R (BGU course)

Jonathan D. Rosenblatt

2017-02-23

Contents

Chapter 1

Preface

This book accompanies BGU's "R" course, at the department of Industrial Engineering and Management.

It has several purposes:

- Help me organize and document the course material.
- Help students during class so that they may focus on listening and not writing.
- Help students after class, so that they may self-study.

At its current state it is experimental. It can thus be expected to change from time to time, and include mistakes. I will be enormously grateful to whoever decides to share with me any mistakes found.

I am enormously grateful to Yihui Xie, who's *bookdown* R package made it possibly to easily write a book which has many mathematical formulae, and R output.

I hope the reader will find this text interesting and useful.

1.1 Acknoledgements

I have consulted many people during the writing of this text. I would like to thank Efrat Vilensky and Liad Shekel in particular for their valuable inputs.

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.
- TakViews: part of CRAN that collects packages per topic.
- Bioconductor: A CRAN-like repository dedicated to the life sciences.
- 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

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 to run lines from editor. - alt+shift+k for RStudio keyboard shortcuts. - Ctrl+alt+j to navigate between 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.

3.1 Simple calculator

R can be used as a simple calculator.

```
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
```

3.2 Probability calculator

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

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 5 7 7 5 4 4 6 6 4
```

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

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

3.3 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()
```

3.4 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: `function(argument=value)`, and understand things like `<<-` 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] 7 4 7 7 4 5 3 7 6 5
```

which is an implicit call to

```
print(x)
```

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

Alternatively, you can assign and print simultaneously

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

```
##  [1] 5 5 4 6 8 4 4 3 6 5
```

Operate on the object

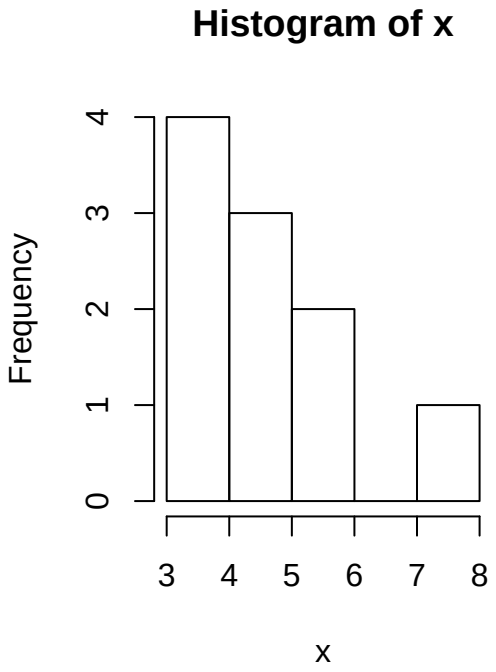
```
mean(x)  # compute mean
```

```
## [1] 5
```

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

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

```
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"
```

[1] S and S-Plus used to save objects on disk. Working from RAM has advantages and disadvantages. More on this in Chapter 15.

or with Ctrl+8 in RStudio.

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

```
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 15.

3.5 Piping

Because R originates in Unix and Linux environments, it inherits much of its flavor. Piping is an idea take 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)
x <- rbinom(n=1000, size=10, prob=0.5)
```

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

3.6 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:

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

```
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
```

Lets assign it to the object named "x":

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

In the case you made a computation you do not want to repeat, you can assign AFTER the computation is finished, since everything is saved by the '.Last.value' variable.

```
c(1,2,3)
y<- .Last.value
y
```

```
## [1] 1 2 3
## [1] "/home/johnros/workspace/Rcourse/docs/index.html"
```

Remark. In line with the linux look and feel, variables starting with a dot (.) are saved but are hidden. To show them see `?ls`.

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
```

3.7 Search paths and packages

R can be easily extended with packages, which are merely a set of functions and other objects, which can be loaded or unloaded at will. Let's look at the function **sum**. We can see its contents by calling it without arguments:

```
print(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: 0x4cec168>
## <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 the packages that are currently loaded:

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

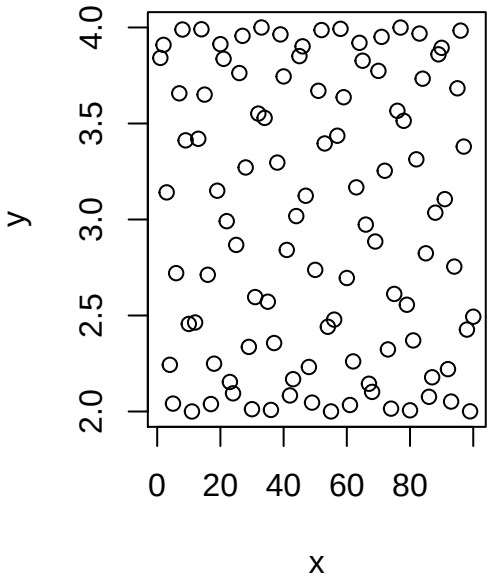
```
## [1] ".GlobalEnv"   "Penicillin"   "cases"        "Penicillin"
## [5] "cases"        "package:grid"
```

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.

3.8 Simple plotting

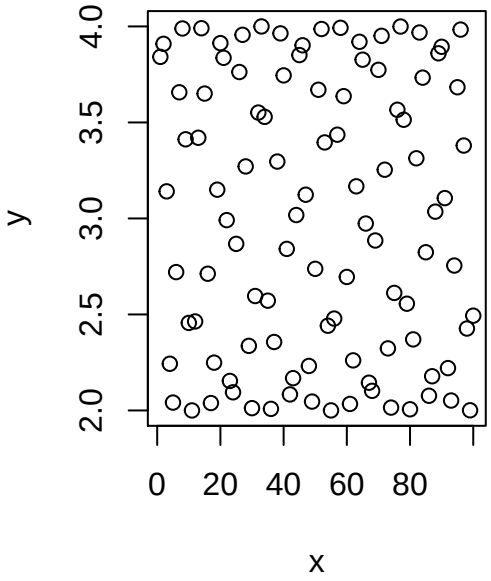
R has many plotting facilities. We start with the simplest facilities, namely, the `plot` function from the **graphics** package, which is loaded by default.

```
x<- 1:100; y<- 3+sin(x) # Create arbitrary data
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 it the "tilde" syntax, which originates in works of Wilkinson and Rogers (1973).

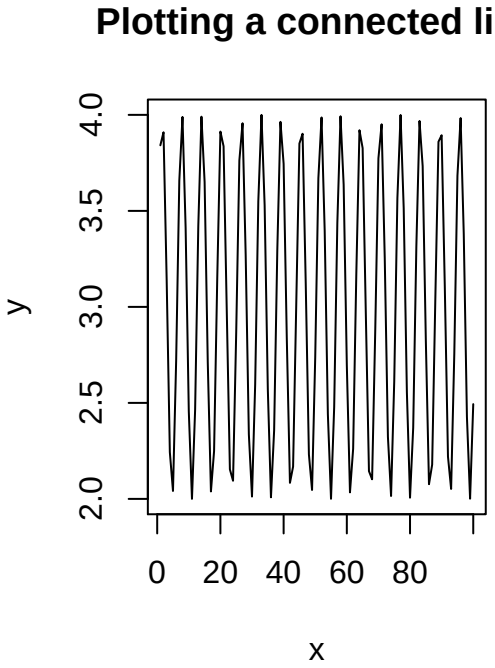
```
plot(y ~ x) # y~x syntax
```



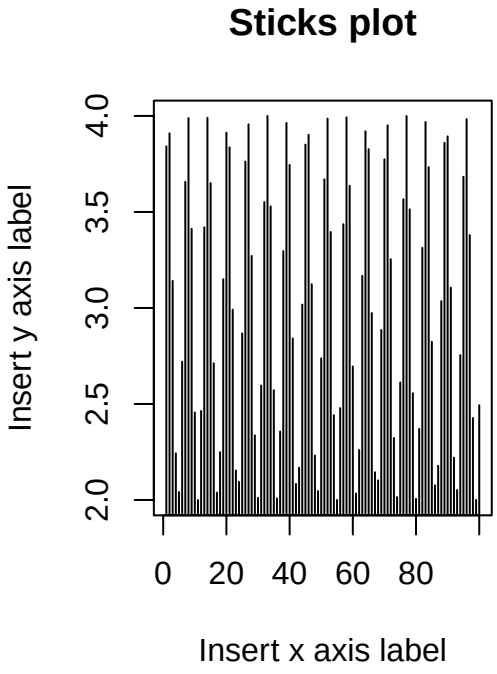
The syntax `y~x` is read as "y is a function of x". We will prefer the `y~x` syntax over the `x,y` syntax since it is easier to read, and will be very useful when we discuss more complicated models.

Here are some arguments that control the plot's appearance:

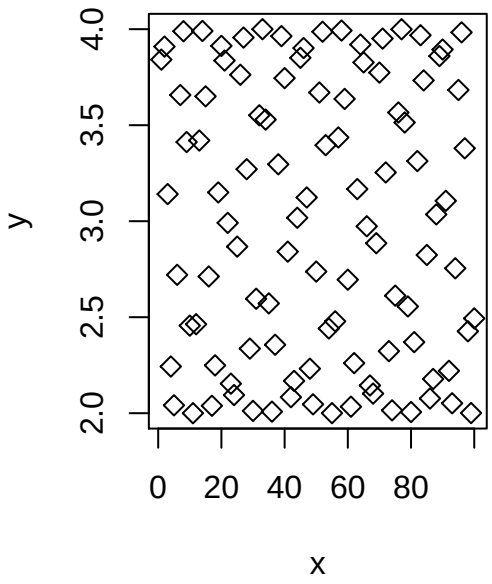
```
plot(y~x, type='l', main='Plotting a connected line') # main title
```



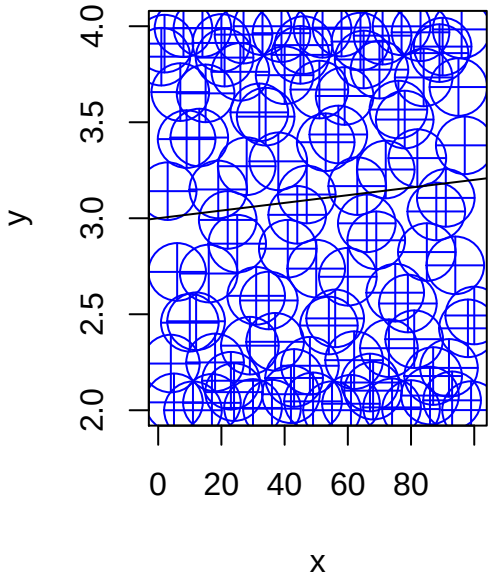

```
plot(y~x, type='h', main='Sticks plot', xlab='Insert x axis label', ylab='Insert y axis label') # axes labe
```




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




```
plot(y~x, pch=10, type='p', col='blue', cex=4) # More point parameters
abline(3, 0.002) # add linear line with slope b and intercept a
```



For more plotting options run these

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

When your plotting gets serious, go to Chapter 11.

3.9 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.
- **numeric** Where each element is a "real" number in double precision floating point.
- **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.
- **factor** Factors are not actually vector objects, but they feel like such. They actually used to encode a finite set of values. This will be very useful when fitting linear model, but may be confusing if you think you are dealing with a character vector when in fact you are dealing with a factor. Be alert!

Vectors can be combined into larger objects. A `matrix` can be thought of as the binding of several vectors of the same type. 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.

3.10 Data Frames

Creating a simple data frame:

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

Lets 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.

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
```

A frame is a vector of column vectors, so its length is merely the number of columns.

```
length(frame1)
```

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

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

3.11 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 all 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, ]
```

```
##   x      sin
## 1 1 3.841471
```

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
```

What did we just extract?

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

```
## NULL
```

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

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

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

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

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

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

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

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

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

```
## NULL
```

The `subset()` function does the same

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

If you are unsatisfied with the output of the `[` mechanism, you can use `[[`, which gets the content of a vector, while stripping the attributes.

```
a <- frame1[1] # [ extraction
b <- frame1[[1]] # [[ extraction
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
```

```
class(a)==class(b)
```

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

The different types of output causes different behaviors

```
a[1]
```

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

```
b[1]
```

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

If you want to learn more about subsetting see Hadley's guide, and our Chapter 13

3.12 Data Import and Export

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

3.12.1 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. The header row was not recognized. Fix with `header=TRUE`:

```
tirgul1 <- read.table(URL, header = TRUE)
head(tirgul1)
```

3.12.2 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 "/".

3.12.3 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.
- The output of `read.table` needs to be explicitly assigned to an object for it to be saved.

3.12.4 Writing Data to Text Files

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

3.12.5 .XLS(X) files

Strongly recommended to convert to .csv in Excel, and then import as csv. If you still insist see here.

3.12.6 Massive files

The above importing and exporting mechanism were not designed for massive files. See the section on Sparse Representation (14) and Out-of-Ram Algorithms (15) for more on working with massive data files.

3.12.7 Databases

R can does not need to read from text files; it can read directly from a data base. This is very useful since it allows the filtering, selecting and joining operations to rely on the database's optimized algorithms. See here.

3.13 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 a an object accessible using its name. We first define a simple function that sums its two arguments

```
my.sum <- function(x,y) {
  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.
- The arguments of the `function`, i.e. `(x,y)`, need to be named but we are not required to specify their type.
- A typical R function does not change objects[4] but rather creates new ones. To save the output pf `my.sum` we will need to assign it using the `<-` operator.
- The function will output its last evaluated expression.

3.14 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
```

[4]This is a classical *functional programming* paradigm. If you are used to *object oriented* programming, you may want to read about references classes which may be required if you are planning to compute with very complicated objects.

3.15 Recursion

The R compiler is really not designed for recursion, and you will rarely need to do so.
See the RCpp Chapter 19 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, as this Fibonacci series example demonstrates.

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

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

3.16 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), or anything else Bill Venables writes. For advanced R programming see Wickham (2014), or anything else Hadley Wickham writes.

Chapter 4

Exploratory Data Analysis

Exploratory Data Analysis (EDA) is a term cast by John W. Tukey in his seminal book Tukey (1977). It is the practice of inspecting, 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 a summary statistic or visually.
- How many variable analyzed simultaneously: univariate, bivariate, or multivariate.
- What type of variable: categorical or continuous.

4.1 Summary Statistics

4.1.1 Categorical Data

Categorical variable 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.

4.1.1.1 Summary of Univariate Categorical 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)

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
```

```
##    11    11
```

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
```

4.1.1.2 Summary of Bivariate Categorical Data

```
library(magrittr)
cbind(gender, drink) %>% head # inspect the raw data
```

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

```
table1 <- table(gender, drink)
table1
```

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

4.1.1.3 Summary of Multivariate Categorical Data

You may be wondering how does R handle tables with more than two dimensions. It is indeed not trivial, and R offers several solutions.

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

```
## , , age = Old
##
##       drink
## gender Coffee Coke Sprite Tea Water
##   Boy       0    2      2   0     0
##   Girl      4    0      0   2     1
##
## , , age = Young
##
##       drink
## gender Coffee Coke Sprite Tea Water
##   Boy       2    3      1   0     0
##   Girl      0    0      0   5     0
```

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

```
##               age Old Young
## gender drink
## Boy    Coffee       0     2
##        Coke         2     3
```

```
##        Sprite       2     1
##        Tea          0     0
##        Water        0     0
## Girl   Coffee       4     0
##        Coke         0     0
##        Sprite       0     0
##        Tea          2     5
##        Water        1     0
```

If you want proportions instead of counts, you need to specify the denominator, i.e., the margins.

```
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
```

4.1.2 Continous Data

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

4.1.2.1 Summary of Univariate Continous Data

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

4.1.2.2 Summary of Location

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

Definition 4.1. The mean, or average, of a sample x of lenth 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 4.2. 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.

We can now define another summary of location, the median.

Definition 4.3. 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 4.4. The α trimmed mean of a sample x, denoted $\bar{x}_\alpha$ is the average of the sample after removing the α largest and α 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)
mean(x) # simple mean
```

```
## [1] 0.8588834
```

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

```
## [1] 0.5913073
```

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

```
## [1] 0.6556664
```

4.1.2.3 Summary of Scale

The scale of the data can be thought of its variability.

Definition 4.5. 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 4.6. 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 the MAD is a robust estimate of $S(x)$.

Definition 4.7. 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] 0.8450222
```

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

```
## [1] 0.5830955
```

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

```
## [1] 0.8890492
```

4.1.2.4 Summary of Asymmetry

The symmetry of a univariate sample is easily understood. Summaries of asymmetry, also known as *skewness* quantify the departure of the x from a symmetric distribution.

Definition 4.8. 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)
  numerator/denominator
}
yule(x)
```

```
##       75%
## 0.2527627
```

4.1.2.5 Summary of Bivariate Continous Data

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

Definition 4.9. 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.

Definition 4.10. Peasrson's correlation coefficient, a.k.a. Pearson's moment product correlation, or simply, the correlation, denoted by 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 4.11. 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 $r(x, y) = Cov(z(x), z(y))$.

4.1.2.6 Summary of Multivariate Continous Data

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

Definition 4.12. Given n observations on p variables, the covariance matrix of the sample, denoted $\hat{\Sigma}$ is defined as

$$\hat{\Sigma}_{i,j} = Cov(x_i, x_j)$$

where x_i, x_j are the n observations on variables x_i and x_j respectively.

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

4.2 Visualization

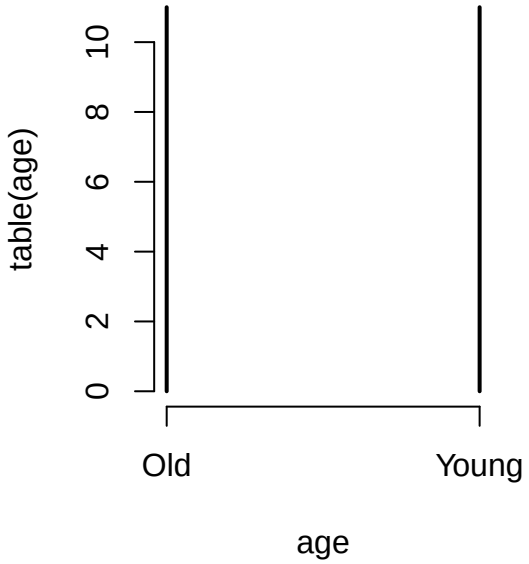
Summarizing the story in a variable to a single number clearly conceals much of the story in the data. This is akin to inspecting a person by its caricature, instead of a picture. Visualizing the data, when possible, is more informative.

4.2.1 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*.

4.2.1.1 Visualizing Univariate Categorical Data

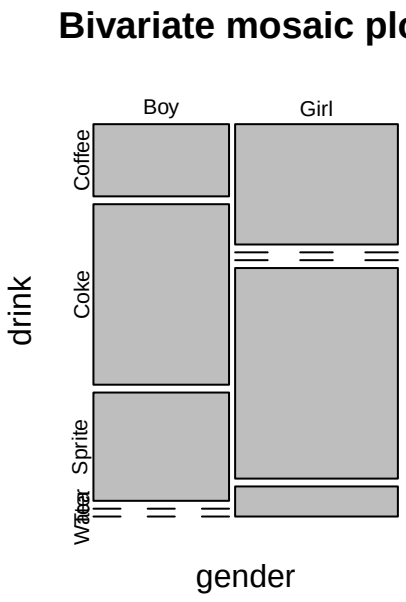
```
plot(table(age))
```



4.2.1.2 Visualizing Bivariate Categorical Data

There are several generalizations of the barplot, aimed to deal with the visualization of bivariate categorical data. There 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.

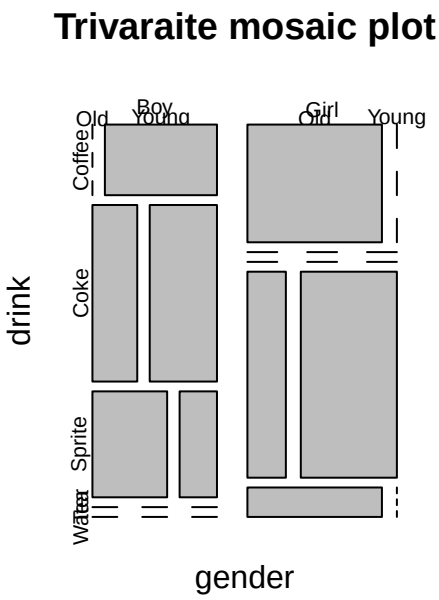
```
plot(table1, main='Bivariate mosaic plot')
```



4.2.1.3 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')
```

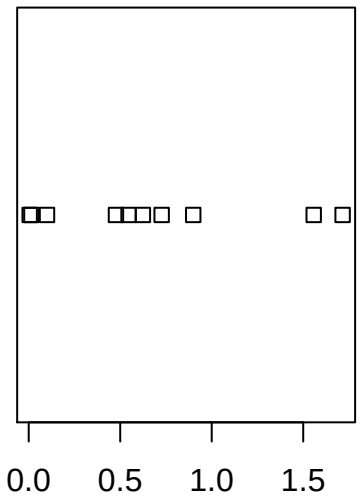


4.2.2 Continous Data

4.2.2.1 Visualizing Univariate Continous Data

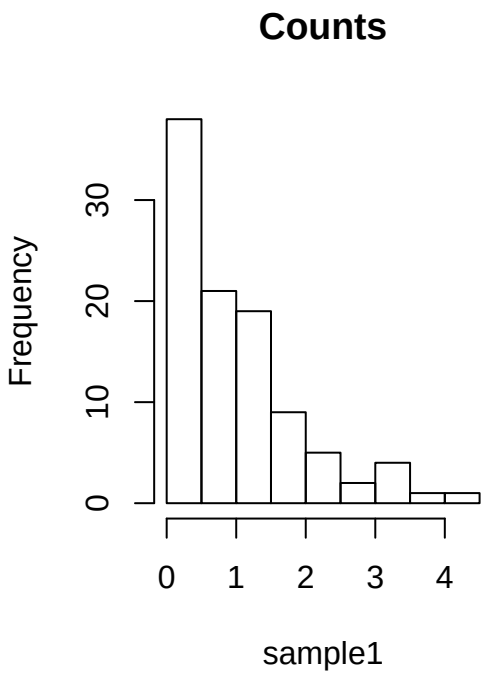
There are endlessly many way to visualize continuous univariate data. The simplest way is to look at the raw data via the `stripcart`.

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

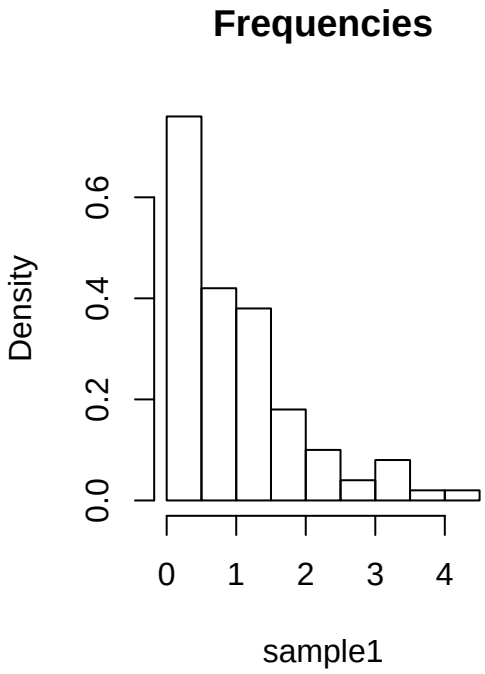


Clearly, if there are many observations, the `stripchart` will be a useless line of black dots. We thus bin them together, and look at the frequency of each bin; this is the *histogram*. R's `histogram` function has very good defaults to choose the number of bins.

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

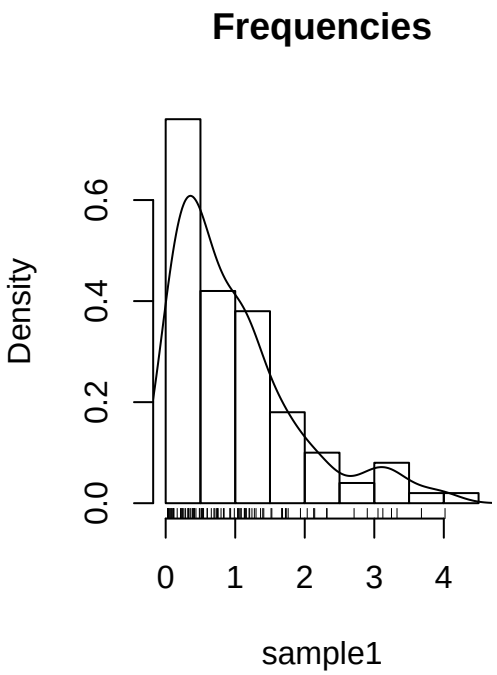



```
hist(sample1, freq=F, main='Frequencies')
```



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 produce with the `density` functions, 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 due to the binning of the histogram, or the smoothing of the density plot.

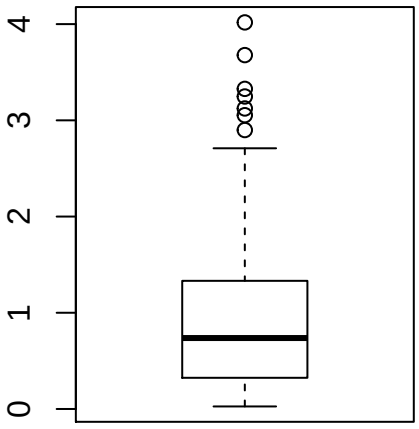
```
hist(sample1, freq=F, main='Frequencies')
lines(density(sample1))
rug(sample1)
```



Remark. Why would it make no sense of making a table, or a barplot, of continous 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.

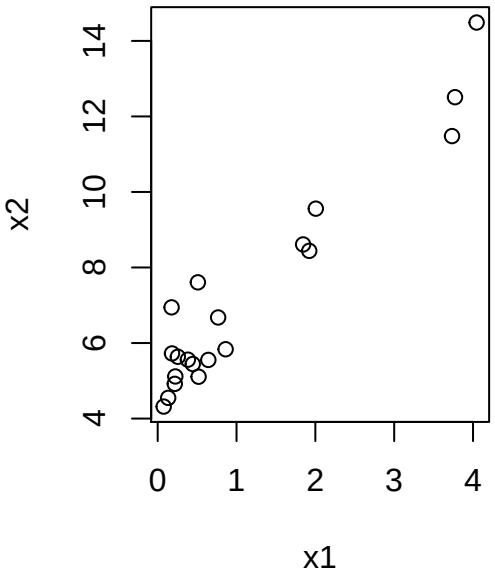
```
boxplot(sample1)
```



4.2.2.2 Visualizing Bivariate Continous Data

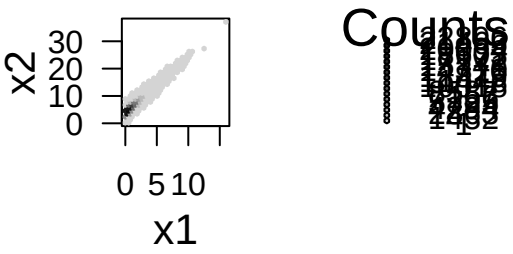
The bivariate counterpart of the `stipchart` is the celebrated scatter plot.

```
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 bivariate plane with hexagons.

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

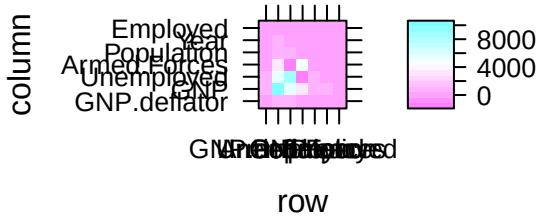


4.2.2.3 Visualizing Multivariate Continous Data

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 (10.1). (ii) To visualize not the data, but the summaries. Like the covariance matrix.

Since the covariance matrix, $\hat{\Sigma}$ is a matrix, it can be visualized as a matrix.

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



4.3 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).

Chapter 5

Linear Models

5.1 Problem Setup

Example 5.1. Consider a randomized experiment designed to study the effects of temperature and pressure on the diameter of a bottle cap.

Example 5.2. 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. We don't care about the causal effects, we just want good predictions.

In this chapter we discuss the causal problem in Example 5.1. This means that when we assume a model, we assume it is the actual *data generating process*. The second type of problems is discussed in the Supervised Learning Chapter 9.

Lets present the linear model. We assume that a response[1] variable is the sum of effects of some factors[2]. Denoting the dependent by y, the factors by x, and the effects by β the linear model assumption implies that

$$E[y] = \sum_j x_j \beta_j = x'\beta. \tag{5.1}$$

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. The implied generative process of a sample of $i = 1, \ldots, n$ observations it thus

$$y_i = \sum x_{i,j}\beta_j + \varepsilon_i, i = 1, \ldots, n. \tag{5.2}$$

or in matrix notation

$$y = X\beta + \varepsilon. \tag{5.3}$$

Lets demonstrate Eq.(5.2):

In our cap example, 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,

[1] The "response" is also know as the "dependent" variable, of the "labels" in the machine learning literature.
[2] The "factors" are also known as the "independent variable", the "design", the "features" and the "attributes".
[3] The "error term" is also known as the "noise", or the "common causes of variability".

