0.75608662
dbp     0.03305511 0.4053117 0.75608662 1.00000000
```

Rank based correlation

The same function is also used to calculate Spearman's correlation which is based on the correlation of the ranks.

```
cor(norsjo86$sbp, norsjo86$dbp, use="complete.obs", method="spearman")
[1] 0.7748761
```

Test

We can also test if there is any correlation in the population $H_0 : \rho = 0$ or get a confidence interval.

```
cor.test(norsjo86$sbp,norsjo86$dbp) # by default NAs are removed

Pearson's product-moment correlation

data: norsjo86$sbp and norsjo86$dbp
t = 18.572, df = 255, p-value < 2.2e-16
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 0.7009239 0.8058481
sample estimates:
      cor
0.7582536
```

A scatterplot can be used to study the pairwise relationship. In the case with sbp and dbp the correlation is high and we can clearly see it in the plot.

```
plot(norsjo86$sbp,norsjo86$dbp,cex=0.7)
```

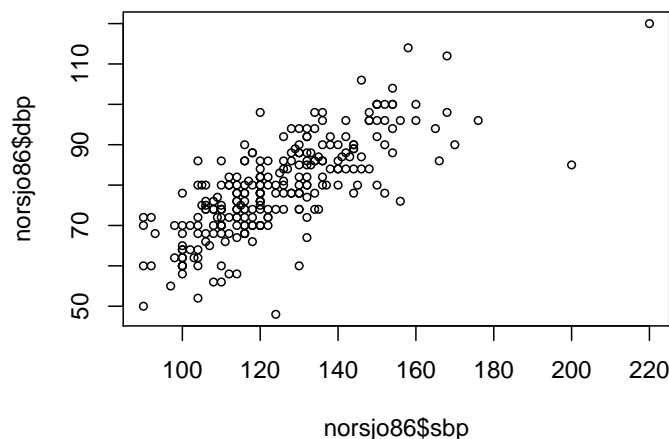


Figure 15.1:

Own experimentation

You may for example calculate Pearson and Spearman's correlation for other variables or subsets and test if $\rho = 0$. Can you test the correlation using Spearman's correlation -check help?

15.2 Test and confidence intervals

What is a statistical test? First of all we need to set up hypotheses; first a so called null hypothesis. It is often about the expected value (theoretical mean value or population mean). Your data can be thought of as a sample from a larger population. If you for example have measured systolic blood pressure in 30 randomly chosen individuals you can calculate the sample mean (\bar{x}). However, if someone else

would do the same thing he/she would end up with a different sample and will also get a different sample mean. The sample mean can be thought of as an estimation of the true theoretical mean in the full population. We cannot observe the theoretical mean because we don't have data on all individuals or we can theoretically have an infinite population. Thus the sample includes observations that are randomly chosen from the population. If the sample includes the full population we don't need statistics, we can then just calculate the population mean.

The null hypothesis, called H_0 is usually what we don't believe is true. We actually want a decision: can we conclude that H_0 is false? If H_0 is false then the alternative hypothesis called H_1 must be true. This is the first step of a test. Assume we want to test if the population mean of systolic blood pressure are equal between men and women, then the null hypothesis may be $H_0 : \mu_M = \mu_W$ where μ_M and μ_W are the population means in men and women. The alternative hypothesis may be $H_1 : \mu_M \neq \mu_W$. Thus if we reject H_0 then H_1 is true. Otherwise we may keep on to H_0 although we cannot be sure it is correct.

To test H_0 we try to find a test statistic, let us call it TS (it can for example be based on the difference of the sample means $\bar{x}_M - \bar{x}_W$) for which the distribution is known if H_0 is true. We would expect the difference of the sample means $\bar{x}_M - \bar{x}_W$ to be close to zero if H_0 is true. If H_0 is false we would expect the difference to be further away from zero (either negative or positive). The question is how far from zero should it be for us to decide that H_0 is false. This is where the known distribution of the test statistic comes in. If H_0 is true we can then calculate the probability to get the TS we got or even more extreme given H_0 is true. If this probability is low we will not believe in H_0 and reject it. The probability is called **p-value**. The limit for when a p-value is regarded as statistically significant is called **significance level** and is most often chosen as 5%

The method to calculate a so called **confidence interval** is similar, utilizing that the distribution of TS is known if H_0 is true. In the example above a 95% confidence interval for $\mu_M - \mu_W$ is an estimate that with 95% probability covers the true $\mu_M - \mu_W$

We may have alternative hypotheses saying for example $H_1 : \mu_M > \mu_W$. Then we are still interested if we can reject H_0 but we are only interested if the mean of men is larger than for women. For a confidence interval it will then be one-sided so that the upper limit is infinity.

15.2.1 T-test

T-test is perhaps the most common statistical test. However it can be performed in different ways; independent two-sample, one-sample or pairwise test. They can also be two-sided or one-sided which has to do with the alternative hypothesis. For two-sample test we can choose between assuming the standard deviations to be equal (t-test) or not (Welch's test) in the two samples. The nice thing in R is that we can use the same function for all different tests just by varying the arguments. The result also includes confidence interval. The confidence level is 95% by default but can be changed using the argument `conf.level = 0.95`.

There are however some requirements for the test to be correctly performed.

- The observations have to be independent
- The observations have to be normally distributed or
- the sample size should exceed 30 per group

15.2.2 Two-sample t-test

We start by creating two groups; smokers and non-smokers as two vectors. We then test the null hypothesis: the population means of systolic blood pressure are the same for smokers and non-smokers.

There are two ways to use the `t.test` function; using two vectors or using the formula method. In the last method the groups are identified by a variable. Observe that you need to give the argument `var.equal=T` to get the two-sample t-test.

```
table(norsjo86$smoker)

 0    1
206  54

# construct two groups, smokers(1) and non-smokers(0)
sbp.smoker<-norsjo86$sbp[norsjo86$smoker==1]
sbp.nonsmoker<-norsjo86$sbp[norsjo86$smoker==0]
summary(sbp.nonsmoker)

  Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
 90.0   113.0   122.0   125.6   136.0   220.0     3

summary(sbp.smoker)

  Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 92.0   104.5   118.0   120.2   131.0   176.0

res.t<-t.test(sbp.nonsmoker,sbp.smoker,var.equal=T)
res.t

Two Sample t-test

data:  sbp.nonsmoker and sbp.smoker
t = 1.8528, df = 255, p-value = 0.06506
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -0.3373127 11.0676539
sample estimates:
mean of x mean of y
125.6059  120.2407
```

The result shows that we don't reject the null hypothesis because the p-value is 0.065 which is over 5%. Furthermore we see that the 95% confidence interval for the difference of the population means is $(-0.34, 11.07)$

```
class(res.t)

[1] "htest"

str(res.t)    # the result object is a list

List of 10
 $ statistic   : Named num 1.85
  .. attr(*, "names")= chr "t"
 $ parameter   : Named num 255
  .. attr(*, "names")= chr "df"
 $ p.value     : num 0.0651
 $ conf.int    : num [1:2] -0.337 11.068
  .. attr(*, "conf.level")= num 0.95
 $ estimate    : Named num [1:2] 126 120
  .. attr(*, "names")= chr [1:2] "mean of x" "mean of y"
 $ null.value  : Named num 0
```

```

..- attr(*, "names")= chr "difference in means"
$ stderr      : num 2.9
$ alternative: chr "two.sided"
$ method      : chr " Two Sample t-test"
$ data.name   : chr "sbp.nonsmoker and sbp.smoker"
- attr(*, "class")= chr "htest"

```

The other alternative is the formula method. The formula method (using a formula object) also makes it easier to analyse subgroups.

```

t.test(sbp~smoker,data=norsjo86,var.equal=T) # here the groups are separated by variable smoker

Two Sample t-test

data:  sbp by smoker
t = 1.8528, df = 255, p-value = 0.06506
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -0.3373127 11.0676539
sample estimates:
mean in group 0 mean in group 1
    125.6059      120.2407

t.test(sbp~smoker,data=subset(norsjo86,agegrp==50),var.equal=T) # we can select a subset

Two Sample t-test

data:  sbp by smoker
t = -0.44741, df = 60, p-value = 0.6562
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -18.20692  11.55094
sample estimates:
mean in group 0 mean in group 1
    129.4902      132.8182

```

15.2.3 One-sided t-test

For a one sided test the alternative hypothesis is either $H_1 : \mu_{NS} < \mu_S$ or $H_1 : \mu_{NS} > \mu_S$. Alternatively we may think of the hypothesis as $H_1 : \mu_{NS} - \mu_S > 0$ or $H_1 : \mu_{NS} - \mu_S < 0$. You can probably guess what NS and S stands for. Now check the difference compared to the two-sided test result.

```

res.t1<-t.test(sbp~smoker,data=norsjo86,var.equal=T,alternative="greater")
res.t1

Two Sample t-test

data:  sbp by smoker
t = 1.8528, df = 255, p-value = 0.03253
alternative hypothesis: true difference in means is greater than 0
95 percent confidence interval:
  0.5848424      Inf
sample estimates:
mean in group 0 mean in group 1
    125.6059      120.2407

```

```
res.t

Two Sample t-test

data:  sbp.nonsmoker and sbp.smoker
t = 1.8528, df = 255, p-value = 0.06506
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -0.3373127 11.0676539
sample estimates:
mean of x mean of y
 125.6059  120.2407
```

The result shows that we reject the null hypothesis because the p-value is 0.033 which is over 5%. Furthermore we see that the 95% confidence interval for the difference of the population means is $(0.58, \infty)$

15.2.4 Welch's test

This test is calculated similarly as a two-sample t-test but without assumption of equal standard deviations in the two groups. The default is "var.equal=F" so we don't need to give this argument.

```
t.test(sbp~smoker,data=norsjo86)

Welch Two Sample t-test

data:  sbp by smoker
t = 1.8184, df = 81.404, p-value = 0.07268
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -0.5049962 11.2353374
sample estimates:
mean in group 0 mean in group 1
 125.6059      120.2407
```

Own experimentation

Try to do a t-test using a different variable or a subset of the data. Try out one-sided test. Study the p-values and the confidence intervals.

15.2.5 One-sample t-test

For this test we only have one sample. Instead our hypotheses are based on the true mean in the population. $H_0 : \mu = \mu_0$ where μ_0 is the mean we want to test. In the examples below we start by the default $H_0 : \mu = 0$ and then $H_0 : \mu = 120$. When we use one-sample test the argument var.equal=T is not relevant.

```
t.test(sbp.smoker) # not realistic null hypothesis (default mu=0)

One Sample t-test

data:  sbp.smoker
```



```

t = 45.552, df = 53, p-value < 2.2e-16
alternative hypothesis: true mean is not equal to 0
95 percent confidence interval:
 114.9463 125.5352
sample estimates:
mean of x
 120.2407

t.test(sbp.smoker,mu=120,alternative="greater")

One Sample t-test

data:  sbp.smoker
t = 0.091201, df = 53, p-value = 0.4638
alternative hypothesis: true mean is greater than 120
95 percent confidence interval:
 115.8216      Inf
sample estimates:
mean of x
 120.2407

```

15.2.6 Paired t-test

So far we have dealt with independent observations. In the case we have paired data, for example observations before and after a treatment there is two observations for each individual. For each measurement there is measurement error but we can also think of a random error for the differences between individuals. Therefore we cannot assume the observations are independent and we should then use a paired test.

We use the Subliminal data for an example.

In an intervention study 18 students were randomized to receive either of two messages with the intention to see if this would affect their performance on the mathematics exam. The control group received neutral messages whereas the intervention group received messages confirming their learning process. All students participated in a summer school in mathematics and were tested at the beginning and end of the intervention.

The dataset contains the following variables:

| | |
|--------------|---|
| Message: | If the student received neutral or confirmatory (positive) messages |
| Before: | Test result at the beginning of the study |
| After: | Test result at the end of the study |
| Improvement: | Improvement of their results (AfterBefore) |

```

# paired t-test
library(haven)
Subliminal <- read_sav("../data/Subliminal.sav") ## find data set
Subliminal

# A tibble: 18 x 4
  Message Before After Improvement
  <chr>    <dbl> <dbl>      <dbl>
1 positive    18    24         6
2 positive    18    25         7
3 positive    21    33        12

```

| | | | |
|-------------|----|----|----|
| 4 positive | 18 | 29 | 11 |
| 5 positive | 18 | 33 | 15 |
| 6 positive | 20 | 36 | 16 |
| 7 positive | 23 | 34 | 11 |
| 8 positive | 23 | 36 | 13 |
| 9 positive | 21 | 34 | 13 |
| 10 positive | 17 | 27 | 10 |
| 11 neutral | 18 | 29 | 11 |
| 12 neutral | 24 | 29 | 5 |
| 13 neutral | 20 | 24 | 4 |
| 14 neutral | 18 | 26 | 8 |
| 15 neutral | 24 | 38 | 14 |
| 16 neutral | 22 | 27 | 5 |
| 17 neutral | 15 | 22 | 7 |
| 18 neutral | 19 | 31 | 12 |

```
t.test(Subliminal$After,Subliminal$Before,paired=T) # Here we compare the two variables (repeated in
```

```
Paired t-test
```

```
data: Subliminal$After and Subliminal$Before
```

```
t = 11.534, df = 17, p-value = 1.839e-09
```

```
alternative hypothesis: true difference in means is not equal to 0
```

```
95 percent confidence interval:
```

```
8.170857 11.829143
```

```
sample estimates:
```

```
mean of the differences
```

```
10
```

```
# what it actually do is however nothing but a one-sample t-test of the null hypothesis:
```

```
# the population mean of the individual differences is zero
```

```
t.test(Subliminal$After-Subliminal$Before,var.equal=T)
```

```
One Sample t-test
```

```
data: Subliminal$After - Subliminal$Before
```

```
t = 11.534, df = 17, p-value = 1.839e-09
```

```
alternative hypothesis: true mean is not equal to 0
```

```
95 percent confidence interval:
```

```
8.170857 11.829143
```

```
sample estimates:
```

```
mean of x
```

```
10
```

15.3 Nonparametric tests

Nonparametric alternatives to t-test like Wilcoxon rank sum test (Mann Witney U test) and Wilcoxon signed rank test are shown below. They all use sums of the ranks to calculate the test statistic instead of using the observed values.

The reason for doing non-parametric test is e.g. when the distribution is not normal and we have less than 30 observations in each group.

15.3.1 Wilcoxon rank sum test (Mann-Whitney test)

This is the non-parametric version of two sample t-test. Also here we have two alternatives, with or without using formula.

```
wilcox.test(sbp.nonsmoker,sbp.smoker) # alternative with two vectors

Wilcoxon rank sum test with continuity correction

data: sbp.nonsmoker and sbp.smoker
W = 6505, p-value = 0.03486
alternative hypothesis: true location shift is not equal to 0

wilcox.test(sbp~smoker,data=norsjo86) # alternative using formula

Wilcoxon rank sum test with continuity correction

data: sbp by smoker
W = 6505, p-value = 0.03486
alternative hypothesis: true location shift is not equal to 0
```

15.3.2 Wilcoxon signed rank test

This is the non-parametric version of a paired t-test.

```
wilcox.test(Subliminal$After,Subliminal$Before,paired=T,exact=F)

Wilcoxon signed rank test with continuity correction

data: Subliminal$After and Subliminal$Before
V = 171, p-value = 0.0002111
alternative hypothesis: true location shift is not equal to 0
```

The non-parametric alternatives are the same as for t-test.

Own experimentation

For example you can compare confidence intervals and p-values for t-test the non-parametric alternative for another variable or a subset (two-sample t-test vs Mann-Whitney test).

15.4 Chi-2 test and Fishers exact test

Here the null hypothesis is that there is no relationship between the column and row variables.

15.4.1 Aggregated data

We start with an example where the relationship between back pain and weight increase during pregnancy was studied. In total 180 pregnant women were asked whether they experienced pain in the back during pregnancy or not. Further their weight increase during pregnancy were measured. We already have the aggregated data.

| Weight increase | Pain | No pain |
|-----------------|------|---------|
| < 15 | 22 | 12 |
| > 15 | 131 | 15 |

The data are prepared in aggregated form in a two by two table (matrix).

```
# chi-2 test (aggregated data)
# generate a aggregated table
pregpain<-matrix(c(22,131,12,15),nrow=2)
rownames(pregpain)<-c("weight increase <15","weight increase >15")
colnames(pregpain)<-c("pain","no pain")
pregpain # pregpain is not a table

      pain no pain
weight increase <15    22    12
weight increase >15  131    15

pregpain<-as.table(pregpain)
prop.table(pregpain,margin=1) # the risk of pain seems to be higer with high weight increase

      pain    no pain
weight increase <15 0.6470588 0.3529412
weight increase >15 0.8972603 0.1027397

# it works with pregpain as a data frame or matrix too
res.c<-chisq.test(pregpain)
res.c

Pearson's Chi-squared test with Yates' continuity correction

data:  pregpain
X-squared = 11.649, df = 1, p-value = 0.0006424
```

The p-value is lower than 5% so the null hypothesis will be rejected. Thus we can conclude that weight increase and pain in the back is related.

```
class(res.c)

[1] "htest"

str(res.c)

List of 9
 $ statistic: Named num 11.6
  .. attr(*, "names")= chr "X-squared"
 $ parameter: Named int 1
  .. attr(*, "names")= chr "df"
 $ p.value   : num 0.000642
 $ method    : chr "Pearson's Chi-squared test with Yates' continuity correction"
 $ data.name : chr "pregpain"
 $ observed  : 'table' num [1:2, 1:2] 22 131 12 15
  .. attr(*, "dimnames")=List of 2
  .. ..$ : chr [1:2] "weight increase <15" "weight increase >15"
  .. ..$ : chr [1:2] "pain" "no pain"
 $ expected : num [1:2, 1:2] 28.9 124.1 5.1 21.9
  .. attr(*, "dimnames")=List of 2
  .. ..$ : chr [1:2] "weight increase <15" "weight increase >15"
  .. ..$ : chr [1:2] "pain" "no pain"
```

```

$ residuals: 'table' num [1:2, 1:2] -1.284 0.619 3.055 -1.474
..- attr(*, "dimnames")=List of 2
.. ..$ : chr [1:2] "weight increase <15" "weight increase >15"
.. ..$ : chr [1:2] "pain" "no pain"
$ stdres : 'table' num [1:2, 1:2] -3.68 3.68 3.68 -3.68
..- attr(*, "dimnames")=List of 2
.. ..$ : chr [1:2] "weight increase <15" "weight increase >15"
.. ..$ : chr [1:2] "pain" "no pain"
- attr(*, "class")= chr "htest"

res.c$expected

              pain no pain
weight increase <15  28.9    5.1
weight increase >15 124.1   21.9

res.c$observed

              pain no pain
weight increase <15   22    12
weight increase >15  131    15

res.c$observed-res.c$expected

              pain no pain
weight increase <15  -6.9    6.9
weight increase >15   6.9   -6.9

```

The test is based on the differences between observed and expected numbers.

15.4.2 Individual data

Here we use the `norsjo86` data set for another example to study the relationship between BMI (split into low and high) and smoking.

We start by aggregating the individual data to a 2 by 2 table as above and then we use individual data directly in the same function. The results are the same.

```

norsjo86$bmic<-cut(norsjo86$bmi,breaks=c(0,25,Inf),labels=c("<25",">25")) # individual data

aggtab<-table(norsjo86$smoker,norsjo86$bmic) # first aggregate
aggtab

  <25 >25
0  112  90
1   32  22

aggtab<-xtabs(~smoker+bmic,data=norsjo86) # alternative table
aggtab

      bmic
smoker <25 >25
0      112  90
1       32  22

res1<-chisq.test(aggtab) # chi-square test
res1

```

```

Pearson's Chi-squared test with Yates' continuity correction

data:  aggtab
X-squared = 0.1207, df = 1, p-value = 0.7283

# We need not do this procedure - using individual data works as well
res2<-chisq.test(norsjo86$smoker,norsjo86$bmic)
res2

Pearson's Chi-squared test with Yates' continuity correction

data:  norsjo86$smoker and norsjo86$bmic
X-squared = 0.1207, df = 1, p-value = 0.7283

res2$observed # compare with aggtab

              norsjo86$bmic
norsjo86$smoker <25 >25
0      112    90
1       32    22

```

Thus we can use either aggregated or individual data in the same function.

15.4.3 Fishers exact test

This test can be useful with small samples. While the p-value from a chi-square test is an approximation (asymptotic result based on normal approximation) Fishers exact test is always correct. Here we need another R function. The function `fisher.test` also gives odds ratio and its confidence interval.

```

res.fa<-fisher.test(aggtab) # aggregated data
res.fi<-fisher.test(norsjo86$smoker,norsjo86$bmic) # Individual data
res.fa

Fisher's Exact Test for Count Data

data:  aggtab
p-value = 0.6461
alternative hypothesis: true odds ratio is not equal to 1
95 percent confidence interval:
 0.4409047 1.6395816
sample estimates:
odds ratio
 0.856075

res.fi

Fisher's Exact Test for Count Data

data:  norsjo86$smoker and norsjo86$bmic
p-value = 0.6461
alternative hypothesis: true odds ratio is not equal to 1
95 percent confidence interval:
 0.4409047 1.6395816
sample estimates:
odds ratio
 0.856075

```

15.4.4 Tests in n by k tables

As an example we test if there is a relationship between age (4 categories) and BMI (2 categories).

```
t22<-table(norsjo86$agegrp,norsjo86$bmic)
round(prop.table(t22,margin=1),2)

      <25  >25
30 0.69 0.31
40 0.59 0.41
50 0.48 0.52
60 0.48 0.52

chisq.test(norsjo86$agegrp,norsjo86$bmic)

Pearson's Chi-squared test

data:  norsjo86$agegrp and norsjo86$bmic
X-squared = 7.6546, df = 3, p-value = 0.05372
```

Close but no significant difference of BMI between the age groups.

Alternative

We can also use summary to calculate chi-square test. Here demonstrated with xtabs but it also works for table.

```
summary(xtabs(~agegrp+bmic, data=norsjo86))

Call: xtabs(formula = ~agegrp + bmic, data = norsjo86)
Number of cases in table: 256
Number of factors: 2
Test for independence of all factors:
Chisq = 7.655, df = 3, p-value = 0.05372
```

15.4.5 Count data in one dimension

It is also possible to do chi-square tests on one variable. Then the aim is not to test relationship but to test if the observations all have the same expected number i.e the observed variation is only random (not systematic). Assume we look at the number of sold houses per month January to June a certain year. Can we be sure that there is a trend or can the numbers be explained by chance?

```
x<-c(17,14,18,26,19,24)
x

[1] 17 14 18 26 19 24

chisq.test(x)

Chi-squared test for given probabilities

data:  x
X-squared = 5.1525, df = 5, p-value = 0.3975
```

15.5 Kolmogorov-Smirnov test

The purpose of this test and plot is to check if a distribution follows a certain distribution. In this example we test if systolic blood pressure in age group 50 years follows a normal distribution. Thus it is important to give the correct parameters as arguments.

```
# KS test
sbp50<-norsjo86$sbp[norsjo86$agegrp==50]

# standardise the observations
m<-mean(sbp50,na.rm=T)
s<-sd(sbp50,na.rm=T)
sbp.st<-(sbp50-m)/s

ks.test(sbp.st,"pnorm") # compare standardised observations to standard normal

One-sample Kolmogorov-Smirnov test

data:  sbp.st
D = 0.10468, p-value = 0.5052
alternative hypothesis: two-sided

ks.test(sbp50,"pnorm",m,s) # compare original observations to normal(mean=m,sd=s)

One-sample Kolmogorov-Smirnov test

data:  sbp50
D = 0.10468, p-value = 0.5052
alternative hypothesis: two-sided

ks.test(sbp50[!duplicated(sbp50)],"pnorm",m,s) # Ties removed (quite a lot)

One-sample Kolmogorov-Smirnov test

data:  sbp50[!duplicated(sbp50)]
D = 0.077041, p-value = 0.9685
alternative hypothesis: two-sided
```

There was a lot of ties and the p-value was quite different when the ties were removed. However, the conclusion was the same: we don't reject the hypothesis that the observations follow a normal distribution. The function `pnorm()` can be found in the help. We can choose any of these functions for the test.

15.6 Q-Q plot

In a Q-Q plot the quantiles of the observed variable is plotted on the y-axis and the expected quantiles with a normal distribution is plotted on the x-axis.


```
qqnorm(sbp50, cex=0.7)  
qqline(sbp50, col=2)
```

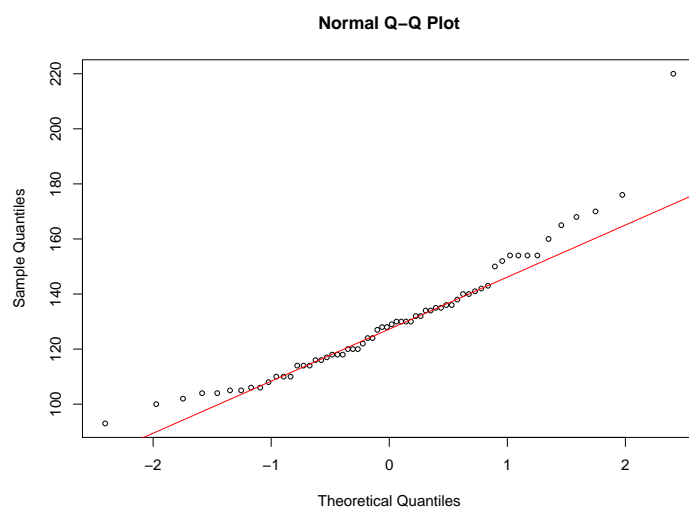


Figure 15.2: Q-Q plot

Chapter 16

ggplot2

16.1 ggplot2

ggplot2 is a very popular package that offers enhanced plotting functionality. It uses a clever way to map numbers to visual representations in plots. Visual representations include for example lines, bars, dots. By varying attributes such as colour, shape or size, even more information can be visualized. *ggplot2* uses the following conceptual framework:

- *Data*: is passed as a dataframe to the *ggplot()* function.
- *Geoms*: points, lines, bars etc are the types of geometric objects that are added in layers to the plot to represent the data
- *Aesthetics*: these serve as the "connection" between the geoms and the data. They map visual properties of the geom, such as (x,y) coordinates, line colour, shapes etc to the variables represented by columns of the dataframe. For example, the y coordinate of lines points may be mapped to a variable in the dataframe.

ggplot2 plots always start with the *ggplot()* function. Then, geoms can be added to the plot by putting a '+' after the () brackets, followed by the geom functions. Geom functions start with *geom_*, for example *geom_line*. The data and the aesthetics are specified as arguments of either the *ggplot()* or the *geom_* functions. The data is provided using the 'data=' argument, and the aesthetics using the 'mapping=' argument followed by the *aes()* function.

If the data and mappings are provided in the *ggplot()* function, they are globally available in all the following *geom_* functions. If provided in a *geom_* function, the data and the aesthetic are defined only for the respective geom.

ggplot2 way of plotting enables to quickly construct complex plots because of flexibility and modularity.

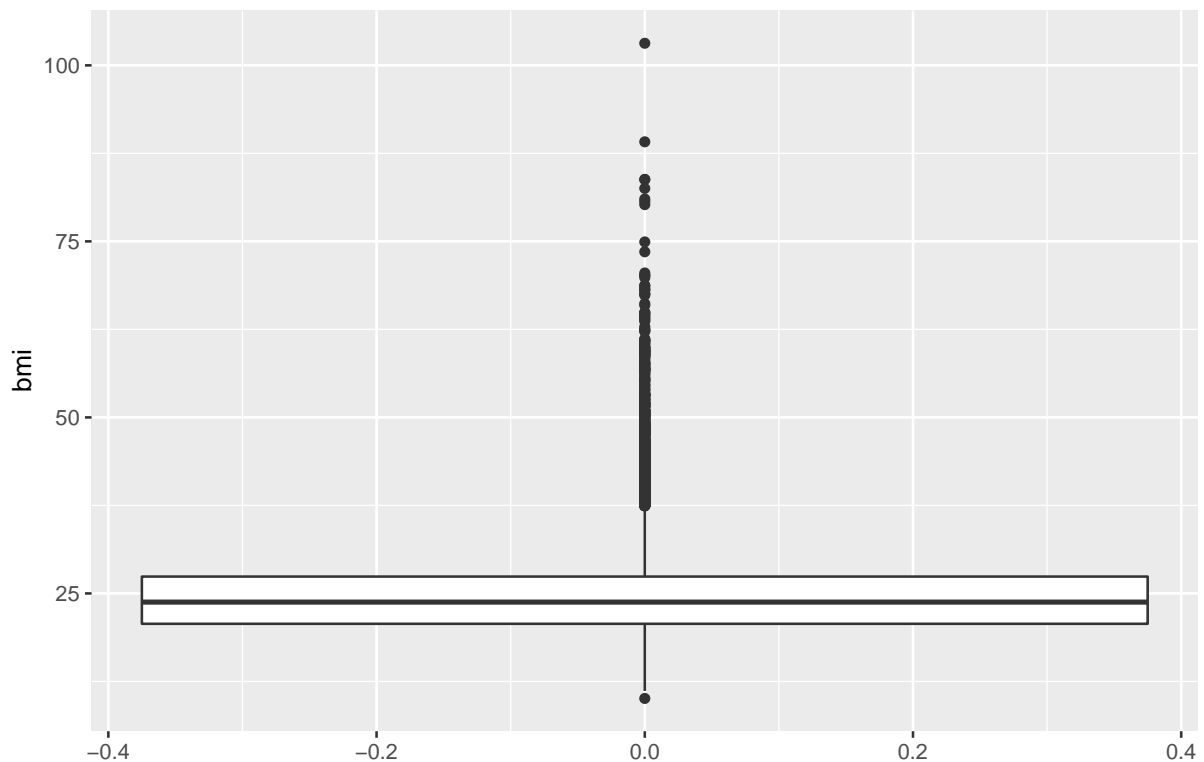
16.2 Plotting distribution of single variables

At the beginning of a statistical analysis it is useful to get an idea of what values are present how often in the data for the different variables. This for example could help to identify outliers, to check assumptions about normally distributed data, or simply to verify if the data has been loaded correctly.

In a first example we create boxplots. Note the *aes()* function that designates *bmi* to be the value that is visualized on the y axis.

