R for Medical Research

S. Willemsen, E. Andrinopoulou

2019-02-27

Contents

Preface

This reader contains some material on the R language for the NIHES course BST02 on R that can be used together with the course slides.

Chapter 1

Introduction

1.1 Why R

Currently R is the statistical package that of choice for statisticians. This popularity is due to the fact that it is focused on data analysis but it is also a complete programming language. This in contrast to on the one hand SPSS and SAS in which programming is very difficult to say the least and Python on the other hand which is a general programming language which is popular for data science but which not created for this purpose from the onset.

One characteristic of R that contributed to its popularity to a large extend is no doubt that the program is completely free of charge. Also its source code is publicly available and in the public domain. So, in principle, anybody can make changes to the language and suggest that these changes are also made to the public distribution. This also means that there is no publisher to whom to address your questions. Fortunately there are enough other sources from which you can get support.

The R language is modular: base R only has limited functionality, but this functionality can be extended by means of various packages. Nowadays there is an enormous amount of extension packages. And the number of packages still seems to be growing at an exponential rate. The quality of these packages varies although there is some kind of quality control on the packages distributed through the official channel CRAN.

1.2 Installation R and R studio

Before we can work with R the program must be installed on the computer. It is recommended to also install a second program called RStudio. This is a separate program (an Integrated Development Environment (IDE)) that makes working with R much more user friendly. There are other IDEs available but we will only discuss RStudio here.

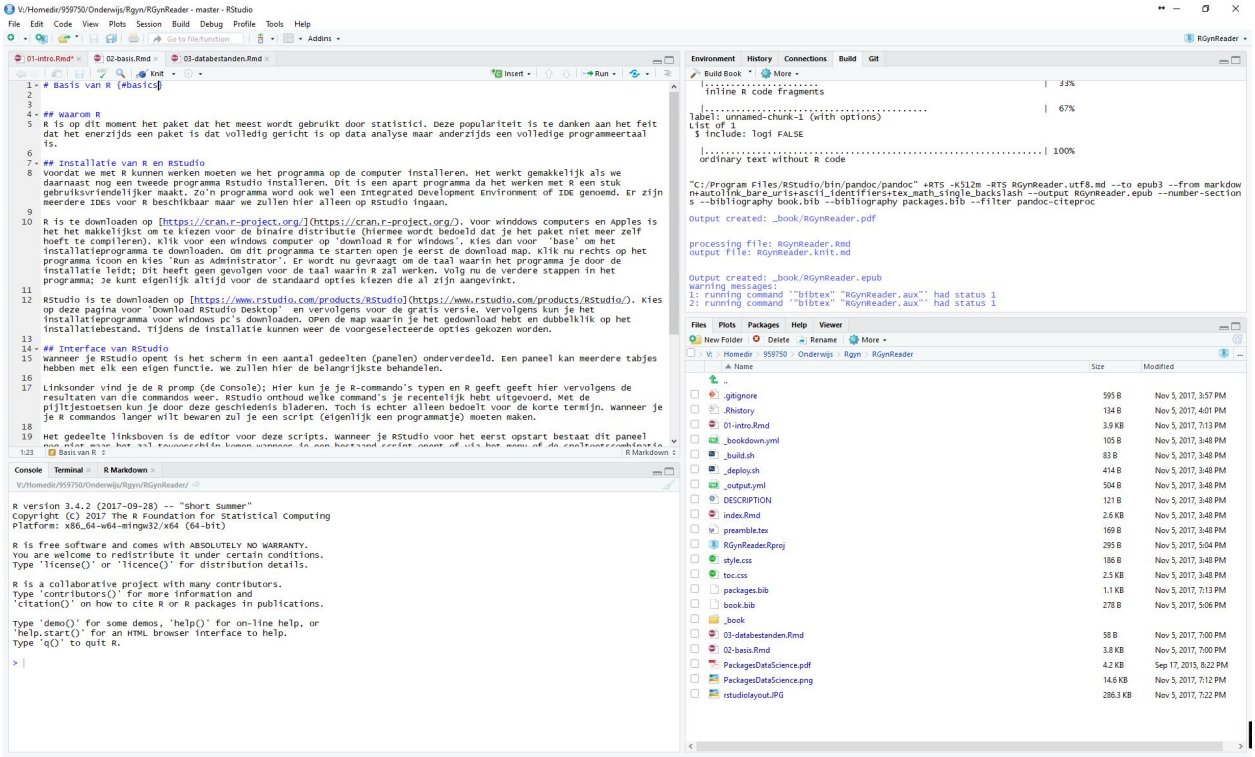
R can be downloaded on https://cran.r-project.org/. For windows and Apple computer it is best to choose the so called binary distribution (which means that we do not have to build the program from its source files ourselves). For a windows PC click on 'download R for Windows'; The choose 'base' to download the installer. To start it simply open the download folder and double click the icon. Now you are requested to select the language in which you are guided through the various installation steps; This does not have any consequences for when we work with R itself. Now proceed with installation, usually the default options in the installer will suffice. When you use Linux the way in which you install R depends on your particular distribution but R is can be installed using the package manager of all popular flavors of Linux.

RStudio can be downloaded at https://www.rstudio.com/products/RStudio. On this page choose 'Download RStudio Desktop' and then choose the free version. Now you can download the installer. On a windows pc

navigate to the downloads folder again and double click the installer's icon to start the installation process. During the installation you can accept the preselected options.

1.3 Interface of RStudio

When RStudio is opened the program window is divided in several parts (panes). A single pane can have several tabs which each a separate function. We will discuss the most important ones.



At the bottom left you can find the R prompt (the Console); Here you can enter your commands and R will return its output. RStudio remembers which commands were recently executed and with the arrow keys you can move back and forth through history. When you want to store your commands for a longer time this method will break down. What you need to do instead is to make an R script (i.e. a program).

The top left pane shows the editor for these scripts. When RStudio is opened for the first time this panel dos not yet exist but it will pop-up whenever you open an existing script or when you press Ctrl-Shift-N to start working on a new script. A script is a sequence of commands, for example to carry out a statistical analysis. You can save a script by choosing 'save' or 'save as' from the 'file' menu. By working with scripts you can always see which commands were executed, and hence it is clear on what the conclusions of your analysis are based. Working with scripts is an important part of reproducible research. An R-script usually has the extension .R. It is in essence a regular plain text file and this it can be opened in any text editor.

You can execute the script by pressing the keys Ctrl+Alt+R simultaneously. It is also possible to execute parts of a script by selecting them and pressing Ctrl-Enter. When pressing this key combination without selecting a part of the script but having the active cursor in a script window the line on which the cursor is located will be executed. It is possible to open several scripts at the same time. These will then be displayed in several tabs.

At the bottom right panel there is a tab called 'Files' with all files in the working directory and a tab called 'Plots' in which the plots we create will be displayed. In the upper right panel there is a tab called 'History' in which we can see all commands that recently have been executed. Here you will also find a tab called

'Environment' where all objects we created are listed.

Chapter 2

R Basics

2.1 R as a calculator

One of the most simple things we can do with R is to use it as a kind of calculator. We can do all elementary arithmetic operations:

```
>1 + 1
[1] 2
>1 - 1
[1] 0
>2 * 3
[1] 6
>2 / 3
[1] 0.6666667
```

Here we use '>' to denote the R prompt. It is used in the console to indicate that R is ready for a new command. The lines in between is the output given by R.

Most of the functionality in R comes from its functions. There are many functions we can use including the standard mathematical ones: >cos(3.1415) [1] -1 >exp(0) [1] 1 >log(1) [1] 0

We will discuss functions later on in this text in much more detail.

2.2 Variables

The results above are displayed but not stored. To do this we must give a name to the result. That is, we store the result in a variable. The results are now no longer displayed but using the variable name we can refer to it later.

Assigning a result in a variable is done using `<-`. For example:

```
>five <- 3+2
> five
[1] 5
> five + 1
[1] 6
```

Each variable has a class that determines for a large extend what we can do with it.

2.3 R packages and the library command

As mentioned above R is a modular language that we can extend using packages. We can load a package using the command `library()`. So if we want to load the `survival` package (in order to do survival analysis) we type:

```
>library(survival)
```

Before we can load a library it must be installed. This can be done using the command `install.packages(packagename)` where `packagename` is the name of the package between single or double quotes. For example:

```
>install.packages('gamlss')
```

Some useful packages are `lme4` for analyses with repeated measurements and `mice` for multiple imputation.

2.4 Documentation

The functions `help` and `?` can be used to read the documentation on a particular subject (mostly functions). The package in which the function is defined must be loaded unless you use the `package` parameter ofn the help function. For example: `help('survreg' , package='survival')`.

Using `help.search` and `??` we can search the documentation files in a more general way. The documentation for all installed packages is searched for topics that have specific words in the title.

```
>?library
>help(install.packages)
>help.search('logistic regression')
```

Some R packages are also documented with so called vignettes. You can browse the vignettes of a package using the function `browseVignettes`, for example `browseVignettes(package="survival")`.

Using the function `RSiteSearch` you can search the documentation of all packages that are on CRAN (so it is like a google for R).

Another great source of help is the site 'Stack Overflow'. A site were people can ask all kinds of programming related questions.

Chapter 3

Data structures

3.1 Importing data files

If you want to learn to use R that is probably because you want to use it to analyse some data. So let's illustrate R based on a data set. We can download and import an SPSS data set containing data on an (imaginary) trial using:

```
library(foreign)
mydata <- read.spss(file='https://github.com/stenw/bst02r/blob/master/data-raw/pain.sav?raw=true',
                    to.data.frame = TRUE)
```

3.2 data.frames

When we have executed the command above the data set will be stored under the name `mydata`. `mydata` is a so called `data.frame`. This is a rectangular table in which every column contains a variable and every row an observation. The variable `mydata` will also appear in the environment tab in the upper right hand corner of RStudio. Right next to the name we can see how many observations and how many variables the `data.frame` has. We can examine the data set more closely by clicking the little icon next to it in this tab. An other way of looking at the data is my typing in the command `View(mydata)` directly in the Console tab.

We see that the first four variables are the patient number, the treatment, the gender and age of the patient(`ptno`, `treatm`, `sex` and `age`). In order to work with one of the variables in the `data.frame` we first type the name of the `data.frame` followed by a dollar sign, and finally the name of the variable inside the `data.frame`. So when we execute the command: `mydata$age` in the console the ages of all patients are displayed. We can calculate the average of them using the function `mean`.

```
mean(mydata$age)
```

```
## [1] 53.52
```

3.3 vectors and datatypes

Each variable in the `data.frame mydata` is a so called vector. A vector is a number of values that each have the same data type (Or mode[1]). The most important data types in R are:[2]:

mode	description
character:	text, for example 'man', 'woman', 'censored', etc.
logical:	TRUE and FALSE
integer: numeric:	whole numers like 0, 1, 2, etc … Posibly fractional numerical values like 1.0, 1.2 or 1e12 (that is 10 raised to the power of 12)

Vectors of these data types are the most elementary data structure in R. All other structures (like the `data.table`) are constructed using these vectors. In R there is also no structure that is smaller than a vector. A single number is not treated differently from a numeric vector of length ten; In fact R sees the single number simply as a numeric vector of length 1. The `length()` of a vector can be obtained by using the function `length()`.

A vector can be created using the function `c()`. (The `c` stands for 'concatenate', 'coerce' or 'combine')

```
c(1, 2, 3)
```

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

```
c('spam', 'ham', 'eggs')
```

```
## [1] "spam" "ham"  "eggs"
```

```
c("double", "quotes", "work",
  'like', 'single')
```

```
## [1] "double" "quotes" "work"   "like"   "single"
```

```
c(TRUE, FALSE)
```

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

In the output we see that R shows the row number of the first element of each row between straight brackets. This makes it easier to refer to a particular element, especially when the vectors are long. We can work with vectors in the same way as with single numbers. In principle all operations are carried out in an element wise fashion.

```
c(1, 2, 3) * c(4, 5, 6)
```

```
## [1]  4 10 18
```

When we try to create a vector that consists of different data types they will be converted to a data type that is capable of containing all of them. For example:

```
c("eleven", 12)
```

```
## [1] "eleven" "12"
```

The second element of the resulting vector is now also of type `character`. In general it is better not to trust this implicit conversion. Instead to it explicitly, in this case by using the function `as.character()`.

An other way to create a vector is by using the function `vector`. `vector('numeric', 8)` creates a numeric vector of length 8. The `vector` function is often used to preallocate room where the results of future computations can be stored.

[1] `mode`, `storage.mode` and `type` are closely related concepts, we will not discuss the differences here. See also `?mode`.
[2] There are a few more like `complex` and `raw` which we will not discuss.

3.4 lists

Elements of a vector are always of the same type. A `list` differs from a vector by also allowing its elements to be of a different type. We can make a `list` using the function `list`.

```
list1 <- list("eleven", 12)
list2 <- list(c(1, 2, 3), c('foo', 'bar'))
```

We can also assign a name to the elements of a list:

```
list3 <- list(numbers=c(1, 2, 3), chars=c('foo', 'bar'))
```

It is also possible for a list to contain other lists.

```
list4 <- list(numbers=c(1, 2, 3), chars=c('aap', 'noot'),
              sublist=list(1,'a'))
```

In many ways lists resemble data.frames. An important difference is that the length of all elements of a data.frame (that is the variables) must be the same. For a list this is not the case.

