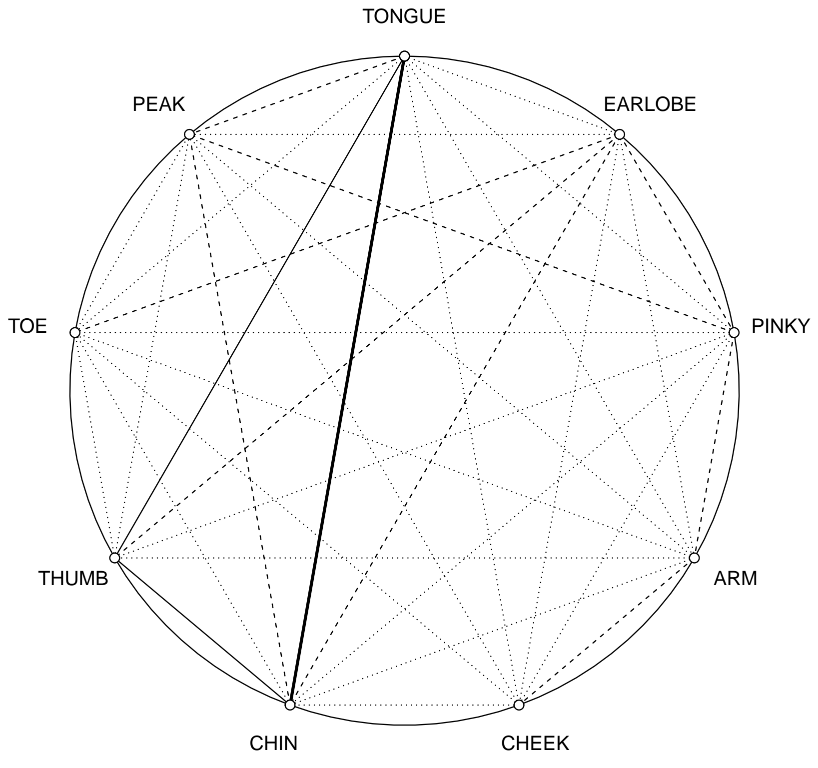
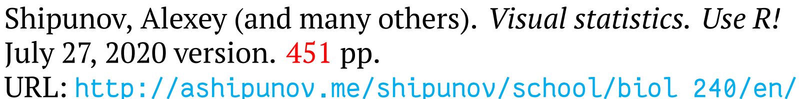
Visual Statistics
Use R!

Alexey Shipunov



July 27, 2020 version



Shipunov, Alexey (and many others). *Visual statistics. Use R!*
July 27, 2020 version. 451 pp.
URL: http://ashipunov.me/shipunov/school/biol_240/en/

On the cover: R plot illustrating correlations between human phenotypic traits. See the page 255 for more explanation.

This book is dedicated to the public domain

Contents

Foreword **9**

I One or two dimensions **13**

1 The data **14**
1.1 Origin of the data 14
1.2 Population and sample 14
1.3 How to obtain the data 15
1.4 What to find in the data 17
1.4.1 Why do we need the data analysis 17
1.4.2 What data analysis can do 17
1.4.3 What data analysis cannot do 18
1.5 Answers to exercises 19

2 How to process the data **20**
2.1 General purpose software 20
2.2 Statistical software 21
2.2.1 Graphical systems 21
2.2.2 Statistical environments 21
2.3 The very short history of the S and R 22
2.4 Use, advantages and disadvantages of the R 22
2.5 How to download and install R 23
2.6 How to start with R 25
2.6.1 Launching R 25
2.6.2 First steps 25
2.6.3 How to type 28
2.6.4 Overgrown calculator 30
2.6.5 How to play with R 32
2.7 R and data 34

2.7.1 How to enter the data from within R 34
2.7.2 How to name your objects 34
2.7.3 How to load the text data 35
2.7.4 How to load data from Internet 38
2.7.5 How to use `read.table()` properly 38
2.7.6 How to load binary data 40
2.7.7 How to load data from clipboard 41
2.7.8 How to edit data in R 42
2.7.9 How to save the results 43
2.7.10 History and scripts 44
2.8 R graphics 46
2.8.1 Graphical systems 46
2.8.2 Graphical devices 53
2.8.3 Graphical options 54
2.8.4 Interactive graphics 56
2.9 Answers to exercises 57

3 Types of data **62**
3.1 Degrees, hours and kilometers: measurement data 62
3.2 Grades and t-shirts: ranked data 68
3.3 Colors, names and sexes: nominal data 70
3.3.1 Character vectors 70
3.3.2 Factors 71
3.3.3 Logical vectors and binary data 76
3.4 Fractions, counts and ranks: secondary data 79
3.5 Missing data 84
3.6 Outliers, and how to find them 86
3.7 Changing data: basics of transformations 87
3.7.1 How to tell the kind of data 89
3.8 Inside R 89
3.8.1 Matrices 89
3.8.2 Lists 93
3.8.3 Data frames 96
3.8.4 Overview of data types and modes 103
3.9 Answers to exercises 106

4 One-dimensional data **111**
4.1 How to estimate general tendencies 111
4.1.1 Median is the best 111
4.1.2 Quartiles and quantiles 113
4.1.3 Variation 115
4.2 1-dimensional plots 118

4.3 Confidence intervals . . . 124
4.4 Normality . . . 128
4.5 How to create your own functions . . . 130
4.6 How good is the proportion? . . . 132
4.7 Answers to exercises . . . 135

5 Two-dimensional data: differences **148**
5.1 What is a statistical test? . . . 148
5.1.1 Statistical hypotheses . . . 149
5.1.2 Statistical errors . . . 149
5.2 Is there a difference? Comparing two samples . . . 151
5.2.1 Two sample tests . . . 151
5.2.2 Effect sizes . . . 166
5.3 If there are more than two samples: ANOVA . . . 169
5.3.1 One way . . . 169
5.3.2 More then one way . . . 183
5.4 Is there an association? Analysis of tables . . . 184
5.4.1 Contingency tables . . . 184
5.4.2 Table tests . . . 189
5.5 Answers to exercises . . . 201
5.5.1 Exercises on two samples . . . 201
5.5.2 Exercises on ANOVA . . . 208
5.5.3 Exercises on tables . . . 212

6 Two-dimensional data: models **221**
6.1 Analysis of correlation . . . 221
6.1.1 Plot it first . . . 222
6.1.2 Correlation . . . 224
6.2 Analysis of regression . . . 230
6.2.1 Single line . . . 230
6.2.2 Many lines . . . 242
6.2.3 More then one way, again . . . 248
6.3 Probability of the success: logistic regression . . . 250
6.4 Answers to exercises . . . 254
6.4.1 Correlation and linear models . . . 254
6.4.2 Logistic regression . . . 266
6.5 How to choose the right method . . . 271

II Many dimensions **274**

7 Draw **275**

7.1 Pictographs . . . 276
7.2 Grouped plots . . . 280
7.3 3D plots . . . 282

8 Discover **289**
8.1 Discovery with primary data . . . 290
8.1.1 Shadows of hyper clouds: PCA . . . 290
8.1.2 Correspondence . . . 298
8.1.3 Projections, unfolds, t-SNE and UMAP . . . 300
8.1.4 Non-negative matrix factorization . . . 304
8.2 Discovery with distances . . . 305
8.2.1 Distances . . . 305
8.2.2 Making maps: multidimensional scaling . . . 308
8.2.3 Making trees: hierarchical clustering . . . 311
8.2.4 How to know the best clustering method . . . 316
8.2.5 How to compare clusterings . . . 319
8.2.6 How good are resulted clusters . . . 322
8.2.7 Making groups: *k*-means and friends . . . 325
8.2.8 How to know cluster numbers . . . 328
8.2.9 Use projection pursuit for clustering . . . 332
8.2.10 How to compare different ordinations . . . 333
8.3 Answers to exercises . . . 334

9 Learn **341**
9.1 Learning with regression . . . 342
9.1.1 Linear discriminant analysis . . . 342
9.1.2 Recursive partitioning . . . 347
9.2 Ensemble learnig . . . 351
9.2.1 Random Forest . . . 351
9.2.2 Gradient boosting . . . 352
9.3 Learning with proximity . . . 354
9.4 Learning with rules . . . 357
9.5 Learning from the black boxes . . . 359
9.5.1 Support Vector Machines . . . 359
9.5.2 Neural Networks . . . 361
9.6 Semi-supervised learning . . . 366
9.7 How to choose the right method . . . 370
9.8 Answers to exercises . . . 373

Appendices 374

A Example of R session 375

A.1 Starting... 376
A.2 Describing... 377
A.3 Plotting... 380
A.4 Testing... 384
A.5 Finishing... 386
A.6 Answers to exercises 387

B Ten Years Later, or use R script 388

B.1 How to make your R script 389
B.2 My R script does not work! 391
B.3 Common pitfalls in R scripting 395
B.3.1 Advices 395
B.3.1.1 Use the Source, Luke!.. 395
B.3.1.2 Keep it simple 395
B.3.1.3 Learn to love errors and warnings 395
B.3.1.4 Subselect by names, not numbers 396
B.3.1.5 About reserved words, again 397
B.3.2 The Case-book of Advanced R user 397
B.3.2.1 A Case of Were-objects 397
B.3.2.2 A Case of Missing Compare 398
B.3.2.3 A Case of Outlaw Parameters 398
B.3.2.4 A Case of Identity 399
B.3.2.5 The Adventure of the Floating Point 400
B.3.2.6 A Case of Twin Files 400
B.3.2.7 A Case of Bad Grammar 401
B.3.2.8 A Case of Double Dipping 402
B.3.2.9 A Case of Factor Join 402
B.3.2.10 A Case of Bad Font 402
B.3.2.11 A Case of Disproportionate Condition 403
B.3.3 Good, Bad, and Not-too-bad 403
B.3.3.1 Good 403
B.3.3.2 Bad 404
B.3.3.3 Not too bad 406
B.4 Answers to exercises 407

C R fragments 409

C.1 R and databases 409
C.2 R and time 412
C.3 R and bootstrap 417

C.4 R and shape 423
C.5 R and Bayes 428
C.6 R, DNA and evolution 429
C.7 R and reporting 430
C.8 R without graphics 432
C.9 Answers to exercises 433

D Most essential R commands **438**

E The short R glossary **440**

F References **448**

G Reference card **451**

Foreword

This book is written for those who want to learn how to analyze data. This challenge arises frequently when you need to determine a previously unknown fact. For example: does this new medicine have an effect on a patient's symptoms? Or: Is there a difference between the public's rating of two politicians? Or: how will the oil prices change in the next week?

You might think that you can find the answer to such a question simply by looking at the numbers. Unfortunately this is often not the case. For example, after surveying 262 people exiting a polling site, it was found that 52% voted for candidate A and 48% for candidate B.

Do the results of this exit poll tell you that candidate A won the election?

Thinking about it, many would say "yes," and then, considering it for a moment, "Well, I don't know, maybe?" But there is a simple (from the point of view of modern computer programs) "proportion test" that tells you not only the answer (in this case, "No, the results of the exit poll do not indicate that Candidate A won the election") but also allows you to calculate how many people you would need to survey to be able to answer that question. In this case, the answer would be "about 5,000 people"—see the explanation at the end of the chapter about one-dimensional data.

The ignorance of the statistical methods can lead to mistakes and misinterpretations. Unfortunately, understanding of these methods is far from common. Many college majors require a course in probability theory and mathematical statistics, but all many of us remember from these courses is horror and/or frustration at complex mathematical formulas filled with Greek letters, some of them wearing hats.

It is true that probability theory forms the basis of most data analysis methods but on the other hand, most people use fridge without knowledge about thermodynamics and Carnot cycle. For the practical purposes of analyzing data, you do not have to be fully fluent in mathematical statistics and probability theory.

Therefore, we tried to follow Steven Hawking who in the "A Brief History of Time" stated that "... someone told me that each equation I included in the book would halve the sales. I therefore resolved not to have any equations at all ..". Consequently, there is *only one equation* in this book. By the way, an interesting exercise is just to **find**[1] it.

Even better, almost ideal approach would be the book similar to R. Munroe's "Thing Explainer"[2] where complicated concepts are explained using dictionary of 1,000 most frequent English words.

To make a long story short, this textbook is the kind of "statistic without math" but with R.

Some caution is required, though, for readers of such books: many methods of statistical analysis have, so to speak, a false bottom. You can apply these methods without delving too deeply into the underlying principles, get results, and discuss these results in your report. But you might find one day that a given method was totally unsuitable for the data you had, and therefore your conclusions are invalid. You must be careful and aware of the limitations of any method you try to use and determine whether they are applicable to your situation.

This book devoted mostly to *biometry*, methods of data analysis applied to biological objects. As Fred Bookstein mentioned in his book (2018), biological objects have at least two important features: morphological *integration* (substantial but inconstant correlation among most pairs of measurements), and the *centrality* of measures of extent (lengths, areas, weights). We will use these features in the book.

* * *

On examples: This book is based on a software which runs data files, and we have made most of the data files used here available to download from

`http://ashipunov.me/data`

We recommend to copy data files to the `data` subdirectory of your working directory; one of possible methods is to open this URL in browser and download all files. Then all code examples should work without Internet connection.

However, you can load data directly from the URL above. If you decide to work online, then the convention is that when the books says `"data/..."`, **replace** it with `"http://ashipunov.me/data/..."`.

---

[1] Usually, small (running) exercises are **boldfaced**.

[2] `https://xkcd.com/thing-explainer/`

Some data is available from `shipunov` package (see below), and also from from author's *open repository* at

http://ashipunov.me/shipunov/open

Most example problems in this book can and should be reproduced independently. These examples are written in `typewriter font` and begin with the > symbol. If an example *does not fit on one line*, a + sign indicates the line's continuation—so

do not type the + (and >) signs when reproducing the code!

All commands used in the text of this book are downloadable as one big R *script* (collection of text commands) from http://ashipunov.me/shipunov/school/biol_240/en/visual_statistics.r

The book also contain *supplements*, they are presented both as zipped and non-zipped folders here:

http://ashipunov.me/shipunov/school/biol_240/en/supp

* * *

Custom functions used in this book could be loaded using `shipunov` package. To install this package from CRAN, run this command in R (explanations will follow):

```
> install.packages("shipunov")
```

To use the most fresh, *development* version from the book Web site, macOS and Linux users should run:

```
> install.packages("http://ashipunov.me/r/shipunov.tar.gz",
+ repos=NULL)
```

whereas Windows (only!) users should run:

```
> install.packages("http://ashipunov.me/r/shipunov.zip",
+ repos=NULL)
```

You need to install this package only once. However, to upgrade the development version of the package, run this above command again in a way customary to your OS.

Then, if you want to use custom commands, load the package first:

```
> library(shipunov)
```

How do you know if the command is custom? They frequently title-cased, but more important is that they labeled in the text like:

```
> ... # shipunov
```

* * *

Of course, many statistical methods including really important ones are not discussed in this book. We almost completely neglect statistical modeling, do not discuss contrasts, do not examine standard distributions besides the normal, do not cover survival curves, factor analysis, geostatistics, we do not talk about how to do multi-factorial or block analysis of variation, multivariate and ordinal regression, design of experiments, and much else. The goal of the first part is just to explain *fundamentals* of statistical analysis with emphasis on biological problems. Having mastered the basics, more advanced methods can be grasped without much difficulty with the help of the scholarly literature, internal documentation, and on-line resources.

This book was first written and published in Russian. The leading author (Alexey Shipunov) of the Russian edition is extremely grateful to all who participated in writing, editing and translating. Some names are listed below: Eugene Baldin, Polina Volkova, Anton Korobeinikov, Sofia Nazarova, Sergei Petrov, Vadim Sufijanov, Alexandra Mushegjan. And many thanks to the editor, Yuta Tamberg who did a great job of the improving and clarifying the text.

Please note that *book is under development*. If you obtained it from somewhere else, do not hesitate to check for the update from the main location (look on the second page for URL).

Happy Data Analysis!

Part I

One or two dimensions

Chapter 1

The data

1.1 Origin of the data

He who would catch fish must find the water first, they say. If you want to analyze data, you need to obtain them. There are many ways of obtaining data but the most important are *observation* and *experiment*.

Observation is the method when observer has the least possible influence on the observed. It is important to understand that zero influence is practically impossible because the observer will always change the environment.

Experiment approaches the nature the other way. In the experiment, influence(s) are strictly controlled. Very important here are precise measurements of effects, removal of all interacting factors and (related) contrasting design. The latter means that one experimental group has no sense, there must be at least two, experiment (influence) and control (no influence). Only then we can equalize all possibly interacting factors and take into account solely the results of our influence. Again, no interaction is practically impossible since everything around us is structurally too complicated. One of the most complicated things are we humans, and this is why several special research methods like blind (when patients do not know what they receive, drug or placebo) or even double blind (when doctor also does not know that) were invented.

1.2 Population and sample

Let us research the simple case: which of two ice-creams is more popular? It would be relatively easy to gather all information if all these ice-creams sold in one shop.

However, the situation is usually different and there are many different sellers which are really hard to control. In situation like that, the best choice is **sampling**. We cannot control everybody but we can control somebody. Sampling is also cheaper, more robust to errors and gives us free hands to perform more data collection and analyses. However, when we receive the information from sampling, another problem will become apparent—how representative are these results? Is it possible to estimate the small piece of sampled information to the whole big *population* (this is not a biological term) of ice-cream data? Statistics (mathematical statistics, including the theory of sampling) could answer this question.

It is interesting that sampling could be more precise than the total investigation. Not only because it is hard to control all variety of cases, and some data will be inevitably mistaken. There are many situations when the smaller size of sample allows to obtain more detailed information. For example, in XIX century many Russian peasants did not remember their age, and all age-related total census data was rounded to tens. However, in this case selective but more thorough sampling (using documents and cross-questioning) could produce better result.

And philosophically, full investigation is impossible. Even most complete research is a subset, sample of something bigger.

1.3 How to obtain the data

There are two main principles of sampling: replication and randomization.

Replication suggests that the same effect will be researched several times. This idea derived from the cornerstone math “big numbers” postulate which in simple words is “the more, the better”. When you count replicates, remember that they must be independent. For example, if you research how light influences the plant growth and use five growing chambers, each with ten plants, then number of replicates is five, not fifty. This is because plants withing each chamber are not independent as they all grow in the same environment but we research differences between environments. Five chambers are replicates whereas fifty plants are *pseudoreplicates*.

Repeated measurements is another complication. For example, in a study of short-term visual memory ten volunteers were planned to look on the same specific object multiple times. The problem here is that people may remember the object and recall it faster towards the end of a sequence. As a result, these multiple times are not replicates, they are repeated measurements which could tell something about learning but not about memory itself. There are only ten true replicates.

Another important question is how many replicates should be collected. There is the immense amount of publications about it, but in essence, there are two answers:

(a) as many as possible and (b) 30. Second answer looks a bit funny but this rule of thumb is the result of many years of experience. Typically, samples which size is less than 30, considered to be a small. Nevertheless, even minuscule samples could be useful, and there are methods of data analysis which work with five and even with three replicates. There are also special methods (*power analysis*) which allow to estimate how many objects to collect (we will give one example due course).

Randomization tells among other that every object should have the equal chances to go into the sample. Quite frequently, researchers think that data was randomized while it was not actually collected in the random way.

For example, how to select the sample of 100 trees in the big forest? If we try simply to walk and select trees which somehow attracted the attention, this sample will not be random because these trees are somehow deviated and this is why we spotted them. Since one of the best ways of randomization is to *introduce the order which is knowingly absent in nature* (or at least not related with the study question), the reliable method is, for example, to use a detailed map of the forest, select two random coordinates, and find the tree which is closest to the selected point. However, trees are not growing homogeneously, some of them (like spruces) tend to grow together whereas others (like oaks) prefer to stay apart. With the method described above, spruces will have a better chance to come into sample so that breaks the rule of randomization. We might employ the second method and make a transect through the forest using rope, then select all trees touched with it, and then select, saying, every fifth tree to make a total of hundred.

> Is the last (second) method appropriate? How to improve it?

Now you know enough to answer another question:

> Once upon a time, there was an experiment with a goal to research the effect of different chemical poisons to weevils. Weevils were hold in jars, chemicals were put on fragments of filter paper. Researcher opened the jar, then picked up the weevil which first came out of jar, put it on the filter paper and waited until weevil died. Then researcher changed chemical, and start the second run of experiment in the same dish, and so on. But for some unknown reason, the first chemical used was always the strongest (weevils died very fast). Why? How to organize this experiment better?

1.4 What to find in the data

1.4.1 Why do we need the data analysis

Well, if everything is so complicated, why to analyze data? It is frequently evident the one shop has more customers than the other, or one drug is more effective, and so on... —This is correct, but only to the some extent. For example, this data

```
2 3 4 2 1 2 2 0
```

is more or less self-explanatory. It is easy to say that here is a tendency, and this tendency is most likely 2. Actually, it is easy to use just a brain to analyze data which contains 5–9 objects. But what about this data?

```
88  22  52  31  51  63  32  57  68  27  15
20  26   3  33   7  35  17  28  32   8  19
60  18  30 104   0  72  51  66  22  44  75
87  95  65  77  34  47 108   9 105  24  29
31  65  12  82
```

(This is the real-word example of some flowers measurements in orchids, you can download it from the book data folder as `dact.txt`.)

It is much harder to say anything about tendency without calculations: there are too many objects. However, sometimes the big sample is easy enough to understand:

```
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2
```

Here everything is so similar than again, methods of data analysis would be redundant.

As a conclusion, we might say that statistical methods are wanted in cases of (1) numerous objects and/or (2) when data is not uniform. And of course, if there are not one (like in examples above) but several variables, our brain does not handle them easily and we again need statistics.

1.4.2 What data analysis can do

1. First of all, data analysis can characterize samples, reveal **central tendency** (of course, if it is here) and **variation**. You may think of them as about target and deviations.

2. Then, data analysis reveals **differences** between samples (usually two samples). For example, in medicine it is very important to understand if there is a difference between physiological characteristics of two groups of patients: those who received the drug of question, and those who received the placebo. There is no other way to understand if the drug works. Statistical tests and effect size estimations will help to understand the reliability of difference numerically.

3. Data analysis might help in understanding *relations* within data. There plenty of relation types. For example, **association** is the situation when two things frequently occur together (like lightning and thunder). The other type is **correlation** where is the way to measure the strength and sign (positive or negative) of relation. And finally, **dependencies** allow not only to spot their presence and to measure their strength but also to understand direction and **predict** the value of effect in unknown situations (this is a *statistical model*).

4. Finally, data analysis might help in understating the **structure** of data. This is the most complicated part of statistics because structure includes multiple objects and multiple variables. The most important outcome of the analysis of structure is **classification** which, in simple words, is an ultimate tool to understand world around us. Without proper classification, most of problems is impossible to resolve.

All of the methods above include both **description** (visualization) methods—which explain the situation, and **inferential** methods—which employ probability theory and other math. Inferential methods include many varieties (some of them explained below in main text and in appendices), e.g., *parametric* and *nonparametric* methods, *robust* methods and *re-sampling* methods. There are also analyses which fall into several of these categories.

1.4.3 What data analysis cannot do

1. Data analysis cannot read your mind. You should start data analysis only if you know what is your data, and which exact questions you need to answer.

2. Data analysis cannot give you certainty. Most inferential methods are based on the theory of *probability*.

3. Data analysis does not reflect the world perfectly. It is always based on a *sample*.

1.5 Answers to exercises

Answer to the exercise about tree sampling. In case of transect, spruces still have a better chance to be selected. Also, this forest could have some specific structure along the transect. So to improve method, one can use several transects and increase distances between selected trees.

* * *

Answer to the weevil question. In that case, first were always most active insects which piked the lethal dose of the chemical mush faster than less active individuals. Rule of replication was also broken here because one dish was used for the sequence of experiments. We think that if you read this explanation and understand it, it already became clear how to improve the experiment.

Chapter 2

How to process the data

Generally, you do not need a computer to process the data. However, contemporary statistics is "heavy" and almost always requires the technical help from some kind of software.

2.1 General purpose software

Almost every computer or smart phone has the **calculator**. Typically, it can do simple arithmetics, sometimes also square roots and degrees. This is enough for the basic data processing. However, to do any statistical analysis, such calculator will need statistical tables which give approximate values of *statistics*, special characteristics of data distribution. Exact calculation of these statistics is too complicated (for example, it might require integration) and most programs use embedded statistical tables. Calculators usually do not have these tables. Even more important disadvantage of the calculator is absence of the ability to work with sequences of numbers.

To deal with many numbers at once, **spreadsheets** were invented. The power of spreadsheet is in data visualization. From the spreadsheet, it is easy to estimate the main parameters of data (especially if the data is small). In addition, spreadsheets have multiple ways to help with entering and converting data. However, as spreadsheets were initially created for the accounting, they oriented still to the tasks typical to that field. If even they have statistical functions, most of them are not contemporary and are not supported well. Multivariate methods are frequently absent, realization of procedures is not optimal (and frequently hidden from the user), there is no specialized reporting system, and so on.

And thinking of data visualization in spreadsheets—what if the data do not fit the window? In that case, the spreadsheet will start to prevent the understanding of data instead of helping it.

Another example—what if you need to work simultaneously with three non-neighboring columns of data? This is also extremely complicated in spreadsheets.

This is why specialized statistical software come to the scene.

2.2 Statistical software

2.2.1 Graphical systems

There are two groups of statistical software. First, *graphical systems* which at a glance do not differ much from spreadsheets but supplied with much more statistical functions and have the powerful graphical and report modules. The typical examples are SPSS and MiniTab.

As all visual systems, they are flexible but only within the given range. If you need something new (new kind of plot, new type of calculation, unusual type of data input), the only possibility is to switch to non-visual side and use macros or subprograms. But even more important is that visual ideology is not working well with more than one user, and does not help if the calculation should be repeated in different place with different people or several years after. That breaks *reproducibility*, one of the most important principle of science. Last but not least, in visual software statistical algorithms are hidden from end-user so if even you find the name of procedure you want, it is not exactly clear what program is going to do.

2.2.2 Statistical environments

This second group of programs uses the command-line interface (CLI). User enters commands, the system reacts. Sounds simple, but in practice, statistical environments belong to the most complicated systems of data analysis. Generally speaking, CLI has many disadvantages. It is impossible, for example, to choose available command from the menu. Instead, user must *remember* which commands are available. Also, this method is so similar to programming that users of statistical environments need to have some programming skills.

As a reward, the user has the *full control* over the system: combine all types of analysis, write command sequences into scripts which could be run later at any time, modify graphic output, easily extend the system and if the system is open source, modify the core statistical environment. The difference between statistical environ-

ment and graphical system is like the difference between supermarket and vending machine!

SAS is the one of the most advanced and powerful statistical environments. This commercial system has extensive help and the long history of development. Unfortunately, SAS is frequently overcomplicated even for the experienced programmer, has many "vestiges" of 1970s (when it was written), closed-source and extremely expensive...

2.3 The very short history of the S and R

R is the statistical environment. In was created as a freeware analog of commercial S-Plus which is in turn was implementation of the S language concept. The S language was first created in 1976 in Bell Labs, and its name was inspired by famous C language (from same Bell Labs). S-Plus started in the end of 1980s, and as many statistical software, was seriously expensive. In August 1993, two New Zealand scientists, Robert Gentleman and Ross Ihaka, decided to make R (this name was, in turn, inspired by S). The idea was to make independent realization of S language concept which would differ from S-Plus in some details (for example, in the way it works with local and global variables).

Practically, R is not an imitation of S-Plus but the new "branch" in the family of S software. In 1990s, R was developing slowly, but when users finally realized its truly amazing opportunities (like the system of R extensions—*packages*, or *libraries*) and started to migrate from other statistical systems, R started to grow exponentially. Now, there are thousands of R packages, and R is used almost everywhere! Without any exaggeration, *R is now the most important software tool for data analysis*.

2.4 Use, advantages and disadvantages of the R

R is used everywhere to work with any kind of data. R is capable to do not only "statistics" in the strict sense but also *all kinds of data analysis* (like visualization plots), *data operations* (similar to databasing) and even *machine learning* and *advanced mathematical modeling* (which is the niche of other software like Python modules, Octave or MATLAB).

There are several extremely useful features of R: *flexibility*, *reproducibility*, *open source* code and (yes!) *command-line interface*. Flexibility allows to create extension packages almost for all purposes. For the common user, it means that almost everything which was described in statistical literature as a method, is available in R. And people who professionally work in the creation of statistical methods, use R for their

research. And (this is rare case) if the method is not available, it is possible to write yourself commands implementing it.

Reproducibility allow to repeat the same analysis, without much additional efforts, with the updated data, or ten years later, or in other institutions.

Openness means that it is always possible to look inside the code and find out how exactly the particular procedure was implemented. It is also possible to correct mistakes in the code (since everything made by humans have mistakes and R is not an exception) in Wikipedia-like communal way.

Command-line interface (CLI) of R is in truth, superior way over GUI (graphical user interface) of other software. User of GUI is just like the ordinary worker whereas CLI user is more similar to foreman who leaves the "dirty work" to the computer, and this is exactly what computers were invented for. CLI also allows to make interfaces, connect R with almost any software.

There is also the R "dark side". R *is difficult to learn*. This is why you are reading this book. After you install R, you see the welcome screen with a > prompt, and that is it. Many commands are hard to remember, and there are no of almost no menus. Sometimes, it is really complicated to find how to do something particular.

* * *

As a difference from S-Plus, R makes all calculations in the operational memory. Therefore if you accidentally power off the computer, all results not written on disk intentionally, will be lost[1].

2.5 How to download and install R

Since R is free, it is possible to download and install it without any additional procedures. There are several ways to do that depending on your operation system, but generally one need to google the uppercase letter "R" which will give the link to the site of R project. Next step is to find there "CRAN", the on-line repository of all R-related software. In fact, there are multiple repositories (mirrors) so next step is to choose the nearest mirror. Then everything is straightforward, links will finally get you to the downloading page.

[1] There is however the SOAR package which overrides this behavior.

If your operating system has the package manager, software center of similar, installing R is even simpler. All that you need is to find R from within the manager and click install[2].

As of 2019, one can install R on Android smartfones. To do that, install Termux application and add its-pointless repository (to know how, check Termux Wiki). You might want also to install Hacker's Keyboard app. However, situation in this field changes rapidly.

Under Windows, R might be installed in two different modes, "one big window with smaller windows inside" (*MDI*) or "multiple independent windows" (*SDI*). We recommended to *use the second (SDI)* as R in other operating systems can work only in SDI mode. It is better to determine the mode during the installation: to make this choice, choose "Custom installation" from the one of first screens. If for some reason, you skipped it, it may be done later, through the menu (R GUI options).

Apart from "graphical" R, both Windows and macOS have terminal R applications. While the functionality of Windows terminal programs is limited, on macOS it runs in a way similar to Linux and therefore makes the appropriate alternative. To start using this terminal R application in macOS, user should run any available terminal (like Terminal.app) first.

There are useful features in macOS graphical R, but also restrictions, especially with saving history of commands (see below). When R is installed on macOS GUI, it is better to uncheck Read history and check Save workspace -> No. On R with macOS Terminal.app, this is not necessary.

* * *

If you are going to work with R, you might enjoy also some companion software: Geany (or Kate) as a text editor, LibreOffice Calc as spreadsheet, muCommander (or Double Commander) as two-panel file manager, Inkscape as vector editor, and also good monospace font like Ubuntu Mono, Hack, or Fantasque Sans Mono. All of these should work on three main OSes and also should be freely available online.

If you use macOS Terminal.app or R on Linux, then useful software are nano text editor (should be installed by default), Midnight Commander two-panel file manager (mc, check how to install it on your OS) and bash command shell with saving history between sessions "on" (this is only for macOS, check online how). On macOS, in addition to the core R, it is recommended to install also XQuartz software.

---

[2] If you do not use these managers or centers, it is recommended to regularly *update* your R, at least once a year.

For the beginner in R, it is better to avoid any R GUI as they hamper the learning process.

2.6 How to start with R

2.6.1 Launching R

Typically, you launch R from the desktop icon or application menu. To launch R from the terminal, type:

```
$ R
```

—and you will see the R screen.

It is even possible to launch R on the remote UNIX server without any graphical system running. In that case, all plots will be written in one PDF file, `Rplots.pdf` which will appear in the working directory.

If you know how to work with R, it is a good idea to check the fresh installation typing, for example, `plot(1:20)` to check if graphics works. If you are a novice to R, proceed to the next section.

2.6.2 First steps

After you successfully opened R, it is good to understand how to exit. After entering empty parentheses, be sure to press `Enter` and answer "n" or "No" on the question:

```
> q()
Save workspace image? [y/n/c]: n
```

This simple example already shows that any *command* (or *function*, this is almost the same) in R has an *argument* inside round brackets, parentheses. If there is no argument, you still need these parentheses. If you forget them, R will show the *definition* of the function instead of quitting:

```
> q
function (save = "default", status = 0, runLast = TRUE)
.Internal(quit(save, status, runLast))
<bytecode: 0x28a5708>
<environment: namespace:base>
```

(For the curious, "bytecode" means that this function was compiled for speed, "environment" shows the way to call this function. If you want to know the function code, it is not always work to call it without parentheses; see the reference card for more advanced methods.)

* * *

How to know more about function? Call the *help*:

```
> help(q)
```

or simply

```
> ?q
```

And you will see the separate window or (under `Linux`) help text in the same window (to exit this help, press `q`)[3].

> But what if *you do not know your command*, and only know what to do? What to do if you were told to perform, saying, analysis of variation (ANOVA) in `R` but do not know which command to use? (*See the answer in the end of chapter*.)

Now back to the `?q`. If you read this help text thoroughly, you might conclude that to quit `R` *without being asked anything*, you may want to enter `q("no")`. Please **try it**.

`"no"` is the *argument* of the exit function `q()`. Actually, not exactly the argument but its *value*, because in some cases you can skip the *name* of argument. The name of argument is `save` so you can type `q(save="no")`. In fact, most of `R` functions look like `function(name="value")`; see more detail in Fig. 2.1.

By the way, on `Linux` systems you may replace `q()` command with `Ctrl+D` key, and on `Windows` with `Crtl+Z` key sequence.

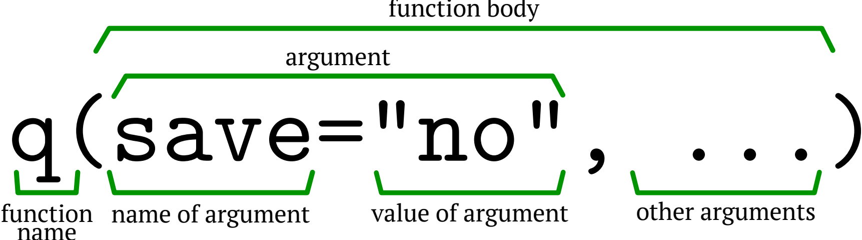


Figure 2.1: Structure of `R` command.

`R` is pretty liberal about arguments. You will receive same answers if you enter any of these variants:

[3]There is command `Xpager()` in the `shipunov` package, it allows to see help in the separate window even if you work in terminal.

```
> round(1.5, digits=0)
[1] 2
> round(1.5, d=0)
[1] 2
> round(d=0, 1.5)
[1] 2
> round(1.5, 0)
[1] 2
> round(1.5,)
[1] 2
> round(1.5)
[1] 2
```

([4] As you see, arguments are matched by *name* and/or by *position*. In output, R frequently prints something like [1], it is just an *index* of resulted number(s). What is round()? **Run** ?round to find out.)

It is possible to mess with arguments as long as R "understands" what you want. Please experiment more yourself and **find out** why this

```
> round(0, 1.5)
[1] 0
```

gives the value you probably do not want.

* * *

If you want to *know arguments* of some function, together with their default values, run args():

```
> args(round)
function (x, digits = 0)
NULL
> args(q)
function (save = "default", status = 0, runLast = TRUE)
NULL
```

There is also an example() function which is quite useful, especially when you learn plotting with R. To run examples supplied with the function, type example(function). Also do not forget to check demo() function which outputs the list of possible *demonstrations*, some of them are really handy, saying, demo(colors).

[4] Within parentheses immediately after example, we are going to provide comments.

Here R shows one of its basic principles which came from Perl language: *there always more than one way to do it*. There are many ways to receive a help in R!

* * *

So default is to ask the "save" question on exit. But why does R ask it? And what will happen if you answer "yes"? In that case, two files will be written into the R working directory: binary file `.RData` and textual file `.Rhistory` (yes, their names start with a dot). First contains all objects you created during the R session. Second contains the full history of entered commands. These files will be *loaded automatically* if you start R from the same directory, and the following message will appear:

```
[Previously saved workspace restored]
```

Frequently, this is *not* a desirable behavior, especially if you are just learning R and therefore often make mistakes. As long as you study with this book, **we strongly recommend to answer "no"**.

If you by chance answered "yes" on the question in the end of the previous R session, you might want to remove unwanted files:

```
> unlink(c(".RData", ".Rhistory"))
```

(*Be extremely careful* here because R deletes files silently! On macOS, file names might be different; in addition, it is better to uncheck Read history file on startup in the Preferences menu.)

If you are bored from answering final questions again and again, and at the same time do not want to enter `q("no")`, there is a third way. Supply R starting command with option `--no-save` (it could be done differently on different operation systems), and you will get rid of it.

2.6.3 How to type

When you work in R, the previous command could be called if you press "arrow up" key (↑). This is extremely useful and saves plenty of time, especially when you need to run the command similar to the preceding.

R on Windows, Linux and macOS Terminal.app (but not GUI) integrates readline command line editing environment. You can think of readline as of rudimentary (but still powerful) text editor which works only with current line of text. Especially useful readline key sequences are:

Ctrl+U to "kill" (delete everything) to the beginning of line (and Ctrl+Y to "yank" it back), this is useful if you mistakenly typed the long command and want to wipe it;

Ctrl+A to move the cursor to the beginning of line and Ctrl+K to "kill" (delete everything) to the end of line;

Ctrl+E to move the cursor to the end of line;

Ctrl+L to clean the screen.

There are many more useful combinations; to know them, check out any "readline cheatsheet" online.

If on Linux you run R in the terminal without scroll bar, the key sequences Shift+PgUp and Shift+PgDn typically help to scroll. Linux also has *backward search* feature (Ctrl+R) which is even more efficient than arrow up.

Another really helpful key is the Tab. To see how it works, start to type long command like read.t ... and then press Tab. It will call *completion* with suggests how to continue. Completion works not only for commands, but also for objects, command arguments and even for file names! To invoke the latter, start to type read.table(" and then press Tab once or twice; all files in the working directory will be shown.

Remember that all brackets (braces, parentheses) *must be always closed*. One of the best ways to make sure of it is to *enter opening and closing brackets together*, and then return your cursor into the middle. Actually, graphic R on macOS does this by default.

Pair also all quotes. R accepts two types of quotes, single '...' and double "..." but they *must be paired with quote of the same type*. The backtick (`) in R is not a quote but a symbol with special meaning.

Good question is when do you need quotes. In general, *quotes belong to character strings*. Rule of thumb is that objects *external* to R need quotes whereas *internal* objects could be called without quotes.

R is sensitive to the case of symbols. Commands ls() and Ls() are *different*! However, spaces do not play any role. These commands are the same:

```
> round   (1.5, digits=0)
> round(1.5, digits=0)
> round  (  1.5 , digits = 0  )
```

* * *

Do not be afraid of making errors. On the contrary—

Make as many mistakes as possible!

The more mistakes you do when you learn, the less you do when you start to work with R on your own.

R is frequently literal when it sees a mistake, and its *error messages* will help you to decipher it. Conversely, R is perfectly silent when you do well. If your input has no errors, R usually says *nothing*.

It is by the way really hard to crash R. If nevertheless your R seems to hang, press Esc button (on Linux, try Ctrl+C instead).

Yet another appeal to users of this book:

Experiment!

Try unknown commands, change options, numbers, names, remove parentheses, load any data, run code from Internet, from help, from your brain. The more you experiment, the better you learn R.

2.6.4 Overgrown calculator

The most simple way is to use R as an advanced calculator:

```
> 2+2
[1] 4
> 2+.2
[2] 2.2
```

(Note that you can skip leading zero in decimal numbers.)

The more complicated example, "`log10(((sqrt(sum(c(2, 2))))^2)*2.5)`" will be calculated as follows:

1. The vector will me created from two twos: `c(2, 2)`.
2. The sum of its elements will be counted: `2+2=4`.
3. Square root calculated: `sqrt(4)=2`.
4. It is raised to the power of 2: `2^2=4`.
5. The result is multiplied by 2.5: `4*2.5=10`.
6. Decimal logarithm is calculated: `log10(10)=1`.

As you see, it is possible to embed pairs of parentheses. It is a good idea to count opening and closing parentheses before you press `Enter`; these numbers must be *equal*. After submission, R will open them, pair by pair, from the deepest pair to the most external one.

So R expressions are in some way similar to Russian doll, or to onion, or to artichoke (Fig. 2.2), and to analyze them, one should peel it.

Figure 2.2: You may think of R syntax as of "artichoke".

Here is also important to say that R (similar to its TEX friend) belongs to one of the most deeply thought software. In essence, R "base" package covers almost 95% needs of the common statistical and data handling work and therefore external tools are often redundant. It is wise to keep things simple with R.

* * *

If there are no parentheses, R will use precedence rules which are similar to the rules known from the middle school.

For example, in `2+3*5` R will multiply first (`3*5=15`), and only then calculate the sum (`2+15=17`). Please **check** it in R yourself. How to make the result 25? Add parentheses.

* * *

Let us feed something mathematically illegal to R. For example, square root or logarithm of –1:

```
> log(-1)
[1] NaN
Warning message:
In log(-1) : NaNs produced
```

If you thought that R will crash, that was wrong. It makes `NaN` instead. `NaN` is *not a number*, one of *reserved words*.

By the way, warnings in general are neither good nor bad. There is a simple rule about them: if you understand the warning, do something (or *decide* not to do); if you do not understand warning—ignore it.

What about division by zero?

```
> 100/0
[1] Inf
```

This is another reserved word, `Inf`, *infinity*.

2.6.5 How to play with R

Now, when we know basics, it is time to do something more interesting in R. Here is the simple task: convert the sequence of numbers from 1 to 9 into the table with three columns. In the spreadsheet or visual statistical software, there will be several steps: (1) make two new columns, (2–3) copy the two pieces into clipboard and paste them and (4) delete extra rows. In R, this is just one command:

```
> bb <- matrix(1:9, ncol=3)
> bb
     [, 1] [, 2] [, 3]
[1, ]    1    4    7
[2, ]    2    5    8
[3, ]    3    6    9
```

(Symbol `<-` is an *assignment* operator, it is read *from right to left*. `bb` is a new R *object* (it is a good custom to name objects with double letters, less chances to intersect with existent R oblects). But what is `1:9`? **Find it** yourself. Hint: it is explained in few pages from this one.)

Again from the above: How to select the sample of 100 trees in the big forest? If you remember, our answer was to produce 100 random pairs of the coordinates. If this forest is split into 10,000 squares (100×100), then required sample might look like:

```
> coordinates <- expand.grid(1:100, 1:100)
> sampled.rows <- sample(1:nrow(coordinates), 100)
> coordinates[sampled.rows, ]
```

```
     Var1 Var2
6751   51   68
5370   70   54
668    68    7
...
```

(First, `expand.grid()` was used above to create all 10,000 combinations of square numbers. Then, powerful `sample()` command randomly selects 100 rows from whatever number of rows is in the table `coordinates`. Note that your results will be likely different since `sample()` uses the *random number generator*.

Command `sample()` was used to create new `samples.rows` object. Finally, this last object was used as an *index* to randomly select 100 rows (pairs of coordinates) from 10,000 combinations. What is left for you now is to go to the forest and find these trees `:-)`)

Let us now play dice and cards with R:

```
> dice <- as.vector(outer(1:6, 1:6, paste))
> sample(dice, 4, replace=TRUE)
[1] "6 4" "4 5" "6 6" "3 5"
> sample(dice, 5, replace=TRUE)
[1] "3 2" "3 1" "3 1" "6 4" "2 3"
...
> cards <- paste(rep(c(6:10,"V","D","K","T"), 4),
+ c("Tr","Bu","Ch","Pi"))
> sample(cards, 6)
[1] "V Tr" "6 Tr" "9 Bu" "T Tr" "T Bu" "8 Bu"
> sample(cards, 6)
[1] "K Tr" "K Ch" "D Bu" "T Pi" "T Tr" "7 Ch"
...
```

(Note here `outer()` command which combines values, `paste()` which joins into the text, `rep()` which repeats some values, and also the `replace=TRUE` argument (by default, `replace` is `FALSE`). What is `replace=FALSE`? Please **find out**. Again, your results could be different from what is shown here. Note also that `TRUE` or `FALSE` must always be fully uppercased.)

2.7 R and data

2.7.1 How to enter the data from within R

We now need to know how to enter data into R. Basic command is `c()` (shortcut of the word *concatenate*):

```
> c(1, 2, 3, 4, 5)
[1] 1 2 3 4 5
```

However, in that way your numbers will be forgotten because R does not remember anything which is not saved into *object*:

```
> aa <- c(2, 3, 4, 5, 6, 7, 8)
> aa
[1] 2 3 4 5 6 7 8
```

(Here we *created* an object `aa`, *assigned* to it vector of numbers from one to five, and then *printed* object with typing its name.)

If you want to create and print object simultaneously, use external parentheses:

```
> (aa <- c(1, 2, 3, 4, 5, 6, 7, 8, 9))
[1] 1 2 3 4 5 6 7 8 9
> aa
[1] 1 2 3 4 5 6 7 8 9
```

(By the way, here we created `aa` object *again*, and R *silently* re-wrote it. R never gives a warning if object already exists!)

In addition to `c()`, we can use commands `rep()`, `seq()`, and also the colon (`:`) *operator*:

```
> rep(1, 5)
[1] 1 1 1 1 1
> rep(1:5, each=3)
[1] 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5
> seq(1, 5)
[1] 1 2 3 4 5
> 1:5
[1] 1 2 3 4 5
```

2.7.2 How to name your objects

R has no strict rules on the naming your objects, but it is better to follow some guidelines:

1. Keep in mind that R is *case-sensitive*, and, for example, X and x are different names.

2. For objects, use only English letters, numbers, dot and (possibly) underscore. Do not put numbers, underscores and dots in the beginning of the name. One of recommended approaches is double-letter (or triple-letter) when you name objects like aa, jjj, xx and so on.

3. In data frames, we recommend to name your columns (characters) with *uppercase letters and dots*. The examples are throughout of this book.

4. Do not reassign names already given to popular functions (like c()), reserved words (especially T, F, NA, NaN, Inf and NULL) and predefined objects like pi[5], LETTERS and letters. If you accidentally did it, there is conflicts() function to help in these situations. To see all reserved words, type ?Reserved.

2.7.3 How to load the text data

In essence, data which need to be processed could be of two kinds: *text* and *binary*. To avoid unnecessary details, we will accept here that *text data* is something which you can read and edit in the *simple text editor* like Geany[6]. But if you want to edit the *binary data*, you typically need a program which outputted this file in the past. Without the specific software, the binary data is not easy to read.

Text data for the statistical processing is usually text tables where every row corresponds with the table row, and columns are separated with *delimiters*, either invisible, like spaces or tab symbols, or visible, like commas or semicolons. If you want R to "ingest" this kind of data, is is necessary to make sure first that the data file is located within the same directory which R regards as a *working directory*:

```
> getwd()
[1] "d:/programs/R/R-3.2.3"
```

If this is not the directory you want, you can change it with the command:

```
> setwd("e:\\wrk\\temp") # Windows only!
> getwd()
[1] "e:/wrk/temp"
```

Note how R works with backslashes under Windows. Instead of one backslash, you need to enter *two*. Only in that case R under Windows will understand it. It is also possible to use slashes under Windows, similar to Linux and macOS:

[5] By the way, if you want the Euler number, *e*, type exp(1).

[6] And also like editor which is embedded into R for Windows or into R macOS GUI, but **not** office software like MS Word or Excel!

```
> setwd("e:/wrk/temp")
> getwd()
[1] "e:/wrk/temp"
```

Please always **start** each of your R session from **changing working directory**. Actually, it is not absolutely necessary to remember long paths. You can copy it from your file manager into R. Then, graphical R under Windows and macOS have rudimentary menu system, and it is sometimes easier to *change working directory though the menu*. Finally, package shipunov contains function Files() which is the textual *file browser*, so it is possible to run setwd(Files()) and then follow screen instructions[7].

The next step after you got sure that the working directory is correct, is to check if your data file is in place, with dir() command:

```
> dir("data")
[1] "mydata.txt"
```

It is really handy to separate data from all other stuff. Therefore, we assumed above that you have subdirectory data in you working directory, and your data files (including mydata.txt) are in that subdirectory. **Please create it** (and of course, **create the working directory**) if you do not have it yet. You can create these with your file manager, or even with R itself:

```
> dir.create("data")
```

* * *

Now you can load your data with read.table() command. But wait a minute! You need to *understand the structure* of your file first.

Command read.table() is sophisticated but it is not smart enough to determine the data structure on the fly[8]. This is why you need to check data. You can open it in any available simple text editor, in your Web browser, or even from inside R with file.show() or url.show() command. It outputs the data "as is". This is what you will see:

```
> file.show("data/mydata.txt")
a;b;c
1;2;3
```

[7] Yet another possibility is to set working directory in preferences (this is quite different between operating systems) but this is not the best solution because you might (and likely will) want different working directories for different tasks.

[8] There is rio package which can determine the structure of data.

```
4;5;6
7;8;9
```

(By the way, if you type `file.show("data/my` and press `Tab`, *completion* will show you if your file is here—if it is really here. This will save both typing file name and checking the presence with `dir()`.)

How did the file `mydata.txt` appear in your data subdirectory? We assume that you already downloaded it from the repository mentioned in the foreword. If you did not do it, please **do it now**. It is possible to perform with any browser and even with R:

```
> download.file("http://ashipunov.me/data/mydata.txt",
+ "data/mydata.txt")
```

(Within parentheses, left part is for URL whereas right tells R how to place and name the downloaded file.)

Alternatively, you can check your file directly from the URL with `url.show()` and then use `read.table()` from the same URL.

* * *

Now time finally came to load data into R. We know that all columns have names, and therefore use `head=TRUE`, and also know that the delimiter is the semicolon, this is why we use `sep=";"`:

```
> mydata <- read.table("data/mydata.txt", sep=";", head=TRUE)
```

Immediately after we loaded the data, we must check the new object. There are three ways:

```
> str(mydata)
'data.frame': 3 obs. of  3 variables:
 $ a: int  1 4 7
 $ b: int  2 5 8
 $ c: int  3 6 9
> head(mydata)
  a b c
1 1 2 3
2 4 5 6
3 7 8 9
```

Third way is to simply type `mydata` but this is not optimal since when data is large, your computer screen will be messed with content. Commands `head()` and `str()` are much more efficient.

To summarize, local data file should be loaded into R in *three steps*:

1. Make sure that you **data is in place**, with `dir()` command, `Tab` completion or through Web browser;
2. **Take a look** on data with `file.show()` or `url.show()` command and determine its structure;
3. **Load** it with `read.table()` command *using appropriate options* (see below).

   ... and immediately **check** the new object with `str()`

2.7.4 How to load data from Internet

Loading remote data takes same three steps from above. However, as the data is not on disk but somewhere else, to check its presence, the best way is to *open it in the Internet browser* using URL which should be given to you; this also makes the second step because you will see its structure in the browser window. It is also possible to check the structure with the command:

```
> url.show("http://ashipunov.me/data/mydata.txt")
```

Then you can run `read.table()` but with URL instead of the file name:

```
> read.table("http://ashipunov.me/data/mydata.txt", sep=";",
+ head=TRUE)
```

(Here and below we will sometimes skip creation of new object step. However, remember that you *must create new object if you want to use the data* in R later. Otherwise, the content will be shown and immediately forgotten.)

2.7.5 How to use `read.table()` properly

Sometimes, you want R to "ingest" not only column names but also row names:

```
> read.table("data/mydata1.txt", head=TRUE)
      a b c
one   1 2 3
two   4 5 6
three 7 8 9
```

(File `mydata1.txt`[9] is constructed in the unusual way: its first row has three items whereas all other rows each have four items delimited with the *tab symbol*—"big in-

[9] Again, download it from Internet to `data` subdirectory first. Alternatively, replace subdirectory with URL and load it into R directly—of course, after you check the structure.

visible space". Please do not forget to check that beforehand, for example using `file.show()` or `url.show()` command.)

Sometimes, there are both spaces (inside cells) and tabs (between cells):

```
> read.table("data/mydata2.txt", sep="\t", quote="", head=TRUE)
              a b c
one           1 2 3
two           4 5 6
three o'clock 7 8 9
```

If we run `read.table()` without `sep="\t"` option (which is "separator is a tab"), R will give an error. **Try it**. But why did it work for `mydata1.txt`? This is because the *default* separator is *both* space and/or tab. If one of them used as the part of data, the other must be stated as separator explicitly.

Note also that since row names contain quote, quoting must be disabled, otherwise data will silently read in a wrong way.

How to know what separator is here, tab or space? This is usually simple as most editors, browsers and `file.show()` / `url.show()` commands visualize tab as a space which is much broader than single letter. However, do not forget to use `monospaced font` in your software, other fonts might be deceiving.

Sometimes, numbers have comma as a decimal separator (this is another worldwide standard). To input this kind of data, use `dec` option:

```
> read.table("data/mydata3.txt", dec=",", se=";", h=T)
```

(Please note the shortcuts. Shortcuts save typing but could be dangerous if they match several possible names. There are only one `read.table()` argument which starts with `se`, but several of them start with `s` (e.g., `skip`); therefore it is impossible to reduce `se` further, into `s`. Note also that `TRUE` and `FALSE` are possible to shrink into `T` and `F`, respectively (but this is the only possible way); but this is not recomemnded and we will avoid it in the book.)

When `read.table()` sees character columns, it converts them into *factors* (see below). To avoid this behavior, use `as.is=TRUE` option.

Command `scan()` is similar to `read.table()` but reads all data into only one "column" (one vector). It has, however, one important feature:

```
> scan()
1: 1
2: 2
3: 3
```

```
4:
Read 3 items
[1] 1 2 3
```

(What did happen here? First, we entered `scan()` with *empty first argument*, and R changed its prompt to numbers allowing to type numerical data in, element after element. To finish, enter *empty row*[10]. One can paste here even numbers from the clipboard!)

2.7.6 How to load binary data

Functions from the `foreign` package (it is installed by default) can read data in MiniTab, S, SAS, SPSS, Stata, Systat, and FoxPro DBF binary formats. To find out more, you may want to call it first with command `library(foreign)` and then call help about all its commands `help(package=foreign)`.

R can upload images. There are multiple packages for this, one of the most developed is `pixmap`. R can also upload GIS maps of different formats including ArcInfo (packages `maps`, `maptools` and others).

R has its *own binary format*. It is very fast to write and to load[11] (useful for big data) but impossible to use with any program other than R:

```
> xx <- "apple"
> save(xx, file="xx.rd") # Save object "xx"
> exists("xx")
[1] TRUE
> rm(xx)
> exists("xx")
[1] FALSE
> dir()
[1] "xx.rd"
> load("xx.rd") # Load object "xx"
> xx
[1] "apple"
```

(Here we used several new commands. To save and to load binary files, one needs `save()` and `load()` commands, respectively; to remove the object, there is `rm()` command. To show you that the object was deleted, we used `exists()` command.)

Note also that everything which is written after "#" symbol on the same text string is a *comment*. R skips all comments without reading.

[10] On macOS, type Enter twice.

[11] With commands `dput()` and `dget()`, R also saves and loads textual representations of objects.

There are many interfaces which connect R to databases including MySQL, PostgresSQL and sqlite (it is possible to call the last one directly from inside R see the documentation for RSQLite and sqldf packages).

* * *

But what most users actually need is to load the *spreadsheet data* made with MS Excel or similar programs (like Gnumeric or LibreOffice Calc). There are three ways.

First way **we recommend to all** users of this book: **convert Excel file into the text**, and then proceed with read.table() command explained above[12].

On macOS, the best way is to save data from spreadsheet as tab-delimited text file. On Windows and Linux, if you copy any piece of spreadsheet into clipboard and then paste it into text editor (including R script editor), it becomes the tab-delimited text automatically. On macOS, the last is also possible but you will need to use the terminal editor (like nano).

Alternative way is to use external packages which convert binary spreadsheets "on the fly". One is readxl package with main command read_excel(), another is xlsx package with main command read.xlsx(). Please note that these packages are not available by default so you need to *download and install* them (see below for the explanations).

2.7.7 How to load data from clipboard

Third way is to use clipboard. It is easy enough: on Linux or Windows you will need to *select* data in the open spreadsheet, *copy* it to clipboard, and then in R window *type* command like:

```
> read.table("clipboard", sep="\t", head=TRUE)
```

On macOS, this is slightly different:

```
> read.table(pipe("pbpaste"), sep="\t", head=TRUE)
```

(Ignore warnings about "incomplete lines" or "closed connection". There is also the package clipr which unifies the work with clipboard on main OSes.)

"Clipboard way" is especially good when your data come out of non-typical software.

[12]This is a bit similar to the joke about mathematician who, in order to boil the kettle full with water, would empty it first and therefore *reduce the problem to one which was already solved*!

Note also that entering `scan()` and then pasting from clipboard (see above) work the same way on all systems. It is also possible to combine pasting from clipboard and `read.table()` like:

```
> read.table(..., text="<paste_text_table_here>")
```

(You will need to insert text from clipboard as a value of `text` argument.)

* * *

Summarizing the above, recommended data workflow in R might look like:

1. Enter data into the spreadsheet;
2. Save it as a text file with known delimiters (tab is preferable), headers and row names (if needed);
3. Load it into R with `read.table()`;
4. If you must change the data in R, write it afterwards to the external file using `write.table()` command (see below);
5. Open it in the spreadsheet program again and proceed to the next round.

One of its big pluses of this workflow is the *separation between data editing and data processing*.

2.7.8 How to edit data in R

If there is a need to change existing objects, you could *edit* them through R. We do not recommend this though, spreadsheets and text editors are much more advanced then R internal tools.

Nevertheless, there is a simple *spreadsheet* sub-program embedded into R which is set to edit table-like objects (matrices or data frames). To start it on `bb` matrix (see above), enter command `fix(bb)` and edit “in place”. Everything which you enter will immediately change your object. This is somewhat contradictory with R principles so there is the similar function `edit()` which does not change the object but *outputs* the result to the R window.

For other types of objects (not table-like), commands `fix()` / `edit()` call internal (on Windows or macOS) or external (on Linux) text editor. To use external editor, you might need to supply an additional argument, `edit(..., editor="name")` where `name` could be any text editor which is available in the system.

R on Linux has vi editor as a default but it is too advanced for the beginner[13]; we recommend to use nano instead[14]. Also, there is a pico() command which is usually equal to edit(..., editor="nano"). nano editor is usually available also through the macOS terminal.

2.7.9 How to save the results

Beginners in R simply copy results of the work (like outputs from statistical tests) from the R console into some text file. This is enough if you are the beginner. Earlier or later, however, it becomes necessary to save larger objects (like data frames):

```
> write.table(file="trees.txt", trees, row.names=FALSE, sep="\t",
+ quote=FALSE)
```

(File trees.txt, which is made from the internal trees data frame, will be written into the working directory.)

Please be really careful with write.table() as R is perfectly silent if the file with the same name trees.txt is already here. Instead of giving you any warning, it simply overwrites it!

By the way, "internal data" means that it is accessible from inside R directly, without preliminary loading. You may want to check which internal data is available with command data().

While a scan() is a single-vector variant of read.table(), write() command is the single-vector variant of write.table().

* * *

It is now a good time to speak about **file name conventions** in this book. We highly recommend to follow these simply rules:

1. Use only lowercase English letters, numbers and underscore for the file and directory names (and also dot, but only to separate file extension).
2. Do not use uppercase letters, spaces and other symbols!
3. Make your names short, preferably shorter than 15–20 symbols.
4. For R command (script) files, use extension *.r

[13]If, by chance, it started and you have no idea how to quit, press uppercase ZQ.
[14]Within nano, use Ctrl+O to save your edits and Ctrl+X to exit.

By the way, for the comfortable work in R, it is strongly recommended to *change those options of your operating system which allow it to hide file extensions*. On macOS, go to Finder preferences, choose Advanced tab and *select* the appropriate box. On Windows, click View tab in File Explorer, choose Options, then View again, *unselect* appropriate box and apply this to all folders. Linux, as a rule, does not hide file extensions.

* * *

But what if we need to write into the external file *our results* (like the output from statistical test)? There is the `sink()` command:

```
> sink("1.txt", split=TRUE)
> 2+2
[1] 4
> sink()
```

(Here the string "`[1] 4`" will be written to the external file.),

We specified `split=TRUE` argument because we wanted to see the result on the screen. Specify also `append=TRUE` if you want to *add* output to the existing file. To stop sinking, use `sink()` without arguments. Be sure that you always close `sink()`!

There are many tools and external packages which enhance R to behave like full-featured *report system* which is not only calculates something for you but also helps you to write the results. One of the simplest is Rresults shell script (http://ashipunov.me/shipunov/r) which works on macOS and Linux. The appendix of the book explains Sweave system. There are also knitr and much more.

2.7.10 History and scripts

To see what you typed during the current R session, run `history()`[15]:

```
> history(100) # 100 last commands
> history(Inf) # all session commands
> history(p="plot") # last plot commands
```

If you want to *save your history of commands*, use `savehistory()` with the appropriate file name (in quotes) as argument[16].

[15] Does not work on graphical macOS.

[16] Under graphical macOS, this command is not accessible, and you need to use application menu.

While you work with this book, it is a good idea to use `savehistory()` and **save all commands from each R session** in the file named, saying, by the date (like `20170116.r`)[17] and store this file in your working folder.

To do that on macOS, use menu R -> Preferences -> Startup -> History, uncheck Read history file on startup and and enter the name of today's history file. When you close R, file will appear in your working directory.

To save all objects in the binary file, type `save.image()`. You may want to use it if, for example, you are experimenting with R.

* * *

R allows to create *scripts* which might be run later to *reproduce* your work. Actually, R scripts could be written in any text editor[18].

In the appendix, there is much more about R scripts, but the following will help you to create your own first one:

1. Open the text editor, or just type `file.edit("hello.r")`[19]
2. Write there the string

   ```
   print("Hello, world!")
   ```

3. Save the file under `hello.r` name *in your working directory*
4. Call it from R using the command `source("hello.r")`
5. ... and you will see `[1] "Hello, world!"` in R console as if you typed it.

(In fact, you can even type in the script `"Hello world!"` without `print()`, R will understand what to do.)

Then, every time you add any R command to the `hello.r`, you will see more and more output. **Try** it.

To see input (commands) and output (results) together, type `source("hello.r", echo=TRUE)`.

Scripting is the "killer feature" of R. If all your data files are in place, and the R script is made, you may easily return to your calculations years later! Moreover, others can do exactly the same with your data and therefore your research becomes fully

[17] Package `shipunov` has the convenient `Save.history()` function.

[18] You can also use `savehistory()` command to make a "starter" script.

[19] On Windows and macOS, this will open internal editor; on Linux, it is better to set `editor` option manually, e.g., `file.edit("hello.r", editor="geany")`.

reproducible. Even more, if you find that your data must be changed, you run the same script and it will output results which take all changes into account.

Command `source()` allows to load commands not only from local file but also from Internet. You only need to replace file name with URL.

2.8 R graphics

2.8.1 Graphical systems

One of the most valuable part of every statistical software is the ability to make diverse plots. R sets here almost a record. In the base, default installation, several dozens of plot types are already present, more are from recommended `lattice` package, and much more are in the external packages from CRAN where more than a half of them (several thousands!) is able to produce at least one unique type of plot. Therefore, there are several thousands plot types in R. But this is not all. All these plots could be enhanced by user! Here we will try to describe fundamental principles of R graphics.

Let us look on this example (Fig. 2.3):

```
> plot(1:20, main="Title")
> legend("topleft", pch=1, legend="My wonderful points")
```

(Curious reader will find here many things to experiment with. What, for example, is `pch`? Change its number in the second row and **find out**. What if you supply `20:1` instead of `1:20`? Please **discover** and explain.)

Command `plot()` *draws* the basic plot whereas the `legend()` *adds* some details to the already drawn output. These commands represent two basic types of R plotting commands:

1. high-level commands which *create* new plot, and
2. low-level commands which *add features* to the existing plot.

Consider the following example:

```
> plot(1:20, type="n")
> mtext("Title", line=1.5, font=2)
> points(1:20)
> legend("topleft", pch=1, legend="My wonderful points")
```

(These commands make almost *the same plot as above*! Why? Please **find out**. And what is different?)

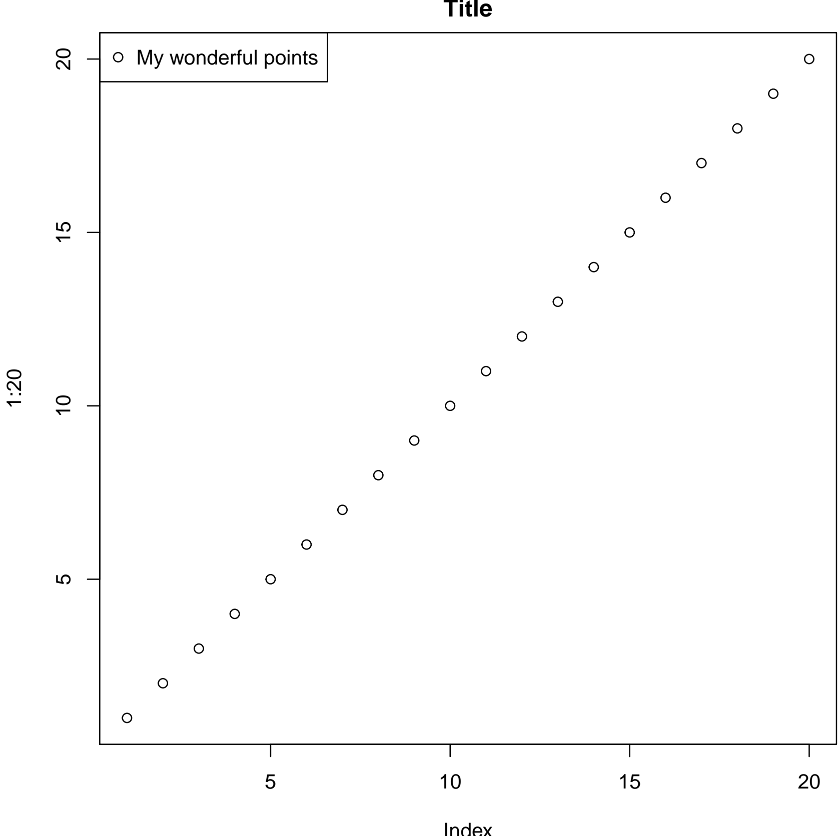


Figure 2.3: Example of the plot with title and legend.

Note also that `type` argument of the `plot()` command has many values, and some produce interesting and potentially useful output. To know more, **try** `p`, `l`, `c`, `s`, `h` and `b` types; check also what `example(plot)` shows.

Naturally, the most important plotting command is the `plot()`. This is a "smart" command[20]. It means that `plot()` "understands" the type of the supplied object, and draws accordingly. For example, `1:20` is a sequence of numbers (numeric vector, see below for more explanation), and `plot()` "knows" that it requires dots with coordinates corresponding to their indices (`x` axis) and actual values (`y` axis). If you supply to the `plot()` something else, the result most likely would be different. Here is an example (Fig. 2.4):

[20]The better term is *generic command*.

```
> plot(cars)
> title(main="Cars from 1920s")
```

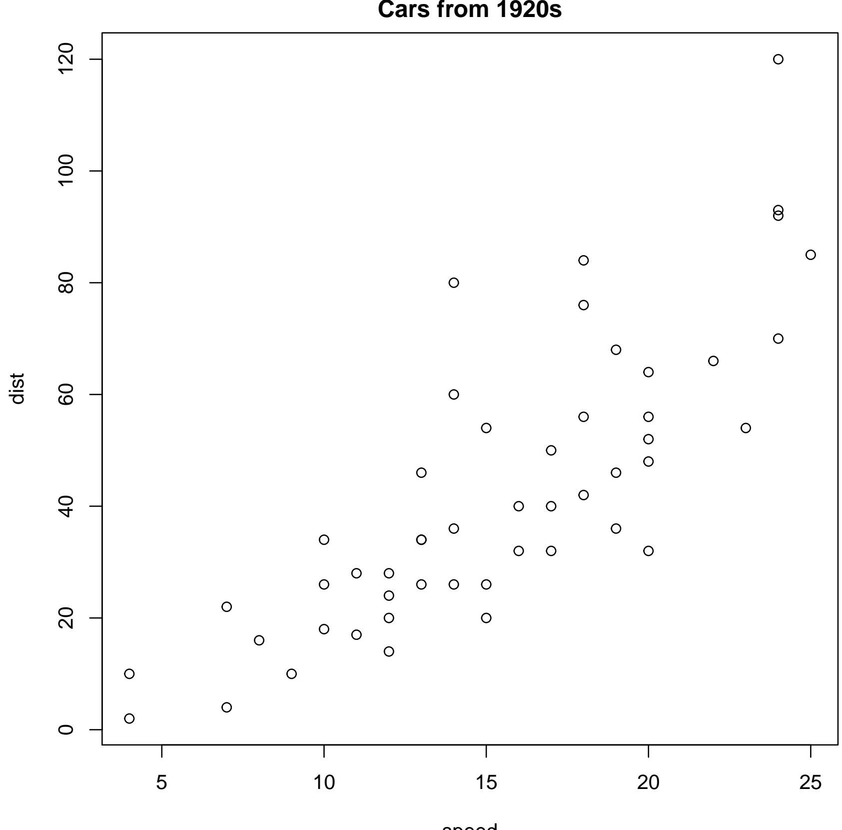


Figure 2.4: Example of plot showing `cars` data.

Here commands of both types are here again, but they were issued in a slightly different way. `cars` is an embedded dataset (you may want to call `?cars` which give you more information). This data is not a vector but *data frame* (sort of table) with two columns, `speed` and `distance` (actually, stopping distance). Function `plot()` chooses the *scatterplot* as a best way to represent this kind of data. On that scatterplot, `x` axis corresponds with the first column, and `y` axis—with the second.

We recommend to **check** what will happen if you supply the data frame with three columns (e.g., embedded `trees` data) or contingency table (like embedded `Titanic` or `HairEyeColor` data) to the `plot()`.

There are innumerable ways to alter the plot. For example, this is a bit more fancy "twenty points":

```
> plot(1:20, pch=3, col=6, main="Title")
```

(Please **run this example** yourself. What are `col` and `pch`? What will happen if you set `pch=0`? If you set `col=0`? Why?)

> Maximal length and maximal width of birds' eggs are likely related. Please make a plot from `eggs.txt` data and confirm (or deny) this hypothesis. Explanations of characters are in companion `eggs_c.txt` file.

* * *

Sometimes, default R plots are considered to be "too laconic". This is simply wrong. Plotting system in R is inherited from S where it was thoroughly developed on the base of systematic research made by W.S. Cleveland and others in Bell Labs. There were many experiments[21]. For example, in order to understand which plot types are easier to catch, they presented different plots and then asked to reproduce data numerically. The research resulted in recommendations of how to make graphic output more understandable and easy to read (please note that it is not always "more attractive"!)

In particular, they ended up with the conclusion that elementary graphical perception tasks should be arranged from easiest to hardest like: position along a scale → length → angle and slope → area → volume → color hue, color saturation and density. So it is easy to *lie with statistics*, if your plot employs perception tasks mostly from the right site of this sequence. (Do you see now why pie charts are particularly bad? This is the reason why they often called "chartjunk".)

They applied this paradigm to S and consequently, in R almost everything (point shapes, colors, axes labels, plotting size) in default plots is based on the idea of intelligible graphics. Moreover, even the order of point and color types represents the sequence from the most easily perceived to less easily perceived features.

> Look on the plot from Fig. 2.5. Guess how was it done, which commands were used?

[21] Cleveland W. S., McGill R. 1985. Graphical perception and graphical methods for analyzing scientific data. Science. 229(4716): 828–833.

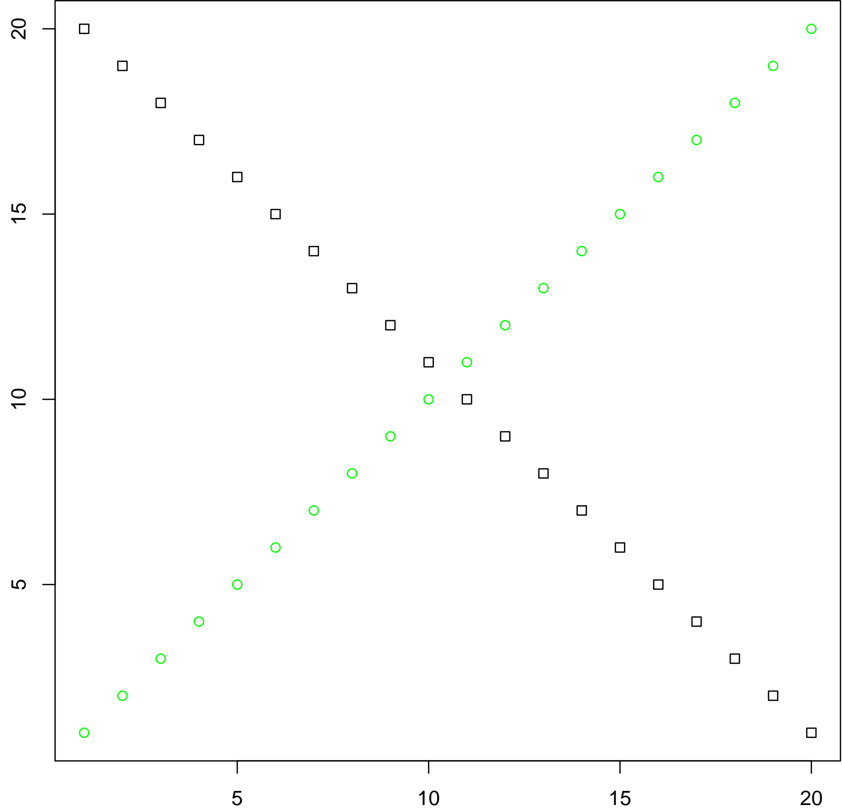


Figure 2.5: Exercise: which commands were used to make this plot?

* * *

Many packages extend the graphical capacities of R. Second well-known R graphical subsystem comes from the `lattice` package (Fig. 2.6):

```
> library(lattice)
> xyplot(1:20 ~ 1:20, main="title")
```

(We repeated `1:20` twice and added tilde because `xyplot()` works slightly differently from the `plot()`. By default, `lattice` should be already installed in your system[22].)

[22] `lattice` came out of later ideas of W.S. Cleveland, *trellis* (conditional) plots (see below for more examples).

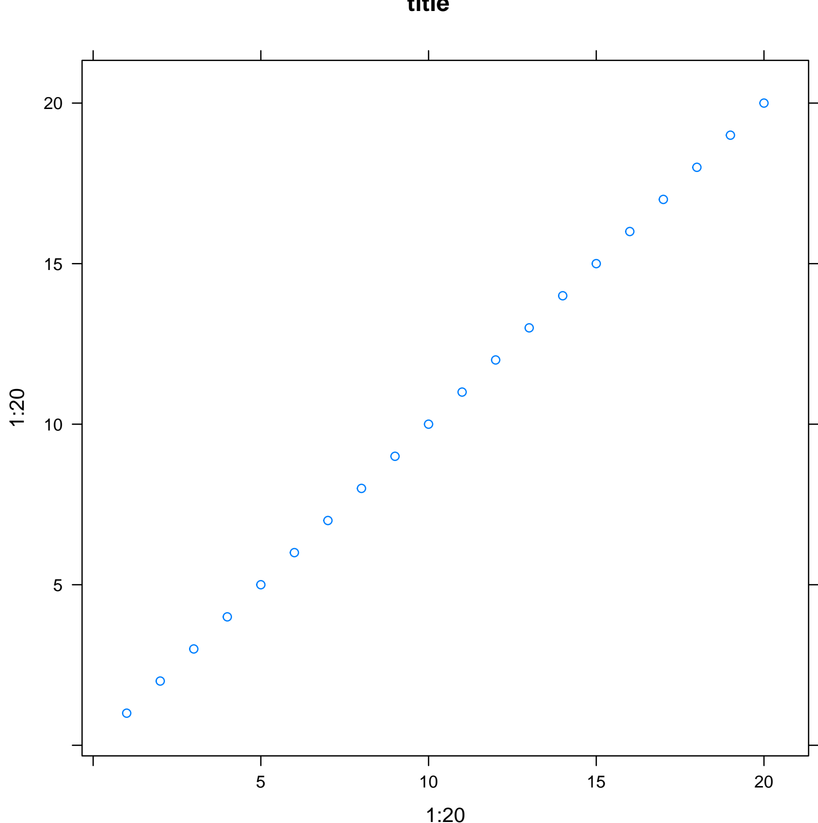


Figure 2.6: Example of plot with a title made with `xyplot()` command from `lattice` package.

Package `lattice` is by default already installed on your system. To know which packages are already installed, type `library()`.

Next, below is what will happen with the same `1:20` data if we apply function `qplot()` from the third popular R graphic subsystem, `ggplot2`[23] package (Fig. 2.7):

```
> library(ggplot2)
> qplot(1:20, 1:20, main="title")
```

[23] `ggplot2` is now the most fashionable R graphic system. Note, however, that it is based on the different "ideology" which related more with SYSTAT visual statistic software and therefore is alien to R.

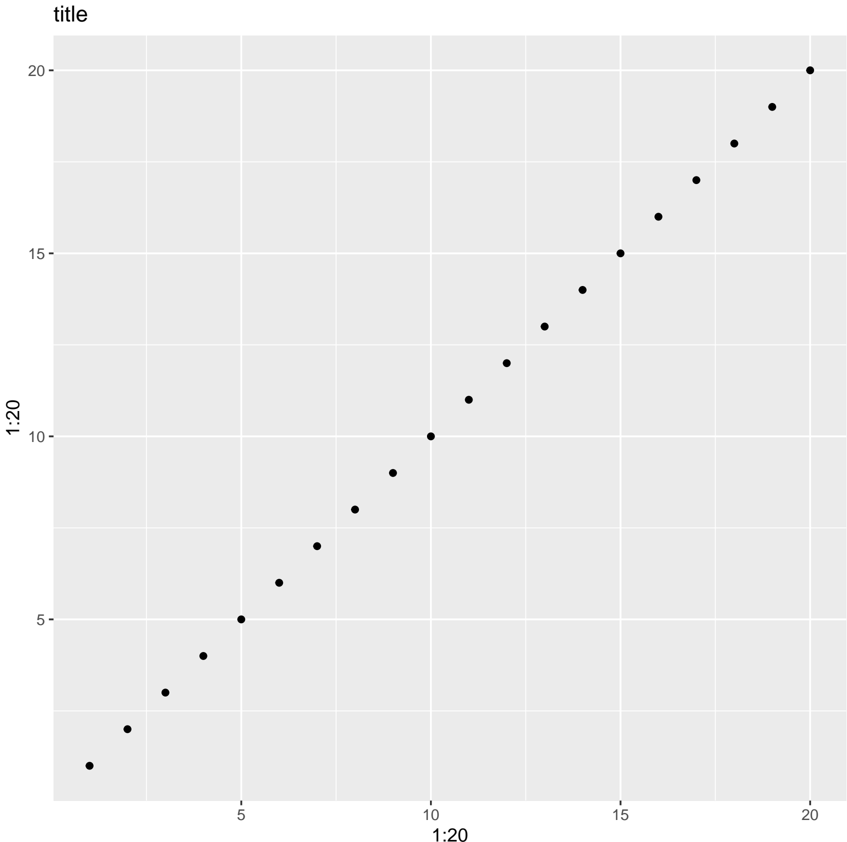


Figure 2.7: Example of plot with a title made with `qplot()` command from `ggplot2` package.

* * *

We already mentioned above that `library()` command loads the package. But what if this package is absent in your installation? `ggplot2` is not installed by default.

In that case, you will need to download it from Internet R archive (CRAN) and install:

```
> install.packages("ggplot2")
```

(Note plural in the command name and quotes in argument).

During the installation, you might be asked first about preferable Internet mirror (it is usually good idea to choose the first). Then, you may be asked about local or system-wide installation (local one works in most cases).

Finally, R for Windows or macOS will simply *unpack* the downloaded archive whereas R on Linux will *compile* the package from source. This takes a bit more time and also could require some additional software to be installed. Actually, some packages want additional software regardless to the system.

After you installed the package, you always need to load it with `library()` command. It is also a good custom to read about the package: `help(package=...)`.

It is also useful to know how to do reverse operation. If you want to remove (unload) the package from R *memory*, use `detach(package:...)`. If you want to remove the package from *disk*, use `remove.packages("...")`. Finally, if you want to use the package command only once, without loading it all into the memory, use `package::command(...)`

2.8.2 Graphical devices

This is the second important concept of R graphics. When you enter `plot()`, R opens screen *graphical device* and starts to draw there. If the next command is of the same type, R will erase the content of the device and start the new plot. If the next command is the "adding" one, like `text()`, R will add something to the existing plot. Finally, if the next command is `dev.off()`, R will close the device.

Most of times, you do not need to call screen devices explicitly. They will open automatically when you type any of main plotting commands (like `plot()`). However, sometimes you need more than one graphical window. In that case, open additional device with `dev.new()` command.

Apart from the screen device, there are many other graphical devices in R, and you will need to remember the most useful. They work as follows:

```
> png(file="01_20.png", bg="transparent")
> plot(1:20)
> text(10, 20, "a")
> dev.off()
```

`png()` command opens the graphical device with the same name, and you may apply some options specific to PNG, e.g., transparency (useful when you want to put the image on the Web page with some background). Then you type all your plotting commands *without seeing the result* because it is now redirected to PNG file connection. When you finally enter `dev.off()`, connection and device will close, and file with a name `01_20.png` will appear in the working directory on disk. Note that R does it silently so if there was the file with the same name, it will be overwritten!

So saving plots in R is as simple as to put elephant into the fridge in three steps (Remember? [1]open fridge – [2]put elephant – [3]close fridge.) Or as simple as to make

a sandwich. This "triple approach" ([1]open device – [2]plot – [3]close device) is the most universal way to save graphics from R. It works on all systems and (what is really important), from the R scripts.

For the beginner, however, difficult is that R is here tacit and does not output anything until the very end. Therefore, it is recommended first to enter plotting commands in a common way, and check what is going on the screen graphic device. Then enter name of file graphic device (like `png()`), and using *arrow up*, repeat commands in proper sequence. Finally, enter `dev.off()`.

`png()` is good for, saying, Web pages but outputs only *raster* images which *do not scale well*. It is frequently recommended to use *vector* images like PDF:

```
> pdf(file="01_20.pdf", width=8)
> plot(1:20)
> text(10, 20, "a")
> dev.off()
```

(Traditionally, PDF width is measured in inches. Since default is 7 inches, the command above makes a bit wider PDF.)

R also can produce files of SVG (scalable vector graphics) format[24].

Important is to **always close the device**! If you did not, there could be strange consequences: for example, new plots do not appear or some files on disk become inaccessible. If you suspect that it is the case, repeat `dev.off()` several times until you receive an error like:

```
> dev.off()
Error in dev.off() : cannot shut down device 1 (the null device)
```

(This is *not* a dangerous error.)

It usually helps.

Another variant is even simpler: just type `graphics.off()` command.

▌ Please create the R script which will make PDF plot by itself.

2.8.3 Graphical options

We already said that R graphics could be tuned in the almost infinite number of ways. One way of the customization is the modification of *graphical options* which are pre-

[24]By the way, both PDF and SVG could be opened and edited with the freely available vector editor Inkscape.

set in R. This is how you, for example, can draw two plots, one under another, in the one window. To do it, change graphical options first (Fig. 2.8):

```
> old.par <- par(mfrow=c(2, 1))
> hist(cars$speed, main="")
> hist(cars$dist, main="")
> par(old.par)
```

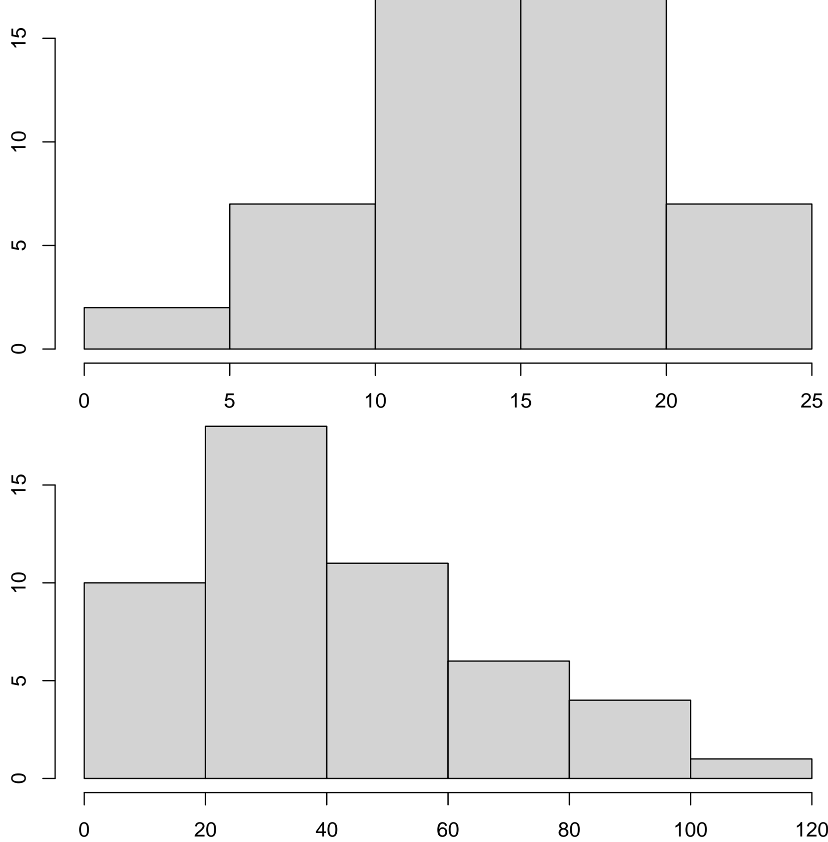


Figure 2.8: Two histograms on the one plot.

(`hist()` command creates *histogram plots*, which break data into bins and then count number of data points in each bin. See more detailed explanations at the end of "one-dimensional" data chapter.)

The key command here is `par()`. First, we changed one of its parameters, namely `mfrow` which regulates number and position of plots within the plotting region. By default `mfrow` is `c(1, 1)` which means "one plot vertically and one horizontally". To

protect the older value of par(), we saved them in the object old.par. At the end, we changed par() again to initial values.

The separate task is to *overlay* plots. That may be done in several ways, and one of them is to change the default par(new=...) value from FALSE to TRUE. Then next high-level plotting command will not erase the content of window but draw over the existed content. Here you should be careful and avoid intersecting axes:

```
> hist(cars$speed, main="", xaxt="n", xlab="")
> old.par <- par(new=TRUE)
> hist(cars$dist, main="", axes=FALSE, xlab="", lty=2)
> par(old.par)
```

(**Try** this plot yourself.)

2.8.4 Interactive graphics

Interactive graphics enhances the data analysis. Interactive tools trace particular points on the plot to their origins in a data, add objects to the arbitrary spots, follow one particular data point across different plots ("brushing"), enhance visualization of multidimensional data, and much more.

The core of R graphical system is not very interactive. Only two interactive commands, identify() and locator() come with the default installation.

With identify(), R displays information about the data point on the plot. In this mode, the click on the default (*left* on Windows and Linux) mouse button near the dot reveals its row number in the dataset. This continues until you right-click the mouse (or Command-Click on macOS).

```
> plot(1:20)
> identify(1:20)
```

Identifying points in 1:20 is practically useless. Consider the following:

```
> plot(USArrests[, c(1, 3)])
> identify(USArrests[, c(1, 3)], labels=row.names(USArrests))
```

By default, plot() does not name states, only print dots. Yes, this is possible to print all state names but this will flood plot window with names. Command identify() will help if you want to see just outliers.

Command `locator()` *returns coordinates* of clicked points. With `locator()` you can add text, points or lines to the plot with the mouse[25]. By default, output goes to the console, but with the little trick you can direct it to the plot:

```
> plot(1:20)
> text(locator(), "My beloved point", pos=4)
```

(Again, left click (Linux & Windows) or click (macOS) will mark places; when you stop this with the right click (Linux & Windows) or Command+Click (macOS), the text will appear in previously marked place(s).)

How to save the plot which was modified interactively? The "triple approach" explained above will not work because it does not allow interaction. When your plot is ready on the screen, use the following:

```
> dev.copy("pdf", "01_20.pdf"); dev.off()
```

This pair of commands (concatenated with command delimiter, semicolon, for the compactness) *copy existing plot* into the specified file.

Plenty of interactive graphics is now available in R through the external packages like `iplot`, `loon`, `manipulate`, `playwith`, `rggobi`, `rpanel`, `TeachingDemos` and many others.

2.9 Answers to exercises

Answer to the question of how to find the R command if you know only what it should do (e.g., "anova"). In order to find this from within R, you may go in several ways. First is to use `help.search()`:

```
> help.search("anova", lib.loc=.Library) # searches only base packages
...
stats::anova            Anova Tables
stats::anova.glm        Analysis of Deviance for Generalized
                        Linear Model Fits
stats::anova.lm         ANOVA for Linear Model Fits
...
stats::stat.anova       GLM Anova Statistics
...
```

(Output might be long because it includes all installed packages. Pay attention to rows started with "base" and "stats".)

[25] Package `shipunov` has game-like command `Miney()`, based on `locator()`; it partly imitates the famous "minesweeper" game.

To search through all available packages, symply type:

```
> ??anova
```

Similar result might be achieved if you start the interactive (Web browser-based) help with `help.start()` and then enter "anova" into the search box.

Second, even simpler way, is to use `apropos()`:

```
> apropos("anova")
[1] "anova"   "manova"   "power.anova.test"   "stat.anova"
[5] "summary.manova"
```

Sometimes, nothing helps:

```
> ??clusterization
No vignettes or demos or help files found with alias or concept or
title matching 'clusterization' using fuzzy matching.
> apropos("clusterization")
character(0)
```

Then start to search in the Internet. It might be done from within R:

```
> RSiteSearch("clusterization")
A search query has been submitted to http://search.r-project.org
The results page should open in your browser shortly
```

In the Web browser, you should see the new tab (or window) with the query results.

If nothing helps, as the R community. Command `help.request()` will guide you through posting sequence.

* * *

Answer to the question about eggs. First, load the data file. To use `read.table()` command, we need to know file structure. To know the structure, (1) we need to look on this file from R with `url.show()` (or without R, in the Internet browser), and also (2) to look on the companion file, `eggs_c.txt`.

From (1) and (2), we conclude that file has three nameless columns from which we will need first and second (egg length and width in mm, respectively). Columns are separated with large space, most likely the `Tab` symbol. Now run `read.table()`:

```
> eggs <- read.table("data/eggs.txt")
```

Next step is **always** to check the structure of new object:

```
> str(eggs)
'data.frame': 555 obs. of  3 variables:
 $ V1: int  51 48 44 48 46 43 48 46 49 45 ...
 $ V2: int  34 33 33 34 32 32 35 32 34 33 ...
 $ V3: chr  "221" "221" "221" "221" ...
```

It is also the good idea to look on first rows of data:

```
> head(eggs)
  V1 V2  V3
1 51 34 221
2 48 33 221
3 44 33 221
4 48 34 221
5 46 32 221
6 43 32 221
```

Our first and second variables received names `V1` (length) and `V2` (width). Now we need to plot variables to see possible relation. The best plot in that case is a *scatterplot*, and to make scatterplot in R, we simply use `plot()` command:

```
> plot(V2 ~ V1, data=eggs, xlab="Length, mm", ylab="Width, mm",
+ pch=21, bg="grey")
```

(Command `plot(y ~ x)` uses R *formula interface*. It is almost the same as `plot(x, y)`[26]; but note the different order in arguments.)

Resulted "cloud" is definitely elongated and slanted as it should be in case of dependence. What would make this more clear, is some kind of the "average" line showing the direction of the relation. As usual, there are several possibilities in R (Fig. 2.9):

```
> abline(line(eggs$V1, eggs$V2), lty=2, lwd=1.5)
> lines(loess.smooth(eggs$V1, eggs$V2), col=2, lty=2, lwd=1.5)
> legend("topleft", lty=2, col=1:2, lwd=1.5, legend=c(
+ "Tukey's median-median line", "LOESS curve"))
```

(Note use of `line()`, `lines()` and `abline()`—all three are really different commands. `lines()` and `abline()` are low-level graphic commands which add line(s) to the existing plot. First uses coordinates while the second uses coefficients. `line()` and `loess.smooth()` do not draw, they *calculate* numbers to use with drawing commands. To see this in more details, run `help()` for every command.)

[26] In the case of our eggs data frame, the command of second style would be `plot(eggs[, 1:2])` or `plot(eggs$V1, eggs$V2)`, see more explanations in the next chapter.