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*.

In Gosset'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 these models are so popular:

1. Before the computer age, these were pretty much the only models that could actually be computed[4]. The whole Analysis of Variance (ANOVA) literature is an instance of linear models.
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 categorical or factorial 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 the 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: How the estimate β, and how to perform inference.

In linear models the estimation of β is done using the method of least squares. For this reason, a linear model with least squares estimation is known as Ordinary Least Squares (OLS). The OLS problem:

$$\hat{\beta}_{OLS} := argmin_\beta \{\sum_i (y_i - x_i'\beta)^2\}, \tag{5.4}$$

and in matrix notation

$$\hat{\beta}_{OLS} := argmin_\beta \{\|y - X\beta\|_2^2\}. \tag{5.5}$$

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

Different software suits, and even different R packages, solve Eq.(5.4) in different ways so that we skip the details of how exactly it is solved.

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

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.
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

$$\hat{\beta}_{OLS} \sim \mathcal{N}(\beta, (X'X)^{-1}\sigma^2) \tag{5.6}$$

The reason I am not too strict about the normality assumption above, is that Eq.(5.6) is approximately correct even if ε is not normal, provided that there are many more observations than factors ($n \gg p$).

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

5.2 OLS Estimation

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 insulation.

```
library(MASS)
data(whiteside)
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 with `lm` function, possibly the most important function in R.

```
lm.1 <- lm(Gas~Temp, data=whiteside[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 only observations before the insulation.
- The result is assigned to the object `lm.1`.

Alternative formulations with the same results would be

```
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)
```

The output is an object of class `lm`.

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

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

Objects of class `lm` are very complicated. It stored a lot of information which will be later 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[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"
```

At this point, we only want $\hat{\beta}_{OLS}$ which can be extracted with the `coef` function.

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

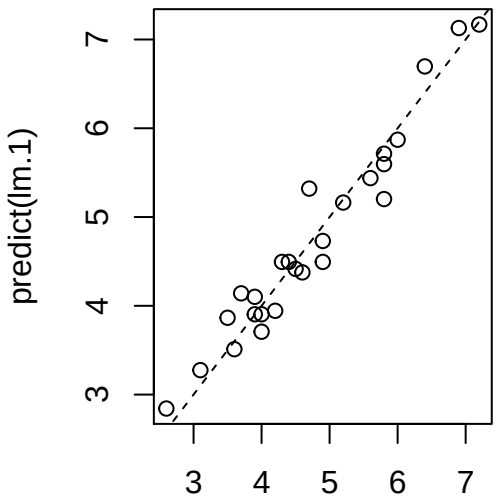
```
## (Intercept)        Temp
##   6.8538277  -0.3932388
```

Things to note:

- R automatically adds an `(Intercept)` term. This means we estimate $y_i = \beta_0 + \beta_1 Gas + \varepsilon$ and not $y_i = \beta_1 Gas + \varepsilon_i$. This makes sense because we are interested in 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 9.

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



whiteside[whiteside$Insul == "Before",]$

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 5.1. The coefficient of determination, denoted R^2, is defined as

$$R^2 := 1 - \frac{\sum_i (y_i - \hat{y}_i)^2}{\sum_i (y_i - \bar{y})^2} \tag{5.7}$$

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(whiteside[whiteside$Insul=='Before',]$Gas, predict(lm.1))
```

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

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

5.3 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.(5.6) 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.(5.6).

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

```
##
## Call:
## lm(formula = Gas ~ Temp, data = whiteside[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
```

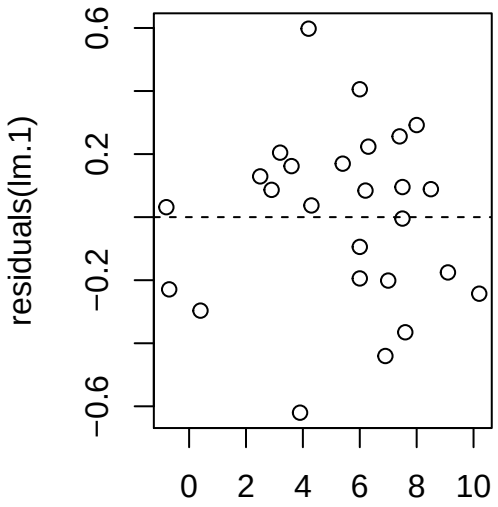
Things to note:

- 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"[5] is $\sqrt{(n-p)^{-1}\sum_i (y_i - \hat{y}_i)^2}$. The "degrees of freedom" are $n - p$ 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.

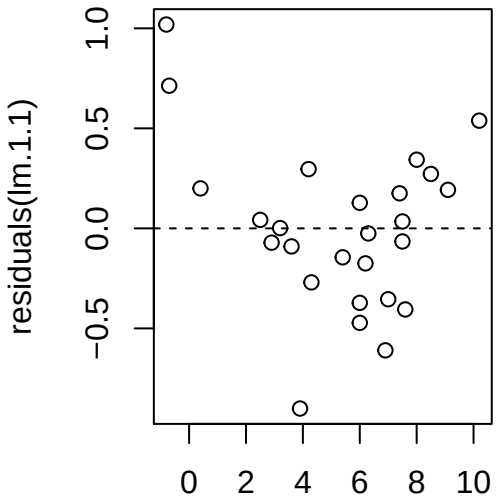
```
plot(residuals(lm.1)~whiteside[whiteside$Insul=='Before',]$Temp); abline(0,0, lty=2)
```



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[whiteside$Insul=='Before',])
plot(residuals(lm.1.1)~whiteside[whiteside$Insul=='Before',]$Temp); abline(0,0, lty=2)
```

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



whiteside[whiteside$Insul == "Before",]$

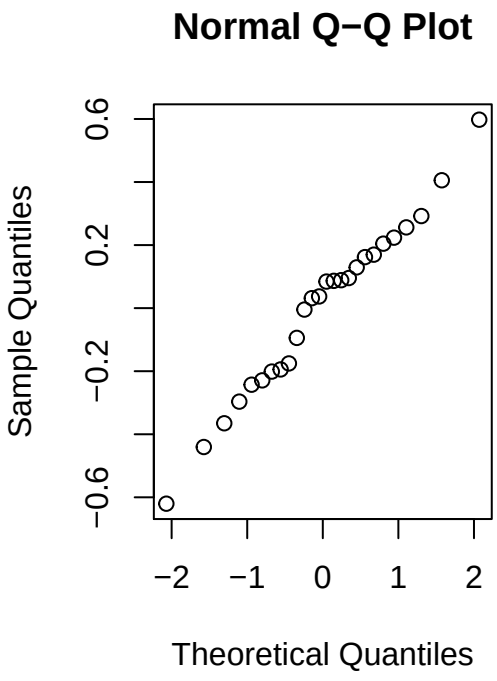
Things to note:

- We used `I(Temp)^2` to specify the model $Gas_i = \beta_0 + \beta_1 Temp^2 + \varepsilon$.
- The residuals have a "belly". Because they are not a cloud of noise 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 is not really needed. If $n \sim p$, i.e., 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 an assumed distribution. If quantiles align along a line, the assumed distribution if OK. If quantiles depart from a line, then clearly the assumed distribution does not fit the sample.

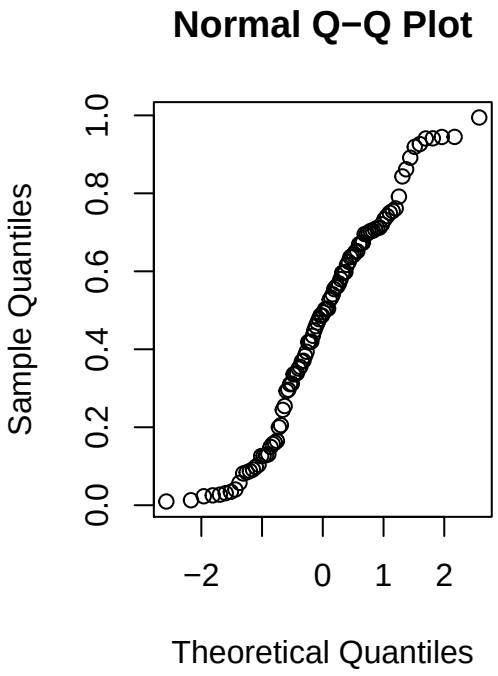
```
qqnorm(resid((lm.1)))
```



The `qqnorm` function plots a qqplot against a normal distribution. 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))
```



5.3.1 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.

5.3.2 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(-1,1) * coefs.lm1[2,2]
```

```
## [1] -0.4128248 -0.3736528
```

5.3.3 Multiple Regression

Remark. *Multiple regression* is not to be confused with *multivariate regression* discussed in Chapter 8.

The data we now use[6] 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 5. This dataset can be conceptualized as a comparison between three stress treatment programs, one using mental methods, one using physical training, and one using medication across genders. The values represent how effective the treatment programs were at reducing participant's stress levels, with higher numbers indicating higher effectiveness.

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

```
##   Treatment   Age StressReduction
## 1    mental young              10
## 2    mental young               9
## 3    mental young               8
## 4    mental   mid               7
## 5    mental   mid               6
## 6    mental   mid               5
```

How many observations per group?

```
table(data$Treatment, data$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~.-1,data=data)
summary(lm.2)
```

```
##
## Call:
## lm(formula = StressReduction ~ . - 1, data = data)
##
## Residuals:
##    Min     1Q Median     3Q    Max
##     -1     -1      0      1      1
##
## Coefficients:
##                    Estimate Std. Error t value Pr(>|t|)
## Treatmentmedical     4.0000     0.3892  10.276 7.34e-10 ***
## Treatmentmental      6.0000     0.3892  15.414 2.84e-13 ***
## Treatmentphysical    5.0000     0.3892  12.845 1.06e-11 ***
## 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:  0.9794, Adjusted R-squared:  0.9747
## F-statistic:   209 on 5 and 22 DF,  p-value: < 2.2e-16
```

Things to note:

- The `StressReduction~.` syntax is read as "Stress reduction as a function of everything else".

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

- The `StressReduction~.-1` means that I do not want an intercept in the model, so that the baseline response is 0.
- All the (main) effects 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. Their numerical values are meaningless, and it has thus 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.

```
head(model.matrix(lm.2))
```

```
##   Treatmentmedical Treatmentmental Treatmentphysical Ageold Ageyoung
## 1                0                1                 0      0        1
## 2                0                1                 0      0        1
## 3                0                1                 0      0        1
## 4                0                1                 0      0        0
## 5                0                1                 0      0        0
## 6                0                1                 0      0        0
```

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  3    693 231.000  282.33 <2e-16 ***
## Age        2    162  81.000   99.00  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 nubmer 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.

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=data)
```

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

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

Lets inspect the model

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

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

```
## lm(formula = StressReduction ~ Treatment + Age + Treatment:Age -
##     1, data = data)
##
## 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.

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) - 1
## 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         0  0      1
```

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

5.3.4 Testing a Hypothesis on a Single Contrast

Returning to `lm.2`.

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

```
##                   Estimate Std. Error   t value     Pr(>|t|)
## Treatmentmedical         4  0.3892495 10.276186 7.336391e-10
## Treatmentmental          6  0.3892495 15.414279 2.835706e-13
```

```
## Treatmentphysical        5  0.3892495 12.845233 1.064101e-11
## 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 levels of a factor 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 ~ . - 1, data = data)
##
## Linear Hypotheses:
##        Estimate Std. Error t value Pr(>|t|)
## 1 == 0   1.0000     0.4264   2.345   0.0284 *
## ---
## 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.

5.4 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).

Chapter 6

Generalized Linear Models

Example 6.1. Consider the relation between cigarettes smoked, and the occurance of lung cancer. Do we expect it to be liner? Probably not. Do we expect the variability to be constant about the trend, be it linear or not? Probably not.

6.1 Problem Setup

In the Linear Models Chapter 5, we assumed the generative process to be

$$y|x = x'\beta + \varepsilon \tag{6.1}$$

This does not allow for assumingly non-linear relations, nor does it allow for the variability of ε to change with x. Generalize linear models (GLM), as the name suggests, are a generalization that allow that.
Remark. Do not confuse *generalized linear models* with *non-linear regression*, or *generalized least squares*. These are different things, that we will not discuss.

To understand GLM, we recall that with the normality of ε, Eq.(6.1) implies that

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

For Example 6.1, we would like something in the lines of

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

More generally, for some distribution $F(\theta)$, with a parameter θ, we would like

$$y|x \sim F(\theta(x)) \tag{6.2}$$

Possible examples include

$$y|x \sim Poisson(\lambda(x)) \tag{6.3}$$
$$y|x \sim Exp(\lambda(x)) \tag{6.4}$$
$$y|x \sim \mathcal{N}(\mu(x), \sigma^2(x)) \tag{6.5}$$

GLMs constrain θ to be some function, g, of a linear combination of the x's. Formally,

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

, where

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

. The function g is called the *link* function.

6.2 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 implied by the logistic regression is the logistic function

$$g(t) = \frac{e^t}{(1+e^t)} \tag{6.6}$$

implying that

$$y|x \sim Binom\left(1, p = \frac{e^{x'\beta}}{1+e^{x'\beta}}\right) \tag{6.7}$$

Before we fit such a model, we try to justify this construction, in particular, this enigmatic link function in Eq.(6.6). 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.

$$p(x=0) = P(y=1|x=0) = \frac{e^{\beta_0}}{(1+e^{\beta_0})} \tag{6.8}$$

$$\Rightarrow \frac{P(y=1|x=0)}{P(y=0|x=0)} = e^{\beta_0} \tag{6.9}$$

$$p(x=1) = P(y=1|x=1) = \frac{e^{\beta_0+\beta_1}}{(1+e^{\beta_0+\beta_1})} \tag{6.10}$$

$$\Rightarrow \frac{P(y=1|x=1)}{P(y=0|x=1)} = e^{\beta_0+\beta_1} \tag{6.11}$$

$$\Rightarrow \frac{P(y=1|x=1)/P(y=0|x=1)}{P(y=1|x=0)/P(y=0|x=0)} = e^{\beta_1} \tag{6.12}$$

$$\Rightarrow \log \frac{P(y=1|x=1)/P(y=0|x=1)}{P(y=1|x=0)/P(y=0|x=0)} = \beta_1. \tag{6.13}$$

The magnitudes in Eqs.(6.8) and (6.11), are known as the *odds*. Odds are the same as probabilities, but instead of of telling me there is a 66% of success, they tell me the odds of success are "2 to 1".

The magnitude in Eq.(6.12) is known as the *odds ratio*. The odds ratio compares between the probabilities of two groups, only that it does not compare them in probability scale, but rather in odds scale.

The magnitude in Eq.(6.13) is known as the *log odds ratio*. Besides some nice theoretical properties of log odds ratios, which we will not discuss, they are important since it demystifies the choice of the link function in (6.6): **it allows us to interpret β of the logistic regression as the odds-ratios (in log scale)**.

Another popular link function is the normal quantile function, a.k.a., the Gaussian inverse CDF, leading to *probit regression* instead of logistic regression.

6.2.1 Logistic Regression with R

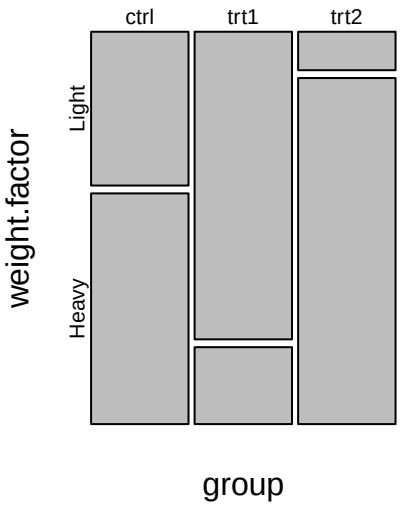
Let's get us some data. The `PlantGrowth` data records the weight of plants under three conditions: control, treatment1, and treatment2.

```
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 two-class 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.

```
attach(PlantGrowth)
weight.factor<- cut(weight, 2, labels=c('Light', 'Heavy'))
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 output is binomial, 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 5.
- As usual, we get the coefficients table, but recall that they are to be interpreted as (log) odd-ratios.
- As usual, we get the significance for the test of no-effect, versus a two-sided alternative.
- The residuals of `glm` are slightly different than the `lm` residuals, and called *Deviance Residuals*.
- For help see `?glm`, `?family`, and `?summary.glm`.

Like for 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.
- In GLMs there is no canonical test (like the F test for `lm`). 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.
- In GLMs there are many types of predictions. The `type` argument controls which type is returned.
- 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 significant predictors.

6.3 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 $g(t) = e^t$. 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. For our example[1] we inspect the number of infected high-school kids, as a function of the days since the 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,
    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)
```

```
## The following objects are masked from cases (pos = 4):
##
##     Days, Students

## The following objects are masked from cases (pos = 6):
##
##     Days, Students

## The following objects are masked from cases (pos = 9):
##
##     Days, Students

## The following objects are masked from cases (pos = 44):
##
##     Days, Students

## The following objects are masked from cases (pos = 49):
##
##     Days, Students
```

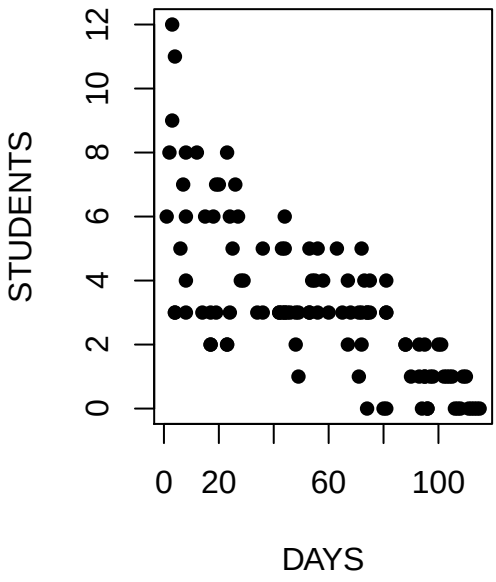
```
head(cases)
```

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

And visually:

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

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



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** by assumption.
- Each day **decreases** the rate of events by a factor of about 0.02.
- For more information see `?glm` and `?family`.

6.4 Extensions

As we already implied, GLMs are a very wide class of models. We do not need to use the default link function,but more importantly, we are not constrained to Binomial, or Poisson distributed response. For exponential, gamma, and other response distributions, see `?glm` or the references in the Bibliographic Notes section.

6.5 Bibliographic Notes

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

Chapter 7

Linear Mixed Models

Example 7.1. Consider the problem of testing for a change in the distribution of the bottle caps produced. Bottle caps are produced by several machines. We could standardize by removing each machine's average. This first practice implies the within-machine variability is the only source of variability. Alternatively, we could ignore the machine of origin. This second practice implies there are two sources of variability: the within-machine variability, and the between-machine variability. The former practice is known as a *fixed effects* model. The latter as a *random effects* model.

Example 7.2. Consider a crossover[1] experimenal design where each subject is given 2 types of diets, and his/hers health condition is recorded. We could standardize by removing each subject's average, before comparing the diets (think of a paired t-test). This first practice implies the within-subject variability is the only source of variability. Alternatively, we could ignore the subject of origin. When doing so, we need to recall that observations from the same subject will be correlated. This second practice implies there are two sources of variability: the within-subject variability and the betwee-subject variability.

The unifying theme of the above two examples, is that the variability we want to infer against has several sources. This is typical in mixed models, which are so popular, 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.
- **Variance Components**: Because as the examples show, variance has more than a single source (like in the Linear Models of Chapter 5).
- **Hirarchial Models**: Because as Example 7.2 demonstrates, we can think of the sampling as hierarchical– first sample a subject, and then sample its response.
- **Repeated Measures**: Because we many have several measurements from each unit, like in 7.2.
- **Longitudinal Data**: Because we follow units over time, like in Example 7.2.
- **Panel Data**: Is the term typically used in econometric for such longitudinal data.

We now emphasize:

1. Mixed effect models 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, i.e., error rates are higher than you think.
2. A mixed effect models, as we will later see, is typically specified via its fixed and random effects. It is possible, however, to specify a mixed effects model by putting all the fixed effects into a linear model, and putting all the random effects into the covariance between ε. For more on this view, see Chapter 8 in (the excellent) Weiss (2005).
3. Like in previous chapters, by "model" we refer to the assumed generative distribution, i.e., the sampling distribution.
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 9 for more on prediction problems.

[1] If you are unfamiliar with design of experiments, have a look at Chapter 6 of my Quality Engineering class notes.

7.1 Problem Setup

$$y|x,u = x'\beta + z'u + \varepsilon \tag{7.1}$$

where x are the factor with fixed effects, β which we may want to study. The factors z, with effects u, are the random effects which contribute to variability. Put differently, we state $y|x,u$ merely as a convenient way to do inference on $y|x$, instead of directly specifying $Var[y|x]$ as a function of u.

Given a sample of n observations (y_i, x_i, z_i) from model (7.1), 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 mixed-models, however, ML is typically replaced with *restricted maximum likelihood* (ReML), because it returns unbiased estimates of $Var[y|x]$ and ML does not.

7.2 Mixed Models with R

We will fit mixed models with the `lmer` function from the **lme4** package. We start with a small simulation demonstrating the importance of acknowledging your sources of variability.

```
n.groups <- 10
n.repeats <- 2
groups <- gl(n = n.groups, k = n.repeats)
n <- length(groups)
z0 <- rnorm(10,0,10)
z <- z0[as.numeric(groups)] # create the random effect vector.
epsilon <- rnorm(n,0,1) # create the measurement error vector.
beta0 <- 2 # create the global mean
y <- beta0 + z  + epsilon # generate synthetic sample
lm.5 <- lm(y~z)  # fit a linear model
library(lme4)
lme.5 <- lmer(y~1|z) # fit a mixed-model
```

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
## -1.8115 -0.4222  0.2243  0.3970  2.8710
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)   1.8614     0.2397   7.764 3.75e-07 ***
## z             0.9770     0.0225  43.422  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.031 on 18 degrees of freedom
## Multiple R-squared:  0.9905, Adjusted R-squared:   0.99
## F-statistic:  1885 on 1 and 18 DF,  p-value: < 2.2e-16
```

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 | z
##
## REML criterion at convergence: 109.7
##
## Scaled residuals:
##      Min       1Q   Median       3Q      Max
## -1.90939 -0.18185  0.03551  0.17218  1.73061
##
## Random effects:
##  Groups   Name        Variance Std.Dev.
##  z        (Intercept) 110.725  10.523
##  Residual               1.515   1.231
## Number of obs: 20, groups:  z, 10
##
## Fixed effects:
##             Estimate Std. Error t value
## (Intercept)    4.733      3.339   1.417
```

Look at the standard error of the global mean, i.e., the intercept: for `lm` it is 0.2397539, and for `lme` it is `summary.lme.5$coefficients[1,2]`. Why this difference? Because `lm` discounts the group effect, while it should treat it as another source of variability. Clearly, inference using `lm` is overly optimistic.

7.2.1 A Single Random Effect

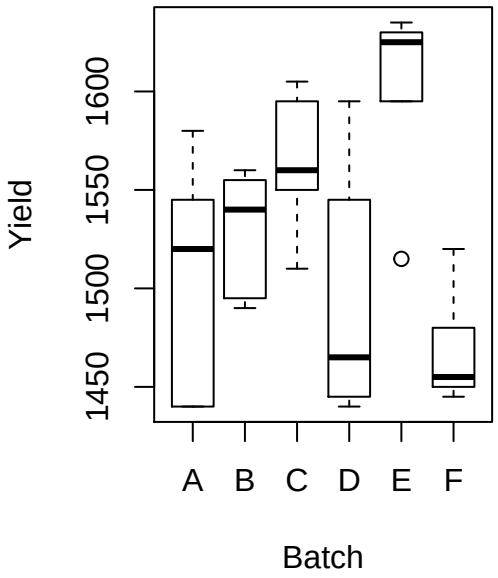
We will use the `Dyestuff` data from the package, which encodes the yield, in grams, of a coloring solution (dyestuff), produced in 6 batches using 5 different preparations.

```
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

```
plot(Yield~Batch)
```



If we want to do inference on the mean yield, we need to account for the two sources of variability: the batch effect, and the measurement error. We thus fit a mixed model, with an intercept and random batch effect (which means this is it not a bona-fide mixed-model, but rather, a simple random-effect model).

```
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:

- As usual, `summary` is content aware and has a different behavior for `lme` class objects.
- 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.
- The output distinguishes between random effects, a source of variability, and fixed effect, white's coefficients we want to study.
- Were we not interested in the variance components, 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 (i) a delicate matter, and (ii) better treated as prediction problems as in the Supervised Learning Chapter 9.

7.2.2 Several Random Effects

Let's make things more interesting. 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).

```
detach(Dyestuff)
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)
```

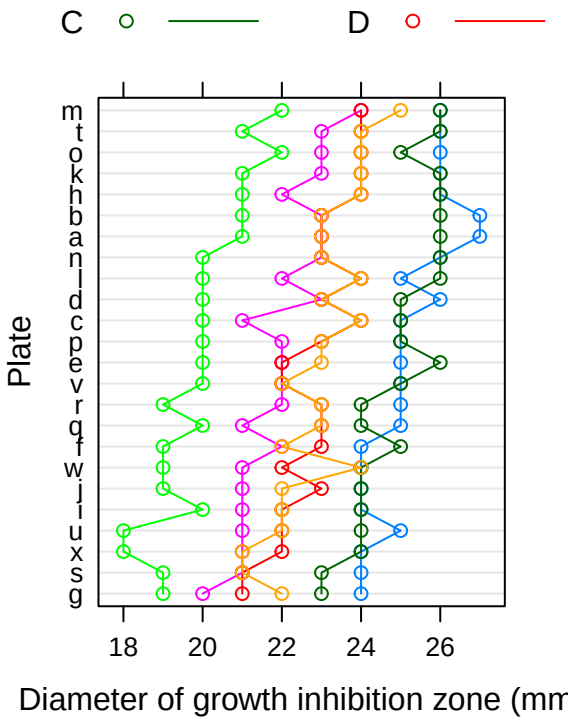
```
## The following objects are masked from Penicillin (pos = 4):
##
##     diameter, plate, sample

## The following objects are masked from Penicillin (pos = 6):
##
##     diameter, plate, sample

## The following objects are masked from Penicillin (pos = 9):
##
##     diameter, plate, sample

## The following objects are masked from Penicillin (pos = 44):
##
##     diameter, plate, sample

## The following objects are masked from Penicillin (pos = 47):
##
##     diameter, plate, sample
```

```
table(sample, plate)
```

```
##       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:

```
lattice::dotplot(reorder(plate, diameter) ~ diameter,data=Penicillin,
              groups = sample,
              ylab = "Plate", xlab = "Diameter of growth inhibition zone (mm)",
              type = c("p", "a"), auto.key = list(columns = 6, lines = TRUE))
```



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)`.

Since we have two random effect, 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.

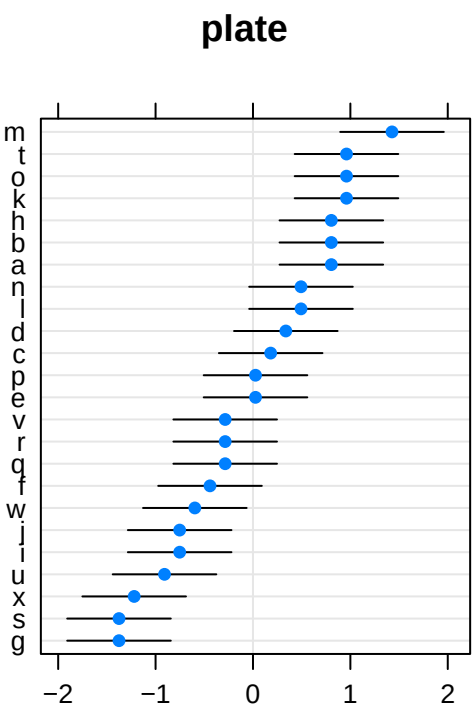
```
random.effect.lme2 <- ranef(lme.2, condVar = TRUE)
qrr2 <- lattice::dotplot(random.effect.lme2, strip = FALSE)
```

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.

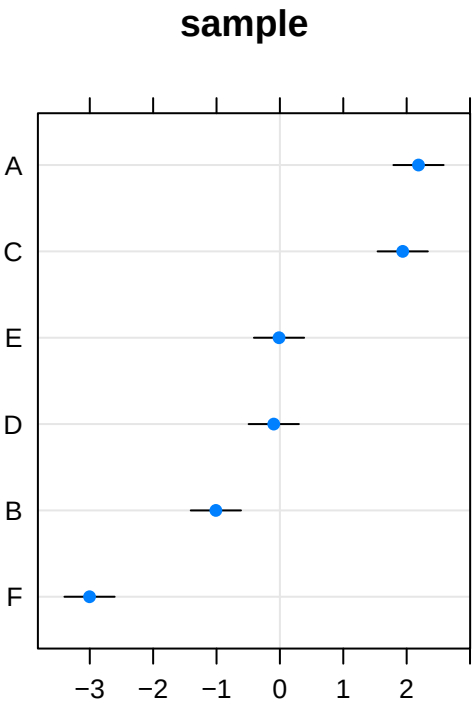
Variability in response for each plate, over various sample types:

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



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

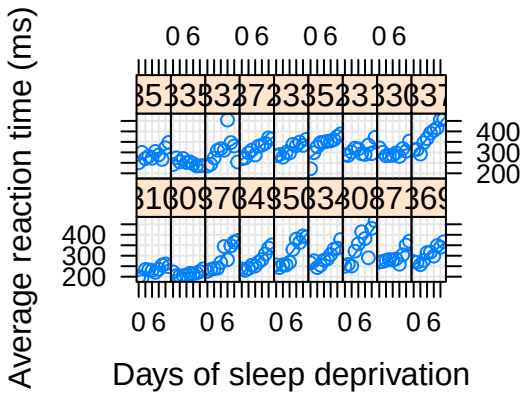
```
print(qrr2[[2]])
```



7.2.3 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`).