3.5 factors

A `factor`is a special kind of vector for categorical data. The factor contains different integers one for every category. Each unique value has an associated 'label' that tell us what the code means. Factors are frequently used when we model categorical data. An advantage of a factor over a `character` is that we can limit the number of possible outcomes. It is also less likely to make mistakes due to typing errors. Factors can be created by means of the function `factor()`.

```
factor(c('male', 'female', 'male'))
```

```
## [1] male   female male
## Levels: female male
```

When a `factor` is displayed R also shows us the unique values the variable can take. These are called the 'levels' of the factor.

3.6 Missing values

Whenever the value of a variable is missing this is denoted by `NA` in R. Usually this means that the values exists however we do not know it. Sometimes the result of a calculation is not finite (for example when we define a positive or negative number by zero). In this case the result is defined to be `Inf` of `-Inf` in R. When a value cannot be computed at all (for example when we divide zero by zero) R will define the result as `NaN`, which stands for 'Not a Number'. Finally, R sometimes uses the special value `NULL` to indicate that a variable is not yet defined. Here we will mostly deal with data that is just missing, that is `NA`.

Chapter 4

Subsetting

Often we want to calculate some things only for a specific subgroup of our patient group. (For example we want to calculate the average for the variables age and weight, but only for the women in the data set and not for the men). So it is important to select specific variables and observations. In R making these selections is called indexing. We start by showing how we can make selections within a single variable (a vector). Afterwards we will see how selecting variables and observations in a `data.frame` works.

4.1 Indexing a vector

As an example we will work with age variable in the `data.frame mydata` ; So let's create a variable that contains this single variable.

```
ages <- mydata$age
```

4.1.1 Making selections based on the position in a vector

The easiest way of selecting elements in a vector is by using the position of the elements in the vector. This can be done by providing the positions of the elements we want to select between square brackets after the name of the vector. So when we want to select the age of the first patient we can run the following code:

```
ages[1]
#> [1] 58
```

We can also do this for multiple elements at the same time.

```
selection <- ages[c(2, 3)]
selection
#> [1] 68 61
```

The vector of indices does not have to be sorted and indices may be duplicated. For example:

```
selection[c(2, 1, 1)]
#> [1] 61 68 68
```

When we use a vector with negative integers we will select all observations *except* those on the specified positions. So if we want to select all the ages except for the first three we can use.

```
ages[-c(1, 2, 3)]
#>  [1] 37 80 51 39 61 58 42 61 66 66 79 72 53 70 59 60 32 48 50 52 41 61 51
```

```
#> [24] 68 38 61 55 45 56 33 50 43 70 59 51 78 64 64 53 55 37 51 73 42 66 35
#> [47] 51 26 58 71 57 73 63 56 45 55 64 26 51 23 55 53 29 69 54 34 59 70 77
#> [70] 74 43 58 40 45 64 46 65 35 40 67 52 46 56 35 40 54 36 56 45 55 52 68
#> [93] 52 35 40 47 64
```

It is not possible to combine positive and negative indices this way.

4.1.2 Selections based on a condition (TRUE / FALSE)

The second way to select elements from a vector is by using a vector of logical values (i.e.. TRUE/FALSE values) between the square brackets. In this way we select all values for which the value between the brackets is TRUE.

```
some_values<- c('foo', 'bar', 'baz')
some_values[c(TRUE, FALSE, TRUE)]
#> [1] "foo" "baz"
```

This way of selecting is especially useful when we use some comparison or condition to select variables.

```
ages[mydata$sex=='Female']
#>  [1] 68 37 79 60 48 50 52 41 68 61 56 33 43 70 51 35 58 56 45 26 51 23 77
#> [24] 45 65 67 46 56 52 35
mydata$ptno[ages>65]
#>  [1]  2  5 12 13 14 15 17 27 36 39 46 48 53 55 67 71 72 73 83 95
```

Often the conditions are based on the comparison operators: | Operator | Meaning | |———-|———————————| | == | Equal to | | != | Not equal to | | > | Greater than | | < | Smaller than | | >= | Greater than or equal to | | <= | Smaller than or equal to |

Conditions can be combined using `&` (and) and `|` or. Finally there is the logical negation operator `!` ()

4.1.3 Making selections using names

When the elements in a vector all have a name we can use these names to select the elements.

```
bp_with_name <- c(sys=135, dia=85)
bp_with_name['dia']
#> dia
#>  85
```

4.2 Making selections in a list

To make selections in a list, we can use single square brackets, double square brackets and the dollar sign. The use of single square brackets works in the same way as it does for vectors.

```
mylist <- list(
  foo=c(1,2, 3),
  bar=c('a', 'b'),
  baz=list(TRUE, c(2, 4))
  )
mylist[c(1,3)]
#> $foo
#> [1] 1 2 3
```

```
#>
#> $baz
#> $baz[[1]]
#> [1] TRUE
#>
#> $baz[[2]]
#> [1] 2 4
mylist[c(1,2,3)==2]
#> $bar
#> [1] "a" "b"
mylist['bar']
#> $bar
#> [1] "a" "b"
```

We can also use double square brackets and the dollar sign for a `list`. There are two important differences between using single and double square brackets: 1. Using double square brackets only allows us to select a single element from the `list`. 2. The result of a selection with double brackets is the element itself, while if we make the selection with single brackets the result is a `list` consisting of the selected elements.

```
mylist[[1]]
#> [1] 1 2 3
```

```
mode(mylist[[1]]) # a vector
#> [1] "numeric"
mode(mylist[1])   # a list with with a numeric vector as its single element
#> [1] "list"
```

There is another way to select a variable from a `list` which is by using the dollar sign ('$'). This is an alternative to using double square brackets in combination with a name. When we use this the result is always a vector.

```
mylist$foo      # results is a vector
#> [1] 1 2 3
```

Instead of using the dollar sign we can use double square brackets. We now need to put the name between quotes like for single square brackets. We can also use the position of the variable using these double square brackets.

```
mydata[['treatm']]
#>   [1] Placebo Placebo Placebo Placebo Placebo Placebo Placebo Placebo
#>   [9] Placebo Placebo Placebo Placebo Placebo Placebo Placebo Placebo
#>  [17] Placebo Placebo Placebo Placebo Placebo Placebo Placebo Placebo
#>  [25] Placebo Placebo Placebo Placebo Placebo Placebo Placebo Placebo
#>  [33] Placebo Placebo Placebo Placebo Placebo Placebo Placebo Placebo
#>  [41] Placebo Placebo Placebo Placebo Placebo Placebo Placebo Placebo
#>  [49] Placebo Placebo Active  Active  Active  Active  Active  Active
#>  [57] Active  Active  Active  Active  Active  Active  Active  Active
#>  [65] Active  Active  Active  Active  Active  Active  Active  Active
#>  [73] Active  Active  Active  Active  Active  Active  Active  Active
#>  [81] Active  Active  Active  Active  Active  Active  Active  Active
#>  [89] Active  Active  Active  Active  Active  Active  Active  Active
#>  [97] Active  Active  Active  Active
#> Levels: Placebo Active
mydata[[1]]
#>   [1]   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17
#>  [18]  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34
```

```
#>  [35]  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51
#>  [52]  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68
#>  [69]  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85
#>  [86]  86  87  88  89  90  91  92  93  94  95  96  97  98  99 100
```

It is not possible to select more than one element using double brackets; The result will always be a vector (instead of a `data.frame`)

4.3 Selecting observations and variables in a `data.frame`

Selecting observations and variables in a `data.frame` works more or less the same for `data.frames` as it does for lists. However because a `data.frame` is two dimensional we can two indices between the square brackets. The first one corresponds to the observations (rows) and the second corresponds to the variables (columns). So, as an example, we can select the first two observations from the third variable in the `data.frame` using the syntax:

```
mydata[c(1, 2), 3]
#> [1] Male   Female
#> Levels: Male Female
```

When the first or second position is left blank all rows or columns are selected. For example:

```
mydata[, 3]       # sex (3rd variable) for all patients
#>   [1] Male   Female Male   Female Male   Male   Male   Male   Male   Male
#>  [11] Male   Male   Male   Female Male   Male   Male   Male   Female Male
#>  [21] Female Female Female Female Male   Male   Female Male   Female Male
#>  [31] Male   Female Female Male   Female Female Male   Male   Male   Male
#>  [41] Male   Male   Male   Male   Female Male   Male   Male   Female Male
#>  [51] Male   Female Male   Male   Male   Male   Female Female Male   Male
#>  [61] Female Female Female Male   Male   Male   Male   Male   Male   Male
#>  [71] Male   Female Male   Male   Male   Male   Female Male   Male   Female
#>  [81] Male   Male   Female Male   Female Male   Male   Male   Male   Male
#>  [91] Female Male   Male   Male   Male   Female Female Male   Male   Male
#> Levels: Male Female
mydata[c(1,2), ]  # all variables for the first two patients
#>   ptno  treatm    sex age height weight comorb painbl painfu  race
#> 1    1 Placebo   Male  58    190     97      1     45     64 Asian
#> 2    2 Placebo Female  68    171     61      1     45     68 Asian
mydata[c(-3,-4), 'sex'] # Negative numbers and names can also be used
#>  [1] Male   Female Male   Male   Male   Male   Male   Male   Male   Male
#> [11] Male   Female Male   Male   Male   Male   Female Male   Female Female
#> [21] Female Female Male   Male   Female Male   Female Male   Male   Female
#> [31] Female Male   Female Female Male   Male   Male   Male   Male   Male
#> [41] Male   Male   Female Male   Male   Male   Female Male   Male   Female
#> [51] Male   Male   Male   Male   Female Female Male   Male   Female Female
#> [61] Female Male   Male   Male   Male   Male   Male   Male   Male   Female
#> [71] Male   Male   Male   Male   Female Male   Male   Female Male   Male
#> [81] Female Male   Female Male   Male   Male   Male   Male   Female Male
#> [91] Male   Male   Male   Female Female Male   Male   Male
#> Levels: Male Female
```

We have to be careful when we want to select a single variable from a `data.frame`, as we do above. The result will now no longer be a `data.frame` but it is transformed to a `vector`. When we want to prevent this we can use `drop=FALSE`, as follows:

```
mydata[, 3, drop=FALSE]       # data.frame met een variabele
#>        sex
#> 1     Male
#> 2   Female
#> 3     Male
#> 4   Female
#> 5     Male
#> 6     Male
#> 7     Male
#> 8     Male
#> 9     Male
#> 10    Male
#> 11    Male
#> 12    Male
#> 13    Male
#> 14  Female
#> 15    Male
#> 16    Male
#> 17    Male
#> 18    Male
#> 19  Female
#> 20    Male
#> 21  Female
#> 22  Female
#> 23  Female
#> 24  Female
#> 25    Male
#> 26    Male
#> 27  Female
#> 28    Male
#> 29  Female
#> 30    Male
#> 31    Male
#> 32  Female
#> 33  Female
#> 34    Male
#> 35  Female
#> 36  Female
#> 37    Male
#> 38    Male
#> 39    Male
#> 40    Male
#> 41    Male
#> 42    Male
#> 43    Male
#> 44    Male
#> 45  Female
#> 46    Male
#> 47    Male
#> 48    Male
#> 49  Female
#> 50    Male
#> 51    Male
```

```
#> 52  Female
#> 53    Male
#> 54    Male
#> 55    Male
#> 56    Male
#> 57  Female
#> 58  Female
#> 59    Male
#> 60    Male
#> 61  Female
#> 62  Female
#> 63  Female
#> 64    Male
#> 65    Male
#> 66    Male
#> 67    Male
#> 68    Male
#> 69    Male
#> 70    Male
#> 71    Male
#> 72  Female
#> 73    Male
#> 74    Male
#> 75    Male
#> 76    Male
#> 77  Female
#> 78    Male
#> 79    Male
#> 80  Female
#> 81    Male
#> 82    Male
#> 83  Female
#> 84    Male
#> 85  Female
#> 86    Male
#> 87    Male
#> 88    Male
#> 89    Male
#> 90    Male
#> 91  Female
#> 92    Male
#> 93    Male
#> 94    Male
#> 95    Male
#> 96  Female
#> 97  Female
#> 98    Male
#> 99    Male
#> 100   Male
```

We can also use a single index between the square brackets. This works as if the `data.frame` was a list of variables (it's columns).

```
mydata[1]
#>     ptno
#> 1      1
#> 2      2
#> 3      3
#> 4      4
#> 5      5
#> 6      6
#> 7      7
#> 8      8
#> 9      9
#> 10    10
#> 11    11
#> 12    12
#> 13    13
#> 14    14
#> 15    15
#> 16    16
#> 17    17
#> 18    18
#> 19    19
#> 20    20
#> 21    21
#> 22    22
#> 23    23
#> 24    24
#> 25    25
#> 26    26
#> 27    27
#> 28    28
#> 29    29
#> 30    30
#> 31    31
#> 32    32
#> 33    33
#> 34    34
#> 35    35
#> 36    36
#> 37    37
#> 38    38
#> 39    39
#> 40    40
#> 41    41
#> 42    42
#> 43    43
#> 44    44
#> 45    45
#> 46    46
#> 47    47
#> 48    48
#> 49    49
#> 50    50
#> 51    51
```

```
#> 52   52
#> 53   53
#> 54   54
#> 55   55
#> 56   56
#> 57   57
#> 58   58
#> 59   59
#> 60   60
#> 61   61
#> 62   62
#> 63   63
#> 64   64
#> 65   65
#> 66   66
#> 67   67
#> 68   68
#> 69   69
#> 70   70
#> 71   71
#> 72   72
#> 73   73
#> 74   74
#> 75   75
#> 76   76
#> 77   77
#> 78   78
#> 79   79
#> 80   80
#> 81   81
#> 82   82
#> 83   83
#> 84   84
#> 85   85
#> 86   86
#> 87   87
#> 88   88
#> 89   89
#> 90   90
#> 91   91
#> 92   92
#> 93   93
#> 94   94
#> 95   95
#> 96   96
#> 97   97
#> 98   98
#> 99   99
#> 100  100
mydata[[1]]
#>   [1]   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17
#>  [18]  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34
#>  [35]  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51
```

```
#>  [52]  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68
#>  [69]  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85
#>  [86]  86  87  88  89  90  91  92  93  94  95  96  97  98  99 100
```

Chapter 5

Functions

For a large part the power of R leas in the ease by which functions can be made. This had lead to lot's of code becoming available (most often in the form of R packages) to solve all kinds of problems.