```
library(ggplot2)
d <- read.csv("../data/BackPain.csv")

ggplot(data = d, mapping= aes(y = bmi)) +
  geom_boxplot(na.rm = T) # boxplot, remove bmi values that are NA
```

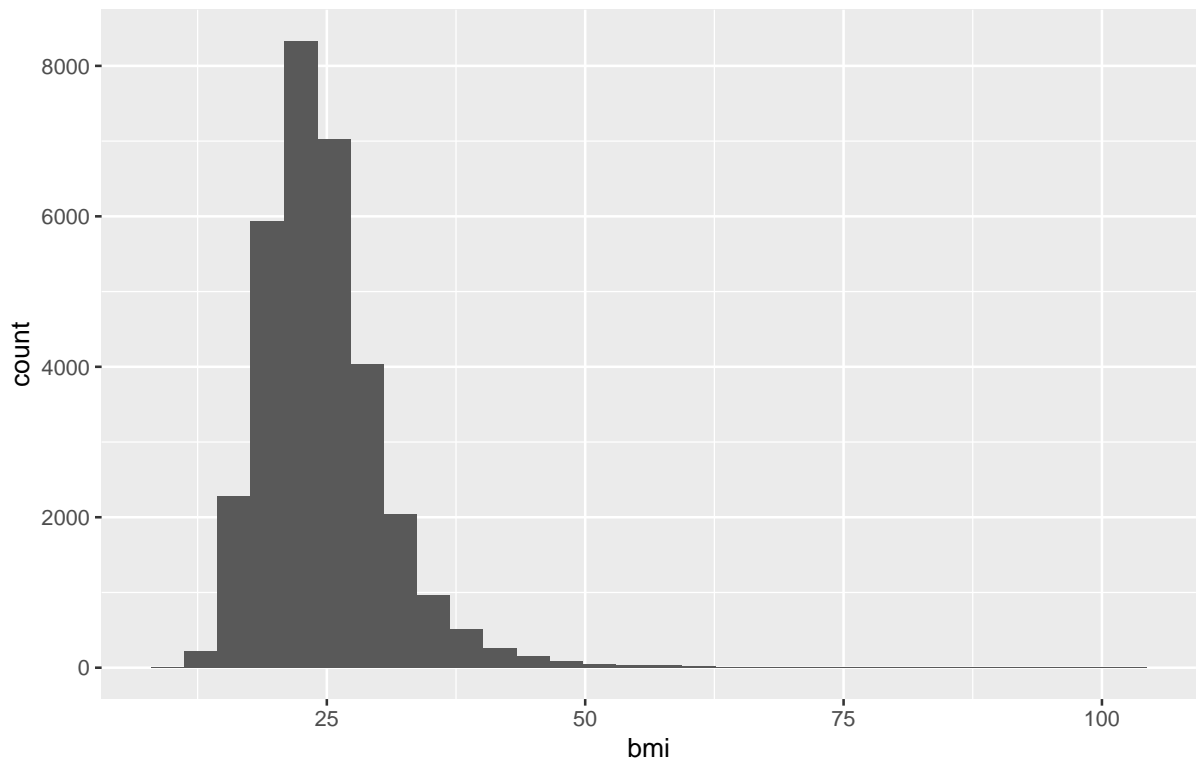


The data and the aesthetic mappings are globally defined in the function `ggplot()`. The `geom_boxplot()` function automatically applies this mapping to the provided data. In addition, NAs are removed

Next, consider a histogram that may offer an even better picture of a distribution.

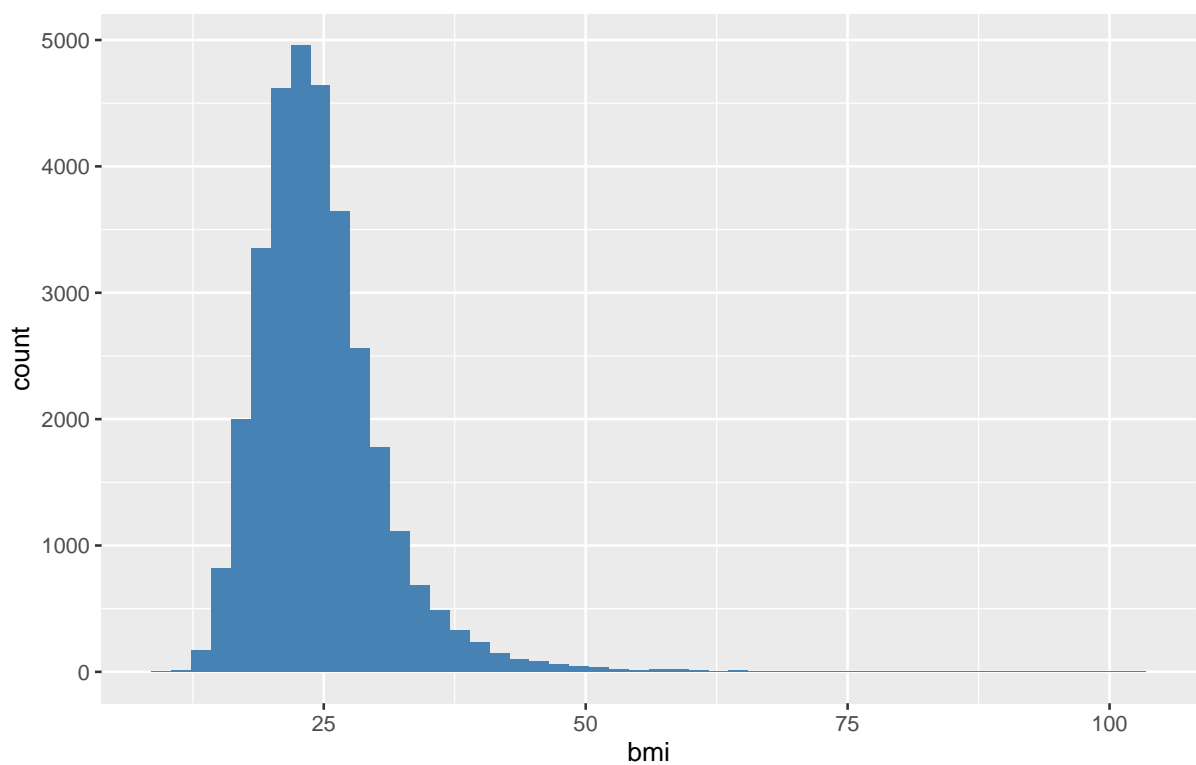
```
# complains about binwidth
ggplot(data = d, mapping= aes(x = bmi)) +
  geom_histogram(na.rm = T)

## 'stat_bin()' using 'bins = 30'. Pick better value with 'binwidth'.
```



If you try to run this code, you would see that `ggplot` complains that you did not pick a good value for 'binwidth'. Let's correct that and manually set a fill color. If you manually set visual properties such as the color, that is without mappings, this should generally happen directly in the `geom` functions outside the `aesthetic()` functions:

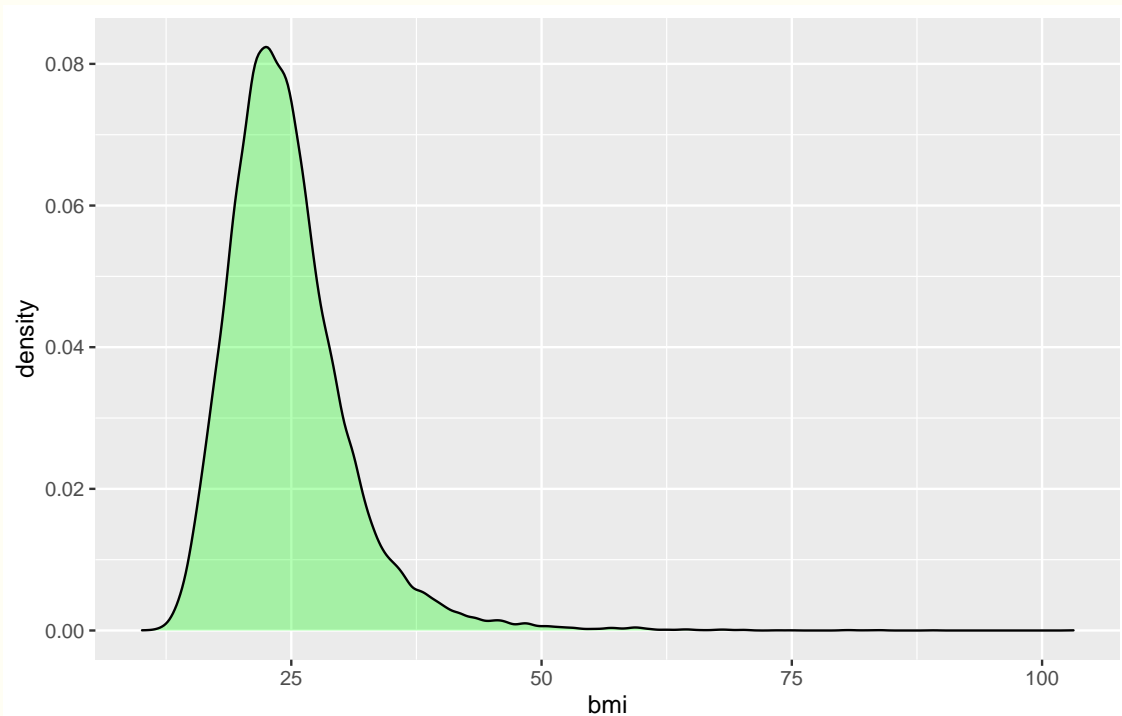
```
# manually define color and set number of bins
ggplot(data = d, mapping = aes(x = bmi)) +
  geom_histogram(na.rm = T, bins = 50, fill = "steelblue")
```



`geom_density()` works similarly as histograms and displays "smooth" histograms.

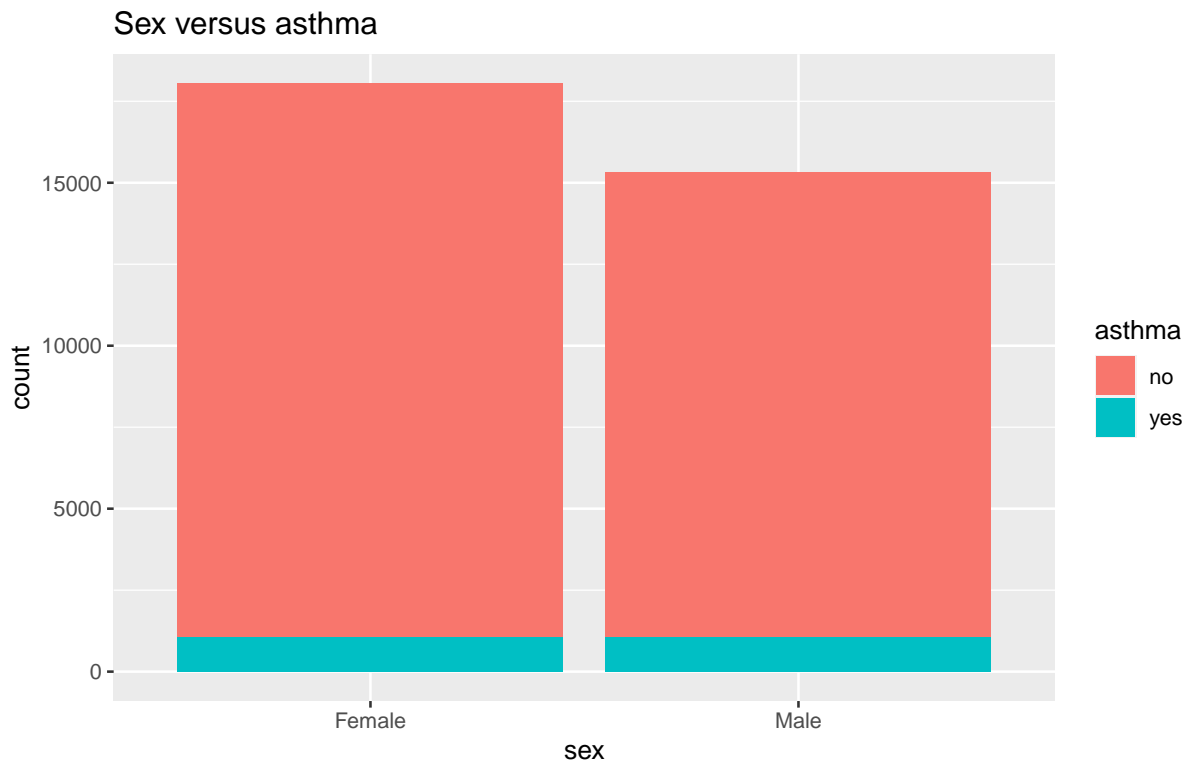
Own experimentation

Try to use `geom_density()` to make a density plot of `bmi`, like so (Hint: set the *alpha* argument):



Now we use barplots to demonstrate how to group sex by colors and how to manually set a title:

```
# A barplot and how to set a plot title
ggplot(data = d[d$asthma != "",], mapping= aes(x = sex, fill = asthma))+
  geom_bar(na.rm = T) +
  ggtitle("Sex versus asthma")
```



16.3 Relationship of variables

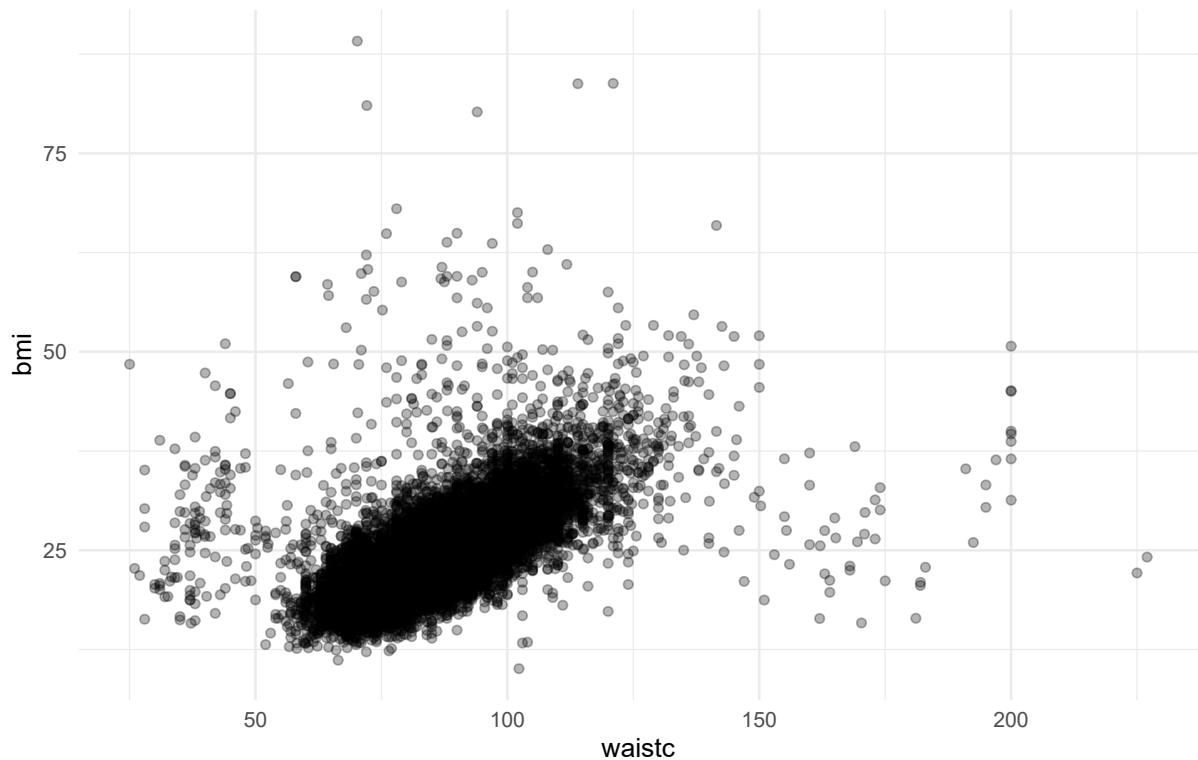
Relationship between different variables are often at the heart of statistical analyses. Visualization of these relationships is crucial to communicate results in an intuitive, understandable manner.

First we prepare a subset of the data to avoid having to deal with NA, and we infer the weight from bmi and height:

```
ds = d[,c("age", "bmi", "waistc", "height", "sex")] # the columns we want to work with
ds$weight <- ds$bmi * (ds$height/100)^2 # recover weight value using the BMI formula
ds <- ds[complete.cases(ds),] # get rid of rows with any NA
```

We start with a simple scatterplot. GGplot also provides several themes to alter the general look of plots. Here we use the minimal theme, which provides a white background. We set *alpha* to 0.3 to better handle overlapping dots by making them transparent:

```
# A points or scatterplot with the 'minimal' theme
ggplot(aes(x = waistc, y = bmi), data = ds)+
  geom_point(alpha = 0.3)+
  theme_minimal()
```

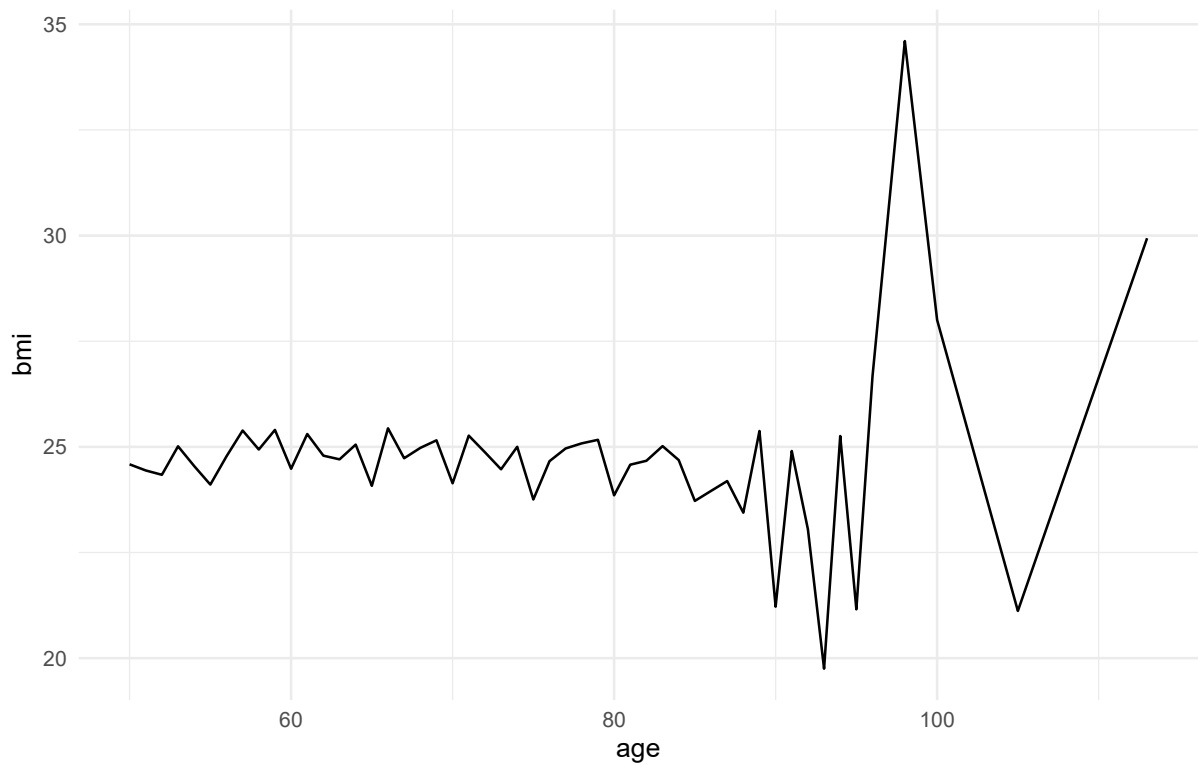


Bmi and waist circumference seem to correlate pretty well.

If we want to plot the mean bmi by age, we first need to calculate it. For that we use the `aggregate()` function that works similarly to `xtabs()`. It creates stratified data according to the formula provided and then applies the provided function to each cell. Here, we categorize the data by age, and then apply the `mean()` function for every strata. The stratification here uses that age is an integer variable. The plot is then stored in the gg object before displaying.

```
da = aggregate(bmi ~ age, ds, mean) # mean of bmi by age (mean function passed as object)
gg = ggplot(aes(x=age, y = bmi), data = da)+
  geom_line() # line plot

# only now plot the plot and apply the theme
gg + theme_minimal()
```

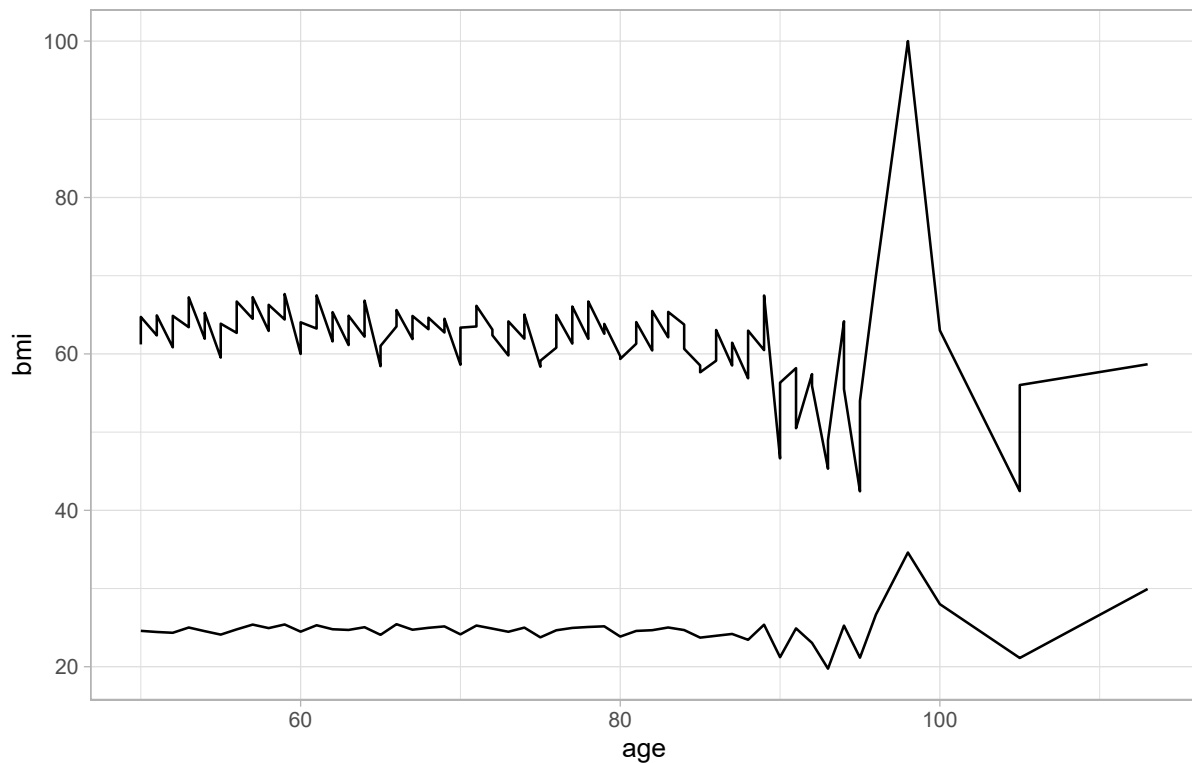



Above an age of 90, the data becomes scarce and the mean is influenced by stark variation.

Thanks to the modular nature of ggplot, it is easy to add things to existing plot. Lets add a line for the weight and change the theme. Note that the new data needs to be provided to the `geom_line()` separately.

```
# calculate mean weight for every age
da2 = aggregate(weight ~ age+sex, ds, mean)

# add a new line based on the new data to the old plot
gg+geom_line(aes(x = age, y = weight), data = da2)+
  theme_light()
```



16.4 Scales and legends

ggplot2 of course also offers large flexibility in customizing how x and y axes, as well as the plot legend look like. To do that, ggplot2 uses *scales*. Scales define how the data is mapped to aesthetics and how the axes of the plot look like.

There are different type of scales in ggplot2. The main distinction is into continuous and discrete scales.

16.4.1 Continuous scales

`scale_x_continuous()` and `scale_y_continuous()` are the principal functions to modify continuous scales, for x and y axes respectively. Here a list of useful arguments:

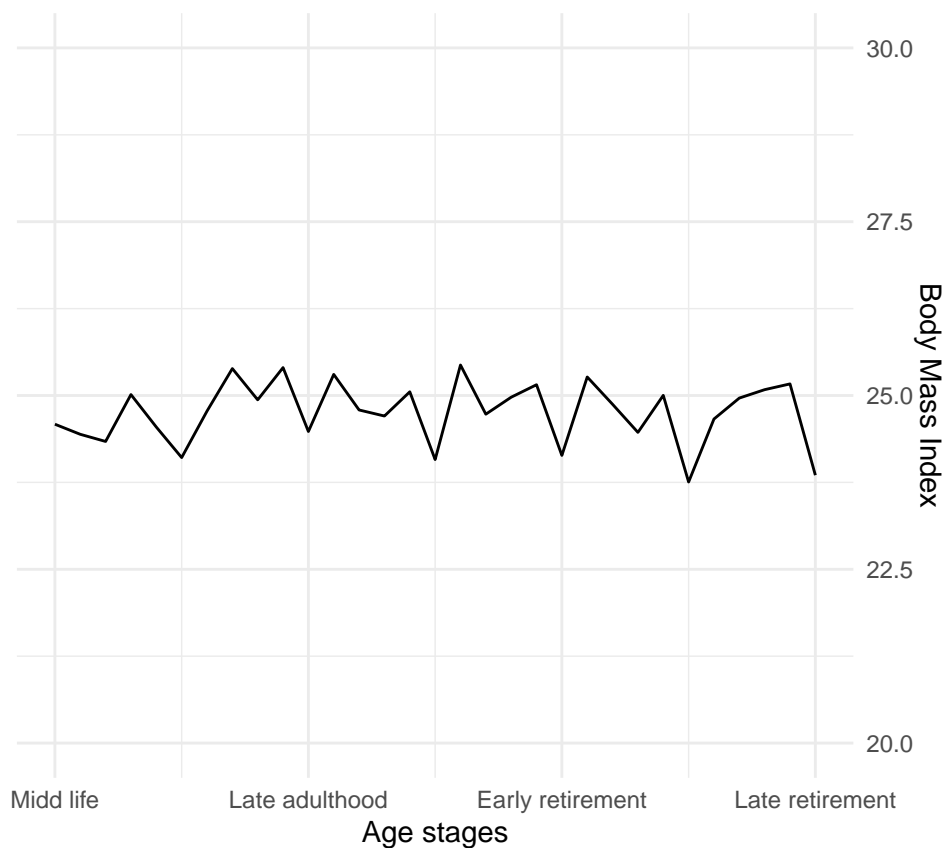
- **name:** Specifies the axis title
- **breaks:** Specifies which values are denoted with axis tickmarks in the form of a vector.
- **labels:** The names for the breaks.
- **limits:** Takes a vector of length 2 whose values indicate the beginning and the end of the axis
- **trans:** Sets a transformation for this axis. Valid values are for example "log", "sqrt", "reverse", "logit". (See the documentation for a full list)
- **position:** Defines the position of the axis. For x, it is either "left" or "right", for y either "top" or "bottom".

The full list of arguments is accessible in the documentation.

For an example, consider again the plot of average weight by age. Lets restrict the plot to ages ≤ 80 years by customizing the axis. Also set titles and restrict the y axes to a bmi of 20 to 30.