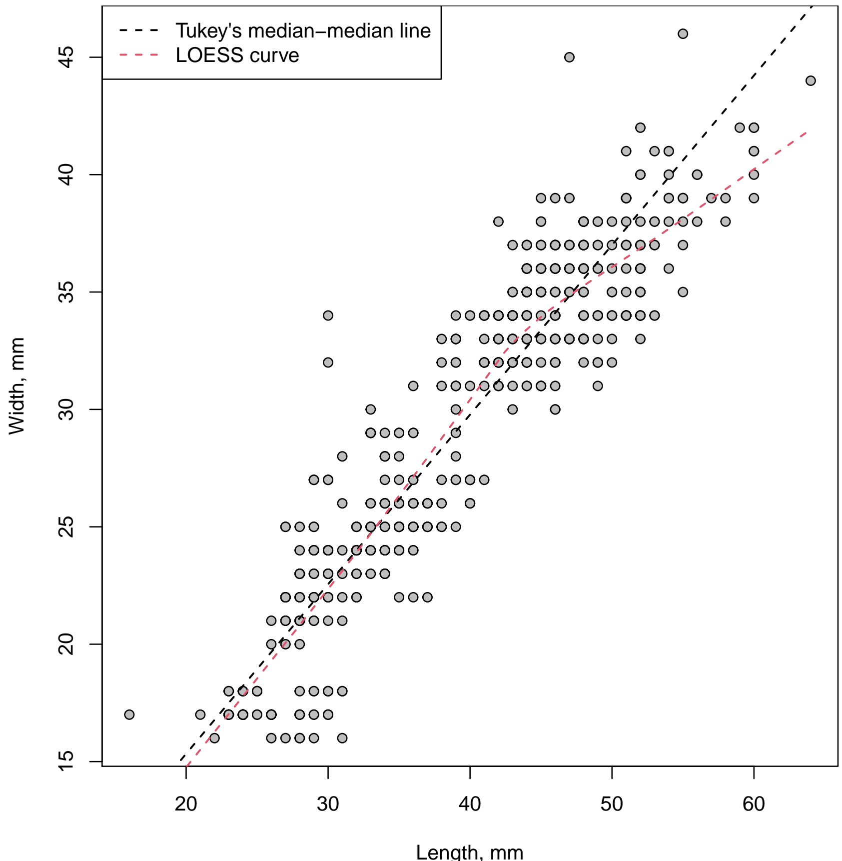


Figure 2.9: Cloud of eggs: scatterplot.

First `line()` approach uses John Tukey's algorithm based on medians (see below) whereas `loess.smooth()` uses more complicated non-linear LOESS (LOcally wEighted Scatterplot Smoothing) which estimates the overall shape of the curve[27]. Both are approximate but robust, exactly what we need to answer the question. Yes, *there is a dependence between egg maximal width and egg maximal length*.

There is one problem though. Look on the Fig. 2.9: many "eggs" are overlaid with other points which have exact same location, and it is not easy to see how many data belong to one point. We will try to access this in next chapter.

[27] Another variant is to use high-level `scatter.smooth()` function which replaces `plot()`. Third alternative is a cubic smoother `smooth.spline()` which calculates numbers to use with `lines()`.

* * *

Answer to the plot question (Fig. 2.5):

```
> plot(1:20, col="green", xlab="", ylab="")
> points(20:1, pch=0)
```

(Here empty `xlab` and `ylab` were used to remove axes labels. Note that `pch=0` is the rectangle.)

Instead of `col="green"`, one can use `col=3`. See below `palette()` command to understand how it works. To know all color names, type `colors()`. Argument `col` could also have *multiple values*. **Check** what happen if you supply, saying, `col=1:3` (pay attention to the very last dots).

To know available point types, run `example(points)` and skip several plots to see the table of points; or simply look on Fig. A.1 in this book (and read comments how it was made).

* * *

Answer to the R script question. It is enough to create (with any text editor) the text file and name it, for example, `my_script1.r`. Inside, type the following:

```
pdf("my_plot1.pdf")
plot(1:20)
dev.off()
```

Create the subdirectory `test` and *copy* your script there. Then *close* R as usual, *open* it again, direct it (through the menu or with `setwd()` command) to make the `test` subdirectory the working directory, and run:

```
> source("my_script1.r", echo=TRUE)
```

If everything is correct, then the file `my_plot1.pdf` will appear in the `test` directory. Please do not forget to **check** it: open it with your PDF viewer. If anything went wrong, it is recommended to delete directory `test` along with all content, modify the master copy of script and repeat the cycle again, until results become satisfactory.

Chapter 3

Types of data

To process data it is not enough just to obtain them. You need to convert it to the appropriate format, typically to numbers. Since Galileo Galilei, who urged to "*measure what can be measured, and make measurable what cannot be measured*", European science aggregated tremendous experience in transferring surrounding events into numbers. Most of our instruments are devices which translate environment features (e.g., temperature, distance) to the numerical language.

3.1 Degrees, hours and kilometers: measurement data

It is extremely important that temperature and distance change smoothly and continuously. This means, that if we have two different measures of the temperature, we can always *imagine an intermediate value*. Any two temperature or distance measurements form an interval including an infinite amount of other possible values. Thus, our first data type is called *measurement*, or *interval*. Measurement data is similar to the ideal endless ruler where every tick mark corresponds to a real number.

However, measurement data do not always change smoothly and continuously from negative infinity to positive infinity. For example, temperature corresponds to a ray and not a line since it is limited with an absolute zero (0°K), and the agreement is that below it no temperature is possible. But the rest of the temperature points along its range are still comparable with real numbers.

It is even more interesting to measure angles. Angles change continuously, but after 359° goes 0°! Instead of a line, there is a segment with only positive values. This is why exists a special *circular statistics* that deals with angles.

Sometimes, collecting measurement data requires expensive or rare equipment and complex protocols. For example, to estimate the colors of flowers as a continuous variable, you would (as minimum) have to use spectrophotometer to measure the wavelength of the reflected light (a numerical representation of visible color).

Now let us consider another example. Say, we are counting the customers in a shop. If on one day there were 947 people, and 832 on another, we can easily imagine values in between. It is also evident that on the first day there were more customers. However, the analogy breaks when we consider two consecutive numbers (like 832 and 831) because, since people are not counted in fractions, there is no intermediate. Therefore, these data correspond better to natural then to real numbers. These numbers are ordered, but not always allow intermediates and are always non-negative. They belong to a different type of measurement data—not continuous, but *discrete*[1].

* * *

Related with definition of measurement data is the idea of *parametricity*. With that approach, inferential statistical methods are divided into *parametric* and *nonparametric*. Parametric methods are working well if:

1. Data type is *continuous measurement*.
2. Sample size is *large* enough (usually no less then 30 individual observations).
3. *Data distribution* is *normal* or close to it. This data is often called "normal", and this feature—"normality".

Should *at least one* of the above assumptions to be violated, the data usually requires *nonparametric methods*. An important advantage of nonparametric tests is their ability to deal with data without prior assumptions about the distribution. On the other hand, *parametric methods are more powerful*: the chance of find an existing pattern is higher because nonparametric algorithms tend to "mask" differences by combining individual observations into groups. In addition, nonparametric methods for two and more samples often suffer from sensitivity to the inequality of sample distributions.

"Normal" data and even "parametric" data are of course jargonisms but these names (together with their opposites) will be used throughout the text for simplicity. Please remember that under "normal" we mean the data which distribution allows to guess that parametric methods are appropriate ways to analyze it.

Let us create normal and non-normal data artificially:

[1] Discrete measurement data are in fact more handy to computers: as you might know, processors are based on 0/1 logic and do not readily understand non-integral, floating numbers.

```
> rnorm(10)
 [1] -0.7370785 0.6138769 -0.8237311 0.3089834 -0.1053840
 [6]  1.0024135 0.6202694 -0.1562604 0.1493745 -1.2335308
> runif(10)
 [1] 0.9468113 0.3505730 0.1066934 0.6470914 0.4819566 0.9502402
 [7] 0.2079193 0.1849797 0.1964943 0.1410300
```

(First command creates 10 random numbers which come from normal distribution. Second creates numbers from uniform distribution[2] Whereas first set of numbers are concentrated around zero, like in darts game, second set are more or less equally spaced.)

But *how to tell normal from non-normal*? Most simple is the visual way, with appropriate plots (Fig. 3.1):

```
> old.par <- par(mfrow=c(2, 1))
> hist(rnorm(100), main="Normal data")
> hist(runif(100), main="Non-normal data")
> par(old.par)
```

(Do you see the difference? Histograms are good to check normality but there are better plots—see next chapter for more advanced methods.)

* * *

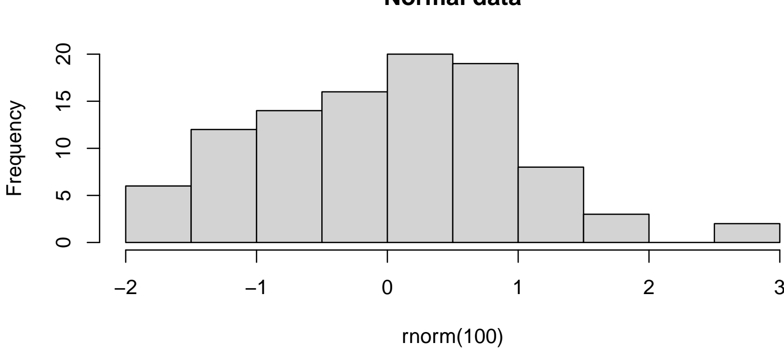
Note again that nonparametric methods are applicable to both "nonparametric" and "parametric" data whereas the opposite is not true (Fig. 3.2).

By the way, this last figure (Euler diagram) was created with R by typing the following commands:

```
> library(plotrix)
> plot(c(-1, 1), c(-1, 1), type="n", xlab="", ylab="", axes=FALSE)
> draw.circle(-.2, 0, .4)
> draw.circle(.1, 0, .9)
> text(-.2, 0, "parametric", cex=1.5)
> text(.1, 0.6, "nonparametric", cex=1.5)
```

(We used `plotrix` package which has the `draw.circle()` command defined. As you see, one may use R even for these exotic purposes. However, diagrams are better to draw in specialized applications like Inkscape.)

[2]For unfamiliar words, please refer to the glossary in the end of book.



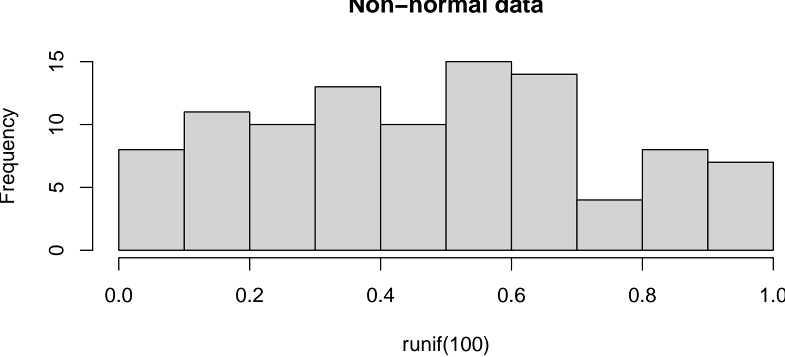


Figure 3.1: Histograms of normal and non-normal data.

* * *

Measurement data are usually presented in R as *numerical vectors*. Often, one vector corresponds with one sample. Imagine that we have data on heights (in cm) of the seven employees in a small firm. Here is how we create a simple vector:

```
> x <- c(174, 162, 188, 192, 165, 168, 172.5)
```

As you learned from the previous chapter, `x` is the name of the R object, `<-` is an *assignment* operator, and `c()` is a function to create vector. Every R object has a *structure*:

```
> str(x)
 num [1:7] 174 162 188 192 165 168 172.5
```

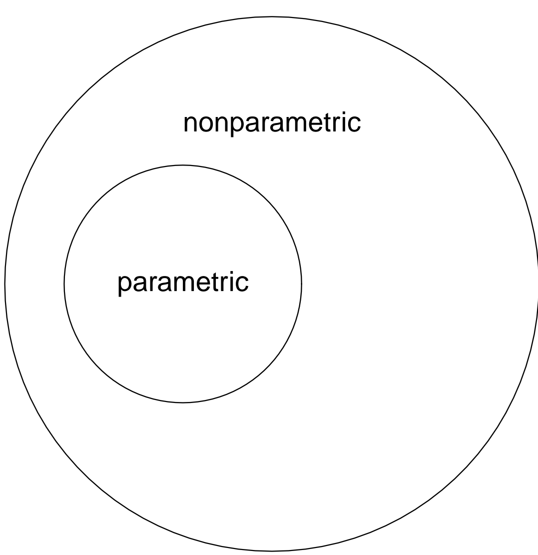


Figure 3.2: Applicability of parametric and nonparametric methods: the Euler diagram.

Function `str()` shows that `x` is a `num`, *numerical vector*. Here is the way to check if an object is a vector:

```
> is.vector(x)
[1] TRUE
```

There are many `is.something()`-like functions in `R`, for example:

```
> is.numeric(x)
[1] TRUE
```

There are also multiple `as.something()`-like *conversion* functions.

To sort heights from smallest to biggest, use:

```
> sort(x)
[1] 162.0 165.0 168.0 172.5 174.0 188.0 192.0
```

To reverse results, use:

```
> rev(sort(x))
[1] 192.0 188.0 174.0 172.5 168.0 165.0 162.0
```

* * *

Measurement data is somehow similar to the common ruler, and R package vegan has a ruler-like linestack() plot useful for plotting linear vectors:

One of simple but useful plots is the linestack() timeline plot from vegan package (Fig. 3.3):

```
> library(vegan)
> phanerozoic <- read.table("data/phanerozoic.txt")
> with(phanerozoic, linestack(541-V2, labels=paste(V1, V2), cex=1))
```

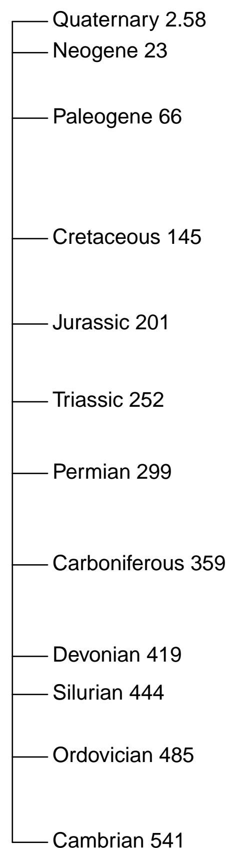


Figure 3.3: Timeline (Mya) of Phanerozoic geological periods.

* * *

In the open repository, file `compositae.txt` contains results of flowering heads measurements for many species of aster family (Compositae). In particular, we measured the overall diameter of heads (variable `HEAD.D`) and counted number of rays ("petals", variable `RAYS`, see Fig. 3.4). Please explore part of this data graphically, with scatterplot(s) and find out if three species (yellow chamomile, *Anthemis tinctoria*; garden cosmos, *Cosmos bipinnatus*; and false chamomile, *Tripleurospermum inodorum*) are different by combination of diameter of heads and number of rays.

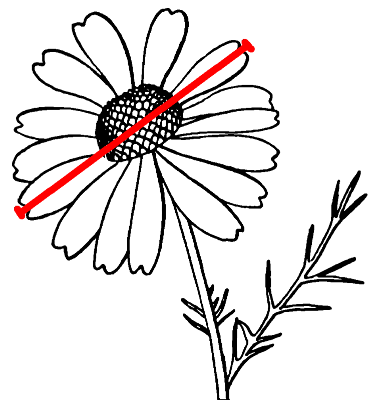


Figure 3.4: Chamomille, *Tripleurospermum.*: leaves and head (diameter shown) with 15 rays.

3.2 Grades and t-shirts: ranked data

Ranked (or *ordinal*) data do not come directly from measurements and do not easily correspond to numbers.

For example, quality of mattresses could be estimated with some numbers, from bad ("0"), to excellent ("5"). These assigned numbers are a matter of convenience. They may be anything. However, they maintain a relationship and continuity. If we grade the most comfortable one as "5", and somewhat less comfortable as "4", it is possible to imagine what is "4.5". This is why many methods designed for measurement variables are applicable to ranked data. Still, we recommend to treat results with caution and keep in mind that these grades are arbitrary.

By default, `R` will identify ranked data as a regular numerical vector. Here are seven employees ranked by their heights:

```
> rr <- c(2, 1, 3, 3, 1, 1, 2)
> str(rr)
 num [1:7] 2 1 3 3 1 1 2
```

Object rr is the same numerical vector, but numbers “1”, “2” and “3” are not measurements, they are ranks, “places”. For example, “3” means that this person belongs to the tallest group.

Function cut() helps to make above three groups automatically:

```
> (hh <- cut(x, 3, labels=c(1:3), ordered_result=TRUE))
[1] 2 1 3 3 1 1 2
Levels: 1 < 2 < 3
```

Result is the *ordered factor* (see below for more explanations). Even if ordered_result is not specified, cut() will still arrange labels in proper order (not necessarily alphabetically as it is typical for factors).

Note that cut() is *irreversible* operation, and “numbers” which you receive are not numbers (heights) you start from:

```
> x
[1] 174.0 162.0 188.0 192.0 165.0 168.0 172.5
> as.numeric(hh)
[1] 2 1 3 3 1 1 2
```

* * *

Ranked data *always* require nonparametric methods. If we still want to use parametric methods, we have to obtain the measurement data (which usually means designing the study differently) and also check it for the normality. However, there is a possibility to re-encode ranked data into the measurement. For example, with the appropriate care the color description could be encoded as red, green and blue channel intensity.

Suppose, we examine the average building height in various cities of the world. Straightforward thing to do would be to put names of places under the variable “city” (nominal data). It is, of cause, the easiest way, but such variable would be almost useless in statistical analysis. Alternatively, we may encode the cities with letters moving from north to south. This way we obtain the ranked data, open for many nonparametric methods. Finally, we may record geographical coordinates of each city. This we obtain the measurement data, which might be suitable for parametric methods of the analysis.

3.3 Colors, names and sexes: nominal data

Nominal, or *categorical*, data, unlike ranked, are impossible to order or align. They are even farther away from numbers. For example, if we assign numerical values to males and females (say, “1” and “2”), it would not imply that one sex is somehow “larger” then the other. An intermediate value (like “1.5”) is also hard to imagine. Consequently, nominal indices may be labeled with any letters, words or special characters—it does not matter.

Regular numerical methods are just *not applicable* to nominal data. There are, however, ways around. The simplest one is *counting*, calculating frequencies for each level of nominal variable. These counts, and other derived measures, are easier to analyze.

3.3.1 Character vectors

R has several ways to store nominal data. First is a character (textual) vector:

```
> sex <- c("male", "female", "male", "male", "female", "male",
+ "male")
> is.character(sex)
[1] TRUE
> is.vector(sex)
[1] TRUE
> str(sex)
 chr [1:7] "male" "female" "male" "male" "female" "male" ...
```

(Please note the function `str()` again. It is **must be used** each time when you deal with new objects!)

By the way, to enter character strings manually, it is easier to start with something like `aa <- c("""")`, then insert commas and spaces: `aa <- c("", "")` and finally insert values: `aa <- c("b", "c")`.

Another option is to enter `scan(what="char")` and then type characters without quotes and commas; at the end, enter empty string.

Let us suppose that vector `sex` records sexes of employees in a small firm. This is how R displays its content:

```
> sex
[1] "male"   "female" "male"   "male"   "female" "male"   "male"
```

To select elements from the vector, use square brackets:

```
> sex[2:3]
```

```
[1] "female" "male"
```

Yes, *square brackets are the command*! They are used to *index* vectors and other R objects. To prove it, **run** `?"["`. Another way to check that is with backticks which allow to use non-trivial calls which are illegal otherwise:

```
> `[`(sex, 2:3)
[1] "female" "male"
```

Smart, object-oriented functions in R may "understand" something about object `sex`:

```
> table(sex)
sex
female   male
     2      5
```

Command `table()` counts items of each type and outputs the *table*, which is one of few numerical ways to work with nominal data (next section tells more about counts).

3.3.2 Factors

But `plot()` could do nothing with the character vector (**check** it yourself). To plot the nominal data, we are to inform R first that this vector has to be treated as *factor*:

```
> sex.f <- factor(sex)
> sex.f
[1] male   female male   male   female male   male
Levels: female male
```

Now `plot()` will "see" what to do. It will invisibly count items and draw a barplot (Fig. 3.5):

```
> plot(sex.f)
```

It happened because character vector was transformed into an object of a type specific to categorical data, a factor with two *levels*:

```
> is.factor(sex.f)
[1] TRUE
> is.character(sex.f)
[1] FALSE
> str(sex.f)
 Factor w/ 2 levels "female","male": 2 1 2 2 1 2 2
> levels(sex.f)
```

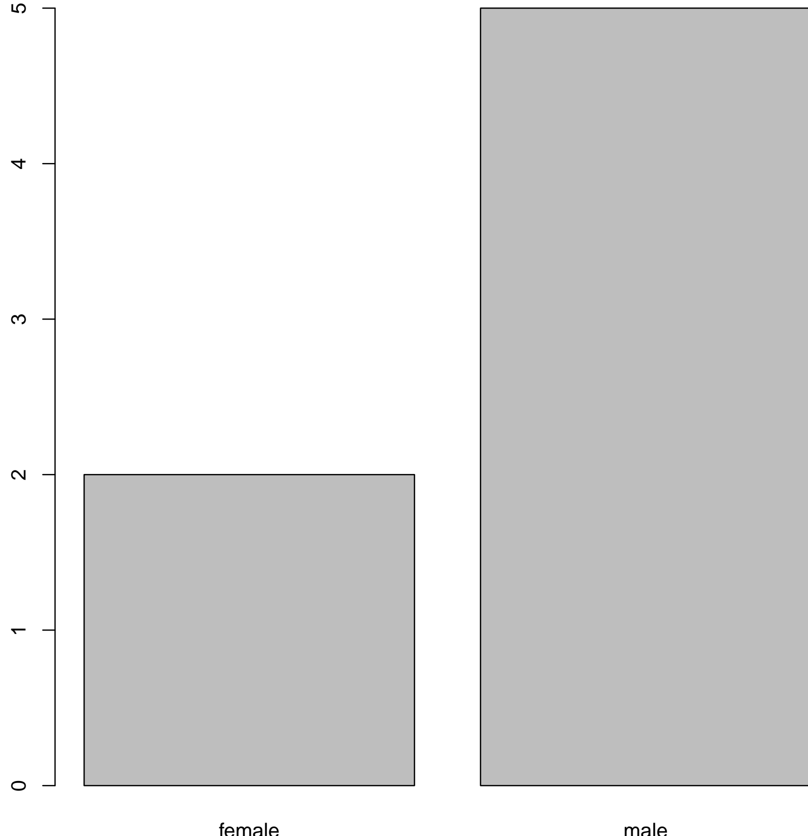


Figure 3.5: This is how `plot()` plots a factor.

```
[1] "female" "male"
> nlevels(sex.f)
[1] 2
```

In R, many functions (including `plot()`) prefer factors to character vectors. Some of them could even transform character into factor, but some not. Therefore, be careful!

There are some other facts to keep in mind.

First (and most important), factors, unlike character vectors, allow for easy transformation into numbers:

```
> as.numeric(sex.f)
[1] 2 1 2 2 1 2 2
```

But why is female 1 and male 2? Answer is really simple: because "female" is the first in alphabetical order. R uses this order every time when factors have to be converted into numbers.

Reasons for such transformation become transparent in a following example. Suppose, we also measured weights of the employees from a previous example:

```
> w <- c(69, 68, 93, 87, 59, 82, 72)
```

We may wish to plot all three variables: height, weight and sex. Here is one possible way (Fig. 3.6):

```
> plot(x, w, pch=as.numeric(sex.f), col=as.numeric(sex.f),
+ xlab="Height, cm", ylab="Weight, kg")
> legend("topleft", pch=1:2, col=1:2, legend=levels(sex.f))
```

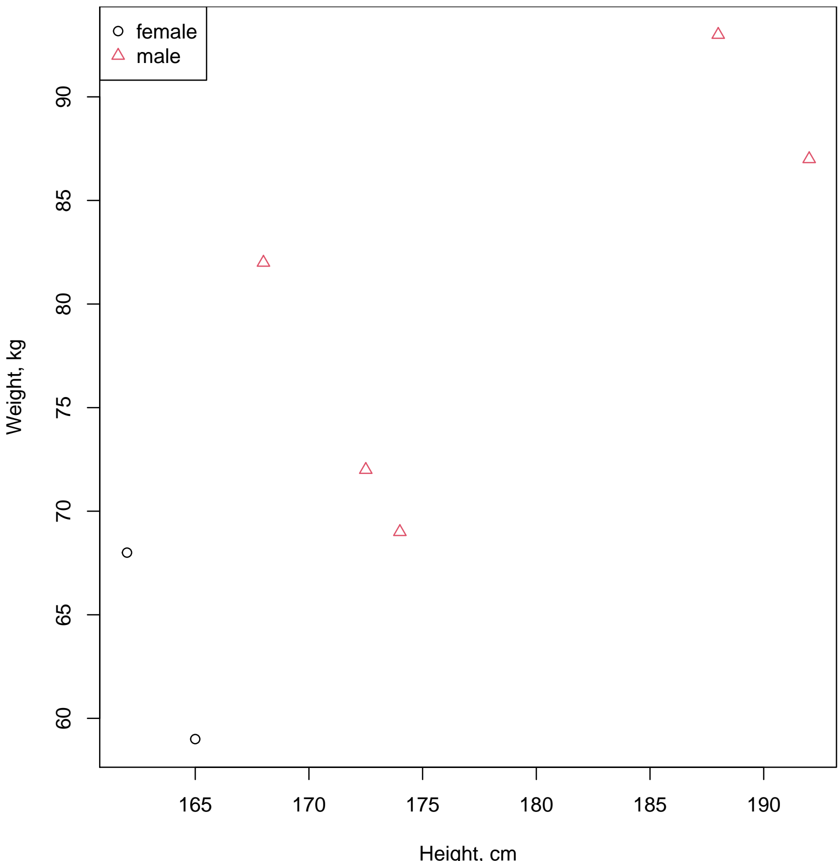


Figure 3.6: A plot with three variables.

Parameters pch (from “print character”) and col (from “color”) define shape and color of the characters displayed in the plot. Depending on the value of the variable sex, data point is displayed as a circle or triangle, and also in black or in red. In general, it is enough to use either shape, or color to distinguish between levels.

Note that colors were printed from numbers in accordance with the current palette. To see which numbers mean which colors, type:

```
> palette()
[1] "black"   "red"     "green3"  "blue"    "cyan"    "magenta"
[7] "yellow" "gray"
```

It is possible to change the default palette using this function with argument. For example, palette(rainbow(8)) will replace default with 8 new “rainbow” colors. To return, type palette("default"). It is also possible to create your own palette, for example with function colorRampPalette() (see examples in next chapters) or using the separate package (like RColorBrewer or cetcolor, the last allows to create *perceptually uniform* palettes).

▌ How to color barplot from Fig. 3.5 in black (female) and red (male)?

If your factor is made from numbers and you want to convert it *back into numbers* (this task is not rare!), convert it first to the characters vector, and only then—to numbers:

```
> (ff <- factor(3:5))
[1] 3 4 5
Levels: 3 4 5
> as.numeric(ff) # incorrect!
[1] 1 2 3
> as.numeric(as.character(ff)) # correct!
[1] 3 4 5
```

Next important feature of factors is that subset of a factor retains by default the original number of levels, even if some of the levels are not here anymore. Compare:

```
> sex.f[5:6]
[1] female male
Levels: female male
> sex.f[6:7]
[1] male male
Levels: female male
```

There are several ways to exclude the unused levels, e.g. with `droplevels()` command, with `drop` argument, or by "back and forth" (factor to character to factor) transformation of the data:

```
> droplevels(sex.f[6:7])
[1] male male
Levels: male
> sex.f[6:7, drop=T]
[1] male male
Levels: male
> factor(as.character(sex.f[6:7]))
[1] male male
Levels: male
```

Third, we may *order* factors. Let us introduce a fourth variable—T-shirt sizes for these seven hypothetical employees:

```
> m <- c("L", "S", "XL", "XXL", "S", "M", "L")
> m.f <- factor(m)
> m.f
[1] L   S   XL  XXL S   M   L
Levels: L M S XL XXL
```

Here levels follow alphabetical order, which is not appropriate because we want `S` (small) to be the first. Therefore, we must tell `R` that these data are ordered:

```
> m.o <- ordered(m.f, levels=c("S", "M", "L", "XL", "XXL"))
> m.o
[1] L   S   XL  XXL S   M   L
Levels: S < M < L < XL < XXL
```

(Now `R` recognizes relationships between sizes, and `m.o` variable could be treated as *ranked*.)

* * *

In this section, we created quite a few new `R` objects. One of skills to develop is to understand which objects are present in your session at the moment. To see them, you might want to *list objects*:

```
> ls()
 [1] "aa"          "bb"          "cards"          "coordinates"   "dice"
...
```

If you want all objects together with their structure, use `ls.str()` command.

There is also a more sophisticated version of object listing, which reports objects in a table:

```
> Ls() # shipunov
          Name      Mode       Type   Obs Vars      Size
1           aa   numeric     vector     5    1  88 bytes
2           bb   numeric     matrix     3    3 248 bytes
3        cards character     vector    36    1      2 Kb
4  coordinates      list data.frame 10000    2   92.2 Kb
5         dice character     vector    36    1      2 Kb
...
```

(To use `Ls()`, install `shipunov` package first, see the preface for explanation.)

`Ls()` is also handy when you start to work with large objects: it helps to clean R memory[3].

3.3.3 Logical vectors and binary data

Binary data (do not mix with a binary file format) are a special case related with both nominal and ranked data. A good example would be "yes" of "no" reply in a questionnaire, or presence *vs.* absence of something. Sometimes, binary data may be ordered (as with presence/absence), sometimes not (as with right or wrong answers). Binary data may be presented either as 0/1 numbers, or as *logical vector* which is the string of `TRUE` or `FALSE` values.

Imagine that we asked seven employees if they like pizza and encoded their "yes"/"no" answers into `TRUE` or `FALSE`:

```
> (likes.pizza <- c(T, T, F, F, T, T, F))
[1]  TRUE  TRUE FALSE FALSE  TRUE  TRUE FALSE
```

Resulted vector is not character or factor, it is *logical*. One of interesting features is that logical vectors participate in arithmetical operations without problems. It is also easy to convert them into numbers directly with `as.numeric()`, as well as to convert numbers into logical with `as.logical()`:

```
> is.vector(likes.pizza)
[1] TRUE
> is.factor(likes.pizza)
[1] FALSE
> is.character(likes.pizza)
```

[3] By default, `Ls()` does not output functions. If required, this behavior could be changed with `Ls(exclude="none")`.

```
[1] FALSE
> is.logical(likes.pizza)
[1] TRUE
> likes.pizza * 1
[1] 1 1 0 0 1 1 0
> as.logical(c(1, 1, 0))
[1]  TRUE  TRUE FALSE
> as.numeric(likes.pizza)
[1] 1 1 0 0 1 1 0
```

This is the most useful feature of binary data. *All other types of data*, from measurement to nominal (the last is most useful), could be converted into logical, and logical is easy to convert into 0/1 numbers:

```
> Tobin(sex, convert.names=FALSE)
     female male
[1,]      0    1
[2,]      1    0
[3,]      0    1
[4,]      0    1
[5,]      1    0
[6,]      0    1
[7,]      0    1
```

Afterwards, many specialized methods, such as logistic regression or binary similarity metrics, will become available even to that initially nominal data.

As an example, this is how to convert the character `sex` vector into logical:

```
> (is.male <- sex == "male")
[1]  TRUE FALSE  TRUE  TRUE FALSE  TRUE  TRUE
> (is.female <- sex == "female")
[1] FALSE  TRUE FALSE FALSE  TRUE FALSE FALSE
```

(We applied *logical expression* on the right side of assignment using "is equal?" double equation symbol operator. This is the second numerical way to work with nominal data. Note that *one* character vector with two types of values became *two* logical vectors.)

Logical vectors are useful also for indexing:

```
> x > 170
[1]  TRUE FALSE  TRUE  TRUE FALSE FALSE  TRUE
> x[x > 170]
[1] 174.0 188.0 192.0 172.5
```

(First, we applied logical expression with greater sign to create the logical vector. Second, we used square brackets to index heights vector; in other words, we *selected* those heights which are greater than 170 cm.)

Apart from greater and equal signs, there are many other *logical operators* which allow to create logical expressions in R (see Table 3.1):

`==`	EQUAL
`<=`	EQUAL OR LESS
`>=`	EQUAL OR MORE
`&`	AND
`|`	OR
`!`	NOT
`!=`	NOT EQUAL
`%in%`	MATCH

Table 3.1: Some logical operators and how to understand them.

AND and OR operators (`&` and `|`) help to build truly advanced and highly useful logical expressions:

```
> ((x < 180) | (w <= 70)) & (sex=="female" | m=="S")
[1] FALSE  TRUE FALSE FALSE  TRUE FALSE FALSE
```

(Here we selected only those people which height is less than 170 cm or weight is 70 kg or less, these people must also be either females or bear small size T-shirts. Note that use of parentheses allows to control the order of calculations and also makes expression more understandable.)

* * *

Logical expressions are even more powerful if you learn how to use them together with command `ifelse()` and operator `if` (the last is frequently supplied with `else`):

```
> ifelse(sex=="female", "pink", "blue")
[1] "blue" "pink" "blue" "blue" "pink" "blue" "blue"
> if(sex[1]=="female") {
+ "pink"
```

```
+ } else {
+ "blue"
+ }
[1] "blue"
```

(Command `ifelse()` is *vectorized* so it goes through multiple conditions at once. Operator `if` takes only one condition.)

Note the use of *curly braces* it the last rows. Curly braces turn a number of expressions into a single (combined) expression. When there is only a single command, the curly braces are optional. Curly braces may contain two commands on one row if they are separated with semicolon.

3.4 Fractions, counts and ranks: secondary data

These data types arise from modification of the "primary", original data, mostly from ranked or nominal data that cannot be analyzed head-on. Close to secondary data is an idea of **compositional data** which are quantitative descriptions of the parts of the whole (probabilities, proportions, percentages etc.)

Percentages, proportions and fractions (ratios) are pretty common and do not need detailed explanation. This is how to calculate percentages (rounded to whole numbers) for our `sex` data:

```
> sex.t <- table(sex)
> round(100*sex.t/sum(sex.t))
sex
female   male
    29     71
```

Since it is so easy to lie with proportions, they must be always supplied with the original data. For example, 50% mortality looks extremely high but if it is discovered that there was only 2 patients, then impression is completely different.

Ratios are particularly handy when measured objects have widely varying absolute values. For example, weight is not very useful in medicine while the height-to-weight ratio allows successful diagnostics.

Counts are just numbers of individual elements inside categories. In R, the easiest way to obtain counts is the `table()` command.

There are many ways to visualize counts and percentages. Bu default, R plots one-dimensional tables (counts) with simple vertical lines (**try** `plot(sex.t)` yourself).

More popular are pie-charts and barplots. However, they represent data badly. There were multiple experiments when people were asked to look on different kinds of plots, and then to report numbers they actually remember. You can **run** this experiment yourself. Figure 3.7 is a barplot of top twelve R commands:

```
> load("data/com12.rd")
> exists("com12") # check if our object is here
[1] TRUE
> com12.plot <- barplot(com12, names.arg="")
> text(com12.plot, par("usr")[3]*2, srt=45, pos=2,
+ xpd=TRUE, labels=names(com12))
```

(We `load()`'ed binary file to avoid using commands which we did not yet learn; to load binary file from Internet, use `load(url(...))`. To make bar labels look better, we applied here the "trick" with rotation. Much more simple but less aesthetic solution is `barplot(com12, las=2)`.)

> Try looking at this barplot for 3–5 minutes, then withdraw from this book and report numbers seen there, from largest to smallest. Compare with the answer from the end of the chapter.

In many experiments like this, researchers found that the most accurately understood graphical feature is the *position along the axis*, whereas length, angle, area, density and color are each less and less appropriate. This is why from the beginning of R history, pie-charts and barplots were recommended to replace with dotcharts (Fig. 3.8):

```
> dotchart(com12)
```

We hope you would agree that the dotchart is easier both to understand and to remember. (Of course, it is possible to make this plot even more understandable with sorting like `dotchart(rev(sort(com12)))`—**try** it yourself. It is also possible to sort bars, but even sorted barplot is worse then dotchart.)

Another useful plot for counts is the *word cloud*, the image where every item is magnified in accordance with its frequency. This idea came out of *text mining* tools. To make word clouds in R, one might use the `wordcloud` package (Fig. 3.9):

```
> com80 <- read.table("data/com80.txt")
> library(wordcloud)
> set.seed(5) # freeze random number generator
> wordcloud(words=com80[, 1], freq=com80[, 2],
+ colors=brewer.pal(8, "Dark2"))
```

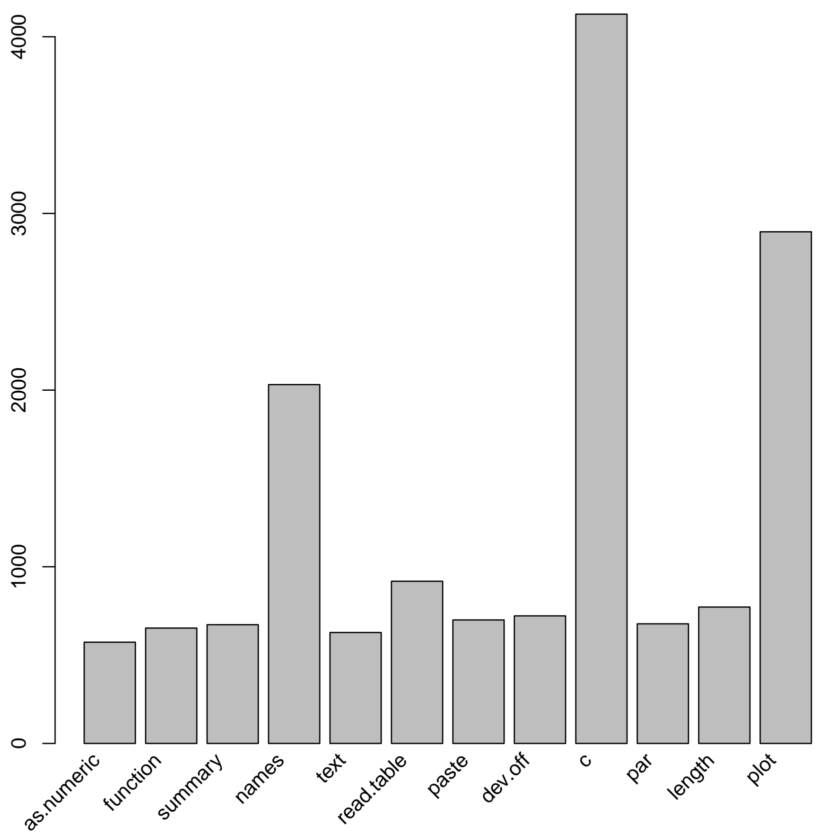


Figure 3.7: Barplot of 12 most frequent R commands.

(New `com80` object is a data frame with two columns—**check** it with `str()` command. Since `wordcloud()` "wants" words and frequencies separately, we supplied columns of `com80` individually to each argument. To select column, we used square brackets with two arguments: e.g., `com80[, 1]` is the first column. See more about this in the "Inside R" section.)

Command `set.seed()` needs more explanation. It freezes random number generator in such a way that immediately after its first use all random numbers are the same on different computers. Word cloud plot uses random numbers, therefore in order to have plots similar between Fig. 3.9 and your computer, it is better run `set.seed()` immediately before plotting. Its argument should be single integer value, same on all computers. To re-initialize random numbers, run `set.seed(NULL)`.

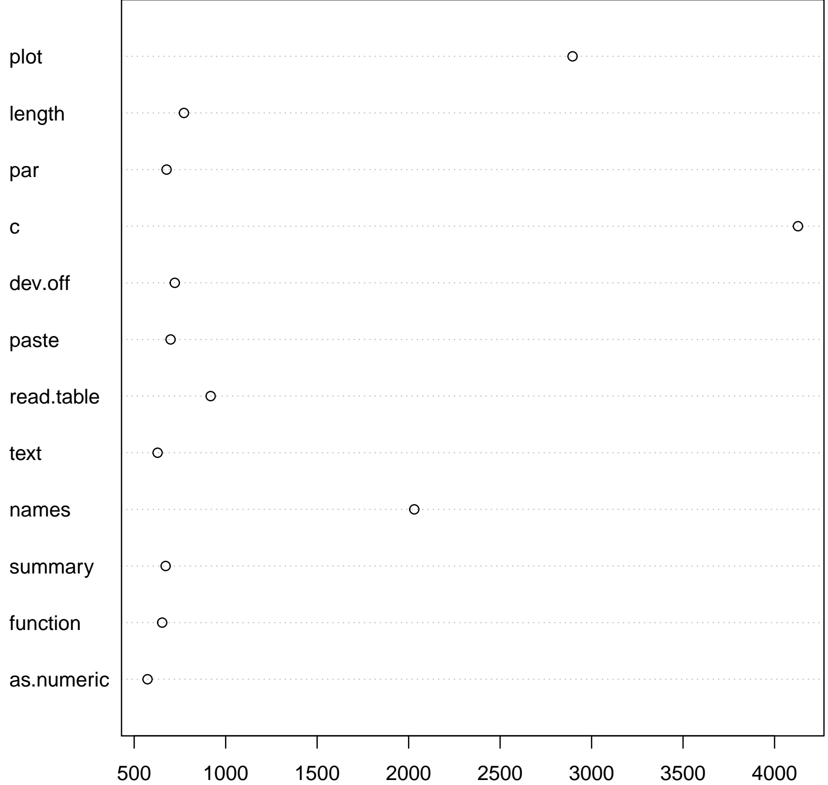


Figure 3.8: Dotchart, or Cleveland dot plot of 12 most frequent R commands.

By the way, `NULL` object is not just an emptiness, it is a really useful tool. For example, it is easy to remove columns from data frame with command like `trees[, 3] <- NULL`. If some command "wants" to plot but you do not need this feature, suppress plotting with `pdf(file=NULL)` command (do not forget to close device with `dev.off()`).

* * *

Compare with your results:

```
> set.seed(1); rnorm(1)
[1] -0.6264538
```

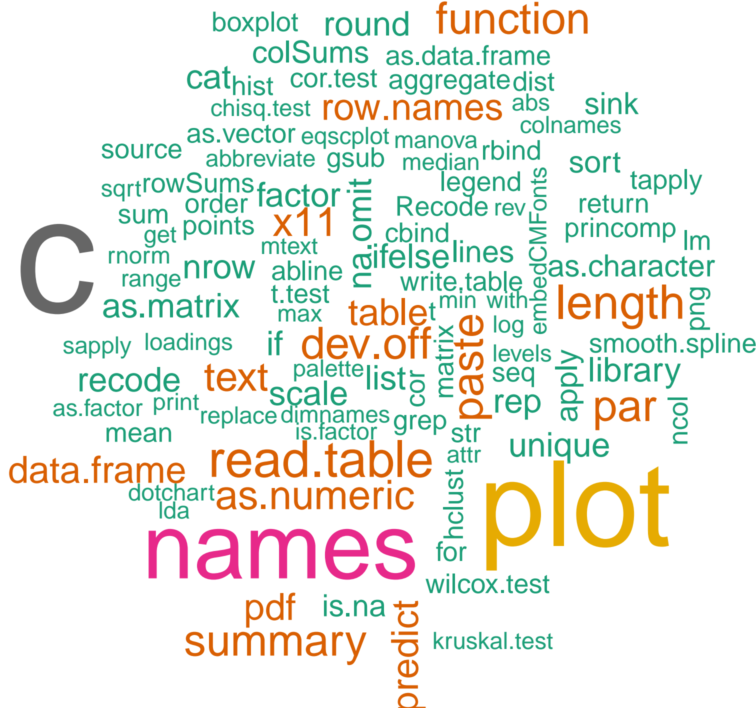


Figure 3.9: Example of word cloud: 80 important R commands.

Word cloud is a fashionable way to show counts but it has one big minus: whereas it possible to tell which word in more frequent, it is impossible to tell how frequent it is. Dotchart of `com80` needs more space (it is better to plot is as big PDF) but there will be both relative and absolute frequencies visible. **Try** it yourself:

```
> dotchart(log(com80[, 2]), labels=com80[, 1], cex=.6)
```

(We used *logarithmic scale* to make counts less dispersed and `cex` parameter to decrease font size.)

* * *

While counts and percentages usually come from categorical (nominal) data, ranks usually come from the measurement data, like our heights:

```
> x.ranks <- x
> names(x.ranks) <- rank(x)
> x.ranks
```

```
    5     1     6     7     2     3     4
174.0 162.0 188.0 192.0 165.0 168.0 172.5
> sort(x.ranks) # easier to spot
    1     2     3     4     5     6     7
162.0 165.0 168.0 172.5 174.0 188.0 192.0
```

(The "trick" here was to use *names* to represent ranks. All R objects, along with values, might bear names.)

Not only integers, but fractions too may serve as rank; the latter happens when there is an even number of equal measurements (i.e., some items are duplicated):

```
> x.ranks2 <- c(x, x[3]) # duplicate the 3rd item
> names(x.ranks2) <- rank(x.ranks2)
> sort(x.ranks2)
    1     2     3     4     5   6.5   6.5     8
162.0 165.0 168.0 172.5 174.0 188.0 188.0 192.0
```

In general, identical original measurements receive identical ranks. This situation is called a "tie", just as in sport. Ties may interfere with some nonparametric tests and other calculations based on ranks:

```
> wilcox.test(x.ranks2)
...
Warning message:
In wilcox.test.default(x.ranks2) : cannot compute exact p-value
with ties
```

(If you did not see R *warnings* before, remember that they might appear even if there is nothing wrong. Therefore, ignore them if you do not understand them. However, sometimes warnings bring useful information.)

R always returns a warning if there are ties. It is possible to avoid ties adding small random noise with `jitter()` command (examples will follow.)

Ranks are widely used in statistics. For example, the popular measure of central tendency, median (see later) is calculated using ranks. They are especially suited for ranked and nonparametric measurement data. Analyses based on ranks are usually more robust but less sensitive.

3.5 Missing data

There is no such thing as a perfect observation, much less a perfect experiment. The larger is the data, the higher is the chance of irregularities. *Missing data* arises from

the almost every source due to imperfect methods, accidents during data recording, faults of computer programs, and many other reasons.

Strictly speaking, there are several types of missing data. The easiest to understand is "unknown", datum that was either not recorded, or even lost. Another type, "both" is a case when condition fits to more then one level. Imagine that we observed the weather and registered sunny days as ones and overcast days with zeros. Intermittent clouds would, in this scheme, fit into both categories. As you see, the presence of "both" data usually indicate poorly constructed methods. Finally, "not applicable", an impossible or forbidden value, arises when we meet something logically inconsistent with a study framework. Imagine that we study birdhouses and measure beak lengths in birds found there, but suddenly found a squirrel within one of the boxes. No beak, therefore no beak length is possible. Beak length is "not applicable" for the squirrel.

In `R`, all kinds of missing data are denoted with two uppercase letters `NA`.

Imagine, for example, that we asked the seven employees about their typical sleeping hours. Five named the average number of hours they sleep, one person refused to answer, another replied "I do not know" and yet another was not at work at the time. As a result, three `NA`'s appeared in the data:

```
> (hh <- c(8, 10, NA, NA, 8, NA, 8))
[1]  8 10 NA NA  8 NA  8
```

We entered `NA` without quotation marks and `R` correctly recognizes it among the numbers. Note that multiple kinds of missing data we had were all labeled identically.

An attempt to just calculate an average (with a function `mean()`), will lead to this:

```
> mean(hh)
[1] NA
```

Philosophically, this is a *correct result* because it is unclear without further instructions how to calculate average of eight values if three of them are not in place. If we still need the numerical value, we can provide one of the following:

```
> mean(hh, na.rm=TRUE)
[1] 8.5
> mean(na.omit(hh))
[1] 8.5
```

The first one allows the function `mean()` to accept (and skip) missing values, while the second creates a temporary vector by throwing `NA`s away from the original vector

hh. The third way is to substitute (*impute*) the missing data, e.g. with the sample mean:

```
> hh.old <- hh
> hh.old
[1]  8 10 NA NA  8 NA  8
> hh[is.na(hh)] <- mean(hh, na.rm=TRUE)
> hh
[1]  8.0 10.0  8.5  8.5  8.0  8.5  8.0
```

Here we selected from hh values that satisfy condition is.na() and *permanently* replaced them with a sample mean[4]. To keep the original data, we saved it in a vector with the other name (hh.old). There are many other ways to *impute missing data*, more complicated are based on bootstrap, regression and/or discriminant analysis. Some are implemented in packages mice and cat.

Package shipunov has Missing.map() function which is useful to determine the "missingness" (volume and relative location of missing data) in big datasets.

3.6 Outliers, and how to find them

Problems arising while typing in data are not limited to empty cells. Mistypes and other kinds of errors are also common, and among them most notorious are *outliers*, highly deviated data values. Some outliers could not be even mistypes, they come from the highly heterogeneous data. Regardless of the origin, they significantly hinder the data analysis as many statistical methods are simply not applicable to the sets with outliers.

The easiest way to catch outliers is to look at maximum and minimum for numerical variables, and at the frequency table for character variables. This could be done with handy summary() function. Among plotting methods, boxplot() (and related boxplot.stats()) is probably the best method to visualize outliers.

While if it is easy enough to spot a value which differs from the normal range of measurements by an order of magnitude, say "17" instead of "170" cm of height, a typing mistake of "171" instead of "170" is nearly impossible to find. Here we rely on the statistical nature of the data—the more measurements we have, the less any individual mistake will matter.

There are multiple *robust statistical procedures* which are not so influenced from outliers. Many of them are also nonparametric, i.e. not sensitive to assumptions about the distribution of data. We will discuss some robust methods later.

[4] We used here *left-hand-side indexing with replacement* via special command, [<-.

* * *

Related with outliers is the common *mistake* in loading data—ignoring headers when they actually exist:

```
> m1 <- read.table("data/mydata.txt", sep=";") # wrong!
> str(m1)
'data.frame': 4 obs. of  3 variables:
 $ V1: chr  "a" "1" "4" "7"
 $ V2: chr  "b" "2" "5" "8"
 $ V3: chr  "c" "3" "6" "9"
> m2 <- read.table("data/mydata.txt", sep=";", h=TRUE) # correct!
> str(m2)
'data.frame': 3 obs. of  3 variables:
 $ a: int  1 4 7
 $ b: int  2 5 8
 $ c: int  3 6 9
```

Command `read.table()` converts whole columns to character vectors even if one data value is not a proper number. This behavior is useful to *identify mistypes*, like "O" (letter O) instead of "0" (zero), but will lead to problems if headers are not defined explicitly. To diagnose problem, use `str()`, it helps to distinguish between the wrong and correct way. Do not forget to use `str()` all the time while you work in R!

3.7 Changing data: basics of transformations

In complicated studies involving many data types: measurements and ranks, percentages and counts, parametric, nonparametric and nominal, it is useful to unify them. Sometimes such transformations are easy. Even nominal data may be understood as continuous, given enough information. For example, sex may be recorded as continuous variable of blood testosterone level, possibly with additional measurements. Another, more common way, is to treat discrete data as continuous—it is usually safe, but sometimes may lead to unpleasant surprises.

Another possibility is to transform measurement data into ranked. R function `cut()` allows to perform this operation and create ordered factors.

What is completely unacceptable is transforming common nominal data into ranks. If values are not, by their nature, ordered, imposing an artificial order can make the results meaningless.

* * *

Data are often transformed to make them closer to parametric and to homogenize standard deviations. Distributions with long tails, or only somewhat bell-shaped (as in Fig. 4.6), might be *log-transformed*. It is perhaps the most common transformation.

There is even a special argument `plot(..., log="axis")`, where "axis" should be substituted with `x` or `y`, presenting it in (natural) logarithmic scale. Another variant is to simply calculate logarithm on the fly like `plot(log(...)`.

Consider some widely used transformations and their implications in R (we assume that your measurements are recorded in the vector `data`):

- Logarithmic: `log(data + 1)`. It may normalize distributions with positive skew (right-tailed), bring relationships between variables closer to linear and equalize variances. It cannot handle zeros, this is why we added a single digit.
- Square root: `sqrt(data)`. It is similar to logarithmic in its effects, but cannot handle negatives.
- Inverse: `1/(data + 1)`. This one stabilizes variances, cannot handle zeros.
- Square: `data^2`. Together with square root, belongs to family of *power transformations*. It may normalize data with negative skew (left-tailed) data, bring relationships between variables closer to linear and equalize variances.
- Logit: `log(p/(1-p))`. It is mostly used on proportions to linearize S-shaped, or sigmoid, curves. Along with logit, these types of data are sometimes treated with arcsine transformation which is `asin(sqrt(p))`. In both cases, `p` must be between 0 and 1.

* * *

While working with multiple variables, keep track of their dimensions. Try not to mix them up, recording one variable in millimeters, and another in centimeters. Nevertheless, in multivariate statistics even data measured in common units might have different nature. In this case, variables are often *standardized*, e.g. brought to the same mean and/or the same variance with `scale()` function. Embedded `trees` data is a good example:

```
> scale(trees)
            Girth     Height      Volume
 [1,] -1.57685421 -0.9416472 -1.20885469
 [2,] -1.48125614 -1.7263533 -1.20885469
```

```
 [3,] -1.41752409 -2.0402357 -1.21493821
 [4,] -0.87580169 -0.6277648 -0.83775985
 [5,] -0.81206964  0.7847060 -0.69175532
...
```

3.7.1 How to tell the kind of data

At the end of data types explanation, we recommend to review a small chart which could be helpful for the determination of data type (Fig. 3.10).

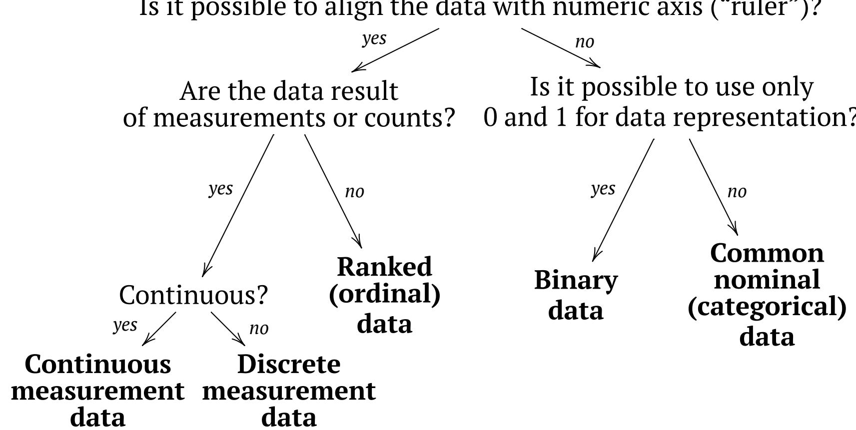


Figure 3.10: How to tell the kind of data.

Many (but not all) data types are inter-convertible. For example, `cut()` allows for conversion from measurement to nominal. Please **think** how to convert other types of data (if it is at all possible).

3.8 Inside R

Vectors in numeric, logical or character modes and factors are enough to represent simple data. However, if the data is structured and/or variable, there is frequently a need for more complicated R objects: matrices, lists and data frames.

3.8.1 Matrices

Matrix is a popular way of presenting tabular data. There are two important things to know about them in R. First, they may have various dimensions. And second—there are, in fact, no true matrices in R.

We begin with the second statement. *Matrix in R is just a specialized type of vector* with additional *attributes* that help to identify values as belonging to rows and columns. Here we create the simple 2×2 matrix from the numerical vector:

```
> m <- 1:4
> m
[1] 1 2 3 4
> ma <- matrix(m, ncol=2, byrow=TRUE)
> ma
     [, 1] [, 2]
[1, ]    1    2
[2, ]    3    4
> str(ma)
 int [1:2, 1:2] 1 3 2 4
> str(m)
 int [1:4] 1 2 3 4
```

As `str()` command reveals, objects `m` and `ma` are very similar. What is different is the way they are *presented* on the screen.

Equality of matrices and vectors in even more clear in the next example:

```
> mb <- m
> mb
[1] 1 2 3 4
> attr(mb, "dim") <- c(2, 2)
> mb
     [, 1] [, 2]
[1, ]    1    3
[2, ]    2    4
```

In looks like a trick but underlying reason is simple. We assign *attribute* `dim` ("dimensions", size) to the vector `mb` and state the value of the attribute as `c(2, 2)`, as 2 rows and 2 columns.

▌ Why are matrices `mb` and `ma` different?

Another popular way to create matrices is *binding* vectors as columns or rows with `cbind()` and `rbind()`. Related command `t()` is used to *transpose* the matrix, *turn it clockwise* by 90°.

To *index* a matrix, use square brackets:

```
> ma[1, 2]
```

```
[1] 2
```

The rule here is simple: *within brackets, first goes first dimension (rows)*, and second to columns. So to index, use `matrix[rows, columns]`. The same rule is applicable to data frames (see below).

Empty index is equivalent to all values:

```
> ma[, ]
     [, 1] [, 2]
[1, ]    1    2
[2, ]    3    4
```

Common ways of indexing matrix do not allow to select diagonal, let alone *L-shaped* ("knight's move") or *sparse selection*. However, R will satisfy even these exotic needs. Let us select the diagonal values of `ma`:

```
> (mi <- matrix(c(1, 1, 2, 2), ncol=2, byrow=TRUE))
     [, 1] [, 2]
[1, ]    1    1
[2, ]    2    2
> ma[mi]
[1] 1 4
```

(Here `mi` is an *indexing matrix*. To index 2-dimensional object, it must have two columns. Each row of indexing matrix describes position of the element to select. For example, the second row of `mi` is equivalent of `[2, 2]`. As an alternative, there is `diag()` command but it works only for diagonals.)

Much less exotic is the indexing with *logical matrix*. We already did similar indexing in the example of missing data imputation. This is how it works in matrices:

```
> (mn <- matrix(c(NA, 1, 2, 2), ncol=2, byrow=TRUE))
     [,1] [,2]
[1,]   NA    1
[2,]    2    2
> is.na(mn)
      [,1]  [,2]
[1,]  TRUE FALSE
[2,] FALSE FALSE
> mn[is.na(mn)] <- 0
> mn
     [,1] [,2]
[1,]    0    1
[2,]    2    2
```

* * *

Since matrices are vectors, *all elements of matrix must be of the same mode*: either numerical, or character, or logical. If we change mode of one element, the rest of them will *change automatically*:

```
> mean(ma)
[1] 2.5
> ma[1, 1] <- "a"
> mean(ma)
[1] NA
Warning message:
In mean.default(ma) :
argument is not numeric or logical: returning NA
> ma
     [, 1] [, 2]
[1, ] "a"  "2"
[2, ] "3"  "4"
```

* * *

Two-dimensional matrices are most popular, but there are also multidimensional *arrays*:

```
> m3 <- 1:8
> dim(m3) <- c(2, 2, 2)
> m3
, , 1
     [, 1] [, 2]
[1, ]    1    3
[2, ]    2    4

, , 2
     [, 1] [, 2]
[1, ]    5    7
[2, ]    6    8
```

(Instead of `attr(..., "dim")` we used analogous `dim(...)` command.)

`m3` is an array, "3D matrix". It cannot be displayed as a single table, and R returns it as a series of tables. There are arrays of higher dimensionality; for example, the built-in dataset `Titatic` is the 4D array. To index arrays, R requires same square brackets but with three or more elements within.

3.8.2 Lists

List is essentially the collection of anything:

```
> l <- list("R", 1:3, TRUE, NA, list("r", 4))
> l
[[1]]
[1] "R"

[[2]]
[1] 1 2 3

[[3]]
[1] TRUE

[[4]]
[1] NA

[[5]]
[[5]][[1]]
[1] "r"

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

Here we see that *list is a composite thing*. Vectors and matrices may only include elements of the same type while lists accommodate anything, including other lists.

List elements could have names:

```
> fred <- list(name="Fred", wife.name="Mary", no.children=3,
+ child.ages=c(1, 5, 9))
> fred
$name
[1] "Fred"
$wife
[1] "Mary"
$no.children
[1] 3
$child.ages
[1] 5 9
```

* * *

Names feature is not unique to lists as many other types of R objects could also have *named elements*. Values inside vectors, and rows and columns of matrices can have their own unique names:

```
> names(w) <- c("Rick", "Amanda", "Peter", "Alex", "Kathryn",
+ "Ben", "George")
> w
Rick Amanda Peter Alex Kathryn Ben George
  69     68    93   87      59  82     72
> row.names(ma) <- c("row1", "row2")
> colnames(ma) <- c("col1", "col2")
> ma
     col1 col2
row1    1    2
row2    3    4
```

To remove names, use:

```
> names(w) <- NULL
> w
[1] 69 68 93 87 59 82 72
```

* * *

Let us now to *index* a list. As you remember, we extracted elements from vectors with square brackets:

```
> hh[3]
[1] 8.5
```

For matrices/arrays, we used several arguments, in case of two-dimensional ones they are row and column numbers:

```
> ma[2, 1]
[1] 3
```

Now, there are at least three ways to get elements from lists. First, we may use the same square brackets:

```
> l[1]
[[1]]
[1] "R"
> str(l[1])
```

```
List of 1
 $ : chr "R"
```

Here the resulting object is *also a list*. Second, we may use *double square brackets*:

```
> l[[1]]
[1] "R"
> str(l[[1]])
 chr "R"
> str(l[[5]])
List of 2
 $ : chr "r"
 $ : num 4
```

After this operation we obtain the *content* of the sub-list, object of the type it had prior to joining into the list. The first object in this example is a character vector, while the fifth is itself a list.

Metaphorically, square brackets take egg out of the basket whereas double square brackets will also shell it.

Third, we may create names for the elements of the list and then call these names with *dollar sign*:

```
> names(l) <- c("first", "second", "third", "fourth", "fifth")
> l$first
[1] "R"
> str(l$first)
 chr "R"
```

Dollar sign is a *syntactic sugar* that allows to write `l$first` instead of more complicated `l[["first"]]`. That last `R` piece might be regarded as a fourth way to index list, with character vector of names.

Now consider the following example:

```
> l$fir
[1] "R"
> l$fi
NULL
```

This happens because dollar sign (and default `[[` too) allow for *partial matching* in the way similar to function arguments. This saves typing time but could potentially be dangerous.

With a dollar sign or character vector, the object we obtain by indexing retains its original type, just as with double square bracket. Note that indexing with dollar sign works only in lists. If you have to index other objects with named elements, use square brackets with character vectors:

```
> names(w) <- c("Rick", "Amanda", "Peter", "Alex", "Kathryn",
+ "Ben", "George")
> w["Jenny"]
Jenny
  68
```

* * *

Lists are so important to learn because many functions in R *store their output as lists*:

```
> x2.wilcox <- wilcox.test(x.ranks2)
...
> str(x2.wilcox)
List of 7
 $ statistic  : Named num 36
  ..- attr(*, "names")= chr "V"
 $ parameter  : NULL
 $ p.value    : num 0.0141
...
```

Therefore, if we want to extract any piece of the output (like p-value, see more in next chapters), we need to use the list indexing principles from the above:

```
> x2.wilcox$p.value
[1] 0.0141474
```

3.8.3 Data frames

Now let us turn to the one most important type of data representation, *data frames*. They bear the closest resemblance with spreadsheets and its kind, and they are most commonly used in R. Data frame is a "hybrid", "chimeric" type of R objects, *unidimensional list of same length vectors*. In other words, *data frame is a list of vectors-columns*[5].

The following scheme (Fig. 3.11) illustrates relationships between most common R data types.

[5] In fact, columns of data frames might be also matrices or *non-atomic* objects like lists or even other data frames, but this feature is rarely useful.

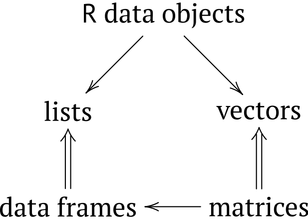


Figure 3.11: Most important R data objects.

Each column of the data frame must contain data of the same type (like in vectors), but columns themselves may be of different types (like in lists). Let us create a data frame from our existing vectors:

```
> d <- data.frame(weight=w, height=x, size=m.o, sex=sex.f)
> row.names(d) <- c("Rick", "Amanda", "Peter", "Alex", "Kathryn",
+ "Ben", "George")
> d
        weight height size    sex
Rick        69  174.0    L   male
Amanda      68  162.0    S female
Peter       93  188.0   XL   male
Alex        87  192.0  XXL   male
Kathryn     59  165.0    S female
Ben         82  168.0    M   male
George      72  172.5    L   male
```

(It was not absolutely necessary to enter `row.names()` since our `w` object could still retain names and they, by rule, will become row names of the whole data frame.)

This data frame represents data in **short form**, with many columns-features. **Long form** of the same data could, for example, look like:

```
Rick        weight    69
Rick        height    174.0
Rick        size      L
Rick        sex       male
Amanda      weight    68
...
```

In long form, features are mixed in one column, whereas the other column specifies feature id. This is really useful when we finally come to the two-dimensional data analysis.

* * *

Commands `row.names()` or `rownames()` specify names of data frame rows (*objects*). For data frame columns (*variables*), use `names()` or `colnames()`.

Alternatively, especially if objects `w`, `x`, `m.o`, or `sex.f` are for some reason absent from the workspace, you can type:

```
> d <- read.table("data/d.txt", h=TRUE, stringsAsFactors=TRUE)
> d$size <- ordered(d$size, levels=c("S", "M", "L", "XL", "XXL"))
```

... and then immediately check the structure:

```
> str(d)
'data.frame': 7 obs. of  4 variables:
 $ weight: int  69 68 93 87 59 82 72
 $ height: num  174 162 188 192 165 ...
 $ size  : Ord.factor w/ 5 levels "S"<"M"<"L"<"XL"<..: 3 1 4
 $ sex   : Factor w/ 2 levels "female","male": 2 1 2 2 1 2 2
```

Since the data frame is in fact a list, we may successfully apply to it all indexing methods for lists. More then that, data frames available for indexing also as two-dimensional matrices:

```
> d[, 1]
[1] 69 68 93 87 59 82 72
> d[[1]]
[1] 69 68 93 87 59 82 72
> d$weight
[1] 69 68 93 87 59 82 72
> d[, "weight"]
[1] 69 68 93 87 59 82 72
> d[["weight"]]
[1] 69 68 93 87 59 82 72
```

To be absolutely sure that any of two these methods output the same, run:

```
> identical(d$weight, d[, 1])
[1] TRUE
```

To select several columns (all these methods give *same* results):

```
> d[, 2:4] # matrix method
        height size    sex
Rick     174.0    L   male
Amanda   162.0    S female
```

```
Peter    188.0   XL   male
...
> d[, c("height", "size", "sex")]
        height size    sex
Rick     174.0    L   male
Amanda   162.0    S female
Peter    188.0   XL   male
...
> d[2:4] # list method
        height size    sex
Rick     174.0    L   male
Amanda   162.0    S female
Peter    188.0   XL   male
...
> subset(d, select=2:4)
        height size    sex
Rick     174.0    L   male
Amanda   162.0    S female
Peter    188.0   XL   male
...
George   172.5    L   male
> d[, -1] # negative selection
        height size    sex
Rick     174.0    L   male
Amanda   162.0    S female
Peter    188.0   XL   male
...
```

(*Three* of these methods work also for this data frame *rows*. Try all of them and **find** which are not applicable. Note also that *negative selection* works only for numerical vectors; to use several negative values, type something like `d[, -(2:4)]`. **Think** why the colon is not enough and you need parentheses here.)

Among all these ways, the most popular is the dollar sign and square brackets (Fig 3.12). While first is shorter, the second is more universal.

Figure 3.12: Two most important ways to select from data frame.

Selection by column indices is easy and saves space but it requires to remember these numbers. Here could help the `Str()` command (note the uppercase) which replaces dollar signs with column numbers (and also indicates with star* sign the presence of NAs, plus shows row names if they are not default):

```
> Str(d) # shipunov
 'data.frame': 7 obs. of  4 variables:
  1 weight: int  69 68 93 87 59 82 72
  2 height: num  174 162 188 192 165 ...
  3 size  : Ord.factor w/ 5 levels "S"<"M"<"L"<"XL"<..: 3 1 4
  4 sex   : Factor w/ 2 levels "female","male": 2 1 2 2 1 2 2
 row.names [1:7] "Rick" "Amanda" "Peter" "Alex" "Kathryn" ...
```

* * *

Now, how to make a *subset*, select several objects (rows) which have particular features? One way is through *logical vectors*. Imagine that we are interesting only in the values obtained from females:

```
> d[d$sex=="female", ]
        weight height size    sex
Amanda      68    162    S female
Kathryn     59    165    S female
```

(To select only rows, we used the *logical expression* `d$sex==female` *before the comma*.)

By itself, the above expression returns a logical vector:

```
> d$sex=="female"
[1] FALSE  TRUE FALSE FALSE  TRUE FALSE FALSE
```

This is why `R` selected only the rows which correspond to `TRUE`: 2nd and 5th rows. The result is just the same as:

```
> d[c(2, 5), ]
        weight height size    sex
Amanda      68    162    S female
Kathryn     59    165    S female
```

Logical expressions could be used to select whole rows and/or columns:

```
> d[, names(d) != "weight"]
        height size    sex
Rick     174.0    L   male
Amanda   162.0    S female
```

```
Peter    188.0   XL   male
...
```

It is also possible to apply more complicated logical expressions:

```
> d[d$size== "M" | d$size== "S", ]
        weight height size    sex
Amanda      68    162    S female
Kathryn     59    165    S female
Ben         82    168    M   male
> d[d$size %in% c("M", "L") & d$sex=="male", ]
       weight height size  sex
Rick       69  174.0    L male
Ben        82  168.0    M male
George     72  172.5    L male
```

(Second example shows how to compare with several character values at once.)

If the process of selection with square bracket, dollar sign and comma looks too complicated, there is another way, with `subset()` command:

```
> subset(d, sex=="female")
        weight height size    sex
Amanda      68    162    S female
Kathryn     59    165    S female
```

However, "classic selection" with `[` is preferable (see the more detailed explanation in `?subset`).

* * *

Selection does not only extract the part of data frame, it also allows to *replace* existing values:

```
> d.new <- d
> d.new[, 1] <- round(d.new[, 1] * 2.20462)
> d.new
        weight height size    sex
Rick       152  174.0    L   male
Amanda     150  162.0    S female
Peter      205  188.0   XL   male
...
```

(Now weight is in pounds.)

Partial matching does not work with the replacement, but there is another interesting effect:

```
> d.new$he <- round(d.new$he * 0.0328084)
> d.new
        weight height size    sex he
Rick       152  174.0    L   male  6
Amanda     150  162.0    S female  5
Peter      205  188.0   XL   male  6
...
```

(A bit mysterious, is not it? However, rules are simple. As usual, expression works *from right to left*. When we called `d.new$he` on the right, independent partial matching substituted it with `d.new$height` and converted centimeters to feet. Then replacement starts. It does not understand partial matching and therefore `d.new$he` on the left returns `NULL`. In that case, *the new column* (variable) is silently created. This is because subscripting with `$` *returns `NULL` if subscript is unknown*, creating a powerful method to add columns to the existing data frame.)

Another example of "data frame magic" is *recycling*. Data frame accumulates shorter objects if they evenly fit the data frame after being repeated several times:

```
> data.frame(a=1:4, b=1:2)
  a b
1 1 1
2 2 2
3 3 1
4 4 2
```

The following table (Table 3.2) provides a summary of R subscripting with "`[`":

subscript	effect
positive numeric vector	selects items with those indices
negative numeric vector	selects all but those indices
character vector	selects items with those names (or dimnames)
logical vector	selects the TRUE (and NA) items
missing	selects all

Table 3.2: Subscription with "`[`".

The "FizzBuzz" game has very simple rules. Any number divisible by 3 is replaced by the word "Fizz" and any number divisible by 5 by the word "Buzz". Numbers divisible by both 3 and 5 become "FizzBuzz". Can you think how to implement FizzBuzz with numbers from 1 to 100 in R? If possible, use recycling and commands `ifelse()` and `paste()`.

* * *

Command `sort()` does not work for data frames. To sort values in a `d` data frame, saying, first with sex and then with height, we have to use more complicated operation:

```
> d[order(d$sex, d$height), ]
        weight height size    sex
Amanda      68  162.0    S female
Kathryn     59  165.0    S female
Ben         82  168.0    M   male
George      72  172.5    L   male
Rick        69  174.0    L   male
Peter       93  188.0   XL   male
Alex        87  192.0  XXL   male
```

The `order()` command creates a numerical, not logical, vector with the future order of the rows:

```
> order(d$sex, d$height)
[1] 2 5 6 7 1 3 4
```

Use `order()` to arrange the *columns* of the `d` matrix in alphabetic order.

3.8.4 Overview of data types and modes

This simple table (Table 3.3) shows the four basic R objects:

(Most, but not all, vectors are also *atomic*, check it with `is.atomic()`.)

You must know the *type* (matrix, data frame *etc*.) and *mode* (numerical, character *etc*.) of object you work with. Command `str()` is especially good for that.

If any procedure wants object of some specific mode or type, it is usually easy to convert into it with `as.<something>()` command.

	linear	rectangular
all the same type	vector	matrix
mixed type	list	data frame

Table 3.3: Basic ojects.

Sometimes, you do not need the conversion at all. For example, matrices are already vectors, and all data frames are already lists (but the reverse is not correct!). On the next page, there is a table (Table 3.4) which overviews R internal data types and lists their most important features.

Data type and mode	What is it?	How to subset?	How to convert?
Vector: numeric, character, or logical	Sequence of numbers, character strings, or `TRUE/FALSE`. Made with `c()`, colon operator `:`, `scan()`, `rep()`, `seq()` *etc.*	With *numbers* like `vec[1]`. With *names* (if named) like `vec["Name"]`. With *logical expression* like `vec[vec > 3]`.	`matrix()`, `rbind()`, `cbind()`, `t()` to matrix; `as.numeric()` and `as.character()` convert modes
Vector: factor	Way of encoding vectors. Has values and *levels* (codes), and sometimes also names.	Just like vector. Factors could be also re-leveled or ordered with `factor()`.	`c()` to numeric vector, `droplevels()` removes unused levels
Matrix	**Vector** with two dimensions. All elements must be of the **same mode**. Made with `matrix()`, `cbind()` *etc.*	`matrix[2, 3]` is a cell; `matrix[2:3, ]` or `matrix[matrix[, 1] > 3, ]` rows; `matrix[, 3]` column	Matrix **is** a vector; `c()` or `dim(...) <- NULL` removes dimensions
List	Collection of anything. Could be nested (hierarchical). Made with `list()`. Most of statistical outputs are lists.	`list[2]` or (if named) `list["Name"]` is **element**; `list[[2]]` or `list$Name` **content of the element**	`unlist()` to vector, `data.frame()` only if all elements have same length
Data frame	**Named list** of anything of **same lengths** but (possibly) different modes. Data could be *short* (ids are columns) and/or *long* (ids are rows). Made with `read.table()`, `data.frame()` *etc.*	Like *matrix*: `df[2, 3]` (with numbers) or `df[, "Name"]` (with names) or `df[df[, 1] > 3, ]` (logical). Like *list*: `df[1]` or `df$Name`. Also possible: `subset(df, Name > 3)`	Data frame **is** a list; `matrix()` converts to matrix (modes will be unified); `t()` transposes and converts to matrix

Table 3.4: Overview of the most important R internal data types and ways to work with them.

3.9 Answers to exercises

Answers to the barplot coloring question:

```
> plot(sex.f, col=1:2)
```

or

```
> plot(sex.f, col=1:nlevels(sex.f))
```

(Please **try** these commands yourself. The second answer is preferable because it will work even in cases when factor has more than two levels.)

* * *

Answers to the barplot counts question. To see frequencies, from highest to smallest, **run**:

```
> rev(sort(com12))
```

or

```
> sort(com12, decreasing=TRUE)
```

* * *

Answer to flowering heads question. First, we need to load the file into R. With url.show() or simply by examining the file in the browser window, we reveal that file has multiple columns divided with wide spaces (likely Tab symbols) and that the first column contains species names *with spaces*. Therefore, header and separator should be defined explicitly:

```
> comp <- read.table(
+ "http://ashipunov.me/shipunov/open/compositae.txt",
+ h=TRUE, sep="\t")
```

Next step is always to check the structure of new object:

```
> str(comp)
'data.frame': 1396 obs. of  8 variables:
 $ SPECIES  : chr  "Achillea cartilaginea" ...
 $ PLANT    : int  118 118 118 118 118 118 ...
 $ HEIGHT   : int  460 460 460 460 460 460 ...
 $ N.CIRCLES: chr  "1" "1" "1" "1"  ...
 $ N.LEAVES : int  2 2 2 2 2 2 2 2 2 2 ...
```

```
 $ HEAD.D   : num  7 8 8 8 10 7 8 9 9 10 ...
 $ DISC.D   : int  2 2 3 3 4 3 3 3 3 3 ...
 $ RAYS     : int  7 7 7 7 7 8 8 8 8 8 ...
```

Two columns (including species) are factors, others are numerical (integer or not). The resulted object is a data frame.

Next is to select our species:

```
> c3 <- comp[comp$SPECIES %in% c("Anthemis tinctoria",
+ "Cosmos bipinnatus", "Tripleurospermum inodorum"), ]
> c3$SPECIES <- factor(c3$SPECIES)
```

To select species, we used logical expression made with `%in%` operator (please **check** how it works with `?"%in%"` command), and then converted it to factor.

Removal of redundant levels will help to use species names for scatterplot:

```
> with(c3, plot(HEAD.D, RAYS, col=as.numeric(SPECIES),
+ pch=as.numeric(SPECIES),
+ xlab="Diameter of head, mm", ylab="Number of rays"))
> legend("topright", pch=1:3, col=1:3, legend=levels(c3$SPECIES))
```

Please **make** this plot yourself. The key is to use `SPECIES` *factor as number*, with `as.numeric()` command. Function `with()` allows to ignore `cc$` and therefore saves typing.

However, there is one big problem which at first is not easy to recognize: in many places, points overlay each other and therefore amount of visible data points is much less then in the data file. What is worse, we cannot say if first and third species are well or not well segregated because we do not see how many data values are located on the "border" between them. This scatterplot problem is well known and there are workarounds:

```
> with(c3, plot(jitter(HEAD.D), jitter(RAYS),
+ col=as.numeric(SPECIES), pch=as.numeric(SPECIES),
+ xlab="Diameter of head, mm", ylab="Number of rays"))
```

Please **run** this code yourself. Function `jitter()` adds random noise to variables and shits points allowing to see what is below. However, it is still hard to understand the amount of overplotted values.

There are also:

```
> with(c3, sunflowerplot(HEAD.D, RAYS))
> with(c3, smoothScatter(HEAD.D, RAYS))
> library(hexbin)
```

```
> with(c3, plot(hexbin(HEAD.D, RAYS)))
```

Try these variants yourself. When you run the first line of code, you will see *sunflower plot*, developed exactly to such "overplotting cases". It reflects how many points are overlaid. However, it is not easy to make `sunflowerplot()` show overplotting *separately for each species*.

Two other approaches, `smoothScatter()` and `plot(hexbin())` show overplotting nicely but suffer from the same problem: they do not allow to use more than one type of points.

Package `shipunov` has function `PPoints()` which overrides this difficulty (Fig. 3.13):

```
> with(c3, plot(HEAD.D, RAYS, type="n",
+ xlab="Diameter of head, mm", ylab="Number of rays"))
> with(c3, PPoints(SPECIES, HEAD.D, RAYS, scale=.9,
+ pchs=1, centers=TRUE))
> legend("topright", pch=1, col=1:3,
+ legend=levels(c3$SPECIES), text.font=3)
```

(If you do not like dots in the centers of point symbols, remove `centers=TRUE`.)

Finally, the answer. As one might see, garden cosmos is really separate from two other species which in turn could be distinguished with some certainty, mostly because number of rays in the yellow chamomile is more than 20. This approach is possible to improve. "Data mining" chapter tells how to do that.

* * *

Answer to the matrix question. While creating matrix `ma`, we defined `byrow=TRUE`, i.e. indicated that elements should be joined into a matrix row by row. In case of `byrow=FALSE` (default) we would have obtained the matrix identical to `mb`:

```
> ma <- matrix(m, ncol=2, byrow=FALSE)
> ma
     [, 1] [, 2]
[1, ]    1    3
[2, ]    2    4
```

* * *

Answer to the "FizzBuzz" question:

```
> jj <- paste0(rep("", 100), c("", "", "Fizz"), c(rep("", 4), "Buzz"))
```

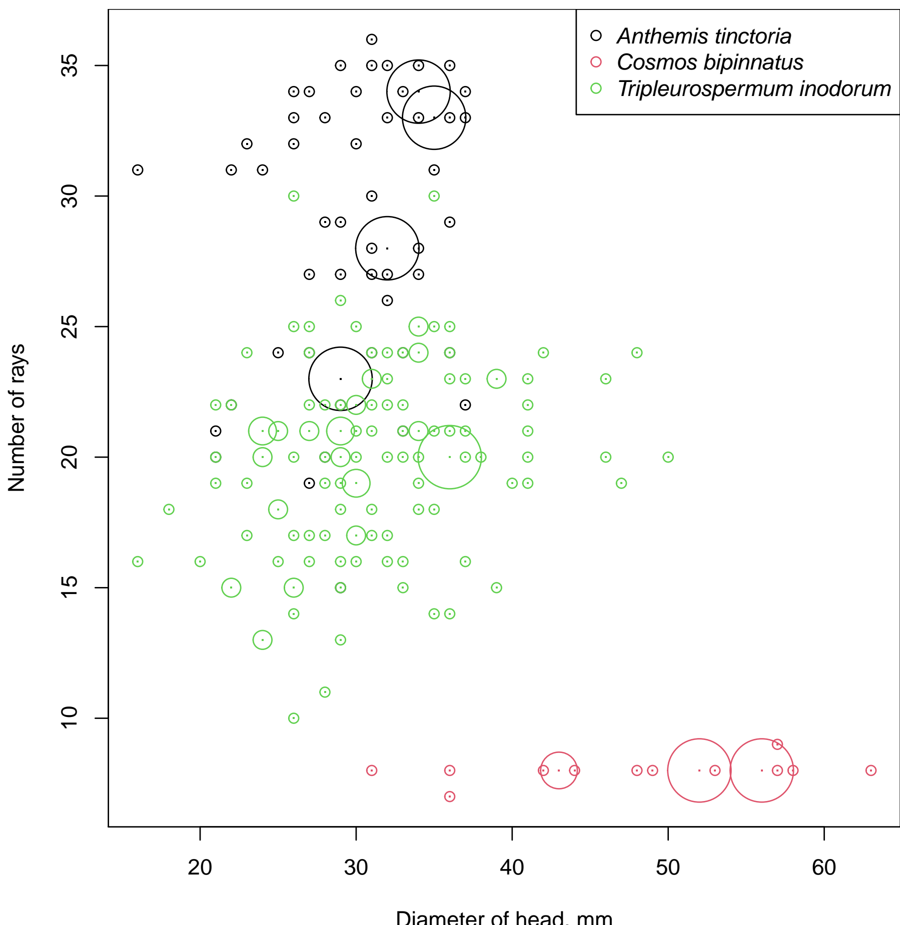


Figure 3.13: Scatterplot which shows density of data points for each species.

```
> ifelse(jj == "", 1:100, jj)
[1]  "1"        "2"        "Fizz"     "4"        "Buzz"     "Fizz"
[7]  "7"        "8"        "Fizz"     "Buzz"     "11"       "Fizz"
[13] "13"       "14"       "FizzBuzz" "16"       "17"       "Fizz"
...
```

The first line uses recycling: shorter second and third vectors recycle along the longest first vector. Here the faster `paste0(...)` command was used but it was possible to use `paste(..., sep="")` instead.

* * *

Answer to the sorting exercise. To work with columns, we have to use square brackets with a comma and place commands to the right:

```
> d.sorted <- d[order(d$sex, d$height), ]
> d.sorted[, order(names(d.sorted))]
        height    sex size weight
Amanda   162.0 female    S     68
Kathryn  165.0 female    S     59
Ben      168.0   male    M     82
George   172.5   male    L     72
Rick     174.0   male    L     69
Peter    188.0   male   XL     93
Alex     192.0   male  XXL     87
```

Please note that we cannot just type `order()` after the comma. This command returns the new order of columns, thus we gave it our column names (`names()` returns column names for a given data frame). By the way, `sort()` would have worked here too, since we only needed to rearrange a single vector.

Chapter 4

One-dimensional data

4.1 How to estimate general tendencies

It is always tempting to describe the sample with just one number "to rule them all". Or only few numbers... This idea is behind *central moments*, two (or sometimes four) numbers which represent the *center* or *central tendency* of sample and its *scale* (variation, variability, instability, dispersion: there are many synonyms).

Third and fourth central moments are not frequently used, they represent asymmetry (shift, *skewness*) and sharpness ("tailedness", *kurtosis*), respectively.

4.1.1 Median is the best

Mean is a parametric method whereas median depends less on the shape of distribution. Consequently, median is more stable, more *robust*. Let us go back to our seven hypothetical employees. Here are their salaries (thousands per year):

```
> salary <- c(21, 19, 27, 11, 102, 25, 21)
```

Dramatic differences in salaries could be explained by fact that Alex is the custodian whereas Kathryn is the owner of company.