```
data(sleepstudy)
lattice::xyplot(Reaction ~ Days | Subject, sleepstudy, aspect = "xy",
             layout = c(9,2), type = c("g", "p", "r"),
             index.cond = function(x,y) coef(lm(y ~ x))[1],
             xlab = "Days of sleep deprivation",
             ylab = "Average reaction time (ms)")
```



We now want to estimate the (fixed) effect of the days of deprivation, while allowing each subject to have his/hers own effect. The fixed effect can thus be thought of as the average slope over subjects.

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

Things to note:

- 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 (i.e. average) day effect is:

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

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

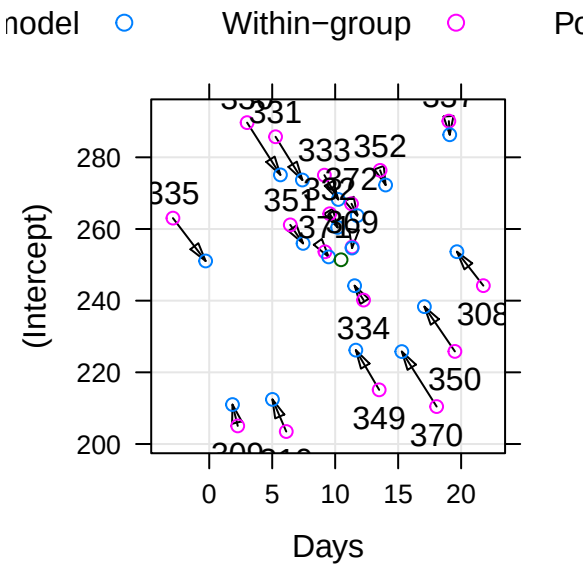
The variability in the average response (intercept) and day effect is

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

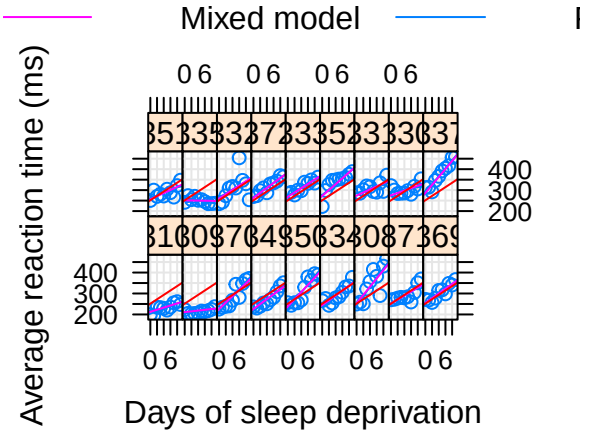
```
## $Subject
##     (Intercept)        Days
## 308   2.2585654   9.1989719
## 309 -40.3985770  -8.6197032
## 310 -38.9602459  -5.4488799
## 330  23.6904985  -4.8143313
## 331  22.2602027  -3.0698946
## 332   9.0395259  -0.2721707
## 333  16.8404312  -0.2236244
## 334  -7.2325792   1.0745761
## 335  -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.

To demonstrate the "strength borrowing", here is a comparison of the subject-wise intercepts of the mixed-model, versus a subject-wise linear model. They are not the same.



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



7.3 Bibliographic Notes

Most of the examples in this chapter are from the documentation of the **lme4** package (Bates et al., 2015). For a more theoretical view see Weiss (2005) or Searle et al. (2009). As usual, a hands on view can be found in Venables and Ripley (2013).

Chapter 8

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 10.

More formally, let y be a p variate random vector, with $E[y] = \mu$. We will discuss the problems of

- **Signal detection**: a.k.a. *multivariate hypothesis testing*, i.e., testing if μ equals μ_0 and for $\mu_0 = 0$ in particular.
- **Signal counting**: Counting the number of elements in μ that differ from μ_0, and for $\mu_0 = 0$ in particular.
- **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.
- **Signal estimation**: a.k.a. *selective inference*, i.e., estimating the magnitudes of the departure of μ from μ_0, and for $\mu_0 = 0$ in particular.
- **Multivariate Regression**: a.k.a. *MANOVA* in statistical literature, and *structured learning* in the machine learning literature.
- **Distribution fitting**: A.k.a. *structure learning* in the machine learning literature, deals with the fitting a distribution to samples from y. In particular, it deals with the identification of independencies between elements of y. For samples from a multivariate Gaussian distribution, learning the distribution implies all the above problem applied to $Var[y]$, instead of $E[y]$.

8.1 Signal Detection

8.2 Signal Counting

8.3 Signal Identification

8.4 Signal Estimation

8.5 Multivariate Regression

8.6 Distribution Fitting

Chapter 9

Supervised Learning

Machine learning is very similar to statistics, but it is certainly not the same. As the name suggests, in machine learning we want machines to learn. This means that we want to replace hard-coded expert algorithm, with data-driven self-learned algorithm.

There are many learning setups, that depend on what is available to the machine. The most common setup, discussed in this chapter, is *supervised learning*. The name takes from the fact that by giving the machine data samples with known inputs (a.k.a. features) and desired outputs (a.k.a. labels), the human is effectively supervising the learning. If we think of the inputs as predictors, and outcomes as predicted, it is no wonder that supervised learning is very similar to statistical prediction. When asked "are these the same?" I like to give the example of internet fraud. If you take a sample of fraud "attacks", a statistical formulation of the problem is highly unlikely. This is because fraud events are not randomly drawn from some distribution, but rather, arrive from an adversary learning the defenses and adapting to it. This instance of supervised learning belongs in game theory, more than it does in statistics.

Other types of machine learning problems include:

- **Unsupervised learning**: See Chapter 10.
- **Semi supervised learning**: Where only part of the samples are labeled. A.k.a. *co-training*, *learning from labeled and unlabeled data*, *transductive learning*.
- **Active learning**: Where the machine is allowed to query the user for labels. Very similar to *adaptive design of experiments*.
- **Reinforcement learning**: Similar to active learning, in that the machine may query for labels. Different from active learning, in that the machine does not receive labels, but *rewards*.
- **Learning on a budget**: A version of active learning where querying for labels induces variable costs.
- **Structure learning**: The learning of the dependence structure between variables.
- **Learning to learn**: Deals with the carriage of "experience" from one learning problem to another. A.k.a. *cummulative learning* and *meta learning*.
- **Manifold learning**: An instance of unsupervised learning, where the goal is to reduce the dimension of the data by embedding it into a lower dimensional manifold. A.k.a. *support estimation*.

9.1 Problem setup

We now present the *empirical risk minimization* to supervised learning.
Remark. We do not discuss purely algorithmic approaches such as K-nearest neighbour and *kernel smoothing* due to space constraints. For a broader review of supervised learning, see the Bibliographic Notes section.

Given n samples with inputs x from some space $\mathcal{X}$ and desired outcome, y, from some space $\mathcal{Y}$. Samples, (x, y) have some distribution we denote P. We want to learn a function that maps inputs to outputs. This function is called a *hypothesis*, or *predictor*, or *classifier* denoted f, that belongs to a hypothesis class $\mathcal{F}$ such that $f : \mathcal{X} \to \mathcal{Y}$. We also choose some other function that fines us for erroneous prediction. This function is called the *loss*, and we denote it by $l : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}^+$.
Remark. The *hypothesis* in machine learning is only vaguely related the *hypothesis* in statistical testing, which is quite confusing.

Remark. The *hypothesis* in machine learning is not a bona-fide *statistical model* since we don't assume it is the data generating process, but rather some function which we choose for its good predictive performance.

The fundamental task in supervised (statistical) learning is to recover a hypothesis that minimizes the average loss in the sample, and not in the population. This is know as the *risk minimization problem*.

$$f^* := argmin_f\{E_P[l(f(x), y)]\} \tag{9.1}$$

To make things more explicit, f may be a linear function, and l a squared error loss, in which case problem (9.1) collapses to

$$f^* := argmin_\beta\{E_P[(x'\beta - y)^2]\} \tag{9.2}$$

Another fundamental problem is that we do not know the distribution of all possible inputs and outputs, P. We typically only have a sample of $(x_i, y_i), i = 1, \ldots, n$. We thus state the *empirical* counterpart of (9.1), which consists of minimizing the average loss. This is known as the *empirical risk miminization* problem (ERM).

$$\hat{f} := argmin_f\{\sum_i l(f(x_i), y_i)\} \tag{9.3}$$

Making things more explicit again by using a linear hypothesis with squared loss, we see that the empirical risk minimization problem collapses to an ordinary least-squares problem:

$$\hat{f} := argmin_\beta\{\sum_i (x_\beta - y_i)^2\} \tag{9.4}$$

When data is samples are independent, then maximum likelihood estimation is also an instance of ERM, when using the (negative) log likelihood as the loss function.

If we don't assume any structure on the hypothesis, f, then $\hat{f}$ from (9.3) will interpolate the data, and will be a very bad predictor. We say, it will *overfit* the observed data, and will have bad performance on new data.

We have several ways to avoid overfitting:

1. Restrict the hypothesis class $\mathcal{F}$ (such as linear functions).
2. Penalize for the complexity of f. The penalty denoted by $\|f\|$.
3. Unbiased risk estimation, where we deal with the overfitted optimism of the empirical risk by debiasing it.

9.1.1 Common Hypothesis Classes

Some common hypothesis classes, $\mathcal{F}$, with restricted complexity, are:

1. **Linear hypotheses**: such as linear models, GLMs, and (linear) support vector machines (SVM).
2. **Neural networks**: a.k.a. *feed-forward* neural nets, *artificial* neural nets, and the celebrated class of *deep* neural nets.
3. **Tree**: a.k.a. *decision rules*, is a class of hypotheses which can be stated as "if-then" rules.
4. **Reproducing Kernel Hilbert Space**: a.k.a. RKHS, is a subset of "the space of all functions[1]" that is both large enough to capture very complicated relations, but small enough so that it is less prone to overfitting, and also surprisingly simple to compute with.
5. **Ensembles**: a "meta" hypothesis class, which consists of taking multiple hypotheses, possibly from different classes, and combining them.

[1] It is even a subset of the Hilbert space, itself a subset of the space of all functions.

9.1.2 Common Complexity Penalties

The most common complexity penalty applied to classes that have a finite dimensional parametric representation, such as a the linear class parametrized via its coefficients β. In such classes we may penalize for the norm of the parameters. Common penalties include:

1. **Ridge penalty**: penalizing the l_2 norm of the parameter. I.e. $\|f\| = \|\beta\|_2^2 = \sum_j \beta_j^2$.
2. **Lasso penalty**: penalizing the l_1 norm of the parameter. I.e., $\|f\| = \|\beta\|_1 = \sum_j |\beta_j|$
3. **Elastic net**: a combination of the lasso and ridge penalty. I.e. ,$\|f\| = \alpha\|\beta\|_2^2 + (a-\alpha)\|\beta\|_1$.

If the hypothesis class $\mathcal{F}$ does not admit a finite dimensional parametric representation, we may penalize it with some functional norm such as $\|f\|_2^2 = \int f(t)^2 dt$.

9.1.3 Unbiased Risk Estimation

The fundamental problem of overfitting, is that the empirical risk, (9.3), is downward biased to the true risk (9.1), a.k.a. *generalization error*, and *test error*. Why is that? Think of estimating a population's mean with the sample minimum. It can be done, but the minimum has to be debiased for it to estimate the population mean. Debiasing methods broadly fall under purely algorithmic *resampling* based approaches, and theory driven debiasing corrections. These corrections feel like the penalties above, but we state them here because unlike the ridge, and lasso, they are designed for a different purpose.

1. **Train,Validate,Test**: The simplest form of validation is to split the data. A *train* set to train a set of hypotheses. A *validation* set to compute the out-of-sample expected loss, and pick the best performing hypothesis. A *test* sample to compute the out-of-sample performance of the selected hypothesis. This is a very simple approach, but it is very "data inefficient", thus motivating the next method.
2. **V-fold cross validation**: By far the most popular performance assessment algorithm, in *V-fold CV* we "fold" the data into V non-overlapping sets. For each of the V sets, we fit a hypothesis to the non-selected fold, and assess the expected loss on the selected loss. We then aggregate results over the V folds, typically by averaging.
3. **AIC**: Akaike's information criterion (AIC) is a theory driven correction of the empirical risk, so that it is unbiased to the true risk. It is appropriate when using the likelihood loss.
4. **Cp**: Mallow's Cp is an instance of AIC for likelihood loss under normal noise.

Other theory driven unbiased risk estimators include the *Bayesian Information Criterion* (BIC, aka SBC, aka SBIC), the *Minimum Description Length* (MDL), *Vapnic's Structural Risk Minimization* (SRM), the *Deviance Information Criterion* (DIC), and the *Hannan-Quinn Information Criterion* (HQC).

Other resampling based unbiased risk estimators include resampling **without replacement** algorithms like *delete-d cross validation* with its many variations, and **resampling with replacement**, like the *bootstrap*, with its many variations.

9.1.4 Collecting the Pieces

An ERM problem with regularization will look like

$$\hat{f} := argmin_f \{\sum_i l(f(x_i), y_i) + \lambda\|f\|\} \tag{9.5}$$

Collecting ideas from the above sections, a typical supervised learning pipeline will include: choosing the hypothesis class, choosing the penalty function and level, choosing the assessment algorithm. We emphasize that choosing the penalty function is not enough, and we need to choose how "hard" to apply it. This if known as the *regularization level*, typically denoted by λ, which enters

Examples of such combos include:

1. Linear regression, no penalty, train-validate test.
2. Linear regression, no penalty, AIC.
3. Linear regression, l_2 penalty, V-fold CV. This combo is typically known as *ridge regression*.

4. Linear regression, l_1 penalty, V-fold CV. This combo is typically known as *lasso regression*.
5. Linear regression, l_1 and l_2 penalty, V-fold CV. This combo is typically known as *elastic net regression*.
6. Logistic regression, l_2 penalty, V-fold CV.
7. SVM classification, l_2 penalty, V-fold CV.
8. Deep network, no penalty, V-fold CV.

For fans of statistical hypothesis testing we will also emphasize: Testing and prediction are related, but are not the same. **It is indeed possible that we will want to ignore a significant predictor, and add a non-significant one!** (Foster and Stine, 2004) Some authors will use hypothesis testing as an initial screening of candidate predictors. This is a useful heuristic, but that is all it is– a heuristic.

9.2 Supervised Learning in R

At this point, we have a rich enough language to do supervised learning with R.

In these examples, I will use two data sets from the **ElemStatLearn** package: `spam` for categorical predictions (spam mail or not spam?), and `prostate` for continuous predictions (size of cancerous tumor). In `spam` we will try to decide if a mail is spam or not. In `prostate` we will try to predict the size of a cancerous tumor. You can now call `?prostate` and `?spam` to learn more about these data sets.

Some boring pre-processing.

```
library(ElemStatLearn) # for data
data("prostate")
data("spam")

library(magrittr) # for piping

# Preparing prostate data
prostate.train <- prostate[prostate$train, names(prostate)!='train']
prostate.test <- prostate[!prostate$train, names(prostate)!='train']
y.train <- prostate.train$lcavol
X.train <- as.matrix(prostate.train[, names(prostate.train)!='lcavol'] )
y.test <- prostate.test$lcavol
X.test <- as.matrix(prostate.test[, names(prostate.test)!='lcavol'] )

# Preparing spam data:
n <- nrow(spam)

train.prop <- 0.66
train.ind <- c(TRUE,FALSE) %>%
  sample(size = n, prob = c(train.prop,1-train.prop), replace=TRUE)
spam.train <- spam[train.ind,]
spam.test <- spam[!train.ind,]

y.train.spam <- spam.train$spam
X.train.spam <- as.matrix(spam.train[,names(spam.train)!='spam'] )
y.test.spam <- spam.test$spam
X.test.spam <-  as.matrix(spam.test[,names(spam.test)!='spam'])

spam.dummy <- spam
spam.dummy$spam <- as.numeric(spam$spam=='spam')
spam.train.dummy <- spam.dummy[train.ind,]
spam.test.dummy <- spam.dummy[!train.ind,]
```

We also load some utility functions that we will require down the road.

```
l2 <- function(x) x^2 %>% sum %>% sqrt
l1 <- function(x) abs(x) %>% sum
```

```
MSE <- function(x) x^2 %>% mean
missclassification <- function(tab) sum(tab[c(2,3)])/sum(tab)
```

9.2.1 Linear Models with Least Squares Loss

Starting with OLS regression, and a train-test data approach. Notice the better in-sample MSE than the out-of-sample. That is overfitting in action.

```
ols.1 <- lm(lcavol~. ,data = prostate.train)
# Train error:
MSE( predict(ols.1)- prostate.train$lcavol)
```

```
## [1] 0.4383709
```

```
# Test error:
MSE( predict(ols.1, newdata = prostate.test)- prostate.test$lcavol)
```

```
## [1] 0.5084068
```

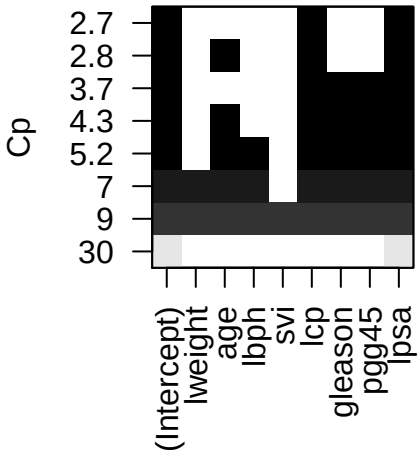
We now implement a V-fold CV, instead of our train-test approach. The assignment of each observation to each fold is encoded in `fold.assignment`. The following implementation is extremely inefficient, but easy to read.

```
folds <- 10
fold.assignment <- sample(1:5, nrow(prostate), replace = TRUE)
errors <- NULL

for (k in 1:folds){
  prostate.cross.train <- prostate[fold.assignment!=k,] # train subset
  prostate.cross.test <-  prostate[fold.assignment==k,] # test subset
  .ols <- lm(lcavol~. ,data = prostate.cross.train) # train
  .predictions <- predict(.ols, newdata=prostate.cross.test)
  .errors <-  .predictions - prostate.cross.test$lcavol # save prediction errors in the fold
  errors <- c(errors, .errors) # aggregate error over folds.
}

# Cross validated prediction error:
MSE(errors)
```

```
## [1] 0.6024089
```

Let's try all possible models, and choose the best performer with respect to the Cp criterion. We see that the best performer has 3 predictors.

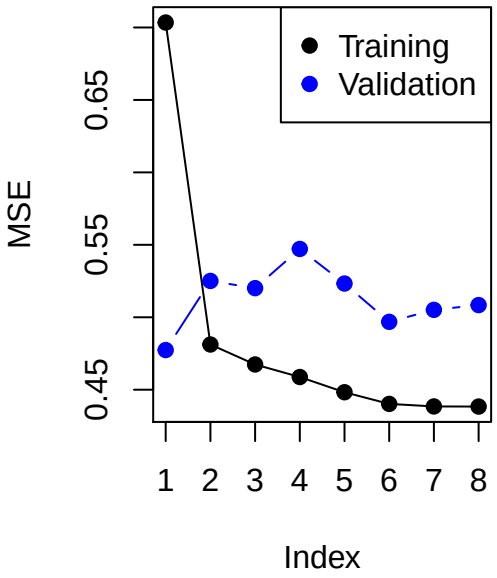
```
library(leaps)
regfit.full <- prostate.train %>%
  regsubsets(lcavol~.,data = ., method = 'exhaustive') # best subset selection
plot(regfit.full, scale = "Cp")
```



Instead of the Cp criterion, we now compute the train and test errors for all the possible predictors[2].

```
model.n <- regfit.full %>% summary %>% length
X.train.named <- prostate.train %>% model.matrix(lcavol ~ ., data = .)
X.train.named <- model.matrix(lcavol ~ ., data = prostate.train )
X.test.named <- model.matrix(lcavol ~ ., data = prostate.test )


val.errors <- rep(NA, model.n)
train.errors <- rep(NA, model.n)
for (i in 1:model.n) {
    coefi <- coef(regfit.full, id = i)

    pred <-  X.train.named[, names(coefi)] %*% coefi
    train.errors[i] <- MSE(y.train - pred)

    pred <-  X.test.named[, names(coefi)] %*% coefi
    val.errors[i] <- MSE(y.test - pred)
}
plot(train.errors, ylab = "MSE", pch = 19, type = "o")
points(val.errors, pch = 19, type = "b", col="blue")
legend("topright",
       legend = c("Training", "Validation"),
       col = c("black", "blue"),
       pch = 19)
```

[2]Example taken from https://lagunita.stanford.edu/c4x/HumanitiesScience/StatLearning/asset/ch6.html



Checking all possible models is computationally very hard. *Forward selection* is a greedy approach that adds one variable at a time, using the AIC criterion. If AIC falls, the variable is added.

```
ols.0 <- lm(lcavol~1 ,data = prostate.train)
model.scope <- list(upper=ols.1, lower=ols.0)
step(ols.0, scope=model.scope, direction='forward', trace = TRUE)
```

```
## Start:  AIC=30.1
## lcavol ~ 1
##
##           Df Sum of Sq     RSS     AIC
## + lpsa     1    54.776  47.130 -19.570
## + lcp      1    48.805  53.101 -11.578
## + svi      1    35.829  66.077   3.071
## + pgg45    1    23.789  78.117  14.285
## + gleason  1    18.529  83.377  18.651
## + lweight  1     9.186  92.720  25.768
## + age      1     8.354  93.552  26.366
## <none>                 101.906  30.097
## + lbph     1     0.407 101.499  31.829
##
## Step:  AIC=-19.57
## lcavol ~ lpsa
##
##           Df Sum of Sq    RSS     AIC
## + lcp      1   14.8895 32.240 -43.009
## + svi      1    5.0373 42.093 -25.143
## + gleason  1    3.5500 43.580 -22.817
## + pgg45    1    3.0503 44.080 -22.053
## + lbph     1    1.8389 45.291 -20.236
## + age      1    1.5329 45.597 -19.785
## <none>                 47.130 -19.570
## + lweight  1    0.4106 46.719 -18.156
##
## Step:  AIC=-43.01
## lcavol ~ lpsa + lcp
##
##           Df Sum of Sq    RSS     AIC
## <none>                 32.240 -43.009
## + age      1   0.92315 31.317 -42.955
```

```
## + pgg45    1   0.29594 31.944 -41.627
## + gleason  1   0.21500 32.025 -41.457
## + lbph     1   0.13904 32.101 -41.298
## + lweight  1   0.05504 32.185 -41.123
## + svi      1   0.02069 32.220 -41.052
```

```
##
## Call:
## lm(formula = lcavol ~ lpsa + lcp, data = prostate.train)
##
## Coefficients:
## (Intercept)         lpsa          lcp
##     0.08798      0.53369      0.38879
```

We now learn a linear predictor on the `spam` data using, with least squares loss, and train-test validation.

```
# train the predictor
ols.2 <- lm(spam~., data = spam.train.dummy)

# make in-sample predictions
.predictions.train <- predict(ols.2) > 0.5
# inspect the confusion matrix
(confusion.train <- table(prediction=.predictions.train, truth=spam.train.dummy$spam))
```

```
##           truth
## prediction    0    1
##      FALSE 1748  234
##      TRUE    84  956
```

```
# compute the train (in sample) misclassification
missclassification(confusion.train)
```

```
## [1] 0.1052283
```

```
# make out-of-sample prediction
.predictions.test <- predict(ols.2, newdata = spam.test.dummy) > 0.5
# inspect the confusion matrix
(confusion.test <- table(prediction=.predictions.test, truth=spam.test.dummy$spam))
```

```
##           truth
## prediction   0   1
##      FALSE 917 129
##      TRUE   39 494
```

```
# compute the train (in sample) misclassification
missclassification(confusion.test)
```

```
## [1] 0.1063965
```

The `glmnet` package is an excellent package that provides ridge, lasso, and elastic net regularization, for all GLMs, so for linear models in particular.

```
suppressMessages(library(glmnet))
ridge.2 <- glmnet(x=X.train, y=y.train, family = 'gaussian', alpha = 0)

# Train error:
MSE( predict(ridge.2, newx =X.train)- y.train)
```

```
## [1] 1.006028
```

```
# Test error:
MSE( predict(ridge.2, newx = X.test)- y.test)
```

```
## [1] 0.7678264
```

Things to note:

- The `alpha=0` parameters tells R to do ridge regression. Setting $alpha = 1$ will do lasso, and any other value, with return an elastic net with appropriate weights.
- The 'family='gaussian' argument tells R to fit a linear model, with least squares loss.

We now use the lasso penalty.

```
lasso.1 <- glmnet(x=X.train, y=y.train, , family='gaussian', alpha = 1)

# Train error:
MSE( predict(lasso.1, newx =X.train)- y.train)
```

```
## [1] 0.5525279
```

```
# Test error:
MSE( predict(lasso.1, newx = X.test)- y.test)
```

```
## [1] 0.5211263
```

We now use `glmnet` for classification.

```
logistic.2 <- cv.glmnet(x=X.train.spam, y=y.train.spam, family = "binomial", alpha = 0)
```

Things to note:

- We used `cv.glmnet` to do an automatic search for the optimal level of regularization (the `lambda` argument in `glmnet`).
- We set `alpha=0` for ridge regression.

```
# Train confusion matrix:
.predictions.train <- predict(logistic.2, newx = X.train.spam, type = 'class')
(confusion.train <- table(prediction=.predictions.train, truth=spam.train$spam))
```

```
##           truth
## prediction email spam
##      email  1753  178
##      spam     79 1012
```

```
# Train misclassification error
missclassification(confusion.train)
```

```
## [1] 0.08504302
```

```
# Test confusion matrix:
.predictions.test <- predict(logistic.2, newx = X.test.spam, type='class')
(confusion.test <- table(prediction=.predictions.test, truth=y.test.spam))
```

```
##           truth
## prediction email spam
##      email   915   93
##      spam     41  530
```

```
# Test misclassification error:
missclassification(confusion.test)
```

```
## [1] 0.08486384
```

9.2.2 SVM

A support vector machine (SVM) is a linear model with a particular loss function known as a *hinge loss*. We learn an SVM with the `svm` function from the **e1071** package, which is merely a wrapper for the **libsvm** C library, which is the most popular implementation of SVM today.

```
library(e1071)
svm.1 <- svm(spam~., data = spam.train)

# Train confusion matrix:
.predictions.train <- predict(svm.1)
(confusion.train <- table(prediction=.predictions.train, truth=spam.train$spam))
```

```
##           truth
## prediction email spam
##      email  1775   98
##      spam     57 1092
```

```
missclassification(confusion.train)
```

```
## [1] 0.05129054
```

```
# Test confusion matrix:
.predictions.test <- predict(svm.1, newdata = spam.test)
(confusion.test <- table(prediction=.predictions.test, truth=spam.test$spam))
```

```
##           truth
## prediction email spam
##      email   920   77
##      spam     36  546
```

```
missclassification(confusion.test)
```

```
## [1] 0.07156428
```

We can also use SVM for regression.

```
svm.2 <- svm(lcavol~., data = prostate.train)

# Train error:
MSE( predict(svm.2)- prostate.train$lcavol)
```

```
## [1] 0.3336868
```

```
# Test error:
MSE( predict(svm.2, newdata = prostate.test)- prostate.test$lcavol)
```

```
## [1] 0.5633183
```

9.2.3 Neural Nets

Neural nets (non deep) can be fitted, for example, with the `nnet` function in the **nnet** package. We start with a nnet regression.

```
library(nnet)
nnet.1 <- nnet(lcavol~., size=20, data=prostate.train, rang = 0.1, decay = 5e-4, maxit = 1000, trace=FALSE)

# Train error:
MSE( predict(nnet.1)- prostate.train$lcavol)
```

```
## [1] 1.174269
```

```
# Test error:
MSE( predict(nnet.1, newdata = prostate.test)- prostate.test$lcavol)
```

```
## [1] 1.476749
```

And nnet classification.

```
nnet.2 <- nnet(spam~., size=5, data=spam.train, rang = 0.1, decay = 5e-4, maxit = 1000, trace=FALSE)

# Train confusion matrix:
.predictions.train <- predict(nnet.2, type='class')
(confusion.train <- table(prediction=.predictions.train, truth=spam.train$spam))
```

```
##           truth
## prediction email spam
##      email  1779   59
##      spam     53 1131
```

```
missclassification(confusion.train)
```

```
## [1] 0.03706155
```

```
# Test confusion matrix:
.predictions.test <- predict(nnet.2, newdata = spam.test, type='class')
(confusion.test <- table(prediction=.predictions.test, truth=spam.test$spam))
```

```
##           truth
## prediction email spam
##      email   915   58
##      spam     41  565
```

```
missclassification(confusion.test)
```

```
## [1] 0.06269791
```

9.2.4 Classification and Regression Trees (CART)

A CART, is not a linear model. It partitions the feature space $\mathcal{X}$, thus creating a set of if-then rules for prediction or classification. This view clarifies the name of the function `rpart`, which *recursively partitions* the feature space.