```
# restrict x axis and set manual axis labels using scales
gg + scale_x_continuous(breaks = c(50,60,70,80),
  labels = c("Mid life", "Late adulthood",
    "Early retirement", "Late retirement"),
  limits = c(50, 80),
  name = "Age stages")+
  scale_y_continuous(limits = c(20,30),
    position = "right",
    name = "Body Mass Index") +
  theme_minimal()

## Warning: Removed 20 row(s) containing missing values (geom_path).
```



ggplot displays a warnign for the values that did not fit within the axis limits.

Note that this way of denoting the x axis may not result in a useful plot, it is likely that direct age numbers are more informative.

Alternatively, the `ylab()`, `xlab()` and `lims()` function can be used to specify axes names and limits.

Own experimentation

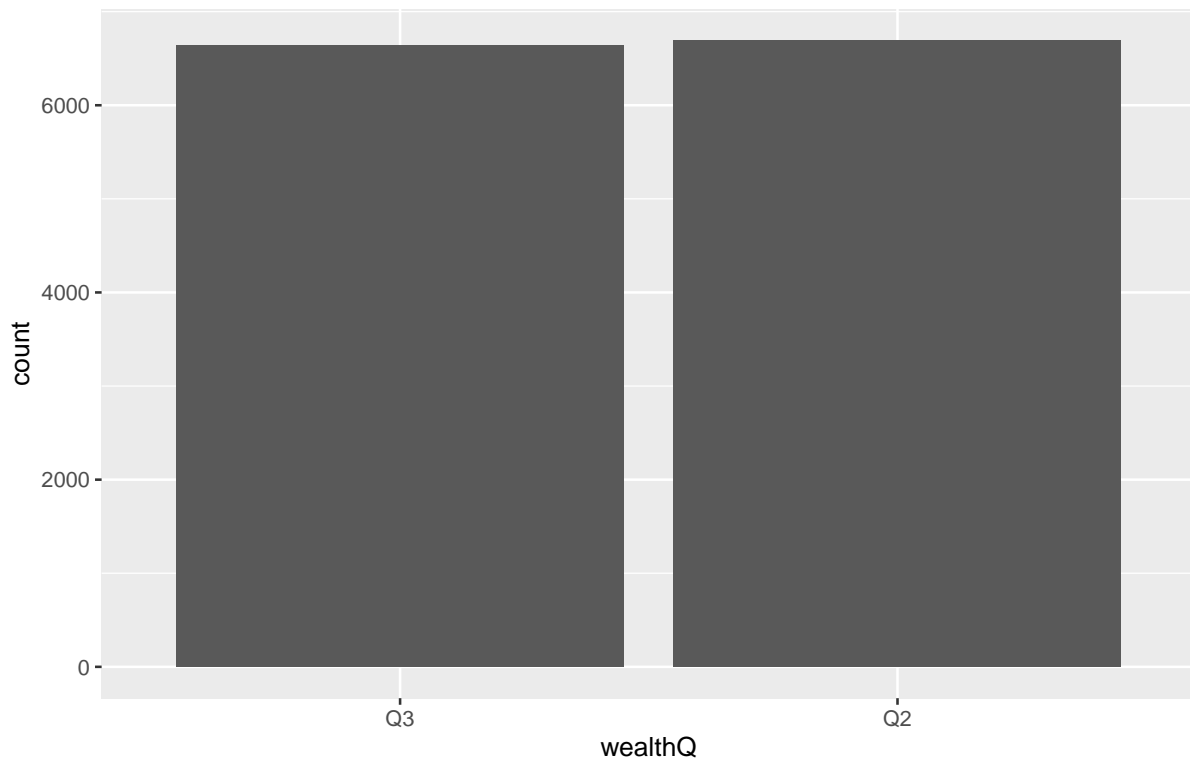
Use the `labs()` and `xlims()` functions to create a plot of average waist circumference for the ages 55 to 75 with a title and a subtitle.

16.4.2 Discrete scales

Axis manipulation of categorical plots is somewhat different. Instead of modifying the tickmarks, ggplot2 allows to manipulate which categories are displayed and how they are displayed. For example, lets create a barplot limited to Q2 and Q3 of *wealthQ*:

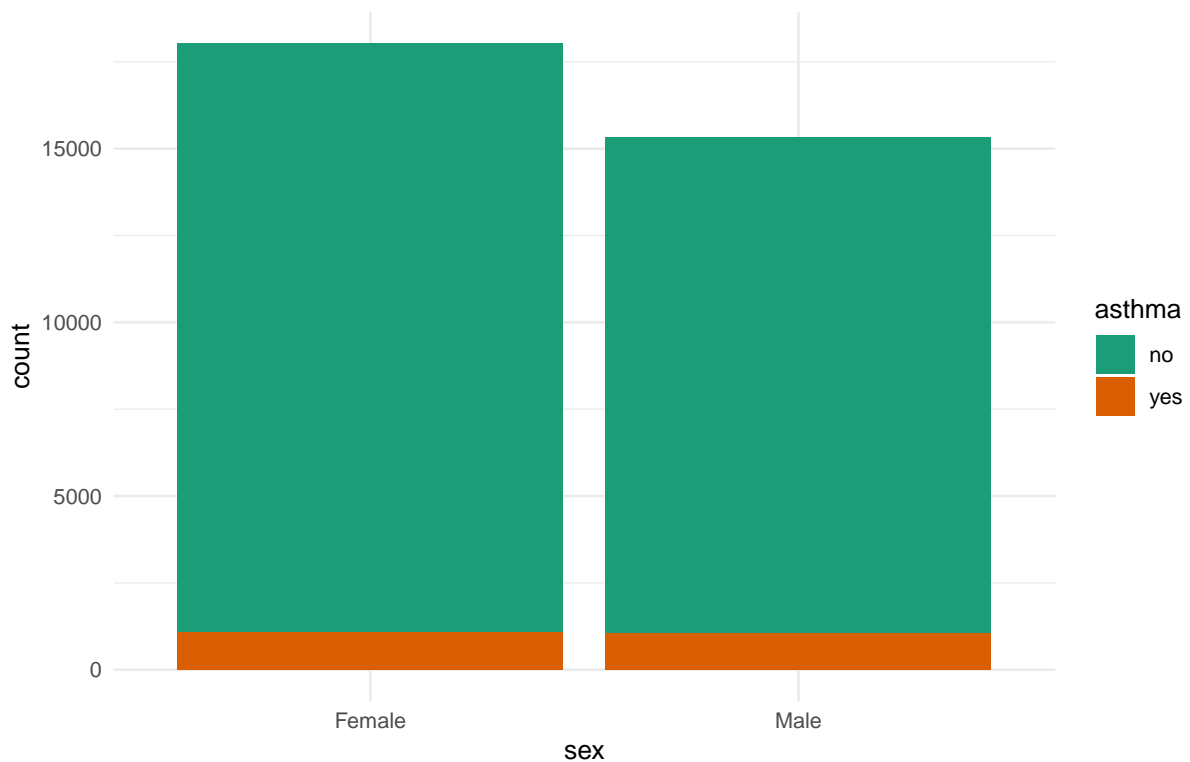
```
# only display two levels of categorical data
ggplot(data=d, mapping=aes(x=wealthQ))+
  geom_bar()+
  scale_x_discrete(limits=c("Q3","Q2"))

## Warning: Removed 20782 rows containing non-finite values (stat_count).
```



Consider again the barplot *asthma* and *sex*. In that case, ggplot uses discrete scales to map *sex* and *asthma* to the aesthetics of the bar geom. To modify the color, we need to change this discrete scale:

```
# use a different color palette by altering the fill scale
ggplot(data = d[d$asthma != "",], mapping= aes(x = sex, fill = asthma))+
  geom_bar(na.rm = T)+
  scale_fill_brewer(type="qual", palette = 2)+
  theme_minimal()
```



We specifically use `scale_fill_brewer` to specify a predefined color palette. This function provides access to predefined color palettes from ColorBrewer. See [here](#) for a list.

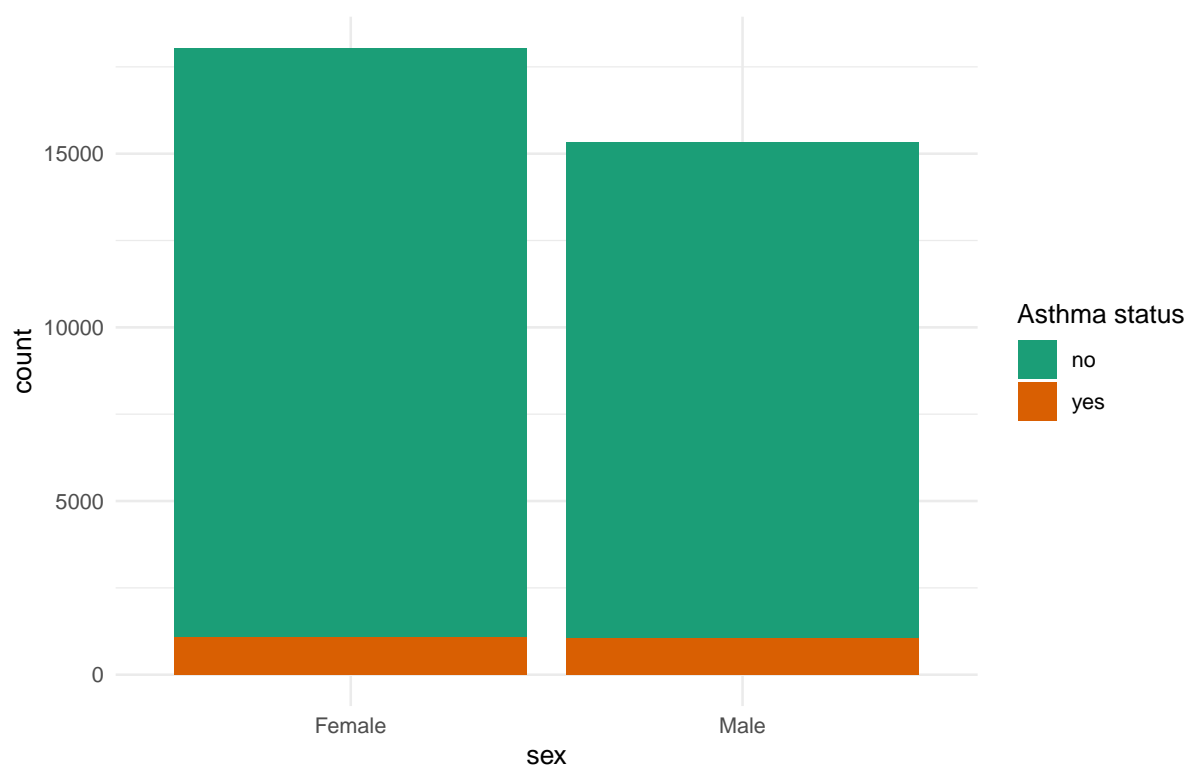
Own experimentation

Use the `scale_color_manual()` function to define two custom colors for the above plot. Hint: You need to use the *values* argument and set the *aesthetics* argument to "fill" to select the correct aesthetic.

16.4.3 Legends

Another concept in `ggplot2` are *guides*. Guides are the legends for the different aesthetic mappings. They can be specified either in the scale functions or as a separate part of a `ggplot` using the `guides()` function. In the following example we use this way to change the legend title from "asthma" to "Asthma status", in the same barplot example as above:

```
# an example using guides to change the legend title
ggplot(data = d[d$asthma != "",], mapping= aes(x = sex, fill = asthma))+
  geom_bar(na.rm = T)+
  scale_fill_brewer(type="qual", palette = 2)+
  guides(fill = guide_legend(title = "Asthma status"))+
  theme_minimal()
```



Many more options are available to customize and style legends and set legend types. See the `ggplot2` documentation for details.

Chapter 17

Functions in tidyverse

Håkan Jonsson < hakan.jonsson@umu.se >

17.1 The packages dplyr and tidyr - the core of tidyverse

These two packages are an important component of the tidyverse. The package dplyr was written by Hadley Wickham to try to simplify, speed-up and regularize commands and their arguments for manipulating data frames. The functions in **dplyr** operate on a dataframe (their first argument) and the output is a data frame. HW refers to these elementary functions as 'Single table verbs'. This allows us to 'pipe' a continuous sequence of commands with each one taking a dataframe as its first argument and passing its output (also a dataframe) to the next command in the pipe. Functions in dplyr are designed to form simple manipulations and in order to do more complicated tasks, the user constructs a sequence of function calls. This is like a 'grammar' of manipulation of data objects. The package is new (2014) and still under development but has been welcomed by the R community as a potentially unifying approach. There are many different ways of manipulating dataframes in R, with inconsistent organisation of arguments and outputs.

The package tidyr complements dplyr with functions that, in the words of Hadley Wickham, 'provides a bunch of tools to help tidy up your messy datasets'.

Note:

Many of these functions have corresponding functions in base R

17.1.1 Data set

We will use the data set BackPain which is a quite large data set both concerning the number of observations and 24 variables which means that it is a little difficult to get an overview. In the examples we will use a random sample (more about the **sample** function in the simulation section) of the full data set. The imported data set is a tibble so all variables are not printed. However this is enabled with the function **glimpse**.

```
BackPain<-read_csv("../data/BackPain.csv") # The resulting object is a tibble
BackPain
# A tibble: 34,122 x 25
```

```

      id residence sex      age wealthQ physical country backPain30 agegr maritalS eduS
    <dbl> <chr>   <chr> <dbl> <chr>   <chr>   <chr>   <chr>   <chr> <chr>   <chr>
  1     1 Urban   Fema~   66 Q3     high ph~ China   No     60-69 Div/Wid~ Comp~
  2     2 Urban   Fema~   63 Q4     mod phy~ China   No     60-69 Div/Wid~ Comp~
  3     3 Urban   Fema~   76 Q5 ric~ high ph~ China   No     70-79 Div/Wid~ No p~
  4     4 Rural   Male    64 Q4     high ph~ Mexico  No     60-69 Married~ Comp~
  5     5 Rural   Fema~   67 Q1 poo~ low phy~ China   No     60-69 Married~ No p~
  6     6 Urban   Fema~   74 Q4     mod phy~ China   No     70-79 Div/Wid~ Comp~
  7     7 Rural   Fema~   51 Q3     low phy~ South ~ No    50-59 Married~ No p~
  8     8 Urban   Fema~   80 Q5 ric~ low phy~ South ~ No    80+   Div/Wid~ Comp~
  9     9 Urban   Male    67 Q4     high ph~ China   No     60-69 Married~ No p~
 10    10 Urban   Male    75 Q5 ric~ mod phy~ China   No     70-79 Married~ Comp~
# ... with 34,112 more rows, and 14 more variables: workS <chr>, bmi <dbl>, bmi4 <chr>,
#   waistc <dbl>, smoke <chr>, alcohol <chr>, arthritis <chr>, angina <chr>,
#   depression <chr>, asthma <chr>, diabetes <chr>, comorb <dbl>, disability <dbl>,
#   height <dbl>

set.seed(1001)
bP<-BackPain[sample(nrow(BackPain),10000),] # let us take a subsample
# spara till senare bP %>% count(country,agegr)
tibble::glimpse(bP) # this is a way of getting an overview of the data

Observations: 10,000
Variables: 25
$ id          <dbl> 27043, 27471, 5323, 9068, 28828, 23775, 24138, 17074, 18398, 25789...
$ residence    <chr> "Urban", "Urban", "Rural", "Rural", "Urban", "Urban", "Rural", "Ur...
$ sex          <chr> "Female", "Female", "Female", "Female", "Female", "Female", "Femal...
$ age          <dbl> 57, 64, 60, 70, 58, 60, 67, 63, 60, 76, 71, 65, 65, 51, 52, 53, 62...
$ wealthQ      <chr> "Q3", "Q3", "Q2", "Q1 poorest", "Q5 richest", "Q1 poorest", "Q2", ...
$ physical     <chr> "mod phys act", "mod phys act", "high phys act", "low phys act", "...
$ country      <chr> "South Africa", "China", "China", "Ghana", "Mexico", "Mexico", "Ch...
$ backPain30   <chr> "Yes", "Yes", "No", "No", "Yes", "Yes", "Yes", "No", "No", "Yes", ...
$ agegr        <chr> "50-59", "60-69", "60-69", "70-79", "50-59", "60-69", "60-69", "60...
$ maritalS     <chr> "Married/Cohab", "Married/Cohab", "Married/Cohab", "Div/Wid/Sep", ...
$ eduS         <chr> "No primary", "No primary", "No primary", "No primary", "Compl Uni...
$ workS        <chr> "currently not working", "never worked", "currently working", "cur...
$ bmi          <dbl> 30.46875, 24.94150, 28.21869, 16.15668, 32.83567, 31.14878, 26.155...
$ bmi4         <chr> "Obese", "Normal", "Pre-Obese", "Underweight", "Obese", "Obese", "...
$ waistc       <dbl> 108.0, 82.0, 80.0, 86.0, 123.0, 101.5, 89.1, 68.0, 72.0, 87.4, 94....
$ smoke        <chr> "Never/Not Daily/Current", "Never/Not Daily/Current", "Never/Not D...
$ alcohol       <chr> "Abstainers", "Abstainers", "Abstainers", NA, "Abstainers", "Absta...
$ arthritis    <chr> "yes", "yes", "no", "no", "no", "yes", "yes", "no", "no", "no", "y...
$ angina       <chr> "yes", "no", "no", NA, "no", "no", "no", "no", "no", "no", "yes", ...
$ depression   <chr> "no", "no", "no", "yes", "no", "yes", "no", "no", "no", "no", "no"...
$ asthma       <chr> "no", "no", "no", "no", "no", "no", "no", "no", "no", "no", "no", ...
$ diabetes     <chr> "yes", "no", "no", "no", "yes", "no", "no", "no", "no", "no", "no"...
$ comorb       <dbl> 2, 1, 0, 1, 1, 2, 1, 0, 0, 0, 2, 2, 0, 0, 0, 0, 2, 1, 0, 0, 1, 0, ...
$ disability   <dbl> 0.000000, 13.888890, 8.333333, 66.666660, 27.777780, 11.111110, 13...
$ height       <dbl> NA, 156, 155, NA, 162, NA, 155, 166, 172, 159, 155, 164, 170, NA, ...

```

There is no text variables converted to factors in the imported data set. However, it is more convenient to use factors so we change all character vectors to factors.

```

bP <- mutate_if(bP,is.character, as.factor)
head(bP)

# A tibble: 6 x 25
      id residence sex      age wealthQ physical country backPain30 agegr maritalS eduS

```



```

  <dbl> <fct>      <fct> <dbl> <fct>  <fct>      <fct>      <fct> <fct> <fct> <fct>
1 27043 Urban      Fema~   57 Q3      mod phy~ South ~ Yes      50-59 Married~ No p~
2 27471 Urban      Fema~   64 Q3      mod phy~ China  Yes      60-69 Married~ No p~
3  5323 Rural      Fema~   60 Q2      high ph~ China  No       60-69 Married~ No p~
4  9068 Rural      Fema~   70 Q1 poo~ low phy~ Ghana  No       70-79 Div/Wid~ No p~
5 28828 Urban      Fema~   58 Q5 ric~ mod phy~ Mexico Yes      50-59 Married~ Comp~
6 23775 Urban      Fema~   60 Q1 poo~ high ph~ Mexico Yes      60-69 Div/Wid~ No p~
# ... with 14 more variables: workS <fct>, bmi <dbl>, bmi4 <fct>, waistc <dbl>,
#   smoke <fct>, alcohol <fct>, arthritis <fct>, angina <fct>, depression <fct>,
#   asthma <fct>, diabetes <fct>, comorb <dbl>, disability <dbl>, height <dbl>

```

17.2 Single table verbs

We will look at the following single table verbs:

| In tidyverse | Examples of corresponding functions in basic R |
|--------------|--|
| select() | matrix form df[,column selection] |
| filter() | subset() or in matrix form df[row selection,] |
| slice() | matrix form df[selection by row numbers,] |
| arrange() | order() |
| rename() | names(df)[column number]<-"newvar" |
| mutate() | df\$newvar<-expression |
| count() | table() or xtabs() |
| group_by() | ? |
| summarise() | ? |

Some of these also have so called scoped variants of these, e.g. mutate_if. See help on scoped.

17.2.1 select()

Column selection can be done using column names (always preferred) or their numerical position.

```

bPs <- select(bP,country, residence, sex, height, disability, diabetes)
head(bPs)

```

```

# A tibble: 6 x 6
  country      residence sex      height disability diabetes
  <fct>      <fct>      <fct>    <dbl>    <dbl> <fct>
1 South Africa Urban      Female    NA         0      yes
2 China      Urban      Female   156      13.9   no
3 China      Rural      Female   155      8.33   no
4 Ghana      Rural      Female    NA      66.7   no
5 Mexico     Urban      Female   162      27.8   yes
6 Mexico     Urban      Female    NA      11.1   no

```

In next example it is using a sequence of column numbers. Observe the alternative printing by using kable.

```

bPs <- select(bP, 1:4)
kable(head(bPs))

```

| X1 | residence | sex | age |
|-------|-----------|--------|-----|
| 27043 | Urban | Female | 57 |
| 27471 | Urban | Female | 64 |
| 5323 | Rural | Female | 60 |
| 9068 | Rural | Female | 70 |
| 28828 | Urban | Female | 58 |
| 23775 | Urban | Female | 60 |

You can use the ":" between named columns, for an interval. You can delete a column by selecting it with a minus sign in front of name or number:

```
bPs <- select(bP, -c(sex,age))
bPs

# A tibble: 10,000 x 23
   id residence wealthQ physical country backPain30 agegr maritalS eduS workS bmi
   <dbl> <fct>    <fct>    <fct>    <fct>    <fct>    <fct> <fct>    <fct> <fct> <dbl>
1 27043 Urban    Q3      mod phy~ South ~ Yes    50-59 Married~ No p~ curr~ 30.5
2 27471 Urban    Q3      mod phy~ China Yes    60-69 Married~ No p~ neve~ 24.9
3 5323 Rural    Q2      high ph~ China No    60-69 Married~ No p~ curr~ 28.2
4 9068 Rural    Q1 poo~ low phy~ Ghana No    70-79 Div/Wid~ No p~ curr~ 16.2
5 28828 Urban    Q5 ric~ mod phy~ Mexico Yes    50-59 Married~ Comp~ curr~ 32.8
6 23775 Urban    Q1 poo~ high ph~ Mexico Yes    60-69 Div/Wid~ No p~ curr~ 31.1
7 24138 Rural    Q2      mod phy~ China Yes    60-69 Married~ No p~ curr~ 26.2
8 17074 Urban    Q5 ric~ mod phy~ China No    60-69 Married~ Comp~ curr~ 17.8
9 18398 Rural    Q2      high ph~ China No    60-69 Married~ No p~ curr~ 21.1
10 25789 Urban    Q5 ric~ low phy~ Mexico Yes    70-79 Married~ Comp~ neve~ 24.6
# ... with 9,990 more rows, and 12 more variables: bmi4 <fct>, waistc <dbl>, smoke <fct>,
# alcohol <fct>, arthritis <fct>, angina <fct>, depression <fct>, asthma <fct>,
# diabetes <fct>, comorb <dbl>, disability <dbl>, height <dbl>
```

or:

```
bPs <- select(bP, -(3:4))
bPs

# A tibble: 10,000 x 23
   id residence wealthQ physical country backPain30 agegr maritalS eduS workS bmi
   <dbl> <fct>    <fct>    <fct>    <fct>    <fct>    <fct> <fct>    <fct> <fct> <dbl>
1 27043 Urban    Q3      mod phy~ South ~ Yes    50-59 Married~ No p~ curr~ 30.5
2 27471 Urban    Q3      mod phy~ China Yes    60-69 Married~ No p~ neve~ 24.9
3 5323 Rural    Q2      high ph~ China No    60-69 Married~ No p~ curr~ 28.2
4 9068 Rural    Q1 poo~ low phy~ Ghana No    70-79 Div/Wid~ No p~ curr~ 16.2
5 28828 Urban    Q5 ric~ mod phy~ Mexico Yes    50-59 Married~ Comp~ curr~ 32.8
6 23775 Urban    Q1 poo~ high ph~ Mexico Yes    60-69 Div/Wid~ No p~ curr~ 31.1
7 24138 Rural    Q2      mod phy~ China Yes    60-69 Married~ No p~ curr~ 26.2
8 17074 Urban    Q5 ric~ mod phy~ China No    60-69 Married~ Comp~ curr~ 17.8
9 18398 Rural    Q2      high ph~ China No    60-69 Married~ No p~ curr~ 21.1
10 25789 Urban    Q5 ric~ low phy~ Mexico Yes    70-79 Married~ Comp~ neve~ 24.6
# ... with 9,990 more rows, and 12 more variables: bmi4 <fct>, waistc <dbl>, smoke <fct>,
# alcohol <fct>, arthritis <fct>, angina <fct>, depression <fct>, asthma <fct>,
# diabetes <fct>, comorb <dbl>, disability <dbl>, height <dbl>
```

17.2.2 filter()

With `filter()` you select **rows** of a data frame. Note the double '=' signs as usual. And between the selection conditions in the default but can be changed. See the help. NA in a filter variable is dropped.

```
bPf <- filter(bP, country == 'China' ,
              residence == 'Rural', sex=='Female',
              diabetes == 'yes')
bPf

# A tibble: 32 x 25
   id residence sex      age wealthQ physical country backPain30 agegr maritalS eduS
   <dbl> <fct>   <fct> <dbl> <fct>   <fct>   <fct>   <fct>   <fct> <fct>   <fct>
1 27913 Rural    Fema~ 66 Q3    low phy~ China    Yes     60-69 Married~ No p~
2 20813 Rural    Fema~ 57 Q2    mod phy~ China    No      50-59 Married~ No p~
3 7794 Rural    Fema~ 52 Q5 ric~ mod phy~ China    No      50-59 Married~ No p~
4 23848 Rural    Fema~ 66 Q2    high ph~ China    Yes     60-69 Married~ No p~
5 12824 Rural    Fema~ 72 Q2    low phy~ China    No      70-79 Married~ No p~
6 13319 Rural    Fema~ 73 Q2    low phy~ China    No      70-79 Div/Wid~ No p~
7 19881 Rural    Fema~ 75 Q1 poo~ mod phy~ China    No      70-79 Div/Wid~ No p~
8 12799 Rural    Fema~ 78 Q1 poo~ low phy~ China    No      70-79 Div/Wid~ No p~
9 15276 Rural    Fema~ 58 Q2    high ph~ China    No      50-59 Married~ No p~
10 9804 Rural    Fema~ 51 Q4    high ph~ China    No      50-59 Married~ No p~
# ... with 22 more rows, and 14 more variables: workS <fct>, bmi <dbl>, bmi4 <fct>,
#   waistc <dbl>, smoke <fct>, alcohol <fct>, arthritis <fct>, angina <fct>,
#   depression <fct>, asthma <fct>, diabetes <fct>, comorb <dbl>, disability <dbl>,
#   height <dbl>
```

We can get rid of those pesky NA's using the `complete.cases` function.