5.1 What are functions

We have seen many examples of functions earlier in the course. For example `read.spss` to read spss data or `mode` to learn about an objects storage mode. So what exactly is a function? Actually a function is just a piece of code - that is a part of a program - that has peen given a name so it can be reused whenever necessary.

In R a function definition looks as follows.

```
function (arguments) {
  body
}
```

The *body* of the code consists for the code we want to execute. We will later come back to the function arguments.

5.2 Using functions

Let's say that we want to compute some summary statistics (the `mean` and the `sd`). Because there can be some outliers it is decided that the smallest and largest values are removed before computing these statistics. We can do this using the code

```
x <- sort(x)
x <- x[c(-1, -length(x))]
rv <- list(mean=mean(x), sd=sd(x))
```

Because we pan to use this code several times for different variables we decide to turn it into a function[1].

```
sum_stats <- function(x){
  x <- sort(x)
  x <- x[c(-1, -length(x))]
```

[1]Normally such a function would also contain some code to check for errors in the input values but we omit that here for simplicity

```
  list(mean=mean(x), sd=sd(x))
}
```

Now we can use this function every time we want to compute the summary statistics:

```
sum_stats(mydata$age)
#> $mean
#> [1] 53.56122
#>
#> $sd
#> [1] 12.53865
```

When we call the function the parameter `x` in the body of the function is replace by the argument `mydata$age` in the function call. The last value in the body of the function is the return value of the function. It can be printed, stored in a variable or used in further calculations.

```
the_summary <- sum_stats(mydata$age)
```

We can make the value that is returned explicit by using the return *keyword*. In this way the value that is return does not have to be the last line of the function.

```
sum_stats <- function(x){
  x <- sort(x)
  x <- x[c(-1, -length(x))]
  rv <- list(mean=mean(x), sd=sd(x))
  return(rv)
}
```

One thing you should be aware of is that while normally the result of a expression is printed when it is not assigned to a variable, this is no longer true within a function. When you do want to show the output of an expression you will have to use the `print` function explicitly.

5.3 Functions with multiple arguments

It is also possible to create functions with multiple arguments:

```
sum_stats <- function(x, outliers){
  x <- sort(x)
  x <- x[c(-(1:outliers), -((length(x)-outliers+1):length(x)))]
  rv <- list(mean=mean(x), sd=sd(x))
  return(rv)
}
sum_stats(mydata$age, 2)
#> $mean
#> [1] 53.58333
#>
#> $sd
#> [1] 12.07142
```

Whenever there are multiple parameters it might be difficult remember which parameter does what. This is why we can also use the names of the parameter in the function call. When we do this we also are more flexible in the order in which we specify the arguments. When the name is not specified the position is still decisive.

```
sum_stats(mydata$age, outliers=2)
#> $mean
```

```
#> [1] 53.58333
#>
#> $sd
#> [1] 12.07142
sum_stats(outliers=2, mydata$age)
#> $mean
#> [1] 53.58333
#>
#> $sd
#> [1] 12.07142
```

5.4 Replacement functions

There are several functions with a name ending in `<-`. For example `is.na<-`. This are so called replacement functions. The arrow at the end of the name indicates that they can be used at the left hand side of an assignment. For example `is.na<-` can be used in the following way to set specific elements of a vector to missing.

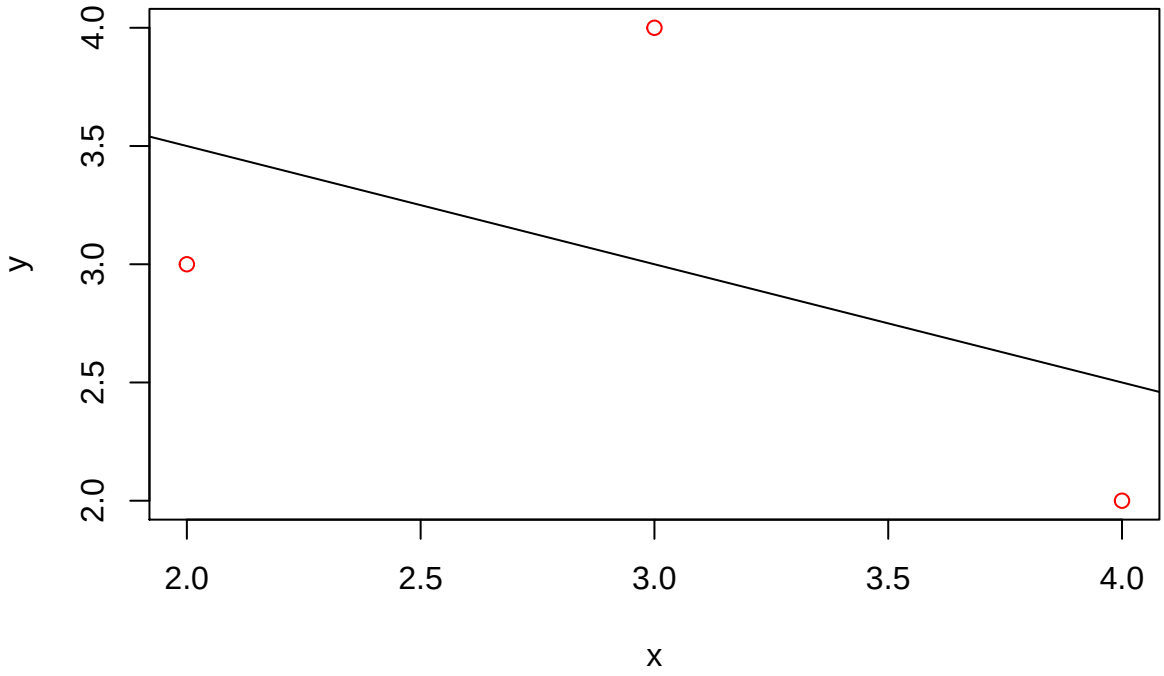
```
a_vector <- 1:5
is.na(a_vector) <- 2
print(a_vector)
#> [1]  1 NA  3  4  5
```

5.5 Ellipses (...)

Sometimes the parameter list of a function contains three dots (also called an ellips). This indicates that an unspecified number of extra unnamed arguments can be given to the function. Some frequently used functions with an ellips in the parameter list are: `c`, `data.frame`, `min`, `max` and `cat` but there are many others.

Often the ellipse is used to pass on parameters to other functions. Like in the fillowing function were the function `plot_regline` passes on arguments to the plot function.

```
plot_regline<- function(x, y, ...){
  lm1 <- lm(y~x)
  plot(x, y, ...)
  abline(lm1)
}
plot_regline(c(2,3,4), c(3,4,2), col='red')
```



Lexical scoping

When a function tries to use a variable that is not one of the parameters and is not determined elsewhere in the function. It looks for the variable the the environment in which the function was defined. This can be the global environment (basically everything outside function definitions) or an other function if the functions are nested. We try to illustrate this with some examples

```
f <- function(x){
  x+y
}
y<-5
f(2)
#> [1] 7
f <- function(x){
  y<-2
  g <- function(z){
    z+y
  }
  g(x)
}
f(2)
#> [1] 4
f <- function(x){
  y<-x
  g <- function(z){
    z+y
  }
  g
```

```
}
h <- f(10)
y <-20
h(2)
#> [1] 12
```

Chapter 6

Loops and controll structures

Often it will not be the case that we can write a program in which we can simply execute every line one by one until we reach the end. What task has to be done, and how many times something has be be done sometimes depends on parameters. Here we will discuss the various constructs to make this happen. Note that you will probably not use any control constructs when you use R in an interactive fashion.

6.1 The if-statement

The simplest (and probably most used) construct to conditionally execute part of a syntax is the if-statement. This statement looks as follows.

```
if (condition){
 code1
} else {
 code2
}
```

Here the code between the first pair of curly brackets is only executed when the condition is true and the code in between the second pair of curly brackets only when it is false. The `else` keyword and the second pair of braces is optional. An example is:

```
if (x >= 0){
 rtx <- sqrt(X)
} else {
 warning('Cannot take the root of a negative number - set to zero')
 rtx <- 0
}
```

Here the first pair of braces could be omitted as their purpose is only to group the code and there is only a single command inside.

```
if (x >= 0)
 rtx <- sqrt(X)
else {
 warning('Cannot take the root of a negative number - set to zero')
 rtx <- 0
}
```

The condition has to result is a single value that is either TRUE or FALSE. If there are more R will only look at the first one and issue a warning.

This is different from the function `ifelse(test, yes, no)`. This function returns a vector of which the element i will be equal to element i of the second parameter if element i of the test is TRUE and equal to element i of the third parameter otherwise.

If returns the value of the last expression that was evaluated. This explains for example how the function NROW works, which is defined as follows:

```
NROW(X)<-function{
  if (length(d <- dim(x))) d[1L] else length(x)
}
```

6.2 for-loops

Sometimes we want to repeat some statements a couple of times. For this purpose we have the `for`-loop. This construct looks as follows:

```
for (index in vector) {
 code
}
```

The variable `index` will take every value in the vector one by one and for each of the values the code between the braces will be executed. Here is a simple example:

```
for (i in c(1, 2, 3)) {
  print(sqrt(i))
}
```

```
## [1] 1
## [1] 1.414214
## [1] 1.732051
```

Within a `for`-loop we can use the commands `next` and `break`. `next` immediately moves to the next iteration of the loop. The remainder of the code between the braces will no longer be executed. With `break` we immediately leave he loop. We will illustrate the use of `next` and `break` with an example.

```
for (i in c(1, 2, 3, 4, 5)) {
  print(i)
  j<- sqrt(i+1)
  if (i==2) {
    next
  }
  if (i==5) {
    break
  }
  print(j)
}
```

```
## [1] 1
## [1] 1.414214
## [1] 2
## [1] 3
## [1] 2
## [1] 4
## [1] 2.236068
## [1] 5
```

6.3 while

Another loop that is sometimes used is the while loop

```
while (condition) {
 code
}
```

This kind of loop executes the code when the condition is TRUE. When the code has been executed the condition is checked again and if it is still TRUE the code is executed again. And so on, until the condition becomes FALSE.

6.4 repeat

The last kind of loop that we will discuss is the repeat loop. It looks as follows.

```
repeat {
 code
}
```

Notice that there is no condition. The body of the loop keeps being executed until we use a `break` statement (or until an error is made).

