R (BGU course)

Jonathan D. Rosenblatt

2018-06-09

Contents

Chapter 1

Preface

These notes are based on my R-Course, at the department of Industrial Engineering and Management, Ben-Gurion University.

1.1 Notation Conventions

In this text we use the following conventions: Lower case x may be a vector or a scalar, random of fixed, as implied by the context. Upper case A will stand for matrices. Equality $=$ is an equality, and $:=$ is a definition. Norm functions are denoted with $\|x\|$ for vector norms, and $\|A\|$ for matrix norms. The type of norm is indicated in the subscript; e.g. $\|x\|_2$ for the Euclidean (l_2) norm. Tag, x' is a transpose. The distribution of a random vector is $\sim$.

Chapter 2

Introduction

2.1 What is R?

R was not designed to be a bona-fide programming language. It is an evolution of the S language, developed at Bell labs (later Lucent) as a wrapper for the endless collection of statistical libraries they wrote in Fortran.

As of 2011, half of R's libraries are actually written in C.

For more on the history of R see AT&T's site, John Chamber's talk at UserR! 2014 or the Introduction to the excellent Venables and Ripley (2013).

2.2 The R Ecosystem

A large part of R's success is due to the ease in which a user, or a firm, can augment it. This led to a large community of users, developers, and protagonists. Some of the most important parts of R's ecosystem include:

- CRAN: a repository for R packages, mirrored worldwide.
- R-help: an immensely active mailing list. Noways being replaced by StackExchange meta-site. Look for the R tags in the StackOverflow and CrossValidated sites.
- Task Views: part of CRAN that collects packages per topic.
- Bioconductor: A CRAN-like repository dedicated to the life sciences.
- Neuroconductor: A CRAN-like repository dedicated to neuroscience, and neuroimaging.
- Books: An insane amount of books written on the language. Some are free, some are not.
- The Israeli-R-user-group: just like the name suggests.
- Commercial R: being open source and lacking support may seem like a problem that would prohibit R from being adopted for commercial applications. This void is filled by several very successful commercial versions such as Microsoft R, with its accompanying CRAN equivalent called MRAN, Tibco's Spotfire, and others.
- RStudio: since its earliest days R came equipped with a minimal text editor. It later received plugins for major integrated development environments (IDEs) such as Eclipse, WinEdit and even VisualStudio. None of these, however, had the impact of the RStudio IDE. Written completely in JavaScript, the RStudio IDE allows the seamless integration of cutting edge web-design technologies, remote access, and other killer features, making it today's most popular IDE for R.

2.3 Bibliographic Notes

2.4 Practice Yourself

Chapter 3

R Basics

We now start with the basics of R. If you have any experience at all with R, you can probably skip this section.

First, make sure you work with the RStudio IDE. Some useful pointers for this IDE include:

- Ctrl+Return(Enter) to run lines from editor.
- Alt+Shift+k for RStudio keyboard shortcuts.
- Ctrl+r to browse the command history.
- Alt+Shift+j to navigate between code sections
- tab for auto-completion
- Ctrl+1 to skip to editor.
- Ctrl+2 to skip to console.
- Ctrl+8 to skip to the environment list.
- Code Folding:
  - Alt+l collapse chunk.
  - Alt+Shift+l unfold chunk.
  - Alt+o collapse all.
  - Alt+Shift+o unfold all.
- Alt+"-" for the assignment operator `<-`.

3.0.1 Other IDEs

Currently, I recommend RStudio, but here are some other IDEs:

1. Jupyter Lab: a very promising IDE, originally designed for Python, that also supports R. At the time of writing, it seems that RStudio is more convenient for R, but it is definetly an IDE to follow closely. See Max Woolf's review.
2. Eclipse: If you are a Java programmer, you are probably familiar with Eclipse, which does have an R plugin: StatEt.
3. Emacs: If you are an Emacs fan, you can find an R plugin: ESS.
4. Vim: Vim-R.
5. Visual Studio also supports R. If you need R for commercial purposes, it may be worthwhile trying Microsoft's R, instead of the usual R. See here for installation instructions.

3.1 File types

The file types you need to know when using R are the following:

- **.R**: An ASCII text file containing R scripts only.

- **.Rmd**: An ASCII text file. If opened in RStudio can be run as an R-Notebook or compiled using knitr, bookdown, etc.

3.2 Simple calculator

R can be used as a simple calculator. Create a new R Notebook (.Rmd file) within RStudio using File-> New -> R Notebook, and run the following commands.

```
10+5
```

```
## [1] 15
```

```
70*81
```

```
## [1] 5670
```

```
2**4
```

```
## [1] 16
```

```
2^4
```

```
## [1] 16
```

```
log(10)
```

```
## [1] 2.302585
```

```
log(16, 2)
```

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

```
log(1000, 10)
```

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

3.3 Probability calculator

R can be used as a probability calculator. You probably wish you knew this when you did your Intro To Probability classes.

The Binomial distribution function:

```
dbinom(x=3, size=10, prob=0.5)  # Compute P(X=3) for X~B(n=10, p=0.5)
```

```
## [1] 0.1171875
```

Notice that arguments do not need to be named explicitly

```
dbinom(3, 10, 0.5)
```

```
## [1] 0.1171875
```

The Binomial cumulative distribution function (CDF):

```
pbinom(q=3, size=10, prob=0.5) # Compute P(X<=3) for X~B(n=10, p=0.5)
```

```
## [1] 0.171875
```

The Binomial quantile function:

```
qbinom(p=0.1718, size=10, prob=0.5) # For X~B(n=10, p=0.5) returns k such that P(X<=k)=0.1718
```

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

Generate random variables:

```
rbinom(n=10, size=10, prob=0.5)
```

```
##  [1] 5 7 4 7 7 6 6 3 4 4
```

R has many built-in distributions. Their names may change, but the prefixes do not:

- **d** prefix for the *distribution* function.
- **p** prefix for the *cummulative distribution* function (CDF).
- **q** prefix for the *quantile* function (i.e., the inverse CDF).
- **r** prefix to generate random samples.

Demonstrating this idea, using the CDF of several popular distributions:

- `pbinom()` for the Binomial CDF.
- `ppois()` for the Poisson CDF.
- `pnorm()` for the Gaussian CDF.
- `pexp()` for the Exponential CDF.

For more information see `?distributions`.

3.4 Getting Help

One of the most important parts of working with a language, is to know where to find help. R has several in-line facilities, besides the various help resources in the R ecosystem.

Get help for a particular function.

```
?dbinom
help(dbinom)
```

If you don't know the name of the function you are looking for, search local help files for a particular string:

```
??binomial
help.search('dbinom')
```

Or load a menu where you can navigate local help in a web-based fashion:

```
help.start()
```

3.5 Variable Asignment

Assignment of some output into an object named "x":

```
x = rbinom(n=10, size=10, prob=0.5) # Works. Bad style.
x <- rbinom(n=10, size=10, prob=0.5)
```

If you are familiar with other programming languages you may prefer the `=` assignment rather than the `<-` assignment. We recommend you make the effort to change your preferences. This is because thinking with `<-` helps to read your code, distinguishes between assignments and function arguments: think of `function(argument=value)` versus `function(argument<-value)`. It also helps understand special assignment operators such as `<<-` and `->`.
Remark. **Style**: We do not discuss style guidelines in this text, but merely remind the reader that good style is extremely important. When you write code, think of other readers, but also think of future self. See Hadley's style guide for more.

To print the contents of an object just type its name

```
x
```

```
##  [1] 6 3 4 5 2 5 7 4 5 5
```

which is an implicit call to

```
print(x)
```

```
##  [1] 6 3 4 5 2 5 7 4 5 5
```

Alternatively, you can assign and print simultaneously using parenthesis.

```
(x <- rbinom(n=10, size=10, prob=0.5))  # Assign and print.
```

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

Operate on the object

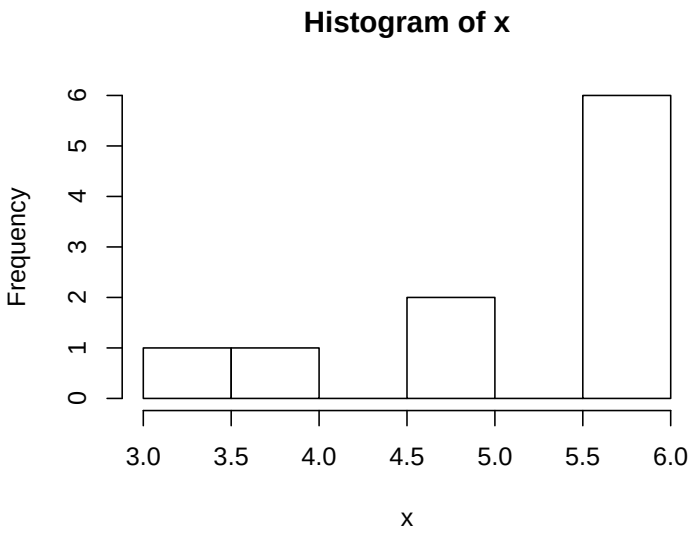
```
mean(x)  # compute mean
```

```
## [1] 5.3
```

```
var(x)  # compute variance
```

```
## [1] 1.122222
```

```
hist(x) # plot histogram
```



R saves every object you create in RAM[1]. The collection of all such objects is the **workspace** which you can inspect with

```
ls()
```

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

or with Ctrl+8 in RStudio.

If you lost your object, you can use `ls` with a text pattern to search for it

```
ls(pattern='x')
```

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

To remove objects from the workspace:

```
rm(x) # remove variable
ls() # verify
```

```
## character(0)
```

You may think that if an object is removed then its memory is freed. This is almost true, and depends on a negotiation mechanism between R and the operating system. R's memory management is discussed in Chapter 13.

[1] S and S-Plus used to save objects on disk. Working from RAM has advantages and disadvantages. More on this in Chapter 13.

3.6 Missing

Unlike typically programming, when working with real life data, you may have **missing** values: measurements that were simply not recorded/stored/etc. *R* has rather sophisticated mechanisms to deal with missing values. It distinguishes between the following types:

1. `NA`: Not Available entries.
2. `NaN`: Not a number.

R tries to defend the analyst, and return an error, or `NA` when the presence of missing values invalidates the calculation:

```
missing.example <- c(10,11,12,NA)
mean(missing.example)
```

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

Most functions will typically have an inner mechanism to deal with these. In the `mean` function, there is an `na.rm` argument, telling *R* how to Remove `NA`s.

```
mean(missing.example, na.rm = TRUE)
```

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

A more general mechanism is removing these manually:

```
clean.example <- na.omit(missing.example)
mean(clean.example)
```

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

3.7 Piping

Because R originates in Unix and Linux environments, it inherits much of its flavor. Piping is an idea taken from the Linux shell which allows to use the output of one expression as the input to another. Piping thus makes code easier to read and write.
Remark. Volleyball fans may be confused with the idea of spiking a ball from the 3-meter line, also called piping. So: (a) These are very different things. (b) If you can pipe, ASA-BGU is looking for you!

Prerequisites:

```
library(magrittr) # load the piping functions
x <- rbinom(n=1000, size=10, prob=0.5) # generate some toy data
```

Examples

```
x %>% var() # Instead of var(x)
x %>% hist()  # Instead of hist(x)
x %>% mean() %>% round(2) %>% add(10)
```

The next example[2] demonstrates the benefits of piping. The next two chunks of code do the same thing. Try parsing them in your mind:

```
# Functional (onion) style
car_data <-
  transform(aggregate(. ~ cyl,
                      data = subset(mtcars, hp > 100),
                      FUN = function(x) round(mean(x, 2))),
            kpl = mpg*0.4251)
```

```
# Piping (magrittr) style
car_data <-
```

[2] Taken from http://cran.r-project.org/web/packages/magrittr/vignettes/magrittr.html

```
mtcars %>%
subset(hp > 100) %>%
aggregate(. ~ cyl, data = ., FUN = . %>% mean %>% round(2)) %>%
transform(kpl = mpg %>% multiply_by(0.4251)) %>%
print
```

Tip: RStudio has a keyboard shortcut for the %>% operator. Try Ctrl+Shift+m.

3.8 Vector Creation and Manipulation

The most basic building block in R is the **vector**. We will now see how to create them, and access their elements (i.e. subsetting). Here are three ways to create the same arbitrary vector:

```
c(10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21) # manually
10:21 # the `:` operator
seq(from=10, to=21, by=1) # the seq() function
```

Let's assign it to the object named "x":

```
x <- c(10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21)
```

Operations usually work element-wise:

```
x+2
```

```
##  [1] 12 13 14 15 16 17 18 19 20 21 22 23
```

```
x*2
```

```
##  [1] 20 22 24 26 28 30 32 34 36 38 40 42
```

```
x^2
```

```
##  [1] 100 121 144 169 196 225 256 289 324 361 400 441
```

```
sqrt(x)
```

```
##  [1] 3.162278 3.316625 3.464102 3.605551 3.741657 3.872983 4.000000
##  [8] 4.123106 4.242641 4.358899 4.472136 4.582576
```

```
log(x)
```

```
##  [1] 2.302585 2.397895 2.484907 2.564949 2.639057 2.708050 2.772589
##  [8] 2.833213 2.890372 2.944439 2.995732 3.044522
```

3.9 Search Paths and Packages

R can be easily extended with packages, which are merely a set of documented functions, which can be loaded or unloaded conveniently. Let's look at the function `read.csv`. We can see its contents by calling it without arguments:

```
read.csv
```

```
## function (file, header = TRUE, sep = ",", quote = "\"", dec = ".",
##     fill = TRUE, comment.char = "", ...)
## read.table(file = file, header = header, sep = sep, quote = quote,
##     dec = dec, fill = fill, comment.char = comment.char, ...)
## <bytecode: 0x9c785f8>
## <environment: namespace:utils>
```

Never mind what the function does. Note the `environment: namespace:utils` line at the end. It tells us that this function is part of the **utils** package. We did not need to know this because it is loaded by default. Here are some packages that I have currently loaded:

```
head(search())
```

```
## [1] ".GlobalEnv"        "package:bindrcpp" "package:ellipse"
## [4] "package:bookdown" "package:doSNOW"   "package:snow"
```

Other packages can be loaded via the `library` function, or downloaded from the internet using the `install.packages` function before loading with `library`. R's package import mechanism is quite powerful, and is one of the reasons for R's success.

If anyone can write a package, can packages be trusted? Well, R is open-source. You can always look at the source code of each function and verify by yourself. Obviously, this is impossible to do with all the packages out there. Luckily, there some packages have a "seal of quality". Packages that ship with R, thus recommended by the R-core team, can be considered no less safe than any commercial software (SAS, SPSS, Stata, …). These packages also recieved an FDA approval (but make sure to read this if you are conducting clinical trials).

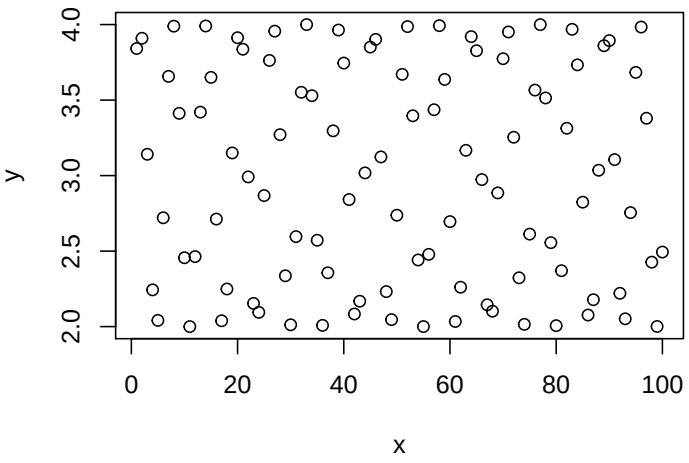
Some "R-authorities" curate lists of recommended packages. These can also be considered safe. See for example Rstudio's list. Other useful curated lists, possibly less authotirative, include Qin Wenfeng, ComputerWorld, yhat.

Finally, despite the great efforts of the CRAN team, in-depth quality control of all available packages is an impossible task. Newly released packages, from non-authoritative authors, should be dealt with caution.

3.10 Simple Plotting

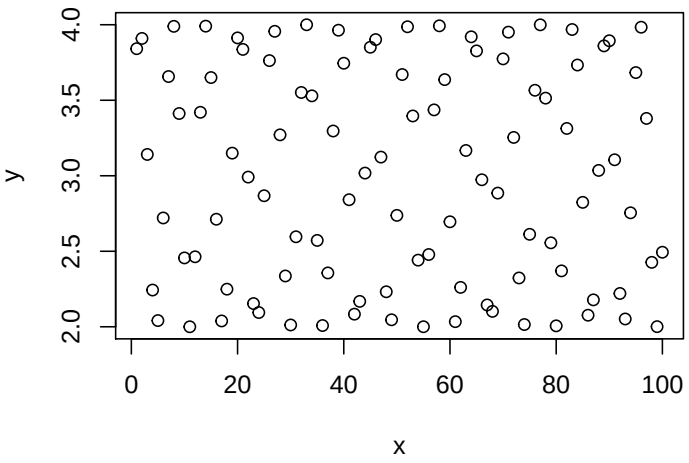
R has many plotting facilities as we will further detail in the Plotting Chapter 10. We start with the simplest facilities, namely, the `plot` function from the **graphics** package, which is loaded by default.

```
x<- 1:100
y<- 3+sin(x)
plot(x = x, y = y) # x,y syntax
```



Given an `x` argument and a `y` argument, `plot` tries to present a scatter plot. We call this the `x,y` syntax. R has another unique syntax to state functional relations. We call `y~x` the "tilde" syntax, which originates in works of Wilkinson and Rogers (1973) and was adopted in the early days of S.

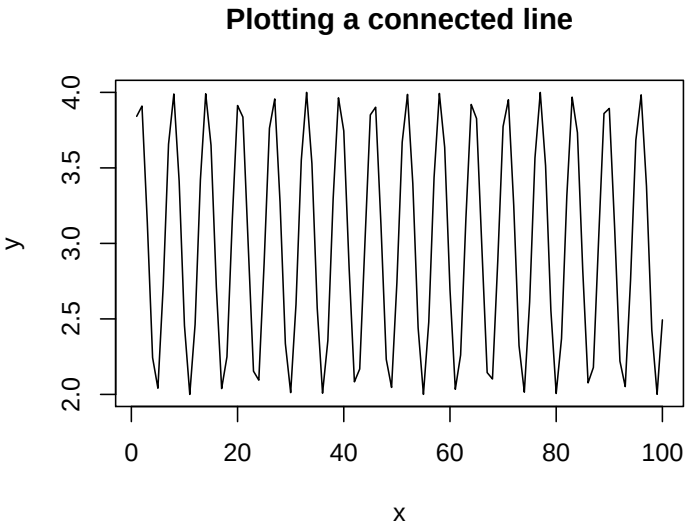
```
plot(y ~ x) # y~x syntax
```



The syntax `y~x` is read as "y is a function of x". We will prefer the `y~x` syntax over the `x,y` syntax since it is easier to read, and will be very useful when we discuss more complicated models.

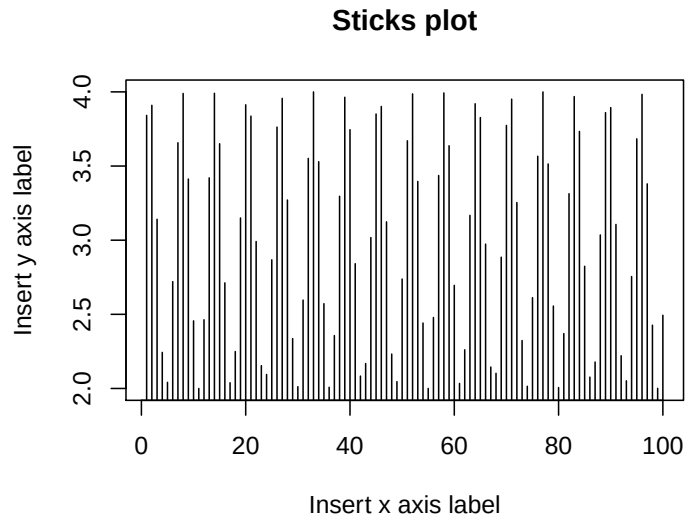
Here are some arguments that control the plot's appearance. We use `type` to control the plot type, `main` to control the main title.

```
plot(y~x, type='l', main='Plotting a connected line')
```



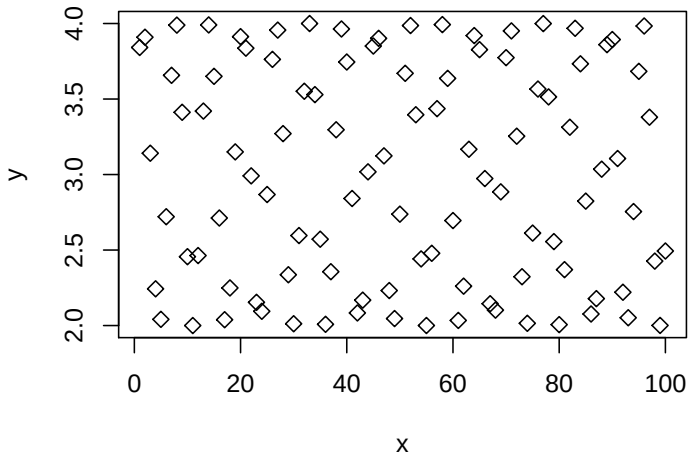
We use `xlab` for the x-axis label, `ylab` for the y-axis.

```
plot(y~x, type='h', main='Sticks plot', xlab='Insert x axis label', ylab='Insert y axis label')
```



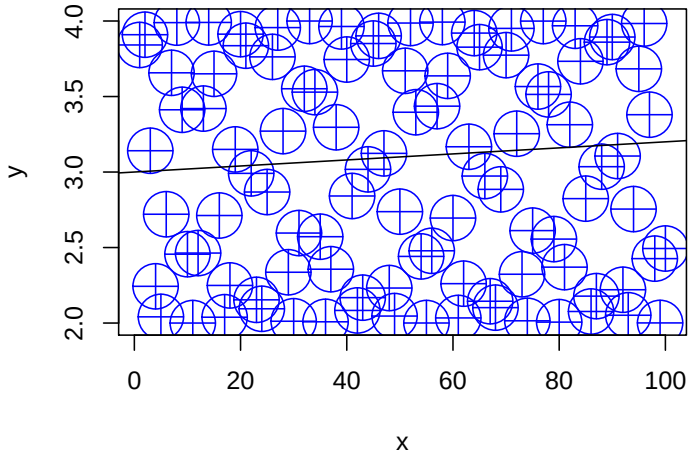
We use `pch` to control the point type.

```
plot(y~x, pch=5) # Point type with pcf
```



We use `col` to control the color, `cex` for the point size, and `abline` to add a straight line.

```
plot(y~x, pch=10, type='p', col='blue', cex=4)
abline(3, 0.002)
```



For more plotting options run these

```
example(plot)
example(points)
?plot
help(package='graphics')
```

When your plotting gets serious, go to Chapter 10.

3.11 Object Types

We already saw that the basic building block of R objects is the vector. Vectors can be of the following types:

- **character** Where each element is a string, i.e., a sequence of alphanumeric symbols.
- **numeric** Where each element is a real number in double precision floating point format.
- **integer** Where each element is an integer.
- **logical** Where each element is either TRUE, FALSE, or NA[3]
- **complex** Where each element is a complex number.
- **list** Where each element is an arbitrary R object.

[3] R uses a **three** valued logic where a missing value (NA) is neither TRUE, nor FALSE.

- **factor** Factors are not actually vector objects, but they feel like such. They are used to encode any finite set of values. SPSS users may think of these as values with built-in labels. Factors will be very useful when fitting models because they include information on contrasts, which can be thought as built-in instruction for level encoding.

Vectors can be combined into larger objects. A `matrix` can be thought of as the binding of several vectors of the same type. In reality, a matrix is merely a vector with a dimension attribute, that tells R to read it as a matrix and not a vector.

If vectors of different types (but same length) are binded, we get a `data.frame` which is the most fundamental object in R for data analysis. Data frames are brilliant, but a lot has been learned since their invention. They have thus been extended in recent years with the `tbl` class, pronounced [Tibble] (https://cran.r-project.org/web/packages/tibble/vignettes/tibble.html), and the `data.table` class.
The latter is discussed in Chapter 4, and is strongly recommended.

3.12 Data Frames

Creating a simple data frame:

```
x<- 1:10
y<- 3 + sin(x)
frame1 <- data.frame(x=x, sin=y)
```

Let's inspect our data frame:

```
head(frame1)
```

```
##   x      sin
## 1 1 3.841471
## 2 2 3.909297
## 3 3 3.141120
## 4 4 2.243198
## 5 5 2.041076
## 6 6 2.720585
```

Now using the RStudio Excel-like viewer:

```
frame1 %>% View()
```

We highly advise against editing the data this way since there will be no documentation of the changes you made. Always transform your data using scripts, so that everything is documented.

Verifying this is a data frame:

```
class(frame1) # the object is of type data.frame
```

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

Check the dimension of the data

```
dim(frame1)
```

```
## [1] 10  2
```

Note that checking the dimension of a vector is different than checking the dimension of a data frame.

```
length(x)
```

```
## [1] 10
```

The length of a `data.frame` is merely the number of columns.

```
length(frame1)
```

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

3.13 Exctraction

R provides many ways to subset and extract elements from vectors and other objects. The basics are fairly simple, but not paying attention to the "personality" of each extraction mechanism may cause you a lot of headache.

For starters, extraction is done with the `[` operator. The operator can take vectors of many types.

Extracting element with by integer index:

```
frame1[1, 2]  # exctract the element in the 1st row and 2nd column.
```

```
## [1] 3.841471
```

Extract **column** by index:

```
frame1[,1]
```

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

Extract column by name:

```
frame1[, 'sin']
```

```
##  [1] 3.841471 3.909297 3.141120 2.243198 2.041076 2.720585 3.656987
##  [8] 3.989358 3.412118 2.455979
```

As a general rule, extraction with `[` will conserve the class of the parent object. There are, however, exceptions. Notice the extraction mechanism and the class of the output in the following examples.

```
class(frame1[, 'sin'])  # extracts a column vector
```

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

```
class(frame1['sin'])  # extracts a data frame
```

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

```
class(frame1[,1:2])  # extracts a data frame
```

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

```
class(frame1[2])  # extracts a data frame
```

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

```
class(frame1[2, ])  # extract a data frame
```

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

```
class(frame1$sin)  # extracts a column vector
```

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

The `subset()` function does the same

```
subset(frame1, select=sin)
subset(frame1, select=2)
subset(frame1, select= c(2,0))
```

If you want to force the stripping of the class attribute when extracting, try the `[[` mechanism instead of `[`.

```
a <- frame1[1] # [ extraction
b <- frame1[[1]] # [[ extraction
class(a)==class(b) # objects have differing classes
```

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

```
a==b # objects are element-wise identical
```

```
##          x
##  [1,] TRUE
##  [2,] TRUE
##  [3,] TRUE
##  [4,] TRUE
##  [5,] TRUE
##  [6,] TRUE
##  [7,] TRUE
##  [8,] TRUE
##  [9,] TRUE
## [10,] TRUE
```

The different types of output classes cause different behaviors. Compare the behavior of `[` on seemingly identical objects.

```
frame1[1][1]
```

```
##     x
## 1   1
## 2   2
## 3   3
## 4   4
## 5   5
## 6   6
## 7   7
## 8   8
## 9   9
## 10 10
```

```
frame1[[1]][1]
```

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

If you want to learn more about subsetting see Hadley's guide.

3.14 Augmentations of the data.frame class

As previously mentioned, the `data.frame` class has been extended in recent years. The best known extensions are the `data.table` and the `tbl`. For beginners, it is important to know R's basics, so we keep focusing on data frames. For more advanced users, I recommend learning the (amazing) `data.table` syntax.

3.15 Data Import and Export

For any practical purpose, you will not be generating your data manually. R comes with many importing and exporting mechanisms which we now present. If, however, you do a lot of data "munging", make sure to see Hadley-verse Chapter **??**. If you work with MASSIVE data sets, read the Memory Efficiency Chapter 13.

3.15.1 Import from WEB

The `read.table` function is the main importing workhorse. It can import directly from the web.

```
URL <- 'http://statweb.stanford.edu/~tibs/ElemStatLearn/datasets/bone.data'
tirgul1 <- read.table(URL)
```

Always look at the imported result!

```
head(tirgul1)
```

```
##      V1    V2     V3          V4
## 1 idnum   age gender      spnbmd
## 2     1  11.7   male  0.01808067
## 3     1  12.7   male  0.06010929
## 4     1 13.75   male 0.005857545
## 5     2 13.25   male  0.01026393
## 6     2  14.3   male   0.2105263
```

Ohh dear. `read.,table` tried to guess the structure of the input, but failed to recognize the header row. Set it manually with `header=TRUE`:

```
tirgul1 <- read.table('data/bone.data', header = TRUE)
head(tirgul1)
```

3.15.2 Export as CSV

Let's write a simple file so that we have something to import

```
head(airquality) #  examine the data to export
```

```
##   Ozone Solar.R Wind Temp Month Day
## 1    41     190  7.4   67     5   1
## 2    36     118  8.0   72     5   2
## 3    12     149 12.6   74     5   3
## 4    18     313 11.5   62     5   4
## 5    NA      NA 14.3   56     5   5
## 6    28      NA 14.9   66     5   6
```

```
temp.file.name <- tempfile() # get some arbitrary file name
write.csv(x = airquality, file = temp.file.name) # export
```

Now let's import the exported file. Being a .csv file, I can use `read.csv` instead of `read.table`.

```
my.data<- read.csv(file=temp.file.name) # import
head(my.data) # verify import
```

```
##   X Ozone Solar.R Wind Temp Month Day
## 1 1    41     190  7.4   67     5   1
## 2 2    36     118  8.0   72     5   2
## 3 3    12     149 12.6   74     5   3
## 4 4    18     313 11.5   62     5   4
## 5 5    NA      NA 14.3   56     5   5
## 6 6    28      NA 14.9   66     5   6
```

Remark. Windows users may need to use "\" instead of "/".

3.15.3 Export non-CSV files

You can export your R objects in endlessly many ways: If instead of the comma delimiter in .csv you want other column delimiters, look into `?write.table`. If you are exporting only for R users, you can consider exporting as binary objects with `saveRDS`, `feather::write_feather`, or `fst::write.fst`. See (http://www.fstpackage.org/) for a comparison.

3.15.4 Reading From Text Files

Some general notes on importing text files via the `read.table` function. But first, we need to know what is the active directory. Here is how to get and set R's active directory:

```
getwd() #What is the working directory?
setwd() #Setting the working directory in Linux
```

We can now call the `read.table` function to import text files. If you care about your sanity, see `?read.table` before starting imports. Some notable properties of the function:

- `read.table` will try to guess column separators (tab, comma, etc.)
- `read.table` will try to guess if a header row is present.
- `read.table` will convert character vectors to factors unless told not to using the `stringsAsFactors=FALSE` argument.
- The output of `read.table` needs to be explicitly assigned to an object for it to be saved.

3.15.5 Writing Data to Text Files

The function `write.table` is the exporting counterpart of `read.table`.

3.15.6 .XLS(X) files

Strongly recommended to convert to .csv in Excel, and then import as csv. If you still insist see the **xlsx** package.

3.15.7 Massive files

The above importing and exporting mechanisms were not designed for massive files. See the section on the **data.table** package (4), Sparse Representation (12), and Out-of-Ram Algorithms (13) for more on working with massive data files.

3.15.8 Databases

R does not need to read from text files; it can read directly from a database. This is very useful since it allows the filtering, selecting and joining operations to rely on the database's optimized algorithms. Then again, if you will only be analyzing your data with R, you are probably better of by working from a file, without the databases' overhead. See Chapter 13 for more on this matter.

3.16 Functions

One of the most basic building blocks of programming is the ability of writing your own functions. A function in R, like everything else, is an object accessible using its name. We first define a simple function that sums its two arguments

```
my.sum <- function(x,y) {
  return(x+y)
}
my.sum(10,2)
```

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

From this example you may notice that:

- The function `function` tells R to construct a function object.
- Unlike some programming languages, a period (`.`) is allowed as part of an object's name.
- The arguments of the `function`, i.e. `(x,y)`, need to be named but we are not required to specify their class. This makes writing functions very easy, but it is also the source of many bugs, and slowness of R compared to type declaring languages (C, Fortran,Java,...).

- A typical R function does not change objects[4] but rather creates new ones. To save the output of `my.sum` we will need to assign it using the `<-` operator.

Here is a (slightly) more advanced function:

```
my.sum.2 <- function(x, y , absolute=FALSE) {
  if(absolute==TRUE) {
    result <- abs(x+y)
  }
  else{
    result <- x+y
  }
  result
}
my.sum.2(-10,2,TRUE)
```

```
## [1] 8
```

Things to note:

- `if(condition){expression1} else{expression2}` does just what the name suggests.
- The function will output its last evaluated expression. You don't need to use the `return` function explicitly.
- Using `absolute=FALSE` sets the default value of `absolute` to `FALSE`. This is overridden if `absolute` is stated explicitly in the function call.

An important behavior of R is the *scoping rules*. This refers to the way R seeks for variables used in functions. As a rule of thumb, R will first look for variables inside the function and if not found, will search for the variable values in outer environments[5]. Think of the next example.

```
a <- 1
b <- 2
x <- 3
scoping <- function(a,b){
  a+b+x
}
scoping(10,11)
```

```
## [1] 24
```

3.17 Looping

The real power of scripting is when repeated operations are done by iteration. R supports the usual `for`, `while`, and `repated` loops. Here is an embarrassingly simple example

```
for (i in 1:5){
    print(i)
    }
```

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

A slightly more advanced example, is vector multiplication

[4] This is a classical *functional programming* paradigm. If you want an object oriented flavor of R programming, see Hadley's Advanced R book.

[5] More formally, this is called Lexical Scoping.

```
result <- 0
n <- 1e3
x <- 1:n
y <- (1:n)/n
for(i in 1:n){
  result <- result+ x[i]*y[i]
}
```

Remark. **Vector Operations**: You should NEVER write your own vector and matrix products like in the previous example. Only use existing facilities such as `%*%`, `sum()`, etc.
Remark. **Parallel Operations**: If you already know that you will be needing to parallelize your work, get used to working with `foreach` loops in the **foreach** package, rather then regular `for` loops.

3.18 Apply

For applying the same function to a set of elements, there is no need to write an explicit loop. This is such en elementary operation that every programming language will provide some facility to **apply**, or **map** the function to all elements of a set. R provides several facilities to perform this. The most basic of which is `lapply` which applies a function over all elements of a list, and return a list of outputs:

```
the.list <- list(1,'a',mean) # a list of 3 elements from different calsses
lapply(X = the.list, FUN = class) # apply the function `class` to each elements
```

```
## [[1]]
## [1] "numeric"
##
## [[2]]
## [1] "character"
##
## [[3]]
## [1] "standardGeneric"
## attr(,"package")
## [1] "methods"
```

```
sapply(X = the.list, FUN = class) # lapply with cleaned output
```

```
## [1] "numeric"         "character"       "standardGeneric"
```

R provides many variations on `lapply` to facilitate programming. Here is a partial list:

- `sapply`: The same as `lapply` but tries to arrange output in a vector or matrix, and not an unstructured list.
- `vapply`: A safer version of `sapply`, where the output class is pre-specified.
- `apply`: For applying over the rows or columns of matrices.
- `mapply`: For applying functions with more than a single input.
- `tapply`: For splitting vectors and applying functions on subsets.
- `rapply`: A recursive version of `lapply`.
- `eapply`: Like `lapply`, only operates on `environments` instead of lists.
- `Map`+`Reduce`: For a Common Lisp look and feel of `lapply`.
- `parallel::parLapply`: A parallel version of `lapply` from the package **parallel**.
- `parallel::parLBapply`: A parallel version of `lapply`, with load balancing from the package **parallel**.

3.19 Recursion

The R compiler is really not designed for recursion, and you will rarely need to do so.
See the RCpp Chapter **??** for linking C code, which is better suited for recursion. If you really insist to write recursions in R, make sure to use the `Recall` function, which, as the name suggests, recalls the function in which it is place. Here is a demonstration with the Fibonacci series.

```
fib<-function(n) {
    if (n < 2) fn<-1
    else fn <- Recall(n - 1) + Recall(n - 2)
    return(fn)
}
fib(5)
```

```
## [1] 8
```

3.20 Dates and Times

[TODO. In the meanwhile, see the **lubridate** pacakge].

3.21 Bibliographic Notes

There are endlessly many introductory texts on R. For a list of free resources see CrossValidated. I personally recommend the official introduction Venables et al. (2004), available online, or anything else Bill Venables writes.

For Importing and Exporting see (https://cran.r-project.org/doc/manuals/r-release/R-data.html). For working with databases see (https://rforanalytics.wordpress.com/useful-links-for-r/odbc-databases-for-r/).

For advanced R programming see Wickham (2014), available online, or anything else Hadley Wickham writes.

For a curated list of recommended packages see here.

3.22 Practice Yourself

1. Load the package **MASS**. That was easy. Now load **ggplot2**, after looking into `install.pacakges()`.
2. Save the numbers 1 to 1,000,000 (`1e6`) into an object named `object`.
3. Write a function that computes the mean of its input. Write a version that uses `sum()`, and another that uses a `for` loop and the summation `+`. Try checking which is faster using `system.time`. Is the difference considerable? Ask me about it in class.
4. Write a function that returns `TRUE` if a number is divisible by 13, `FALSE` if not, and a nice warning to the user if the input is not an integer number.
5. Apply the previous function to all the numbers in `object`. Try using a `for` loop, but also a mapping/apply function.
6. Make a matrix of random numbers using `A <- matrix(rnorm(40), ncol=10, nrow=4)`. Compute the mean of each columns. Do it using your own loop and then do the same with `lapply` or `apply`.
7. Make a data frame (`dataA`) with three columns, and 100 rows. The first column with 100 numbers generated from the $\mathcal{N}(10, 1)$ distribution, second column with samples from $Poiss(\lambda = 4)$. The third column contains only `1`.
   Make another data frame (`dataB`) with three columns and 100 rows. Now with $\mathcal{N}(10, 0.5^2)$, $Poiss(\lambda = 4)$ and `2`. Combine the two data frames into an object named `dataAB` with `rbind`. Make a scatter plot of `dataAB` where the x-axes is the first column, the y-axes is the second and define the shape of the points to be the third column.
8. In a sample generated of 1,000 observations from the $\mathcal{N}(10, 1)$ distribution:
   1. What is the proportion of samples smaller than 12.4 ?
   2. What is the 0.23 percentile of the sample?
9. Nothing like cleaning a dataset, to paractice your R basics. Have a look at RACHAEL TATMAN collected several datasets which BADLY need some cleansing.

Chapter 4

data.table

`data.table` is an excellent extension of the `data.frame` class. If used as a `data.frame` it will look and feel like a data frame. If, however, it is used with it's unique capabilities, it will prove faster and easier to manipulate.

Let's start with importing some freely available car sales data from Kaggle.

```
library(data.table)
library(magrittr)
auto <- fread('data/autos.csv')
```

```
View(auto)
```

```
dim(auto) #  Rows and columns
```

```
## [1] 371824     20
```

```
names(auto) # Variable names
```

```
##  [1] "dateCrawled"         "name"                "seller"
##  [4] "offerType"           "price"               "abtest"
##  [7] "vehicleType"         "yearOfRegistration"  "gearbox"
## [10] "powerPS"             "model"               "kilometer"
## [13] "monthOfRegistration" "fuelType"            "brand"
## [16] "notRepairedDamage"   "dateCreated"         "nrOfPictures"
## [19] "postalCode"          "lastSeen"
```

```
class(auto) # Object class
```

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

```
file.info('data/autos.csv') # File info on disk
```

```
##                    size isdir mode               mtime               ctime
## data/autos.csv 68439217 FALSE  664 2016-11-28 15:47:00 2018-06-08 15:15:48
##                              atime  uid  gid   uname  grname
## data/autos.csv 2018-06-09 20:42:35 1000 1000 johnros johnros
```

```
gdata::humanReadable(68439217)
```

```
## [1] "65.3 MiB"
```

```
object.size(auto) %>% print(units = 'auto') # File size in memory
```

```
## 97.9 Mb
```

Things to note:

- The import has been done with `fread` instead of `read.csv`. This is more efficient, and directly creates a `data.table` object.

- The import is very fast.
- The data after import is slightly larger than when stored on disk (in this case).