```
bPnf <- select(bP,bmi,waistc,age, height)
summary(bPnf)

      bmi          waistc          age          height
Min.   :11.70   Min.   : 28.00   Min.   : 50.00   Min.   : 65.0
1st Qu.:20.67   1st Qu.: 78.00   1st Qu.: 55.00   1st Qu.:153.0
Median :23.84   Median : 85.50   Median : 62.00   Median :160.0
Mean    :24.72   Mean    : 86.57   Mean    : 63.58   Mean    :159.2
3rd Qu.:27.55   3rd Qu.: 95.00   3rd Qu.: 70.00   3rd Qu.:167.0
Max.    :82.53   Max.    :225.00   Max.    :100.00   Max.    :210.0
NA's    :594     NA's    :683                      NA's    :4116

bPf <- filter(bPnf,complete.cases(bmi,waistc,age)) # complete data in bmi,waistc and age
# complete.cases operate on variables whil omit.na operates on a data frame
summary(bPf)

      bmi          waistc          age          height
Min.   :11.70   Min.   : 28.00   Min.   :50.00   Min.   : 80.0
1st Qu.:20.61   1st Qu.: 77.90   1st Qu.:55.00   1st Qu.:153.0
Median :23.76   Median : 85.50   Median :62.00   Median :160.0
Mean    :24.65   Mean    : 86.53   Mean    :63.34   Mean    :159.2
3rd Qu.:27.46   3rd Qu.: 95.00   3rd Qu.:70.00   3rd Qu.:166.0
Max.    :82.53   Max.    :225.00   Max.    :98.00   Max.    :210.0
NA's    :3625
```

17.2.3 slice()

Rows can also be selected by position using `slice()`.

```
bPs5 <- slice(bP, 1:5)
bPs5

# A tibble: 5 x 25
  id residence sex      age wealthQ physical country backPain30 agegr maritalS eduS
  <dbl> <fct>   <fct> <dbl> <fct>   <fct>   <fct>   <fct>   <fct> <fct>   <fct>
1 27043 Urban   Fema~  57 Q3    mod phy~ South ~ Yes    50-59 Married~ No p~
2 27471 Urban   Fema~  64 Q3    mod phy~ China  Yes    60-69 Married~ No p~
3 5323 Rural    Fema~  60 Q2    high ph~ China  No     60-69 Married~ No p~
4 9068 Rural    Fema~  70 Q1 poo~ low phy~ Ghana  No     70-79 Div/Wid~ No p~
5 28828 Urban   Fema~  58 Q5 ric~ mod phy~ Mexico Yes    50-59 Married~ Comp~
# ... with 14 more variables: workS <fct>, bmi <dbl>, bmi4 <fct>, waistc <dbl>,
#   smoke <fct>, alcohol <fct>, arthritis <fct>, angina <fct>, depression <fct>,
#   asthma <fct>, diabetes <fct>, comorb <dbl>, disability <dbl>, height <dbl>

bPs10 <- slice(bP, seq(1, nrow(bP), by = 10)) # Select every 10th observation
bPs10

# A tibble: 1,000 x 25
  id residence sex      age wealthQ physical country backPain30 agegr maritalS eduS
  <dbl> <fct>   <fct> <dbl> <fct>   <fct>   <fct>   <fct>   <fct> <fct>   <fct>
1 27043 Urban   Fema~  57 Q3    mod phy~ South ~ Yes    50-59 Married~ No p~
2 8120 Urban   Fema~  71 Q2    mod phy~ China  No     70-79 Married~ Comp~
3 26658 Rural    Fema~  63 Q4    high ph~ Russia~ Yes    60-69 Married~ Comp~
4 21774 Rural    Male  58 Q4    high ph~ Ghana  Yes    50-59 Married~ No p~
5 17090 Urban    Male  54 Q3    high ph~ China  No     50-59 Married~ Comp~
6 30346 Urban    Fema~  65 Q4    low phy~ China  Yes    60-69 Married~ No p~
7 17100 Rural    Male  72 Q2    low phy~ Ghana  No     70-79 Div/Wid~ Comp~
8 16852 Rural    Fema~  86 Q2    low phy~ China  No     80+   Div/Wid~ No p~
9 21191 Rural    Male  65 Q3    mod phy~ Ghana  No     60-69 Married~ Comp~
10 258 Urban     Male  54 Q5 ric~ high ph~ India  No     50-59 Married~ Comp~
# ... with 990 more rows, and 14 more variables: workS <fct>, bmi <dbl>, bmi4 <fct>,
#   waistc <dbl>, smoke <fct>, alcohol <fct>, arthritis <fct>, angina <fct>,
#   depression <fct>, asthma <fct>, diabetes <fct>, comorb <dbl>, disability <dbl>,
#   height <dbl>
```

17.2.4 arrange()

The function `arrange()` is used to reorder rows. You provide a column name to control the ordering; if you want to resolve ties, add more column names. Default is ascending order. Because there are many variables we limit the example to a few of them.

```
bPa <- arrange(bP, waistc)
head(select(bPa, residence, sex, waistc, age, wealthQ), 10) # default is ascending order

# A tibble: 10 x 5
  residence sex      waistc age wealthQ
  <fct>   <fct>   <dbl> <dbl> <fct>
1 Rural   Female    28    59 Q2
2 Rural   Male     28    75 Q3
3 Rural   Female    28    68 Q2
4 Rural   Male     29    67 Q1 poorest
```

| | | | | | |
|----|-------|--------|------|----|------------|
| 5 | Rural | Male | 29 | 58 | Q2 |
| 6 | Urban | Male | 29.5 | 50 | Q4 |
| 7 | Rural | Female | 30.4 | 53 | Q2 |
| 8 | Rural | Male | 31 | 50 | Q1 poorest |
| 9 | Urban | Male | 31 | 83 | Q4 |
| 10 | Rural | Male | 32 | 66 | Q1 poorest |

We can sort by more variables and choose descending order is done as follows:

```
bPa <- arrange(bP, desc(waistc), age)
head(select(bPa, residence, sex, waistc, age, wealthQ), 10)
```

A tibble: 10 x 5

| | residence | sex | waistc | age | wealthQ |
|----|-----------|--------|--------|-------|------------|
| | <fct> | <fct> | <dbl> | <dbl> | <fct> |
| 1 | Urban | Male | 225 | 55 | Q3 |
| 2 | Urban | Female | 200 | 52 | Q2 |
| 3 | Urban | Female | 200 | 53 | Q4 |
| 4 | Rural | Female | 200 | 63 | Q3 |
| 5 | Urban | Female | 200 | 80 | Q4 |
| 6 | Urban | Female | 195 | 61 | Q4 |
| 7 | Urban | Male | 193 | 59 | Q1 poorest |
| 8 | Urban | Female | 189 | 53 | Q5 richest |
| 9 | Rural | Male | 178 | 61 | Q4 |
| 10 | Urban | Female | 174 | 65 | Q5 richest |

17.2.5 rename()

Renaming columns is straightforward:

```
bPr <- rename(bP, wealthQuantile = wealthQ) # New name = old name
head(bPr)
```

A tibble: 6 x 25

| | X1 | residence | sex | age | wealthQuantile | physical | country | backPain30 | agegr | maritalS |
|---|-------|-----------|-------|-------|----------------|----------|---------|------------|-------|----------|
| | <dbl> | <fct> | <fct> | <dbl> | <fct> | <fct> | <fct> | <fct> | <fct> | <fct> |
| 1 | 27043 | Urban | Fema~ | 57 | Q3 | mod phy~ | South ~ | Yes | 50-59 | Married~ |
| 2 | 27471 | Urban | Fema~ | 64 | Q3 | mod phy~ | China | Yes | 60-69 | Married~ |
| 3 | 5323 | Rural | Fema~ | 60 | Q2 | high ph~ | China | No | 60-69 | Married~ |
| 4 | 9068 | Rural | Fema~ | 70 | Q1 poorest | low phy~ | Ghana | No | 70-79 | Div/Wid~ |
| 5 | 28828 | Urban | Fema~ | 58 | Q5 richest | mod phy~ | Mexico | Yes | 50-59 | Married~ |
| 6 | 23775 | Urban | Fema~ | 60 | Q1 poorest | high ph~ | Mexico | Yes | 60-69 | Div/Wid~ |

... with 15 more variables: eduS <fct>, workS <fct>, bmi <dbl>, bmi4 <fct>,
waistc <dbl>, smoke <fct>, alcohol <fct>, arthritis <fct>, angina <fct>,
depression <fct>, asthma <fct>, diabetes <fct>, comorb <dbl>, disability <dbl>,
height <dbl>

17.2.6 mutate()

The function mutate() adds new columns which are calculated from old columns:

```
bPm <- mutate(bP, heightInches = height/2.54)
head(select(bPm, residence, sex, age, waistc, height, heightInches))
```

A tibble: 6 x 6

| | residence | sex | age | waistc | height | heightInches |
|---|-----------|--------|-------|--------|--------|--------------|
| | <fct> | <fct> | <dbl> | <dbl> | <dbl> | <dbl> |
| 1 | Urban | Female | 57 | 108 | NA | NA |
| 2 | Urban | Female | 64 | 82 | 156 | 61.4 |
| 3 | Rural | Female | 60 | 80 | 155 | 61.0 |
| 4 | Rural | Female | 70 | 86 | NA | NA |
| 5 | Urban | Female | 58 | 123 | 162 | 63.8 |
| 6 | Urban | Female | 60 | 102. | NA | NA |

It can also be used to change/add multiple columns:

```
bPm <- mutate(bP, heightM = height/100,
              wHr = waistc/height,
              sAge = age-50)
select(bPm, residence, sex, age, heightM, wHr, sAge)
```

A tibble: 10,000 x 6

| ## | residence | sex | age | heightM | wHr | sAge |
|-------|--------------------------|--------|-------|---------|-------|-------|
| ## | <fct> | <fct> | <dbl> | <dbl> | <dbl> | <dbl> |
| ## 1 | Urban | Female | 57 | NA | NA | 7 |
| ## 2 | Urban | Female | 64 | 1.56 | 0.526 | 14 |
| ## 3 | Rural | Female | 60 | 1.55 | 0.516 | 10 |
| ## 4 | Rural | Female | 70 | NA | NA | 20 |
| ## 5 | Urban | Female | 58 | 1.62 | 0.759 | 8 |
| ## 6 | Urban | Female | 60 | NA | NA | 10 |
| ## 7 | Rural | Female | 67 | 1.55 | 0.575 | 17 |
| ## 8 | Urban | Female | 63 | 1.66 | 0.410 | 13 |
| ## 9 | Rural | Male | 60 | 1.72 | 0.419 | 10 |
| ## 10 | Urban | Male | 76 | 1.59 | 0.550 | 26 |
| ## # | ... with 9,990 more rows | | | | | |

17.2.7 count()

We can summarise individual factors with counts of their levels using the function `count()`. It reminds of `ftable` but the result is a tibble/data frame.

```
count(bP, wealthQ)
```

A tibble: 6 x 2

| | wealthQ | n |
|---|------------|-------|
| | <fct> | <int> |
| 1 | Q1 poorest | 1889 |
| 2 | Q2 | 1983 |
| 3 | Q3 | 1888 |
| 4 | Q4 | 2054 |
| 5 | Q5 richest | 2159 |
| 6 | <NA> | 27 |

```
bPc <- count(bP, country, residence, wealthQ)
bPc
```

A tibble: 66 x 4

```

country residence wealthQ      n
<fct>    <fct>    <fct>    <int>
1 China   Rural    Q1 poorest  514
2 China   Rural    Q2          513
3 China   Rural    Q3          355
4 China   Rural    Q4          335
5 China   Rural    Q5 richest  194
6 China   Rural    <NA>         7
7 China   Urban    Q1 poorest  254
8 China   Urban    Q2          249
9 China   Urban    Q3          384
10 China  Urban    Q4          434
# ... with 56 more rows

```

The result **bPc** is a multi-way table in long format and the resulting object is a tibble. Thus all of it will not be printed out. If you want to see all of it try **as.data.frame(bPc)**

Notice the warnings are there because there are a significant number of NA in the data.

17.2.8 base::summary()

It can sometimes be useful to use the `base::summary()` function after selecting only those columns which are of interest. For factors you get one-way frequency tables.

```

summary(select(bP, residence, country, agegr, age))

```

| residence | country | agegr | age |
|------------|---------------------|-------|------------|
| Rural:4902 | China | :3810 | 50-59:4126 |
| Urban:5098 | Ghana | :1300 | 60-69:3065 |
| | India | :1945 | 70-79:2041 |
| | Mexico | : 661 | 80+ : 768 |
| | Russian Federation: | 1169 | |
| | South Africa | :1115 | |

Min. : 50.00
1st Qu.: 55.00
Median : 62.00
Mean : 63.58
3rd Qu.: 70.00
Max. :100.00

17.3 Pipes

One characteristic with tidyverse is that pipes "%>%" are allowed. In this section we shall look at some examples how they are used. A pipe is a link leading from one tibble (or data frame) to another via a verb (function). A quick command for "%>%" is "ctrl shift m".

First we want to select some variables and print the first 4 rows. Compare with the earlier examples when bP was an argument.

```

bP %>% select(age,sex,wealthQ,physical,bmi) %>% head(4)

```

```

# A tibble: 4 x 5
  age sex    wealthQ    physical    bmi
<dbl> <fct> <fct>    <fct>    <dbl>
1   57 Female Q3      mod phys act  30.5
2   64 Female Q3      mod phys act  24.9
3   60 Female Q2      high phys act  28.2
4   70 Female Q1 poorest low phys act  16.2

```

We can also in the same command set the result to an object (tibble).

```
bPp<-bP %>% select(age,sex,wealthQ,physical,bmi) %>%
  filter(sex=="Female",age<60) %>% arrange(bmi)
bPp

# A tibble: 2,197 x 5
   age sex    wealthQ physical    bmi
<dbl> <fct> <fct>    <fct>    <dbl>
1    55 Female Q2      low phys act 13.1
2    50 Female Q1 poorest high phys act 13.2
3    58 Female Q2      high phys act 13.3
4    54 Female Q2      mod phys act 13.9
5    55 Female Q4      high phys act 14.0
6    56 Female Q1 poorest high phys act 14.1
7    55 Female Q2      mod phys act 14.1
8    54 Female Q1 poorest high phys act 14.2
9    58 Female Q4      mod phys act 14.2
10   52 Female Q5 richest high phys act 14.4
# ... with 2,187 more rows

bPp<-bP %>% filter(sex=="Female",age<60) %>%
  mutate(bmi2=cut(bmi,c(0,25,Inf))) %>% count(bmi2)
bPp

# A tibble: 3 x 2
  bmi2      n
<fct> <int>
1 (0,25] 1120
2 (25,Inf] 983
3 <NA>    94

bPp<-bP %>% filter(sex=="Female",age<60) %>%
  mutate(bmi2=cut(bmi,c(0,25,Inf)),agesq=age^2) %>% slice(1:3) %>%
  select(-c(5,7:11,13:24))
bPp

# A tibble: 3 x 9
   id residence sex    age physical    workS      height bmi2    agesq
<dbl> <fct>    <fct> <dbl> <fct>    <fct>    <dbl> <fct>    <dbl>
1 27043 Urban  Female  57 mod phys act currently not working    NA (25,Inf] 3249
2 28828 Urban  Female  58 mod phys act currently working      162 (25,Inf] 3364
3 22008 Urban  Female  51 low phys act currently working      156 (0,25] 2601
```

17.4 Manipulating factors with tidyverse tools

Factors (basically categorical variables) are somewhat complicated objects but they can be very useful if we can learn how handle them.

We have already seen how to do the conversion of all character variables in a dataframe to factors:

```
bP <- mutate_if(bP, is.character, as.factor)
```

We note that this should be done with care, because there will certainly be times when conversion of character variables to factors is inappropriate - for which reason, the tidyverse dataframe functions do not automatically convert character variables to factors, which is the default in older R functions.

Here we will take a closer look at how to manipulate factors. We'll begin here by selecting a factor variable from the backPain data frame - country, in which the order of the 'levels' (categories) of the factor is not significant (nominal variable). Notice that R saves storage of factors by saving them as numerics and relating the numerics to the levels.

```
levels(bP$country)

[1] "China"          "Ghana"          "India"          "Mexico"
[5] "Russian Federation" "South Africa"

str(select(bP, country) )

Classes 'tbl_df', 'tbl' and 'data.frame': 10000 obs. of  1 variable:
 $ country: Factor w/ 6 levels "China","Ghana",...: 6 1 1 2 4 4 1 1 1 4 ...

bP %>% select(country) %>% mutate(numeric.country=as.numeric(country)) %>% slice(1:5)

# A tibble: 5 x 2
  country      numeric.country
  <fct>          <dbl>
1 South Africa      6
2 China             1
3 China             1
4 Ghana             2
5 Mexico            4

# print numeric values of first 5 elements
```

Often coding for levels (input codes) is abbreviated (or sometimes more lengthy than we might like for display). By default, when plotting the level names are used on the plot. If the names are not suitable, we can then define a more suitable name for the level (we 're-code' it) and then the new code name will be used in the plots and at the same time R will re-name the level in the dataframe.

There is considerable confusion about the use of 'labels' with levels and tidyverse offers tidier (and clearer) solutions!

Basically we want to be able to do four things:

1. Coerce numeric and/or character variables to factors where it is appropriate,
2. Re-order to something more sensible. The default alphabetical order of the levels is often unsatisfactory for presentation purposes, either in a table or a plot. Use `fct_relevel`.
3. Re-name unnecessarily terse, long or meaningless names. Use `fct_recode`
4. Cut a numeric variable into named groups to create a new factor variable. Use one of:
 - `cut_interval` which makes n groups with equal range,
 - `cut_number` which makes n groups each with approximately equal numbers of observations or
 - `cut_width`, which makes groups with a specified 'bin width'

You can see examples of the cut functions here.

17.4.1 Coercing numeric and character variables to a factor

Here is how to coerce a simple numeric variable. (In this case `comorb` is coded as the number of comorbidities, 0, 1, 2 - where 2 may be 2 or more.)

```
bP %>% select(comorb) %>% head(5)

# A tibble: 5 x 1
  comorb
  <dbl>
1      2
2      1
3      0
4      1
5      1

bPm<-bP %>% select(comorb) %>% mutate(comorb=as.factor(comorb)) %>% head(5)
str(bPm$comorb)

Factor w/ 3 levels "0","1","2": 3 2 1 2 2
```

17.4.2 Changing the codes of a factor

We sometimes need to change the codes to give us meaningful labels for presentation, we use `fct_recode`. The recoding definition in the `fct_recode` function is "new name" = "old name". We start with converting `comorb` to a factor

```
bPm<-bP %>% mutate(comorb=as.factor(comorb))%>%
  mutate(comorb = fct_recode(comorb,"None" = "0","One" = "1","Two or more"= "2"))
bPm %>% select(comorb)

# A tibble: 10,000 x 1
  comorb
  <fct>
1 Two or more
2 One
3 None
4 One
5 One
6 Two or more
7 One
8 None
9 None
10 None
# ... with 9,990 more rows
```

17.4.3 Re-ordering factor levels

To re-order, you must specify the levels in your desired order as the arguments of `fct_relevel` following the variable name. The spelling must be exactly as in the original!

```
levels(bP$bmi4)

[1] "Normal"      "Obese"      "Pre-Obese"  "Underweight"

bPm <- bP %>% mutate(bmi4 = fct_relevel(bmi4, "Underweight", "Normal", "Pre-Obese", "Obese"))
levels(bPm$bmi4)

[1] "Underweight" "Normal"      "Pre-Obese"  "Obese"
```

17.4.4 Re-coding (renaming) categories (levels) in factors

If you simply want to change the coding of individual levels, this is easily done with `fct_recode`. Here is the call to reduce the length of the Russian Federation code (and we can deal with South Africa at the same time):

```
bP %>% count(country)

# A tibble: 6 x 2
  country          n
  <fct>          <int>
1 China          3810
2 Ghana          1300
3 India          1945
4 Mexico          661
5 Russian Federation 1169
6 South Africa    1115

bPr <- bP %>%
  mutate(country = fct_recode(country,
                              "Russian Fed" = "Russian Federation",
                              "Sth Africa" = "South Africa") )

bPr %>% count(country)

# A tibble: 6 x 2
  country          n
  <fct>          <int>
1 China          3810
2 Ghana          1300
3 India          1945
4 Mexico          661
5 Russian Fed    1169
6 Sth Africa     1115
```

If you look back at the preceding script you will see that there we had the level name "Russian Federation". We've changed the associated level name to "Russian Fed" and a similar change was made to "South Africa"

17.4.5 Converting numerics to factors using the `cut_` functions

The function `cut()` provides a quick way of converting numeric data to grouped factors. The `forcats` package (included in `tidyverse`) simplifies and extends these calls a little.

Here's how to create 6 levels with approximately the same number of observations in each group:

```
bPc6 <- bP %>% mutate(height6 = cut_number(height,
      n = 6,
      labels = c("Very Short", "Short",
                 "Average", "Tall", "Very Tall", "Extremely Tall")))
summary(bPc6$height6)
```

| | | | | | |
|------------|-------|---------|------|-----------|----------------|
| Very Short | Short | Average | Tall | Very Tall | Extremely Tall |
| 1174 | 967 | 1124 | 1040 | 917 | 662 |
| NA's | | | | | |
| 4116 | | | | | |

We use `cut_interval` to cut with approximately equal ranges:


```

                                "very tall" = "Extremely Tall"))
xtabs(~h6_to_4+height6,data=bP64)

##           height6
## h6_to_4      Very Short Short Average Tall Very Tall Extremely Tall
##   short           1174   967         0    0           0           0
##   Average           0     0       1124    0           0           0
##   Tall              0     0         0 1040           0           0
##   very tall         0     0         0    0          917          662

```

The package `forcats` is also very useful for dealing with factors. See Chapter 15 of ‘R for data science’ [Wickham2017] for detailed examples and extensions on the use of the functions we have seen here.

17.5 group_by()

The `summarise()` function and those we have discussed above, become much more powerful when we use grouping operations with the verb/function `group_by()`.

The other verbs are affected by grouping as follows:

- Grouped `select()` is the same as ungrouped `select()`, excepted that grouping variables are always retained.
- Grouped `arrange()` orders first by grouping variables
- The `slice()` function extracts rows within each group.
- The `count()` function counts the number of rows with each unique value of variable, so it is particularly useful for counting the frequency of levels in factors.
- The `summarise()` function is particularly useful when applied to grouped variables, and is explained in detail below.

17.5.1 Summarising groups

In summarising groups we can add columns containing the statistics (mean, sd, max, IQR etc) for every group combination of the set specified. In our bP data, we group using factor variables.

```

bPs <- bP %>%
  group_by(residence) %>%
  summarise(mean.disability = mean(disability))
bPs

# A tibble: 2 x 2
  residence mean.disability
  <fct>      <dbl>
1 Rural      19.4
2 Urban      16.0

```

Here’s a further example in which we add more groups and statistics for more variables.

```
bPs <- bP %>% group_by(country, residence, sex) %>%
  summarise(mean.disability = mean(disability), disIQR = IQR(disability), Bmi = mean(bmi, na.rm=T))
bPs
```

```
# A tibble: 24 x 6
# Groups:   country, residence [12]
  country residence sex    mean.disability disIQR    Bmi
  <fct>    <fct>    <fct>          <dbl>   <dbl> <dbl>
1 China   Rural    Female         12.4    13.9  23.8
2 China   Rural    Male           8.57   11.1  23.1
3 China   Urban    Female         8.16   11.1  24.8
4 China   Urban    Male           6.62    8.33  24.6
5 Ghana   Rural    Female        27.4    25.   22.2
6 Ghana   Rural    Male         19.2    25.0  21.3
7 Ghana   Urban    Female        25.9    30.6  25.3
8 Ghana   Urban    Male         18.9    30.6  24.4
9 India   Rural    Female        33.6    25    20.7
10 India  Rural    Male         24.9    25.0  19.8
# ... with 14 more rows
```

Observe the `na.rm=T` which was necessary because of missing values.

We may want to change the order in the table:

```
bPs <- bP %>% group_by(sex, country, residence) %>%
  summarise(mean.disability = mean(disability), disIQR = IQR(disability), Bmi = mean(bmi, na.rm=T))
bPs
```

```
# A tibble: 24 x 6
# Groups:   sex, country [12]
  sex    country residence mean.disability disIQR    Bmi
  <fct> <fct>    <fct>          <dbl>   <dbl> <dbl>
1 Female China   Rural         12.4    13.9  23.8
2 Female China   Urban          8.16   11.1  24.8
3 Female Ghana   Rural        27.4    25.   22.2
4 Female Ghana   Urban        25.9    30.6  25.3
5 Female India   Rural        33.6    25    20.7
6 Female India   Urban        27.8    27.8  22.4
7 Female Mexico  Rural        22.2    27.8  27.6
8 Female Mexico  Urban        19.7    25.0  29.1
9 Female Russian Fed Rural    22.1    25.0  30.9
10 Female Russian Fed Urban    23.8    27.8  29.3
# ... with 14 more rows
```

..or filter to look at only one country. Here also the counts of observations in the group are included (function `n()`)

```
bPs <- bP %>%
  filter(country == "China") %>%
  group_by(sex, residence) %>%
  summarise(mean.disability = mean(disability), disIQR = IQR(disability), Bmi = mean(bmi, na.rm=T),
    , count=n())
bPs
```

```
# A tibble: 4 x 6
# Groups:   sex [2]
  sex    residence mean.disability disIQR    Bmi count
  <fct> <fct>          <dbl>   <dbl> <dbl> <dbl>
1 Female Rural         12.4    13.9  23.8     2
2 Female Urban          8.16   11.1  24.8     2
3 Female Rural        27.4    25.   22.2     2
4 Female Urban        25.9    30.6  25.3     2
```

| | <fct> | <fct> | <dbl> | <dbl> | <dbl> | <int> |
|---|--------|-------|-------|-------|-------|-------|
| 1 | Female | Rural | 12.4 | 13.9 | 23.8 | 970 |
| 2 | Female | Urban | 8.16 | 11.1 | 24.8 | 1054 |
| 3 | Male | Rural | 8.57 | 11.1 | 23.1 | 948 |
| 4 | Male | Urban | 6.62 | 8.33 | 24.6 | 838 |

When groups vary significantly in size it is prudent to always include counts of observations:

```
bPs <- group_by(bP, country, sex, residence, wealthQ) %>%
  summarise(count=n(), mean.disability=mean(disability))
bPs

# A tibble: 130 x 6
# Groups:   country, sex, residence [24]
  country sex residence wealthQ count mean.disability
  <fct>   <fct> <fct>   <fct>   <int>      <dbl>
1 China  Female Rural   Q1 poorest  277      15.7
2 China  Female Rural   Q2          260      12.4
3 China  Female Rural   Q3          172      11.8
4 China  Female Rural   Q4          156       9.99
5 China  Female Rural   Q5 richest   99       7.10
6 China  Female Rural   <NA>         6      24.5
7 China  Female Urban   Q1 poorest  137      16.7
8 China  Female Urban   Q2          146      10.8
9 China  Female Urban   Q3          210       8.90
10 China Female Urban   Q4          237       5.92
# ... with 120 more rows
```

17.6 left_join(): Merging two data frames

There is a set of functions in dplyr for merging data frames. Here we'll just demonstrate the left_join() function.

We'll first create a data set iDf with individual ID (id) and a household ID (hhID). Then we create a second data set on households (hhDf) which will relate to the first data set through the household ID (hhID) variable.