Chapter 7

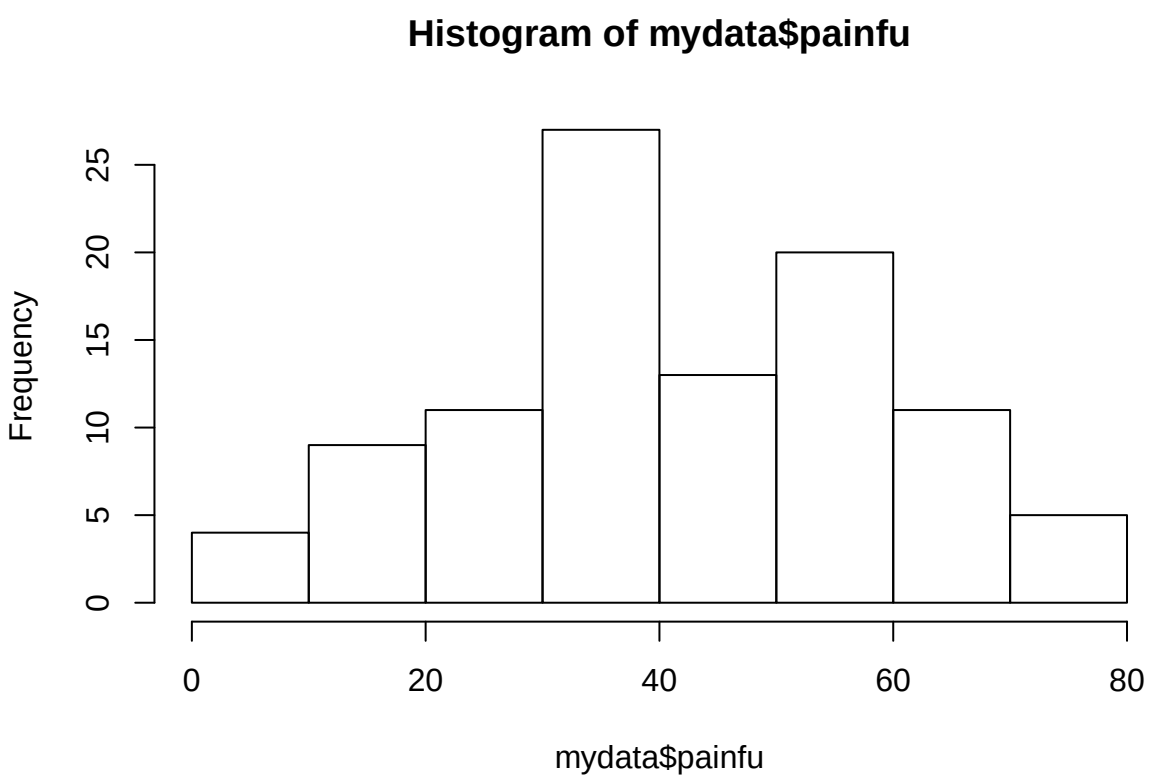
"R plots"

Often you get more information about a data.set by making graphs than by making tables. Here we will show some basic commands that R provides for this. In R there are several packages for creating plots. The main ones are: * graphics * lattice * ggplot2 Here we will discuss the base graphics from the graphics library. This package is the oldest of the graphical packages, it comes with the R installation (so you do not have to use `install.packages`), and is loaded automatically when R starts. For these reasons the type of graphics it creates are called 'base' plots. The plotting functions in the `graphics` package can be divided in two types. The so called higher-level plotting functions and the lower-level plotting functions. The distinction is that the former create a more or less complete plot while the latter perform very specific functions (like adding specific points at certain locations, adding a legend and so on). Often we build complicated functions step by step by first using a higher-level plotting function and then adding thing to it using lower-level plotting functions.

7.1 Histograms

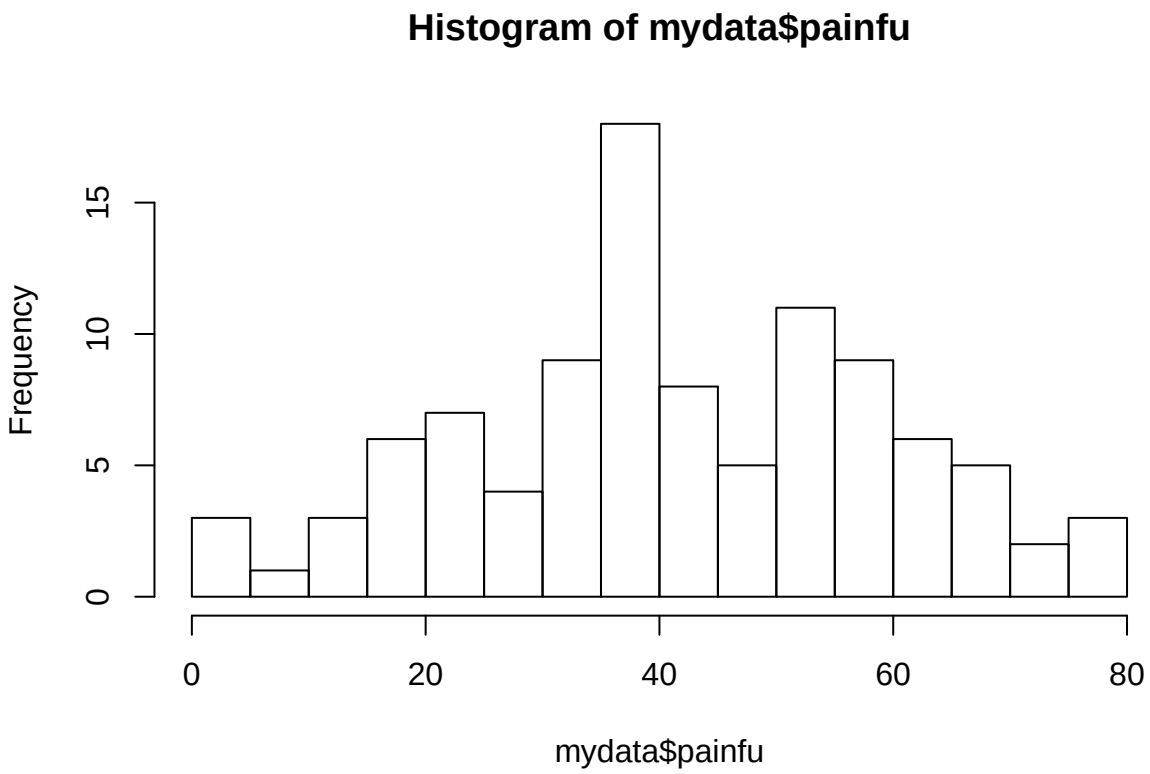
If we want to learn something about the distribution of a single continous variable we can make a histogram. In R this is done using the function `hist`. Let's look at the pain score at the follow-up.

```
hist(mydata$painfu)
```



If we run this code in RStudio the plot is displayed in the bottom right-hand corner. From there we can save the plot as an pdf, or image file by clicking 'export' if we want to keep it. We can have more control over the number of bins that is used by using the `breaks` parameter.

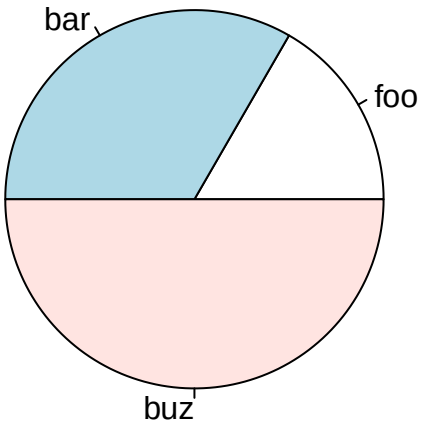
```
hist(mydata$painfu, breaks=20)
hist(mydata$painfu, breaks=seq(0,80,5))
```



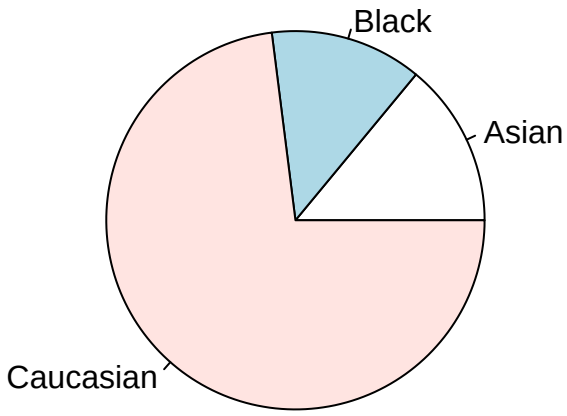
7.2 Pie charts, bar charts and dot charts

When we look at a single categorical variable a pie chart is often used to see how the observations are divided over the levels of the variable. The `pie` function takes a numeric vector of frequencies as input (together with a set of labels). It is often more convenient to work with a table.:

```
pie(c(2, 4, 6), labels=c('foo', 'bar', 'buz') )
```

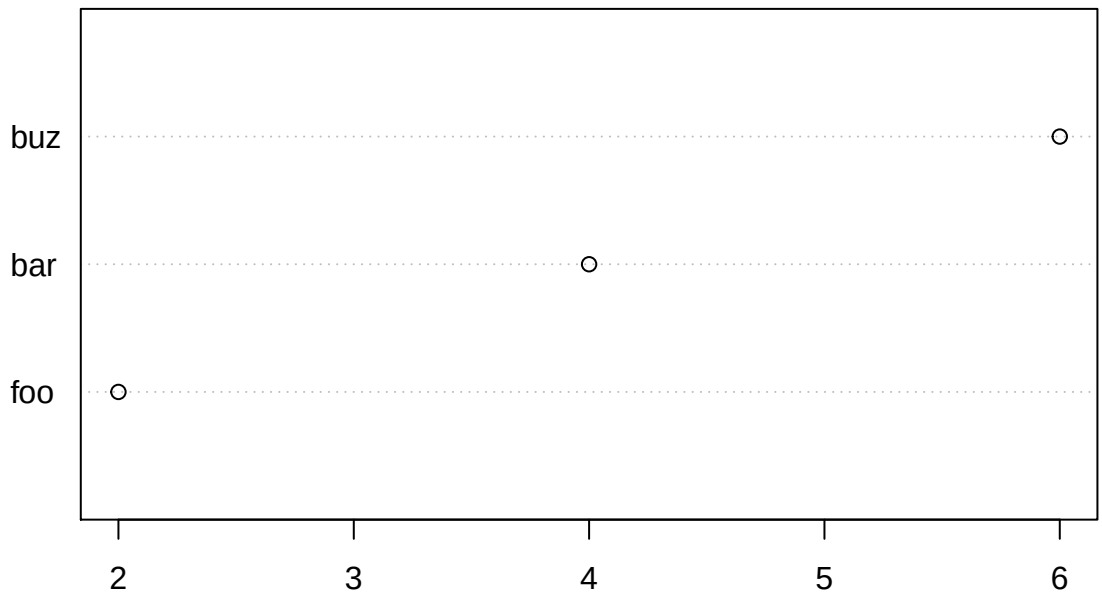



```
pie(table(mydata$race))
```

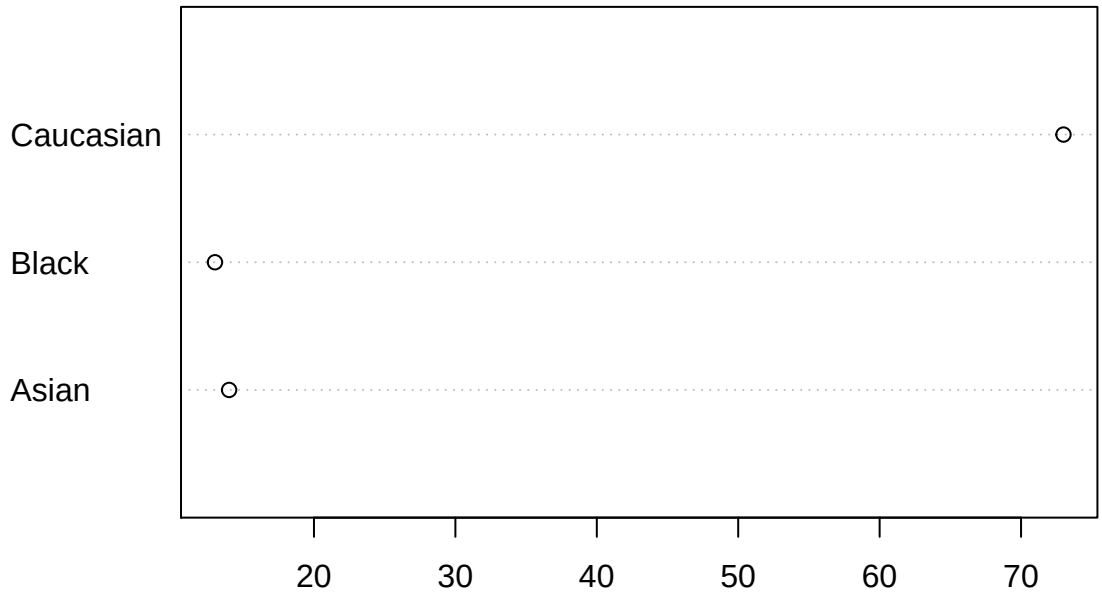


Some people recommend against using pie charts because people are bad at judging relative areas. Often a so called dotchart is suggested as an alternative.

```
dotchart(c(2, 4, 6), labels=c('foo', 'bar', 'buz') )
```