Let's start with verifying that it behaves like a `data.frame` when expected.

```
auto[,2] %>% head
```

```
##                                                    name
## 1:                                           Golf_3_1.6
## 2:                                  A5_Sportback_2.7_Tdi
## 3:                         Jeep_Grand_Cherokee_"Overland"
## 4:                                  GOLF_4_1_4__3T\xdcRER
## 5:                         Skoda_Fabia_1.4_TDI_PD_Classic
## 6: BMW_316i___e36_Limousine___Bastlerfahrzeug__Export
```

```
auto[[2]] %>% head
```

```
## [1] "Golf_3_1.6"
## [2] "A5_Sportback_2.7_Tdi"
## [3] "Jeep_Grand_Cherokee_\"Overland\""
## [4] "GOLF_4_1_4__3T\xdcRER"
## [5] "Skoda_Fabia_1.4_TDI_PD_Classic"
## [6] "BMW_316i___e36_Limousine___Bastlerfahrzeug__Export"
```

```
auto[1,2] %>% head
```

```
##          name
## 1: Golf_3_1.6
```

But notice the difference between `data.frame` and `data.table` when subsetting multiple rows. Uhh!

```
auto[1:3] %>% dim # data.table will exctract *rows*
```

```
## [1]  3 20
```

```
as.data.frame(auto)[1:3] %>% dim # data.frame will exctract *columns*
```

```
## [1] 371824      3
```

Just use columns (`,`) and be explicit regarding the dimension you are extracting…

Now let's do some `data.table` specific operations. The general syntax has the form `DT[i,j,by]`. SQL users may think of `i` as `WHERE`, `j` as `SELECT`, and `by` as `GROUP BY`. We don't need to name the arguments explicitly. Also, the `Tab` key will typically help you to fill in column names.

```
auto[,vehicleType,] %>% table # Exctract column and tabulate
```

```
## .
##                andere        bus     cabrio      coupe kleinwagen
##      37899       3362      30220      22914      19026      80098
##      kombi  limousine        suv
##      67626      95963      14716
```

```
auto[vehicleType=='coupe',,] %>% dim # Exctract rows
```

```
## [1] 19026    20
```

```
auto[,gearbox:model,] %>% head # exctract column range
```

```
##      gearbox powerPS model
## 1:   manuell       0  golf
## 2:   manuell     190
## 3: automatik     163 grand
## 4:   manuell      75  golf
## 5:   manuell      69 fabia
## 6:   manuell     102   3er
```

```
auto[,gearbox,] %>% table
```

```
## .
##           automatik   manuell
##     20223     77169    274432
```

```
auto[vehicleType=='coupe' & gearbox=='automatik',,] %>% dim # intersect conditions
```

```
## [1] 6008   20
```

```
auto[,table(vehicleType),] # uhh? why would this even work?!?
```

```
## vehicleType
##                andere        bus     cabrio      coupe kleinwagen
##      37899       3362      30220      22914      19026      80098
##      kombi  limousine        suv
##      67626      95963      14716
```

```
auto[, mean(price), by=vehicleType] # average price by car group
```

```
## Warning in gmean(price): Group 9 summed to more than type 'integer'
## can hold so the result has been coerced to 'numeric' automatically, for
## convenience.
```

```
##    vehicleType         V1
## 1:              20124.688
## 2:       coupe  25951.506
## 3:         suv  13252.392
## 4:  kleinwagen   5691.167
## 5:   limousine  11111.107
## 6:      cabrio  15072.998
## 7:         bus  10300.686
## 8:       kombi   7739.518
## 9:      andere 676327.100
```

The `.N` operator is very useful if you need to count the length of the result. Notice where I use it:

```
auto[.N-1,,] # will exctract the *last* row
```

```
##            dateCrawled                   name seller offerType price
## 1: 2016-03-20 19:41:08 VW_Golf_Kombi_1_9l_TDI privat   Angebot  3400
##    abtest vehicleType yearOfRegistration gearbox powerPS model kilometer
## 1:   test       kombi               2002 manuell     100  golf    150000
##    monthOfRegistration fuelType      brand notRepairedDamage
## 1:                   6   diesel volkswagen
##            dateCreated nrOfPictures postalCode            lastSeen
## 1: 2016-03-20 00:00:00            0      40764 2016-03-24 12:45:21
```

```
auto[,.N] # will count rows
```

```
## [1] 371824
```

```
auto[,.N, vehicleType] # will count rows by type
```

```
##    vehicleType     N
## 1:             37899
## 2:       coupe 19026
## 3:         suv 14716
## 4:  kleinwagen 80098
## 5:   limousine 95963
## 6:      cabrio 22914
## 7:         bus 30220
## 8:       kombi 67626
```

```
## 9:      andere  3362
```

You may concatenate results into a vector:

```
auto[,c(mean(price), mean(powerPS)),]
```

```
## [1] 17286.2996   115.5414
```

This `c()` syntax no longer behaves well if splitting:

```
auto[,c(mean(price), mean(powerPS)), by=vehicleType]
```

```
##     vehicleType           V1
##  1:              20124.68801
##  2:                 71.23249
##  3:       coupe  25951.50589
##  4:       coupe    172.97614
##  5:         suv  13252.39182
##  6:         suv    166.01903
##  7:  kleinwagen   5691.16738
##  8:  kleinwagen     68.75733
##  9:   limousine  11111.10661
## 10:   limousine    132.26936
## 11:      cabrio  15072.99782
## 12:      cabrio    145.17684
## 13:         bus  10300.68561
## 14:         bus    113.58137
## 15:       kombi   7739.51760
## 16:       kombi    136.40654
## 17:      andere 676327.09964
## 18:      andere    102.11154
```

Use a `list()` instead of `c()`, within `data.table` commands:

```
auto[,list(mean(price), mean(powerPS)), by=vehicleType]
```

```
## Warning in gmean(price): Group 9 summed to more than type 'integer'
## can hold so the result has been coerced to 'numeric' automatically, for
## convenience.
```

```
##    vehicleType         V1        V2
## 1:              20124.688  71.23249
## 2:       coupe  25951.506 172.97614
## 3:         suv  13252.392 166.01903
## 4:  kleinwagen   5691.167  68.75733
## 5:   limousine  11111.107 132.26936
## 6:      cabrio  15072.998 145.17684
## 7:         bus  10300.686 113.58137
## 8:       kombi   7739.518 136.40654
## 9:      andere 676327.100 102.11154
```

You can add names to your new variables:

```
auto[,list(Price=mean(price), Power=mean(powerPS)), by=vehicleType]
```

```
## Warning in gmean(price): Group 9 summed to more than type 'integer'
## can hold so the result has been coerced to 'numeric' automatically, for
## convenience.
```

```
##    vehicleType      Price     Power
## 1:              20124.688  71.23249
## 2:       coupe  25951.506 172.97614
## 3:         suv  13252.392 166.01903
```

```
## 4:  kleinwagen   5691.167  68.75733
## 5:   limousine  11111.107 132.26936
## 6:      cabrio  15072.998 145.17684
## 7:         bus  10300.686 113.58137
## 8:       kombi   7739.518 136.40654
## 9:      andere 676327.100 102.11154
```

You can use `.()` to replace the longer `list()` command:

```
auto[,.(Price=mean(price), Power=mean(powerPS)), by=vehicleType]
```

```
## Warning in gmean(price): Group 9 summed to more than type 'integer'
## can hold so the result has been coerced to 'numeric' automatically, for
## convenience.
```

```
##    vehicleType      Price     Power
## 1:              20124.688  71.23249
## 2:       coupe  25951.506 172.97614
## 3:         suv  13252.392 166.01903
## 4:  kleinwagen   5691.167  68.75733
## 5:   limousine  11111.107 132.26936
## 6:      cabrio  15072.998 145.17684
## 7:         bus  10300.686 113.58137
## 8:       kombi   7739.518 136.40654
## 9:      andere 676327.100 102.11154
```

And split by multiple variables:

```
auto[,.(Price=mean(price), Power=mean(powerPS)), by=.(vehicleType,fuelType)] %>% head
```

```
## Warning in gmean(price): Group 37 summed to more than type 'integer'
## can hold so the result has been coerced to 'numeric' automatically, for
## convenience.
```

```
##    vehicleType fuelType     Price     Power
## 1:              benzin 11820.443  70.14477
## 2:       coupe   diesel 51170.248 179.48704
## 3:         suv   diesel 15549.369 168.16115
## 4:  kleinwagen   benzin  5786.514  68.74309
## 5:  kleinwagen   diesel  4295.550  76.83666
## 6:   limousine   benzin  6974.360 127.87025
```

Compute with variables created on the fly:

```
auto[,sum(price<1e4),] # Count prices higher than 10,000
```

```
## [1] 310497
```

```
auto[,mean(price<1e4),] # Proportion of prices larger than 10,000
```

```
## [1] 0.8350644
```

```
auto[,.(Power=mean(powerPS)), by=.(PriceRange=price>1e4)]
```

```
##    PriceRange    Power
## 1:      FALSE 101.8838
## 2:       TRUE 185.9029
```

You may sort along one or more columns

```
auto[order(-price), price,] %>% head # Order along price. Descending
```

```
## [1] 2147483647   99999999   99999999   99999999   99999999   99999999
```

```
auto[order(price, -lastSeen), price,] %>% head# Order along price and last seen . Ascending and descending.
```

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

You may apply a function to ALL columns using a Subset of the Data using `.SD`

```
count.uniques <- function(x) length(unique(x))
auto[,lapply(.SD, count.uniques), vehicleType]
```

```
##    vehicleType dateCrawled  name seller offerType price abtest
## 1:                  36714 32891      1         2  1378      2
## 2:       coupe      18745 13182      1         2  1994      2
## 3:         suv      14549  9707      1         1  1667      2
## 4:  kleinwagen      75591 49302      2         2  1927      2
## 5:   limousine      89352 58581      2         1  2986      2
## 6:      cabrio      22497 13411      1         1  2014      2
## 7:         bus      29559 19651      1         2  1784      2
## 8:       kombi      64415 41976      2         1  2529      2
## 9:      andere       3352  3185      1         1   562      2
##    yearOfRegistration gearbox powerPS model kilometer monthOfRegistration
## 1:                101       3     374   244        13                  13
## 2:                 75       3     414   117        13                  13
## 3:                 73       3     342   122        13                  13
## 4:                 75       3     317   163        13                  13
## 5:                 83       3     506   210        13                  13
## 6:                 88       3     363    95        13                  13
## 7:                 65       3     251   106        13                  13
## 8:                 64       3     393   177        13                  13
## 9:                 81       3     230   162        13                  13
##    fuelType brand notRepairedDamage dateCreated nrOfPictures postalCode
## 1:        8    40                3          65            1       6304
## 2:        8    35                3          51            1       5159
## 3:        8    37                3          61            1       4932
## 4:        8    38                3          68            1       7343
## 5:        8    39                3          82            1       7513
## 6:        7    38                3          70            1       5524
## 7:        8    33                3          63            1       6112
## 8:        8    38                3          75            1       7337
## 9:        8    38                3          41            1       2220
##    lastSeen
## 1:    32813
## 2:    16568
## 3:    13367
## 4:    59354
## 5:    65813
## 6:    19125
## 7:    26094
## 8:    50668
## 9:     3294
```

Things to note:

- `.SD` is the data subset after splitting along the `by` argument.
- Recall that `lapply` applies the same function to all elements of a list. In this example, to all columns of `.SD`.

If you want to apply a function only to a subset of columns, use the `.SDcols` argument

```
auto[,lapply(.SD, count.uniques), by=vehicleType, .SDcols=price:gearbox]
```

```
##    vehicleType price abtest vehicleType yearOfRegistration gearbox
## 1:        1378      2           1                101       3
```

```
## 2:       coupe  1994      2          1                75       3
## 3:         suv  1667      2          1                73       3
## 4:  kleinwagen  1927      2          1                75       3
## 5:   limousine  2986      2          1                83       3
## 6:      cabrio  2014      2          1                88       3
## 7:         bus  1784      2          1                65       3
## 8:       kombi  2529      2          1                64       3
## 9:      andere   562      2          1                81       3
```

4.1 Make your own variables

It is very easy to compute new variables

```
auto[,log(price/powerPS),] %>% head # This makes no sense
```

```
## [1]      Inf 4.567632 4.096387 2.995732 3.954583 1.852000
```

And if you want to store the result in a new variable, use the `:=` operator

```
auto[,newVar:=log(price/powerPS),]
```

Or create multiple variables at once. The syntax `c("A","B"):=.(expression1,expression2)`is read "save the **list** of results from expression1 and expression2 using the **vector** of names A, and B".

```
auto[,c('newVar','newVar2'):=.(log(price/powerPS),price^2/powerPS),]
```

4.2 Join

data.table can be used for joining. A *join* is the operation of aligning two (or more) data frames/tables along some index. The index can be a single variable, or a combination thereof.

Here is a simple example of aligning age and gender from two different data tables:

```
DT1 <- data.table(Names=c("Alice","Bob"), Age=c(29,31))
DT2 <- data.table(Names=c("Alice","Bob","Carl"), Gender=c("F","M","M"))
setkey(DT1, Names)
setkey(DT2, Names)
DT1[DT2,,]
```

```
##    Names Age Gender
## 1: Alice  29      F
## 2:   Bob  31      M
## 3:  Carl  NA      M
```

```
DT2[DT1,,]
```

```
##    Names Gender Age
## 1: Alice      F  29
## 2:   Bob      M  31
```

Things to note:

- A join with `data.tables` is performed by indexing one `data.table` with another. Which is the outer and which is the inner will affect the result.
- The indexing variable needs to be set using the `setkey` function.

There are several types of joins:

- **Inner join**: Returns the rows along the intersection of keys, i.e., rows that appear in **all** data sets.
- **Outer join**: Returns the rows along the union of keys, i.e., rows that appear in **any** of the data sets.
- **Left join**: Returns the rows along the index of the "left" data set.

- **Right join**: Returns the rows along the index of the "right" data set.

Assuming `DT1` is the "left" data set, we see that `DT1[DT2,,]` is a right join, and `DT2[DT1,,]` is a left join. For an inner join use the `nomath=0` argument:

```
DT1[DT2,,,nomatch=0]
```

```
##    Names Age Gender
## 1: Alice  29      F
## 2:   Bob  31      M
```

```
DT2[DT1,,,nomatch=0]
```

```
##    Names Gender Age
## 1: Alice      F  29
## 2:   Bob      M  31
```

4.3 Reshaping data

Data sets (i.e. frames or tables) may arrive in a "wide" form or a "long" form. The difference is best illustrated with an example. The `ChickWeight` data encodes the weight of various chicks. It is "long" in that a variable encodes the time of measurement, making the data, well, simply long:

```
ChickWeight %>%  head
```

```
## Grouped Data: weight ~ Time | Chick
##   weight Time Chick Diet
## 1     42    0     1    1
## 2     51    2     1    1
## 3     59    4     1    1
## 4     64    6     1    1
## 5     76    8     1    1
## 6     93   10     1    1
```

The `mtcars` data encodes 10 characteristics of 32 types of automobiles. It is "wide" since the various characteristics are encoded in different variables, making the data, well, simply wide.

```
mtcars %>% head
```

```
##                    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
```

Most of *R*'s functions, with exceptions, will prefer data in the long format. There are thus various facilities to convert from one format to another. We will focus on the `melt` and `dcast` functions to convert from one format to another.

4.3.1 Wide to long

`melt` will convert from wide to long.

```
dimnames(mtcars)
```

```
## [[1]]
##  [1] "Mazda RX4"           "Mazda RX4 Wag"       "Datsun 710"
##  [4] "Hornet 4 Drive"      "Hornet Sportabout"   "Valiant"
##  [7] "Duster 360"          "Merc 240D"           "Merc 230"
## [10] "Merc 280"            "Merc 280C"           "Merc 450SE"
```

```
## [13] "Merc 450SL"          "Merc 450SLC"         "Cadillac Fleetwood"
## [16] "Lincoln Continental" "Chrysler Imperial"   "Fiat 128"
## [19] "Honda Civic"         "Toyota Corolla"      "Toyota Corona"
## [22] "Dodge Challenger"    "AMC Javelin"         "Camaro Z28"
## [25] "Pontiac Firebird"    "Fiat X1-9"           "Porsche 914-2"
## [28] "Lotus Europa"        "Ford Pantera L"      "Ferrari Dino"
## [31] "Maserati Bora"       "Volvo 142E"
##
## [[2]]
##  [1] "mpg"  "cyl"  "disp" "hp"   "drat" "wt"   "qsec" "vs"   "am"   "gear"
## [11] "carb"
```

```
mtcars$type <- rownames(mtcars)
melt(mtcars, id.vars=c("type")) %>% head
```

```
##                type variable value
## 1         Mazda RX4      mpg  21.0
## 2     Mazda RX4 Wag      mpg  21.0
## 3        Datsun 710      mpg  22.8
## 4    Hornet 4 Drive      mpg  21.4
## 5 Hornet Sportabout      mpg  18.7
## 6           Valiant      mpg  18.1
```

Things to note:

- The car type was originally encoded in the rows' names, and not as a variable. We thus created an explicit variable with the cars' type using the `rownames` function.
- The `id.vars` of the `melt` function names the variables that will be used as identifiers. All other variables are assumed to be measurements. These can have been specified using their index instead of their name.
- If not all variables are measurements, we could have names measurement variables explicitly using the `measure.vars` argument of the `melt` function. These can have been specified using their index instead of their name.
- By default, the molten columns are automatically named `variable` and `value`.

We can replace the automatic namings using `variable.name` and `value.name`:

```
melt(mtcars, id.vars=c("type"), variable.name="Charachteristic", value.name="Measurement") %>% head
```

```
##                type Charachteristic Measurement
## 1         Mazda RX4             mpg        21.0
## 2     Mazda RX4 Wag             mpg        21.0
## 3        Datsun 710             mpg        22.8
## 4    Hornet 4 Drive             mpg        21.4
## 5 Hornet Sportabout             mpg        18.7
## 6           Valiant             mpg        18.1
```

4.3.2 Long to wide

`dcast` will conver from long to wide:

```
dcast(ChickWeight, Chick~Time, value.var="weight")
```

```
##    Chick  0  2  4  6   8  10  12  14  16  18  20  21
## 1     18 39 35 NA NA  NA  NA  NA  NA  NA  NA  NA  NA
## 2     16 41 45 49 51  57  51  54  NA  NA  NA  NA  NA
## 3     15 41 49 56 64  68  68  67  68  NA  NA  NA  NA
## 4     13 41 48 53 60  65  67  71  70  71  81  91  96
## 5      9 42 51 59 68  85  96  90  92  93 100 100  98
## 6     20 41 47 54 58  65  73  77  89  98 107 115 117
## 7     10 41 44 52 63  74  81  89  96 101 112 120 124
```

```
## 8      8 42 50 61 71  84  93 110 116 126 134 125  NA
## 9     17 42 51 61 72  83  89  98 103 113 123 133 142
## 10    19 43 48 55 62  65  71  82  88 106 120 144 157
## 11     4 42 49 56 67  74  87 102 108 136 154 160 157
## 12     6 41 49 59 74  97 124 141 148 155 160 160 157
## 13    11 43 51 63 84 112 139 168 177 182 184 181 175
## 14     3 43 39 55 67  84  99 115 138 163 187 198 202
## 15     1 42 51 59 64  76  93 106 125 149 171 199 205
## 16    12 41 49 56 62  72  88 119 135 162 185 195 205
## 17     2 40 49 58 72  84 103 122 138 162 187 209 215
## 18     5 41 42 48 60  79 106 141 164 197 199 220 223
## 19    14 41 49 62 79 101 128 164 192 227 248 259 266
## 20     7 41 49 57 71  89 112 146 174 218 250 288 305
## 21    24 42 52 58 74  66  68  70  71  72  72  76  74
## 22    30 42 48 59 72  85  98 115 122 143 151 157 150
## 23    22 41 55 64 77  90  95 108 111 131 148 164 167
## 24    23 43 52 61 73  90 103 127 135 145 163 170 175
## 25    27 39 46 58 73  87 100 115 123 144 163 185 192
## 26    28 39 46 58 73  92 114 145 156 184 207 212 233
## 27    26 42 48 57 74  93 114 136 147 169 205 236 251
## 28    25 40 49 62 78 102 124 146 164 197 231 259 265
## 29    29 39 48 59 74  87 106 134 150 187 230 279 309
## 30    21 40 50 62 86 125 163 217 240 275 307 318 331
## 31    33 39 50 63 77  96 111 137 144 151 146 156 147
## 32    37 41 48 56 68  80  83 103 112 135 157 169 178
## 33    36 39 48 61 76  98 116 145 166 198 227 225 220
## 34    31 42 53 62 73  85 102 123 138 170 204 235 256
## 35    39 42 50 61 78  89 109 130 146 170 214 250 272
## 36    38 41 49 61 74  98 109 128 154 192 232 280 290
## 37    32 41 49 65 82 107 129 159 179 221 263 291 305
## 38    40 41 55 66 79 101 120 154 182 215 262 295 321
## 39    34 41 49 63 85 107 134 164 186 235 294 327 341
## 40    35 41 53 64 87 123 158 201 238 287 332 361 373
## 41    44 42 51 65 86 103 118 127 138 145 146  NA  NA
## 42    45 41 50 61 78  98 117 135 141 147 174 197 196
## 43    43 42 55 69 96 131 157 184 188 197 198 199 200
## 44    41 42 51 66 85 103 124 155 153 175 184 199 204
## 45    47 41 53 66 79 100 123 148 157 168 185 210 205
## 46    49 40 53 64 85 108 128 152 166 184 203 233 237
## 47    46 40 52 62 82 101 120 144 156 173 210 231 238
## 48    50 41 54 67 84 105 122 155 175 205 234 264 264
## 49    42 42 49 63 84 103 126 160 174 204 234 269 281
## 50    48 39 50 62 80 104 125 154 170 222 261 303 322
```

Things to note:

- `dcast` uses a formula interface (`~`) to specify the row identifier and the variables. The LHS is the row identifier, and the RHS for the variables to be created.
- The measurement of each LHS at each RHS, is specified using the `value.var` argument.

4.4 Bibliographic Notes

`data.table` has excellent online documentation. See here. See here for joining data.tables. See here for more on reshaping data.tables. See here for a comparison of the `data.frame` way, versus the `data.table` way. For some advanced tips and tricks see Andrew Brooks' blog.

4.5 Practice Yourself

Chapter 5

Exploratory Data Analysis

Exploratory Data Analysis (EDA) is a term coined by John W. Tukey in his seminal book (Tukey, 1977). It is also (arguably) known as *Visual Analytics*, or *Descriptive Statistics*. It is the practice of inspecting, and exploring your data, before stating hypotheses, fitting predictors, and other more ambitious inferential goals. It typically includes the computation of simple *summary statistics* which capture some property of interest in the data, and *visualization*. EDA can be thought of as an assumption free, purely algorithmic practice.

In this text we present EDA techniques along the following lines:

- How we explore: with summary-statistics, or visually?
- How many variables analyzed simultaneously: univariate, bivariate, or multivariate?
- What type of variable: categorical or continuous?

5.1 Summary Statistics

5.1.1 Categorical Data

Categorical variables do not admit any mathematical operations on them. We cannot sum them, or even sort them. We can only **count** them. As such, summaries of categorical variables will always start with the counting of the frequency of each category.

5.1.1.1 Summary of Univariate Categorical Data

```
# Make some data
gender <- c(rep('Boy', 10), rep('Girl', 12))
drink <- c(rep('Coke', 5), rep('Sprite', 3), rep('Coffee', 6), rep('Tea', 7), rep('Water', 1))
age <-  sample(c('Young', 'Old'), size = length(gender), replace = TRUE)
# Count frequencies
table(gender)
```

```
## gender
##  Boy Girl
##   10   12
```

```
table(drink)
```

```
## drink
## Coffee   Coke Sprite    Tea  Water
##      6      5      3      7      1
```

```
table(age)
```

```
## age
##   Old Young
##    10    12
```

If instead of the level counts you want the proportions, you can use `prop.table`

```
prop.table(table(gender))
```

```
## gender
##       Boy      Girl
## 0.4545455 0.5454545
```

5.1.1.2 Summary of Bivariate Categorical Data

```
library(magrittr)
cbind(gender, drink) %>% head # bind vectors into matrix and inspect
```

```
##      gender drink
## [1,] "Boy"  "Coke"
## [2,] "Boy"  "Coke"
## [3,] "Boy"  "Coke"
## [4,] "Boy"  "Coke"
## [5,] "Boy"  "Coke"
## [6,] "Boy"  "Sprite"
```

```
table1 <- table(gender, drink) # count frequencies of bivariate combinations
table1
```

```
##       drink
## gender Coffee Coke Sprite Tea Water
##   Boy       2    5      3   0     0
##   Girl      4    0      0   7     1
```

5.1.1.3 Summary of Multivariate Categorical Data

You may be wondering how does R handle tables with more than two dimensions. It is indeed not trivial to report this in a human-readable way. R offers several solutions: `table` is easier to compute with, and `ftable` is human readable.

```
table2.1 <- table(gender, drink, age) # A machine readable table.
table2.1
```

```
## , , age = Old
##
##       drink
## gender Coffee Coke Sprite Tea Water
##   Boy       1    2      1   0     0
##   Girl      2    0      0   3     1
##
## , , age = Young
##
##       drink
## gender Coffee Coke Sprite Tea Water
##   Boy       1    3      2   0     0
##   Girl      2    0      0   4     0
```

```
table.2.2 <- ftable(gender, drink, age) # A human readable table.
table.2.2
```

```
##               age Old Young
## gender drink
```

```
## Boy    Coffee       1     1
##        Coke         2     3
##        Sprite       1     2
##        Tea          0     0
##        Water        0     0
## Girl   Coffee       2     2
##        Coke         0     0
##        Sprite       0     0
##        Tea          3     4
##        Water        1     0
```

If you want proportions instead of counts, you need to specify the denominator, i.e., the margins. Think: what is the margin in each of the following outputs?

```
prop.table(table1, margin = 1)
```

```
##       drink
## gender     Coffee       Coke     Sprite        Tea      Water
##   Boy  0.20000000 0.50000000 0.30000000 0.00000000 0.00000000
##   Girl 0.33333333 0.00000000 0.00000000 0.58333333 0.08333333
```

```
prop.table(table1, margin = 2)
```

```
##       drink
## gender    Coffee      Coke    Sprite       Tea     Water
##   Boy  0.3333333 1.0000000 1.0000000 0.0000000 0.0000000
##   Girl 0.6666667 0.0000000 0.0000000 1.0000000 1.0000000
```

5.1.2 Continous Data

Continuous variables admit many more operations than categorical. We can compute sums, means, quantiles, and more.

5.1.2.1 Summary of Univariate Continuous Data

We distinguish between several types of summaries, each capturing a different property of the data.

5.1.2.2 Summary of Location

Capture the "location" of the data. These include:

Definition 5.1 (Average)**.** The mean, or average, of a sample $x := (x_1, \ldots, x_n)$, denoted $\bar{x}$ is defined as

$$\bar{x} := n^{-1} \sum x_i.$$

The sample mean is **non robust**. A single large observation may inflate the mean indefinitely. For this reason, we define several other summaries of location, which are more robust, i.e., less affected by "contaminations" of the data.

We start by defining the sample quantiles, themselves **not** a summary of location.

Definition 5.2 (Quantiles)**.** The α quantile of a sample x, denoted x_α, is (non uniquely) defined as a value above $100\alpha\%$ of the sample, and below $100(1-\alpha)\%$.

We emphasize that sample quantiles are non-uniquely defined. See `?quantile` for the 9(!) different definitions that R provides.

Using the sample quantiles, we can now define another summary of location, the **median**.

Definition 5.3 (Median)**.** The median of a sample x, denoted $x_{0.5}$ is the $\alpha = 0.5$ quantile of the sample.

A whole family of summaries of locations is the **alpha trimmed mean**.

Definition 5.4 (Alpha Trimmed Mean)**.** The α trimmed mean of a sample x, denoted $\bar{x}_\alpha$ is the average of the sample after removing the α proportion of largest and α proportion of smallest observations.

The simple mean and median are instances of the alpha trimmed mean: $\bar{x}_0$ and $\bar{x}_{0.5}$ respectively.

Here are the R implementations:

```
x <- rexp(100) # generate some random data
mean(x) # simple mean
```

```
## [1] 1.074307
```

```
median(x) # median
```

```
## [1] 0.7814793
```

```
mean(x, trim = 0.2) # alpha trimmed mean with alpha=0.2
```

```
## [1] 0.7988495
```

5.1.2.3 Summary of Scale

The *scale* of the data, sometimes known as *spread*, can be thought of its variability.

Definition 5.5 (Standard Deviation)**.** The standard deviation of a sample x, denoted $S(x)$, is defined as

$$S(x) := \sqrt{(n-1)^{-1} \sum (x_i - \bar{x})^2}.$$

For reasons of robustness, we define other, more robust, measures of scale.

Definition 5.6 (MAD)**.** The Median Absolute Deviation from the median, denoted as $MAD(x)$, is defined as

$$MAD(x) := c\,|x - x_{0.5}|_{0.5}.$$

where c is some constant, typically set to $c = 1.4826$ so that MAD and $S(x)$ have the same large sample limit.

Definition 5.7 (IQR)**.** The Inter Quantile Range of a sample x, denoted as $IQR(x)$, is defined as

$$IQR(x) := x_{0.75} - x_{0.25}.$$

Here are the R implementations

```
sd(x) # standard deviation
```

```
## [1] 1.070065
```

```
mad(x) # MAD
```

```
## [1] 0.7046802
```

```
IQR(x) # IQR
```

```
## [1] 0.9900286
```

5.1.2.4 Summary of Asymmetry

Summaries of asymmetry, also known as *skewness*, quantify the departure of the x from a symmetric sample.

Definition 5.8 (Yule)**.** The Yule measure of assymetry, denoted $Yule(x)$ is defined as

$$Yule(x) := \frac{1/2\,(x_{0.75} + x_{0.25}) - x_{0.5}}{1/2\,IQR(x)}.$$

Here is an R implementation

```
yule <- function(x){
  numerator <- 0.5 * (quantile(x,0.75) + quantile(x,0.25))-median(x)
  denominator <- 0.5* IQR(x)
  c(numerator/denominator, use.names=FALSE)
}
yule(x)
```

```
## [1] 0.2080004
```

5.1.2.5 Summary of Bivariate Continuous Data

When dealing with bivariate, or multivariate data, we can obviously compute univariate summaries for each variable separately. This is not the topic of this section, in which we want to summarize the association **between** the variables, and not within them.

Definition 5.9 (Covariance)**.** The covariance between two samples, x and y, of same length n, is defined as

$$Cov(x,y) := (n-1)^{-1} \sum (x_i - \bar{x})(y_i - \bar{y})$$

We emphasize this is not the covariance you learned about in probability classes, since it is not the covariance between two *random variables* but rather, between two *samples*. For this reasons, some authors call it the *empirical covariance*, or *sample covariance*.

Definition 5.10 (Pearson's Correlation Coefficient)**.** Peasrson's correlation coefficient, a.k.a. Pearson's moment product correlation, or simply, the correlation, denoted `r(x,y)`, is defined as

$$r(x,y) := \frac{Cov(x,y)}{S(x)S(y)}.$$

If you find this definition enigmatic, just think of the correlation as the covariance between x and y after transforming each to the unitless scale of z-scores.

Definition 5.11 (Z-Score)**.** The z-scores of a sample x are defined as the mean-centered, scale normalized observations:

$$z_i(x) := \frac{x_i - \bar{x}}{S(x)}.$$

We thus have that $r(x,y) = Cov(z(x), z(y))$.

5.1.2.6 Summary of Multivariate Continuous Data

The covariance is a simple summary of association between two variables, but it certainly may not capture the whole "story" when dealing with more than two variables. The most common summary of multivariate relation, is the **covariance matrix**, but we warn that only the simplest multivariate relations are fully summarized by this matrix.

Definition 5.12 (Sample Covariance Matrix)**.** Given n observations on p variables, denote $x_{i,j}$ the i'th observation of the j'th variable. The *sample covariance matrix*, denoted $\hat{\Sigma}$ is defined as

$$\hat{\Sigma}_{k,l} = (n-1)^{-1} \sum_i [(x_{i,k} - \bar{x}_k)(x_{i,l} - \bar{x}_l)],$$

where $\bar{x}_k := n^{-1} \sum_i x_{i,k}$. Put differently, the k,l'th entry in $\hat{\Sigma}$ is the sample covariance between variables k and l.

Remark. $\hat{\Sigma}$ is clearly non robust. How would you define a robust covariance matrix?

5.2 Visualization

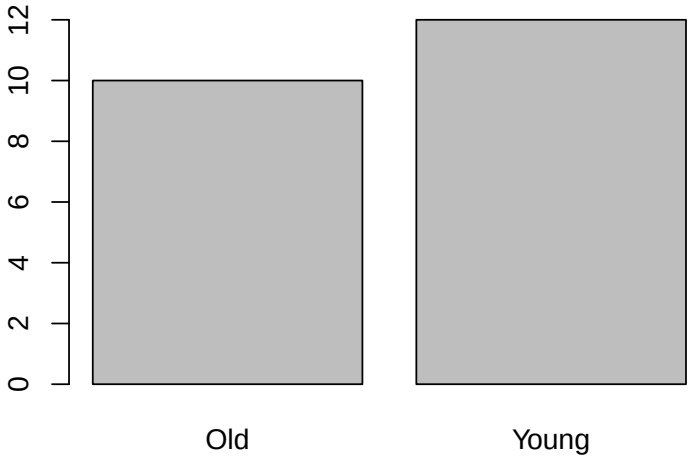
Summarizing the information in a variable to a single number clearly conceals much of the story in the sample. This is akin to inspecting a person using a caricature, instead of a picture. Visualizing the data, when possible, is more informative.

5.2.1 Categorical Data

Recalling that with categorical variables we can only count the frequency of each level, the plotting of such variables are typically variations on the *bar plot*.

5.2.1.1 Visualizing Univariate Categorical Data

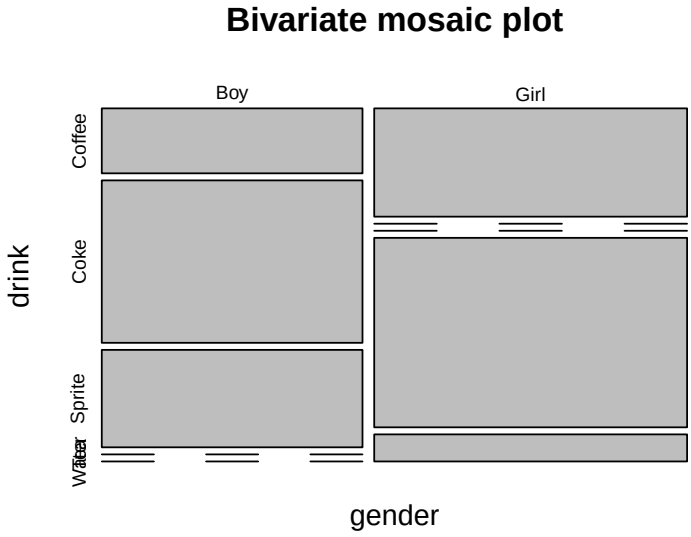
```
barplot(table(age))
```



5.2.1.2 Visualizing Bivariate Categorical Data

There are several generalizations of the barplot, aimed to deal with the visualization of bivariate categorical data. They are sometimes known as the *clustered bar plot* and the *stacked bar plot*. In this text, we advocate the use of the *mosaic plot* which is also the default in R.

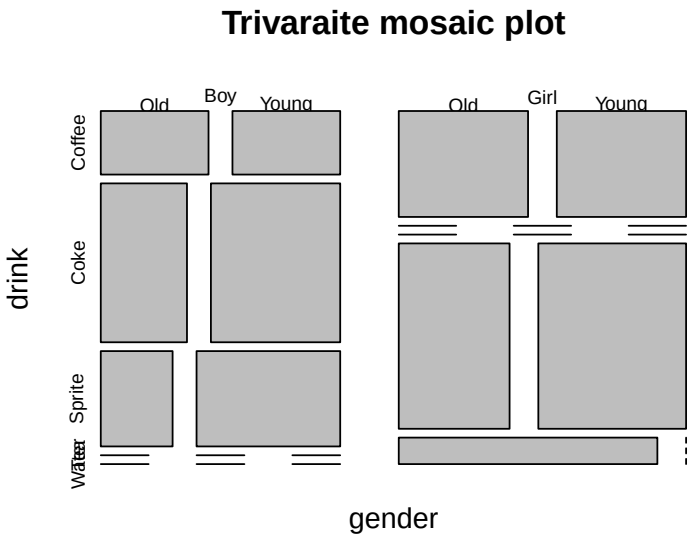
```
plot(table1, main='Bivariate mosaic plot')
```



5.2.1.3 Visualizing Multivariate Categorical Data

The *mosaic plot* is not easy to generalize to more than two variables, but it is still possible (at the cost of interpretability).

```
plot(table2.1, main='Trivaraite mosaic plot')
```

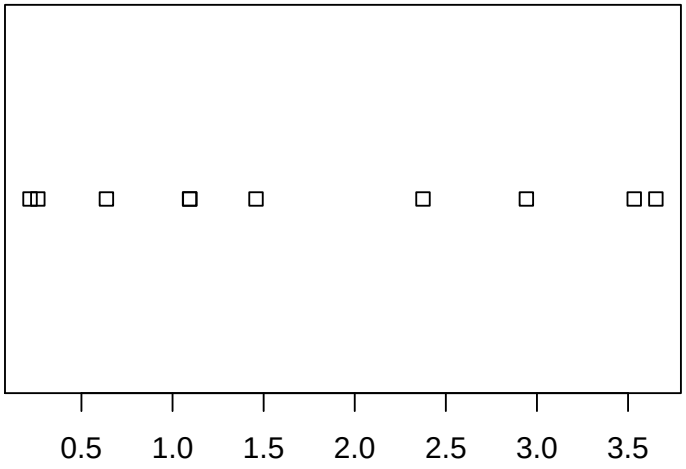


5.2.2 Continuous Data

5.2.2.1 Visualizing Univariate Continuous Data

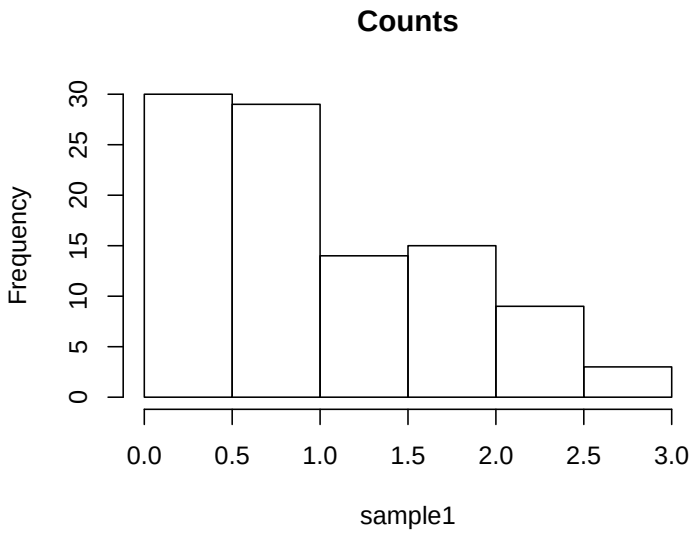
Unlike categorical variables, there are endlessly many way to visualize continuous variables. The simplest way is to look at the raw data via the `stripchart`.

```
sample1 <- rexp(10)
stripchart(sample1)
```



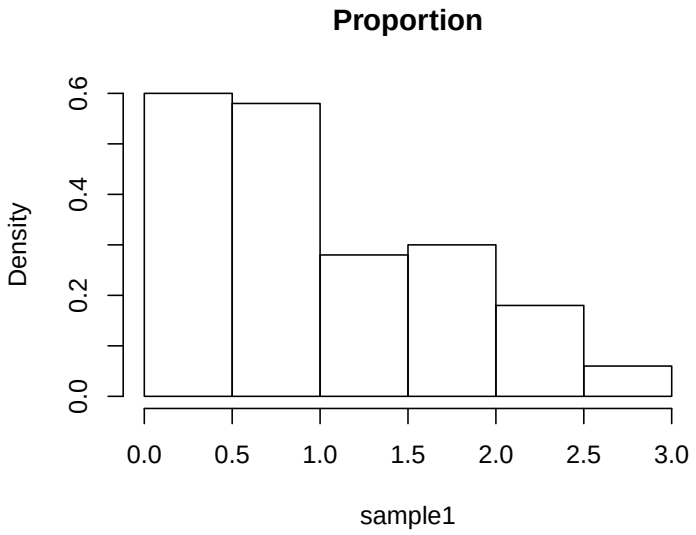
Clearly, if there are many observations, the `stripchart` will be a useless line of black dots. We thus bin them together, and look at the frequency of each bin; this is the *histogram*. R's `histogram` function has very good defaults to choose the number of bins. Here is a histogram showing the counts of each bin.

```
sample1 <- rexp(100)
hist(sample1, freq=T, main='Counts')
```



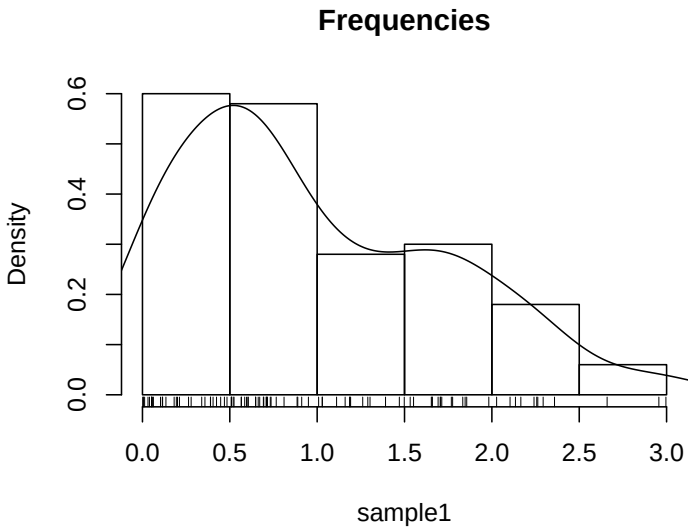
The bin counts can be replaced with the proportion of each bin using the `freq` argument.

```
hist(sample1, freq=F, main='Proportion')
```



The bins of a histogram are non overlapping. We can adopt a sliding window approach, instead of binning. This is the *density plot* which is produced with the `density` function, and added to an existing plot with the `lines` function. The `rug` function adds the original data points as ticks on the axes, and is strongly recommended to detect artifacts introduced by the binning of the histogram, or the smoothing of the density plot.

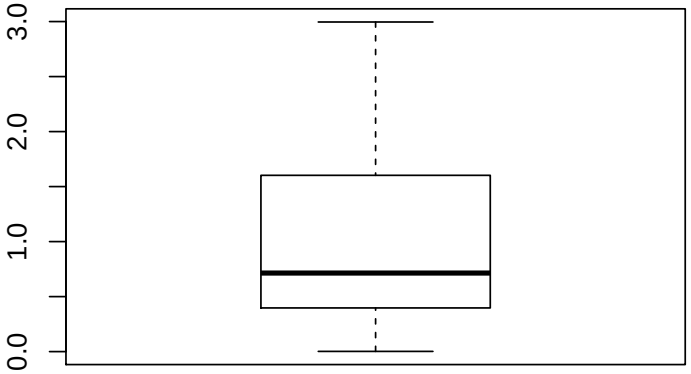
```
hist(sample1, freq=F, main='Frequencies')
lines(density(sample1))
rug(sample1)
```



Remark. Why would it make no sense to make a table, or a barplot, of continuous data?

One particularly useful visualization, due to John W. Tukey, is the *boxplot*. The boxplot is designed to capture the main phenomena in the data, and simultaneously point to outlines.