```
ID <- 1:15
hhID <- c(1,1,1,1,1,2,2,3,3,3,4,4,4,4,4)
iData1 <- LETTERS[1:15]
iData2 <- letters[12:26]

iDf <- data.frame(id = as.factor(ID),
                  hhID = as.factor(hhID),
                  iD1 = iData1,
                  iD2 = iData2)

iDf

##      id hhID iD1 iD2
## 1    1    1   A   l
## 2    2    1   B   m
## 3    3    1   C   n
## 4    4    1   D   o
## 5    5    1   E   p
## 6    6    2   F   q
```

```
## 7 7 2 G r
## 8 8 3 H s
## 9 9 3 I t
## 10 10 3 J u
## 11 11 4 K v
## 12 12 4 L w
## 13 13 4 M x
## 14 14 4 N y
## 15 15 4 O z

hhID <- 1:4
hData1 <- c("X1", "X2", "X3", "X4")
hData2 <- letters[5:8]
hhDf <- data.frame(hhID = as.factor(hhID),
                   hD1 = hData1,
                   hD2 = hData2)

hhDf

##   hhID hD1 hD2
## 1    1 X1  e
## 2    2 X2  f
## 3    3 X3  g
## 4    4 X4  h
```

Now let's see if we can create a combined data frame in which the individual data frame rows are maintained, but have added to them the variables from the household data frame with values of those variables corresponding to the household listed in the individual's data frame.

We'll firstly try a `left_join` using the common household ID (`hhID`) as the 'key', and then we'll do it without specifying the key. Notice how data in each household are repeated for individuals in the same household.

```
merged <- left_join(iDf, hhDf, by="hhID")
merged

##   id hhID iD1 iD2 hD1 hD2
## 1  1    1  A  l  X1  e
## 2  2    1  B  m  X1  e
## 3  3    1  C  n  X1  e
## 4  4    1  D  o  X1  e
## 5  5    1  E  p  X1  e
## 6  6    2  F  q  X2  f
## 7  7    2  G  r  X2  f
## 8  8    3  H  s  X3  g
## 9  9    3  I  t  X3  g
## 10 10    3  J  u  X3  g
## 11 11    4  K  v  X4  h
## 12 12    4  L  w  X4  h
## 13 13    4  M  x  X4  h
## 14 14    4  N  y  X4  h
## 15 15    4  O  z  X4  h

merged2 <- left_join(iDf, hhDf)

## Joining, by = "hhID"

merged2

##   id hhID iD1 iD2 hD1 hD2
```



```
## 1 1 1 A l X1 e
## 2 2 1 B m X1 e
## 3 3 1 C n X1 e
## 4 4 1 D o X1 e
## 5 5 1 E p X1 e
## 6 6 2 F q X2 f
## 7 7 2 G r X2 f
## 8 8 3 H s X3 g
## 9 9 3 I t X3 g
## 10 10 3 J u X3 g
## 11 11 4 K v X4 h
## 12 12 4 L w X4 h
## 13 13 4 M x X4 h
## 14 14 4 N y X4 h
## 15 15 4 O z X4 h
```

There is also a base R function `base::merge` that can be used to merge (link) data in a similar way.

17.7 Tidyr functions

Håkan Jonsson <hakan.jonsson@umu.se>

17.7.1 Functions for converting between long and wide format

Since data can be stored in different ways there is sometimes a need to convert the data to the desired form. Two types of storage is wide and long format. Let us think of an example where timber volume (cubic meters) is measured with three different methods in three areas. The data can then be stored as wide format where the volume for each method is represented as separate variables (vectors). The functions we will use are from the `tidyr` package.

```
[1] "Wide format"
  area method1 method2 method3
1    1      210      242      207
2    2      135      135      111
3    3      187      201      214
```

In long format the vectors for each of the three methods are stacked. This means that there will be three times as many rows in the new data frame. We now only need one column for these values but we also need a column (here called `method`) with information on the type of method.

```
[1] "Long format"
  area method volume
1    1      1      210
2    2      1      135
3    3      1      187
4    1      2      242
5    2      2      135
6    3      2      201
7    1      3      207
8    2      3      111
9    3      3      214
```

17.7.2 Example - the dataset Subliminal

Let us first import the dataset. Data in the dataset Subliminal are from an intervention study where 18 students were randomized to receive either of two messages with the intention to see if this would affect their performance on the mathematics exam. The control group received neutral messages whereas the intervention group received messages confirming their learning process. All students participated in a summer school in mathematics and were tested at the beginning and end of the intervention. The dataset contains the following variables:

| | |
|--------------|--|
| Message: | If the student received neutral or confirmatory messages |
| Before: | Test result at the beginning of the study |
| After: | Test result at the end of the study |
| Improvement: | Improvement of their results (AfterBefore) |

Observe that we need to give two new variable names **key** which holds the information of the kind of data (variable) here chosen as Time and **value** here chosen as Result while Message, Improvement and ind.nr is kept.

```
library(haven)
Subliminal <- read_sav("../data/Subliminal.sav")

sub1<-Subliminal %>% cbind(ind.nr=1:nrow(Subliminal))

head(sub1)

  Message Before After Improvement ind.nr
1 positive    18   24          6      1
2 positive    18   25          7      2
3 positive    21   33         12      3
4 positive    18   29         11      4
5 positive    18   33         15      5
6 positive    20   36         16      6

nrow(sub1)

[1] 18

sub2<-sub1 %>% gather(key=Time,value=Result,Before,After)
# only individual 1 to 5 are shown
sub2 %>% filter(ind.nr<=5)

  Message Improvement ind.nr   Time Result
1 positive          6      1 Before    18
2 positive          7      2 Before    18
3 positive         12      3 Before    21
4 positive         11      4 Before    18
5 positive         15      5 Before    18
6 positive          6      1 After    24
7 positive          7      2 After    25
8 positive         12      3 After    33
9 positive         11      4 After    29
10 positive        15      5 After    33

nrow(sub2)

[1] 36
```

```
sub3<-sub2 %>% spread(key=Time,value=Result) %>% arrange(ind.nr)

head(sub3)

  Message Improvement ind.nr After Before
1 positive          6      1    24     18
2 positive          7      2    25     18
3 positive         12      3    33     21
4 positive         11      4    29     18
5 positive         15      5    33     18
6 positive         16      6    36     20

nrow(sub3)

[1] 18
```

In fact the different variables that is gathered does not have to be repeated measurements of the same kind. It can be completely different measures. Let us see an example using the sample of the BackPain data.

```
bPg<- bP %>% gather(key=var,value=value,disability,bmi,age)
bPg %>% count(var)

# A tibble: 3 x 2
  var      n
  <chr>  <int>
1 age    10000
2 bmi    10000
3 disability 10000

# the first three rows in each group
bPg %>% group_by(var) %>% select(var,value,residence,sex,wealthQ,physical,country) %>% slice(1:3)

# A tibble: 9 x 7
# Groups:   var [3]
  var      value residence sex    wealthQ physical    country
  <chr>    <dbl> <fct>   <fct> <fct>  <fct>      <fct>
1 age      57    Urban   Female Q3    mod phys act South Africa
2 age      64    Urban   Female Q3    mod phys act China
3 age      60    Rural   Female Q2    high phys act China
4 bmi     30.5    Urban   Female Q3    mod phys act South Africa
5 bmi     24.9    Urban   Female Q3    mod phys act China
6 bmi     28.2    Rural   Female Q2    high phys act China
7 disability 0     Urban   Female Q3    mod phys act South Africa
8 disability 13.9   Urban   Female Q3    mod phys act China
9 disability 8.33   Rural   Female Q2    high phys act China
```

Own experimentation

The last tibble bP is cut to show only the first three rows per group. Check if it looks ok also further down. Can you restore it to the original by use of spread? Try out other combinations of variables when using gather.

Chapter 18

Statistical methods part 2

Håkan Jonsson < hakan.jonsson@umu.se >

18.1 The formula object

The formula object is useful and important in estimation of linear models in R. It is described in the help as "a generic function formula and its specific methods provide a way of extracting formulae which have been included in other objects". It will be easier to understand how it is used from examples e.g. in the linear models below.

```
f1<-y~x+z # f1 is a formula  
  
class(f1)  
[1] "formula"  
  
typeof(f1)  
[1] "language"
```

18.2 Linear regression

Linear regression is a model where the observed value (dependent variable) is explained by one or more variables called linear predictors (or independent variables, explanatory variables, background variables or covariates). When only one predictor variable is used we talk about simple regression and with two or more predictors we call it multiple regression (sometimes multivariable regression or (incorrectly) multivariate regression)

18.2.1 Simple linear regression

The model can be written $y_i = a + bx_i + \epsilon_i$ where x_i is the linear predictor. In the model x_i is not random. The only random variable is ϵ_i a random error assumed to be independent between observations and follows a normal distribution $N(\mu, \sigma)$. The parameters a and b are constants which are estimated. The interpretation is that b (slope) is the expected change of y per one unit increase of x while a (intercept) is the expected y -value given $x = 0$. The intercept is most often not of interest or

relevant e.g. if y_i is the number of sold Volvo cars in Umeå a certain year x_i during 1990-2010. The intercept is then an estimate of the number of Volvo cars sold in Umeå year 0. However, the intercept is important if we are interested to use the model for prediction of the expected value on the dependent variable. The estimation is shown below. In the example we estimate how cholesterol can be predicted by BMI in the norsjo86 data set.

We will use the norsjo86 data set which has been earlier described. We first remove some missing data.

```
library(haven)
norsjo86 <- read_sav("../data/norsjo86.sav")

summary(norsjo86) # There are some missing, especially in health
```

| agegrp | | health | | sex | | height | | weight | |
|---------|--------|---------|---------|---------|--------|---------|--------|---------|---------|
| Min. | :30.00 | Min. | :0.0000 | Min. | :1.000 | Min. | :145.0 | Min. | : 47.00 |
| 1st Qu. | :40.00 | 1st Qu. | :0.0000 | 1st Qu. | :1.000 | 1st Qu. | :165.0 | 1st Qu. | : 64.00 |
| Median | :40.00 | Median | :0.0000 | Median | :2.000 | Median | :171.0 | Median | : 74.00 |
| Mean | :45.12 | Mean | :0.4046 | Mean | :1.508 | Mean | :170.9 | Mean | : 73.89 |
| 3rd Qu. | :60.00 | 3rd Qu. | :1.0000 | 3rd Qu. | :2.000 | 3rd Qu. | :178.0 | 3rd Qu. | : 81.00 |
| Max. | :60.00 | Max. | :1.0000 | Max. | :2.000 | Max. | :191.0 | Max. | :116.00 |
| | | NA's | :40 | | | NA's | :4 | NA's | :4 |

| sbp | | dbp | | kolester | | smoker | | bmi | |
|---------|--------|---------|---------|----------|---------|---------|---------|---------|--------|
| Min. | : 90.0 | Min. | : 48.00 | Min. | : 3.300 | Min. | :0.0000 | Min. | :16.71 |
| 1st Qu. | :110.0 | 1st Qu. | : 72.00 | 1st Qu. | : 5.400 | 1st Qu. | :0.0000 | 1st Qu. | :22.58 |
| Median | :120.0 | Median | : 78.00 | Median | : 6.300 | Median | :0.0000 | Median | :24.52 |
| Mean | :124.5 | Mean | : 79.17 | Mean | : 6.536 | Mean | :0.2077 | Mean | :25.27 |
| 3rd Qu. | :135.0 | 3rd Qu. | : 86.00 | 3rd Qu. | : 7.400 | 3rd Qu. | :0.0000 | 3rd Qu. | :27.28 |
| Max. | :220.0 | Max. | :120.00 | Max. | :11.600 | Max. | :1.0000 | Max. | :45.03 |
| NA's | :3 | NA's | :3 | NA's | :3 | | | NA's | :4 |

```
nrow(norsjo86)

[1] 260

# remove the variable health and remove missing in the remaining variables
norsjo<-na.omit(norsjo86[,~2])

nrow(norsjo) # 8 observations removed

[1] 252

res<-lm(norsjo$kolester~norsjo$bmi) # this is one method but somewhat prolix
res<-lm(kolester~bmi,data=norsjo)  # this is a more convenient way

res                                     # printing the results don't give a comprehensive output

Call:
lm(formula = kolester ~ bmi, data = norsjo)

Coefficients:
(Intercept)          bmi
    5.27744         0.04915

class(res)

[1] "lm"

summary(res)                                     # using summary gives a reasonable amount of information

Call:
lm(formula = kolester ~ bmi, data = norsjo)
```

```

Residuals:
    Min       1Q   Median       3Q      Max
-3.1892 -1.0369 -0.1388  0.8443  4.5600

Coefficients:
              Estimate Std. Error t value Pr(>|t|)
(Intercept)  5.27744    0.59932   8.806  <2e-16 ***
bmi          0.04915    0.02346   2.095   0.0372 *
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 1.431 on 250 degrees of freedom
Multiple R-squared:  0.01725, Adjusted R-squared:  0.01332
F-statistic: 4.387 on 1 and 250 DF,  p-value: 0.03722

```

From these results we can see the estimates of a and b , the latter called bmi although it is the coefficient for BMI. The BMI coefficient is estimated at 0.049 and the p-value is 0.037 for the test of the hypothesis $b = 0$. This means that we reject the null hypothesis ($H_0 : b = 0$) if we use the common significance level 0.05.

Another information which can be of value is the multiple R-squared. It is a measure of the proportion of the total variation (of the independent variable) that is explained by the regression model.

`str(res)` The result object include a lot of information. We can also plot the data and look at the result by the `plot` default for an object of this class.

```

#par(mfrow=c(1,2))
plot(kolester~bmi,data=res$model,cex=0.7) # same as plot(kolester~bmi,data=norsjo)
abline(res$coefficients,col=2)

abline(coef(res),col=2) # alternative

```

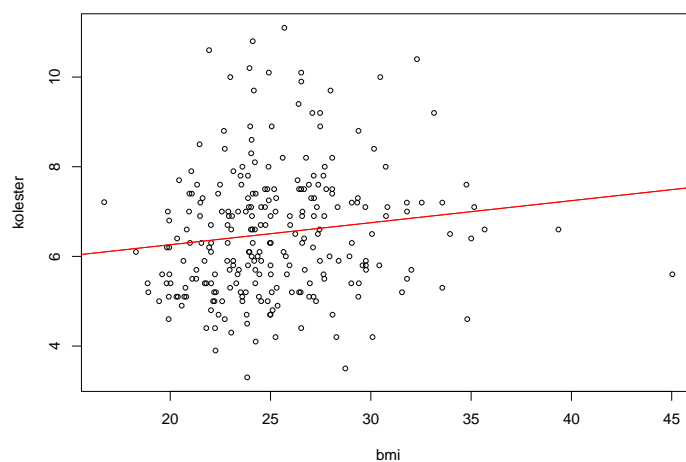
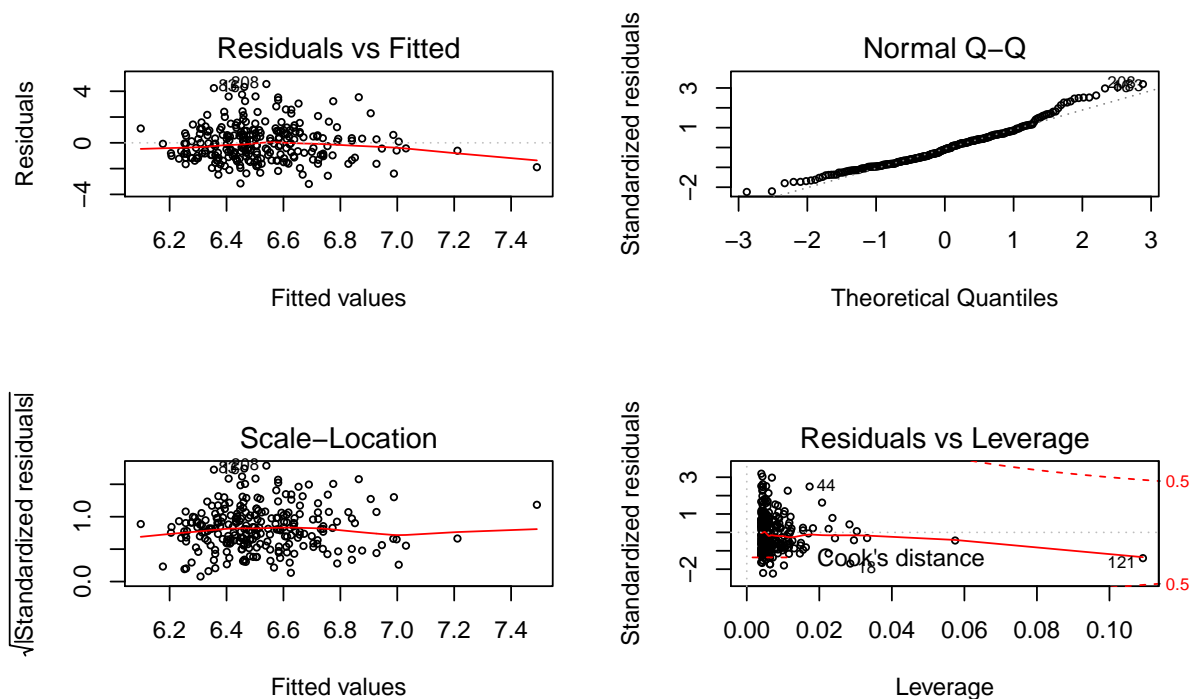


Figure 18.1: Cholesterol vs BMI and fitted model

```

par(mfrow=c(2,2))
plot(res,cex=0.6) # plot is generic, here it gives four plots
par(mfrow=c(1,1))

```



Residuals are $y_i - \hat{y}_i$ where \hat{y}_i is the estimated y_i given x_i . Thus it is the deviation of the observed value from the predicted value. Residuals are very useful for analysing the goodness of fit of the model. However, in multiple regression where the possibilities of graphic illustration of y vs the independent variables are limited residuals are more useful.

18.2.2 Multiple linear regression

Multiple linear regression is similar as simple regression but include two or more predictors. One difference is that in simple regression you can plot y vs x . It is not possible with a multiple model. Assume we want to extend the model above including age. Let us first check the age variable by making a table on its values. Observe how the formula object is used.

```
table(norsjo$agegrp)

30 40 50 60
62 66 61 63

res<-lm(kolester~bmi+agegrp,data=norsjo)
summary(res)

Call:
lm(formula = kolester ~ bmi + agegrp, data = norsjo)

Residuals:
    Min       1Q   Median       3Q      Max
-3.2112 -0.9551 -0.0558  0.6956  4.0167

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  3.592065   0.586747   6.122 3.57e-09 ***
bmi           0.015922   0.021682   0.734  0.463
agegrp       0.056147   0.007472   7.514 1.03e-12 ***
```



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

Residual standard error: 1.294 on 249 degrees of freedom
Multiple R-squared:  0.1989, Adjusted R-squared:  0.1925
F-statistic: 30.91 on 2 and 249 DF,  p-value: 1.019e-12
```

In the results we can see the estimates of intercept and the parameters for BMI and agegrp. The agegrp coefficient is estimated at 0.056 and the p-value is 0 for the test of the hypothesis saying the parameter is zero. An interpretation is that for each year of increase in age the mean cholesterol level will increase 0.056.

You can also see that the estimate for BMI has changed to 0.016. Also its p-value has changed to 0.463 and is no longer significant.

An interpretation of the change of estimation and p-value can be that age is explaining most of the relationship between BMI and cholesterol we saw in the simple regression model.

18.2.3 Including factors in the regression model

The variable agegroup was estimated as a continuous variable. However, only individuals aged 30, 40, 50 and 60 years are included so we can also think of agegrp as a categorical variable. If we convert agegrp to a factor in the model we get a different result.

```
res<-lm(kolester~bmi+as.factor(agegrp),data=norsjo)
summary(res)

Call:
lm(formula = kolester ~ bmi + as.factor(agegrp), data = norsjo)

Residuals:
    Min       1Q   Median       3Q      Max
-3.1476 -0.9234 -0.0442  0.7456  3.8675

Coefficients:
              Estimate Std. Error t value Pr(>|t|)
(Intercept)    5.35783    0.54969   9.747 < 2e-16 ***
bmi             0.01353    0.02188   0.619  0.5368
as.factor(agegrp)40 0.43900    0.23005   1.908  0.0575 .
as.factor(agegrp)50 1.24829    0.23957   5.211 3.97e-07 ***
as.factor(agegrp)60 1.60695    0.23535   6.828 6.64e-11 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 1.296 on 247 degrees of freedom
Multiple R-squared:  0.2031, Adjusted R-squared:  0.1902
F-statistic: 15.74 on 4 and 247 DF,  p-value: 1.731e-11
```

The estimates `as.factor(agegrp)40`, `as.factor(agegrp)50`, `as.factor(agegrp)60` are actually the differences from the reference group which is 30 years. We can see that a variable for age 30 is not printed out. Since there are 4 age groups but the estimates are differences from age 30 there will be only 3 estimates. For example age 40 has 0.439 higher cholesterol than age 30.

When you use categories (factors) in the model they are converted to dummy variables which are used in the model instead of the original factor. A dummy variable has either the value 1 or 0. If the factor have two levels one dummy variable will be created and used. In our example with 4

levels there will be 3 dummies. The estimates in the results are the coefficients for the dummies. The **contrasts** function give information on the dummy variables used with the dummy variables in the columns below. For example for age group 30 all three dummies are = 0 but for age group 50 the dummy variable for 50 is = 1 and the other two dummies = 0.

However, the converting of factors to dummy variables or other kinds of variables (contrasts) can be made in different ways. The default we will use is calculated by **contr.treatment** based on the levels, see below. Let us look at agegrp as a factor. It can be a good idea to include the new variable in the data frame. We see that it has four levels; 30, 40, 50 and 60.

```
fage<-as.factor(norsjo$agegrp)
norsjo<-cbind(norsjo,fage) # add the factor to the data frame

levels(norsjo$fage)

[1] "30" "40" "50" "60"

contrasts(norsjo$fage)

      40 50 60
30    0  0  0
40    1  0  0
50    0  1  0
60    0  0  1

contr.treatment(1:4)

      2 3 4
1 0 0 0
2 1 0 0
3 0 1 0
4 0 0 1

contr.treatment(levels(norsjo$fage))

      40 50 60
30    0  0  0
40    1  0  0
50    0  1  0
60    0  0  1
```

Some examples of other contrasts.

```
contr.sum(levels(norsjo$fage))

      [,1] [,2] [,3]
30      1    0    0
40      0    1    0
50      0    0    1
60     -1   -1   -1

contr.SAS(1:4)

      1 2 3
1 1 0 0
2 0 1 0
3 0 0 1
4 0 0 0

options()$contrasts # using contr.treatment is the default
```

| unordered | ordered |
|-------------------|--------------|
| "contr.treatment" | "contr.poly" |

If we use the new variable `fage` in the regression we see that the names of the estimates was changed but everything else are the same.

```
res.n2<-lm(kolester~bmi+fage,data=norsjo)
summary(res.n2)

Call:
lm(formula = kolester ~ bmi + fage, data = norsjo)

Residuals:
    Min       1Q   Median       3Q      Max
-3.1476 -0.9234 -0.0442  0.7456  3.8675

Coefficients:
              Estimate Std. Error t value Pr(>|t|)
(Intercept)   5.35783     0.54969   9.747 < 2e-16 ***
bmi            0.01353     0.02188   0.619  0.5368
fage40         0.43900     0.23005   1.908  0.0575 .
fage50         1.24829     0.23957   5.211 3.97e-07 ***
fage60         1.60695     0.23535   6.828 6.64e-11 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 1.296 on 247 degrees of freedom
Multiple R-squared:  0.2031, Adjusted R-squared:  0.1902
F-statistic: 15.74 on 4 and 247 DF, p-value: 1.731e-11
```

Changing the reference level of a factor

What if we don't want the default reference category (30 years). We can change the reference level by the `relevel` function. Assume we want to change it to age 50.

```
norsjo$newfage<-relevel(norsjo$fage,"50")

levels(norsjo$newfage)

[1] "50" "30" "40" "60"

res<-lm(kolester~bmi+newfage,data=norsjo)
summary(res)

Call:
lm(formula = kolester ~ bmi + newfage, data = norsjo)

Residuals:
    Min       1Q   Median       3Q      Max
-3.1476 -0.9234 -0.0442  0.7456  3.8675

Coefficients:
              Estimate Std. Error t value Pr(>|t|)
(Intercept)   6.60612     0.60050  11.001 < 2e-16 ***
bmi            0.01353     0.02188   0.619 0.536762
```

```

newfage30    -1.24829    0.23957   -5.211  3.97e-07 ***
newfage40    -0.80929    0.23251   -3.481  0.000591 ***
newfage60     0.35866    0.23310    1.539  0.125170
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 1.296 on 247 degrees of freedom
Multiple R-squared:  0.2031, Adjusted R-squared:  0.1902
F-statistic: 15.74 on 4 and 247 DF,  p-value: 1.731e-11

```

Now the newfage30 estimate the difference compared to newfage50. Compare with the earlier result with age 30 as reference. It is only the sign that have changed.

18.3 Analysis of variance - ANOVA

Let us start looking at the linear regression above including agegrp only. We can there get an estimate and test of three of the age groups compared to the reference group. This is similar as above but without BMI.

```

lm.age<-lm(kolester~fage,data=norsjo)
summary(lm.age)

Call:
lm(formula = kolester ~ fage, data = norsjo)

Residuals:
    Min       1Q   Median       3Q      Max
-3.1143 -0.9409 -0.0727  0.7665  3.8369

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   5.6823     0.1644   34.568 < 2e-16 ***
fage40         0.4512     0.2289    1.971  0.0498 *
fage50         1.2809     0.2334    5.487 1.01e-07 ***
fage60         1.6320     0.2315    7.049 1.78e-11 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 1.294 on 248 degrees of freedom
Multiple R-squared:  0.2019, Adjusted R-squared:  0.1922
F-statistic: 20.91 on 3 and 248 DF,  p-value: 4.127e-12

```

18.3.1 One-way anova

The p-values for example fage40 at 0.05 tell us that the cholesterol level for age group 40 is significantly different from the reference age group 30 years. If we want to test age group as one factor meaning the mean level of cholesterol is equal for all the four age groups (the null hypothesis) the most straightforward test is ANOVA (one-way analysis of variance) by using the function `aov`. Here the contrasts does not matter.

```

aov.age<-aov(kolester~fage,data=norsjo)
summary(aov.age)

```

```

      Df Sum Sq Mean Sq F value    Pr(>F)
fage      3  105.1    35.03   20.91 4.13e-12 ***
Residuals 248  415.5     1.68
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

```

The p-value is calculated from the test statistic F-value 20.91 and is statistically significant.

ANOVA is based on the sums of squares (Sum Sq) measuring the variation between and within the groups. The ratio between them is the basis for F so a large F occurs if there is more variation between the groups than expected had the null hypothesis been true. The null hypothesis state that the population mean values are the same in all groups.

However, if you take a look at the last row of the result from the linear regression model with age groups above you will find the same F-value and test result there. The null hypothesis says that all parameters = 0 except the intercept.

18.3.2 Likelihood ratio test

We can generalize the F test above. It is actually a comparison between two nested models. As an example we compare the model including age group only compared to a model including both sbp and bmi.

```

lm.m1<-lm(kolester~fage,data=norsjo)
lm.m2<-lm(kolester~fage+sbp+bmi,data=norsjo)

anova(lm.m1,lm.m2)

Analysis of Variance Table

Model 1: kolester ~ fage
Model 2: kolester ~ fage + sbp + bmi
  Res.Df  RSS Df Sum of Sq    F  Pr(>F)
1    248 415.47
2    246 394.28  2    21.183 6.6081 0.001602 **
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

```

OBS remember that we first removed all missing data from the data. This comparison can only be made if the number of observations are the same in the two models. We can see that the result of the **anova** (Likelihood ratio test) is significant saying that we can reject the hypothesis that the parameters of bmi and sbp both =0. Remember that earlier we showed that BMI was not significant so probably most of the result is due to sbp. Let us check.

```

round(summary(lm.m2)$coef,4)

      Estimate Std. Error t value Pr(>|t|)
(Intercept)  3.8115      0.6892  5.5305  0.0000
fage40        0.3944      0.2251  1.7523  0.0810
fage50        1.0251      0.2422  4.2325  0.0000
fage60        1.3267      0.2429  5.4626  0.0000
sbp           0.0171      0.0048  3.5798  0.0004
bmi          -0.0038      0.0219 -0.1749  0.8613

```

We can see that sbp is statistically significant but not bmi as suspected. The result of summary was here restricted using **coef**.