```
> mean(salary)
[1] 32.28571
> median(salary)
[1] 21
```

We can see that mean does not reflect typical wages very well—it is influenced by higher Kathryn's salary. Median does a better job because it is calculated in a way

radically different from mean. **Median is a value that cuts off a half of ordered sample**. To illustrate the point, let us make another vector, similar to our `salary`:

```
> sort(salary1 <- c(salary, 22))
[1]  11  19  21  21  22  25  27 102
> median(salary1)
[1] 21.5
```

Vector `salary1` contains an even number of values, eight, so its median lies in the middle, between two central values (21 and 22).

There is also a way to make mean more robust to outliers, *trimmed mean* which is calculated after removal of marginal values:

```
> mean(salary, trim=0.2)
[1] 22.6
```

This trimmed mean is calculated after 10% of data was taken from each end and it is significantly closer to the median.

There is another measure of central tendency aside from median and mean. It is *mode*, the *most frequent value* in the sample. It is rarely used, and mostly applied to nominal data. Here is an example (we took the variable `sex` from the last chapter):

```
> sex <- c("male", "female", "male", "male", "female", "male",
+ "male")
> t.sex <- table(sex)
> mode <- names(t.sex[which.max(t.sex)])
> mode
[1] "male"
```

Here the most common value is `male`[1].

* * *

Often we face the task of calculating mean (or median) for the data frames. There are at least three different ways:

```
> attach(trees)
> mean(Girth)
[1] 13.24839
> mean(Height)
[1] 76
> detach(trees)
```

[1] Package `DescTools` has the handy `Mode()` function to calculate mode.

The first way uses `attach()` and adds columns from the table to the list of "visible" variables. Now we can address these variables using their names only, omitting the name of the data frame. If you choose to use this command, do not forget to `detach()` the table. Otherwise, there is a risk of loosing track of what is and is not attached. It is particularly problematic if variable names repeat across different data frames. Note that any changes made to variables will be forgotten after you `detach()`.

The second way uses `with()`) which is similar to attaching, only here attachment happens *within* the function body:

```
> with(trees, mean(Volume)) # Second way
[1] 30.17097
```

The third way uses the fact that a data frame is just a list of columns. It uses grouping functions from `apply()` family[2], for example, `sapply()` ("apply and simplify"):

```
> sapply(trees, mean)
   Girth   Height   Volume
13.24839 76.00000 30.17097
```

What if you must supply an argument to the function which is inside `sapply()`? For example, missing data will return `NA` without proper argument. In many cases this is possible to specify directly:

```
> trees.n <- trees
> trees.n[2, 1] <- NA
> sapply(trees.n, mean)
   Girth   Height   Volume
      NA 76.00000 30.17097

> sapply(trees.n, mean, na.rm=TRUE)
   Girth   Height   Volume
13.40333 76.00000 30.17097
```

In more complicated cases, you might want to define *anonymous function* (see below).

4.1.2 Quartiles and quantiles

Quartiles are useful in describing sample variability. Quartiles are values cutting the sample at points of 0%, 25%, 50%, 75% and 100% of the total distribution[3]. *Median is*

[2] While it is possible to run here a *cycle* using `for` operator, `apply`-like functions are always preferable.

[3] In the book, we include minimum and maximum into quartiles.

nothing else then the third quartile (50%). The first and the fifth quartiles are *minimum* and *maximum* of the sample.

The concept of quartiles may be expanded to obtain cut-off points at *any* desired interval. Such measures are called *quantiles* (from quantum, an increment), with many special cases, e.g. percentiles for percentages. Quantiles are used also to check the normality (see later). This will calculate quartiles:

```
> quantile(salary, c(0, 0.25, 0.5, 0.75, 1))
  0%  25%  50%  75% 100%
  11   20   21   26  102
```

Another way to calculate them:

```
> fivenum(salary)
[1]  11  20  21  26 102
```

(These two functions sometimes output slightly different results, but this is insignificant for the research. To know more, use help. Boxplots (see below) use `fivenum()`.)

The third and most commonly used way is to run `summary()`:

```
> summary(salary)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  11.00   20.00   21.00   32.29   26.00  102.00
```

`summary()` function is *generic* so it returns different results for different object types (e.g., for data frames, for measurement data and nominal data):

```
> summary(PlantGrowth)
     weight       group
 Min.   :3.590   ctrl:10
 1st Qu.:4.550   trt1:10
 Median :5.155   trt2:10
 Mean   :5.073
 3rd Qu.:5.530
 Max.   :6.310
```

In addition, `summary()` shows the number of missing data values:

```
> summary(hh)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
    8.0     8.0     8.0     8.5     8.5    10.0       3
```

* * *

Command summary() is also very useful at the first stage of analysis, for example, when we check the quality of data. It shows missing values and returns minimum and maximum:

```
> err <- read.table("data/errors.txt", sep="\t",
+ h=TRUE, stringsAsFactors=TRUE)
> str(err)
'data.frame': 7 obs. of  3 variables:
 $ AGE   : Factor w/ 6 levels "12","22","23",..: 3 4 3 5 1
 $ NAME  : Factor w/ 6 levels "","John","Kate",..: 2 3 1 4
 $ HEIGHT: num  172 163 161 16.1 132 155 183
> summary(err)
 AGE        NAME       HEIGHT
 12:1          :1   Min.   : 16.1
 22:1   John :2   1st Qu.:143.5
 23:2   Kate :1   Median :161.0
 24:1   Lucy :1   Mean   :140.3
 56:1   Penny:1   3rd Qu.:167.5
 a :1   Sasha:1   Max.   :183.0
```

We read the data file (converting text to factors) into a table and checked its structure with str(), and found that variable AGE (which must be the number) has unexpectedly turned into a factor. Output of the summary() explains why: one of age measures was mistyped as a letter a. Moreover, one of the names is empty—apparently, it should have contained NA. Finally, the minimum height is 16.1 cm! This is quite impossible even for the newborns. Most likely, the decimal point was misplaced.

4.1.3 Variation

Most common parametric measures of variation are *variance* and *standard deviation*:

```
> var(salary); sqrt(var(salary)); sd(salary)
[1] 970.9048
[1] 31.15934
[1] 31.15934
```

(As you see, standard deviation is simply the square root of variance; in fact, this function was absent from S language.)

Useful non-parametric variation measures are IQR and MAD:

```
> IQR(salary); mad(salary)
```

```
[1] 6
[1] 5.9304
```

The first measure, *inter-quartile range* (IQR), the distance between the second and the fourth quartiles. Second robust measurement of the dispersion is *median absolute deviation*, which is based on the median of absolute differences between each value and sample median.

To report central value and variability together, one of frequent approaches is to use "center ± variation". Sometimes, they do mean ± standard deviation (which mistakenly called "SEM", ambiguous term which must be avoided), but this is not robust. Non-parametric, robust methods are always preferable, therefore "median ± IQR", "median ± IQR/2" or "median ± MAD" will do the best:

```
> with(trees, paste(median(Height), IQR(Height)/2, sep="±"))
[1] "76±4"
> paste(median(trees$Height), mad(trees$Height), sep="±")
[1] "76±5.9304"
```

(Do not forget to report exactly *which* measures were used.)

To report variation only, there are more ways. For example, one can use the interval where 95% of sample lays:

```
> paste(quantile(trees$Height, c(0.025, 0.975)), collapse="-")
[1] "63.75-86.25"
```

Note that this is *not* a confidence interval because quantiles and all other descriptive statistics are about sample, not about population! However, bootstrap (described in Appendix) might help to use 95% quantiles to estimate confidence interval.

... or 95% range together with a *median*:

```
> paste(quantile(trees$Girth, c(0.025, 0.5, 0.975)), collapse="-")
[1] "8.525-12.9-18.65"
```

... or scatter of "whiskers" from the boxplot:

```
> paste(boxplot.stats(trees$Height)$stats[c(1, 5)], collapse="-")
[1] "63-87"
```

Related with scale measures are *maximum* and *minimum*. They are easy to obtain with `range()` or separate `min()` and `max()` functions. Taking alone, they are not so useful because of possible outliers, but together with other measures they might be included in the report:

```
> HS <- fivenum(trees$Height)
```

```
> paste("(", HS[1], ")", HS[2], "-", HS[4], "(", HS[5], ")", sep="")
[1] "(63)72-80(87)"
```

(Here boxplot hinges were used for the main interval.)

The figure (Fig. 4.1) summarizes most important ways to report central tendency and variation with the same Euler diagram which was used to show relation between parametric and nonparametric approaches (Fig. 3.2).

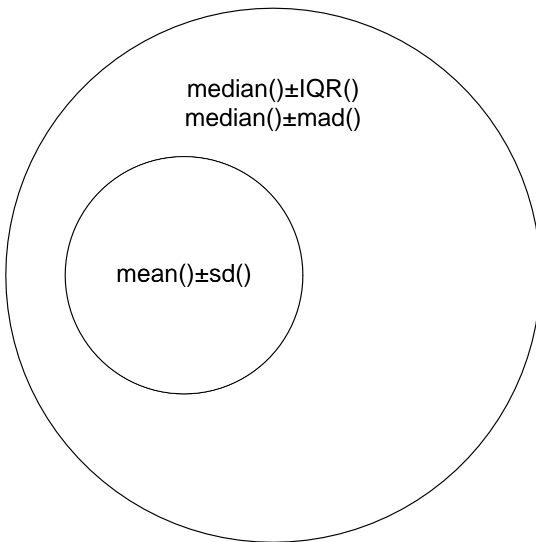


Figure 4.1: How to report center and variation in parametric (smaller circle) and all other cases (bigger circle).

* * *

To *compare the variability* of characters (especially measured in different units) one may use a dimensionless *coefficient of variation*. It has a straightforward calculation: standard deviation divided by mean and multiplied by 100%. Here are variation coefficients for trees characteristics from a built-in dataset (`trees`):

```
> 100*sapply(trees, sd)/colMeans(trees)
    Girth    Height    Volume
23.686948  8.383964 54.482331
```

(To make things simpler, we used `colMeans()` which calculated means for each column. It comes from a family of similar commands with self-explanatory names: `rowMeans()`, `colSums()` and `rowSums()`.)

4.2 1-dimensional plots

Our firm has just seven workers. How to analyze the bigger data? Let us first imagine that our hypothetical company prospers and hired one thousand new workers! We add them to our seven data points, with their salaries drawn randomly from interquartile range of the original sample (Fig. 4.2):

```
> new.1000 <- sample((median(salary) - IQR(salary)) :
+ (median(salary) + IQR(salary)), 1000, replace=TRUE)
> salary2 <- c(salary, new.1000)
> boxplot(salary2, log="y")
```

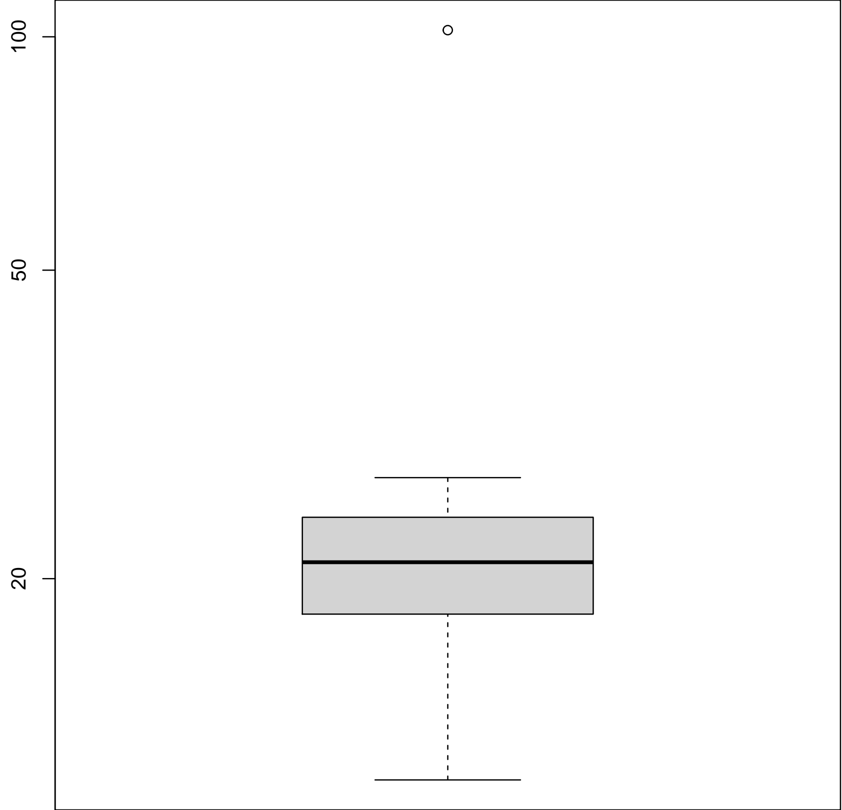


Figure 4.2: The boxplot.

In a code above we also see an example of data generation. Function `sample()` draws values randomly from a distribution or interval. Here we used `replace=TRUE`,

since we needed to pick a lot of values from a much smaller sample. (The argument `replace=FALSE` might be needed for imitation of a card game, where each card may only be drawn from a deck once.) Please keep in mind that sampling is random and therefore each iteration will give slightly different results.

Let us look at the plot. This is the boxplot ("box-and-whiskers" plot). Kathryn's salary is the highest dot. It is so high, in fact, that we had to add the parameter `log="y"` to better visualize the rest of the values. The box (main rectangle) itself is bound by second and fourth quartiles, so that its height equals IQR. Thick line in the middle is a median. By default, the "whiskers" extend to the most extreme data point which is no more than 1.5 times the interquartile range from the box. Values that lay farther away are drawn as separate points and are considered *outliers*. The scheme (Fig. 4.3) might help in understanding boxplots[4].

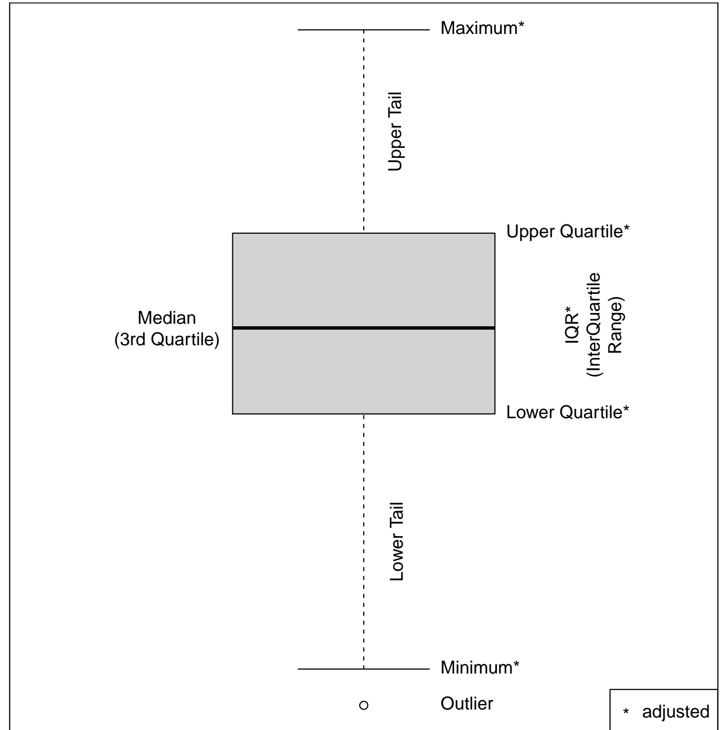


Figure 4.3: The structure of the boxplot ("box-and-whiskers" plot).

Numbers which make the boxplot might be returned with `fivenum()` command. Boxplot representation was created by a famous American mathematician John W.

[4]If you want this boxplot scheme in your `R`, run command `Ex.boxplot()` from `shipunov` package

Tukey as a quick, powerful and consistent way of reflecting main distribution-independent characteristics of the sample. In R, `boxplot()` is *vectorized* so we can draw several boxplots at once (Fig. 4.4):

```
> boxplot(scale(trees))
```

(Parameters of trees were measured in different units, therefore we `scale()`'d them.)

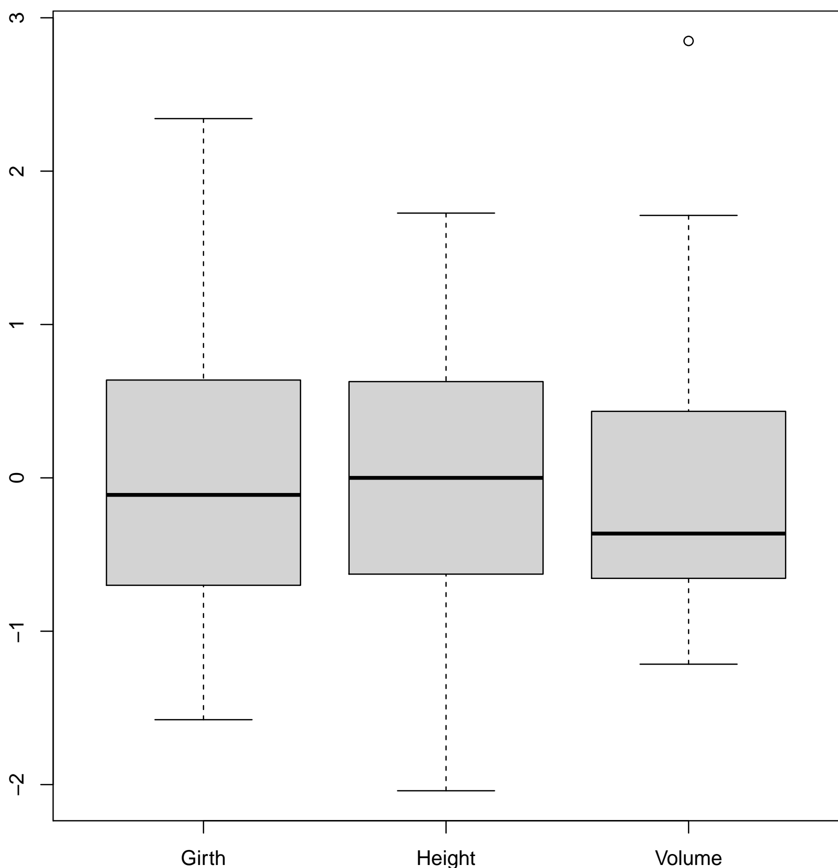


Figure 4.4: Three boxplots, each of them represents one column of the data.

Histogram is another graphical representation of the sample where range is divided into intervals (bins), and consecutive bars are drawn with their height proportional to the count of values in each bin (Fig. 4.5):

```
> hist(salary2, breaks=20, main="", xlab=0)
```

(By default, the command `hist()` would have divided the range into 10 bins, but here we needed 20 and therefore set them manually.)

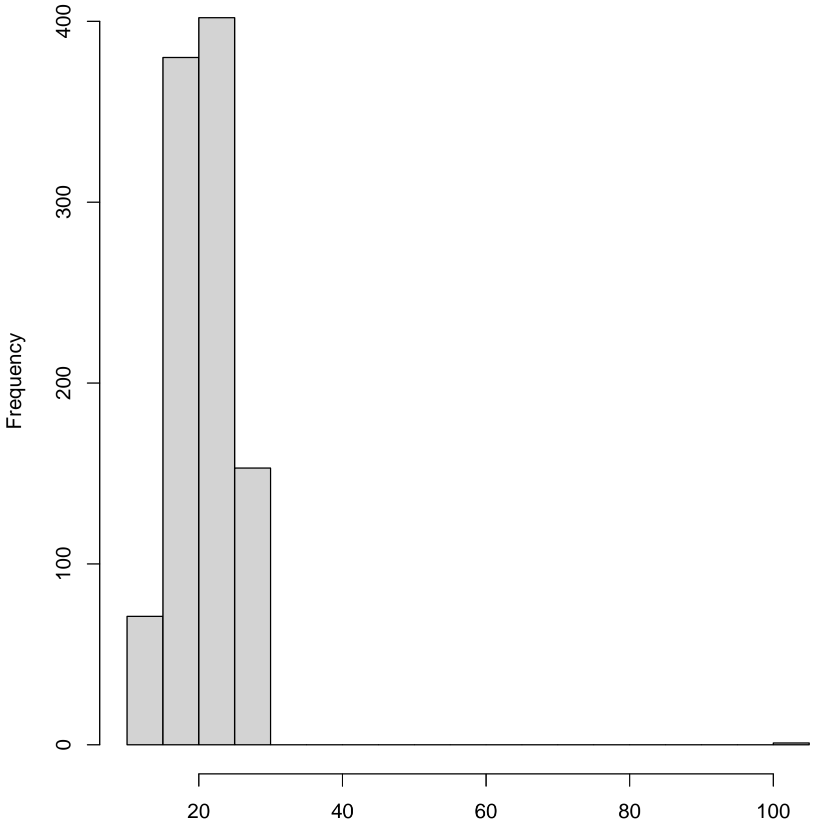


Figure 4.5: Histogram of the 1007 hypothetical employees' salaries.

Histogram is sometimes a rather cryptic way to display the data. Command `Histr()` from the `shipunov` package will plot histogram together with curve (normal or density); this is really useful, for example, to check normality. Please **try** it yourself.

A numerical analog of a histogram is the function `cut()`:

```
> table(cut(salary2, 20))
(10.9, 15.5]   (15.5, 20]   (20, 24.6] (24.6, 29.1] (29.1, 33.7]
          76          391          295          244            0
...
```

There are other graphical functions, conceptually similar to histograms. The first is *stem-and-leafs* plot. `stem()` is a kind of *pseudograph*, text histogram. Let us see how it treats the original vector `salary`:

```
> stem(salary, scale=2)
  The decimal point is 1 digit(s) to the right of the |
   1 | 19
   2 | 1157
   3 |
   4 |
   5 |
   6 |
   7 |
   8 |
   9 |
  10 | 2
```

The bar | symbol is a "stem" of the graph. The numbers in front of it are leading digits of the raw values. As you remember, our original data ranged from 11 to 102—therefore we got leading digits from 1 to 10. Each number to the left comes from the next digit of a datum. When we have several values with identical leading digit, like 11 and 19, we place their last digits in a sequence, as "leafs", to the left of the "stem". As you see, there are two values between 10 and 20, five values between 20 and 30, *etc.* Aside from a histogram-like appearance, this function performs an efficient ordering.

Another univariate instrument requires more sophisticated calculations. It is a graph of distribution density, *density plot* (Fig. 4.6):

```
> plot(density(salary2, adjust=2), main="", xlab="", ylab="")
> rug(salary2)
```

(We used an additional graphic function `rug()` which supplies an existing plot with a "ruler" which marks areas of highest data density.)

Here the histogram is *smoothed*, turned into a continuous function. The degree to which it is "rounded" depends on the parameter `adjust`.

Aside from boxplots and a variety of histograms and alike, R and external packages provide many more instruments for univariate plotting. *Beeswarm plot* requires the external package. It is similar to the base R stripchart (see `example(stripchart)` for the help) but has several advanced methods to disperse points, plus an ability to control the type of individual points (Fig. 4.7):

```
> library(beeswarm)
> trees.s <- data.frame(scale(trees), class=cut(trees$Girth, 3,
+ labels=c("thin", "medium", "thick")))
> beeswarm(trees.s[, 1:3], cex=2, col=1:3,
+ pwpch=rep(as.numeric(trees.s$class), 3))
```

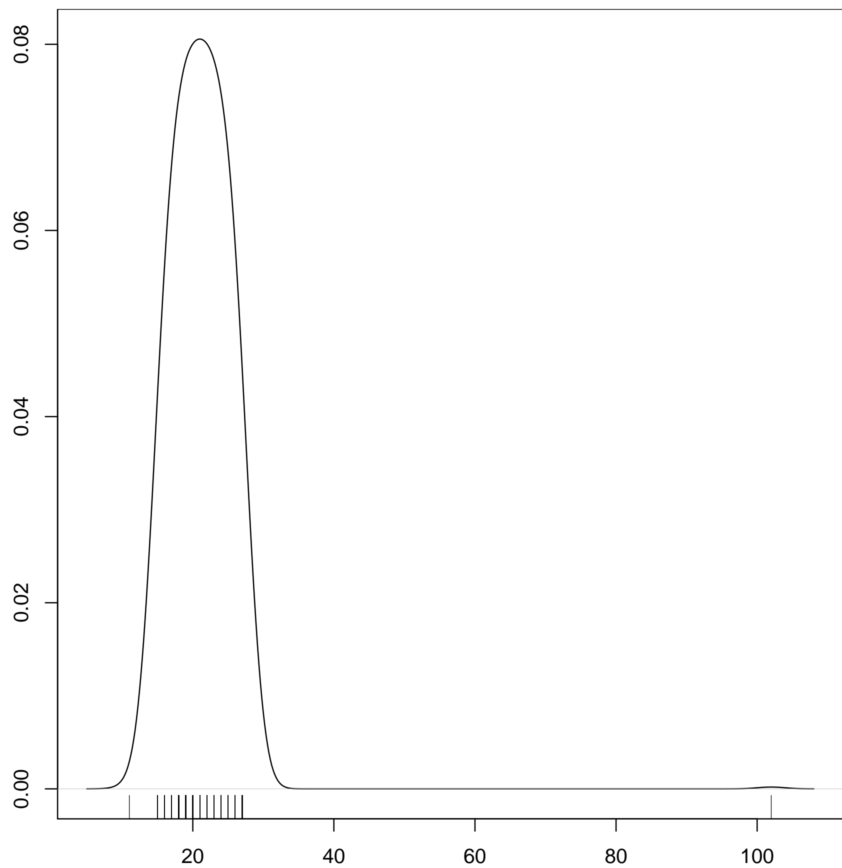


Figure 4.6: Distribution density of the 1007 hypothetical employees' salaries.

```
> bxplot(trees.s[, 1:3], add=TRUE)
> legend("top", pch=1:3, legend=levels(trees.s$class))
```

(Here with `bxplot()` command we added boxplot lines to a beehive graph in order to visualize quartiles and medians. To overlay, we used an argument `add=TRUE`.)

And one more useful 1-dimensional plot. It is a similar to both boxplot and density plot (Fig. 4.8):

```
> library(beanplot)
> beanplot(trees.s[, 1:3], col=list(1, 2, 3),
+ border=1:3, beanlines="median")
```

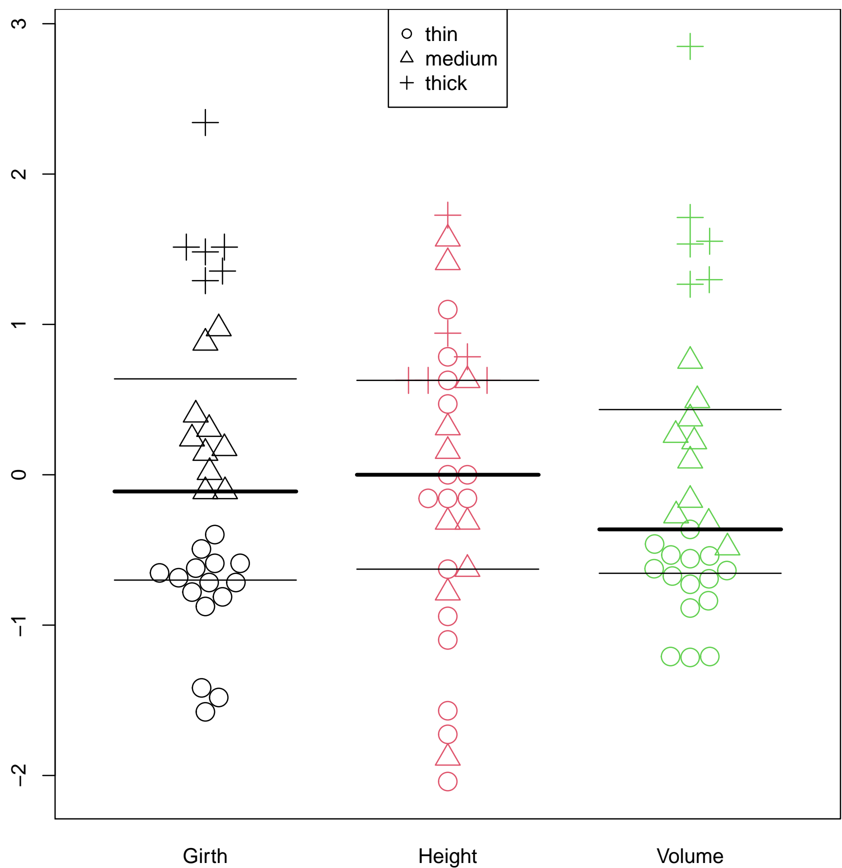


Figure 4.7: Beeswarm plot with boxplot lines.

4.3 Confidence intervals

We are ready now to make the first step in the world of inferential statistics and use *statistical tests*. They were invented to solve the main question of statistical analysis (Fig. 4.9): how to estimate anything about *population* using only its *sample*?

This sounds like a magic. How to estimate the whole population if we know nothing about it? However, it is possible if we know some data law, feature which our population should follow. For example, the population could exhibit one of *standard data distributions*.

Let us first to calculate *confidence interval*. This interval *predict* with a given probability (usually 95%) where the particular central tendency (mean or median) is located

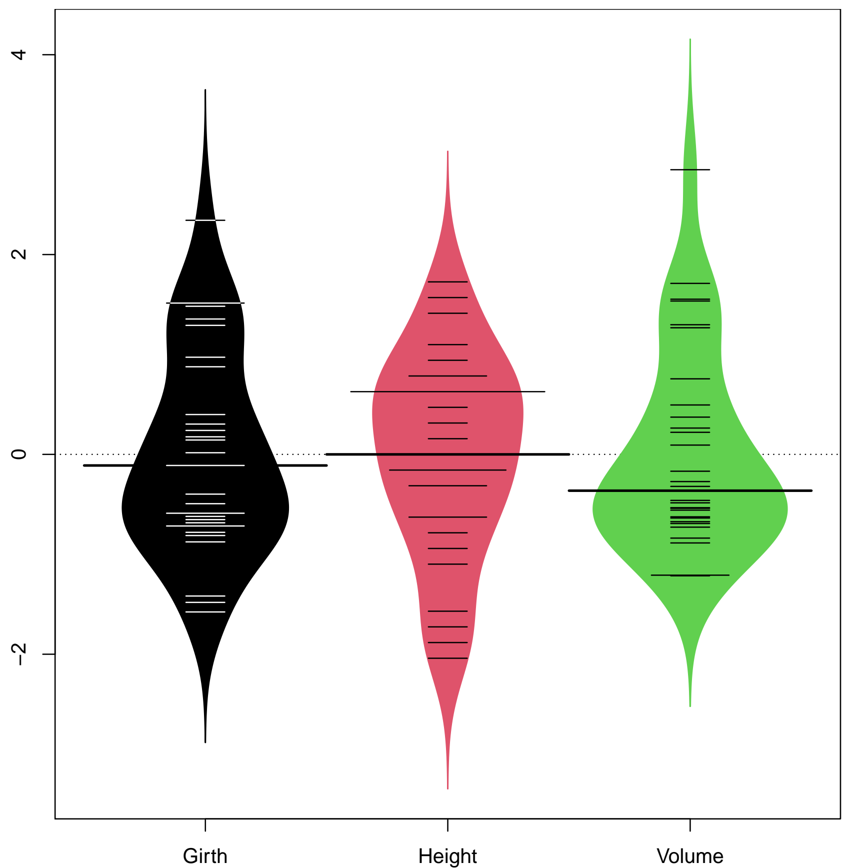


Figure 4.8: Bean plot with overall line and median lines (default lines are means).

within population. Do not mix it with the 95% quantiles, these measures have a different nature.

We start from checking the *hypothesis* that the *population mean is equal to 0*. This is our *null hypothesis*, H_0, that we wish to accept or reject based on the test results.

```
> t.test(trees$Height)
One Sample t-test
data:  trees$Height
t = 66.41, df = 30, p-value < 2.2e-16
alternative hypothesis: true mean is not equal to 0
95 percent confidence interval:
 73.6628 78.3372
sample estimates:
```

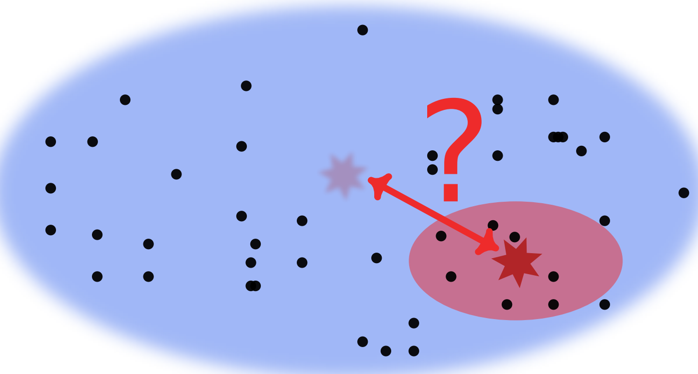


Figure 4.9: Graphic representation of the main statistical question: how to estimate population (blue) from sample (red)? Red arrow relates with the *confidence interval*. To answer "big red" question, one needs the *p-value*.

```
mean of x
       76
```

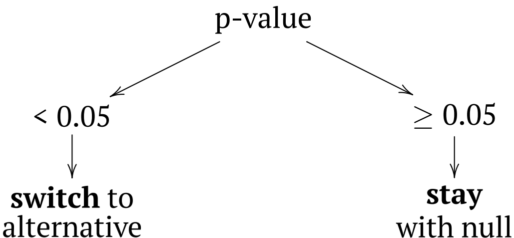
Here we used a variant of *t-test* for univariate data which in turn uses the standard *Student's t-distribution*. First, this test obtains a specific *statistic* from the original data set, so-called *t-statistic*. The test statistic is a single measure of some attribute of a sample; it reduces all the data to one value and with a help of standard distribution, allows to re-create the "virtual population".

Student test comes with some price: you should assume that your population is "parametric", "normal", i.e. interpretable with a normal distribution (dart game distribution, see the glossary).

Second, this test estimates if the statistic derived from our data can reasonably come from the distribution defined by our original assumption. This principle lies at the heart of calculating *p-value*. The latter is the probability of obtaining our test statistic if the initial assumption, *null hypothesis* was true (in the above case, mean tree height equals 0).

What do we see in the output of the test? *t-statistic* equals 66.41 at 30 degrees of freedom (df = 30). P-value is really low (2.2×10^{-16}), almost zero, and definitely much lower then the "sacred" confidence level of 0.05.

Therefore, *we reject the null hypothesis*, or our initial assumption that mean tree height equals to 0 and consequently, go with the *alternative hypothesis* which is a logical opposite of our initial assumption (i.e., "height is *not* equal to 0"):



The importance of p-value has been often exaggerated. To maintain the realistic approach to the hypothesis testing, statisticians recently invented the S-value. It shows how many bits of information are against the null hypothesis:

```
> S.value(t.test(trees$Height)) # shipunov
[1] 110.9296
```

This is very high S-value, analogous to getting 110 heads in a trial of "fairness" with 110 coin tosses.

However, the most important at the moment is the *confidence interval*—a range into which the true, population mean should fall with given probability (95%). Here it is narrow, spanning from 73.7 to 78.3 and *does not include zero*. The last means again that null hypothesis is not supported.

* * *

If your data does not go well with normal distribution, you need more universal (but less powerful) *Wilcoxon* rank-sum test. It uses *median* instead of mean to calculate the test statistic V. Our null hypothesis will be that *population median is equal to zero*:

```
> wilcox.test(salary, conf.int=TRUE)
Wilcoxon signed rank test with continuity correction
data:  salary
V = 28, p-value = 0.02225
alternative hypothesis: true location is not equal to 0
95 percent confidence interval:
 15.00002 64.49995
sample estimates:
(pseudo)median
            23
...
```

(Please ignore warning messages, they simply say that our data has ties: two salaries are identical.)

Here we will also reject our null hypothesis with a high degree of certainty. Passing an argument `conf.int=TRUE` will return the confidence interval for population median—it is broad (because sample size is small) but does not include zero.

4.4 Normality

How to decide which test to use, parametric or non-parametric, t-test or Wilcoxon? We need to know if the distribution follows or at least approaches normality. This could be checked visually (Fig. 4.10):

```
> qqnorm(salary2, main=""); qqline(salary2, col=2)
```

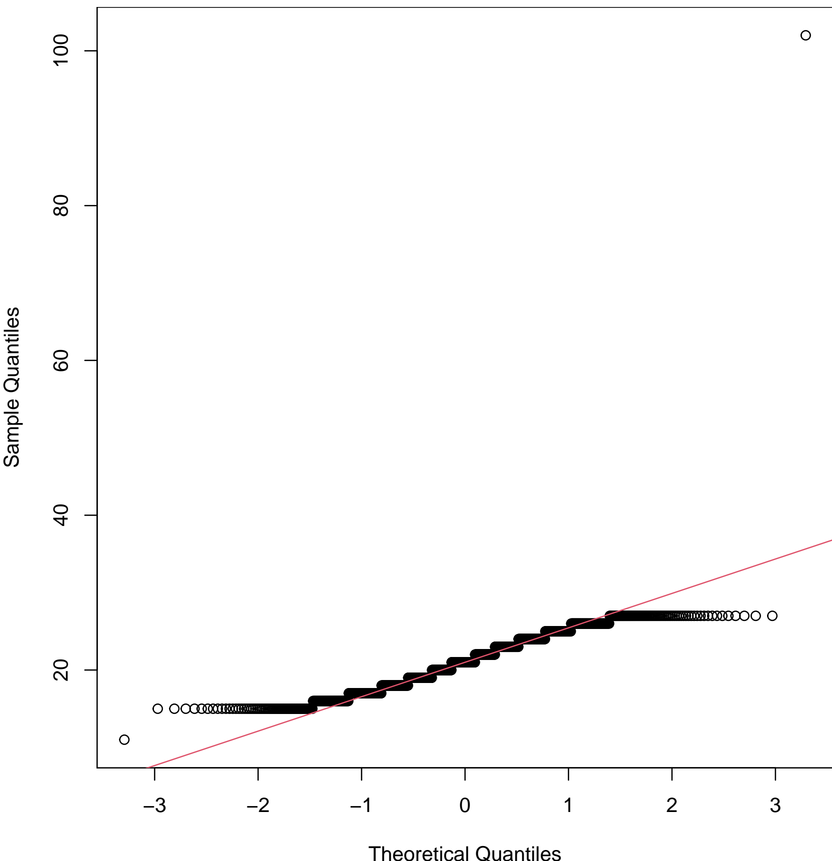


Figure 4.10: Graphical check for the normality.

How does QQ plot work? First, data points are ordered and each one is assigned to a quantile. Second, a set of theoretical quantiles—positions that data points should

have occupied in a *normal distribution*—is calculated. Finally, theoretical and empirical quantiles are paired off and plotted.

We have overlaid the plot with a line coming through quartiles. When the dots follow the line closely, the empirical distribution is normal. Here a lot of dots at the tails are far. Again, we conclude, that the original distribution is not normal.

R also offers numerical instruments that check for normality. The first among them is Shapiro-Wilk test (please **run** this code yourself):

```
> shapiro.test(salary)
> shapiro.test(salary2)
```

Here the output is rather terse. P-values are small, but what was the null hypothesis? Even the built-in help does not state it. To understand, we may run a simple experiment:

```
> set.seed(1638) # freeze random number generator
> shapiro.test(rnorm(100))
Shapiro-Wilk normality test
data:  rnorm(100)
W = 0.9934, p-value = 0.9094
```

The command `rnorm()` generates random numbers that follow normal distribution, as many of them as stated in the argument. Here we have obtained a p-value approaching unity. Clearly, the null hypothesis was "the empirical distribution is normal".

Armed with this little experiment, we may conclude that distributions of both `salary` and `salary2` are not normal.

Kolmogorov-Smirnov test works with two distributions. The null hypothesis is that both samples came from the same population. If we want to test one distribution against normal, second argument should be `pnorm`:

```
> ks.test(scale(salary2), "pnorm")
    One-sample Kolmogorov-Smirnov test
data:  scale(salary2)
D = 0.094707, p-value = 2.856e-08
alternative hypothesis: two-sided
...
```

(The result is comparable with the result of Shapiro-Wilk test. We scaled data because by default, the second argument uses scaled normal distribution.)

Function `ks.test()` accepts any type of the second argument and therefore could be used to check how reliable is to approximate current distribution with *any* theoretical distribution, not necessarily normal. However, Kolmogorov-Smirnov test often returns the wrong answer for samples which size is < 50, so it is less powerful then Shapiro-Wilks test.

`2.2e-16` us so-called *exponential notation*, the way to show really small numbers like this one (2.2×10^{-16}). If this notation is not comfortable to you, there is a way to get rid of it:

```
> old.options <- options(scipen=100)
> ks.test(scale(salary2), "pnorm")
    One-sample Kolmogorov-Smirnov test
data:  scale(salary2)
D = 0.094707, p-value = 0.00000002856
alternative hypothesis: two-sided
...
> options(old.options)
```

(Option `scipen` equals to the maximal allowable number of zeros.)

Most of times these three ways to determine normality are in agreement, but this is not a surprise if they return different results. Normality check is not a death sentence, it is just an opinion based on probability.

Again, if sample size is small, statistical tests and even quantile-quantile plots frequently fail to detect non-normality. In these cases, simpler tools like stem plot or histogram, would provide a better help.

4.5 How to create your own functions

Shapiro-Wilk test is probably the fastest way to check normality but its output is not immediately understandable. It is also not easy to apply for whole data frames. Let us create the **function** which overcomes these problems:

```
> Normality <- function(a)
+ {
+ ifelse(shapiro.test(a)$p.value < 0.05, "NOT NORMAL", "NORMAL")
+ }
```

(We used here the fact that in `R`, test output is usually a *list* and each component is possible to extract using `$`-name approach described in previous chapter. How to know what to extract? Save test output into object and run `str(obj)`!)

Package `shipunov` contains slightly more advanced version of the `Normality()` which takes into account that Shapiro-Wilks test is not so reliable for small size (< 25) samples.

To make this `Normality()` function work, you need to copy the above text into R console, or into the separate file (preferably with `*.r` extension), and then load it with `source()` command. Next step is to call the function:

```
> Normality(salary) # shipunov
> sapply(trees, Normality)
> sapply(log(trees+0.01), Normality)
```

(Note that logarithmic conversion could change the normality. **Check** yourself if square root does the same.)

This function not only runs Shapiro-Wilks test several times but also outputs an easily readable result. Most important is the third row which uses p-value extracted from the test results. Extraction procedure is based on the knowledge of the internal structure of `shapiro.test()` output.

> How to know the structure of `shapiro.test()` output object without going into help?

* * *

In many cases, "stationary", named function is not necessary as user need some piece of code which runs only once (but runs in relatively complicated way). Here helps the *anonymous function*. It is especially useful within functions of `apply()` family. This is how to calculate mode simultaneously in multiple columns:

```
> sapply(chickwts,
+ function(.x) names(table(.x))[which.max(table(.x))])
   weight      feed

    "248" "soybean"
```

(Here we followed the agreement that in the anonymous functions, argument names must start with a dot.)

Even more useful—simultaneous confidence intervals:

```
> old.options <- options(warn=-1)
> sapply(trees,
+ function(.x) wilcox.test(.x, conf.int=TRUE)$conf.int)
```

```
       Girth   Height   Volume
[1,] 11.84995 73.50001 22.00000
[2,] 14.44990 78.50003 36.05001
> options(old.options)
```

(Here we suppressed multiple "ties" warnings. **Do not** do it yourself without a strong reason!)

> File `betula.txt` in the open data repository contains measurements of several birch morphological characters. Are there any characters which could be analyzed with parametric methods?

> Please make the user function `CV()` which calculates coefficient of variation (CV) and apply it to `betula` data. Which character is most variable? (File `betula_c.txt` contains explanation of variables.)

> In the first chapter, we used `dact.txt` data to illustrate situation when it is really hard to say anything about data without statistical analysis. Now, time came to make this analysis. Provide as many *relevant* description characteristics as possible, calculate the appropriate confidence interval and plot this data.

> In the open repository, file `nymphaeaceae.txt` contains counts of flower parts taken from two members of water lily family (Nymphaeaceae), *Nuphar lutea* (yellow water lily) and *Nymphaea candida* (white water lily). Using plots and, more importantly, confidence intervals, find which characters (except `SEPALS`) distinguish these species most. Provide the answer in the form "if the character is ..., then species is ..".

4.6 How good is the proportion?

So far, out methods are good for measurement data (and to some extent, for ranked data). But how about categorical data? There are two possible ways to convert them into numbers: either make dummy binary variables or simply count entries of each type, *tabulate* them:

```
> table(sex)
sex
```

```
female   male
     2      5
> table(sex)/length(sex)
sex
   female      male
0.2857143 0.7142857
```

So we found that proportion of females in our small firm is about 29%. Now we need to ask the main statistical question: what is the proportion of females in the whole population (all similar firms)? Can we estimate it from our sample?

```
> prop.test(table(sex))
1-sample proportions test with continuity correction
data:  table(sex), null probability 0.5
X-squared = 0.57143, df = 1, p-value = 0.4497
alternative hypothesis: true p is not equal to 0.5
95 percent confidence interval:
 0.05112431 0.69743997
sample estimates:
        p
0.2857143
Warning message:
In prop.test(table(sex)) : Chi-squared approximation may be incorrect
```

The *test of proportions* tried to find the confidence interval for the proportion of females and also to check the null hypothesis if this proportion might be 50% (just a half). As you see, confidence interval is really broad and it is better to stay with null. Despite the first impression, it is possible that firms of this type have equal number of male and female employees.

Here is another example. In hospital, there was a group of 476 patients undergoing specific treatment and 356 among them are smokers (this is the old data). In average, proportion of smokers is slightly less than in our group (70% *versus* 75%, respectively). To check if this difference is real, we can run the *proportions test*:

```
> prop.test(x=356, n=476, p=0.7, alternative="two.sided")
1-sample proportions test with continuity correction
data:  356 out of 476, null probability 0.7
X-squared = 4.9749, df = 1, p-value = 0.02572
alternative hypothesis: true p is not equal to 0.7
95 percent confidence interval:
 0.7059174 0.7858054
sample estimates:
        p
```

```
0.7478992
```

(We used `two.sided` option to check both variants of inequality: larger and smaller. To check one of them ("one tail"), we need `greater` or `less`[5].)

Confidence interval is narrow. Since the null hypothesis was that "true probability of is equal to 0.7" and p-value was less than 0.05, we *reject* it in favor to alternative hypothesis, "true probability of is not equal to 0.7". Consequently, proportion of smokers in our group is *different* from their proportion in the whole hospital.

* * *

Now to the example from foreword. Which candidate won, A or B? Here the proportion test will help again[6]:

```
> prop.test(x=0.52*262, n=262, p=0.5, alternative="greater")
1-sample proportions test with continuity correction
data:  0.52 * 262 out of 262, null probability 0.5
X-squared = 0.34302, df = 1, p-value = 0.279
alternative hypothesis: true p is greater than 0.5
95 percent confidence interval:
 0.4673901 1.0000000
sample estimates:
   p
0.52
```

According to the confidence interval, the real proportion of people voted for candidate A varies from 100% to 47%. This might change completely the result of elections!

Large p-value suggests also that we cannot reject the null hypothesis. We must conclude that "true p is not greater then 0.5". Therefore, using only that data it is *impossible* to tell if candidate A won the elections.

All in all, proportion tests and more advanced methods (like Chi-square test, see below) help to avoid stereotyping errors.

> This exercise is related with phyllotaxis (Fig. 4.11), botanical phenomenon when leaves on the branch are distributed in accordance to the particular rule. Most amazingly, this rule (*formulas of phyllotaxis*) is quite often the *Fibonacci rule*, kind

[5] Note that these options must be set *a priori*, before you run the test. It is not allowed to change alternatives in order to find a better p-values.

[6] Look also into the end of this chapter.

of fraction where numerators and denominators are members of the famous Fibonacci sequence. We made R function `Phyllotaxis()` which produces these fractions:

```
> sapply(1:10, Phyllotaxis) # shipunov
 [1] "1/2"    "1/3"    "2/5"    "3/8"    "5/13"   "8/21"
 [7] "13/34"  "21/55"  "34/89"  "55/144"
```

In the open repository, there is a data file `phyllotaxis.txt` which contains measurements of phyllotaxis in nature. Variables `N.CIRCLES` and `N.LEAVES` are numerator and denominator, respectively. Variable `FAMILY` is the name of plant family. Many formulas in this data file belong to "classic" Fibonacci group (see above), but some do not. Please count proportions of non-classic formulas per family, determine which family is the most deviated and check if the proportion of non-classic formulas in this family is statistically different from the average proportion (calculated from the whole data).

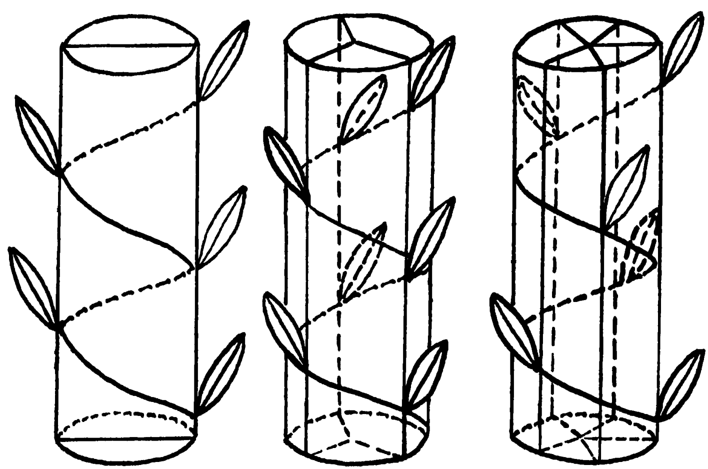


Figure 4.11: Phyllotaxis. From left to right: leaves arranged by ½, ⅓ and ⅖ formulas of phyllotaxis.

4.7 Answers to exercises

Answer to the question of `shapiro.test()` output structure. First, we need to recollect that almost everything what we see on the R console, is the result of `print()`'ing some lists. To extract the component from a list, we can call it by dollar sign and

name, or by square brackets and number (if component is not named). Let us check the structure with `str()`:

```
> str(shapiro.test(rnorm(100)))
List of 4
 $ statistic: Named num 0.992
  ..- attr(*, "names")= chr "W"
 $ p.value  : num 0.842
 $ method   : chr "Shapiro-Wilk normality test"
 $ data.name: chr "rnorm(100)"
 - attr(*, "class")= chr "htest"
```

Well, p-value most likely comes from the `p.value` component, this is easy. Check it:

```
> set.seed(1683)
> shapiro.test(rnorm(100))$p.value
[1] 0.8424077
```

This is what we want. Now we can insert it into the body of our function.

* * *

Answer to the "birch normality" exercise. First, we need to check the data and understand its structure, for example with `url.show()`. Then we can read it into R, check its variables and apply `Normality()` function to all appropriate columns:

```
> betula <- read.table(
+ "http://ashipunov.me/shipunov/open/betula.txt", h=TRUE)
> Str(betula) # shipunov
 'data.frame': 229 obs. of  10 variables:
  1  LOC      : int  1 1 1 1 1 1 1 1 1 1 ...
  2  LEAF.L   : int  50 44 45 35 41 53 50 47 52 42 ...
  3  LEAF.W   : int  37 29 37 26 32 37 40 36 39 40 ...
  4  LEAF.MAXW: int  23 20 19 15 18 25 21 21 23 19 ...
  5  PUB      : int  0 1 0 0 0 0 0 0 1 0 ...
  6  PAPILLAE : int  1 1 1 1 1 1 1 1 1 1 ...
  7  CATKIN   : int  31 25 21 20 24 22 40 25 14 27 ...
  8  SCALE.L  : num  4 3 4 5.5 5 5 6 5 5 5 ...
  9  LOBES    * int  0 0 1 0 0 0 0 0 1 0 ...
 10  WINGS    * int  0 0 0 0 0 0 1 0 0 1 ...
> sapply(betula[, c(2:4, 7:8)], Normality) # shipunov
      LEAF.L       LEAF.W    LEAF.MAXW       CATKIN      SCALE.L
"NOT NORMAL" "NOT NORMAL" "NOT NORMAL"     "NORMAL" "NOT NORMAL"
```

(Note how only non-categorical columns were selected for the normality check. We used Str() because it helps to check numbers of variables, and shows that two variables, LOBES and WINGS have missing data. There is no problem in using str() instead.)

Only CATKIN (length of female catkin) is available to parametric methods here. It is a frequent case in biological data.

What about the graphical check for the normality, histogram or QQ plot? Yes, it should work but we need to repeat it 5 times. However, lattice package allows to make it in two steps and fit on one *trellis plot* (Fig. 4.12):

```
> betula.s <- stack(betula[, c(2:4, 7:8)])
> qqmath(~ values | ind, data=betula.s,
+ panel=function(x) {panel.qqmathline(x); panel.qqmath(x)})
```

(Library lattice requires *long data format* where all columns stacked into one and data supplied with identifier column, this is why we used stack() function and formula interface.

There are many trellis plots. Please **check** the *trellis histogram* yourself:

```
> bwtheme <- standard.theme("pdf", color=FALSE)
> histogram(~ values | ind, data=betula.s, par.settings=bwtheme)
```

(There was also an example of how to apply grayscale theme to these plots.)

As one can see, SCALE.L could be also accepted as "approximately normal". Among others, LEAF.MAXW is "least normal".

* * *

Answer to the birch characters variability exercise. To create a function, it is good to start from *prototype*:

```
> CV <- function(x) {}
```

This prototype does nothing, but on the next step you can improve it, for example, with fix(CV) command. Then test CV() with some simple argument. If the result is not satisfactory, fix(CV) again. At the end of this process, your function (actually, it "wraps" CV calculation explained above) might look like:

```
> CV <- function(x)
+ {
+ 100*sd(x, na.rm=TRUE)/mean(x, na.rm=TRUE)
+ }
```

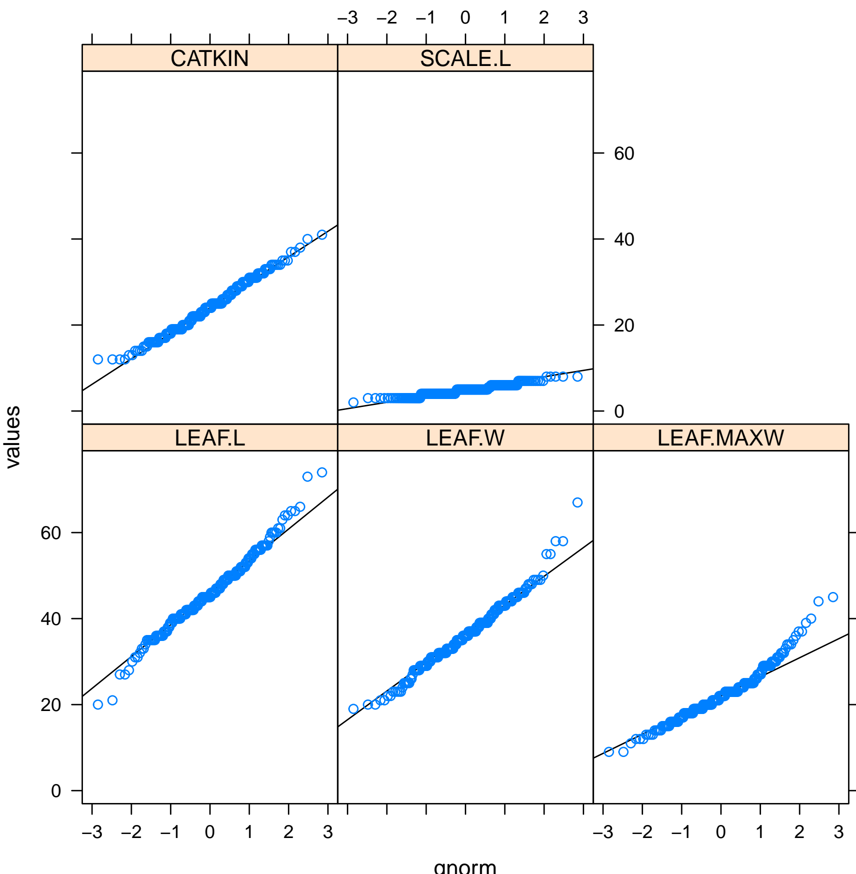


Figure 4.12: Normality QQ trellis plots for the five measurement variables in `betula` dataset (variables should be read from bottom to top).

Then `sapply()` could be used to check variability of each measurement column:

```
> sapply(betula[, c(2:4, 7:8)], CV)
   LEAF.L     LEAF.W LEAF.MAXW    CATKIN   SCALE.L
 17.93473   20.38630  26.08806  24.17354  24.72061
```

As one can see, `LEAF.MAXW` (location of the maximal leaf width) has the biggest variability. In the `shipunov` package, there is `CVs()` function which implements this and three other measurements of relative variation.

* * *

Answer to question about `dact.txt` data. Companion file `dact_c.txt` describes it as a random extract from some plant measurements. From the first chapter, we know that it is just one sequence of numbers. Consequently, `scan()` would be better than `read.table()`. First, load and check:

```
> dact <- scan("data/dact.txt")
Read 48 items
> str(dact)
 num [1:48] 88 22 52 31 51 63 32 57 68 27 ...
```

Now, we can check the normality with our new function:

```
> Normality(dact) # shipunov
[1] "NOT NORMAL"
```

Consequently, we must apply to `dact` only those analyses and characteristics which are robust to non-normality:

```
> summary(dact)[-4] # no mean
   Min. 1st Qu.  Median 3rd Qu.    Max.
   0.00   22.00   33.50   65.25  108.00
> IQR(dact)
[1] 43.25
> mad(dact)
[1] 27.4281
```

Confidence interval for the median:

```
> wilcox.test(dact, conf.int=TRUE)$conf.int
[1] 34.49995 53.99997
attr(,"conf.level")
[1] 0.95
Warning messages:
... # please ignore warnings
```

(Using the idea that every test output is a *list*, we extracted the confidence interval from output directly. Of course, we knew beforehand that name of a component we need is `conf.int`; this knowledge could be obtained from the function help (section "Value"). The resulted interval is broad.)

To plot single numeric data, histogram (Fig. 4.13) is preferable (boxplots are better for comparison between variables):

```
> Histr(dact, xlab="", main="") # shipunov
```

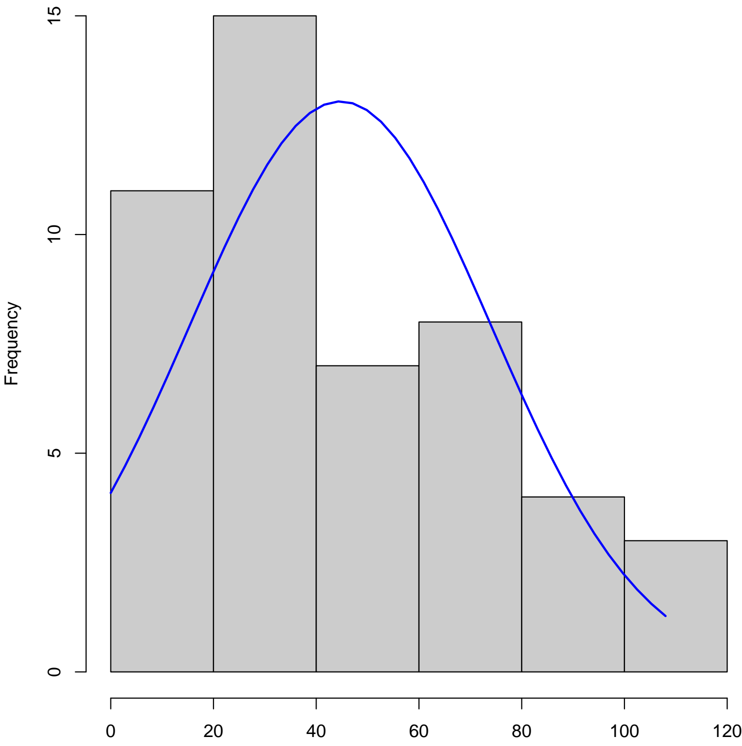


Figure 4.13: Histogram with overlaid normal distribution curve for `dact` data.

Similar to histogram is the steam-and-leaf plot:

```
> stem(dact)
  The decimal point is 1 digit(s) to the right of the |
   0 | 0378925789
   2 | 0224678901122345
   4 | 471127
   6 | 035568257
   8 | 2785
  10 | 458
```

In addition, here we will calculate *skewness* and *kurtosis*, third and fourth central moments (Fig. 4.14). Skewness is a measure of how asymmetric is the distribution, kurtosis is a measure of how spiky is it. Normal distribution has both skewness and kurtosis zero whereas "flat" uniform distribution has skewness zero and kurtosis approximately −1.2 (**check** it yourself).

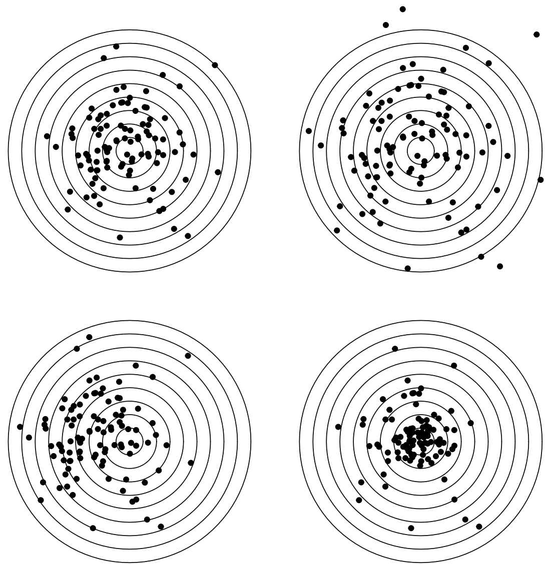


Figure 4.14: Central moments (left to right, top to bottom): default, different scale, different skewness, different kurtosis.

What about `dact` data? From the histogram (Fig. 4.13) and stem-and-leaf we can predict positive skewness (asymmetricity of distribution) and negative kurtosis (distribution flatter than normal). To check, one need to load library `e1071` first:

```
> library(e1071)
> skewness(dact)
[1] 0.5242118
> kurtosis(dact)
[1] -0.8197875
```

* * *

Answer to the question about water lilies. First, we need to check the data, load it into R and check the resulted object:

```
> ny <- read.table(
+ "http://ashipunov.me/shipunov/open/nymphaeaceae.txt",
+ h=TRUE, sep="\t")
> Str(ny) # shipunov
 'data.frame': 267 obs. of  5 variables:
  1 SPECIES: chr  "Nuphar lutea" "Nuphar lutea" ...
```

```
 2 SEPALS : int  4 5 5 5 5 5 5 5 5 5 ...
 3 PETALS : int  14 10 10 11 11 11 12 12 12 12 ...
 4 STAMENS* int  131 104 113 106 111 119 73 91 102 ...
 5 STIGMAS* int  13 12 12 11 13 15 10 12 12 11 ...
```

(Function `Str()` shows column numbers and the presence of `NA`.)

One of possible ways to proceed is to examine differences between species by each character, with four paired boxplots. To make them in one row, we will employ `for()` cycle:

```
> oldpar <- par(mfrow=c(2, 2))
> for (i in 2:5) boxplot(ny[, i] ~ ny[, 1], main=names(ny)[i])
> par(oldpar)
```

(Not here, but in many other cases, `for()` in R is better to replace with commands of `apply()` family. Boxplot function accepts "ordinary" arguments but in this case, formula interface with tilde is much more handy.)

Please **review this plot** yourself.

It is even better, however, to compare *scaled characters* in the *one* plot. First variant is to load `lattice` library and create trellis plot similar to Fig. 7.8 or Fig. 7.7:

```
> library(lattice)
> ny.s <- stack(as.data.frame(scale(ny[ ,2:5])))
> ny$SPECIES <- as.factor(ny$SPECIES)
> ny.s$SPECIES <- ny$SPECIES
> bwplot(SPECIES ~ values | ind, ny.s, xlab="")
```

(As usual, trellis plots "want" long form and formula interface.)

Please **check** this plot yourself.

Alternative is the `Boxplots()` (Fig. 4.15) command. It is not a trellis plot, but designed with a similar goal to compare many things at once:

```
> Boxplots(ny[, 2:5], ny$SPECIES, srt=0, adj=c(.5, 1)) # shipunov
```

(By default, `Boxplots()` rotates character labels, but this behavior is not necessary with 4 characters. This plot uses `scale()` so y-axis is, by default, not provided.)

Or, with even more crisp `Linechart()` (Fig. 4.16):

```
Linechart
> Linechart(ny[, 2:5], ny[, 1], se.lwd=2) # shipunov
```

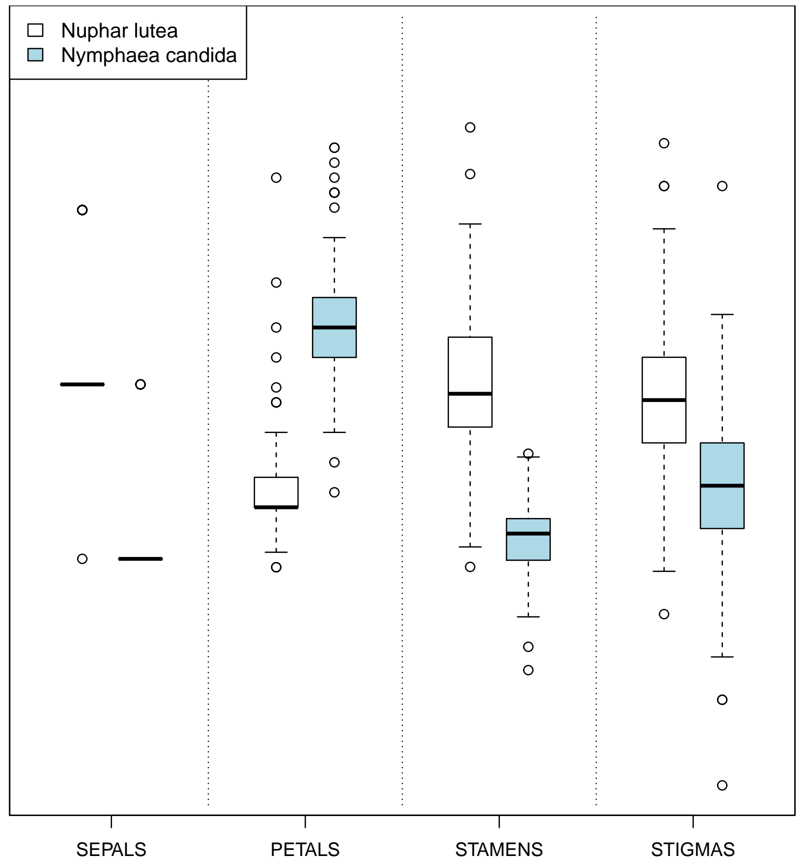


Figure 4.15: Grouped boxplots with `Boxplots()` function.

(Sometimes, IQRs are better to percept if you add `grid()` to the plot. **Try** it yourself. By the way, if you have just one species, use `Dotchart3()` function.)

Evidently (after `SEPALS`), `PETALS` and `STAMENS` make the best species resolution. To obtain numerical values, it is better to *check the normality* first.

Note that species identity is the natural, internal feature of our data. Therefore, it is theoretically possible that the same character in one species exhibit normal distribution whereas in another species does not. This is why normality should be checked *per character per species*. This idea is close to the concept of *fixed effects* which are so useful in linear models (see next chapters). Fixed effects oppose the random effects which are not natural to the objects studied (for example, if we sample *only one* species of water lilies in the lake *two times*).

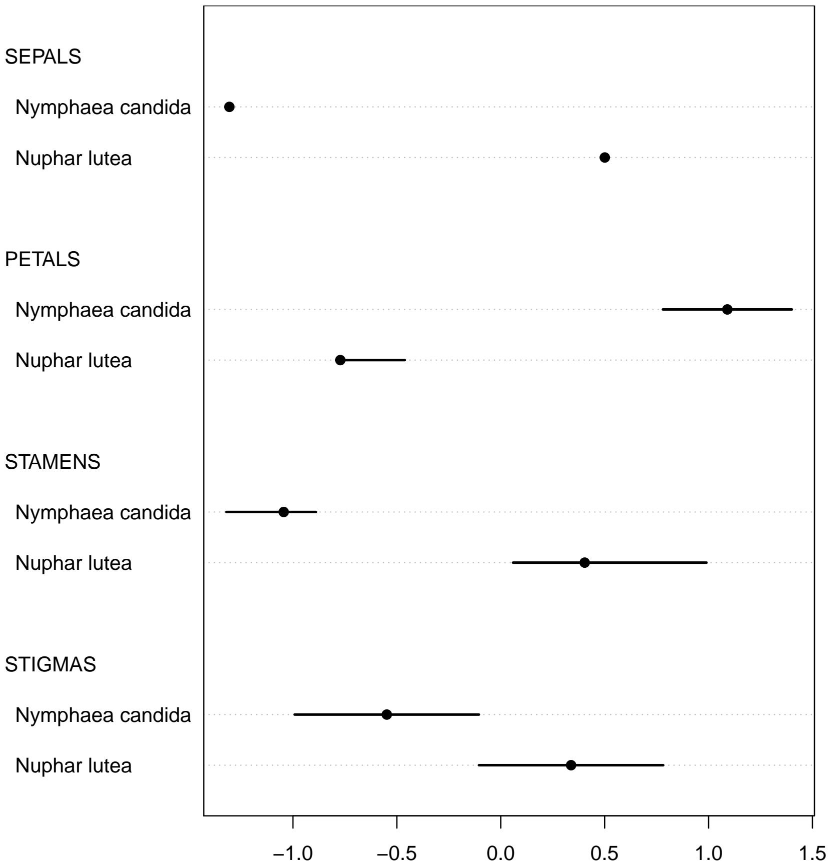


Figure 4.16: Grouped medians and IQRs with `Linechart()` function.

```
> aggregate(ny[, 3:4], by=list(SPECIES=ny[, 1]), Normality) # shipunov
          SPECIES     PETALS    STAMENS
1     Nuphar lutea NOT NORMAL NOT NORMAL
2 Nymphaea candida NOT NORMAL NOT NORMAL
```

(Function `aggregate()` does not only apply anonymous function to all elements of its argument, but also splits it on the fly with `by` list of factor(s). Similar is `tapply()` but it works only with one vector. Another variant is to use `split()` and then `apply()` reporting function to the each part separately.)

By the way, the code above is good for learning but in our particular case, normality check is not required! This is because numbers of petals and stamens are *discrete* characters and therefore must be treated with nonparametric methods *by definition*.

Thus, for confidence intervals, we should proceed with nonparametric methods:

```
> aggregate(ny[, 3:4],
+ by=list(SPECIES=ny[, 1]),
+ function(.x) wilcox.test(.x, conf.int=TRUE)$conf.int)
          SPECIES PETALS.1 PETALS.2 STAMENS.1 STAMENS.2
1     Nuphar lutea 14.49997 14.99996 119.00003 125.50005
2 Nymphaea candida 25.49997 27.00001  73.99995  78.49997
```

Confidence intervals reflect the possible location of central value (here median). But we still need to report our centers and ranges (confidence interval is not a range!). We can use either `summary()` (**try** it yourself), or some customized output which, for example, can employ median absolute deviation:

```
> aggregate(ny[, 3:4], by=list(SPECIES=ny[, 1]), function(.x)
+ paste(median(.x, na.rm=TRUE), mad(.x, na.rm=TRUE), sep="±"))
          SPECIES    PETALS     STAMENS
1     Nuphar lutea 14±1.4826 119±19.2738
2 Nymphaea candida 26±2.9652  77±10.3782
```

Now we can give the answer like "if there are 12–16 petals and 100–120 stamens, this is likely a yellow water lily, otherwise, if there are 23–29 petals and 66–88 stamens, this is likely a white water lily".

* * *

Answer to the question about phyllotaxis. First, we need to look on the data file, either with `url.show()`, or in the browser window and determine its structure. There are four tab-separated columns with headers, and at least the second column contains spaces. Consequently, we need to tell `read.table()` about both separator and headers and then immediately check the "anatomy" of new object:

```
> phx <- read.table(
+ "http://ashipunov.me/shipunov/open/phyllotaxis.txt",
+ h=TRUE, sep="\t")
> str(phx)
 $ FAMILY   : chr  "Anacardiaceae" ...
 $ SPECIES  : chr  "Cotinus coggygria" ...
 $ N.CIRCLES: int  2 2 2 2 2 2 2 2 2 2 ...
 $ N.LEAVES : int  4 4 5 5 5 5 5 5 5 5 ...
> nlevels(factor((phx$FAMILY)))
[1] 11
```

As you see, we have 11 families and therefore 11 proportions to create and analyze:

```
> phx10 <- sapply(1:10, Phyllotaxis)
> phx.all <- paste(phx$N.CIRCLES, phx$N.LEAVES, sep="/")
> phx.tbl <- table(phx$FAMILY, phx.all %in% phx10)
> Dotchart(sort(phx.tbl[,"FALSE"]/(rowSums(phx.tbl)))) # shipunov
```

(Here we used `Dotchart()` function which is a modified variant of classic `dotchart()` with better defaults and improved margins.)

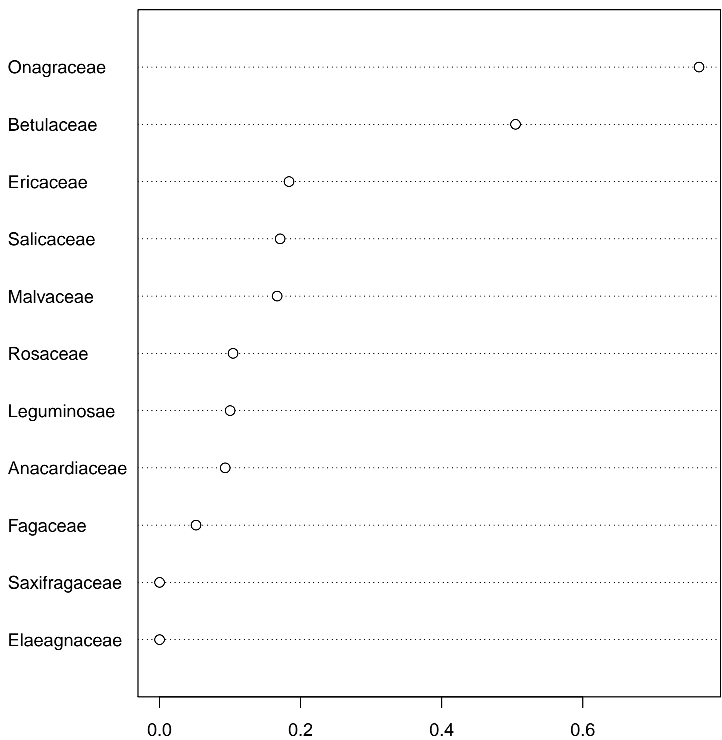


Figure 4.17: Dotchart shows proportions of non-classic formulas of phyllotaxis.

Here we created 10 first classic phyllotaxis formulas (ten is enough since higher order formulas are extremely rare), then made these formulas (classic and non-classic) from data and finally made a table from the logical expression which checks if real world formulas are present in the artificially made classic sequence. Dotchart (Fig. 4.17) is probably the best way to visualize this table. Evidently, Onagraceae (evening primrose family) has the highest proportion of `FALSE`'s. Now we need actual proportions and finally, proportion test:

```
> mean.phx.prop <- sum(phx.tbl[, 1])/sum(phx.tbl)
> prop.test(phx.tbl["Onagraceae", 1], sum(phx.tbl["Onagraceae", ]),
```

```
+ mean.phx.prop)
1-sample proportions test with continuity correction
data:  phx.tbl["Onagraceae", 1] out of sum(phx.tbl["Onagraceae", ]),
 null probability mean.phx.prop
X-squared = 227.9, df = 1, p-value < 2.2e-16
alternative hypothesis: true p is not equal to 0.2712202
95 percent confidence interval:
 0.6961111 0.8221820
sample estimates:
        p
0.7647059
```

As you see, proportion of non-classic formulas in Onagraceae (almost 77%) is statistically different from the average proportion of 27%.

* * *

Answer to the exit poll question from the “Foreword”. Here is the way to calculate how many people we might want to ask to be sure that our sample 48% and 52% are “real” (represent the population):

```
> power.prop.test(p1=0.48, p2=0.52, power=0.8)
     Two-sample comparison of proportions power calculation
              n = 2451.596
             p1 = 0.48
             p2 = 0.52
      sig.level = 0.05
          power = 0.8
    alternative = two.sided
 NOTE: n is number in *each* group
```

We need to ask almost 5,000 people!

To calculate this, we used a kind of *power test* which are frequently used for planning experiments. We made `power=0.8` since it is the typical value of power used in social sciences. The next chapter gives definition of *power* (as a statistical term) and some more information about power test output.

Chapter 5

Two-dimensional data: differences

All methods covered in this chapter based on the idea of statistical test and side-by-side comparison. If even there are methods which seemingly accept multiple samples (like ANOVA or analysis of tables), they internally do the same: compare two pooled variations, or expected and observed frequencies.

5.1 What is a statistical test?

Suppose that we compared two sets of numbers, measurements which came from two samples. From comparison, we found that they are different. But how to know if this difference did not arise by chance? In other words, how to decide that our two samples are truly different, i.e. did not come from the one population?

These samples could be, for example, measurements of systolic blood pressure. If we study the drug which potentially lowers the blood pressure, it is sensible to mix it randomly with a placebo, and then ask members of the group to report their blood pressure on the first day of trial and, saying, on the tenth day. Then the difference between two measurements will allow to decide if there is any effect:

```
> bpress <- read.table("data/bpress.txt", h=TRUE)
> head(bpress)
  PLACEBO.1 PLACEBO.10 DRUG.1 DRUG.10
1       180        170    140     120
2       140        150    120     100
3       160        155    180     140
```

```
4       170         180    160     140
5       150         150    160     160
> drug.d <- bpress$DRUG.10 - bpress$DRUG.1
> placebo.d <- bpress$PLACEBO.10 - bpress$PLACEBO.1
> mean(drug.d - placebo.d)
[1] -21
> boxplot(bpress)
```

Now, there is a promising effect, sufficient difference between blood pressure differences with drug and with placebo. This is also visible well with boxplots (**check** it yourself). How to test it? We already know how to use p-value, but it is the end of logical chain. Let us start from the beginning.

5.1.1 Statistical hypotheses

Philosophers postulated that science can never prove a theory, but only *disprove* it. If we collect 1000 facts that support a theory, it does not mean we have proved it—it is possible that the 1001st piece of evidence will disprove it.

This is why in statistical testing we commonly use two hypotheses. The one we are trying to prove is called the alternative hypothesis (H_1). The other, default one, is called the null hypothesis (H_0). The null hypothesis is a proposition of absence of something (for example, difference between two samples or relationship between two variables). We cannot prove the alternative hypothesis, but we can reject the null hypothesis and therefore switch to the alternative. If we cannot reject the null hypothesis, then we must stay with it.

5.1.2 Statistical errors

With two hypotheses, there are four possible outcomes (Table 5.1).

The first (a) and the last (d) outcomes are ideal cases: we either accept the null hypothesis which is correct for the population studied, or we reject H_0 when it is wrong.

If we have accepted the alternative hypothesis, when it is not true, we have committed a *Type I statistical error*—we have found a pattern that does not exist. This situation is often called "false positive", or "false alarm". The probability of committing a Type I error is connected with a p-value which is always reported as one of results of a statistical test. In fact, **p-value is a probability to have same or greater effect if the null hypothesis is true**.

Imagine security officer on the night duty who hears something strange. There are two choices: jump and check if this noise is an indication of something important, or continue to relax. If the noise outside is not important or even not real but officer

Sample \ Population	Null is true	Alternative is true
Accept null	(a)	(c)
Accept alternative	(b)	(d)

Table 5.1: Statistical hypotheses, including illustrations of (b) Type I and (c) Type II errors. Bigger dots are samples, all dots are population(s).

jumped, this is the Type I error. The probability to hear the suspicious noise when actually nothing happens in a p-value.

For the security officer, it is probably better to commit Type I error than to skip something important. However, in science the situation is opposite: we always stay with the H_0 when the probability of committing a Type I error is *too high*. Philosophically, this is a variant of *Occam's razor*: scientists always prefer not to introduce anything (i.e., switch to alternative) without necessity.

This approach could be found also in other spheres of our life. Read the Wikipedia article about Stanislav Petrov (https://en.wikipedia.org/wiki/Stanislav_Petrov); this is another example when false alarm is too costly.

The obvious question is what probability is "too high"? The conventional answer places that threshold (Fig. 5.1) at 0.05—the alternative hypothesis is accepted if the p-value is less than 5% (more than 95% confidence level). In medicine, with human lives as stake, the thresholds are set even more strictly, at 1% or even 0.1%. Contrary, in social sciences, it is frequent to accept 10% as a threshold. Whatever was chosen as a threshold, it must be set *a priori*, before any test. It is not allowed to modify threshold in order to find an excuse for statistical decision in mind.

Accept the null hypothesis when in fact the alternative is true is a *Type II statistical error*—failure to detect a pattern that actually exists. This is called "false negative", "carelessness". If the careless security officer did not jump when the noise outside

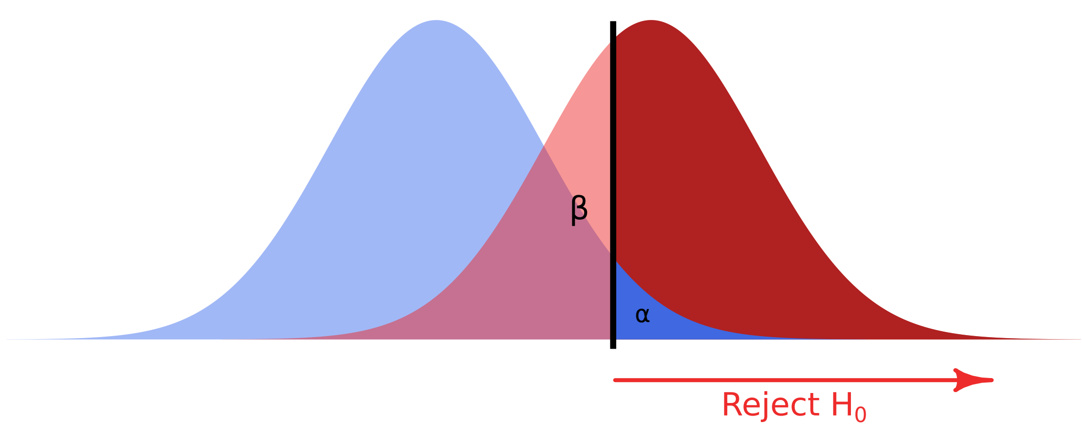


Figure 5.1: Scheme of statistical decision (for 1-tailed test). Blue "bell" is the null distribution, red is alternative, α is the probability of Type I error, β—of Type II error. Before the test, we must set the α threshold, usually to 0.05 (black line). Then we use the original data to calculate statistic (x axis values). Next, we use statistic to calculate p-value. Finally, if p-value is less then α threshold (statistic is to the right of black line), we reject the null hypothesis.

is really important, this is Type II error. Probability of committing Type II error is related with *power* of the statistical test. The smaller is this probability, the more powerful is the test.

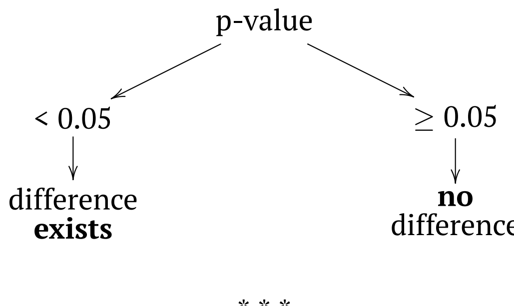
5.2 Is there a difference? Comparing two samples

5.2.1 Two sample tests

Studying two samples, we use the same approach with two hypotheses. The typical *null hypothesis* is "there is no difference between these two samples"—in other words, they are both drawn from the same population. The *alternative hypothesis* is "there is a difference between these two samples". There are many other ways to say that:

- **Null**: difference equal to 0 $\approx$ samples similar $\approx$ samples related $\approx$ samples came from the same population
- **Alternative**: difference not equal to 0 $\approx$ samples different $\approx$ samples non-related $\approx$ samples came from different populations

And, in terms of p-value:



* * *

If the data are "parametric", then a parametric *t-test* is required. If the variables that we want to compare were obtained on different objects, we will use a *two-sample t-test for independent variables*, which is called with the command `t.test()`:

```
> sapply(data.frame(placebo.d, drug.d), Normality) # shipunov
placebo.d    drug.d
 "NORMAL"  "NORMAL"
...
> t.test(placebo.d, drug.d)
Welch Two Sample t-test
data:  placebo.d and drug.d
t = 2.8062, df = 6.7586, p-value = 0.02726
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
  3.175851 38.824149
sample estimates:
mean of x mean of y
        1       -20
> S.value(t.test(placebo.d, drug.d)) # shipunov
[1] 5.197132
```

There is a long output. Please note the following:

- Apart from the normality, there is a second assumption of the classic t-test, homogeneity of variances. However, `R` by default performs more complicated *Welch test* which does not require homogeneity. This is why degrees of freedom are not a whole number.
- `t` is a *t statistic* and `df` are *degrees of freedom* (related with number of cases), they both needed to calculate the p-value.

- *Confidence interval* is the second most important output of the R `t.test()`. It is recommended to supply confidence intervals and effect sizes (see below) wherever possible.

- p-value is small, therefore the probability to "raise the false alarm" when "nothing happens" is also small. Consequently, we *reject the null hypothesis* ("nothing happens", "no difference", "no effect") and therefore switch to the alternative hypothesis ("there is a difference between drugs".)

- S-value close to 5 means that there are five bits of information against null hypothesis, analogous to having 5 heads in a trial of "fairness" with 5 coin tosses.

We can use the following order from most to least important:

1. *p-value* (and S-value) are first because they helps to make decision;
2. *confidence interval*;
3. *t statistic*;
4. *degrees of freedom*.

Results of t-test did not come out of nowhere. Let us calculate the same thing manually (actually, half-manually because we will use degrees of freedom from the above test results):

```
> df <- 6.7586
> v1 <- var(placebo.d)
> v2 <- var(drug.d)
> (t.stat <- (mean(placebo.d) - mean(drug.d))/sqrt(v1/5 + v2/5))
[1] 2.806243
> (p.value <- 2*pt(-abs(t.stat), df))
[1] 0.02725892
> S.value(p.value) # shipunov
[1] 5.197128
```

(Function `pt()` calculates values of the Student distribution, the one which is used for t-test. Actually, instead of direct calculation, this and similar functions *estimate* p-values using tables and approximate formulas. This is because the direct calculation of exact probability requires *integration*, determining the square under the curve, like α from Fig. 5.1.)

Using t statistic and degrees of freedom, one can calculate p-value *without* running test. This is why to *report* result of t-test (and related Wilcoxon test, see later), most researchers list statistic, degrees of freedom (for t-test only) and p-value.

* * *

Instead of "short form" from above, you can use a "long form" when the first column of the data frame contains all data, and the second indicates groups:

```
> long <- stack(data.frame(placebo.d, drug.d))
> head(long)
  values       ind
1    -10 placebo.d
2     10 placebo.d
3     -5 placebo.d
4     10 placebo.d
5      0 placebo.d
6    -20    drug.d
> t.test(values ~ ind, data=long)
... Welch Two Sample t-test
data:  values by ind
t = -2.8062, df = 6.7586, p-value = 0.02726
...
```

(Note the *formula interface* which usually comes together with a long form.)

Long form is handy also for plotting and data manipulations (**check** the plot yourself):

```
> boxplot(values ~ ind, data=long)
> tapply(long$values, long$ind, sd)
   drug.d placebo.d
14.142136  8.944272

> aggregate(values ~ ind, range, data=long)
        ind values.1 values.2
1    drug.d      -40        0
2 placebo.d      -10       10
```

Another example of long form is the embedded `beaver2` data:

```
> tapply(beaver2$temp, beaver2$activ, Normality) # shipunov
       0         1
"NORMAL" "NORMAL"
> boxplot(temp ~ activ, data=beaver2)
> t.test(temp ~ activ, data=beaver2)
Welch Two Sample t-test
data:  temp by activ
```

```
t = -18.548, df = 80.852, p-value < 2.2e-16
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -0.8927106 -0.7197342
sample estimates:
mean in group 0 mean in group 1
       37.09684        37.90306
```

(**Check** the boxplot yourself. We assumed that temperature was measured randomly.)

Again, p-value is much less than 0.05, and we must reject the null hypothesis that temperatures are not different when beaver is active or not.

To convert long form into short, use `unstack()` function:

```
> uu <- unstack(beaver2, temp ~ activ)
> str(uu)
List of 2
 $ 0: num [1:38] 36.6 36.7 36.9 37.1 37.2 ...
 $ 1: num [1:62] 38 38 38 38.2 38.1 ...
```

(Note that result is a list because numbers of observations for active and inactive beaver are *different*. This is another plus of long form: it can handle subsets of unequal size.)

* * *

If measurements were obtained on one object, a *paired* t-test should be used. In fact, it is just one-sample t-test applied to differences between each pair of measurements. To do paired t-test in R, use the parameter `paired=TRUE`. It is not illegal to choose common t-test for paired data, but paired tests are usually more powerful:

```
> sapply(bpress, Normality) # shipunov
 PLACEBO.1 PLACEBO.10     DRUG.1    DRUG.10
  "NORMAL"   "NORMAL"   "NORMAL"   "NORMAL"
> attach(bpress)
> t.test(DRUG.1, DRUG.10, paired=TRUE)
Paired t-test
data:  DRUG.1 and DRUG.10
t = 3.1623, df = 4, p-value = 0.03411
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
  2.440219 37.559781
```

```
sample estimates:
mean of the differences
                     20

> t.test(DRUG.10 - DRUG.1, mu=0) # same results
...
t = -3.1623, df = 4, p-value = 0.03411
...
> t.test(DRUG.1, DRUG.10) # non-paired
Welch Two Sample t-test
...
t = 1.3868, df = 8, p-value = 0.2029 # much larger!
...
> detach(bpress)
```

If the case of blood pressure measurements, common t-test does not "know" which factor is responsible more for the differences: drug influence or individual variation between people. Paired t-test excludes individual variation and allows each person to serve as its own control, this is why it is more precise.

Also more precise (if the alternative hypothesis is correctly specified) are *one-tailed* tests:

```
> attach(bpress)
> t.test(PLACEBO.10, DRUG.10, alt="greater") # one-tailed test
Welch Two Sample t-test
data:  PLACEBO.10 and DRUG.10
t = 2.4509, df = 6.4729, p-value = 0.0234
alternative hypothesis: true difference in means is greater than 0
95 percent confidence interval:
 6.305348      Inf
sample estimates:
mean of x mean of y
      161       132

> t.test(PLACEBO.10, DRUG.10) # "common" test
Welch Two Sample t-test
data:  PLACEBO.10 and DRUG.10
t = 2.4509, df = 6.4729, p-value = 0.04681 # larger!
...
> detach(bpress)
```

(Here we used another alternative hypothesis: instead of guessing difference, we guessed that blood pressure in "placebo" group was *greater* on 10th day.)

Highly important note: all decisions related with the statistical tests (parametric or nonparametric, paired or non-paired, one-sided or two-sided, 0.05 or 0.01) must be done *a priori*, *before* the analysis.
The "hunting for the p-value" is illegal!

* * *

If we work with *nonparametric data*, nonparametric *Wilcoxon test* (also known as a Mann-Whitney test) is required, under the command `wilcox.test()`:

```
> tapply(ToothGrowth$len, ToothGrowth$supp, Normality) # shipunov
         OJ            VC
"NOT NORMAL"     "NORMAL"
> boxplot(len ~ supp, data=ToothGrowth, notch=TRUE)
> wilcox.test(len ~ supp, data=ToothGrowth)
Wilcoxon rank sum test with continuity correction
data:  len by supp
W = 575.5, p-value = 0.06449
alternative hypothesis: true location shift is not equal to 0
...
```

(Please run the boxplot code and note the use of *notches*. It is commonly accepted that *overlapping notches is a sign of no difference*. And yes, Wilcoxon test supports that. Notches are not default because in many cases, boxplots are visually not overlapped. By the way, we assumed here that only supp variable is present and ignored `dose` (see `?ToothGrowth` for more details).)

And yes, it is really tempting to conclude something except "stay with null" if p-value is 0.06 (Fig. 5.2) but no. This is not allowed.

Like in the t-test, paired data requires the parameter `paired=TRUE`:

```
> w0 <- ChickWeight$weight[ChickWeight$Time == 0]
> w2 <- ChickWeight$weight[ChickWeight$Time == 2]
> sapply(data.frame(w0, w2), Normality)
         w0            w2
"NOT NORMAL" "NOT NORMAL"
> boxplot(w0, w2)
> wilcox.test(w0, w2, paired=TRUE)
Wilcoxon signed rank test with continuity correction
data:  w0 and w2
```

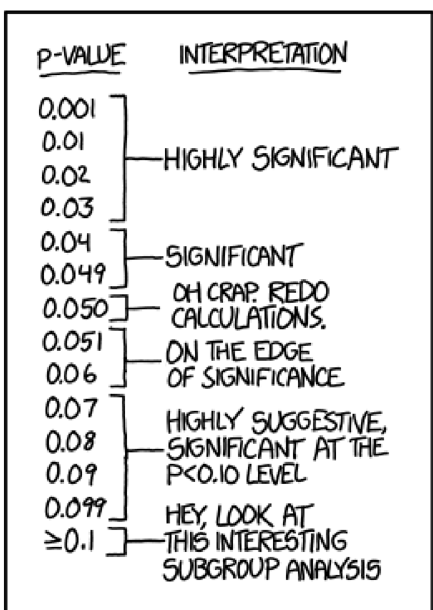


Figure 5.2: How **not** to interpret p-values (taken from XKCD, https://xkcd.com/1478/)

```
V = 8, p-value = 1.132e-09
alternative hypothesis: true location shift is not equal to 0
```

(Chicken weights are really different between hatching and second day! Please **check** the boxplot yourself.)

Nonparametric tests are generally more universal since they do not assume any particular distribution. However, they are less powerful (prone to Type II error, "carelessness"). Moreover, nonparametric tests based on ranks (like Wilcoxon test) are sensitive to the heterogeneity of variances[1]. All in all, parametric tests are preferable when data comply with their assumptions. Table 5.2 summarizes this simple procedure.

* * *

Embedded in R is the classic data set used in the original work of Student (the pseudonym of mathematician William Sealy Gossett who worked for Guinness brew-

[1] There is a workaround though, *robust rank order test*, look for the function `Rro.test()` in the `shipunov` package.

	Paired: one object, two measures	Non-paired
Normal	`t.test(..., paired=TRUE)`	`t.test(...)`
Non-normal	`wilcox.test(..., paired=TRUE)`	`wilcox.test(...)`

Table 5.2: How to choose two-sample test in R. This table should be read from the top right cell.

ery and was not allowed to use his real name for publications). This work was concerned with comparing the effects of two drugs on the duration of sleep for 10 patients.

In R these data are available under the name `sleep` (Fig. 5.3 shows corresponding boxplots). The data is in the long form: column `extra` contains the increase of the sleep times (in hours, positive or negative) while the column `group` indicates the group (type of drug).

```
> plot(extra ~ group, data=sleep)
```

(Plotting uses the "model formula": in this case, `extra ~ group`. R is smart enough to understand that `group` is the "splitting" factor and should be used to make two boxplots.)

The effect of each drug on each person is individual, but the average length by which the drug prolongs sleep can be considered a reasonable representation of the "strength" of the drug. With this assumption, we will attempt to use a two sample test to determine whether there is a significant difference between the means of the two samples corresponding to the two drugs. First, we need to determine which test to use:

```
> tapply(sleep$extra, sleep$group, Normality) # shipunov
       1        2
"NORMAL" "NORMAL"
```

(Data in the long form is perfectly suitable for `tapply()` which splits first argument in accordance with second, and then apply the third argument to all subsets.)

Since the data comply with the normality assumption, we can now employ parametric paired t-test:

```
> t.test(extra ~ group, data=sleep, paired=TRUE)
Paired t-test
data:  extra by group
t = -4.0621, df = 9, p-value = 0.002833
```

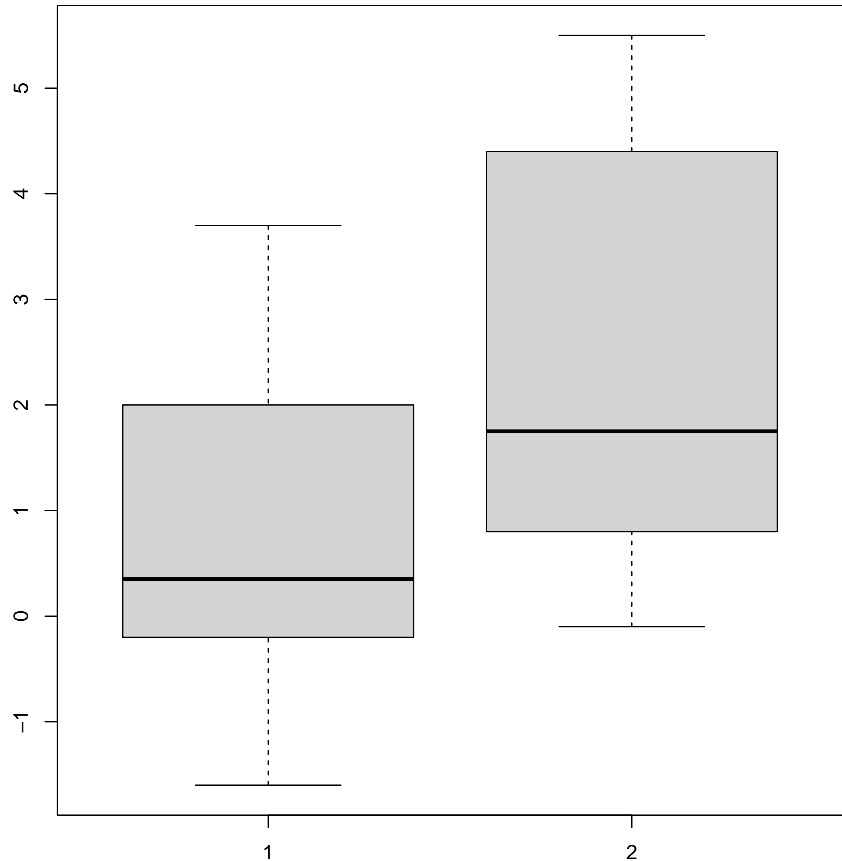


Figure 5.3: The average increase of the sleep with two drugs.