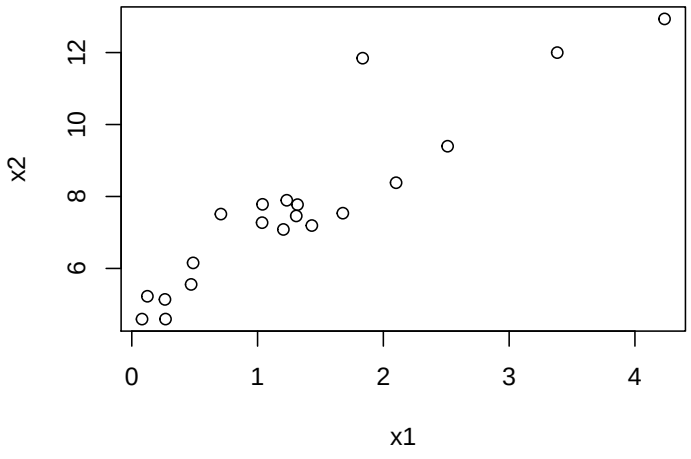
```
boxplot(sample1)
```



5.2.2.2 Visualizing Bivariate Continuous Data

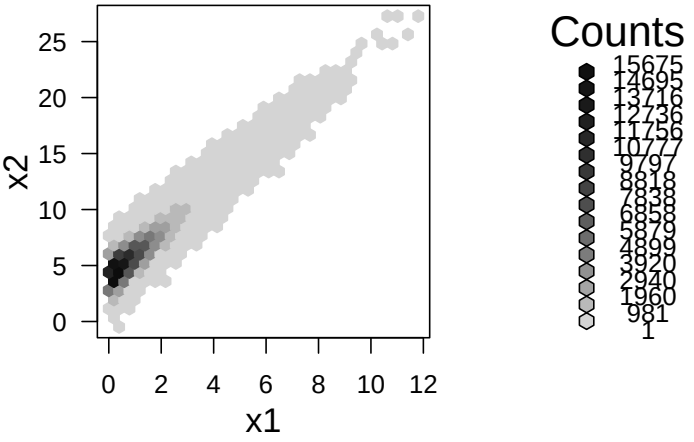
The bivariate counterpart of the `stipchart` is the celebrated scatter plot.

```
n <- 20
x1 <- rexp(n)
x2 <- 2* x1 + 4 + rexp(n)
plot(x2~x1)
```



Like the univariate `stripchart`, the scatter plot will be an uninformative mess in the presence of a lot of data. A nice bivariate counterpart of the univariate histogram is the *hexbin plot*, which tessellates the plane with hexagons, and reports their frequencies.

```
library(hexbin) # load required library
n <- 2e5
x1 <- rexp(n)
x2 <- 2* x1 + 4 + rnorm(n)
plot(hexbin(x = x1, y = x2))
```

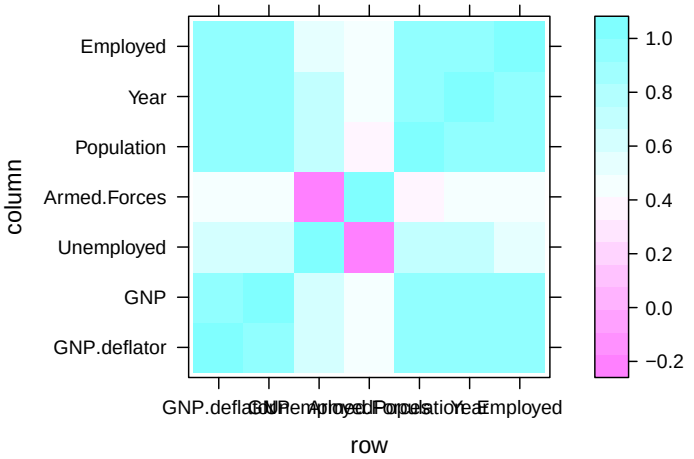


5.2.2.3 Visualizing Multivariate Continuous Data

Visualizing multivariate data is a tremendous challenge given that we cannot grasp 4 dimensional spaces, nor can the computer screen present more than 2 dimensional spaces. We thus have several options: (i) To project the data to 2D. This is discussed in the Dimensionality Reduction Section **??**. (ii) To visualize not the raw data, but rather its summaries, like the covariance matrix.

Since the covariance matrix, $\hat{\Sigma}$ is a matrix, it can be visualized as an image. Note the use of the `::` operator, which is used to call a function from some package, without loading the whole package. We will use the `::` operator when we want to emphasize the package of origin of a function.

```
covariance <- cov(longley) # The covariance of the longley dataset
correlations <- cor(longley) # The correlations of the longley dataset
lattice::levelplot(correlations)
```



5.2.2.4 Parallel Coordinate Plots

TODO

5.3 Mixed Type Data

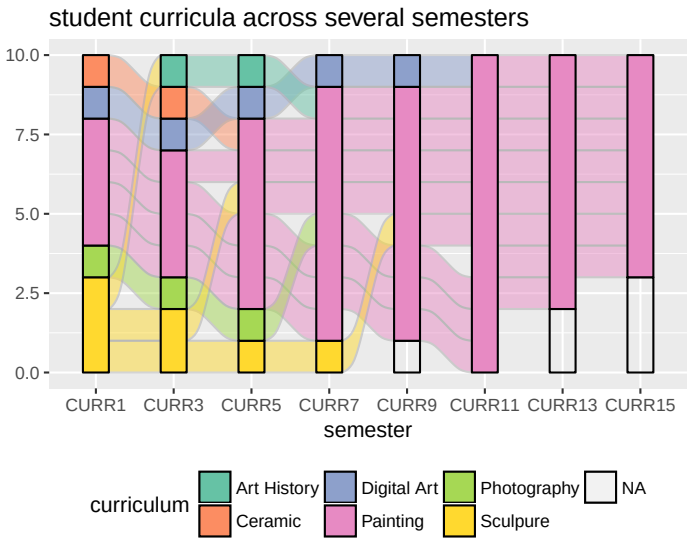
Most real data sets will be of mixed type: both categorical and continous. One approach to view such data, is to visualize the continous variables separatly, for each level of the categorical variables. There are, however, interesting dedicated visualization for such data.

5.3.1 Alluvian Diagram

An Alluvian plot is a type of Parallel Coordinate Plot for multivariate categorical data. It is particularly interesting when the x axis is a discretized time variable, and it is used to visualize flow.

The following example, from the **ggalluvial** package Vignette by Jason Cory Brunson, demonstrates the flow of students between different majors, as semesters evolve.

```
library(ggalluvial)
data(majors)
majors$curriculum <- as.factor(majors$curriculum)
ggplot(majors,
       aes(x = semester, stratum = curriculum, alluvium = student,
           fill = curriculum, label = curriculum)) +
  scale_fill_brewer(type = "qual", palette = "Set2") +
  geom_flow(stat = "alluvium", lode.guidance = "rightleft",
            color = "darkgray") +
  geom_stratum() +
  theme(legend.position = "bottom") +
  ggtitle("student curricula across several semesters")
```



Things to note:

- We used the **galluvian** package of the **ggplot2** ecosystem. More on **ggplot2** in the Plotting Chapter.
- Time is on the x axis. Categories are color coded.

5.4 Bibliographic Notes

Like any other topic in this book, you can consult Venables and Ripley (2013). The seminal book on EDA, written long before R was around, is Tukey (1977). For an excellent text on robust statistics see Wilcox (2011).

5.5 Practice Yourself

1. Read about the Titanic data set using `?Titanic`. Inspect it with the `table` and with the `ftable` commands. Which do you prefer?
2. Inspect the Titanic data with a plot. Start with `plot(Titanic)` Try also `lattice::dotplot`. Which is the passenger category with most survivors? Which plot do you prefer? Which scales better to more categories?
3. Read about the women data using `?women`.
    1. Compute the average of each variable. What is the average of the heights?
    2. Plot a histogram of the heights. Add ticks using `rug`.
    3. Plot a boxplot of the weights.
    4. Plot the heights and weights using a scatter plot. Add ticks using `rug`.
4. Choose α to define a new symmetry measure: $1/2(x_\alpha + x_{1-\alpha}) - x_{0.5}$. Write a function that computes it, and apply it on women's heights data.
5. Compute the covariance matrix of women's heights and weights. Compute the correlation matrix. View the correlation matrix as an image using `lattice::levelplot`.
6. Pick a dataset with two LONG continous variables from `?datasets`. Plot it using `hexbin::hexbin`.

Chapter 6

Linear Models

6.1 Problem Setup

Example 6.1 (Bottle Cap Production)**.** Consider a randomized experiment designed to study the effects of temperature and pressure on the diameter of manufactured a bottle cap.

Example 6.2 (Rental Prices)**.** Consider the prediction of rental prices given an appartment's attributes.

Both examples require some statistical model, but they are very different. The first is a *causal inference* problem: we want to design an intervention so that we need to recover the causal effect of temperature and pressure. The second is a *prediction* problem, a.k.a. a *forecasting* problem, in which we don't care about the causal effects, we just want good predictions.

In this chapter we discuss the causal problem in Example 6.1. This means that when we assume a model, we assume it is the actual *data generating process*, i.e., we assume the *sampling distribution* is well specified. The second type of problems is discussed in the Supervised Learning Chapter **??**.

Here are some more examples of the types of problems we are discussing.

Example 6.3 (Plant Growth)**.** Consider the treatment of various plants with various fertilizers to study the fertilizer's effect on growth.

Example 6.4 (Return to Education)**.** Consider the study of return to education by analyzing the incomes of individuals with different education years.

Example 6.5 (Drug Effect)**.** Consider the study of the effect of a new drug for hemophilia, by analyzing the level of blood coagulation after the administration of various amounts of the new drug.

Let's present the linear model. We assume that a response[1] variable is the sum of effects of some factors[2]. Denoting the response variable by y, the factors by $x = (x_1, \dots, x_p)$, and the effects by $\beta := (\beta_1, \dots, \beta_p)$ the linear model assumption implies that the expected response is the sum of the factors effects:

$$E[y] = x_1\beta_1 + \cdots + x_p\beta_p = \sum_{j=1}^{p} x_j\beta_j = x'\beta. \tag{6.1}$$

Clearly, there may be other factors that affect the the caps' diameters. We thus introduce an error term[3], denoted by ε, to capture the effects of all unmodeled factors and measurement error[4]. The implied generative process of a sample of $i = 1, \dots, n$ observations it thus

[1] The "response" is also know as the "dependent" variable in the statistical literature, or the "labels" in the machine learning literature.

[2] The "factors" are also known as the "independent variable", or "the design", in the statistical literature, and the "features", or "attributes" in the machine learning literature.

[3] The "error term" is also known as the "noise", or the "common causes of variability".

[4] You may philosophize if the measurement error is a mere instance of unmodeled factors or not, but this has no real implication for our purposes.

$$y_i = x_i'\beta + \varepsilon_i = \sum_j x_{i,j}\beta_j + \varepsilon_i, i = 1, \ldots, n. \tag{6.2}$$

or in matrix notation

$$y = X\beta + \varepsilon. \tag{6.3}$$

Let's demonstrate Eq.(6.2). In our cap example (6.1), assuming that pressure and temperature have two levels each (say, high and low), we would write $x_{i,1} = 1$ if the pressure of the i'th measurement was set to high, and $x_{i,1} = -1$ if the **pressure** was set to low. Similarly, we would write $x_{i,2} = 1$, and $x_{i,2} = -1$, if the **temperature** was set to high, or low, respectively. The coding with $\{-1, 1\}$ is known as *effect coding*. If you prefer coding with $\{0, 1\}$, this is known as *dummy coding*. The choice of coding has no real effect on the results, provided that you remember what coding you used when interpreting $\hat{\beta}$.
Remark. In Galton's classical regression problem, where we try to seek the relation between the heights of sons and fathers then $p = 1$, y_i is the height of the i'th father, and x_i the height of the i'th son.

There are many reasons linear models are very popular:

1. Before the computer age, these were pretty much the only models that could actually be computed[5]. The whole Analysis of Variance (ANOVA) literature is an instance of linear models, that relies on sums of squares, which do not require a computer to work with.
2. For purposes of prediction, where the actual data generating process is not of primary importance, they are popular because they simply work. Why is that? They are simple so that they do not require a lot of data to be computed. Put differently, they may be biased, but their variance is small enough to make them more accurate than other models.
3. For non continuous predictors, **any** functional relation can be cast as a linear model.
4. For the purpose of *screening*, where we only want to show the existence of an effect, and are less interested in the magnitude of that effect, a linear model is enough.
5. If the true generative relation is not linear, but smooth enough, then the linear function is a good approximation via Taylor's theorem.

There are still two matters we have to attend: (i) How the estimate β? (ii) How to perform inference?

In the simplest linear models the estimation of β is done using the method of least squares. A linear model with least squares estimation is known as Ordinary Least Squares (OLS). The OLS problem:

$$\hat{\beta} := argmin_\beta\{\sum_i (y_i - x_i'\beta)^2\}, \tag{6.4}$$

and in matrix notation

$$\hat{\beta} := argmin_\beta\{\|y - X\beta\|_2^2\}. \tag{6.5}$$

Remark. Personally, I prefer the matrix notation because it is suggestive of the geometry of the problem. The reader is referred to Friedman et al. (2001), Section 3.2, for more on the geometry of OLS.

Different software suits, and even different R packages, solve Eq.(6.4) in different ways so that we skip the details of how exactly it is solved. These are discussed in Chapters **??** and **??**.

The last matter we need to attend is how to do inference on $\hat{\beta}$. For that, we will need some assumptions on ε. A typical set of assumptions is the following:

[5] By "computed" we mean what statisticians call "fitted", or "estimated", and computer scientists call "learned".

1. **Independence**: we assume ε_i are independent of everything else. Think of them as the measurement error of an instrument: it is independent of the measured value and of previous measurements.
2. **Centered**: we assume that $E[\varepsilon] = 0$, meaning there is no systematic error, sometimes it called The "Linearity assumption".
3. **Normality**: we will typically assume that $\varepsilon \sim \mathcal{N}(0, \sigma^2)$, but we will later see that this is not really required.

We emphasize that these assumptions are only needed for inference on $\hat{\beta}$ and not for the estimation itself, which is done by the purely algorithmic framework of OLS.

Given the above assumptions, we can apply some probability theory and linear algebra to get the distribution of the estimation error:

$$\hat{\beta} - \beta \sim \mathcal{N}(0, (X'X)^{-1}\sigma^2). \tag{6.6}$$

The reason I am not too strict about the normality assumption above, is that Eq.(6.6) is approximately correct even if ε is not normal, provided that there are many more observations than factors ($n \gg p$).

6.2 OLS Estimation in R

We are now ready to estimate some linear models with R. We will use the `whiteside` data from the **MASS** package, recording the outside temperature and gas consumption, before and after an apartment's insulation.

```
library(MASS) # load the package
library(data.table) # for some data manipulations
data(whiteside) # load the data
head(whiteside) # inspect the data
```

```
##    Insul Temp Gas
## 1 Before -0.8 7.2
## 2 Before -0.7 6.9
## 3 Before  0.4 6.4
## 4 Before  2.5 6.0
## 5 Before  2.9 5.8
## 6 Before  3.2 5.8
```

We do the OLS estimation on the pre-insulation data with `lm` function, possibly the most important function in R.

```
whiteside <- data.table(whiteside)
lm.1 <- lm(Gas~Temp, data=whiteside[Insul=='Before']) # OLS estimation
```

Things to note:

- We used the tilde syntax `Gas~Temp`, reading "gas as linear function of temperature".
- The `data` argument tells R where to look for the variables Gas and Temp. We used `Insul=='Before'` to subset observations before the insulation.
- The result is assigned to the object `lm.1`.

Like any other language, spoken or programable, there are many ways to say the same thing. Some more elegant than others...

```
lm.1 <- lm(y=Gas, x=Temp, data=whiteside[whiteside$Insul=='Before',])
lm.1 <- lm(y=whiteside[whiteside$Insul=='Before',]$Gas,x=whiteside[whiteside$Insul=='Before',]$Temp)
lm.1 <- whiteside[whiteside$Insul=='Before',] %>% lm(Gas~Temp, data=.)
```

The output is an object of class `lm`.

```
class(lm.1)
```

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

Objects of class `lm` are very complicated. They store a lot of information which may be used for inference, plotting, etc. The `str` function, short for "structure", shows us the various elements of the object.

```
str(lm.1)
```

```
## List of 12
##  $ coefficients : Named num [1:2] 6.854 -0.393
##   ..- attr(*, "names")= chr [1:2] "(Intercept)" "Temp"
##  $ residuals    : Named num [1:26] 0.0316 -0.2291 -0.2965 0.1293 0.0866 ...
##   ..- attr(*, "names")= chr [1:26] "1" "2" "3" "4" ...
##  $ effects      : Named num [1:26] -24.2203 -5.6485 -0.2541 0.1463 0.0988 ...
##   ..- attr(*, "names")= chr [1:26] "(Intercept)" "Temp" "" "" ...
##  $ rank         : int 2
##  $ fitted.values: Named num [1:26] 7.17 7.13 6.7 5.87 5.71 ...
##   ..- attr(*, "names")= chr [1:26] "1" "2" "3" "4" ...
##  $ assign       : int [1:2] 0 1
##  $ qr           :List of 5
##   ..$ qr   : num [1:26, 1:2] -5.099 0.196 0.196 0.196 0.196 ...
##   .. ..- attr(*, "dimnames")=List of 2
##   .. .. ..$ : chr [1:26] "1" "2" "3" "4" ...
##   .. .. ..$ : chr [1:2] "(Intercept)" "Temp"
##   .. ..- attr(*, "assign")= int [1:2] 0 1
##   ..$ qraux: num [1:2] 1.2 1.35
##   ..$ pivot: int [1:2] 1 2
##   ..$ tol  : num 1e-07
##   ..$ rank : int 2
##   ..- attr(*, "class")= chr "qr"
##  $ df.residual  : int 24
##  $ xlevels      : Named list()
##  $ call         : language lm(formula = Gas ~ Temp, data = whiteside[Insul == "Before"])
##  $ terms        :Classes 'terms', 'formula'  language Gas ~ Temp
##   .. ..- attr(*, "variables")= language list(Gas, Temp)
##   .. ..- attr(*, "factors")= int [1:2, 1] 0 1
##   .. .. ..- attr(*, "dimnames")=List of 2
##   .. .. .. ..$ : chr [1:2] "Gas" "Temp"
##   .. .. .. ..$ : chr "Temp"
##   .. ..- attr(*, "term.labels")= chr "Temp"
##   .. ..- attr(*, "order")= int 1
##   .. ..- attr(*, "intercept")= int 1
##   .. ..- attr(*, "response")= int 1
##   .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
##   .. ..- attr(*, "predvars")= language list(Gas, Temp)
##   .. ..- attr(*, "dataClasses")= Named chr [1:2] "numeric" "numeric"
##   .. .. ..- attr(*, "names")= chr [1:2] "Gas" "Temp"
##  $ model        :'data.frame':   26 obs. of  2 variables:
##   ..$ Gas : num [1:26] 7.2 6.9 6.4 6 5.8 5.8 5.6 4.7 5.8 5.2 ...
##   ..$ Temp: num [1:26] -0.8 -0.7 0.4 2.5 2.9 3.2 3.6 3.9 4.2 4.3 ...
##   ..- attr(*, "terms")=Classes 'terms', 'formula'  language Gas ~ Temp
##   .. .. ..- attr(*, "variables")= language list(Gas, Temp)
##   .. .. ..- attr(*, "factors")= int [1:2, 1] 0 1
##   .. .. .. ..- attr(*, "dimnames")=List of 2
##   .. .. .. .. ..$ : chr [1:2] "Gas" "Temp"
##   .. .. .. .. ..$ : chr "Temp"
##   .. .. ..- attr(*, "term.labels")= chr "Temp"
##   .. .. ..- attr(*, "order")= int 1
##   .. .. ..- attr(*, "intercept")= int 1
##   .. .. ..- attr(*, "response")= int 1
##   .. .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
```

```
##   .. .. ..- attr(*, "predvars")= language list(Gas, Temp)
##   .. .. ..- attr(*, "dataClasses")= Named chr [1:2] "numeric" "numeric"
##   .. .. .. ..- attr(*, "names")= chr [1:2] "Gas" "Temp"
##  - attr(*, "class")= chr "lm"
```

In RStudio it is particularly easy to extract objects. Just write `your.object$` and press `tab` after the `$` for autocompletion.

If we only want $\hat{\beta}$, it can also be extracted with the `coef` function.

```
coef(lm.1)
```

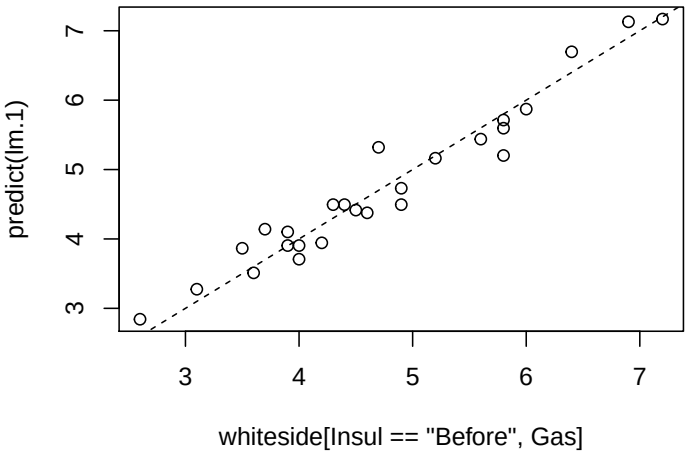
```
## (Intercept)        Temp
##   6.8538277  -0.3932388
```

Things to note:

- R automatically adds an `(Intercept)` term. This means we estimate $Gas = \beta_0 + \beta_1 Temp + \varepsilon$ and not $Gas = \beta_1 Temp + \varepsilon$. This makes sense because we are interested in the contribution of the temperature to the variability of the gas consumption about its **mean**, and not about zero.
- The effect of temperature, i.e., $\hat{\beta}_1$, is -0.39. The negative sign means that the higher the temperature, the less gas is consumed. The magnitude of the coefficient means that for a unit increase in the outside temperature, the gas consumption decreases by 0.39 units.

We can use the `predict` function to make predictions, but we emphasize that if the purpose of the model is to make predictions, and not interpret coefficients, better skip to the Supervised Learning Chapter **??**.

```
plot(predict(lm.1)~whiteside[Insul=='Before',Gas])
abline(0,1, lty=2)
```



The model seems to fit the data nicely. A common measure of the goodness of fit is the *coefficient of determination*, more commonly known as the R^2.

Definition 6.1 (R2)**.** The coefficient of determination, denoted R^2, is defined as

$$R^2 := 1 - \frac{\sum_i (y_i - \hat{y}_i)^2}{\sum_i (y_i - \bar{y})^2}, \tag{6.7}$$

where $\hat{y}_i$ is the model's prediction, $\hat{y}_i = x_i \hat{\beta}$.

It can be easily computed

```
R2 <- function(y, y.hat){
  numerator <- (y-y.hat)^2 %>% sum
  denominator <- (y-mean(y))^2 %>% sum
  1-numerator/denominator
```

```
}
R2(y=whiteside[Insul=='Before',Gas], y.hat=predict(lm.1))
```

```
## [1] 0.9438081
```

This is a nice result implying that about 94% of the variability in gas consumption can be attributed to changes in the outside temperature.

Obviously, R does provide the means to compute something as basic as R^2, but I will let you find it for yourselves.

6.3 Inference

To perform inference on $\hat{\beta}$, in order to test hypotheses and construct confidence intervals, we need to quantify the uncertainly in the reported $\hat{\beta}$. This is exactly what Eq.(6.6) gives us.

Luckily, we don't need to manipulate multivariate distributions manually, and everything we need is already implemented. The most important function is `summary` which gives us an overview of the model's fit. We emphasize that that fitting a model with `lm` is an assumption free algorithmic step. Inference using `summary` is **not** assumption free, and requires the set of assumptions leading to Eq.(6.6).

```
summary(lm.1)
```

```
##
## Call:
## lm(formula = Gas ~ Temp, data = whiteside[Insul == "Before"])
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -0.62020 -0.19947  0.06068  0.16770  0.59778
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  6.85383    0.11842   57.88   <2e-16 ***
## Temp        -0.39324    0.01959  -20.08   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.2813 on 24 degrees of freedom
## Multiple R-squared:  0.9438, Adjusted R-squared:  0.9415
## F-statistic: 403.1 on 1 and 24 DF,  p-value: < 2.2e-16
```

Things to note:

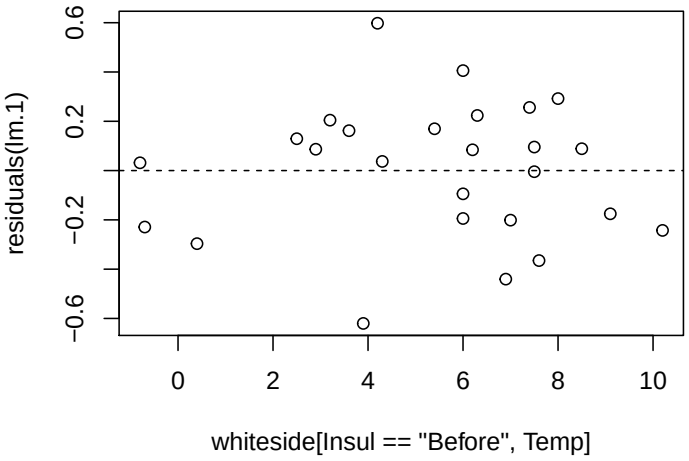
- The estimated $\hat{\beta}$ is reported in the 'Coefficients' table, which has point estimates, standard errors, t-statistics, and the p-values of a two-sided hypothesis test for each coefficient $H_{0,j} : \beta_j = 0, j = 1, \ldots, p$.
- The R^2 is reported at the bottom. The "Adjusted R-squared" is a variation that compensates for the model's complexity.
- The original call to `lm` is saved in the `Call` section.
- Some summary statistics of the residuals $(y_i - \hat{y}_i)$ in the `Residuals` section.
- The "residuals standard error"[6] is $\sqrt{(n-p)^{-1}\sum_i (y_i - \hat{y}_i)^2}$. The denominator of this expression is the *degrees of freedom*, $n - p$, which can be thought of as the hardness of the problem.

As the name suggests, `summary` is merely a summary. The full `summary(lm.1)` object is a monstrous object. Its various elements can be queried using `str(sumary(lm.1))`.

Can we check the assumptions required for inference? Some. Let's start with the linearity assumption. If we were wrong, and the data is not arranged about a linear line, the residuals will have some shape. We thus plot the residuals as a function of the predictor to diagnose shape.

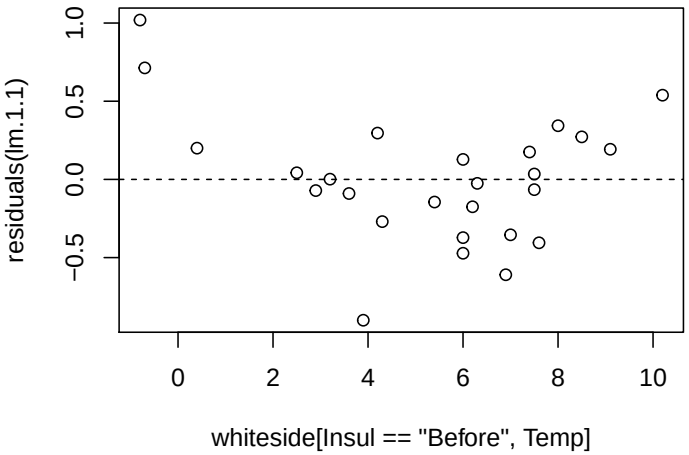
[6] Sometimes known as the Root Mean Squared Error (RMSE).

```
plot(residuals(lm.1)~whiteside[Insul=='Before',Temp])
abline(0,0, lty=2)
```



I can't say I see any shape. Let's fit a **wrong** model, just to see what "shape" means.

```
lm.1.1 <- lm(Gas~I(Temp^2), data=whiteside[Insul=='Before',])
plot(residuals(lm.1.1)~whiteside[Insul=='Before',Temp]); abline(0,0, lty=2)
```



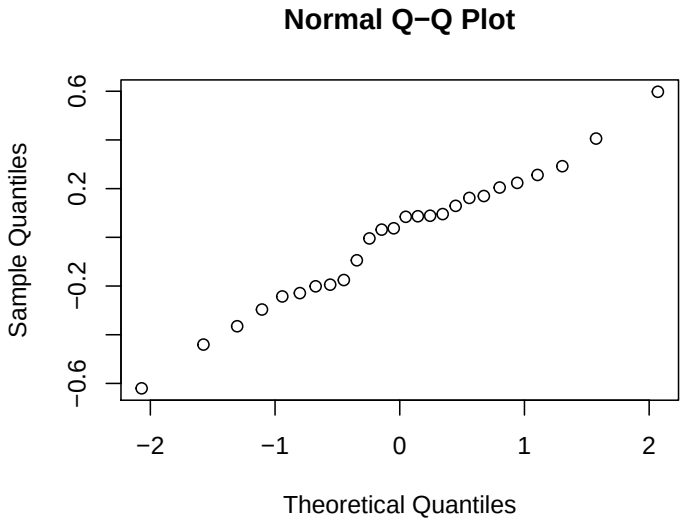
Things to note:

- We used `I(Temp)^2` to specify the model $Gas = \beta_0 + \beta_1 Temp^2 + \varepsilon$.
- The residuals have a "belly". Because they are not a cloud around the linear trend, and we have the wrong model.

To the next assumption. We assumed ε_i are independent of everything else. The residuals, $y_i - \hat{y}_i$ can be thought of a sample of ε_i. When diagnosing the linearity assumption, we already saw their distribution does not vary with the x's, `Temp` in our case. They may be correlated with themselves; a positive departure from the model, may be followed by a series of positive departures etc. Diagnosing these *auto-correlations* is a real art, which is not part of our course.

The last assumption we required is normality. As previously stated, if $n \gg p$, this assumption can be relaxed. If n is in the order of p, we need to verify this assumption. My favorite tool for this task is the *qqplot*. A qqplot compares the quantiles of the sample with the respective quantiles of the assumed distribution. If quantiles align along a line, the assumed distribution if OK. If quantiles depart from a line, then the assumed distribution does not fit the sample.

```
qqnorm(resid(lm.1))
```

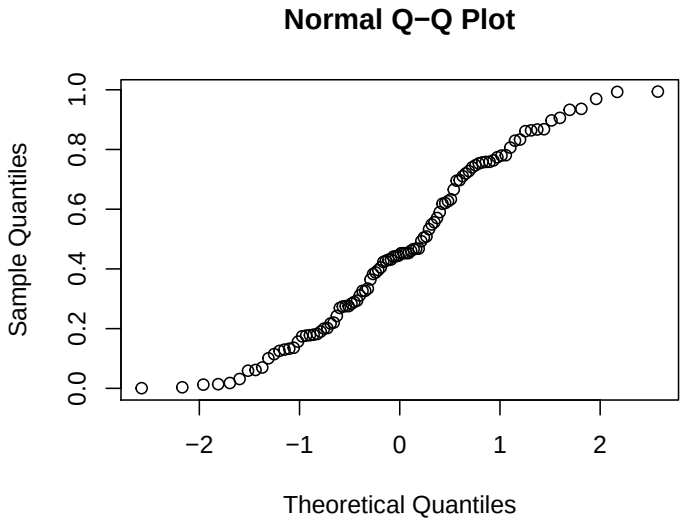


Things to note:

- The `qqnorm` function plots a qqplot against a normal distribution. For non-normal distributions try `qqplot`.
- `resid(lm.1)` extracts the residuals from the linear model, i.e., the vector of $y_i - x_i'\hat{\beta}$.

Judging from the figure, the normality assumption is quite plausible. Let's try the same on a non-normal sample, namely a uniformly distributed sample, to see how that would look.

```
qqnorm(runif(100))
```



6.3.1 Testing a Hypothesis on a Single Coefficient

The first inferential test we consider is a hypothesis test on a single coefficient. In our gas example, we may want to test that the temperature has no effect on the gas consumption. The answer for that is given immediately by `summary(lm.1)`

```
summary.lm1 <- summary(lm.1)
coefs.lm1 <- summary.lm1$coefficients
coefs.lm1
```

```
##               Estimate Std. Error   t value     Pr(>|t|)
## (Intercept)  6.8538277 0.11842341  57.87561 2.717533e-27
## Temp        -0.3932388 0.01958601 -20.07754 1.640469e-16
```

We see that the p-value for $H_{0,1} : \hat{\beta}_1 = 0$ against a two sided alternative is effectively 0, so that β_1 is unlikely to be 0.

6.3.2 Constructing a Confidence Interval on a Single Coefficient

Since the `summary` function gives us the standard errors of $\hat{\beta}$, we can immediately compute $\hat{\beta}_j \pm 2\sqrt{Var[\hat{\beta}_j]}$ to get ourselves a (roughly) 95% confidence interval. In our example the interval is

```
coefs.lm1[2,1] + c(-2,2) * coefs.lm1[2,2]
```

```
## [1] -0.4324108 -0.3540668
```

Wait! A confidence interval is something so basic! How can it not be already in R?

```
confint(lm.1)
```

```
##                 2.5 %     97.5 %
## (Intercept)  6.6094138  7.0982416
## Temp        -0.4336624 -0.3528153
```

6.3.3 Multiple Regression

Remark. Multiple regression is not to be confused with *multivariate regression* discussed in Chapter 9.

The `swiss` dataset encodes the fertility at each of Switzerland's 47 French speaking provinces, along other socio-economic indicators. Let's see if these are statistically related:

```
head(swiss)
```

```
##              Fertility Agriculture Examination Education Catholic
## Courtelary        80.2        17.0          15        12     9.96
## Delemont          83.1        45.1           6         9    84.84
## Franches-Mnt      92.5        39.7           5         5    93.40
## Moutier           85.8        36.5          12         7    33.77
## Neuveville        76.9        43.5          17        15     5.16
## Porrentruy        76.1        35.3           9         7    90.57
##              Infant.Mortality
## Courtelary               22.2
## Delemont                 22.2
## Franches-Mnt             20.2
## Moutier                  20.3
## Neuveville               20.6
## Porrentruy               26.6
```

```
lm.5 <- lm(data=swiss, Fertility~Agriculture+Examination+Education+Education+Catholic+Infant.Mortality)
summary(lm.5)
```

```
##
## Call:
## lm(formula = Fertility ~ Agriculture + Examination + Education +
##     Education + Catholic + Infant.Mortality, data = swiss)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -15.2743  -5.2617   0.5032   4.1198  15.3213
##
## Coefficients:
##                  Estimate Std. Error t value Pr(>|t|)
## (Intercept)      66.91518   10.70604   6.250 1.91e-07 ***
## Agriculture      -0.17211    0.07030  -2.448  0.01873 *
## Examination      -0.25801    0.25388  -1.016  0.31546
```

```
## Education        -0.87094    0.18303  -4.758 2.43e-05 ***
## Catholic          0.10412    0.03526   2.953  0.00519 **
## Infant.Mortality  1.07705    0.38172   2.822  0.00734 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 7.165 on 41 degrees of freedom
## Multiple R-squared:  0.7067, Adjusted R-squared:  0.671
## F-statistic: 19.76 on 5 and 41 DF,  p-value: 5.594e-10
```

Things to note:

- The ~ syntax allows to specify various predictors separated by the + operator.
- The summary of the model now reports the estimated effect, i.e., the regression coefficient, of each of the variables.

Clearly, naming each variable explicitely is a tedios task if there are many. The use of `Fertility~.` in the next example reads: "Fertility as a function of all other variables in the `swiss` data.frame".

```
lm.5 <- lm(data=swiss, Fertility~.)
summary(lm.5)
```

```
##
## Call:
## lm(formula = Fertility ~ ., data = swiss)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -15.2743  -5.2617   0.5032   4.1198  15.3213
##
## Coefficients:
##                  Estimate Std. Error t value Pr(>|t|)
## (Intercept)      66.91518   10.70604   6.250 1.91e-07 ***
## Agriculture      -0.17211    0.07030  -2.448  0.01873 *
## Examination      -0.25801    0.25388  -1.016  0.31546
## Education        -0.87094    0.18303  -4.758 2.43e-05 ***
## Catholic          0.10412    0.03526   2.953  0.00519 **
## Infant.Mortality  1.07705    0.38172   2.822  0.00734 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 7.165 on 41 degrees of freedom
## Multiple R-squared:  0.7067, Adjusted R-squared:  0.671
## F-statistic: 19.76 on 5 and 41 DF,  p-value: 5.594e-10
```

6.3.4 ANOVA (*)

Our next example[7] contains a hypothetical sample of 60 participants who are divided into three stress reduction treatment groups (mental, physical, and medical) and two gender groups (male and female). The stress reduction values are represented on a scale that ranges from 1 to 10. The values represent how effective the treatment programs were at reducing participant's stress levels, with larger effects indicating higher effectiveness.

```
twoWay <- read.csv('data/dataset_anova_twoWay_comparisons.csv')
head(twoWay)
```

```
##   Treatment   Age StressReduction
## 1    mental young              10
## 2    mental young               9
## 3    mental young               8
```

[7] The example is taken from http://rtutorialseries.blogspot.co.il/2011/02/r-tutorial-series-two-way-anova-with.html

```
## 4    mental   mid              7
## 5    mental   mid              6
## 6    mental   mid              5
```

How many observations per group?

```
table(twoWay$Treatment, twoWay$Age)
```

```
##
##            mid old young
##   medical    3   3     3
##   mental     3   3     3
##   physical   3   3     3
```

Since we have two factorial predictors, this multiple regression is nothing but a *two way ANOVA*. Let's fit the model and inspect it.

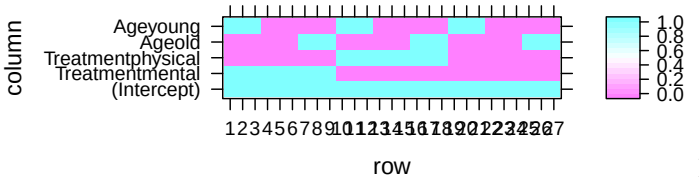
```
lm.2 <- lm(StressReduction~.,data=twoWay)
summary(lm.2)
```

```
##
## Call:
## lm(formula = StressReduction ~ ., data = twoWay)
##
## Residuals:
##    Min     1Q Median     3Q    Max
##     -1     -1      0      1      1
##
## Coefficients:
##                   Estimate Std. Error t value Pr(>|t|)
## (Intercept)         4.0000     0.3892  10.276 7.34e-10 ***
## Treatmentmental     2.0000     0.4264   4.690 0.000112 ***
## Treatmentphysical   1.0000     0.4264   2.345 0.028444 *
## Ageold             -3.0000     0.4264  -7.036 4.65e-07 ***
## Ageyoung            3.0000     0.4264   7.036 4.65e-07 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.9045 on 22 degrees of freedom
## Multiple R-squared:  0.9091, Adjusted R-squared:  0.8926
## F-statistic:    55 on 4 and 22 DF,  p-value: 3.855e-11
```

Things to note:

- The `StressReduction~.` syntax is read as "Stress reduction as a function of everything else".
- All the (main) effects and the intercept seem to be significant.
- The data has 2 factors, but the coefficients table has 4 predictors. This is because `lm` noticed that `Treatment` and `Age` are factors. Each level of each factor is thus encoded as a different (dummy) variable. The numerical values of the factors are meaningless. Instead, R has constructed a dummy variable for each level of each factor. The names of the effect are a concatenation of the factor's name, and its level. You can inspect these dummy variables with the `model.matrix` command.

```
model.matrix(lm.2) %>% lattice::levelplot()
```



If you don't want the default dummy coding, look at `?contrasts`.

If you are more familiar with the ANOVA literature, or that you don't want the effects of each level separately, but rather, the effect of **all** the levels of each factor, use the `anova` command.

```
anova(lm.2)
```

```
## Analysis of Variance Table
##
## Response: StressReduction
##           Df Sum Sq Mean Sq F value    Pr(>F)
## Treatment  2     18   9.000      11 0.0004883 ***
## Age        2    162  81.000      99     1e-11 ***
## Residuals 22     18   0.818
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Things to note:

- The ANOVA table, unlike the `summary` function, tests if **any** of the levels of a factor has an effect, and not one level at a time.
- The significance of each factor is computed using an F-test.
- The degrees of freedom, encoding the number of levels of a factor, is given in the `Df` column.
- The StressReduction seems to vary for different ages and treatments, since both factors are significant.

If you are extremely more comfortable with the ANOVA literature, you could have replaced the `lm` command with the `aov` command all along.

```
lm.2.2 <- aov(StressReduction~.,data=twoWay)
class(lm.2.2)
```

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

```
summary(lm.2.2)
```

```
##             Df Sum Sq Mean Sq F value   Pr(>F)
## Treatment    2     18    9.00      11 0.000488 ***
## Age          2    162   81.00      99    1e-11 ***
## Residuals   22     18    0.82
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Things to note:

- The `lm` function has been replaced with an `aov` function.
- The output of `aov` is an `aov` class object, which extends the `lm` class.
- The summary of an `aov` does not like the summary of an `lm` object, but rather, like an ANOVA table.

As in any two-way ANOVA, we may want to ask if different age groups respond differently to different treatments. In the statistical parlance, this is called an *interaction*, or more precisely, an *interaction of order 2*.

```
lm.3 <- lm(StressReduction~Treatment+Age+Treatment:Age-1,data=twoWay)
```

The syntax `StressReduction~Treatment+Age+Treatment:Age-1` tells R to include main effects of Treatment, Age, and their interactions. Here are other ways to specify the same model.

```
lm.3 <- lm(StressReduction ~ Treatment * Age - 1,data=twoWay)
lm.3 <- lm(StressReduction~(.)^2 - 1,data=twoWay)
```

The syntax `Treatment * Age` means "mains effects with second order interactions". The syntax `(.)^2` means "everything with second order interactions"

Let's inspect the model

```
summary(lm.3)
```

```
##
## Call:
```

```
## lm(formula = StressReduction ~ Treatment + Age + Treatment:Age -
##     1, data = twoWay)
##
## Residuals:
##    Min     1Q Median     3Q    Max
##     -1     -1      0      1      1
##
## Coefficients:
##                              Estimate Std. Error t value Pr(>|t|)
## Treatmentmedical            4.000e+00  5.774e-01   6.928 1.78e-06 ***
## Treatmentmental             6.000e+00  5.774e-01  10.392 4.92e-09 ***
## Treatmentphysical           5.000e+00  5.774e-01   8.660 7.78e-08 ***
## Ageold                     -3.000e+00  8.165e-01  -3.674  0.00174 **
## Ageyoung                    3.000e+00  8.165e-01   3.674  0.00174 **
## Treatmentmental:Ageold      4.246e-16  1.155e+00   0.000  1.00000
## Treatmentphysical:Ageold    1.034e-15  1.155e+00   0.000  1.00000
## Treatmentmental:Ageyoung   -3.126e-16  1.155e+00   0.000  1.00000
## Treatmentphysical:Ageyoung  5.128e-16  1.155e+00   0.000  1.00000
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1 on 18 degrees of freedom
## Multiple R-squared:  0.9794, Adjusted R-squared:  0.9691
## F-statistic:    95 on 9 and 18 DF,  p-value: 2.556e-13
```

Things to note:

- There are still 5 main effects, but also 4 interactions. This is because when allowing a different average response for every $Treatment * Age$ combination, we are effectively estimating $3 * 3 = 9$ cell means, even if they are not parametrized as cell means, but rather as main effect and interactions.
- The interactions do not seem to be significant.
- The assumptions required for inference are clearly not met in this example, which is there just to demonstrate R's capabilities.

Asking if all the interactions are significant, is asking if the different age groups have the same response to different treatments. Can we answer that based on the various interactions? We might, but it is possible that no single interaction is significant, while the combination is. To test for all the interactions together, we can simply check if the model without interactions is (significantly) better than a model with interactions. I.e., compare `lm.2` to `lm.3`. This is done with the `anova` command.

```
anova(lm.2,lm.3, test='F')
```

```
## Analysis of Variance Table
##
## Model 1: StressReduction ~ Treatment + Age
## Model 2: StressReduction ~ Treatment + Age + Treatment:Age - 1
##   Res.Df RSS Df   Sum of Sq F Pr(>F)
## 1     22  18
## 2     18  18  4 -3.5527e-15
```

We see that `lm.3` is **not** (significantly) better than `lm.2`, so that we can conclude that there are no interactions: different ages have the same response to different treatments.