18.3.3 Two-way anova

Now we study both the factors agegrp and sbp split into three categories.

```
sbpcat<-cut(norsjo$sbp,c(0,110,130,Inf))
is.factor(sbpcat)

[1] TRUE

levels(sbpcat)

[1] "(0,110]" "(110,130]" "(130,Inf]"

norsjo<-cbind(norsjo,sbpcat) # alternative to add a variable to a data frame
head(norsjo)

  agegrp sex height weight sbp dbp kolester smoker    bmi fage newfage  sbpcat
1     60   2   157    61 110   70     6.7      0 24.74745   60     60 (0,110]
2     60   2   157    97 150  100     6.6      0 39.35251   60     60 (130,Inf]
3     60   1   170    74 136   96     8.2      0 25.60554   60     60 (130,Inf]
4     60   2   163    66 156   76     7.5      0 24.84098   60     60 (130,Inf]
5     60   2   166    66 110   70    10.2      0 23.95123   60     60 (0,110]
6     60   2   168    61 130   78     7.3      0 21.61281   60     60 (110,130]

aov.agesbp<-aov(kolester~fage+sbpcat,data=norsjo)
summary(aov.agesbp)

              Df Sum Sq Mean Sq F value    Pr(>F)
fage              3  105.1   35.03   21.666 1.74e-12 ***
sbpcat            2   17.7    8.85    5.474 0.00472 **
Residuals       246   397.8    1.62
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

18.3.4 Interaction

The effects we have estimated so far is usually called main effects. When we analyse a two-way anova we may also want to include the interaction between the two factors in the model. Interaction can be described as if the effect of factor 1 is dependent on the level of factor 2. For example if the dependent outcome variable is blood pressure and the two factors are smoking (yes/no) and sex. Without interaction we can estimate the general effect of smoking on blood pressure. However, with interaction the effect of smoking is different for men and women.

Interaction can be estimated in all linear models. It can also be estimated for two continuous variables e.g. cholesterol and bmi or one factor and one continuous variable, eg. bmi and smoking.

In R formula we use the * which adds the interaction parameters in the model. Also : can be used but it has a somewhat different meaning.

```
aov.agesbp.int<-aov(kolester~fage*sbpcat,data=norsjo)
summary(aov.agesbp.int)

              Df Sum Sq Mean Sq F value    Pr(>F)
fage              3  105.1   35.03   21.278 2.95e-12 ***
sbpcat            2   17.7    8.85    5.376 0.0052 **
fage:sbpcat       6    2.6    0.44    0.266 0.9525
Residuals       240   395.1    1.65
```

```

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

aov.agesbp.int<-aov(kolester~fage:sbpcat,data=norsjo)
summary(aov.agesbp.int)

              Df Sum Sq Mean Sq F value    Pr(>F)
fage:sbpcat   11  125.4   11.402    6.925 3.86e-10 ***
Residuals    240   395.1    1.646
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

```

18.4 Generalized linear models

We will not go through the theory behind generalised linear models but show some examples how to estimate and interpret the results. Within this family we can find some common distributions which can be estimated in a similar way as for linear regression with only minor modifications. The function used for this is **glm**. Normal distribution also belong to this family. Actually we don't seem necessary because of the functions used so far but let start to compare with the results from those. Let us also compare the class of the resulting objects.

```

glm.m2<-glm(kolester~fage+sbp+bmi,data=norsjo)
summary(glm.m2)

Call:
glm(formula = kolester ~ fage + sbp + bmi, data = norsjo)

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-3.0829  -0.8835  -0.0724   0.7454   4.0417

Coefficients:
              Estimate Std. Error t value Pr(>|t|)
(Intercept)  3.811453    0.689174   5.530 8.15e-08 ***
fage40        0.394414    0.225086   1.752 0.080972 .
fage50        1.025092    0.242196   4.232 3.27e-05 ***
fage60        1.326732    0.242874   5.463 1.15e-07 ***
sbp           0.017069    0.004768   3.580 0.000414 ***
bmi          -0.003833    0.021919  -0.175 0.861326
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for gaussian family taken to be 1.602775)

    Null deviance: 520.56  on 251  degrees of freedom
Residual deviance: 394.28  on 246  degrees of freedom
AIC: 841.95

Number of Fisher Scoring iterations: 2

summary(lm.m2) # Same result with glm and lm

Call:
lm(formula = kolester ~ fage + sbp + bmi, data = norsjo)

Residuals:

```

```

      Min       1Q   Median       3Q      Max
-3.0829 -0.8835 -0.0724  0.7454  4.0417

Coefficients:
              Estimate Std. Error t value Pr(>|t|)
(Intercept)  3.811453   0.689174   5.530 8.15e-08 ***
fage40       0.394414   0.225086   1.752 0.080972 .
fage50       1.025092   0.242196   4.232 3.27e-05 ***
fage60       1.326732   0.242874   5.463 1.15e-07 ***
sbp          0.017069   0.004768   3.580 0.000414 ***
bmi         -0.003833   0.021919  -0.175 0.861326
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 1.266 on 246 degrees of freedom
Multiple R-squared:  0.2426, Adjusted R-squared:  0.2272
F-statistic: 15.76 on 5 and 246 DF, p-value: 1.861e-13

```

As we can see the results are similar for `lm` and `glm` and have a similar structure but there are some differences. If you check the help for `glm` you can see that there are some more arguments than for `lm`.

One of the arguments for `glm` is `family` (= gaussian by default). Gaussian is the same as normal distribution which is the reason why we got the same result as above - it is actually the same model. However, choosing another family we will get something completely different. You may check the alternatives for family in the help.

The link function is describing how the expected observations relates to the linear predictors, e.g. linear or log-linear models with linear and log links, respectively. We don't need to bother about the link and other specifications except family, because the default arguments for a given family is most often what we want.

If we check the class we can see that `glm` belong to the same class as `lm` but also to another namely `glm`.

```

class(glm.m2)

[1] "glm" "lm"

class(lm.m2)

[1] "lm"

```

18.4.1 Logistic regression

Here the response variable is binary i.e. it can only have two values e.g. yes/no, 1/0, sick/healthy etc. We have to change the family object to binomial (or similarly binomial()). Below we analyse if bmi can explain the probability of having high diastolic blood pressure. If the independent variable is a factor the "event" is the second level.

```

dbpcat<-cut(norsjo$dbp,c(0,80,Inf))
levels(dbpcat)

[1] "(0,80]" "(80,Inf]"

norsjo<-cbind(norsjo,dbpcat)

```



```
bin.m1<-glm(dbpcat~bmi,data=norsjo,family=binomial)
summary(bin.m1)
```

Call:
glm(formula = dbpcat ~ bmi, family = binomial, data = norsjo)

Deviance Residuals:

| Min | 1Q | Median | 3Q | Max |
|---------|---------|---------|--------|--------|
| -2.0385 | -0.9238 | -0.6849 | 1.1251 | 2.0695 |

Coefficients:

| | Estimate | Std. Error | z value | Pr(> z) |
|-------------|----------|------------|---------|--------------|
| (Intercept) | -6.18343 | 1.09608 | -5.641 | 1.69e-08 *** |
| bmi | 0.22784 | 0.04283 | 5.319 | 1.04e-07 *** |

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

(Dispersion parameter for binomial family taken to be 1)

Null deviance: 339.36 on 251 degrees of freedom
Residual deviance: 303.38 on 250 degrees of freedom
AIC: 307.38

Number of Fisher Scoring iterations: 4

The estimated parameters here are log(odds ratio). The positive estimate of bmi indicate an increasing odds (risk/(1-risk)) for high diastolic blood pressure with increasing bmi and it is statistically significant. However, usually it is more interesting to get an estimate of the odds ratio (see below).

18.4.2 Confidence intervals

A convenient way to get the confidence intervals and also the corresponding exponentiated estimates and confidence intervals can be calculated by functions in the library **Epi**. The function **ci.lin** can be used for the linear scale e.g. output from **lm** or **glm**. It may be convenient to round off the result. You can change the confidence level (95% is default) with the argument **alpha**. In log-linear models like logistic regression we may be more interested in odds ratios than log(odds ratios). To get these estimates and confidence intervals we can use the corresponding function **ci.exp**. By default the p-value is not included but can be added. The p-value is the same as for the ordinary result on the log scale.

```
library(Epi)
round(ci.lin(lm.m2,alpha=0.1),3) # with 90% confidence interval
```

| | Estimate | StdErr | z | P | 5.0% | 95.0% |
|-------------|----------|--------|--------|-------|--------|-------|
| (Intercept) | 3.811 | 0.689 | 5.530 | 0.000 | 2.678 | 4.945 |
| fage40 | 0.394 | 0.225 | 1.752 | 0.080 | 0.024 | 0.765 |
| fage50 | 1.025 | 0.242 | 4.232 | 0.000 | 0.627 | 1.423 |
| fage60 | 1.327 | 0.243 | 5.463 | 0.000 | 0.927 | 1.726 |
| sbp | 0.017 | 0.005 | 3.580 | 0.000 | 0.009 | 0.025 |
| bmi | -0.004 | 0.022 | -0.175 | 0.861 | -0.040 | 0.032 |

```
round(ci.lin(bin.m1),3) # log(odds) ratio (95% confidence interval - default)
```

| | Estimate | StdErr | z | P | 2.5% | 97.5% |
|-------------|----------|--------|--------|---|--------|--------|
| (Intercept) | -6.183 | 1.096 | -5.641 | 0 | -8.332 | -4.035 |
| bmi | 0.228 | 0.043 | 5.319 | 0 | 0.144 | 0.312 |

```
round(ci.exp(bin.m1,pval=T),3) # odds ratio (p-value added)
```

| | exp(Est.) | 2.5% | 97.5% | P |
|-------------|-----------|-------|-------|---|
| (Intercept) | 0.002 | 0.000 | 0.018 | 0 |
| bmi | 1.256 | 1.155 | 1.366 | 0 |

18.5 Survival analysis

Survival analysis means methods used to analyse data where the response is a waiting time for a certain event. There are two things that are distinctive; the observations are always positive and all events may not have happened at the time of the analysis. These non-complete observations, called censorings, also carry information and have to be included in the analysis to avoid bias.

18.5.1 Survival data

Anderson data set

The data Anderson.dta consists of remission time data for two groups of leukemia patients with 21 patients in each group.

| | |
|------------|---|
| survt: | Remission time in weeks is denoted and the variable |
| status: | censored=0, relapse (which is the event)=1 |
| sex: | female=0, male=1 |
| logwbc: | a well-known prognostic indicator of survival for leukemia patients |
| rx: | (Treatment group =0, Placebo group=1) |
| catlogwbc: | the variable logWBC divided into low=1, medium=2 and high values=3 |

```
library(haven)
adf <- read_dta("../data/Anderson.dta")
head(adf)

# A tibble: 6 x 6
  survt      status      sex logwbc      rx catlogwbc
  <dbl>    <dbl+lbl> <dbl+lbl> <dbl>    <dbl+lbl>    <dbl>
1    19 0 [censored] 0 [female]  2.05 0 [treatment]      1
2    17 0 [censored] 0 [female]  2.16 0 [treatment]      1
3    13 1 [relapse]  0 [female]  2.88 0 [treatment]      2
4    11 0 [censored] 0 [female]  2.6  0 [treatment]      2
5    10 0 [censored] 0 [female]  2.7  0 [treatment]      2
6    10 1 [relapse]  0 [female]  2.96 0 [treatment]      2
```

18.5.2 The survival object

The observed data (independent variable) have two dimensions: time to event and status (status=if the time is an event or a censored observation). There is a function **Surv** for defining such survival object.

```
library(survival)
survtime<-Surv(adf$survt,adf$status)
class(survtime)

[1] "Surv"
```

18.5.3 Survival curves - Kaplan Meier

A survival curve show the probability of the waiting time to exceed the value on the x-axis. It is the same as one minus the cumulative distribution function (for a continuous variable).

```
adf<-cbind(adf,survtime)
sf<-survfit(survtime~1,data=adf)
sf
```

Call: `survfit(formula = survtime ~ 1, data = adf)`

| | n | events | median | 0.95LCL | 0.95UCL |
|--|----|--------|--------|---------|---------|
| | 42 | 30 | 12 | 8 | 22 |

```
summary(sf) # events but not censorings are shown
```

Call: `survfit(formula = survtime ~ 1, data = adf)`

| time | n.risk | n.event | survival | std.err | lower 95% CI | upper 95% CI |
|------|--------|---------|----------|---------|--------------|--------------|
| 1 | 42 | 2 | 0.952 | 0.0329 | 0.8901 | 1.000 |
| 2 | 40 | 2 | 0.905 | 0.0453 | 0.8202 | 0.998 |
| 3 | 38 | 1 | 0.881 | 0.0500 | 0.7883 | 0.985 |
| 4 | 37 | 2 | 0.833 | 0.0575 | 0.7279 | 0.954 |
| 5 | 35 | 2 | 0.786 | 0.0633 | 0.6709 | 0.920 |
| 6 | 33 | 3 | 0.714 | 0.0697 | 0.5899 | 0.865 |
| 7 | 29 | 1 | 0.690 | 0.0715 | 0.5628 | 0.845 |
| 8 | 28 | 4 | 0.591 | 0.0764 | 0.4588 | 0.762 |
| 10 | 23 | 1 | 0.565 | 0.0773 | 0.4325 | 0.739 |
| 11 | 21 | 2 | 0.512 | 0.0788 | 0.3783 | 0.692 |
| 12 | 18 | 2 | 0.455 | 0.0796 | 0.3227 | 0.641 |
| 13 | 16 | 1 | 0.426 | 0.0795 | 0.2958 | 0.615 |
| 15 | 15 | 1 | 0.398 | 0.0791 | 0.2694 | 0.588 |
| 16 | 14 | 1 | 0.369 | 0.0784 | 0.2437 | 0.560 |
| 17 | 13 | 1 | 0.341 | 0.0774 | 0.2186 | 0.532 |
| 22 | 9 | 2 | 0.265 | 0.0765 | 0.1507 | 0.467 |
| 23 | 7 | 2 | 0.189 | 0.0710 | 0.0909 | 0.395 |

For example at time=17 you can see that there are 13 at risk (just before) and 1 death. At time 22 there are 9 at risk. Thus there must be 3 censorings between time 17 and 22. It can be seen in the plot.

18.5.4 Comparing Survival curves - log rank test

The same function `survfit` is also used to calculate Kaplan-meier estimates for different groups and to test if they are equal. For a test of the null hypothesis: the survival is the same in the groups, we use log-rank test (function `survdif`). In the example we compare treated ($rx=0$) vs not treated ($rx=1$).

```
adf<-cbind(adf,frx=factor(adf$rx,labels=c("treatment","placebo")))
sf2<-survfit(survtime~frx,data=adf)
par(mfrow=c(1,2))

survdif(survtime~rx,data=adf) # log-rank test
```

Call:

```
survdif(formula = survtime ~ rx, data = adf)
```

| | N | Observed | Expected | (O-E) ² /E | (O-E) ² /V |
|--|---|----------|----------|-----------------------|-----------------------|
|--|---|----------|----------|-----------------------|-----------------------|

```
par(mfrow=c(1,2))
plot(sf) # 95% confidence interval by default
plot(sf,conf.int=F,mark.time=T) # remove CI and add the censorings
```

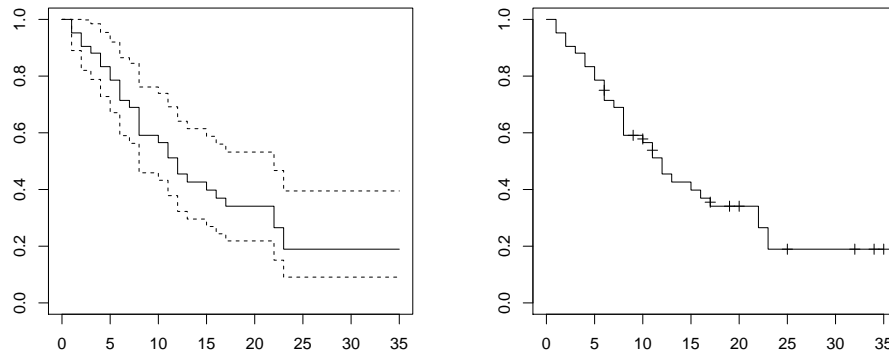


Figure 18.2: Kaplan Meier curves with confidence interval and with censorings, respectively

| | | | | | |
|------|----|----|------|------|------|
| rx=0 | 21 | 9 | 19.3 | 5.46 | 16.8 |
| rx=1 | 21 | 21 | 10.7 | 9.77 | 16.8 |

Chisq= 16.8 on 1 degrees of freedom, p= 4e-05

```
par(mfrow=c(1,2))
plot(sf2,col=c(1,2),lty=c(1,2)) # different color and pattern of the curves added
legend(20,0.95,legend=c(levels(adf$frx)),col=c(1,2),lty=c(1,2),cex=0.8)
plot(sf2,col=c(1,2),lty=c(1,2),fun="cloglog") # plot of log-log(S(t))
```

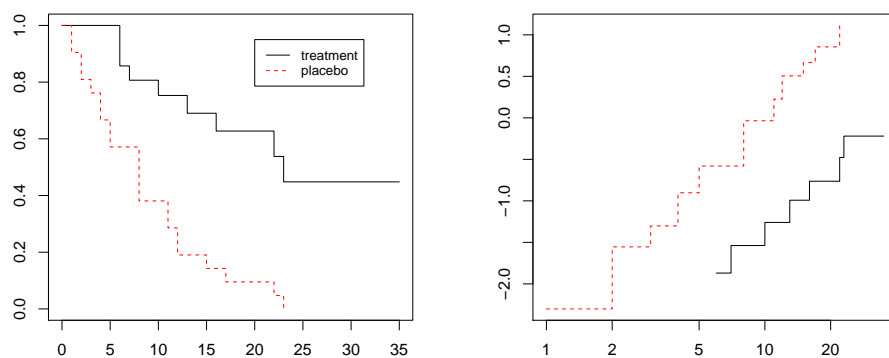


Figure 18.3: Kaplan Meier and log-log transformation of Kaplan Meier

The log-rank test is based on the observed and expected numbers. In this example there is a statistically significant difference between treated and not treated.

Hazard is a rate which can be thought of as e.g. mortality at a certain age or at a certain time of

follow-up. The second plot is showing the log-log transformed curve which can be used to check if the so called hazard functions are proportional. If the curves are parallel this is fulfilled.

18.6 Cox models

A Cox model is a regression method similar as the ones we have seen above where the outcome (independent) variable is a survival time and include censorings. It can be described as $\lambda(t; \mathbf{z}) = \lambda_0(t) \exp(z_1 \beta_1 + z_2 \beta_2 + \dots)$ where λ is the hazard function, \mathbf{z} is a vector of predictors and β_i the parameters. It is not part of the generalised linear models family so we need another function for the estimation. It is a so called semi-parametric method. The parameters are estimates in a similar way as for other regression models but the survival is not estimated by a parametric method. However, there is an assumption that the hazards should be proportional. This can also be described as; the hazard ratios (HR) are independent of time.

Data set Addicts

We will use the Addicts data set for an example. In a 1991 Australian study by Caplehorn et al., two methadone treatment clinics for heroin addicts were compared to assess patient time remaining under methadone treatment. A patients treatment time was determined as the time, in days, until the person dropped out of the clinic

| | |
|-------------|--|
| ID-Patient: | ID |
| Clinic: | Indicated which methadone treatment clinic the patient attended (coded 1 or 2) |
| Status: | Indicates if the individual dropped out (coded 1) or was censored (coded 0) |
| Survt: | The time (in days) until the patient dropped out of the clinic. |
| Prison: | Indicates whether the patient had a prison record (coded 1) or not (coded 0) |
| Dose: | A continuous variable for the patients maximum methadone dose (mg/day). |
| Dosekat: | 1 if Dose <65, 2 if Dose 65 or higher |

```
addicts <- read_dta("../data/addicts.dta")
addicts

# A tibble: 238 x 6
  Clinic Status Survt Prison Dose Dosekat
  <dbl>   <dbl> <dbl>   <dbl> <dbl>   <dbl>
1     1     1    428     0    50     1
2     1     1    275     1    55     1
3     1     1    262     0    55     1
4     1     1    183     0    30     1
5     1     1    259     1    65     2
6     1     1    714     0    55     1
7     1     1    438     1    65     2
8     1     0    796     1    60     1
9     1     1    892     0    50     1
10    1     1    393     1    65     2
# ... with 228 more rows

dim(addicts)

[1] 238  6
```

As before we have to start creating a survival object.

```

survtime<-Surv(addicts$Survtime,addicts$Status)
fclinic<-as.factor(addicts$Clinic)
addicts<-cbind(addicts,survtime,fclinic) # add to the data frame

fitc<-coxph(survtime~Dose+fclinic,data=addicts)
summary(fitc)

Call:
coxph(formula = survtime ~ Dose + fclinic, data = addicts)

n= 195, number of events= 123
(43 observations deleted due to missingness)

              coef exp(coef)    se(coef)      z Pr(>|z|)
Dose      -0.028668  0.971739  0.007191 -3.987 6.69e-05 ***
fclinic2  -0.939289  0.390906  0.232238 -4.045 5.24e-05 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

              exp(coef) exp(-coef) lower .95 upper .95
Dose           0.9717      1.029    0.9581    0.9855
fclinic2       0.3909      2.558    0.2480    0.6162

Concordance= 0.627 (se = 0.031 )
Likelihood ratio test= 36.83 on 2 df,  p=1e-08
Wald test              = 32.59 on 2 df,  p=8e-08
Score (logrank) test = 34.07 on 2 df,  p=4e-08

```

18.6.1 Model check

We start with looking at the plot of survival and log-log plot for the two clinics. The curves in the second plot is not parallel, i.e. the condition of proportional hazards don't seem to be fulfilled.

```

sfc<-survfit(survtime~fclinic,data=addicts)
par(mfrow=c(1,2))
plot(sfc,col=c(1,2))
plot(sfc,fun="cloglog",col=c(1,2))

```

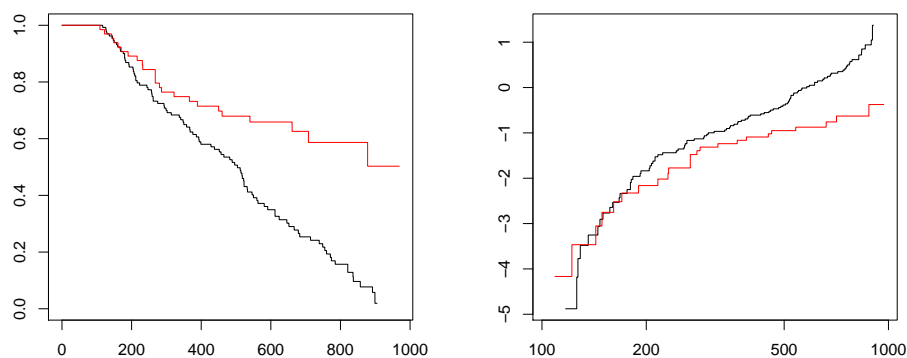


Figure 18.4: Kaplan Meier and log-log transformation of Kaplan Meier for clinic

Proportional hazards can also be tested looking at time specific estimates using the `cox.zph` function. This function also provides a plot of an estimate of the time-dependent coefficient $\beta(t)$. If the proportional hazards assumption holds then the true $\beta(t)$ function would be a horizontal line.

```
ph<-cox.zph(fitc)
ph
```

```
      chisq df      p
Dose    0.00717 1 0.9325
fclinic 7.38630 1 0.0066
GLOBAL  7.42611 2 0.0244
```

```
par(mfrow=c(1,2))
plot(ph)
```

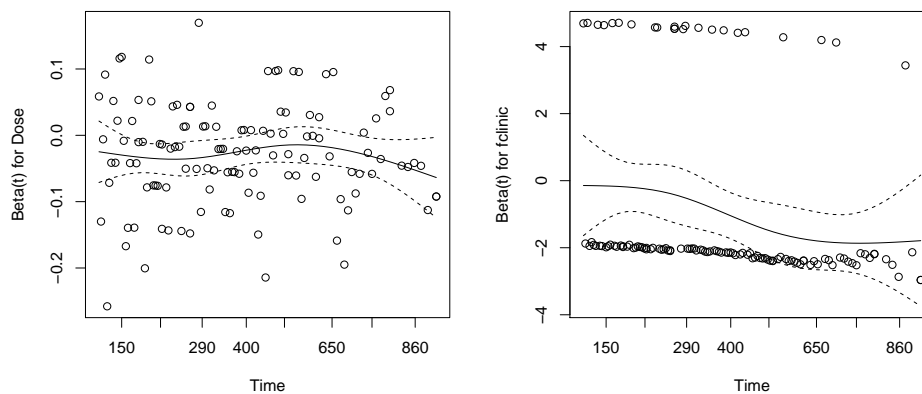


Figure 18.5: Kaplan Meier and log-log transformation of Kaplan Meier for clinic

18.6.2 Stratification

Now although the results show that both dose and clinic are statistically significant it seems inappropriate to use this model. However, we can overcome this problem (especially if we are not specifically interested in the hazard ratio of clinic) by stratifying on clinic.

```
library(survival)
fitcs<-coxph(survtime~Dose+strata(fclinic),data=addicts)
summary(fitcs)
```

Call:

```
coxph(formula = survtime ~ Dose + strata(fclinic), data = addicts)
```

```
n= 195, number of events= 123
(43 observations deleted due to missingness)
```

```
      coef exp(coef)  se(coef)      z Pr(>|z|)
Dose -0.028061  0.972329  0.007264 -3.863 0.000112 ***
```

```
---
```

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

```
      exp(coef) exp(-coef) lower .95 upper .95
Dose    0.9723    1.028    0.9586    0.9863
```

```
Concordance= 0.602 (se = 0.031 )  
Likelihood ratio test= 14.81 on 1 df, p=1e-04  
Wald test = 14.93 on 1 df, p=1e-04  
Score (logrank) test = 15.13 on 1 df, p=1e-04
```

If we compare the results of HR for dose the new result 0.972 using stratification on clinic does not differ much from the model without stratification. This is not always the case.

Chapter 19

Graphics with ggplot2 - part 2

Håkan Jonsson < hakan.jonsson@umu.se >

19.1 Overplotting

With integer data or data with many categories it is usually a problem because many points can be at the same position. We can to some extent overcome this by adding some randomness into the plot using `geom_jitter`. We use the BackPain data for illustration. The data set is large so we use only a random sample and remove the missing data.