We start with a regression tree.

```
library(rpart)
tree.1 <- rpart(lcavol~., data=prostate.train)

# Train error:
MSE( predict(tree.1)- prostate.train$lcavol)
```

```
## [1] 0.4909568
```

```
# Test error:
MSE( predict(tree.1, newdata = prostate.test)- prostate.test$lcavol)
```

```
## [1] 0.5623316
```

Tree are very prone to overfitting. To avoid this, we reduce a tree's complexity by *pruning* it. This is done with the `prune` function.

We now fit a classification tree.

```
tree.2 <- rpart(spam~., data=spam.train)

# Train confusion matrix:
.predictions.train <- predict(tree.2, type='class')
(confusion.train <- table(prediction=.predictions.train, truth=spam.train$spam))
```

```
##           truth
## prediction email spam
##      email  1755  216
##      spam     77  974
```

```
missclassification(confusion.train)
```

```
## [1] 0.09695566
```

```
# Test confusion matrix:
.predictions.test <- predict(tree.2, newdata = spam.test, type='class')
(confusion.test <- table(prediction=.predictions.test, truth=spam.test$spam))
```

```
##           truth
## prediction email spam
##      email   909  137
##      spam     47  486
```

```
missclassification(confusion.test)
```

```
## [1] 0.1165294
```

9.2.5 K-nearest neighbour (KNN)

KNN is not an ERM problem. For completeness, we still show how to fit such a hypothesis.

```
library(class)
knn.1 <- knn(train = X.train.spam, test = X.test.spam, cl =y.train.spam, k = 1)

# Test confusion matrix:
.predictions.test <- knn.1
(confusion.test <- table(prediction=.predictions.test, truth=spam.test$spam))
```

```
##           truth
## prediction email spam
##      email   814  153
##      spam    142  470
```

```
missclassification(confusion.test)
```

```
## [1] 0.1868271
```

9.2.6 Linear Discriminant Analysis (LDA)

LDA is equivalent to least squares classification 9.2.1. There are, however, some dedicated functions to fit it.

```
library(MASS)
lda.1 <- lda(spam~., spam.train)

# Train confusion matrix:
.predictions.train <- predict(lda.1)$class
(confusion.train <- table(prediction=.predictions.train, truth=spam.train$spam))
```

```
##           truth
## prediction email spam
##      email  1748  234
##      spam     84  956
```

```
missclassification(confusion.train)
```

```
## [1] 0.1052283
```

```
# Test confusion matrix:
.predictions.test <- predict(lda.1, newdata = spam.test)$class
(confusion.test <- table(prediction=.predictions.test, truth=spam.test$spam))
```

```
##           truth
## prediction email spam
##      email   917  125
##      spam     39  498
```

```
missclassification(confusion.test)
```

```
## [1] 0.1038632
```

9.2.7 Naive Bayes

A Naive-Bayes classifier is also not part of the ERM framework. It is, however, very popular, so we present it.

```
library(e1071)
nb.1 <- naiveBayes(spam~., data = spam.train)

# Train confusion matrix:
.predictions.train <- predict(nb.1, newdata = spam.train)
(confusion.train <- table(prediction=.predictions.train, truth=spam.train$spam))
```

```
##           truth
## prediction email spam
##      email  1068   74
##      spam    764 1116
```

```
missclassification(confusion.train)
```

```
## [1] 0.2772998
```

```
# Test confusion matrix:
.predictions.test <- predict(nb.1, newdata = spam.test)
(confusion.test <- table(prediction=.predictions.test, truth=spam.test$spam))
```

```
##           truth
## prediction email spam
##      email   548   27
##      spam    408  596
```

```
missclassification(confusion.test)
```

```
## [1] 0.2754908
```

9.3 Bibliographic Notes

The ultimate reference on (statistical) machine learning is Friedman et al. (2001). For a softer introduction, see James et al. (2013). A statistician will also like Ripley (2007). For an R oriented view see Lantz (2013). For a very algorithmic view, see the seminal Leskovec et al. (2014) or Conway and White (2012). For a much more theoretical reference, see Mohri et al. (2012), Vapnik (2013), Shalev-Shwartz and Ben-David (2014). Terminology taken from Sammut and Webb (2011). For a review of resampling based unbiased risk estimation (i.e. cross validation) see the exceptional review of Arlot et al. (2010).

Chapter 10

Unsupervised Learning

This chapter deals with machine learning problems which are unsupervised. This means the machine has access to a set of inputs, x, but the desired outcome, y is not available. Clearly, learning a relation between inputs and outcomes is impossible, but there are still a lot of problems of interest. In particular, we may want to find a compact representation of the inputs, be it for visualization of further processing. This is the problem of *dimensionality reduction*. For the same reasons we may want to group similar inputs. This is the problem of *clustering*.

In the statistical terminology, and with some exceptions, this chapter can be thought of as multivariate **exploratory** statistics. For multivariate **inference**, see Chapter 8.

10.1 Dimensionality Reduction

Example 10.1. Consider the heights and weights of a sample of individuals. The data may seemingly reside in 2 dimensions but given the height, we have a pretty good guess of a persons weight, and vice versa. We can thus state that heights and weights are not really two dimensional, but roughly lay on a 1 dimensional subspace of $\mathbb{R}^2$.

Example 10.2. Consider the correctness of the answers to a questionnaire with p questions. The data may seemingly reside in a p dimensional space, but assuming there is a thing as "skill", then given the correctness of a person's reply to a subset of questions, we have a good idea how he scores on the rest. Put differently, we don't really need a 200 question questionnaire– 100 is more than enough. If skill is indeed a one dimensional quality, then the questionnaire data should organize around a single line in the p dimensional cube.

Example 10.3. Consider n microphones recording an individual. The digitized recording consists of p samples. Are the recordings really a shapeless cloud of n points in $\mathbb{R}^p$? Since they all record the same sound, one would expect the n p-dimensional points to arrange around the source sound bit: a point in $\mathbb{R}^p$. If microphones have different distances to the source, volumes may differ. We would thus expect the n points to arrange about a line that ends at the source.

10.1.1 Principal Component Analysis

Principal Component Analysis (PCA) is such a basic technique, it has been rediscovered and renamed independently in many fields. It can be found under the names of Discrete Karhunen–Loève Transform; Hotteling Transform; Proper Orthogonal Decomposition; Eckart–Young Theorem; Schmidt–Mirsky Theorem; Empirical Orthogonal Functions; Empirical Eigenfunction Decomposition; Empirical Component Analysis; Quasi-Harmonic Modes; Spectral Decomposition; Empirical Modal Analysis, and possibly more[1]. The many names are quite interesting as they offer an insight into the different problems that led to PCA's (re)discovery.

Return to the BMI problem in Example 10.1. Assume you wish to give each individual a "size score", that is a **linear** combination of height and weight: PCA does just that. It returns the linear combination that has the largest variability, i.e., the combination which best distinguishes between individuals.

[1] http://en.wikipedia.org/wiki/Principal_component_analysis

The variance maximizing motivation above was the one that guided Hotelling (1933). But 30 years before him, Pearson (1901) derived the same procedure with a different motivation in mind. Pearson was also trying to give each individual a score. He did not care about variance maximization, however. He simply wanted a small set of coordinates in some (linear) space that approximates the original data well. Before we proceed, we give an example to fix ideas. Consider the crime rate data in `USArrests`, which encodes reported murder events, assaults, rapes, and the urban population of each american state.

```
head(USArrests)
```

```
##            Murder Assault UrbanPop Rape
## Alabama      13.2     236       58 21.2
## Alaska       10.0     263       48 44.5
## Arizona       8.1     294       80 31.0
## Arkansas      8.8     190       50 19.5
## California    9.0     276       91 40.6
## Colorado      7.9     204       78 38.7
```

Following Hotelling's motivation, we may want to given each state a "crimilality score". PCA returns the sequence of $1, \ldots, 4$ scores that best separate between states.

```
USArrests.1 <- USArrests[,-3] %>% scale # note the scaling, which is required by some
pca.1 <- prcomp(USArrests.1, scale = TRUE)
pca.1
```

```
## Standard deviations:
## [1] 1.5357670 0.6767949 0.4282154
##
## Rotation:
##                PC1        PC2        PC3
## Murder  -0.5826006  0.5339532 -0.6127565
## Assault -0.6079818  0.2140236  0.7645600
## Rape    -0.5393836 -0.8179779 -0.1999436
```

Things to note:

- Distinguishing between states, i.e., finding the variance maximizing scores, should be indifferent to the **average** of each variable. We also don't want the score to be sensitive to the measurement **scale**. We thus perform PCA in the z-score scale of each variable, obtained with the `scale` function.
- PCA is performed with the `prcomp` function. It returns the contribution (weight) of the original variables, to the new crimeness score.
  These weights are called the *loadings*.
- The number of possible scores, is the same as the number of original variables in the data.
- The new scores are called the *principal components*, labeled `PC1`,…,`PC4` in our output.
- The loadings on PC1 tell us that the best separation between states is along the average crime rate. Why is this? Because all the 3 crime variables have a similar loading on PC1.
- The other PCs are slightly harder to interpret, but it is an interesting exercise.

If we now represent each state, not with its original 4 variables, but only with the first 2 PCs (for example), we have reduced the dimensionality of the data.

10.1.2 Preliminaries

Before presenting methods other than PCA, we need some terminology.

- **Variable**: A.k.a. *dimension*, or *feature*, or *column* for reasons that will be obvious in the next item.
- **Data**: A.k.a. *sample*, *observations*. Will typically consist of n, p dimensional vectors. We typically denote the data as a $n \times p$ matrix X.
- **Manifold**: A generalization of a linear space, which is regular enough so that, **locally**, it has all the properties of a linear space. We will denote an arbitrary manifold by $\mathcal{M}$, and by $\mathcal{M}_q$ a q dimensional[2] manifold.

[2] You are probably used to thinking of the **dimension** of linear spaces. We will not rigorously define what is the dimension of a manifold,

- **Embedding**: Informally speaking: a "shape preserving" mapping of a space into another.
- **Linear Embedding**: An embedding done via a linear operation (thus representable by a matrix).
- **Generative Model**: Known to statisticians as the **sampling distribution**. The assumed stochastic process that generated the observed X.

There are many motivations for dimensionality reduction:

1. **Scoring**: Give each observation an interpretable, simple score (Hotelling's motivation).
2. **Latent structure**: Recover unobservables from indirect measurements. E.g: Blind signal reconstruction, CT scan, cryo-electron microscopy, etc.
3. **Signal to Noise**: Denoise measurements before further processing like clustering, supervised learning, etc.
4. **Compression**: Save on RAM ,CPU, and communication when operating on a lower dimensional representation of the data.

10.1.3 Latent Variable Approaches

All generative approaches to dimensionality reduction will include some unobserved set of variables, which we can try to recover from the observable X. The unobservable variables will typically have a lower dimension than the observables, thus, dimension is reduced. We start with the simplest case of linear Factor Analysis.

10.1.3.1 Factor Analysis (FA)

FA originates from the psychometric literature. We thus revisit the IQ (actually g-factor[3]) Example~10.2:

Example 10.4. Assume n respondents answer p quantitative questions: $x_i \in \mathbb{R}^p, i = 1, \ldots, n$. Also assume, their responses are some linear function $A \in \mathbb{R}^p$ of a single personality attribute, s_i. We can think of s_i as the subject's "intelligence". We thus have

$$x_i = As_i + \varepsilon_i \tag{10.1}$$

And in matrix notation, for $q < p$ latent attributes:

$$X = SA + \varepsilon, \tag{10.2}$$

where A is the $q \times p$ matrix of factor loadings, and S the $n \times q$ matrix of latent personality traits. In our particular example where $q = 1$, the problem is to recover the unobservable intelligence scores, $s_1, \ldots, s_n$, from the observed answers X.

We may try to estimate SA by assuming some distribution on S and ε and apply maximum likelihood. Under standard assumptions on the distribution of S and ε, recovering S from $\widehat{SA}$ is still impossible as there are infinitely many such solutions. In the statistical parlance we say the problem is *non identifiable*, and in the applied mathematics parlance we say the problem is *ill posed*. To see this, consider an orthogonal *rotation* matrix R ($R'R = I$). For each such R: \$ S A = A R' R S = $S^* A^*$ \$. While both solve Eq.(10.2), A and A^* may have very different interpretations. This is why many researchers find FA an unsatisfactory inference tool.
Remark. The non-uniqueness (non-identifiability) of the FA solution under variable rotation is never mentioned in the PCA context. Why is this? This is because the methods solve different problems. The reason the solution to PCA is well defined is that PCA does not seek a single S but rather a **sequence** of S_q with dimensions growing from $q = 1$ to $q = p$.
Remark. In classical FA in Eq.(10.2) is clearly an embedding to a linear space. The one spanned by S. Under the classical probabilistic assumptions on S and ε the embedding itself is also linear, and is sometimes solved with PCA. Being a generative model, there is no restriction for the embedding to be linear, and there certainly exists sets of assumptions for which the FA embedding is non linear.

but you may think of it as the number of free coordinates needed to navigate along the manifold.
[3] https://en.wikipedia.org/wiki/G_factor_(psychometrics)

The FA terminology is slightly different than PCA:

- **Factors**: The unobserved attributes S. Not to be confused with the *principal components* in the context of PCA.
- **Loading**: The A matrix; the contribution of each factor to the observed X.
- **Rotation**: An arbitrary orthogonal re-combination of the factors, S, and loadings, A, which changes the interpretation of the result.

The FA literature does offer several heuristics to "fix" the solution of the FA. These are known as *rotations*, and go under the names of *Varimax*, *Quartimax*, *Equimax*, *Oblimin*, *Promax*, and possibly others.

10.1.3.2 Independent Component Analysis (ICA)

Like FA, *independent compoent analysis* (ICA) is a family of latent space models, thus, a *meta-method*. It assumes data is generated as some function of the latent variables S. In many cases this function is assumed to be linear in S so that ICA is compared, if not confused, with PCA and even more so with FA.

The fundamental idea of ICA is that S has a joint distribution of **non-Gaussian**, **independent** variables. This independence assumption, solves the the non-uniqueness of S in FA.

Being a generative model, estimation of S can then be done using maximum likelihood, or other estimation principles.

ICA is a popular technique in signal processing, where A is actually the signal, such as sound in Example 10.3. Recovering A is thus recovering the original signals mixing in the recorded X.

10.1.4 Purely Algorithmic Approaches

We now discuss dimensionality reduction approaches that are not stated via their generative model, but rather, directly as an algorithm. This does not mean that they cannot be cast via their generative model, but rather they were not motivated as such.

10.1.4.1 Multidimensional Scaling (MDS)

MDS can be thought of as a variation on PCA, that begins with a distance graph[4]}.

MDS aims at embedding a graph of distances, while preserving the original distances. Basic results in graph/network theory (Graham, 1988) suggest that the geometry of a graph cannot be preserved when embedding it into lower dimensions. The different types of MDSs, such as *Classical MDS*, and *Sammon Mappings*, differ in the *stress function* penalizing for geometric distortion.

10.1.4.2 Local Multidimensional Scaling (Local MDS)

Example 10.5. Consider data of coordinates on the globe. At short distances, constructing a dissimilarity graph with Euclidean distances will capture the true distance between points. At long distances, however, the Euclidean distances as grossly inappropriate. A more extreme example is coordinates on the brain's cerebral cortex. Being a highly folded surface, the Euclidean distance between points is far from the true geodesic distances along the cortex's surface[5].

Local MDS is aimed at solving the case where we don't know how to properly measure distances. It is an algorithm that compounds both the construction of the dissimilarity graph, and the embedding. The solution of local MDS, as the name suggests, rests on the computation of *local* distances, where the Euclidean assumption may still be plausible, and then aggregate many such local distances, before calling upon regular MDS for the embedding.

Because local MDS ends with a regular MDS, it can be seen as a non-linear embedding into a linear $\mathcal{M}$.

[4]The term Graph is typically used in this context instead of Network. But a graph allows only yes/no relations, while a network, which is a weighted graph, allows a continuous measure of similarity (or dissimilarity). *Network* is thus more appropriate than *graph*.

[5]Then again, it is possible that the true distances are the white matter fibers connecting going within the cortex, in which case, Euclidean distances are more appropriate than geodesic distances. We put that aside for now.

Local MDS is not popular. Why is this? Because it makes no sense: If we believe the points reside in a non-Euclidean space, thus motivating the use of geodesic distances, why would we want to wrap up with regular MDS, which embeds in a linear space?!

10.1.4.3 Isometric Feature Mapping (IsoMap)

Like localMDS, only that the embedding, and not only the computation of the distances, is local.

10.1.4.4 Local Linear Embedding (LLE)

Very similar to IsoMap 10.1.4.3.

10.1.4.5 Kernel PCA

TODO

10.1.4.6 Simplified Component Technique LASSO (SCoTLASS)

TODO

10.1.4.7 Sparse Principal Component Analysis (sPCA)

TODO

10.1.4.8 Sparse kernel principal component analysis (skPCA)

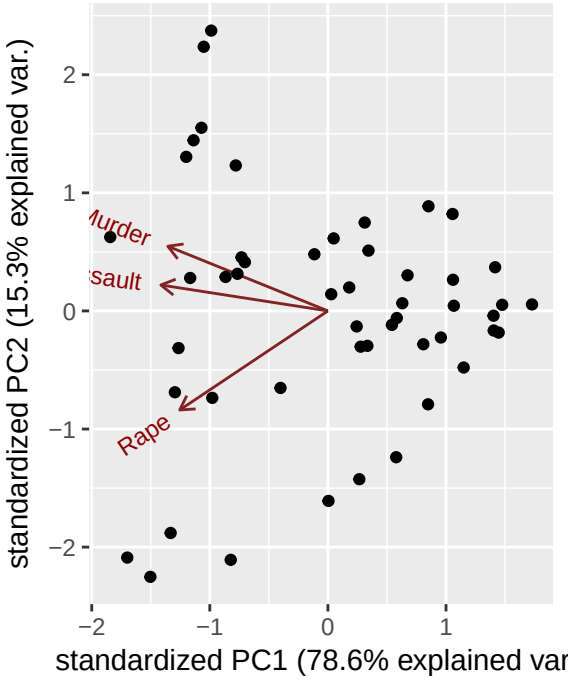
TODO

10.1.5 Dimensionality Reduction in R

10.1.5.1 PCA

We already saw the basics of PCA in 10.1.5.1. The fitting is done with the `procomp` function. The *bi-plot* is a useful way to visualize the output of PCA.

```
library(devtools)
# install_github("vqv/ggbiplot")
ggbiplot::ggbiplot(pca.1)
```

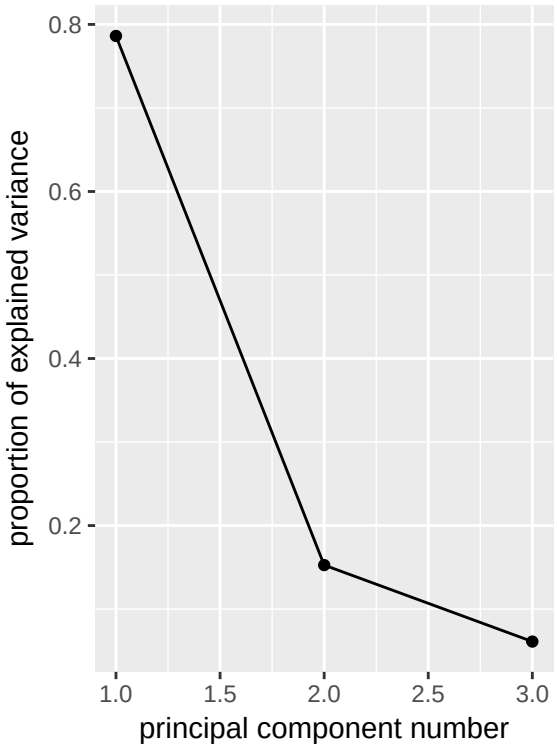


Things to note:

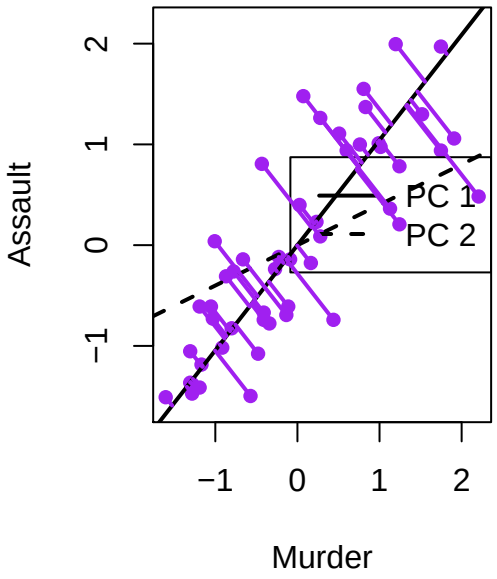
- The bi-plot plots each data point along its PCs.
- We used the `ggbiplot` function from the **ggbiplot** (available from github, but not from CRAN), because it has a nicer output than `stats::biplot`.
- The bi-plot also plots the loadings as arrows. The coordinates of the arrows belong to the weight of each of the original variables in each PC. For example, the x-value of each arrow is the loadings on the first PC (on the x-axis). Since the weights of Murder, Assault, and Rape are almost the same, and larger then UrbanPop, we conclude that PC1 captures the average crime rate in each state.

The *scree plot* depicts the quality of the approximation of X as q grows. This is depicted using the proportion of variability in X that is removed by each added PC. It is customary to choose q as the first PC that has a relative low contribution to the approximation of X.

```
ggbiplot::ggscreeplot(pca.1)
```



See how the first PC captures the variability in the Assault levels and Murder levels, with a single score.



More implementations of PCA:

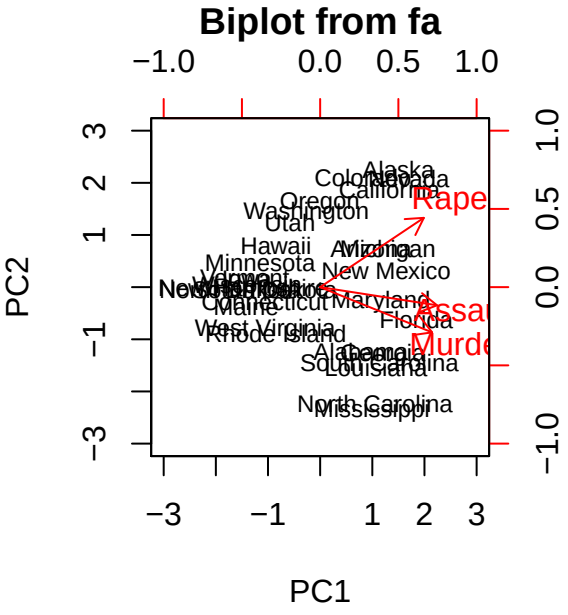
```
# FAST solutions:
gmodels::fast.prcomp()

# More detail in output:
FactoMineR::PCA()

# For flexibility in algorithms and visualization:
ade4::dudi.pca()

# Another one...
amap::acp()
```

10.1.5.2 FA

```
fa.1 <- psych::principal(USArrests.1, nfactors = 2, rotate = "none")
fa.1
```

```
## Principal Components Analysis
## Call: psych::principal(r = USArrests.1, nfactors = 2, rotate = "none")
## Standardized loadings (pattern matrix) based upon correlation matrix
##          PC1   PC2   h2     u2 com
## Murder  0.89 -0.36 0.93 0.0688 1.3
## Assault 0.93 -0.14 0.89 0.1072 1.0
## Rape    0.83  0.55 0.99 0.0073 1.7
##
##                        PC1  PC2
## SS loadings           2.36 0.46
## Proportion Var        0.79 0.15
## Cumulative Var        0.79 0.94
## Proportion Explained  0.84 0.16
## Cumulative Proportion 0.84 1.00
##
## Mean item complexity =  1.4
## Test of the hypothesis that 2 components are sufficient.
##
## The root mean square of the residuals (RMSR) is  0.05
##  with the empirical chi square  0.87  with prob <  NA
##
## Fit based upon off diagonal values = 0.99
```

```
biplot(fa.1, labels =  rownames(USArrests.1))
```




```
# Numeric comparison with PCA:
fa.1$loadings
```

```
##
## Loadings:
##         PC1    PC2
## Murder   0.895 -0.361
## Assault  0.934 -0.145
## Rape     0.828  0.554
```

```
##
##                    PC1   PC2
## SS loadings    2.359 0.458
## Proportion Var 0.786 0.153
## Cumulative Var 0.786 0.939
```

```
pca.1$rotation
```

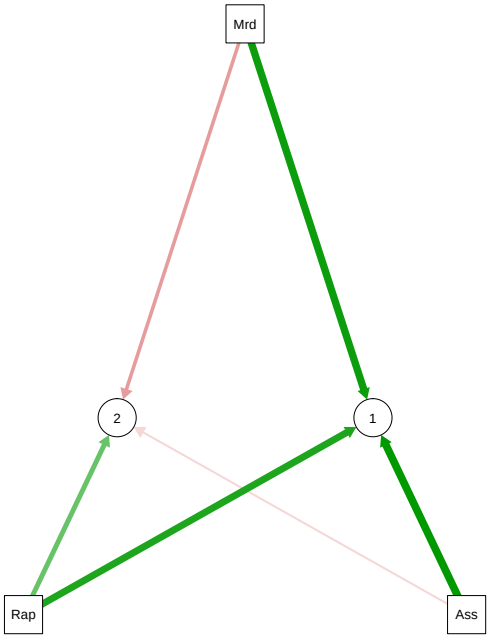
```
##                PC1        PC2        PC3
## Murder  -0.5826006  0.5339532 -0.6127565
## Assault -0.6079818  0.2140236  0.7645600
## Rape    -0.5393836 -0.8179779 -0.1999436
```

Things to note:

- We perform FA with the `psych::principal` function.
- The first factor (`fa.1$loadings`) has different weights than the first PC (`pca.1$rotation`) because of normalization. They are the same, however, in that the first PC, and the first factor, capture average crime levels.

Graphical model fans will like the following plot, where the contribution of each variable to each factor is encoded in the width of the arrow.

```
# Graph comparison: loadings encoded in colors
qgraph::qgraph(fa.1)
```



Let's add a rotation (Varimax), and note that the rotation has indeed changed the loadings of the variables, thus the interpretation of the factors.