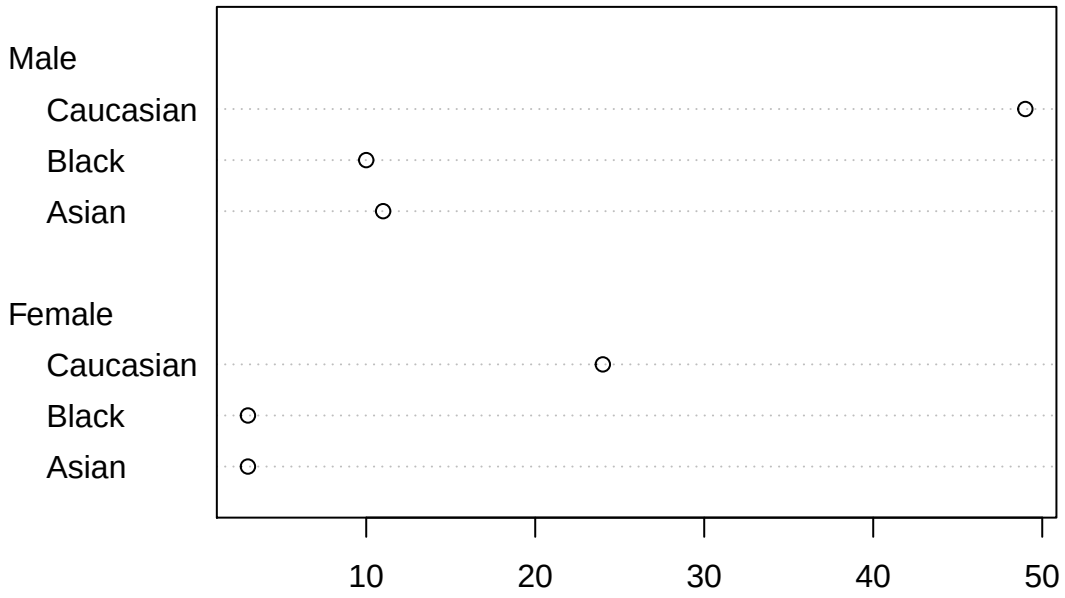



```
dotchart(c(table(mydata$race)))
```



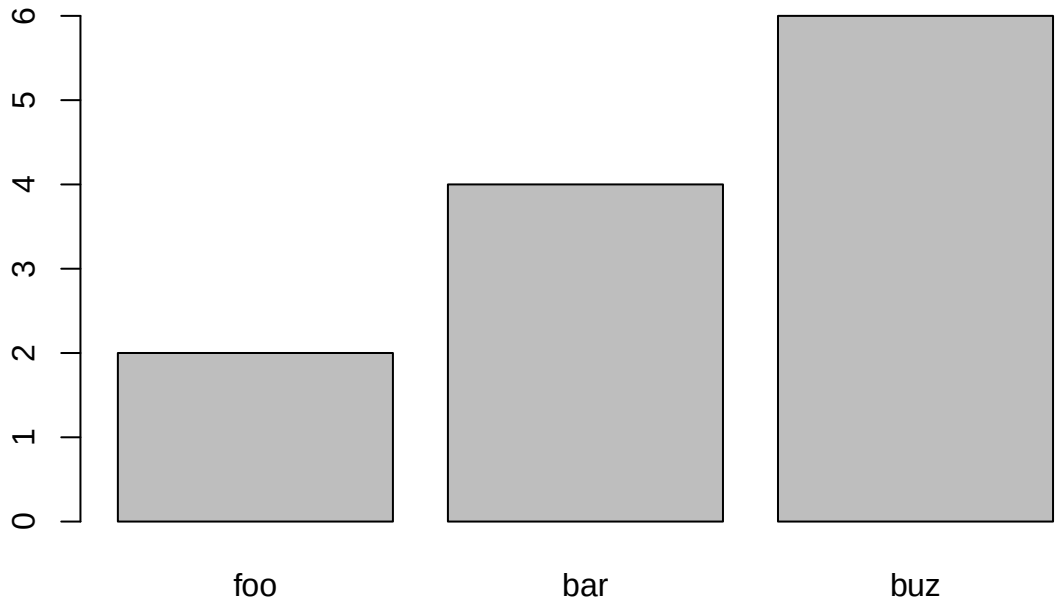
Note that we were required to transform the table to a vector by using the function `c`. Instead of a vector we can also use a matrix (or a two dimensional table which can readily be converted to it) as input for the function.

```
dotchart(table(mydata$race, mydata$sex))
```

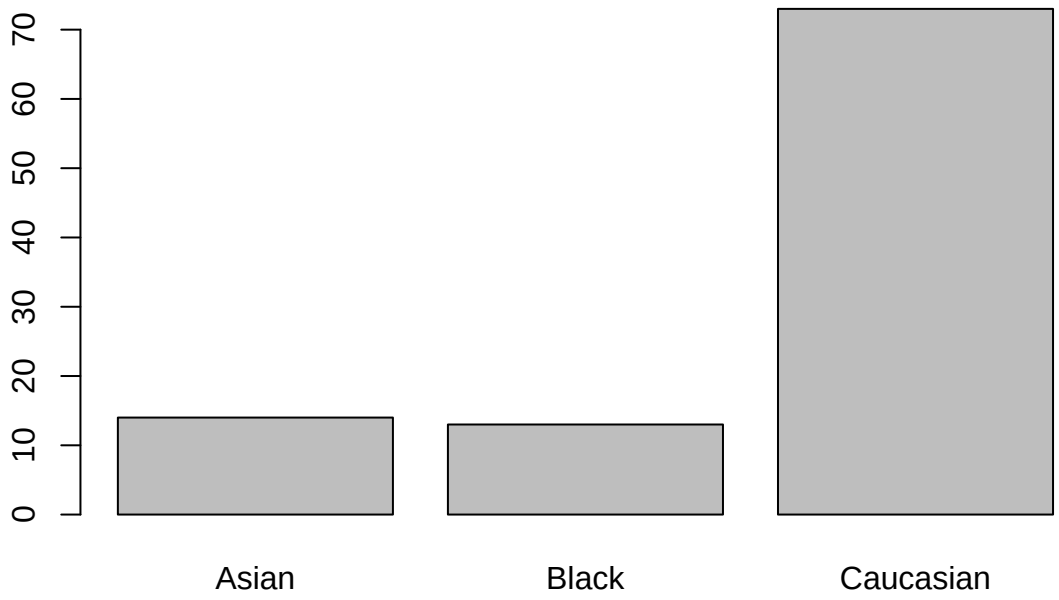


An other alternative for this type of data is a bar chart (which resembles a histogram but the x-axis is not a continuous scale but instead represents the different levels of the variable). In R we use the command `barplot`. The names of the categories are now specified using the parameter 'names.arg' instead of 'labels'.

```
barplot(c(2, 4, 6), names.arg =c('foo', 'bar', 'buz')  )
```




```
barplot(table(mydata$race))
```

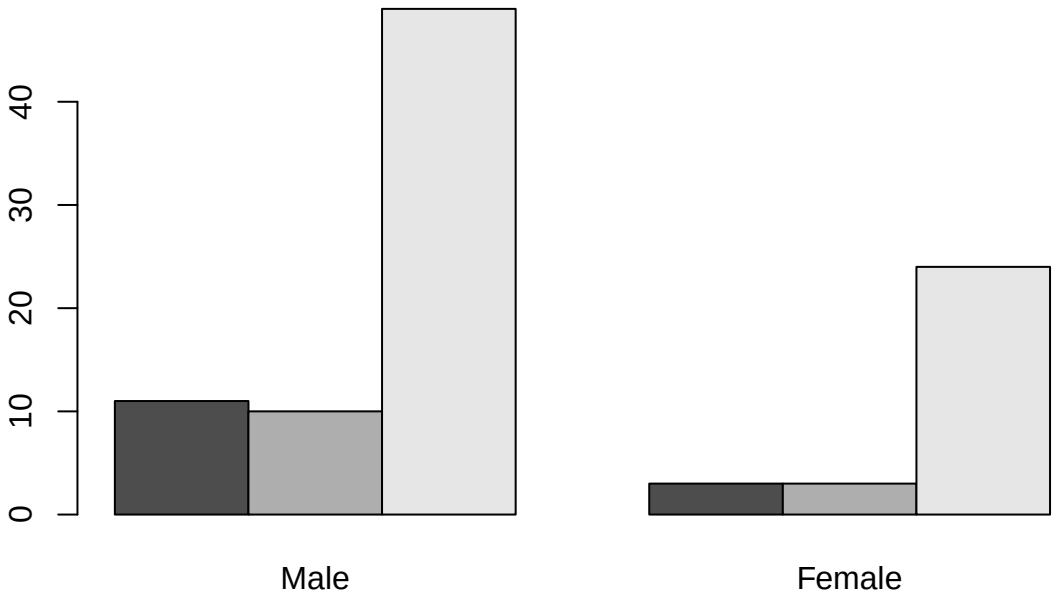


We can also use a matrix or 2-dimensional table.

```
barplot(table(mydata$race, mydata$sex))
```

It would be better to add a legend to the plot to explain what the colors mean. We will come back to that later. It is also possible to unstack the different categories using

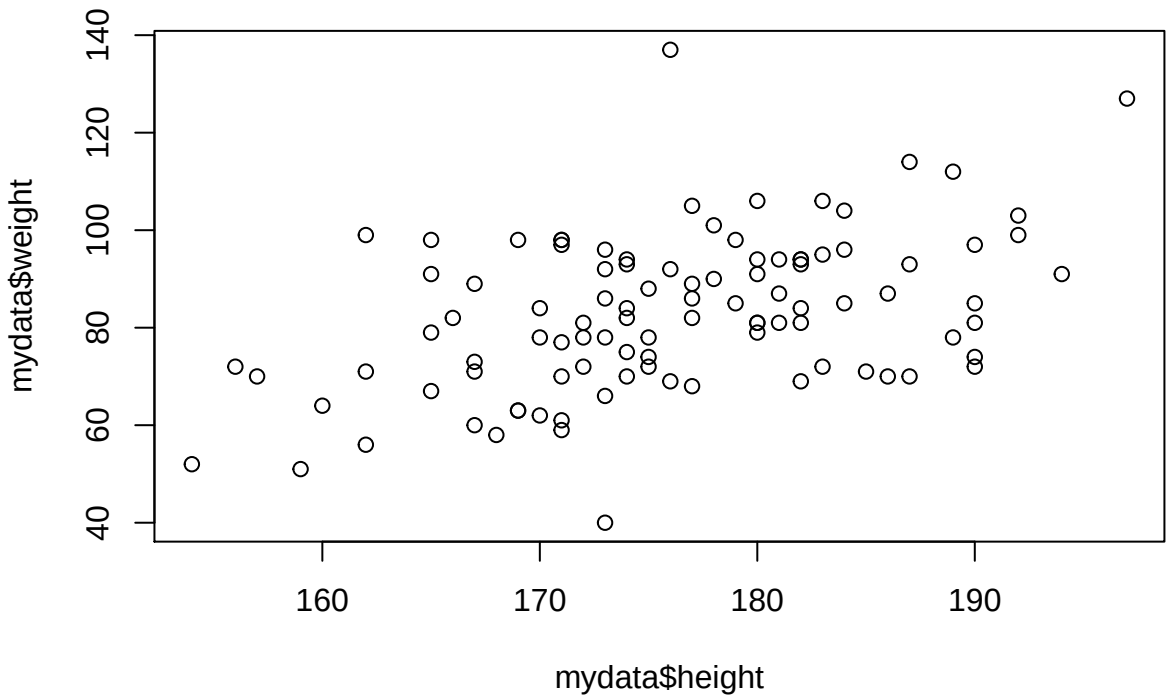
```
barplot(table(mydata$race, mydata$sex),beside = TRUE)
```



7.3 Scatterplots

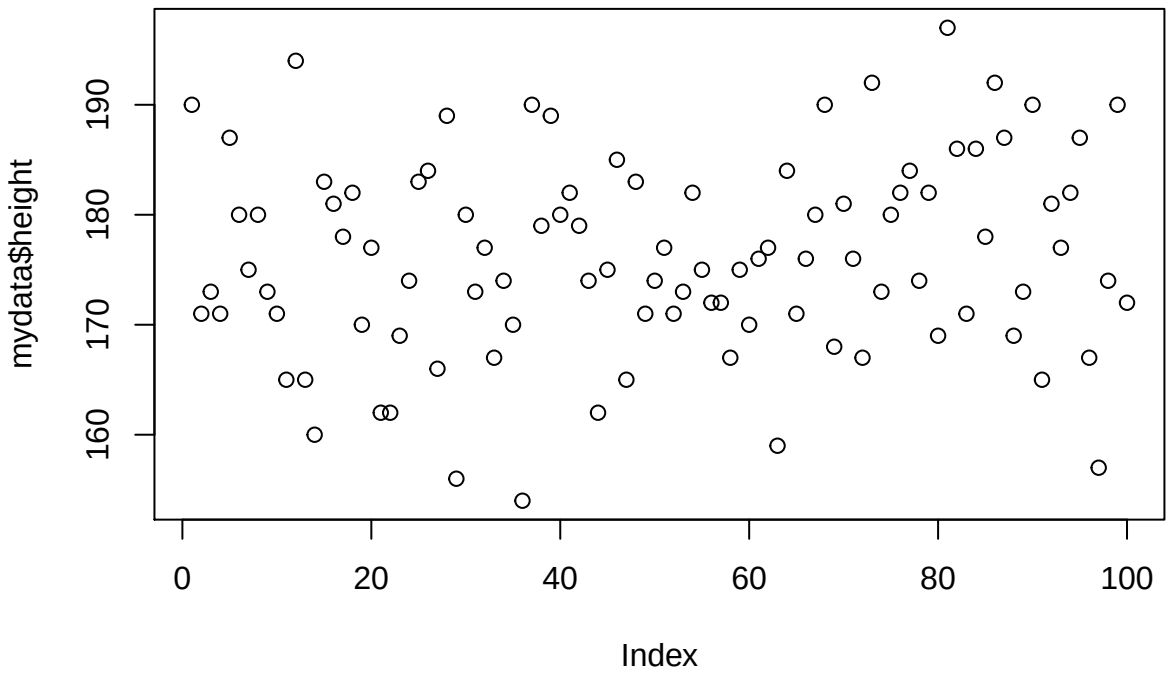
If we want to look at the relation between two variables we can use the `plot` command to make a scatter plot.

```
plot(mydata$height, mydata$weight)
```