```
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -2.4598858 -0.7001142
sample estimates:
mean of the differences
                  -1.58
```

(Yes, we should reject null hypothesis about no difference.)

How about the probability of Type II errors (false negatives)? It is related with *statistical power*, and could be calculated through *power test*:

```
> power.t.test(n=10, sig.level=0.05, d=1.58)
     Two-sample t test power calculation
```

```
              n = 10
          delta = 1.58
             sd = 1
      sig.level = 0.05
          power = 0.9160669
    alternative = two.sided
```

Therefore, if we want the level of significance 0.05, sample size 10 and the effect (difference between means) 1.58, then probability of false negatives should be approximately $1 - 0.92 = 0.08$ which is really low. Altogether, this makes close to 100% our *positive predictive value* (PPV), probability of our positive result (observed difference) to be truly positive for the whole statistical population. Package `caret` is able to calculate PPV and other values related with statistical power.

It is sometimes said that t-test can handle the number of samples as low as just four. This is not absolutely correct since the power is suffering from small sample sizes, but it is true that main reason to invent t-test was to work with small samples, smaller then "rule of 30" discussed in first chapter.

* * *

Both t-test and Wilcoxon test check for differences only between measures of *central tendency* (for example, means). These homogeneous samples

```
> (aa <- 1:9)
[1] 1 2 3 4 5 6 7 8 9
> (bb <- rep(5, 9))
[1] 5 5 5 5 5 5 5 5 5
```

have the same mean but different variances (`check` it yourself), and thus the difference would **not** be detected with t-test or Wilcoxon test. Of course, tests for `scale` measures (like `var.test()`) also exist, and they *might find* the difference. You might **try** them yourself. The third homogeneous sample complements the case:

```
> (xx <- 51:59)
[1] 51 52 53 54 55 56 57 58 59
```

as differences in centers, not in ranges, will now be detected (`check` it).

There are many other two sample tests. One of these, the *sign test*, is so simple that it does not exist in R by default. The sign test first calculates differences between every pair of elements in two samples of equal size (it is a *paired* test). Then, it considers only the *positive values* and disregards others. The idea is that if samples were taken from the same distribution, then approximately *half* the differences should

be positive, and the *proportions test* will not find a significant difference between 50% and the proportion of positive differences. If the samples are different, then the proportion of positive differences should be significantly more or less than half.

> Come up with R code to carry out sign test, and test two samples that were mentioned at the beginning of the section.

* * *

The standard data set `airquality` contains information about the amount of ozone in the atmosphere around New York City from May to September 1973. The concentration of ozone is presented as a rounded mean for every day. To analyze it conservatively, we use nonparametric methods.

> Determine how close to normally distributed the monthly concentration measurements are.

Let us test the hypothesis that ozone levels in May and August were the same:

```
> wilcox.test(Ozone ~ Month, data=airquality,
+ subset = Month %in% c(5, 8), conf.int=TRUE)
Wilcoxon rank sum test with continuity correction
data:  Ozone by Month
W = 127.5, p-value = 0.0001208
alternative hypothesis: true location shift is not equal to 0
95 percent confidence interval:
 -52.99999 -14.99998
sample estimates:
difference in location
             -31.99996
Warning messages:
1: ...
```

(Since `Month` is a discrete variable as the "number" simply represents the month, the values of `Ozone` will be grouped by month. We used the parameter `subset` with the operator `%in%`, which chooses May and August, the 5th and 8th month. To obtain the confidence interval, we used the additional parameter `conf.int`. *W* is the statistic employed in the calculation of p-values. Finally, there were warning messages about ties which we ignored.)

The test rejects the null hypothesis, of equality between the distribution of ozone concentrations in May and August, fairly confidently. This is plausible because the

ozone level in the atmosphere strongly depends on solar activity, temperature and wind.

Differences between samples are well represented by box plots (Fig. 5.4):

```
> boxplot(Ozone ~ Month, data=airquality,
+ subset=Month %in% c(5, 8), notch=TRUE)
```

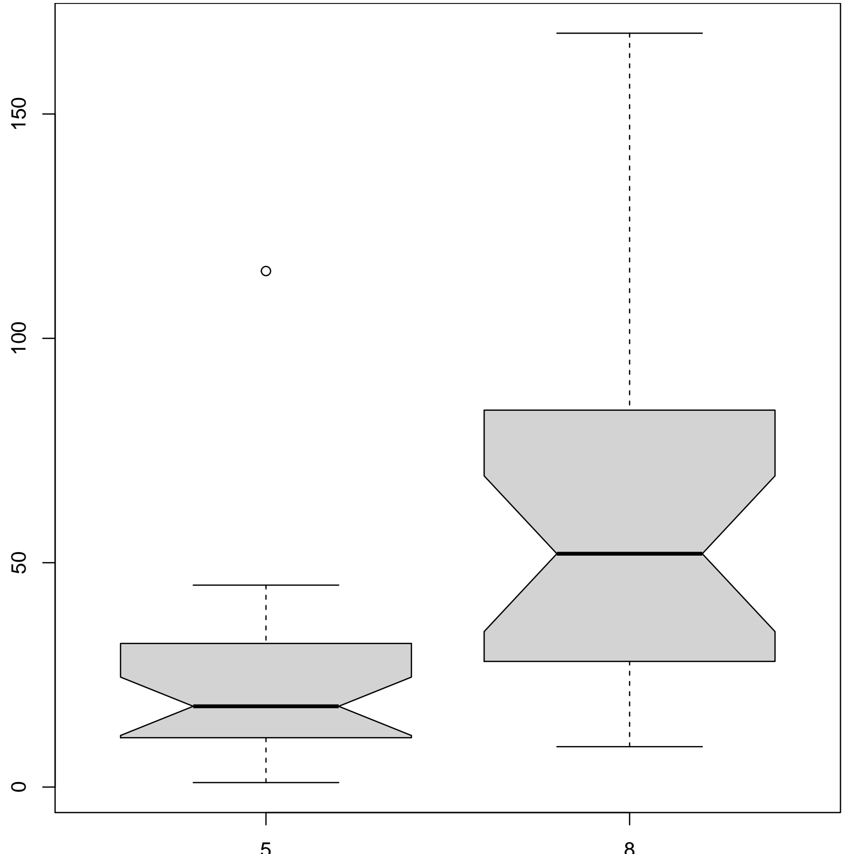


Figure 5.4: Distribution of ozone in May and June.

(Note that in the `boxplot()` command we use the same formula as the statistical model. Option `subset` is alternative way to select from data frame.)

It is conventionally considered that if the boxes overlap by more than a third of their length, the samples are not significantly different.

* * *

The last example in this section is related with the discovery of argon. At first, there was no understanding that inert gases exist in nature as they are really hard to discover chemically. But in the end of XIX century, data start to accumulate that something is wrong with nitrogen gas (N_2). Physicist Lord Rayleigh presented data which show that densities of nitrogen gas produced from ammonia and nitrogen gas produced from air are different:

```
> ar <- read.table("data/argon.txt")
> unstack(ar, form=V2 ~ V1)
      air chemical
1 2.31017  2.30143
2 2.30986  2.29890
3 2.31010  2.29816
4 2.31001  2.30182
5 2.31024  2.29869
6 2.31010  2.29940
7 2.31028  2.29849
8      NA  2.29869
```

As one might see, the difference is really small. However, it was enough for chemist Sir William Ramsay to accept it as a challenge. Both scientists performed series of advanced experiments which finally resulted in the discovery of new gas, argon. In 1904, they received two Nobel Prizes, one in physical science and one in chemistry. From the statistical point of view, most striking is how the visualization methods perform with this data:

```
> means <- tapply(ar$V2, ar$V1, mean, na.rm=TRUE)
> oldpar <- par(mfrow=1:2)
> boxplot(V2 ~ V1, data=ar)
> barplot(means)
> par(oldpar)
```

The Figure 5.5 shows as clear as possible that boxplots have great advantage over traditional barplots, especially in cases of two-sample comparison.

We recommend therefore to avoid barplots, and by all means avoid so-called "dynamite plots" (barplots with error bars on tops). Beware of dynamite!

Their most important disadvantages are (1) they hide primary data (so they are not exploratory), and in the same time, do not illustrate any statistical test (so they are not inferential); (2) they (frequently wrongly) assume that data is symmetric and parametric; (3) they use space inefficiently, have low data-to-ink ratio; (4) they cause

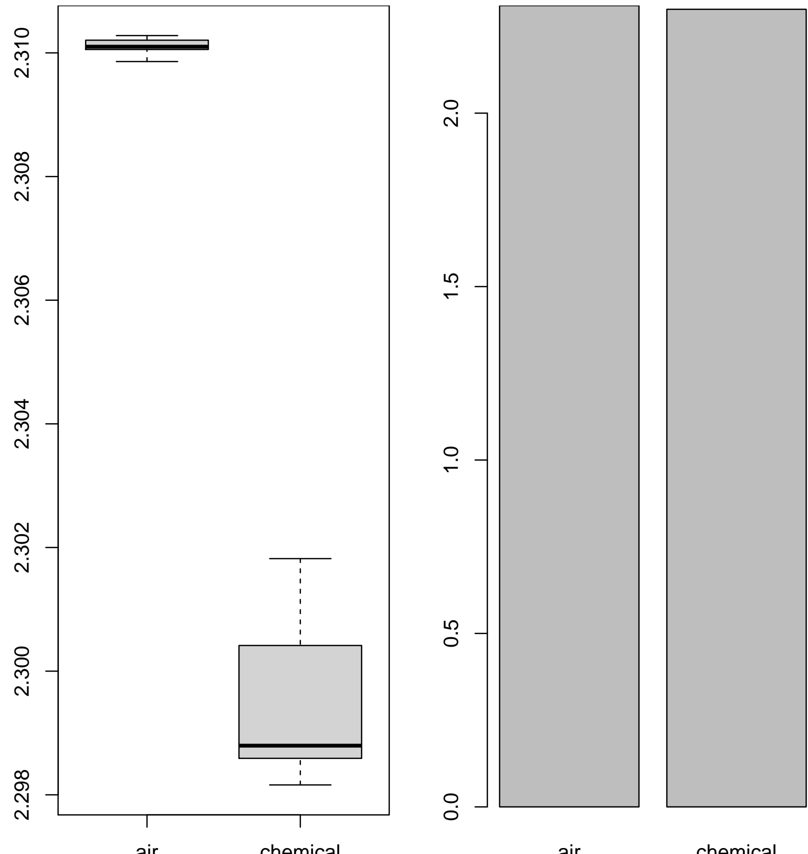


Figure 5.5: Which of these two plots would help Lord Rayleigh and Sir William Ramsay more to receive their Nobel Prizes? (The idea from Tukey, 1977.)

an optical illusion in which the reader adds some of the error bar to the height of the main bar when trying to judge the heights of the main bars; (5) the standard deviation error bar (typical there) has no direct relation even with comparing two samples (see above how t-test works), and has almost nothing to do with comparison of multiple samples (see below how ANOVA works). And, of course, they do not help Lord Rayleigh and Sir William Ramsay to receive their Nobel prizes.

> Please check the Lord Rayleigh data with the appropriate statistical test and report results.

So what to do with dynamite plots? Replace them with boxplots. The only disadvantage of boxplots is that they are harder to draw with hand which sounds funny in the

era of computers. This, by the way, explains partly why there are so many dynamite around: they are sort of legacy pre-computer times.

> A supermarket has two cashiers. To analyze their work efficiency, the length of the line at each of their registers is recorded several times a day. The data are recorded in `cashiers.txt`. Which cashier processes customers more quickly?

5.2.2 Effect sizes

Statistical tests allow to make *decisions* but do not show *how different* are samples. Consider the following examples:

```
> wilcox.test(1:10, 1:10 + 0.001, paired=TRUE)
Wilcoxon signed rank test with continuity correction
data:  1:10 and 1:10 + 0.001
V = 0, p-value = 0.005355
alternative hypothesis: true location shift is not equal to 0
...
> wilcox.test(1:10, 1:10 + 0.0001, paired=TRUE)
Wilcoxon signed rank test with continuity correction
data:  1:10 and 1:10 + 1e-04
V = 0, p-value = 0.004237
alternative hypothesis: true location shift is not equal to 0
...
```

(Here difference decreases but p-value does not grow!)

One of the beginner's mistakes is to think that p-values measure differences, but this is really wrong.

P-values are probabilities and are not supposed to measure anything. They could be used only in one, binary, yes/no way: to help with statistical decisions.

In addition, the researcher can almost always obtain a reasonably good p-value, even if effect is minuscule, like in the second example above.

To estimate the extent of differences between populations, *effect sizes* were invented. They are strongly recommended to *report together with p-values*.

* * *

Package `effsize` calculates several effect size metrics and provides interpretations of their magnitude.

Cohen's d is the parametric effect size metric which indicates difference between two means:

```
> library(effsize)
> cohen.d(extra ~ group, data=sleep)
Cohen's d
d estimate: -0.8321811 (large)
95 percent confidence interval:
       inf        sup
-1.8691015  0.2047393
```

(Note that in the last example, effect size is large with confidence interval including zero; this spoils the "large" effect.)

If the data is nonparametric, it is better to use *Cliff's Delta*:

```
> cliff.delta(1:10, 1:10+0.001)
Cliff's Delta
delta estimate: -0.1 (negligible)
95 percent confidence interval:
       inf        sup
-0.5344426  0.3762404

> cliff.delta(Ozone ~ Month, data=airquality,
+ subset = Month %in% c(5, 8))
Cliff's Delta
delta estimate: -0.2991453 (small)
95 percent confidence interval:
       inf        sup
-0.6255598  0.1163964
```

* * *

Now we have quite a few measurements to keep in memory. The simple table below emphasizes most frequently used ones:

	Center	Variation	Test	Effect
Parametric	Mean	Standard deviation	t-test	Cohen's D
Non-parametric	Median	IQR, MAD	Wilcoxon test	Cliff's Delta

Table 5.3: Most frequently used numerical tools, both for one and two samples.

* * *

There are many measures of effect sizes. In biology, useful is *coefficient of divergence* (*K*) discovered by Alexander Lyubishchev in 1959, and related with the recently introduced squared *strictly standardized mean difference* (SSSMD):

```
> K(extra ~ group, data=sleep) # shipunov
0.3462627
> summary(K(extra ~ group, data=sleep))
Lubischew's K       Effect            P
         0.35         Weak         0.34
```

(Method `summary()` shows also the magnitude and *P*, which is the *probability of misclassification*.)

Lyubishchev noted that good biological species should have $K > 18$, this means no transgression.

Coefficient of divergence is robust to *allometric changes*:

```
> summary(K(aa*3, aa*10)) # shipunov
  Lubischew's K          Effect              P
            1.5 Fairly moderate           0.19
> cliff.delta(aa*3, aa*10)
Cliff's Delta
delta estimate: -0.7777778 (large)
95 percent confidence interval:
       inf        sup
-0.9493811 -0.2486473
```

There is also MAD-based *nonparametric* variant of *K*:

```
> summary(K(1:10, 1:10+0.001, mad=TRUE)) # shipunov
 Lubischew's K         Effect              P
             0 Extremely weak            0.5
> (dd <- K(Ozone ~ Month,
+ data=airquality[airquality$Month %in% c(5, 8), ],
+ mad=TRUE)) # shipunov
0.6141992
> summary(dd)
Lubischew's K       Effect             P
         0.61   Fairly weak         0.29
```

> In the data file `grades.txt` are the grades of a particular group of students for the first exam (in the column labeled `A1`) and the second exam (`A2`), as well as the grades of a second group of students for the first exam (`B1`). Do the A class grades for the first and second exams differ? Which class did better in the first exam, A or B? Report significances, confidence intervals and effect sizes.

> In the open repository, file `aegopodium.txt` contains measurements of leaves of sun and shade *Aegopodium podagraria* (ground elder) plants. Please find the character which is most different between sun and shade and apply the appropriate statistical test to find if this difference is significant. Report also the confidence interval and effect size.

5.3 If there are more than two samples: ANOVA

5.3.1 One way

What if we need to know if there are differences between *three* samples? The first idea might be to make the series of statistical tests between each pair of the sample. In case of three samples, we will need three t-tests or Wilcoxon tests. What is unfortunate is that number of required tests will grow dramatically with the number of samples. For example, to compare six samples we will need to perform 15 tests!

Even more serious problem is that all tests are based on the idea of probability. Consequently, the chance to make of the Type I error (false alarm) will grow every time we perform more simultaneous tests on the same sample.

Let us imagine two samples which belong to one population. Therefore, when we test them, p-value should help us to stay with null. However, with many similar two-sample tests, chance is growing that at least in one of them, p-value, just *by random*, will dictate us to switch to alternative (Fig. E.2).

Or, in other words, we typically accept that probability to reject null by mistake in our case should be very small. However, with every new test, this probability will grow and if there are many tests, it will eventually become dangerously high. This is called the *problem of multiple comparisons*. To avoid the problem, we need either to correct p-values (e.g., magnify so they will be less prone to these random errors), or to invent the new method of the analysis.

One of most striking examples of multiple comparisons is a "dead salmon case". In 2009, group of researches published results of MRI testing which *detected the brain*

activity in a dead fish! But that was simply because they purposely *did not account for multiple comparisons*[2].

* * *

The special technique, ANalysis Of VAriance (ANOVA) was invented to avoid multiple comparisons in case of more than two samples.

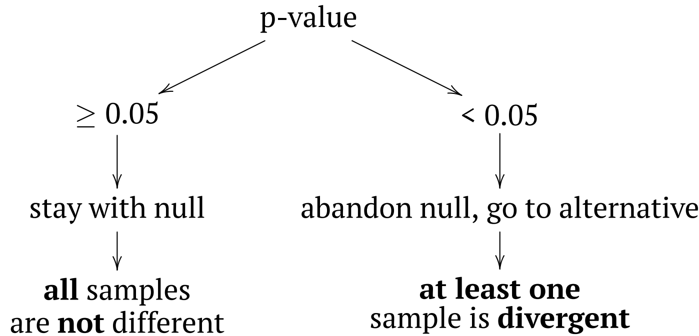
In R formula language, ANOVA might be described as

`response ~ factor`

where `response` is the *measurement* variable. Note that the only difference from two-sample case above is that `factor` in ANOVA has *more then two* levels.

The null hypothesis here is that *all samples* belong to the same population ("are not different"), and the alternative hypothesis is that *at least one sample* is divergent, does not belong to the same population ("samples are different").

In terms of p-values:



The idea of ANOVA is to *compare variances*: (1) *grand* variance within whole dataset, (2) total variance *within samples* (subsets in long form or columns in short form) and (3) variance *between samples* (columns, subsets). Figure 5.6 explains it on example of multiple apple samples mixed with divergent tomato sample.

Note that the same idea (compare variances) underlies t-test so you might think of two samples techniques as of reduced multiple samples (ANOVA and alike).

If any sample came from different population, then variance between samples should be at least comparable with (or larger then) variation within samples; in other words,

[2] Bennett C.M., Wolford G.L., Miller M.B. 2009. The principled control of false positives in neuroimaging. Social cognitive and affective neuroscience 4(4): 417–422, https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2799957/

F-value (or F-ratio) should be ≥ 1. To check that inferentially, *F-test* is applied. If p-value is small enough, then at least one sample (subset, column) is divergent.

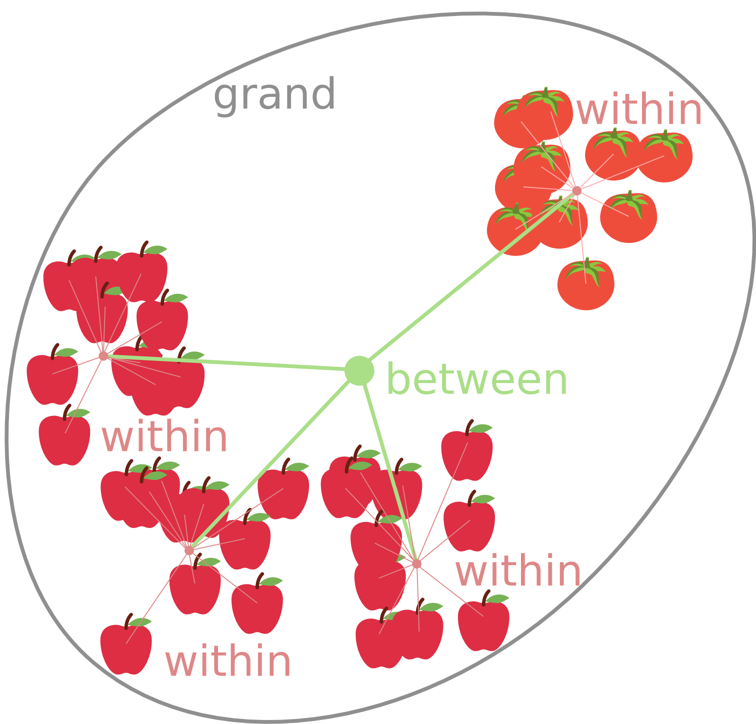


Figure 5.6: The basic idea of ANOVA: compare grand, within and between variances.

ANOVA does not reveal *which* sample is different. This is because variances in ANOVA are pooled. But what if we still need to know that? Then we should apply *post hoc* tests. In is not required to run them *after* ANOVA; what is required is to perform them carefully and always apply *p-value adjustment* for multiple comparisons. This adjustment typically *increases* p-value to avoid accumulation from multiple tests. ANOVA and *post hoc* tests answer *different* research questions, therefore this is up to the researcher to decide which and when to perform.

* * *

ANOVA is a *parametric* method, and this typically goes well with its first assumption, normal distribution of residuals (deviations between observed and expected values). Typically, we check normality of the whole dataset because ANOVA uses pooled data anyway. It is also possible to check normality of residuals directly (see below). Please note that ANOVA tolerates mild deviations from normality, both in

data and in residuals. But if the data is clearly nonparametric, it is recommended to use other methods (see below).

Second assumption is homogeinety of variance (homoscedasticity), or, simpler, *similarity of variances*. This is more important and means that sub-samples were collected with similar methods.

Third assumption is more general. It was already described in the first chapter: independence of samples. “Repeated measurements ANOVA” is however possible, but requires more specific approach.

All assumptions must be checked before analysis.

* * *

The best way of data organization for the ANOVA is the *long form* explained above: two variables, one of them contains numerical data, whereas the other describes grouping (in R terminology, it is a factor). Below, we use the artificial data which describes three types of hair color, height (in cm) and weight (in kg) of 90 persons:

```
> str(hwc) # shipunov
'data.frame': 90 obs. of  3 variables:
 $ COLOR : Factor w/ 3 levels "black","blond",..: 1 1 1 ...
 $ WEIGHT: int  80 82 79 80 81 79 82 83 78 80 ...
 $ HEIGHT: int  166 170 170 171 169 171 169 170 167 166 ...
> boxplot(WEIGHT ~ COLOR, data=hwc, ylab="Weight, kg")
```

(Note that notches and other “bells and whistles” do not help here because we want to estimate joint differences; raw boxplot is probably the best choice.)

```
> sapply(hwc[sapply(hwc, is.numeric)], Normality) # shipunov
  WEIGHT   HEIGHT
"NORMAL" "NORMAL"
> tapply(hwc$WEIGHT, hwc$COLOR, var)
   black    blond    brown
8.805747 9.219540 8.896552
```

(Note the use of double `sapply()` to check normality only for measurement columns.)

It looks like both assumptions are met: variance is at least similar, and variables are normal. Now we run the core ANOVA:

```
> wc.aov <- aov(WEIGHT ~ COLOR, data=hwc)
> summary(wc.aov)
            Df Sum Sq Mean Sq F value   Pr(>F)
```

```
COLOR         2  435.1  217.54   24.24 4.29e-09 ***
Residuals    87  780.7    8.97
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

This output is slightly more complicated then output from two-sample tests, but contains similar elements (from most to least important):

1. p-value (expressed as `Pr(>F)`) and its significance;
2. statistic (`F value`);
3. degrees of freedom (`Df`)

All above numbers should go to the report. In addition, there are also:

4. variance within columns (`Sum Sq` for `Residuals`);
5. variance between columns (`Sum Sq` for `COLOR`);
6. mean variances (`Sum Sq` divided by `Df`)

(Grand variance is just a sum of variances between and within columns.)

If degrees of freedom are already known, it is easy enough to calculate F value and p-value manually, step by step:

```
> df1 <- 2
> df2 <- 87
> group.size <- 30
> (sq.between <- sum(tapply(hwc$WEIGHT, hwc$COLOR,
+ function(.x) (mean(.x) - mean(hwc$WEIGHT))^2))*group.size)
[1] 435.0889
> (mean.sq.between <- sq.between/df1)
[1] 217.5444
> (sq.within <- sum(tapply(hwc$WEIGHT, hwc$COLOR,
+ function(.x) sum((.x - mean(.x))^2))))
[1] 780.7333
> (mean.sq.within <- sq.within/df2)
[1] 8.973946
> (f.value <- mean.sq.between/mean.sq.within)
[1] 24.24178
> (p.value <- (1 - pf(f.value, df1, df2)))
[1] 4.285683e-09
```

Of course, R calculates all of that automatically, plus also takes into account all possible variants of calculations, required for data with another structure. Related to

the above example is also that to *report* ANOVA, most researches list three things: two values for degrees of freedom, F value and, of course, p-value.

All in all, this ANOVA p-value is so small that H_0 should be rejected in favor of the hypothesis that *at least one sample* is different. Remember, ANOVA does not tell *which* sample is it, but boxplots (Fig. 5.7) suggest that this might be people with black hairs.

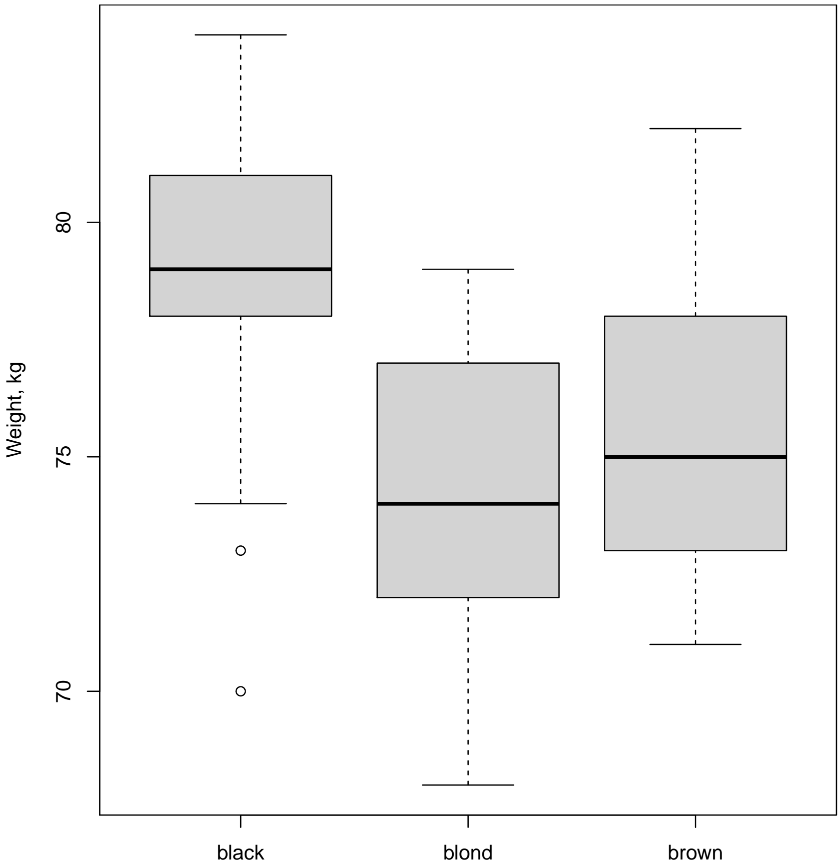


Figure 5.7: Is there a weight difference between people with different hair color? (Artificial data.)

* * *

To check the second assumption of ANOVA, that *variances should be at least similar*, *homogeneous*, it is sometimes enough to look on the variance of each group with `tapply()` as above or with `aggregate()`:

```
> aggregate(hwc[, -1], by=list(COLOR=hwc[, 1]), var)
  COLOR   WEIGHT   HEIGHT
1 black 8.805747 9.154023
2 blond 9.219540 8.837931
3 brown 8.896552 9.288506
```

But better is to *test* if variances are equal with, for example, `bartlett.test()` which has the same formula interface:

```
> bartlett.test(WEIGHT ~ COLOR, data=hwc)
Bartlett test of homogeneity of variances
data:  WEIGHT by COLOR
Bartlett's K-squared = 0.016654, df = 2, p-value = 0.9917
```

(The null hypothesis of the Bartlett test is the equality of variances.)

Alternative is nonparametric Fligner-Killeen test:

```
> fligner.test(WEIGHT ~ COLOR, data=hwc)
Fligner-Killeen test of homogeneity of variances
data:  WEIGHT by COLOR
Fligner-Killeen:med chi-squared = 1.1288, df = 2, p-value = 0.5687
```

(Null is the same as in Bartlett test.)

The first assumption of ANOVA could also be checked here directly:

```
> Normality(wc.aov$residuals)
[1] "NORMAL"
```

* * *

Effect size of ANOVA is called η^2 (eta squared). There are many ways to calculate eta squared but simplest is derived from the linear model (see in next sections). It is handy to define η^2 as a function:

```
> Eta2 <- function(aov)
+ {
+ summary.lm(aov)$r.squared
+ }
```

and then use it for results of both classic ANOVA and one-way test (see below):

```
> (ewc <- Eta2(wc.aov))
[1] 0.3578557
> Mag(ewc) # shipunov
```

```
[1] "high"
```

The second function is an interpreter for η^2 and similar effect size measures (like *r* correlation coefficient or R^2 from linear model).

If there is a need to calculate effect sizes for each pair of groups, two-sample effect size measurements like coefficient of divergence (Lyubishchev's *K*) are applicable.

* * *

One more example of classic one-way ANOVA comes from the data embedded in R (**make** boxplot yourself):

```
> Normality(chickwts$weight)
[1] "NORMAL"
> bartlett.test(weight ~ feed, data=chickwts)
Bartlett test of homogeneity of variances
data:  weight by feed
Bartlett's K-squared = 3.2597, df = 5, p-value = 0.66

> boxplot(weight ~ feed, data=chickwts)
> chicks.aov <- aov(weight ~ feed, data=chickwts)
> summary(chicks.aov)
            Df Sum Sq Mean Sq F value   Pr(>F)
feed         5 231129   46226   15.37 5.94e-10 ***
Residuals   65 195556    3009
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
> Eta2(chicks.aov)
[1] 0.5416855
> Mag(Eta2(chicks.aov)) # shipunov
[1] "very high"
```

Consequently, there is a very high difference between weights of chickens on different diets.

* * *

If there is a goal to find the divergent sample(s) statistically, one can use *post hoc* pairwise t-test which takes into account *the problem of multiple comparisons* described above; this is just a compact way to run many t-tests and adjust resulted p-values:

```
> pairwise.t.test(hwc$WEIGHT, hwc$COLOR)
Pairwise comparisons using t tests with pooled SD
data:  hwc$WEIGHT and hwc$COLOR
      black   blond
blond 1.7e-08 -
brown 8.4e-07 0.32
P value adjustment method: holm
```

(This test uses by default the Holm method of p-value correction. Another way is Bonferroni correction explained below. All available ways of correction are accessible trough the `p.adjust()` function.)

Similar to the result of pairwise t-test (but more detailed) is the result of Tukey Honest Significant Differences test (Tukey HSD):

```
> TukeyHSD(wc.aov)
  Tukey multiple comparisons of means
    95% family-wise confidence level
Fit: aov(formula = WEIGHT ~ COLOR, data=hwc)
$COLOR
                  diff       lwr       upr     p adj
blond-black -5.0000000 -6.844335 -3.155665 0.0000000
brown-black -4.2333333 -6.077668 -2.388999 0.0000013
brown-blond  0.7666667 -1.077668  2.611001 0.5843745
```

▌ Are our groups different also by heights? If yes, are black-haired still different?

Post hoc tests output p-values so they do not measure anything. If there is a need to calculate group-to-group effect sizes, two samples effect measures (like Lyubishchev's *K*) are generally applicable. To understand pairwise effects, you might want to use the custom function `pairwise.Eff()` which is based on double `sapply()`:

```
> pairwise.Eff(hwc$WEIGHT, hwc$COLOR, eff="cohen.d") # shipunov
      black        blond         brown
black
blond 1.67 (large)
brown 1.42 (large) -0.25 (small)
```

* * *

Next example is again from the embedded data (**make** boxplot yourself):

```
> Normality(PlantGrowth$weight)
[1] "NORMAL"
> bartlett.test(weight ~ group, data=PlantGrowth)
Bartlett test of homogeneity of variances
data:  weight by group
Bartlett's K-squared = 2.8786, df = 2, p-value = 0.2371

> plants.aov <- aov(weight ~ group, data=PlantGrowth)
> summary(plants.aov)
            Df Sum Sq Mean Sq F value Pr(>F)
group        2  3.766  1.8832   4.846 0.0159 *
Residuals   27 10.492  0.3886
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

> Eta2(plants.aov)
[1] 0.2641483
> Mag(Eta2(plants.aov)) # shipunov
[1] "high"

> boxplot(weight ~ group, data=PlantGrowth)
> with(PlantGrowth, pairwise.t.test(weight, group))
Pairwise comparisons using t tests with pooled SD
data:  weight and group
     ctrl  trt1
trt1 0.194 -
trt2 0.175 0.013
P value adjustment method: holm
```

As a result, yields of plants from two treatment condition are different, but there is no difference between each of them and the control. However, the overall effect size if this experiment is high.

* * *

If variances are *not similar*, then `oneway.test()` will replace the simple (one-way) ANOVA:

```
> boxplot(WEIGHT ~ COLOR, data=hwc2) # shipunov
> sapply(hwc2[, 2:3], Normality) # shipunov
  WEIGHT   HEIGHT
"NORMAL" "NORMAL"
```

```
> tapply(hwc2$WEIGHT, hwc2$COLOR, var)
   black    blond    brown
62.27126 23.45862 31.11379 # suspicious!

> bartlett.test(WEIGHT ~ COLOR, data=hwc2)
Bartlett test of homogeneity of variances
data:  WEIGHT by COLOR
Bartlett's K-squared = 7.4914, df = 2, p-value = 0.02362 # bad!

> oneway.test(WEIGHT ~ COLOR, data=hwc2)
One-way analysis of means (not assuming equal variances)
data:  WEIGHT and COLOR
F = 7.0153, num df = 2.000, denom df = 56.171, p-value = 0.001907
> (e2 <- Eta2(aov(WEIGHT ~ COLOR, data=hwc2)))
[1] 0.1626432
> Mag(e2)
[1] "medium"

> pairwise.t.test(hwc2$WEIGHT, hwc2$COLOR) # most applicable post hoc
...
```

(Here we used another data file where variables are normal but group variances are not homogeneous. Please **make** boxplot and **check** results of *post hoc* test yourself.)

* * *

Now, what if the data is *not normal*?

The first workaround is to apply some transformation which might convert data into normal:

```
> Normality(InsectSprays$count) # shipunov
[1] "NOT NORMAL"
> Normality(sqrt(InsectSprays$count))
[1] "NORMAL"
```

However, the same transformation could influence variance:

```
> bartlett.test(sqrt(count) ~ spray, data=InsectSprays)$p.value
[1] 0.5855673 # bad for ANOVA, use one-way test
```

Frequently, it is better to use the nonparametric ANOVA replacement, *Kruskall-Wallis test*:

```
> boxplot(WEIGHT ~ COLOR, data=hwc3)
> sapply(hwc3[, 2:3], Normality) # shipunov
      WEIGHT       HEIGHT
"NOT NORMAL" "NOT NORMAL"

> kruskal.test(WEIGHT ~ COLOR, data=hwc3)
Kruskal-Wallis rank sum test
data:  WEIGHT by COLOR
Kruskal-Wallis chi-squared = 32.859, df = 2, p-value = 7.325e-08
```

(Again, another variant of the data file was used, here variables are not even normal. Please **make** boxplot yourself.)

Effect size of Kruskall-Wallis test could be calculated with ϵ^2:

```
> Epsilon2 <- function(kw, n) # n is the number of cases
+ {
+ unname(kw$statistic/((n^2 - 1)/(n+1)))
+ }

> kw <- kruskal.test(WEIGHT ~ COLOR, data=hwc3)
> Epsilon2(kw, nrow(hwc3))
[1] 0.3691985
> Mag(Epsilon2(kw, nrow(hwc3))) # shipunov
[1] "high"
```

The overall efefct size is high, it also visible well on the boxplot (**make** it yourself):

```
> boxplot(WEIGHT ~ COLOR, data=hwc3)
```

To find out *which* sample is deviated, use nonparametric *post hoc* test:

```
> pairwise.wilcox.test(hwc3$WEIGHT, hwc3$COLOR)
Pairwise comparisons using Wilcoxon rank sum test
data:  hwc3$WEIGHT and hwc3$COLOR
      black   blond
blond 1.1e-06 -
brown 1.6e-05 0.056
P value adjustment method: holm
...
```

(There are multiple warnings about ties. To get rid of them, replace the first argument with `jitter(hwc3$HEIGHT)`. However, since `jitter()` adds random noise, it is better to be careful and repeat the analysis several times if p-values are close to the threshold like here.)

Another *post hoc* test for nonparametric one-way layout is Dunn's test. There is a separate `dunn.test` package:

```
> library(dunn.test)
> dunn.test(hwc3$WEIGHT, hwc3$COLOR, method="holm", altp=TRUE)
  Kruskal-Wallis rank sum test
data: x and group
Kruskal-Wallis chi-squared = 32.8587, df = 2, p-value = 0
               Comparison of x by group
                        (Holm)
Col Mean-|
Row Mean |      black      blond
---------+----------------------
   blond |   5.537736
         |    0.0000*
         |
   brown |   4.051095  -1.486640
         |    0.0001*     0.1371
alpha = 0.05
Reject Ho if p <= alpha
```

(Output is more advanced but overall results are similar. More *post hoc* tests like Dunnett's test exist in the `multcomp` package.)

It is *not necessary to check homogeneity of variance* before Kruskall-Wallis test, but please note that it assumes that distribution shapes are not radically different between samples. If it is not the case, one of workarounds is to transform the data first, either logarithmically or with square root, or to the ranks[3], or even in the more sophisticated way.

Another option is to apply permutation tests (see Appendix). As a *post hoc* test, is is possible to use `pairwise.Rro.test()` from `shipunov` package which does not assume similarity of distributions.

* * *

Next figure (Fig. 5.8) contains the Euler diagram which summarizes what was said above about different assumptions and ways of simple ANOVA-like analyses. Please note that there are much more *post hoc* tests procedures then listed, and many of them are implemented in various R packages.

[3] Like it is implemented in the `ARTool` package (`art()` function); there also possible to use multi-way nonparametric designs.

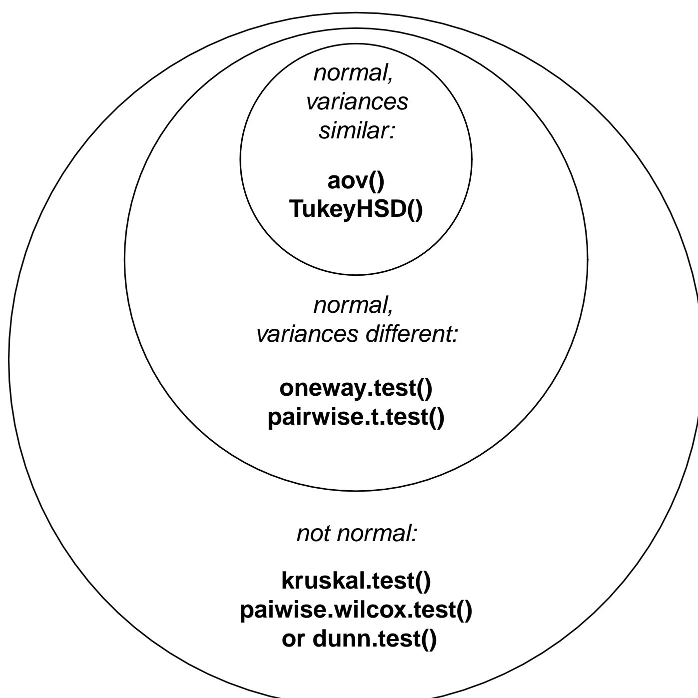


Figure 5.8: Applicability of different ANOVA-like procedures and related *post hoc.* tests. Please read it from bottom to the top.

The typical sequence of procedures related with one-way analysis is listed below:

- Check if data structure is suitable (`head()`, `str()`, `summary()`), is it long or short
- Plot (e.g., `boxplot()`, `beanplot()`)
- Normality, with plot or `Normality()`-like function
- Homogeneity of variance (homoscedasticity) (with `bartlett.test()` or `fligner.test()`)
- **Core procedure** (classic `aov()`, `oneway.test()` or `kruskal.test()`)
- Optionally, effect size (η^2 or ϵ^2 with appropriate formula)
- *Post hoc* test, for example `TukeyHSD()`, `pairwise.t.test()`, `dunn.test()` or `pairwise.wilcox.test()`

In the open repository, data file `melampyrum.txt` contains results of cow-wheat (*Melampyrum* spp.) measurements in multiple localities. Please find if there is a difference in plant height and leaf length between plants from different localities. Which localities are divergent in each case? To understand the structure of data, use companion file `melampyrum_c.txt`.

* * *

All in all, if you have two or more samples represented with measurement data, the following table will help to research *differences*:

	two samples	more then two samples
Step 1. Graphic	`boxplot()`; `beanplot()`	
Step 2. Normality *etc.*	`Normality()`; `hist()`; `qqnorm()` and `qqine()`; optionally (if required): `bartlett.test()` or `fligner.test()`	
Step 3. Test	`t.test()`; `wilcoxon.test()`	`aov()`; `oneway.test()`; `kruskal.test()`
Step 4. Effect	`cohen.d()`; `cliff.delta()`	optionally: `Eta2()`; `Epsilon2()`
Step 5. Pairwise	`NA`	`TukeyHSD()`; `pairwise.t.test()`; `dunn.test()`

Table 5.4: How to research differences between numerical samples in R.

5.3.2 More then one way

Simple, one-way ANOVA uses only one factor in formula. Frequently, however, we need to analyze results of more sophisticated experiments or observations, when data is split two or more times and possibly by different principles.

Our book is not intended to go deeper, and the following is just an introduction to the world of *design and analysis of experiment*. Some terms, however, are important to explain:

Two-way This is when data contains two *independent* factors. See, for example, `?ToothGrowth` data embedded in R. With more factors, three- and more ways layouts are possible.

Repeated measurements This is analogous to paired two-sample cases, but with three and more measurements on each subject. This type of layout might require specific approaches. See `?Orange` or `?Loblolly` data.

Unbalanced When groups have different sizes and/or some factor combinations are absent, then design is unbalanced; this sometimes complicates calculations.

Interaction If there are more than one factor, they could work together (interact) to produce response. Consequently, with two factors, analysis should include statistics for each of them plus separate statistic for interaction, three values in total. We will return to interaction later, in section about ANCOVA ("Many lines"). Here we only mention the useful way to show interactions visually, with *interaction plot* (Fig. 5.9):

```
> with(ToothGrowth, interaction.plot(supp, dose, len))
```

(It is, for example, easy to see from this interaction plot that with dose 2, type of supplement does not matter.)

Random and fixed effects Some factors are irrelevant to the research but participate in response, therefore they must be included into analysis. Other factors are planned and intentional. Respectively, they are called *random* and *fixed* effects. This difference also influences calculations.

5.4 Is there an association? Analysis of tables

5.4.1 Contingency tables

How do you compare samples of *categorical* data? These frequently are text only, there are have no numbers, like in classic "Fisher's tea drinker" example[4]. A British woman claimed to be able to distinguish whether milk or tea was added to the cup first. To test, she was given 8 cups of tea, in four of which milk was added first:

```
> tea <- read.table("data/tea.txt", h=TRUE)
> head(tea)
  GUESS REALITY
1  Milk    Milk
2  Milk    Milk
```

[4]Fisher R.A. 1971. The design of experiments. 9th ed. P. 11.

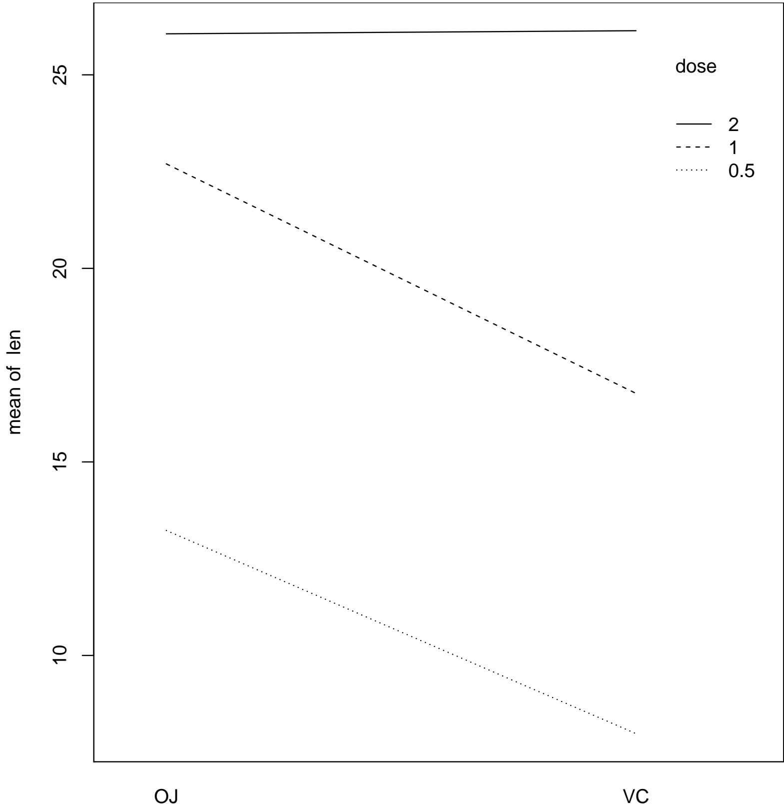


Figure 5.9: Interaction plot for `ToothGrowth` data.

```
3  Milk    Milk
4  Milk     Tea
5   Tea     Tea
6   Tea     Tea
```

The only way is to convert it to numbers, and the best way to convert is to count cases, make *contingency table*:

```
> (tea.t <- table(tea))
      REALITY
GUESS  Milk Tea
  Milk    3   1
  Tea     1   3
```

Contingency table is *not* a matrix or data frame, it is the special type of R object called "table".

In R formula language, contingency tables are described with simple formula

```
~ factor(s)
```

To use this formula approach, run `xtabs()` command:

```
> xtabs(~ GUESS + REALITY, data=tea)
      REALITY
GUESS  Milk Tea
  Milk    3   1
  Tea     1   3
```

(More than one factors have to be connected with + sign.)

* * *

If there are more than two factors, R can build a multidimensional table and print it as a series of two-dimensional tables. Please call the embedded `Titanic` data to see how 4-dimensional contingency table looks. A "flat" contingency table can be built if all the factors except one are combined into one multidimensional factor. To do this, use the command `ftable()`:

```
> ftable(Titanic)
                   Survived  No Yes
Class Sex    Age
1st   Male   Child            0   5
             Adult          118  57
      Female Child            0   1
             Adult            4 140
2nd   Male   Child            0  11
             Adult          154  14
      Female Child            0  13
             Adult           13  80
3rd   Male   Child           35  13
             Adult          387  75
      Female Child           17  14
             Adult           89  76
Crew  Male   Child            0   0
             Adult          670 192
      Female Child            0   0
             Adult            3  20
```

The function `table` can be used simply for calculation of frequencies (including missing data, if needed):

```
> d <- rep(LETTERS[1:3], 10)
> is.na(d) <- 3:4
> d
 [1] "A" "B" NA  NA  "B" "C" ...
> table(d, useNA="ifany")
d
   A    B    C <NA>
   9   10    9    2
```

The function mosaicplot() creates a graphical representation of a contingency table (Fig. 5.10):

```
> titanic <- apply(Titanic, c(1, 4), sum)
> titanic
      Survived
Class   No Yes
  1st  122 203
  2nd  167 118
  3rd  528 178
  Crew 673 212
> mosaicplot(titanic, col=c("#485392", "#204F15"),
+ main="", cex.axis=1)
```

(Function apply(..., c(1, 4), ...) does the job along first and fourth dimension and outputs 2D matrix. Command mosaicplot() is for matrices; for tables plot() calls mosaic plot automatically.)

Contingency tables are easy enough to make even from numerical data. Suppose that we need to look on association between month and comfortable temperatures in New York. If the temperatures from 64 to 86°F (from 18 to 30°C) are comfort temperatures, then:

```
> comfort <- ifelse(airquality$Temp < 64 | airquality$Temp > 86,
+ "uncomfortable", "comfortable")
```

Now we have two categorical variables, comfort and airquality$Month and can proceed to the table:

```
> comf.month <- table(comfort, airquality$Month)
> comf.month
comfort          5  6  7  8  9
  comfortable   18 25 24 21 24
  uncomfortable 13  5  7 10  6
```

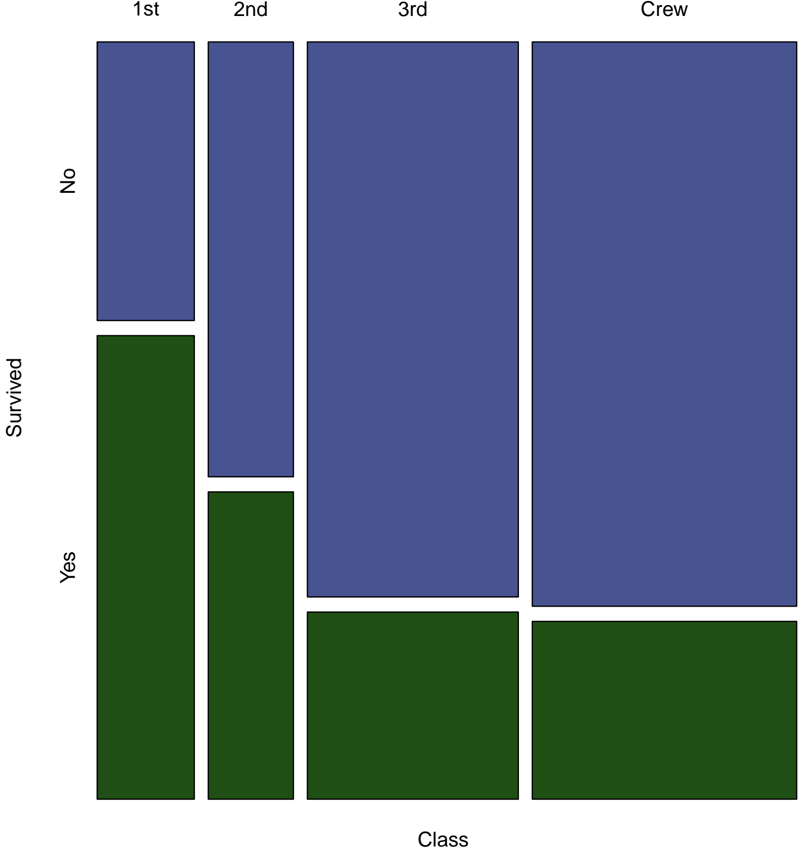


Figure 5.10: Survived on the "Titanic"

Spine plot (Fig. 5.11) is good for this kind of table, it looks like a visually advanced "hybrid" between histogram, barplot and mosaic plot:

```
> spineplot(t(comf.month))
```

(Another variant to plot these two-dimensional tables is the `dotchart()`, please **try** it yourself. Dotchart is good also for 1-dimensional tables, but sometimes you might need to use the replacement `Dotchart1()` from `shipunov` package—it keeps space for `y` axis label.)

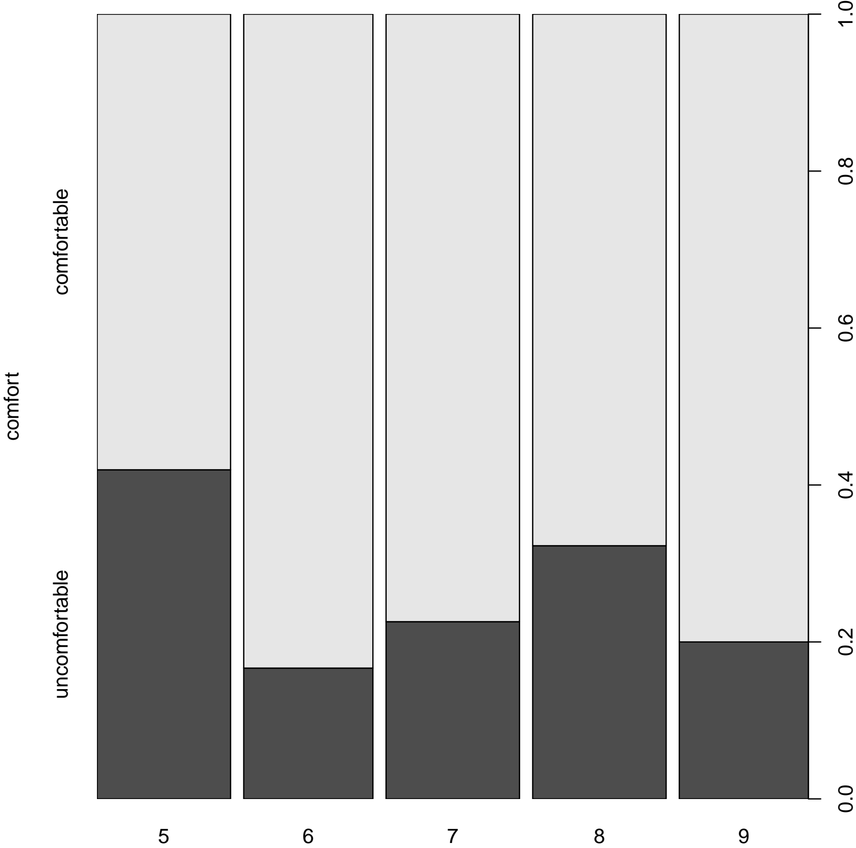


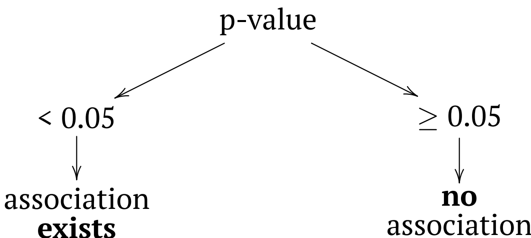
Figure 5.11: Spine plot: when is better to visit New York City.

5.4.2 Table tests

To find if there is an association in a table, one should compare two frequencies in each cell: predicted (theoretical) and observed. The serious difference is the sign of association. Null and alternative hypotheses pairs are typically:

- Null: independent distribution of factors ≈ no pattern present ≈ no association present
- Alternative: concerted distribution of factors ≈ pattern present ≈ there is an association

In terms of p-values:



Function `chisq.test()` runs a *chi-squared test*, one of two most frequently used tests for contingency tables. Two-sample chi-squared (or χ^2) test requires either contingency table or two factors of the same length (to calculate table from them first).

Now, what about the table of temperature comfort? `assocplot(comf.month)` shows some "suspicious" deviations. To check if these are statistically significant:

```
> chisq.test(comf.month)
Pearson's Chi-squared test
data:  comf.month
X-squared = 6.6499, df = 4, p-value = 0.1556
```

No, they are *not* associated. As before, there is nothing mysterious in these numbers. Everything is based on differences between expected and observed values:

```
> df <- 4
> (expected <- outer(rowSums(comf.month),
+ colSums(comf.month), "*")/sum(comf.month))
                    5         6        7        8         9
comfortable   22.69281 21.960784 22.69281 22.69281 21.960784
uncomfortable  8.30719  8.039216  8.30719  8.30719  8.039216
> (chi.squared <- sum((comf.month - expected)^2/expected))
[1] 6.649898
> (p.value <- 1 - pchisq(chi.squared, df))
[1] 0.1555872
```

(Note how expected values calculated and how they look: expected (null) are *equal proportions* between both rows and columns. June and September have 30 days each, hence slight differences in values—but not in expected proportions.)

Let us see now whether hair color and eye color from the 3-dimensional embedded `HairEyeColor` data are associated. First, we can examine associations graphically with `assocplot()` (Fig. 5.12):

```
> (HE <- margin.table(HairEyeColor, 1:2))
       Eye
Hair    Brown Blue Hazel Green
```

```
  Black     68   20    15     5
  Brown    119   84    54    29
  Red       26   17    14    14
  Blond      7   94    10    16
> assocplot(HE)
```

(Instead of `apply()` used in the previous example, we employed `margin.table()` which essentially did the same job.)

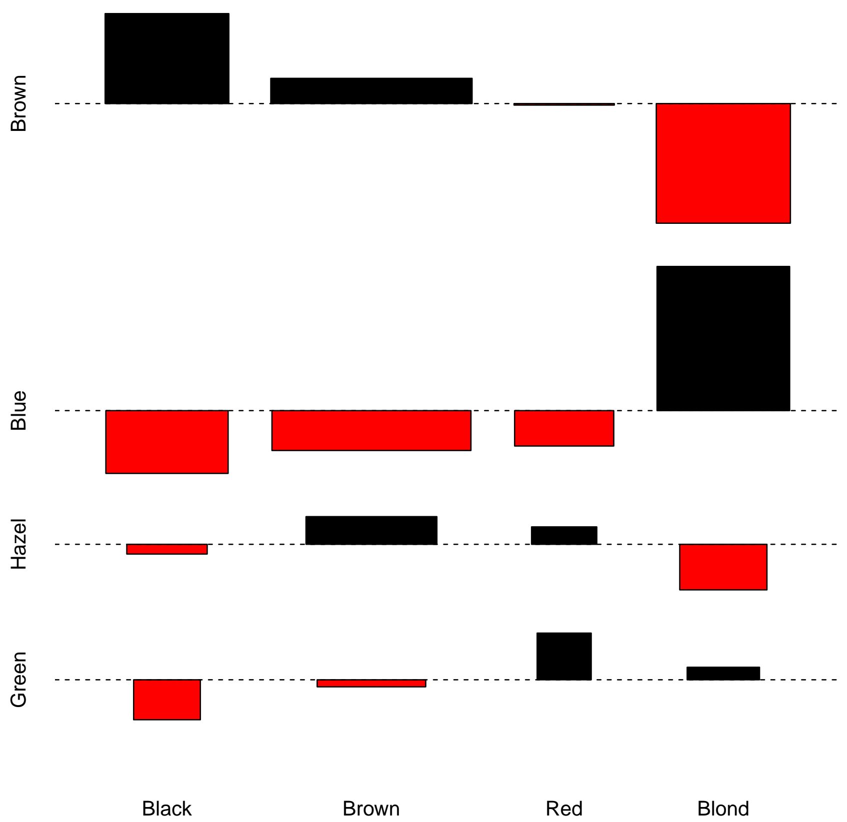


Figure 5.12: Association between hair color and eye color.

Association plot shows several things: the *height* of bars reflects the contribution of each cell into the total chi-squared, this allows, for example, to detect outliers. *Square* of rectangle corresponds with difference between observed and expected value, thus *big tall rectangles indicate more association* (to understand this better,

compare this current plot with `assocplot(comf.month)`). *Color and position* of rectangle show the sign of the difference.

Overall, it is likely that there is an association. Now we need to check this hypothesis with a test:

```
> chisq.test(HE)
Pearson's Chi-squared test
data:  HE
X-squared = 138.29, df = 9, p-value < 2.2e-16
```

The chi-squared test takes as null hypothesis "no pattern", "no association". Therefore, in our example, since we reject the null hypothesis, we find that the factors are associated.

And what about survival on the "Titanic"?

```
> chisq.test(titanic)
Pearson's Chi-squared test
data:  titanic
X-squared = 190.4, df = 3, p-value < 2.2e-16
```

Yes (as reader might remember from the famous movie), survival was associated with being in the particular class.

General chi-squared test shows only if asymmetry presents anywhere in the table. This means that if it is significant, then *at least one* group of passengers has the difference in survival. Like ANOVA, test does not show *which* one. *Post hoc*, or *pairwise* table test is able do show this:

```
> pairwise.Table2.test(titanic) # shipunov
Pairwise comparisons using Pearson's Chi-squared test
data:  titanic
     1st     2nd     3rd
2nd  4.7e-07 -       -
3rd  < 2e-16 8.3e-07 -
Crew < 2e-16 3.9e-08 0.6
P value adjustment method: BH
```

From the table of p-values, it is apparent that 3rd class and crew members were not different by survival rates. Note that *post hoc* tests apply *p-value adjustment for multiple comparisons*; practically, it means that because 7 tests were performed simultaneously, p-values were magnified with some method (here, Benjamini & Hochberg method is default).

> The file `seedlings.txt` contains results of an experiment examining germination of seeds infected with different types of fungi. In all, three fungi were tested, 20 seeds were tested for each fungus, and therefore with the controls 80 seeds were tested. Do the germination rates of the infected seeds differ?

Let us examine now the more complicated example. A large group of epidemiologists gathered for a party. The next morning, many woke up with symptoms of food poisoning. Because they were epidemiologists, they decided to remember what each of them ate at the banquet, and thus determine what was the cause of the illness. The gathered data take the following format:

```
> tox <- read.table("data/poisoning.txt", h=TRUE)
> head(tox)
  ILL CHEESE CRABDIP CRISPS BREAD CHICKEN RICE CAESAR TOMATO
1   1      1       1      1     2       1    1      1      1
2   2      1       1      1     2       1    2      2      2
3   1      2       2      1     2       1    2      1      2
4   1      1       2      1     1       1    2      1      2
...
  ICECREAM CAKE JUICE WINE COFFEE
1        1    1     1    1      1
2        1    1     1    1      2
3        1    1     2    1      2
4        1    1     2    1      2
...
```

(We used `head()` here because the table is really long.)

The first variable (`ILL`) tells whether the participant got sick or not (1 or 2 respectively); the remaining variables correspond to different foods.

A simple glance at the data will not reveal anything, as the banquet had 45 participants and 13 different foods. Therefore, statistical methods must be used. Since the data are nominal, we will use contingency tables:

```
> tox.1 <- lapply(tox[, -1], function(.x) table(tox[, 1], .x))
> tox.2 <- array(unlist(tox.1),
+ dim=c(dim(tox.1[[1]]), length(tox.1))) # or simply c(2, 2, 13)
> dimnames(tox.2) <- list(c("ill","not ill"),
+ c("took","didn't take"), names(tox.1))
```

(First, we ran `ILL` variable against every column and made a list of small contingency tables. Second, we converted list into 3-dimensional array, just like the `Titanic` data is, and also made sensible names of dimensions.)

Now our data consists of small contingency tables which are elements of array:

```
> tox.2[,,"TOMATO"]
        took didn't take
ill       24           5
not ill    6          10
```

(Note two commas which needed to tell R that we want the third dimension of the array.)

Now we need a kind of *stratified* (with every type of food) table analysis. Since every element in the tox.2 is 2×2 table, *fourfold plot* will visualize this data well (Fig. 5.13):

```
> fourfoldplot(tox.2, conf.level=0, col=c("yellow","black"))
```

(In fourfold plots, association corresponds with the difference between two pairs of diagonal sectors. Since we test multiple times, confidence rings are suppressed.)

There are some apparent differences, especially for CAESAR, BREAD and TOMATO. To check their significance, we will at first apply some table test multiple times and check out p-values:

```
> cbind(apply(tox.2, 3, function(.x) chisq.test(.x)$p.value))
                 [, 1]
CHEESE   0.8408996794
CRABDIP  0.9493138514
CRISPS   1.0000000000
BREAD    0.3498177243
CHICKEN  0.3114822175
RICE     0.5464344359
CAESAR   0.0002034102
TOMATO   0.0059125029
ICECREAM 0.5977125948
CAKE     0.8694796709
JUICE    1.0000000000
WINE     1.0000000000
COFFEE   0.7265552461
Warning messages:
1: In chisq.test(.x) : Chi-squared approximation may be incorrect
...
```

(An apply() allows us not to write the code for the test 13 times. You may omit cbind() since it used only to make output prettier. There were multiple warnings, and we will return to them soon.)

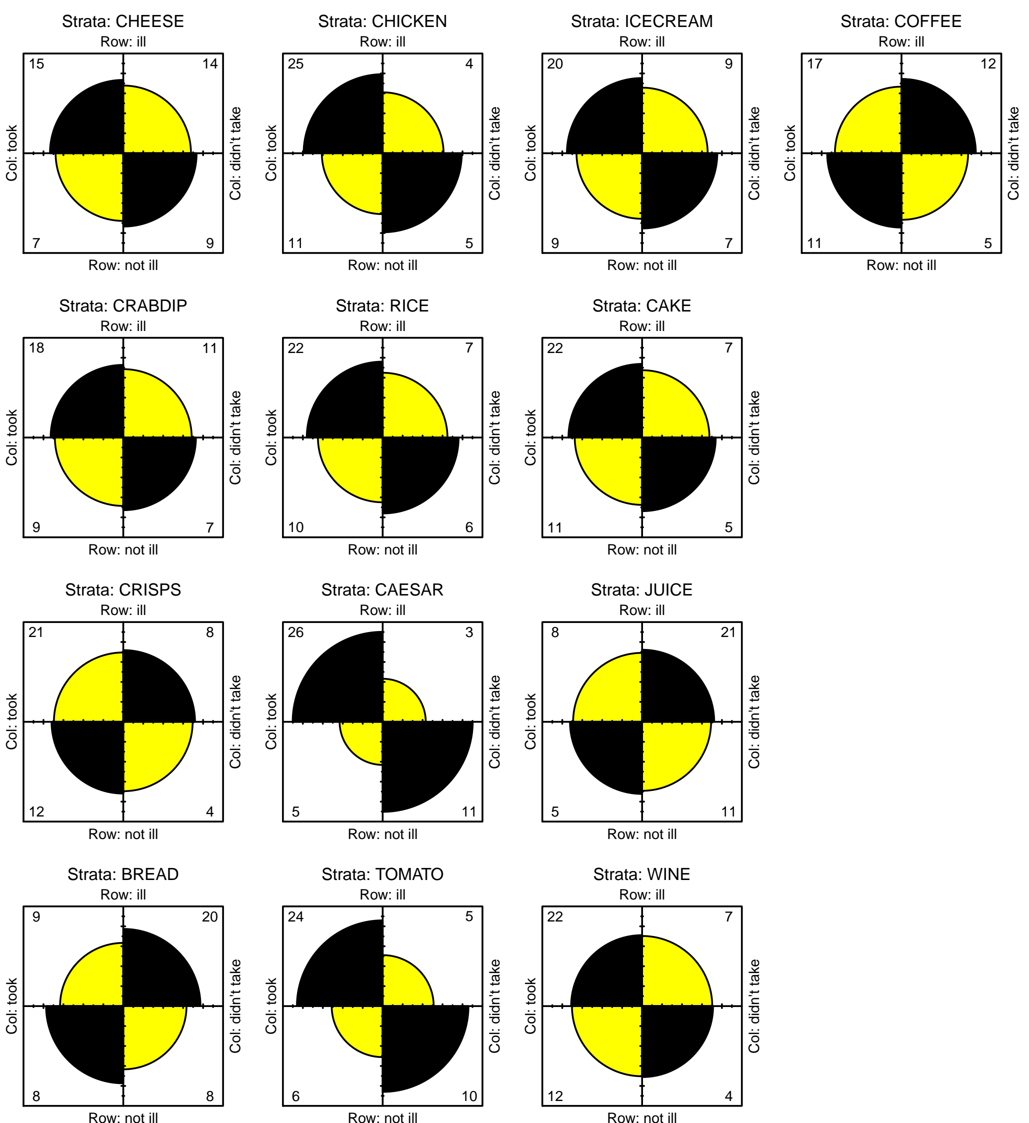


Figure 5.13: Association between food taken and illness.

The result is that two foods exhibit significant associations with illness—Caesar salad and tomatoes. The culprit is identified! Almost. After all, it is unlikely that both dishes were contaminated. Now we must try to determine what was the main cause of the food poisoning. We will return to this subject later.

> Let us discuss one more detail. Above, we applied chi-squared test simultaneously several times. To account for multiple comparisons, we must *adjust p-values*, magnify them in accordance with the particular rule, for example, with widely known Bonferroni correction rule, or with (more reliable) Benjamini and Hochberg correction rule like in the following example:
>
> ```
> > p.adjust(c(0.005, 0.05, 0.1), method="BH")
> [1] 0.015 0.075 0.100
> ```
>
> Now you know how to apply p-value corrections for multiple comparisons. Try to do this for our toxicity data. Maybe, it will help to identify the culprit?

* * *

The special case of chi-squared test is the *goodness-of-fit test*, or *G-test*. We will apply it to the famous data, results of Gregor Mendel first experiment. In this experiment, he crossed pea plants which grew out of round and angled seeds. When he counted seeds from the first generation of hybrids, he found that among 7,324 seeds, 5,474 were round and 1850 were angled. Mendel guessed that true ratio in this and six other experiments is 3:1[5]:

```
> chisq.test(c(5474, 1850), p=c(3/4, 1/4))
Chi-squared test for given probabilities
data:  c(5474, 1850)
X-squared = 0.26288, df = 1, p-value = 0.6081
```

Goodness-of-fit test uses the null that frequencies in the first argument (interpreted as one-dimensional contingency table) are *not* different from probabilities in the second argument. Therefore, 3:1 ratio is statistically supported. As you might note, it is not radically different from the proportion test explained in the previous chapter.

Without p parameter, G-test simply checks if probabilities are equal. Let us check, for example, if numbers of species in supergroups of living organisms on Earth are equal:

---

[5]Mendel G. 1866. Versuche über Pflanzen-Hybriden. Verhandlungen des naturforschenden Vereines in Brünn. Bd. 4, Abhandlungen: 12. http://biodiversitylibrary.org/page/40164750

```
> sp <- read.table("data/species.txt", sep="\t")
> species <- sp[, 2]
> names(species) <- sp[, 1]
> Dotchart(rev(sort(log10(species))),
+ xlab="Decimal logarithm of species number")
> chisq.test(species)
Chi-squared test for given probabilities
data:  species
X-squared = 4771700, df = 7, p-value < 2.2e-16
```

Naturally, numbers of species are not equal between supergroups. Some of them like bacteria (supergroup Monera) have surprisingly low number of species, others like insects (supergroup Ecdysozoa)—really large number (Fig. 5.14).

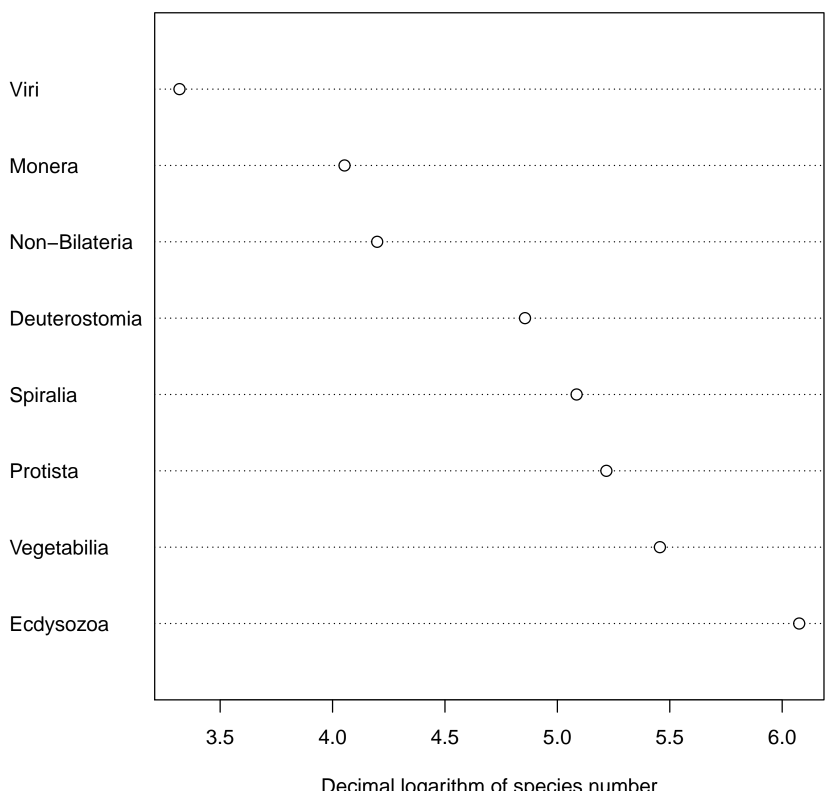


Figure 5.14: Numbers of species in supergroups of living organisms.

* * *

Chi-squared test works well when the number of cases per cell is more then 5. If there are less cases, R gives at least three workarounds.

First, instead of p-value *estimated* from the theoretical distribution, there is a way to calculate it directly, with *Fisher exact test*. Tea drinker table contains less then 5 cases per cell so it is a good example:

```
> fisher.test(tea.t)
Fisher's Exact Test for Count Data
data:  tea.t
p-value = 0.4857
alternative hypothesis: true odds ratio is not equal to 1
95 percent confidence interval:
   0.2117329 621.9337505
sample estimates:
odds ratio
  6.408309
```

Fisher test checks the null if odds ratio is just one. Although in this case, calculation gives odds ratio $(3:1)/(1:3) = 9$, there are only 8 observations, and confidence interval still includes one. Therefore, contrary to the first impression, the test does not support the idea that aforementioned woman is a good guesser.

Fourfold plot (please **check** it yourself) gives the similar result:

```
> fourfoldplot(tea.t)
```

While there is apparent difference between diagonals, confidence rings significantly intersect.

Fisher test is computationally intensive so it is not recommended to use it for large number of cases. However, if you must use it with large data, it is possible to run with `simulate.p.value=TRUE` option, like Chi-square test (see below).

The second workaround is the *Yates continuity correction* which in R is default for chi-squared test on 2×2 tables. We use now data from the original Yates (1934)[6] publication, data is taken from study of the influence of breast and artificial feeding on teeth formation:

```
> ee <- read.table("data/teeth.txt", h=TRUE)
> chisq.test(table(ee))
```

[6]Yates F. 1934. Contingency tables involving small numbers and the χ^2 test. Journal of the Royal Statistical Society. 1(2): 217–235.

```
Pearson's Chi-squared test with Yates' continuity correction
data:  table(ee)
X-squared = 1.1398, df = 1, p-value = 0.2857
Warning message:
In chisq.test(table(ee)) :
    Chi-squared approximation may be incorrect
```

(Note the warning in the end.)

Yates correction is *not* a default for the `summary.table()` function:

```
> summary(table(ee)) # No correction in summary.table()
Number of cases in table: 42
Number of factors: 2
Test for independence of all factors:
Chisq = 2.3858, df = 1, p-value = 0.1224
Chi-squared approximation may be incorrect
```

(Note different p-value: this is an effect of no correction. For all other kind of tables (e.g., non 2×2), results of `chisq.test()` and `summary.table()` should be similar.)

The third way is to *simulate* chi-squared test p-value with replication:

```
> chisq.test(table(ee), simulate.p.value=TRUE)
Pearson's Chi-squared test with simulated p-value (based on 2000
replicates)
data:  table(ee)
X-squared = 2.3858, df = NA, p-value = 0.1754
```

(Note that since this algorithm is based on random procedure, p-values might differ.)

* * *

How to calculate an *effect size for the association* of categorical variables? One of them is *odds ratio* from the Fisher test (see above). There are also several different effect size measures changing from 0 (no association) to (theoretically) 1 (which is an extremely strong association). If you do not want to use external packages, one of them, ϕ coefficient is easy to calculate from the χ-squared statistic.

```
> sqrt(chisq.test(tea.t, correct=FALSE)$statistic/sum(tea.t))
...
     0.5
```

Φ coefficient works only for two binary variables. If variables are not binary, there are *Tschuprow's T* and *Cramér's V* coefficients. Now it is better to use the external code from the `shipunov` package:

```
> (x <- margin.table(Titanic, 1:2))
      Sex
Class  Male Female
  1st   180    145
  2nd   179    106
  3rd   510    196
  Crew  862     23
> VTcoeffs(x) # shipunov
               coefficients    values comments
1                 Cramer's V 0.3987227   medium
2    Cramer's V (corrected) 0.3970098   medium
3             Tschuprow's T 0.3029637
4 Tschuprow's T (corrected) 0.3016622
```

R package vcd has function assocstats() which calculates odds ratio, ϕ, Cramér V and several other effect measures.

> In the open repository, file cochlearia.txt contains measurements of morphological characters in several populations (locations) of scurvy-grass, *Cochlearia*. One of characters, binary IS.CREEPING reflects the plant life form: creeping or upright stem. Please check if numbers of creeping plants are different between locations, provide effect sizes and p-values.

* * *

There are many table tests. For example, *test of proportions* from the previous chapter could be easily extended for two samples and therefore could be used as a table test. There is also mcnemar.test() which is used to compare proportions when they belong to same objects (*paired proportions*). You might want to check the help (and especially examples) in order to understand how they work.

> In the betula (see above) data, there are two binary characters: LOBES (position of lobes on the flower bract) and WINGS (the relative size of fruit wings). Please find if proportions of plants with 0 and 1 values of LOBES are different between location 1 and location 2.

> Are proportions of LOBES and WING values different in the whole dataset?

* * *

The typical sequence of procedures related with analysis of tables is listed below:

- Check the phenomenon of association: `table()`, `xtabs()`
- Plot it first: `mosaicplot()`, `spineplot()`, `assocplot()`
- Decide is association is statistically significant: `chisq.test()`, `fisher.test()`
- Measure how strong is an association: `VTCoeffs()`
- Optionally, if there are more then two groups per case involved, run *post hoc* pairise tests with the appropriate correction: `pairwise.Table2.test()`

* * *

To conclude this "differences" chapter, here is the Table 5.5 which will guide the reader through *most frequently* used types of analysis. Please note also the much more detailed Table 6.1 in the appendix.

	Normal	**Non-normal**	
		measurement or ranked	**nominal**
= **2 samples**	Student's test	Wilcoxon test	Chi-squared test
> **2 samples**	ANOVA or one-way + some *post hoc* test	Kruskall-Wallis + some *post hoc* test	(+ *post-hoc* test)

Table 5.5: Methods, most frequently used to analyze differences and patterns. This is the simplified variant of Table 6.1.

5.5 Answers to exercises

5.5.1 Exercises on two samples

Answer to the sign test question. It is enough to write:

```
> aa <- c(1, 2, 3, 4, 5, 6, 7, 8, 9)
> bb <- c(5, 5, 5, 5, 5, 5, 5, 5, 5)
> dif <- aa - bb
> pos.dif <- dif[dif > 0]
```

```
> prop.test(length(pos.dif), length(dif))
1-sample proportions test with continuity correction
data:  length(pos.dif) out of length(dif), null probability 0.5
X-squared = 0, df = 1, p-value = 1
alternative hypothesis: true p is not equal to 0.5
95 percent confidence interval:
 0.1534306 0.7734708
sample estimates:
        p
0.4444444
```

Here the sign test failed to find obvious differences because (like t-test and Wilcoxon test) it considers only central values.

* * *

Answer to the ozone question. To know if our data are normally distributed, we can apply the `Normality()` function:

```
> ozone.month <- airquality[, c("Ozone", "Month")]
> ozone.month.list <- unstack(ozone.month)
> sapply(ozone.month.list, Normality) # shipunov
           5            6            7            8            9
"NOT NORMAL"     "NORMAL"     "NORMAL"     "NORMAL" "NOT NORMAL"
```

(Here we applied `unstack()` function which segregated our data by months.)

* * *

Answer to the argon question. First, we need to check assumptions:

```
> sapply(unstack(ar, form=V2 ~ V1), Normality)
         air     chemical
    "NORMAL" "NOT NORMAL"
```

It is clear that in this case, nonparametric test will work better:

```
> wilcox.test(jitter(V2) ~ V1, data=ar)
Wilcoxon rank sum test
data:  jitter(V2) by V1
W = 56, p-value = 0.0003108
alternative hypothesis: true location shift is not equal to 0
```

(We used `jitter()` to break ties. However, be careful and try to check if this random noise does not influence the p-value. Here, it does not.)

And yes, boxplots (Fig. 5.5) told the truth: there is a statistical difference between two set of numbers.

* * *

Answer to the cashiers question.

```
> cashiers <- read.table("data/cashiers.txt", h=TRUE)
> head(cashiers)
  CASHIER.1 CASHIER.2
1         3        12
2        12        12
3        13         9
4         5         6
5         4         2
6        11         9
```

The data is **not** normal by definition because it is discrete (we cannot dissect people with statistics).

Now, we compare medians:

```
> (cashiers.m <- sapply(cashiers, median))
CASHIER.1 CASHIER.2
        8         9
```

It is likely that second cashier has slightly bigger lines. To test it, we will employ one-sided hypothesis (`"greater"` since we believe that second variable is probably greater):

```
> with(cashiers, wilcox.test(CASHIER.1, CASHIER.2, alt="greater"))

Wilcoxon rank sum test with continuity correction
data:  CASHIER.1 and CASHIER.2
W = 236, p-value = 0.3523
alternative hypothesis: true location shift is greater than 0
Warning message:
...
```

The difference, if even exists, is not significant. Please **calculate** effect sizes yourself.

* * *

Answer to the grades question. First, check the normality:

```
> grades <- read.table("data/grades.txt")
> classes <- split(grades$V1, grades$V2)
> sapply(classes, Normality) # shipunov
          A1           A2           B1
"NOT NORMAL" "NOT NORMAL" "NOT NORMAL"
```

(Function `split()` created three new variables in accordance with the grouping factor; it is similar to `unstack()` from previous answer but can accept groups of unequal size.)

Check data (it is also possible to plot boxplots):

```
> sapply(classes, median, na.rm=TRUE)
A1 A2 B1
 4  4  5
```

It is likely that the first class has results similar between exams but in the first exam, the second group might have better grades. Since data is not normal, we will use nonparametric methods:

```
> wilcox.test(classes$A1, classes$A2, paired=TRUE, conf.int=TRUE)

Wilcoxon signed rank test with continuity correction
data:  classes$A1 and classes$A2
V = 15.5, p-value = 0.8605
alternative hypothesis: true location shift is not equal to 0
95 percent confidence interval:
 -1.499923  1.500018
sample estimates:
(pseudo)median
             0
Warning messages:
...

> wilcox.test(classes$B1, classes$A1, alt="greater", conf.int=TRUE)

Wilcoxon rank sum test with continuity correction
data:  classes$B1 and classes$A1
W = 306, p-value = 0.02382
alternative hypothesis: true location shift is greater than 0
```

```
95 percent confidence interval:
 6.957242e-05          Inf
sample estimates:
difference in location
          6.160018e-05
Warning messages:
...
```

For the first class, we applied the paired test since grades in first and second exams belong to the same people. To see if differences between different classes exist, we used one-sided alternative hypothesis because we needed to understand not if the second class is different, but if it is *better*.

As a result, grades of the first class are not significantly different between exams, but the second class performed significantly better than first. First confidence interval includes zero (as it should be in the case of no difference), and second is not of much use.

Now effect sizes with suitable nonparametric Cliff's Delta:

```
> cliff.delta(classes$A1, classes$A2)

Cliff's Delta
delta estimate: 0.03557312 (negligible)
95 percent confidence interval:
       inf        sup
-0.2620344  0.3270022

> cliff.delta(classes$B1, classes$A1)

Cliff's Delta
delta estimate: 0.2670807 (small)
95 percent confidence interval:
        inf         sup
-0.01307644  0.50835763
```

Therefore, results of the second class are only *slightly better* which could even be negligible since confidence interval includes 0.

* * *

Answer to the question about ground elder leaves (Fig. 5.15).

First, check data, load it and check the object:

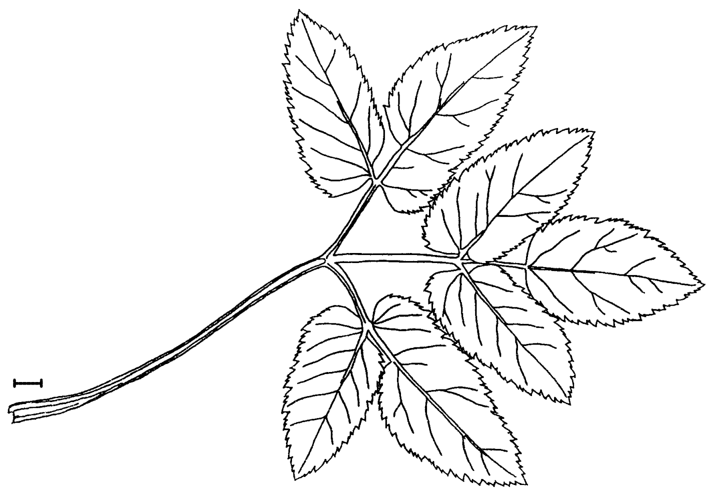


Figure 5.15: Leaf of *Aegopodium podagraria.*, ground elder. Scale bar is approximately 10 mm.

```
> aa <- read.table(
+ "http://ashipunov.me/shipunov/open/aegopodium.txt", h=TRUE)
> aa$SUN <- factor(aa$SUN, labels=c("shade","sun"))
> Str(aa)
 'data.frame': 100 obs. of  5 variables:
  1  PET.L : num  25.5 24.1 27.4 24.9 26.2 37.4 18 30.3 26.1 ...
  2  TERM.L: num  8.9 5.8 8.2 6.8 8.2 13.1 7.2 7.8 7.1 5.8 ...
  3  LEAF.L: num  34.4 29.9 35.6 31.7 34.4 50.5 25.2 38.1 33.2 ...
  4  BLADES: int  5 7 7 7 7 4 7 6 7 7 ...
  5  SUN   : Factor w/ 2 levels "shade","sun": 1 1 1 1 1 1 1 1 ...
```

(We also converted SUN variable into factor and supplied the proper labels.)

Let us check the data for the normality and for the most different character (Fig. 5.16):

```
> aggregate(aa[, -5], list(light=aa[, 5]), Normality) # shipunov
  light  PET.L      TERM.L LEAF.L     BLADES
1 shade NORMAL      NORMAL NORMAL NOT NORMAL
2   sun NORMAL NOT NORMAL NORMAL NOT NORMAL
> Linechart(aa[, 1:4], aa[, 5], xmarks=FALSE, lcolor=1,
+ se.lwd=2, mad=TRUE) # shipunov
```

TERM.L (length of the terminal leaflet, it is the rightmost one on Fig. 5.15), is likely most different between sun and shade. Since this character is normal, we will run more precise parametric test:

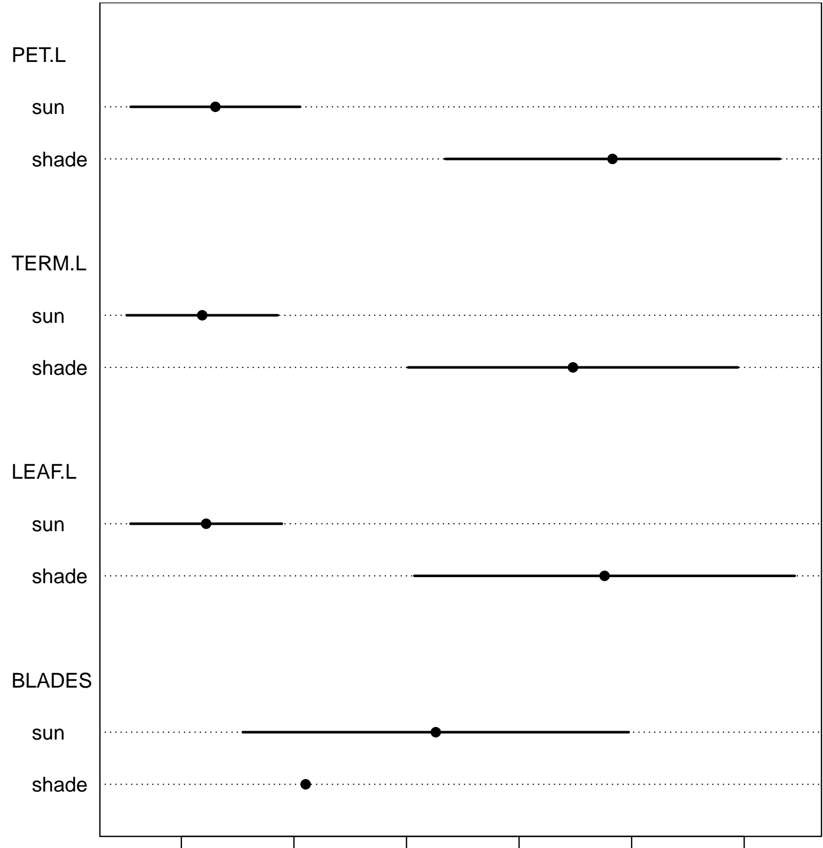


Figure 5.16: Medians with MADs in leaves data.

```
> t.test(LEAF.L ~ SUN, data=aa)
Welch Two Sample t-test
data:  LEAF.L by SUN
t = 14.846, df = 63.691, p-value < 2.2e-16
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 14.20854 18.62746
sample estimates:
mean in group shade   mean in group sun
             35.534              19.116
```

To report t-test result, one needs to provide degrees of freedom, statistic and p-value, e.g., like "in a Welch test, t statistic is 14.85 on 63.69 degrees of freedom, p-value is close to zero, thus we rejected the null hypothesis".

Effect sizes are usually concerted with p-values but provide additional useful information about the magnitude of differences:

```
> library(effsize)
> cohen.d(LEAF.L ~ SUN, data=aa)
Cohen's d
d estimate: 2.969218 (large)
95 percent confidence interval:
     inf      sup
2.384843 3.553593
> summary(K(LEAF.L ~ SUN, data=aa))
Lyubishchev's K          Effect
           4.41          Strong
```

Both Cohen's d and Lyubishchev's K (coefficient of divergence) are large.

5.5.2 Exercises on ANOVA

Answer to the height and color questions. Yes on both questions:

```
> summary(aov(HEIGHT ~ COLOR, data=hwc))
            Df Sum Sq Mean Sq F value Pr(>F)
COLOR        2 2787.3  1393.6   153.3 <2e-16 ***
Residuals   87  791.1     9.1
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

> pairwise.t.test(hwc$HEIGHT, hwc$COLOR)
Pairwise comparisons using t tests with pooled SD
data:  hwc$HEIGHT and hwc$COLOR
      black   blond
blond < 2e-16 -
brown 1.7e-10 3.3e-16
P value adjustment method: holm
```

There are significant differences between all three groups.

* * *

Answer to the question about differences between cow-wheats (Fig. 5.17) from seven locations.

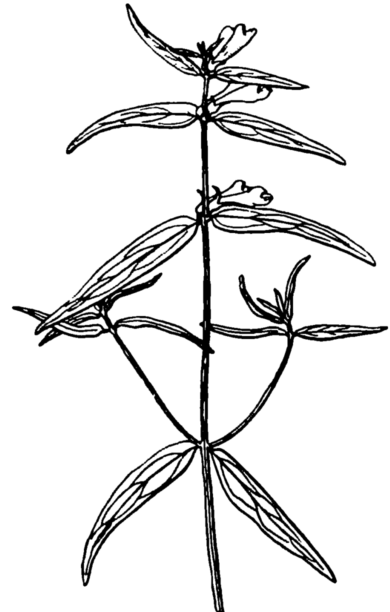


Figure 5.17: Top of the cow-wheat plant, *Melampyrum* sp. Size of fragment is approximately 10 cm.

Load the data and check its structure:

```
> mm <- read.table(
+ "http://ashipunov.me/shipunov/open/melampyrum.txt", h=TRUE)
> Str(mm)
 'data.frame': 126 obs. of  9 variables:
  1  LOC      : int  1 1 1 1 1 1 1 1 1 1 ...
  2  P.HEIGHT : int  246 235 270 260 300 250 205 190 275 215 ...
  3  NODES    * int  NA NA NA NA NA NA NA NA NA NA ...
  4  V.NODES  * int  NA NA NA NA NA NA NA NA NA NA ...
  5  LEAF.L   * int  23 27 35 20 38 46 17 22 42 26 ...
  6  LEAF.W   * int  5 3 4 4 6 5 3 3 4 3 ...
  7  LEAF.MAXW* int  3 5 5 4 6 11 5 4 5 3 ...
  8  TEETH    * int  3 2 2 6 2 4 4 5 4 3 ...
  9  TOOTH.L  * int  4 2 2 5 6 2 11 3 5 4 ...
```

Plot it first (Fig. 5.18):

```
> old.par <- par(mfrow=c(2, 1), mai=c(0.5, 0.5, 0.1, 0.1))
> boxplot(P.HEIGHT ~ LOC, data=mm, col=grey(0.8))
> boxplot(LEAF.L ~ LOC, data=mm, col=rgb(173, 204, 90, max=255))
> par(old.par)
```

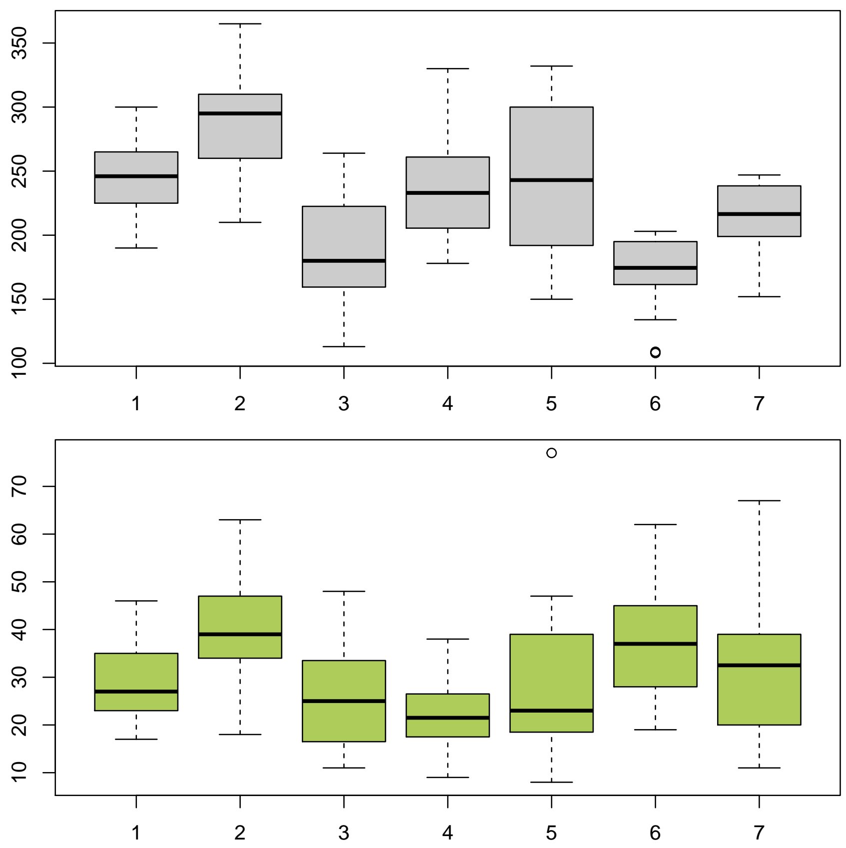


Figure 5.18: Cow-wheat stem heights (top) and leaf lengths (bottom) across seven different locations.

Check assumptions:

```
> sapply(mm[, c(2, 5)], Normality)
    P.HEIGHT       LEAF.L
    "NORMAL" "NOT NORMAL"
> bartlett.test(P.HEIGHT ~ LOC, data=mm)
Bartlett test of homogeneity of variances
```

```
data:  P.HEIGHT by LOC
Bartlett's K-squared=17.014, df=6, p-value=0.00923
```

Consequently, leaf length must be analyzed with non-parametric procedure, and plant height—with parametric which does not assume homogeneity of variance (one-way test):

```
> oneway.test(P.HEIGHT ~ LOC, data=mm)
One-way analysis of means (not assuming equal variances)
data:  P.HEIGHT and LOC
F = 18.376, num df = 6.000, denom df = 49.765, p-value = 4.087e-11

> pairwise.t.test(mm$P.HEIGHT, mm$LOC)
Pairwise comparisons using t tests with pooled SD
data:  mm$P.HEIGHT and mm$LOC
  1       2       3       4       5       6
2 0.05381 -       -       -       -       -
3 0.00511 1.1e-08 -       -       -       -
4 1.00000 0.00779 0.00511 -       -       -
5 1.00000 0.04736 0.00041 1.00000 -       -
6 4.2e-05 1.8e-11 0.62599 2.3e-05 1.1e-06 -
7 0.28824 1.9e-05 0.39986 0.39986 0.09520 0.01735
P value adjustment method: holm
```

Now the leaf length:

```
> kruskal.test(LEAF.L ~ LOC, data=mm)
Kruskal-Wallis rank sum test
data:  LEAF.L by LOC
Kruskal-Wallis chi-squared = 22.6, df = 6, p-value = 0.0009422

> pairwise.wilcox.test(mm$LEAF.L, mm$LOC)
Pairwise comparisons using Wilcoxon rank sum test
data:  mm$LEAF.L and mm$LOC
  1      2      3      4      5      6
2 0.6249 -      -      -      -      -
3 1.0000 0.1538 -      -      -      -
4 0.4999 0.0064 1.0000 -      -      -
5 1.0000 0.5451 1.0000 1.0000 -      -
6 0.6599 1.0000 0.1434 0.0028 0.5451 -
7 1.0000 1.0000 1.0000 0.5904 1.0000 1.0000

P value adjustment method: holm
```

```
There were 21 warnings (use warnings() to see them)
```

All in all, location pairs 2–4 and 4–6 are divergent statistically in both cases. This is visible also on boxplots (Fig. 5.18). There are more significant differences in plant heights, location #6, in particular, is quite outstanding.