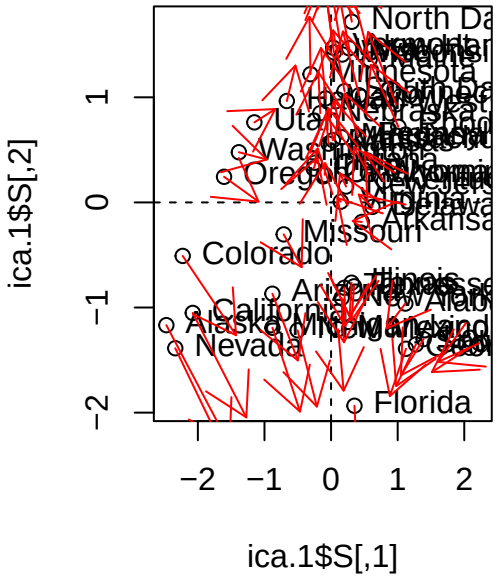
```
fa.2 <- psych::principal(USArrests.1, nfactors = 2, rotate = "varimax")

fa.2$loadings
```

```
##
## Loadings:
##          RC1   RC2
## Murder  0.930 0.257
## Assault 0.829 0.453
## Rape    0.321 0.943
##
##                  RC1   RC2
## SS loadings    1.656 1.160
```

```
## Proportion Var 0.552 0.387
## Cumulative Var 0.552 0.939
```

10.1.5.3 ICA

```
ica.1 <- fastICA::fastICA(USArrests.1, n.com=2) # Also performs projection pursuit

plot(ica.1$S)
abline(h=0, v=0, lty=2)
text(ica.1$S, pos = 4, labels = rownames(USArrests.1))

# Compare with two PCA (first two PCs):
arrows(x0 = ica.1$S[,1], y0 = ica.1$S[,2], x1 = pca.1$x[,2], y1 = pca.1$x[,1], col='red', pch=19, cex=0.5)
```



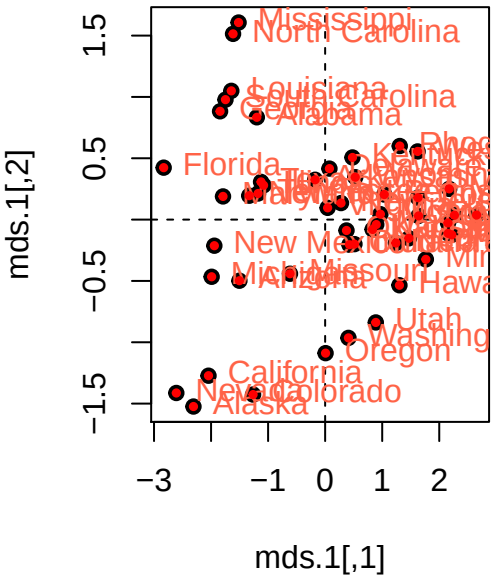
Things to note:

- ICA is fitted with `fastICA::fastICA`.
- The ICA components are very different than the PCA components.

10.1.5.4 MDS

Classical MDS, also compared with PCA.

```
# We first need a dissimarity matrix/graph:
state.disimilarity <- dist(USArrests.1)

mds.1 <- cmdscale(state.disimilarity)

plot(mds.1, pch = 19)
abline(h=0, v=0, lty=2)
USArrests.2 <- USArrests[,1:2] %>%  scale
text(mds.1, pos = 4, labels = rownames(USArrests.2), col = 'tomato')

# Compare with two PCA (first two PCs):
points(pca.1$x[,1:2], col='red', pch=19, cex=0.5)
```

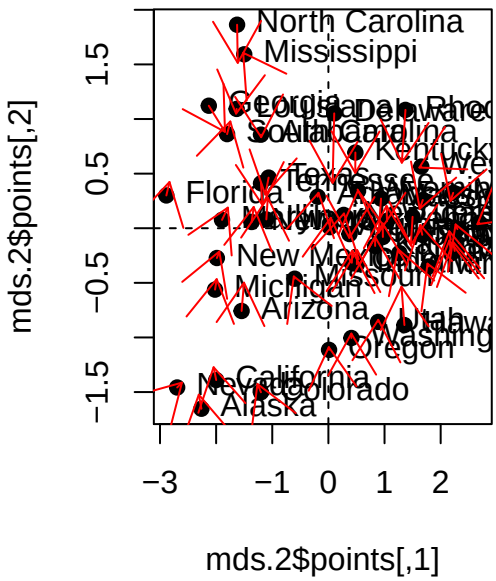


Things to note:

- For MDS, we first compute a dissimilarity graph with `dist`, and then learn the embedding with `cmdscale`.
- As previously stated, the embedding of PCA is the same as classical MDS with Euclidean distances.
- See the `cluster::daisy` function for more dissimilarity measures.

Let's try other strain functions for MDS, like Sammon's strain, and compare it with the PCs.

```
mds.2 <- MASS::sammon(state.disimilarity, trace = FALSE)
plot(mds.2$points, pch = 19)
abline(h=0, v=0, lty=2)
text(mds.2$points, pos = 4, labels = rownames(USArrests.2))

# Compare with two PCA (first two PCs):
arrows(
  x0 = mds.2$points[,1], y0 = mds.2$points[,2],
  x1 = pca.1$x[,1], y1 = pca.1$x[,2],
  col='red', pch=19, cex=0.5)
```



Things to note:

- `MASS::sammon` does the fitting.
- The embedding returned by the Sammon strain is different than that of the first two PCs.

10.1.5.5 Sparse PCA

```
# Compute similarity graph
state.similarity <- MASS::cov.rob(USArrests.1)$cov

spca1 <- elasticnet::spca(state.similarity, K=2, type="Gram", sparse="penalty", trace=FALSE, para=c(0.06,0.1
spca1$loadings
```

```
##                PC1 PC2
## Murder  -0.7143088   0
## Assault -0.2023409  -1
## Rape    -0.6699411   0
```

10.1.5.6 Kernel PCA

```
kernlab::kpca()
```

10.2 Clustering

Example 10.6. Consider the tagging of your friends' pictures on Facebook. If you tagged some pictures, Facebook may try to use a supervised approach to automatically label photos. If you never tagged pictures, a supervised approach is impossible. It is still possible to group simiar pictures together.

Example 10.7. Consider the problem of spam detection. It would be nice if each user could label several thousands emails, to apply a supervised learning approach to spam detection. This is an unrealistic demand, so a pre-clustering stage is useful: the user only needs to tag a couple dozens of homogenous clusters, before solving the supervisedl learning problem.

In clustering problems, we seek to group observations that are similar.

There are many motivations for clustering:

1. **Understanding**: The most common use of clustering is probably as a an exploratory step, to identify homogeneous groups in the data.
2. **Dimensionality reduction**: Clustering may be seen as a method for dimensionality reduction. Unlike the approaches in the Dimensionality Reduction Section 10.1, it does not "compress" variables but rather observations. Each group of homogeneous observations may then be represented as a single prototypical observation of the group.
3. **Pre-Labelling**: Clustering may be performed as a pre-processing step for supervised learning, when labeling all the samples is impossible due to "budget" constraints, like in Example 10.7. This is sometimes known as *pre-clustering*.

Clustering, like dimensionality reduction, may rely on some latent variable generative model, or on purely algorithmic approaches.

10.2.1 Latent Variable Approaches

10.2.1.1 Finite Mixture

Example 10.8. Consider the distribution of heights in some population. Given the gender, heights have a nice bell shaped distribution. If genders have not been recorded, We can view it as a *latent*, i.e., unobservale, with $K = 2$ levels: males and females.

A *finite mixture* is the marginal distribution of K distinct classes, when the class variable is *latent*. This is useful for clustering since we can assume the number of classes, K, and the distribution of each class. We can then use maximum likelihood estimation to fit the mixture distribution and assign observations to the most probable class.

10.2.2 Purely Algorithmic Approaches

10.2.2.1 K-means

The *K-means* algorithm is possibly the most popular clustering algorithm. The goal behind K-means clustering algorithm is finding a representative point for each of K clusters, and assign each data point to one of these clusters. As each cluster has a representative point, this is also a *prototype method* The clusters are defined so that they minimize the average Euclidean distance between all points to the center of the cluster.

In K-means, the clusters are first defined, and then similarities computed. This is thus a *top-down* method.

K-means clustering requires the raw features X as inputs, and not only a similarity graph. This is evident when examining the algorithm below.

The k-means algorithm works as follows:

1. Choose the number of clusters K.
2. Arbitrarily assign points to clusters.
3. While clusters keep changing:
    1. Compute the cluster centers as the average of their points.
    2. Assign each point to its closest cluster center (in Euclidean distance).
4. Return Cluster assignments and means.

Remark. If are trained as a statistician, you may wonder- what population quantity is K-means actually estimating? The estimand of K-means is known as the *K principal points*. Principal points are points which are *self consistent*, i.e., they are the mean of their neighbourhood.

10.2.2.2 K-means++

K-means++ is a fast version of K-means thanks to a smart initialization.

10.2.2.3 K-medoids

If a Euclidean distance is inappropriate for a particular set of variables, or that robustness to corrupt observations is required, or that we wish to constrain the cluster centers to be actual observations, then the *K-Medoids* algorithm is an adaptation of K-means that allows this. It is also known under the name *partition around medoids* (PAM) clustering.

The k-medoids algorithm works as follows.

1. Given a dissimilarity graph.
2. Choose the number of clusters K.
3. Arbitrarily assign points to clusters.
4. While clusters keep changing:
    1. Within each cluster, set the center as the data point that minimizes the sum of distances to other points in the cluster.
    2. Assign each point to its closest cluster center.
5. Return Cluster assignments and centers.

10.2.2.4 Hirarchial Clustering

Hierarchical clustering algorithms take dissimilarity graphs as inputs. Hierarchical clustering is a class of greedy *graph-partitioning* algorithms. Being hierarchical by design, they have the attractive property that the evolution of the clustering can be presented with a *dendogram*, i.e., a tree plot.
A particular advantage of these methods is that they do not require an a-priori choice of the number of cluster (K).

Two main sub-classes of algorithms are *agglomerative*, and *divisive*.

Agglomerative clustering algorithms are **bottom-up** algorithm which build clusters by joining smaller clusters. To decide which clusters are joined at each iteration some measure of closeness between clusters is required.

- **Single Linkage**: Cluster distance is defined by the distance between the two **closest** members.

- **Complete Linkage**: Cluster distance is defined by the distance between the two **farthest** members.
- **Group Average**: Cluster distance is defined by the **average** distance between members.
- **Group Median**: Like Group Average, only using the median.

Divisive clustering algorithms are **top-down** algorithm which build clusters by splitting larger clusters.

10.2.2.5 Fuzzy Clustering

Can be thought of as a purely algorithmic view of the finite-mixture in Section 10.2.1.1.

10.2.3 Clustering in R

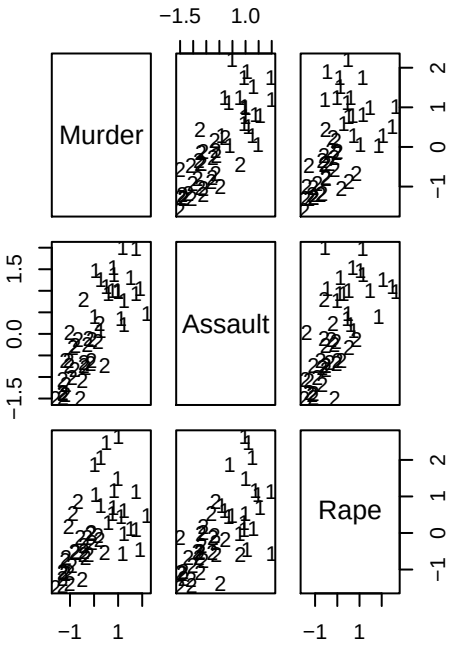
10.2.3.1 K-Means

The following code is an adaptation from David Hitchcock.

```
k <- 2
kmeans.1 <- stats::kmeans(USArrests.1, centers = k)
head(kmeans.1$cluster) # cluster asignments
```

```
##    Alabama     Alaska    Arizona   Arkansas California   Colorado
##          1          1          1          2          1          1
```

```
# Visualize using scatter plots of the original features
pairs(USArrests.1, panel=function(x,y) text(x,y,kmeans.1$cluster))
```



Things to note:

- The `stats::kmeans` function does the clustering.
- The cluster assignment is given in the `cluster` element of the `stats::kmeans` output.
- The visual inspection confirms that similar states have been assigned to the same cluster.

10.2.3.2 K-means ++

K-Means++ is a smart initialization for K-Means. The following code is taken from the r-help mailing list.

```
# Write my own K-means++ function.
kmpp <- function(X, k) {

  n <- nrow(X)
  C <- numeric(k)
  C[1] <- sample(1:n, 1)

  for (i in 2:k) {
    dm <- pracma::distmat(X, X[C, ])
    pr <- apply(dm, 1, min); pr[C] <- 0
    C[i] <- sample(1:n, 1, prob = pr)
  }

  kmeans(X, X[C, ])
}

# Examine output:
kmeans.2 <- kmpp(USArrests.1, k)
head(kmeans.2$cluster)
```

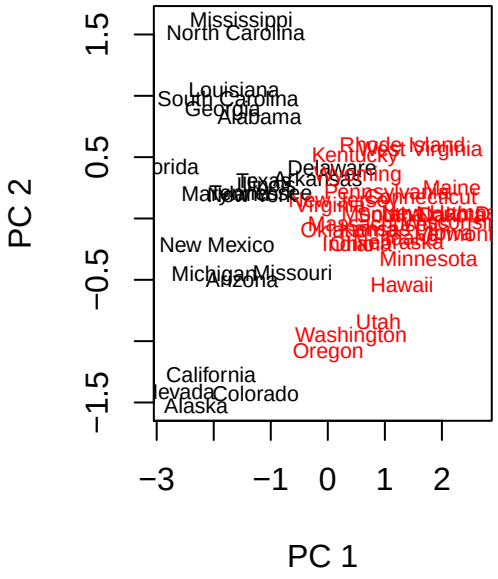
```
##    Alabama     Alaska    Arizona   Arkansas California   Colorado
##          1          1          1          2          1          1
```

10.2.3.3 K-medoids

```
state.disimilarity <- dist(USArrests.1)
kmed.1 <- cluster::pam(x= state.disimilarity, k=2)
head(kmed.1$clustering)
```

```
##    Alabama     Alaska    Arizona   Arkansas California   Colorado
##          1          1          1          1          1          1
```

```
plot(pca.1$x[,1], pca.1$x[,2], xlab="PC 1", ylab="PC 2", type ='n', lwd=2)
text(pca.1$x[,1], pca.1$x[,2], labels=rownames(USArrests.1), cex=0.7, lwd=2, col=kmed.1$cluster)
```



Things to note:

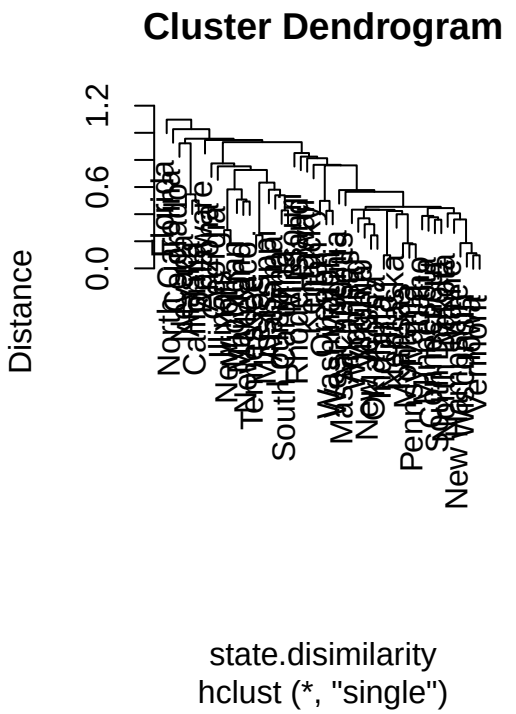
- K-medoids starts with the computation of a dissimilarity graph, done by the `dist` function.
- The clustering is done by the `cluster::pam` function.

- Inspecting the output confirms that similar states have been assigned to the same cluster.
- Many other similarity measures can be found in `proxy::dist()`.
- See `cluster::clara()` for a big-data implementation of PAM.

10.2.3.4 Hirarchial Clustering

We start with agglomerative clustering with single-linkage.

```
# Single linkage:
hirar.1 <- hclust(state.disimilarity, method='single')
plot(hirar.1, labels=rownames(USArrests.1), ylab="Distance")
```

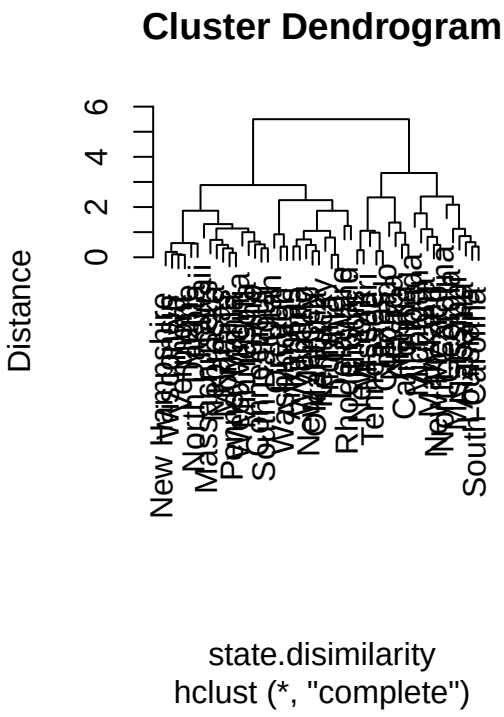


Things to note:

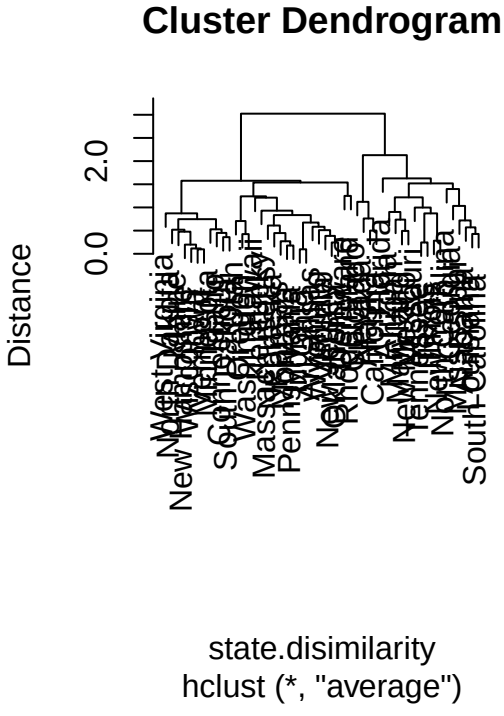
- The clustering is done with the `hclust` function.
- We choose the single-linkage distance using the `method='single'` argument.
- We did not need to a-priori specify the number of clusters, K.
- The `plot` function has a particular method for `hclust` class objects, and plots them as dendograms.

We not try other types of linkages, to verify that the indeed affect the clustering.

```
# Complete linkage:
hirar.2 <- hclust(state.disimilarity, method='complete')
plot(hirar.2, labels=rownames(USArrests.1), ylab="Distance")
```




```
# Average linkage:
hirar.3 <- hclust(state.disimilarity, method='average')
plot(hirar.3, labels=rownames(USArrests.1), ylab="Distance")
```



If we know how many clusters we want, we can use `cuttree` to get the class assignments.

```
# Fixing the number of clusters:
cut.2.2 <- cutree(hirar.2, k=2)
head(cut.2.2)   # printing the "clustering vector"
```

```
##    Alabama     Alaska    Arizona   Arkansas California   Colorado
##          1          1          1          2          1          1
```

10.3 Bibliographic Notes

For more on PCA see my Dimensionality Reduction Class Notes and references therein. For more on everything, see Friedman et al. (2001). For a softer introduction, see James et al. (2013).

Chapter 11

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 high-level philosophy. The **graphics** package originates from the main-frame days. Computers had no graphical interface, and the output of the plot was immediately sent to a printer. For this reason, once a plot has been produced with the **graphics** package, it cannot be queryied 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 small set of building blocks. The building blocks in **ggplot2** are the ones stated by Wilkinson (2006). The objects and grammar of **ggplot2** have later evolved to allow more complicated plotting and in particular, interactive plotting, in other packages.

Interactive plotting is a very important feature for EDA, and for reporting. The major leap in interactive plotting was made possible by the advancement of web technologies, such as JavaScript. 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 reply on the web-browsers capabilities for interactivity.

One of the latest contributions to interactive plotting, is the **Shiny** framework by RStudio (RStudio, Inc, 2013). **Shiny**, unlike other interactive plotting systems, is not a static web-site. It is a web-server, that can query R, with all its facilities.

11.1 The graphics System

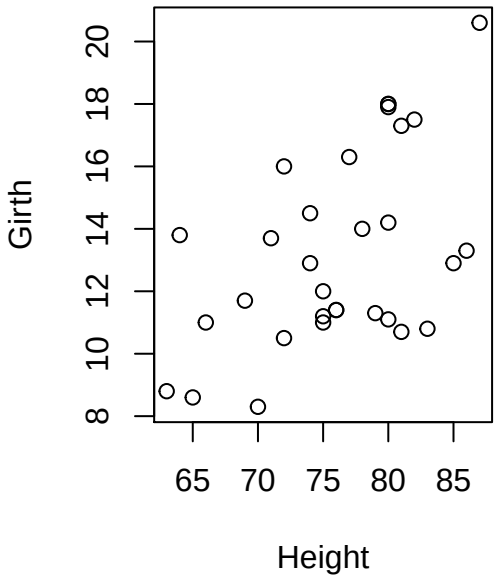
The R code from the Basics Chapter 3 is a demonstration of the **graphics** package and system. We make a quick review of the basics.

11.1.1 Using Existing Plotting Functions

11.1.1.1 Scatter Plot

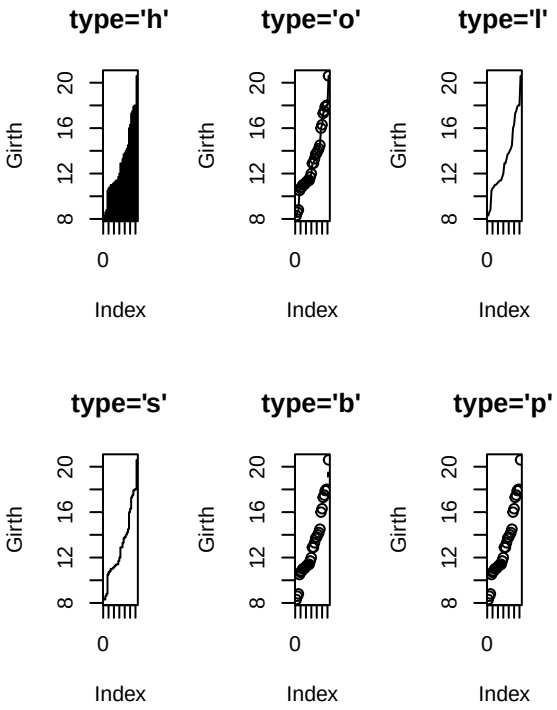
A simple scatter plot.

```
attach(trees)
plot(Girth ~ Height)
```



Various types of plots.

```
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'")
```



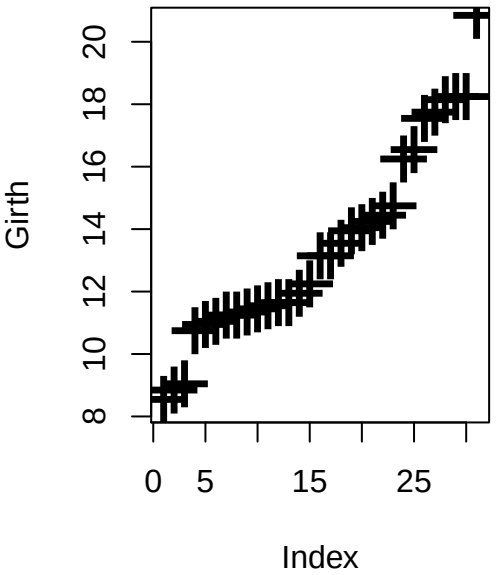

```
par(mfrow=c(1,1))
```

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 `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.

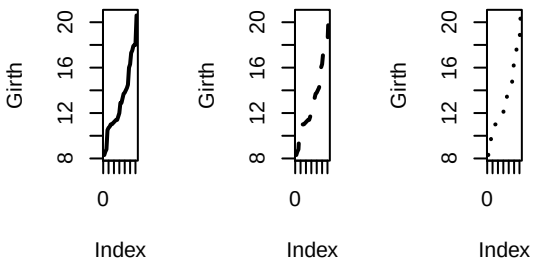
```
plot(Girth, pch='+', cex=3)
```

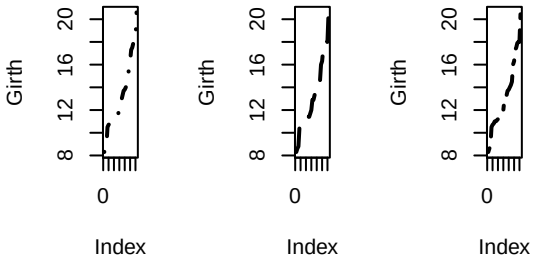


Control the line's type with `lty` argument, and width with `lwd`.

```
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)
```

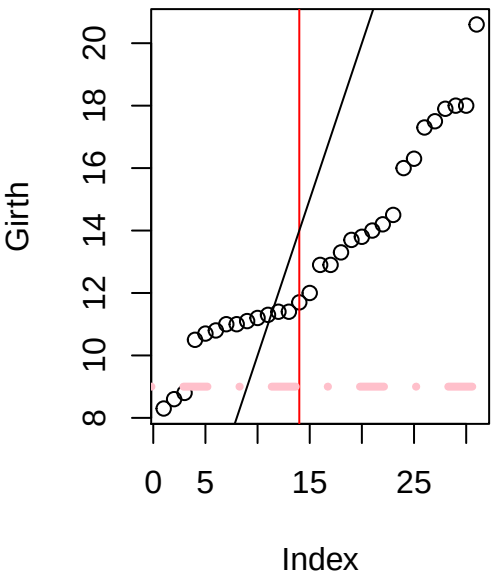





```
par(mfrow=c(1,1))
```

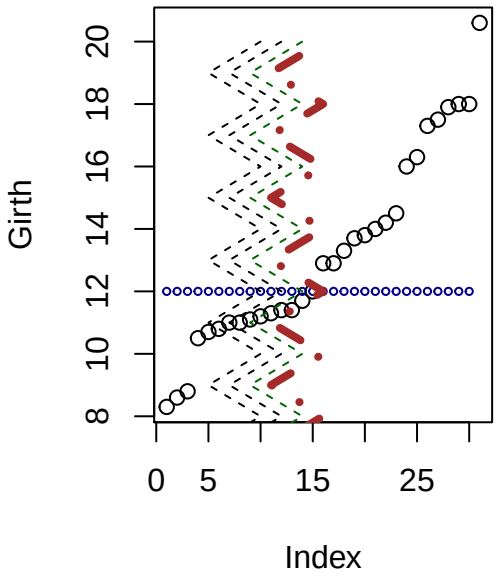
Add line by slope and intercept with `abline`.

```
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')
```

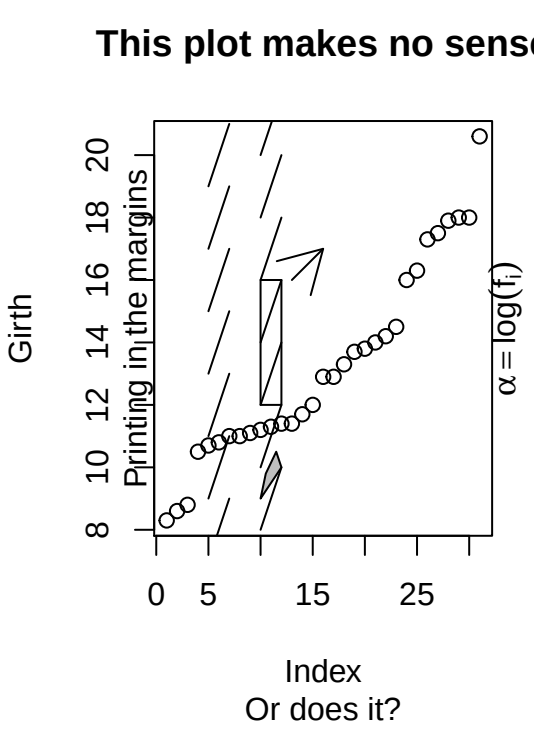
```
lines(x=rep(c(5,10), 7)+6, y=7:20, lty=4 , col='brown', lwd=4)
```



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.

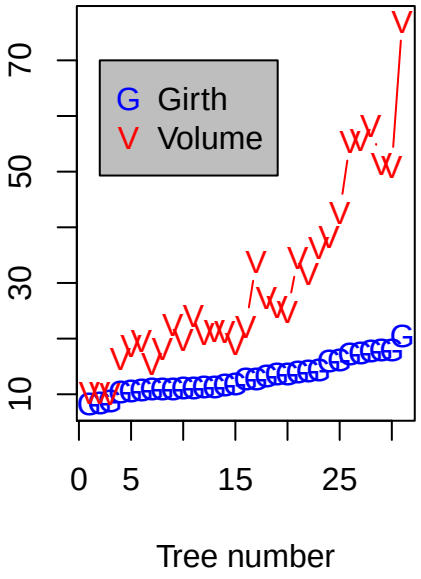
```
plot(Girth)
segments(x0=rep(c(5,10), 7), y0=7:20, x1=rep(c(5,10), 7)+2, y1=(7:20)+2 )
arrows(x0=13,y0=16,x1=16,y1=17, )
rect(xleft=10, ybottom=12,  xright=12, ytop=16)
polygon(x=c(10,11,12,11.5,10.5), y=c(9,9.5,10,10.5,9.8), col='grey')
title(main='This plot makes no sense', sub='Or does it?')
mtext('Printing in the margins', side=2)
mtext(expression(alpha==log(f[i])), side=4)
```



Things to note: - The following functions add the elements they are names after: `segments`, `arrows`, `rect`, `polygon`, `title`. - `mtext` adds mathematical text. For more information for mathematical annotation see `?plotmath`.