6.3.5 Testing a Hypothesis on a Single Contrast (*)

Returning to the model without interactions, `lm.2`.

```
coef(summary(lm.2))
```

```
##                   Estimate Std. Error   t value     Pr(>|t|)
```

```
## (Intercept)              4  0.3892495 10.276186 7.336391e-10
## Treatmentmental          2  0.4264014  4.690416 1.117774e-04
## Treatmentphysical        1  0.4264014  2.345208 2.844400e-02
## Ageold                  -3  0.4264014 -7.035624 4.647299e-07
## Ageyoung                 3  0.4264014  7.035624 4.647299e-07
```

We see that the effect of the various treatments is rather similar. It is possible that all treatments actually have the same effect. Comparing the effects of factor levels is called a *contrast*. Let's test if the medical treatment, has in fact, the same effect as the physical treatment.

```
library(multcomp)
my.contrast <- matrix(c(-1,0,1,0,0), nrow =  1)
lm.4 <- glht(lm.2, linfct=my.contrast)
summary(lm.4)
```

```
##
##   Simultaneous Tests for General Linear Hypotheses
##
## Fit: lm(formula = StressReduction ~ ., data = twoWay)
##
## Linear Hypotheses:
##        Estimate Std. Error t value Pr(>|t|)
## 1 == 0  -3.0000     0.7177   -4.18 0.000389 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
## (Adjusted p values reported -- single-step method)
```

Things to note:

- A contrast is a linear function of the coefficients. In our example $H_0 : \beta_1 - \beta_3 = 0$, which justifies the construction of `my.contrast`.
- We used the `glht` function (generalized linear hypothesis test) from the package **multcompt**.
- The contrast is significant, i.e., the effect of a medical treatment, is different than that of a physical treatment.

6.4 Bibliographic Notes

Like any other topic in this book, you can consult Venables and Ripley (2013) for more on linear models. For the theory of linear models, I like Greene (2003).

6.5 Practice Yourself

1. Inspect women's heights and weights with `?women`.
    1. What is the change in weight per unit change in height? Use the `lm` function.
    2. Is the relation of height on weight significant? Use `summary`.
    3. Plot the residuals of the linear model with `plot` and `resid`.
    4. Plot the predictions of the model using `abline`.
    5. Inspect the normality of residuals using `qqnorm`.
    6. Inspect the design matrix using `model.matrix`.
2. Write a function that takes an `lm` class object, and returns the confidence interval on the first coefficient. Apply it on the height and weight data.
3. Use the `ANOVA` function to test the significance of the effect of height.
4. Read about the "mtcars" dataset using `? mtcars`.Inspect the dependency of the fuel consumption (mpg) in the weight (wt) and the 1/4 mile time (qsec).
    1. Make a pairs scatter plot with `plot(mtcars[,c("mpg","wt","qsec")])` Does the connection look linear?

2. Fit a multiple linear regression with `lm`. Call it `model1`.
3. Try to add the transmission (am) as independent variable. Let R know this is a categorical variable with `factor(am)`. Call it `model2`.
4. Compare the "Adjusted R-squared" measure of the two models (we can't use the regular R2 to compare two models with a different number of variables).
5. Do the coefficients significant?
6. Inspect the normality of residuals and the linearity assumptions.
7. Now Inspect the hypothesis that the effect of weight is different between the transmission types with adding interaction to the model `wt*factor(am)`.
8. According to this model, what is the addition of one unit of weight in a manual transmission to the fuel consumption (-2.973-4.141=-7.11)?

Chapter 7

Generalized Linear Models

Example 7.1. Consider the relation between cigarettes smoked, and the occurance of lung cancer. Do we expect the probability of cancer to be linear in the number of cigarettes? Probably not. Do we expect the variability of events to be constant about the trend? Probably not.

Example 7.2. Consider the relation between the travel times to the distance travelled. Do you agree that the longer the distance travelled, then not only the travel times get longer, but they also get more variable?

7.1 Problem Setup

In the Linear Models Chapter 6, we assumed the generative process to be linear in the effects of the predictors x. We now write that same linear model, slightly differently:

$$y|x \sim \mathcal{N}(x'\beta, \sigma^2).$$

This model not allow for the non-linear relations of Example 7.1, nor does it allow for the distrbituion of ε to change with x, as in Example 7.2. *Generalize linear models* (GLM), as the name suggests, are a generalization of the linear models in Chapter 6 that allow that[1].

For Example 7.1, we would like something in the lines of

$$y|x \sim Binom(1, p(x))$$

For Example 7.2, we would like something in the lines of

$$y|x \sim \mathcal{N}(x'\beta, \sigma^2(x)),$$

or more generally

$$y|x \sim \mathcal{N}(\mu(x), \sigma^2(x)),$$

or maybe not Gaussian

$$y|x \sim Pois(\lambda(x)).$$

Even more generally, for some distribution $F(\theta)$, with a parameter θ, we would like to assume that the data is generated via

$$y|x \sim F(\theta(x)) \tag{7.1}$$

Possible examples include

[1] Do not confuse *generalized linear models* with *non-linear regression*, or *generalized least squares*. These are different things, that we do not discuss.

$$y|x \sim Poisson(\lambda(x)) \tag{7.2}$$
$$y|x \sim Exp(\lambda(x)) \tag{7.3}$$
$$y|x \sim \mathcal{N}(\mu(x), \sigma^2(x)) \tag{7.4}$$

GLMs allow models of the type of Eq.(7.1), while imposing some constraints on F and on the relation $\theta(x)$. GLMs assume the data distribution F to be in a "well-behaved" family known as the *Natural Exponential Family* of distributions. This family includes the Gaussian, Gamma, Binomial, Poisson, and Negative Binomial distributions. These five include as special cases the exponential, chi-squared, Rayleigh, Weibull, Bernoulli, and geometric distributions.

GLMs also assume that the distribution's parameter, θ, is some simple function of a linear combination of the effects. In our cigarettes example this amounts to assuming that each cigarette has an additive effect, but not on the probability of cancer, but rather, on some simple function of it. Formally

$$g(\theta(x)) = x'\beta,$$

and we recall that

$$x'\beta = \beta_0 + \sum_j x_j\beta_j.$$

The function g is called the *link* function, its inverse, g^{-1} is the *mean function*. We thus say that "the effects of each cigarette is linear **in link scale**". This terminology will later be required to understand R's output.

7.2 Logistic Regression

The best known of the GLM class of models is the *logistic regression* that deals with Binomial, or more precisely, Bernoulli-distributed data. The link function in the logistic regression is the *logit function*

$$g(t) = log\left(\frac{t}{(1-t)}\right) \tag{7.5}$$

implying that under the logistic model assumptions

$$y|x \sim Binom\left(1, p = \frac{e^{x'\beta}}{1+e^{x'\beta}}\right). \tag{7.6}$$

Before we fit such a model, we try to justify this construction, in particular, the enigmatic link function in Eq.(7.5). Let's look at the simplest possible case: the comparison of two groups indexed by x: $x = 0$ for the first, and $x = 1$ for the second. We start with some definitions.

Definition 7.1 (Odds)**.** The *odds*, of a binary random variable, y, is defined as

$$\frac{P(y=1)}{P(y=0)}.$$

Odds are the same as probabilities, but instead of telling me there is a 66% of success, they tell me the odds of success are "2 to 1". If you ever placed a bet, the language of "odds" should not be unfamiliar to you.

Definition 7.2 (Odds Ratio)**.** The *odds ratio* between two binary random variables, y_1 and y_2, is defined as the ratio between their odds. Formally:

$$OR(y_1, y_2) := \frac{P(y_1=1)/P(y_1=0)}{P(y_2=1)/P(y_2=0)}.$$

Odds ratios (OR) compare between the probabilities of two groups, only that it does not compare them in probability scale, but rather in odds scale. You can also think of ORs as a measure of distance between two Brenoulli distributions. ORs have better mathematical properties than other candidate distance measures, such as $P(y_1 = 1) - P(y_2 = 1)$.

Under the logit link assumption formalized in Eq.(7.6), the OR between two conditions indexed by $y|x = 1$ and $y|x = 0$, returns:

$$OR(y|x = 1, y|x = 0) = \frac{P(y = 1|x = 1)/P(y = 0|x = 1)}{P(y = 1|x = 0)/P(y = 0|x = 0)} = e^{\beta_1}. \tag{7.7}$$

The last equality demystifies the choice of the link function in the logistic regression: **it allows us to interpret β of the logistic regression as a measure of change of binary random variables, namely, as the (log) odds-ratios due to a unit increase in x.**
Remark. Another popular link function is the normal quantile function, a.k.a., the Gaussian inverse CDF, leading to *probit regression* instead of logistic regression.

7.2.1 Logistic Regression with R

Let's get us some data. The `PlantGrowth` data records the weight of plants under three conditions: control, treatment1, and treatment2.

```
head(PlantGrowth)
```

```
##   weight group
## 1   4.17  ctrl
## 2   5.58  ctrl
## 3   5.18  ctrl
## 4   6.11  ctrl
## 5   4.50  ctrl
## 6   4.61  ctrl
```

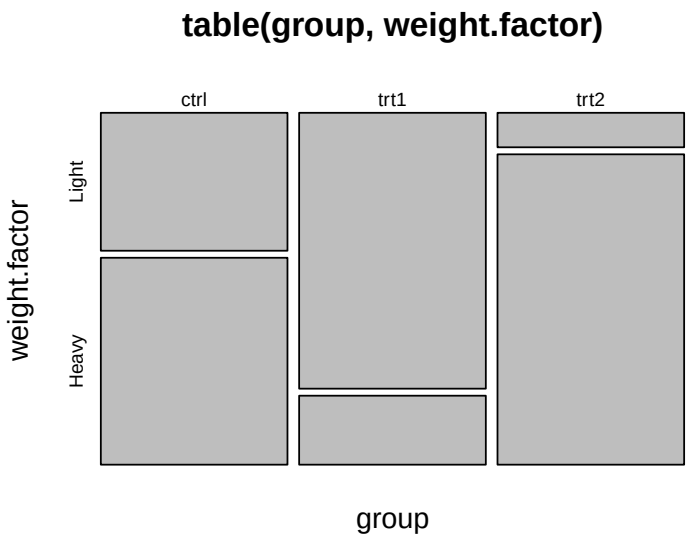
We will now `attach` the data so that its contents is available in the workspace (don't forget to `detach` afterwards, or you can expect some conflicting object names). We will also use the `cut` function to create a binary response variable for Light, and Heavy plants (we are doing logistic regression, so we need a two-class response). As a general rule of thumb, when we discretize continuous variables, we lose information. For pedagogical reasons, however, we will proceed with this bad practice.

Look at the following output and think: how many `group` effects do we expect? What should be the sign of each effect?

```
attach(PlantGrowth)
```

```
## The following objects are masked from PlantGrowth (pos = 3):
##
##     group, weight
```

```
weight.factor<- cut(weight, 2, labels=c('Light', 'Heavy')) # binarize weights
plot(table(group, weight.factor))
```



Let's fit a logistic regression, and inspect the output.

```
glm.1<- glm(weight.factor~group, family=binomial)
summary(glm.1)
```

```
##
## Call:
## glm(formula = weight.factor ~ group, family = binomial)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -2.1460  -0.6681   0.4590   0.8728   1.7941
##
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept)   0.4055     0.6455   0.628   0.5299
## grouptrt1    -1.7918     1.0206  -1.756   0.0792 .
## grouptrt2     1.7918     1.2360   1.450   0.1471
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 41.054  on 29  degrees of freedom
## Residual deviance: 29.970  on 27  degrees of freedom
## AIC: 35.97
##
## Number of Fisher Scoring iterations: 4
```

Things to note:

- The `glm` function is our workhorse for all GLM models.
- The `family` argument of `glm` tells R the respose variable is brenoulli, thus, performing a logistic regression.
- The `summary` function is content aware. It gives a different output for `glm` class objects than for other objects, such as the `lm` we saw in Chapter 6. In fact, what `summary` does is merely call `summary.glm`.
- As usual, we get the coefficients table, but recall that they are to be interpreted as (log) odd-ratios, i.e., in "link scale". To return to probabilities ("response scale"), we will need to undo the logistic transformation.
- As usual, we get the significance for the test of no-effect, versus a two-sided alternative. P-values are asymptotic, thus, only approximate (and can be very bad approximations in small samples).
- The residuals of `glm` are slightly different than the `lm` residuals, and called *Deviance Residuals*.
- For help see `?glm`, `?family`, and `?summary.glm`.

Like in the linear models, we can use an ANOVA table to check if treatments have any effect, and not one treatment

at a time. In the case of GLMs, this is called an *analysis of deviance* table.

```
anova(glm.1, test='LRT')
```

```
## Analysis of Deviance Table
##
## Model: binomial, link: logit
##
## Response: weight.factor
##
## Terms added sequentially (first to last)
##
##
##       Df Deviance Resid. Df Resid. Dev Pr(>Chi)
## NULL                     29     41.054
## group  2   11.084        27     29.970 0.003919 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Things to note:

- The `anova` function, like the `summary` function, are content-aware and produce a different output for the `glm` class than for the `lm` class. All that `anova` does is call `anova.glm`.
- In GLMs there is no canonical test (like the F test for `lm`). `LRT` implies we want an approximate Likelihood Ratio Test. We thus specify the type of test desired with the `test` argument.
- The distribution of the weights of the plants does vary with the treatment given, as we may see from the significance of the `group` factor.
- Readers familiar with ANOVA tables, should know that we computed the GLM equivalent of a type I sum-of-squares. Run `drop1(glm.1, test='Chisq')` for a GLM equivalent of a type III sum-of-squares.
- For help see `?anova.glm`.

Let's predict the probability of a heavy plant for each treatment.

```
predict(glm.1, type='response')
```

```
##   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18
## 0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.2 0.2 0.2 0.2 0.2 0.2 0.2 0.2
##  19  20  21  22  23  24  25  26  27  28  29  30
## 0.2 0.2 0.9 0.9 0.9 0.9 0.9 0.9 0.9 0.9 0.9 0.9
```

Things to note:

- Like the `summary` and `anova` functions, the `predict` function is aware that its input is of `glm` class. All that `predict` does is call `predict.glm`.
- In GLMs there are many types of predictions. The `type` argument controls which type is returned. Use `type=response` for predictions in probability scale; use 'type=link' for predictions in log-odds scale.
- How do I know we are predicting the probability of a heavy plant, and not a light plant? Just run `contrasts(weight.factor)` to see which of the categories of the factor `weight.factor` is encoded as 1, and which as 0.
- For help see `?predict.glm`.

Let's detach the data so it is no longer in our workspace, and object names do not collide.

```
detach(PlantGrowth)
```

We gave an example with a factorial (i.e. discrete) predictor. We can do the same with multiple continuous predictors.

```
data('Pima.te', package='MASS') # Loads data
head(Pima.te)
```

```
##   npreg glu bp skin  bmi   ped age type
## 1     6 148 72   35 33.6 0.627  50  Yes
## 2     1  85 66   29 26.6 0.351  31   No
## 3     1  89 66   23 28.1 0.167  21   No
```

```
## 4     3  78 50   32 31.0 0.248  26  Yes
## 5     2 197 70   45 30.5 0.158  53  Yes
## 6     5 166 72   19 25.8 0.587  51  Yes
```

```
glm.2<- step(glm(type~., data=Pima.te, family=binomial))
```

```
## Start:  AIC=301.79
## type ~ npreg + glu + bp + skin + bmi + ped + age
##
##         Df Deviance    AIC
## - skin   1   286.22 300.22
## - bp     1   286.26 300.26
## - age    1   286.76 300.76
## <none>       285.79 301.79
## - npreg  1   291.60 305.60
## - ped    1   292.15 306.15
## - bmi    1   293.83 307.83
## - glu    1   343.68 357.68
##
## Step:  AIC=300.22
## type ~ npreg + glu + bp + bmi + ped + age
##
##         Df Deviance    AIC
## - bp     1   286.73 298.73
## - age    1   287.23 299.23
## <none>       286.22 300.22
## - npreg  1   292.35 304.35
## - ped    1   292.70 304.70
## - bmi    1   302.55 314.55
## - glu    1   344.60 356.60
##
## Step:  AIC=298.73
## type ~ npreg + glu + bmi + ped + age
##
##         Df Deviance    AIC
## - age    1   287.44 297.44
## <none>       286.73 298.73
## - npreg  1   293.00 303.00
## - ped    1   293.35 303.35
## - bmi    1   303.27 313.27
## - glu    1   344.67 354.67
##
## Step:  AIC=297.44
## type ~ npreg + glu + bmi + ped
##
##         Df Deviance    AIC
## <none>       287.44 297.44
## - ped    1   294.54 302.54
## - bmi    1   303.72 311.72
## - npreg  1   304.01 312.01
## - glu    1   349.80 357.80
```

```
summary(glm.2)
```

```
##
## Call:
## glm(formula = type ~ npreg + glu + bmi + ped, family = binomial,
##     data = Pima.te)
##
```

```
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -2.9845  -0.6462  -0.3661   0.5977   2.5304
##
## Coefficients:
##              Estimate Std. Error z value Pr(>|z|)
## (Intercept) -9.552177   1.096207  -8.714  < 2e-16 ***
## npreg        0.178066   0.045343   3.927  8.6e-05 ***
## glu          0.037971   0.005442   6.978  3.0e-12 ***
## bmi          0.084107   0.021950   3.832 0.000127 ***
## ped          1.165658   0.444054   2.625 0.008664 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 420.30  on 331  degrees of freedom
## Residual deviance: 287.44  on 327  degrees of freedom
## AIC: 297.44
##
## Number of Fisher Scoring iterations: 5
```

Things to note:

- We used the `~.` syntax to tell R to fit a model with all the available predictors.
- Since we want to focus on significant predictors, we used the `step` function to perform a *step-wise* regression, i.e. sequentially remove non-significant predictors. The function reports each model it has checked, and the variable it has decided to remove at each step.
- The output of `step` is a single model, with the subset of selected predictors.

7.3 Poisson Regression

Poisson regression means we fit a model assuming $y|x \sim Poisson(\lambda(x))$. Put differently, we assume that for each treatment, encoded as a combinations of predictors x, the response is Poisson distributed with a rate that depends on the predictors.

The typical link function for Poisson regression is the logarithm: $g(t) = log(t)$. This means that we assume $y|x \sim Poisson(\lambda(x) = e^{x'\beta})$. Why is this a good choice? We again resort to the two-group case, encoded by $x = 1$ and $x = 0$, to understand this model: $\lambda(x = 1) = e^{\beta_0+\beta_1} = e^{\beta_0}\ e^{\beta_1} = \lambda(x = 0)\ e^{\beta_1}$. We thus see that this link function implies that a change in x **multiples** the rate of events by e^{β_1}.

For our example[2] we inspect the number of infected high-school kids, as a function of the days since an outbreak.

```
cases <-
structure(list(Days = c(1L, 2L, 3L, 3L, 4L, 4L, 4L, 6L, 7L, 8L,
8L, 8L, 8L, 12L, 14L, 15L, 17L, 17L, 17L, 18L, 19L, 19L, 20L,
23L, 23L, 23L, 24L, 24L, 25L, 26L, 27L, 28L, 29L, 34L, 36L, 36L,
42L, 42L, 43L, 43L, 44L, 44L, 44L, 44L, 45L, 46L, 48L, 48L, 49L,
49L, 53L, 53L, 53L, 54L, 55L, 56L, 56L, 58L, 60L, 63L, 65L, 67L,
67L, 68L, 71L, 71L, 72L, 72L, 72L, 73L, 74L, 74L, 74L, 75L, 75L,
80L, 81L, 81L, 81L, 81L, 88L, 88L, 90L, 93L, 93L, 94L, 95L, 95L,
95L, 96L, 96L, 97L, 98L, 100L, 101L, 102L, 103L, 104L, 105L,
106L, 107L, 108L, 109L, 110L, 111L, 112L, 113L, 114L, 115L),
    Students = c(6L, 8L, 12L, 9L, 3L, 3L, 11L, 5L, 7L, 3L, 8L,
    4L, 6L, 8L, 3L, 6L, 3L, 2L, 2L, 6L, 3L, 7L, 7L, 2L, 2L, 8L,
    3L, 6L, 5L, 7L, 6L, 4L, 4L, 3L, 3L, 5L, 3L, 3L, 3L, 5L, 3L,
    5L, 6L, 3L, 3L, 3L, 3L, 2L, 3L, 1L, 3L, 3L, 5L, 4L, 4L, 3L,
```

[2] Taken from http://www.theanalysisfactor.com/generalized-linear-models-in-r-part-6-poisson-regression-count-variables/

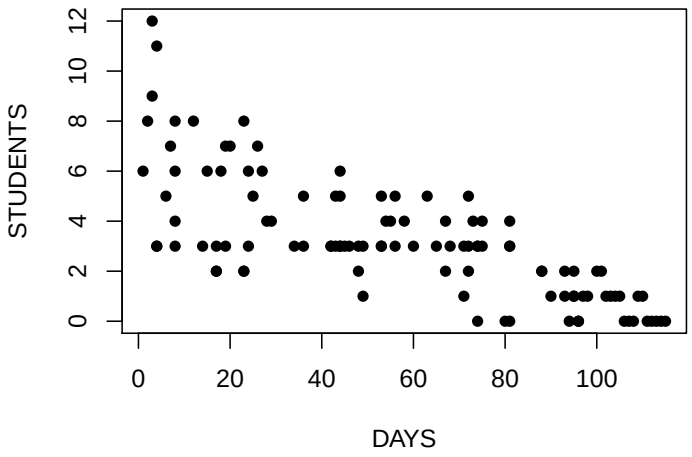
```
    5L, 4L, 3L, 5L, 3L, 4L, 2L, 3L, 3L, 1L, 3L, 2L, 5L, 4L, 3L,
    0L, 3L, 3L, 4L, 0L, 3L, 3L, 4L, 0L, 2L, 2L, 1L, 1L, 2L, 0L,
    2L, 1L, 1L, 0L, 0L, 1L, 1L, 2L, 2L, 1L, 1L, 1L, 1L, 0L, 0L,
    0L, 1L, 1L, 0L, 0L, 0L, 0L, 0L)), .Names = c("Days", "Students"
), class = "data.frame", row.names = c(NA, -109L))
attach(cases)
head(cases)
```

```
##   Days Students
## 1    1        6
## 2    2        8
## 3    3       12
## 4    3        9
## 5    4        3
## 6    4        3
```

Look at the following plot and think:

- Can we assume that the errors have constant variace?
- What is the sign of the effect of time on the number of sick students?
- Can we assume a linear effect of time?

```
plot(Days, Students, xlab = "DAYS", ylab = "STUDENTS", pch = 16)
```



We now fit a model to check for the change in the rate of events as a function of the days since the outbreak.

```
glm.3 <- glm(Students ~ Days, family = poisson)
summary(glm.3)
```

```
##
## Call:
## glm(formula = Students ~ Days, family = poisson)
##
## Deviance Residuals:
##      Min        1Q    Median        3Q       Max
## -2.00482  -0.85719  -0.09331   0.63969   1.73696
##
## Coefficients:
##              Estimate Std. Error z value Pr(>|z|)
## (Intercept)  1.990235   0.083935   23.71   <2e-16 ***
## Days        -0.017463   0.001727  -10.11   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for poisson family taken to be 1)
```

```
##
##     Null deviance: 215.36  on 108  degrees of freedom
## Residual deviance: 101.17  on 107  degrees of freedom
## AIC: 393.11
##
## Number of Fisher Scoring iterations: 5
```

Things to note:

- We used `family=poisson` in the `glm` function to tell R that we assume a Poisson distribution.
- The coefficients table is there as usual. When interpreting the table, we need to recall that the effect, i.e. the $\hat{\beta}$, are **multiplicative** due to the assumed link function.
- Each day **decreases** the rate of events by a factor of about $e^{\beta_1} = 0.02$.
- For more information see `?glm` and `?family`.

7.4 Extensions

7.4.1 Probit Regression and Changing the Link Function

Probit regression is the same as *logistic regression*, only using a different link function.

```
attach(PlantGrowth)
```

```
## The following objects are masked from PlantGrowth (pos = 3):
##
##     group, weight
```

```
glm.1.2<- glm(weight.factor~group, family=binomial(link = "probit"))
summary(glm.1.2)
```

```
##
## Call:
## glm(formula = weight.factor ~ group, family = binomial(link = "probit"))
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -2.1460  -0.6681   0.4590   0.8728   1.7941
##
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept)   0.2533     0.4010   0.632   0.5275
## grouptrt1    -1.0950     0.6041  -1.813   0.0699 .
## grouptrt2     1.0282     0.6730   1.528   0.1266
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 41.054  on 29  degrees of freedom
## Residual deviance: 29.970  on 27  degrees of freedom
## AIC: 35.97
##
## Number of Fisher Scoring iterations: 4
```

```
detach(PlantGrowth)
```

Things to note:

- Like logistic regression, we use the `glm` function, with the `family=binomial` argument.

- Unlike the logistic regression, we explicitly use the `link=` argument of the family, to force R to use the probit link, instead of the default logistic link. Not all links are supported by all families. See `?family` for more.

7.4.2 Marginal Effects

In linear models one may ask the question "what is the effect of some predictor `x`?", and get answered with a number. In non-linear models, the answer to that same question is "depends what are the values of the other variables". One convention that simplifies this matter, is to report the effect of `x` at some average value of the other predictors. If this matter interestes you, see the margins package.

7.5 Bibliographic Notes

The ultimate reference on GLMs is McCullagh (1984). For a less technical exposition, we refer to the usual Venables and Ripley (2013).

7.6 Practice Yourself

1. Try using `lm` for analyzing the plant growth data in `weight.factor` as a function of `group` in the `PlantGrowth` data.
2. Generate some synthetic data for a logistic regression:
   a. Generate two predictor variables of length 100. They can be random from your favorite distribution.
   b. Fix `beta<- c(-1,2)`, and generate the response with:`rbinom(n=100,size=1,prob=exp(x %*% beta)/(1+exp(x %*% beta)))`. Think: why is this the model implied by the logistic regression?
   c. Fit a Logistic regression to your synthetic data using `glm`.
   d. Are the estimated coefficients similar to the true ones you used?
   e. What is the estimated probability of an event at `x=1,1`? Use `predict.glm` but make sure to read the documentation on the `type` argument.
3. Read about the `epil` dataset using `? MASS::epil`. Inspect the dependency of the number of seizures (y) in the age of the patient (`age`) and the treatment (`trt`).
   1. Fit a Poisson regression with `glm` and `family = "poisson"`.
   2. Are the coefficients significant?
   3. Does the treatment reduce the frequency of the seizures?
   4. According to this model, what would be the number of seizures for 20 years old patient with progabide treatment?

Chapter 8

Linear Mixed Models

Example 8.1 (Dependent Samples on the Mean)**.** Consider inference on a population's mean. Supposdly, more observations imply more infotmation on the mean. This, however, is not the case if samples are completely dependant. More observations do not add any new information. From this example one may think that dependence is a bad thing. This is a false intuitiont: negative correlations imply oscilations about the mean, so they are actually more informative on the mean than independent observations.

Example 8.2 (Repeated Measures)**.** Consider a prospective study, i.e., data that originates from selecting a set of subjects and making measurements on them over time. Also assume that some subjects received some treatment, and other did not. When we want to infer on the population from which these subjects have been sampled, we need to recall that some series of observations came from the same subject. If we were to ignore the subject of origin, and treat each observation as an independent sample point, we will think we have more information in our data than we actually do. For a rough intuition, think of a case where observatiosn within subject are perfectly dependent.

The sources of variability, i.e. noise, are known in the statistical literature as "random effects". Specifying these sources determines the correlation structure in our measurements. In the simplest linear models of Chapter 6, we thought of the variability as a measurement error, independent of anything else. This, however, is rarely the case when time or space are involved.

The variability in our data is rarely the object of interest. It is merely the source of uncertainty in our measurements. The effects we want to infer on are assumingly non-random, thus known as "fixed-effects". A model which has several sources of variability, i.e. random-effects, and several deterministic effects to study, i.e. fixed-effects, is known as a "mixed effects" model. If the model is also linear, it is known as a *linear mixed model* (LMM). Here are some examples of such models.

Example 8.3 (Fixed and Random Machine Effect)**.** Consider the problem of testing for a change in the distribution of diamteters of manufactured bottle caps. We want to study the (fixed) effect of time: before versus after. Bottle caps are produced by several machines. Clearly there is variablity in the diameters within-machine and between-machines. Given many measurements on many bottle caps from many machines, we could standardize measurements by removing each machine's average. This implies the within-machine variability is the only source of variability we care about, because the substration of the machine effect, removed information on the between-machine variability.
Alternatively, we could treat the between-machine variability as another source of noise/uncertainty when inferring on the temporal fixed effect.

Example 8.4 (Fixed and Random Subject Effect)**.** Consider an experimenal design where each subject is given 2 types of diets, and his health condition is recorded. We could standardize over subjects by removing the subject-wise average, before comparing diets. This is what a paired t-test does. This also implies the within-subject variability is the only source of variability we care about. Alternatively, for inference on the population of "all subjects" we need to adress the between-subject variability, and not only the within-subject variability.

The unifying theme of the above examples, is that the variability in our data has several sources. Which are the sources of variability that need to concern us? This is a delicate matter which depends on your goals. As a rule of thumb, we will suggest the following view: **If information of an effect will be available at the time of prediction, treat it as a fixed effect. If it is not, treat it as a random-effect.**

LMMs are so fundamental, that they have earned many names:

- **Mixed Effects**: Because we may have both *fixed effects* we want to estimate and remove, and *random effects* which contribute to the variability to infer against.
- **Variance Components**: Because as the examples show, variance has more than a single source (like in the Linear Models of Chapter 6).
- **Hirarchial Models**: Because as Example 8.4 demonstrates, we can think of the sampling as hierarchical– first sample a subject, and then sample its response.
- **Multilevel Analysis**: For the same reasons it is also known as Hierarchical Models.
- **Repeated Measures**: Because we make several measurements from each unit, like in Example 8.4.
- **Longitudinal Data**: Because we follow units over time, like in Example 8.4.
- **Panel Data**: Is the term typically used in econometric for such longitudinal data.
- **MANOVA**: Many of the problems that may be solved with a multivariate analysis of variance (MANOVA), may be solved with an LMM for reasons we detail in 9.
- **Structured Prediction**: In the machine learning literature, predicting outcomes with structure, such as correlated vectors, is known as Structured Learning. Because LMMs merely specify correlations, using a LMM for making predictions may be thought of as an instance of structured prediction.

Whether we are aiming to infer on a generative model's parameters, or to make predictions, there is no "right" nor "wrong" approach. Instead, there is always some implied measure of error, and an algorithm may be good, or bad, with respect to this measure (think of false and true positives, for instance). This is why we care about dependencies in the data: ignoring the dependence structure will probably yield inefficient algorithms. Put differently, if we ignore the statistical dependence in the data we will probably me making more errors than possible/optimal.

We now emphasize:

1. Like in previous chapters, by "model" we refer to the assumed generative distribution, i.e., the sampling distribution.
2. LMMs are a way to infer against the right level of variability. Using a naive linear model (which assumes a single source of variability) instead of a mixed effects model, probably means your inference is overly anti-conservative. Put differently, the uncertainty in your estimates is higher than the linear model from Chapter 6 may suggest.
3. In a LMM we will specify the dependence structure via the hierarchy in the sampling scheme (e.g. caps within machine, students within class, etc.). Not all dependency models can be specified in this way. Dependency structures that are not hierarchical include temporal dependencies (AR, ARIMA, ARCH and GARCH), spatial, Markov Chains, and more. To specify dependency structures that are no hierarchical, see Chapter 8 in (the excellent) Weiss (2005).
4. If you are using the model merely for predictions, and not for inference on the fixed effects or variance components, then stating the generative distribution may be be useful, but not necessarily. See the Supervised Learning Chapter **??** for more on prediction problems. Also recall that machine learning from non-independent observations (such as LMMs) is a delicate matter that is rarely treated in the literature.

8.1 Problem Setup

$$y|x,u = x'\beta + z'u + \varepsilon \tag{8.1}$$

where x are the factors with fixed effects, β, which we may want to study. The factors z, with effects u, are the random effects which contribute to variability. In our repeated measures example (8.2) the treatment is a fixed effect, and the subject is a random effect. In our bottle-caps example (8.3) the time (before vs. after) is a fixed effect, and the machines may be either a fixed or a random effect (depending on the purpose of inference). In our diet example (8.4) the diet is the fixed effect and the family is a random effect.

Notice that we state $y|x,z$ merely as a convenient way to do inference on $y|x$, instead of directly specifying $Var[y|x]$. This is exactly the power of LMMs: we specify the covariance not via the matrix $Var[y,z]$, but rather via the sampling hierarchy.

Given a sample of n observations (y_i, x_i, z_i) from model (8.1), we will want to estimate (β, u). Under some assumption on the distribution of ε and z, we can use *maximum likelihood* (ML). In the context of LMMs, however, ML is typically replaced with *restricted maximum likelihood* (ReML), because it returns unbiased estimates of $Var[y|x]$ and ML does not.

8.1.1 Non-Linear Mixed Models

The idea of random-effects can also be implemented for non-linear mean models. Formally, this means that $y|x,z = f(x,z,\varepsilon)$ for some non-linear f. This is known as *non-linead-mixed-models*, which will not be discussed in this text.

8.1.2 Generalized Linear Mixed Models (GLMM)

You can marry the ideas of random effects, with non-linear link functions, and non-Gaussian distribution of the response. These are known as Generalized Linear Mixed Models. Wikidot has a nice comparison of several software suits for GLMMs.

8.2 Mixed Models with R

We will fit mixed models with the `lmer` function from the **lme4** package, written by the mixed-models Guru Douglas Bates. We start with a small simulation demonstrating the importance of acknowledging your sources of variability. Our demonstration consists of fitting a linear model that assumes independence, when data is clearly dependent.

```
# Simulation parameters
n.groups <- 4 # number of groups
n.repeats <- 2 # sample per group
groups <- rep(1:n.groups, each=n.repeats) %>% as.factor
n <- length(groups)
z0 <- rnorm(n.groups,0,10) # generate group effects
(z <- z0[as.numeric(groups)]) # generate and inspect random group effects
```

```
## [1]   8.901364   8.901364  -4.318889  -4.318889   9.708611   9.708611
## [7] -10.693773 -10.693773
```

```
epsilon <- rnorm(n,0,1) # generate measurement error

# Generate data
beta0 <- 2 # set global mean
y <- beta0 + z + epsilon # generate synthetic sample
```

We can now fit the linear and mixed models.

```
lm.5 <- lm(y~z)  # fit a linear model assuming independence
library(lme4)
lme.5 <- lmer(y~1|groups) # fit a mixed-model that deals with the group dependence
```

The summary of the linear model

```
summary.lm.5 <- summary(lm.5)
summary.lm.5
```

```
##
## Call:
## lm(formula = y ~ z)
##
```

```
## Residuals:
##      Min       1Q   Median       3Q      Max
## -1.69518 -0.74814  0.06391  0.78223  1.35721
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  2.36672    0.41639   5.684  0.00128 **
## z            1.05698    0.04757  22.221 5.43e-07 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.171 on 6 degrees of freedom
## Multiple R-squared:  0.988,  Adjusted R-squared:  0.986
## F-statistic: 493.8 on 1 and 6 DF,  p-value: 5.432e-07
```

The summary of the mixed-model

```
summary.lme.5 <- summary(lme.5)
summary.lme.5
```

```
## Linear mixed model fit by REML ['lmerMod']
## Formula: y ~ 1 | groups
##
## REML criterion at convergence: 41
##
## Scaled residuals:
##      Min       1Q   Median       3Q      Max
## -1.15395 -0.50048  0.04306  0.55891  0.99797
##
## Random effects:
##  Groups   Name        Variance Std.Dev.
##  groups   (Intercept) 111.962  10.581
##  Residual               2.012   1.418
## Number of obs: 8, groups:  groups, 4
##
## Fixed effects:
##             Estimate Std. Error t value
## (Intercept)    3.317      5.314   0.624
```

Look at the standard error of the global mean, i.e., the intercept: for `lm` it is 0.4163865, and for `lme` it is 5.3143276. Why this difference? Because `lm` treats the group effect[1] as a fixed while the mixed model treats the group effect as a source of noise/uncertainty. Clearly, inference using `lm` underestimates our uncertainty in the estimated population mean (β_0).

Now let's adopt the paired t-test view, which removes the group mean, so that it implicitly ignores the between-group variability. Which is the model compatible with this view?

```
diffs <- tapply(y, groups, diff)
diffs # Q:what is this estimating? A: epsilon+epsilon.
```

```
##         1         2         3         4
## -1.411024 -1.598983 -1.493730  3.052394
```

```
sd(diffs) #
```

```
## [1] 2.278119
```

So we see that a paired t-test infers only against the within-group variability. Q:Is this a good think? A: depends…

[1] A.k.a. the *cluster effect*.

8.2.1 A Single Random Effect

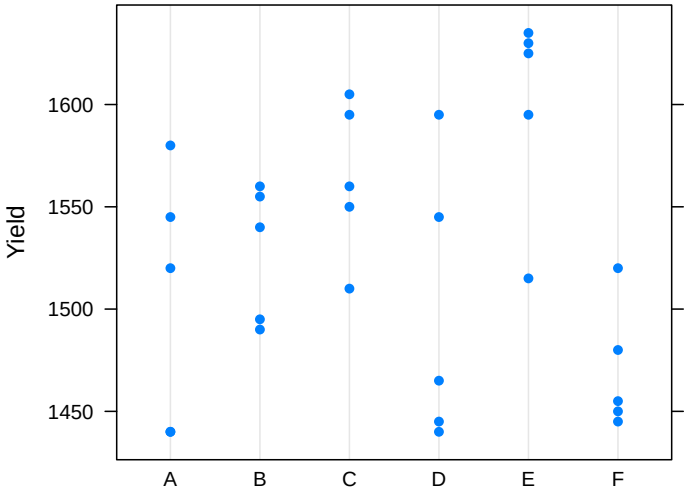
We will use the `Dyestuff` data from the **lme4** package, which encodes the yield, in grams, of a coloring solution (`dyestuff`), produced in 6 batches using 5 different preparations.

```
data(Dyestuff, package='lme4')
attach(Dyestuff)
head(Dyestuff)
```

```
##   Batch Yield
## 1     A  1545
## 2     A  1440
## 3     A  1440
## 4     A  1520
## 5     A  1580
## 6     B  1540
```

And visually

```
lattice::dotplot(Yield~Batch)
```



If we want to do inference on the (global) mean yield, we need to account for the two sources of variability: the within-batch variability, and the between-batch variability We thus fit a mixed model, with an intercept and random batch effect.

```
lme.1<- lmer( Yield ~ 1  | Batch  , Dyestuff )
summary(lme.1)
```

```
## Linear mixed model fit by REML ['lmerMod']
## Formula: Yield ~ 1 | Batch
##    Data: Dyestuff
##
## REML criterion at convergence: 319.7
##
## Scaled residuals:
##     Min      1Q  Median      3Q     Max
## -1.4117 -0.7634  0.1418  0.7792  1.8296
##
## Random effects:
##  Groups   Name        Variance Std.Dev.
##  Batch    (Intercept) 1764     42.00
##  Residual             2451     49.51
## Number of obs: 30, groups:  Batch, 6
##
```

```
## Fixed effects:
##             Estimate Std. Error t value
## (Intercept)  1527.50      19.38    78.8
```

Things to note:

- The syntax `Yield ~ 1 | Batch` tells R to fit a model with a global intercept (`1`) and a random Batch effect (`|Batch`). More on that later.
- As usual, `summary` is content aware and has a different behavior for `lme` class objects.
- The output distinguishes between random effects (u), a source of variability, and fixed effect (β), which we want to study. The mean of the random effect is not reported because it is unassumingly 0.
- Were we not interested in the variance components, and only in the coefficients or predictions, an (almost) equivalent `lm` formulation is `lm(Yield ~ Batch)`.

Some utility functions let us query the `lme` object. The function `coef` will work, but will return a cumbersome output. Better use `fixef` to extract the fixed effects, and `ranef` to extract the random effects. The model matrix (of the fixed effects alone), can be extracted with `model.matrix`, and predictions made with `predict`. Note, however, that predictions with mixed-effect models are better treated as prediction problems as in the Supervised Learning Chapter **??**, but are a very delicate matter.

```
detach(Dyestuff)
```

8.2.2 Multiple Random Effects

Let's make things more interesting by allowing more than one random effect. One-way ANOVA can be thought of as the fixed-effects counterpart of the single random effect.

In the `Penicillin` data, we measured the diameter of spread of an organism, along the plate used (a to x), and penicillin type (A to F). We will now try to infer on the diameter of typical organism, and compute its variability over plates and Penicillin types.

```
head(Penicillin)
```

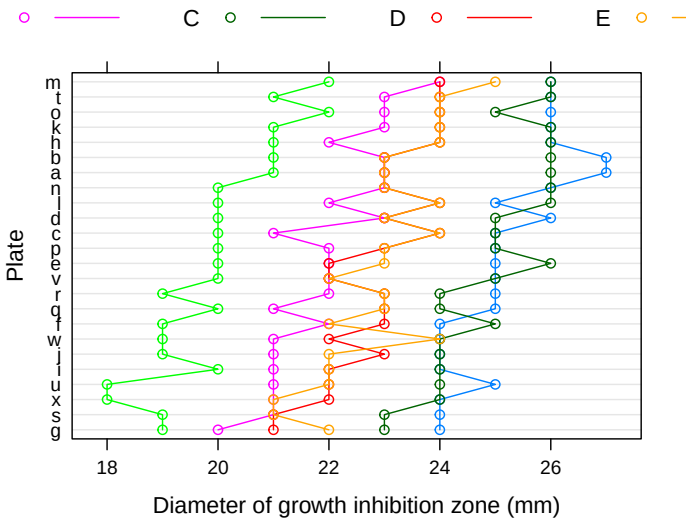
```
##   diameter plate sample
## 1       27     a      A
## 2       23     a      B
## 3       26     a      C
## 4       23     a      D
## 5       23     a      E
## 6       21     a      F
```

One sample per combination:

```
attach(Penicillin)
table(sample, plate) # how many observations per plate & type?
```

```
##       plate
## sample a b c d e f g h i j k l m n o p q r s t u v w x
##      A 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##      B 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##      C 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##      D 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##      E 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##      F 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

And visually:



Let's fit a mixed-effects model with a random plate effect, and a random sample effect:

```
lme.2 <- lmer ( diameter ~  1  + (1|plate )+(1|sample) , Penicillin )
fixef(lme.2) # Fixed effects
```

```
## (Intercept)
##    22.97222
```

```
ranef(lme.2) # Random effects
```

```
## $plate
##   (Intercept)
## a  0.80454704
## b  0.80454704
## c  0.18167191
## d  0.33739069
## e  0.02595313
## f -0.44120322
## g -1.37551591
## h  0.80454704
## i -0.75264078
## j -0.75264078
## k  0.96026582
## l  0.49310948
## m  1.42742217
## n  0.49310948
## o  0.96026582
## p  0.02595313
## q -0.28548443
## r -0.28548443
## s -1.37551591
## t  0.96026582
## u -0.90835956
## v -0.28548443
## w -0.59692200
## x -1.21979713
##
## $sample
##   (Intercept)
## A  2.18705797
## B -1.01047615
## C  1.93789946
```

```
## D -0.09689497
## E -0.01384214
## F -3.00374417
```

Things to note:

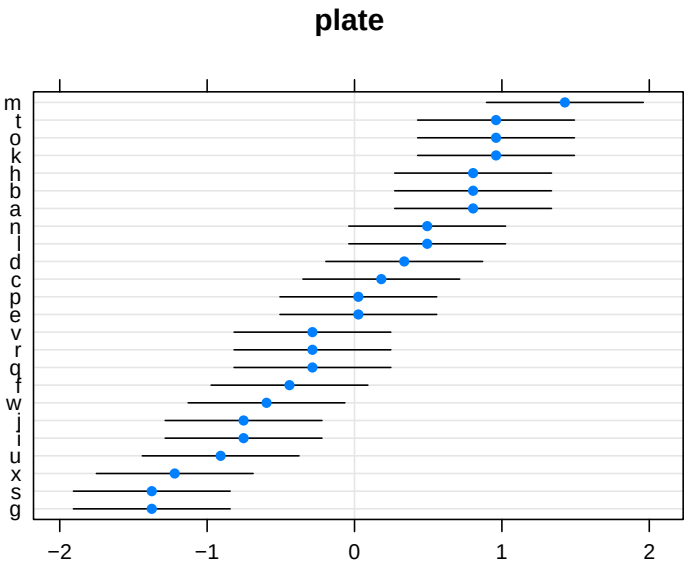
- The syntax `1+ (1| plate ) + (1| sample )` fits a global intercept (mean), a random plate effect, and a random sample effect.
- Were we not interested in the variance components, an (almost) equivalent `lm` formulation is `lm(diameter ~ plate + sample)`.
- The output of `ranef` is somewhat controversial. Think about it: Why would we want to plot the estimates of a random variable?