5.5.3 Exercises on tables

Answer to the seedlings question. Load data and check its structure:

```
> pr <- read.table("data/seedlings.txt", h=TRUE)
> str(pr)
'data.frame': 80 obs. of  2 variables:
 $ CID    : int  63 63 63 63 63 63 63 63 63 63 ...
 $ GERM.14: int  1 1 1 1 1 1 1 1 1 1 ...
```

Now, what we need is to examine the table because both variables only look like numbers; in fact, they are categorical. Dotchart (Fig. 5.19) is a good way to explore 2-dimensional table:

```
> (pr.t <- table(pr))
     GERM.14
CID    0  1
  0    1 19
  63   3 17
  80  17  3
  105 10 10
> Dotchart(t(pr.t))
```

To explore possible associations visually, we employ vcd package:

```
> library(vcd)
> assoc(pr.t, shade=TRUE, gp=shading_Friendly2,
+ gp_args=list(interpolate=c(1, 1.8)))
```

Both table output and vcd association plot (Fig. 5.20) suggest some asymmetry (especially for CID80) which is a sign of possible association. Let us check it numerically, with the chi-squared test:

```
> chisq.test(pr.t, simulate=TRUE)
Pearson's Chi-squared test with simulated p-value (based on 2000
replicates)
data:  pr.t
X-squared = 33.443, df = NA, p-value = 0.0004998
```

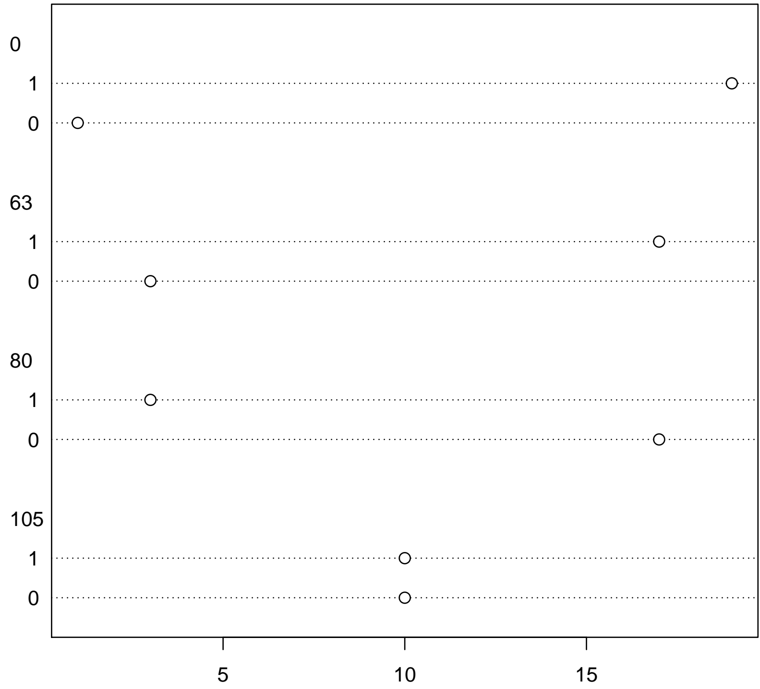


Figure 5.19: Dotchart to explore table made from seedlings data.

Yes, there is an association between fungus (or their absence) and germination. How to know differences between particular samples? Here we need a *post hoc* test:

```
> pairwise.Table2.test(table(pr), exact=TRUE)
Pairwise comparisons using Fisher's Exact Test
data:  table(pr)
    0       63      80
63  0.6050  -       -
80  2.4e-06 5.8e-05 -
105 0.0067  0.0489  0.0489
P value adjustment method: BH
```

(Exact Fisher test was used because some counts were really small.)

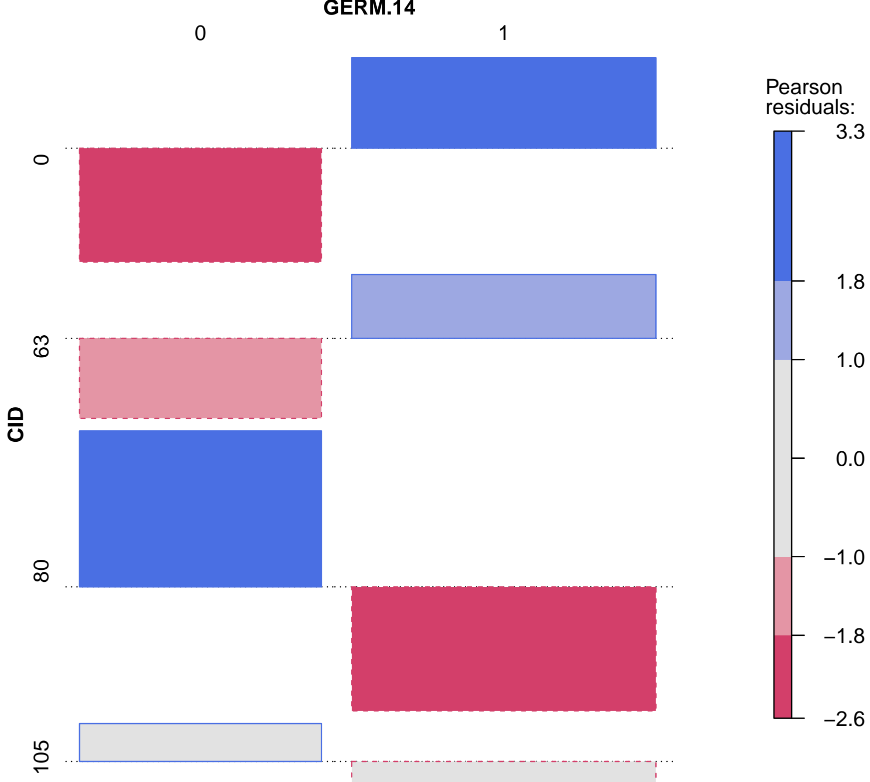


Figure 5.20: Advanced association plot of seedlings data.

It is now clear that germination patterns form two fungal infections, CID80 and CID105, are significantly different from germination in the control (CID0). Also, significant association was found in the every comparison between three infections; this means that all three germination patterns are statistically different. Finally, one fungus, CID63 produces germination pattern which is *not* statistically different from the control.

* * *

Answer to the question about multiple comparisons of toxicity. Here we will go the slightly different way. Instead of using array, we will extract p-values right from the original data, and will avoid warnings with the exact test:

```
> tox <- read.table("data/poisoning.txt", h=TRUE)
> tox.p.values <- apply(tox[, -1], 2,
```

```
+ function(.x) fisher.test(table(tox[, 1], .x))$p.value)
```

(We cannot use `pairwise.Table2.test()` from the previous answer since our comparisons have different structure. But we used exact test to avoid warnings related with small numbers of counts.)

Now we can adjust p-values:

```
> sort(round(p.adjust(tox.p.values, method="BH"), 3))
  CAESAR   TOMATO   CHEESE  CRABDIP   BREAD  CHICKEN    RICE  ICECREAM
   0.001    0.021    0.985    0.985   0.985    0.985   0.985     0.985
    CAKE   COFFEE   CRISPS    JUICE    WINE
   0.985    0.985    1.000    1.000   1.000
```

Well, now we can say that Caesar salad and tomatoes are statistically supported as culprits. But why table tests always show us two factors? This could be due to the interaction: in simple words, it means that people who took the salad, frequently took tomatoes with it.

* * *

Answer to the scurvy-grass question. Check the data file, load and check result:

```
> cc <-read.table(
+ "http://ashipunov.me/shipunov/open/cochlearia.txt", h=TRUE)
> cc$LOC <- factor(cc$LOC, labels=paste0("loc", levels(cc$LOC)))
> cc$IS.CREEPING <- factor(cc$IS.CREEPING,
+ labels=c("upright", "creeping"))
> str(cc)
'data.frame': 174 obs. of  8 variables:
 $ LOC        : Factor w/ 8 levels "loc1","loc2",..: 1 1 1 ...
 $ STEM       : int  162 147 170 432 146 207 91 166 204 172 ...
 $ IS.CREEPING: Factor w/ 2 levels "upright","creeping": 1 1 1 ...
 $ LATERAL    : int  0 0 0 0 0 0 0 0 0 0 ...
 $ PETAL.L    : num  NA NA NA NA NA NA NA 3 NA NA ...
 $ FRUIT.L    : int  6 7 9 7 7 6 7 6 7 8 ...
 $ FRUIT.W    : int  4 3 7 5 4 5 4 4 5 5 ...
 $ SEED.S     : int  2 2 1 1 1 1 1 2 1 1 ...
```

(In addition, we converted `LOC` and `IS.CREEPING` to factors and provided new level labels.)

Next step is the visual analysis (Fig. 5.21):

```
> s.cols <- colorRampPalette(c("white", "forestgreen"))(5)[3:4]
```

```
> spineplot(IS.CREEPING ~ LOC, data=cc, col=s.cols)
```

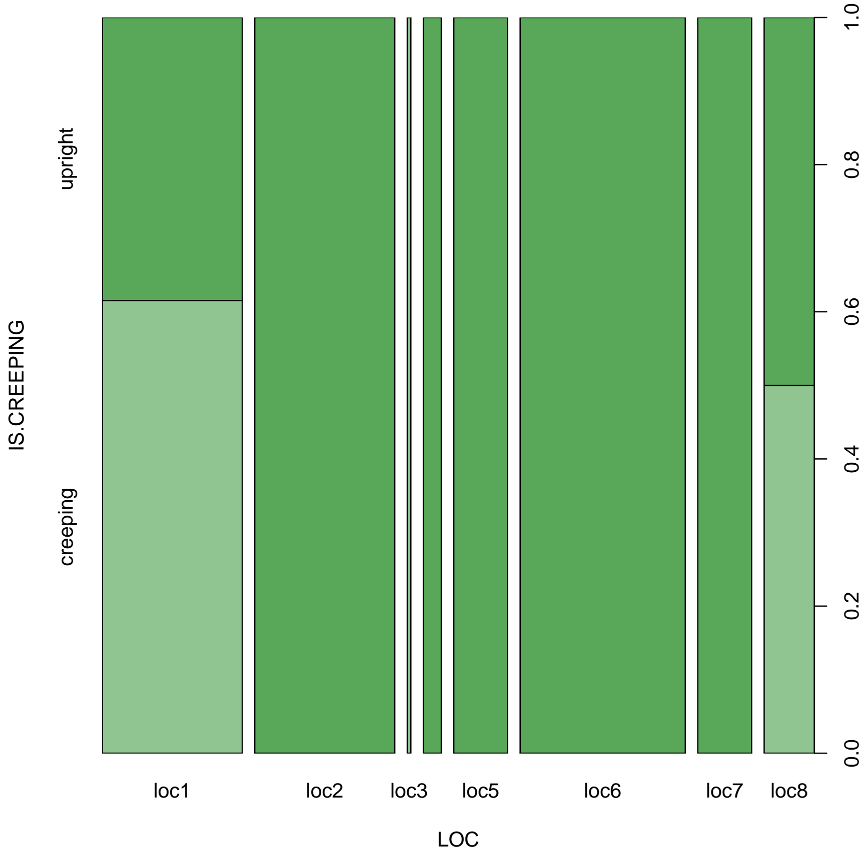


Figure 5.21: Spine plot: locality *vs.* life form of scurvy-grass.

Some locations look different. To analyze, we need contingency table:

```
> (cc.lc <- xtabs(~ LOC + IS.CREEPING, data=cc))
      IS.CREEPING
LOC    upright creeping
  loc1      15       24
  loc2      39        0
  loc3       0        1
  loc4       5        0
  loc5      15        0
  loc6      46        0
  loc7      15        0
```

```
  loc8       7        7
```

Now the test and effect size:

```
> chisq.test(cc.lc, simulate.p.value=TRUE)
Pearson's Chi-squared test with simulated p-value (based on 2000
replicates)
data:  cc.lc
X-squared = 89.177, df = NA, p-value = 0.0004998

> VTcoeffs(cc.lc)[2, ] # shipunov
            coefficients    values comments
2 Cramer's V (corrected) 0.6872265    large
```

(**Run** `pairwise.Table2.test(cc.lc)` yourself to understand differences in details.)

Yes, there is a large, statistically significant association between locality and life form of scurvy-grass.

* * *

Answer to the question about equality of proportions of `LOBES` character in two birch localities. First, we need to select these two localities (1 and 2) and count proportions there. The shortest way is to use the `table()` function:

```
> (betula.ll <- table(betula[betula$LOC < 3, c("LOC","LOBES")]))
   LOBES
LOC  0  1
  1 17  4
  2 14 16
```

Spine plot (Fig. 5.22) helps to make differences in the table even more apparent:

```
> birch.cols <- colorRampPalette(c("black", "forestgreen"))(5)[3:4]
> spineplot(betula.ll, col=birch.cols)
```

(Please also note how to create two colors intermediate between black and dark green.)

The most natural choice is `prop.test()` which is applicable directly to the `table()` output:

```
> prop.test(betula.ll)
2-sample test for equality of proportions with continuity correction
data:  betula.ll
X-squared = 4.7384, df = 1, p-value = 0.0295
```

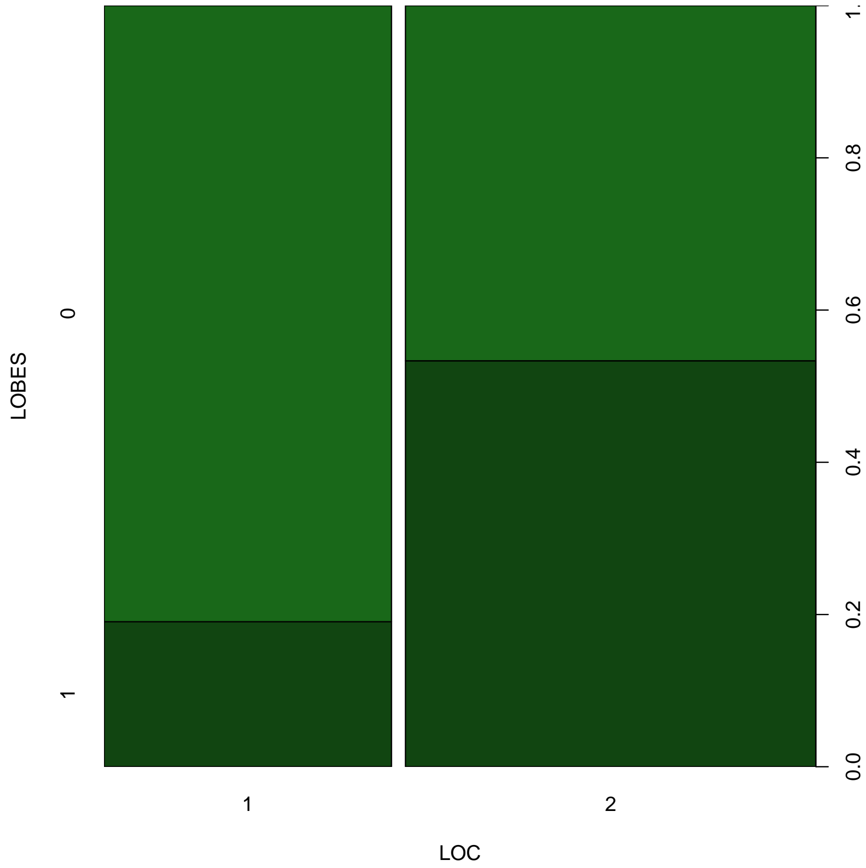


Figure 5.22: Spine plot of two birch characters.

```
alternative hypothesis: two.sided
95 percent confidence interval:
 0.05727638 0.62843791
sample estimates:
   prop 1    prop 2
0.8095238 0.4666667
```

Instead of proportion test, we can use Fisher exact:

```
> fisher.test(betula.ll)
Fisher's Exact Test for Count Data
data:  betula.ll
p-value = 0.01987
alternative hypothesis: true odds ratio is not equal to 1
```

```
95 percent confidence interval:
  1.156525 23.904424
sample estimates:
odds ratio
  4.704463
```

... or chi-squared with simulation (note that one cell has only 4 cases), or with default Yates' correction:

```
> chisq.test(betula.ll)
Pearson's Chi-squared test with Yates' continuity correction
data:  betula.ll
X-squared = 4.7384, df = 1, p-value = 0.0295
```

All in all, yes, proportions of plants with different position of lobes are different between location 1 and 2.

And what about effect size of this association?

```
> VTcoeffs(betula.ll)[2, ] # shipunov
            coefficients    values comments
2 Cramer's V (corrected) 0.3159734   medium
```

* * *

Answer to the question about proportion equality in the whole `betula` dataset. First, make table:

```
> (betula.lw <- table(betula[, c("LOBES", "WINGS")]))
     WINGS
LOBES  0  1
    0 61 69
    1 50 45
```

There is no apparent asymmetry. Since `betula.lw` is 2×2 table, we can apply fourfold plot. It shows differences not only as different sizes of sectors, but also allows to check 95% confidence interval with marginal rings (Fig. 5.23):

```
> fourfoldplot(betula.lw, col=birch.cols)
```

Also not suggestive... Finally, we need to test the association, if any. Noe that samples are *related*. This is because `LOBES` and `WINGS` were measured on the *same plants*. Therefore, instead of the chi-squared or proportion test we should run McNemar's test:

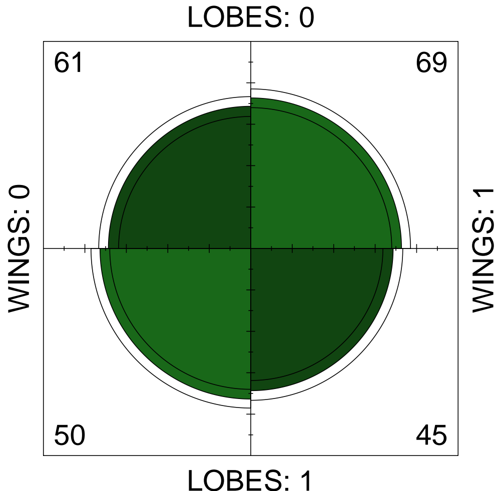


Figure 5.23: Fourfold plot of two birch characters.

```
> mcnemar.test(betula.lw)

McNemar's Chi-squared test with continuity correction

data:  betula.lw
McNemar's chi-squared = 2.7227, df = 1, p-value = 0.09893
```

We conclude that proportions of two character states in each of characters are not statistically different.

Chapter 6

Two-dimensional data: models

Here we finally come to the world of *statistical models*, the study of not just differences but *how exactly* things are related. One of the most important features of models is an ability to *predict* results. Modeling expands into thousands of varieties, there are experiment planning, Bayesian methods, maximal likelihood, and many others—but we will limit ourself with correlation, core linear regression, analysis of covariation, and introduction to logistic models.

6.1 Analysis of correlation

To start with relationships, one need first to find a *correlation*, e.g., to measure the *extent* and *sign* of relation, and to prove if this is statistically reliable.

Note that *correlation does not reflect the nature of relationship* (Fig. 6.1). If we find a significant correlation between variables, this could mean that A depends on B, B depends on A, A and B depend on each other, or A and B depend on a third variable C but have no relation to each other. A famous example is the correlation between ice cream sales and home fires. It would be strange to suggest that eating ice cream causes people to start fires, or that experiencing fires causes people to buy ice cream. In fact, both of these parameters depend on air temperature[1].

Numbers alone could be misleading, so there is a simple rule: *plot it first*.

[1] There are, however, advanced techniques with the goal to understand the difference between causation and correlation: for example, those implemented in `bnlearn` package.

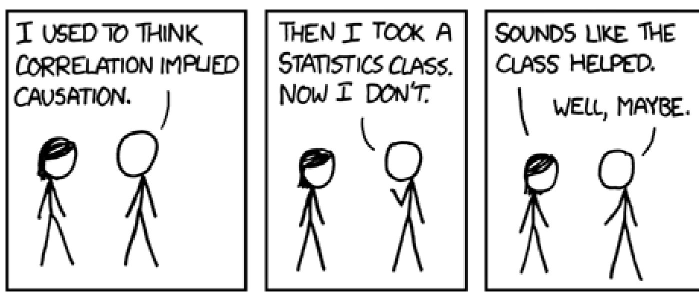


Figure 6.1: Correlation and causation (taken from XKCD, http://xkcd.com/552/).

6.1.1 Plot it first

The most striking example of relationships where numbers alone do to provide a reliable answer, is the *Anscombe's quartet*, four sets of two variables which have almost identical means and standard deviations:

```
> classic.desc <- function(.x) {c(mean=mean(.x, na.rm=TRUE),
+ var=var(.x, na.rm=TRUE))}
> sapply(anscombe, classic.desc)
     x1 x2 x3 x4       y1       y2      y3       y4
mean  9  9  9  9 7.500909 7.500909 7.50000 7.500909
var  11 11 11 11 4.127269 4.127629 4.12262 4.123249
```

(Data anscombe is embedded into R. To compact input and output, several tricks were used. Please **find** them yourself.)

Linear model coefficients (see below) are also quite similar but if we plot these data, the picture (Fig 6.2) is radically different from what is reflected in numbers:

```
> a.vars <- data.frame(i=c(1, 5), ii=c(2, 6),
+ iii=c(3, 7), iv=c(4, 8))
> oldpar <- par(mfrow=c(2, 2), mar=c(4, 4, 1, 1))
> for (i in 1:4) { plot(anscombe[a.vars[, i]], pch=19, cex=1.2);
+ abline(lm(anscombe[rev(a.vars[, i])]), lty=2) }
```

(For aesthetic purposes, we put all four plots on the same figure. Note the for operator which produces *cycle* repeating one sequence of commands four times. To know more, **check** ?"for".)

To the credit of nonparametric and/or robust numerical methods, they are not so easy to deceive:

```
> robust.desc <- function(.x) {c(median=median(.x, na.rm=TRUE),
```

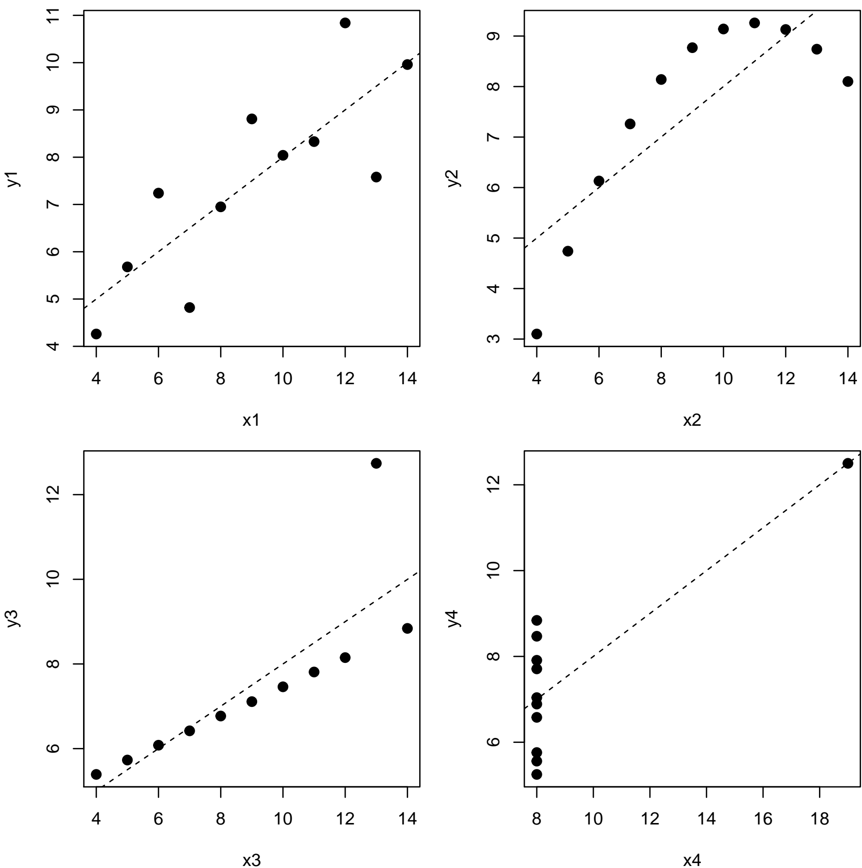


Figure 6.2: Anscombe's quartet, plotted together with lines from linear models.

```
+ IQR=IQR(.x, na.rm=TRUE), mad=mad(.x, na.rm=TRUE))}
> sapply(anscombe, robust.desc)
           x1     x2     x3 x4       y1       y2       y3       y4
median 9.0000 9.0000 9.0000  8 7.580000 8.140000 7.110000 7.040000
IQR    5.0000 5.0000 5.0000  0 2.255000 2.255000 1.730000 2.020000
mad    4.4478 4.4478 4.4478  0 1.823598 1.467774 1.527078 1.897728
```

This is correct to guess that boxplots should also show the difference. Please **try** to plot them yourself.

6.1.2 Correlation

To measure the extent and sign of linear relationship, we need to calculate *correlation coefficient*. The absolute value of the correlation coefficient varies from 0 to 1. Zero means that the values of one variable are unconnected with the values of the other variable. A correlation coefficient of 1 or −1 is an evidence of a linear relationship between two variables. A positive value of means the correlation is positive (the higher the value of one variable, the higher the value of the other), while negative values mean the correlation is negative (the higher the value of one, the lower of the other).

It is easy to calculate correlation coefficient in R:

```
> cor(5:15, 7:17)
[1] 1
> cor(5:15, c(7:16, 23))
[1] 0.9375093
```

(By default, R calculates the parametric Pearson correlation coefficient *r*.)

In the simplest case, it is given two arguments (vectors of equal length). It can also be called with one argument if using a matrix or data frame. In this case, the function cor() calculates a *correlation matrix*, composed of correlation coefficients between *all pairs* of data columns.

```
> cor(trees)
           Girth    Height    Volume
Girth  1.0000000 0.5192801 0.9671194
Height 0.5192801 1.0000000 0.5982497
Volume 0.9671194 0.5982497 1.0000000
```

As correlation is in fact the effect size of *covariance*, joint variation of two variables, to calculate it manually, one needs to know individual variances and variance of the difference between variables:

```
> with(trees, cor(Girth, Height))
[1] 0.5192801
> (v1 <- var(trees$Girth))
[1] 9.847914
> (v2 <- var(trees$Height))
[1] 40.6
> (v12 <- var(trees$Girth - trees$Height))
[1] 29.68125
> (pearson.r <- (v1 + v2 - v12)/(2*sqrt(v1)*sqrt(v2)))
[1] 0.5192801
```

Another way is to use `cov()` function which calculates covariance directly:

```
> with(trees, cov(Girth, Height)/(sd(Girth)*sd(Height)))
[1] 0.5192801
```

To interpret correlation coefficient values, we can use either `symnum()` or `Topm()` functions (see below), or `Mag()` together with `apply()`:

```
> noquote(apply(cor(trees), 1:2,
+ function(.x) Mag(.x, squared=FALSE)))  # shipunov
       Girth     Height    Volume
Girth  very high high      very high
Height high      very high high
Volume very high high      very high
```

If the numbers of observations in the columns are *unequal* (some columns have missing data), the parameter `use` becomes important. Default is `everything` which returns `NA` whenever there are any missing values in a dataset. If the parameter `use` is set to `complete.obs`, observations with missing data are automatically *excluded*. Sometimes, missing data values are so dispersed that `complete.obs` will not leave much of it. In that last case, use `pairwise.complete.obs` which removes missing values pair by pair.

Pearson's parametric correlation coefficients characteristically fail with the Anscombe's data:

```
> diag(cor(anscombe[, 1:4], anscombe[, 5:8]))
[1] 0.8164205 0.8162365 0.8162867 0.8165214  # correlation
```

To overcome the problem, one can use Spearman's ρ ("rho", or *rank correlation* coefficient) which is most frequently used *nonparametric correlation coefficient*:

```
> with(trees, cor(Girth, Height, method="spearman"))
[1] 0.4408387
> diag(cor(anscombe[, 1:4], anscombe[, 5:8], method="s"))
[1] 0.8181818 0.6909091 0.9909091 0.5000000
```

(Spearman's correlation is definitely more robust!)

The third kind of correlation coefficient in R is nonparametric Kendall's τ ("tau"):

```
> with(trees, cor(Girth, Height, method="k"))
[1] 0.3168641
> diag(cor(anscombe[, 1:4], anscombe[, 5:8], method="k"))
[1] 0.6363636 0.5636364 0.9636364 0.4264014
```

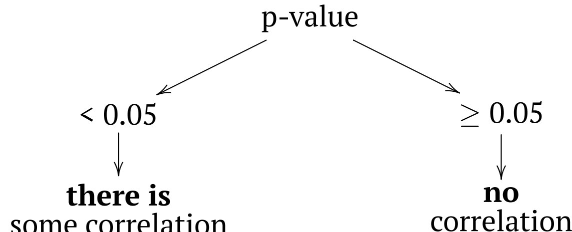
It is often used to measure association between two ranked or binary variables, i.e. as an alternative to effect sizes of the association in contingency tables.

* * *

How to check if correlation is statistically significant? As a *null hypothesis*, we could accept that correlation coefficient is equal to zero (*no correlation*). If the null is rejected, then correlation is significant:

```
> with(trees, cor.test(Girth, Height))
Pearson's product-moment correlation
data:  Girth and Height
t = 3.2722, df = 29, p-value = 0.002758
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 0.2021327 0.7378538
sample estimates:
      cor
0.5192801
```

The logic of `cor.test()` is the same as in tests before (Table 5.1, Fig. 5.1). In terms of p-value:



The probability of obtaining the test statistic (correlation coefficient), given the initial assumption of zero correlation between the data is very low—about 0.3%. We would reject H_0 and therefore accept an alternative hypothesis that correlation between variables is present. Please note the confidence interval, it indicates here that the true value of the coefficient lies between 0.2 and 0.7. with 95% probability.

* * *

It is not always easy to read the big correlation table, like in the following example of `longley` macroeconomic data. Fortunately, there are several workarounds, for example, the `symnum()` function which replaces numbers with letters or symbols in accordance to their value:

```
> symnum(cor(longley))
             GNP. GNP U A P Y E
GNP.deflator 1
GNP          B    1
Unemployed   ,    ,   1
Armed.Forces .    .     1
Population   B    B   , . 1
Year         B    B   , . B 1
Employed     B    B   . . B B 1
attr(,"legend")
[1] 0 ‘ ’ 0.3 ‘.’ 0.6 ‘,’ 0.8 ‘+’ 0.9 ‘*’ 0.95 ‘B’ 1
```

The second way is to represent the correlation matrix with a plot. For example, we may use the *heatmap*: split the range (we need absolute values because correlation changes from −1 to +1) into equal intervals, assign the color for each interval and show these colors (Fig. 6.3):

```
> library(lattice)
> mycorr <- abs(cor(iris[, -5]))
> levelplot(mycorr, xlab="", ylab="") # axes names a redundant
```

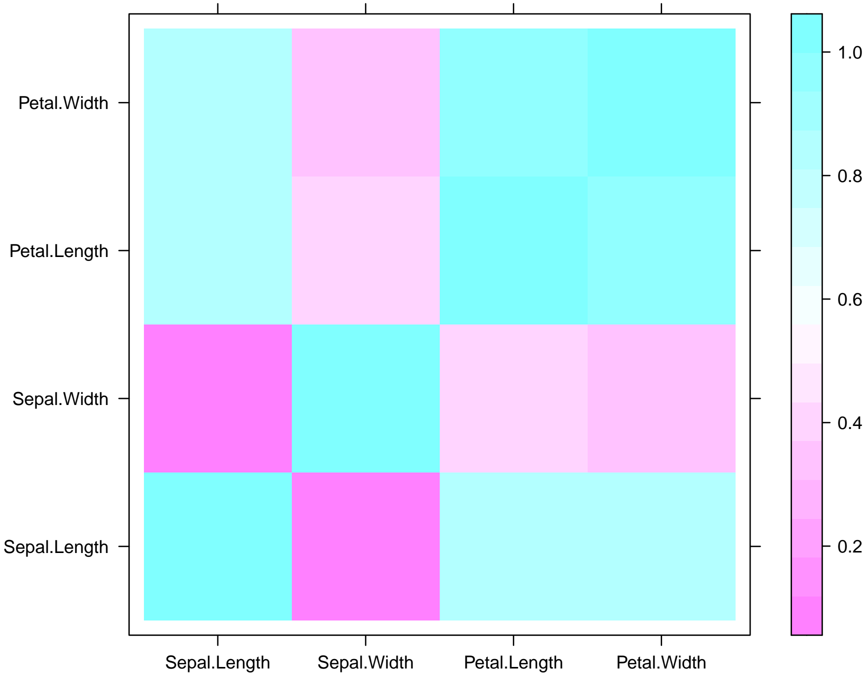


Figure 6.3: Heatmap: graphical representation of the correlation matrix.

The other interesting way of representing correlations are correlation ellipses (from ellipse package). In that case, correlation coefficients are shown as variously compressed ellipses; when coefficient is close to −1 or +1, ellipse is more narrow (Fig. 6.4). The slope of ellipse represents the sign of correlation (negative or positive):

```
> library(ellipse)
> colors <- cm.colors(7)
> cor.l <- cor(longley)
> plotcorr(cor.l, type="lower", col=colors[5*cor.l + 2])
```

GNP.deflator GNP Unemployed Armed.Forces Population Year
GNP
Unemployed
Armed.Forces
Population
Year
Employed

Figure 6.4: Correlation coefficients as ellipses.

* * *

The package shipunov presents several useful ways to visualize and analyze correlations:

```
> tox.cor <- cor(tox, method="k")
> Pleiad(tox.cor, corr=TRUE, lcol="black") # shipunov
```

We calculated here Kendall's correlation coefficient for the binary toxicity data to make the picture used on the title page. `Pleiad()` not only showed (Fig. 6.5) that illness is associated with tomato and Caesar salad, but also found two other correlation pleiads: coffee/rice and crab dip/crisps. (By the way, pleiads show one more application of R: *analysis of networks*.)

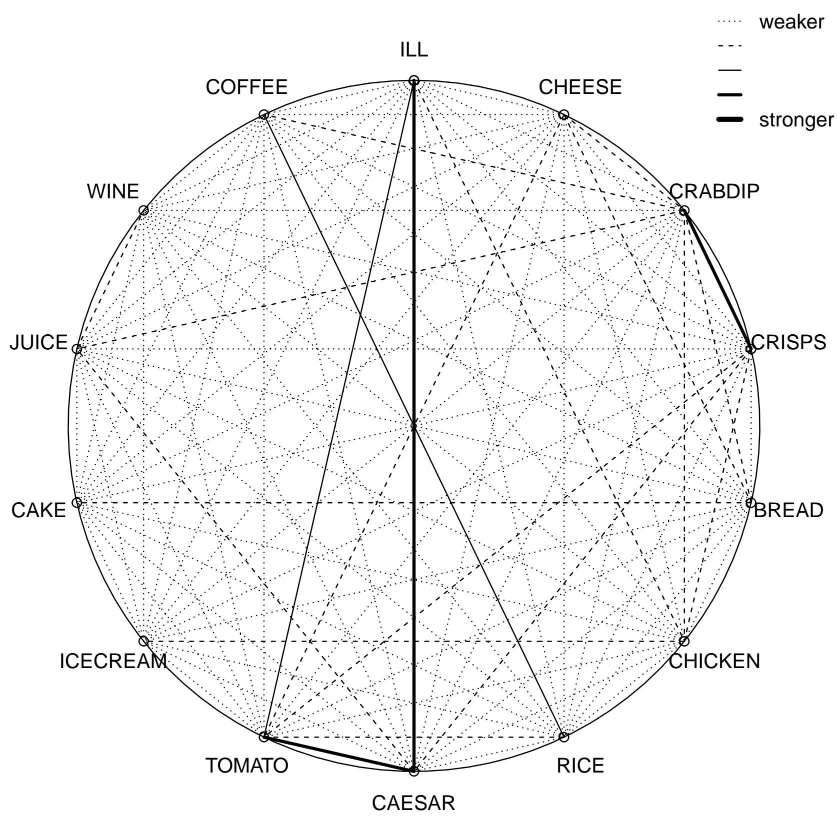


Figure 6.5: Correlation pleiads for the toxicity data.

Function `Cor()` outputs correlation matrix together with asterisks for the significant correlation tests:

```
> Cor(tox, method="kendall", dec=2) # shipunov
          ILL CHEESE CRABDIP CRISPS  BREAD CHICKEN CAESAR TOMATO
ILL         -   0.08    0.06  -0.03  -0.19    0.21    0.6*  0.46*
CHEESE   0.08      -  -0.38* -0.11   0.34*   0.04    0.08   0.22
CRABDIP  0.06 -0.38*       -  0.64*  -0.3*   0.27    0.04   0.19
```

```
CRISPS   -0.03  -0.11   0.64*     -  -0.05   0.33*  0.25  0.21
BREAD    -0.19   0.34*  -0.3* -0.05     -    0.05   0.03 -0.03
CHICKEN   0.21   0.04   0.27   0.33*  0.05     -    0.02  0.12
RICE      0.14   0.03   0.18   0.17   0.09   0.17    0.1  0.28
CAESAR     0.6*  0.08   0.04   0.25   0.03   0.02     -   0.64*
TOMATO    0.46*  0.22   0.19   0.21  -0.03   0.12   0.64*    -
...
```

(Please ignore warnings about ties.)

Finally, function `Topm()` shows largest correlations by rows:

```
> Topm(tox.cor, level=0.4) # shipunov
    Var1    Var2     Value Magnitude
1 TOMATO  CAESAR 0.6449020      high
2 CRISPS CRABDIP 0.6359727      high
3 CAESAR     ILL 0.6039006      high
4 TOMATO     ILL 0.4595725    medium
5 COFFEE    RICE 0.4134925    medium
```

> Data file `traits.txt` contains results of the survey where most genetically apparent human phenotype characters were recorded from many individuals. Explanation of these characters are in `trait_c.txt` file. Please analyze this data with correlation methods.

6.2 Analysis of regression

6.2.1 Single line

Analysis of correlation allows to determine if variables are dependent and calculate the strength and sign of the dependence. However, if the goal is to understand the other features of dependence (like direction), and, even more important, predict (extrapolate) results (Fig. 6.6) we need another kind of analysis, the *analysis of regression*.

It gives much more information on the relationship, but requires us to assign variables *beforehand* to one of two categories: *influence* (predictor) or *response*. This approach is rooted in the nature of the data: for example, we may use air temperature to predict ice cream sales, but hardly the other way around.

The most well-known example is a simple *linear regression*:

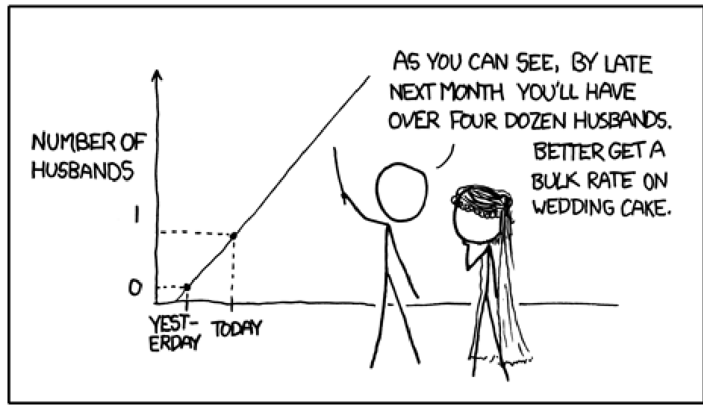


Figure 6.6: Extrapolation (taken from XKCD, `http://xkcd.com/605/`).

$$\text{response} = \text{intercept} + \text{slope} \times \text{influence}$$

or, in R formula language, even simpler:

`response ~ influence`

That model estimates the average value of *response* if the value of *influence* is known (note that both effect and influence are *measurement* variables). The differences between observed and predicted values are model *errors* (or, better, *residuals*). The goal is to *minimize residuals* (Fig. 6.8); since residuals could be both positive and negative, it is typically done via squared values, this method is called *least squares*.

Ideally, residuals should have the normal distribution with zero mean and constant variance which is not dependent on effect and influence. In that case, residuals are homogeneous. In other cases, residuals could show heterogeneity. And if there is the *dependence* between residuals and influence, then most likely the overall model should be non-linear and therefore requires the other kind of analysis.

Linear regression model is based on the several assumptions:

- Linearity of the relationship. It means that for a unit change in influence, there should always be a corresponding change in effect. Units of change in response variable should retain the same size and sign throughout the range of influence.

- Normality of *residuals*. Please note that normality of data is not an assumption! However, if you want to get rid of most other assumptions, you might want to use other regression methods like LOESS.

- Homoscedasticity of residuals. Variability within residuals should *remain constant* across the whole range of influence, or else we could not predict the effect reliably.

The null hypothesis states that *nothing* in the variability of response is explained by the model. Numerically, *R-squared* coefficient is the the degree to which the variability of response is explained by the model, therefore null hypothesis is that R-squared *equals zero*, this approach uses F-statistics (Fisher's statistics), like in ANOVA. There are also checks of additional null hypotheses that both *intercept and slope are zeros*. If all *three* p-values are smaller than the level of significance (0.05), the whole model is statistically significant.

Here is an example. The embedded `women` data contains observations on the height and weight of 15 women. We will try to understand the dependence between weight and height, graphically at first (Fig. 6.7):

```
> women.lm <- lm(weight ~ height, data=women)
> plot(weight ~ height, data=women,
+ xlab="Height, in", ylab="Weight, lb")
> grid()
> abline(women.lm, col="red")
> Cladd(women.lm, data=women) # shipunov
> legend("bottomright", col=2:1, lty=1:2,
+ legend=c("linear relationship", "95% confidence bands"))
```

(Here we used function `Cladd()` which adds *confidence bands* to the plot[2].)

Let us visualize residuals better (Fig. 6.8):

```
> plot(weight ~ height, data=women, pch=19, col="red")
> abline(women.lm)
> with(women, segments(height, fitted(women.lm), height, weight,
+ col="red"))
```

To look on the results of model analysis, we can employ `summary()`:

```
> summary(women.lm)
Call:
lm(formula = weight ~ height, data=women)
```

[2]Function `Cladd()` is applicable only to simple linear models. If you want confidence bands in more complex cases, check the `Cladd()` code to see what it does exactly.

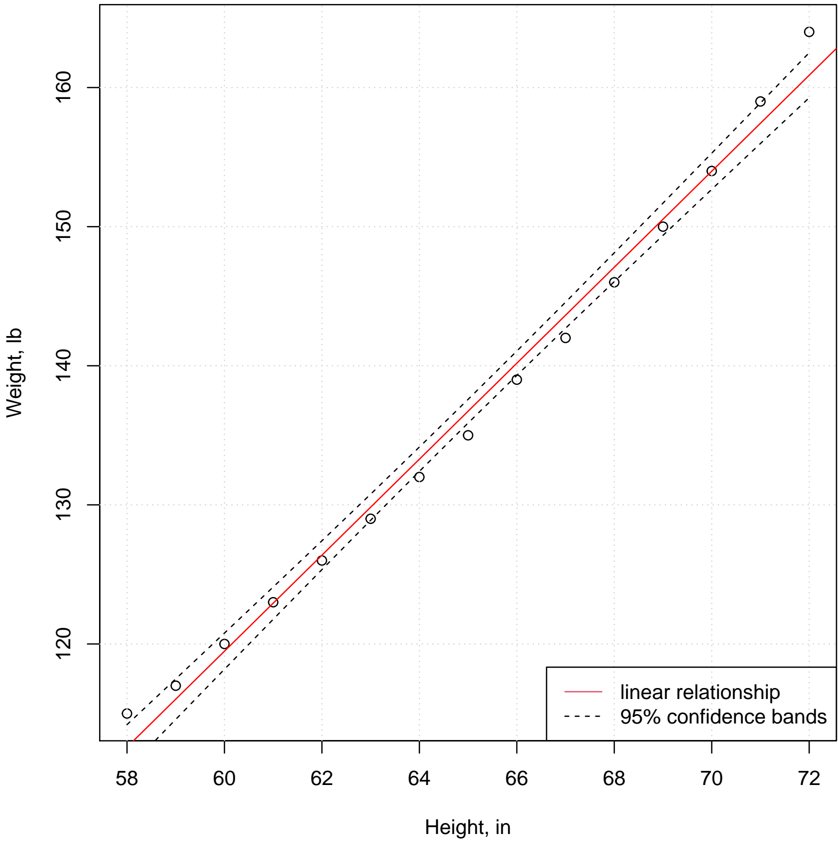


Figure 6.7: The relation between height and weight.

```
Residuals:
    Min      1Q  Median      3Q     Max
-1.7333 -1.1333 -0.3833  0.7417  3.1167
Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) -87.51667    5.93694  -14.74 1.71e-09 ***
height        3.45000    0.09114   37.85 1.09e-14 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 1.525 on 13 degrees of freedom
Multiple R-squared:  0.991,Adjusted R-squared:  0.9903
F-statistic:  1433 on 1 and 13 DF,  p-value: 1.091e-14
```

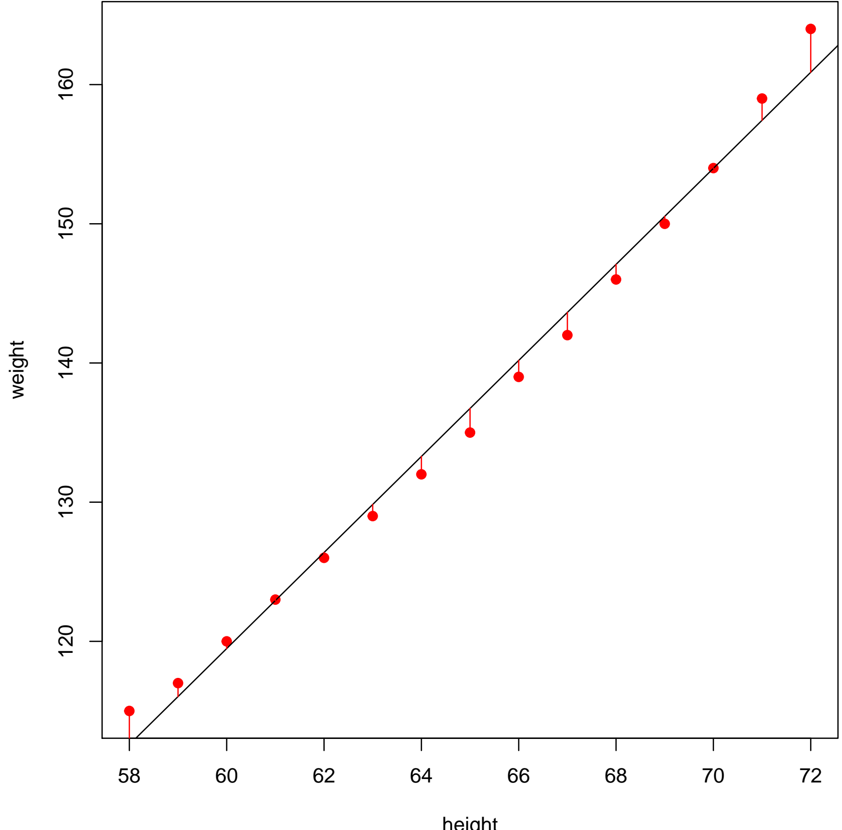


Figure 6.8: Residuals of the women weight *vs.* height linear model.

This long output is better to read from bottom to the top. We can say that:

- The significance of relation (reflected with R-squared) is high from the statistical point of view: F-statistics is 1433 with overall `p-value: 1.091e-14`.
- The R-squared (use `Adjusted R-squared` because this is better suited for the model) is really big, $R^2 = 0.9903$. This means that almost all variation in response variable (weight) is explained by predictor (height).

  R-squared is related with the coefficient of correlation and might be used as the measure of *effect size*. Since it is squared, high values start from 0.25:

```
> Mag(0.9903) # shipunov
[1] "very high"
```

- Both coefficients are statistically different from zero, this might be seen via "stars" (like `***`), and also via actual p-values `Pr(>|t|)`: `1.71e-09` for intercept, and `1.09e-14` for `height`, which represents the slope.

  To calculate slope in degrees, one might run:

  ```
  > (atan(women.lm$coefficients[[2]]) * 180)/pi
  [1] 73.8355
  ```

  This is the steep regression line.

- Overall, our model is:

  `Weight (estimated) = -87.51667 + 3.45 * Height,`

  so if the height grows by 4 inches, the weight will grow on approximately 14 pounds.

- The maximal positive residual is 3.1167 lb, maximal negative is −1.7333 lb.

- Half of residuals are quite close to the median (within approximately ±1 interval).

On the first glance, the model summary looks fine. However, before making any conclusions, we must also *check assumptions* of the model. The command `plot(women.lm)` returns four consecutive plots:

- First plot, *residuals vs. fitted values*, is most important. Ideally, it should show *no structure* (uniform variation and no trend); this satisfies both linearity and homoscedascicity assumptions.

  Unfortunately, `women.lm` model has an obvious trend which indicates nonlinearity. Residuals are positive when fitted values are small, negative for fitted values in the mid-range, and positive again for large fitted values. Clearly, the first assumption of the linear regression analysis is violated.

  To understand residuals *vs.* fitted plots better, please **run** the following code yourself and look on the resulted plots:

  ```
  > oldpar <- par(mfrow=c(3, 3))
  > ## Uniform variation and no trend:
  > for (i in 1:9) plot(1:50, rnorm(50), xlab="Fitted",
  + ylab="Residuals")
  > title("'Good' Residuals vs. Fitted", outer=TRUE, line=-2)
  > ## Non-uniform variation plus trend:
  > for (i in 1:9) plot(1:50, ((1:50)*rnorm(50) + 50:1),
  + xlab="Fitted",ylab="Residuals")
  ```

```
> title("'Bad' Residuals vs. Fitted", outer=TRUE, line=-2)
> par(oldpar)
```

- On the the next plot, standardized residuals do not follow the normal line perfectly (see the explanation of the QQ plot in the previous chapter), but they are "good enough". To review different variants of these plots, **run** the following code yourself:

  ```
  > oldpar <- par(mfrow=c(3, 3))
  > for (i in 1:9) { aa <- rnorm(50);
  + qqnorm(aa, main=""); qqline(aa) }
  > title("'Good' normality QQ plots", outer=TRUE, line=-2)
  > for (i in 1:9) { aa <- rnorm(50)^2;
  + qqnorm(aa, main=""); qqline(aa) }
  > title("'Bad' normality QQ plots", outer=TRUE, line=-2)
  > par(oldpar)
  ```

  Test for the normality should also work:

  ```
  > Normality(women.lm$residuals) # shipunov
  [1] "NORMAL"
  ```

- The third, *Scale-Location* plot, is similar to the residuals *vs.* fitted, but instead of "raw" residuals it uses the square roots of their standardized values. It is also used to reveal trends in the magnitudes of residuals. In a good model, these values should be more or less randomly distributed.

- Finally, the last plot demonstrates which values exert most influence over the final shape of the model. Here the two values with most leverage are the first and the last measurements, those, in fact, that stay furtherest away from linearity.

(If you need to know more about summary and plotting of linear models, check help pages with commands `?summary.lm` and `?plot.lm`. By the way, as ANOVA has many similarities to the linear model analysis, in R you can run same diagnostic plots for any ANOVA model.)

Now it is clear that our first linear model *does not work well* for our data which is likely *non-linear*. While there are many non-linear regression methods, let us modify it first in a more simple way to introduce non-linearity. One of simple ways is to add the *cubed term*, because weight relates with volume, and volume is a cube of linear sizes:

```
> women.lm3 <- lm(weight ~ height + I(height^3), data=women)
> summary(women.lm3)
> plot(women.lm3, which=1) # just residuals vs. fitted
```

(Function `I()` was used to tell `R` that `height^3` is arithmetical operation and not the part of model formula.)

The quick look on the residuals *vs.* fitted plot (Fig 6.9) shows that this second model fits much better! Confidence bands and predicted line are also look more appropriate :

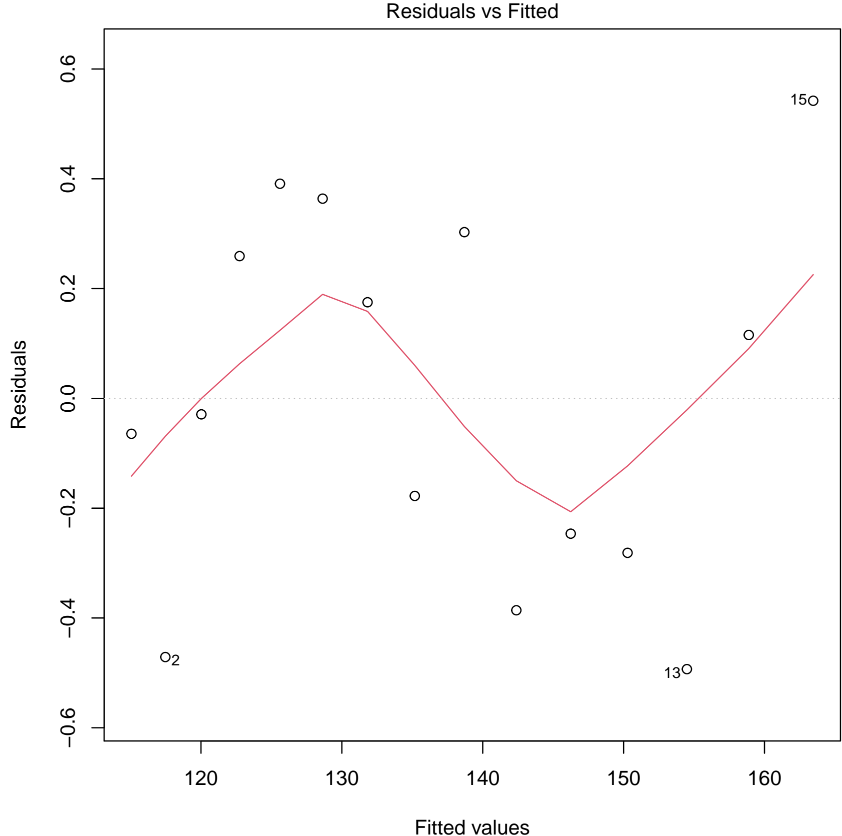


Figure 6.9: Residuals *vs.* fitted plot for the women height/weight model with the cubed term.

You may want also to see the *confidence intervals* for linear model parameters. For that purpose, **use** `confint(women.lm)`.

* * *

Another example is from egg data studied graphically in the second chapter (Fig 2.9). Does the length of the egg linearly relate with with of the egg?

```
> eggs <- read.table("data/eggs.txt")
> eggs.lm <- lm(V2 ~ V1, data=eggs)
```

We can analyze the assumptions first:

```
> plot(eggs.lm)
```

The most important, residuals vs. fitted is not perfect but could be considered as "good enough" (please **check** it yourself): there is no obvious trend, and residuals seem to be more or less equally spread (homoscedasticity is fulfilled). Distribution of residuals is close to normal. Now we can interpret the model summary:

```
> summary(eggs.lm)
Call:
lm(formula = V2 ~ V1, data = eggs)
Residuals:
    Min      1Q  Median      3Q     Max
-6.8039 -1.7064 -0.1321  1.6394 11.7090
Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  0.90641    0.67195   1.349    0.178
V1           0.71282    0.01772  40.228   <2e-16 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
Residual standard error: 2.725 on 344 degrees of freedom
Multiple R-squared:  0.8247,Adjusted R-squared:  0.8242
F-statistic:  1618 on 1 and 344 DF,  p-value: < 2.2e-16
```

Significance of the slope means that the line is definitely *slanted* (this is actually what is called "relation" in common language). However, intercept is not significantly different from zero:

```
> confint(eggs.lm)
                 2.5 %    97.5 %
(Intercept) -0.4152310 2.2280551
V1           0.6779687 0.7476725
```

(Confidence interval for intercept includes zero.)

To check the magnitude of effect size, one can use:

```
> Mag(summary(eggs.lm)$adj.r.squared)
[1] "very high"
```

This is a really large effect.

* * *

Third example is based on a simple idea to check if the success in multiple choice test depends on time spent with it. Data presents in `exams.txt` file which contains results of two multiple choice tests in a large class:

```
> ee <- read.table("http://ashipunov.me/data/exams.txt", h=TRUE)
> str(ee)
'data.frame': 201 obs. of  3 variables:
 $ EXAM.N   : int  3 3 3 3 3 3 3 3 3 3 ...
 $ ORDER    : int  1 2 3 4 5 6 7 8 9 10 ...
 $ POINTS.50: int  42 23 30 32 27 19 37 30 36 28 ...
```

First variable is the number of test, two others are order of finishing the work, and resulted number of points (out of 50). We assume here that the order reflects the time spent on test. Select one of two exams:

```
> ee3 <- ee[ee$EXAM.N == 3,]
```

... and plot it first (please **check** this plot yourself):

```
> plot(POINTS.50 ~ ORDER, data=ee3)
```

Well, no visible relation occurs. Now we approach it inferentially:

```
> ee3.lm <- lm(POINTS.50 ~ ORDER, data=ee3)
> summary(ee3.lm)
Call:
lm(formula = POINTS.50 ~ ORDER, data = ee3)
Residuals:
     Min       1Q   Median       3Q      Max
-16.0118  -4.7561   0.4708   4.4344  13.4695
Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 33.35273    1.24634  26.761   <2e-16 ***
ORDER       -0.02005    0.02143  -0.936    0.352
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
Residual standard error: 6.185 on 98 degrees of freedom
Multiple R-squared:  0.008859,Adjusted R-squared:  -0.001254
```

```
F-statistic: 0.876 on 1 and 98 DF,  p-value: 0.3516
```

As usual, this output is read from bottom to the top. First, statistical significance of the relation is absent, and relation (adjusted R-squared) itself is almost zero. Even if intercept is significant, slope is not and therefore could easily be zero. There is no relation between time spent and result of the test.

To double check if the linear model approach was at all applicable in this case, **run** diagnostic plots yourself:

```
> plot(ee3.lm)
```

And as the final touch, **try** the regression line and confidence bands:

```
> abline(ee3.lm)
> Cladd(ee3.lm, data=ee3)
```

Almost horizontal—no relation. It is also interesting to check if the other exam went the same way. Please **find out** yourself.

> In the open repository, data file `erophila.txt` contains measurements of spring draba plant (*Erophila verna*). Please find which morphological measurement characters are most correlated, and check the linear model of their relationships.

> In the `shipunov` package, there is a dataset `drosera` which contains data from morphological measurements of more than thousand plants and multiple populations of the carnivorous sundew (*Drosera*) plant. Please find which pair of morphological characters is most correlated and analyze the linear model which includes these characters. Also, check if length of leaf is different between the three biggest populations of sundew.

* * *

As the linear models and ANOVA have many in common, there is no problem in the analysis of multiple groups with the default linear regression methods. Consider our ANOVA data:

```
> newcolor <- relevel(hwc$COLOR, "brown")
> summary(lm(cbind(WEIGHT, HEIGHT) ~ newcolor, data=hwc))
Response WEIGHT :
...
Coefficients:
```

```
               Estimate Std. Error t value Pr(>|t|)
(Intercept)     75.0000     0.5469 137.129  < 2e-16 ***
newcolorblack    4.2333     0.7735   5.473  4.2e-07 ***
newcolorblond   -0.7667     0.7735  -0.991    0.324
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 2.996 on 87 degrees of freedom
Multiple R-squared:  0.3579,Adjusted R-squared:  0.3431
F-statistic: 24.24 on 2 and 87 DF,  p-value: 4.286e-09

Response HEIGHT :
...
Coefficients:
               Estimate Std. Error t value Pr(>|t|)
(Intercept)    164.2333     0.5506 298.302  < 2e-16 ***
newcolorblack    5.6333     0.7786   7.235 1.72e-10 ***
newcolorblond   -7.9333     0.7786 -10.189  < 2e-16 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 3.016 on 87 degrees of freedom
Multiple R-squared:  0.7789,Adjusted R-squared:  0.7738
F-statistic: 153.3 on 2 and 87 DF,  p-value: < 2.2e-16
```

This example shows few additional “tricks”. First, this is how to analyze several response variables at once. This is applicable also to `aov()`—**try** it yourself.

Next, it shows how to re-level factor putting one of proximal levels first. That helps to compare coefficients. In our case, it shows that blonds do not differ from browns by weight. Note that “intercepts” here have no clear relation with plotting linear relationships.

It is also easy to calculate the effect size because *R-squared is the effect size*.

Last but not least, please **check** assumptions of the linear model with `plot(lm(...))`. At the moment in R, this works only for singular response.

▌ Is there the linear relation between the weight and height in our ANOVA `hwc` data?

6.2.2 Many lines

Sometimes, there is a need to analyze not just linear relationships between variables, but to answer second order question: *compare several regression lines*.

In formula language, this is described as

```
response ~ influence * factor
```

where factor is a categorical variable responsible for the distinction between regression lines, and star (*) indicates that we are simultaneously checking (1) response from influence (predictor), (2) response from factor and (3) response from *interaction* between influence and factor.

This kind of analysis is frequently called ANCOVA, "ANalysis of COVAriation". The ANCOVA will check if there is any difference between intercept and slope of the first regression line and intercepts and slopes of all other regression lines where each line corresponds with one factor level.

Let us start from the example borrowed from M.J. Crawley's "R Book". 40 plants were treated in two groups: grazed (in first two weeks of the cultivation) and not grazed. Rootstock diameter was also measured. At the end of season, dry fruit production was measured from both groups. First, we analyze the data graphically:

```
> ipo <- read.table("data/ipomopsis.txt", h=TRUE)
> with(ipo, plot(Root, Fruit, pch=as.numeric(Grazing)))
> abline(lm(Fruit ~ Root, data=subset(ipo, Grazing=="Grazed")))
> abline(lm(Fruit ~ Root, data=subset(ipo, Grazing=="Ungrazed")),
+ lty=2)
> legend("topleft", lty=1:2, legend=c("Grazed","Ungrazed"))
```

As it is seen on the plot (Fig. 6.10), regression lines for grazed and non-grazed plants are likely different. Now to the ANCOVA model:

```
> ipo.lm <- lm(Fruit ~ Root * Grazing, data=ipo)
> summary(ipo.lm)
Call:
lm(formula = Fruit ~ Root * Grazing, data=ipo)

Residuals:
     Min       1Q   Median       3Q      Max
-17.3177  -2.8320   0.1247   3.8511  17.1313

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

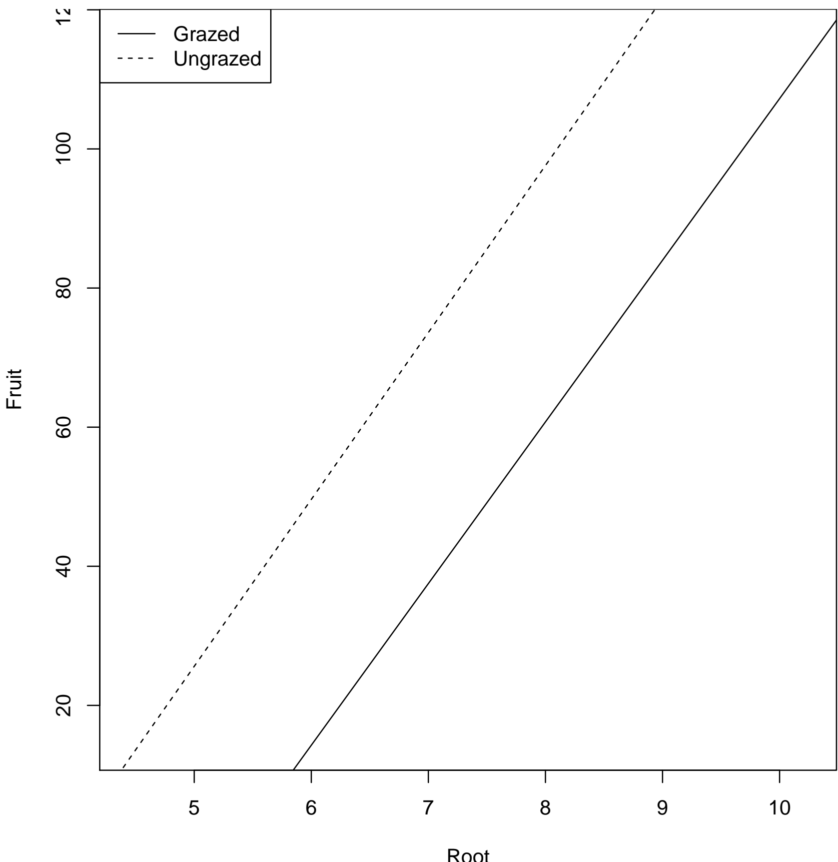


Figure 6.10: Grazed *vs*. non-grazed plants: linear models.

```
(Intercept)           -125.173     12.811  -9.771 1.15e-11 ***
Root                    23.240      1.531  15.182  < 2e-16 ***
GrazingUngrazed         30.806     16.842   1.829   0.0757 .
Root:GrazingUngrazed     0.756      2.354   0.321   0.7500
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 6.831 on 36 degrees of freedom
Multiple R-squared:  0.9293,Adjusted R-squared:  0.9234
F-statistic: 157.6 on 3 and 36 DF,  p-value: < 2.2e-16
```

Model output is similar to the linear model but one more term is present. This term indicated *interaction* which labeled with colon. Since `Grazing` factor has two level ar-

ranged alphabetically, first level (`Grazed`) used as default and therefore `(Intercept)` belongs to grazed plants group. The intercept of non-grazed group is labeled as `GrazingUngrazed`. In fact, this is not even an intercept but difference between intercept of non-grazed group and intercept of grazed group. Analogously, slope for grazed is labeled as `Root`, and difference between slopes of non-grazed and grazed labeled as `Root:GrazingUngrazed`. This difference is interaction, or how grazing affects the shape of relation between rootstock size and fruit weight. To convert this output into regression formulas, some calculation will be needed:

`Fruit = -125.174 + 23.24 * Root` (grazed)
`Fruit = (-125.174 + 30.806) + (23.24 + 0.756) * Root` (non-grazed)

Note that difference between slopes is not significant. Therefore, interaction could be ignored. Let us check if this is true:

```
> ipo.lm2 <- update(ipo.lm, . ~ . - Root:Grazing)
> summary(ipo.lm2)
> AIC(ipo.lm)
> AIC(ipo.lm2)
```

First, we updated our first model by removing the interaction term. This is the *additive* model. Then `summary()` told us that all coefficients are now significant (**check** its output yourself). This is definitely better. Finally, we employed AIC (Akaike's Information Criterion). AIC came from the theory of information and typically reflects the entropy, in other words, adequacy of the model. The smaller is AIC, the better is a model. Then the second model is the unmistakable winner.

By the way, we could specify the same additive model using plus sign instead of star in the model formula.

What will the AIC tell about our previous example, women data models?

```
> AIC(women.lm)
> AIC(women.lm3)
```

Again, the second model (with the cubed term) is better.

* * *

It is well known that in the analysis of voting results, dependence between attendance and the number of people voted for the particular candidate, plays a great role. It is possible, for example, to elucidate if elections were falsified. Here we will use the `elections.txt` data file containing voting results for three different Russian parties in more than 100 districts:

```
> elections <- read.table("data/elections.txt", h=TRUE)
> str(elections)
'data.frame': 153 obs. of  7 variables:
 $ DISTRICT: int  1 2 3 4 5 6 7 8 9 10 ...
 $ VOTER   : int  329786 144483 709903 696114 ...
 $ INVALID  : int  2623 859 5656 4392 3837 4715 ...
 $ VALID: int  198354 97863 664195 619629 ...
 $ CAND.1  : int  24565 7884 30491 54999 36880 ...
 $ CAND.2  : int  11786 6364 11152 11519 10002 ...
 $ CAND.3  : int  142627 68573 599105 525814 ...
```

To simplify typing, we will attach() the data frame (if you do the same, do not forget to detach() it at the end) and calculate proportions of voters and the overall attendance:

```
> attach(elections)
> PROP <- cbind(CAND.1, CAND.2, CAND.3) / VOTER
> ATTEN <- (VALID + INVALID) / VOTER
```

Now we will look on the dependence between attendance and voting graphically (Fig. 6.11):

```
> lm.1 <- lm(CAND.1/VOTER ~ ATTEN)
> lm.2 <- lm(CAND.2/VOTER ~ ATTEN)
> lm.3 <- lm(CAND.3/VOTER ~ ATTEN)
> plot(CAND.3/VOTER ~ ATTEN, xlim=c(0, 1), ylim=c(0, 1),
+ xlab="Attendance", ylab="Percent voted for the candidate")
> points(CAND.1/VOTER ~ ATTEN, pch=2)
> points(CAND.2/VOTER ~ ATTEN, pch=3)
> abline(lm.3)
> abline(lm.2, lty=2)
> abline(lm.1, lty=3)
> legend("topleft", lty=c(3, 2, 1),
+ legend=c("Party 1", "Party 2", "Party 3"))
> detach(elections)
```

So the third party had a voting process which was suspiciously different from voting processes for two other parties. It was clear even from the graphical analysis but we might want to test it inferentially, using ANCOVA:

```
> elections2 <- cbind(ATTEN, stack(data.frame(PROP)))
> names(elections2) <- c("atten", "percn", "cand")
> str(elections2)
'data.frame': 459 obs. of  3 variables:
```

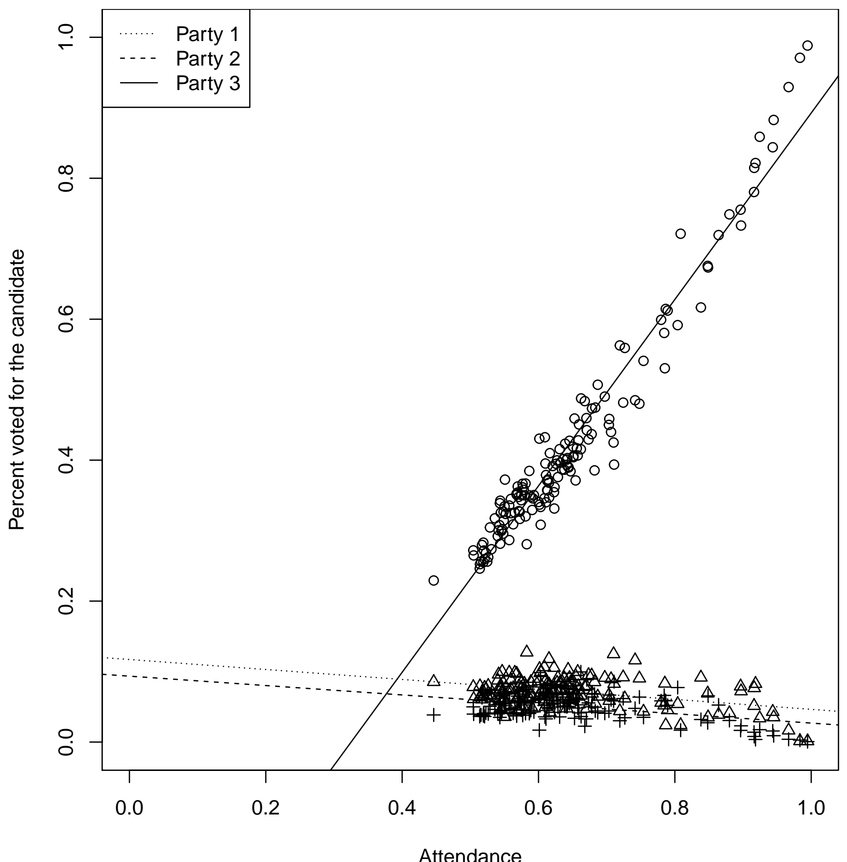


Figure 6.11: Voting results *vs.* attendance for every party.

```
 $ atten: num  0.609 0.683 0.944 0.896 0.916 ...
 $ percn: num  0.0745 0.0546 0.043 0.079 0.0514 ...
 $ cand : Factor w/ 3 levels "CAND.1","CAND.2",..: 1 1 1 ...
```

(Here we created and checked the new data frame. In `elections2`, all variables are now `stack()`'ed in two columns, and the third column contains the party code.)

```
> ancova.v <- lm(percn ~ atten * cand, data=elections2)
> summary(ancova.v)

Call:
lm(formula = percn ~ atten * cand, data=elections2)

Residuals:
```

```
      Min        1Q    Median        3Q       Max
-0.116483 -0.015470 -0.000699  0.014825  0.102810

Coefficients:
                  Estimate Std. Error t value Pr(>|t|)
(Intercept)       0.117115   0.011973   9.781  < 2e-16 ***
atten            -0.070726   0.018266  -3.872 0.000124 ***
candCAND.2       -0.023627   0.016933  -1.395 0.163591
candCAND.3       -0.547179   0.016933 -32.315  < 2e-16 ***
atten:candCAND.2  0.004129   0.025832   0.160 0.873074
atten:candCAND.3  1.393336   0.025832  53.938  < 2e-16 ***
---
Signif. codes: 0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 0.02591 on 453 degrees of freedom
Multiple R-squared: 0.9824,Adjusted R-squared: 0.9822
F-statistic:  5057 on 5 and 453 DF,  p-value: < 2.2e-16
```

Here `(Intercept)` belongs specifically to the model for first party. Its p-value indicates if it differs significantly from zero. Second coefficient, `atten`, belongs to the continuous predictor, attendance. It is not an intercept but slope of a regression. It is also compared to zero.

Next four rows represent differences from the first party, two for intercepts and two for slopes (this is the traditional way to structure output in R). Last two items represent interactions. We were most interested if there is an interaction between attendance and voting for the third party, this interaction is common in case of falsifications and our results support this idea.

> In the open repository, file `heterostyly.txt` contains results of flower parts measurements in populations of two primroses, *Primula veris* and *P. vulgaris*. Primroses are famous with their *heterostyly* (Fig. 6.12, phenomenon when in each population there are mostly two types of plants: with flowers bearing long stamens and short style, and with flowers bearing long style and short stamens. It was proved that heterostyly helps in pollination. Please check if the linear dependencies between lengths of style and stamens are different in these two species. Find also which model is better, full (multiplicative, with interactions) or reduced (additive, without interactions).

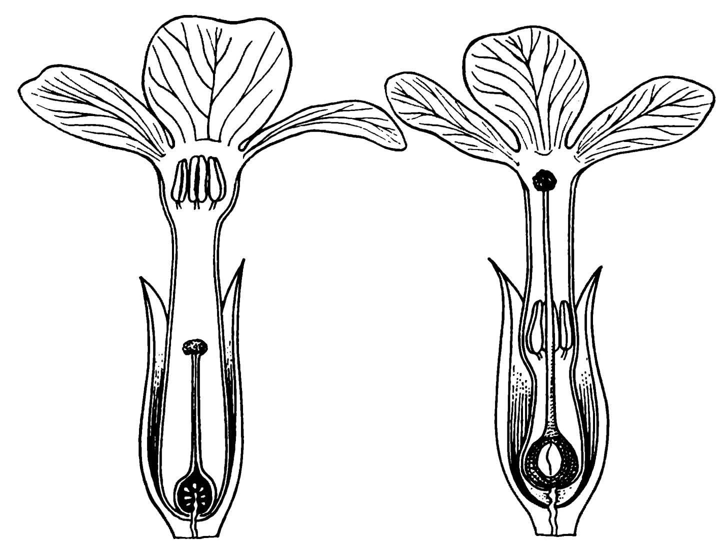


Figure 6.12: Heterostyly in primroses: flowers from the different plants of one population.

6.2.3 More then one way, again

Armed with the knowledge about AIC, multiplicative and additive models, we can return now to the ANOVA two-way layouts, briefly mentioned before. Consider the following example:

```
> ToothGrowth.1 <- with(ToothGrowth,
+ data.frame(len, supp, fdose=factor(dose)))
> str(ToothGrowth.1)
'data.frame': 60 obs. of  3 variables:
 $ len  : num  4.2 11.5 7.3 5.8 6.4 10 11.2 11.2 5.2 7 ...
 $ supp : Factor w/ 2 levels "OJ","VC": 2 2 2 2 2 2 2 2 2 2 ...
 $ fdose: Factor w/ 3 levels "0.5","1","2": 1 1 1 1 1 1 1 1 1 1 ...
> Normality(ToothGrowth$len)
[1] "NORMAL"
> with(ToothGrowth, fligner.test(split(len, list(dose, supp))))
Fligner-Killeen test of homogeneity of variances
data:  split(len, list(dose, supp))
Fligner-Killeen:med chi-squared = 7.7488, df = 5, p-value = 0.1706
```

(To start, we converted dose into factor. Otherwise, our model will be ANCOVA instead of ANOVA.)

Assumptions met, now the core analysis:

```
> summary(aov(len ~ supp * fdose, data=ToothGrowth.1))
            Df Sum Sq Mean Sq F value   Pr(>F)
supp         1  205.4   205.4  15.572 0.000231 ***
fdose        2 2426.4  1213.2  92.000  < 2e-16 ***
supp:fdose   2  108.3    54.2   4.107 0.021860 *
Residuals   54  712.1    13.2
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
> AIC(aov(len ~ supp * fdose, data=ToothGrowth.1))
[1] 332.7056

> summary(aov(len ~ supp + fdose, data=ToothGrowth.1))
            Df Sum Sq Mean Sq F value   Pr(>F)
supp         1  205.4   205.4   14.02 0.000429 ***
fdose        2 2426.4  1213.2   82.81  < 2e-16 ***
Residuals   56  820.4    14.7
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
> AIC(aov(len ~ supp + fdose, data=ToothGrowth.1))
[1] 337.2013
```

Now we see what was already visible on the interaction plot (Fig. 5.9): model with interactions is better, and significant are *all three terms*: dose, supplement, and interaction between them.

Effect size is really high:

```
> Eta2(aov(len ~ supp * fdose, data=ToothGrowth.1))
[1] 0.7937246
```

Post hoc tests are typically more dangerous in two-way analysis, simply because there are much more comparisons. However, it is possible to run `TukeyHSD()`:

```
> TukeyHSD(aov(len ~ supp * fdose, data=ToothGrowth.1))
  Tukey multiple comparisons of means
    95% family-wise confidence level
Fit: aov(formula = len ~ supp * fdose, data = ToothGrowth.1)

$supp
      diff       lwr       upr     p adj
VC-OJ -3.7 -5.579828 -1.820172 0.0002312
```

```
$fdose
        diff       lwr       upr   p adj
1-0.5  9.130  6.362488 11.897512 0.0e+00
2-0.5 15.495 12.727488 18.262512 0.0e+00
2-1    6.365  3.597488  9.132512 2.7e-06

$`supp:fdose`
               diff        lwr        upr     p adj
VC:0.5-OJ:0.5 -5.25 -10.048124 -0.4518762 0.0242521
OJ:1-OJ:0.5    9.47   4.671876 14.2681238 0.0000046
...
```

The rest of comparisons is here omitted, but `TukeyHSD()` has plotting method allowing to plot the single or last element (Fig. 6.13):

```
> plot(TukeyHSD(aov(len ~ supp * fdose, data=ToothGrowth.1)), las=1)
```

6.3 Probability of the success: logistic regression

There are a few analytical methods working with categorical variables. Practically, we are restricted here with proportion tests and chi-squared. However, the goal sometimes is more complicated as we may want to check not only the presence of the correspondence but also its *features*—something like regression analysis but for the nominal data. In formula language, this might be described as

`factor ~ influence`

Below is an example using data from hiring interviews. Programmers with different months of professional experience were asked to write a program on paper. Then the program was entered into the memory of a computer and if it worked, the case was marked with "S" (success) and "F" (failure) otherwise:

```
> l <- read.table("data/logit.txt")
> l$V2 <- as.factor(l$V2)
> head(l)
  V1 V2
1 14  F
2 29  F
3  6  F
4 25  S
5 18  S
6  4  F
```

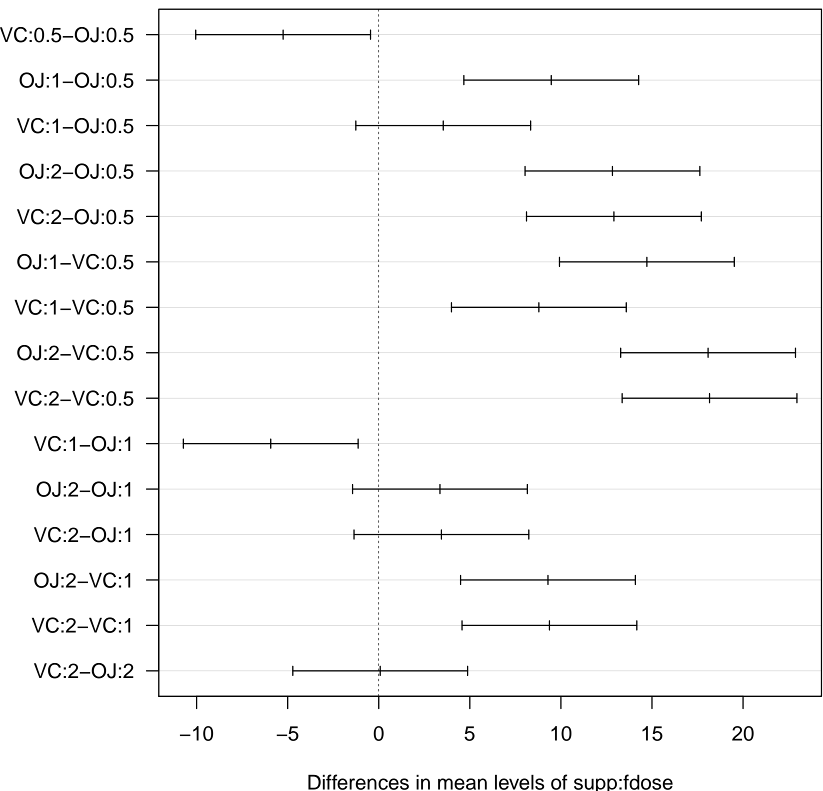


Figure 6.13: `TukeyHSD()` plot for supplement-dose multiple comparisons (`ToothGrowth` data).

It is more or less obvious more experienced programmers are more successful. This is even possible to check visually, with `cdplot()` (Fig 6.14):

```
> cdplot(V2 ~ V1, data=l,
+ xlab="Experience, months", ylab="Success")
```

But is it possible to determine numerically the dependence between years of experience and programming success? Contingency tables is not a good solution because `V1` is a measurement variable. Linear regression will not work because the response here is a factor. But there is a solution. We can research the model where the response is not a success/failure but the *probability of success* (which, as all probabilities is a measurement variable changing from 0 to 1):

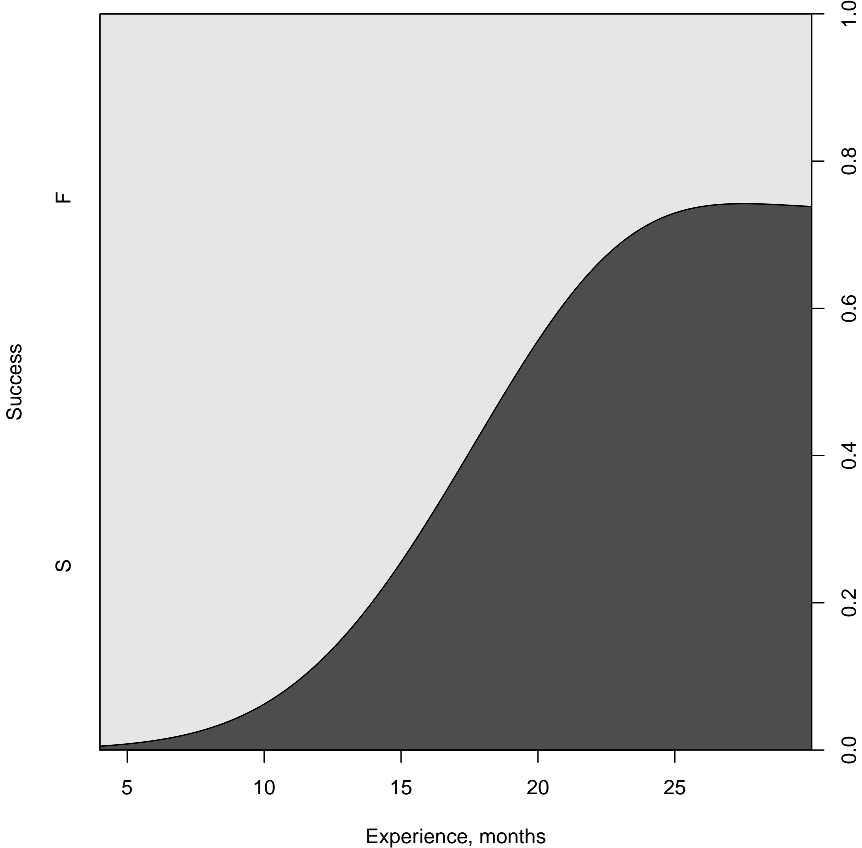


Figure 6.14: Conditional density plot shows the probability of programmer's success.

```
> l.logit <- glm(V2 ~ V1, family=binomial, data=l)
> summary(l.logit)

Call:
glm(formula = V2 ~ V1, family = binomial, data=l)

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-1.9987  -0.4584  -0.2245   0.4837   1.5005

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept)  -4.9638     2.4597  -2.018   0.0436 *
```

```
V1             0.2350     0.1163   2.021   0.0432 *
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 18.249  on 13  degrees of freedom
Residual deviance: 10.301  on 12  degrees of freedom
AIC: 14.301

Number of Fisher Scoring iterations: 5
```

Not going deeply into details, we can see here that both parameters of the regression are significant since p-values are small. This is enough to say that the experience influences the programming success.

> The file `seeing.txt` came from the results of the following experiment. People were demonstrated some objects for the short time, and later they were asked to describe these objects. First column of the data file contains the person ID, second—the number of object (five objects were shown to each person in sequence) and the third column is the success/failure of description (in binary 0/1 format). Is there dependence between the object number and the success?

* * *

The output of `summary.glm()` contains the AIC value. It is accepted that smaller AIC corresponds with the more optimal model. To show it, we will return to the intoxication example from the previous chapter. Tomatoes or salad?

```
> tox.logit <- glm(formula=I(2-ILL) ~ CAESAR + TOMATO,
+ family=binomial, data=tox)
> tox.logit2 <- update(tox.logit, . ~ . - TOMATO)
> tox.logit3 <- update(tox.logit, . ~ . - CAESAR)
```

At first, we created the logistic regression model. Since it “needs” the binary response, we subtracted the `ILL` value from 2 so the illness became encoded as 0 and no illness as 1. `I()` function was used to avoid the subtraction to be interpret as a model formula, and our minus symbol had only arithmetical meaning. On the next step, we used `update()` to modify the starting model removing tomatoes, then we removed the salad (dots mean that we use all initial influences and responses). Now to the AIC:

```
> tox.logit$aic
[1] 47.40782
> tox.logit2$aic
[1] 45.94004
> tox.logit3$aic
[1] 53.11957
```

The model without tomatoes but with salad is the most optimal. It means that the poisoning agent was most likely the Caesar salad alone.

* * *

Now, for the sake of completeness, readers might have question if there are methods similar to logistic regression but using not two but *many factor levels* as response? And methods using *ranked* (ordinal) variables as response? (As a reminder, measurement variable as a response is a property of linear regression and similar.) Their names are *multinomial regression* and *ordinal regression*, and appropriate functions exist in several R packages, e.g., `nnet`, `rms` and `ordinal`.

> File `juniperus.txt` in the open repository contains measurements of morphological and ecological characters in several Arctic populations of junipers (*Juniperus*). Please analyze how measurements are distributed among populations, and check specifically if the needle length is different between locations.
>
> *Another problem* is that junipers of smaller size (height less than 1 m) and with shorter needles (less than 8 mm) were frequently separated from the common juniper (*Juniperus communis*) into another species, *J. sibirica*. Please check if plants with *J. sibirica* characters present in data, and does the probability of being *J. sibirica* depends on the amount of shading pine trees in vicinity (character `PINE.N`).

6.4 Answers to exercises

6.4.1 Correlation and linear models

Answer to the question of human traits. Inspect the data, load it and check the object:

```
> traits <- read.table("data/traits.txt", h=TRUE, row.names=1)
> Str(traits)
 'data.frame': 21 obs. of  9 variables:
  1 TONGUE : int  0 0 0 0 0 0 1 0 1 1 ...
```

```
  2 EARLOBE: int  0 0 0 1 0 0 0 1 1 1 ...
  3 PINKY  : int  0 1 1 1 1 0 1 0 1 0 ...
  4 ARM    : int  0 1 0 1 1 0 0 1 1 0 ...
  5 CHEEK  : int  1 0 0 0 1 0 0 1 0 0 ...
  6 CHIN   : int  1 1 1 1 1 1 1 1 0 0 ...
  7 THUMB  : int  0 0 0 1 0 0 1 1 0 1 ...
  8 TOE    : int  0 1 0 0 1 1 1 0 0 0 ...
  9 PEAK   : int  0 1 1 1 0 0 1 1 0 0 ...
 row.names [1:21] "A" "B" "C" "D" "E" ...
```

Data is binary, so Kendall's correlation is most natural:

```
> Cor(traits, method="kendall", dec=1) # shipunov
        TONGUE EARLOBE PINKY   ARM CHEEK  CHIN THUMB   TOE  PEAK
TONGUE       -     0.1  -0.2  -0.1  -0.1 -0.6*  0.5*     0  -0.2
EARLOBE    0.1       -  -0.2   0.1  -0.1  -0.4   0.3  -0.3     0
PINKY     -0.2    -0.2     -   0.2   0.1     0  -0.1     0   0.2
ARM       -0.1     0.1   0.2     -   0.2     0  -0.1   0.1     0
CHEEK     -0.1    -0.1   0.1   0.2     -  -0.1     0  -0.1  -0.1
CHIN     -0.6*    -0.4     0     0  -0.1     - -0.5*     0   0.4
THUMB     0.5*     0.3  -0.1  -0.1     0 -0.5*     -   0.1   0.1
TOE          0    -0.3     0   0.1  -0.1     0   0.1     -     0
PEAK      -0.2       0   0.2     0  -0.1   0.4   0.1     0     -
> traits.c <- cor(traits, method="kendall")
> Topm(traits.c) # shipunov
   Var1   Var2      Value Magnitude
1   CHIN TONGUE -0.6264145      high
2 THUMB TONGUE  0.5393599      high
3 THUMB   CHIN -0.4853627    medium
```

We will visualize correlation with `Pleiad()`, one of advantages of it is to show which correlations are connected, grouped—so-called "correlation pleiads":

```
> Pleiad(traits.c, corr=TRUE, lcol=1, legend=FALSE, off=1.12,
+ pch=21, bg="white", cex=1.1) # shipunov
```

(Look on the title page to see correlations. One pleiad, `CHIN`, `TONGUE` and `THUMB` is the most apparent.)

* * *

Answer to the question of the linear dependence between height and weight for the artificial data. Correlation is present but the dependence is weak (Fig. 6.15):

```
> cor.test(hwc$WEIGHT, hwc$HEIGHT)

Pearson's product-moment correlation

data:  hwc$WEIGHT and hwc$HEIGHT
t = 5.0682, df = 88, p-value = 2.199e-06
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 0.2975308 0.6212688
sample estimates:
      cor
0.4753337
> w.h <- lm(WEIGHT ~ HEIGHT, data=hwc)
> summary(w.h)

Call:
lm(formula = WEIGHT ~ HEIGHT, data=hwc)

Residuals:
   Min     1Q Median     3Q    Max
-7.966 -2.430  0.305  2.344  5.480

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 30.86387    8.94310   3.451  0.00086 ***
HEIGHT       0.27707    0.05467   5.068  2.2e-06 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 3.27 on 88 degrees of freedom
Multiple R-squared:  0.2259,Adjusted R-squared:  0.2171
F-statistic: 25.69 on 1 and 88 DF,  p-value: 2.199e-06

> plot(WEIGHT ~ HEIGHT, data=hwc, xlab="Height, cm",
+ ylab="Weight, kg")
> abline(w.h)
> Cladd(w.h, data=hwc) # shipunov
```

The conclusion about weak dependence was made because of low R-squared which means that predictor variable, height, does not explain much of the dependent variable, weight. In addition, many residuals are located outside of IQR. This is also easy

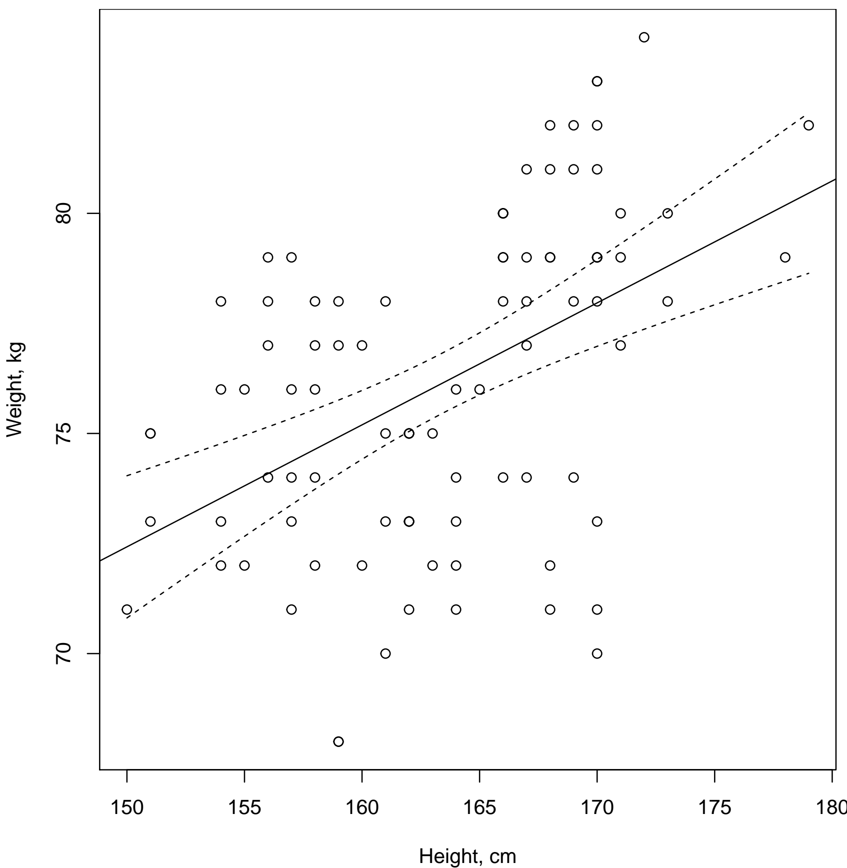


Figure 6.15: The dependence of weight from height (artificial data)

to see on the plot where many data points are distant from the regression line and even from 95% confidence bands.

* * *

Answer to spring draba question. Check file, load and check the object:

```
> ee <- read.table(
+ "http://ashipunov.me/shipunov/open/erophila.txt", h=TRUE)
> Str(ee) # shipunov
 'data.frame': 111 obs. of  5 variables:
  1  LOC       : int  1 1 1 1 1 1 1 1 1 1 ...
  2  FRUIT.L   : num  4.8 5.1 4.9 4.7 4.7 5 4.7 4.8 5.5 4.5 ...
```

```
  3  FRUIT.W   : num  1.8 2 2.3 2.1 2.4 2.8 2.8 2.8 2.8 1.8 ...
  4  FRUIT.MAXW: num  3.5 3 3 2.5 2.8 3.3 1.5 2 3 2.5 ...
  5  FRUIT.A   : int  2 2 2 2 2 1 0 0 2 2 ...
```

Now, check normality and correlations with the appropriate method:

```
> sapply(ee[, 2:4], Normality) # shipunov
     FRUIT.L      FRUIT.W   FRUIT.MAXW
    "NORMAL"     "NORMAL" "NOT NORMAL"
> Topm(cor(ee[, 2:4], method="spearman")) # shipunov
        Var1    Var2     Value Magnitude
1 FRUIT.MAXW FRUIT.L 0.7109781 very high
2    FRUIT.W FRUIT.L 0.4642429    medium
> with(ee, cor.test(FRUIT.L, FRUIT.MAXW, method="spearman"))

Spearman's rank correlation rho

data:  FRUIT.L and FRUIT.MAXW
S = 65874, p-value < 2.2e-16
alternative hypothesis: true rho is not equal to 0
sample estimates:
      rho
0.7109781
```

Therefore, `FRUIT.L` and `FRUIT.MAXW` are best candidates for linear model analysis. We will plot it first (Fig. 6.16):

```
> ee.lm <- lm(FRUIT.MAXW ~ FRUIT.L, data=ee)
> plot(FRUIT.MAXW ~ FRUIT.L, data=ee, type="n")
> Points(ee$FRUIT.L, ee$FRUIT.MAXW, scale=.5) # shipunov
> Cladd(ee.lm, ee, ab.lty=1) # shipunov
```

(`Points()` is a “single” variant of `PPoints()` from the above, and was used because there are multiple overlaid data points.)

Finally, check the linear model and assumptions:

```
> summary(ee.lm)
...