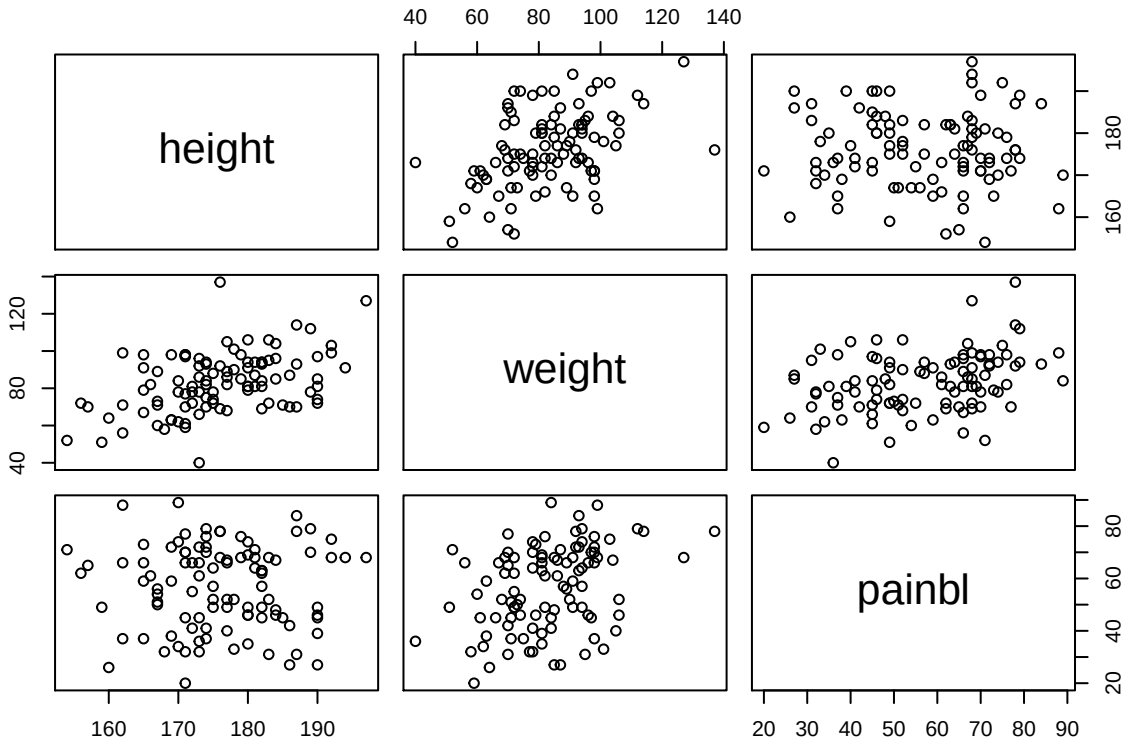
If we use only a single input vector the values are plotted against the observation numbers (which is not very useful here).

```
plot(mydata$height)
```



Sometimes we want to study the relation ship between several continuous variables at the same time. In this case we can make a scatter plot matrix. As the name implies this is basically a matrix of scatter plots with the scatter plots of all pairwise combinations of variables as off-diagonal elements.

```
pairs(~height+weight+painbl, data=mydata)
```



7.4 Common elements to plot commands

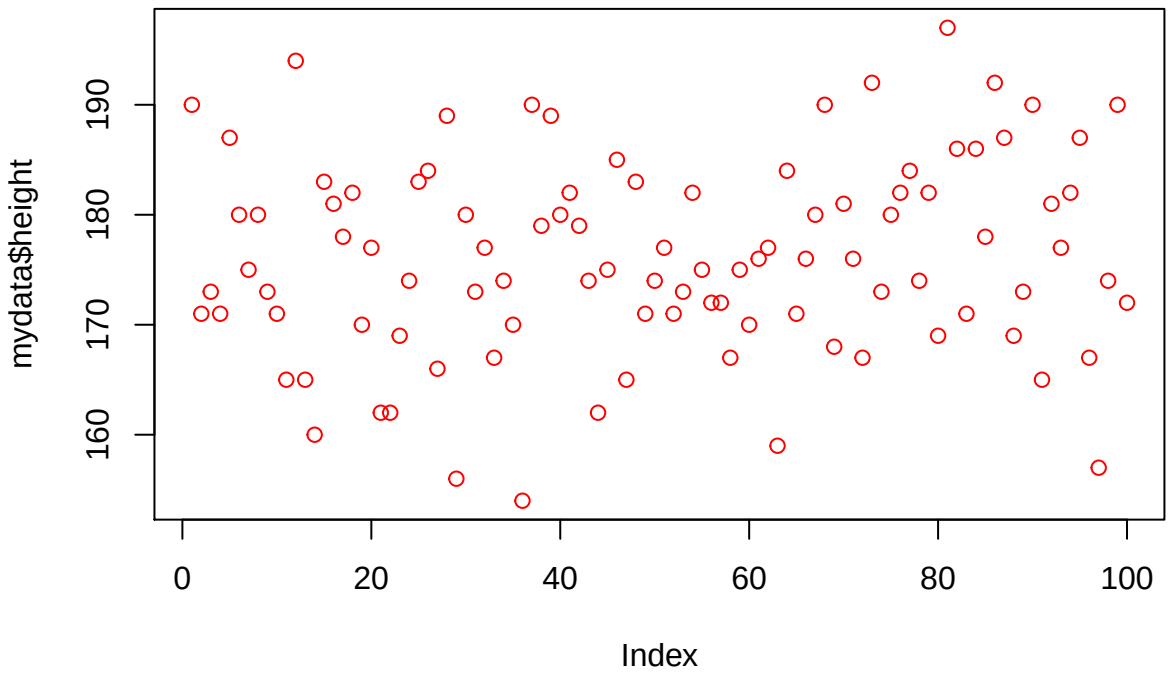
The plots above were very basic and we can do much to improve them by adding colors, changing the thickness and style of the lines and so on. Many of these things can be done by using extra parameters of the plotting function. For example `plot(mydata$height, col='red')`. A number of parameters (like `col` to set the color or `lwd` to set the thickness of lines) is common to most of the plotting commands discussed above and I will discuss the most important ones here.

Another way to set these plotting parameters is through the function `par`. For example: `par(col='red')` Some of the plotting parameters (like `mar` and `mfrow`) can even only be set in this fashion. A difference between using the parameters on the plot functions themselves is that when they are used in the plot functions they change an attribute for the current plot only but when they are used on the `par` function they change the attribute for all following plots.

7.4.1 col

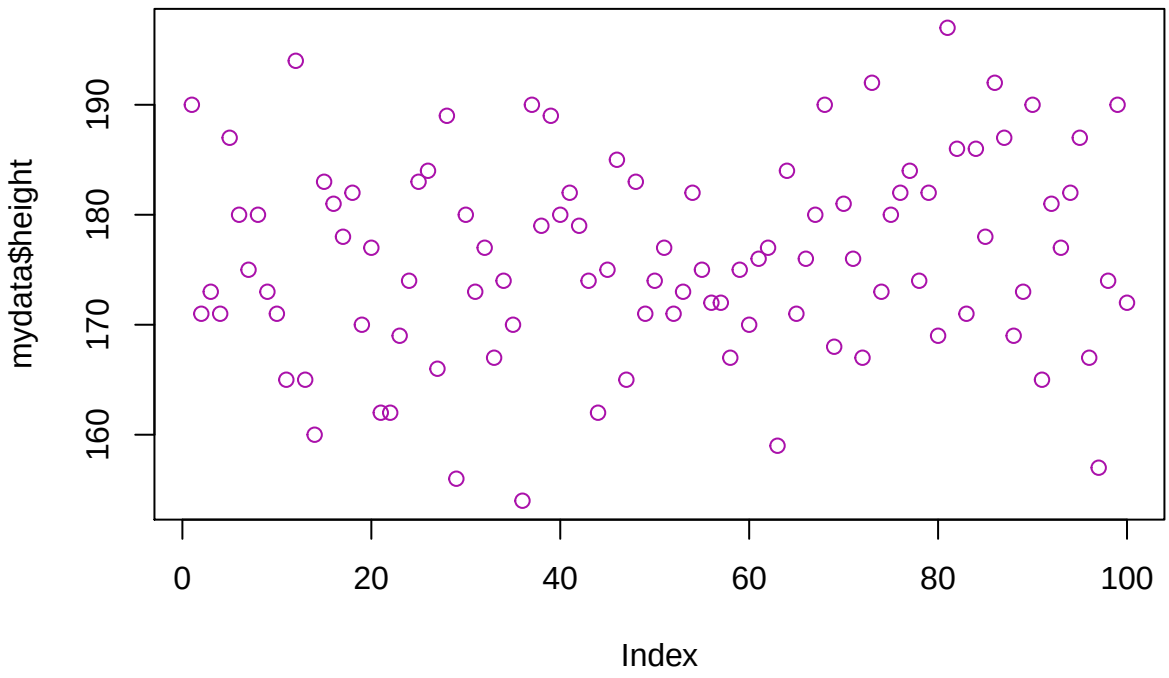
The parameter `col` can often be used to specify what collor to use to plot something. There are several ways do do this. The first is by using the name of the color. You can see all available color names by specifying the `colors()` function.

```
plot(mydata$height, col='red')
```



We can also use integers to specify the colors. The integers refer to positions in the current color palette. By default 1 is black, 2 is red, 3 is green, 4 is blue, 5 is cyan, 6 is magenta, 7 is yellow and 8 is grey. The color palette can be viewed or changed using the `palette` function. Finally we can specify colors by using a string representing the hexadecimal (that is 16-base, one does not use the normal 10 digits but the 16 'digits' 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E and F) color values. Basically this is a string where the first two characters indicate how strong the red signal is on a scale from 0-255 the next two characters indicate how strong the green signal is and the last two characters indicate how strong the blue signal is.

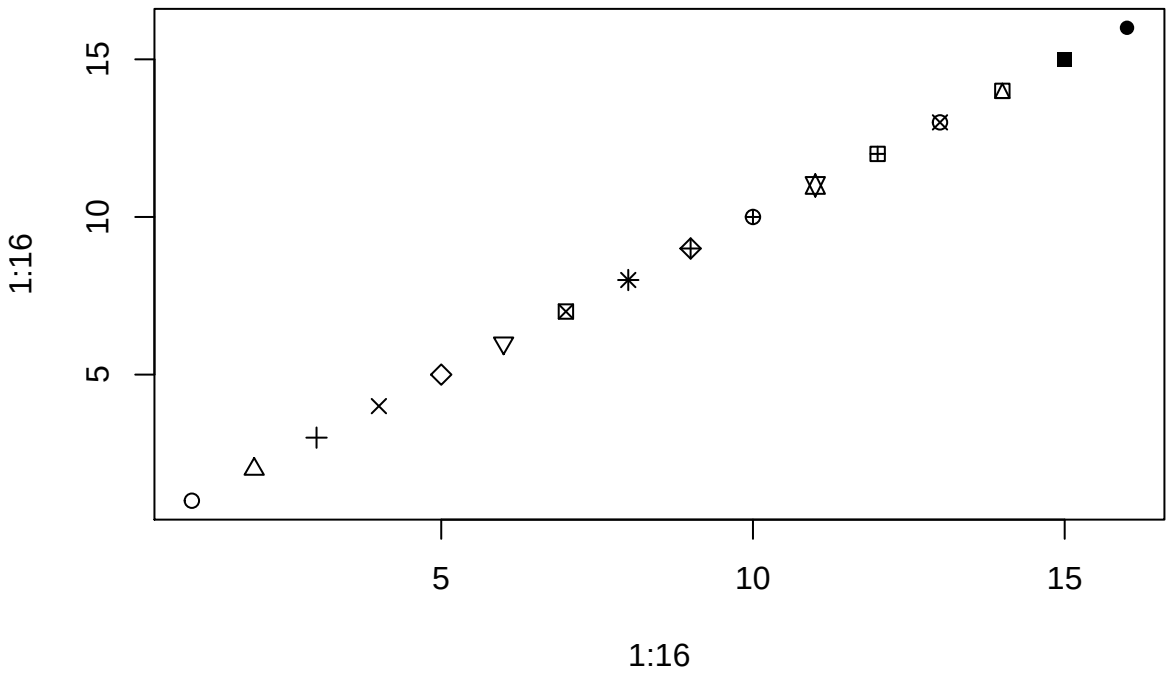
```
plot(mydata$height, col='#AA11AA')
```



7.4.2 pch

pch indicates the 'plot character'. Usually an integer is used. Each number stands for a specific symbol.