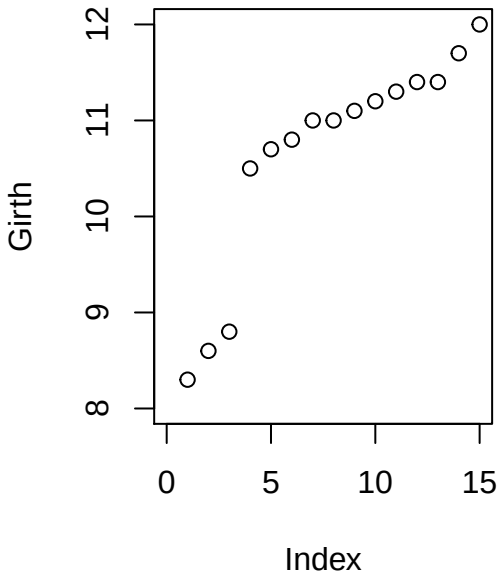
Add a legend.

```
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`.

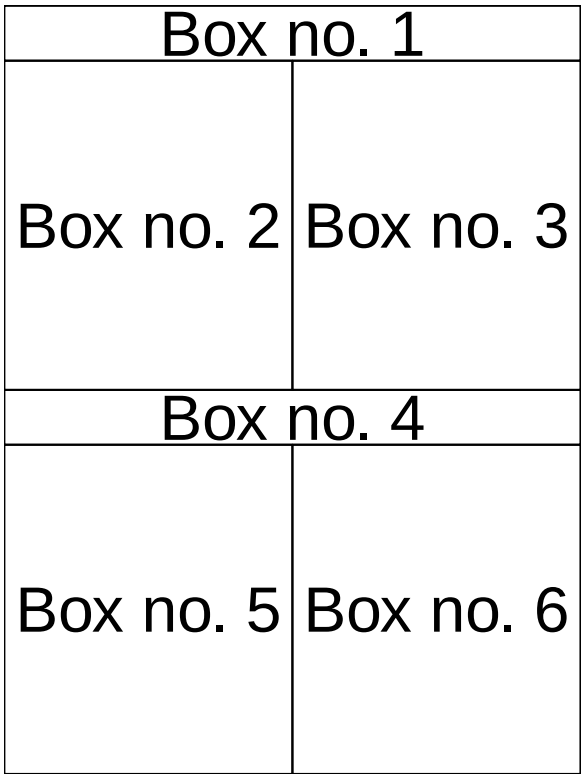
```
plot(Girth, xlim=c(0,15), ylim=c(8,12))
```



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)
}
```



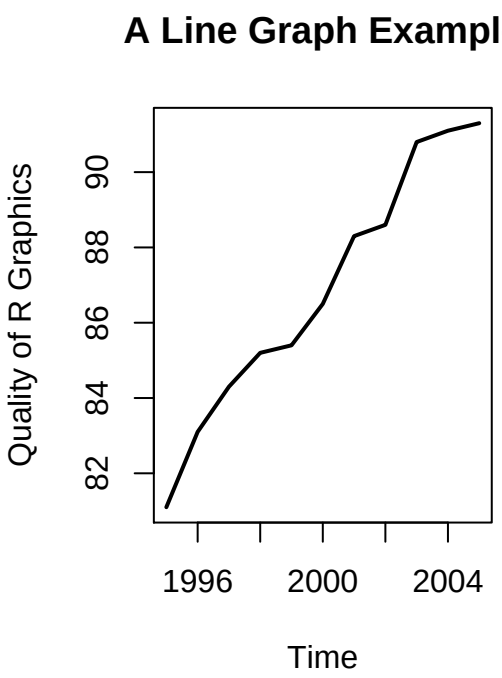
Always detach.

```
detach(trees)
```

11.1.2 The Power of the graphics device

Building a line graph from scratch.

```
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()
```

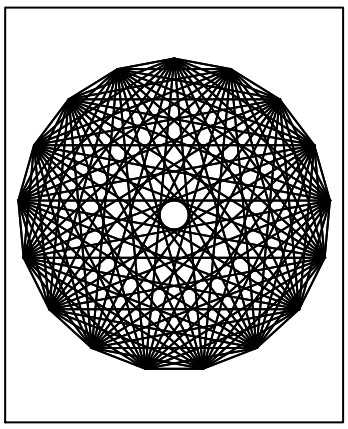


Things to note:

- `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.

Rosette.

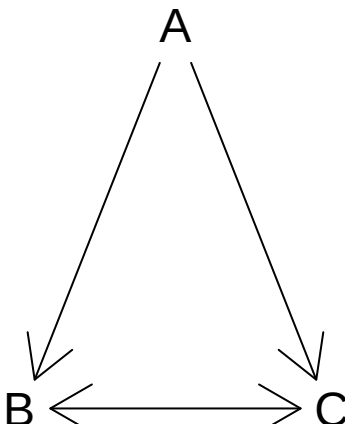
```
n = 17
theta = seq(0, 2 * pi, length = n + 1)[1:n]
x = sin(theta)
y = cos(theta)
v1 = rep(1:n, n)
v2 = rep(1:n, rep(n, n))
plot.new()
plot.window(xlim = c(-1, 1), ylim = c(-1, 1), asp = 1)
segments(x[v1], y[v1], x[v2], y[v2])
box()
```



Arrows.

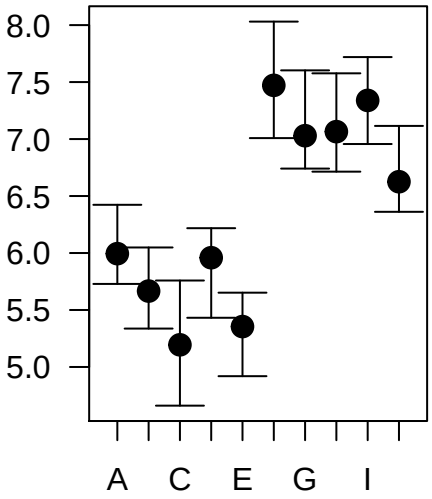
```
plot.new()
plot.window(xlim = c(0, 1), ylim = c(0, 1))
arrows(.05, .075, .45, .9, code = 1)
```

```
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)
```



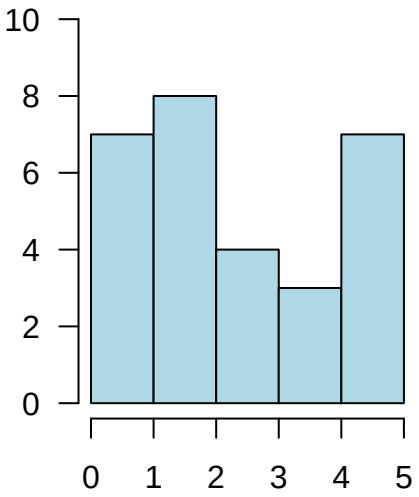
Arrows as error bars.

```
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()
```



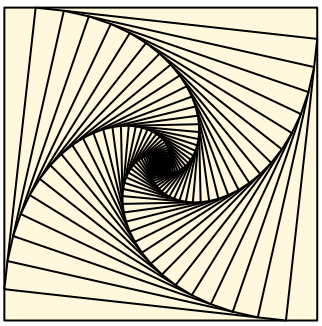
A histogram is nothing but a bunch of rectangle elements.

```
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)
```



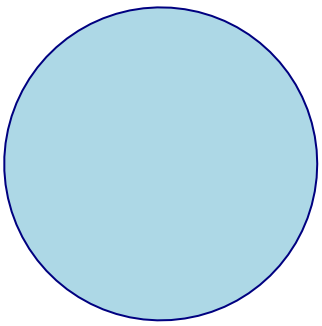
Spiral Squares.

```
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")
}
```



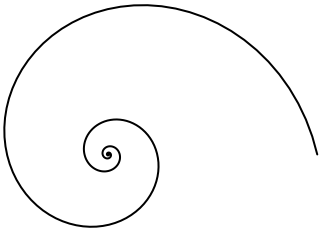
Circles are just dense polygons.

```
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")
```



Spiral- just a bunch of lines.

```
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)
```



11.1.3 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` is close 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.

```
tiff(filename='graphicExample%d.tiff') #Creates a sequence of files
plot(rnorm(100))
boxplot(rnorm(100))
hist(rnorm(100))
dev.off()
```

```
## pdf
##   2
```

To see the list of all open devices use `dev.list()`. To close **all** device, (not the last one), use `graphics.off()`.

See `?pdf` and `?jpeg` for more info.

11.2 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 queryied, 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 this function, 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:

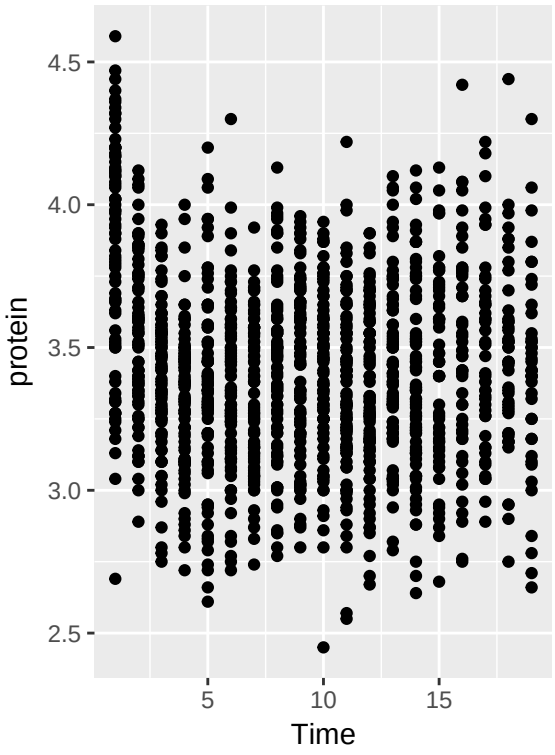
- **Data** are the variables mapped to aesthetic features of the graph.
- **Aes** is the mapping between objects to their visualization.
- **Geoms** are the objects/shapes you see on the graph.
- **Stats** are statistical transformations that summarize data, such as the mean or confidence intervals.
- **Scales** define which aesthetic values are mapped to data values. Legends and axes display these mappings.
- **Coordiante systems** define the plane on which data are mapped on the graphic.
- **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()
```

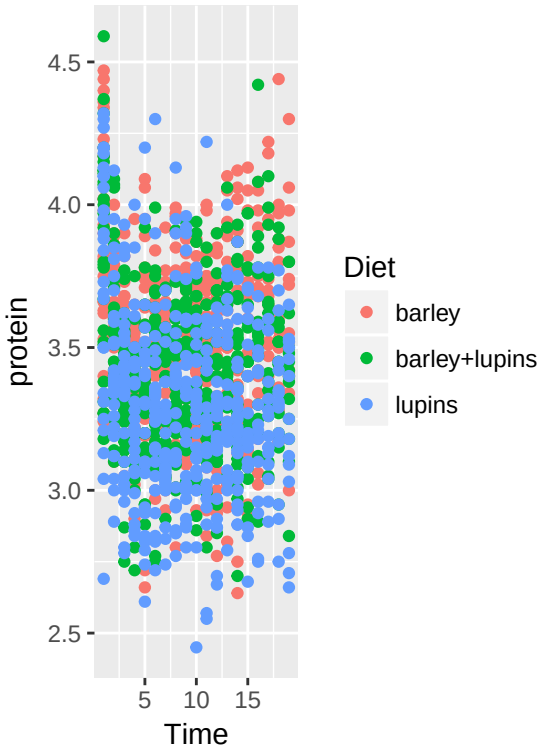


Things to note:

- 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.

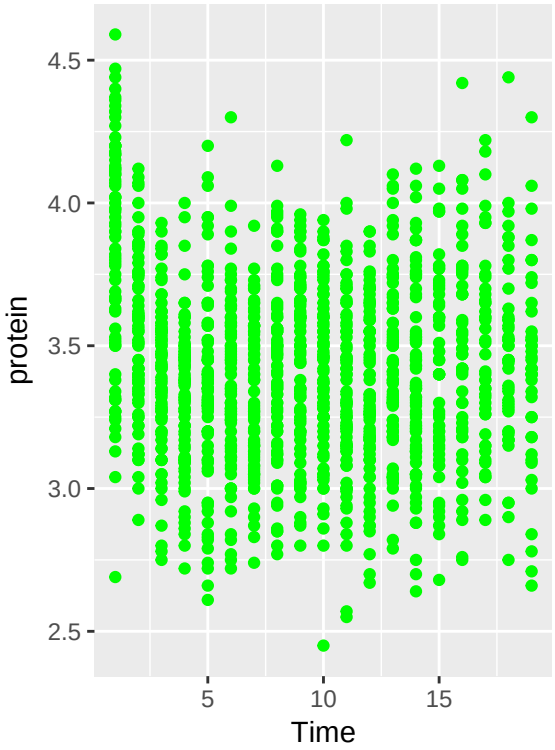
Let's add some color.

```
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. 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")
```



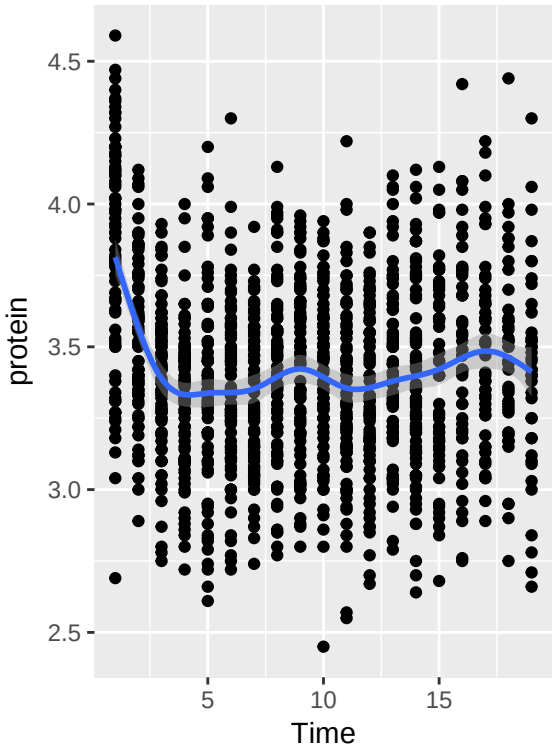
Let's save the **ggplot2** object so we can reuse it. Notice it is not plotted.

```
p <- ggplot(data = Milk, aes(x=Time, y=protein)) +
  geom_point()
```

We can add **layers** using the + operator. Here, we add a smoothing line.

```
p + geom_smooth()
```

```
## `geom_smooth()` using method = 'gam'
```

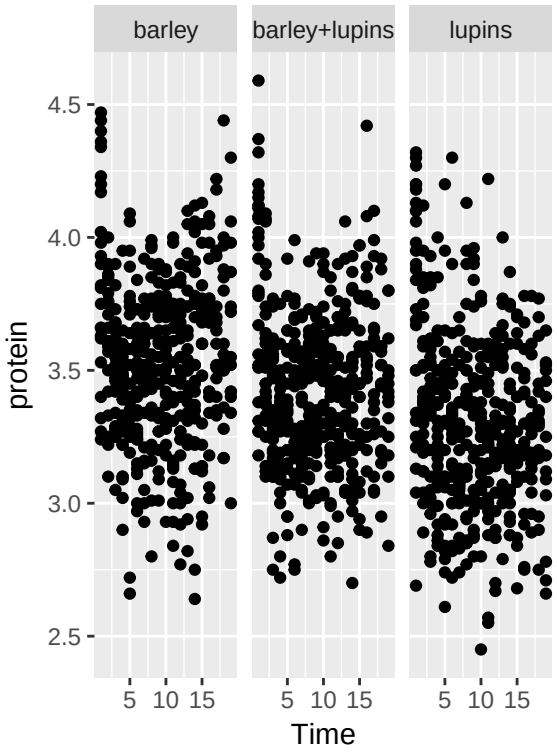


Things to note:

- The smoothing line is a layer added with the `geom_smooth()` function.
- Lacking any arguments, the new layer will inherit the `aes` of the original object, x and y variables in particular.

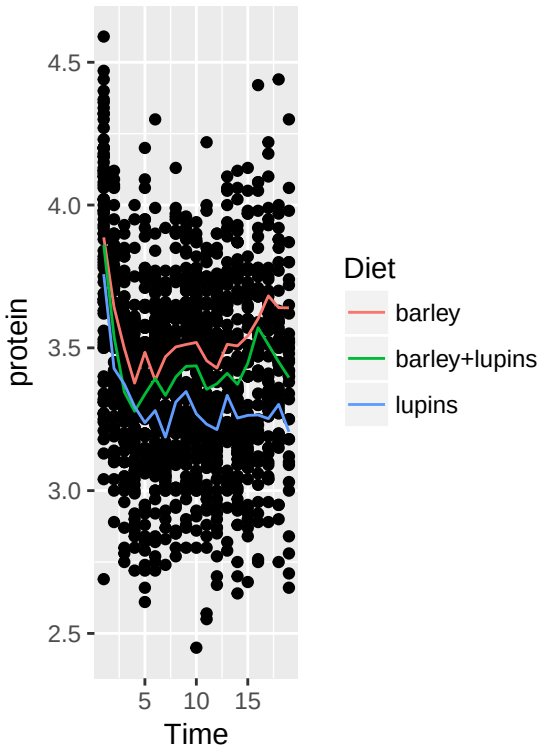
To split the plot along some variable, we use faceting, done with the `facet_wrap` function.

```
p + facet_wrap(~Diet)
```



We now 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.
- 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.

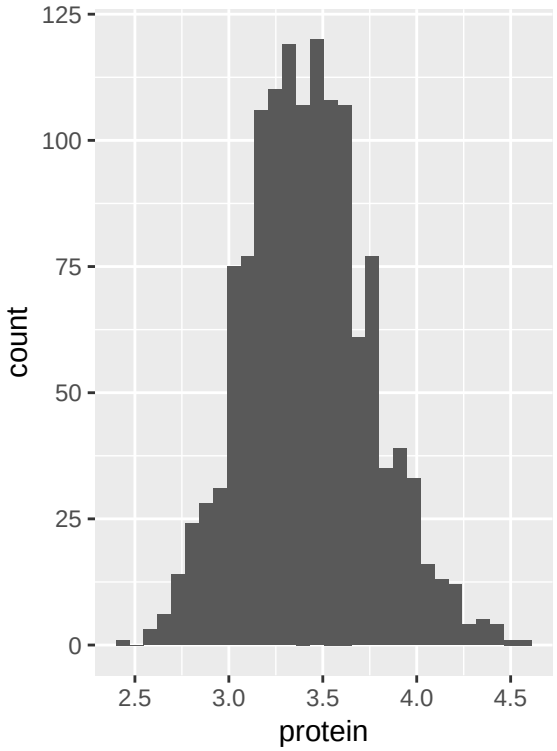
What layers can be added using the **geoms** family of functions?

- **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.

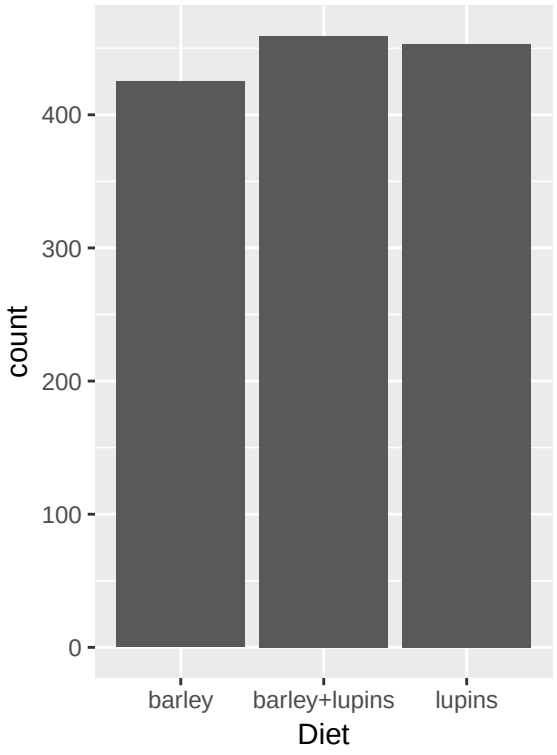
```
pro <- ggplot(Milk, aes(x=protein))
pro + geom_histogram()
```

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



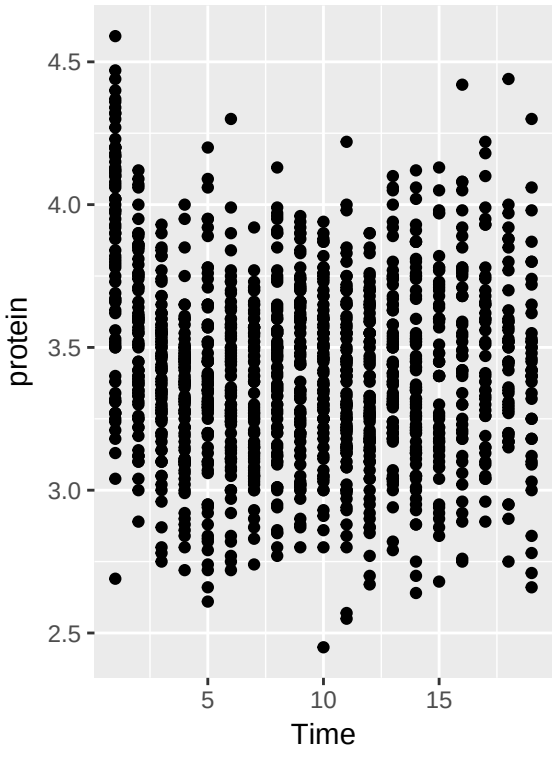
A bar plot.

```
ggplot(Milk, aes(x=Diet)) +
  geom_bar()
```



A scatter plot.

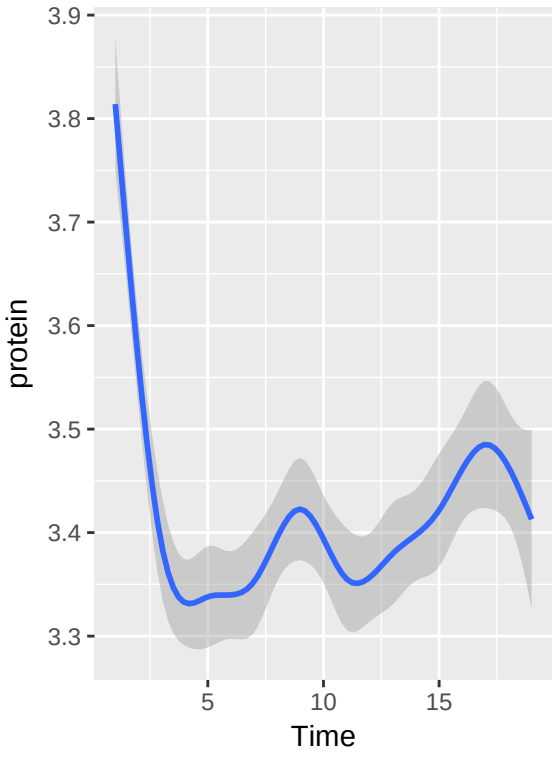
```
tp <- ggplot(Milk, aes(x=Time, y=protein))
tp + geom_point()
```



A smooth regression plot, reusing the `tp` object.

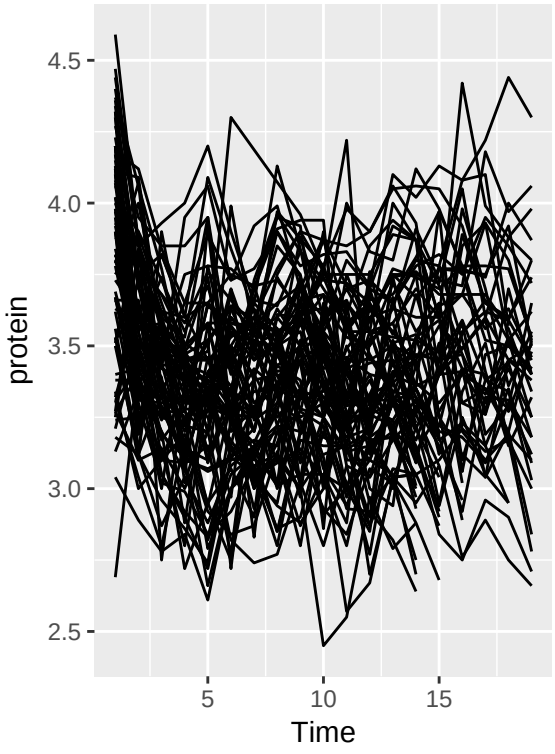
```
tp + geom_smooth()
```

```
## `geom_smooth()` using method = 'gam'
```



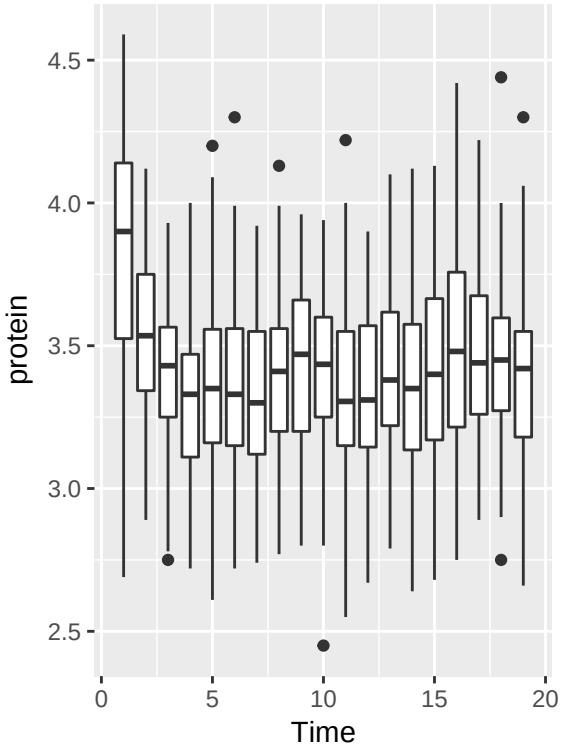
And now, a simple line plot, reusing the `tp` object, and connecting lines along `Cow`.

```
tp + geom_line(aes(group=Cow))
```



The line plot is completely incomprehensible. Better look at boxplots along time (even if committing the `Cow` information).

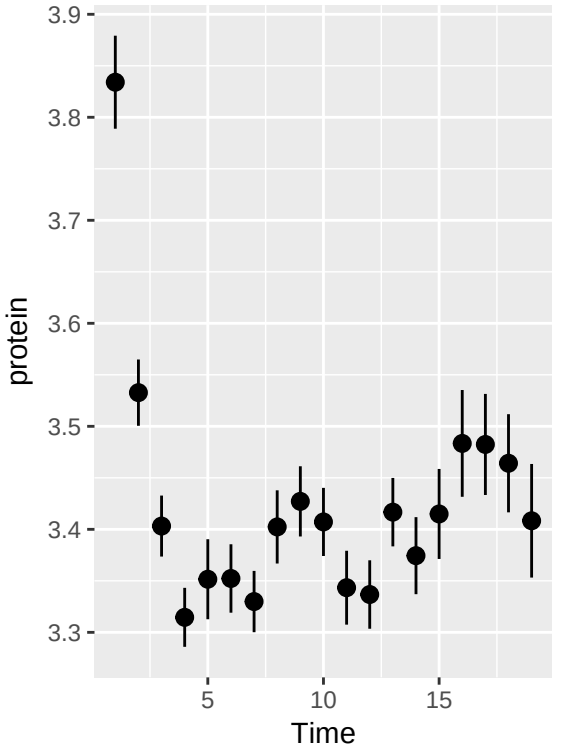
```
tp + geom_boxplot(aes(group=Time))
```



We can do some statistics for each subgroup. The following will compute the mean and standard errors (default of `stat_summary`) of `protein` at each time point.