Since we have two random effects, we may compute the variability of the global mean (the only fixed effect) as we did before. Perhaps more interestingly, we can compute the variability in the response, for a particular plate or sample type.

```
random.effect.lme2 <- ranef(lme.2, condVar = TRUE)
qrr2 <- lattice::dotplot(random.effect.lme2, strip = FALSE)
```

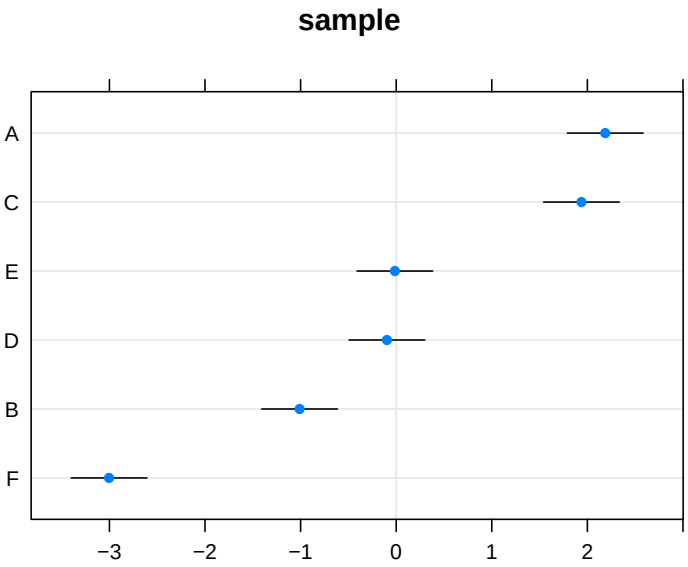
Variability in response for each plate, over various sample types:

```
print(qrr2[[1]])
```



Variability in response for each sample type, over the various plates:

```
print(qrr2[[2]])
```



Things to note:

- The `condVar` argument of the `ranef` function tells R to compute the variability in response conditional on each random effect at a time.
- The `dotplot` function, from the **lattice** package, is only there for the fancy plotting.

We used the penicillin example to demonstrate the incorporation of two random-effects. We could have, however, compared between penicillin types. For this matter, penicillin types are fixed effects to infer on, and not part of the uncertainty in the mean diameter. The appropriate model is the following:

```
lme.2.2 <- lmer( diameter ~  1  + sample + (1|plate) , Penicillin )
```

I may now ask myself: does the `sample`, i.e. penicillin, have any effect? This is what the ANOVA table typically gives us. The next table can be thought of as a "repeated measures ANOVA":

```
anova(lme.2.2)
```

```
## Analysis of Variance Table
##        Df Sum Sq Mean Sq F value
## sample  5 449.22  89.844  297.09
```

Ugh! No p-values. Why is this? Because Doug Bates, the author of **lme4** makes a strong argument against current methods of computing p-values in mixed models. If you insist on an p-value, you may recur to other packages that provide that, at your own caution:

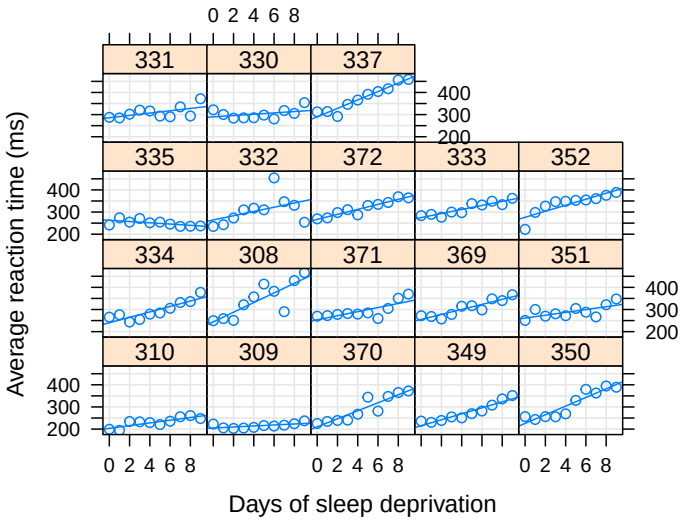
```
car::Anova(lme.2.2)
```

```
## Analysis of Deviance Table (Type II Wald chisquare tests)
##
## Response: diameter
##         Chisq Df Pr(>Chisq)
## sample 1485.4  5  < 2.2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

… and yes; the penicillin type has a significant effect on the diameter.

8.2.3 A Full Mixed-Model

In the `sleepstudy` data, we recorded the reaction times to a series of tests (`Reaction`), after various subject (`Subject`) underwent various amounts of sleep deprivation (`Day`).



We now want to estimate the (fixed) effect of the days of sleep deprivation on response time, while allowing each subject to have his/hers own effect. Put differently, we want to estimate a *random slope* for the effect of `day`. The fixed `Days` effect can be thought of as the average slope over subjects.

```
lme.3 <- lmer ( Reaction ~ Days + ( Days | Subject ) , data= sleepstudy )
```

Things to note:

- `~Days` specifies the fixed effect.
- We used the `Days|Subect` syntax to tell R we want to fit the model `~Days` within each subject.
- Were we fitting the model for purposes of prediction only, an (almost) equivalent `lm` formulation is `lm(Reaction~Days*Subject)`.

The fixed day effect is:

```
fixef(lme.3)
```

```
## (Intercept)        Days
##   251.40510    10.46729
```

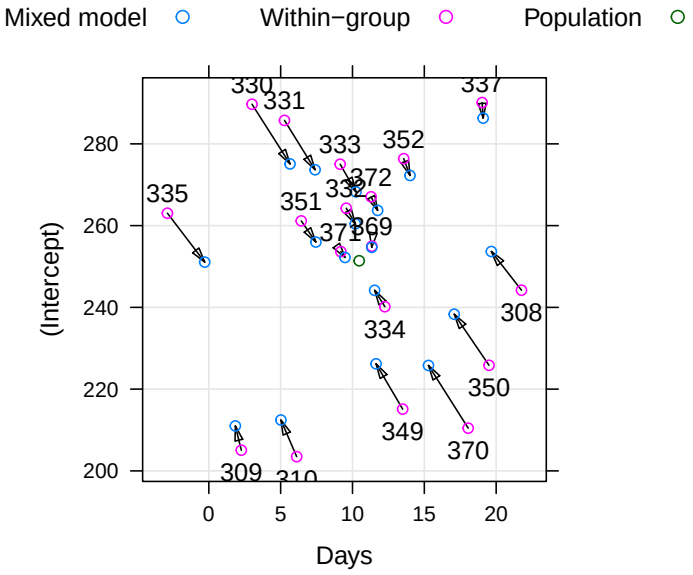
The variability in the average response (intercept) and day effect is

```
ranef(lme.3)
```

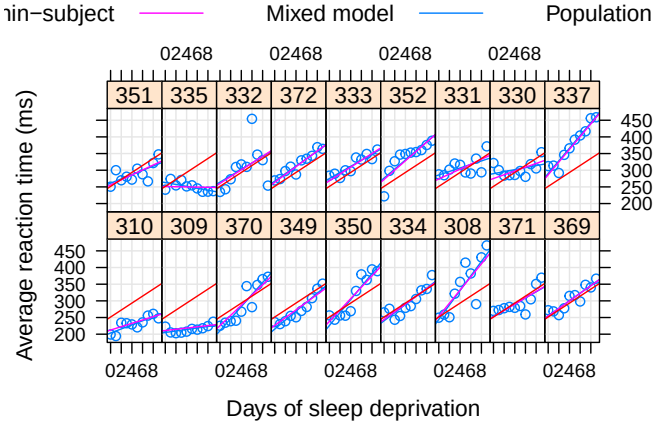
```
## $Subject
##     (Intercept)        Days
## 308   2.2585654   9.1989719
## 309 -40.3985770  -8.6197032
## 310 -38.9602459  -5.4488799
## 330  23.6904985  -4.8143313
## 331  22.2602027  -3.0698946
## 332   9.0395259  -0.2721707
## 333  16.8404312  -0.2236244
## 334  -7.2325792   1.0745761
## 335  -0.3336959 -10.7521591
## 337  34.8903509   8.6282839
## 349 -25.2101104   1.1734143
## 350 -13.0699567   6.6142050
## 351   4.5778352  -3.0152572
## 352  20.8635925   3.5360133
## 369   3.2754530   0.8722166
## 370 -25.6128694   4.8224646
## 371   0.8070397  -0.9881551
## 372  12.3145394   1.2840297
```

Did we really need the whole `lme` machinery to fit a within-subject linear regression and then average over subjects? The answer is yes. The assumptions on the distribution of random effect, namely, that they are normally distributed, allows us to pool information from one subject to another. In the words of John Tukey: "we borrow strength over subjects". Is this a good thing? If the normality assumption is true, it certainly is. If, on the other hand, you have a lot of samples per subject, and you don't need to "borrow strength" from one subject to another, you can simply fit within-subject linear models without the mixed-models machinery.

To demonstrate the "strength borrowing", here is a comparison of the lme, versus the effects of fitting a linear model to each subject separately.



Here is a comparison of the random-day effect from `lme` versus a subject-wise linear model. They are not the same.




```
detach(Penicillin)
```

8.3 Serial Correlations

As previously stated, a hierarchical model is a very convenient way to state correlations. The hierarchical sampling scheme will always yield correlations in blocks. What is the correlation does not have a block structure? Like a smooth temporal decay for time-series, or a smooth spatial decay for geospatial data?

One way to go about, is to find a dedicated package. For instance, in the Spatio-Temporal Data task view, or the Ecological and Environmental task view. Fans of vector-auto-regression should have a look at the vars package.

Instead, we will show how to solve this matter using the **nlme** package. This is because **nlme** allows to specify both a block-covariance structure using the mixed-models framework, and the smooth parametric covariances we find in temporal and spatial data.

The `nlme::Ovary` data is panel data of number of ovarian follicles in different mares (female horse), at various times.

with an AR(1) temporal correlation, alongside random-effects, we take an example from the help of `nlme::corAR1`.

```
library(nlme)
head(nlme::Ovary)
```

```
## Grouped Data: follicles ~ Time | Mare
##   Mare        Time follicles
## 1    1 -0.13636360        20
## 2    1 -0.09090910        15
## 3    1 -0.04545455        19
## 4    1  0.00000000        16
## 5    1  0.04545455        13
## 6    1  0.09090910        10
```

```
fm1Ovar.lme <- nlme::lme(fixed=follicles ~ sin(2*pi*Time) + cos(2*pi*Time),
                  data = Ovary,
                  random = pdDiag(~sin(2*pi*Time)),
                  correlation=corAR1() )
summary(fm1Ovar.lme)
```

```
## Linear mixed-effects model fit by REML
##  Data: Ovary
##        AIC     BIC   logLik
##   1563.448 1589.49 -774.724
##
## Random effects:
##  Formula: ~sin(2 * pi * Time) | Mare
##  Structure: Diagonal
##         (Intercept) sin(2 * pi * Time) Residual
## StdDev:    2.858385           1.257977 3.507053
##
## Correlation Structure: AR(1)
##  Formula: ~1 | Mare
##  Parameter estimate(s):
##       Phi
## 0.5721866
## Fixed effects: follicles ~ sin(2 * pi * Time) + cos(2 * pi * Time)
##                        Value Std.Error  DF   t-value p-value
## (Intercept)        12.188089 0.9436602 295 12.915760  0.0000
## sin(2 * pi * Time) -2.985297 0.6055968 295 -4.929513  0.0000
## cos(2 * pi * Time) -0.877762 0.4777821 295 -1.837159  0.0672
##  Correlation:
##                    (Intr) s(*p*T
## sin(2 * pi * Time)  0.000
## cos(2 * pi * Time) -0.123  0.000
##
## Standardized Within-Group Residuals:
##         Min          Q1         Med          Q3         Max
## -2.34910093 -0.58969626 -0.04577893  0.52931186  3.37167486
##
## Number of Observations: 308
## Number of Groups: 11
```

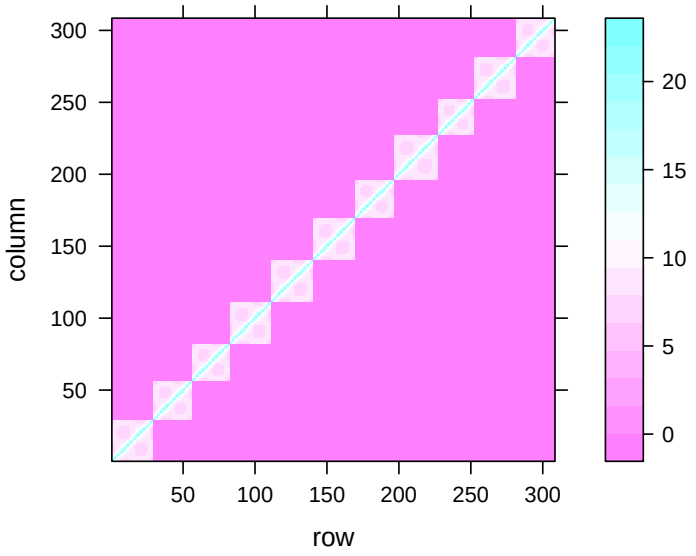
Things to note:

- The fitting is done with the `nlme::lme` function, and not `lme4::lmer` (which does not allow for non blocked covariance models).
- `sin(2*pi*Time) + cos(2*pi*Time)` is a fixed effect that captures seasonality.

- The temporal covariance, is specified using the `correlations=` argument.
- AR(1) was assumed by calling `correlation=corAR1()`. See `nlme::corClasses` for a list of supported correlation structures.
- From the summary, we see that a `Mare` random effect has also been added. Where is it specified? It is implied by the `random=` argument. Read `?lme` for further details.

We can now inspect the contrivance implied by our model's specification:

```
the.cov <- mgcv::extract.lme.cov(fm1Ovar.lme, data = Ovary)
lattice::levelplot(the.cov)
```



8.4 Extensions

8.4.1 Cluster Robust Standard Errors

As previously stated, random effects are nothing more than a convenient way to specify dependencies within a level of a random effect, i.e., within a group/cluster. This is also the motivation underlying *cluster robust* inference, which is immensely popular with econometricians, but less so elsewhere. What is the difference between the two?

Mixed models framework is a bona-fide generalization of cluster robust inference. This author thus recommends using the **lme4** and **nlme** packages for mixed models to deal with correlations within cluster.

For a longer comparison between the two approaches, see Michael Clarck's guide.

8.4.2 Linear Models for Panel Data

nlme and **lme4** will probably provide you with all the functionality you need for panel data. If, however, you are trained as an econometrist, prefer the econometric parlance, and are not using non-linead models, then the plm package is just for you. In particular, it allows for cluster-robust covariance estimates, and Durbin–Wu–Hausman test for random effects. The **plm** package vignette also has a comparison to the **nlme** package.

8.4.3 Testing Hypotheses on Correlations

After working so hard to model the correlations in observation, we may want to test if it was all required. Douglas Bates, the author of **nlme** and **lme4** wrote a famous cautionary note, found here, on hypothesis testing in mixed models. Many practitioners, however, do not adopt Doug's view. Many of the popular tests, particularly the ones in the econometric literature, can be found in the **plm** package (see Section 6 in the package vignette). These include

tests for poolability, Hausman test, tests for serial correlations, tests for cross-sectional dependence, and unit root tests.

8.5 Relation to Other Estimators

8.5.1 Fixed Effects in the Econometric Literature

Fixed effects in the statistical literature, as discussed herein, are different than those in the econometric literature. See Section 7 of the **plm** package vignette for a comparison.

8.5.2 Relation to Generalized Least Squares (GLS)

GLS is the solution to a decorrelated least squares problem:

$$\hat{\beta}_{GLS} := argmin_{\beta}\{(X'\beta - y)'\Sigma^{-1}(X'\beta - y)'\}.$$

This estimator can be viewed as a least squares estimator that accounts for correlations in the data. It is also a maximum likelihood estimator under a Gaussian error assumption. Viewed as the latter, then linear mixed models under a Gaussian error assumption, collapses to a GLS estimator.

8.5.3 Relation to Conditional Gaussian Fields

In the geo-spatial literature, geo-located measurements are typically assumed to be sampled from a *Gaussian Random Field*. All the models discussed in this chapter can be stated in terms of these random fields. In the random field nomenclature, the fixed effects are known as the *drift*, or the *mean field*, and the covariance in errors is known as the *correlation function*, or *kernel operator*. Assuming stationarity, these simplify to the *power spectrum*.

8.5.4 Relation to Empirical Risk Minimization (ERM)

ERM is more general than mixed-models estimation since it allows loss functions that are not the (log) likelihood. ERM is less general than LMM, in that ERM (typically) does not account for correlations in the data.

8.5.5 Relation to M-Estimation

M-estimation is term in the statistical literature for ERM.

8.5.6 Relation to Generalize Estimating Equations (GEE)

The first order condition of the LMM problem returns a set of (non-linear) estimating equations. In this sense, GEE can be seen as more general than LMM in that the GEE need not be the derivative of the (log) likelihood.

8.5.7 Relation to Generalized Method of Moments (GMM)

Moment matching estimators are more general than likelihood based estimators in that assuming the likelihood implies all momemnts are assumed, where as in GMMs, not all moments are assumed. For GMM estimators in R, see the **gmm** pacakge.

8.5.8 Relation to MANOVA

Multivariate analysis of variance (MANOVA) deals with the estimation of effect on **vector valued** outcomes. Put differently: in ANOVA the response, y, is univariate. In MANOVA, the outcome is multivariate. MANOVA is useful when there are correlations among the entries of y. Otherwise- one may simply solve many ANOVA problems, instead of a single MANOVA.

Now assume that the outcome of a MANOVA is measurements of an individual at several time periods. The measurements are clearly correlated, so that MANOVA may be useful. But one may also treat the subject as a random effect, with a univariate response. We thus see that this seemingly MANOVA problem can be solved with the mixed models framework.

What MANOVA problems cannot be solved with mixed models? There may be cases where the covariance of the multivariate outcome, y, is very complicated. If the covariance in y may not be stated using a combination of random and fixed effects, then the covariance has to be stated explicitly in the MANOVA framework. It is also possible to consider mixed-models with multivariate outcomes, i.e., a *mixed MANOVA*, or *hirarchial MANOVA*. The R functions we present herein permit this.

8.6 The Variance-Components View

TODO

8.7 Bibliographic Notes

Most of the examples in this chapter are from the documentation of the **lme4** package (Bates et al., 2015). For a general and very applied treatment, see Pinero and Bates (2000). As usual, a hands on view can be found in Venables and Ripley (2013), and also in an excellent blog post by Kristoffer Magnusson For a more theoretical view see Weiss (2005) or Searle et al. (2009). Sometimes it is unclear if an effect is random or fixed; on the difference between the two types of inference see Rosset and Tibshirani (2018) and references therein. For more on predictions in linear mixed models see Robinson (1991), Rabinowicz and Rosset (2018), and references therein. See Michael Clarck's guide for various ways of dealing with correlations within groups. For the geo-spatial view and terminology of correlated data, see Christakos (2000), Diggle et al. (1998), Allard (2013), and Cressie (2015).

8.8 Practice Yourself

1. Computing the variance of the sample mean given dependent correlations. How does it depend on the covariance between observations? When is the sample most informative on the population mean?
2. Return to the `Penicillin` data set. Instead of fitting an LME model, fit an LM model with `lm`. I.e., treat all random effects as fixed.
    a. Compare the effect estimates.
    b. Compare the standard errors.
    c. Compare the predictions of the two models.
3. [Very Advanced!] Return to the `Penicillin` data and use the `gls` function to fit a generalized linear model, equivalent to the LME model in our text.
4. Read about the "oats" dataset using `? MASS::oats`.Inspect the dependency of the yield (Y) in the Varieties (V) and the Nitrogen treatment (N).
    1. Fit a linear model, does the effect of the treatment significant? The interaction between the Varieties and Nitrogen is significant?
    2. An expert told you that could be a variance between the different blocks (B) which can bias the analysis. fit a LMM for the data.
    3. Do you think the blocks should be taken into account as "random effect" or "fixed effect"?

5. Return to the temporal correlation in Section 8.3, and replace the AR(1) covariance, with an ARMA covariance. Visualize the data's covariance matrix, and compare the fitted values.

Chapter 9

Multivariate Data Analysis

The term "multivariate data analysis" is so broad and so overloaded, that we start by clarifying what is discussed and what is not discussed in this chapter. Broadly speaking, we will discuss statistical *inference*, and leave more "exploratory flavored" matters like clustering, and visualization, to the Unsupervised Learning Chapter **??**.

We start with an example.

Example 9.1. Consider the problem of a patient monitored in the intensive care unit. At every minute the monitor takes p physiological measurements: blood pressure, body temperature, etc. The total number of minutes in our data is n, so that in total, we have $n \times p$ measurements, arranged in a matrix. We also know the typical measurements for this patient when healthy: μ_0.

Formally, let y be single (random) measurement of a p-variate random vector. Denote $\mu := E[y]$. Here is the set of problems we will discuss, in order of their statistical difficulty.

- **Signal detection**: a.k.a. *multivariate hypothesis testing*, i.e., testing if μ equals μ_0 and for $\mu_0 = 0$ in particular. In our example: "are the current measurement different than a typical one?"
- **Signal counting**: Counting the number of elements in μ that differ from μ_0, and for $\mu_0 = 0$ in particular. In our example: "how many measurements differ than their typical values?"
- **Signal identification**: a.k.a. *multiple testing*, i.e., testing which of the elements in μ differ from μ_0 and for $\mu_0 = 0$ in particular. In the ANOVA literature, this is known as a **post-hoc** analysis. In our example: "which measurements differ than their typical values?"
- **Signal estimation**: Estimating the magnitudes of the departure of μ from μ_0, and for $\mu_0 = 0$ in particular. If estimation follows a *signal detection* or *signal identification* stage, this is known as a *selective estimation* problem. In our example: "what is the value of the measurements that differ than their typical values?"
- **Multivariate Regression**: a.k.a. *MANOVA* in statistical literature, and *structured learning* in the machine learning literature. In our example: "what factors affect the physiological measurements?"

Example 9.2. Consider the problem of detecting regions of cognitive function in the brain using fMRI. Each measurement is the activation level at each location in a brain's region. If the region has a cognitive function, the mean activation differs than $\mu_0 = 0$ when the region is evoked.

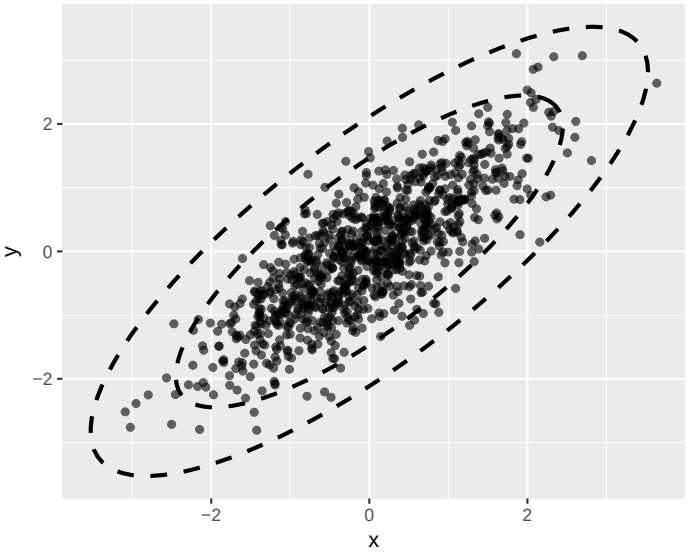
Example 9.3. Consider the problem of detecting cancer encoding regions in the genome. Each measurement is the vector of the genetic configuration of an individual. A cancer encoding region will have a different (multivariate) distribution between sick and healthy. In particular, μ of sick will differ from μ of healthy.

Example 9.4. Consider the problem of the simplest multiple regression. The estimated coefficient, $\hat{\beta}$ are a random vector. Regression theory tells us that its covariance is $(X'X)^{-1}\sigma^2$, and null mean of β. We thus see that inference on the vector of regression coefficients, is nothing more than a multivaraite inference problem.

Remark. In the above, "signal" is defined in terms of μ. It is possible that the signal is not in the location, μ, but rather in the covariance, Σ. We do not discuss these problems here, and refer the reader to Nadler (2008).

Another possible question is: does a multivariate analysis gives us something we cannot get from a mass-univariate analysis (i.e., a multivariate analysis on each variable separately). In Example 9.1 we could have just performed multiple univariate tests, and sign an alarm when any of the univariate detectors was triggered. The reason we want a multivariate detector, and not multiple univariate detectors is that it is possible that each measurement alone is borderline, but together, the signal accumulates. In our ICU example is may mean that the pulse is borderline, the body temperature is borderline, etc. Analyzed simultaneously, it is clear that the patient is in distress.

The next figure[1] illustrates the idea that some bi-variate measurements may seem ordinary univariately, while very anomalous when examined bi-variately.
Remark. The following figure may also be used to demonstrate the difference between Euclidean Distance and Mahalanobis Distance.



9.1 Signal Detection

Signal detection deals with the detection of the departure of μ from some μ_0, and especially, $\mu_0 = 0$. This problem can be thought of as the multivariate counterpart of the univariate hypothesis t-test.

9.1.1 Hotelling's T2 Test

The most fundamental approach to signal detection is a mere generalization of the t-test, known as *Hotelling's T^2 test*.

Recall the univariate t-statistic of a data vector x of length n:

$$t^2(x) := \frac{(\bar{x} - \mu_0)^2}{Var[\bar{x}]} = (\bar{x} - \mu_0) Var[\bar{x}]^{-1} (\bar{x} - \mu_0), \tag{9.1}$$

where $Var[\bar{x}] = S^2(x)/n$, and $S^2(x)$ is the unbiased variance estimator $S^2(x) := (n-1)^{-1} \sum (x_i - \bar{x})^2$.

Generalizing Eq(9.1) to the multivariate case: μ_0 is a p-vector, $\bar{x}$ is a p-vector, and $Var[\bar{x}]$ is a $p \times p$ matrix of the covariance between the p coordinated of $\bar{x}$. When operating with vectors, the squaring becomes a quadratic form, and the division becomes a matrix inverse. We thus have

$$T^2(x) := (\bar{x} - \mu_0)' Var[\bar{x}]^{-1} (\bar{x} - \mu_0), \tag{9.2}$$

[1] My thanks to Efrat Vilneski for the figure.

which is the definition of Hotelling's T^2 test statistic. We typically denote the covariance between coordinates in x with $\hat{\Sigma}(x)$, so that $\widehat{\Sigma}_{k,l} := \widehat{Cov}[x_k, x_l] = (n-1)^{-1}\sum(x_{k,i} - \bar{x}_k)(x_{l,i} - \bar{x}_l)$. Using the Σ notation, Eq.(9.2) becomes

$$T^2(x) := n(\bar{x} - \mu_0)'\hat{\Sigma}(x)^{-1}(\bar{x} - \mu_0), \tag{9.3}$$

which is the standard notation of Hotelling's test statistic.

For inference, we need the null distribution of Hotelling's test statistic. For this we introduce some vocabulary[2]:

1. **Low Dimension**: We call a problem *low dimensional* if $n \gg p$, i.e. $p/n \approx 0$. This means there are many observations per estimated parameter.
2. **High Dimension**: We call a problem *high dimensional* if $p/n \to c$, where $c \in (0, 1)$. This means there are more observations than parameters, but not many.
3. **Very High Dimension**: We call a problem *very high dimensional* if $p/n \to c$, where $1 < c < \infty$. This means there are less observations than parameter.

Hotelling's T^2 test can only be used in the low dimensional regime. For some intuition on this statement, think of taking $n = 20$ measurements of $p = 100$ physiological variables. We seemingly have 20 observations, but there are 100 unknown quantities in μ. Would you trust your conclusion that $\bar{x}$ is different than μ_0 based on merely 20 observations.

The above criticism is formalized in Bai and Saranadasa (1996). For modern applications, Hotelling's T^2 is not recommended, since many modern alternatives have been made available. See Rosenblatt et al. (2016) and references for a review.

9.1.2 Various Types of Signal to Detect

In the previous, we assumed that the signal is a departure of μ from some μ_0. For vactor-valued data y, that is distributed F, we may define "signal" as any departure from some F_0. This is the multivaraite counterpart of goodness-of-fit (GOF) tests.

Even when restricting "signal" to departures of μ from μ_0, we may try to detect various types of signal:

1. **Dense Signal**: when the departure is in all coordinates of μ.
2. **Sparse Signal**: when the departure is in a subset of coordinates of μ.

A manufactoring motivation is consistent with a dense signal: if a manufacturing process has failed, we expect a change in many measurements (i.e. coordinates of μ). A brain-imaging motivation is consistent with a dense signal: if a region encodes cognitive function, we expect a change in many brain locations (i.e. coordinates of μ.) A genetic motivation is consistent with a sparse signal: if susceptibility of disease is genetic, only a small subset of locations in the genome will encode it.

Hotelling's T^2 statistic is designed for dense signal. The following is a simple statistic designed for sparse signal.

9.1.3 Simes' Test

Hotelling's T^2 statistic has currently two limitations: It is designed for dense signals, and it requires estimating the covariance, which is a very difficult problem.

An algorithm, that is sensitive to sparse signal and allows statistically valid detection under a wide range of covariances (even if we don't know the covariance) is known as *Simes' Test*. The statistic is defined vie the following algorithm:

1. Compute p variable-wise p-values: $p_1, \ldots, p_j$.
2. Denote $p_{(1)}, \ldots, p_{(j)}$ the sorted p-values.
3. Simes' statistic is $p_{Simes} := min_j\{p_{(j)} \times p/j\}$.
4. Reject the "no signal" null hypothesis at significance α if $p_{Simes} < \alpha$.

[2]This vocabulary is not standard in the literature, so when you read a text, you need to verify yourself what the author means.

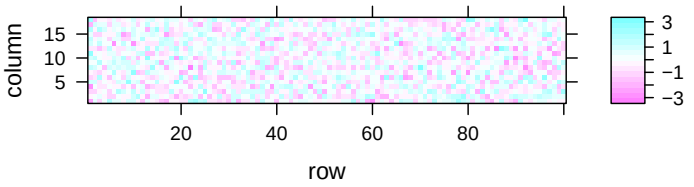
9.1.4 Signal Detection with R

Let's generate some data with no signal.

```
library(mvtnorm)
n <- 100 # observations
p <- 18 # parameter dimension
mu <- rep(0,p) # no signal
x <- rmvnorm(n = n, mean = mu)
dim(x)
```

```
## [1] 100  18
```

```
lattice::levelplot(x)
```



Now make our own Hotelling function.

```
hotellingOneSample <- function(x, mu0=rep(0,ncol(x))){
  n <- nrow(x)
  p <- ncol(x)
  stopifnot(n > 5* p)
  bar.x <- colMeans(x)
  Sigma <- var(x)
  Sigma.inv <- solve(Sigma)
  T2 <- n * (bar.x-mu0) %*% Sigma.inv %*% (bar.x-mu0)
  p.value <- pchisq(q = T2, df = p, lower.tail = FALSE)
  return(list(statistic=T2, pvalue=p.value))
}
hotellingOneSample(x)
```

```
## $statistic
##          [,1]
## [1,] 17.22438
##
## $pvalue
##           [,1]
## [1,] 0.5077323
```

Things to note:

- `stopifnot(n > 5 * p)` is a little verification to check that the problem is indeed low dimensional. Otherwise, the χ^2 approximation cannot be trusted.
- `solve` returns a matrix inverse.
- `%*%` is the matrix product operator (see also `crossprod()`).
- A function may return only a single object, so we wrap the statistic and its p-value in a `list` object.

Just for verification, we compare our home made Hotelling's test, to the implementation in the **rrcov** package. The statistic is clearly OK, but our χ^2 approximation of the distribution leaves room to desire. Personally, I would never trust a Hotelling test if n is not much greater than p, in which case I would use a high-dimensional adaptation (see Bibliography).

```
rrcov::T2.test(x)
```

```
##
##  One-sample Hotelling test
##
```

```
## data:  x
## T2 = 17.22400, F = 0.79259, df1 = 18, df2 = 82, p-value = 0.703
## alternative hypothesis: true mean vector is not equal to (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)'
##
## sample estimates:
##                       [,1]       [,2]       [,3]       [,4]      [,5]
## mean x-vector -0.01746212 0.03776332 0.1006145 -0.2083005 0.1026982
##                       [,6]         [,7]       [,8]       [,9]       [,10]
## mean x-vector -0.05220043 -0.009497987 -0.1139856 0.02851701 -0.03089953
##                      [,11]      [,12]      [,13]      [,14]      [,15]
## mean x-vector -0.02457798 -0.1270753 0.04717076 0.01683591 0.03085023
##                    [,16]       [,17]     [,18]
## mean x-vector 0.1499485 -0.07630663 0.1004852
```

Let's do the same with Simes':

```
Simes <- function(x){
  p.vals <- apply(x, 2, function(z) t.test(z)$p.value) # Compute variable-wise pvalues
  p <- ncol(x)
  p.Simes <- p * min(sort(p.vals)/seq_along(p.vals)) # Compute the Simes statistic
  return(c(pvalue=p.Simes))
}
Simes(x)
```

```
##    pvalue
## 0.6398998
```

And now we verify that both tests can indeed detect signal when present. Are p-values small enough to reject the "no signal" null hypothesis?

```
mu <- rep(x = 10/p,times=p) # inject signal
x <- rmvnorm(n = n, mean = mu)
hotellingOneSample(x)
```

```
## $statistic
##          [,1]
## [1,] 686.8046
##
## $pvalue
##               [,1]
## [1,] 3.575926e-134
```

```
Simes(x)
```

```
##       pvalue
## 2.765312e-10
```

... yes. All p-values are very small, so that all statistics can detect the non-null distribution.

9.2 Signal Counting

There are many ways to approach the *signal counting* problem. For the purposes of this book, however, we will not discuss them directly, and solve the signal counting problem as a signal identification problem: if we know **where** μ departs from μ_0, we only need to count coordinates to solve the signal counting problem.
Remark. In the sparsity or multiple-testing literature, what we call "signal counting" is known as "adapting to sparsit", or "adaptivity".

9.3 Signal Identification

The problem of *signal identification* is also known as *selective testing*, or more commonly as *multiple testing*.

In the ANOVA literature, an identification stage will typically follow a detection stage. These are known as the *omnibus F test*, and *post-hoc* tests, respectively. In the multiple testing literature there will typically be no preliminary detection stage. It is typically assumed that signal is present, and the only question is "where?"

The first question when approaching a multiple testing problem is "what is an error"? Is an error declaring a coordinate in μ to be different than μ_0 when it is actually not? Is an error an overly high proportion of falsely identified coordinates? The former is known as the *family wise error rate* (FWER), and the latter as the *false discovery rate* (FDR).
Remark. These types of errors have many names in many communities. See the Wikipedia entry on ROC for a table of the (endless) possible error measures.

9.3.1 Signal Identification in R

One (of many) ways to do signal identification involves the `stats::p.adjust` function. The function takes as inputs a p-vector of the variable-wise **p-values**. Why do we start with variable-wise p-values, and not the full data set?

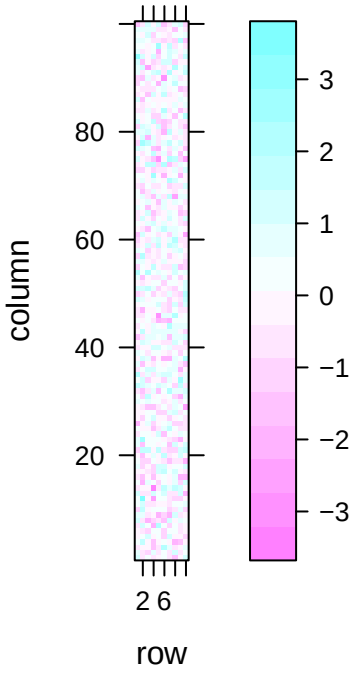
a. Because we want to make inference variable-wise, so it is natural to start with variable-wise statistics.
b. Because we want to avoid dealing with covariances if possible. Computing variable-wise p-values does not require estimating covariances.
c. So that the identification problem is decoupled from the variable-wise inference problem, and may be applied much more generally than in the setup we presented.

We start be generating some high-dimensional multivariate data and computing the coordinate-wise (i.e. hypothesis-wise) p-value.

```
library(mvtnorm)
n <- 1e1
p <- 1e2
mu <- rep(0,p)
x <- rmvnorm(n = n, mean = mu)
dim(x)
```

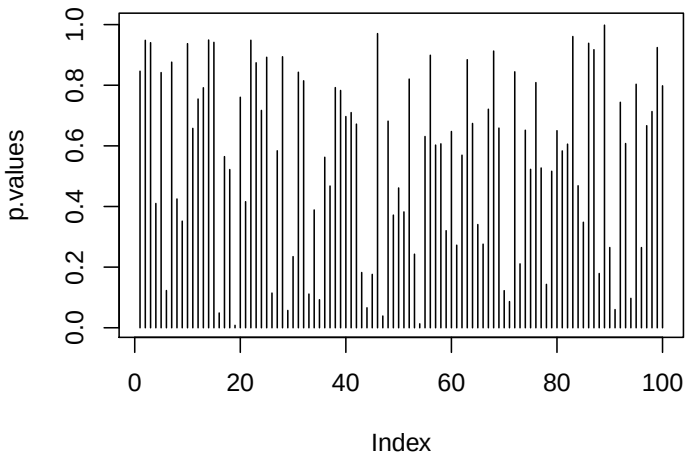
```
## [1]  10 100
```

```
lattice::levelplot(x)
```



We now compute the pvalues of each coordinate. We use a coordinate-wise t-test. Why a t-test? Because for the purpose of demonstration we want a simple test. In reality, you may use any test that returns valid p-values.

```
t.pval <- function(y) t.test(y)$p.value
p.values <- apply(X = x, MARGIN = 2, FUN = t.pval)
plot(p.values, type='h')
```



Things to note:

- `t.pval` is a function that merely returns the p-value of a t.test.
- We used the `apply` function to apply the same function to each column of `x`.
- `MARGIN=2` tells `apply` to compute over columns and not rows.
- The output, `p.values`, is a vector of 100 p-values.

We are now ready to do the identification, i.e., find which coordinate of μ is different than $\mu_0 = 0$. The workflow for identification has the same structure, regardless of the desired error guarantees:

1. Compute an `adjusted p-value`.
2. Compare the adjusted p-value to the desired error level.

If we want $FWER \leq 0.05$, meaning that we allow a 5% probability of making any mistake, we will use the `method="holm"` argument of `p.adjust`.

```
alpha <- 0.05
p.values.holm <- p.adjust(p.values, method = 'holm' )
which(p.values.holm < alpha)
```

```
## integer(0)
```

If we want $FDR \leq 0.05$, meaning that we allow the proportion of false discoveries to be no larger than 5%, we use the `method="BH"` argument of `p.adjust`.

```
alpha <- 0.05
p.values.BH <- p.adjust(p.values, method = 'BH' )
which(p.values.BH < alpha)
```

```
## integer(0)
```

We now inject some strong signal in μ just to see that the process works. We will artificially inject signal in the first 10 coordinates.