```
library(readr)
BackPain<-read_csv("../data/BackPain.csv") # The resulting object is a tibble

# remove missing data in some variables
bp<-BackPain %>% filter(complete.cases(bmi,residence,physical,waistc,height))

set.seed(1001)
bp<-bp[sample(nrow(bp),10000),]
bp

# A tibble: 10,000 x 25
   id residence sex    age wealthQ physical country backPain30 agegr maritalS eduS
   <dbl> <chr>   <chr> <dbl> <chr>   <chr>   <chr>   <chr>   <chr> <chr>   <chr>
1    679 Rural   Male   56 Q2    high ph~ India   No      50-59 Married~ No p~
2   8863 Rural   Fema~  56 Q3    high ph~ China   No      50-59 Married~ No p~
3  33223 Urban   Male   58 Q4    high ph~ Russia~ Yes     50-59 Married~ Comp~
4  30317 Urban   Fema~  73 Q2    high ph~ China   Yes     70-79 Married~ No p~
5  15269 Urban   Male   53 Q5 ric~ mod phy~ China   No      50-59 Married~ Comp~
6  30488 Urban   Fema~  69 Q5 ric~ mod phy~ Russia~ Yes     60-69 Div/Wid~ Comp~
7  20081 Rural   Male   71 Q2    low phy~ Russia~ No      70-79 Married~ Comp~
8   5997 Urban   Fema~  60 Q5 ric~ mod phy~ China   No      60-69 Div/Wid~ Comp~
9  31587 Urban   Fema~  72 Q3    mod phy~ China   Yes     70-79 Married~ Comp~
10 13102 Rural   Male   64 Q4    low phy~ Mexico  No      60-69 Married~ Comp~
# ... with 9,990 more rows, and 14 more variables: workS <chr>, bmi <dbl>, bmi4 <chr>,
# waistc <dbl>, smoke <chr>, alcohol <chr>, arthritis <chr>, angina <chr>,
# depression <chr>, asthma <chr>, diabetes <chr>, comorb <dbl>, disability <dbl>,
# height <dbl>

p<-ggplot(bp,aes(y=disability,x=age))
p1<-p+geom_point(size=0.7) # not informative with integer or lattice data. There are more points th
```

```
# p+geom_point(size=1.3,color="red",shape=21)
p2<-p+geom_jitter(height=0.5,width=0.5,size=1,shape=21) # Better
```

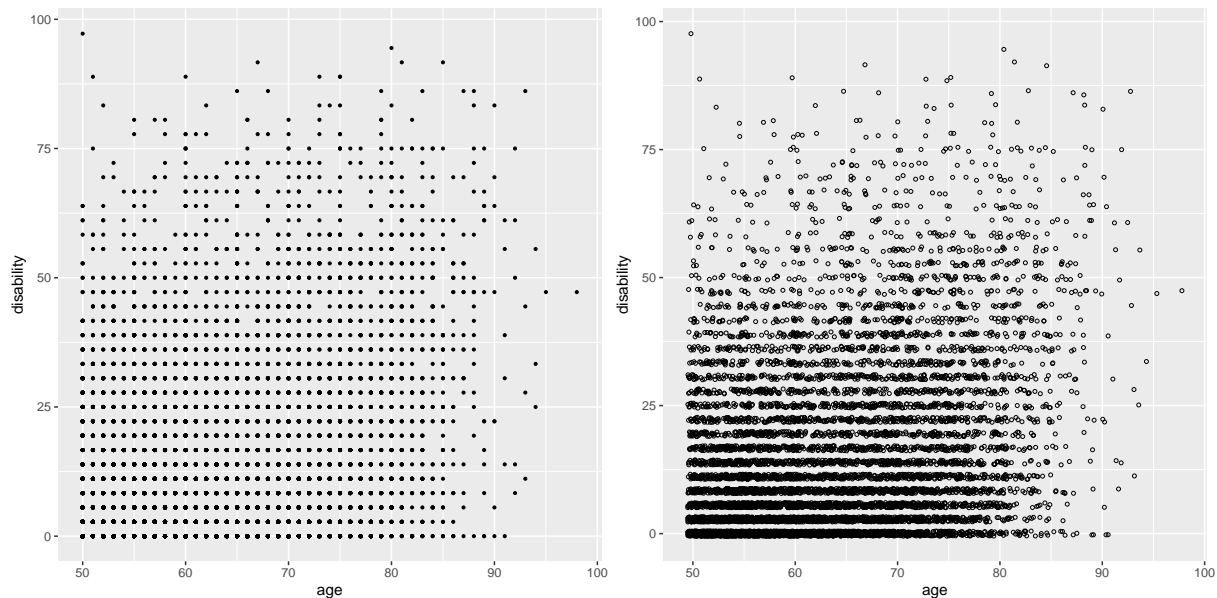


Figure 19.1: Scatterplots; normal and using geom_jitter

19.2 Multiple plots

Using lines, symbols and color. Don't forget that you can always use the R-studio menu Plots/Zoom or Plots/Export to get a better view of the plot.

19.2.1 Using different data sets in the same plot

Let us now import the data file norsjo86.

| | | |
|-----------|--------------------------|--------------------------------|
| agegrp: | Age group | (30, 40, 50 ,60 years) |
| health: | Health status | (0=good, 1=not quite good/bad) |
| sex: | Sex | (1=man, 2=woman) |
| height: | Body height | (cm) |
| weight: | Body weight | (kg) |
| sbp: | Systolic blood pressure | |
| dbp: | Diastolic blood pressure | |
| kolester: | Cholesterol | |
| smoker: | Smoking status | (0=non-smoker, 1=smoker) |
| bmi: | Body mass index | (kg/m^2) |

```
# import spss file norsjo86
library(haven)
norsjo86 <- read_sav("../data/norsjo86.sav")

norsjo86<- norsjo86 %>% filter(complete.cases(sbp,bmi,kolester))
norsjo86
```

```
# A tibble: 252 x 10
  agegrp health sex height weight sbp dbp kolester smoker bmi
<dbl+lbl> <dbl+lbl> <dbl+lb> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl+lbl> <dbl>
1 60 [60 year~ 0 [good] 2 [Woma~ 157 61 110 70 6.7 0 [Non-s~ 24.7
2 60 [60 year~ 1 [not quite ~ 2 [Woma~ 157 97 150 100 6.6 0 [Non-s~ 39.4
3 60 [60 year~ 0 [good] 1 [Man] 170 74 136 96 8.2 0 [Non-s~ 25.6
4 60 [60 year~ 0 [good] 2 [Woma~ 163 66 156 76 7.5 0 [Non-s~ 24.8
5 60 [60 year~ 0 [good] 2 [Woma~ 166 66 110 70 10.2 0 [Non-s~ 24.0
6 60 [60 year~ 0 [good] 2 [Woma~ 168 61 130 78 7.3 0 [Non-s~ 21.6
7 60 [60 year~ 1 [not quite ~ 2 [Woma~ 159 67 122 74 5.2 0 [Non-s~ 26.5
8 60 [60 year~ 0 [good] 1 [Man] 172 62 142 88 7.4 0 [Non-s~ 21.0
9 60 [60 year~ 1 [not quite ~ 2 [Woma~ 153 68 150 100 7.2 0 [Non-s~ 29.0
10 60 [60 year~ 1 [not quite ~ 1 [Man] 179 87 133 85 7.8 0 [Non-s~ 27.2
# ... with 242 more rows
```

```
gr1<-norsjo86 %>% filter(sex==1 & smoker==0)
gr2<-norsjo86 %>% filter(sex==2 & smoker==1)

p1<-ggplot(gr1,aes(y=sbp,x=bmi))+
  geom_point(size=1.5,color="blue")+
  geom_smooth(method=lm,se=F,color="blue")+
  labs(title="Non-smoking men")

p2<-p1+geom_point(data=gr2,size=1.5,color="red")+
  geom_smooth(data=gr2,method=lm,se=F,color="red")+
  labs(title="Smoking women (red) and non-smoking men ( blue)")
```

19.2.2 Using aesthetics (aes argument) to make multiple plots

The different parameters in aesthetics can be found at <https://cran.r-project.org/web/packages/ggplot2/vignettes/ggplot-specs.html>

Histogram

```
p<-ggplot(bp,aes(x=bmi))
p1<-p+geom_histogram(binwidth=1,colour="black",fill="red",alpha=0.4) # alpha=1 is full color
p2<-p+geom_histogram(aes(fill=sex),binwidth=1,colour="black",alpha=1)
```

Boxplots

```
p<-ggplot(bp,aes(x=country,y=bmi))
p1<-p+geom_boxplot(colour="blue",size=0.5) # size=thickness of box lines
p2<-p+geom_boxplot(aes(fill=sex))
p3<-p+geom_boxplot(aes(fill=agegrp),alpha=0.8)
p4<-p+geom_boxplot(aes(color=agegrp))
```

```
## 'geom_smooth()' using formula 'y ~ x'
## 'geom_smooth()' using formula 'y ~ x'
## 'geom_smooth()' using formula 'y ~ x'
```



Figure 19.2: Scatterplot with regression line, one and two subgroups

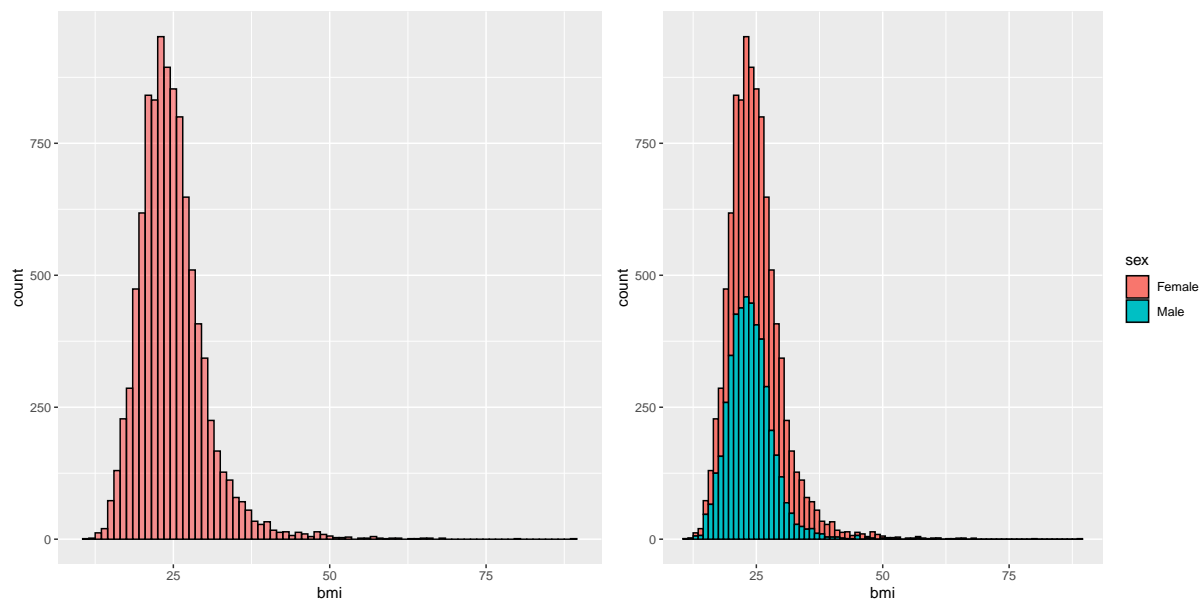


Figure 19.3: Histograms; total and with aes(fill=sex)

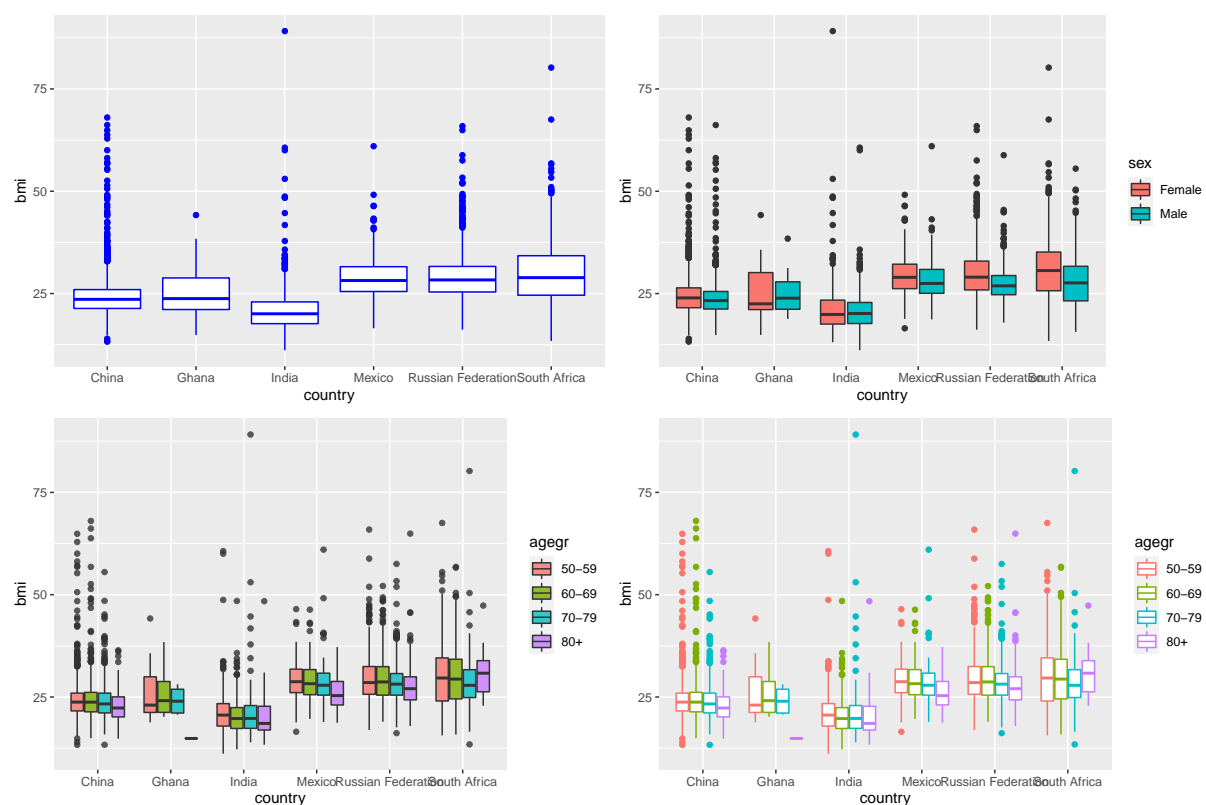


Figure 19.4: Boxplots; examples of aesthetic mappings on agegroup and sex

Interaction

We can also use interaction of two factors in ggplot and in aesthetics.

```
p<-ggplot(bp,aes(x=interaction(sex,residence),y=bmi))
p1<-p+geom_boxplot(colour="blue",size=0.5)
p2<-p+geom_boxplot(size=0.5,aes(fill=sex))
p3<-p+geom_boxplot(colour="blue",size=0.5,aes(fill=interaction(sex,residence)))

p<-ggplot(bp,aes(x=agegr,y=bmi))
p4<-p+geom_boxplot(colour="blue",size=0.5,aes(fill=interaction(sex,residence)))
```

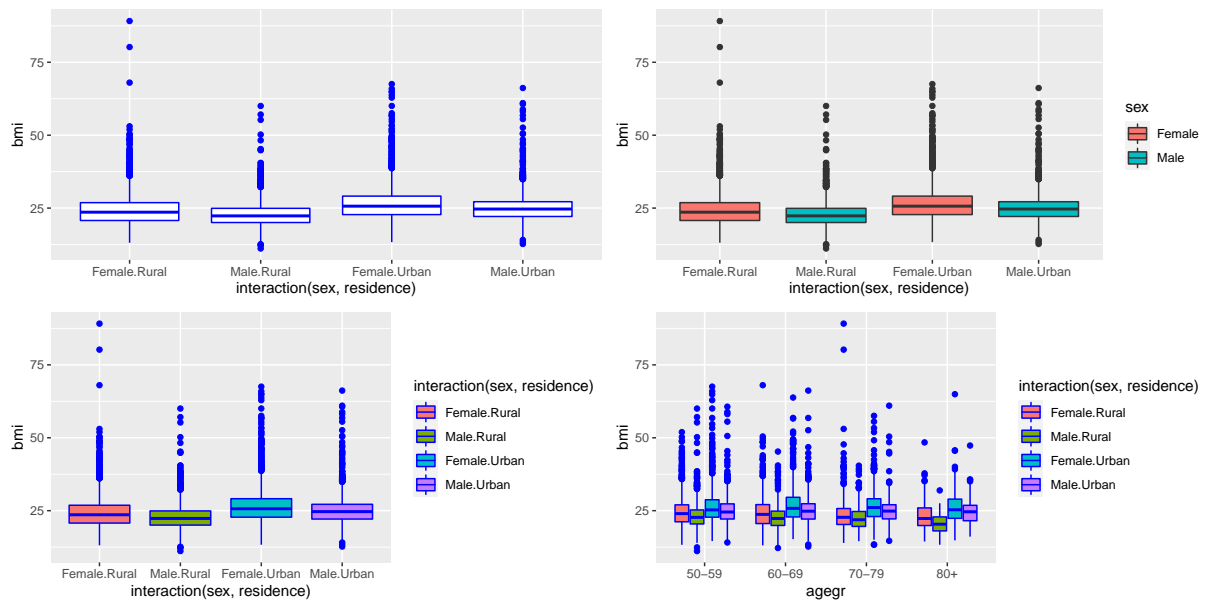


Figure 19.5: Boxplot; examples of using interaction and aes

Scatterplots

```
p<-ggplot(norsjo86,aes(y=sbp,x=bmi,shape=as_factor(sex)))
p1<-p+geom_point(size=1.5,aes(color=as_factor(sex)))
p2<-p+geom_point(aes(size=kolester/2))
p3<-p+geom_point(shape=21,colour="black",fill="red",aes(size=kolester/2,stroke=bmi/10))
# we can add regression lines
p4<-p1+geom_smooth(method=lm,se=F,aes(color=as_factor(sex)))
```

```
## 'geom_smooth()' using formula 'y ~ x'
```

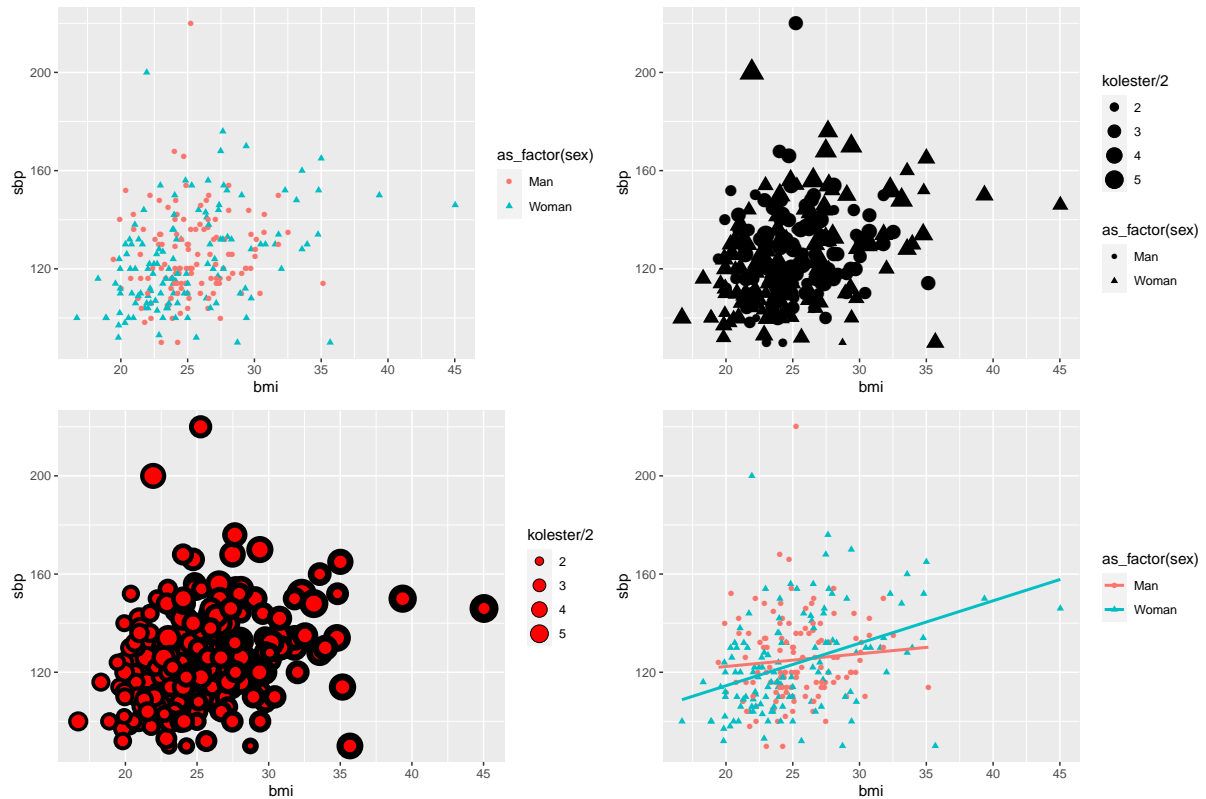


Figure 19.6: Scatterplots; examples of aesthetics

Trend plots

A trend plot is a plot where values on the x-axis is categorical and there is only one y-value per x-value.

```
norsjo86<-norsjo86 %>% mutate(sex=factor(sex,labels=c("men","women")),
                               age=as.double(agegrp))
nor.sum<-norsjo86 %>% group_by(age,sex) %>%
  summarise(mean.sbp=mean(sbp),min.sbp=min(sbp),max.sbp=max(sbp))
nor.sum
```

A tibble: 8 x 5
Groups: age [4]

| | age | sex | mean.sbp | min.sbp | max.sbp |
|---|-------|-------|----------|---------|---------|
| | <dbl> | <fct> | <dbl> | <dbl> | <dbl> |
| 1 | 30 | men | 119. | 90 | 148 |
| 2 | 30 | women | 110. | 97 | 142 |

| | | | | | |
|---|----|-------|------|-----|-----|
| 3 | 40 | men | 121. | 98 | 152 |
| 4 | 40 | women | 116. | 90 | 156 |
| 5 | 50 | men | 128. | 102 | 220 |
| 6 | 50 | women | 132. | 93 | 176 |
| 7 | 60 | men | 133. | 100 | 168 |
| 8 | 60 | women | 134. | 104 | 200 |

```
p1<-ggplot(nor.sum,aes(y=mean.sbp,x=age,shape=sex))+
  geom_line(aes(linetype=sex),size=1)+
  geom_point(size=2)+
  labs(y="Mean systolic blood pressure")
```

```
p2<-p1+geom_point(aes(y=max.sbp),color="black",size=2)+
  geom_line(aes(y=max.sbp,linetype=sex),size=1)+
  geom_point(aes(y=min.sbp),color="black",size=2)+
  geom_line(aes(y=min.sbp,linetype=sex),size=1)+
  labs(y="Systolic blood pressure",title="Mean, maximum and minimum systolic blood pressure by sex")
```

```
p3<-p1+geom_point(color="black")+
  geom_text(aes(y=min.sbp,color=sex),label="Min",size=3)+
  geom_label(aes(y=max.sbp,color=sex),label="Max",size=3)
# label can also be a vector with length=nrow(data.frame)
```

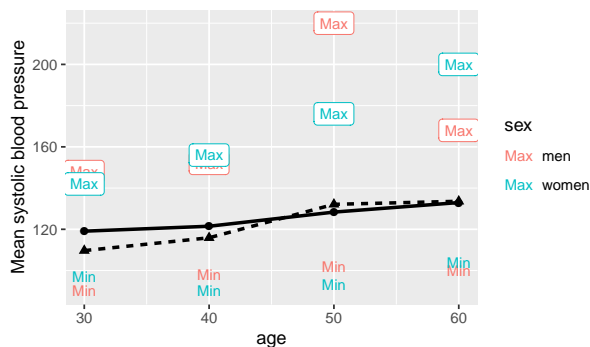
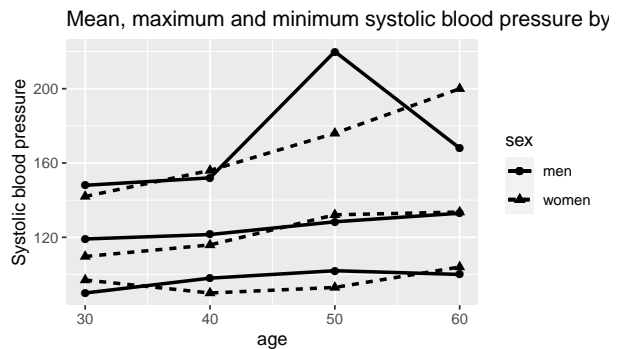
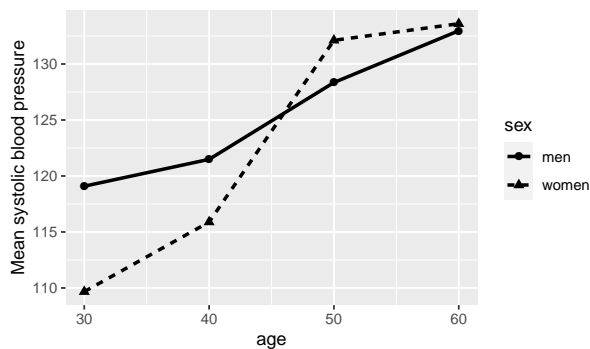


Figure 19.7: Trend plots including summary measures

19.3 Using facets

This facility makes it easy to split the plots by categorical data using the formula object.

```
p<-ggplot(norsjo86,aes(y=sbp,x=bmi))+
  geom_point(size=1.5)
```



```
p1<-p+facet_grid(.~as_factor(sex))  
p2<-p+facet_grid(as_factor(sex)~agegrp)
```

p1

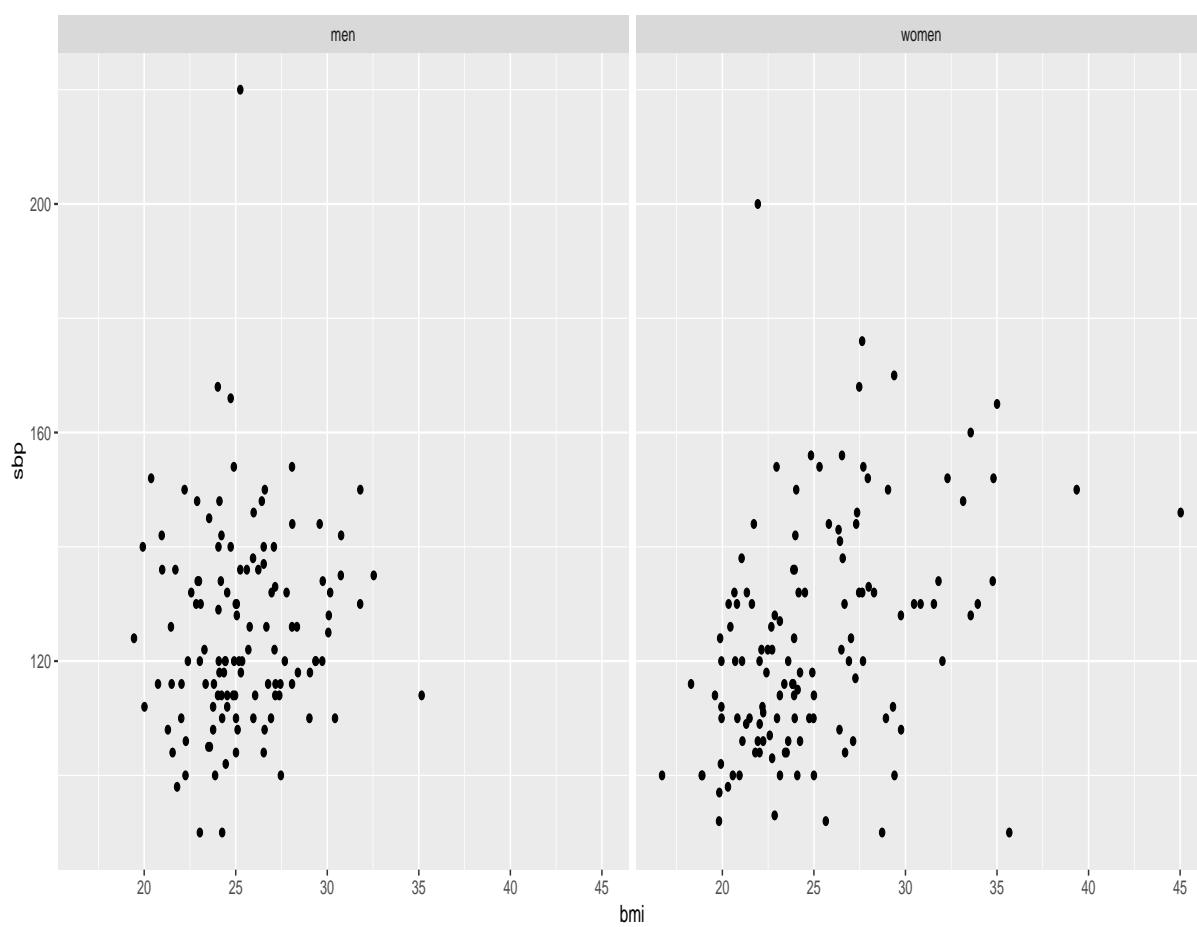


Figure 19.8: Scatterplots of sbp vs bmi split by sex

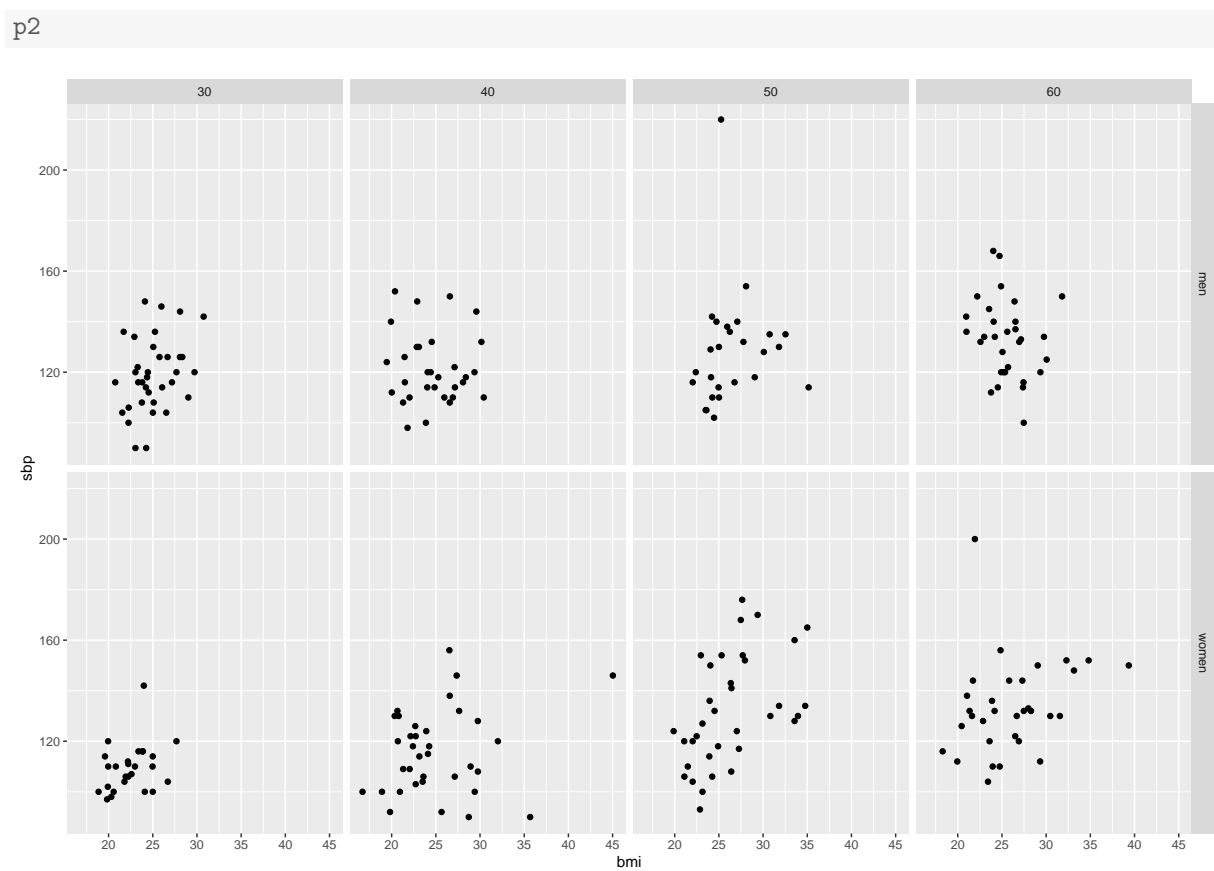


Figure 19.9: Scatterplots of sbp vs bmi split by sex and age