```
ggplot(Milk, aes(x=Time, y=protein)) +
  stat_summary()
```

```
## No summary function supplied, defaulting to `mean_se()
```

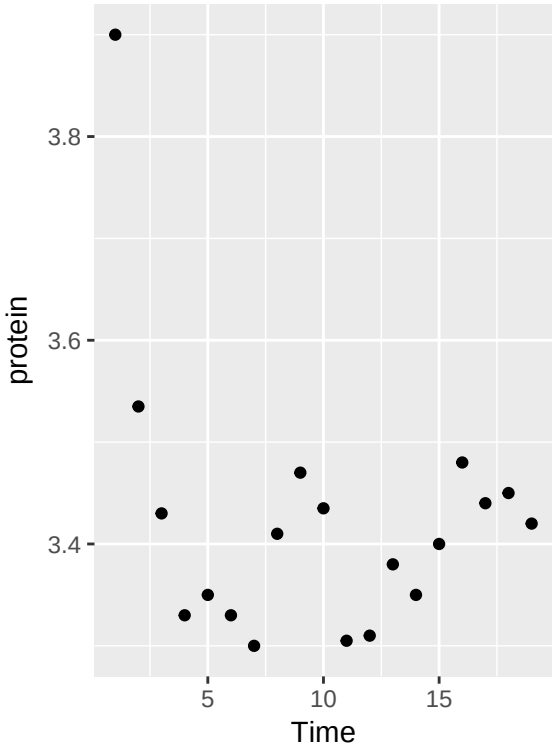


Some popular statistical summaries, have gained their own functions:

- `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).

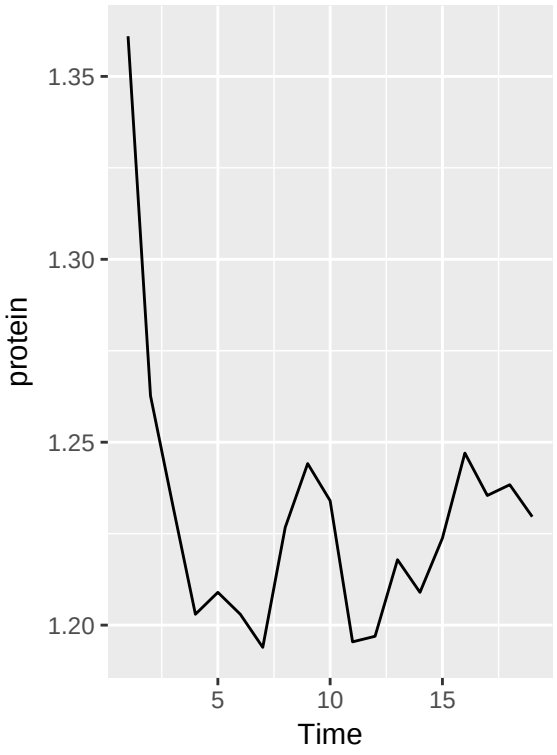
For less popular statistical summaries, we may specify the statistical function in `stat_summary`. The median is a first example.

```
ggplot(Milk, aes(x=Time, y=protein)) +
  stat_summary(fun.y="median", geom="point")
```



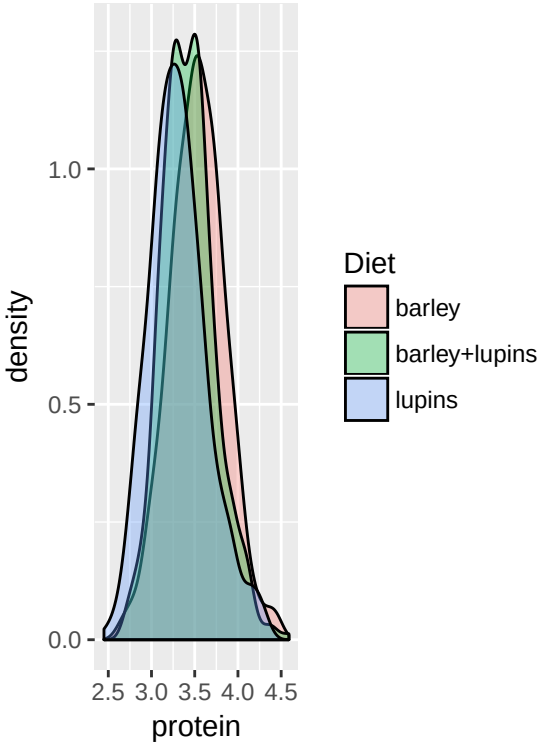
We can also define our own statistical summaries.

```
medianlog <- function(y) {median(log(y))}
ggplot(Milk, aes(x=Time, y=protein)) +
  stat_summary(fun.y="medianlog", geom="line")
```



Scales define the actual mapping of an aesthetics values to data values.

```
ggplot(Milk, aes(x=protein, fill=Diet)) +
  geom_density(alpha=1/3) +
  scale_fill_hue()
```

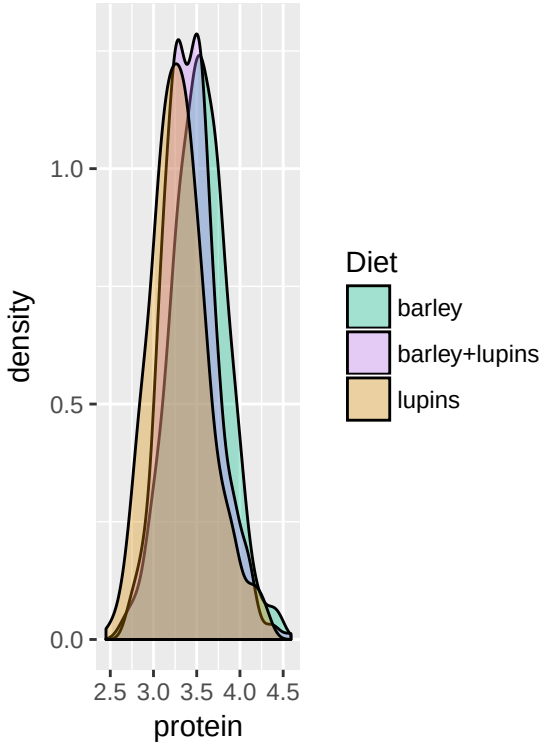


Things to note:

- The `geom_density` function tells R to plot density plots.
- The `alpha=1/3` parameter controls the transparency. Set to `alpha=1` for opaque, and `alpha=0` for transparent.
- The `scale_fill_hue` function, which is the default (thus can be omitted), tells R how to map factors to colors.

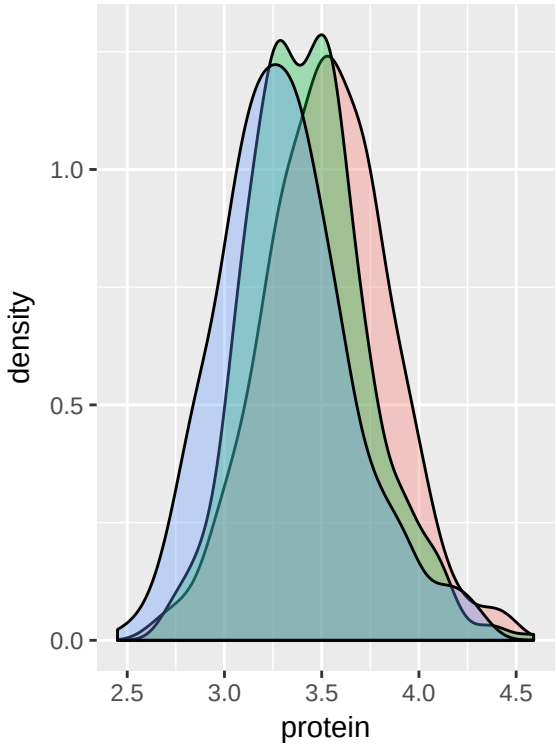
Let's change the default color mapping.

```
ggplot(Milk, aes(x=protein, fill=Diet)) +
  geom_density(alpha=1/3) +
  scale_fill_hue(h.start=150)
```



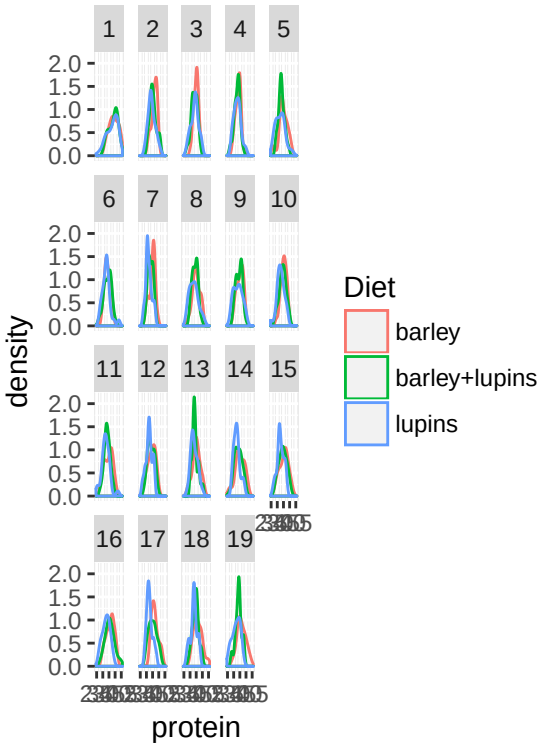
The legend is controlled with the `guides` function.

```
ggplot(Milk, aes(x=protein, fill=Diet)) +
  geom_density(alpha=1/3) +
  guides(fill="none")
```



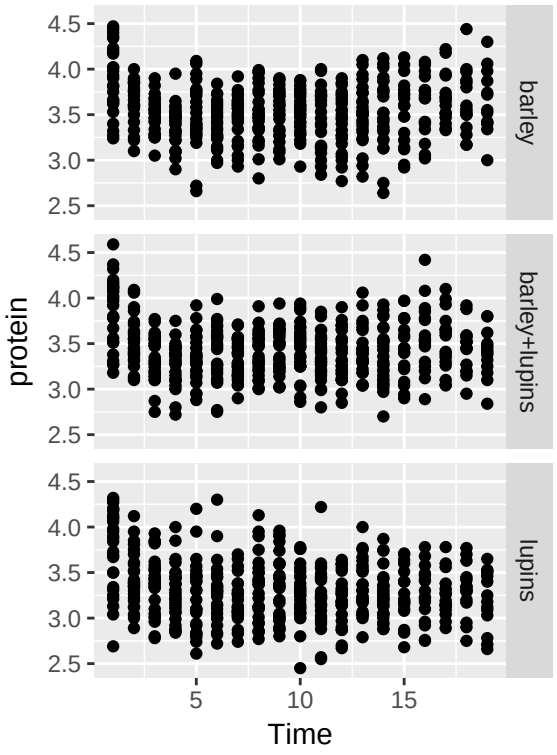
___Faceting allows to split the plotting along some variable. `face_wrap` tells R to compute the number of columns and rows of plots automatically.

```
ggplot(Milk, aes(x=protein, color=Diet)) +
  geom_density() +
  facet_wrap(~Time)
```



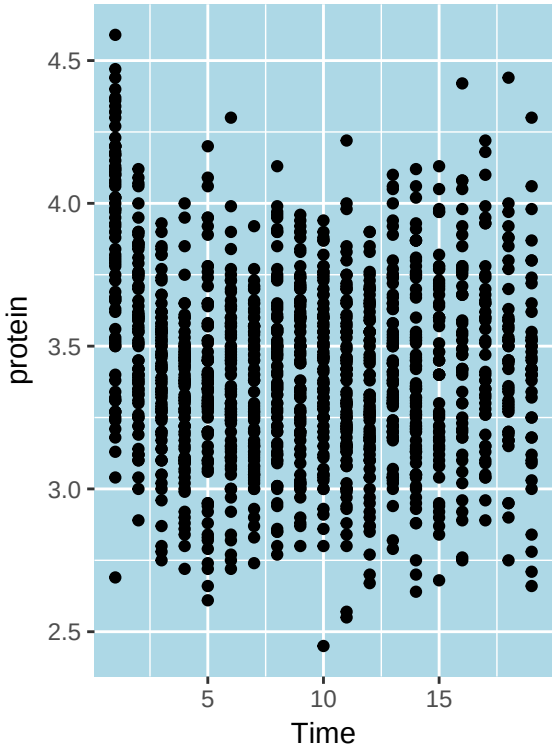
`facet_grid` forces the plot to appear allow rows or columns, using the `~` syntax.

```
ggplot(Milk, aes(x=Time, y=protein)) +
  geom_point() +
  facet_grid(Diet~.)
```

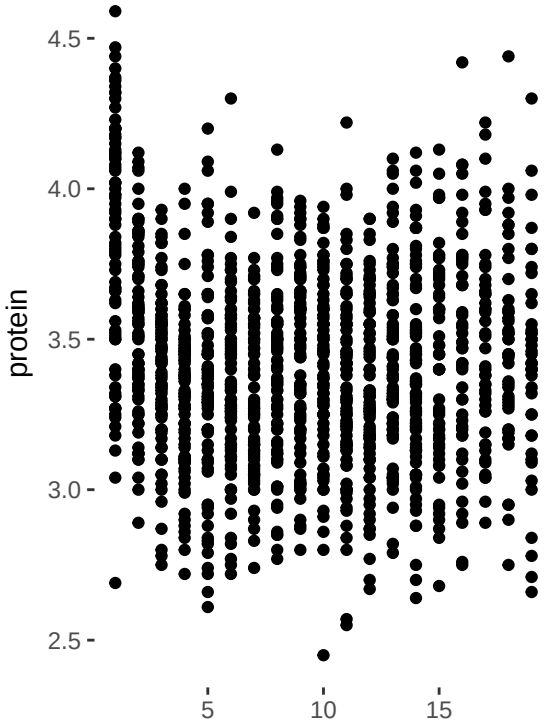


To control the looks of the plot, **ggplot2** uses **themes**.

```
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 the `pdf` function, but possibly easier with the `ggsave` function.

Finally, what every user of **ggplot2** constantly uses, is the online documentation at http://docs.ggplot2.org.

11.3 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. These libraries are more visually pleasing, and computationally efficient, than anything they could have developed themselves.

Some noteworthy interactive plotting systems are the following:

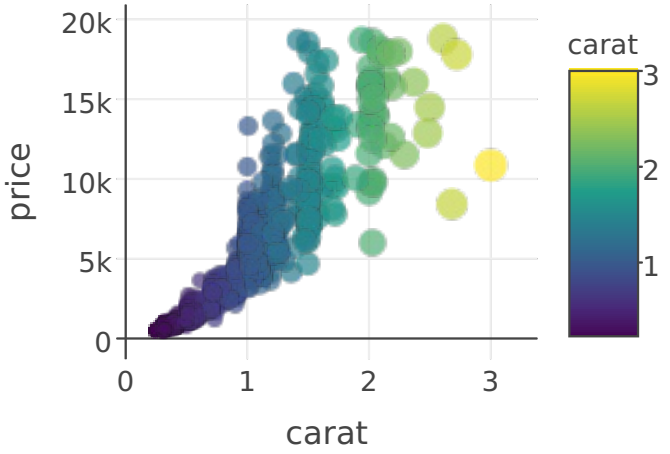
- **plotly**: The **plotly** package (Sievert et al., 2016) uses the (brilliant!) visualization framework of the Plotly company to provide local, or web-publishable, interactive graphics.
- **dygraphs**: The dygraphs JavaScript library is intended for interactive visualization of time series. The **dygraphs** R package is an interface allowing the plotting of R objects with this library. For more information see here.
- **rCharts**: If you like the **lattice** plotting system, the **rCharts** package will allow you to produce interactive plots from R using the **lattice** syntax. For more information see here.
- **clickme**: Very similar to **rCharts**.
- **ggv2**: Vega is a grammar for plots, i.e., a syntax that describes a plots elements, along with the appropriate JavaScript visualization libraries. **ggv2** is an an experimental package that produces Vega interactive plots from R. For more information see here.
- **rVega**: Same purpose as **ggv2**.
- **googleVis**: TODO
- **HTML Widgets**: 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.

11.3.1 Plotly

```
library(plotly)
set.seed(100)
d <- diamonds[sample(nrow(diamonds), 1000), ]
plot_ly(d, x = ~carat, y = ~price, color = ~carat,
        size = ~carat, text = ~paste("Clarity: ", clarity))
```

```
## No trace type specified:
##   Based on info supplied, a 'scatter' trace seems appropriate.
##   Read more about this trace type -> https://plot.ly/r/reference/#scatter
```

```
## No scatter mode specifed:
##   Setting the mode to markers
##   Read more about this attribute -> https://plot.ly/r/reference/#scatter-mode
```



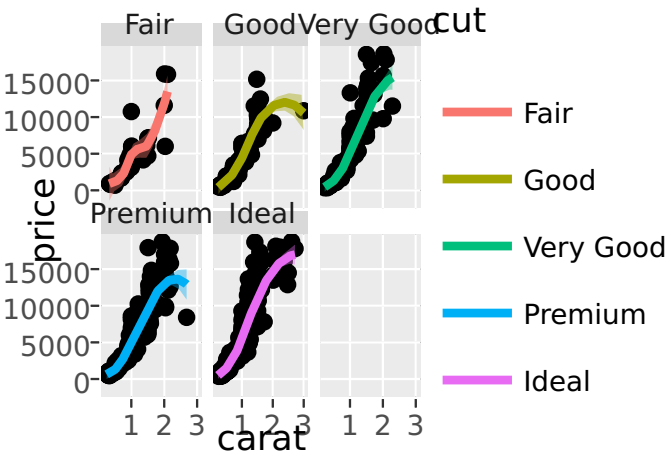
If you are comfortable with **ggplot2**, you may use the **ggplot2** syntax, and export the final result to **plotly**.

```
p <- ggplot(data = d, aes(x = carat, y = price)) +
  geom_point(aes(text = paste("Clarity:", clarity))) +
  geom_smooth(aes(colour = cut, fill = cut)) + facet_wrap(~ cut)
```

```
## Warning: Ignoring unknown aesthetics: text
```

```
ggplotly(p)
```

```
## `geom_smooth()` using method = 'loess'
```



For more on **plotly** see https://plot.ly/r/.

11.3.2 HTML Widgets

11.4 Bibliographic Notes

For the **graphics** package, see R Core Team (2016). For **ggplot2** see Wickham (2009).

Chapter 12

Reports

If you 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 to documents, typically 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, compile it, 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 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. This book 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. **Shiney**: The previous reporting frameworks are static in that R "dumps" its output and is never queryied again. This does not mean that output is static, but only that R is not called. **Shiny** (Chang et al., 2017) is different. 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).

12.1 knitr

12.1.1 Installation

To run **knitr** you will need to install the package.

[1] Recall, S was the original software from which R evolved.

```
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.

12.1.2 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~` ($like_{this}$), and superscripts are enclosed in carets `like^this^` ($like^{this}$). For `verbatim` use "`verbatim`".

For links use `[text](link)`, like `[my site](www.john-ros.com)`. Image is the same as a link, starting with an exclamation, like this `![image title](image path)`.

An itemized list simply starts with hyphens:

```
- 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 here.

12.1.3 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:

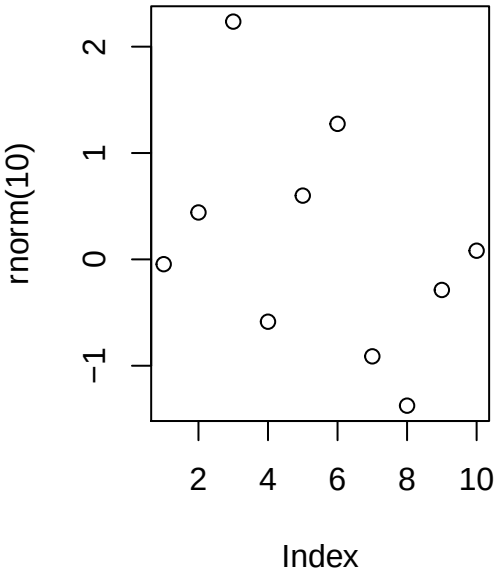
- The evaluated expression is added in a chunk of highlighted text.
- The output is added prefixed with `##`.
- The `eval=` argument is not required, since it is set to `eval=TRUE` by default. It does demonstrate how to set the options of the code chunk.

In the same way, we may add a plot:

````
```{r  eval=FALSE}
plot(rnorm(10))
```
````

which compiles into

```
plot(rnorm(10))
```



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.

12.1.4 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.

12.2 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.

12.3 Shiny

Shiny (RStudio, Inc, 2013) 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 they layout of the web-site, and setting up the web-server, is made with several simple R commands, with no need for web-programming. For this purpose, Shiny uses the Bootstrap web development technology. 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.

12.3.1 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.

12.3.2 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")
```

The site's layout, is specified via *layout functions* in the `iu.R` file. 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 queryied 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 `io.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.

```
knitr::include_url('http://shiny.rstudio.com/gallery/example-01-hello.html')
```

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(),

      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.
- `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:

```
knitr::include_url('https://shiny.rstudio.com/gallery/tabsets.html')
```

12.3.3 Beyond the Basics

Now that we have seen the basics, we may consider extensions to the basic report.

12.3.3.1 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.

```
knitr::include_url('https://shiny.rstudio.com/gallery/widget-gallery.html')
```

12.3.3.2 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.

12.4 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.

Chapter 13

The Hadleyverse

The *Hadleyverse*, short for "Hadley Wickham's universe", is a set of packages that make it easier to handle data. If you are developing packages, you should be careful since using these packages may create many dependencies and compatibility issues. If you are analyzing data, and the portability of your functions to other users, machines, and operating systems is not of a concern, you will LOVE these packages. The term Hadleyverse refers to **all** of Hadley's packages, but here, we mention only a useful subset, which can be collectively installed via the **tidyverse** package:

- **ggplot2** for data visualization. See the Plotting Chapter 11.
- **dplyr** for data manipulation.
- **tidyr** for data tidying.
- **readr** for data import.
- **stringr** for character strings.
- **anytime** for time data.

13.1 readr

The **readr** package (Wickham et al., 2016) replaces base functions for importing and exporting data such as `read.table`. It is faster, with a cleaner syntax.

We will not go into the details and refer the reader to the official documentation here and the R for data sciecne book.

13.2 dplyr

When you think of data frame operations, think **dplyr** (Wickham and Francois, 2016). Notable utilities in the package include:

- `select()` Select columns from a data frame.
- `filter()` Filter rows according to some condition(s).
- `arrange()` Sort / Re-order rows in a data frame.
- `mutate()` Create new columns or transform existing ones.
- `group_by()` Group a data frame by some factor(s) usually in conjunction to summary.
- `summarize()` Summarize some values from the data frame or across groups.
- `inner_join(x,y,by="col")`return all rows from 'x' where there are matching values in 'x', and all columns from 'x' and 'y'. If there are multiple matches between 'x' and 'y', all combination of the matches are returned.
- `left_join(x,y,by="col")` return all rows from 'x', and all columns from 'x' and 'y'. Rows in 'x' with no match in 'y' will have 'NA' values in the new columns. If there are multiple matches between 'x' and 'y', all combinations of the matches are returned.
- `right_join(x,y,by="col")` return all rows from 'y', and all columns from 'x' and y. Rows in 'y' with no match in 'x' will have 'NA' values in the new columns. If there are multiple matches between 'x' and 'y', all combinations of the matches are returned.

- `anti_join(x,y,by="col")` return all rows from 'x' where there are not matching values in 'y', keeping just columns from 'x'.

The following example involve `data.frame` objects, but **dplyr** can handle other classes. In particular `data.tables` from the **data.table** package (Dowle and Srinivasan, 2017), which is designed for very large data sets.

dplyr can work with data stored in a database. In which case, it will convert your command to the appropriate SQL syntax, and issue it to the database. This has the advantage that (a) you do not need to know the specific SQL implementation of your database, and (b), you can enjoy the optimized algorithms provided by the database supplier. For more on this, see the databses vignette.

The following examples are taken from Kevin Markham. The `nycflights13::flights` has delay data for US flights.

```
library(nycflights13)
flights
```

```
## # A tibble: 336,776 × 19
##     year month   day dep_time sched_dep_time dep_delay arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>
## 1   2013     1     1      517            515         2      830
## 2   2013     1     1      533            529         4      850
## 3   2013     1     1      542            540         2      923
## 4   2013     1     1      544            545        -1     1004
## 5   2013     1     1      554            600        -6      812
## 6   2013     1     1      554            558        -4      740
## 7   2013     1     1      555            600        -5      913
## 8   2013     1     1      557            600        -3      709
## 9   2013     1     1      557            600        -3      838
## 10  2013     1     1      558            600        -2      753
## # ... with 336,766 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 <dttm>
```

The data is of class `tbl_df` which is an extension of the `data.frame` class, designed for large data sets. Notice that the printing of `flights` is short, even without calling the `head` function. This is a feature of the `tbl_df` class (`print(data.frame)` would try to load all the data, thus take a long time).

```
class(flights) # a tbl_df is an extension of the data.frame class
```

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

Let's filter the observations from the first day of the first month. Notice how much better (i.e. readable) is the **dplyr** syntax, with piping, compared to the basic syntax.

```
flights[flights$month == 1 & flights$day == 1, ] # old style

library(dplyr)
filter(flights, month == 1, day == 1) #dplyr style
flights %>% filter(month == 1, day == 1) # dplyr with piping.
```

More filtering.

```
filter(flights, month == 1 | month == 2) # First OR second month.
slice(flights, 1:10) # selects first ten rows.

arrange(flights, year, month, day) # sort
arrange(flights, desc(arr_delay)) # sort descending

select(flights, year, month, day) # select columns year, month, and day
select(flights, year:day) # select column range
select(flights, -(year:day)) # drop columns
rename(flights, tail_num = tailnum) # rename column
```

```
# add a new computed colume
mutate(flights,
  gain = arr_delay - dep_delay,
  speed = distance / air_time * 60)

# you can refer to columns you just created! (gain)
mutate(flights,
  gain = arr_delay - dep_delay,
  gain_per_hour = gain / (air_time / 60)
)

# keep only new variables, not all data frame.
transmute(flights,
  gain = arr_delay - dep_delay,
  gain_per_hour = gain / (air_time / 60)
)

# simple statistics
summarise(flights,
  delay = mean(dep_delay, na.rm = TRUE)
  )
# random subsample
sample_n(flights, 10)
sample_frac(flights, 0.01)
```

We now perform operations on subgroups. we group observations along the plane's tail number (`tailnum`), and compute the count, average distance traveled, and average delay. We group with `group_by`, and compute subgroup statistics with `summarise`.

```
by_tailnum <- group_by(flights, tailnum)

delay <- summarise(by_tailnum,
  count = n(),
  avg.dist = mean(distance, na.rm = TRUE),
  avg.delay = mean(arr_delay, na.rm = TRUE))

delay
```

```
## # A tibble: 4,044 × 4
##    tailnum count avg.dist  avg.delay
##      <chr> <int>    <dbl>      <dbl>
## 1   D942DN     4 854.5000 31.5000000
## 2   N0EGMQ   371 676.1887  9.9829545
## 3   N10156   153 757.9477 12.7172414
## 4   N102UW    48 535.8750  2.9375000
## 5   N103US    46 535.1957 -6.9347826
## 6   N104UW    47 535.2553  1.8043478
## 7   N10575   289 519.7024 20.6914498
## 8   N105UW    45 524.8444 -0.2666667
## 9   N107US    41 528.7073 -5.7317073
## 10  N108UW    60 534.5000 -1.2500000
## # ... with 4,034 more rows
```

We can group along several variables, with a hierarchy. We then collapse the hierarchy one by one.

```
daily <- group_by(flights, year, month, day)
per_day   <- summarise(daily, flights = n())
per_month <- summarise(per_day, flights = sum(flights))
```

```
per_year  <- summarise(per_month, flights = sum(flights))
```

Things to note:

- Every call to `summarise` collapses one level in the hierarchy of grouping. The output of `group_by` recalls the hierarchy of aggregation, and collapses along this hierarchy.

We can use **dplyr** for two table operations, i.e., *joins*. For this, we join the flight data, with the airplane data in `airplanes`.