```
mu[1:10] <- 2 # inject signal in first 10 variables
x <- rmvnorm(n = n, mean = mu) # generate data
p.values <- apply(X = x, MARGIN = 2, FUN = t.pval)
p.values.BH <- p.adjust(p.values, method = 'BH' )
which(p.values.BH < alpha)
```

```
##  [1]  1  2  3  4  5  6  7  9 10 55
```

Indeed- we are now able to detect that the first coordinates carry signal, because their respective coordinate-wise null hypotheses have been rejected.

9.4 Signal Estimation (*)

The estimation of the elements of μ is a seemingly straightforward task. This is not the case, however, if we estimate only the elements that were selected because they were significant (or any other data-dependent criterion). Clearly, estimating only significant entries will introduce a bias in the estimation. In the statistical literature, this is known as *selection bias*. Selection bias also occurs when you perform inference on regression coefficients after some model selection, say, with a lasso, or a forward search[3].

Selective inference is a complicated and active research topic so we will not offer any off-the-shelf solution to the matter. The curious reader is invited to read Rosenblatt and Benjamini (2014), Javanmard and Montanari (2014), or Will Fithian's PhD thesis (Fithian, 2015) for more on the topic.

9.5 Multivariate Regression (*)

Multivaraite regression, a.k.a. *MANOVA*, similar to *structured learning* in machine learning, is simply a regression problem where the outcome, y, is not scalar values but vector valued. It is not to be confused with *multiple regression* where the predictor, x, is vector valued, but the outcome is scalar.

If the linear models generalize the two-sample t-test from two, to multiple populations, then multivariate regression generalizes Hotelling's test in the same way.

When the entries of y are independent, MANOVA collapses to multiple univariate regressions. It is only when entries in y are correlated that we can gain in accuracy and power by harnessing these correlations through the MANOVA framework.

9.5.1 Multivariate Regression with R

TODO

9.6 Graphical Models (*)

Fitting a multivariate distribution, i.e. learning a *graphical model*, is a very hard task. To see why, consider the problem of p continuous variables. In the simplest case, where we can assume normality, fitting a distributions means estimating the p parameters in the expectation, μ, and $p(p+1)/2$ parameters in the covariance, Σ. The number of observations required for this task, n, may be formidable.

A more humble task, is to identify **independencies**, known as *structure learning* in the machine learning literature. Under the multivariate normality assumption, this means identifying zero entries in Σ, or more precisely, zero entries in Σ^{-1}. This task can be approached as a **signal identification** problem (9.3). The same solutions may be applied to identify non-zero entries in Σ, instead of μ as discussed until now.

If multivariate normality cannot be assumed, then identifying independencies cannot be done via the covariance matrix Σ and more elaborate algorithms are required.

9.6.1 Graphical Models in R

TODO

[3] You might find this shocking, but it does mean that you cannot trust the `summary` table of a model that was selected from a multitude of models.

9.7 Biblipgraphic Notes

For a general introduction to multivariate data analysis see Anderson-Cook (2004). For an R oriented introduction, see Everitt and Hothorn (2011). For more on the difficulties with high dimensional problems, see Bai and Saranadasa (1996). For some cutting edge solutions for testing in high-dimension, see Rosenblatt et al. (2016) and references therein. Simes' test is not very well known. It is introduced in Simes (1986), and proven to control the type I error of detection under a PRDS type of dependence in Benjamini and Yekutieli (2001). For more on multiple testing, and signal identification, see Efron (2012). For more on the choice of your error rate see Rosenblatt (2013). For an excellent review on graphical models see Kalisch and Bühlmann (2014). Everything you need on graphical models, Bayesian belief networks, and structure learning in R, is collected in the Task View.

9.8 Practice Yourself

1. Generate multivariate data with:

```
set.seed(3)
mean<-rexp(50,6)
multi<-  rmvnorm(n = 100, mean = mean)
```

   a. Use Hotelling's test to determine if μ equals $\mu_0 = 0$. Can you detect the signal?
   b. Perform t.test on each variable and extract the p-value. Try to identify visually the variables which depart from μ_0.
   c. Use `p.adjust` to identify in which variables there are any departures from $\mu_0 = 0$. Allow 5% probability of making any false identification.
   d. Use `p.adjust` to identify in which variables there are any departures from $\mu_0 = 0$. Allow a 5% proportion of errors within identifications.

2. Generate multivariate data from two groups: `rmvnorm(n = 100, mean = rep(0,10))` for the first, and `rmvnorm(n = 100, mean = rep(0.1,10))` for the second.

   a. Do we agree the groups differ?
   b. Implement the two-group Hotelling test described in Wikipedia: (https://en.wikipedia.org/wiki/Hotelling%27s_T-squared_distribution#Two-sample_statistic).
   c. Verify that you are able to detect that the groups differ.
   d. Perform a two-group t-test on each coordinate. On which coordinates can you detect signal while controlling the FWER? On which while controlling the FDR? Use `p.adjust`.

3. Return to the previous problem, but set `n=9`. Verify that you cannot compute your Hotelling statistic.

Chapter 10

Plotting

Whether you are doing EDA, or preparing your results for publication, you need plots. R has many plotting mechanisms, allowing the user a tremendous amount of flexibility, while abstracting away a lot of the tedious details. To be concrete, many of the plots in R are simply impossible to produce with Excel, SPSS, or SAS, and would take a tremendous amount of work to produce with Python, Java and lower level programming languages.

In this text, we will focus on two plotting packages. The basic **graphics** package, distributed with the base R distribution, and the **ggplot2** package.

Before going into the details of the plotting packages, we start with some philosophy. The **graphics** package originates from the mainframe days. Computers had no graphical interface, and the output of the plot was immediately sent to a printer. Once a plot has been produced with the **graphics** package, just like a printed output, it cannot be queried nor changed, except for further additions.

The philosophy of R is that **everyting is an object**. The **graphics** package does not adhere to this philosophy, and indeed it was soon augmented with the **grid** package (R Core Team, 2016), that treats plots as objects. **grid** is a low level graphics interface, and users may be more familiar with the **lattice** package built upon it (Sarkar, 2008).

lattice is very powerful, but soon enough, it was overtaken in popularity by the **ggplot2** package (Wickham, 2009). **ggplot2** was the PhD project of Hadley Wickham, a name to remember… Two fundamental ideas underlay **ggplot2**: (i) everything is an object, and (ii), plots can be described by a simple grammar, i.e., a language to describe the building blocks of the plot. The grammar in **ggplot2** are is the one stated by Wilkinson (2006). The objects and grammar of **ggplot2** have later evolved to allow more complicated plotting and in particular, interactive plotting.

Interactive plotting is a very important feature for EDA, and reporting. The major leap in interactive plotting was made possible by the advancement of web technologies, such as JavaScript and D3.JS. Why is this? Because an interactive plot, or report, can be seen as a web-site. Building upon the capabilities of JavaScript and your web browser to provide the interactivity, greatly facilitates the development of such plots, as the programmer can rely on the web-browsers capabilities for interactivity.

10.1 The graphics System

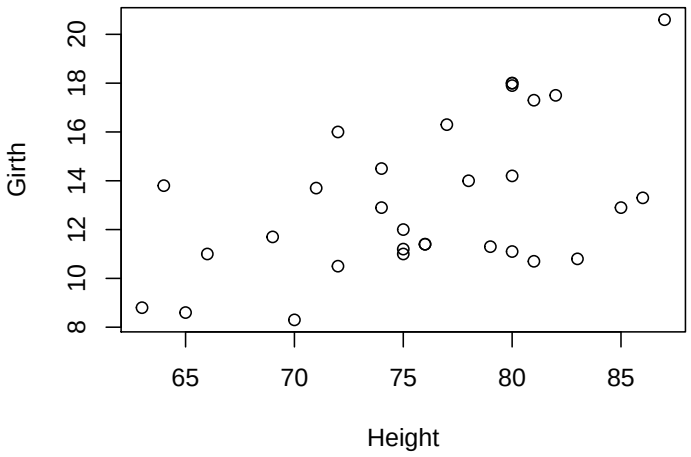
The R code from the Basics Chapter 3 is a demonstration of the **graphics** package and plotting system. We make a quick review of the basics.

10.1.1 Using Existing Plotting Functions

10.1.1.1 Scatter Plot

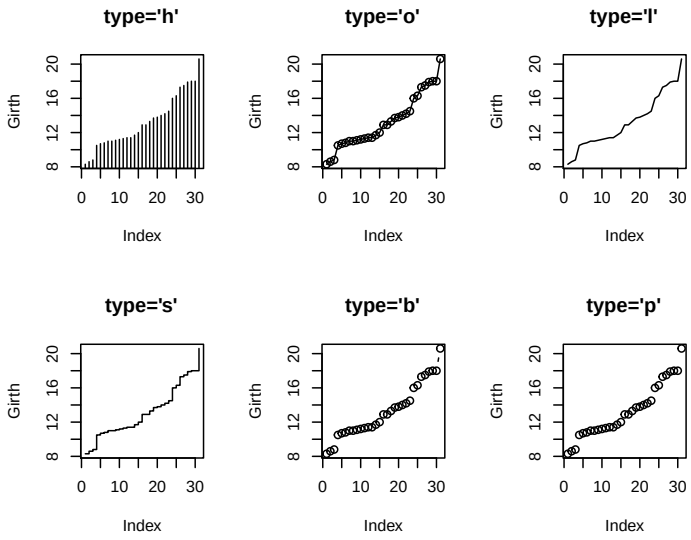
A simple scatter plot.

```
attach(trees)
plot(Girth ~ Height)
```



Various types of plots.

```
par.old <- par(no.readonly = TRUE)
par(mfrow=c(2,3))
plot(Girth, type='h', main="type='h'")
plot(Girth, type='o', main="type='o'")
plot(Girth, type='l', main="type='l'")
plot(Girth, type='s', main="type='s'")
plot(Girth, type='b', main="type='b'")
plot(Girth, type='p', main="type='p'")
```



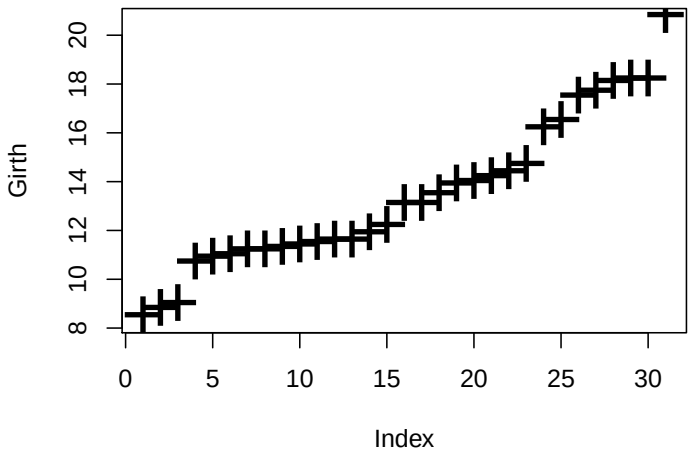

```
par(par.old)
```

Things to note:

- The `par` command controls the plotting parameters. `mfrow=c(2,3)` is used to produce a matrix of plots with 2 rows and 3 columns.
- The `par.old` object saves the original plotting setting. It is restored after plotting using `par(par.old)`.
- The `type` argument controls the type of plot.
- The `main` argument controls the title.
- See `?plot` and `?par` for more options.

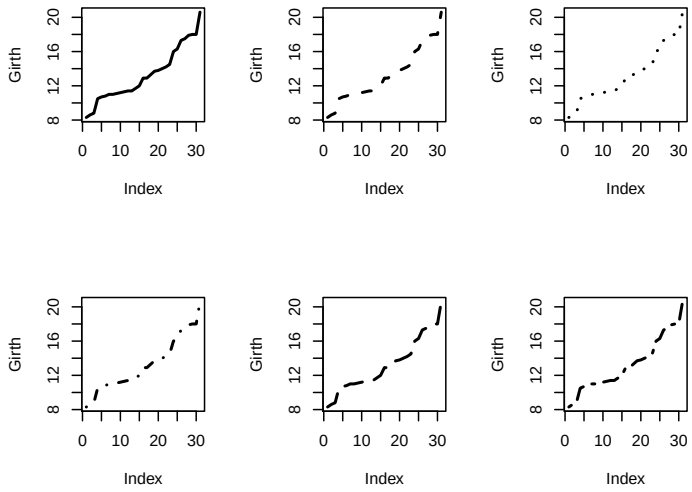
Control the plotting characters with the `pch` argument, and size with the `cex` argument.

```
plot(Girth, pch='+', cex=3)
```



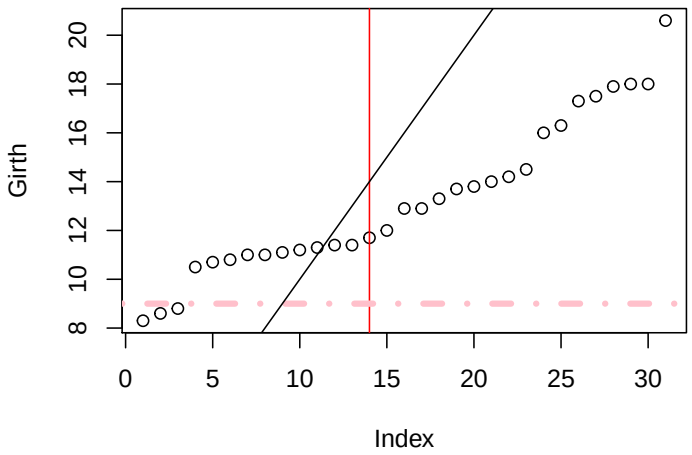
Control the line's type with `lty` argument, and width with `lwd`.

```
par(mfrow=c(2,3))
plot(Girth, type='l', lty=1, lwd=2)
plot(Girth, type='l', lty=2, lwd=2)
plot(Girth, type='l', lty=3, lwd=2)
plot(Girth, type='l', lty=4, lwd=2)
plot(Girth, type='l', lty=5, lwd=2)
plot(Girth, type='l', lty=6, lwd=2)
```

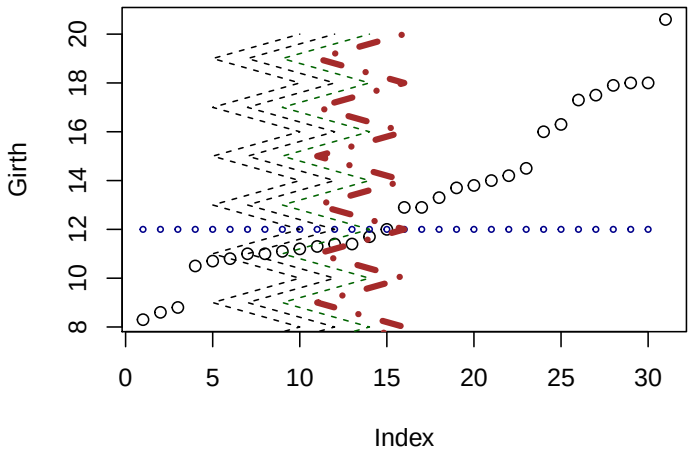


Add line by slope and intercept with `abline`.

```
plot(Girth)
abline(v=14, col='red') # vertical line at 14.
abline(h=9, lty=4,lwd=4, col='pink') # horizontal line at 9.
abline(a = 0, b=1) # linear line with intercept a=0, and slope b=1.
```




```
plot(Girth)
points(x=1:30, y=rep(12,30), cex=0.5, col='darkblue')
lines(x=rep(c(5,10), 7), y=7:20, lty=2 )
lines(x=rep(c(5,10), 7)+2, y=7:20, lty=2 )
lines(x=rep(c(5,10), 7)+4, y=7:20, lty=2 , col='darkgreen')
lines(x=rep(c(5,10), 7)+6, y=7:20, lty=4 , col='brown', lwd=4)
```



Things to note:

- `points` adds points on an existing plot.
- `lines` adds lines on an existing plot.
- `col` controls the color of the element. It takes names or numbers as argument.
- `cex` controls the scale of the element. Defaults to `cex=1`.

Add other elements.

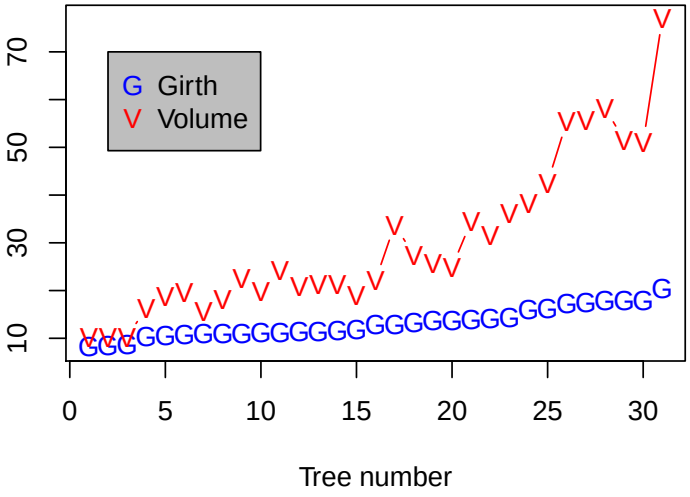
```
plot(Girth)
segments(x0=rep(c(5,10), 7), y0=7:20, x1=rep(c(5,10), 7)+2, y1=(7:20)+2 ) # line segments
arrows(x0=13,y0=16,x1=16,y1=17) # arrows
rect(xleft=10, ybottom=12,  xright=12, ytop=16) # rectangle
polygon(x=c(10,11,12,11.5,10.5), y=c(9,9.5,10,10.5,9.8), col='grey') # polygon
title(main='This plot makes no sense', sub='Or does it?')
mtext('Printing in the margins', side=2) # math text
mtext(expression(alpha==log(f[i])), side=4)
```

Things to note:

- The following functions add the elements they are named after: `segments`, `arrows`, `rect`, `polygon`, `title`.
- `mtext` adds mathematical text, which needs to be wrapped in `expression()`. For more information for mathematical annotation see `?plotmath`.

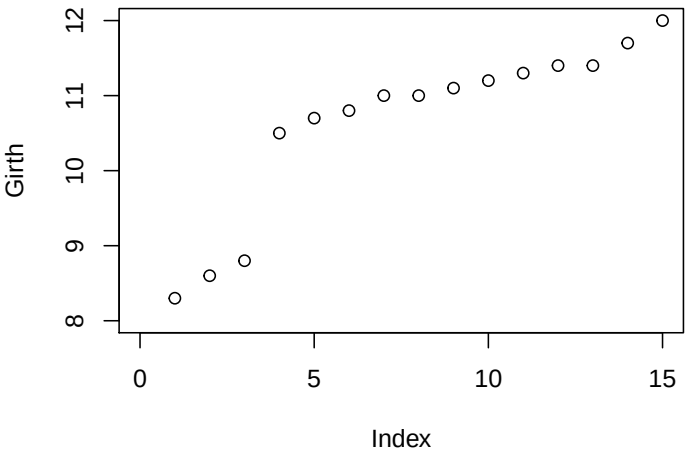
Add a legend.

```
plot(Girth, pch='G',ylim=c(8,77), xlab='Tree number', ylab='', type='b', col='blue')
points(Volume, pch='V', type='b', col='red')
legend(x=2, y=70, legend=c('Girth', 'Volume'), pch=c('G','V'), col=c('blue','red'), bg='grey')
```



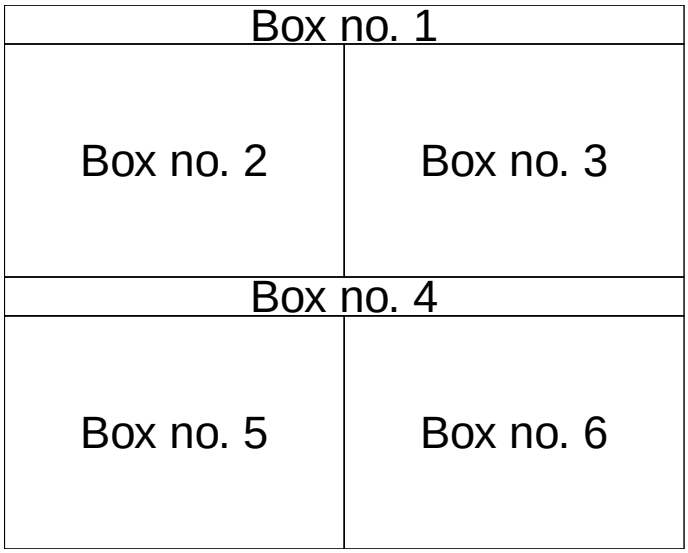
Adjusting Axes with `xlim` and `ylim`.

```
plot(Girth, xlim=c(0,15), ylim=c(8,12))
```



Use `layout` for complicated plot layouts.

```
A<-matrix(c(1,1,2,3,4,4,5,6), byrow=TRUE, ncol=2)
layout(A,heights=c(1/14,6/14,1/14,6/14))

oma.saved <- par("oma")
par(oma = rep.int(0, 4))
par(oma = oma.saved)
o.par <- par(mar = rep.int(0, 4))
for (i in seq_len(6)) {
    plot.new()
    box()
    text(0.5, 0.5, paste('Box no.',i), cex=3)
}
```



Always detach.

```
detach(trees)
```

10.1.2 Exporting a Plot

The pipeline for exporting graphics is similar to the export of data. Instead of the `write.table` or `save` functions, we will use the `pdf`, `tiff`, `png`, functions. Depending on the type of desired output.

Check and set the working directory.

```
getwd()
setwd("/tmp/")
```

Export tiff.

```
tiff(filename='graphicExample.tiff')
plot(rnorm(100))
dev.off()
```

Things to note:

- The `tiff` function tells R to open a .tiff file, and write the output of a plot.
- Only a single (the last) plot is saved.
- `dev.off` to close the tiff device, and return the plotting to the R console (or RStudio).

If you want to produce several plots, you can use a counter in the file's name. The counter uses the printf format string.

```
tiff(filename='graphicExample%d.tiff') #Creates a sequence of files
plot(rnorm(100))
boxplot(rnorm(100))
hist(rnorm(100))
dev.off()
```

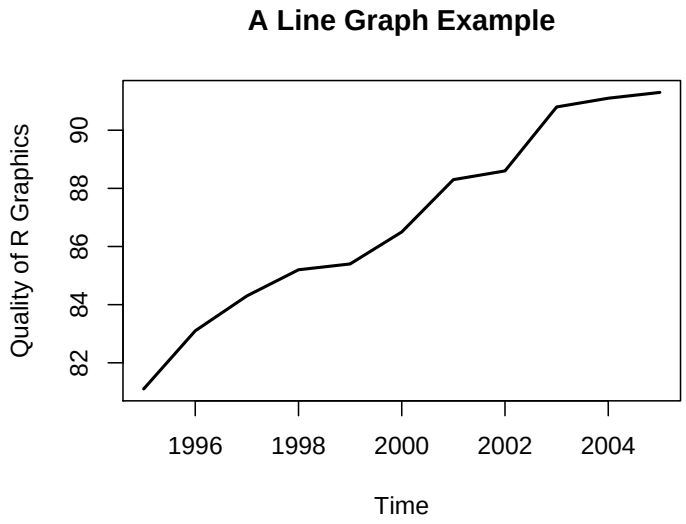
To see the list of all open devices use `dev.list()`. To close **all** device, (not only the last one), use `graphics.off()`.

See `?pdf` and `?jpeg` for more info.

10.1.3 Fancy graphics Examples

10.1.3.1 Line Graph

```
x = 1995:2005
y = c(81.1, 83.1, 84.3, 85.2, 85.4, 86.5, 88.3, 88.6, 90.8, 91.1, 91.3)
plot.new()
plot.window(xlim = range(x), ylim = range(y))
abline(h = -4:4, v = -4:4, col = "lightgrey")
lines(x, y, lwd = 2)
title(main = "A Line Graph Example",
        xlab = "Time",
        ylab = "Quality of R Graphics")
axis(1)
axis(2)
box()
```

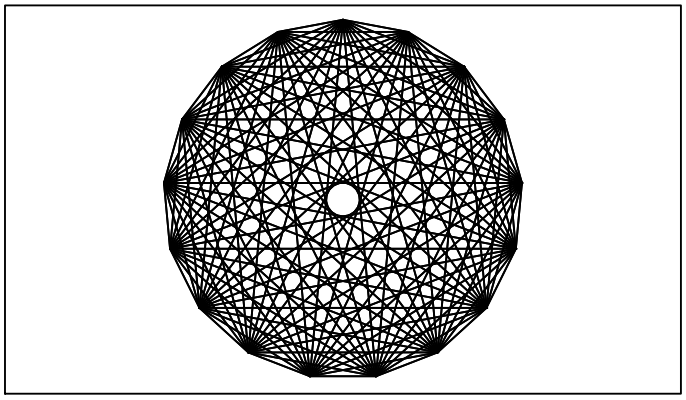


Things to note:

- `plot.new` creates a new, empty, plotting device.
- `plot.window` determines the limits of the plotting region.
- `axis` adds the axes, and `box` the framing box.
- The rest of the elements, you already know.

10.1.3.2 Rosette

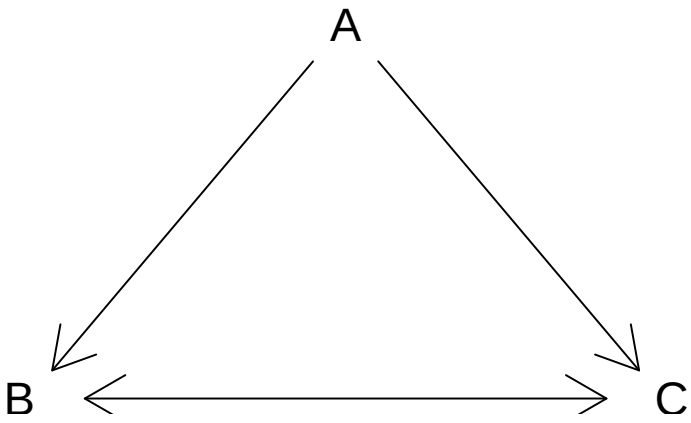
```
n = 17
theta = seq(0, 2 * pi, length = n + 1)[1:n]
x = sin(theta)
y = cos(theta)
v1 = rep(1:n, n)
v2 = rep(1:n, rep(n, n))
plot.new()
plot.window(xlim = c(-1, 1), ylim = c(-1, 1), asp = 1)
segments(x[v1], y[v1], x[v2], y[v2])
box()
```



10.1.3.3 Arrows

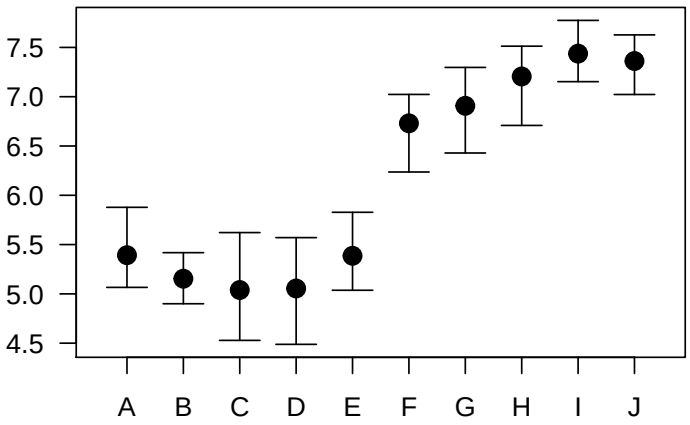
```
plot.new()
plot.window(xlim = c(0, 1), ylim = c(0, 1))
arrows(.05, .075, .45, .9, code = 1)
arrows(.55, .9, .95, .075, code = 2)
```

```
arrows(.1, 0, .9, 0, code = 3)
text(.5, 1, "A", cex = 1.5)
text(0, 0, "B", cex = 1.5)
text(1, 0, "C", cex = 1.5)
```



10.1.3.4 Arrows as error bars

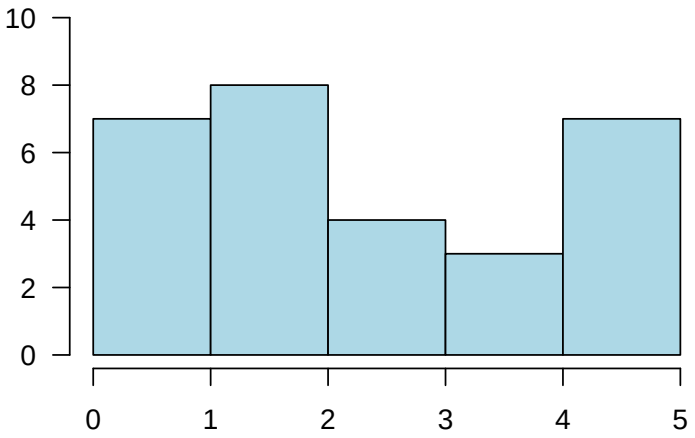
```
x = 1:10
y = runif(10) + rep(c(5, 6.5), c(5, 5))
yl = y - 0.25 - runif(10)/3
yu = y + 0.25 + runif(10)/3
plot.new()
plot.window(xlim = c(0.5, 10.5), ylim = range(yl, yu))
arrows(x, yl, x, yu, code = 3, angle = 90, length = .125)
points(x, y, pch = 19, cex = 1.5)
axis(1, at = 1:10, labels = LETTERS[1:10])
axis(2, las = 1)
box()
```



10.1.3.5 Histogram

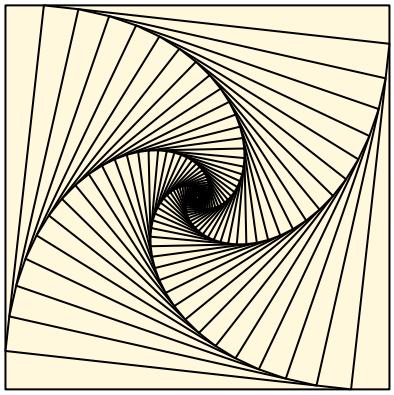
A histogram is nothing but a bunch of rectangle elements.

```
plot.new()
plot.window(xlim = c(0, 5), ylim = c(0, 10))
rect(0:4, 0, 1:5, c(7, 8, 4, 3), col = "lightblue")
axis(1)
axis(2, las = 1)
```



10.1.3.5.1 Spiral Squares

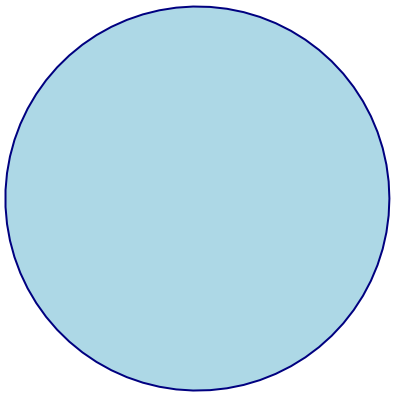
```
plot.new()
plot.window(xlim = c(-1, 1), ylim = c(-1, 1), asp = 1)
x = c(-1, 1, 1, -1)
y = c( 1, 1, -1, -1)
polygon(x, y, col = "cornsilk")
vertex1 = c(1, 2, 3, 4)
vertex2 = c(2, 3, 4, 1)
for(i in 1:50) {
    x = 0.9 * x[vertex1] + 0.1 * x[vertex2]
    y = 0.9 * y[vertex1] + 0.1 * y[vertex2]
    polygon(x, y, col = "cornsilk")
}
```



10.1.3.6 Circles

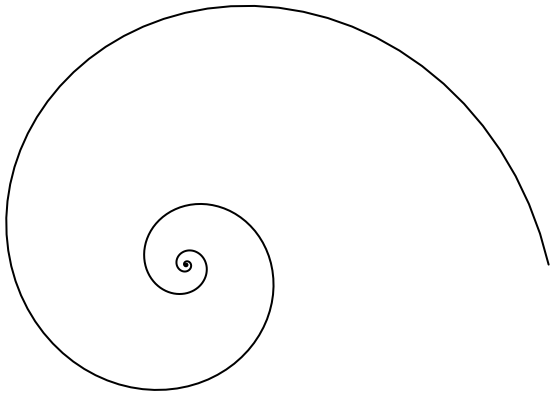
Circles are just dense polygons.

```
R = 1
xc = 0
yc = 0
n = 72
t = seq(0, 2 * pi, length = n)[1:(n-1)]
x = xc + R * cos(t)
y = yc + R * sin(t)
plot.new()
plot.window(xlim = range(x), ylim = range(y), asp = 1)
polygon(x, y, col = "lightblue", border = "navyblue")
```



10.1.3.7 Spiral

```
k = 5
n = k * 72
theta = seq(0, k * 2 * pi, length = n)
R = .98^(1:n - 1)
x = R * cos(theta)
y = R * sin(theta)
plot.new()
plot.window(xlim = range(x), ylim = range(y), asp = 1)
lines(x, y)
```



10.2 The ggplot2 System

The philosophy of **ggplot2** is very different from the **graphics** device. Recall, in **ggplot2**, a plot is a object. It can be queried, it can be changed, and among other things, it can be plotted.

ggplot2 provides a convenience function for many plots: `qplot`. We take a non-typical approach by ignoring `qplot`, and presenting the fundamental building blocks. Once the building blocks have been understood, mastering `qplot` will be easy.

The following is taken from UCLA's idre.

A **ggplot2** object will have the following elements:

- **Data** the data frame holding the data to be plotted.
- **Aes** defines the mapping between variables to their visualization.
- **Geoms** are the objects/shapes you add as layers to your graph.
- **Stats** are statistical transformations when you are not plotting the raw data, such as the mean or confidence intervals.

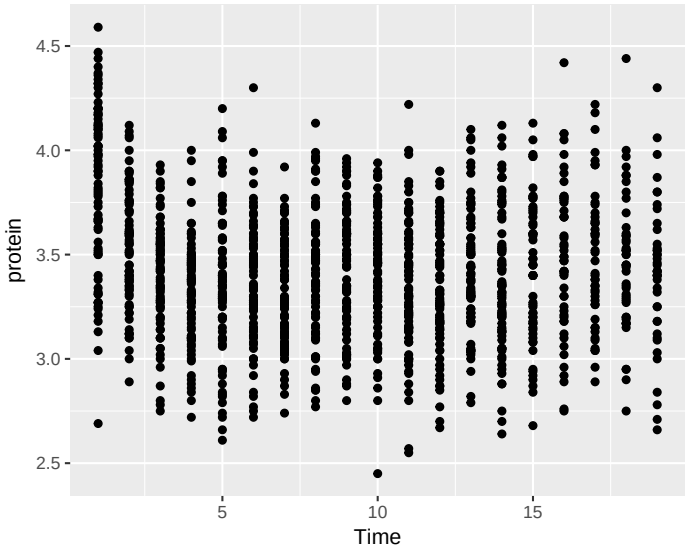
- **Faceting** splits the data into subsets to create multiple variations of the same graph (paneling).

The `nlme::Milk` dataset has the protein level of various cows, at various times, with various diets.

```
library(nlme)
data(Milk)
head(Milk)
```

```
## Grouped Data: protein ~ Time | Cow
##   protein Time Cow   Diet
## 1    3.63    1 B01 barley
## 2    3.57    2 B01 barley
## 3    3.47    3 B01 barley
## 4    3.65    4 B01 barley
## 5    3.89    5 B01 barley
## 6    3.73    6 B01 barley
```

```
library(ggplot2)
ggplot(data = Milk, aes(x=Time, y=protein)) +
  geom_point()
```

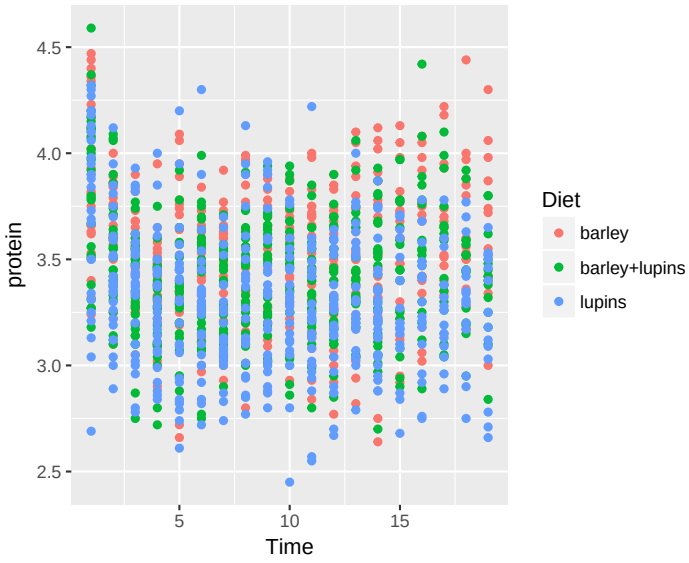


Things to note:

- The `ggplot` function is the constructor of the **ggplot2** object. If the object is not assigned, it is plotted.
- The `aes` argument tells R that the `Time` variable in the `Milk` data is the x axis, and protein is y.
- The `geom_point` defines the **Geom**, i.e., it tells R to plot the points as they are (and not lines, histograms, etc.).
- The **ggplot2** object is build by compounding its various elements separated by the `+` operator.
- All the variables that we will need are assumed to be in the `Milk` data frame. This means that (a) the data needs to be a data frame (not a matrix for instance), and (b) we will not be able to use variables that are not in the `Milk` data frame.

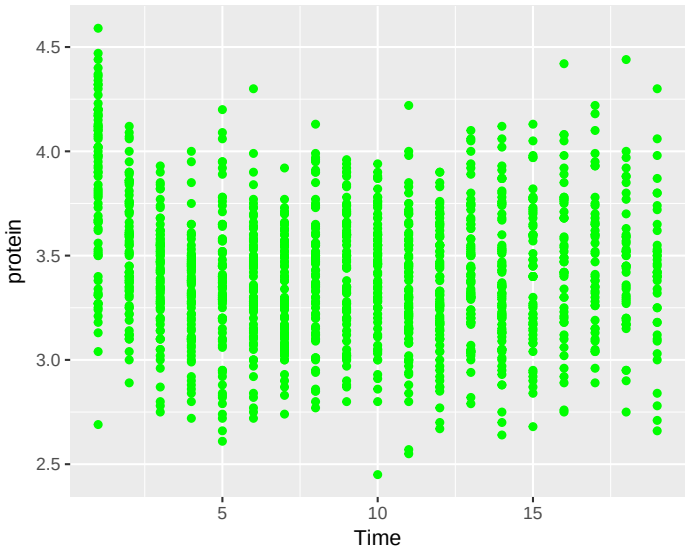
Let's add some color.

```
ggplot(data = Milk, aes(x=Time, y=protein)) +
  geom_point(aes(color=Diet))
```



The `color` argument tells R to use the variable `Diet` as the coloring. A legend is added by default. If we wanted a fixed color, and not a variable dependent color, `color` would have been put outside the `aes` function.

```
ggplot(data = Milk, aes(x=Time, y=protein)) +
  geom_point(color="green")
```

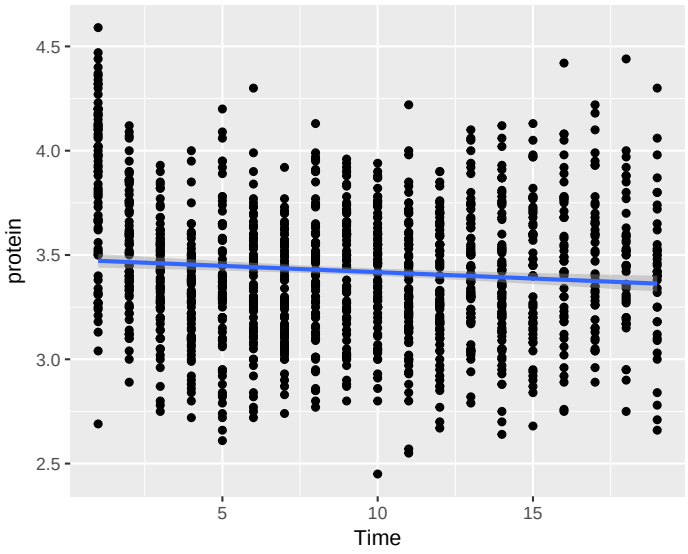


Let's save the **ggplot2** object so we can reuse it. Notice it is not plotted.

```
p <- ggplot(data = Milk, aes(x=Time, y=protein)) +
  geom_point()
```

We can changeˆ{In the Object-Oriented Programming lingo, this is known as mutating} existing plots using the `+` operator. Here, we add a smoothing line to the plot `p`.

```
p + geom_smooth(method = 'gam')
```

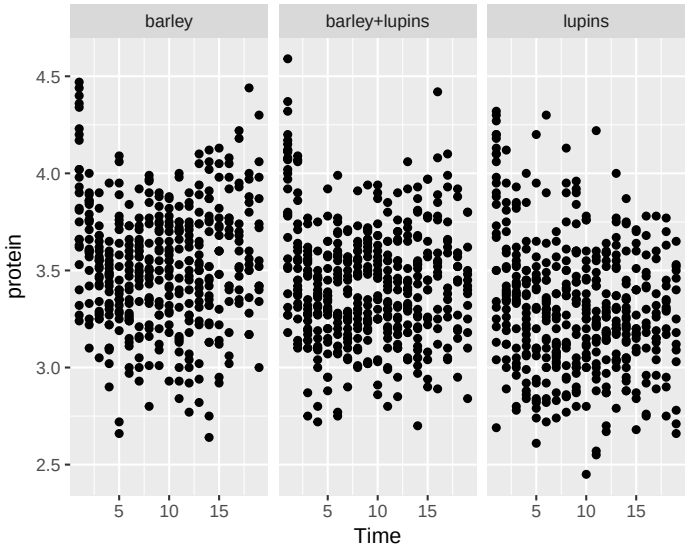


Things to note:

- The smoothing line is a layer added with the `geom_smooth()` function.
- Lacking arguments of its own, the new layer will inherit the `aes` of the original object, x and y variables in particular.

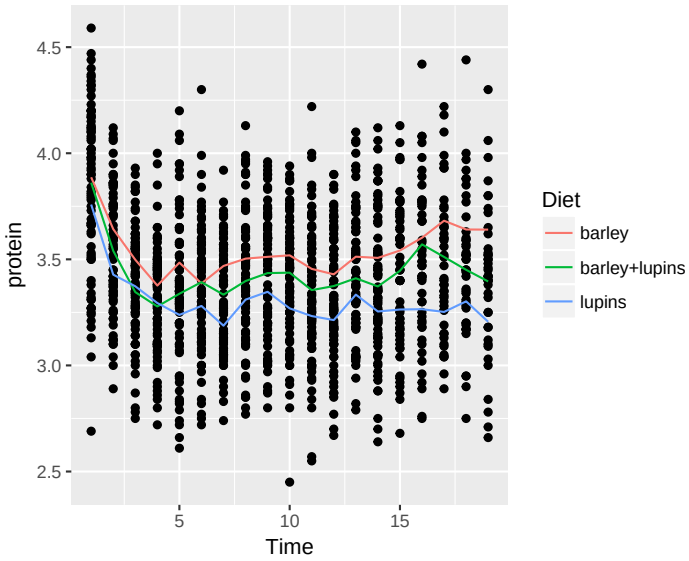
To split the plot along some variable, we use faceting, done with the `facet_wrap` function.

```
p + facet_wrap(~Diet)
```



Instead of faceting, we can add a layer of the mean of each `Diet` subgroup, connected by lines.

```
p + stat_summary(aes(color=Diet), fun.y="mean", geom="line")
```



Things to note:

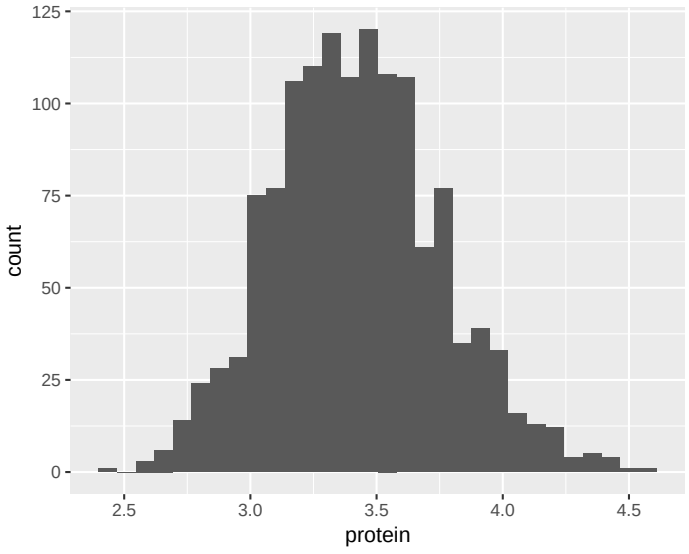
- `stat_summary` adds a statistical summary.
- The summary is applied along `Diet` subgroups, because of the `color=Diet` aesthetic, which has already split the data.
- The summary to be applied is the mean, because of `fun.y="mean"`.
- The group means are connected by lines, because of the `geom="line"` argument.

What layers can be added using the **geoms** family of functions?

- `geom_bar`: bars with bases on the x-axis.
- `geom_boxplot`: boxes-and-whiskers.
- `geom_errorbar`: T-shaped error bars.
- `geom_histogram`: histogram.
- `geom_line`: lines.
- `geom_point`: points (scatterplot).
- `geom_ribbon`: bands spanning y-values across a range of x-values.
- `geom_smooth`: smoothed conditional means (e.g. loess smooth).

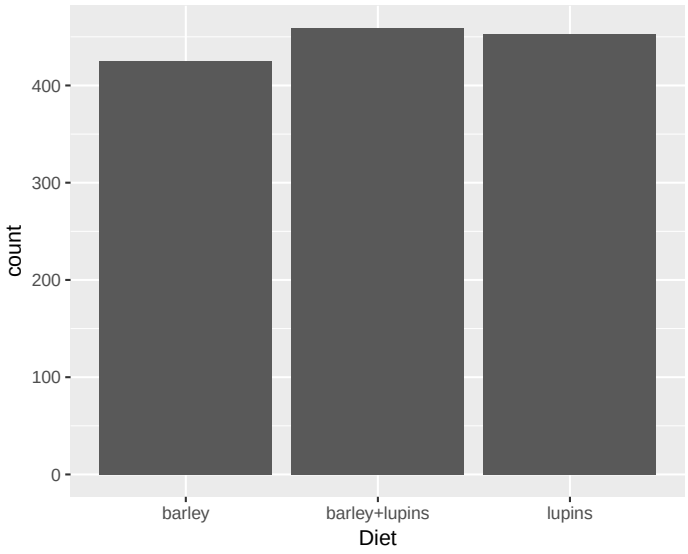
To demonstrate the layers added with the `geoms_*` functions, we start with a histogram.