```
p<-ggplot(norsjo86,aes(y=sbp,x=bmi))+  
  geom_point(size=1.5)+  
  facet_grid(as_factor(sex)~agegrp)  
  
ps1<-p+geom_smooth(method=lm,se=T) # Add smoothing with linear regression  
ps2<-p+geom_smooth(se=F) # by default it is smoothed using loess  
ps3<-p+geom_smooth(se=F,span=0.4) # less stiff  
  
ps4<-p+geom_smooth(se=F,span=10,color="black") # more stiff
```

```
## 'geom_smooth()' using formula 'y ~ x'
## 'geom_smooth()' using method = 'loess' and formula 'y ~ x'
## 'geom_smooth()' using method = 'loess' and formula 'y ~ x'
## 'geom_smooth()' using method = 'loess' and formula 'y ~ x'
```

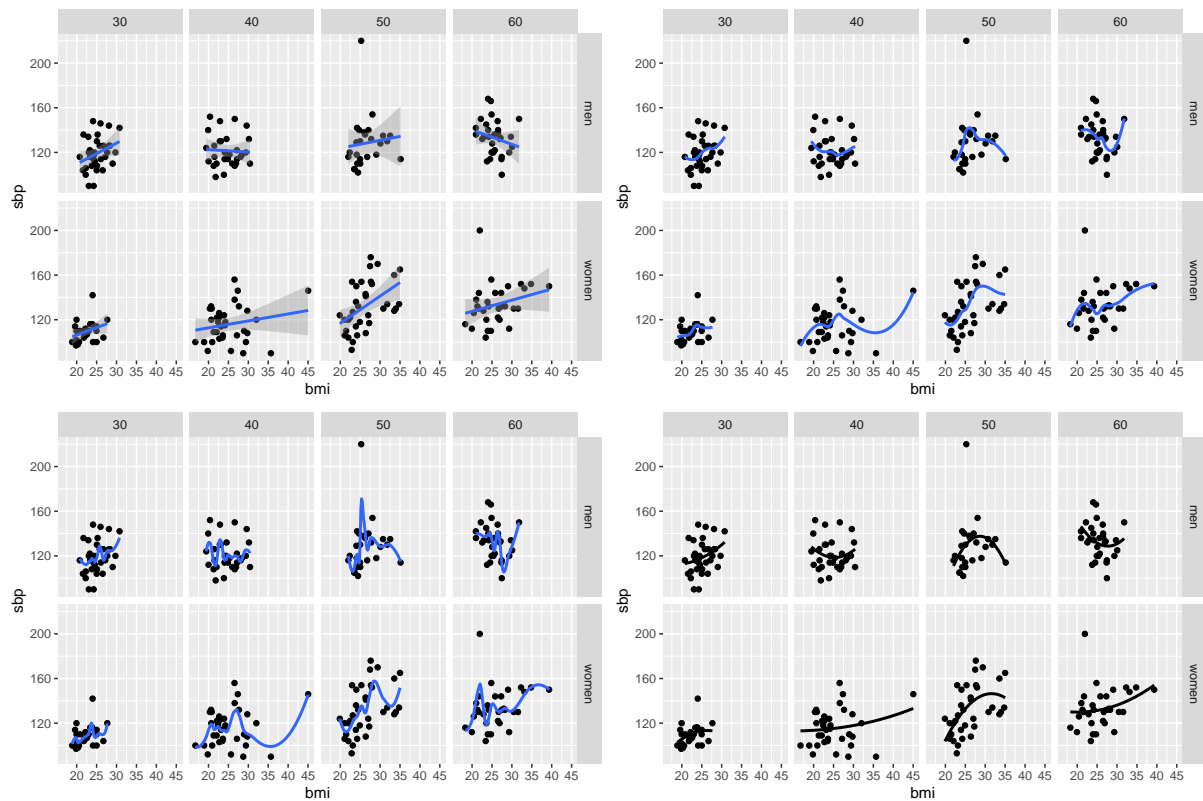


Figure 19.10: Scatterplots with facets and smoothing

```
p<-ggplot(norsjo86,aes(y=sbp,x=bmi))+  
  facet_grid(as_factor(sex)~agegrp)  
p1<-p+geom_point(size=2,shape=21,aes(fill=as_factor(smoker)))  
p2<-p+geom_point(size=2,aes(color=cut(kolester,c(0,6,100)),shape=as_factor(smoker)))
```

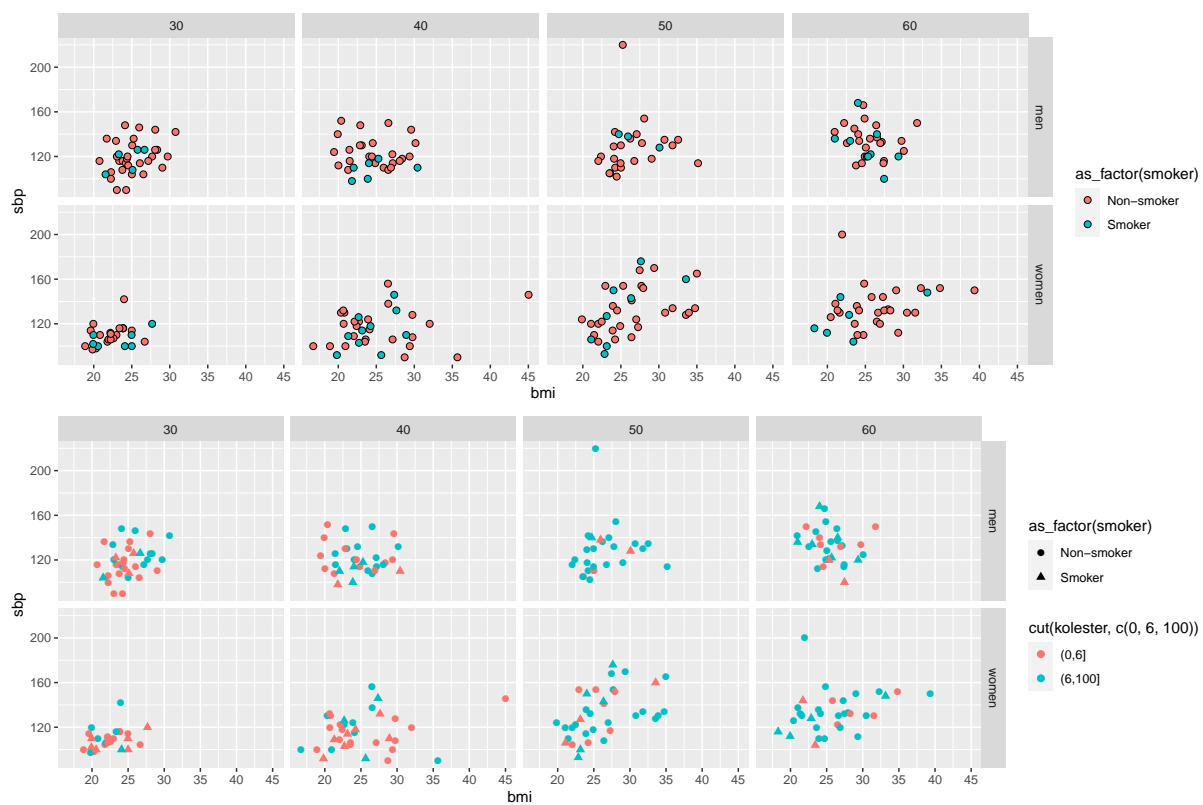


Figure 19.11: Scatterplots with facets and aes

19.3.1 Using facets and other grouping together with converting data into long format

We first use the `tidyr::gather` function to store the data into long format.

```
bp.lf<- bp %>% gather(key=var,value=value,sex,agegr) # now data is duplicated for sex and agegr
bp.lf %>% select(1:5,var,value) %>% head(5)

# A tibble: 5 x 7
  id residence    age wealthQ    physical    var    value
  <dbl> <chr>      <dbl> <chr>      <chr>      <chr> <chr>
1  679 Rural        56 Q2        high phys act sex    Male
2  8863 Rural        56 Q3        high phys act sex    Female
3  33223 Urban        58 Q4        high phys act sex    Male
4  30317 Urban        73 Q2        high phys act sex    Female
5  15269 Urban        53 Q5 richest mod phys act sex    Male

bp.lf %>% select(1:5,var,value) %>% tail(5)

# A tibble: 5 x 7
  id residence    age wealthQ    physical    var    value
  <dbl> <chr>      <dbl> <chr>      <chr>      <chr> <chr>
1  11966 Urban        65 Q5 richest high phys act agegr 60-69
2   8301 Rural        53 Q5 richest high phys act agegr 50-59
3  26439 Urban        70 Q1 poorest mod phys act agegr 70-79
4  13910 Rural        54 Q2        high phys act agegr 50-59
5  14725 Urban        51 Q3        high phys act agegr 50-59

table(bp.lf$var)

agegr    sex
10000 10000

p1<-ggplot(bp.lf,aes(y=disability,x=value,fill=var))+
geom_boxplot(colour="blue",size=0.5) # disability is plotted vs both sex and agegr

p2<-p1+facet_grid(residence~.)          # split by residence

p3<-p1+facet_grid(residence~country) # also split by country

# this time we gather continuous variables
bp.lf<- bp %>% gather(key=var,value=value,bmi,height)

p1<-ggplot(bp.lf,aes(x=waistc,y=value,fill=var))+
geom_point(size=1.3,shape=21)

p2<-p1+facet_grid(residence~physical)+
  geom_smooth(method=lm,se=F,color="black")
```

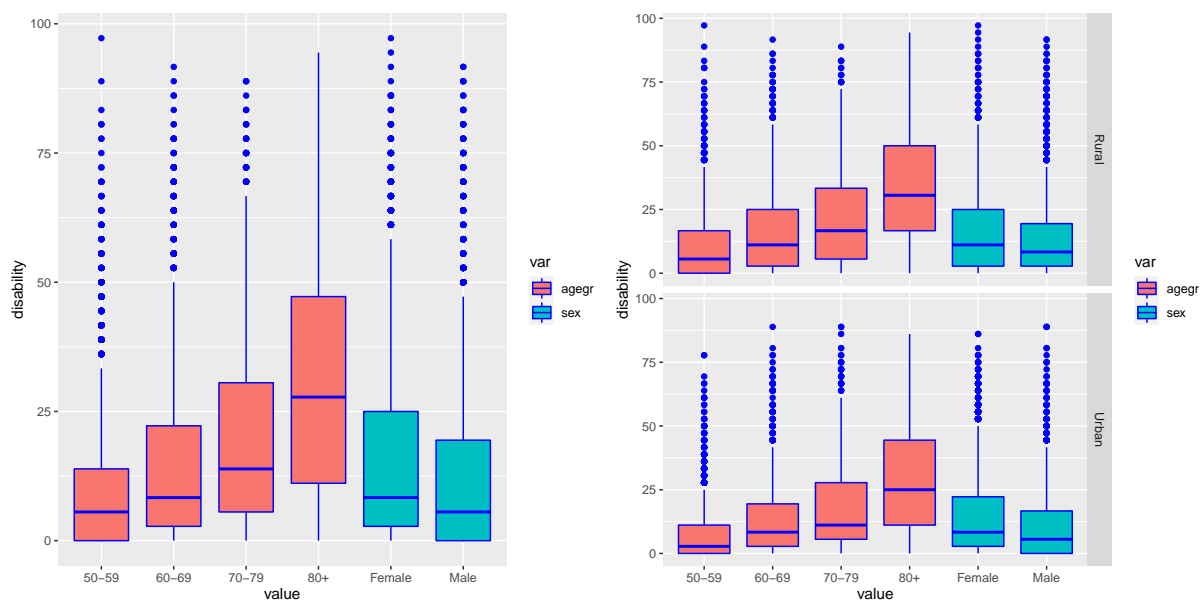



Figure 19.12: Boxplots using long format (two variables in one)

p3

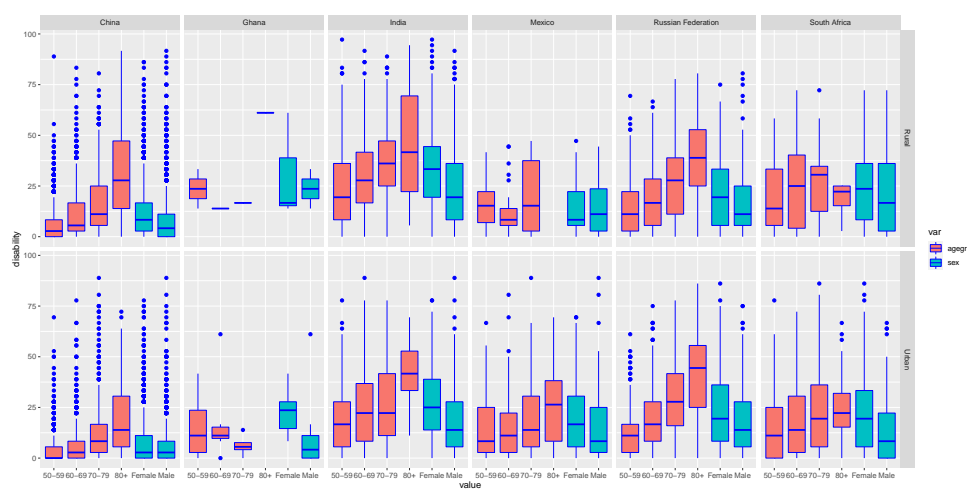


Figure 19.13: Boxplots using long format and facets

p1



Figure 19.14: Plot of two continuous variables in one using gather

p2

```
## 'geom_smooth()' using formula 'y ~ x'
```

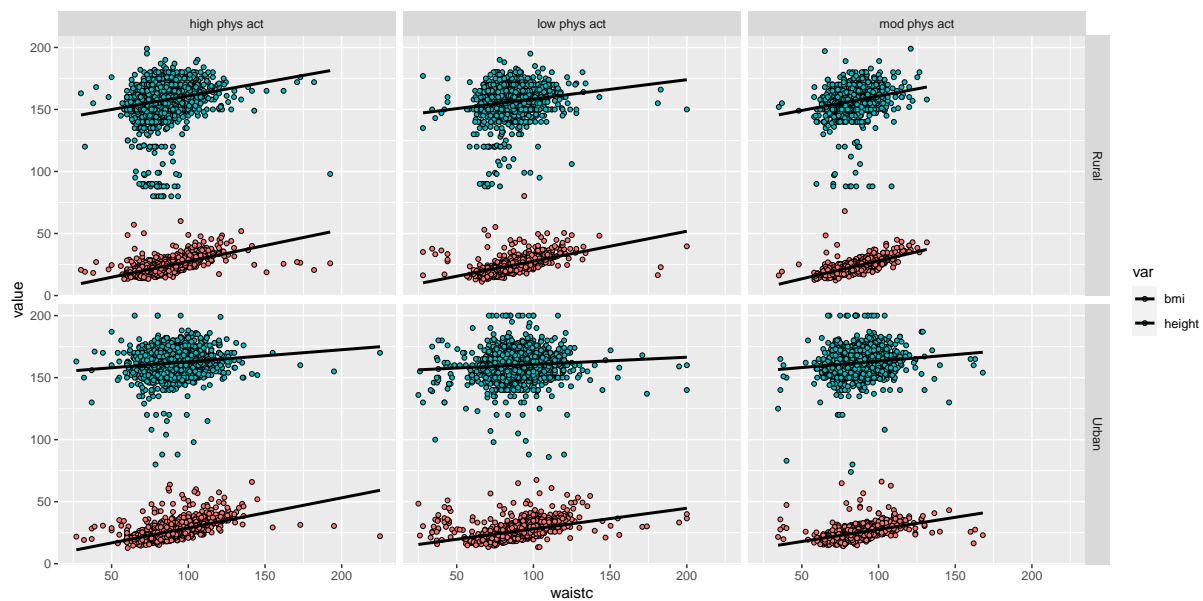


Figure 19.15: Plot of two continuous variables in one using gather and facets

Chapter 20

Matrix calculations

Håkan Jonsson < hakan.jonsson@umu.se >

20.1 Matrix definition

A matrix is a two-dimension collections of elements of the same type, very useful in mathmatics. Often a matrix is thought of as a number of vectors of the same length. A matrix is like a data frame but without column names.

```
m<-matrix(1:10,nrow=4)
```

```
m
```

| | [,1] | [,2] | [,3] |
|------|------|------|------|
| [1,] | 1 | 5 | 9 |
| [2,] | 2 | 6 | 10 |
| [3,] | 3 | 7 | 1 |
| [4,] | 4 | 8 | 2 |

```
class(m)
```

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

```
as.data.frame(m)
```

| | V1 | V2 | V3 |
|---|----|----|----|
| 1 | 1 | 5 | 9 |
| 2 | 2 | 6 | 10 |
| 3 | 3 | 7 | 1 |
| 4 | 4 | 8 | 2 |

Vi can observe

- nrow=4 decides the numer of rows
- the data 1:10 are filled column by column
- the number of coumns is decided by the data (input numbers)
- if the length of the input numbers is not a factor of the number of rows the last column is filled starting from the beginning

```
# we can define a matrix of characters but it is not so useful for calculations
matrix(letters[1:8],ncol=2)

      [,1] [,2]
[1,] "a"  "e"
[2,] "b"  "f"
[3,] "c"  "g"
[4,] "d"  "h"

m1<-matrix(1:12,ncol=4)
m1

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

nrow(m1)

[1] 4

ncol(m1)

[1] 3

dim(m1)

[1] 4 3

m2<-m1[2:4,] # row 2-4 of m2
# specifying rows and columns work like as for a data frame
m2

      [,1] [,2] [,3]
[1,]     2     6    10
[2,]     3     7    11
[3,]     4     8    12

dim(m2)

[1] 3 3
```

Own experimentation

Create matrixes of varying dimensions. Instead of `nrow` you can use `ncol`. Then the dimension of the matrix is decided by number of columns. It is possible to change the order of filling a matrix using the argument `byrow = TRUE`. Can you create a matrix of logics (TRUE/FALSE)?

20.2 Calculations

Matrixes can be transposed i.e. rows and columns are switched. Two matrixes can be added or subtracted if they have the same dimension. The product of two matrixes A and B with dimensions $(R_A \times C_A)$ and $(R_B \times C_B)$ written AB is only possible if $C_A = R_B$. The resulting product written AB has dimension $(R_A \times C_B)$. OBS that BA has dimension $(R_B \times C_A)$. It is therefore not obvious that $AB=BA$ or that BA is even defined. Observe the operator for multiplication. The (i, j) th element in the

resulting matrix product AB is the inner product of the vectors from row i in A and column j in B .

```
m1<-matrix(1:12,nrow=4)
m1

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

dim(m1)

[1] 4 3

t(m1) # this is the transpose of the matrix

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

dim(t(m1))

[1] 3 4

m2<-matrix(11:25,nrow=3)
m2

      [,1] [,2] [,3] [,4] [,5]
[1,]   11   14   17   20   23
[2,]   12   15   18   21   24
[3,]   13   16   19   22   25

dim(m2)

[1] 3 5

m1+m1 # the numer of rows and column must match

      [,1] [,2] [,3]
[1,]    2   10   18
[2,]    4   12   20
[3,]    6   14   22
[4,]    8   16   24

m1+m2 # this does not work

Error in m1 + m2: non-conformable arrays

3*m2

      [,1] [,2] [,3] [,4] [,5]
[1,]   33   42   51   60   69
[2,]   36   45   54   63   72
[3,]   39   48   57   66   75

m1%*%m2 # the product of two matrixes

      [,1] [,2] [,3] [,4] [,5]
[1,]  188  233  278  323  368
[2,]  224  278  332  386  440
[3,]  260  323  386  449  512
[4,]  296  368  440  512  584
```

```

m1[3,]

[1] 3 7 11

m2[,2]

[1] 14 15 16

sum(m1[3,]*m2[,2]) # the element in row 3 column 2 of the product m1%%m2

[1] 323

(m1%%m2)[3,2]

[1] 323

m2%%m1 # this does not work because the number of columns

Error in m2 %% m1: non-conformable arguments

#           in the first and the number of rows in the second does not match

m1%%m1 # only possible if the number of rows=number of columns

Error in m1 %% m1: non-conformable arguments

```

20.3 Equation system and matrix inverse

Assume we want to solve the equations $\begin{cases} x + 3y = 4 \\ 2x - y = 3 \end{cases}$

This can be written in matrix form as $AX = b$ where X is the vector (x, y) , b the vector $(4, 3)$ and

$$A = \begin{pmatrix} 1 & 3 \\ 2 & -1 \end{pmatrix}$$

The equation system can be solved by taking the inverse of A and the solution is the result of $A^{-1}b$

```

A<-matrix(c(1,2,3,-1),nrow=2)
A

      [,1] [,2]
[1,]    1    3
[2,]    2   -1

b<-matrix(c(4,3),nrow=2)
b

      [,1]
[1,]    4
[2,]    3

Am1<-solve(A)
Am1

      [,1] [,2]
[1,] 0.1428571 0.4285714
[2,] 0.2857143 -0.1428571

Am1%%b # this is the straightforward solution

```

```

      [,1]
[1,] 1.8571429
[2,] 0.7142857

X<-solve(A,b) # this is the recommended solution
X

      [,1]
[1,] 1.8571429
[2,] 0.7142857

A%*%X # thus the resulting x is the solution to the equation system

      [,1]
[1,] 4
[2,] 3

round(Am1%*%A,10) # The identity matrix

      [,1] [,2]
[1,] 1 0
[2,] 0 1

diag(2) # it can easily be generated

      [,1] [,2]
[1,] 1 0
[2,] 0 1

diag(10)

      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,] 1 0 0 0 0 0 0 0 0 0
[2,] 0 1 0 0 0 0 0 0 0 0
[3,] 0 0 1 0 0 0 0 0 0 0
[4,] 0 0 0 1 0 0 0 0 0 0
[5,] 0 0 0 0 1 0 0 0 0 0
[6,] 0 0 0 0 0 1 0 0 0 0
[7,] 0 0 0 0 0 0 1 0 0 0
[8,] 0 0 0 0 0 0 0 1 0 0
[9,] 0 0 0 0 0 0 0 0 1 0
[10,] 0 0 0 0 0 0 0 0 0 1

```

20.4 Arrays

An array is a generalisation of a matrix. A vector can be seen as a matrix with only one row or column. The dimension of a matrix is ($R \times C$). An array is a generalisation of a matrix to more than two dimensions. As for a matrix the elements have to be of the same type, almost always numeric.

```

x<-array(1,dim=c(3,3))
x

      [,1] [,2] [,3]
[1,] 1 1 1
[2,] 1 1 1
[3,] 1 1 1

```

```

class(x)

[1] "matrix"

y<-array(1:18,dim=c(3,3,2))
y

, , 1

      [,1] [,2] [,3]
[1,]     1     4     7
[2,]     2     5     8
[3,]     3     6     9

, , 2

      [,1] [,2] [,3]
[1,]    10    13    16
[2,]    11    14    17
[3,]    12    15    18

class(y)

[1] "array"

dim(x)

[1] 3 3

dim(y)

[1] 3 3 2

y[, , 1]

      [,1] [,2] [,3]
[1,]     1     4     7
[2,]     2     5     8
[3,]     3     6     9

y[, , 2]

      [,1] [,2] [,3]
[1,]    10    13    16
[2,]    11    14    17
[3,]    12    15    18

y[1, , 2]  # the first dimension=1 (first row) and third dimension=2

[1] 10 13 16

is.vector(y[1, , 2])

[1] TRUE

```

Let us import the data file norsjo86.

| | | |
|-----------|--------------------------|--------------------------------|
| agegrp: | Age group | (30, 40, 50 ,60 years) |
| health: | Health status | (0=good, 1=not quite good/bad) |
| sex: | Sex | (1=man, 2=woman) |
| height: | Body height | (cm) |
| weight: | Body weight | (kg) |
| sbp: | Systolic blood pressure | |
| dbp: | Diastolic blood pressure | |
| kolester: | Cholesterol | |
| smoker: | Smoking status | (0=non-smoker, 1=smoker) |
| bmi: | Body mass index | (kg/m^2) |

```
# import spss file norsjo86
library(haven)
norsjo86 <- read_sav("../data/norsjo86.sav")
head(norsjo86,3)

# A tibble: 3 x 10
  agegrp health sex height weight sbp dbp kolester smoker bmi
  <dbl+lbl> <dbl+lbl> <dbl+1> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl+lbl> <dbl>
1 60 [60 years~ 0 [good] 2 [Wom~ 157 61 110 70 6.7 0 [Non-sm~ 24.7
2 60 [60 years~ 1 [not quite ~ 2 [Wom~ 157 97 150 100 6.6 0 [Non-sm~ 39.4
3 60 [60 years~ 0 [good] 1 [Man] 170 74 136 96 8.2 0 [Non-sm~ 25.6

xtabs(~sex+agegrp,data=norsjo86)

  agegrp
sex 30 40 50 60
  1 36 31 27 34
  2 27 37 35 33

tab<-xtabs(~sex+agegrp+cut(sbp,c(0,120,Inf)),data=norsjo86)
tab

, , cut(sbp, c(0, 120, Inf)) = (0,120]

  agegrp
sex 30 40 50 60
  1 23 19 13 9
  2 26 22 12 8

, , cut(sbp, c(0, 120, Inf)) = (120,Inf]

  agegrp
sex 30 40 50 60
  1 13 12 14 24
  2 1 14 23 24

is.array(tab)
[1] TRUE

tab[,2,] # gives the table for 40 year olds

  cut(sbp, c(0, 120, Inf))
sex (0,120] (120,Inf]
  1 19 12
  2 22 14

tab[, "40",] # alternative

  cut(sbp, c(0, 120, Inf))
sex (0,120] (120,Inf]
  1 19 12
  2 22 14
```