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) -0.14037    0.30475  -0.461    0.646
FRUIT.L      0.59877    0.06091   9.830   <2e-16 ***
---
```

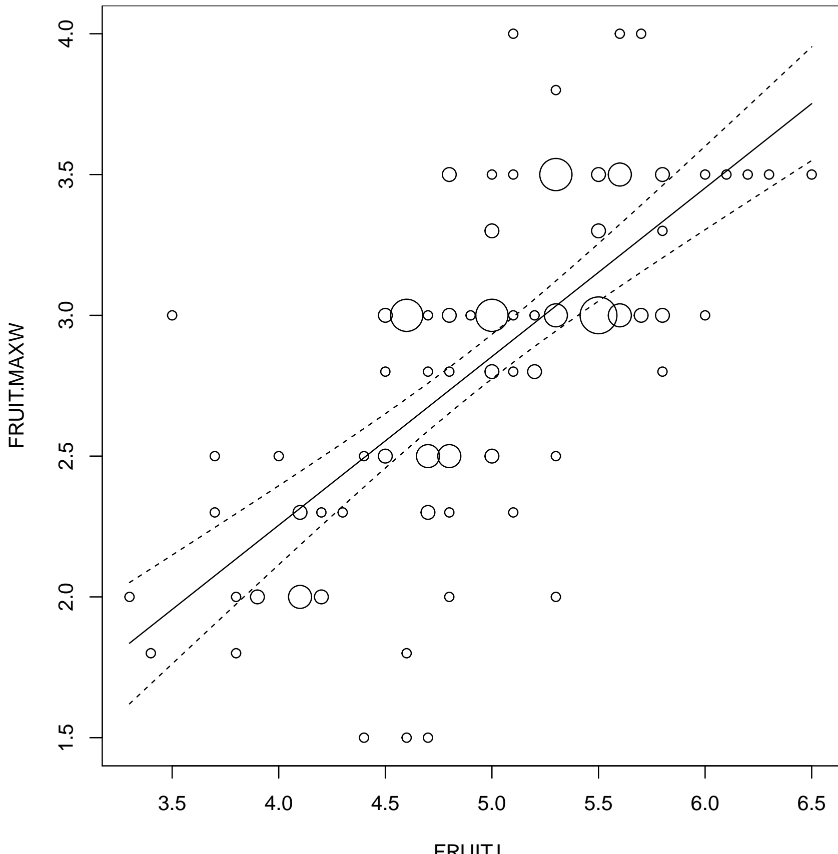


Figure 6.16: Linear relationship between fruit characteristics of spring draba.

```
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 0.4196 on 109 degrees of freedom
Multiple R-squared:  0.4699,Adjusted R-squared:  0.4651
F-statistic: 96.64 on 1 and 109 DF,  p-value: < 2.2e-16

> plot(ee.lm, which=1)
```

There is a reliable model (`p-value: < 2.2e-16`) which has a high R-squared value (`sqrt(0.4651) = 0.6819824`). Slope coefficient is significant whereas intercept is not. Homogeneity of residuals is apparent, their normality is also out of question:

```
> Normality(ee.lm$residuals)
[1] "NORMAL"
```

* * *

Answer to the heterostyly question. First, inspect the file, load the data and check it:

```
> he <- read.table(
+ "http://ashipunov.me/shipunov/open/heterostyly.txt", h=TRUE)
> he$SPECIES <- as.factor(he$SPECIES)
> str(he)
'data.frame': 993 obs. of  6 variables:
...
```

This is how to visualize the phenomenon of heterostyly for all data:

```
> boxplot((STYLE.L-STAMEN.L) ~ (STYLE.L-STAMEN.L)>0,
+ names=c("short","long"), data=he)
```

(Please **review** this plot yourself.)

Now we need to visualize linear relationships of question. There are many overlaid data points so the best way is to employ the `PPoints()` function (Fig. 6.17):

```
> plot(STYLE.L ~ STAMEN.L, data=he, type="n",
+ xlab="Length of stamens, mm", ylab="Length of style, mm")
> PPoints(he$SPECIES, he$STAMEN.L, he$STYLE.L, scale=.9, cols=1)
> abline(lm(STYLE.L ~ STAMEN.L, data=he[he$SPECIES=="veris", ]))
> abline(lm(STYLE.L ~ STAMEN.L, data=he[he$SPECIES=="vulgaris", ]),
+ lty=2)
> legend("topright", pch=1:2, lty=1:2, legend=c("Primula veris",
+ "P. vulgaris"), text.font=3)
```

Now to the models. We will assume that length of stamens is the independent variable. Explore, check assumptions and AIC for the full model:

```
> summary(he.lm1 <- lm(STYLE.L ~ STAMEN.L * SPECIES, data=he))
...
Coefficients:
                          Estimate Std. Error t value Pr(>|t|)
(Intercept)               19.13735    1.17900  16.232  < 2e-16 ***
STAMEN.L                  -0.74519    0.09137  -8.156 1.05e-15 ***
SPECIESvulgaris           -1.84688    1.23060  -1.501   0.1337
STAMEN.L:SPECIESvulgaris   0.24272    0.09509   2.552   0.0108 *
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

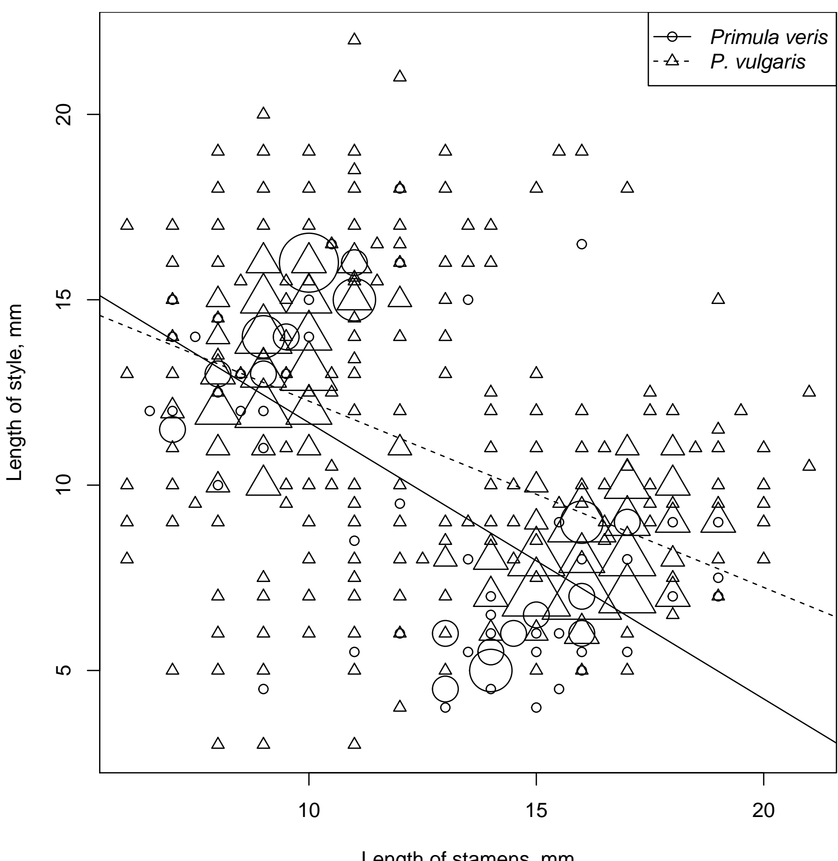


Figure 6.17: Linear relationships within flowers of two primrose species. Heterostyly is visible as two dense "clouds" of data points.

```
Residual standard error: 2.921 on 989 degrees of freedom
Multiple R-squared:  0.3077,Adjusted R-squared:  0.3056
F-statistic: 146.5 on 3 and 989 DF,  p-value: < 2.2e-16

> plot(he.lm1, which=1:2)
> AIC(he.lm1)
[1] 4953.172
```

Reduced (additive) model:

```
> summary(he.lm2 <- update(he.lm1, . ~ . - STAMEN.L:SPECIES))
...
Coefficients:
```

```
                Estimate Std. Error t value Pr(>|t|)
(Intercept)     16.34607    0.44188  36.992  < 2e-16 ***
STAMEN.L        -0.52109    0.02538 -20.536  < 2e-16 ***
SPECIESvulgaris  1.18400    0.32400   3.654 0.000271 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 2.93 on 990 degrees of freedom
Multiple R-squared:  0.3031,Adjusted R-squared:  0.3017
F-statistic: 215.3 on 2 and 990 DF,  p-value: < 2.2e-16

> AIC(he.lm2)
[1] 4957.692
```

Full model is better, most likely because of strong interactions. To check interactions graphically is possible also with the *interaction plot* which will treat independent variable as factor:

```
> with(he, interaction.plot(cut(STAMEN.L, quantile(STAMEN.L)),
+ SPECIES, STYLE.L))
```

This technical plot (**check it** yourself) shows the reliable differences between lines of different species. This differences are bigger when stamens are longer. This plot is more suitable for the complex ANOVA but as you see, works also for linear models.

* * *

Answer to the question about sundew (*Drosera*) populations. First, inspect the file, then load it and check the structure of object:

```
> Str(drosera) # shipunov
 'data.frame': 1165 obs. of  11 variables:
  1  POP     : Factor w/ 19 levels "A","B","C","D",..: 1 1 1 ...
  2  YOUNG.L * int  0 0 0 1 0 1 0 1 0 1 ...
  3  MATURE.L* int  3 2 5 5 3 5 4 4 3 3 ...
  4  OLD.L   * int  3 2 5 2 1 2 1 1 3 2 ...
  5  INSECTS * int  3 2 10 5 0 5 7 3 1 1 ...
  6  INFL.L  * int  0 0 128 0 0 0 0 0 0 0 ...
  7  STALK.L * int  0 0 100 0 0 0 0 0 0 0 ...
  8  N.FLOW  * int  0 0 3 0 0 0 0 0 0 0 ...
  9  LEAF.L  * num  4 4 6 5 3 5 6 4 3 4 ...
 10  LEAF.W  * num  4 4 6 6 4 6 7 5 4 5 ...
 11  PET.L   * int  10 7 16 15 4 5 13 14 11 10 ...
```

Since we a required to calculate correlation, check the normality first:

```
> sapply(drosera[, -1], Normality)
     YOUNG.L      MATURE.L         OLD.L       INSECTS
"NOT NORMAL" "NOT NORMAL" "NOT NORMAL" "NOT NORMAL"
       INFL.L       STALK.L
 "NOT NORMAL" "NOT NORMAL"
      N.FLOW        LEAF.L        LEAF.W         PET.L
"NOT NORMAL" "NOT NORMAL" "NOT NORMAL" "NOT NORMAL"
```

Well, to this data we can apply only nonparametric methods:

```
> dr.cor <- cor(drosera[, -1], method="spearman", use="pairwise")
> Topm(dr.cor) # shipunov
     Var1     Var2     Value Magnitude
1 STALK.L   INFL.L 0.9901613 very high
2  N.FLOW  STALK.L 0.9774198 very high
3  N.FLOW   INFL.L 0.9593589 very high
4  LEAF.W   LEAF.L 0.8251841 very high
5   PET.L   LEAF.W 0.7129433 very high
6   PET.L   LEAF.L 0.6972402      high
7   PET.L   INFL.L 0.4795218    medium
8   PET.L  STALK.L 0.4661593    medium
9 INSECTS MATURE.L 0.4644699    medium

> Pleiad(dr.cor, corr=TRUE, legtext=2, legpos="bottom",
+ leghoriz=TRUE, pch=19, cex=1.2) # shipunov
```

(Note that "pairwise" was employed, there are many NAs.)

The last plot (Fig. 6.18) shows two most important correlation pleiads: one related with leaf size, and another—with inflorescence.

Since we know now which characters are most correlated, proceed to linear model. Since in the development of sundews stalk formed first, let us accept `STALK.L` as independent variable (influence), and `INFL.L` as dependent variable (response):

```
> summary(dr.lm <- lm(INFL.L ~ STALK.L, data=drosera))
...
Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) -0.046294   0.212065  -0.218    0.827
STALK.L      1.139452   0.004637 245.719   <2e-16 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

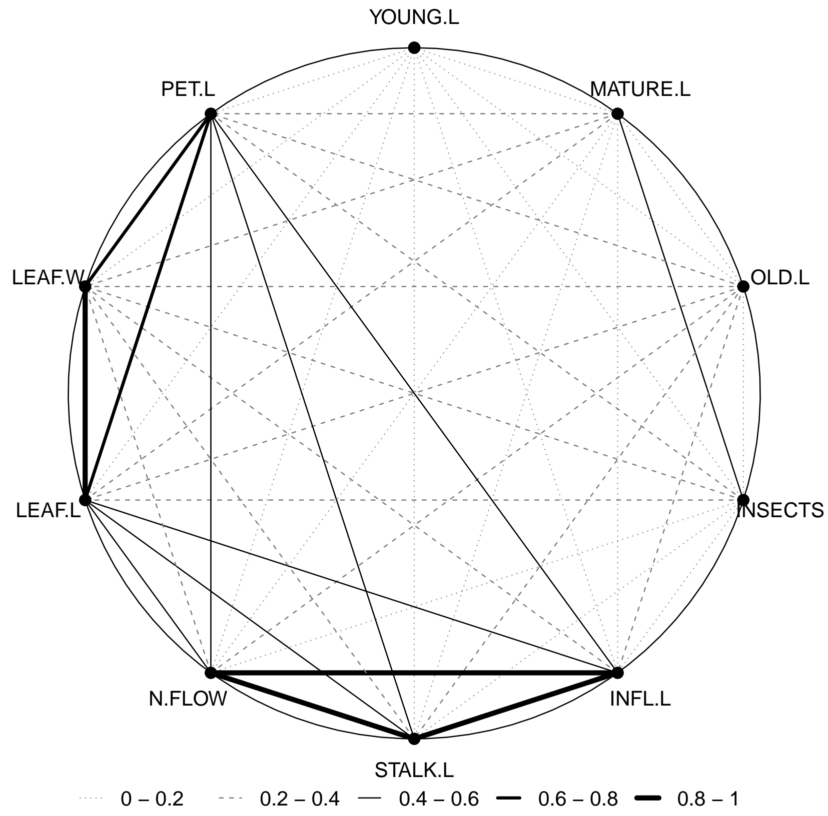


Figure 6.18: Correlations in sundew data.

```
Residual standard error: 6.285 on 1158 degrees of freedom
  (5 observations deleted due to missingness)
Multiple R-squared:  0.9812,Adjusted R-squared:  0.9812
F-statistic: 6.038e+04 on 1 and 1158 DF,  p-value: < 2.2e-16

> plot(dr.lm, which=1:2)
```

Reliable model with high R-squared. However, normality of residuals is not perfect (please **check** model plots yourself).

Now to the analysis of leaf length. Determine which three populations are largest and subset the data:

```
> (largest3 <- rev(sort(table(drosera[, 1])))[1:3])
 Q1   L  N1
211 201 144
> dr3 <- drosera[drosera$POP %in% names(largest3), ]
> dr3$POP <- droplevels(dr3$POP)
```

Now we need to plot them and check if there are visual differences:

```
> boxplot(LEAF.L ~ POP, data=dr3)
```

Yes, they probably exist (please **check** the plot yourself.)

It is worth to look on similarity of ranges:

```
> tapply(dr3$LEAF.L, dr3$POP, mad, na.rm=TRUE)
     L     N1     Q1
1.4826 1.4826 1.4826
> fligner.test(LEAF.L ~ POP, data=dr3)
Fligner-Killeen test of homogeneity of variances
data:  LEAF.L by POP
Fligner-Killeen:med chi-squared = 8.1408, df = 2, p-value = 0.01707
```

The robust range statistic, MAD (median absolute deviation) shows that variations are similar. We also ran the nonparametric analog of Bartlett test to see the statistical significance of this similarity. Yes, variances are statistically similar.

Since we have three populations to analyze, we will need something ANOVA-like, but nonparametric:

```
> kruskal.test(LEAF.L ~ POP, data=dr3)
Kruskal-Wallis rank sum test
data:  LEAF.L by POP
Kruskal-Wallis chi-squared = 97.356, df = 2, p-value < 2.2e-16
```

Yes, there is at least one population where leaf length is different from all others. To see which, we need a *post hoc*, pairwise test:

```
> pairwise.wilcox.test(dr3$LEAF.L, dr3$POP)
Pairwise comparisons using Wilcoxon rank sum test
data:  dr3$LEAF.L and dr3$POP
   L       N1
N1 5.2e-16 -
Q1 0.74    < 2e-16
P value adjustment method: holm
```

Population N1 is most divergent whereas Q1 is not really different from L.

6.4.2 Logistic regression

Answer to the question about demonstration of objects. We will go the same way as in the example about programmers. After loading data, we attach it for simplicity:

```
> seeing <- read.table("data/seeing.txt")
> attach(seeing)
```

Check the model:

```
> seeing.logit <- glm(V3 ~ V2, family=binomial, data=seeing)
> summary(seeing.logit)

Call:
glm(formula = V3 ~ V2, family = binomial)

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-2.4029  -0.8701   0.4299   0.7825   1.5197

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept)  -1.6776     0.7923  -2.117  0.03423 *
V2            0.9015     0.2922   3.085  0.00203 **
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 62.687  on 49  degrees of freedom
Residual deviance: 49.738  on 48  degrees of freedom
AIC: 53.738

Number of Fisher Scoring iterations: 4
```

(Calling variables, we took into account the fact that `R` assign names like `V1`, `V2`, `V3` *etc.* to “anonymous” columns.)

As one can see, the model is significant. It means that some learning takes place within the experiment.

It is possible to represent the logistic model graphically (Fig. 6.19):

```
> tries <- seq(1, 5, length=50) # exactly 50 numbers from 1 to 5
> seeing.p <- predict(seeing.logit, list(V2=tries),
```

```
+ type="response")
> plot(V3 ~ jitter(V2, amount=.1), data=seeing, xlab="", ylab="")
> lines(tries, seeing.p)
```

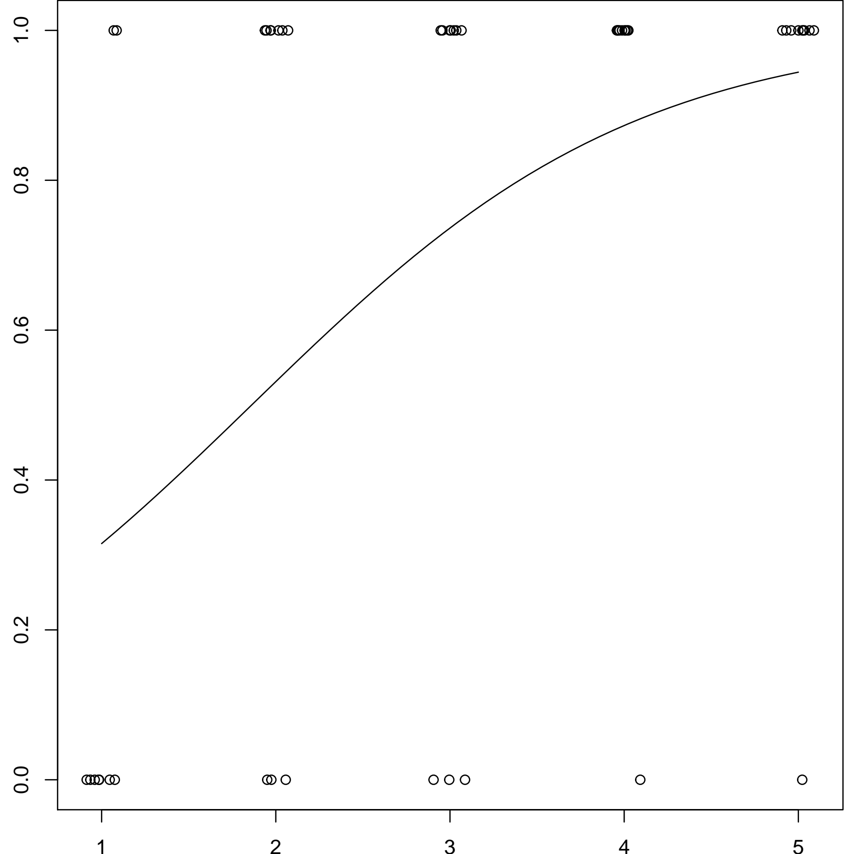


Figure 6.19: Graphical representation of the logistic model.

We used `predict()` function to calculate probabilities of success for non-existed attempts, and also added small random noise with function `jitter()` to avoid the overlap.

* * *

Answer to the juniper questions. Check file, load it, check the object:

```
> jj <- read.table(
+ "http://ashipunov.me/shipunov/open/juniperus.txt", h=TRUE)
```

```
> jj$LOC <- factor(jj$LOC, labels=paste0("loc", levels(jj$LOC)))
> Str(jj) # shipunov
 'data.frame': 61 obs. of  7 variables:
  1  LOC     : Factor w/ 3 levels "loc1","loc2",..: 1 1 1 ...
  2  HEIGHT  : num  90 55 20 80 80 65 25 40 55 40 ...
  3  WIDTH   : num  40 25 45 100 135 35 55 25 45 55 ...
  4  NEEDLE.L: int  8 8 5 6 10 6 6 9 5 5 ...
  5  PROTR   : num  1 1 1.5 1 0 1 0 0.5 1 1 ...
  6  STEM.D  * num  2.4 2.3 3.5 6 2.6 4.5 3.2 0.5 NA 1.7 ...
  7  PINE.N  : int  1 0 2 2 0 2 3 2 0 3 ...
```

Analyze morphological and ecological characters graphically (Fig. 6.20):

```
> j.cols <- colorRampPalette(c("steelblue", "white"))(5)[2:4]
> Boxplots(jj[, 2:7], jj$LOC, legpos="top",
+ boxcols=j.cols) # shipunov
```

Now plot length of needles against location (Fig. 6.21):

```
> spineplot(LOC ~ NEEDLE.L, data=jj, col=j.cols)
```

(As you see, spine plot works with measurement data.)

Since there is a measurement character and several locations, the most appropriate is ANOVA-like approach. We need to check assumptions first:

```
> Normality(jj$NEEDLE.L)
[1] "NORMAL"
> tapply(jj$NEEDLE.L, jj$LOC, var)
    loc1     loc2     loc3
2.461905 3.607895 2.407895
> bartlett.test(NEEDLE.L ~ LOC, data=jj)

Bartlett test of homogeneity of variances

data:  NEEDLE.L by LOC
Bartlett's K-squared = 1.0055, df = 2, p-value = 0.6049
```

Since variation is not homogeneous, one-way test with *post hoc* pairwise t-test is the best:

```
> oneway.test(NEEDLE.L ~ LOC, data=jj)

One-way analysis of means (not assuming equal variances)

data:  NEEDLE.L and LOC
```

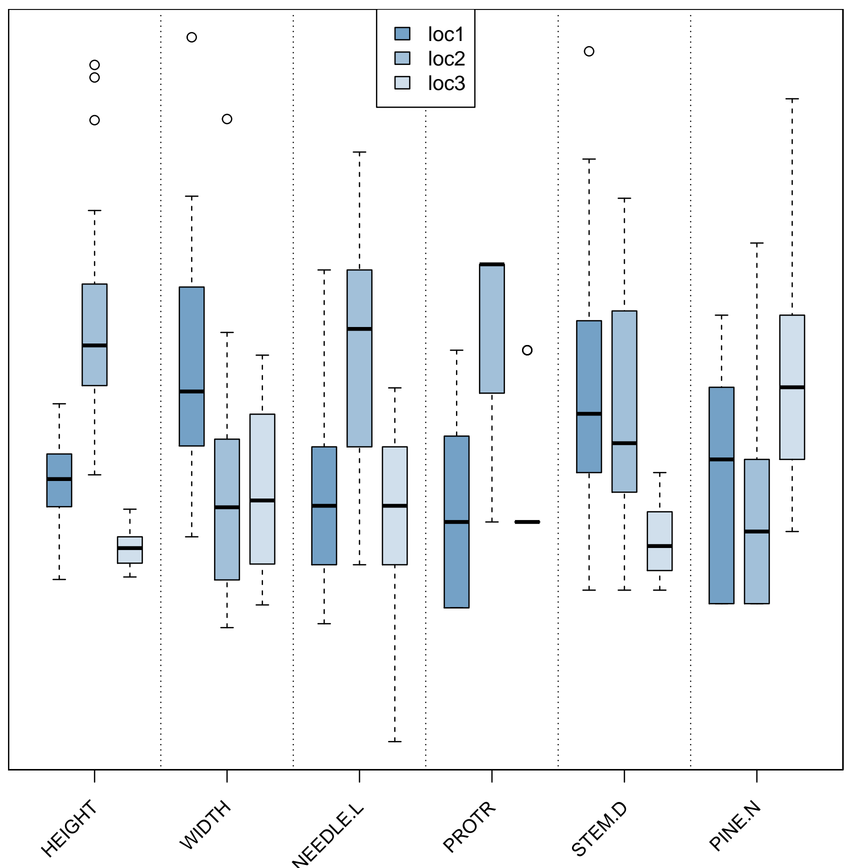


Figure 6.20: Boxplots show distribution of measurements among juniper populations.

```
F = 14.129, num df = 2.000, denom df = 38.232, p-value = 2.546e-05

> (eta.squared <-
+ summary(lm(NEEDLE.L ~ LOC, data=jj))$adj.r.squared)
[1] 0.3337755
> pairwise.t.test(jj$NEEDLE.L, jj$LOC)

Pairwise comparisons using t tests with pooled SD

data:  jj$NEEDLE.L and jj$LOC

     loc1    loc2
```

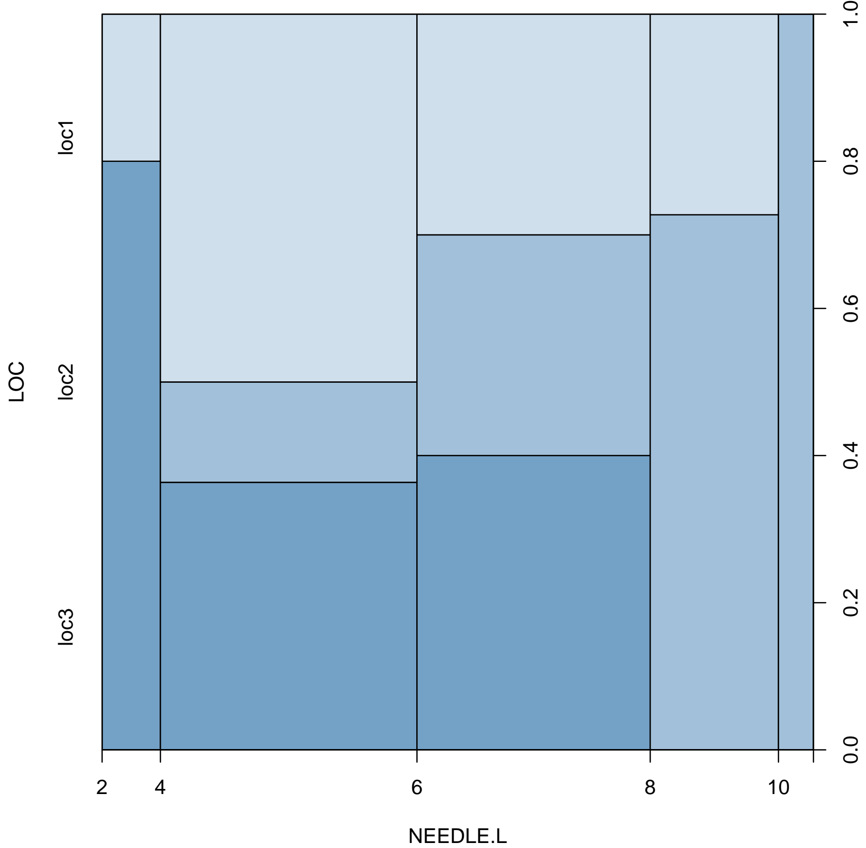


Figure 6.21: Spine plot: locality *vs.* needle length of junipers.

```
loc2 0.00031 -
loc3 0.14564 3.1e-06

P value adjustment method: holm
```

(Note how we calculated eta-squared, the effect size of ANOVA. As you see, this could be done through linear model.)

There is significant difference between the second and two other locations.

And to the second problem. First, we make new variable based on *logical expression* of character differences:

```
> is.sibirica <- with(jj, (NEEDLE.L < 8 & HEIGHT < 100))
> sibirica <- factor(is.sibirica, labels=c("communis", "sibirica"))
```

```
> summary(sibirica)
communis sibirica
      24       37
```

There are both “species” in the data. Now, we plot conditional density and analyze logistic regression:

```
> cdplot(sibirica ~ PINE.N, data=jj, col=j.cols[c(1, 3)])
> summary(glm(sibirica ~ PINE.N, data=jj, family=binomial))

Call:
glm(formula = sibirica ~ PINE.N, family = binomial, data=jj)

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-1.8549  -1.0482   0.7397   1.0042   1.3123

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept)  -0.3117     0.4352  -0.716   0.4738
PINE.N        0.3670     0.1776   2.066   0.0388 *
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 81.772  on 60  degrees of freedom
Residual deviance: 77.008  on 59  degrees of freedom
AIC: 81.008

Number of Fisher Scoring iterations: 4
```

Conditional density plot (Fig. 6.22) shows an apparent tendency, and model summary outputs significance for slope coefficient.

6.5 How to choose the right method

On the next page, there is a table (Table 6.1) with a key which could help to choose the right inferential method if you know number of samples and type of the data.

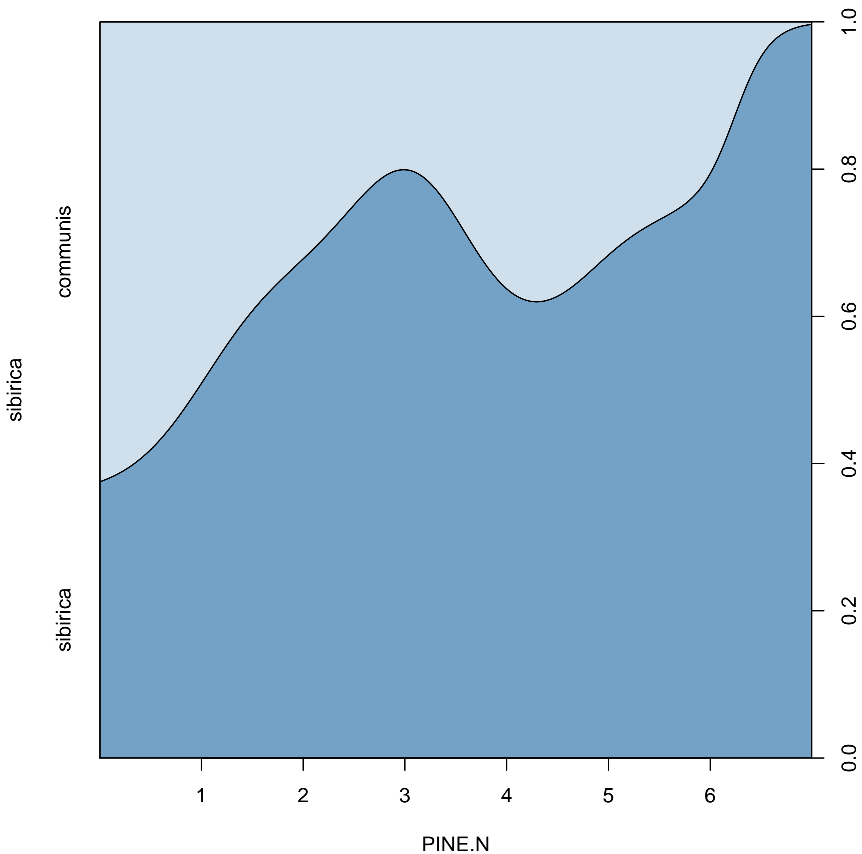


Figure 6.22: Conditional density of being *Juniperus sibirica* on the presence of some pine trees.

Type of data	One variable	Two variables	Many variables
Measurement, normally distributed	t-test	*Difference*: t-test (paired and non-paired), F-test (scale) *Effect size*: Cohen's d, Lyubishchev's K *Relation*: correlation, linear models	Linear models, ANOVA, one-way test, Bartlett test (scale) *Post hoc*: pairwise t-test, Tukey HSD *Effect size*: R-squared
Measurement and ranked	Wilcoxon test, Shapiro-Wilk test	*Difference*: Wilcoxon test (paired and non-paired), sign test, robust rank order test, Ansari-Bradley test (scale) *Effect size*: Cliff's delta, Lyubishchev's K *Relation*: nonparametric correlation	Linear models, LOESS, Kruskal-Wallis test, Friedman test, Fligner-Killeen test (scale) *Post hoc*: pairwise Wilcoxon test, pairwise robust rank order test *Effect size*: R-squared
Categorical	One sample test of proportions, goodness-of-fit test	*Association*: Chi-squared test, Fisher's exact test, test of proportions, G-test, McNemar's test (paired) *Effect size*: Cramér's V, Tschuprow's T, odds ratio	*Association* tests (see on the left); generalized linear models of binomial family (= logistic regression) *Post hoc*: pairwise table test

Table 6.1: Key to the most important inferential statistical methods (except multivariate). After you narrow the search with couple of methods, proceed to the main text.

Part II

Many dimensions

Chapter 7

Draw

“Data Mining”, “Big Data”, “Machine Learning”, “Pattern Recognition” phrases often mean all statistical methods, analytical and visual which help to understand the structure of data.

Data might be of any kind, but it is usually *multidimensional*, which is best represented with the table of multiple columns a.k.a. variables, or features (which might be of different types: measurement, ranked or categorical) and rows a.k.a objects. So more traditional name for these methods is “multivariate data analysis” or “multivariate statistics”.

Data mining is often based on the idea of *classification*, arranging objects into non-intersecting, and possibly hierarchical groups.

We use classification all the time (but sometimes do not realize it). We open the door and enter the room, the first thing is to recognize (classify) what is inside. Our brain has the outstanding power of classification, but computers and software are speedily advancing and becoming more brain-like. This is why data mining is related with artificial intelligence.

* * *

In the following chapters we will frequently use the embedded `iris` data taken from works of Ronald Fisher[1]. There are four characters measured on three species of irises (Fig. 7.1), and the fifth column is the species name.

[1] Fisher R.A. 1936. The use of multiple measurements in taxonomic problems. Annals of Eugenics. 7(2): 179–188.

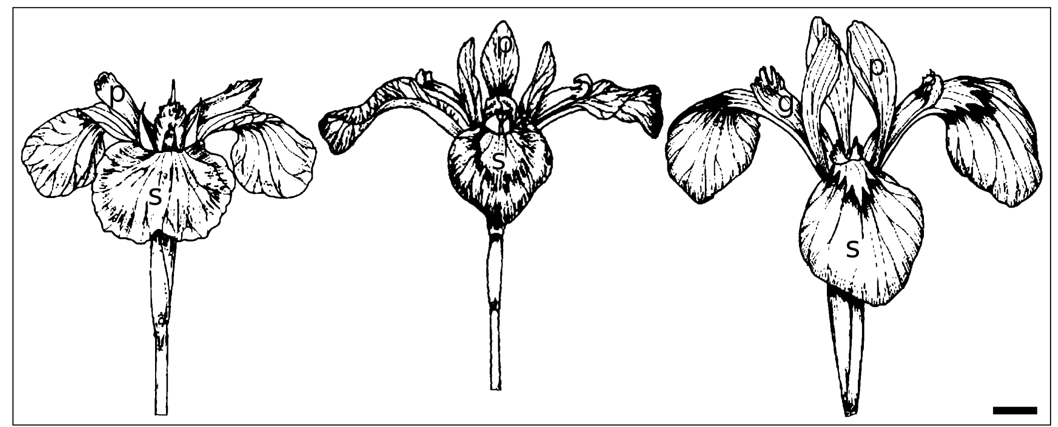


Figure 7.1: Flowers of irises from the `iris` data (from left to right): *Iris setosa* Pall., *I. versicolor* L. and *I. virginica* L. Sepals labeled with *s*, petals with *p*; there are also petal-like stigma lobes (labeled once with *g*). Scale bar is approximately 10 mm.

* * *

The simplest way to work with multidimensional data is to draw it. This chapter discusses how.

7.1 Pictographs

Pictograph is a plot where each element represents one of objects, and every feature of the element corresponds with one character of the primary object. *If the every row of data is unique*, pictographs might be useful. Here is the *star plot* (Fig. 7.2) example:

```
> eq8 <- read.table("data/eq8.txt", h=TRUE)
> Str(eq8) # shipunov
 'data.frame': 832 obs. of  9 variables:
  1  SPECIES * Factor w/ 8 levels "arvense","fluviatile",..: 1 ...
  2  DL.R    : num  424 339 321 509 462 350 405 615 507 330 ...
  3  DIA.ST  * num  2.3 2 2.5 3 2.5 1.8 2 2.2 2 1.8 ...
  4  N.REB   : int  13 11 15 14 12 9 14 11 10 8 ...
  5  N.ZUB   : int  12 12 14 14 13 9 15 11 10 8 ...
  6  DL.OSN.Z* num  2 1 2 1.5 1.1 1.1 1 1 1 1 ...
  7  DL.TR.V * num  5 4 5 5 4 4 4 4 5 5 ...
  8  DL.BAZ  : num  3 2.5 2.3 2.2 2.1 2 2 2 1.9 1 ...
  9  DL.PER  : num  25 13 13 23 12 15 13 14 10 9 ...
> eq8m <- Aggregate1(eq8[, 2:9], eq8[, 1],
```

```
+  median, na.rm=TRUE) # shipunov
> stars(eq8m, cex=1.2, lwd=1.2, col.stars=rep("darkseagreen", 8))
```

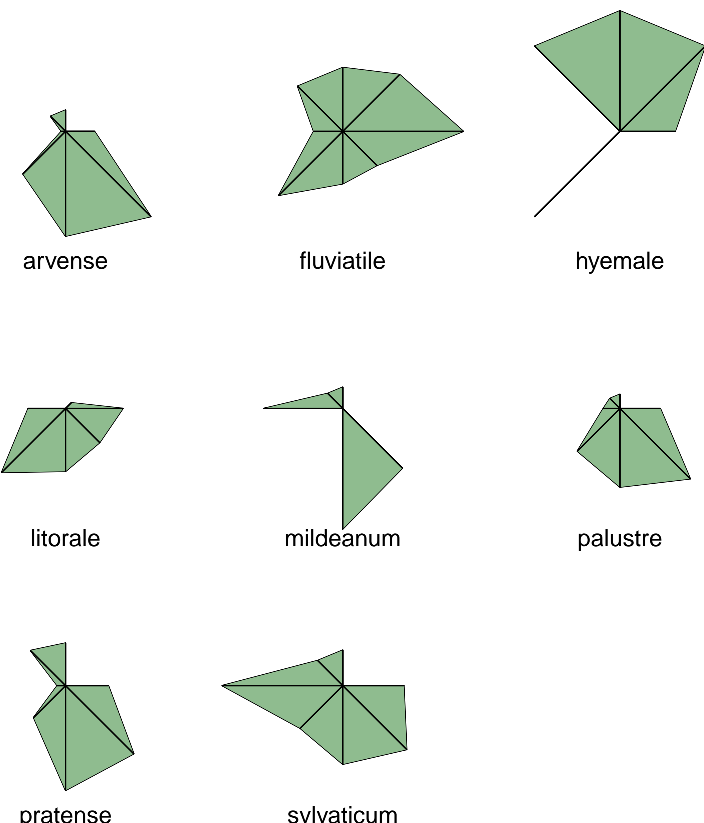


Figure 7.2: Stars show different horsetail species.

(We made each element to represent the species of horsetails. Length of the particular ray corresponds with some morphological character. It is easy to see, as an example, similarities between *Equisetum* $\times$*litorale* and *E. fluviatile*.)

Slightly more exotic pictograph is *Chernoff's faces* where features of elements are shown as human face characters (Fig. 7.3):

```
> library(TeachingDemos)
> faces(eq8m)
```

(Original Chernoff's faces have been implemented in the `faces2()` function, there is also another variant in `symbols()` package.)

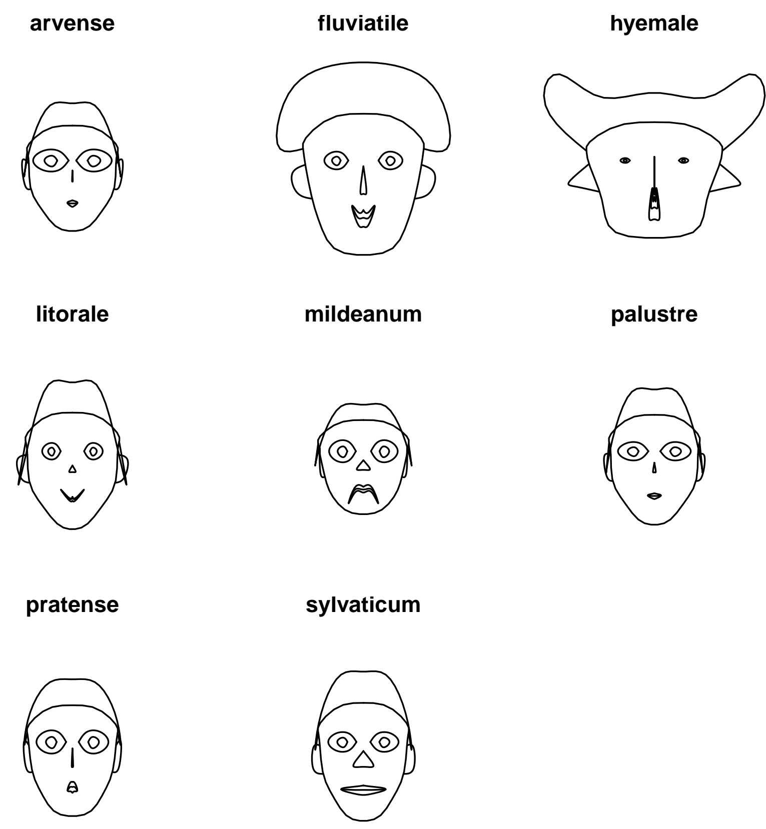


Figure 7.3: Chernoff's faces show different horsetail species.

* * *

Related to pictographs are ways to overview the *whole* numeric dataset, matrix or data frame. First, `image()` allows for plots like on Fig. 7.4:

```
> image(scale(iris[, -5]), axes=FALSE)
> axis(2, at=seq(0, 1, length.out=4),
+ labels=abbreviate(colnames(iris[, -5])), las=2)
```

(This is a "portrait" or iris matrix, useful in many ways. For example, it is well visible that highest, most red, values of `Pt.L` (abbreviated from `Petal.Length`) correspond with lowest values of `Sp.W` (`Sepal.Width`).)

More advanced is the *parallel coordinates plot* (Fig. 7.5):

```
> library(MASS)
```

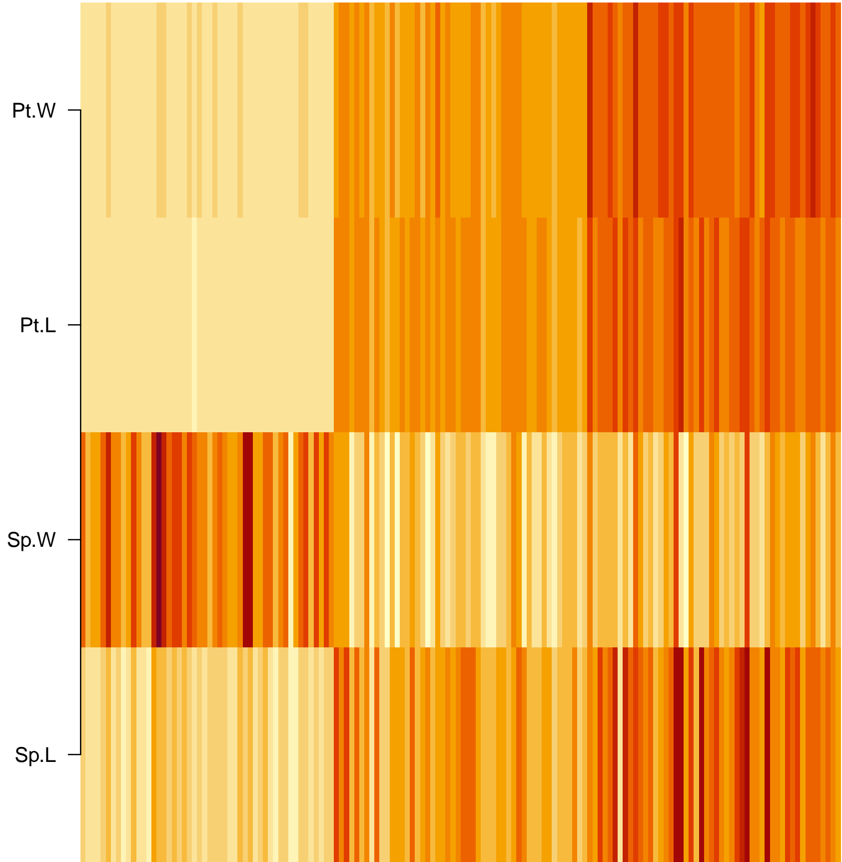


Figure 7.4: Results of plotting `iris` data with the `image()` command. Redder colors correspond with higher values of scaled characters.

```
> parcoord(iris[, -5], col=as.numeric(iris[, 5]), lwd=2)
> legend("top", bty="n", lty=1, lwd=2, col=1:3,
+ legend=names(table(iris[, 5])))
```

This is somewhat like the multidimensional stripchart. Each character represented with one axis. Then, for every plant, these values are connected with lines. It is easy to see, for example, that petal characters are more distinguishing than sepal. It is also visible well that *Iris setosa* is more distinct from two other species.

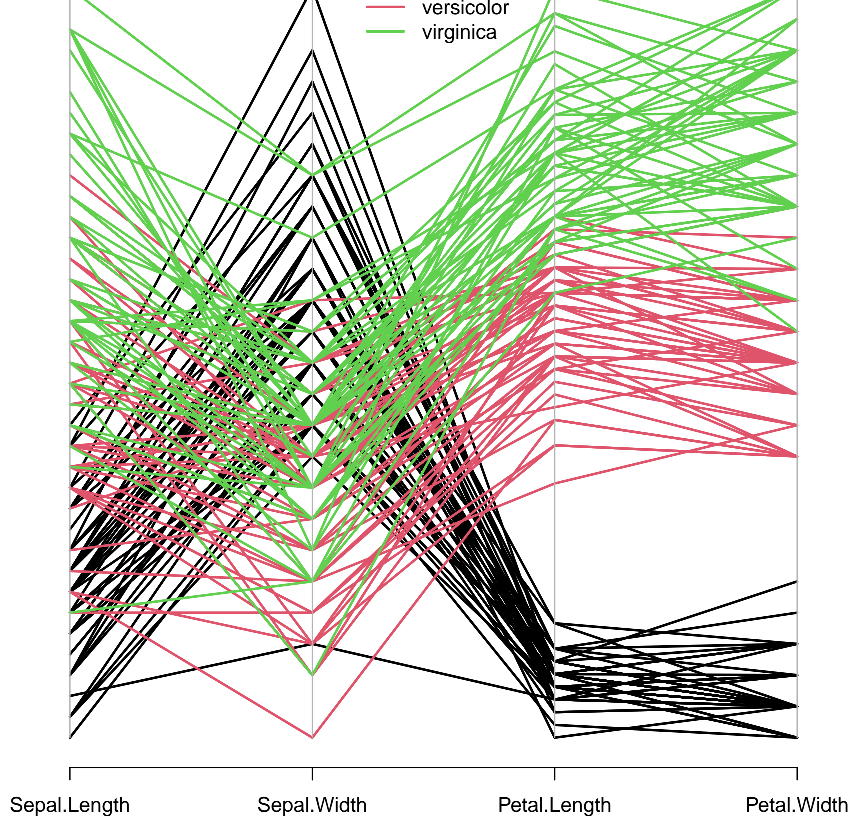


Figure 7.5: Parallel coordinates plot.

7.2 Grouped plots

Even boxplots and dotcharts could represent multiple characters of multiple groups, but you will need either to scale them first and then manually control positions of plotted elements, or use `Boxplots()` and `Linechart()` described in the previous chapter:

```
> Boxplots(iris[, 1:4], iris[, 5], srt=0, adj=c(.5, 1),
+ legpos="topright") # shipunov
> Linechart(iris[, 1:4], iris[, 5], mad=TRUE) # shipunov
```

(Please **make** these plots yourself.)

Function `matplot()` allows to place multiple scatterplots in one frame:

```
> matplot(iris[, 1:2], iris[, 3:4], pch=1,
+ xlab="Sepals", ylab="Petals")
> legend("topleft", pch=1, col=1:2, legend=c("Length", "Width"))
```

(Please **make** this plot yourself.)

Function `symbols()` places multiple smaller plots in desired locations, and function `pairs()` shows multiple scatterplots as a matrix (Fig. 7.6).

```
> pairs(iris[, 1:4], pch=19, col=as.numeric(iris[, 5]),
+ oma=c(2, 2, 4, 2))
> oldpar <- par(xpd=TRUE)
> legend(0, 1.06, horiz=TRUE, legend=levels(iris[, 5]),
+ pch=19, col=1:3, bty="n")
> par(oldpar)
```

(This matrix plot shows dependencies between each possible pair of five variables simultaneously. Most commands from above help to place the legend.)

Matrix plot is just one of the big variety of R *trellis* plots. Many of them are in the `lattice` package (Fig. 7.7):

```
> betula <- read.table(
+ "http://ashipunov.me/shipunov/open/betula.txt", h=TRUE)
> library(lattice)
> d.tmp <- do.call(make.groups, betula[, c(2:4, 7:8)])
> d.tmp$LOC <- betula$LOC
> bwplot(data ~ factor(LOC) | which, data=d.tmp, ylab="")
```

(Note how to use `make.groups()` and `do.call()` to stack all columns into the long variable (it is also possible to do it with `stack()`, see above). When `LOC` was added to temporary dataset, it was recycled five times—exactly what we need.)

Library `lattice` offers multiple trellis variants of common R plots. For example, one could make the trellis dotchart which will show differences between horsetail species (Fig. 7.8)

```
> eq.s <- stack(as.data.frame(scale(eq8m)))
> eq.s$SPECIES <- row.names(eq8m)
> dotplot(SPECIES ~ values | ind, eq.s, xlab="")
```

(Here we stacked all numerical columns into one with `stack()`.)

Few trellis plots are available in the core R. This is our election data from the previous chapter (Fig. 7.9):

```
> coplot(percn ~ atten | cand, data=elections2, col="red",
```

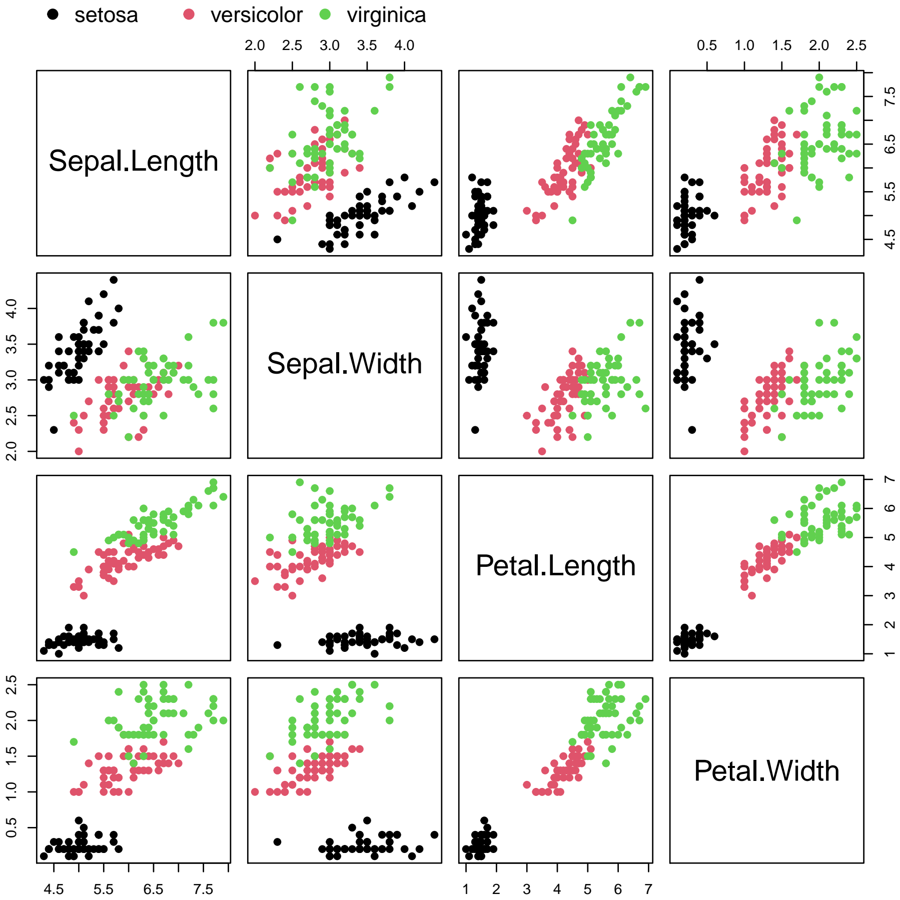


Figure 7.6: Matrix plot.

```
+ bg="pink", pch=21, bar.bg=c(fac="lightblue"))
```

7.3 3D plots

If there just three numerical variables, we can try to plot all of them with 3-axis plots. Frequently seen in geology, metallurgy and some other fields are *ternary plots*. They implemented, for example, in the vcd package. They use triangle coordinate system which allows to reflect simultaneously three measurement variables and some more categorical characters (via colors, point types *etc.*):

```
> library(vcd)
```

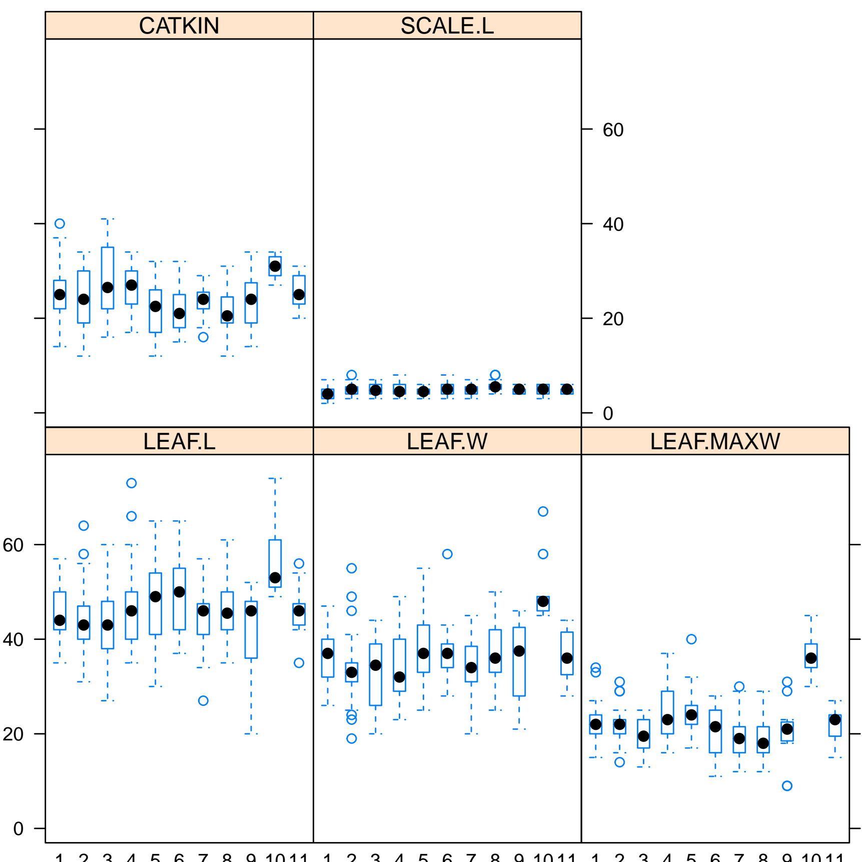


Figure 7.7: The example of trellis plot: for each measurement character, boxplots represent differences between locations.

```
> ternaryplot(scale(iris[, 2:4], center=FALSE), grid_color="black",
+ cex=0.3, col=iris[, 5], main="")
> grid_legend(0.8, 0.7, pch=19, size=.5, col=1:3, levels(iris[, 5]))
```

The "brick" 3D plot could be done, for example, with the package `scatterplot3d` (Fig. 7.11):

```
> library(scatterplot3d)
> i3d <- scatterplot3d(iris[, 2:4], color=as.numeric(iris[, 5]),
+ type="h", pch=14 + as.numeric(iris[, 5]), xlab="Sepal width",
+ ylab="", zlab="Petal width")
> dims <- par("usr")
> x <- dims[1]+ 0.82*diff(dims[1:2])
```

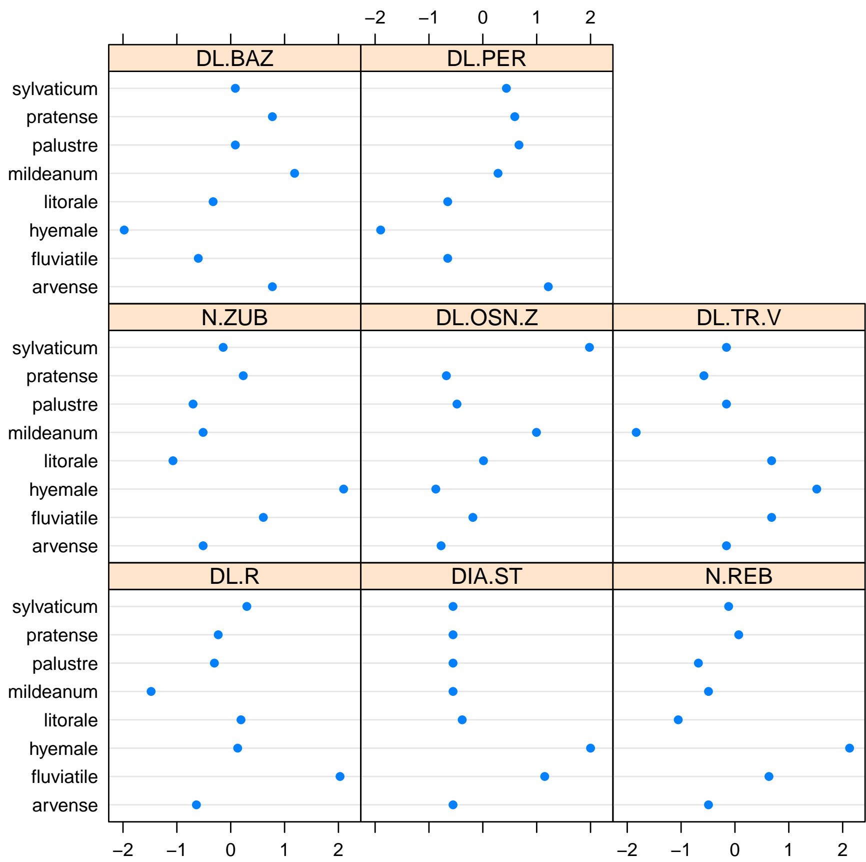


Figure 7.8: Trellis dotchart of the horsetail species (character values are scaled). These plots are typically read from the bottom.

```
> y <- dims[3]+ 0.1*diff(dims[3:4])
> text(x, y, "Petal length", srt=40)
> legend(i3d$xyz.convert(3.8, 6.5, 1.5), col=1:3, pch=(14 + 1:3),
+ legend=levels(iris[, 5]), bg="white")
```

(Here some additional efforts were used to make y-axis label slanted.)

These 3D scatterplots look attractive, but what if some points were hidden from the view? How to rotate and find the best projection? Library RGL will help to create the *dynamic* 3D plot:

```
> library(rgl)
> plot3d(iris[, 1:3], col=as.numeric(iris[, 5]))
```

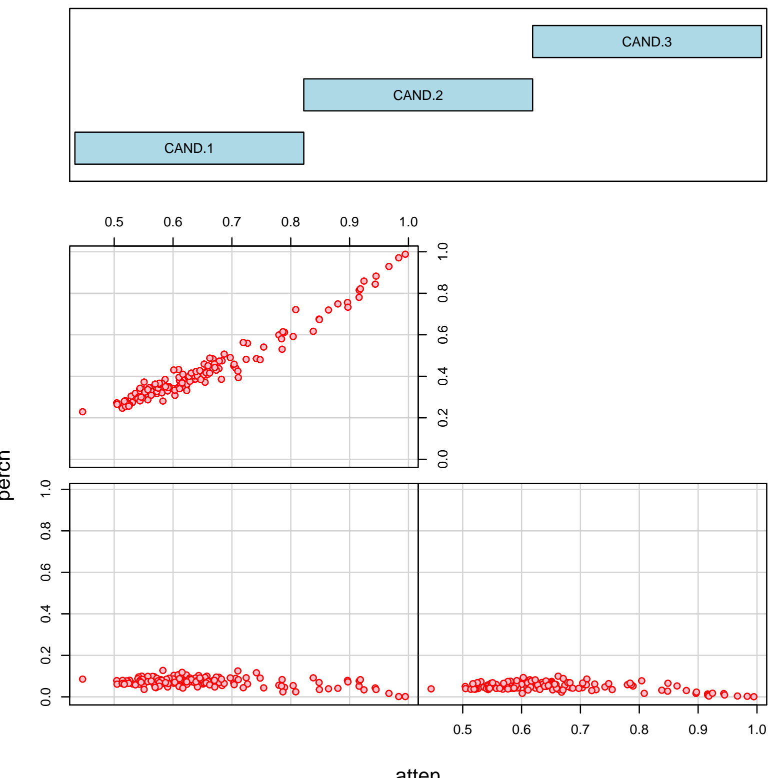


Figure 7.9: Voting data from previous chapter represented with `coplot()` function.

The size of window and projection in RGL plots (Fig. 7.12) are controlled with mouse. Please rotate it, this helps to understand better the position of every point. In case of `iris` data, it is visible clearly that one of the species (*Iris setosa*) is more distinct than two others, and the most "splitting" character is the length of petals (`Petal.Length`). There are *four* characters on the plot, because color was used to distinguish species.

To save current RGL plot, run `rgl.snapshot()` or `rgl.postscript()` function. Please also note that `RGL` package depends on the external `OpenGL` library and therefore on some systems, additional installations might be required.

Another 3D possibility is `cloud()` from `lattice` package. It is a static plot with the relatively heavy code but important is that user can use different rotations (Fig. 7.13):

```
> library(lattice)
```

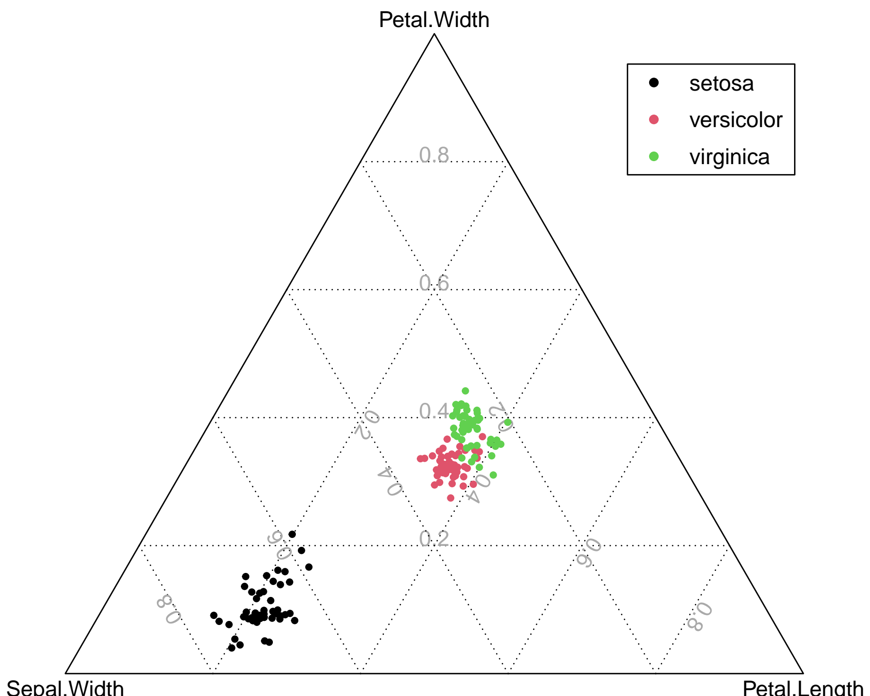


Figure 7.10: Ternary plot for `iris` data.

```
> p <- cloud(Sepal.Length ~ Petal.Length * Petal.Width, data=iris,
+ groups=Species, par.settings=list(clip=list(panel="off")),
+ auto.key=list(space="top", columns=3, points=TRUE))
> update(p[rep(1, 4)], layout=c(2, 2), function(..., screen)
+ panel.cloud(..., screen=list(z=c(-70, 110)[current.column()],
+ x=-70, y=c(140, 0)[current.row()])))
```

Plots similar to the above are also available in the `plot3D` package.

* * *

We see now that plotting multivariate data always has some problems: either there are too many elements (e.g., in parallel coordinates) which are hard to understand, or there is a need of some grouping operation (e.g., median or range) which will result in the lost of information. Also, 3D plots are in fact fake 3D because they must be plotted on 2D devices. Then, some points will frequently hide others, and the best view angle is possible to find only with try-and-error approach.

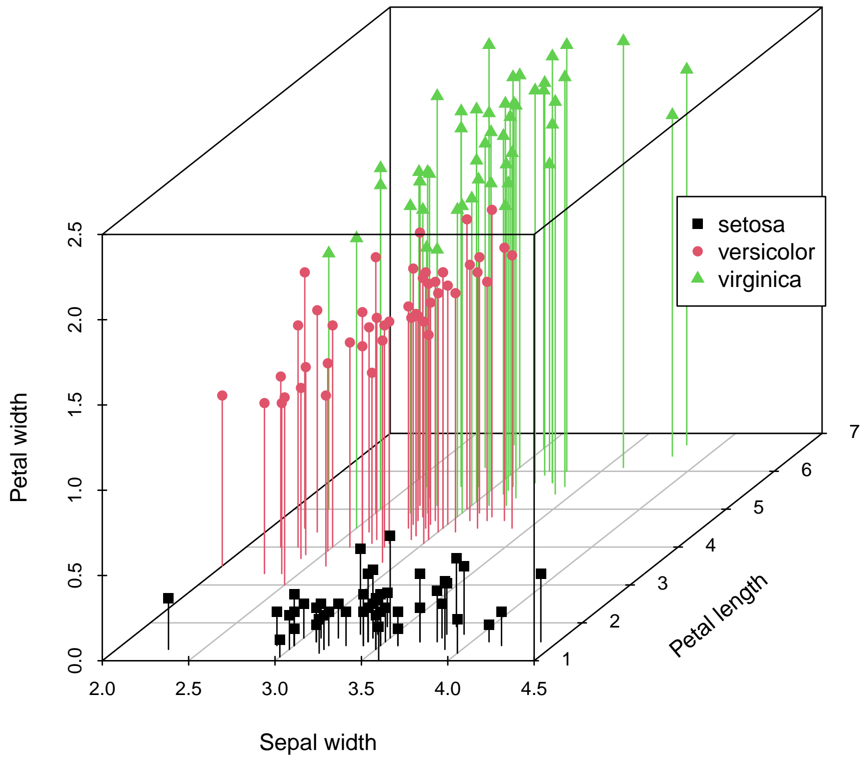


Figure 7.11: Static 3D scatterplot of `iris` data.

What will be really helpful is to simplify the data first. For example, *reduce dimensions* to two or three, keeping the best view angle. These techniques are described in next chapter.

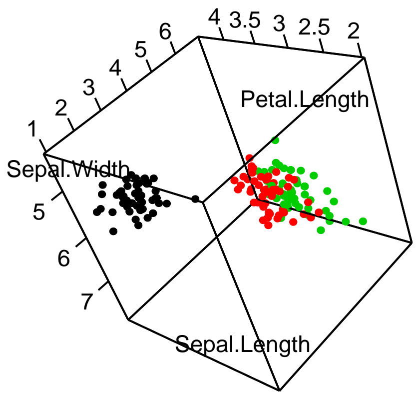


Figure 7.12: RGL 3D scatterplot: one of positions.

Figure 7.13: Static 3D cloud plot of `iris` data with several rotations.

Chapter 8

Discover

As Sherlock Holmes said (at least sometimes), it is not a good practice to formulate hypotheses before we know the data. Discovery methods ("non-supervised classification", "classification without learning") always start from scratch and output results regardless to what researcher actually wants. There are many discovery methods. Below, we split them in two groups: methods which use original, *primary data* and methods which start from *distance calculation*.

Another approach is to split discovery methods into (a) *manifold* and (b) *layout* methods (Fig. 8.1) where the first group is about (a) space reduction (manifold learning, shape discovery, ordination) and the second is about (b) layout reduction (order learning, structure discovery of structure).

Mathematically, *manifold* methods are mainly based on hyperspace geometry and return projections or deconvolutions (unfolds), whereas *layout* methods frequently use graph theory and return simple classifications (piles, groups) or more complex hierarchical objects like trees or decision keys.

There are also methods which combine manifold and layout discovery (see, for example, the `clustrd` package), and methods which use them together with machine learning (i.e., employ training data). Typical discovery methods do not use learning, but could be leveraged or educated (for example, clustering methods frequently require education: they want users to specify the desired number of clusters).

As manifold methods reduce dimensionality, they typically provide a good ground for multidimensional *visualizations*.

Discovery methods might be able also to reveal hidden factors (latent variables), hidden structure and *variable importance* (thus, help to perform the feature selection).

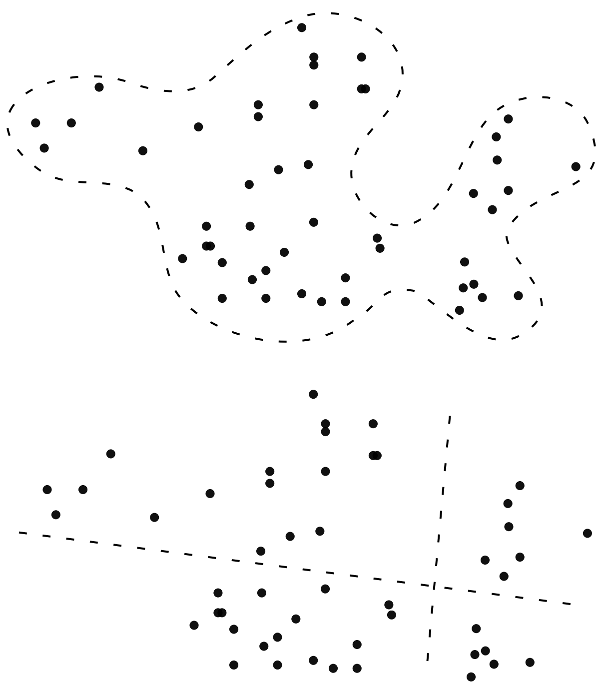


Figure 8.1: The difference between manifold (above) and layout (below) approaches.

Package `Boruta` is especially good for the all relevant feature selection. Package `bnlearn` performs Bayesian learning of the data structure.

8.1 Discovery with primary data

Primary data is what come directly from observation, and was not yet processed in any further way (to make secondary data).

8.1.1 Shadows of hyper clouds: PCA

RGL (see above) allows to find the best projection manually, with a mouse. However, it is possible to do programmatically, with *principal component analysis*, PCA.

PCA treats the data as points in the virtual multidimensional space where every dimension is the one character. These points make together the multidimensional cloud. The goal of the analysis is to find a line which crosses this cloud through the most elongated part, like pear on the stick (Fig. 8.2). This is the first, most informative *principal component*. The second is perpendicular to the first and again span the

second most elongated part of the cloud. These two lines make the plane on which every point is projected.

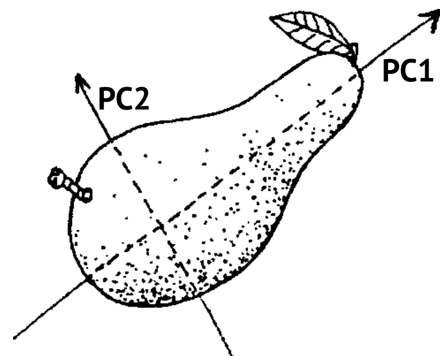


Figure 8.2: Principal component analysis is like the pear on the stick.

Let us prove this practically. We will load the two-dimensional (hence only two principal components) black and white pear image and see what PCA does with it:

```
> library(png) # package to read PNG images as data
> aa <- readPNG("data/pear2d_bw.png")
> bb <- which(aa == 0, arr.ind=TRUE) # pixels to coordinates
> ## plot together original (green) and PCA-rotated (gray):
> bbs <- scale(bb)
> pps <- scale(prcomp(bb)$x[, 1:2]) # only two PCs
> xx <- range(c(bbs[, 1], pps[, 1]))
> yy <- range(c(bbs[, 2], pps[, 2]))
> plot(pps, pch=".", col=adjustcolor("black", alpha=0.5),
+ xlim=xx, ylim=yy)
> points(bbs, pch=".", col=adjustcolor("green", alpha=0.5))
> legend("bottomright", fill=adjustcolor(c("green", "black"),
+ alpha=0.5), legend=c("Original", "PCA-rotated"),
+ bty="n", border=0)
```

The number of principal components is the same as the number of initial characters but first two or three usually include all necessary information. This is why it is possible to use them for 2D visualization of multidimensional data.

This is an example from the open repository presenting measurements of four different populations of sedges:

```
> ca <- read.table(
+ "http://ashipunov.me/shipunov/open/carex.txt", h=TRUE)
```

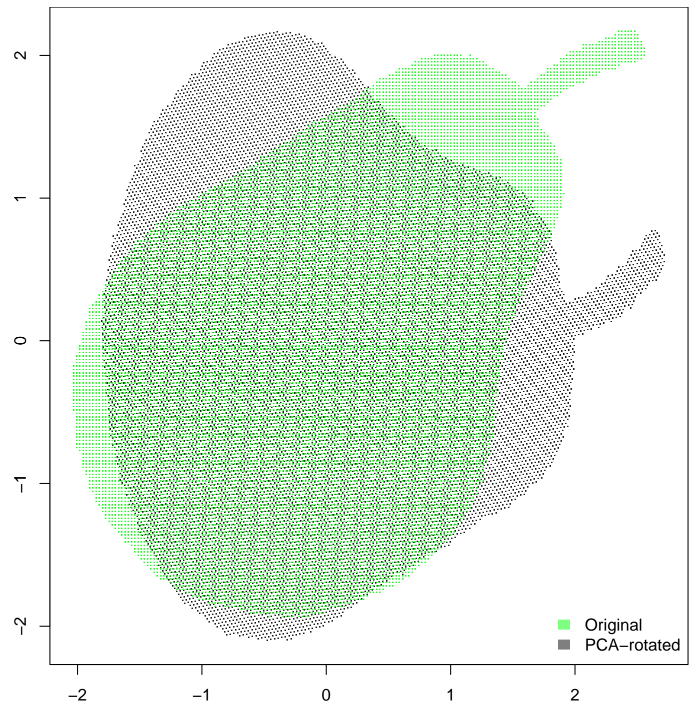


Figure 8.3: Shadow of the pear: how PCA projects the image.

```
> Str(ca)
 'data.frame': 62 obs. of  5 variables:
  1  LOC    : int  1 1 1 1 1 1 1 1 1 1 ...
  2  HEIGHT : int  157 103 64 77 21 27 19 35 43 92 ...
  3  LEAF.W : num  2.5 2.5 2 2 1.5 1.5 2 1.5 2 2 ...
  4  SPIKE.L: num  9.5 9 7.5 7 4 5 3.5 6 6 6.5 ...
  5  SPIKE.W: num  6.5 6.5 6 5 4 4 3.5 5 5 5.5 ...
> ca.pca <- princomp(scale(ca[, -1]))
```

(Function `scale()` standardizes all variables.)

The following (Fig. 8.4) plot is technical *screeplot* which shows the relative importance of each component:

```
> plot(ca.pca, main="")
```

Here it is easy to see that among four components (same number as initial characters), two first have the highest importances. There is a way to have the same without plotting:

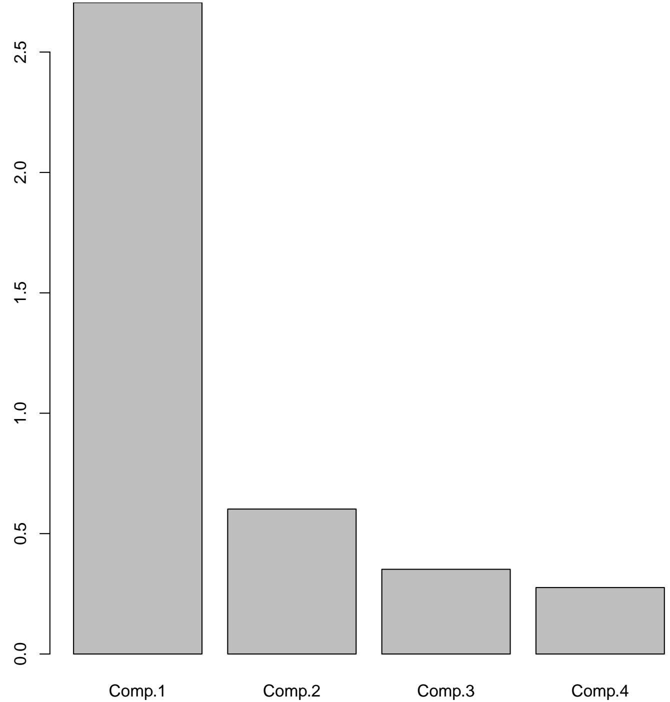


Figure 8.4: Plot showing the importance of each component.

```
> summary(ca.pca)
Importance of components:
                          Comp.1    Comp.2     Comp.3     Comp.4
Standard deviation     1.6448264 0.7759300 0.59318563 0.52544578
Proportion of Variance 0.6874514 0.1529843 0.08940939 0.07015485
Cumulative Proportion  0.6874514 0.8404358 0.92984515 1.00000000
```

First two components together explain about 84% percents of the total variance.

Visualization of PCA is usually made using scores from PCA model (Fig. 8.5):

```
> ca.p <- ca.pca$scores[, 1:2]
> plot(ca.p, type="n", xlab="PC1", ylab="PC2")
> text(ca.p, labels=ca[, 1], col=ca[, 1])
> Hulls(ca.p, ca[, 1]) # shipunov
```

(Last command draws hulls which help to conclude that first sedges from the third population are intermediate between first and second, they might be even hybrids.)

It is tempting to *measure* the intersection between hulls:

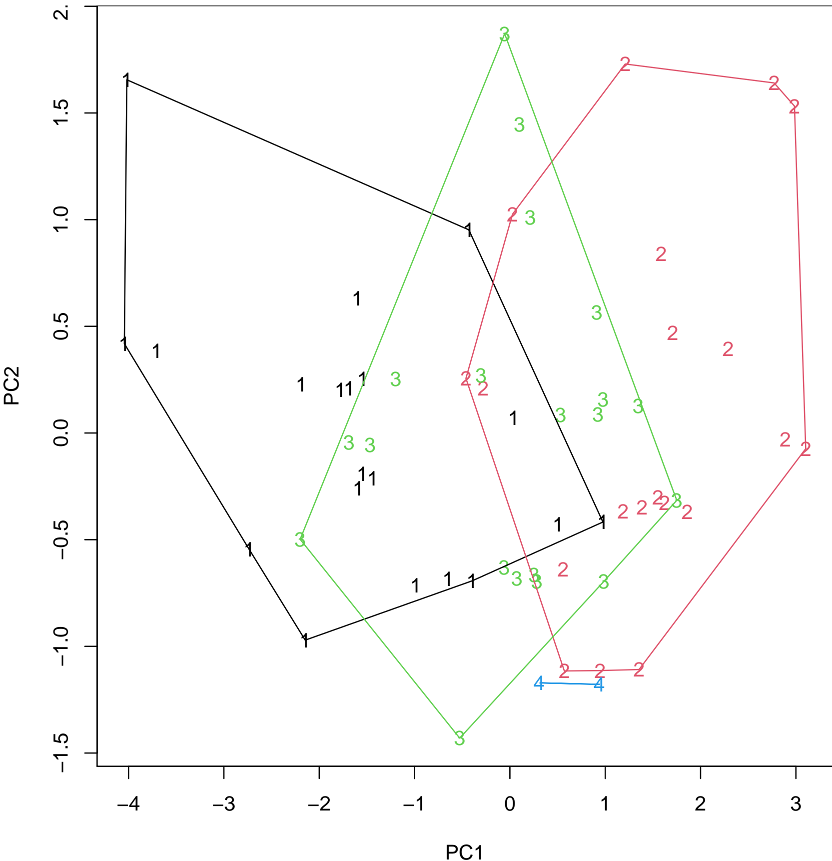


Figure 8.5: Diversity of sedges on the plot of two first principal components.

```
> ca.h <- Hulls(ca.p, ca[, 1], plot=FALSE) # shipunov
> ca.o <- Overlap(ca.h) # shipunov
Loading required package: PBSmapping
...
> summary(ca.o)
Overlaps for each hull, %:
  mean.overlap total.overlap
1        25.81         51.63
2        22.91         45.83
3        44.68         89.36
4          NaN          0.00
Mean overlap for the whole dataset 31.14 %
```

Sometimes, PCA results are useful to present as a biplot (Fig. 8.6):

```
> biplot(ca.pca, xlabs=ca[, 1])
```

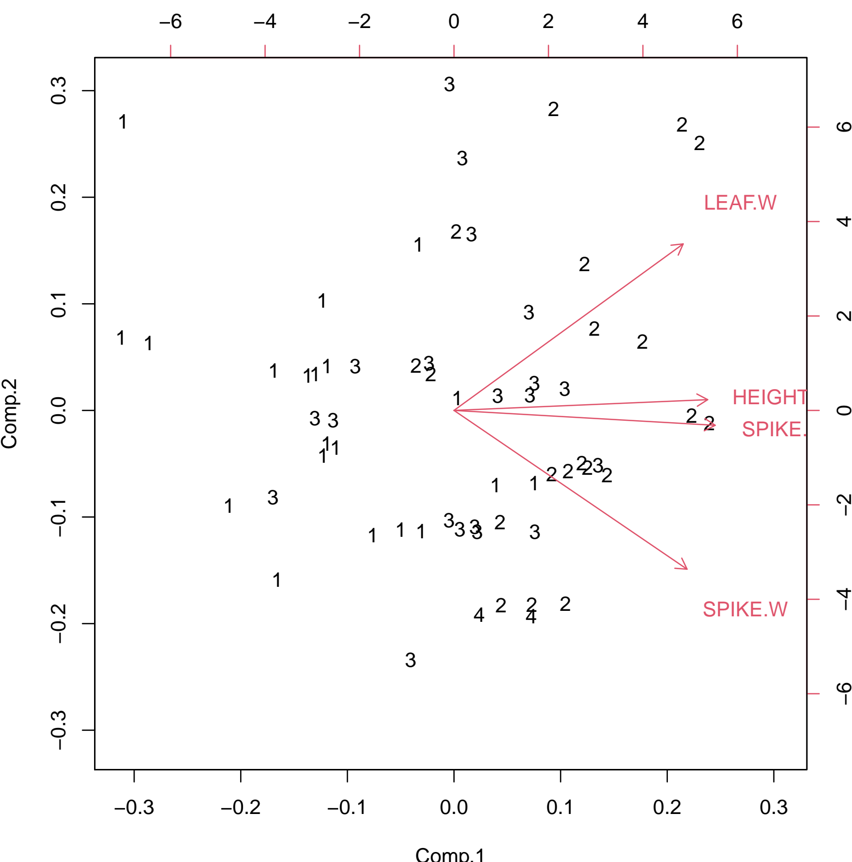


Figure 8.6: Biplot shows the load of each character into two first components.

Biplot helps to understand visually how large is the load of each initial character into first two components. For example, characters of height and spike length (but spike width) have a biggest loads into the first component which distinguishes populations most. Function `loadings()` allows to see this information in the numerical form:

```
> loadings(ca.pca)
Loadings:
        Comp.1 Comp.2 Comp.3 Comp.4
HEIGHT  -0.518         0.845 -0.125
LEAF.W  -0.468  0.721 -0.263  0.437
SPIKE.L -0.534        -0.432 -0.724
```

```
SPIKE.W -0.476 -0.688 -0.178  0.518
...
```

R has two variants of PCA calculation, first (already discussed) with `princomp()`, and second with `prcomp()`. The difference lays in the way how exactly components are calculated. First way is traditional, but second is recommended:

```
> iris.pca <- prcomp(iris[, -5])
> iris.pca$rotation
                     PC1         PC2         PC3        PC4
Sepal.Length  0.36138659 -0.65658877  0.58202985  0.3154872
Sepal.Width  -0.08452251 -0.73016143 -0.59791083 -0.3197231
Petal.Length  0.85667061  0.17337266 -0.07623608 -0.4798390
Petal.Width   0.35828920  0.07548102 -0.54583143  0.7536574
> iris.p <- iris.pca$x[, 1:2]
> plot(iris.p, type="n", xlab="PC1", ylab="PC2")
> text(iris.p, labels=abbreviate(iris[, 5], 1, method="both.sides"),
+ col=as.numeric(iris[, 5]))
> Ellipses(iris.p[, 1:2], as.numeric(iris[, 5])) # shipunov
```

Example above shows some differences between two PCA methods. First, `prcomp()` conveniently accepts scale option. Second, loadings are taken from the `rotation` element. Third, scores are in the the element with `x` name. Please **run** the code yourself to see how to add 95% confidence ellipses to the 2D ordination plot. One might see that *Iris setosa* (letter "s" on the plot) is seriously divergent from two other species, *Iris versicolor* ("v") and *Iris virginica* ("a").

(Function `prcomp()` has the convenient `scale=TRUE` option, but in case of `iris` data where measurement variables are uniform, scaling only makes species less separable so we avoided it.)

* * *

Packages `ade4` and `vegan` offer many variants of PCA (Fig. 8.7):

```
> library(ade4)
> iris.dudi <- dudi.pca(iris[, 1:4], scannf=FALSE)
> s.class(iris.dudi$li, iris[, 5])
```

(The plot is similar to the shown on Fig. 8.5; however, the differences between groups are here more clear.)

In addition, this is possible to use the inferential approach for the PCA:

```
> iris.between <- bca(iris.dudi, iris[, 5], scannf=FALSE)
```

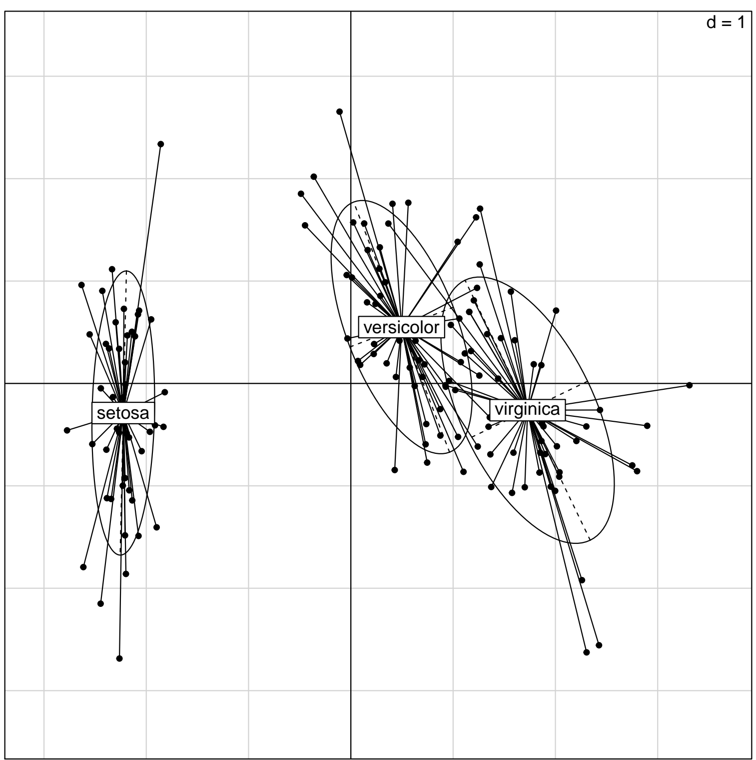


Figure 8.7: Diversity of irises on the plot of two first principal components (ade4 package)

```
> randtest(iris.between)

Monte-Carlo test
Call: randtest.between(xtest = iris.between)
Observation: 0.7224358
Based on 999 replicates
Simulated p-value: 0.001
Alternative hypothesis: greater
...
```

Monte-Carlo randomization allows to understand numerically how well are *Iris* species separated with this PCA. The high `Observation` value (72.2% which is larger than 50%) is the sign of reliable differences.

There are other variants of permutation tests for PCA, for example, with `anosim()` from the `vegan` package:

```
> library(vegan)
> anosim(iris.p, iris$Species, permutations=99, distance="euclidean")
...
ANOSIM statistic R:  0.78
      Significance: 0.01
...
```

(ANOSIM statistics behaves similarly to correlation coefficient so the closer is it to 0, the more random is the correspondence between grouping and new dimensions.)

Please note that principal component analysis is a *linear* technique similar to the analysis of correlations, and it can fail in some complicated cases. Generally, PCA also wants data to be measurement. Package `Gifi` contains PCA modifications for other data types, for example, `princals()` which is categorical PCA. The other common workaround is to calculate distances (see below) from any kind of data and then use them for PCA or PCA-related techniques.

8.1.2 Correspondence

Correspondence analysis is the family of techniques similar to PCA, but applicable to categorical data (primary or in contingency tables). Simple variant of the correspondence analysis is implemented in `corresp()` from `MASS` package (Fig. 8.8) which works with contingency tables:

```
> library(MASS)
> HE <- margin.table(HairEyeColor, 1:2)
> HE.df <- Table2df(HE) # shipunov
> biplot(corresp(HE.df, nf = 2), xpd=TRUE)
> legend("topleft", fill=1:2, legend=c("hair","eyes"))
```

(We converted here “table” object `HE` into the data frame. `xpd=TRUE` was used to allow text to go out of the plotting box.)

This example uses `HairEyeColor` data from previous chapter. One can see there that black hairs and brown eyes have similar pattern of occurrence. The position of these words is more distant from the center (which is designated with cross) because numerical values of these characters are remote.

The possibility to visualize several aspects of data simultaneously on the one plot is the impressive feature of correspondence analysis (Fig. 8.9):

```
> library(vegan)
> alla <- read.table("data/lakesea_abio.txt", sep="\t", h=TRUE)
> allc <- read.table("data/lakesea_bio.txt", sep="\t", h=TRUE)
> oldnames <- names(alla) # names are too long for the plot
```

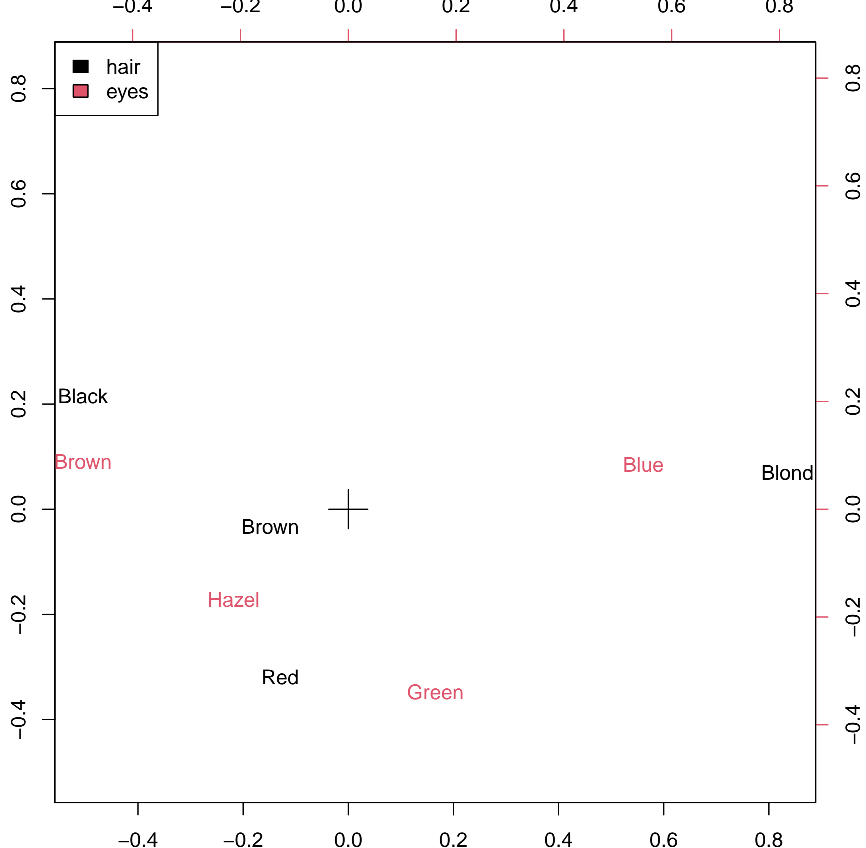


Figure 8.8: Correspondence plot of contingency table.