```
pro <- ggplot(Milk, aes(x=protein))
pro + geom_histogram(bins=30)
```



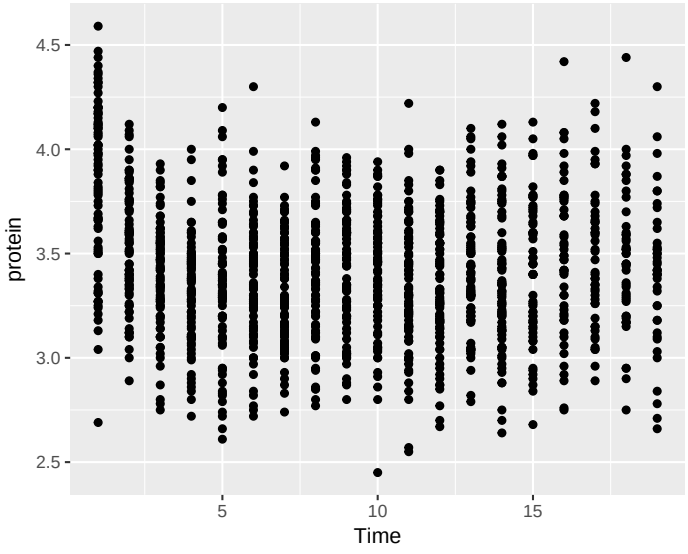
A bar plot.

```
ggplot(Milk, aes(x=Diet)) +
  geom_bar()
```



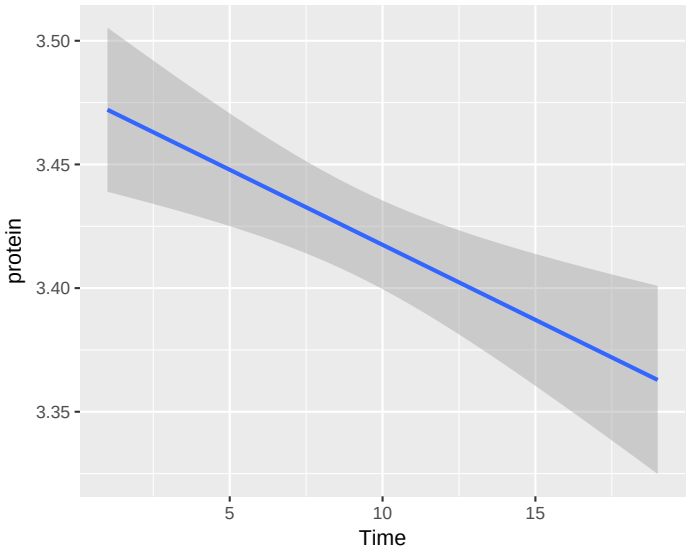
A scatter plot.

```
tp <- ggplot(Milk, aes(x=Time, y=protein))
tp + geom_point()
```



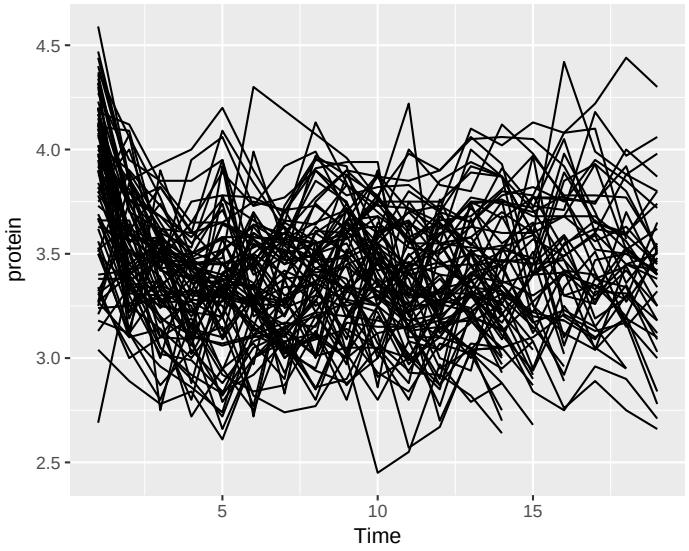
A smooth regression plot, reusing the `tp` object.

```
tp + geom_smooth(method='gam')
```



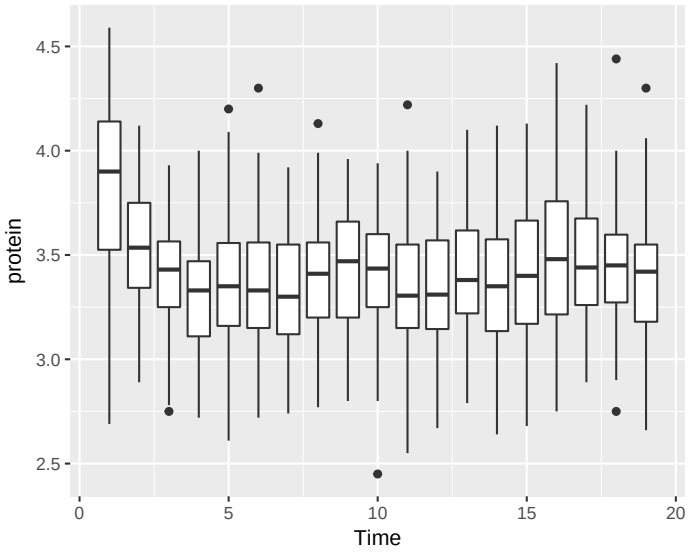
And now, a simple line plot, reusing the `tp` object, and connecting lines along `Cow`.

```
tp + geom_line(aes(group=Cow))
```



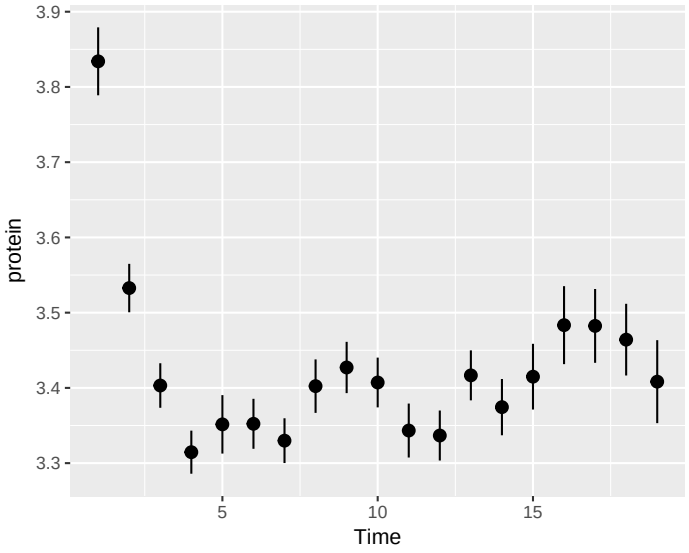
The line plot is completely incomprehensible. Better look at boxplots along time (even if omitting the `Cow` information).

```
tp + geom_boxplot(aes(group=Time))
```



We can do some statistics for each subgroup. The following will compute the mean and standard errors of `protein` at each time point.

```
ggplot(Milk, aes(x=Time, y=protein)) +
  stat_summary(fun.data = 'mean_se')
```

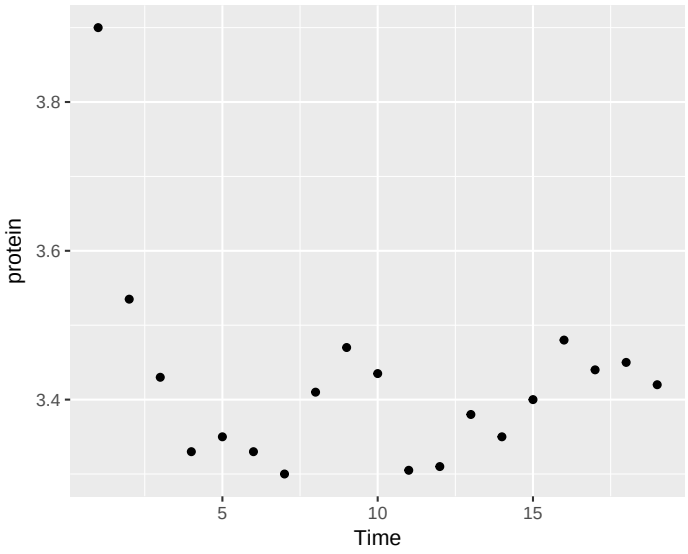


Some popular statistical summaries, have gained their own functions:

- `mean_cl_boot`: mean and bootstrapped confidence interval (default 95%).
- `mean_cl_normal`: mean and Gaussian (t-distribution based) confidence interval (default 95%).
- `mean_dsl`: mean plus or minus standard deviation times some constant (default constant=2).
- `median_hilow`: median and outer quantiles (default outer quantiles = 0.025 and 0.975).

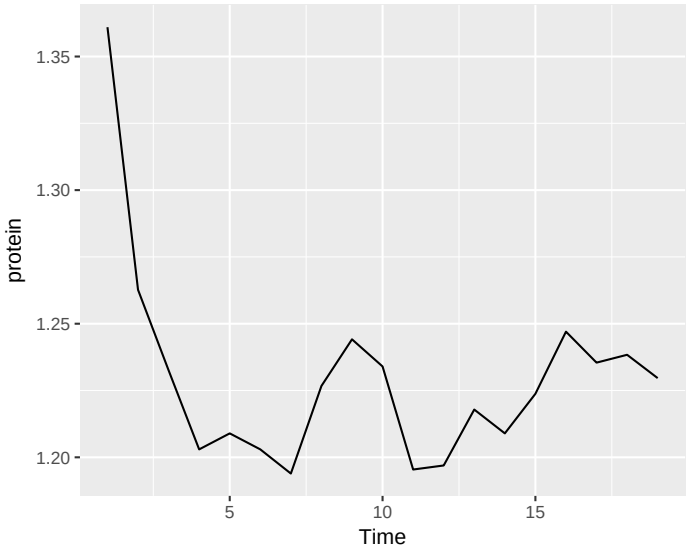
For less popular statistical summaries, we may specify the statistical function in `stat_summary`. The median is a first example.

```
ggplot(Milk, aes(x=Time, y=protein)) +
  stat_summary(fun.y="median", geom="point")
```



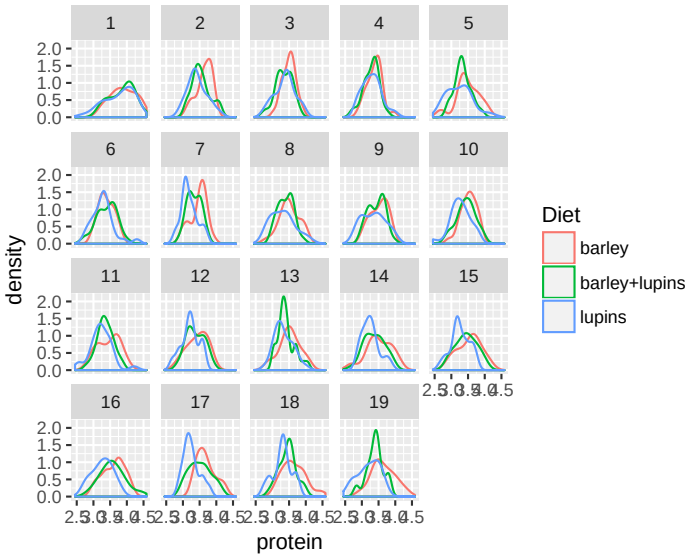
We can also define our own statistical summaries.

```
medianlog <- function(y) {median(log(y))}
ggplot(Milk, aes(x=Time, y=protein)) +
  stat_summary(fun.y="medianlog", geom="line")
```



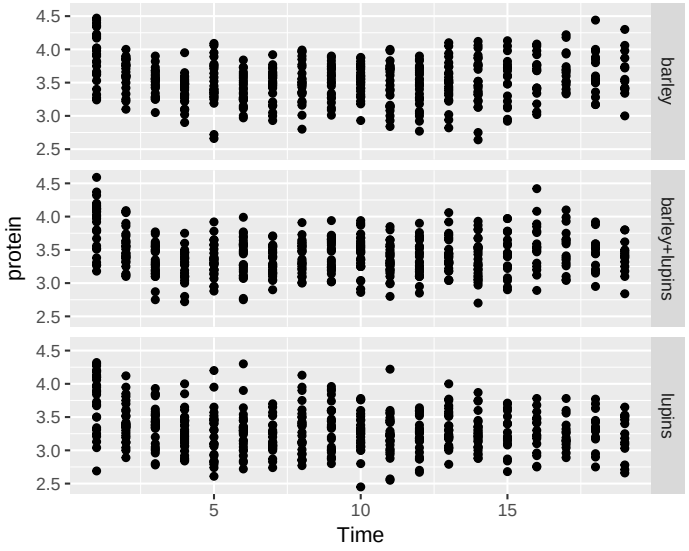
Faceting allows to split the plotting along some variable. `face_wrap` tells R to compute the number of columns and rows of plots automatically.

```
ggplot(Milk, aes(x=protein, color=Diet)) +
  geom_density() +
  facet_wrap(~Time)
```



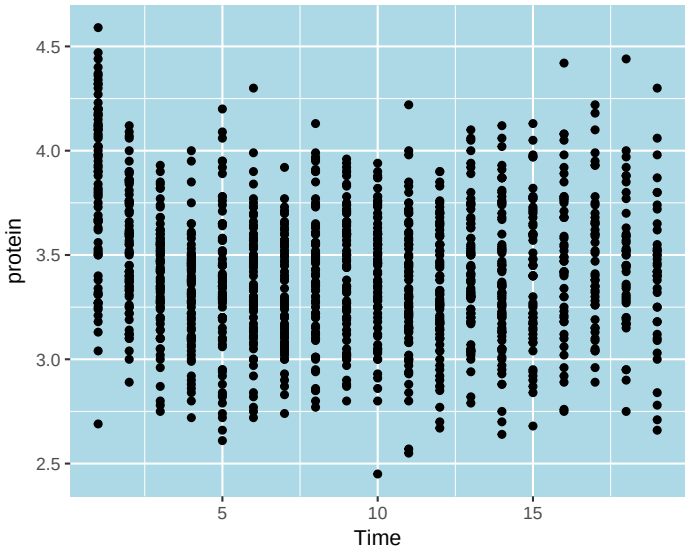
`facet_grid` forces the plot to appear allow rows or columns, using the `~` syntax.

```
ggplot(Milk, aes(x=Time, y=protein)) +
  geom_point() +
  facet_grid(Diet~.) # `.~Diet` to split along columns and not rows.
```

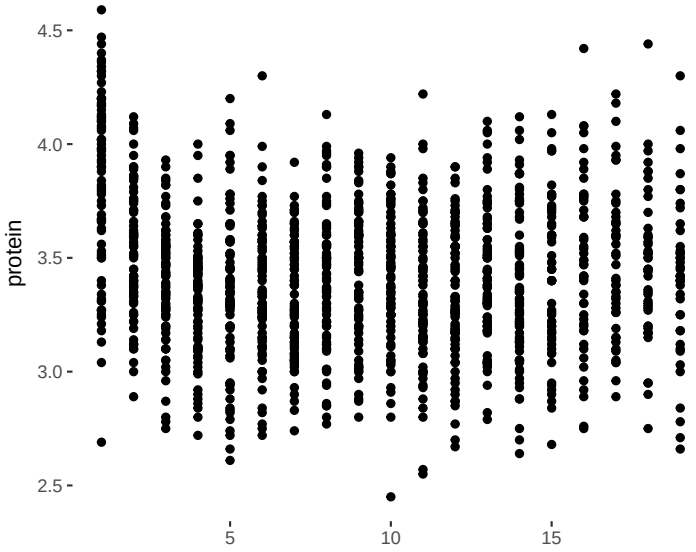


To control the looks of the plot, **ggplot2** uses **themes**.

```
ggplot(Milk, aes(x=Time, y=protein)) +
  geom_point() +
  theme(panel.background=element_rect(fill="lightblue"))
```




```
ggplot(Milk, aes(x=Time, y=protein)) +
  geom_point() +
  theme(panel.background=element_blank(),
        axis.title.x=element_blank())
```



Saving plots can be done using `ggplot2::ggsave`, or with `pdf` like the **graphics** plots:

```
pdf(file = 'myplot.pdf')
print(tp) # You will need an explicit print command!
dev.off()
```

Finally, what every user of **ggplot2** constantly uses, is the (excellent!) online documentation at http://docs.ggplot2.org.

10.2.1 Extensions of the ggplot2 System

Because **ggplot2** plots are R objects, they can be used for computations and altered. Many authors, have thus extended the basic **ggplot2** functionality. A list of **ggplot2** extensions is curated by Daniel Emaasit at http://www.ggplot2-exts.org. The RStudio team has its own list of recommended packages at RStartHere.

10.3 Interactive Graphics

As already mentioned, the recent and dramatic advancement in interactive visualization was made possible by the advances in web technologies, and the D3.JS JavaScript library in particular. This is because it allows developers to rely on existing libraries designed for web browsing instead of re-implementing interactive visualizations. These libraries are more visually pleasing, and computationally efficient, than anything they could have developed themselves.

The **htmlwidgets** package does not provide visualization, but rather, it facilitates the creation of new interactive visualizations. This is because it handles all the technical details that are required to use R output within JavaScript visualization libraries.

For a list of interactive visualization tools that rely on **htmlwidgets** see the RStartsHere page. In the following sections, we discuss a selected subset.

10.3.1 Plotly

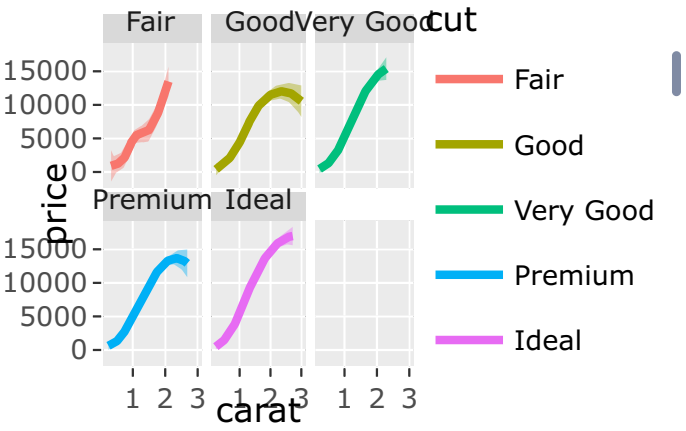
You can create nice interactive graphs using `plotly::plot_ly`:

```
library(plotly)
set.seed(100)
d <- diamonds[sample(nrow(diamonds), 1000), ]
```

```
plot_ly(data = d, x = ~carat, y = ~price, color = ~carat, size = ~carat, text = ~paste("Clarity: ", clarity)
```

More conveniently, any **ggplot2** graph can be made interactive using `plotly::ggplotly`:

```
p <- ggplot(data = d, aes(x = carat, y = price)) +
  geom_smooth(aes(colour = cut, fill = cut), method = 'loess') +
  facet_wrap(~ cut) # make ggplot
ggplotly(p) # from ggplot to plotly
```



How about exporting **plotly** objects? Well, a **plotly** object is nothing more than a little web site: an HTML file. When showing a **plotly** figure, RStudio merely servers you as a web browser. You could, alternatively, export this HTML file to send your colleagues as an email attachment, or embed it in a web site. To export these, use the `plotly::export` or the `htmlwidgets::saveWidget` functions.

For more on **plotly** see https://plot.ly/r/.

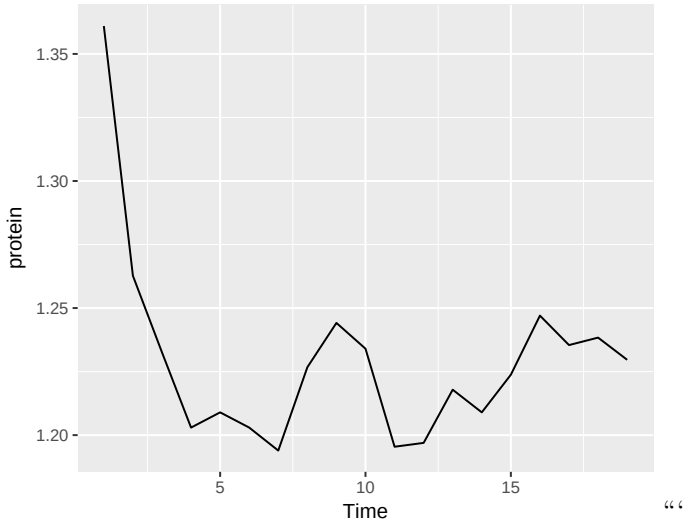
10.4 Bibliographic Notes

For the **graphics** package, see R Core Team (2016). For **ggplot2** see Wickham (2009). For the theory underlying **ggplot2**, i.e. the Grammer of Graphics, see Wilkinson (2006). A video by one of my heroes, Brian Caffo, discussing **graphics** vs. **ggplot2**.

10.5 Practice Yourself

1. Go to the Fancy Graphics Section 10.1.3. Try parsing the commands in your head.
2. Recall the `medianlog` example and replace the `medianlog` function with a harmonic mean.

```
medianlog <- function(y) {median(log(y))}
ggplot(Milk, aes(x=Time, y=protein)) +
  stat_summary(fun.y="medianlog", geom="line")
```



“‘

3. Write a function that creates a boxplot from scratch. See how I built a line graph in Section 10.1.3.
4. Export my plotly example using the RStudio interface and send it to yourself by email.

ggplot2:

1. Read about the “oats” dataset using `? MASS::oats`.
    1. Inspect, visually, the dependency of the yield (Y) in the Varieties (V) and the Nitrogen treatment (N).
    2. Compute the mean and the standard error of the yield for every value of Varieties and Nitrogen treatment.
    3. Change the axis labels to be informative with `labs` function and give a title to the plot with `ggtitle` function.
2. Read about the “mtcars” data set using `? mtcars`.
    1. Inspect, visually, the dependency of the Fuel consumption (mpg) in the weight (wt)
    2. Inspect, visually, the assumption that the Fuel consumption also depends on the number of cylinders.
    3. Is there an interaction between the number of cylinders to the weight (i.e. the slope of the regression line is different between the number of cylinders)? Use `geom_smooth`.

Chapter 11

Reports

If you have ever written a report, you are probably familiar with the process of preparing your figures in some software, say R, and then copy-pasting into your text editor, say MS Word. While very popular, this process is both tedious, and plain painful if your data has changed and you need to update the report. Wouldn't it be nice if you could produce figures and numbers from within the text of the report, and everything else would be automated? It turns out it is possible. There are actually several systems in R that allow this. We start with a brief review.

1. **Sweave**: *LaTeX* is a markup language that compiles to *Tex* programs that compile, in turn, to documents (typically PS or PDFs). If you never heard of it, it may be because you were born the the MS Windows+MS Word era. You should know, however, that *LaTeX* was there much earlier, when computers were mainframes with text-only graphic devices. You should also know that *LaTeX* is still very popular (in some communities) due to its very rich markup syntax, and beautiful output. *Sweave* (Leisch, 2002) is a compiler for *LaTeX* that allows you do insert R commands in the *LaTeX* source file, and get the result as part of the outputted PDF. It's name suggests just that: it allows to weave S[1] output into the document, thus, Sweave.
2. **knitr**: *Markdown* is a text editing syntax that, unlike *LaTeX*, is aimed to be human-readable, but also compilable by a machine. If you ever tried to read HTML or *LaTeX* source files, you may understand why human-readability is a desirable property. There are many *markdown* compilers. One of the most popular is Pandoc, written by the Berkeley philosopher(!) Jon MacFarlane. The availability of Pandoc gave Yihui Xie, a name to remember, the idea that it is time for Sweave to evolve. Yihui thus wrote **knitr** (Xie, 2015), which allows to write human readable text in *Rmarkdown*, a superset of *markdown*, compile it with R and the compile it with Pandoc. Because Pandoc can compile to PDF, but also to HTML, and DOCX, among others, this means that you can write in Rmarkdown, and get output in almost all text formats out there.
3. **bookdown**: **Bookdown** (Xie, 2016) is an evolution of **knitr**, also written by Yihui Xie, now working for RStudio. The text you are now reading was actually written in **bookdown**. It deals with the particular needs of writing large documents, and cross referencing in particular (which is very challenging if you want the text to be human readable).
4. **Shiny**: Shiny is essentially a framework for quick web-development. It includes (i) an abstraction layer that specifies the layout of a web-site which is our report, (ii) the command to start a web server to deliver the site. For more on Shiny see Chang et al. (2017).

11.1 knitr

11.1.1 Installation

To run **knitr** you will need to install the package.

```
install.packages('knitr')
```

It is also recommended that you use it within RStudio (version>0.96), where you can easily create a new `.Rmd` file.

[1] Recall, S was the original software from which R evolved.

11.1.2 Pandoc Markdown

Because **knitr** builds upon *Pandoc markdown*, here is a simple example of markdown text, to be used in a `.Rmd` file, which can be created using the *File-> New File -> R Markdown* menu of RStudio.

Underscores or asterisks for `_italics1_` and `*italics2*` return *italics1* and *italics2*. Double underscores or asterisks for `__bold1__` and `**bold2**` return **bold1** and **bold2**. Subscripts are enclosed in tildes, `like~this~` (like$_{\text{this}}$), and superscripts are enclosed in carets `like^this^` (like$^{\text{this}}$).

For links use `[text](link)`, like `[my site](www.john-ros.com)`. An image is the same as a link, starting with an exclamation, like this `![image caption](image path)`.

An itemized list simply starts with hyphens preceeded by a blank line (don't forget that!):

```
- bullet
- bullet
    - second level bullet
    - second level bullet
```

Compiles into:

- bullet
- bullet
  - second level bullet
  - second level bullet

An enumerated list starts with an arbitrary number:

```
1. number
1. number
    1. second level number
    1. second level number
```

Compiles into:

1. number
2. number
   1. second level number
   2. second level number

For more on markdown see https://bookdown.org/yihui/bookdown/markdown-syntax.html.

11.1.3 Rmarkdown

Rmarkdown, is an extension of *markdown* due to RStudio, that allows to incorporate R expressions in the text, that will be evaluated at the time of compilation, and the output automatically inserted in the outputted text. The output can be a `.PDF`, `.DOCX`, `.HTML` or others, thanks to the power of *Pandoc*.

The start of a code chunk is indicated by three backticks and the end of a code chunk is indicated by three backticks. Here is an example.

````
```{r  eval=FALSE}
rnorm(10)
```
````

This chunk will compile to the following output (after setting `eval=FALSE` to `eval=TRUE`):

```
rnorm(10)
```

```
##  [1] -1.4462875  0.3158558 -0.3427475 -1.9313531  0.2428210 -0.3627679
##  [7]  2.4327289  0.5920912 -0.5762008  0.4066282
```

Things to note:

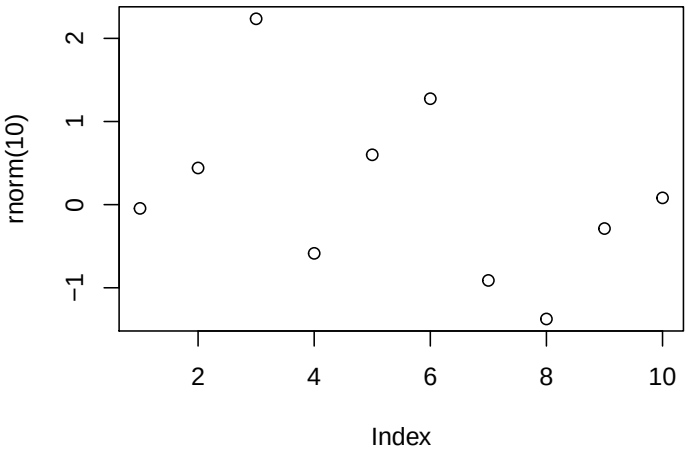
- The evaluated expression is added in a chunk of highlighted text, before the R output.
- The output is prefixed with `##`.
- The `eval=` argument is not required, since it is set to `eval=TRUE` by default. It does demonstrate how to set the options of the code chunk.

In the same way, we may add a plot:

````
```{r  eval=FALSE}
plot(rnorm(10))
```
````

which compiles into

```r
plot(rnorm(10))
```



TODO: more code chunk options.

You can also call r expressions inline. This is done with a single tick and the `r` argument. For instance:

> `` `r rnorm(1)` `` is a random Gaussian

will output

> 0.3378953 is a random Gaussian.

11.1.4 BibTex

BibTex is both a file format and a compiler. The bibtex compiler links documents to a reference database stored in the `.bib` file format.

Bibtex is typically associated with Tex and LaTex typesetting, but it also operates within the markdown pipeline.

Just store your references in a `.bib` file, add a `bibliography: yourFile.bib` in the YML preamble of your Rmarkdown file, and call your references from the Rmarkdown text using `@referencekey`. Rmarkdow will take care of creating the bibliography, and linking to it from the text.

11.1.5 Compiling

Once you have your `.Rmd` file written in RMarkdown, **knitr** will take care of the compilation for you. You can call the `knitr::knitr` function directly from some `.R` file, or more conveniently, use the RStudio (0.96) Knit button above the text editing window. The location of the output file will be presented in the console.

11.2 bookdown

As previously stated, **bookdown** is an extension of **knitr** intended for documents more complicated than simple reports– such as books. Just like **knitr**, the writing is done in **RMarkdown**. Being an extension of **knitr**, **bookdown** does allow some markdowns that are not supported by other compilers. In particular, it has a more powerful cross referencing system.

11.3 Shiny

Shiny (Chang et al., 2017) is different than the previous systems, because it sets up an interactive web-site, and not a static file. The power of Shiny is that the layout of the web-site, and the settings of the web-server, is made with several simple R commands, with no need for web-programming. Once you have your app up and running, you can setup your own Shiny server on the web, or publish it via Shinyapps.io. The freemium versions of the service can deal with a small amount of traffic. If you expect a lot of traffic, you will probably need the paid versions.

11.3.1 Installation

To setup your first Shiny app, you will need the **shiny** package. You will probably want RStudio, which facilitates the process.

```
install.packages('shiny')
```

Once installed, you can run an example app to get the feel of it.

```
library(shiny)
runExample("01_hello")
```

Remember to press the **Stop** button in RStudio to stop the web-server, and get back to RStudio.

11.3.2 The Basics of Shiny

Every Shiny app has two main building blocks.

1. A user interface, specified via the `ui.R` file in the app's directory.
2. A server side, specified via the `server.R` file, in the app's directory.

You can run the app via the **RunApp** button in the RStudio interface, of by calling the app's directory with the `shinyApp` or `runApp` functions– the former designed for single-app projects, and the latter, for multiple app projects.

```
shiny::runApp("my_app") # my_app is the app's directory.
```

The site's layout, is specified in the `ui.R` file using one of the *layout functions*. For instance, the function `sidebarLayout`, as the name suggest, will create a sidebar. More layouts are detailed in the layout guide.

The active elements in the UI, that control your report, are known as *widgets*. Each widget will have a unique `inputId` so that it's values can be sent from the UI to the server. More about widgets, in the widget gallery.

The `inputId` on the UI are mapped to `input` arguments on the server side. The value of the `mytext inputId` can be queried by the server using `input$mytext`. These are called *reactive values*. The way the server "listens" to the UI, is governed by a set of functions that must wrap the `input` object. These are the `observe`, `reactive`, and `reactive*` class of functions.

With `observe` the server will get triggered when any of the reactive values change. With `observeEvent` the server will only be triggered by specified reactive values. Using `observe` is easier, and `observeEvent` is more prudent programming.

A `reactive` function is a function that gets triggered when a reactive element changes. It is defined on the server side, and reside within an `observe` function.

We now analyze the `1_Hello` app using these ideas. Here is the `ui.R` file.

```
library(shiny)

shinyUI(fluidPage(

  titlePanel("Hello Shiny!"),

  sidebarLayout(
    sidebarPanel(
      sliderInput(inputId = "bins",
                  label = "Number of bins:",
                  min = 1,
                  max = 50,
                  value = 30)
    ),

    mainPanel(
      plotOutput(outputId = "distPlot")
    )
  )
))
```

Here is the `server.R` file:

```
library(shiny)

shinyServer(function(input, output) {

  output$distPlot <- renderPlot({
    x    <- faithful[, 2]  # Old Faithful Geyser data
    bins <- seq(min(x), max(x), length.out = input$bins + 1)

    hist(x, breaks = bins, col = 'darkgray', border = 'white')
  })
})
```

Things to note:

- `ShinyUI` is a (deprecated) wrapper for the UI.
- `fluidPage` ensures that the proportions of the elements adapt to the window side, thus, are fluid.
- The building blocks of the layout are a title, and the body. The title is governed by `titlePanel`, and the body is governed by `sidebarLayout`. The `sidebarLayout` includes the `sidebarPanel` to control the sidebar, and the `mainPanel` for the main panel.
- `sliderInput` calls a widget with a slider. Its `inputId` is `bins`, which is later used by the server within the `renderPlot` reactive function.
- `plotOutput` specifies that the content of the `mainPanel` is a plot (`textOutput` for text). This expectation is satisfied on the server side with the `renderPlot` function (`renderText`).
- `shinyServer` is a (deprecated) wrapper function for the server.
- The server runs a function with an `input` and an `output`. The elements of `input` are the `inputId`s from the UI. The elements of the `output` will be called by the UI using their `outputId`.

This is the output.

Here is another example, taken from the RStudio Shiny examples.

ui.R:

```
library(shiny)

fluidPage(

  titlePanel("Tabsets"),

  sidebarLayout(
    sidebarPanel(
      radioButtons(inputId = "dist",
                   label = "Distribution type:",
                   c("Normal" = "norm",
                     "Uniform" = "unif",
                     "Log-normal" = "lnorm",
                     "Exponential" = "exp")),
      br(), # add a break in the HTML page.

      sliderInput(inputId = "n",
                  label = "Number of observations:",
                   value = 500,
                   min = 1,
                   max = 1000)
    ),

    mainPanel(
      tabsetPanel(type = "tabs",
        tabPanel(title = "Plot", plotOutput(outputId = "plot")),
        tabPanel(title = "Summary", verbatimTextOutput(outputId = "summary")),
        tabPanel(title = "Table", tableOutput(outputId = "table"))
      )
    )
  )
)
```

server.R:

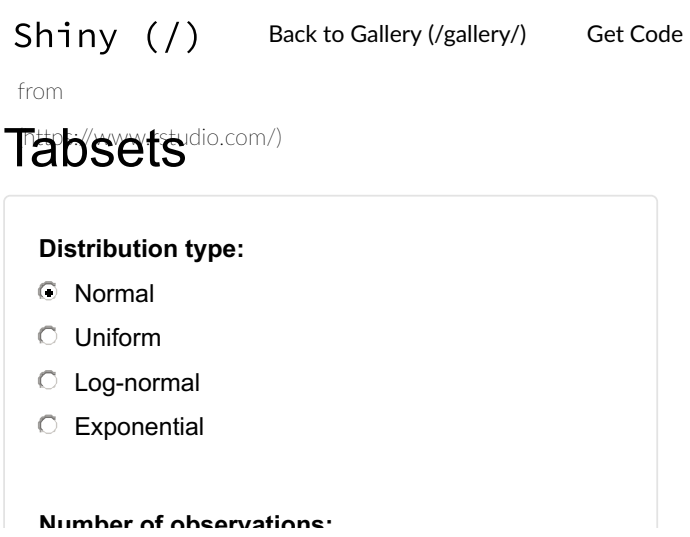
```
library(shiny)

# Define server logic for random distribution application
function(input, output) {

  data <- reactive({
    dist <- switch(input$dist,
```

```
                    norm = rnorm,
                    unif = runif,
                    lnorm = rlnorm,
                    exp = rexp,
                    rnorm)

    dist(input$n)
  })

  output$plot <- renderPlot({
    dist <- input$dist
    n <- input$n

    hist(data(), main=paste('r', dist, '(', n, ')', sep=''))
  })

  output$summary <- renderPrint({
    summary(data())
  })

  output$table <- renderTable({
    data.frame(x=data())
  })

}
```

Things to note:

- We reused the `sidebarLayout`.
- As the name suggests, `radioButtons` is a widget that produces radio buttons, above the `sliderInput` widget. Note the different `inputId`s.
- Different widgets are separated in `sidebarPanel` by commas.
- `br()` produces extra vertical spacing (break).
- `tabsetPanel` produces tabs in the main output panel. `tabPanel` governs the content of each panel. Notice the use of various output functions (`plotOutput`,`verbatimTextOutput`, `tableOutput`) with corresponding `outputId`s.
- In `server.R` we see the usual `function(input,output)`.
- The `reactive` function tells the server the trigger the function whenever `input` changes.
- The `output` object is constructed outside the `reactive` function. See how the elements of `output` correspond to the `outputId`s in the UI.

This is the output:



11.3.3 Beyond the Basics

Now that we have seen the basics, we may consider extensions to the basic report.

11.3.3.1 Widgets

- `actionButton` Action Button.
- `checkboxGroupInput` A group of check boxes.
- `checkboxInput` A single check box.
- `dateInput` A calendar to aid date selection.
- `dateRangeInput` A pair of calendars for selecting a date range.
- `fileInput` A file upload control wizard.
- `helpText` Help text that can be added to an input form.
- `numericInput` A field to enter numbers.
- `radioButtons` A set of radio buttons.
- `selectInput` A box with choices to select from.
- `sliderInput` A slider bar.
- `submitButton` A submit button.
- `textInput` A field to enter text.

See examples here.

11.3.3.2 Output Elements

The `ui.R` output types.

- `htmlOutput` raw HTML.
- `imageOutput` image.
- `plotOutput` plot.
- `tableOutput` table.
- `textOutput` text.
- `uiOutput` raw HTML.
- `verbatimTextOutput` text.

The corresponding `server.R` renderers.

- `renderImage` images (saved as a link to a source file).
- `renderPlot` plots.
- `renderPrint` any printed output.
- `renderTable` data frame, matrix, other table like structures.
- `renderText` character strings.
- `renderUI` a Shiny tag object or HTML.

Your Shiny app can use any R object. The things to remember:

- The working directory of the app is the location of `server.R`.
- The code before `shinyServer` is run only once.
- The code inside '`shinyServer` is run whenever a reactive is triggered, and may thus slow things.

To keep learning, see the RStudio's tutorial, and the Biblipgraphic notes herein.

11.3.4 shinydashboard

A template for Shiny to give it s modern look.

11.4 flexdashboard

If you want to quickly write an interactive dashboard, which is simple enough to be a static HTML file and does not need an HTML server, then Shiney may be an overkill. With **flexdashboard** you can write your dashboard a single .Rmd file, which will generate an interactive dashboard as a static HTML file.

See [http://rmarkdown.rstudio.com/flexdashboard/] for more info.

11.5 Bibliographic Notes

For RMarkdown see here. For everything on **knitr** see Yihui's blog, or the book Xie (2015). For a **bookdown** manual, see Xie (2016). For a Shiny manual, see Chang et al. (2017), the RStudio tutorial, or Zev Ross's excellent guide. Video tutorials are available here.

11.6 Practice Yourself

1. Generate a report using **knitr** with your name as title, and a scatter plot of two random variables in the body. Save it as PDF, DOCX, and HTML.
2. Recall that this book is written in **bookdown**, which is a superset of **knitr**. Go to the source .Rmd file of the first chapter, and parse it in your head: (https://raw.githubusercontent.com/johnros/Rcourse/master/02-r-basics.Rmd)

Chapter 12

Sparse Representations

Analyzing "bigdata" in R is a challenge because the workspace is memory resident, i.e., all your objects are stored in RAM. As a rule of thumb, fitting models requires about 5 times the size of the data. This means that if you have 1 GB of data, you might need about 5 GB to fit a linear models. We will discuss how to compute *out of RAM* in the Memory Efficiency Chapter 13. In this chapter, we discuss efficient representations of your data, so that it takes less memory. The fundamental idea, is that if your data is *sparse*, i.e., there are many zero entries in your data, then a naive `data.frame` or `matrix` will consume memory for all these zeroes. If, however, you have many recurring zeroes, it is more efficient to save only the non-zero entries.

When we say *data*, we actually mean the `model.matrix`. The `model.matrix` is a matrix that R grows, converting all your factors to numeric variables that can be computed with. *Dummy coding* of your factors, for instance, is something that is done in your `model.matrix`. If you have a factor with many levels, you can imagine that after dummy coding it, many zeroes will be present.

The **Matrix** package replaces the `matrix` class, with several sparse representations of matrix objects.

When using sparse representation, and the **Matrix** package, you will need an implementation of your favorite model fitting algorithm (e.g. `lm`) that is adapted to these sparse representations; otherwise, R will cast the sparse matrix into a regular (non-sparse) matrix, and you will have saved nothing in RAM.
Remark. If you are familiar with MATLAB you should know that one of the great capabilities of MATLAB, is the excellent treatment of sparse matrices with the `sparse` function.

Before we go into details, here is a simple example. We will create a factor of letters with the `letters` function. Clearly, this factor can take only 26 values. This means that 25/26 of the `model.matrix` will be zeroes after dummy coding. We will compare the memory footprint of the naive `model.matrix` with the sparse representation of the same matrix.

```
library(magrittr)
reps <- 1e6 # number of samples
y<-rnorm(reps)
x<- letters %>%
  sample(reps, replace=TRUE) %>%
  factor
```

The object `x` is a factor of letters:

```
head(x)
```

```
## [1] n x z f a i
## Levels: a b c d e f g h i j k l m n o p q r s t u v w x y z
```

We dummy code `x` with the `model.matrix` function.