```
library(dplyr)
airlines
```

```
## # A tibble: 16 × 2
##    carrier                        name
##      <chr>                       <chr>
## 1       9E           Endeavor Air Inc.
## 2       AA      American Airlines Inc.
## 3       AS        Alaska Airlines Inc.
## 4       B6             JetBlue Airways
## 5       DL        Delta Air Lines Inc.
## 6       EV    ExpressJet Airlines Inc.
## 7       F9      Frontier Airlines Inc.
## 8       FL AirTran Airways Corporation
## 9       HA      Hawaiian Airlines Inc.
## 10      MQ                  Envoy Air
## 11      OO       SkyWest Airlines Inc.
## 12      UA       United Air Lines Inc.
## 13      US             US Airways Inc.
## 14      VX              Virgin America
## 15      WN      Southwest Airlines Co.
## 16      YV          Mesa Airlines Inc.
```

```
# select the subset of interesting flight data.
flights2 <- flights %>% select(year:day, hour, origin, dest, tailnum, carrier)

# join on left table with automatic matching.
flights2 %>% left_join(airlines)
```

```
## Joining, by = "carrier"
```

```
## # A tibble: 336,776 × 9
##     year month   day  hour origin  dest tailnum carrier
##    <int> <int> <int> <dbl>  <chr> <chr>   <chr>   <chr>
## 1   2013     1     1     5    EWR   IAH  N14228      UA
## 2   2013     1     1     5    LGA   IAH  N24211      UA
## 3   2013     1     1     5    JFK   MIA  N619AA      AA
## 4   2013     1     1     5    JFK   BQN  N804JB      B6
## 5   2013     1     1     6    LGA   ATL  N668DN      DL
## 6   2013     1     1     5    EWR   ORD  N39463      UA
## 7   2013     1     1     6    EWR   FLL  N516JB      B6
## 8   2013     1     1     6    LGA   IAD  N829AS      EV
## 9   2013     1     1     6    JFK   MCO  N593JB      B6
## 10  2013     1     1     6    LGA   ORD  N3ALAA      AA
## # ... with 336,766 more rows, and 1 more variables: name <chr>
```

```
flights2 %>% left_join(weather)
```

```
## Joining, by = c("year", "month", "day", "hour", "origin")
```

```
## # A tibble: 336,776 × 18
##     year month   day  hour origin  dest tailnum carrier  temp  dewp humid
```

```
##    <dbl> <dbl> <int> <dbl>  <chr> <chr>   <chr>   <chr> <dbl> <dbl> <dbl>
## 1   2013     1     1     5    EWR   IAH  N14228      UA    NA    NA    NA
## 2   2013     1     1     5    LGA   IAH  N24211      UA    NA    NA    NA
## 3   2013     1     1     5    JFK   MIA  N619AA      AA    NA    NA    NA
## 4   2013     1     1     5    JFK   BQN  N804JB      B6    NA    NA    NA
## 5   2013     1     1     6    LGA   ATL  N668DN      DL 39.92 26.06 57.33
## 6   2013     1     1     5    EWR   ORD  N39463      UA    NA    NA    NA
## 7   2013     1     1     6    EWR   FLL  N516JB      B6 39.02 26.06 59.37
## 8   2013     1     1     6    LGA   IAD  N829AS      EV 39.92 26.06 57.33
## 9   2013     1     1     6    JFK   MCO  N593JB      B6 39.02 26.06 59.37
## 10  2013     1     1     6    LGA   ORD  N3ALAA      AA 39.92 26.06 57.33
## # ... with 336,766 more rows, and 7 more variables: wind_dir <dbl>,
## #   wind_speed <dbl>, wind_gust <dbl>, precip <dbl>, pressure <dbl>,
## #   visib <dbl>, time_hour <dttm>
```

```
# join with named matching
flights2 %>% left_join(planes, by = "tailnum")
```

```
## # A tibble: 336,776 × 16
##    year.x month   day  hour origin  dest tailnum carrier year.y
##     <int> <int> <int> <dbl>  <chr> <chr>   <chr>   <chr>  <int>
## 1    2013     1     1     5    EWR   IAH  N14228      UA   1999
## 2    2013     1     1     5    LGA   IAH  N24211      UA   1998
## 3    2013     1     1     5    JFK   MIA  N619AA      AA   1990
## 4    2013     1     1     5    JFK   BQN  N804JB      B6   2012
## 5    2013     1     1     6    LGA   ATL  N668DN      DL   1991
## 6    2013     1     1     5    EWR   ORD  N39463      UA   2012
## 7    2013     1     1     6    EWR   FLL  N516JB      B6   2000
## 8    2013     1     1     6    LGA   IAD  N829AS      EV   1998
## 9    2013     1     1     6    JFK   MCO  N593JB      B6   2004
## 10   2013     1     1     6    LGA   ORD  N3ALAA      AA     NA
## # ... with 336,766 more rows, and 7 more variables: type <chr>,
## #   manufacturer <chr>, model <chr>, engines <int>, seats <int>,
## #   speed <int>, engine <chr>
```

```
# join with explicit column matching
flights2 %>% left_join(airports, by= c("dest" = "faa"))
```

```
## # A tibble: 336,776 × 15
##     year month   day  hour origin  dest tailnum carrier
##    <int> <int> <int> <dbl>  <chr> <chr>   <chr>   <chr>
## 1   2013     1     1     5    EWR   IAH  N14228      UA
## 2   2013     1     1     5    LGA   IAH  N24211      UA
## 3   2013     1     1     5    JFK   MIA  N619AA      AA
## 4   2013     1     1     5    JFK   BQN  N804JB      B6
## 5   2013     1     1     6    LGA   ATL  N668DN      DL
## 6   2013     1     1     5    EWR   ORD  N39463      UA
## 7   2013     1     1     6    EWR   FLL  N516JB      B6
## 8   2013     1     1     6    LGA   IAD  N829AS      EV
## 9   2013     1     1     6    JFK   MCO  N593JB      B6
## 10  2013     1     1     6    LGA   ORD  N3ALAA      AA
## # ... with 336,766 more rows, and 7 more variables: name <chr>, lat <dbl>,
## #   lon <dbl>, alt <int>, tz <dbl>, dst <chr>, tzone <chr>
```

Types of join with SQL equivalent.

```
# Create simple data
(df1 <- data_frame(x = c(1, 2), y = 2:1))
```

```
## # A tibble: 2 × 2
```

```
##       x     y
##   <dbl> <int>
## 1     1     2
## 2     2     1
```

```
(df2 <- data_frame(x = c(1, 3), a = 10, b = "a"))
```

```
## # A tibble: 2 × 3
##       x     a     b
##   <dbl> <dbl> <chr>
## 1     1    10     a
## 2     3    10     a
```

```
# Return only matched rows
df1 %>% inner_join(df2) # SELECT * FROM x JOIN y ON x.a = y.a
```

```
## Joining, by = "x"
```

```
## # A tibble: 1 × 4
##       x     y     a     b
##   <dbl> <int> <dbl> <chr>
## 1     1     2    10     a
```

```
# Return all rows in df1.
df1 %>% left_join(df2) # SELECT * FROM x LEFT JOIN y ON x.a = y.a
```

```
## Joining, by = "x"
```

```
## # A tibble: 2 × 4
##       x     y     a     b
##   <dbl> <int> <dbl> <chr>
## 1     1     2    10     a
## 2     2     1    NA  <NA>
```

```
# Return all rows in df2.
df1 %>% right_join(df2) # SELECT * FROM x RIGHT JOIN y ON x.a = y.a
```

```
## Joining, by = "x"
```

```
## # A tibble: 2 × 4
##       x     y     a     b
##   <dbl> <int> <dbl> <chr>
## 1     1     2    10     a
## 2     3    NA    10     a
```

```
# Return all rows.
df1 %>% full_join(df2) # SELECT * FROM x FULL JOIN y ON x.a = y.a
```

```
## Joining, by = "x"
```

```
## # A tibble: 3 × 4
##       x     y     a     b
##   <dbl> <int> <dbl> <chr>
## 1     1     2    10     a
## 2     2     1    NA  <NA>
## 3     3    NA    10     a
```

```
# Like left_join, but returning only columns in df1
df1 %>% semi_join(df2, by = "x")  # SELECT * FROM x WHERE EXISTS (SELECT 1 FROM y WHERE x.a = y.a)
```

```
## # A tibble: 1 × 2
##       x     y
##   <dbl> <int>
## 1     1     2
```

13.3 tidyr

13.4 reshape2

13.5 stringr

13.6 anytime

13.7 Biblipgraphic Notes

Chapter 14

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 1GB of data, you might need about 3GB to fit a linear models. We will discuss how to compute *out of RAM* in the Memory Efficiency Chapter 15. 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 save 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 in the background, 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 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`. You will typically **not** use this syntax, which is there only for demonstration.
- 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
##   3.976   0.112   4.088
```

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
##   1.060   0.024   1.083
```

We now do the same with the sparse representation:

```
system.time(lm.2 <- MatrixModels:::lm.fit.sparse(X.2,y))
```

```
##    user  system elapsed
##   0.228   0.032   0.265
```

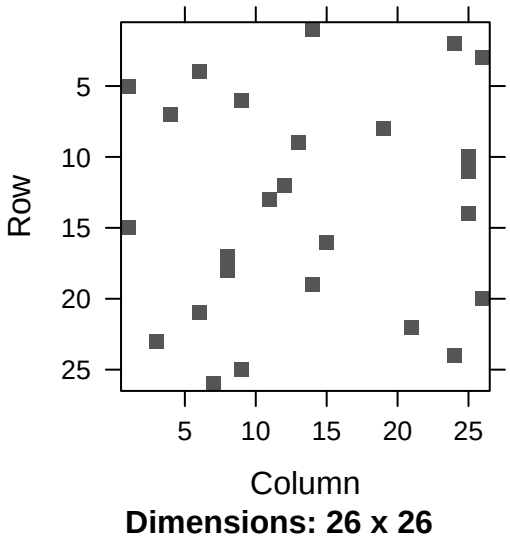
It is only left to verify that the returned coefficients are the same:

```
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])
```



14.1 Sparse Matrix Representations

We start with some terminology of sparse matrices. 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*.

14.1.1 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.

14.1.2 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 colum index vectors where the row index has to increase. The following figure may clarify this simple idea.

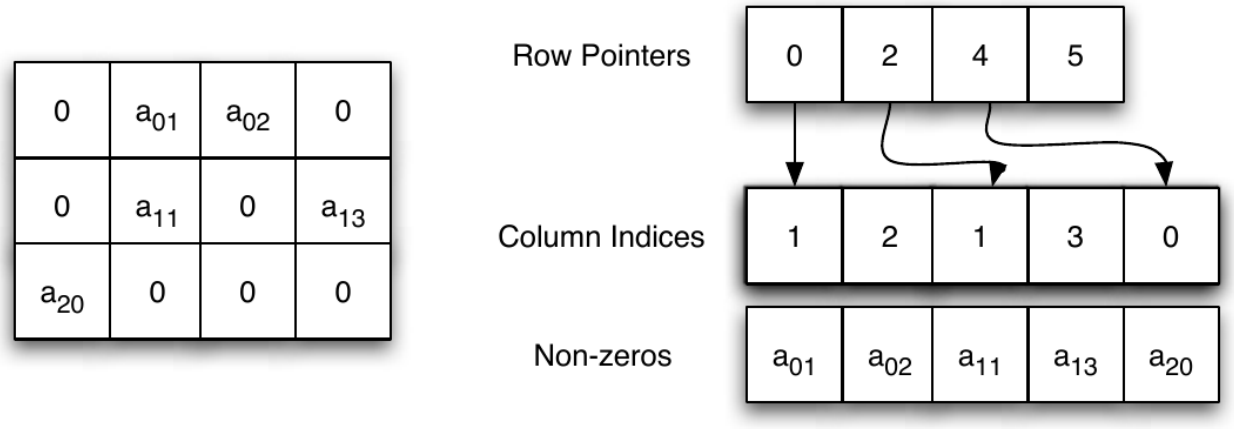


Figure 14.1: The CSC data structure. From Shah and Gilbert (2004). Remember that MATLAB is written in C, where the indexing starts at 0, and not 1.

The nature of statistical applications is such, that CSC representation is typically the most economical, justifying its popularity.

14.1.3 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.

14.1.4 Sparse Algorithms

We will go into the details of some algorithms in the Numerical Linear Algebra Chapter 17. For our current purposes two things need to be emphasized:

1. A mathematician may write $Ax = b \Rightarrow x = A^{-1}b$. A computer, however, would **never** compute A^{-1} in order to find x, but rather use one of many endlessly many numerical algorithms.

14.2 Sparse Matrices and Sparse Models in R

14.2.1 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 also 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.

We 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.

14.2.2 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.

```
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

  sparse.times <- system.time(fit1 <- glmnet(sx, y)) # fit with sparse glmnet
  full.times <- system.time(fit2 <- glmnet(x, y)) # fit with 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],
```

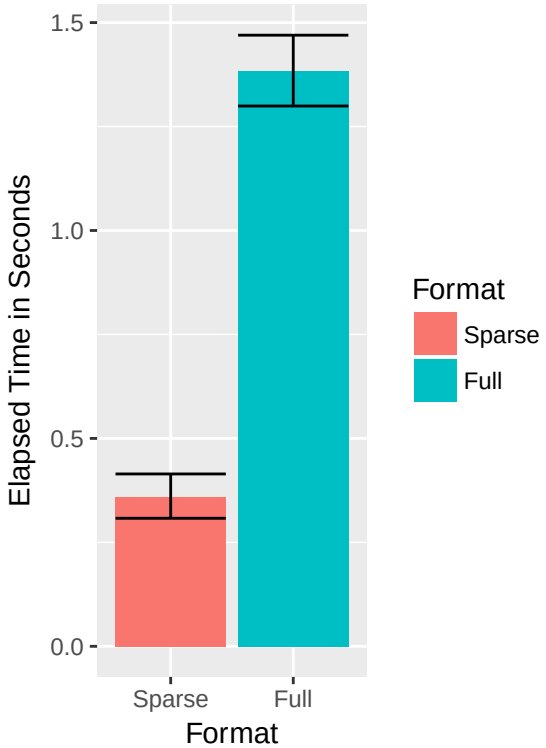
```
                                            Size = full.size))
}
```

Things to note:

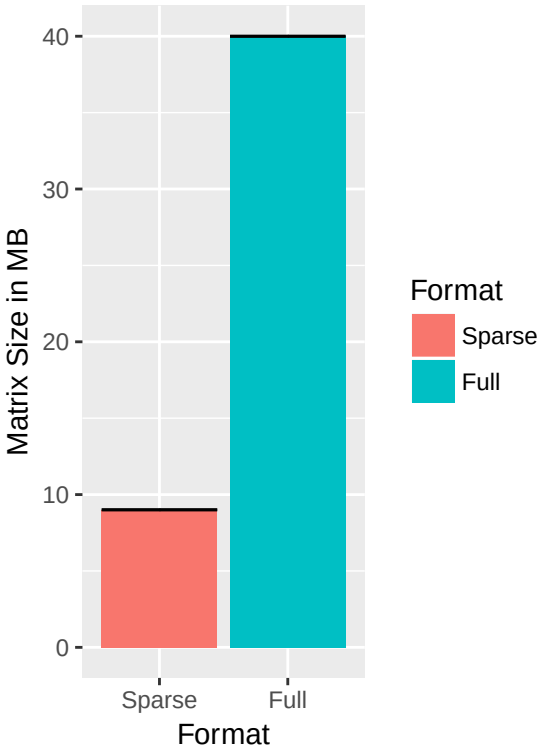
- The simulation calls `glmnet` twice. Once with the non-sparse object `x`, and once with its sparse version `sx`.
- The degree of sparsity of `sx` is 85%. We know this because we "injected" zeroes in 0.85 of the locations of `x`.
- Because `y` is continuous `glmnet` will fit a simple OLS model. We will 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 elasticnet regularization.

14.3 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).

Chapter 15

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 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 14, 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, the solution will typically be to replace the algorithm you are using. This will typically mean that 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. This will allow you to use the algorithms provided by your database, or let R use an EMA while loading from your database. Alternatively, and preferably, you may avoid using a database, 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 database, 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.

15.1 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 14, which is relevant when you have factors, which will typically map to sparse model matrices. Another way is to use memory efficient algorithms.

The `giblm::giblm` 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** packages.

15.1.1 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
##   6.528   0.000   6.530
```

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
##   0.320   0.000   0.322
```

The winner is obvious. Let's compare filteting (i.e. row subsets, i.e. SQL's SELECT).

```
system.time(
  tbl_df %>% filter(v1 == "1")
  )
```

```
##    user  system elapsed
##   0.016   0.000   0.016
```

```
system.time(
  tbl_dt[v1 == "1"]
  )
```

```
##    user  system elapsed
##   0.012   0.000   0.012
```

15.2 Computing from a Database

The early solutions to oversized data relied on storing your data in some relational database such as *MySQL*, *PostgresSQL*, *SQLite*, *H2*, *Oracle*, etc. Several R packages provide interfaces to these databases, such as **sqldf**, **RDBI**, **RSQite**. Some will even include the database as part of the package itself.

Storing your data in a database has the advantage that you can typically rely on database 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 database. For this reason, and others, we prefer to compute from efficient file structures, as described in Section 15.3.

If, however, you have a powerful database 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 13, will take care of translating the **dplyr** syntax, to the SQL syntax of the database. 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 starts an empty table, managed by SQLite, at the desired path.
- `copy_to` copies data from R to the database.
- Typically, setting up a database 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 database.

```
select(flights_sqlite, year:day, dep_delay, arr_delay)
```

```
## Source:   query [?? x 5]
## Database: sqlite 3.11.1 [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:   query [?? x 19]
## Database: sqlite 3.11.1 [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:   query [?? x 1]
## Database: sqlite 3.11.1 [my_db.sqlite3]
##
##     delay
##     <dbl>
## 1 1349.11
```

15.3 Computing From Efficient File Structrures

It is possible to save your data on your storage device, without the database layer to manage it. This has several advantages:

- You don't need to manage a database.
- You don't have the computational overhead of the database.
- You may optimize the file structure for statistical modelling, and not for join and summary operations, as in relational databases.

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 16.

	Reading	Transforming	Fitting
bigmemory	968.6	105.5	1501.7
ff	1111.3	528.4	1988.0
RRE	851.7	107.5	189.4

We now demonstrate the workflow of the **bigmemory** package.

```
# download.file("http://www.cms.gov/Research-Statistics-Data-and-Systems/Statistics-Trends-and-Reports/BSAPU
# unzip(zipfile="2010_Carrier_PUF.zip")

library("bigmemory")
x <- read.big.matrix("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, `airline.desc` 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.

15.4 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 database. We do not cover this topic here, and refer the reader to the **RHipe**, **RHadoop**, and **RSpark** packages and references therein.

15.5 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.

Chapter 16

Parallel Computing

You would think that because you have an expensive multicore computer your computations will speed up. Well, no. At least not if you don't make sure they do. By default, no matter how many cores you have, the operating system will allocate each R session to a single core.

For starters, we need to distinguish between two types of parallelism:

1. **Explicit parallelism**: where the user handles the parallelisation.
2. **Implicit parallelism**: where the compiler handles the nonparallel.

Clearly, implicit parallelism is more desirable, but the state of mathematical computing is such that no sufficiently general implicit parallelism framework exists. The R Consortium is currently financing a major project for a A "Unified Framework For Distributed Computing in R" so we can expect things to change soon. In the meanwhile, most of the parallel implementations are explicit.

16.1 Explicit Parallelism

R provides many frameworks for explicit parallelism. Because the parallelism is initiated by the user, we first need to decide **when to parallelize?** As a rule of thumb, you want to parallelise when you encounter a CPU bottleneck, and not a memory bottleneck. Memory bottlenecks are released with Sparsity (Chapter 14), or efficient representation (Chapter 15).

Several ways to diagnose your bottleneck include:

- Keep your Windows Task Manager, or Linux `top` open, and look for the CPU load.
- The computation takes a long time, and when you stop it pressing ESC, R is immediately responsive (if it is not immediately responsive, you have a memory bottleneck).
- Profile your code. See Hadley's guide.

For reasons detailed in Kane et al. (2013), we will present the **foreach** parallelisation package (Analytics and Weston, 2015). It will allow us to:

1. Decouple between our parallel algorithm and the parallelisation mechanism: we write parallelisable code once, and can then switch the underlying parallelisation mechanism.

2. Combine with the `big.matrix` object from Chapter 15 for *shared memory parallisation*: all the machines may see the same data, so that we don't need to export objects from machine to machine.

What do we mean by "switch the underlying parallesation mechanism"? It means there are several packages that will handle communication between machines. Some are very general and will work on any cluster that does not even need to be on the same machine. Some are more specific and will work only on a single multicore machine (not a cluster) with a particular operating system. These mechanisms include **multicore**, **snow**, **parallel**, and **Rmpi**. The compatibility between these mechanisms and **foreach** is provided by another set of packages: **doMC** , **doMPI**, **doRedis**, **doParallel**, and **doSNOW**.

Remark. I personally prefer the **multicore** mechanism, with the **doMC** adapter for **foreach**. I will not use this combo, however, because **multicore** will not work on Windows machines. I will thus use the more general **snow** and **doParallel** combo. If you do happen to run on Linux, or Unix, you will want to replace all **doParallel** functionality with **doMC**.

Let's start with a simple example, taken from "Getting Started with doParallel and foreach".

```
library(doParallel)
cl <- makeCluster(2)
registerDoParallel(cl)
result <- foreach(i=1:3) %dopar% sqrt(i)
class(result)
```

```
## [1] "list"
```

```
result
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] 1.414214
##
## [[3]]
## [1] 1.732051
```

Things to note:

- `makeCluster` creates an object with the information our cluster. One a single machine it is very simple. On a cluster of machines, you will need to specify the i.p. addresses or other identifiers of the machines.
- `registerDoParallel` is used to inform the **foreach** package of the presence of our cluster.
- The `foreach` function handles the looping. In particular note the `%dopar` operator that ensures that looping is in parallel. `%dopar%` can be replaced by `%do%` if you want serial looping (like the `for` loop), for instance, for debugging.
- The output of the various machines is collected by `foreach` to a list object.
- In this simple example, no data is shared between machines so we are not putting the shared memory capabilities to the test.
- We can check how many workers were involved using the `getDoParWorkers()` function.
- We can check the parallelisation mechanism used with the `getDoParName()` function.

Here is a more involved example. We now try to make Bootstrap inference on the coefficients of a logistic regression. Bootstrapping means that in each iteration, we resample the data, and refit the model.

```
x <- iris[which(iris[,5] != "setosa"), c(1,5)]
trials <- 1e4
ptime <- system.time({
 r <- foreach(icount(trials), .combine=cbind) %dopar% {
 ind <- sample(100, 100, replace=TRUE)
 result1 <- glm(x[ind,2]~x[ind,1], family=binomial(logit))
 coefficients(result1)
 }
 })[3]
```

```
## Warning: closing unused connection 8 (<-localhost:11728)
```

```
## Warning: closing unused connection 7 (<-localhost:11728)
```

```
## Warning: closing unused connection 6 (<-localhost:11728)
```

```
## Warning: closing unused connection 5 (<-localhost:11728)
```

```
ptime
```

```
## elapsed
```

```
##  20.218
```

Things to note:

- As usual, we use the `foreach` function with the `%dopar%` operator to loop in parallel.
- The `icounts` function generates a counter.
- The `.combine=cbind` argument tells the `foreach` function how to combine the output of different machines, so that the returned object is not the default list.

How long would that have taken in a simple (serial) loop? We only need to replace `%dopar%` with `%do%` to test.

```
stime <- system.time({
 r <- foreach(icount(trials), .combine=cbind) %do% {
 ind <- sample(100, 100, replace=TRUE)
 result1 <- glm(x[ind,2]~x[ind,1], family=binomial(logit))
 coefficients(result1)
 }
 })[3]
stime
```

```
## elapsed
##  37.555
```

Yes. Parallelising is clearly faster.

Let's see how we can combine the power of **bigmemory** and **foreach** by creating a file mapped `big.matrix` object, which is shared by all machines. The following example is taken from Kane et al. (2013), and uses the `big.matrix` object we created in Chapter 15.

```
library(bigmemory)
x <- attach.big.matrix("airline.desc")

library(foreach)
library(doSNOW)
cl <- makeSOCKcluster(rep("localhost", 4)) # make a cluster of 4 machines
registerDoSNOW(cl) # register machines for foreach()
```

Get a "description" of the `big.matrix` object that will be used to call it from each machine.

```
xdesc <- describe(x)
```

Split the data along values of `BENE_AGE_CAT_CD`.

```
G <- split(1:nrow(x), x[, "BENE_AGE_CAT_CD"])
```

Define a function that computes quantiles of `CAR_LINE_ICD9_DGNS_CD`.

```
GetDepQuantiles <- function(rows, data) {
 quantile(data[rows, "CAR_LINE_ICD9_DGNS_CD"], probs = c(0.5, 0.9, 0.99),
 na.rm = TRUE)
}
```

We are all set up to loop, in parallel, and compute quantiles of `CAR_LINE_ICD9_DGNS_CD` for each value of `BENE_AGE_CAT_CD`.

```
qs <- foreach(g = G, .combine = rbind) %dopar% {
 require("bigmemory")
 x <- attach.big.matrix(xdesc)
 GetDepQuantiles(rows = g, data = x)
}
qs
```

```
##          50% 90% 99%
## result.1 558 793 996
## result.2 518 789 996
```

```
## result.3 514 789 996
## result.4 511 789 996
## result.5 511 790 996
## result.6 518 796 995
```

16.2 Implicit Parallelism

We will not elaborate on implicit parallelism except mentioning the following:

- You can enjoy parallel linear algebra by replacing the linear algebra libraries with BLAS and LAPACK as described here.
- You should read the "Parallel computing: Implicit parallelism" section in the excellent High Performance Computing task view, for the latest developments in implicit parallelism.

16.3 Bibliographic Notes

For a brief and excellent explanation on parallel computing in R see Schmidberger et al. (2009). For a full review see Chapple et al. (2016). For an up-to-date list of packages supporting parallel programming see the High Performance Computing R task view.

Chapter 17

Numerical Linear Algebra

Chapter 18

Convex Optimization

Chapter 19

RCpp

Chapter 20

Writing Packages

Chapter 21

Bibliography

Bibliography

Analytics, R. and Weston, S. (2015). *foreach: Provides Foreach Looping Construct for R.* R package version 1.4.3.

Arlot, S., Celisse, A., et al. (2010). A survey of cross-validation procedures for model selection. *Statistics surveys*, 4:40–79.

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.

Chang, W., Cheng, J., Allaire, J., Xie, Y., and McPherson, J. (2017). *shiny: Web Application Framework for R.* R package version 1.0.0.

Chapple, S. R., Troup, E., Forster, T., and Sloan, T. (2016). *Mastering Parallel Programming with R.* Packt Publishing Ltd.

Conway, D. and White, J. (2012). *Machine learning for hackers.* " O'Reilly Media, Inc.".

Davis, T. A. (2006). *Direct methods for sparse linear systems.* SIAM.

Dowle, M. and Srinivasan, A. (2017). *data.table: Extension of 'data.frame'.* R package version 1.10.4.

Foster, D. P. and Stine, R. A. (2004). Variable selection in data mining: Building a predictive model for bankruptcy. *Journal of the American Statistical Association*, 99(466):303–313.

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.

Graham, R. (1988). Isometric embeddings of graphs. *Selected Topics in Graph Theory*, 3:133–150.

Greene, W. H. (2003). *Econometric analysis.* Pearson Education India.

Hotelling, H. (1933). Analysis of a complex of statistical variables into principal components. *Journal of educational psychology*, 24(6):417.

James, G., Witten, D., Hastie, T., and Tibshirani, R. (2013). *An introduction to statistical learning*, volume 6. Springer.

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.

Lantz, B. (2013). *Machine learning with R.* Packt Publishing Ltd.

Leisch, F. (2002). Sweave: Dynamic generation of statistical reports using literate data analysis. In *Compstat*, pages 575–580. Springer.

Leskovec, J., Rajaraman, A., and Ullman, J. D. (2014). *Mining of massive datasets.* Cambridge University Press.

McCullagh, P. (1984). Generalized linear models. *European Journal of Operational Research*, 16(3):285–292.

Mohri, M., Rostamizadeh, A., and Talwalkar, A. (2012). *Foundations of machine learning.* MIT press.

Pearson, K. (1901). Liii. on lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 2(11):559–572.

R Core Team (2016). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria.

Ripley, B. D. (2007). *Pattern recognition and neural networks*. Cambridge university press.

RStudio, Inc (2013). *Easy web applications in R*. URL: http://www.rstudio.com/shiny/.

Sammut, C. and Webb, G. I. (2011). *Encyclopedia of machine learning*. Springer Science & Business Media.

Sarkar, D. (2008). *Lattice: Multivariate Data Visualization with R*. Springer, New York. ISBN 978-0-387-75968-5.

Schmidberger, M., Morgan, M., Eddelbuettel, D., Yu, H., Tierney, L., and Mansmann, U. (2009). State of the art in parallel computing with r. *Journal of Statistical Software*, 47(1).

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.

Shalev-Shwartz, S. and Ben-David, S. (2014). *Understanding machine learning: From theory to algorithms*. Cambridge university press.

Sievert, C., Parmer, C., Hocking, T., Chamberlain, S., Ram, K., Corvellec, M., and Despouy, P. (2016). *plotly: Create Interactive Web Graphics via 'plotly.js'*. R package version 4.5.6.

Tukey, J. W. (1977). *Exploratory data analysis*. Reading, Mass.

Vapnik, V. (2013). *The nature of statistical learning theory*. Springer science & business media.

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.

Wickham, H. and Francois, R. (2016). *dplyr: A Grammar of Data Manipulation*. R package version 0.5.0.

Wickham, H., Hester, J., and Francois, R. (2016). *readr: Read Tabular Data*. R package version 1.0.0.

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.