```
> names(alla) <- LETTERS[1:ncol(alla)]
> all.cca <- cca(allc, alla[, -14])
> plot(all.cca, display=c("sp", "cn"))
> points(all.cca, display="sites", pch=ifelse(alla[, 14], 15, 0))
> legend("topleft", pch=c(15, 0, 3), col=c(1, 1, 2),
+ legend=c("lake sites", "sea sites", "plant species"))
> legend("bottomright", pch=names(alla)[-14], col="blue",
+ legend=oldnames[-14])
> text(-1.6, -4.2, "Carex lasiocarpa", pos=4)
```

This is much more advanced than simple biplot. Data used here contained both abiotic (ecotopes) and biotic factors (plant species), plus the geography of some Arctic islands: whether these are lake islands or sea islands. The plot was able to arrange

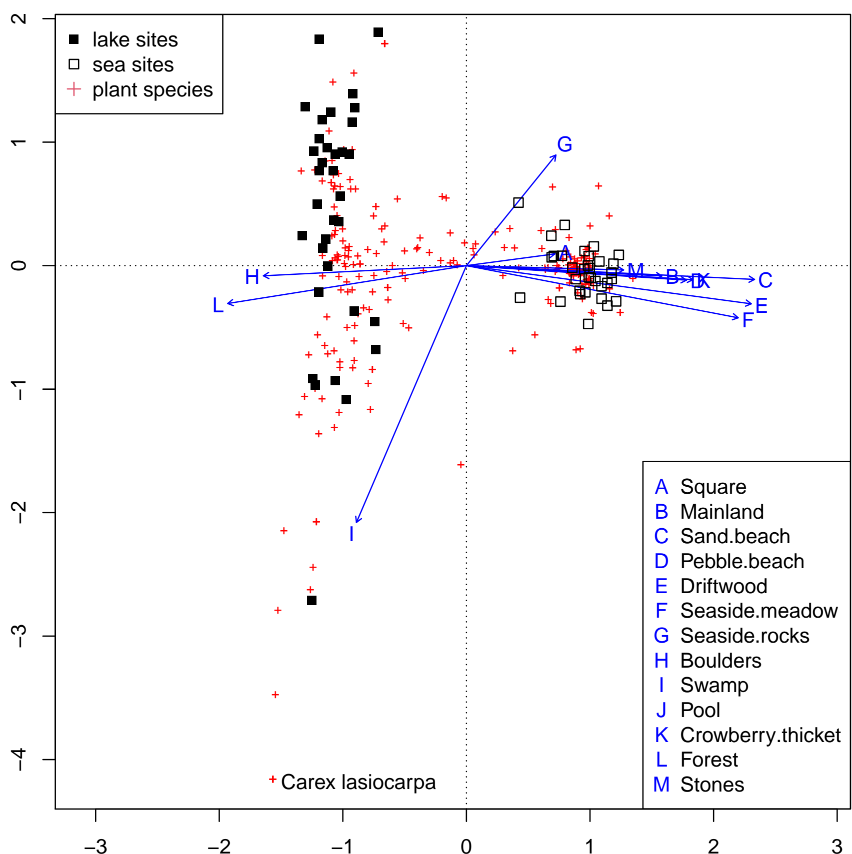


Figure 8.9: Canonical correlation analysis plot showing Arctic islands (squares), species (crosses) and habitat factors (arrows)

all of these data: for abiotic factors, it used arrows, for biotic—pluses, and for sites (islands themselves as characterized by the sum of all available factors, biotic and abiotic)—squares of different color, depending on geographic origin.

All pluses could be identified with the `identify(plot.all.cca, "species")` command. We did it just for one most outstanding species, *Carex lasiocarpa* (woolly-fruit sedge) which is clearly associated with lake islands, and also with swamps.

8.1.3 Projections, unfolds, t-SNE and UMAP

PCA and many other methods like correspondence analysis, factor analysis and multidimensional scaling (MDS, see below) do not preserve local distances—these meth-

ods are *projections* that typically "squeeze" data to preserve global picture more than local.

Some other dimension reduction methods (like *isomap* which is available in R packages dimRed, Rdimtools and vegan) tend to preserve local distances more: they are *deconvolutions*, "unfolds". Unfolds flatten or distort data but locally close points will likely remain close.

One disadvantage of these methods is that they are usually slow. On the other hand, there is the tapkee R package based on the fast dimension reduction library with the same name[1]. Below we will try to open the Swiss roll with several tapkee methods:

```
> library(tapkee)
> SR <- Gen.dr.data("swissroll")
> COL <- rainbow(1100)[1:1000] # separates colors better
> TM <- c("lle", "npe", "ltsa", "hlle", "la", "isomap", "spe", "pca")
> names(TM) <- c("Locally Linear Embedding",
+ "Neighborhood Preserving1 Embd.",
+ "Local Tangent Space Alignment",
+ "Hessian Locally Linear Embd.",
+ "Laplacian Eigenmaps", "Isomap",
+ "Stochastic Proximity Embd.", "PCA")
> oldpar <- par(mfrow=c(3, 3), xaxt="n", yaxt="n")
> scatterplot3d(SR, color=COL, pch=20, main="Swiss Roll",
+ cex.symbols=1.2, xlab="", ylab="", zlab="",
+ axis=FALSE, tick.marks=FALSE,
+ label.tick.marks=FALSE, mar=c(1, 1, 2, 1))
> for (n in 1:length(TM)) plot(Tapkee(SR, method=TM[n]),
+ col=COL, pch=20, main=names(TM)[n])
> par(oldpar)
```

Now if you look on Fig. 8.10, you will spot that some methods (like Hessian locally linear embedding, or isomap) truly unroll the structure whereas others (like PCA or neighborhood preserving embedding) see it from one of it sides. First methods are deconvolutions, second are projections.

There are also methods which are in between projections and unfolds, for example, t-SNE and UMAP.

With the really big number of samples, *t-SNE algorithm* (name stands for "t-Distributed Stochastic Neighbor Embedding") often performs better than classical PCA. t-SNE is frequently used for the shape recognition.

[1]You will need to install this library separately, please run package?tapkee for instructions.

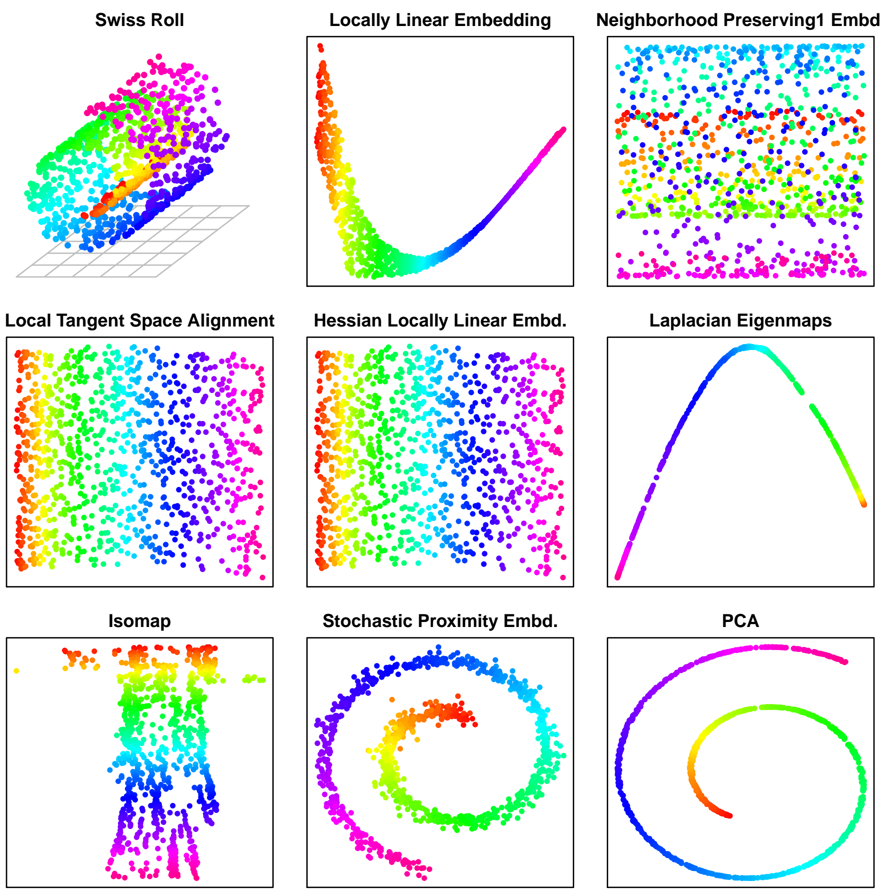


Figure 8.10: How to open the Swiss roll with `tapkee` package: eight different methods.

UMAP (Uniform Manifold Approximation and Projection) is close to t-SNE but performs more efficiently, and often returns even better results. One of recent applications of UMAP is especially famous. All integers might be represented with prime factors of divisibility. If we make the binary matrix of prime divisibility and "feed" it to UMAP, amazing structures (Fig. 8.11) appear, more similar to fireworks or maps of Star Wars galaxy then to just prime factors.

It is easy enough to employ both t-SNE and UMAP in `R` (Fig. 8.12):

```
> library(uwot)
> library(tapkee)
> ## library(Rtsne)
```

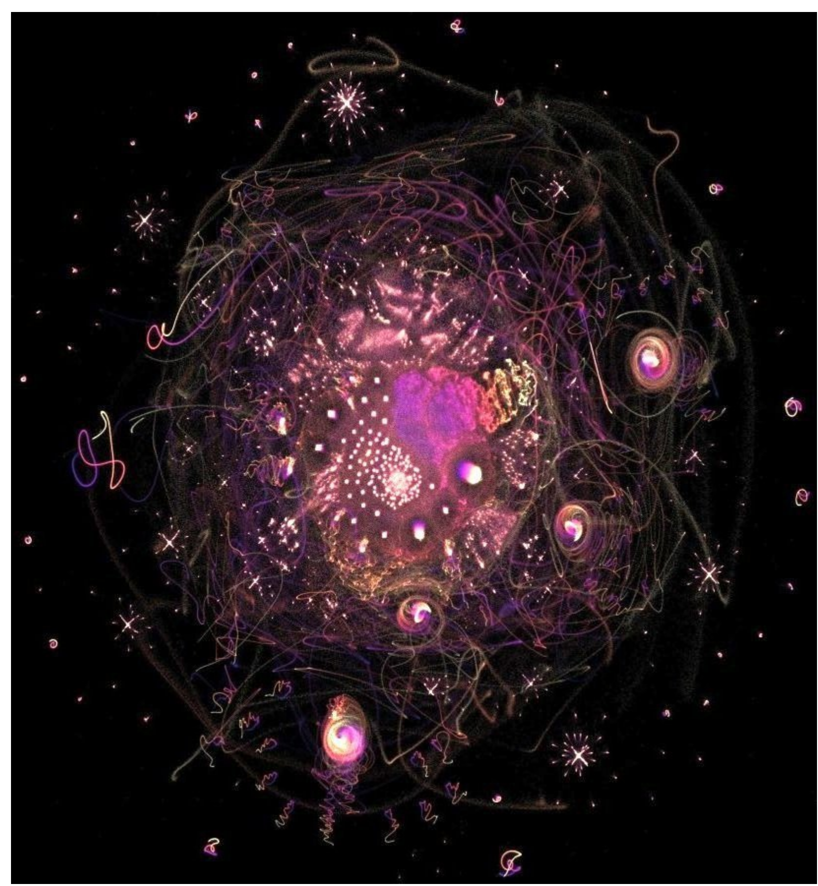


Figure 8.11: UMAP visualization of 8,000,000 integers represented with binary vectors of prime factors, colored by density of points.

```
> SR <- Gen.dr.data("swissroll")
> COL <- rainbow(1100)[1:1000] # separates colors better
> oldpar <- par(mfrow=c(1, 2), mar=c(1, 1, 3, 1), xaxt="n", yaxt="n")
> ## plot(Rtsne(SR)$Y, col=COL, pch=20, main="t-SNE (Rtsne)")
> plot(Tapkee(SR, method="t-sne"),
+ col=COL, pch=20, main="t-SNE (tapkee)")
> plot(umap(SR, n_neighbors=15, approx_pow=TRUE, pca=50),
+ col=COL, pch=20, main="UMAP (uwot)")
> par(oldpar)
```

(You can run t-SNE either from Rtsne or from tapkee package, the second is slightly faster so the first is commented out. Note umap() parameters, they speed up the calculations.)

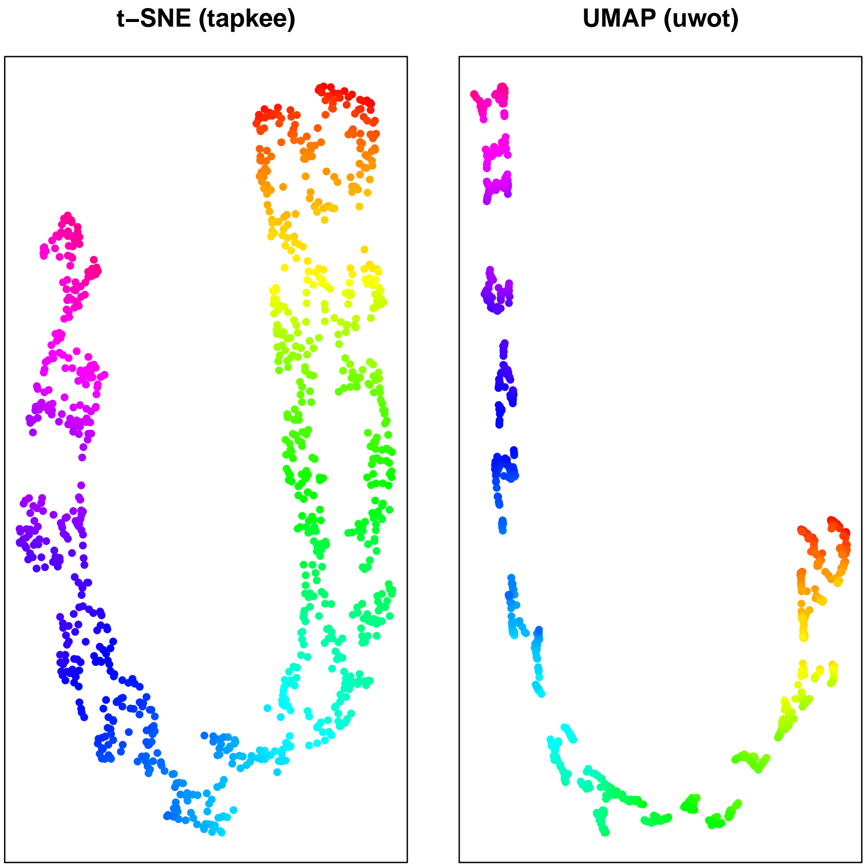


Figure 8.12: How to t-SNE and UMAP try to open the Swiss roll keeping its overall structure.

8.1.4 Non-negative matrix factorization

Non-negative matrix factorization (NNF) is based on the simple idea. If our data set consists of multiple rows and columns, we can make it the product (or approximate product) of two much smaller matrices. One of them (traditionally called W) will contain row scores, and the second (H)—column scores. These row scores might be used as new dimensions. Package NMFN provides NNF for R:

```
> library(NMFN)
```

```
> set.seed(1)
> iris.nnmf <- nnmf(iris[, -5], k=2) # we want two factors
Multiplicative Update Algorithm
Iter =  50 ...
...
Iter =  1000 ...
> plot(iris.nnmf$W, col=iris$Species, xlab="F1", ylab="F2")
> iris.nh <- Hulls(iris.nnmf$W, iris$Species) # shipunov
> summary(Overlap(iris.nh)) # shipunov
Overlaps for each hull, %:
           mean.overlap total.overlap
setosa              NaN          0.00
versicolor         6.73          6.73
virginica          4.98          4.98
Mean overlap for the whole dataset 5.85 %
```

(Please **review** the plot yourself.)

Generally, NNF is the method of data compression. It has similarities with PCA, with *k*-means clustering (see below) and with support vector machines (SVM, also below). Typically, NNF results are comparable (or slightly better) to results of PCA.

8.2 Discovery with distances

Important way of non-supervised classification is to work with **distances** instead of original data. Distance-based methods need the *dissimilarities* between each pair of objects to be calculated first. Advantage of these methods is that dissimilarities could be calculated from data of any type: measurement, ranked or nominal.

8.2.1 Distances

There are myriads of ways to calculate dissimilarity (or similarity which is essentially the reverse dissimilarity)[2]. One of these ways already explained above is a (reverse absolute) correlation. Other popular ways are Euclidean (square) distance and Manhattan (block) distance. Both of them (Fig. 8.13) are useful for measurement variables (but Manhattan is more suitable for high-dimensional datasets).

Manhattan distances are similar to driving distances, especially when there are not many roads available. The example below are driving distances between biggest North Dakota towns:

[2]For example, "Encyclopedia of Distances" (2009) mentions about 1,500!

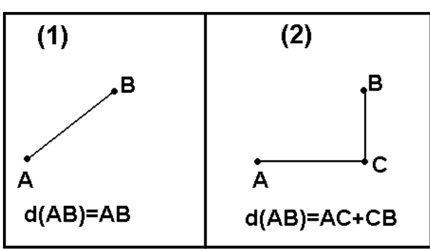


Figure 8.13: Euclidean (1) and Manhattan (2) distances between A and B

```
> nd <- read.table("data/nd.txt", h=TRUE, sep="\t", row.names=1)
> nd.d <- as.dist(nd)
> str(nd.d)
 'dist' int [1:28] 110 122 170 208 268 210 173 219 101 135 ...
 - attr(*, "Labels")= chr [1:8] "Minot" "Bismarck" ...
...
```

In most cases, however, distances should be calculated from raw variables. The basic way is to use `dist()`. Note that ranked and binary variables usually require different approaches which are implemented in the `vegan` (function `vegdist()`) and `cluster` packages (function `daisy()`). The last function recognizes the type of variable and applies the most appropriate metric (including the universal Gower distance); it also accepts the metric specified by user:

```
> library(cluster)
> iris.dist <- daisy(iris[, 1:4], metric="manhattan")
> str(iris.dist)
 'dissimilarity' num [1:11175] 0.7 0.8 1 0.2 1.2 ...
 - attr(*, "Size")= int 150
 - attr(*, "Metric")= chr "manhattan"
```

Gower distance (and similar Wishart and Podani distances) could use any type of variable, including categorical:

```
> iris.gower <- Gower.dist(iris) # shipunov
> str(iris.gower)
 'dist' num [1:11175] 0.0528 0.0506 0.0645 0.0139 0.0768 ...
 - attr(*, "Labels")= chr [1:150] "1" "2" "3" "4" ...
```

(Please read the code carefully and you will see that character number 5 (species name) is not removed, as we do almost everywhere in this book! This is because `Gower.dist()` recognizes the variable type automatically and then uses it in distance calculation.)

* * *

Again, there are myriads of distance measures, based, for example, on linguistic string similarity or on physical potential energy. In biology, one can use *Smirnov taxonomic distances*, available from `smirnov` package. In the following example, we use plant species distribution data on small islands.

The next plot intends to help the reader to understand them better. It is just a kind of map which reflects geographical locations and sizes of islands:

```
> library(MASS)
> eqscplot(moldino_l$LON, moldino_l$LAT,
+ cex=round(log(moldino_l$SQUARE))-5.5,
+ axes=FALSE, xlab="", ylab="", xpd=TRUE) # shipunov
> text(moldino_l$LON, moldino_l$LAT, labels=row.names(moldino_l),
+ pos=4, offset=1, cex=.9)
```

(Please **plot** it yourself.)

Now we will calculate and visualize Smirnov's distances:

```
> m1 <- t((moldino > 0) * 1) # convert to 0/1 and transpose
> library(smirnov)
> m1.Txy <- smirnov(m1)
> m1.s <- (1 - m1.Txy) # convert similarity into dissimilarity
> dimnames(m1.s) <- list(row.names(m1))
> symnum(m1.s)
Bobrovyj         +
Ekslibris        + +
Gnutyj           , + *
Leda             , + , *
Malinovyj.Kruglyj + + + + +
Slitnyj          , , , , + *
Tajnik           , , , , + , *
Verik            + + + + , , + +
Zakhar           , + + , + + , + +
attr(,"legend")
[1] 0 ‘ ’ 0.3 ‘.’ 0.6 ‘,’ 0.8 ‘+’ 0.9 ‘*’ 0.95 ‘B’ 1
```

Smirnov's distances have an interesting feature: instead of 0 or 1, diagonal of the similarity matrix is filled with the *coefficient of uniqueness* values (Txx):

```
> m1.Txx <- diag(m1.Txy)
> names(m1.Txx) <- row.names(m1)
> rev(sort(round(m1.Txx, 3)))
```

```
Verik Malinovyj.Kruglyj          Ekslibris          Bobrovyj
0.189              0.130              0.124              0.106
...
```

This means that Verik island is a most unique in regards to plant species occurrence.

8.2.2 Making maps: multidimensional scaling

There are many things to do with the distance matrix. One of most straightforward is the *multidimensional scaling*, MDS (the other name is "principal coordinate analysis", PCoA). MDS is the manifold learning method, similar to PCA but applied to distances:

```
> nd.d <- as.dist(nd)
> nd.c <- cmdscale(nd.d)
> new.names <- sub("y C", "y\nC", row.names(nd))
> library(MASS)
> eqscplot(nd.c, type="n", axes=FALSE, xlab="", ylab="")
> points(nd.c, pch=19)
> text(nd.c, labels=new.names, xpd=TRUE, pos=3, cex=0.8)
```

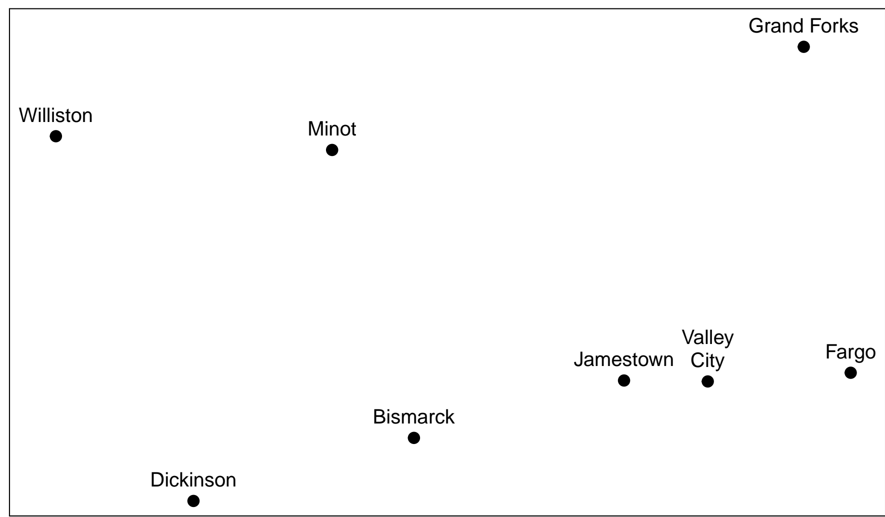


Figure 8.14: It is not a map of North Dakota towns. It is `cmdscale()` output from the driving distance data.

Compare the plot (Fig. 8.14) with geographical map. You will see that `cmdscale()` allows to re-create the map! In essence, MDS is a task reverse to navigation (finding

driving directions from map): it uses "driving directions" and makes a map from them.

Another, less impressive but more useful example (Fig. 8.15) is from raw data of Fisher's irises:

```
> library(KernSmooth)
> iris.c <- cmdscale(iris.dist)
> est <- bkde2D(iris.c, bandwidth=c(.7, 1.5))
> plot(iris.c, type="n", xlab="Dim. 1", ylab="Dim. 2")
> text(iris.c,
+ labels=abbreviate(iris[, 5], 1, method="both.sides"))
> contour(est$x1, est$x2, est$fhat, add=TRUE,
+ drawlabels=FALSE, lty=3)
```

To make the plot "prettier", we added here density lines of point closeness estimated with `bkde2D()` function from the `KernSmooth` package. Another way to show density is to plot 3D surface (Fig. 8.16):

```
> persp(est$x1, est$x2, est$fhat, theta=135, phi=45,
+ col="purple3", shade=0.75, border=NA,
+ xlab="Dim. 1", ylab="Dim. 2", zlab="Density")
```

In addition to `cmdscale()`, `MASS` package implements the non-metric multidimensional scaling (functions `isoMDS()` and `sammon()`), and package `vegan` has the advanced non-metric `metaMDS()`. Non-metric multidimensional scaling does not have analogs to PCA loadings (importances of variables) and proportions of variance explained by each component, but it is possible to calculate surrogate metrics:

```
> iris.dist2 <- dist(iris[, 1:4], method="manhattan")
> ## to remove zero distances:
> iris.dist2[iris.dist2 == 0] <- abs(jitter(0))
> library(MASS)
> iris.m <- isoMDS(iris.dist2)
initial  value 5.444949
iter   5 value 4.117042
final  value 4.075094
converged
> cor(iris[, 1:4], iris.m$points) # variable importance ("loadings")
                   [, 1]        [, 2]
Sepal.Length  0.9041779 -0.30455326
Sepal.Width  -0.3996573 -0.87872539
Petal.Length  0.9956071  0.04209809
Petal.Width   0.9661085 -0.01138353
```

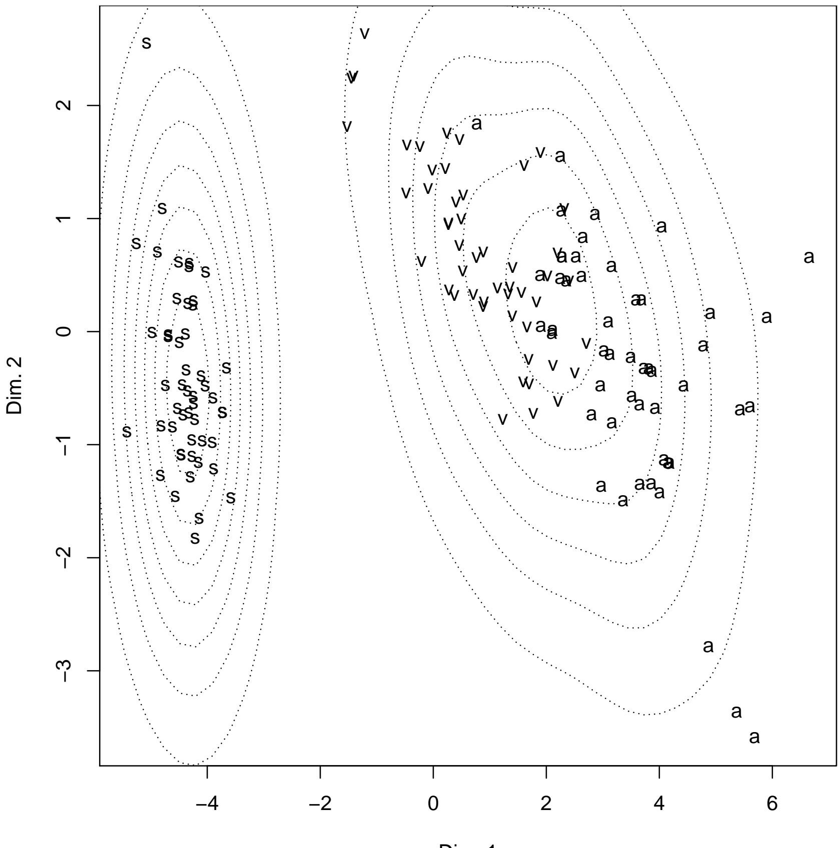


Figure 8.15: The result of the multidimensional scaling of the `iris` data. Visualization uses the estimation of density.

```
> (vv <- MDSv(iris.m$points)) # shipunov
[1] 97.509738  2.490262 # dimension importance ("explained variance")
```

With `Biarrows()`, it is possible to visualize variable importance in a way similar to PCA biplot:

```
> xxlab <- paste0("Dim 1 (", round(vv[1], 2), "%)")
> yylab <- paste0("Dim 1 (", round(vv[2], 2), "%)")
> aabb <- abbreviate(iris$Species, 1, method="both.sides")
> plot(iris.m$points, pch=aabb, xlab=xxlab, ylab=yylab)
> Biarrows(iris.m$points, iris[, -5]) # shipunov
```

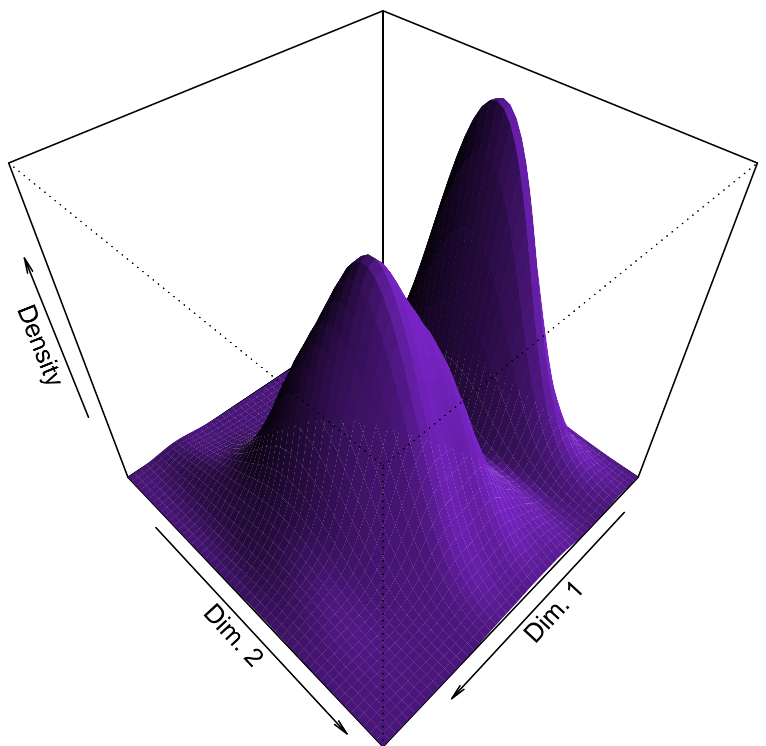


Figure 8.16: 3D density surface of multidimensionally scaled `iris` data.

(Please check this plot yourself. Note that `Biarrows()` works well for most dimension reduction techniques.)

Similarly to PCA, the sepal width character influences second dimension much more than three other characters. We can also guess that with this non-metric solution, the first dimension importance is about 98%.

8.2.3 Making trees: hierarchical clustering

From this point, we switch to another set of discovery methods, *layout* learning methods. They also called “clusterization” or even “classification”.

Layout methods are based on distances but their results are more similar to connected nodes (graphs) then to maps. Graphs are in many ways more helpful to researcher because they facilitate decisions. They also convenient for studying time-based processes such as evolution.

Hierarchical clustering is the way to process the distance matrix stepwise. It returns *dendrograms*, or trees, which are "one and a half dimensional[3]" plots (Fig. 8.17):

```
> aa <- t(atmospheres) # shipunov
> aa.dist <- dist(aa) # planets are columns
> aa.hclust <- hclust(aa.dist, method="ward.D")
> plot(aa.hclust, xlab="", ylab="", sub="")
```

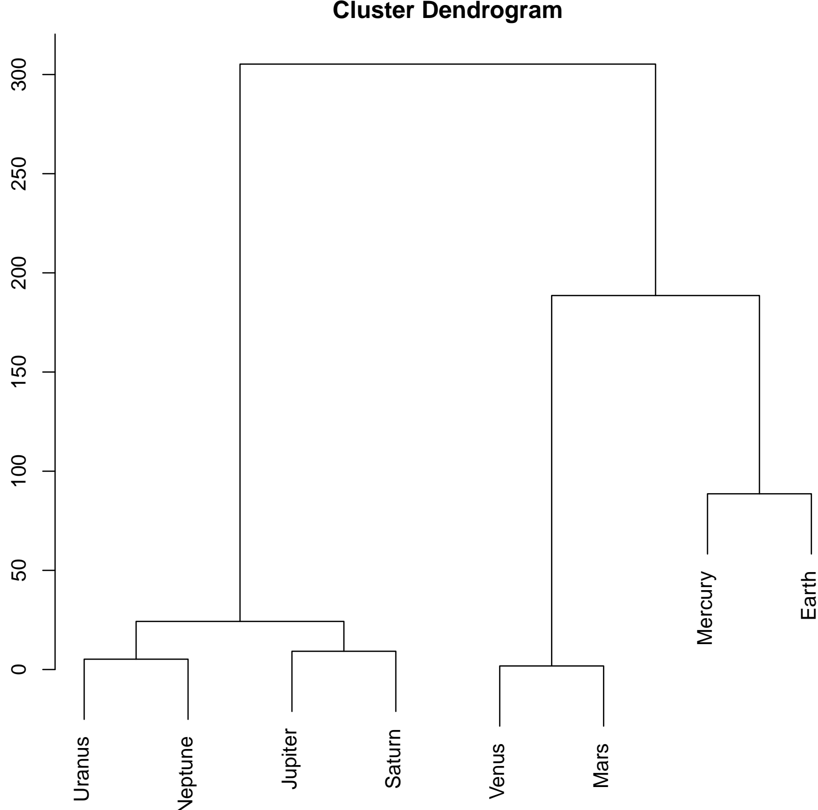


Figure 8.17: Dendrogram which reflects similarities between atmospheres of Solar system planets.

Ward's method of clustering is well known to produce sharp, well-separated clusters (this, however, might lead to false conclusions if data has no apparent structure). Distant planets are most similar (on the height $\approx$ 25), similarity between Venus and Mars is also high (dissimilarity is $\approx$ 0). Earth is more outstanding, similarity with

[3]This is not actually a metaphor, many trees have fractal dimensionality between 1 and 2.

Mercury is lower, on the height ≈ 100; but since Mercury has no true atmosphere, it could be ignored.

The following classification could be produced from this plot:

- Earth group: Venus, Mars, Earth, [Mercury]
- Jupiter group: Jupiter, Saturn, Uranus, Neptune

Instead of this "speculative" approach, we can use `cutree()` function to produce classification explicitly:

```
> cutree(aa.hclust, k=2) # we want 2 groups
Mercury   Venus   Earth    Mars Jupiter  Saturn  Uranus Neptune
      1       1       1       1       2       2       2       2
```

Now we can try to cluster the `iris` data:

```
> library(cluster) # advanced clustering package
> iris.dist <- daisy(iris[, 1:4], metric="manhattan") # distances
> iris.hclust <- hclust(iris.dist)
> iris.3 <- cutree(iris.hclust, 3) # we want 3 groups
> head(iris.3) # show cluster numbers
[1] 1 1 1 1 1 1
> Misclass(iris.3, iris[, 5], best=TRUE) # shipunov
Best classification table:
    obs
pred setosa versicolor virginica
   1     50          0         0
   2      0         50        16
   3      0          0        34
Misclassification errors (%):
    setosa versicolor  virginica
         0          0         32
Mean misclassification error: 10.7%
```

`Misclass()` calculates the *confusion matrix*, which is a simple way to assess the effectiveness of the classification (we added `best=TRUE` because `cutree()` does not know species names).

As you can see from the confusion matrix, 32% of *Iris virginica* were misclassified. The last is possible to improve, if we change either distance metric, or clustering method. For example, Ward's method of clustering gives more separated clusters and slightly better misclassification rates. Please **try** it yourself.

Note that Ward method minimizes the variance between clusters and works better with Euclidean distances. Sometimes, you need to do something like `dist(dist(data, method=...))` which converts non-Euclidean distance (for eexample, binary) to Euclidean. By the way, this transformation allows also to use `cmdscale()` with non-metric distances.

There are many other indices which allow to compare clusterings. For example, *adjusted Rand index* measures similarity between clusters:

```
> Adj.Rand(iris.3, iris[, 5]) # shipunov
[1] 0.7322981
```

* * *

Hierarchical clustering does not by default return any variable importance. However, it is still possible to assist the feature selection with heatmap (Fig. 8.18):

```
> library(cetcolor)
> heatmap(aa, col=cet_pal(12, "coolwarm"), margins=c(9, 6))
```

(Here we also used `cetcolor` package which allows to create perceptually uniform color palettes.)

Heatmap separately clusters rows and columns and places result of the `image()` function in the center. Then it become visible which characters influence which object clusters and *vice versa*. For example, Mars and Venus cluster together mostly because of similar levels of carbon dioxide.

* * *

we did not plot the dendrogram of irises above because there are too many of them. Of course, we can select, say, every 5th and then dendrogram start to be more readable. Another method is to use function `Ploth()` (Fig. 8.19):

```
> Ploth(iris.hclust, col=as.numeric(iris[, 5]),
+ pch=16, col.edges=TRUE, horiz=TRUE, leaflab="none") # shipunov
> legend("topleft", fill=1:nlevels(iris[, 5]),
+ legend=levels(iris[, 5]))
```

Another `Ploth()` example shows how just to rotate dendrogram. Please **check** the following yourself:

```
> oldpar <- par(mar=c(2, 0, 0, 4))
> tox.dist <- as.dist(1 - abs(tox.cor))
```

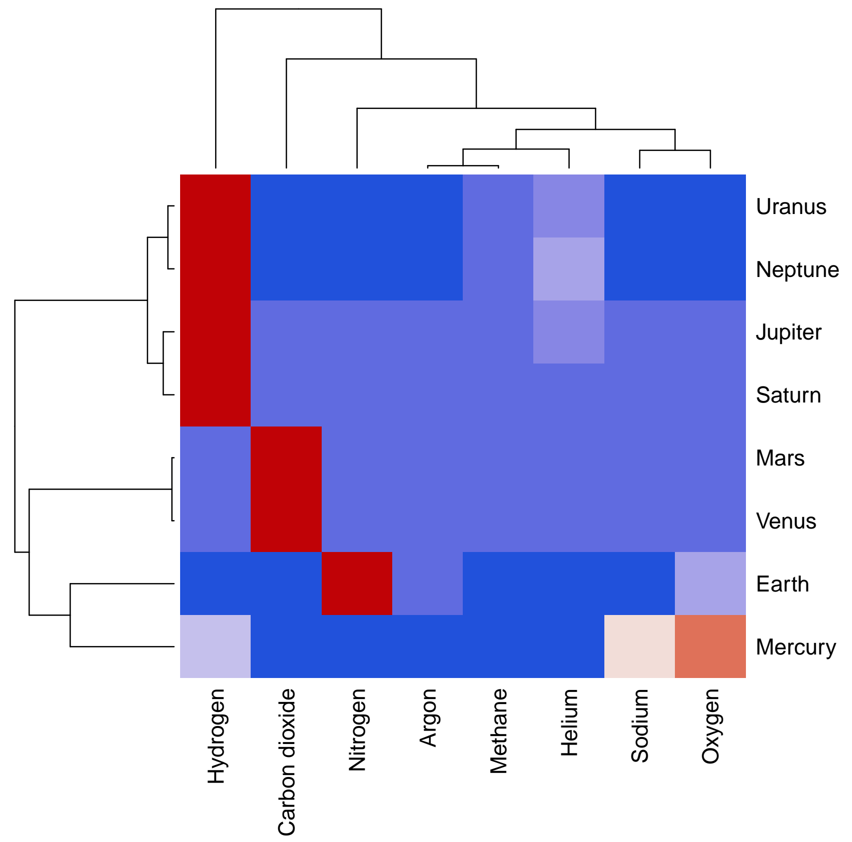


Figure 8.18: Clustering heatmap for atmosphere data.

```
> Ploth(hclust(tox.dist, method="ward.D"), horiz=TRUE) # shipunov
> par(oldpar)
```

(This is also a demonstration of how to use correlation for the distance. As you will see, the same connection between Caesar salad, tomatoes and illness could be visualized with dendrogram. There visible also some other interesting relations.)

> Make the hierarchical classification of beer varieties. Data was collected in 2001, in Russia and written to the `beer.txt` file, characters used described in the `beer_c.txt` file.

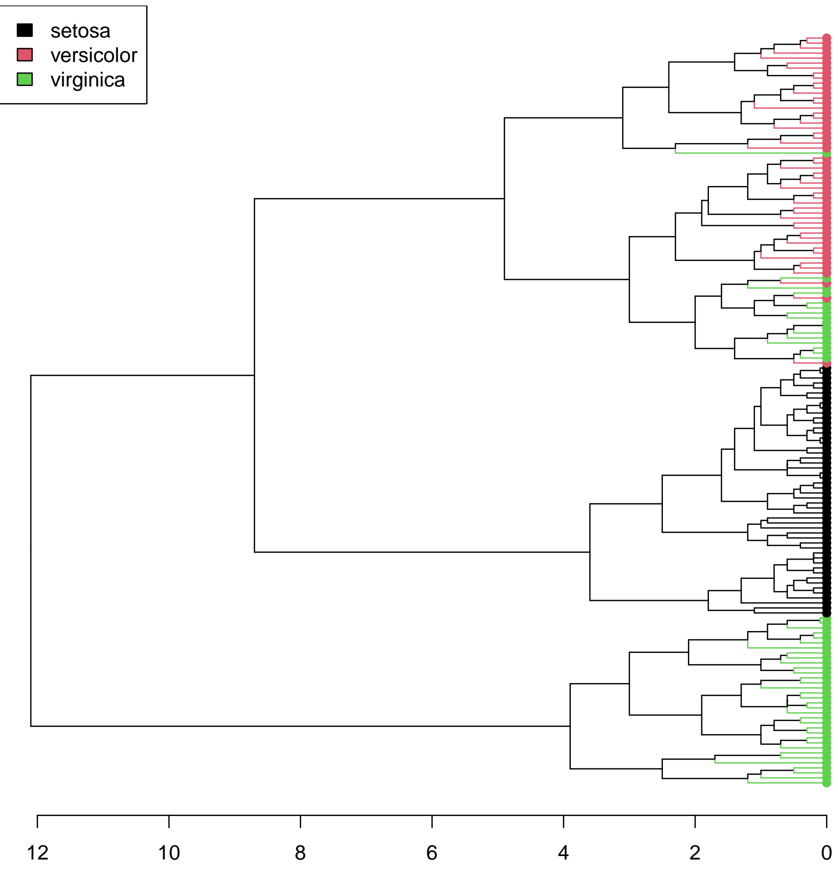


Figure 8.19: Hierarchical clustering of `iris` data.

> Planet Aqua is entirely covered by shallow water. This ocean is inhabited with various flat organisms (Fig. 8.20). These creatures (we call them "kubricks") can photosynthesize and/or eat other organisms or their parts (which match with the shape of their mouths), and move (only if they have no stalks). Provide the dendrogram for kubrick species based on result of hierarchical clustering.

8.2.4 How to know the best clustering method

Hierarchical cluster analysis and relatives (e.g., phylogeny trees) are visually appealing, but there are three important questions which need to be solved: (1) which distance is the best (this also relevant to other distance-based methods); (2) which

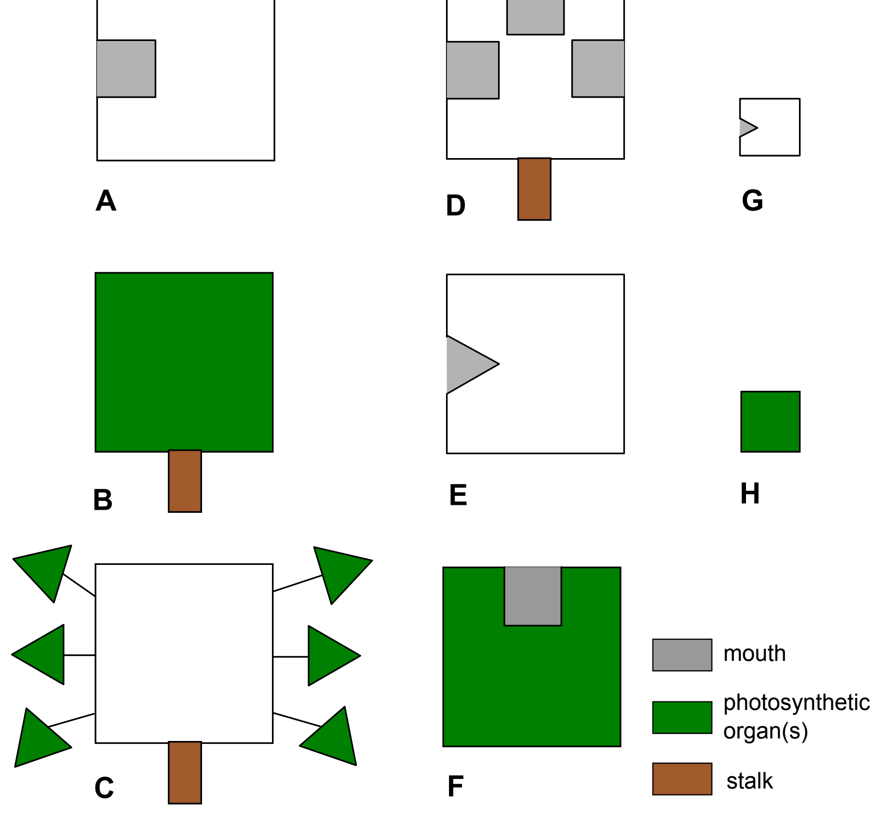


Figure 8.20: Eight species of kubricks.

hierarchical clustering method is the best; and (3) how to assess the quality of clusters.

Second question is relatively easy to answer. We can calculate consistency between distance object and hierachical clusterization as correlation between initial distances and distances from *cophenetic structure* (function `cophenetic()`) of the dendrogram.

Function `PlotBest.hclust()` helps to choose the most consistent clustering method:

```
> PlotBest.hclust(as.dist(m1.s)) # shipunov
```

(**Make and review** this plot yourself. Which clustering is better?)

Note, however, these "best" scores are **not** always best for you. For example, one might still decide to use `ward.D` regardless to its score, because it makes clusters sharp and visually separated.

* * *

To choose the best distance method, one might use the visually similar approach:

```
> PlotBest.dist(m1) # shipunov
```

(Again, please **review** the plot yourself.)

It just visualizes the correlation between multidimensional scaling of distances and principal component analysis of raw data. Nevertheless, it is still useful.

* * *

Another approach to choose the best method is to make artificial clusters and check how different methods perform. This is how function `Gen.cl.data()` works:

```
> palette(c("#377EB8", "#FF7F00", "#4DAF4A", "#F781BF", "#A65628",
+ "#984EA3", "#999999", "#E41A1C", "#DEDE00", "#000000"))
> nsam <- 500 # how many points to use
> K <- 3 # how many clusters to target
> noisy.moons <- Gen.cl.data(type="moons", N=nsam, noise=0.05)
> no.struct <- list(samples=cbind(runif(nsam), runif(nsam)),
+ labels=rep(1, nsam))
> varied <- Gen.cl.data(type="blobs", N=nsam, bnoise=c(1, 2.5, 0.5))
> oldpar <- par(mfrow=c(3, 3), mar=c(1, 1, 2, 1), xaxt="n", yaxt="n")
> for (n in c("no.struct", "noisy.moons", "varied")) {
+ newlabels <- cutree(hclust(dist(get(n)$samples),
+ method="ward.D2"), k=K)
+ plot(get(n)$samples, col=newlabels, pch=19,
+ main=paste0("Ward: ", n))
+ newlabels <- cutree(hclust(dist(get(n)$samples),
+ method="single"), k=K)
+ plot(get(n)$samples, col=newlabels, pch=19,
+ main=paste0("Single linkage: ", n))
+ newlabels <- kmeans(get(n)$samples, centers=K)$cluster
+ plot(get(n)$samples, col=newlabels, pch=19,
+ main=paste0("k-means: ", n))
+ }
> par(oldpar)
> palette("default")
```

On the Fig. 8.21, it is clearly visible that single linkage works best for intersecting objects and also does not reveal clusters when they are absent (some dots are colored

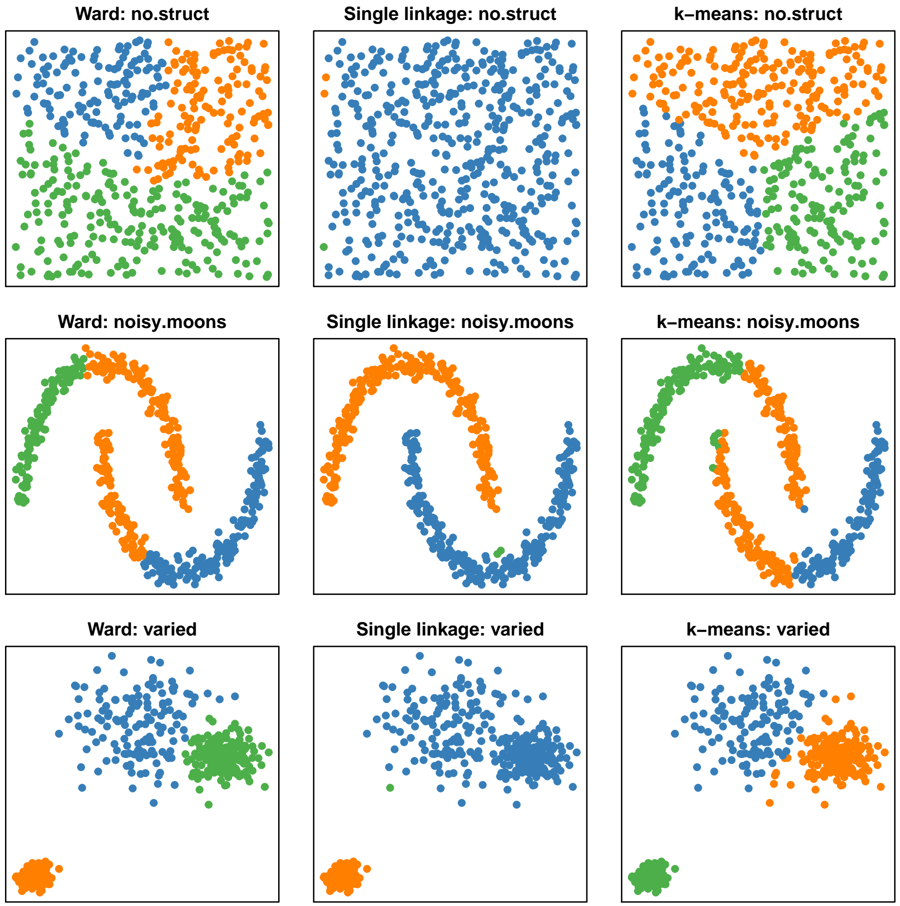


Figure 8.21: Three types of clusterings on three types of artificial clusters.

but it is only because we forced three clusters). Ward's method, in turn, is good to separate the blurred data.

8.2.5 How to compare clusterings

Hierarchical clustering are dendrograms and it is not easy to compare them "out of the box". We can, for example, employ methods associated with biological phylogenies (because phylogeny trees are essentially dendrograms).

Suppose that there are two clusterings:

```
> aa.d1 <- hclust(dist(aa))
```

```
> aa.d2 <- hclust(as.dist(1 - abs(cor(atmospheres, method="s"))),
+ method="ward.D") # shipunov
```

Library ape has `dist.topo()` function which calculates topological distance between trees, and library phangorn calculates more indexes:

```
> library(ape)
> aa.ph1 <- unroot(as.phylo(aa.d1)) # convert
> aa.ph2 <- unroot(as.phylo(aa.d2))
> dist.topo(aa.ph1, aa.ph2)
      tree1
tree2     2
> phangorn::treedist(aa.ph1, aa.ph2)
     symmetric.difference   branch.score.difference
                 2.000000                109.201423
     path.difference
 3.872983
quadratic.path.difference
               598.444274
```

Next possibility is to plot two trees side-by-side and show differences with lines connecting same tips (Fig. 8.22):

```
> ass <- cbind(aa.ph1$tip.label, aa.ph1$tip.label)
> aa.ph2r <- rotate(aa.ph2, c("Earth", "Neptune"))
> cophyloplot(aa.ph1, aa.ph2r, assoc=ass, space=30, lty=2)
```

(Note that this plot is just a visualization and could be misleading. Sometimes, you need to rotate some branches with `rotate()` function. Rotation does not change dendrogram.)

There is also possible to plot *consensus tree* which shows only those clusters which appear in both clusterings:

```
> plot(consensus(aa.ph1, aa.ph2r))
```

(Please **make** this plot yourself.)

* * *

Heatmap could also be used to visualize similarities between two dendrograms:

```
> aa12.match <- Hclust.match(aa.d1, aa.d2) # shipunov
> library(cetcolor)
> cols <- cet_pal(max(aa12.match), "blues")
```

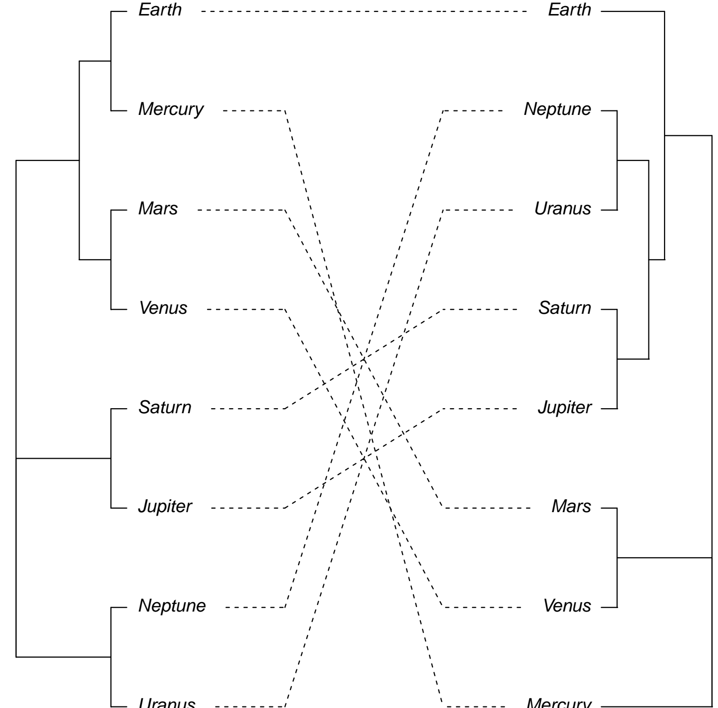


Figure 8.22: Side-by-side dendrogram plot for athmosphere data.

```
> heatmap(aa12.match, scale="none", col=cols)
```

(`Hclust.match()` counts matches between two dendrograms (if they utilize the the same data) and then `heatmap()` plots these counts as colors, and also supplies the consensus configuration as two identical dendrograms on the top and on the left. Please **make** this plot yourself.)

* * *

If we find a way to "reverse" hierarchical clustering and make a matrix from it, this will help, for example, to merge different clustering and calculate the joint result. Function `MRH()` (Matrix Representation of Hierarchical clustering) from `shipunov` package can do that:

```
> aa.h <- hclust(dist(t(atmospheres)))
> MRH(aa.h, method="branches") # shipunov
        [,1] [,2] [,3] [,4] [,5] [,6] [,7]
```

```
Mercury    0    0    0    0    1    1    1
Venus      1    0    0    0    0    1    1
Earth      0    0    0    0    1    1    1
Mars       1    0    0    0    0    1    1
Jupiter    0    0    1    1    0    0    1
Saturn     0    0    1    1    0    0    1
Uranus     0    1    0    1    0    0    1
Neptune    0    1    0    1    0    0    1
```

Method `branches` returns all clusters as binary columns. Please **check** yourself how other `MRH()` methods work.

* * *

Both multidimensional scaling and hierarchical clustering are distance-based methods. Please **make and review** the following plot (from the `vegan3d` package) to understand how to use them together:

```
> m1.dist <- as.dist(m1.s)
> m1.hclust <- hclust(m1.dist)
> m1.c <- cmdscale(m1.dist)
> library(vegan3d)
> orditree3d(m1.c, m1.hclust, text=attr(m1.dist, "Labels"), type="t")
```

8.2.6 How good are resulted clusters

There are several ways to check how good are resulted clusters, and many are based on the *bootstrap replication* (see Appendix).

Function `Jclust()` presents a method to bootstrap cluster memberships and plot the consensus membership tree with support values (Fig. 8.23):

```
> (m1.j <- Jclust(m1, 3, iter=1000)) # shipunov
 Bootstrap support for 3 clusters, 1000 iterations:
 support cluster                       members
   87.25       2 Gnutyj, Leda, Slitnyj, Tajnik
   84.30       1   Bobrovyj, Ekslibris, Zakhar
   62.50       3      Malinovyj.Kruglyj, Verik
> plot(m1.j, rect.lty=2, sub="")
```

(Note that `Jclust()` uses `cutree()` and therefore works only if it "knows" the number of desired clusters. Since consensus result relates with cluster number, plots with different numbers of clusters will be different.)

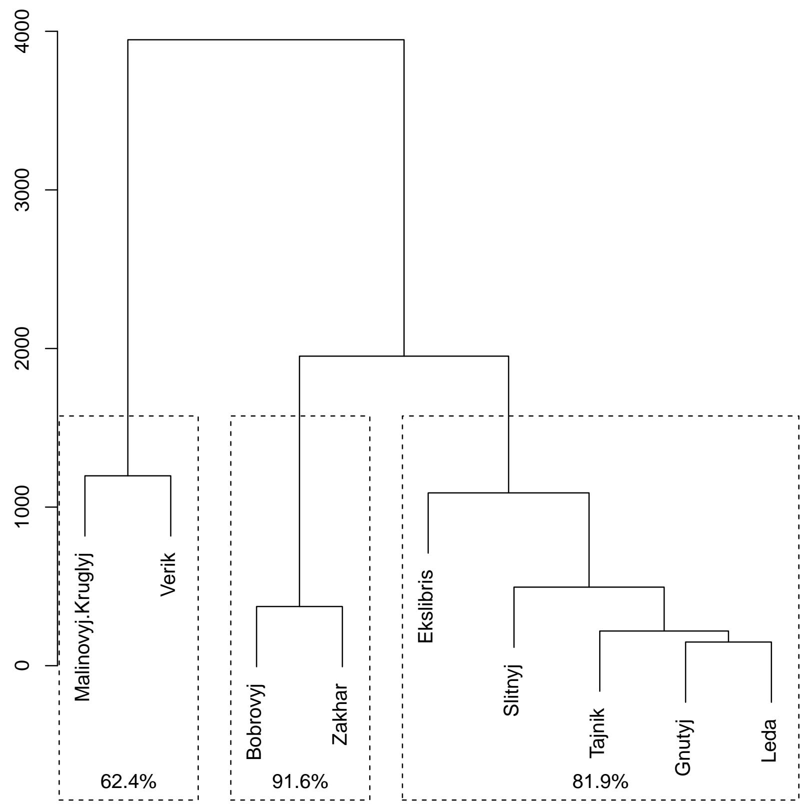


Figure 8.23: Bootstrap stability of 3-cluster solution for lake islands data

* * *

Another way is to use pvclust package which has an ability to calculate the support for clusters via bootstrap (Fig. 8.24):

```
> library(pvclust)
> m1.pvclust <- pvclust(t(m1), method.dist="manhattan",
+ nboot=100, parallel=TRUE)
Creating a temporary cluster...done:
socket cluster with 3 nodes on host 'localhost'
Multiscale bootstrap... Done.
> plot(m1.pvclust, col.pv=c(au="red", bp="darkgreen", edge=0), main="")
```

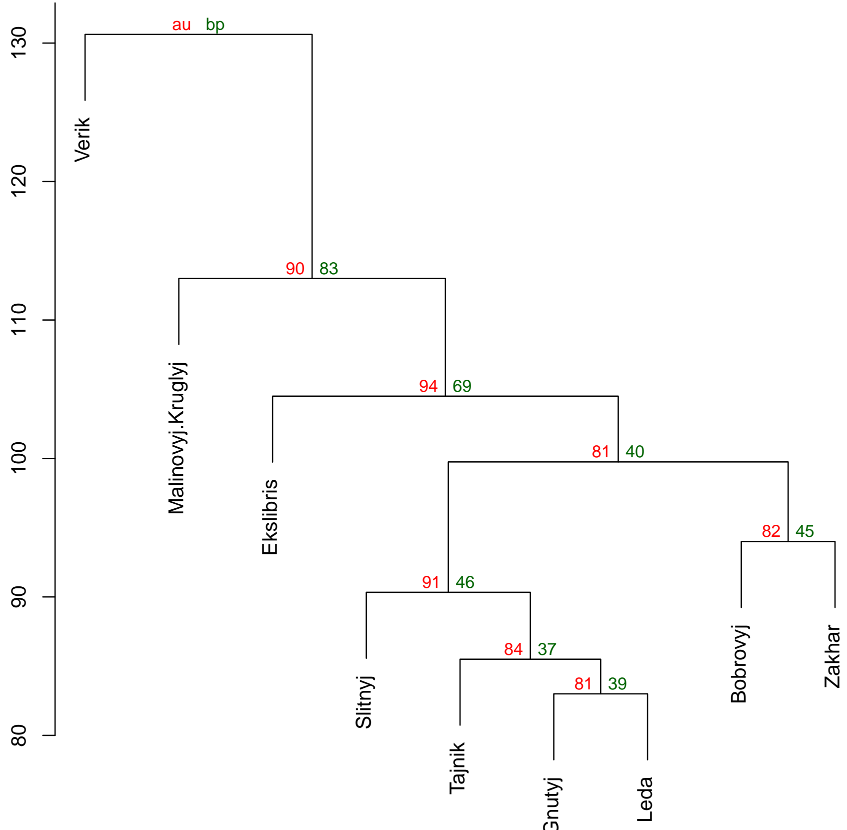


Figure 8.24: Dendrogram with supporting values (`pvclust` package)

(Function `pvclust()` clusterizes columns, not rows, so we have to transpose data again. On the plot, numerical values of cluster stability (`au` and `bp`) are located above each node. The closer are these values to 100, the better.)

Note that when number of variables is too low (less then "sacred" 25–35), bootstrap values (like `bp` in the above case) tend to be smaller then expected. In that case, *hyper-binding* (duplication of variablles like `cbind(data, data)` might be recommened as a workaround.)

* * *

There is also `BootA()` function in `shipunov` package which allows to bootstrap clustering with methods from phylogenetic package `ape`:

```
> m1.ba <- BootA(m1, FUN=function(.x) # shipunov
+ as.phylo(hclust(dist(.x, method="minkowski"),
+ method="average")), iter=100, mc.cores=4)
Running parallel bootstraps... done.
Calculating bootstrap values... done.
> plot(m1.ba$boot.tree, show.node.label=TRUE)
> plot(m1.ba$cons.tree) # majority-rule consensus
```

(This method requires to make an anonymous function which uses methods you want. It also plots both consensus tree (without support values) and original tree with support values. Please **make** these trees. Note that by default, only support values greater then 50% are shown.)

Function `Bclust()` from the same package does the simple bootstrap of the hierarchical clustering. The main difference from `Jclust()` is that it bootstraps clusters themselves, not memberships. Therefore, it is possible to check the support for all clusters within one run:

```
> aa.bcl <- Bclust(t(atmospheres), iter=100) # shipunov
...
> plot(aa.bcl$hclust)
> Bclabels(aa.bcl$hclust, aa.bcl$values, pos=3, offset=0.1) # shipunov
```

(Please **review** this plot yourself.)

`Bclust()` is more atomized and therefore more flexible than other cluster bootstrapping functions. Like `BootA()`, it allows to plot consensus tree using `$consensus` component.

8.2.7 Making groups: *k*-means and friends

There are many other ways of clustering. Most of them are *partitioning* methods which provide only cluster membership. They also frequently require the desired number of clusters (thus, they use some learning). For example, *k-means clustering* tries to obtain the *a priori* specified number of clusters from the raw data. *k*-means calculates distances internally so does not need the distance matrix to be supplied. It does not plot trees; instead, for every object it returns the number of its cluster:

```
> iris.km <- kmeans(iris[, -5], nstart=5, centers=3) # 3 clusters
> str(iris.km$cluster)
 int [1:150] 1 1 1 1 1 1 1 1 1 1 ...
> Misclass(iris$Species, iris.km$cluster, best=TRUE)
Best classification table:
            obs
```

```
pred          1  3  2
  setosa     50  0  0
  versicolor  0 48  2
  virginica   0 14 36
Misclassification errors (%):
   1    3    2
 0.0 22.6  5.3
Mean misclassification error: 9.3%
```

(Note that *K*-means is the relatively stable but *stochastic* method, so in your case, results might be different. To make results more robust, we added here `nstart=5` which sets five random starts instead of one.)

* * *

Spectral clustering from `kernlab` package is the method capable to separate non-linearly tangled elements:

```
> library(kernlab)
> data(spirals)
> set.seed(1)
> sc <- specc(spirals, centers=2)
> plot(spirals, col=sc, xlab="", ylab="")
```

Kernel methods (like spectral clustering) recalculate the primary data to make it more suitable for the analysis (for example, make it linear). Support vector machines (SVM, see below) is another example of kernel methods. Package `kernlab` contains kernel variants of other multivariate methods, for example, kernel PCA (function `kpca()`).

* * *

Next group of clustering methods is based on *fuzzy logic* and takes into account the *fuzziness* of relations. There is always the possibility that particular object classified in the cluster A belongs to the different cluster B, and fuzzy clustering tries to measure this possibility:

```
> library(cluster)
> iris.f <- fanny(iris[, 1:4], 3)
> head(data.frame(sp=iris[, 5], iris.f$membership))

      sp        X1         X2         X3
1 setosa 0.9142273 0.03603116 0.04974153
```

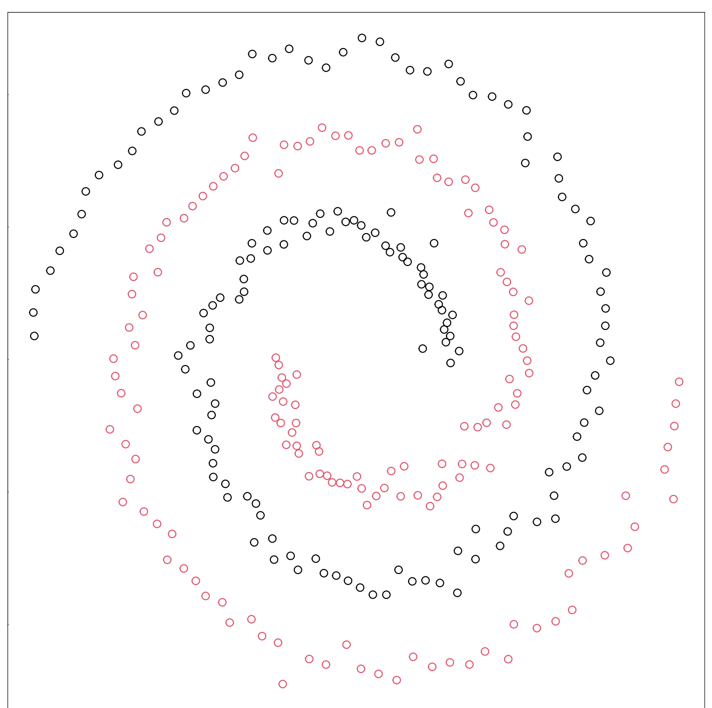


Figure 8.25: Kernel-based spectral clustering is capable to separate two spirals.

```
2 setosa 0.8594576 0.05854637 0.08199602
3 setosa 0.8700857 0.05463714 0.07527719
4 setosa 0.8426296 0.06555926 0.09181118
5 setosa 0.9044503 0.04025288 0.05529687
6 setosa 0.7680227 0.09717445 0.13480286
```

Textual part of the `fanny()` output is most interesting. Every row contains multiple membership values which represent the *probability of this object to be in the particular cluster*. For example, sixth plant most likely belongs to the first cluster but there is also visible attraction to the third cluster. In addition, `fanny()` can round memberships and produce hard clustering like other cluster methods:

```
> Misclass(iris.f$clustering, iris[, 5], best=TRUE) # shipunov
Best classification table:
    obs
pred setosa virginica versicolor
   1     50         0          0
   2      0        41          4
```

```
   3      0         9         46
Misclassification errors (%):
    setosa  virginica versicolor
         0         18          8
Mean misclassification error: 8.7%
```

(We used `best=TRUE` to find the best corresponding partition because `fanny()` does not know species names.)

8.2.8 How to know cluster numbers

Partitioning methods usually want to know the number of clusters before they start. So how to know *a priori* how many clusters are in data?

The visual analysis of *banner plot* (invented by Kaufman & Rousseeuw, 1990) could predict this number (Fig. 8.26):

```
> library(cluster)
> eq.a <- agnes(eq[, -1])
> plot(eq.a, which=1, col=c(0, "maroon"))
```

White bars on the left represent unclustered data, maroon lines on the right show height of possible clusters. Therefore, two clusters is the most natural solution, four clusters should be the next possible option.

* * *

Model-based clustering allows to determine how many clusters present in data and also cluster membership. The method assumes that clusters have the particular nature and multidimensional shapes:

```
> library(mclust)
...
> iris.mclust <- Mclust(iris[, -5])
fitting ...
> summary(iris.mclust) # 2 clusters
----------------------------------------------------
Gaussian finite mixture model fitted by EM algorithm
----------------------------------------------------

Mclust VEV (ellipsoidal, equal shape) model with 2 components:

 log.likelihood   n df       BIC       ICL
```

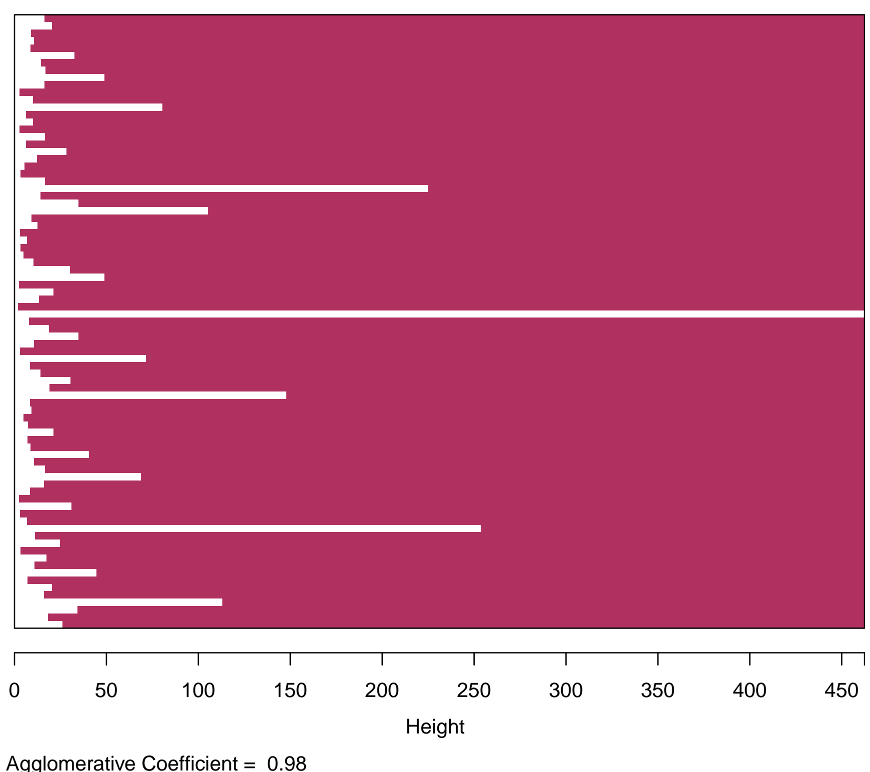


Figure 8.26: Banner plot. White bars suggest possible cluster partitions.

```
      -215.726 150 26 -561.7285 -561.7289

Clustering table:
  1   2
 50 100
> iris.mclust$classification
  [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 ...
```

(It reveals only two clusters because in `iris` data two species are much more similar then the third one.)

* * *

DBSCAN is the powerful algorithm for the big data (like raster images which consist of billions of pixels) and there is the R package with the same name (in lowercase). DBSCAN reveals how many clusters are in data at particular resolution:

```
> library(dbscan)
> kNNdistplot(iris[, -5], k = 5) # look on the knee
> abline(h=.5, col = "red", lty=2)
> (iris.dbscan <- dbscan(iris[, -5], eps = .5, minPts = 5))
DBSCAN clustering for 150 objects.
Parameters: eps = 0.5, minPts = 5
The clustering contains 2 cluster(s) and 17 noise points.
 0  1  2
17 49 84
...
> Misclass(iris.dbscan$cluster, iris$Species, best=TRUE, ignore="0")
Best classification table:
  setosa virginica versicolor
1     49         0          0
2      0        40         44
       0         0          0
Misclassification errors (%):
    setosa  virginica versicolor
         0          0        100
Mean misclassification error: 33.3%
Note: data contains NAs
```

(We used here another feature of `Misclass()` which allows to "ignore" (actually, convert to `NA`) some class labels; in that case, zeros which which designate "noise" have been ignored.)

```
> plot(iris.p, type="n", xlab="", ylab="")
> text(iris.p, labels=abbreviate(iris[, 5], 1,
+ method="both.sides"), col=iris.dbscan$cluster+1)
```

(Plots above are not shown, please **make** then yourself.)

Note that while DBSCAN was not able to recover all three species, it recovered well two species groups, and also placed marginal points in the "noise" group. Parameter `eps` allows to change "resolution" of clustering and to find more (or less) clusters.

Like *k*-means, DBSCAN is based on specifically calculated proximities. It is relates also with t-SNE (see above) and with supervised methods based on proximity (like *k*-NN, see below).

DBSCAN can also be *hierarchical* and return dendrograms:

```
> iris.hdbscan <- hdbscan(iris[, -5], minPts=10)
> plot(iris.hdbscan$hc)
```

(Please **check** this plot yourself.)

Note that *k*-means and DBSCAN are based on specifically calculated *proximities*, not directly on distances.

> Data `stars` contains information about 50 brightest stars in the night sky, their location and constellations. Please use DBSCAN to make artificial constellations on the base of star proximity. How are they related to real constellations?
>
> Note that location (right ascension and declination) is given in degrees or hours (sexagesimal system), they must be converted into decimals.

* * *

"Mean-shift" method searches for *modes* within data, which in essence, is similar to finding proximities. The core mean-shift algorithm is slow so approximate "blurring" version is typically preferable:

```
> library(MeanShift)
> bandwidth <- quantile(dist(iris[, -5]), 0.25)
> (bmsClustering(t(iris[, -5]), h=bandwidth))
Running blurring mean-shift algorithm...
Blurring mean-shift algorithm ran successfully.
Finding clusters...
The algorithm found 3 clusters.
$components
                 mode1    mode2    mode3
Sepal.Length 5.0007162 6.179920 7.104628
Sepal.Width  3.4178891 2.869793 3.070013
Petal.Length 1.4701932 4.800869 5.957860
Petal.Width  0.2442342 1.640072 2.046728
$labels
...
 [38] 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 ...
```

Function `meanShift()` from `meanShiftR` package is generally faster but less flexible.

* * *

There are some simple ways to check the possible number of clusters. We can, for example, employ the dimension reduction or even use the "raw" distance matrix:

```
> aa <- prcomp(iris[, -5])$x[, 1] # only PC1
> Histr(aa, overlay="density", breaks=10)
> bb <- c(dist(iris[, -5])) # coverts "distance" object into vector
> Histr(bb, overlay="density", breaks=10)
```

(Please **check** these plots yourself. Two peaks are easily visible; this gives a hope that the data contains at least two clusters.)

Package `kpeaks` (see below) contains functions which automatize the search for the best cluster number.

* * *

Finally, the integrative package `NbClust` allows to use many methods to assess the putative number of clusters:

```
> library(NbClust)
> iris.nb <- NbClust(iris[, -5], method="ward.D") # wait!
...
* Among all indices:
* 9 proposed 2 as the best number of clusters
* 10 proposed 3 as the best number of clusters
* 3 proposed 6 as the best number of clusters
* 1 proposed 10 as the best number of clusters
* 1 proposed 15 as the best number of clusters
                   ***** Conclusion *****r
* According to the majority rule, the best number of clusters is  3
```

8.2.9 Use projection pursuit for clustering

Projection pursuit is a method close to PCA and LDA (see below) and analogous to rotating RGL plot with mouse to find the best projection. It is useful as is, and also in connection with clustering methods because it frequently outputs well separated clusters:

```
> library(PPCI)
> iris.mcdr <- mcdr(iris[, -5], p=3) # dimension reduction
> plot(iris.mcdr$fitted, col=iris$Species)
> iris.mcdc <- mcdc(iris[, -5], K=3) # clustering
> plot(iris.mcdc, labels=iris$Species)
> Misclass(iris.mcdc$cluster, iris$Species, best=TRUE) # shipunov
Best classification table:
    obs
pred versicolor setosa virginica
   1         48      0         1
   2          0     50         0
   3          2      0        49
Misclassification errors (%):
versicolor     setosa  virginica
         4          0          2
Mean misclassification error: 2%
```

Above, we used *maximum clusterability* approach. It gives one of the lowest misclassification rates and also provides nicely separated groups on the plots (please **review** them yourself.)

8.2.10 How to compare different ordinations

Most of classification methods result in some ordination, 2D plot which includes all data points. This allow to compare them with Procrustes analysis (see Appendix for more details) which rotates ans scales one data matrix to make in maximally similar with the second (target) one. Let us compare results of classic PCA and t-SNE:

```
> library(Rtsne)
> iris.unique <- unique(iris)
> tsne.out <- Rtsne(as.matrix(iris.unique[, -5]))
> irisu.pca <- prcomp(iris.unique[, -5])
> irisu.p <- irisu.pca$x[, 1:2]
> library(vegan)
> irisu.pr <- procrustes(irisu.p, tsne.out$Y)
> plot(irisu.pr, ar.col=iris.unique$Species, pch=" ",
+ xlab="", ylab="", main="") # arrows point to target (PCA)
> points(irisu.pr$Yrot, col=iris$Species, pch=21, bg="white")
> with(iris.unique, legend("topright", lty=1, col=1:nlevels(Species),
+ legend=levels(Species), bty="n"))
> legend("bottomright", lty=2:1, legend=c("PCA", "t-SNE"), bty="n")
```

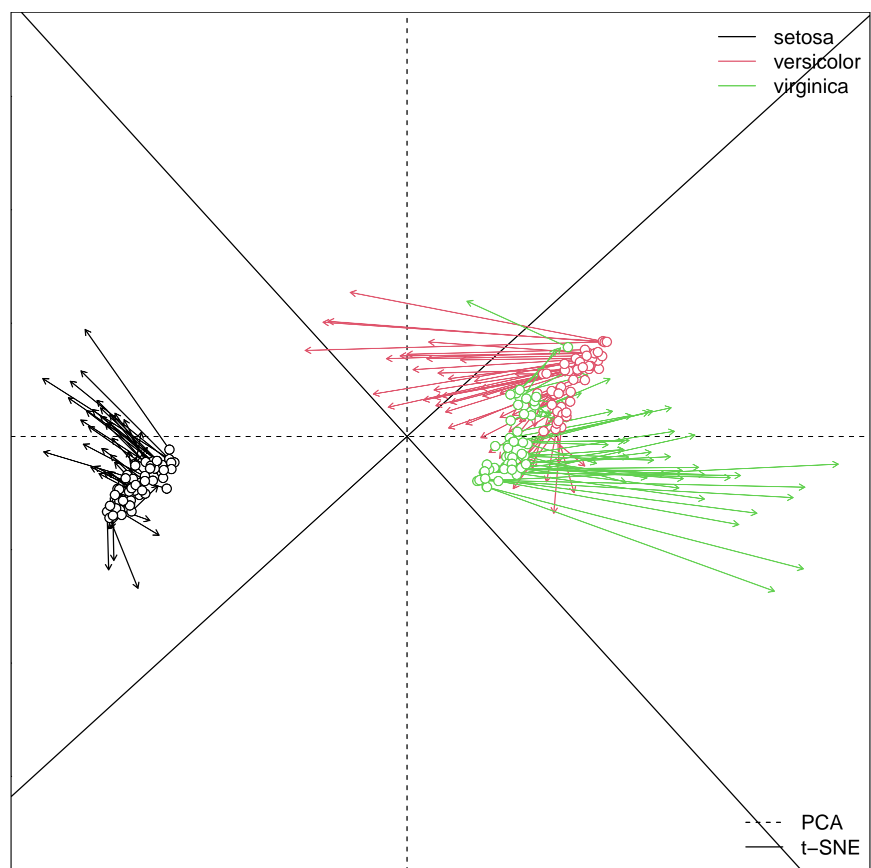


Figure 8.27: Procrustes plot which show t-SNE ordination against the target PCA ordination.

Resulted plot (Fig. 8.27) shows how dense are points in t-SNE and how PCA spreads them. Which of methods makes better grouping? **Find** it yourself.

8.3 Answers to exercises

Answer to the stars question.

First, load the data and as suggested above, convert coordinates into decimals:

```
> s50 <- read.table("data/stars.txt", h=TRUE, sep="\t",
+ as.is=TRUE, quote="")
> str(s50)
```

```
'data.frame': 50 obs. of  13 variables:
...
 $ RA       : chr  "06 45 08.9" "06 23 57.1" ...
 $ DEC      : chr  "-16 42 58" "-52 41 45" ...
...
> RA10 <- as.numeric(substr(s50$RA, 1, 2)) + # hours
+ as.numeric(substr(s50$RA, 4, 5))/60 + # minutes
+ as.numeric(substr(s50$RA, 7, 10))/3600 # seconds
> DEC10 <- sign(as.numeric(substr(s50$DEC, 1, 3))) *
+ (as.numeric(substr(s50$DEC, 2, 3)) +
+ as.numeric(substr(s50$DEC, 5, 6))/60 +
+ as.numeric(substr(s50$DEC, 8, 9))/3600)
> coo <- cbind(RA10, DEC10)
```

Next, some preliminary plots (please **make** them yourself):

```
> oldpar <- par(bg="black", fg="white", mar=rep(0, 4))
> plot(coo, pch="*", cex=(3 - s50$VMAG)) # constellation plot
> Hulls(coo, as.numeric(factor(s50$CONSTEL)), # shipunov
+ usecolors=rep("white", nlevels(factor(s50$CONSTEL))))
> points(runif(100, min(RA10), max(RA10)),
+ runif(100, min(DEC10), max(DEC10)), pch=".") # random "stars"
> par(oldpar)
> plot(coo, type="n") # constellation names
> text(coo, s50$CONSTEL.A)
> plot(coo, type="n") # star names
> text(coo, s50$NAME)
```

Now, load dbscan package and try to find where number of “constellations” is maximal:

```
> library(dbscan)
> for (eps in 1:20) cat(c(eps, ":",
+ names(table(dbscan(coo, eps=eps)$cluster))), "\n")
...
7 : 0 1
8 : 0 1 2
9 : 0 1 2 3 # maximum reached
10 : 0 1 2 3
11 : 0 1 2
12 : 0 1 2
13 : 0 1 2
14 : 0 1
```

```
...
```

Plot the prettified "night sky" (Fig. 8.28) with new "constellations" found:

```
> s50.db <- dbscan(coo, eps=9)
> oldpar <- par(bg="black", fg="white", mar=rep(0, 4))
> plot(coo, pch=8, cex=(3 - s50$VMAG))
> Hulls(coo, s50.db$cluster, # shipunov
+ usecolors=c("black", "white", "white", "white"))
> points(runif(100, min(RA10), max(RA10)),
+ runif(100, min(DEC10), max(DEC10)), pch=".")
> par(oldpar)
```

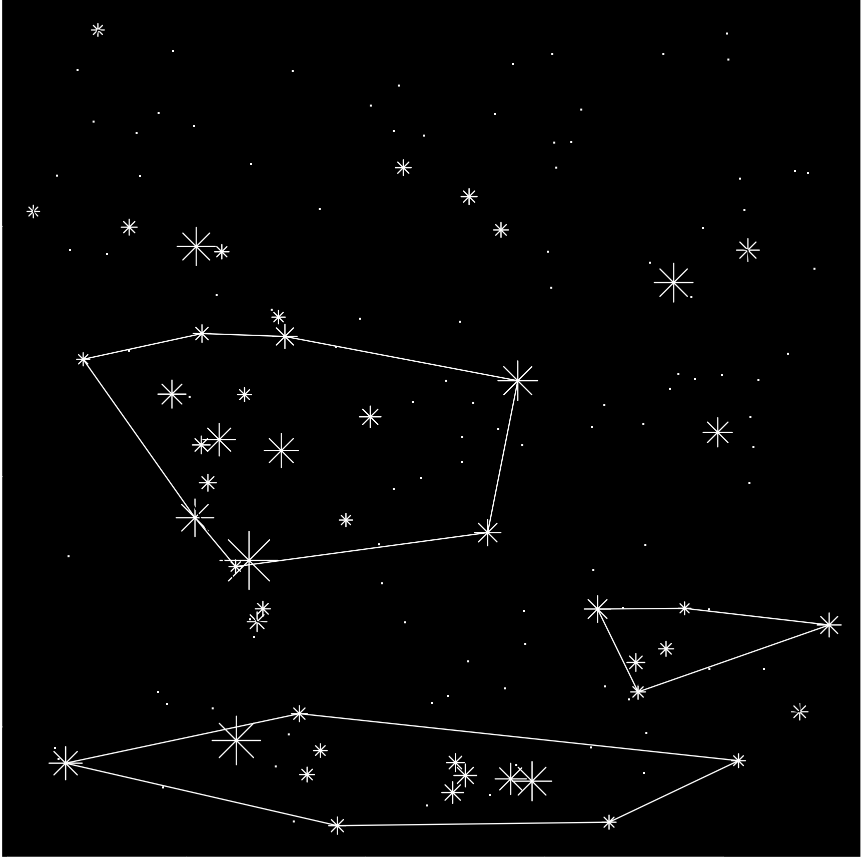


Figure 8.28: Fifty brightest stars with "constellations" found with DBSCAN.

To access agreement between two classifications (two systems of constellations) we might use adjusted Rand index which counts correspondences:

```
> Adj.Rand(as.numeric(factor(s50$CONSTEL)), s50.db$cluster) # shipunov
[1] 0.1061416
```

(It is of course low.)

* * *

Answer to the beer classification exercise. To make hierarchical classification, we need first to make the distance matrix. Let us look on the data:

```
> beer <- read.table("data/beer.txt", sep="\t", h=TRUE)
> head(beer)
              C01 C02 C03 C04 C05 C06 C07 C08 C09 C10
Afanas.light    0   1   0   1   0   0   0   0   1   0
Baltika.3       0   1   0   0   0   0   1   0   1   1
Baltika.6       0   1   1   1   1   0   1   0   1   1
Baltika.9       0   1   1   1   1   0   1   0   1   1
Bochk.light     0   1   0   0   1   1   0   1   1   1
Budweiser       0   1   1   0   1   0   1   0   1   1
...
```

Data is binary and therefore we need the specific method of distance calculation. We will use here Jaccard distance implemented in `vegdist()` function from the `vegan` package. It is also possible to use here other methods like "binary" from the core `dist()` function. Next step would be the construction of dendrogram (Fig. 8.29):

```
> library(vegan)
> beer.d <- vegdist(beer, "jaccard")
> plot(hclust(beer.d, method="ward.D"), main="", xlab="", sub="")
```

There are two big groups (on about 1.7 dissimilarity level), we can call them "Baltika" and "Budweiser". On the next split (approximately on 1.4 dissimilarity), there are two subgroups in each group. All other splits are significantly deeper. Therefore, it is possible to make the following hierarchical classification:

- Baltika group
  - Baltika subgroup: Baltika.6, Baltika.9, Ochak.dark, Afanas.light, Sibirskoe, Tula.hard
  - Tula subgroup: Zhigulevsk, Khamovn, Tula.arsenal, Tula.orig
- Budweiser group
  - Budweiser subgroup: Sinebryukh, Vena.porter, Sokol.soft, Budweiser, Sokol.light

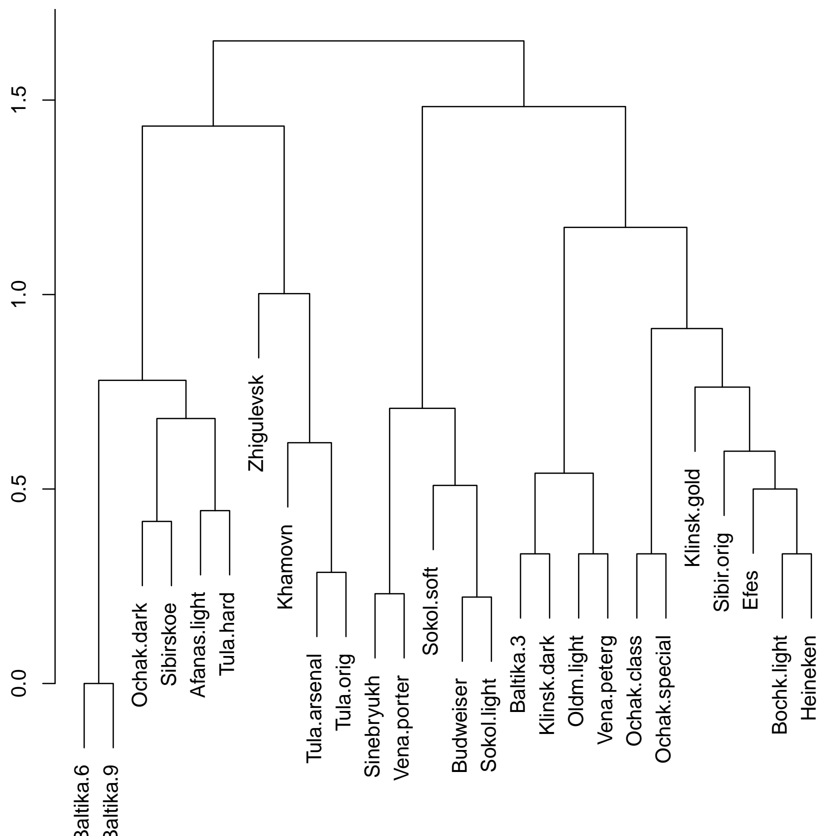


Figure 8.29: Hierarchical classification of Russian beer types.

– Ochak subgroup: Baltika.3, Klinsk.dark, Oldm.light, Vena.peterg, Ochak.class, Ochak.special, Klinsk.gold, Sibir.orig, Efes, Bochk.light, Heineken

It is also a good idea to check the resulted classification with any other classification method, like non-hierarchical clustering, multidimensional scaling or even PCA. The more consistent is the above classification with this second approach, the better.

* * *

Answer to the kubricks (Fig. 8.20) question. This is just a plan as you will still need to perform these steps individually:

1. Open R, open Excel or any spreadsheet software and create the data file. This data file should be the table where kubrick species are rows and characters are columns (variables). Every row should start with a name of kubrick (i.e., letter), and every column should have a header (name of character). For characters, short uppercased names with no spaces are preferable.

   The top left cell might stay empty. In every other cell, there should be either 1 (character present) or 0 (character absent). For the character, you might use "presence of stalk" or "presence of three mouths", or "ability to make photosynthesis", or something alike. Since there are 8 kubricks, it is recommended to invent $N+1$ (in this case, 9) characters.

2. Save your table as a text file, preferably tab-separated (use approaches described in the second chapter), then load it into R with `read.table(.., h=TRUE, row.names=1)`.

3. Apply hierarchical clustering with the distance method applicable for binary (0/1) data, for example `binary` from `dist()` or another method (like Jaccard) from the `vegan::vegdist()`.

4. Make the dendrogram with `hclust()` using the appropriate clustering algorithm.

In the data directory, there is an example data file, `kubricks.txt`. It is just an example so it is not necessarily correct and does not contain descriptions of characters. Since it is pretty obvious how to perform hierarchical clustering (see the "beer" example above), we present below two other possibilities.

First, we use MDS plus the MST, *minimum spanning tree*, the set of lines which show the shortest path connecting all objects on the ordination plot (Fig 8.30):

```
> kubricks <- read.table("data/kubricks.txt", h=TRUE, row.names=1)
> kubricks.d <- dist(kubricks, method="binary")
> kubricks.c <- cmdscale(kubricks.d)
> plot(kubricks.c, type="n", axes=FALSE)
> text(kubricks.c, labels=row.names(kubricks), cex=2)
> library(vegan)
> lines(spantree(kubricks.d), kubricks.c[, 1:2], lty=2)
```

(Package `emstreeR` has more advanced methods for minimal spanning trees.)

Second, we can take into account that kubricks are biological objects. Therefore, with the help of packages `ape` and `phangorn` we can try to construct the most parsimonious (i.e., shortest) *phylogeny tree* for kubricks. Let us accept that kubrick H is the *outgroup*, the most primitive one:

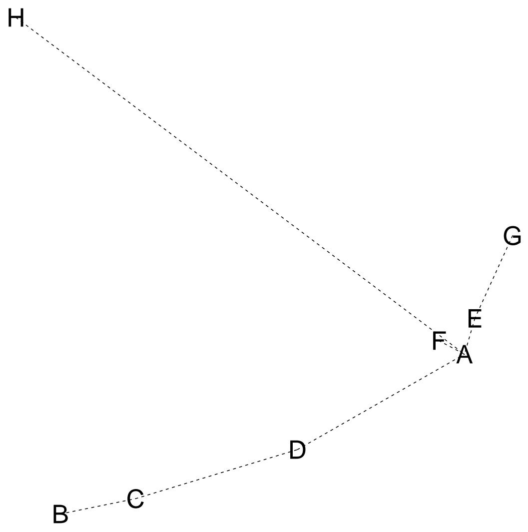


Figure 8.30: Minimum spanning tree of kubricks.