```
X.1 <- model.matrix(~x-1)
head(X.1)
```

```
##   xa xb xc xd xe xf xg xh xi xj xk xl xm xn xo xp xq xr xs xt xu xv xw xx
```

```
## 1  0  0  0  0  0  0  0  0  0  0  0  0  0  1  0  0  0  0  0  0  0  0  0  0
## 2  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1
## 3  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
## 4  0  0  0  0  0  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
## 5  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
## 6  0  0  0  0  0  0  0  0  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
##   xy xz
## 1  0  0
## 2  0  0
## 3  0  1
## 4  0  0
## 5  0  0
## 6  0  0
```

We call **MatrixModels** for an implementation of `model.matrix` that supports sparse representations.

```
suppressPackageStartupMessages(library(MatrixModels))
X.2<- as(x,"sparseMatrix") %>% t # Makes sparse dummy model.matrix
head(X.2)
```

```
## 6 x 26 sparse Matrix of class "dgCMatrix"
```

```
##    [[ suppressing 26 column names 'a', 'b', 'c' ... ]]
```

```
##
## [1,] . . . . . . . . . . . . 1 . . . . . . . . . . . .
## [2,] . . . . . . . . . . . . . . . . . . . . . . . . 1 .
## [3,] . . . . . . . . . . . . . . . . . . . . . . . . . 1
## [4,] . . . . . 1 . . . . . . . . . . . . . . . . . . . .
## [5,] 1 . . . . . . . . . . . . . . . . . . . . . . . . .
## [6,] . . . . . . . . 1 . . . . . . . . . . . . . . . . .
```

Notice that the matrices have the same dimensions:

```
dim(X.1)
```

```
## [1] 1000000      26
```

```
dim(X.2)
```

```
## [1] 1000000      26
```

The memory footprint of the matrices, given by the `pryr::object_size` function, are very very different.

```
pryr::object_size(X.1)
```

```
## 264 MB
```

```
pryr::object_size(X.2)
```

```
## 12 MB
```

Things to note:

- The sparse representation takes a whole lot less memory than the non sparse.
- The `as(,"sparseMatrix")` function grows the dummy variable representation of the factor `x`.
- The **pryr** package provides many facilities for inspecting the memory footprint of your objects and code.

With a sparse representation, we not only saved on RAM, but also on the computing time of fitting a model. Here is the timing of a non sparse representation:

```
system.time(lm.1 <- lm(y ~ X.1))
```

```
##    user  system elapsed
##   3.048   0.124   3.172
```

Well actually, `lm` is a wrapper for the `lm.fit` function. If we override all the overhead of `lm`, and call `lm.fit` directly, we gain some time:

```
system.time(lm.1 <- lm.fit(y=y, x=X.1))
```

```
##    user  system elapsed
##   1.196   0.028   1.232
```

We now do the same with the sparse representation:

```
system.time(lm.2 <- MatrixModels:::lm.fit.sparse(X.2,y))
```

```
##    user  system elapsed
##   0.212   0.004   0.215
```

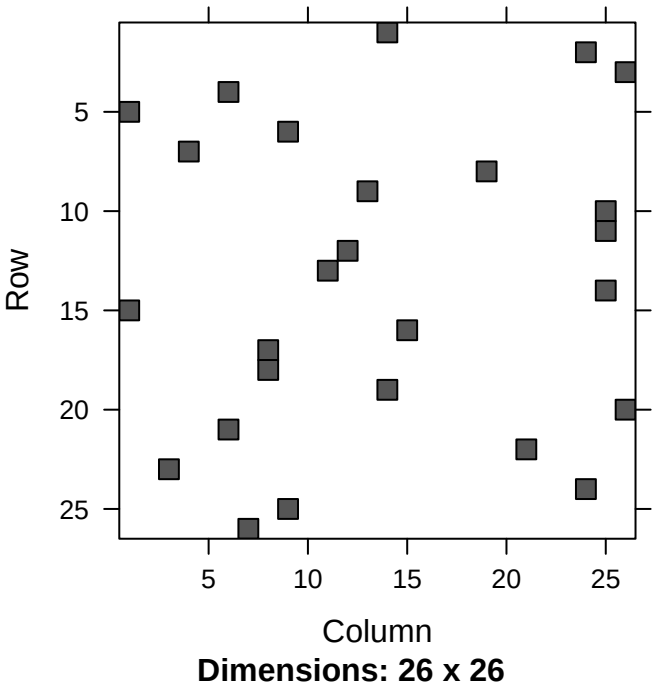
It is only left to verify that the returned coefficients are the same:

```
all.equal(lm.2, unname(lm.1$coefficients), tolerance = 1e-12)
```

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

You can also visualize the non zero entries, i.e., the sparsity structure.

```
image(X.2[1:26,1:26])
```



12.1 Sparse Matrix Representations

We first distinguish between the two main goals of the efficient representation: (i) efficient writing, i.e., modification; (ii) efficient reading, i.e., access. For our purposes, we will typically want efficient reading, since the `model.matrix` will not change while a model is being fitted.

Representations designed for writing include the *dictionary of keys*, *list of lists*, and a *coordinate list*. Representations designed for efficient reading include the *compressed sparse row* and *compressed sparse column*.

12.1.1 Coordinate List Representation

A *coordinate list representation*, also known as *COO*, or *triplet represantation* is simply a list of the non zero entries. Each element in the list is a triplet of the column, row, and value, of each non-zero entry in the matrix.

12.1.2 Compressed Column Oriented Representation

A *compressed column oriented representation*, also known as *compressed sparse column*, or *CSC*, where the **column** index is similar to COO, but instead of saving the row indexes, we save the locations in the column index vectors where the row index has to increase. The following figure may clarify this simple idea.

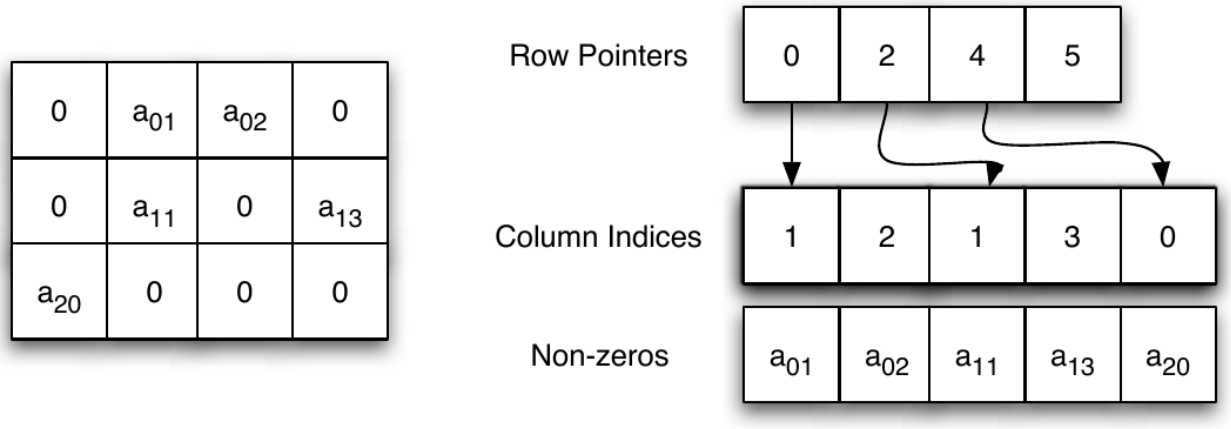


Figure 12.1: The CSC data structure. From Shah and Gilbert (2004). Remember that MATLAB is written in C, where the indexing starts at 0, and not 1.

The nature of statistical applications is such, that CSC representation is typically the most economical, justifying its popularity.

12.1.3 Compressed Row Oriented Representation

A *compressed row oriented representation*, also known as *compressed sparse row*, or *CSR*, is very similar to CSC, after switching the role of rows and columns. CSR is much less popular than CSC.

12.1.4 Sparse Algorithms

We will go into the details of some algorithms in the Numerical Linear Algebra Chapter **??**. For our current purposes two things need to be emphasized:

1. Working with sparse representations requires using a function that is aware of the representation you are using.
2. A mathematician may write $Ax = b \Rightarrow x = A^{-1}b$. This is a predicate of x,i.e., a property that x satisfies, which helps with its analysis. A computer, however, would **never** compute A^{-1} in order to find x, but rather use one of many endlessly many numerical algorithms. A computer will typically "search" various x's until it finds the one that fulfils the predicate.

12.2 Sparse Matrices and Sparse Models in R

12.2.1 The Matrix Package

The **Matrix** package provides facilities to deal with real (stored as double precision), logical and so-called "pattern" (binary) dense and sparse matrices. There are provisions to provide integer and complex (stored as double precision complex) matrices.

The sparse matrix classes include:

- `TsparseMatrix`: a virtual class of the various sparse matrices in triplet representation.
- `CsparseMatrix`: a virtual class of the various sparse matrices in CSC representation.
- `RsparseMatrix`: a virtual class of the various sparse matrices in CSR representation.

For matrices of real numbers, stored in *double precision*, the **Matrix** package provides the following (non virtual) classes:

- `dgTMatrix`: a **general** sparse matrix of **doubles**, in **triplet** representation.
- `dgCMatrix`: a **general** sparse matrix of **doubles**, in **CSC** representation.
- `dsCMatrix`: a **symmetric** sparse matrix of **doubles**, in **CSC** representation.
- `dtCMatrix`: a **triangular** sparse matrix of **doubles**, in **CSC** representation.

Why bother with distinguishing between the different shapes of the matrix? Because the more structure is assumed on a matrix, the more our (statistical) algorithms can be optimized. For our purposes `dgCMatrix` will be the most useful.

12.2.2 The MatrixModels Package

TODO

12.2.3 The glmnet Package

As previously stated, an efficient storage of the `model.matrix` is half of the story. We now need implementations of our favorite statistical algorithms that make use of this representation. At the time of writing, a very useful package that does that is the **glmnet** package, which allows to fit linear models, generalized linear models, with ridge, lasso, and elastic net regularization. The **glmnet** package allows all of this, using the sparse matrices of the **Matrix** package.

The following example is taken from John Myles White's blog, and compares the runtime of fitting an OLS model, using `glmnet` with both sparse and dense matrix representations.

```
suppressPackageStartupMessages(library('glmnet'))

set.seed(1)
performance <- data.frame()

for (sim in 1:10){
  n <- 10000
  p <- 500

  nzc <- trunc(p / 10)

  x <- matrix(rnorm(n * p), n, p) #make a dense matrix
  iz <- sample(1:(n * p),
               size = n * p * 0.85,
               replace = FALSE)
  x[iz] <- 0 # sparsify by injecting zeroes
  sx <- Matrix(x, sparse = TRUE) # save as a sparse object

  beta <- rnorm(nzc)
```

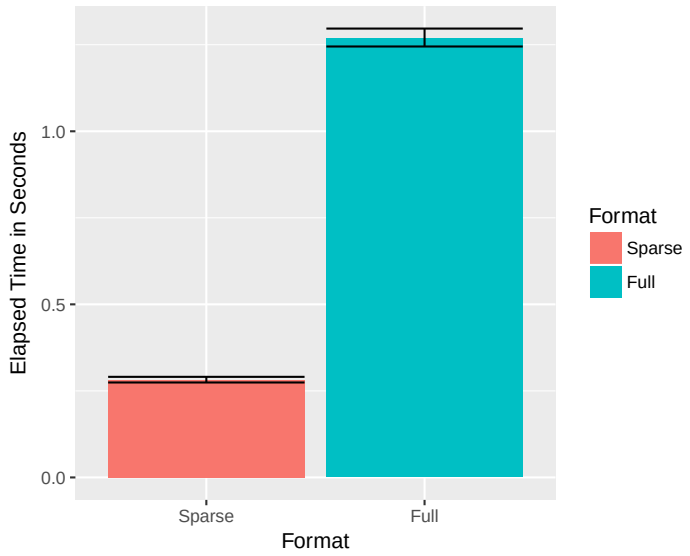
```
  fx <- x[, seq(nzc)] %*% beta

  eps <- rnorm(n)
  y <- fx + eps # make data

  # Now to the actual model fitting:
  sparse.times <- system.time(fit1 <- glmnet(sx, y)) # sparse glmnet
  full.times <- system.time(fit2 <- glmnet(x, y)) # dense glmnet

  sparse.size <- as.numeric(object.size(sx))
  full.size <- as.numeric(object.size(x))

  performance <- rbind(performance, data.frame(Format = 'Sparse',
                                               UserTime = sparse.times[1],
                                               SystemTime = sparse.times[2],
                                               ElapsedTime = sparse.times[3],
                                               Size = sparse.size))
  performance <- rbind(performance, data.frame(Format = 'Full',
                                               UserTime = full.times[1],
                                               SystemTime = full.times[2],
                                               ElapsedTime = full.times[3],
                                               Size = full.size))
}
```

Things to note:

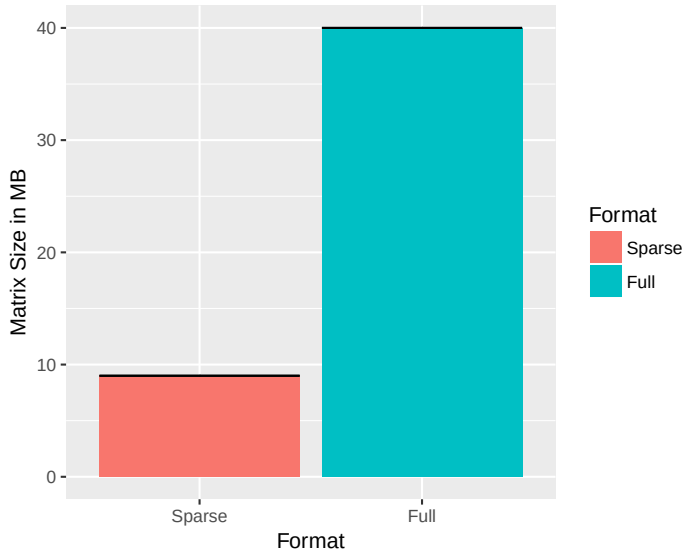
- The simulation calls `glmnet` twice. Once with the non-sparse object `x`, and once with its sparse version `sx`.
- The degree of sparsity of `sx` is 85%. We know this because we "injected" zeroes in 0.85 of the locations of `x`.
- Because `y` is continuous `glmnet` will fit a simple OLS model. We will see later how to use it to fit GLMs and use lasso, ridge, and elastic-net regularization.

We now inspect the computing time, and the memory footprint, only to discover that sparse representations make a BIG difference.

```
suppressPackageStartupMessages(library('ggplot2'))
ggplot(performance, aes(x = Format, y = ElapsedTime, fill = Format)) +
  stat_summary(fun.data = 'mean_cl_boot', geom = 'bar') +
  stat_summary(fun.data = 'mean_cl_boot', geom = 'errorbar') +
  ylab('Elapsed Time in Seconds')
```



```
ggplot(performance, aes(x = Format, y = Size / 1000000, fill = Format)) +
  stat_summary(fun.data = 'mean_cl_boot', geom = 'bar') +
  stat_summary(fun.data = 'mean_cl_boot', geom = 'errorbar') +
  ylab('Matrix Size in MB')
```



How do we perform other types of regression with the **glmnet**? We just need to use the `family` and `alpha` arguments of `glmnet::glmnet`. The `family` argument governs the type of GLM to fit: logistic, Poisson, probit, or other types of GLM. The `alpha` argument controls the type of regularization. Set to `alpha=0` for ridge, `alpha=1` for lasso, and any value in between for elastic-net regularization.

12.2.4 The SparseM Package

12.3 Bibliographic Notes

The best place to start reading on sparse representations and algorithms is the vignettes of the **Matrix** package. Gilbert et al. (1992) is also a great read for some general background. For the theory on solving sparse linear systems see Davis (2006). For general numerical linear algebra see Gentle (2012).

12.4 Practice Yourself

Chapter 13

Memory Efficiency

As put by Kane et al. (2013), it was quite puzzling when very few of the competitors, for the Million dollars prize in the Netflix challenge, were statisticians. This is perhaps because the statistical community historically uses SAS, SPSS, and R. The first two tools are very well equipped to deal with big data, but are very unfriendly when trying to implement a new method. R, on the other hand, is very friendly for innovation, but was not equipped to deal with the large data sets of the Netflix challenge. A lot has changed in R since 2006. This is the topic of this chapter.

As we have seen in the Sparsity Chapter 12, an efficient representation of your data in RAM will reduce computing time, and will allow you to fit models that would otherwise require tremendous amounts of RAM. Not all problems are sparse however. It is also possible that your data does not fit in RAM, even if sparse. There are several scenarios to consider:

1. Your data fits in RAM, but is too big to compute with.
2. Your data does not fit in RAM, but fits in your local storage (HD, SSD, etc.)
3. Your data does not fit in your local storage.

If your data fits in RAM, but is too large to compute with, a solution is to replace the algorithm you are using. Instead of computing with the whole data, your algorithm will compute with parts of the data, also called *chunks*, or *batches*. These algorithms are known as *external memory algorithms* (EMA).

If your data does not fit in RAM, but fits in your local storage, you have two options. The first is to save your data in a *database management system* (DBMS). This will allow you to use the algorithms provided by your DBMS, or let R use an EMA while "chunking" from your DBMS. Alternatively, and preferably, you may avoid using a DBMS, and work with the data directly form your local storage by saving your data in some efficient manner.

Finally, if your data does not fit on you local storage, you will need some external storage solution such as a distributed DBMS, or distributed file system.
Remark. If you use Linux, you may be better of than Windows users. Linux will allow you to compute with larger datasets using its *swap file* that extends RAM using your HD or SSD. On the other hand, relying on the swap file is a BAD practice since it is much slower than RAM, and you can typically do much better using the tricks of this chapter. Also, while I LOVE Linux, I would never dare to recommend switching to Linux just to deal with memory contraints.

13.1 Efficient Computing from RAM

If our data can fit in RAM, but is still too large to compute with it (recall that fitting a model requires roughly 5-10 times more memory than saving it), there are several facilities to be used. The first, is the sparse representation discussed in Chapter 12, which is relevant when you have factors, which will typically map to sparse model matrices. Another way is to use *external memory algorithms* (EMA).

The `biglm::biglm` function provides an EMA for linear regression. The following if taken from the function's example.

```
data(trees)
ff<-log(Volume)~log(Girth)+log(Height)
```

```
chunk1<-trees[1:10,]
chunk2<-trees[11:20,]
chunk3<-trees[21:31,]

library(biglm)
a <- biglm(ff,chunk1)
a <- update(a,chunk2)
a <- update(a,chunk3)

coef(a)
```

```
## (Intercept)  log(Girth) log(Height)
##   -6.631617    1.982650    1.117123
```

Things to note:

- The data has been chunked along rows.
- The initial fit is done with the `biglm` function.
- The model is updated with further chunks using the `update` function.

We now compare it to the in-memory version of `lm` to verify the results are the same.

```
b <- lm(ff, data=trees)
rbind(coef(a),coef(b))
```

```
##      (Intercept) log(Girth) log(Height)
## [1,]   -6.631617    1.98265    1.117123
## [2,]   -6.631617    1.98265    1.117123
```

Other packages that follow these lines, particularly with classification using SVMs, are **LiblineaR**, and **RSofia**.

13.1.1 Summary Statistics from RAM

If you are not going to do any model fitting, and all you want is efficient filtering, selection and summary statistics, then a lot of my warnings above are irrelevant. For these purposes, the facilities provided by **base**, **stats**, and **dplyr** are probably enough. If the data is large, however, these facilities may be too slow. If your data fits into RAM, but speed bothers you, take a look at the **data.table** package. The syntax is less friendly than **dplyr**, but **data.table** is BLAZING FAST compared to competitors. Here is a little benchmark[1].

First, we setup the data.

```
library(data.table)

n <- 1e6 # number of rows
k <- c(200,500) # number of distinct values for each 'group_by' variable
p <- 3 # number of variables to summarize

L1 <- sapply(k, function(x) as.character(sample(1:x, n, replace = TRUE) ))
L2 <- sapply(1:p, function(x) rnorm(n) )

tbl <- data.table(L1,L2) %>%
  setnames(c(paste("v",1:length(k),sep=""), paste("x",1:p,sep="") ))

tbl_dt <- tbl
tbl_df <- tbl %>% as.data.frame
```

We compare the aggregation speeds. Here is the timing for **dplyr**.

[1] The code was contributed by Liad Shekel.

```
system.time( tbl_df %>%
               group_by(v1,v2) %>%
               summarize(
                 x1 = sum(abs(x1)),
                 x2 = sum(abs(x2)),
                 x3 = sum(abs(x3))
                 )
             )
```

```
##    user  system elapsed
##   9.000   0.260   9.265
```

And now the timing for **data.table**.

```
system.time(
  tbl_dt[ ,  .( x1 = sum(abs(x1)), x2 = sum(abs(x2)), x3 = sum(abs(x3)) ), .(v1,v2)]
  )
```

```
##    user  system elapsed
##   0.872   0.052   0.923
```

The winner is obvious. Let's compare filtering (i.e. row subsets, i.e. SQL's SELECT).

```
system.time(
  tbl_df %>% filter(v1 == "1")
  )
```

```
##    user  system elapsed
##   0.480   0.044   0.528
```

```
system.time(
  tbl_dt[v1 == "1"]
  )
```

```
##    user  system elapsed
##   0.016   0.004   0.017
```

13.2 Computing from a Database

The early solutions to oversized data relied on storing your data in some DBMS such as *MySQL*, *PostgresSQL*, *SQLite*, *H2*, *Oracle*, etc. Several R packages provide interfaces to these DBMSs, such as **sqldf**, **RDBI**, **RSQite**. Some will even include the DBMS as part of the package itself.

Storing your data in a DBMS has the advantage that you can typically rely on DBMS providers to include very efficient algorithms for the queries they support. On the downside, SQL queries may include a lot of summary statistics, but will rarely include model fitting[2]. This means that even for simple things like linear models, you will have to revert to R's facilities– typically some sort of EMA with chunking from the DBMS. For this reason, and others, we prefer to compute from efficient file structures, as described in Section 13.3.

If, however, you have a powerful DBMS around, or you only need summary statistics, or you are an SQL master, keep reading.

The package **RSQLite** includes an SQLite server, which we now setup for demonstration. The package **dplyr**, discussed in the Hadleyverse Chapter **??**, will take care of translating the **dplyr** syntax, to the SQL syntax of the DBMS. The following example is taken from the **dplyr** Databases vignette.

```
library(RSQLite)
library(dplyr)

file.remove('my_db.sqlite3')
```

[2]This is slowly changing. Indeed, Microsoft's SQL Server 2016 is already providing in-database-analytics, and other will surely follow.

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

```
my_db <- src_sqlite(path = "my_db.sqlite3", create = TRUE)

library(nycflights13)
flights_sqlite <- copy_to(
  dest= my_db,
  df= flights,
  temporary = FALSE,
  indexes = list(c("year", "month", "day"), "carrier", "tailnum"))
```

Things to note:

- `src_sqlite` to start an empty table, managed by SQLite, at the desired path.
- `copy_to` copies data from R to the database.
- Typically, setting up a DBMS like this makes no sense, since it requires loading the data into RAM, which is precisely what we want to avoid.

We can now start querying the DBMS.

```
select(flights_sqlite, year:day, dep_delay, arr_delay)
```

```
## # Source:   lazy query [?? x 5]
## # Database: sqlite 3.19.3 [my_db.sqlite3]
##     year month   day dep_delay arr_delay
##    <int> <int> <int>     <dbl>     <dbl>
##  1  2013     1     1         2        11
##  2  2013     1     1         4        20
##  3  2013     1     1         2        33
##  4  2013     1     1        -1       -18
##  5  2013     1     1        -6       -25
##  6  2013     1     1        -4        12
##  7  2013     1     1        -5        19
##  8  2013     1     1        -3       -14
##  9  2013     1     1        -3        -8
## 10  2013     1     1        -2         8
## # ... with more rows
```

```
filter(flights_sqlite, dep_delay > 240)
```

```
## # Source:   lazy query [?? x 19]
## # Database: sqlite 3.19.3 [my_db.sqlite3]
##     year month   day dep_time sched_dep_time dep_delay arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>
##  1  2013     1     1      848           1835       853     1001
##  2  2013     1     1     1815           1325       290     2120
##  3  2013     1     1     1842           1422       260     1958
##  4  2013     1     1     2115           1700       255     2330
##  5  2013     1     1     2205           1720       285       46
##  6  2013     1     1     2343           1724       379      314
##  7  2013     1     2     1332            904       268     1616
##  8  2013     1     2     1412            838       334     1710
##  9  2013     1     2     1607           1030       337     2003
## 10  2013     1     2     2131           1512       379     2340
## # ... with more rows, and 12 more variables: sched_arr_time <int>,
## #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dbl>
```

```
summarise(flights_sqlite, delay = mean(dep_time))
```

```
## # Source:   lazy query [?? x 1]
## # Database: sqlite 3.19.3 [my_db.sqlite3]
##     delay
##     <dbl>
## 1 1349.11
```

13.3 Computing From Efficient File Structrures

It is possible to save your data on your storage device, without the DBMS layer to manage it. This has several advantages:

- You don't need to manage a DBMS.
- You don't have the computational overhead of the DBMS.
- You may optimize the file structure for statistical modelling, and not for join and summary operations, as in relational DBMSs.

There are several facilities that allow you to save and compute directly from your storage:

1. **Memory Mapping**: Where RAM addresses are mapped to a file on your storage. This extends the RAM to the capacity of your storage (HD, SSD,...). Performance slightly deteriorates, but the access is typically very fast. This approach is implemented in the **bigmemory** package.
2. **Efficient Binaries**: Where the data is stored as a file on the storage device. The file is binary, with a well designed structure, so that chunking is easy. This approach is implemented in the **ff** package, and the commercial **RevoScaleR** package.

Your algorithms need to be aware of the facility you are using. For this reason each facility (**bigmemory**, **ff**, **RevoScaleR**,...) has an eco-system of packages that implement various statistical methods using that facility. As a general rule, you can see which package builds on a package using the *Reverse Depends* entry in the package description. For the **bigmemory** package, for instance, we can see that the packages **bigalgebra**, **biganalytics**, **bigFastlm**, **biglasso**, **bigpca**, **bigtabulate**, **GHap**, and **oem**, build upon it. We can expect this list to expand.

Here is a benchmark result, from Wang et al. (2015). It can be seen that **ff** and **bigmemory** have similar performance, while **RevoScaleR** (RRE in the figure) outperforms them. This has to do both with the efficiency of the binary representation, but also because **RevoScaleR** is inherently parallel. More on this in the Parallelization Chapter **??**.

	Reading	Transforming	Fitting
bigmemory	968.6	105.5	1501.7
ff	1111.3	528.4	1988.0
RRE	851.7	107.5	189.4

13.3.1 bigmemory

We now demonstrate the workflow of the **bigmemory** package. We will see that **bigmemory**, with it's `big.matrix` object is a very powerful mechanism. If you deal with big numeric matrices, you will find it very useful. If you deal with big data frames, or any other non-numeric matrix, **bigmemory** may not be the appropriate tool, and you should try **ff**, or the commercial **RevoScaleR**.

```
# download.file("http://www.cms.gov/Research-Statistics-Data-and-Systems/Statistics-Trends-and-Reports/BSAP
# unzip(zipfile="2010_Carrier_PUF.zip")

library("bigmemory")
x <- read.big.matrix("data/2010_BSA_Carrier_PUF.csv", header = TRUE,
```

```
                    backingfile = "airline.bin",
                    descriptorfile = "airline.desc",
                    type = "integer")
dim(x)
```

```
## [1] 2801660      11
```

```
pryr::object_size(x)
```

```
## 616 B
```

```
class(x)
```

```
## [1] "big.matrix"
## attr(,"package")
## [1] "bigmemory"
```

Things to note:

- The basic building block of the **bigmemory** ecosystem, is the `big.matrix` class, we constructed with `read.big.matrix`.
- `read.big.matrix` handles the import to R, and the saving to a memory mapped file. The implementation is such that at no point does R hold the data in RAM.
- The memory mapped file will be there after the session is over. It can thus be called by other R sessions using `attach.big.matrix("airline.desc")`. This will be useful when parallelizing.
- `pryr::object_size` return the size of the object. Since `x` holds only the memory mappings, it is much smaller than the 100MB of data that it holds.

We can now start computing with the data. Many statistical procedures for the `big.matrix` object are provided by the **biganalytics** package. In particular, the `biglm.big.matrix` and `bigglm.big.matrix` functions, provide an interface from `big.matrix` objects, to the EMA linear models in `biglm::biglm` and `biglm::bigglm`.

```
library(biganalytics)
biglm.2 <- bigglm.big.matrix(BENE_SEX_IDENT_CD~CAR_LINE_HCPCS_CD, data=x)
coef(biglm.2)
```

```
##       (Intercept) CAR_LINE_HCPCS_CD
##      1.537848e+00      1.210282e-07
```

Other notable packages that operate with `big.matrix` objects include:

- **bigtabulate**: Extend the bigmemory package with 'table', 'tapply', and 'split' support for 'big.matrix' objects.
- **bigalgebra**: For matrix operation.
- **bigpca**: principle components analysis (PCA), or singular value decomposition (SVD).
- **bigFastlm**: for (fast) linear models.
- **biglasso**: extends lasso and elastic nets.
- **GHap**: Haplotype calling from phased SNP data.

13.4 ff

The **ff** packages replaces R's in-RAM storage mechanism with on-disk (efficient) storage. Unlike **bigmemory**, **ff** supports all of R vector types such as factors, and not only numeric. Unlike `big.matrix`, which deals with (numeric) matrices, the `ffdf` class can deal with data frames.

Here is an example. First open a connection to the file, without actually importing it using the `LaF::laf_open_csv` function.

```
.dat <- LaF::laf_open_csv(filename = "data/2010_BSA_Carrier_PUF.csv",
                    column_types = c("integer", "integer", "categorical", "categorical", "categorical", "int
                    column_names = c("sex", "age", "diagnose", "healthcare.procedure", "typeofservice", "ser
                    skip = 1)
```

Now write the data to local storage as an ff data frame, using `laf_to_ffdf`.

```
data.ffdf <- ffbase::laf_to_ffdf(laf = .dat)
head(data.ffdf)
```

```
## ffdf (all open) dim=c(2801660,6), dimorder=c(1,2) row.names=NULL
## ffdf virtual mapping
##                              PhysicalName VirtualVmode PhysicalVmode  AsIs
## sex                                   sex      integer       integer FALSE
## age                                   age      integer       integer FALSE
## diagnose                        diagnose      integer       integer FALSE
## healthcare.procedure healthcare.procedure      integer       integer FALSE
## typeofservice              typeofservice      integer       integer FALSE
## service.count              service.count      integer       integer FALSE
##                      VirtualIsMatrix PhysicalIsMatrix PhysicalElementNo
## sex                            FALSE            FALSE                 1
## age                            FALSE            FALSE                 2
## diagnose                       FALSE            FALSE                 3
## healthcare.procedure           FALSE            FALSE                 4
## typeofservice                  FALSE            FALSE                 5
## service.count                  FALSE            FALSE                 6
##                      PhysicalFirstCol PhysicalLastCol PhysicalIsOpen
## sex                                 1               1           TRUE
## age                                 1               1           TRUE
## diagnose                            1               1           TRUE
## healthcare.procedure                1               1           TRUE
## typeofservice                       1               1           TRUE
## service.count                       1               1           TRUE
## ffdf data
##           sex   age diagnose healthcare.procedure typeofservice
## 1       1     1        NA                  99213         M1B
## 2       1     1        NA                  A0425         O1A
## 3       1     1        NA                  A0425         O1A
## 4       1     1        NA                  A0425         O1A
## 5       1     1        NA                  A0425         O1A
## 6       1     1        NA                  A0425         O1A
## 7       1     1        NA                  A0425         O1A
## 8       1     1        NA                  A0425         O1A
## :           :     :        :                   :             :
## 2801653 2     6        V82                 85025         T1D
## 2801654 2     6        V82                 87186         T1H
## 2801655 2     6        V82                 99213         M1B
## 2801656 2     6        V82                 99213         M1B
## 2801657 2     6        V82                 A0429         O1A
## 2801658 2     6        V82                 G0328         T1H
## 2801659 2     6        V86                 80053         T1B
## 2801660 2     6        V88                 76856         I3B
##         service.count
## 1                   1
## 2                   1
## 3                   1
## 4                   2
## 5                   2
## 6                   3
## 7                   3
## 8                   4
## :                   :
## 2801653             1
```

```
## 2801654         1
## 2801655         1
## 2801656         1
## 2801657         1
## 2801658         1
## 2801659         1
## 2801660         1
```

We can verify that the `ffdf` data frame has a small RAM footprint.

```
pryr::object_size(data.ffdf)
```

```
## 343 kB
```

The **ffbase** package provides several statistical tools to compute with `ff` class objects. Here is simple table.

```
ffbase:::table.ff(data.ffdf$age)
```

```
##
##      1      2      3      4      5      6
## 517717 495315 492851 457643 419429 418705
```

The EMA implementation of `biglm::biglm` and `biglm::bigglm` have their **ff** versions.

```
library(biglm)
mymodel.ffdf <- biglm(payment ~ factor(sex) + factor(age) + place.served,
                      data = data.ffdf)
summary(mymodel.ffdf)
```

```
## Large data regression model: biglm(payment ~ factor(sex) + factor(age) + place.served, data = data.ffdf)
## Sample size =  2801660
##                  Coef    (95%     CI)     SE      p
## (Intercept)  97.3313 96.6412 98.0214 0.3450 0.0000
## factor(sex)2 -4.2272 -4.7169 -3.7375 0.2449 0.0000
## factor(age)2  3.8067  2.9966  4.6168 0.4050 0.0000
## factor(age)3  4.5958  3.7847  5.4070 0.4056 0.0000
## factor(age)4  3.8517  3.0248  4.6787 0.4135 0.0000
## factor(age)5  1.0498  0.2030  1.8965 0.4234 0.0132
## factor(age)6 -4.8313 -5.6788 -3.9837 0.4238 0.0000
## place.served -0.6132 -0.6253 -0.6012 0.0060 0.0000
```

Things to note:

- `biglm::biglm` notices the input of of class `ffdf` and calls the appropriate implementation.
- The model formula, `payment ~ factor(sex) + factor(age) + place.served`, includes factors which cause no difficulty.
- You cannot inspect the factor coding (dummy? effect?) using `model.matrix`. This is because EMAs never really construct the whole matrix, let alone, save it in memory.

13.5 matter

Memory-efficient reading, writing, and manipulation of structured binary data on disk as vectors, matrices, arrays, lists, and data frames.

TODO

13.6 iotools

A low level facility for connecting to on-disk binary storage. Unlike **ff**, and **bigmemory**, it behaves like native R objects, with their copy-on-write policy. Unlike **reader**, it allows chunking. Unlike `read.csv`, it allows fast I/O.

iotools is thus a potentially very powerfull facility. See Arnold et al. (2015) for details.

TODO

13.7 HDF5

Like **ff**, HDF5 is an on-disk efficient file format. The package **h5** is interface to the "HDF5" library supporting fast storage and retrieval of R-objects like vectors, matrices and arrays.

TODO

13.8 DelayedArray

Delayed operations on array-like objects

TODO

13.8.1 DelayedMatrixStats

Functions that Apply to Rows and Columns of **DelayedMatrix** Objects.

13.8.2 beachmat

Provides a consistent C++ class interface for a variety of commonly used matrix types, including sparse and HDF5-backed matrices.

TODO

13.8.3 restfulSE

Functions and classes to interface with remote data stores.

TODO

13.9 Computing from a Distributed File System

If your data is SOOO big that it cannot fit on your local storage, you will need a distributed file system or DBMS. We do not cover this topic here, and refer the reader to the **RHipe**, **RHadoop**, and **RSpark** packages and references therein.

13.10 Bibliographic Notes

An absolute SUPERB review on computing with big data is Wang et al. (2015), and references therein (Kane et al. (2013) in particular). For an up-to-date list of the packages that deal with memory constraints, see the **Large memory and out-of-memory data** section in the High Performance Computing R task view.

13.11 Practice Yourself

Chapter 14

Causal Inferense

Recall this fun advertisement

```
knitr::include_graphics(rep("art/wor.jpg", 3))
```

How come everyone in the past did not know what every kid knows these days: that cigarettes are bad for you. The reason is the difficulty in causal inference. Scientists knew about the correlations between smoking and disease, but

no one could prove one caused the other. These could have been nothing more than correlations, with some external cause.

Cigarettes were declared dangerous without any direct causal evidence. It was in the USA's surgeon general report of 1964 that it was decided that despite of the impossibility of showing a direct causal relation, the circumstantial evidence is just too strong, and declared cigarettes as dangerous.

14.1 Causal Inference From Designed Experiments

14.1.1 Design of Experiments

https://cran.r-project.org/web/views/ExperimentalDesign.html

TODO

14.1.2 Randomized Inference

https://dimewiki.worldbank.org/wiki/Randomization_Inference

TODO

14.2 Causal Inference from Observational Data

14.2.1 Instrumental Variables

TODO

14.2.2 Propensity Scores

TODO

14.2.3 Direct Lieklihood

TODO

14.2.4 Regression Discontinuity

14.3 Bibliographic Notes

14.4 Practice Yourself

Bibliography

Allard, D. (2013). J.-p. chilès, p. delfiner: Geostatistics: Modeling spatial uncertainty.

Anderson-Cook, C. M. (2004). An introduction to multivariate statistical analysis. *Journal of the American Statistical Association*, 99(467):907–909.

Arnold, T., Kane, M., and Urbanek, S. (2015). iotools: High-performance i/o tools for r. *arXiv preprint arXiv:1510.00041*.

Bai, Z. and Saranadasa, H. (1996). Effect of high dimension: by an example of a two sample problem. *Statistica Sinica*, pages 311–329.

Bates, D., Mächler, M., Bolker, B., and Walker, S. (2015). Fitting linear mixed-effects models using lme4. *Journal of Statistical Software*, 67(1):1–48.

Benjamini, Y. and Yekutieli, D. (2001). The control of the false discovery rate in multiple testing under dependency. *Annals of statistics*, pages 1165–1188.

Chang, W., Cheng, J., Allaire, J., Xie, Y., and McPherson, J. (2017). *shiny: Web Application Framework for R*. R package version 1.0.0.

Christakos, G. (2000). *Modern spatiotemporal geostatistics*, volume 6. Oxford University Press.

Cressie, N. (2015). *Statistics for spatial data*. John Wiley & Sons.

Davis, T. A. (2006). *Direct methods for sparse linear systems*. SIAM.

Diggle, P. J., Tawn, J., and Moyeed, R. (1998). Model-based geostatistics. *Journal of the Royal Statistical Society: Series C (Applied Statistics)*, 47(3):299–350.

Efron, B. (2012). *Large-scale inference: empirical Bayes methods for estimation, testing, and prediction*, volume 1. Cambridge University Press.

Everitt, B. and Hothorn, T. (2011). *An introduction to applied multivariate analysis with R*. Springer Science & Business Media.

Fithian, W. (2015). *Topics in Adaptive Inference*. PhD thesis, STANFORD UNIVERSITY.

Friedman, J., Hastie, T., and Tibshirani, R. (2001). *The elements of statistical learning*, volume 1. Springer series in statistics Springer, Berlin.

Gentle, J. E. (2012). *Numerical linear algebra for applications in statistics*. Springer Science & Business Media.

Gilbert, J. R., Moler, C., and Schreiber, R. (1992). Sparse matrices in matlab: Design and implementation. *SIAM Journal on Matrix Analysis and Applications*, 13(1):333–356.

Greene, W. H. (2003). *Econometric analysis*. Pearson Education India.

Javanmard, A. and Montanari, A. (2014). Confidence intervals and hypothesis testing for high-dimensional regression. *Journal of Machine Learning Research*, 15(1):2869–2909.

Kalisch, M. and Bühlmann, P. (2014). Causal structure learning and inference: a selective review. *Quality Technology & Quantitative Management*, 11(1):3–21.

Kane, M. J., Emerson, J., Weston, S., et al. (2013). Scalable strategies for computing with massive data. *Journal of Statistical Software*, 55(14):1–19.

Leisch, F. (2002). Sweave: Dynamic generation of statistical reports using literate data analysis. In *Compstat*, pages 575–580. Springer.

McCullagh, P. (1984). Generalized linear models. *European Journal of Operational Research*, 16(3):285–292.

Nadler, B. (2008). Finite sample approximation results for principal component analysis: A matrix perturbation approach. *The Annals of Statistics*, pages 2791–2817.

Pinero, J. and Bates, D. (2000). Mixed-effects models in s and s-plus (statistics and computing).

R Core Team (2016). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria.

Rabinowicz, A. and Rosset, S. (2018). Assessing prediction error at interpolation and extrapolation points. *arXiv preprint arXiv:1802.00996*.

Robinson, G. K. (1991). That blup is a good thing: the estimation of random effects. *Statistical science*, pages 15–32.

Rosenblatt, J. (2013). A practitioner's guide to multiple testing error rates. *arXiv preprint arXiv:1304.4920*.

Rosenblatt, J., Gilron, R., and Mukamel, R. (2016). Better-than-chance classification for signal detection. *arXiv preprint arXiv:1608.08873*.

Rosenblatt, J. D. and Benjamini, Y. (2014). Selective correlations; not voodoo. *NeuroImage*, 103:401–410.

Rosset, S. and Tibshirani, R. J. (2018). From fixed-x to random-x regression: Bias-variance decompositions, covariance penalties, and prediction error estimation. *Journal of the American Statistical Association*, (just-accepted).

Sarkar, D. (2008). *Lattice: Multivariate Data Visualization with R*. Springer, New York. ISBN 978-0-387-75968-5.

Searle, S. R., Casella, G., and McCulloch, C. E. (2009). *Variance components*, volume 391. John Wiley & Sons.

Shah, V. and Gilbert, J. R. (2004). Sparse matrices in matlab* p: Design and implementation. In *International Conference on High-Performance Computing*, pages 144–155. Springer.

Simes, R. J. (1986). An improved bonferroni procedure for multiple tests of significance. *Biometrika*, 73(3):751–754.

Tukey, J. W. (1977). *Exploratory data analysis*. Reading, Mass.

Venables, W. N. and Ripley, B. D. (2013). *Modern applied statistics with S-PLUS*. Springer Science & Business Media.

Venables, W. N., Smith, D. M., Team, R. D. C., et al. (2004). An introduction to r.

Wang, C., Chen, M.-H., Schifano, E., Wu, J., and Yan, J. (2015). Statistical methods and computing for big data. *arXiv preprint arXiv:1502.07989*.

Weiss, R. E. (2005). *Modeling longitudinal data*. Springer Science & Business Media.

Wickham, H. (2009). *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York.

Wickham, H. (2014). *Advanced R*. CRC Press.

Wilcox, R. R. (2011). *Introduction to robust estimation and hypothesis testing*. Academic Press.

Wilkinson, G. and Rogers, C. (1973). Symbolic description of factorial models for analysis of variance. *Applied Statistics*, pages 392–399.

Wilkinson, L. (2006). *The grammar of graphics*. Springer Science & Business Media.

Xie, Y. (2015). *Dynamic Documents with R and knitr*, volume 29. CRC Press.

Xie, Y. (2016). *bookdown: Authoring Books and Technical Documents with R Markdown*. CRC Press.