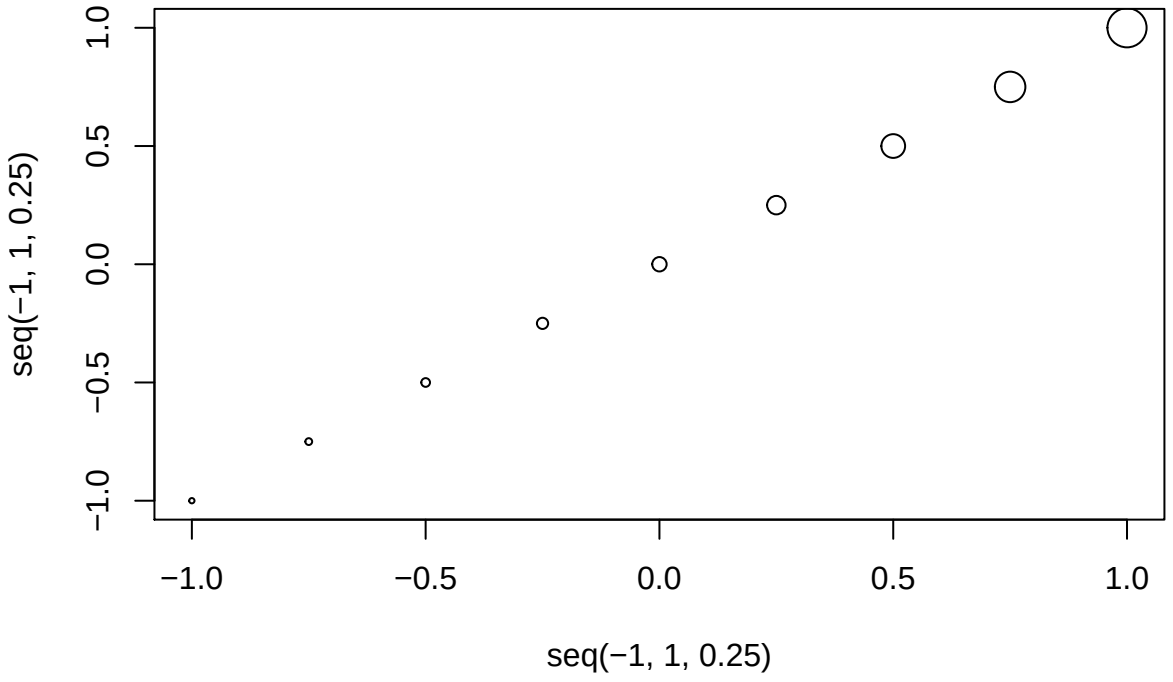
```
plot(x=1:16, 1:16, pch=1:16)
```



7.4.3 cex

This parameter specifies how large symbols and text should be.

```
plot(x=seq(-1,1,0.25), y=seq(-1,1,0.25), cex=exp(seq(-1,1,0.25)))
```



7.4.4 lwd

The parameter lwd indicates how thick the lines should be.

7.4.5 lty

This parameter indicates the line type: 1 is solid, 2 is dashed and 3 dotted. More complex patterns are also possible (see ?par).

7.4.6 xlim / ylim

Limits of the x and y axis. By default a little bit of extra space is added so the extreme values do not fall at the very edge of the plot.

7.4.7 xlab / ylab

Labels of the x and y axis

7.4.8 main

This parameter can be used to set the title of a plot. It is an alternative for the `title` command (See below).

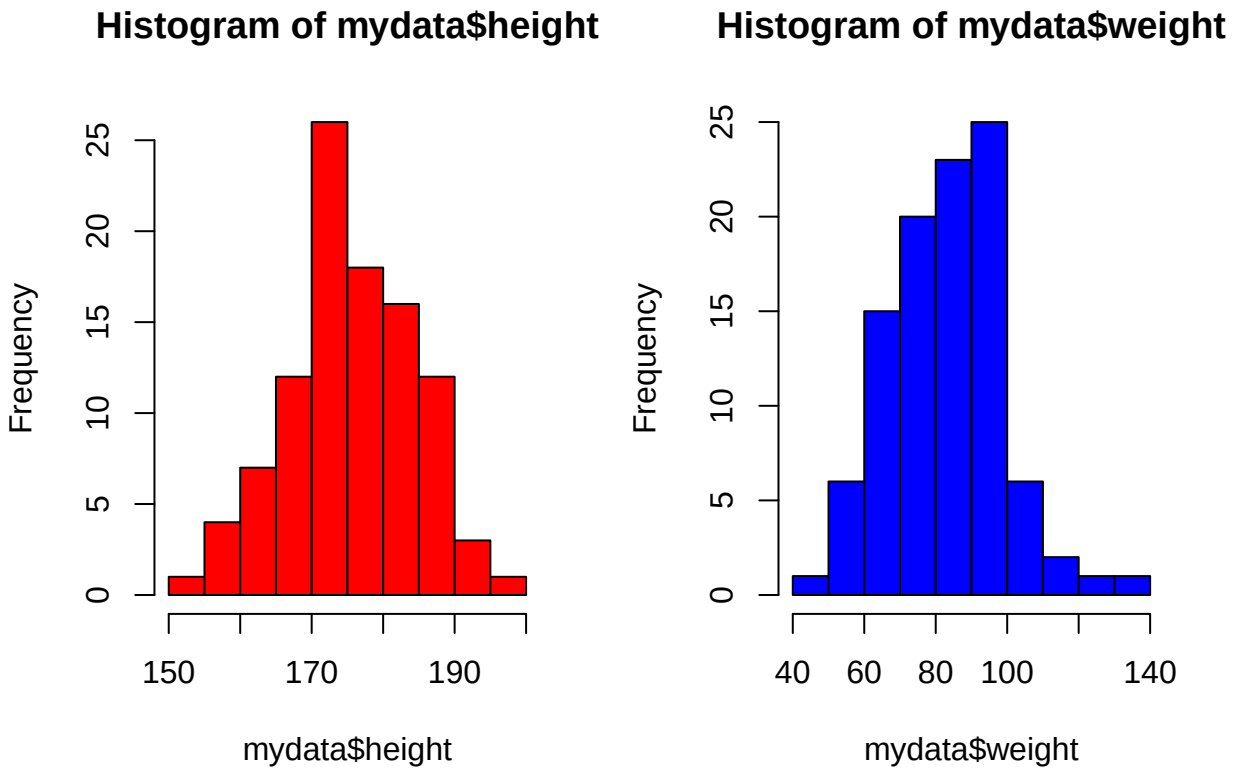
7.4.9 mar

This graphical parameter indicates the size of the margins in the plot. The argument should take the form of a numerical vector of length 4 (exactly). The margins should be specified in the order: lower margin, left margin, upper margin and right margin. Note that this parameter can only be set using `par`.

7.4.10 mfow

This parameter controls the number of subplots. Normally there is only a single subplot but this can be changed. The parameter `mfow` takes a vector of length 2. The first number specifies the number of rows of subfigures and the second the number of columns of subfigures.

```
opar <- par(mfrow=c(1, 2))
hist(mydata$height, col='red')
hist(mydata$weight, col='blue')
```




```
par(opar)
```

Notice how the return value of the par function can be used to restore all plotting parameters to their defaults.

7.5 some lower-level plot functions

As mentioned above the lower-level plot functions do not create a complete plot but instead they perform a very specific task. Common lower-level plotting functions are:

7.5.1 points(x, y, ...)

To add points to a plot.

7.5.2 lines(x, y, ...)

To add lines to a plot.

7.5.3 text(x, y, ...)

To add texts to a plot.

7.5.4 abline(...).

This function can be used to add reference lines. Horizontal lines can be added using `abline(h=y)` were y is the y coordinate of the line. Vertical lines can be drawn using `abline(v=x)`. Finally a diagonal line with equation $y = a + bx$ can be drawn using `abline(a=a, b=b)` (This is were the name of the function comes from).

7.5.5 legend(x, y, legend, ...)

With this function we can add a legend to a plot. Besides using x and y coordinates for the plot we can also indicate the position using text values as 'top', 'bottom', 'left' and 'right'.

7.5.6 title(main, sub, ...)

This function can be used to add a title and a subtitle to a plot.

7.5.7 polygon(x, y, border, col,...)

Draws a polygon. THis works the same way as the lines command but the end is connected with the beginning. The parameter `col` specifies the color with which the polygon will be filled while `border` specifies the color of the border.

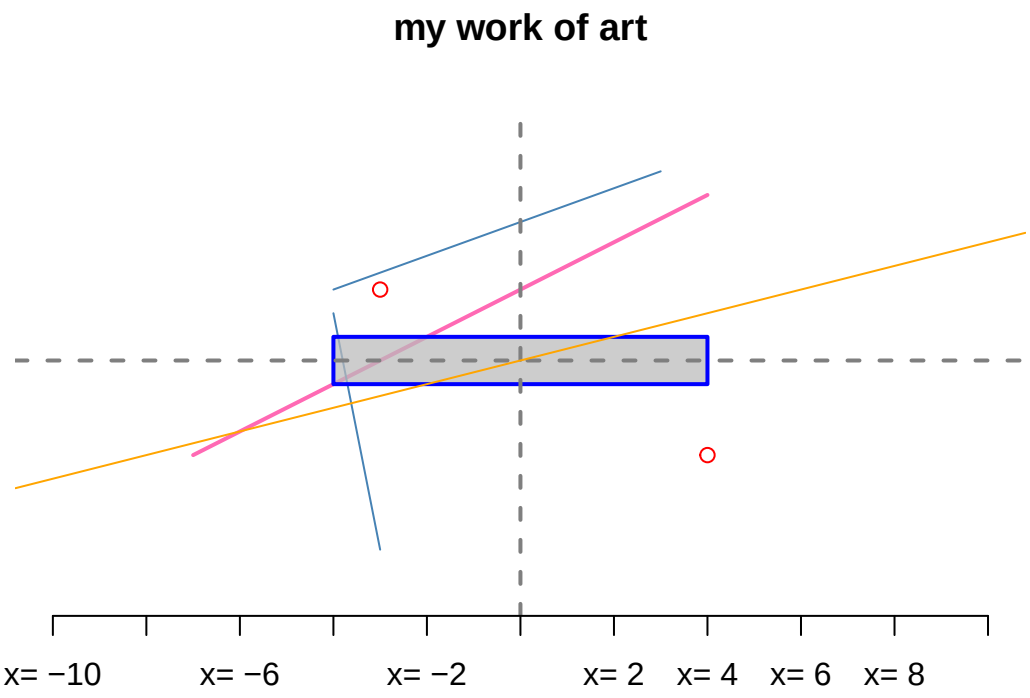
7.5.8 segments(x0, y0, x1, x1, ...)

Draws line segments from the points (x0, y0) to (x1, y1). Unlike the function `lines` the lines are not automatically connected.

Examine the code below and the resulting plot. We use many of the above functions and some additional ones. Most commands should be relatively self explanatory. Otherwise you can always use the documentation.

```
plot.new()
plot.window(xlim=c(-10,10), ylim=c(-10, 10))
points(c(-3, 4), c(3, -4), col= 'red')
lines(c(4, -7), c(7, -4), lwd=2,
      col= 'hotpink')
segments(x0=c(-4, -4), x1=c(3,-3), y0=c(3,2), y1=c(8,-8),
         col='steelblue')
```

```
polygon(c(4,4,-4,-4), c(-1,1,1,-1),
        border = 'blue', col = '#BBBBBBBB', lwd=2)
abline(v=0, lty=2, lwd=2, col='grey50') # vertical line x=0
abline(h=0, lty=2, lwd=2, col='grey50') # horizontal line y=0
abline(a=0, b=0.5, col='orange') # line denifed by y=a+bx
title('my work of art')
axis(side = 1, at = seq(-10,10, 2),
     labels=paste('x=',seq(-10,10, 2)))
```



7.5.9 Illustration of a the creation of a (somewhat) complicated plot

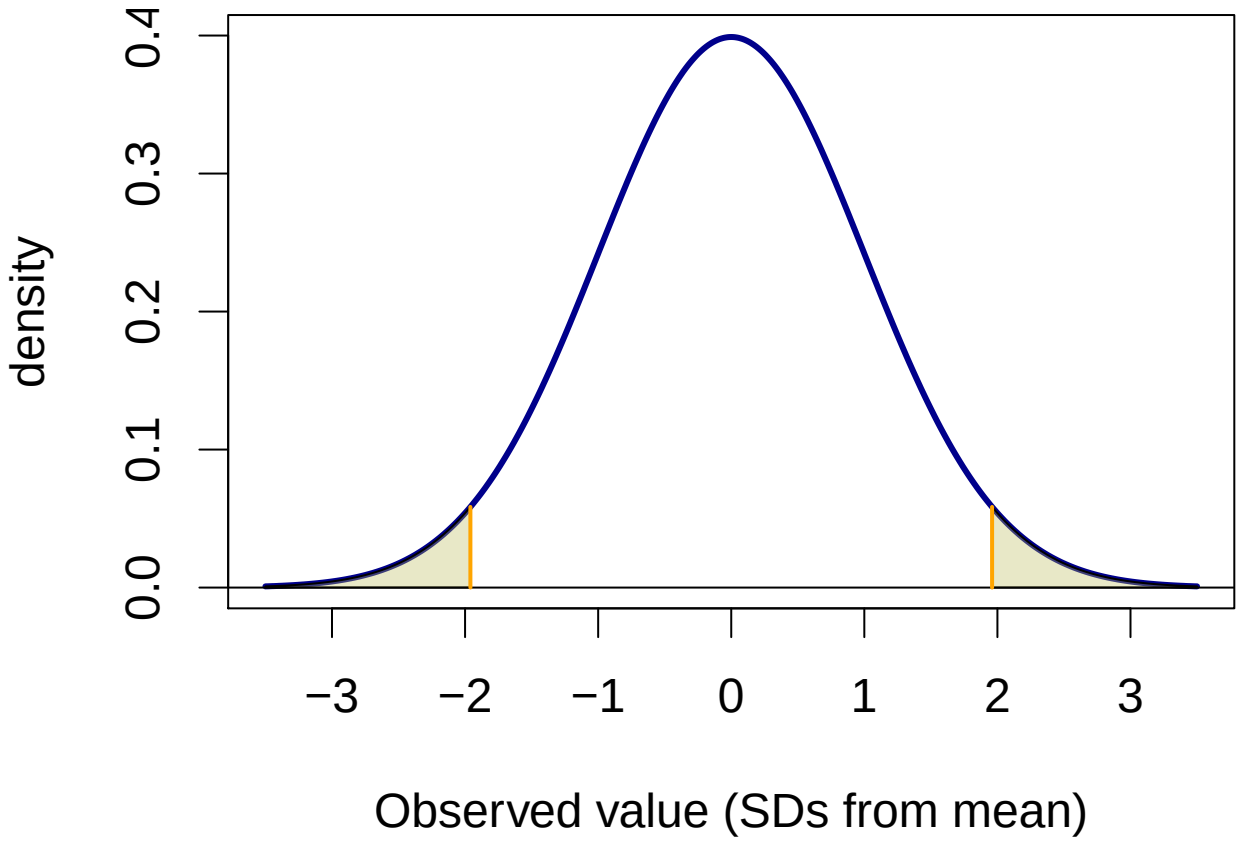
While the plot above shows a lot of different low-level commands. The resulting plot is not very useful. Below we show a more practical example: some code that was used to generate an illustrate of the normal distribution.

```
x <- seq(-3.5, 3.5, 0.005)
y <- dnorm(x, 0,1)
par(mar=c(4.6,4.1,0.1,0.1), cex=1.5)
plot(x=x,y=y, col='darkblue', type='l', lwd=3,
     xlab='Observed value (SDs from mean)', ylab='density')
q <- qnorm(0.025, 0, 1)
p <- dnorm(q, 0, 1)
abline(h=0)

cord.x <- c(-3.5, seq(-3.5, q, 0.005),  q)
```

```
cord.y <-c(0, dnorm(seq(-3.5,q, 0.005),0,1),0)
polygon(cord.x,cord.y,col='#BABA5555')
cord.x <- c(3.5, seq(3.5, -q, -0.005),  -q)
cord.y <-c(0, dnorm(seq(3.5,-q, -0.005),0,1),0)
polygon(cord.x,cord.y,col='#BABA5555')

lines(x=c(q, q), y=c(0, p), col= 'orange', lwd=2 )
lines(x=c(-q, -q), y=c(0, p), col= 'orange', lwd=2 )
```



Chapter 8

Basic statistical tests

8.1 Basic parametric Statistical tests for continuous data

8.1.1 Single sample

The simplest situation in which we can do a statistical test is perhaps that where we have a sample from a population of an outcome which we assume to have a normal distribution. We want to test the null hypothesis if the mean of this population is equal to some value. For example let's assume that we want to test if the average IQ of children with a particular syndrome deviates from that of the population (defined to be 100). We have a sample of 10 IQ scores (The values obtained were 74, 87, 100, 75, 98, 74, 79, 103, 95 and 110). So our null hypothesis is that the mean is 100 while the alternative hypothesis is that it is not. We decide that we will test the hypothesis using an alpha of 5%. (Of course we should have consulted a statistician to discuss the power of the test.) In R we can perform the test using the `t.test` function.

```
IQs<-c(74, 87, 100, 75, 98, 74, 79, 103, 95, 110)
t.test(IQs, mu=100)
#>
#>  One Sample t-test
#>
#> data:  IQs
#> t = -2.473, df = 9, p-value = 0.0354
#> alternative hypothesis: true mean is not equal to 100
#> 95 percent confidence interval:
#>  79.89508 99.10492
#> sample estimates:
#> mean of x
#>      89.5
```

From the output we see that the average in the sample was 89.5 points a difference of more than 10 points from the value of 100 we compare against. The p-value is 0.035 so we can reject the null. We also see that the confidence interval of the sample average runs from 79.9 to 99.1 so our estimate based on this small sample is still rather imprecise.

8.1.2 Two samples

8.1.2.1 Unpaired data

Arguably the most common situation in which we perform a statistical test is when we have two independent groups for which we want to compare an outcome that is (approximately) normally distributed. In this situation we can use the two sample t-test for unrelated samples. Imagine we have conducted a trial in which we measure the pain score at follow-up between two treatment arms. The data from this trial is included in the data set `pain`. We can perform the two sample t-test for unpaired data using the same `t.test` command we used earlier.

```
data(pain, package='BST02R')
t.test(painfu ~ treatm ,data=pain)
#>
#>  Welch Two Sample t-test
#>
#> data:  painfu by treatm
#> t = 4.7809, df = 95.585, p-value = 6.32e-06
#> alternative hypothesis: true difference in means is not equal to 0
#> 95 percent confidence interval:
#>   9.040738 21.879262
#> sample estimates:
#> mean in group Placebo  mean in group Active
#>                 49.72                34.26
```

From the output we see that the pain at follow-up was 50 in the active group while it was 50 for the patients that were given a placebo. If the null hypothesis that the mean pain score in the two groups is the same in the whole population would be true the probability of finding such a large difference would be very small as indicated by the p-value. So we reject the null hypothesis and conclude that the treatment helps to reduce pain.

Actually here we performed the variant of the t-test that does not make the assumption that the variance in the outcome is the same in the groups that are compared (the Welch two sample t-test). The classical t-test that is found in most statistical textbooks actually does make this additional assumption. If we wanted to perform the test this way in R we could have done this using the extra argument `var.equal=TRUE`.

8.1.2.2 Paired data

It is also interesting to see if the pain at follow-up is different from that at baseline. However, here we cannot do this using the two sample t-test described above as the patents whose pain is measured at the follow-up are the same as those which are measured at baseline. For this we need the two sample t-test for *paired* data (which is actually the same as the one sample t-test performed at the difference between the pain levels at follow-up and baseline). Let's look at this in the Placebo group.

```
data_placebo <- pain[pain$treatm=='Placebo',]
with(data_placebo, t.test(x=painbl, y=painfu))
#>
#>  Welch Two Sample t-test
#>
#> data:  painbl and painfu
#> t = 2.1923, df = 97.941, p-value = 0.03072
#> alternative hypothesis: true difference in means is not equal to 0
#> 95 percent confidence interval:
#>   0.7148663 14.3651337
#> sample estimates:
```

```
#> mean of x mean of y
#>     57.26     49.72
```

We see that the pain score is decreased by 7.5 points. So the difference is statistically significant.

8.1.3 More than two samples

The independent samples t-test can be generalized to compare more than two groups. This is called ANOVA (ANalysis Of VAriance). The null hypothesis that is tested is that the group means are all equal. This hypothesis is tested by comparing the variance among the group means with the variance within each group (hence the name of the statistical technique).

Let's use R to test if pain scores at baseline differ among the different ethnicities.

```
anova_eth <- aov(painbl ~ race, data=pain)
summary(anova_eth)
#>              Df Sum Sq Mean Sq F value Pr(>F)
#> race          2      9    4.63   0.018  0.982
#> Residuals    97  25171  259.49
```

So in this case we cannot reject the null; There is no evidence that the pain level differs among the ethnicities.

8.2 Basic parametric Statistical tests for categorical data

If he data we want to compare is categorical instead of continuous we cannot use a t-test. However there are other tests that we can use.

8.2.1 Single sample

Say we want to test if the fraction of men in the trial is equal to 0.5 (so the proportion of men is the same as that of women). If R we can use the command `prop.test(x, n, p)`. Where `x` is the number of 'events' (that is women), `n` is the total number of individuals and `p` is the proportion we are comparing against.

```
# frst make a cross table
table(pain$sex)
#>
#>   Male Female
#>     70     30
prop.test(sum(pain$sex=='Male'), n=NROW(pain), p = 0.5)
#>
#>  1-sample proportions test with continuity correction
#>
#> data:  sum(pain$sex == "Male") out of NROW(pain), null probability 0.5
#> X-squared = 15.21, df = 1, p-value = 9.619e-05
#> alternative hypothesis: true p is not equal to 0.5
#> 95 percent confidence interval:
#>  0.5989396 0.7854574
#> sample estimates:
#>   p
#> 0.7
```

From the output we see that the fraction of men is actually 70% with a confidence interval from 60% to 79%. So it is clear that the proportion is significantly higher than 50%. We could also have obtained the result by using the function `prop.test` on the frequency table.

```
prop.test(table(pain$sex))
```

8.2.2 Two (or more) independent samples

We can also use the function `prop.test` to compare a proportion between two groups. This test is called the chi square test. Let's compare whether the proportion of men is the same in the two arms of the trial.

```
(tab <- table(pain$treatm, pain$sex))
#>
#>           Male Female
#>   Placebo   34     16
#>   Active    36     14
prop.test(tab)
#>
#>  2-sample test for equality of proportions with continuity
#>  correction
#>
#> data:  tab
#> X-squared = 0.047619, df = 1, p-value = 0.8273
#> alternative hypothesis: two.sided
#> 95 percent confidence interval:
#>  -0.2394625  0.1594625
#> sample estimates:
#> prop 1 prop 2
#>   0.68   0.72
```

From the output we see that the proportion of men does not differ a lot between the arms. We cannot reject the null hypothesis of no difference.

The same test can be used when we want to compare more than two groups.

```
(tab <- table(pain$race, pain$sex))
#>
#>             Male Female
#>   Asian       11      3
#>   Black       10      3
#>   Caucasian   49     24
prop.test(tab)
#> Warning in prop.test(tab): Chi-squared approximation may be incorrect
#>
#>  3-sample test for equality of proportions without continuity
#>  correction
#>
#> data:  tab
#> X-squared = 1.0742, df = 2, p-value = 0.5844
#> alternative hypothesis: two.sided
#> sample estimates:
#>    prop 1    prop 2    prop 3
#> 0.7857143 0.7692308 0.6712329
```

In this case we get a warning. This is because some of the (expected) counts in the cross table are very

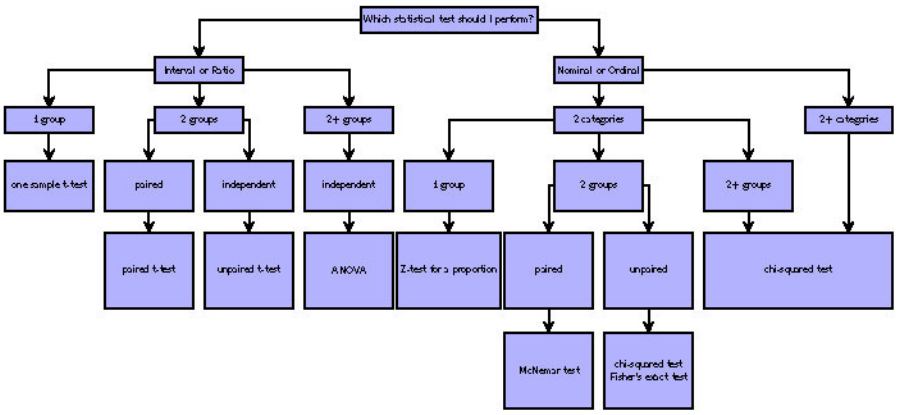


Figure 8.1: Flowchart

low. This makes the approximation of the distribution of the test statistic under the null that is used rather inaccurate.

8.2.3 Two dependent samples

When we want to test paired proportions we can use the McNemar test. Let's use this test to check if the proportion of people that have a pain score larger than 50 differs between baseline and follow-up.

```
pain$painbl50 <- pain$painbl > 50
pain$painfu50 <- pain$painfu > 50

(tab <- table(pain$painbl50, pain$painfu50))
#>
#>         FALSE TRUE
#>   FALSE    27   12
#>   TRUE     37   24
mcnemar.test(tab)
#>
#>  McNemar's Chi-squared test with continuity correction
#>
#> data:  tab
#> McNemar's chi-squared = 11.755, df = 1, p-value = 0.0006068
```

The McNemar test uses only the discordant pairs (i.e. the combinations were the outcome at baseline is the same as at follow-up are not used). The question is whether there are more people that have a high pain score at baseline and not at follow up than the other way around. Here this seems to be the case and we can reject the null that this proportion is the same.