```
> library(phangorn)
> k <- as.phyDat(data.frame(t(kubricks)), type="USER",
+ levels = c(0, 1))
> kd <- dist.hamming(k) # Hamming distance for morphological data
> kdnj <- NJ(kd) # neighbor-joining tree for starter
> kp <- optim.parsimony(kdnj, k) # find most parsimonous tree
> ktree <- root(kp, outgroup="H", resolve.root=TRUE) # re-root
> plot(ktree, cex=2)
```

(**Make and review** this plot yourself.)

Chapter 9

Learn

Methods explained in this chapter frequently called “classification with learning”, “supervised classification”, “machine learning”, or just “classification”. All of them are based on the idea of *learning*:

> … He scrambled through and rose to his feet. … He saw nothing but colours—colours that refused to form themselves into things. Moreover, he knew nothing yet well enough to see it: *you cannot see things till you know roughly what they are*[1]. His first impression was of a bright, pale world—a watercolour world out of a child’s paint-box; a moment later he recognized the flat belt of light blue as a sheet of water, or of something like water, which came nearly to his feet. They were on the shore of a lake or river…
>
> C.S.Lewis. *Out of the Silent Planet*.

Most learning methods have similar workflow. Firstly, small part of data where identity is already known (*training dataset*) used to develop (fit) the model of classification (Fig 9.1). Secondly, this model is used to classify objects with unknown identity (*testing dataset*). Finally, in most of these methods, it is possible to estimate the quality of the classification and also assess the significance of the every character.

Let us first to create training and testing datasets from `iris` data:

```
> iris.train <- iris[seq(1, nrow(iris), 5), ]
> iris.unknown <- iris[-seq(1, nrow(iris), 5), ]
```

[1]Emphasis mine.

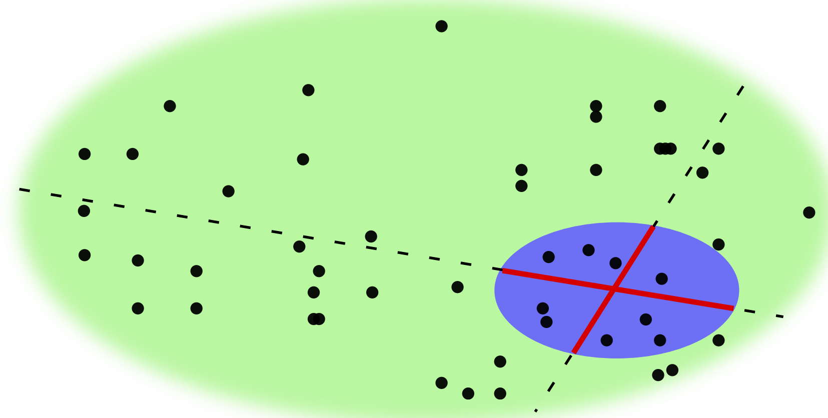


Figure 9.1: Graphic representation of the statistical machine learning. Blue is a training dataset, red lines is classification model, green is a testing dataset, dashed lines show how prediction works.

Choosing every fifth row has a lot of sense for `iris` data because it is sorted by species. `iris.unknown` is of course the fake unknown so to use it properly, we must specify `iris.unknown[, -5]`. However, species information will help in cross-validation (see below).

9.1 Learning with regression

9.1.1 Linear discriminant analysis

One of the simplest methods of classification is the linear discriminant analysis (LDA). The basic idea is to create the set of linear functions which "decide" how to classify the particular object. It close to the idea of "supervised PCA".

```
> library(MASS)
> iris.lda <- lda(Species ~ . , data=iris.train)
> iris.predicted <- predict(iris.lda, iris.unknown[, 1:4])
> Misclass(iris.predicted$class, iris.unknown[, 5]) # shipunov
Classification table:
            obs
pred         setosa versicolor virginica
  setosa         40          0         0
  versicolor      0         40         7
```

```
  virginica      0          0        33
Misclassification errors (%):
    setosa versicolor  virginica
       0.0        0.0       17.5
Mean misclassification error: 5.8%
```

Training resulted in the hypothesis which allowed almost all plants (with an exception of seven *Iris virginica*) to be placed into the proper group. Please note that LDA does not require scaling of variables.

* * *

It is possible to check LDA results with inferential methods. Multidimensional analysis of variation (MANOVA) allows to understand the relation between data and model (classification from LDA):

```
> ldam <- manova(as.matrix(iris.unknown[, 1:4]) ~
+ iris.predicted$class)
> summary(ldam, test="Wilks")
           Df    Wilks approx F num Df den Df    Pr(>F)
iris.ldap   2 0.026878   145.34      8    228 < 2.2e-16 ***
Residuals 117
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

Important here are both p-value based on Fisher statistics, and also the value of Wilks' statistics which is the *likelihood ratio* (in our case, the probability that groups are *not different*).

It is also possible to check the relative importance of each character in LDA with ANOVA-like techniques:

```
> summary(aov(as.matrix(iris.unknown[, 1:4]) ~
+ iris.predicted$class))
 Response Sepal.Length :
                      Df   Sum Sq   Mean Sq  F value     Pr(>F)
iris.predicted$class   2 51.04735 25.523675 108.8983 < 2.22e-16 ***
Residuals            117 27.42257  0.234381
---
...
```

Non-parametric MANOVA-like analysis also exists, one of the best is probably the `adonis()` function from the `vegan` package:

```
> library(vegan)
> adonis(as.matrix(iris.unknown[, 1:4]) ~ iris.predicted$class)
...
Terms added sequentially (first to last)
                      Df SumsOfSqs MeanSqs F.Model      R2 Pr(>F)
iris.predicted$class   2   1.85808 0.92904  437.53 0.88206  0.001 ***
Residuals            117   0.24843 0.00212         0.11794
Total                119   2.10651                 1.00000
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

Note that all these ideas are applicable to other classification methods.

* * *

Finally, it is possible to visualize LDA results (Fig. 9.2) in a way similar to PCA:

```
> iris.lda2 <- lda(iris[, 1:4], iris[, 5])
> iris.ldap2 <- predict(iris.lda2, dimen=2)$x
> plot(iris.ldap2, type="n", xlab="LD1", ylab="LD2")
> text(iris.ldap2, labels=abbreviate(iris[, 5], 1,
+ method="both.sides"))
> Ellipses(iris.ldap2, as.numeric(iris[, 5]),
+ centers=TRUE) # shipunov
```

(Please note 95% confidence ellipses with centers.)

(To place all points on the plot, we simply used all data as training.)

* * *

Please note that while LDA was developed on biological material, this kind of data rarely meets two key assumptions of this method: (1) multivariate normality and (2) multivariate homoscedasticity. Even Fisher’s *Iris* data with which LDA was invented, does not fully meet these assumptions. Therefore, we recommend to use LDA with caution. On the other hand, LDA could be understood as leveraged dimension reduction method similar to PCA. In that case, LDA restrictions should not play so important role.

LDA, as name suggests, is a linear method but there are non-linear extensions like QDA (Quadratic Discriminant Analysis), see the `MASS::qda()` function.

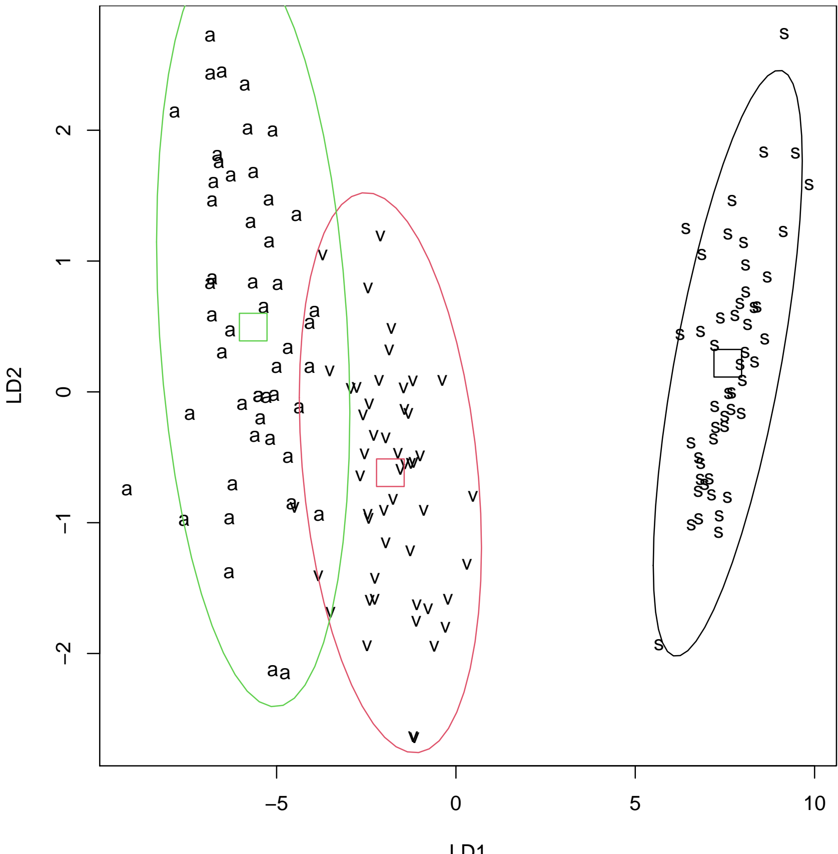


Figure 9.2: Graphical representation of the linear discriminant analysis results. 95% confidence ellipses and their centers added with `Ellipses()` function.

* * *

With LDA, it is easy to illustrate one more important concepts of machine learning, the *quality of training*. Consider the following example:

```
> set.seed(16881)
> iris.sample2 <- sample(nrow(iris), 30)
> set.seed(NULL) # set random number generator default
> iris.train2 <- iris[iris.sample2, ]
> iris.unknown2 <- iris[-iris.sample2, ]
> iris.lda2 <- lda(Species ~ . , data=iris.train2)
> iris.predicted2 <- predict(iris.lda2, iris.unknown2[, 1:4])
```

```
> Misclass(iris.predicted2$class, iris.unknown2[, 5]) # shipunov
Classification table:
            obs
pred         setosa versicolor virginica
  setosa         44          0         0
  versicolor      0         32        12
  virginica       0          0        32
Misclassification errors (%):
    setosa versicolor  virginica
       0.0        0.0       27.3
Mean misclassification error: 9.1%
```

Misclassification error here is much bigger than usual! Why?

```
> table(iris.train2$Species)
    setosa versicolor  virginica
         6         18          6
```

Well, using `sample()` (and particular `set.seed()` value) resulted in biased training sample, this is why our second model was trained so poorly. Our first way to sample (every 5th iris) was better, and if there is a need to use `sample()`, consider to sample each species *separately*. This is called *stratified sampling*.

Stratified sampling is easy to perform with `Class.sample()` function:

```
> sam <- Class.sample(iris[, 5], nsam=10) # shipunov
> nrow(iris[sam, ])
 [1] 30
```

(Here we used 10 samples per class. Please note that `Class.sample()` outputs logical vector so to make "unknown", testing object, one must use `iris[!sam, ]`.)

* * *

As you see from the above, misclassification table (confusion matrix) allows to assess the predictive power of model in a simple way. More advanced technique of same sort is called *cross-validation*.

As an example, user might split data into 10 equal parts (e.g., with `cut()`) and then in turn, make each part an "unknown" whereas the rest will become training subset. This is called *k-fold cross-validation*.

Another cross-validation method is the *repeated random sub-sampling*. It is better to perform using the function:

```
> RSS <- function() {
+  sam <- Class.sample(iris[, 5], 5) # 5 samples per class
+  iris.lda <- lda(Species ~ ., data=iris[sam, ])
+  iris.ldp <- predict(iris.lda, iris[!sam, -5])
+  iris.ldm <- Misclass(iris.ldp$class, iris[!sam, 5], quiet=TRUE)
+  mean(((CS <- colSums(iris.ldm)) - diag(iris.ldm))/CS) * 100
+ }
> res <- replicate(100, RSS())
> quantile(res, c(0.025, 0.5, 0.975))
     2.5%       50%     97.5%
 1.481481  4.444444 12.592593
> hist(res)
```

(Please **review** the histogram yourself. What do you think, why is the right tail longer?)

9.1.2 Recursive partitioning

To develop further the linear discriminant analysis, multiple methods with similar background ideas were invented. Recursive partitioning, or *decision trees* (regression trees, classification trees), allow, among other, to make and visualize the sort of discrimination key where every step results in splitting objects in two groups (Fig. 9.3):

```
> library(tree)
> iris.tree <- tree(Species ~ ., data=iris)
> plot(iris.tree)
> text(iris.tree)
```

We loaded first the `tree` package containing `tree()` function. Then we used the whole dataset as training data. The plot shows that all plants with petal length less than 2.45 cm belong to *Iris setosa*, and from the rest those plants which have petal width less than 1.75 cm and petal length more than 4.95 cm, are *I. versicolor*; all other irises belong to *I. virginica*.

Now, we predict *Iris* species using training data set:

```
> iris.tree2 <- tree(Species ~ ., data=iris.train)
> iris.tp2 <- predict(iris.tree2, iris.unknown[, -5], type="class")
> Misclass(iris.tp2, iris.unknown[, 5]) # shipunov
Classification table:
            obs
pred          setosa versicolor virginica
  setosa          40          0         0
  versicolor       0         38         6
```

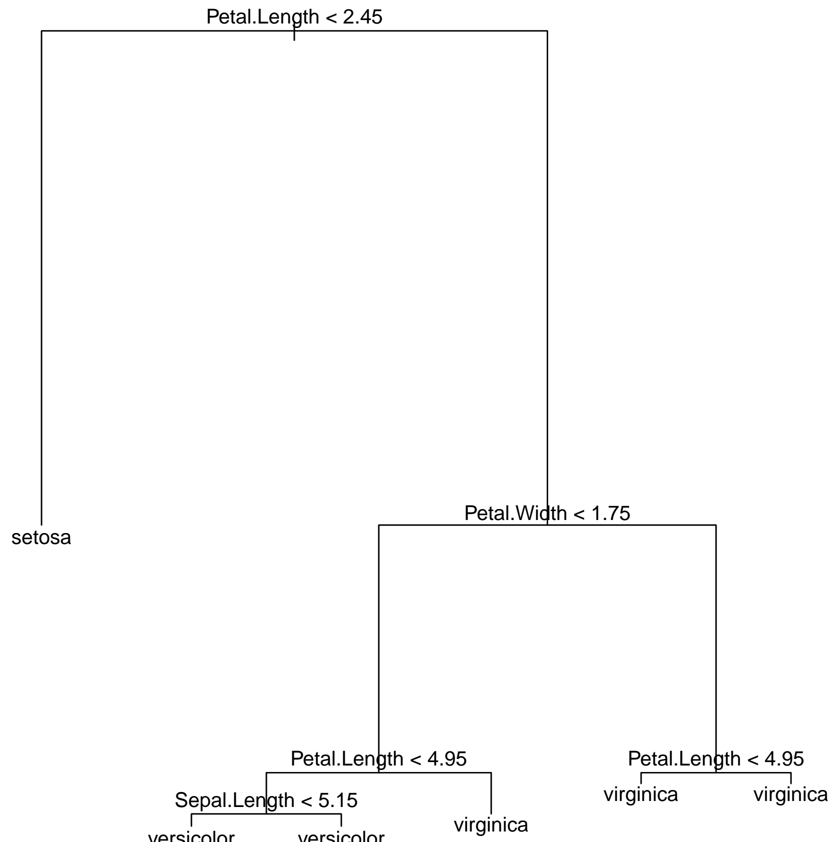


Figure 9.3: Classification tree for the `iris` data from `tree` package.

```
  virginica       0           2        34
Misclassification errors (%):
    setosa versicolor  virginica
         0          5         15
Mean misclassification error: 6.7%
```

> Try to find out which characters distinguish two species of horsetails described in `eq` dataset from `shipunov` package. Run `?eq` to see descriptions of characters.

Package `rpart` has very similar interface but in general more advanced. There the main command to produce recursive partitioning is `rpart()`.

* * *

Package `party` offers sophisticated recursive partitioning methods together with advanced tree plots (Fig. 9.4):

```
> library(party)
> SA <- abbreviate(iris$Species, 1, method="both.sides")
> iris.ct <- ctree(factor(SA) ~ ., data=iris[, 1:4])
> plot(iris.ct)
> Misclass(predict(iris.ct), SA) # shipunov
Classification table:
    obs
pred  a  s  v
   a 49  0  5
   s  0 50  0
   v  1  0 45
Misclassification errors (%):
 a  s  v
 2  0 10
Mean misclassification error: 4%
```

(For species names, we used one-letter abbreviations.)

* * *

Increasing the number of feature variables should definitely improve the quality of recursive partition, but here is another danger: with more variables, the model becomes more and more data-specific and therefore less general; it will not handle well the new data.

Neural networks and recursive partitioning are especially sensitive to *overfitting*. The recipe is to stop early, prune large models, and remove the less significant data-specific features. Cross-validation, even its simplest variants (like in `Misclass()` examples), help to recognize overfitting.

Another approach to avoid overfitting is to use One Rule algorithms. They are basically multiple 1-level recursive partitioning:

```
> library(OneR)
> iris.one <- OneR(Species ~ ., data=iris.train)
...
> summary(iris.one)
...
```

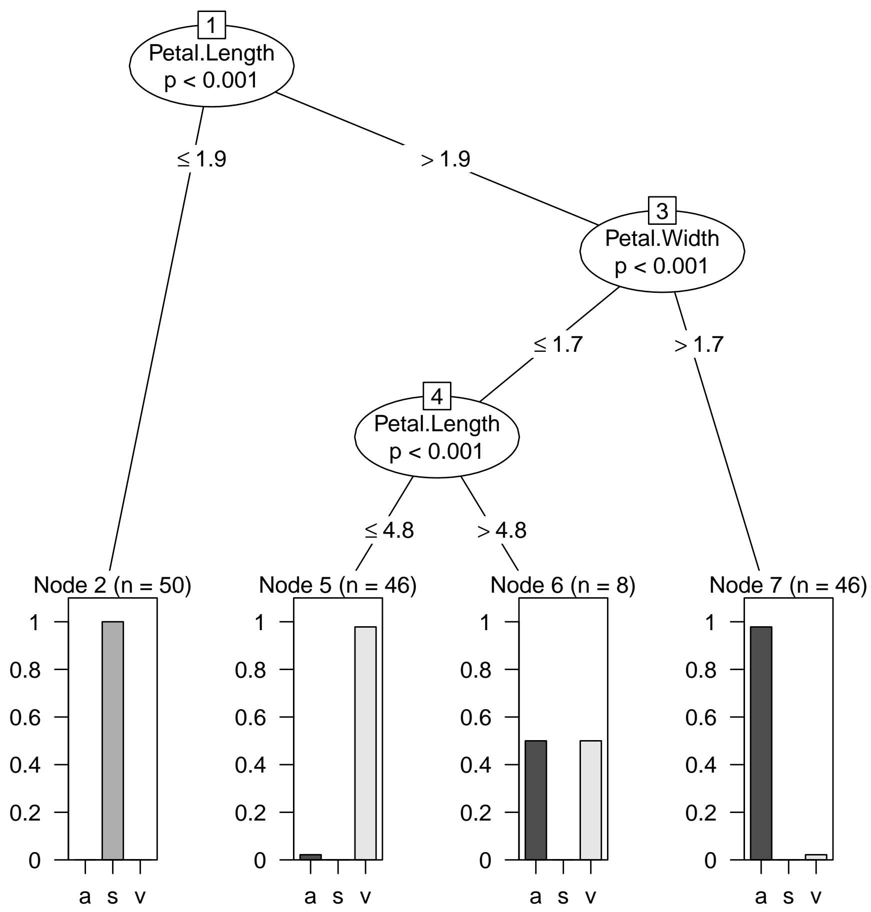


Figure 9.4: Classification tree for the `iris` data from `party` package.

```
Rules:
If Petal.Width = (0.198,0.66] then Species = setosa
If Petal.Width = (0.66,1.12]  then Species = versicolor
If Petal.Width = (1.12,1.58]  then Species = versicolor
If Petal.Width = (1.58,2.04]  then Species = virginica
If Petal.Width = (2.04,2.5]   then Species = virginica
...
Accuracy:
28 of 30 instances classified correctly (93.33%)
...
Pearson's Chi-squared test:
X-squared = 52.8, df = 8, p-value = 1.179e-08
```

```
> iris.pre <- predict(iris.one, iris.unknown[, -5])
> Misclass(as.character(iris.pre), iris.unknown$Species,
+ ignore="UNSEEN", useNA="ifany", best=TRUE) # shipunov
Best classification table:
            obs
pred         setosa versicolor virginica
  setosa         35          0         0
  versicolor      0         37         3
  virginica       0          3        37
  <NA>            5          0         0
Misclassification errors (%):
    setosa versicolor  virginica
      12.5        7.5        7.5
Mean misclassification error: 9.2%
Note: data contains NAs
```

The idea is, as you see, to find just one well-working rule which is able to predict most (in that case, 93%) of the training data.

9.2 Ensemble learnig

These methods use collective learning: employ multiple instances of some learning algorithm and then make the joint model.

9.2.1 Random Forest

Random Forest simultaneously uses numerous decision trees and builds the complex classification model. It belongs to *bagging ensemble methods* and uses bootstrap (see in Appendix) to multiply the number of trees in the model (hence "forest"). Below is an example of Random Forest classifier made from the `iris` data:

```
> library(randomForest)
> set.seed(17) # Random Forest is stochastic method
> iris.rf <- randomForest(Species ~ ., data=iris.train)
> iris.rfp <- predict(iris.rf, iris.unknown[, -5])
> Misclass(iris.rfp, iris.unknown[, 5]) # shipunov
Classification table:
            obs
pred         setosa versicolor virginica
  setosa         40          0         0
  versicolor      0         39         7
  virginica       0          1        33
```

```
Misclassification errors (%):
    setosa versicolor  virginica
       0.0        2.5       17.5
Mean misclassification error: 6.7%
```

Random Forest can also clarify the importance of each character (with function `importance()`), and reveal proximities between all objects of training subset. These proximities are frequences with which objects appear together in the terminal nodes of underlying trees.

Function `randomForest()` can make one more step forward and run in the unsupervised mode. In that case, artificial data generated, and model trained to distinguish between real and artificial data. Resulted proximities can serve as distances for any distance-based discovery method (Fig. 9.5):

```
> set.seed(17)
> iris.urf <- randomForest(iris[, -5]) # no response variable
> iris.rfc <- cmdscale(1 - iris.urf$proximity)
> plot(iris.rfc, xlab="", ylab="",
+ pch=abbreviate(iris[, 5], 1, method="both.sides"))
> Hulls(iris.rfc, as.numeric(iris[, 5]), outliers=FALSE,
+ coef=0.8, usecolor=rep(1, 3), lty=2) # shipunov
> Biarrows(iris.rfc, iris[, -5], shift=c(-0.1, 0)) # shipunov
```

(Please also note how to show convex hulls without outliers.)

Package `ranger` implements even faster variant of Random Forest algorithm, it also can employ parallel calculations.

9.2.2 Gradient boosting

There are many weak classification methods which typically make high misclassification errors. However, as many of them are also ultra-fast, it is possible to apply many weak learners sequentially to make the strong one. This approach is called boosting. Gradient boosting employs multi-step optimization and is now among most frequently using learning techniques. In R, there are several gradient boosting packages, for example, `xgboost` and `gbm`:

```
> library(gbm)
> set.seed(4)
> iris.gbm <- gbm(Species ~ .,
+ data=rbind(iris.train, iris.train)) # to make training bigger
... Distribution not specified, assuming multinomial ...
> iris.gbm.p1 <- predict(iris.gbm, iris.unknown[, -5],
```

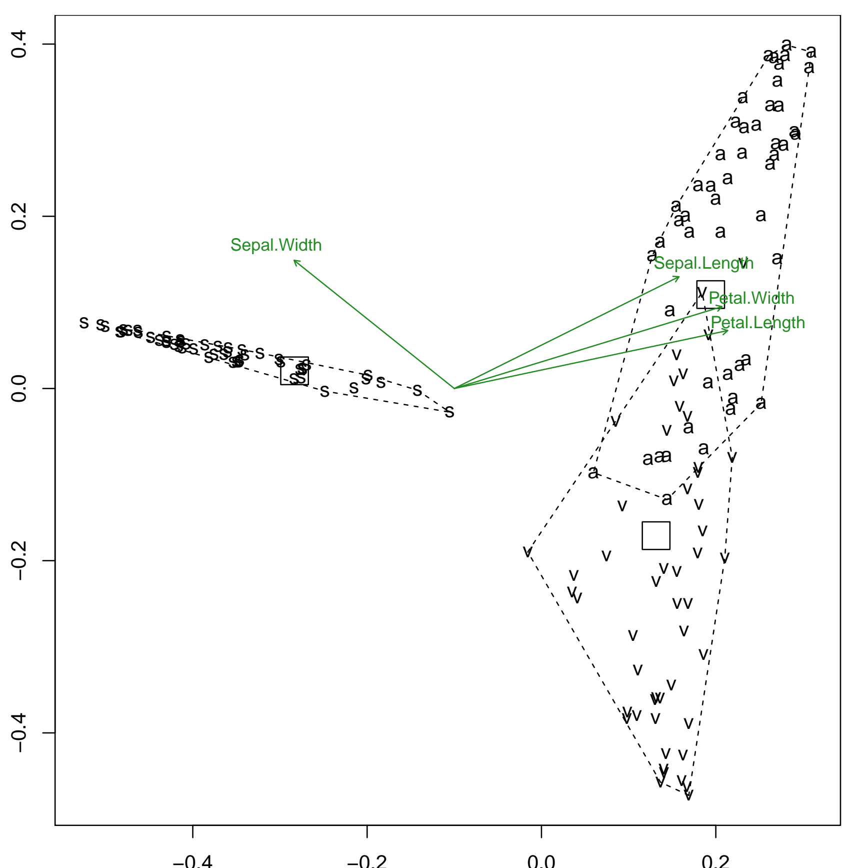


Figure 9.5: Visualization of iris data with the help of “Random Forest”. Hulls and their centroids added with Hulls() function.

```
+ n.trees=iris.gbm$n.trees)
> iris.gbm.p2 <- apply(iris.gbm.p1, 1, # membership trick
+ function(.x) colnames(iris.gbm.p1)[which.max(.x)])
> Misclass(iris.gbm.p2, iris.unknown[, 5]) # shipunov
Classification table:
            obs
pred          setosa versicolor virginica
  setosa          40          0         0
  versicolor       0         38         5
  virginica        0          2        35
Misclassification errors (%):
    setosa versicolor  virginica
```

```
       0.0        5.0       12.5
Mean misclassification error: 5.8%
```

(Function `gbm()` uses regressions as weak learners and works better if data distribution is known. "Membership trick" selects the "best species" from three alternatives as `gbm()` reports classification result in fuzzy form.)

* * *

In addition to boosting and bagging methods, there are also *stacking* ensemble learning. In that case, several different classifiers used to output their predictions. On the second step, new classifier uses these predictions as variables to output the final result. To use stacking ensemble learning, check out the `SuperLearner` package.

9.3 Learning with proximity

k-Nearest Neighbors algorithm (or *k*-NN) is the "lazy classifier" because it does not work until unknown data is supplied:

```
> library(class)
> iris.knn.pred <- knn(train=iris.train[, -5],
+ test=iris.unknown[, -5], cl=iris.train[, 5], k=5)
> Misclass(iris.knn.pred, iris.unknown[, 5]) # shipunov
Classification table:
            obs
pred         setosa versicolor virginica
  versicolor      0         40        11
  virginica       0          0        29
Misclassification errors (%):
    setosa versicolor  virginica
       0.0        0.0       27.5
Mean misclassification error: 9.2%
```

k-NN is based on *distance calculation* and "voting". It calculates distances from every unknown object to the every object of the training set. Next, it considers several (5 in the case above) nearest neighbors with known identity and finds which is prevalent. This prevalent identity then assigned to the unknown member.

To illustrate idea of nearest neighbors, we use *Voronoi decomposition*, the technique which is close to both *k*-NN and distance calculation:

```
> iris.p <- prcomp(iris[, -5])$x[, 1:2]
> iris.p1 <- iris.p[seq(1, nrow(iris.p), 5), ]
```

```
> iris.p2 <- iris.p[-seq(1, nrow(iris.p), 5), ]
> library(tripack)
> iris.v <- voronoi.mosaic(iris.p1[, 1], iris.p1[, 2],
+ duplicate="remove")
> plot(iris.v, do.points=FALSE, main="", sub="")
> points(iris.p1[, 1:2], col=iris.train$Species, pch=16, cex=2)
> points(iris.p2[, 1:2], col=iris.unknown$Species)
```

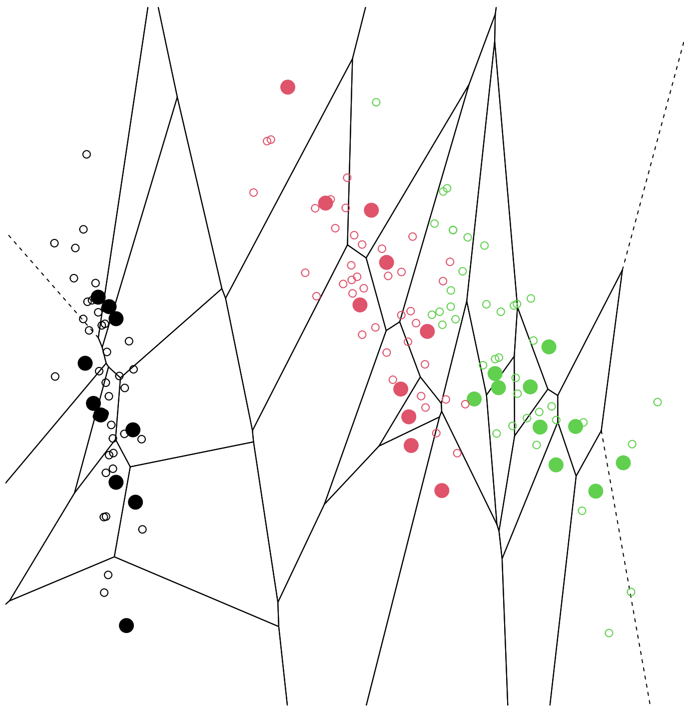


Figure 9.6: Visualization of training data points neighborhoods with Voronoi decomposition.

The plot (Fig. 9.6) contains multiple cells which represent neighborhoods of training sample (big dots). This is not exactly what *k*-NN does, but idea is similar. In fact, Voronoi plot is a good tool to visualize any distance-based approach.

* * *

Distance-based k-NN classification (DNN) is the super-lazy classifier because it even does not calculate the distance itself. The user should supply it with some pre-

calculated distance matrix, and DNN does the rest of the job: assigns testing points to classes on the base of distances between testing and training points.

This feature allows to use the whole variety of distance metrics, for example, the universal Gower distance (e.g., `Gower.dist()` from package `shipunov`) and analogous universal Wishart and Podani distances (package `kmed`).

Here we give just one but impressive example. Below, we convert `iris` data into the English text! On the next step, we use the *string distance metric* which is developed for text data analysis (the chapter of data analysis which we almost entirely missed):

```
> library(stringdist)
> library(english)
> iris.en <- apply(round(iris[, -5]), 1,
+ function(.x) paste(as.character(as.english(.x)), collapse=" "))
> head(iris.en)
[1] "five four one zero"  "five three one zero" "five three one zero"
[4] "five three two zero" "five four one zero"  "five four two zero"
> iris.end <- stringdistmatrix(iris.en)
> cl1 <- iris$Species
> sam <- Class.sample(iris[, 5], nsam=10, uniform=TRUE)
> cl1[!sam] <- NA # DNN wants unknown classes as NAs
> iris.pred <- DNN(dst=iris.end, cl=cl1, k=5)
> Misclass(iris.pred, iris$Species[is.na(cl1)]) # shipunov
Classification table:
            obs
pred         setosa versicolor virginica
  setosa         40          3         1
  versicolor      0         34         5
  virginica       0          3        34
Misclassification errors (%):
    setosa versicolor  virginica
         0         15         15
Mean misclassification error: 10%
```

(So even after rounding and conversion into the spelled text, there is 90% of accuracy!)

Again, the biggest advantage of DNN is to use distance method of your choice. However, whereas `DNN()` function is very fast, distance calculations might be significantly slower than methods internal to more common k-NN-based classification algorithms.

* * *

Depth classification based on how close an arbitrary point of the space is located to an implicitly defined center of a multidimensional data cloud:

```
> library(ddalpha)
> iris.dd <- ddalpha.train(Species ~ ., data=iris.train)
Selected columns:  Species,   Sepal.Length,   Sepal.Width,
Petal.Length,   Petal.Width
> iris.p <- predict(iris.dd, iris.unknown[, -5]) # default method
> Misclass(unlist(iris.p), iris.unknown[, 5]) # shipunov
Classification table:
            obs
pred         setosa versicolor virginica
  setosa         40          0         0
  versicolor      0         40         7
  virginica       0          0        33
Misclassification errors (%):
    setosa versicolor  virginica
       0.0        0.0       17.5
Mean misclassification error: 5.8%
> iris.pp <- predict(iris.dd, iris.unknown[, -5],
+ outsider.method="Ignore")
> sapply(iris.pp, as.character) # shows points outside train clouds
  [1] "Ignored"    "Ignored" ...
```

9.4 Learning with rules

Naïve Bayes classifier tries to classify objects based on the probabilities of previously seen attributes. It is typically a good classifier:

```
> library(e1071)
> iris.nb <- naiveBayes(Species ~ ., data=iris.train)
> iris.nb
Naive Bayes Classifier for Discrete Predictors
...
A-priori probabilities:
Y
    setosa versicolor  virginica
 0.3333333  0.3333333  0.3333333
Conditional probabilities:
            Sepal.Length
```

```
Y             [,1]      [,2]
  setosa     5.16 0.2988868
...
> iris.nbp <- predict(iris.nb, iris.unknown[, -5])
> Misclass(iris.nbp, iris.unknown[, 5]) # shipunov
Classification table:
            obs
pred         setosa versicolor virginica
  setosa         40          0         0
  versicolor      0         39        13
  virginica       0          1        27
Misclassification errors (%):
    setosa versicolor  virginica
       0.0        2.5       32.5
Mean misclassification error: 11.7%
```

Note that Naïve Bayes classifier could use both measurement and categorical variables.

* * *

Apriori method is similar to regression trees but instead of classifying objects, it researches *association rules* between classes of objects. This method could be used not only to find these rules but also to make classification. Note that measurement `iris` data is less suitable for association rules then nominal data, and it needs *discretization*:

```
> library(arulesCBA)
> irisd <- as.data.frame(lapply(iris[1:4], discretize, breaks=9))
> irisd$Species <- iris$Species
> irisd.train <- irisd[seq(1, nrow(irisd), 5), ]
> irisd.unknown <- irisd[-seq(1, nrow(irisd), 5), ]
> irisd.cba <- CBA(Species ~ ., irisd.train, supp=0.05, conf=0.8)
> inspect(irisd.cba$rules)
    lhs                             rhs                    support
[1] {Petal.Length=[1.00,1.66)}  => {Species=setosa}     0.26666667
[2] {Petal.Width=[0.100,0.367)} => {Species=setosa}     0.26666667
[3] {Petal.Length=[4.28,4.93)}  => {Species=versicolor} 0.23333333
[4] {Petal.Length=[5.59,6.24)}  => {Species=virginica}  0.20000000
[5] {Petal.Width=[2.233,2.500]} => {Species=virginica}  0.20000000
[6] {Petal.Width=[1.167,1.433)} => {Species=versicolor} 0.20000000
[7] {Petal.Length=[4.93,5.59)}  => {Species=virginica}  0.10000000
```

```
[8] {Petal.Width=[0.900,1.167)} => {Species=versicolor} 0.06666667
...
> irisd.cbap <- predict(irisd.cba, irisd.unknown)
> Misclass(irisd.cbap, irisd.unknown$Species) # shipunov
Classification table:
            obs
pred         setosa versicolor virginica
  setosa         40          1         4
  versicolor      0         37         6
  virginica       0          2        30
Misclassification errors (%):
    setosa versicolor  virginica
       0.0        7.5       25.0
Mean misclassification error: 10.8%
```

9.5 Learning from the black boxes

"Black box" methods try to find the best classification model generally without regard to the natural laws, and mostly by some abstract, try-and-error or similar algorithmic procedures. As a result, they frequently predict very well but their results are not easy to explain.

9.5.1 Support Vector Machines

Famous SVM, *Support Vector Machines* is a *kernel* technique (i.e., it modifies the original data). It searches for the best parameters of the hyper-planes which divide groups in the multidimensional space of characters:

```
> library(e1071)
> iris.svm <- svm(Species ~ ., data=iris.train)
> iris.svmp <- predict(iris.svm, iris.unknown[, -5])
> Misclass(iris.svmp, iris.unknown[, 5]) # shipunov
Classification table:
            obs
pred         setosa versicolor virginica
  setosa         40          0         0
  versicolor      0         40        14
  virginica       0          0        26
Misclassification errors (%):
    setosa versicolor  virginica
         0          0         35
```

```
Mean misclassification error: 11.7%
```

Classification, or prediction *grid* often helps to illustrate the SVM method. Data points are arranged with PCA to reduce dimensionality, and then classifier predicts the identity for the every point in the artificially made grid (Fig. 9.7). This is possible to perform manually but `Gradd()` function simplifies a plotting of classification grid:

```
> iris.p <- prcomp(iris[, -5])$x[, 1:2]
> iris.svm.pca <- svm(Species ~ ., data=cbind(iris[5], iris.p))
> plot(iris.p, type="n", xlab="", ylab="")
> Gradd(iris.svm.pca, iris.p, cex=0.4) # shipunov
> text(iris.p, col=as.numeric(iris[, 5]),
+ labels=abbreviate(iris[, 5], 1, method="both.sides"))
```

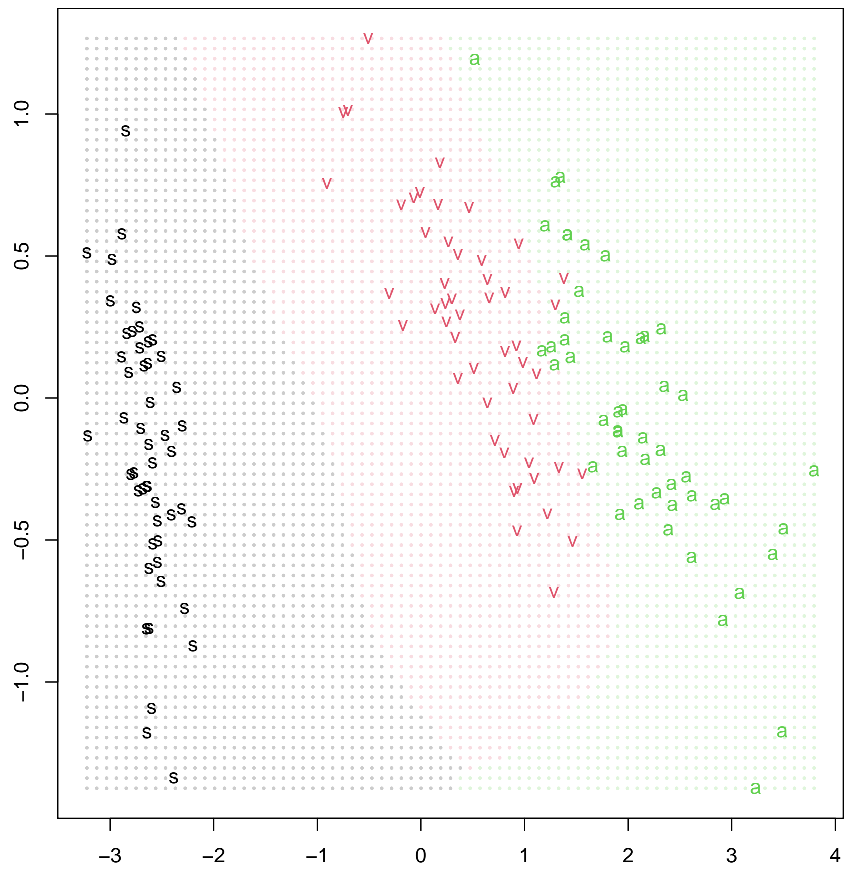


Figure 9.7: Classification grid which illustrates the SVM algorithm. Data points are arranged with PCA.

9.5.2 Neural Networks

And finally, *neural networks*! This name is used for the statistical technique based on some features of living neural cells, *neurons*. Nowadays, "neural networks" frequently means something really complicated and powerful, but practically, this technique is not far away from linear regression and LDA.

In fact, neural networks consist of two things: the system of equations and the algorithm which updates their coefficients. This algorithm works cyclically, and in the end, updated coefficients should allow for class prediction.

First, we will make the neural network of multiple neurons, not with just input and output but also with "in between" neurons (*hidden layer*). There are many R packages which have these networks implemented, for example, the package nnet:

```
> library(nnet)
> iris.nn <- nnet(Species ~ . , data=iris.train, size=4)
# weights:  35
initial  value 37.203588
iter  10 value 9.507574
iter  20 value 0.009795
final  value 0.000077
converged
> iris.nnp <- predict(iris.nn, iris.unknown[, -5], type="class")
> Misclass(iris.nnp, iris.unknown$Species) # shipunov
Classification table:
            obs
pred          setosa versicolor virginica
  setosa          40          0         0
  versicolor       0         37         0
  virginica        0          3        40
Misclassification errors (%):
    setosa versicolor  virginica
       0.0        7.5        0.0
Mean misclassification error: 2.5%
```

Neural nets could be plotted with command plotn() from the NeuralNetTools package:

```
> library(NeuralNetTools)
> oldpar <- par(mar=c(0, 2, 0, 1), xpd=TRUE)
> plotnet(iris.nn)
> par(oldpar)
```

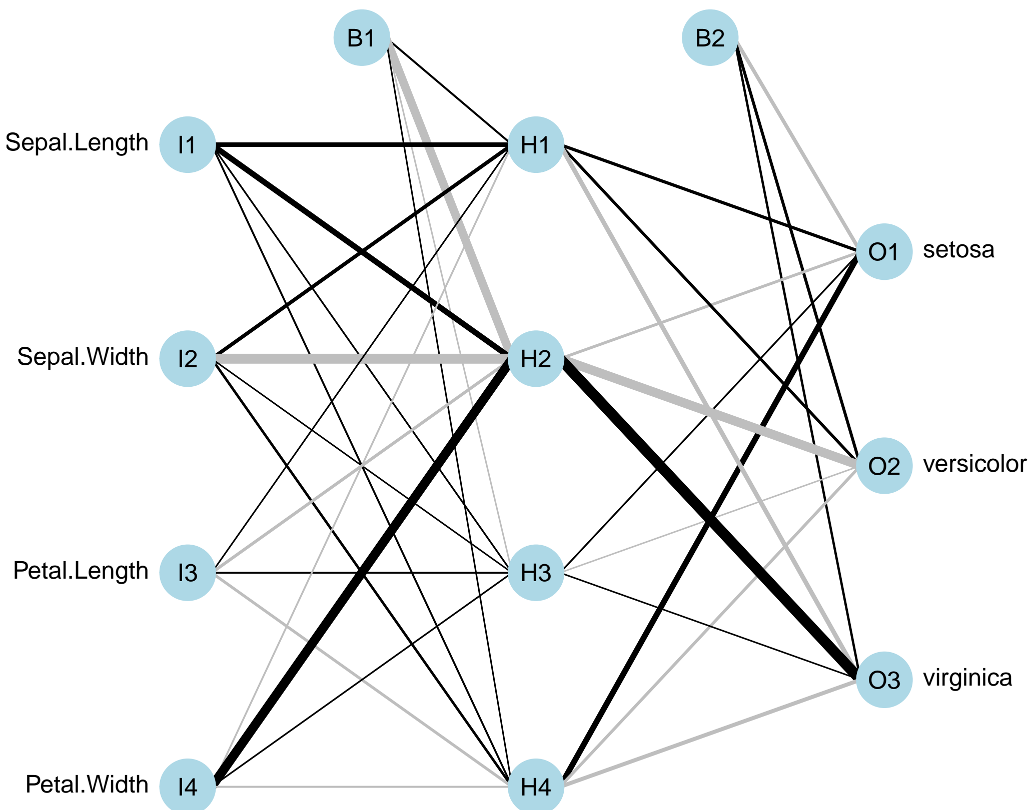


Figure 9.8: Neural network with four-neuron hidden layer.

The resulted network (Fig. 9.8) has four inputs (one per column), three outputs (one per species) and multiple connecting coefficients-weights (including bias-related, labeled with "B".)

* * *

To understand neural networks better, we will employ the very simple example or *perceptron*.

Perceptron is just one "neuron" which is however can tell apart two groups within data (but only if these two groups are linearly separable). Perceptron is based on one equation, and if the data has two characters (two columns), then perceptron should have three coefficients: weight of the first character, weight of the second character and bias.

The perceptron algorithm should work cyclically (we call it "learning") so it is better to make perceptron a function and then run it many times:

```
> Perceptron <- function(x, y, eta, n) {
+  ## x dataset, y classes, eta learning speed, n cycles
+  ## make initial weights randomly
+  weight <- runif(ncol(x) + 1)
+  ## make initial errors vector: we need it for visualization
+  errors <- rep(0, n)
+  ## loop over number of epochs (learning cycles) n
+  for (jj in 1:n) {
+   ## loop through training data set
+   for (ii in 1:length(y)) {
+    ## main equation: "w1*x + w2*y + ... + b" (_w_eights and _b_ias)
+    z <- sum(as.numeric(x[ii, ]) * weight[-1]) + weight[1]
+    ## decision step: Heaviside activation
+    ypred <- as.numeric(z > 0) # all > 0 become 1
+    ## update weight, it changes if the predicted value is incorrect
+    weightdiff <- eta * (y[ii] - ypred) *  c(1, as.numeric(x[ii, ]))
+    weight <- weight + weightdiff
+    ## update error vector
+    if ((y[ii] - ypred) != 0) errors[jj] <- errors[jj] + 1
+    }
+   }
+  ## name coefficients
+  names(weight) <- c("bias", names(x))
+  ## output will be a list
+  list(weight=weight, errors=errors)
+ }
```

(Please read all comments carefully. Function includes two cycles but is not extremely complicated; if you understand the connection between main equation and how weights are updated, you got it.)

This new `Perceptron()` function is good to learn from *any* linearly separable data with two subgroups. We will use it for our `iris` data, but at first will simplify it:

```
> ## new "species": "setosa" and "non-setosa" (1 or 0)
> y <- as.numeric(iris[, 5] == "setosa")
> ## make sample (10% of each class)
> sam <- Class.sample(y, prop=0.1, uniform=TRUE) # shipunov
> ## training classes
> y.trn <- y[sam]
```

```
> ## select two characters (but more characters is usually better)
> x <- iris[, c(1, 3)]
> ## check if everything is separable
> plot(x, col=y+1)
> ## training data
> x.trn <- x[sam, ]
```

(Please **check** the plot yourself. Does it show that data is linearly separable?)

Now, we run the `Perceptron()` with `x.trn` and `y.trn` data ten times:

```
> (P <- Perceptron(x=x.trn, y=y.trn, eta=0.5, n=10))
$weight
        bias Sepal.Length Petal.Length
    1.025410     1.776865    -4.441380
$errors
 [1] 1 3 2 3 1 0 0 0 0 0
```

As you might see, number of errors grows at first but then goes to zero. This is a good sign.

Weights are the most important part of the result. Perceptron is a typical black box: we do not know what these weights mean but they will hopefully aid in species identification. Let us check it with "new", testing data:

```
> ## make testing sample
> x.tst <- x[!sam, ]
> ## select weights from perceptron object
> W <- P$weight
> ## predict each row of x.tst with the main Perceptron() equation
> z.tst <- apply(x.tst, 1, function(.x) sum(.x * W[-1] + W[1]))
> ## decision
> z.tst <- as.numeric(z.tst > 0)
> ## testing classes to compare with prediction
> y.tst <- y[!sam]
> Misclass(z.tst, y.tst) # shipunov
Classification table:
    obs
pred  0  1
   0 90  0
   1  0 45
Misclassification errors (%):
0 1
0 0
```

```
Mean misclassification error: 0%
```

Wonderful results! Our perceptron, just on a base of three numbers was able to tell apart *setosa* and non-*setosa* irises using only two characters, `Sepal.Length` and `Petal.Length`.

```
> library(NeuralNetTools)
> oldpar <- par(mar=c(0, 2, 0, 1), xpd=TRUE)
> plotnet(W, struct=c(2, 1), x_names=names(x), y_names="setosa")
> par(oldpar)
```

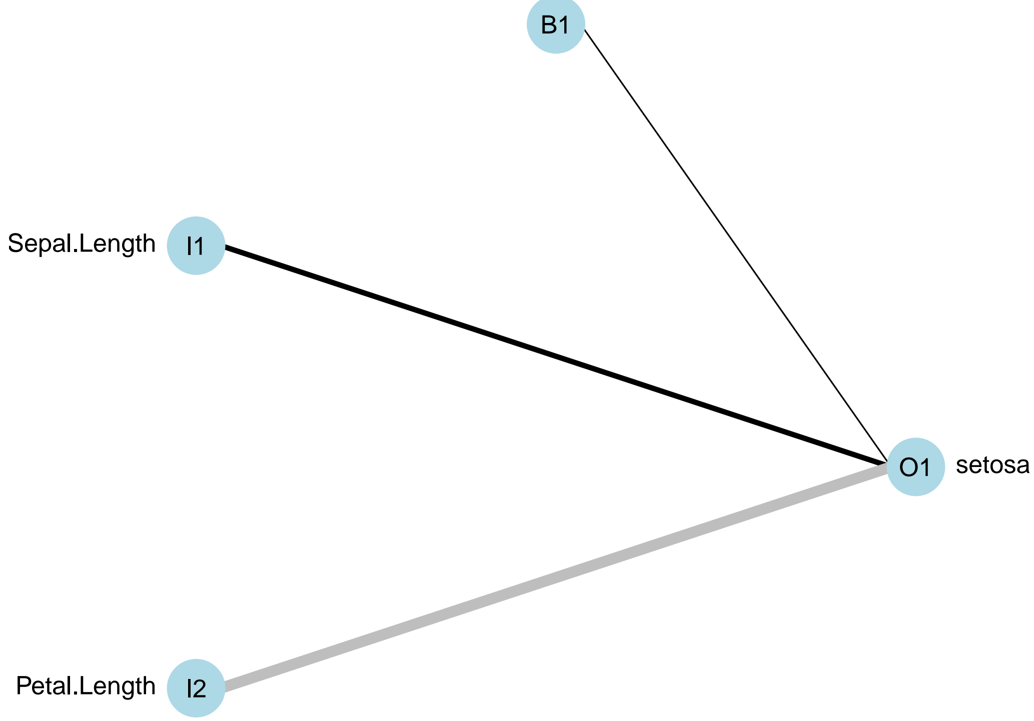


Figure 9.9: The perceptron.

Nothing fancy but the main features of our most simple neural network are clearly visible (Fig. 9.9): two input characters, one output node (*setosa* or not *setosa*) and bias node.

9.6 Semi-supervised learning

Completely new multivariate methods are rare, and many methods are either (1) extensions of some older techniques ("B is like A but for nominal data"); (2) "level-up" methods ("B analyzes data outputted by A") or (3) "hybrid", mixed methods ("B uses A on first step and C on second step").

The last group is especially easy to create. There is no deep distinction between supervised and non-supervised methods, some of non-supervised (like SOM or PCA) could use training whereas some supervised (LDA, Random Forest, recursive partitioning) are useful directly as visualizations. Therefore, it is easy to mix supervised and non-supervised methods and invent *semi-supervised* learning. It takes into account both data features and data labeling (Fig. 9.10).

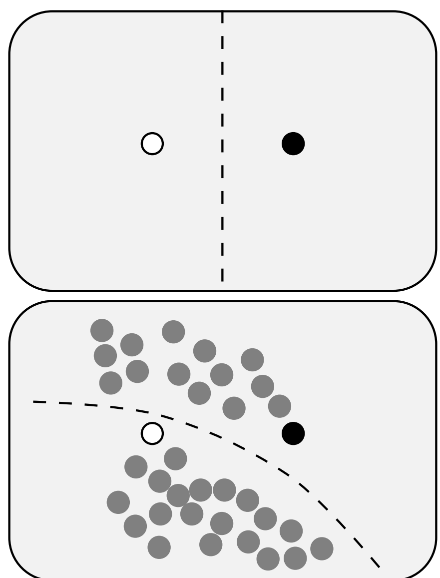


Figure 9.10: How semi-supervised learning can improve learning results. If only labeled data used, then the most logical split is vertical. However, if we look on the testing set, it become apparent that training points are parts of more complicated structures, and the actual split goes in the other direction.

One of the most important features of SSL is an ability to work with the very small training sample. Many really bright ideas are embedded in SSL, here we just one of them on the example of `sslSelfTrain()` function from the `SSL` package. To install SSL package, you need to install `devtools` package first, and then run installation from Github:

```
> library(devtools)
> install_github("cran/SSL")
```

Function `sslSelfTrain()` uses *self-learning* which means that classification is developed in multiple cycles. On each cycle, testing points which are most confident, are labeled and added to the training set:

```
> library(SSL)
> iris.10 <- seq(1, nrow(iris), 10) # only 10 labeled points!
> iris.sslt1 <- sslSelfTrain(iris[iris.10, -5], iris[iris.10, 5],
+ iris[-iris.10, -5], nrounds=20, n=5)
> iris.sslt2 <- levels(iris$Species)[iris.sslt1]
> Misclass(iris.sslt2, iris[-iris.10, 5]) # shipunov
Classification table:
            obs
pred         setosa versicolor virginica
  setosa         45          0         0
  versicolor      0         43         6
  virginica       0          2        39
Misclassification errors (%):
    setosa versicolor  virginica
       0.0        4.4       13.3
Mean misclassification error: 5.9%
```

As you see, with only 10 training points (approximately 7% of data vs. 33% of data in `iris.train`), semi-supervised self-leaning (in this case, based on gradient boosting) reached 94% of accuracy!

* * *

Semi-supervised learning includes most unusual but well working techniques. For example, learning methods require class labels. What if we *calculate* class labels using the discovery approach?

```
> library(kpeaks)
> nclusters <- max(findk(iris[, -5])$pcounts) # step 1
> iris.cl <- kmeans(iris[, -5], centers=nclusters)$cluster # step 2
> sel <- Class.sample(iris.cl, 10) # shipunov
> iris.svm <- svm(as.factor(iris.cl[sel]) ~ ., data=iris[sel, -5])
> iris.svmp <- predict(iris.svm, iris[!sel, -5]) # step 3
> Misclass(iris.svmp, iris[!sel, 5], best=TRUE) # shipunov
Best classification table:
    obs
```

```
pred virginica setosa versicolor
   1        28      0          2
   2         0     40          0
   3        10      0         40
Misclassification errors (%):
 virginica     setosa versicolor
      26.3        0.0        4.8
Mean misclassification error: 10.4%
```

There are three steps. First, with the help of `kpeaks::findk()` function, we determined the number of clusters. Second, we performed *k*-means clustering to find cluster labels (note that `kmeans()` function *does not know* species names).

Third, we selected 10 class labels per class, trained SVM model and predicted the rest of class labels. It is simply amazing that this approach reaches 89% of accuracy against the *real* species names!

This process of learning without pre-defined class labels we can call *automated learning*.

* * *

Another hybrid method might employ learning for dimension reduction. If there are class labels, then projection pursuit might use it to find the most separated variant:

```
> iris.dms <- Classproj(iris[, -5], iris$Species) # shipunov
> plot(iris.dms$proj, col=iris$Species)
> text(iris.dms$centers, levels(iris$Species), col=1:3)
```

(Please **review** the plot yourself.)

We call this approach *leveraged* (sometimes, they call it also supervised dimension reduction) because the dimension reduction tool "knows" all data labels beforehand.

What if we know only some class labels? This we call *educated* (semi-supervised dimension reduction) approach:

```
> sam <- Class.sample(iris$Species, 10) # shipunov
> newclasses <- iris$Species
> newclasses[!sam] <- NA
> iris.dms <- Classproj(iris[, -5], newclasses) # shipunov
> plot(iris.dms$proj, col=iris$Species, pch=ifelse(sam, 19, 1))
> text(iris.dms$centers, levels(iris$Species), col=1:3)
```

(Please **review** the plot yourself.)

Both leveraged and educated approaches for dimension reduction are implemented also in `umap()` function from `uwot` package. It is possible also to imagine leveraged, educated and even automated LDA (if it is used as dimension reduction tool.)

By the way, how to implement an *automated* `Classproj()`? Please **think**.

* * *

Yet another approach is to use data features (for example, distances between points) to improve layout (to "sharpen clusters"). The package `ksharp` allows to improve cluster separation, using the concept of noise points (similarly to DBSCAN):

```
> library(ksharp)
> iris.km <- kmeans(iris[, -5], centers=3)
> Misclass(iris.km$cluster, iris$Species, best=TRUE) # shipunov
Best classification table:
    obs
pred setosa versicolor virginica
   1     50          0         0
   2      0         48        14
   3      0          2        36
Misclassification errors (%):
    setosa versicolor  virginica
         0          4         28
Mean misclassification error: 10.7%
> names(iris.km$cluster) <- row.names(iris) # ksharp requirement
> iris.ksharp <- ksharp(iris.km, data=iris[, -5])
> plot(prcomp(iris[, -5])$x, col=iris$Species,
+ pch=ifelse(iris.ksharp$cluster == 0, -1, 14)+iris.km$cluster)
> legend("topright", col=c(1:3, rep(1, 4)),
+ pch=c(rep(19, 3), 15:17, 1),
+ legend=c("I. setosa", "I. versicolor", "I. virginica",
+ "Cluster I", "Cluster II", "Cluster III", "'noise'"))
> Misclass(iris.ksharp$cluster, iris$Species, ignore=0, best=TRUE)
Best classification table:
    obs
pred setosa versicolor virginica
   1     50          0         0
   2      0         44         7
   3      0          0        34
Misclassification errors (%):
    setosa versicolor  virginica
```

```
       0.0          0.0         17.1
Mean misclassification error: 5.7%
Note: data contains NAs
```

(In the example above, ksharp() used features of the kmeans() object but it can use any clustering. On the next step, it calculated sharpness measures and used them to correct cluster memberships. Please **review** the plot yourself. Do you understand how legend is done?)

As you see, cluster sharpening allowed to reduce misclassification error twice, and also to reveal the most weak, "noisy" points.

* * *

Finally, we frequently supply the desired number of clusters to clustering methods. What if we teach them more? We might supply the information which data points are better place in one cluster, and which data points we prefer not to unite in the one cluster.

In the above clusterings of planet atmospheres, it was annoying to see that Mercury and Earth are constantly clustered together. Let us set a *constraint* of not uniting them in one cluster:

```
> aad <- dist(t(atmospheres))
> aadu <- Updist(aad, unlink=list(c("Earth", "Mercury"))) # shipunov
> plot(hclust(aadu))
```

(Please **review** this plot yourself. Do you see how Earth is now placed?)

Function Updist() does its job but it is very straightforward. More efficient implementation (but only for *k*-means-like clustering) of these educated methods could be found in the conclust package (for example, in mpckm() function).

9.7 How to choose the right method

At the end of the chapter, we decided to place the simple decision tree (Fig. 9.11) which allows to select some most important multivariate methods. Please note that if you decide to transform your data (for example, make a distance matrix from it), then you might access other methods.

So which classification method to use? There are generally two answers: (1) this which work best with your data and (2) as many as possible. The second makes the perfect sense because human perception works the same way, using all possible models until it reaches stability in recognition. Remember some optical illusions

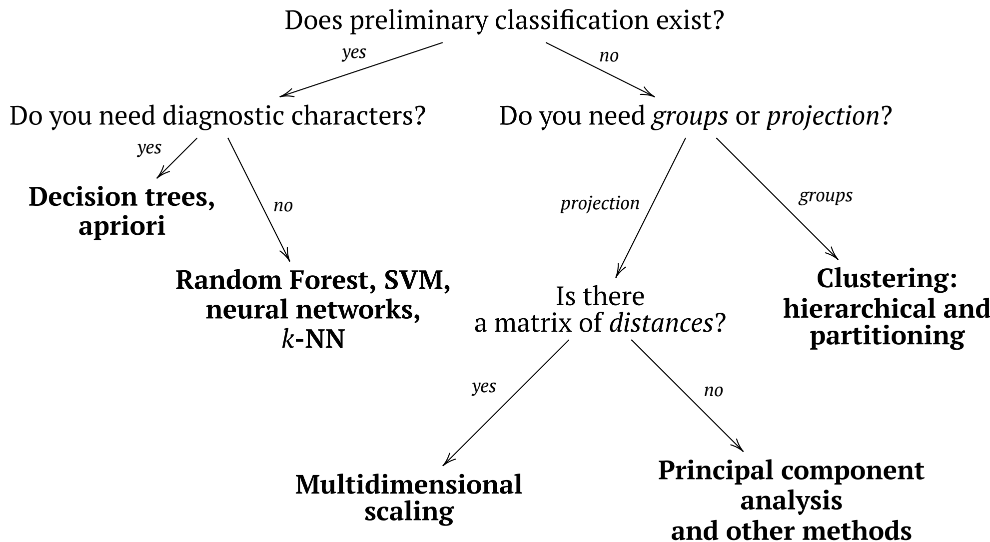


Figure 9.11: How to find the correct multivariate method.

(e.g., the famous duck-rabbit image, Fig. 9.12) and Rorschach inkblot test. They illustrate how flexible is human cognition and how many models we really use to recognize objects.

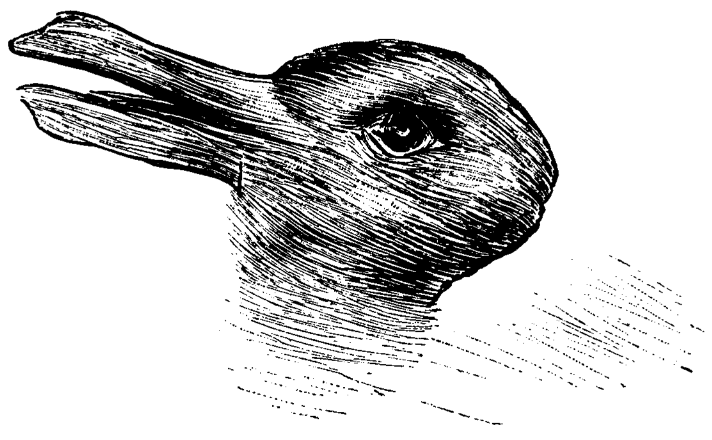


Figure 9.12: Duck-rabbit image presents two alternative recognition models.

9.8 Answers to exercises

Answer to the plant species classification tree exercise. The tree is self-explanatory but we need to build it first (Fig. 9.13):

```
> eq.tree <- tree(eq[, 1] ~ ., eq[, -1]) # shipunov
> plot(eq.tree); text(eq.tree)
```

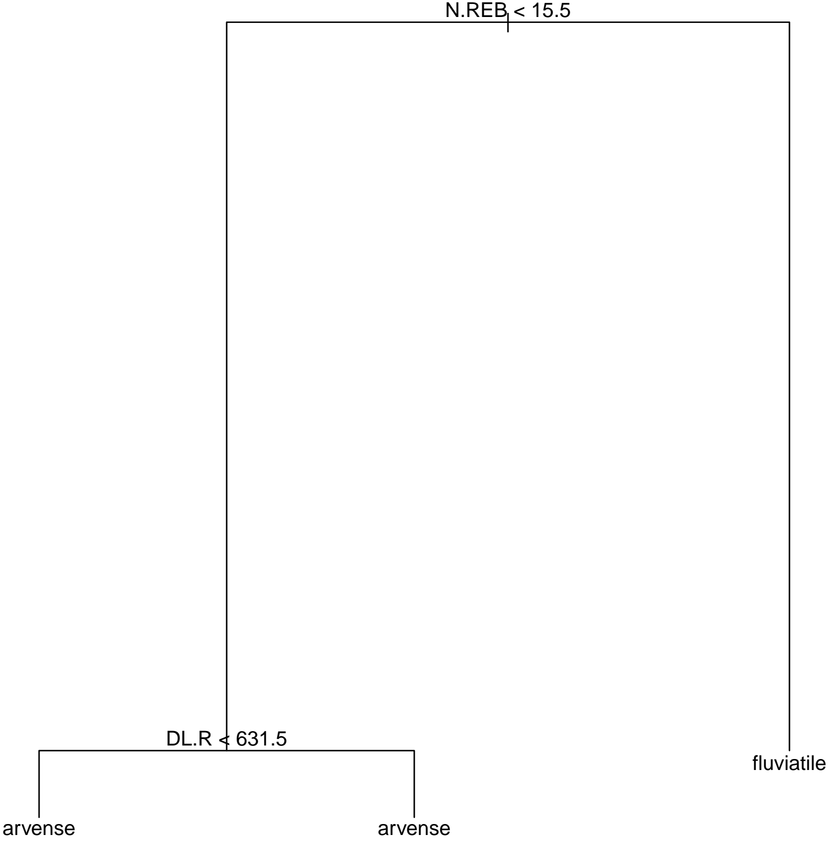


Figure 9.13: Classification tree shows that two horsetail species differ by `N.REB` character (number of stem ridges)

Appendices

Appendix A

Example of R session

The following is for impatient readers who prefer to learn R in a speedy way. They will need to type all commands listed below. Please do not copy-paste them but exactly *type* from the keyboard: that way, they will be much easier to remember and consequently to learn. For each command, we recommend to read the help (call it with ?command). As an exception from most others parts of this book, R output and plots are generally not shown below. You will need to **check and get** them yourself. We strongly recommend also to "play" with commands: modify them and look how they work.

All of the following relates with an imaginary data file containing information about some insects. Data file is a table of four columns separated with tabs:

```
SEX     COLOR   WEIGHT  LENGTH
0       1       10.68   9.43
1       1       10.02   10.66
0       2       10.18   10.41
1       1       8.01    9
0       3       10.23   8.98
1       3       9.7     9.71
1       2       9.73    9.09
0       3       11.22   9.23
1       1       9.19    8.97
1       2       11.45   10.34
```

Companion file bugs_c.txt contains information about these characters:

```
# Imaginary insects
SEX     females 0, males 1
```

```
COLOR   red 1, blue 2, green 3
LENGTH  length of the insect in millimeters
```

A.1 Starting...

There are two possibilities to load your data:

I. Save your data on the computer first:

Create the working directory (use only lowercase English letters, numbers and underscore symbols for the name); inside working directory, create the directory `data`.

Open R. Using menu or `setwd()` command (with the full path and `/` slashes as argument), point R to the working directory (not to `data`!).

Download your file:

```
> download.file("http://ashipunov.me/shipunov/data/bugs.txt",
+ "data/bugs.txt")
```

... and press `ENTER` key (press it on the end of every command).

To check if the file is now in proper location, type

```
> dir("data")
```

Among other, this command should output the name of file, `bugs.txt`.

Now read the data file and create in R memory the object `data` which will be the working copy of the data file. Type:

```
> data <- read.table("data/bugs.txt", h=TRUE)
```

II. Alternatively, you can read your data directly from URL:

```
> data <- read.table("http://ashipunov.me/data/bugs.txt", h=TRUE)
```

(This is much faster. But your data is not kept, it went directly into R memory.)

* * *

Now look on the data file:

```
> head(data)
```

Attention! If anything looks wrong, note that it is not quite handy to change data from inside R. The more sensible approach is to change the initial text file (for example, in Excel) and then `read.table()` it from disk again.

Look on the data structure: how many characters (variables, columns), how many observations, what are names of characters and what is their type and order:

```
> str(data)
```

Please note that SEX and COLOR are represented with numbers whereas they are categorical variables.

Create new object which contains data only about females (SEX is 0):

```
> data.f <- data[data$SEX == 0, ]
```

Now—the object containing data about big (more than 10 mm) males:

```
> data.m.big <- data[data$SEX == 1 & data$LENGTH > 10, ]
```

By the way, this command is easier not to type but create from the previous command (this way is preferable in R). To repeat the previous command, press ↑ key on the keyboard.

"==" and "&" are logical statements "equal to" and "and", respectively. They were used for *data selection*. Selection also requires square brackets, and if the data is tabular (like our data), there should be a comma inside square brackets which separates statements about rows from statements concerning columns.

Add new character (columns) to the data file: the relative weight of bug (the ratio between weight and length)— WEIGHT.R:

```
> data$WEIGHT.R <- data$WEIGHT/data$LENGTH
```

Check if new character is now in data table: run str(data) (use ↑ from the keyboard!)

This new character was added only to the memory copy of your data file. It will disappear when you close R. You may want to save new version of the data file under the new name bugs_new.txt in your data subdirectory:

```
> write.table(data, file="data/bugs_new.txt", quote=FALSE)
```

A.2 Describing...

Firstly, look on the basic characteristics of every character:

```
> summary(data)
```

Since SEX and COLOR are categorical, the output in these columns has no sense, but you may want to convert these columns into "true" categorical data. There are multiple possibilities but the simplest is the conversion into *factor*:

```
> data1 <- data
> data1$SEX <- factor(data1$SEX, labels=c("female", "male"))
> data1$COLOR <- factor(data1$COLOR,
+ labels=c("red", "blue", "green"))
```

(To retain the original data, we copied it first into new object `data1`. Please **check** it now with `summary()` yourself.)

Now back to the initial `data` file. `summary()` command is applicable not only to the whole data frame but also to individual characters (or variables, or columns):

```
> summary(data$WEIGHT)
```

It is possible to calculate characteristics from `summary()` one by one. Maximum and minimum:

```
> min(data$WEIGHT)
> max(data$WEIGHT)
```

… median:

```
> median(data$WEIGHT)
```

… mean for `WEIGHT` and for each character:

```
> mean(data$WEIGHT)
```

and

```
> colMeans(data)
```

… and also round the result to one decimal place:

```
> round(colMeans(data), 1)
```

(Again, the output of `colMeans()` has no sense for `SEX` and `COLOR`.)

Unfortunately, the commands above (but not `summary()`) do not work if the data have missed values (`NA`):

```
> data2 <- data
> data2[3, 3] <- NA
> mean(data2$WEIGHT)
```

To calculate mean without noticing missing data, enter

```
> mean(data2$WEIGHT, na.rm=TRUE)
```

Another way is to remove rows with `NA` from the data with:

```
> data2.o <- na.omit(data2)
```

Then, `data2.o` will be free from missing values.

* * *

Sometimes, you need to calculate the sum of all character values:

```
> sum(data$WEIGHT)
```

… or the sum of all values in one row (we will try the second row):

```
> sum(data[2, ])
```

… or the sum of all values for *every* row:

```
> apply(data, 1, sum)
```

(These summarizing exercises are here for training purposes only.)

For the categorical data, it is sensible to look how many times every value appear in the data file (and that also help to know all values of the character):

```
> table(data$SEX)
> table(data$COLOR)
```

Now transform frequencies into percents (100% is the total number of bugs):

```
> 100*prop.table(table(data$SEX))
```

One of the most important characteristics of data variability is the *standard deviation*:

```
> sd(data$WEIGHT)
```

Calculate standard deviation for each numerical column (columns 3 and 4):

```
> sapply(data[, 3:4], sd)
```

If you want to do the same for data with a missed value, you need something like:

```
> sapply(data2[, 3:4], sd, na.rm=TRUE)
```

Calculate also the *coefficient of variation* (CV):

```
> 100*sd(data$WEIGHT)/mean(data$WEIGHT)
```

We can calculate any characteristic separately for males and females. Means for insect weights:

```
> tapply(data$WEIGHT, data$SEX, mean)
```

How many individuals of each color are among males and females?

```
> table(data$COLOR, data$SEX)
```

(Rows are colors, columns are males and females.)

Now the same in percents:

```
> 100*prop.table(table(data$COLOR, data$SEX))
```

Finally, calculate mean values of weight separately for every combination of color and sex (i.e., for red males, red females, green males, green females, and so on):

```
> tapply(data$WEIGHT, list(data$SEX, data$COLOR), mean)
```

A.3 Plotting...

At the beginning, visually check the distribution of data. Make histogram:

```
> hist(data$WEIGHT, breaks=3)
```

(To see more detailed histogram, increase the number of `breaks`.)

If for the histogram you want to split data in the specific way (for example, by 20 units, starting from 0 and ending in 100), type:

```
> hist(data$WEIGHT, breaks=c(seq(0, 100, 20))
```

Boxplots show outliers, maximum, minimum, quartile range and median for any measurement variable:

```
> boxplot(data$LENGTH)
```

... now for males and females separately, using *formula* interface:

```
> boxplot(data$LENGTH ~ data$SEX)
```

There are two commands which together help to check normality of the character:

```
> qqnorm(data$WEIGHT); qqline(data$WEIGHT)
```

(These two separate commands work together to make a single plot, this is why we used semicolon. The more dots on the resulting plot are deviated from the line, the more non-normal is the data.)

Make scatterplot where all bugs represented with small circles. X axis will represent the length whereas Y axis—the weight:

```
> plot(data$LENGTH, data$WEIGHT, type="p")
```

(type="p" is the default for plot(), therefore it is usually omitted.)

It is possible to change the size of dots varying the cex parameter. Compare

```
> plot(data$LENGTH, data$WEIGHT, type="p", cex=0.5)
```

with

```
> plot(data$LENGTH, data$WEIGHT, type="p", cex=2)
```

How to compare? The best way is to have more than one graphical window on the desktop. To start new window, type dev.new().

It is also possible to change the type of plotting symbol. Figure A.1 shows their numbers. If you want this table on the computer, you might want to load the shipunov package and run:

```
> Ex.points() # shipunov
```

To obtain similar graphic examples about types of lines, default colors, font faces and plot types, run:

```
> Ex.lines() # shipunov
> Ex.cols() # shipunov
> Ex.fonts() # shipunov
> Ex.types() # shipunov
```

* * *

Use symbol 2 (empty triangle):

```
> plot(data$LENGTH, data$WEIGHT, type="p", pch=2)
```

Use text codes (0/1) for the SEX instead of graphical symbol:

```
> plot(data$LENGTH, data$WEIGHT, type="n")
> text(data$LENGTH, data$WEIGHT, labels=data$SEX)
```

(Here both commands make one plot together. The first one plots the empty field with axes, the second add there text symbols.)

The same plot is possible to make with the single command, but this works only for one-letter labels:

```
> plot(data$LENGTH, data$WEIGHT, pch=as.character(data$SEX))
```

If we want these numbers to have different colors, type:

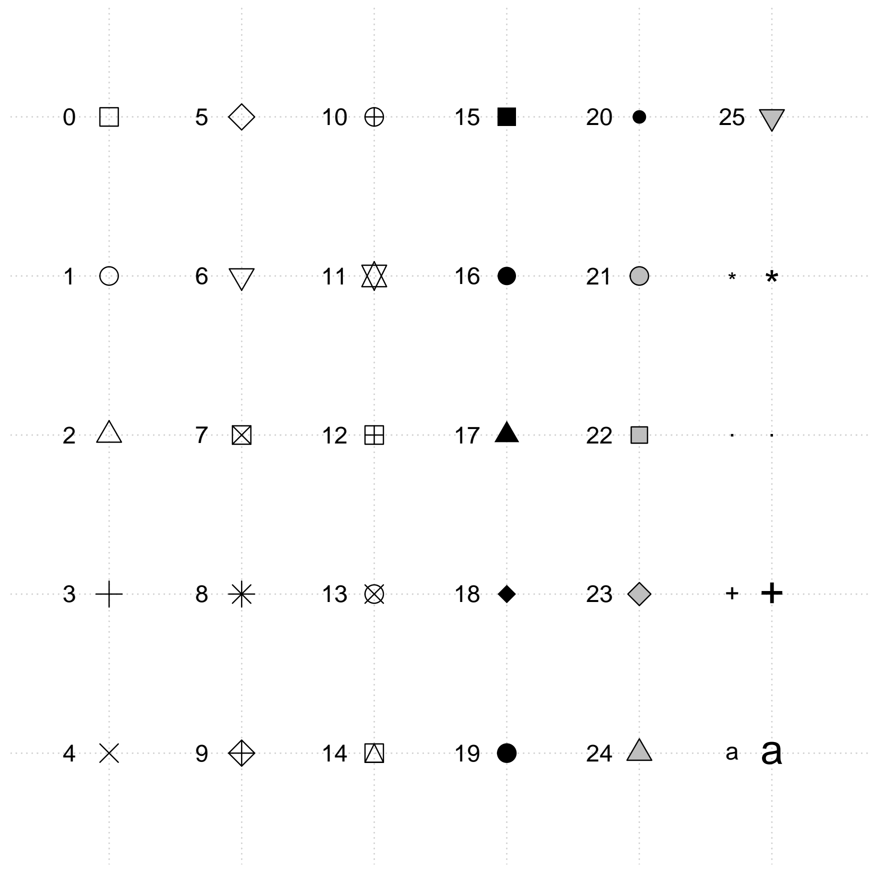


Figure A.1: Point types in R standard plots. For types 21–25, it is possible to specify different background and frame colors.

```
> plot(data$LENGTH, data$WEIGHT, type="n")
> text(data$LENGTH, data$WEIGHT, labels=data$SEX, col=data$SEX+1)
```

(Again, both commands make one plot. We added `+1` because otherwise female signs would be of `0` color, which is "invisible".)

Different symbols for males and females:

```
> plot(data$LENGTH, data$WEIGHT, type="n")
> points(data$LENGTH, data$WEIGHT, pch=data$SEX)
```

The more complicated variant—use symbols from Hershey fonts[1] which are internal in R (Fig. A.2):

```
> plot(data$LENGTH^3, data$WEIGHT, type="n",
+ xlab=expression("Volume (cm"^3*")"), ylab="Weight")
> text(data$LENGTH^3, data$WEIGHT,
+ labels=ifelse(data$SEX, "\\MA", "\\VE"),
+ vfont=c("serif","plain"), cex=1.5)
```

(Note also how `expression()` was employed to make advanced axes labels. Inside `expression()`, different parts are joined with star `*`. To know more, run `?plotmath`.)

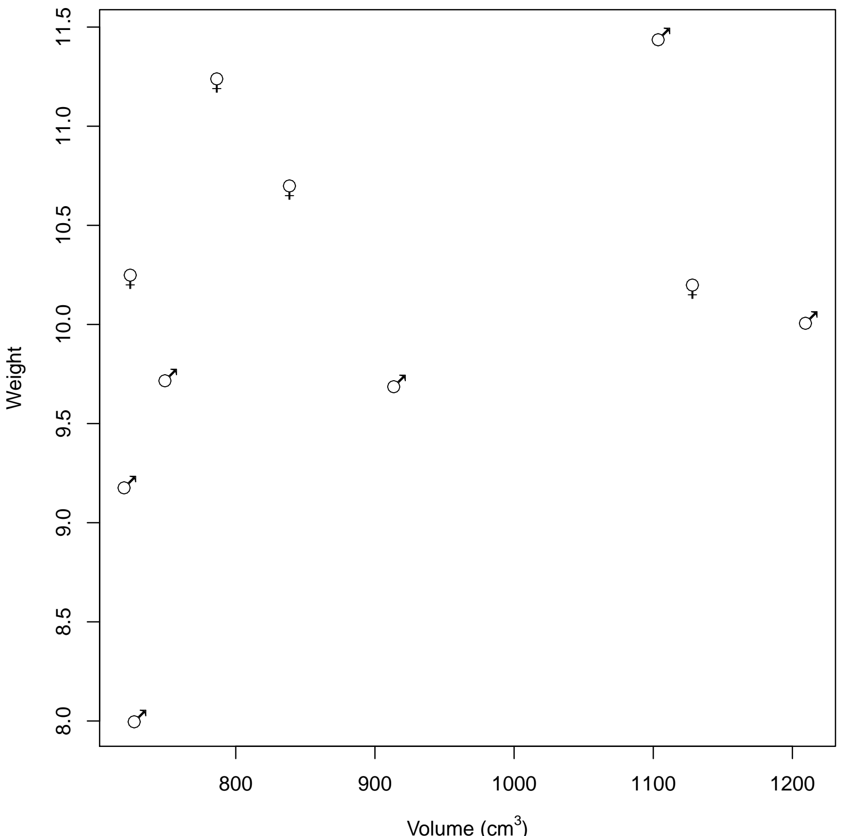


Figure A.2: Distribution of male and female bugs by size and weight (Hershey fonts used)

[1] To know which symbols are available, run `demo(Hershey)`.

We can paint symbols with different colors:

```
> plot(data$LENGTH, data$WEIGHT, type="n")
> points(data$LENGTH, data$WEIGHT, pch=data$SEX*3, col=data$SEX+1)
```

Finally, it is good to have a legend:

```
> legend("bottomright", legend=c("male", "female"),
+ pch=c(0, 3), col=1:2)
```

And then save the plot as PDF file:

```
> dev.copy(pdf, "graph.pdf")
> dev.off()
```

Attention! Saving into the external file, never forget to type `dev.off()`!

If you do not want any of axis and main labels, insert options `main=""`, `xlab=""`, `ylab=""` into your `plot()` command.

There is also a better way to save plots because it does not duplicate to screen and therefore works better in R scripts:

```
> pdf("graph.pdf")
> plot(data$LENGTH, data$WEIGHT, type="n")
> points(data$LENGTH, data$WEIGHT, pch=data$SEX*3, col=data$SEX+1)
> legend("bottomright", legend=c("male", "female"),
+ pch=c(0, 3), col=1:2)
> dev.off()
```

(Please note here that R issues no warning if the file with the same name is already exist on the disk, it simply erases it and saves the new one. Be careful!)

A.4 Testing...

The significance of difference between means for paired parametric data (t-test for paired data):

```
> t.test(data$WEIGHT, data$LENGTH, paired=TRUE)
```

... t-test for independent data:

```
> t.test(data$WEIGHT, data$LENGTH, paired=FALSE)
```

(Last example is for learning purpose only because our data is paired since every row corresponds with one animal. Also, `"paired=FALSE"` is the default for the `t.test()`, therefore one can skip it.)

Here is how to compare values of one character between two groups using formula interface:

```
> t.test(data$WEIGHT ~ data$SEX)
```

Formula was used because our weight/sex data is in the *long form*:

```
> data[, c("WEIGHT", "SEX")]
```

Convert weight/sex data into the *short form* and test:

```
> data3 <- unstack(data[, c("WEIGHT", "SEX")])
> t.test(data3[[1]], data3[[2]])
```

(Note that test results are exactly the same. Only format was different.)

If the p-value is equal or less than 0.05, then the difference is statistically supported. R does not require you to check if the dispersion is the same.

Nonparametric Wilcoxon test for the differences:

```
> wilcox.test(data$WEIGHT, data$LENGTH, paired=TRUE)
```

One-way test for the differences between three and more groups (the simple variant of ANOVA, analysis of variation):

```
> wilcox.test(data$WEIGHT ~ data$SEX)
```

Which pair(s) are significantly different?

```
> pairwise.t.test(data$WEIGHT, data$COLOR, p.adj="bonferroni")
```

(We used Bonferroni correction for multiple comparisons.)

Nonparametric Kruskal-Wallis test for differences between three and more groups:

```
> kruskal.test(data$WEIGHT ~ data$COLOR)
```

Which pairs are significantly different in this nonparametric test?

```
> pairwise.wilcox.test(data$WEIGHT, data$COLOR)
```

The significance of the correspondence between categorical data (nonparametric Pearson chi-squared, or χ^2 test):

```
> chisq.test(data$COLOR, data$SEX)
```

The significance of proportions (nonparametric):

```
> prop.test(sum(data$SEX), length(data$SEX), 0.5)
```

(Here we checked if this is true that the proportion of male is different from 50%.)

The significance of linear correlation between variables, parametric way (Pearson correlation test):

```
> cor.test(data$WEIGHT, data$LENGTH, method="pearson")
```

... and nonparametric way (Spearman's correlation test):

```
> cor.test(data$WEIGHT, data$LENGTH, method="spearman")
```

The significance (and many more) of the linear model describing relation of one variable on another:

```
> summary(lm(data$LENGTH ~ data$SEX))
```

... and analysis of variation (ANOVA) based on the linear model:

```
> aov(lm(data$LENGTH ~ data$SEX))
```

A.5 Finishing...

Save command history from the menu (on macOS) or with command

```
> savehistory("bugs.r")
```

(on Windows or Linux.)

Attention! **Always save** everything which you did in R!

Quit R typing

```
> q("no")
```

Later, you can open the saved `bugs.r` in any text editor, change it, remove possible mistakes and redundancies, add more commands to it, copy fragments from it into the R window, and finally, *run* this file as *R script*, either from within R, with command `source("bugs.r", echo=TRUE)`, or even without starting the interactive R session, typing in the console window something like `Rscript bugs.r`.

> There is just one R mistake in this chapter. Please find it. Do not look on the next page.

A.6 Answers to exercises

Answer to the question about mistake. This is it:

```
> hist(data$WEIGHT, breaks=c(seq(0, 100, 20))
```

Here should be

```
> hist(data$WEIGHT, breaks=c(seq(0, 100, 20)))
```

By the way, non-paired brackets (and also non-paired quotes) are among the most frequent mistakes in R.

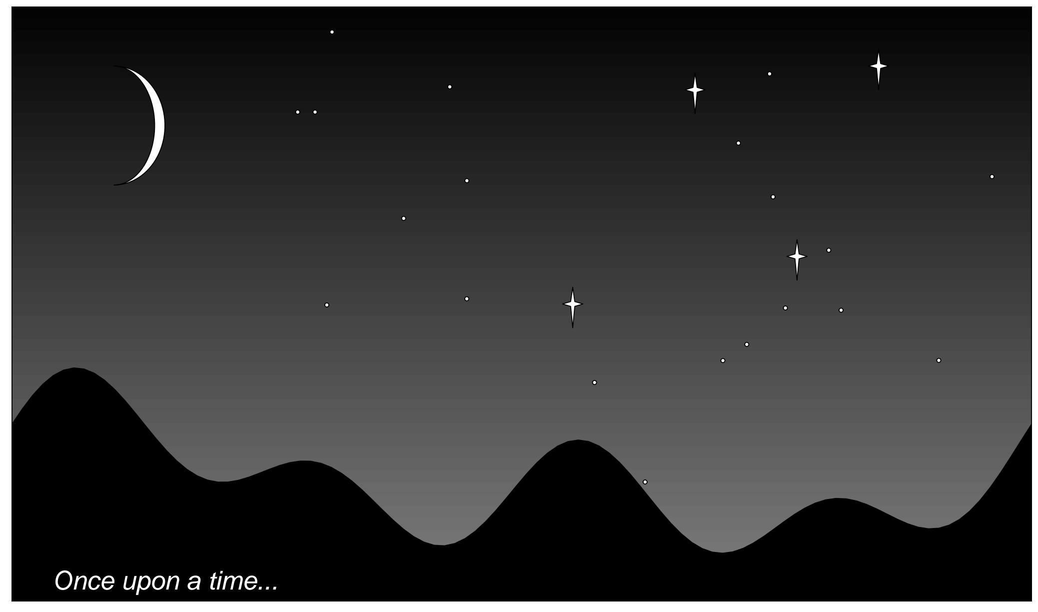
Even more, function `seq()` makes vector so function `c()` is unnecessary and the better variant of the same command is

```
> hist(data$WEIGHT, breaks=seq(0, 100, 20))
```

Now the truth is that there are *two* mistakes in the text. We are sorry about it, but we believe it will help you to understand R code better. Second is not syntactic mistake, it is more like inconsistency between the text and example. Please **find** it yourself.

Appendix B

Ten Years Later, or use R script



... there was a master student. He studied kubricks, and published a nice paper with many plots made in R. Then he graduated, started a family, and they lived happily ever after until ... ten years later, some new kubricks were discovered and he was asked to update his old plots with new data!

(By the way, the above image was made with R! Here is how—thanks to the first edition of Paul Murrell's "R Graphics" book and his `grid` package:)

```
> Gridmoon() # shipunov
```

* * *

There are recommendations for those R users who want to make their research reproducible in different labs, on different computers, and also on your own computer but 10 years later (or sometimes just 10 days after). How to proceed? Use R script!

B.1 How to make your R script

Script is a core tool for reproducible, evaluable data analysis. Every R user must know *how to make scripts*.

This is a short instruction (check also Fig. B.1) for unexperienced user:

1. Save your history of commands, just in case.

2. Then copy-paste all necessary commands from your R console into the text editor (e.g., open blank file in R editor with `file.edit()` command[1]).

   Recommendations:

   (a) *Interactive commands* which need user attention, like `help()`, `identify()`, `install.packages()`, `dev.new()`, or `url.show()`) should *not* go into the script.

   (b) All *plotting commands* should be within `pdf(...)` / `dev.off()` or similar.

   (c) It is also a good idea to place your package/script *loading commands* first, then your data loading commands like `read.table()` and finally actual calculations and plotting.

   (d) The best way to *load data* is from `data` subdirectory (stable URL is the less preferable option), *do not* use absolute paths specific to your system.

   (e) To add the single function, you may (1) *type* function name without parentheses, (2) *copy-paste* function name and output into the script and (3) after the name of function, *insert* assignment operator.

   (f) Try to *optimize* your script, remove all unnecessary commands. For example, pay attention to those which do not assign or plot anything. Some of them, however, might be useful to show your results on the screen.

[1] Linux users might want to add option `editor=`.

(g) To learn how to write your scripts better, read style guides, e.g., Google's R Style Guide on `https://google.github.io/styleguide/Rguide.xml`[2].

3. Save your script. We recommend the `.r` extension. Anyway, please do not forget to tell your OS to *show file extensions*, this could be really important.

4. Close R, *do not save workspace*.

5. Make a `test` directory inside your working directory, or (if it already exists) *remove* it (with all contents) and then *make again* from scratch.

6. *Copy* your script and `data` subdirectory into `test` directory. Note that the master version of script (were you will insert changes) should stay outside of `test` directory.

7. Start R, make `test` the working directory.

8. Run your script from within R via `source(script_name.r, echo=TRUE)`

   Note that: (a) R runs your script *two times*, first it checks for errors, second performs commands; and (b) all warnings will concentrate at the end of output (so please do not worry).

   It is really important to check your script *exactly* as described above, because in this case commands and objects saved in a previous session will not interfere with your script commands.[3]

9. If everything is well (please check especially if all plot files exist and open correctly in your independent viewer), then your script is ready.

   If not, open script in the editor and try to find a mistake (see below), then correct, close R, re-create (delete old and make new) `test` directory and repeat.

When your script is ready, you may use it as the most convenient way to protocol and even to report your work. The most important is that your script is self-sufficient, loads all data, loads all packages and makes all plots itself.

* * *

Actually, this book is the one giant R script. When I run it, all R plots are re-created. This is the first plus: the exact correspondence between code and plots. Second plus is that all code is checked with R, and if there is a mistake, it will simply stop. I do not control textual output because I want to modify it, e.g., to make it fit better with the text.

[2]Package `lintr` contains `lint()` command which checks R scripts.

[3]Alternatively, you can use non-interactive way with `Rresults` shell script (see below).

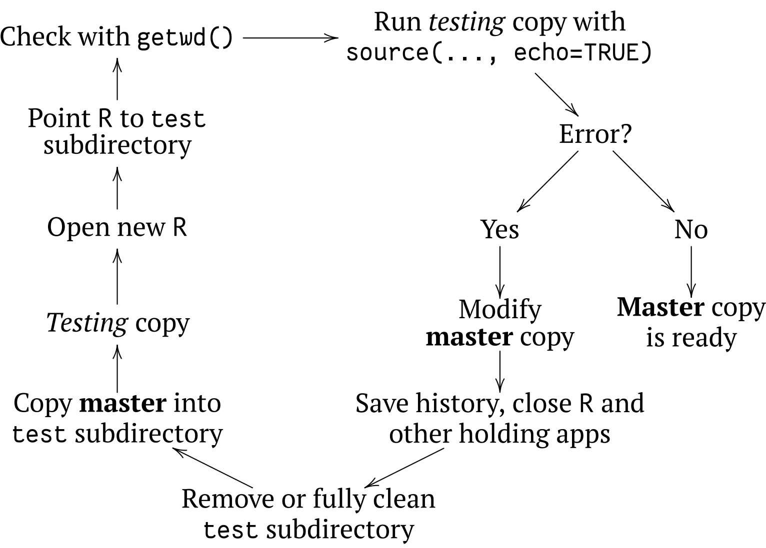


Figure B.1: How to test your R script.

▌ Some R code in this book does not go trough R. Can you find some?

B.2 My R script does not work!

What if your script does not work?

Most likely, there is some message (which you probably do not understand) but outputs nothing or something inappropriate. You will need to *debug* (Fig. B.2) your script!

- First is to find *where exactly* your script fails. If you run `source()` command with `echo=TRUE` option, this is possible just by looking into output. If this is still not clear, *run the script piece by piece*: open it in any simple text editor and copy-paste pieces of the script from the beginning to the end into the R window.

- Above mentioned is related with one of the most important principles of debugging: **minimize** your code as much as possible, and find the *minimal example which still does not work*. It is likely that you will see the mistake after minimization. If not, that minimal example will be appropriate to post somewhere with a question.

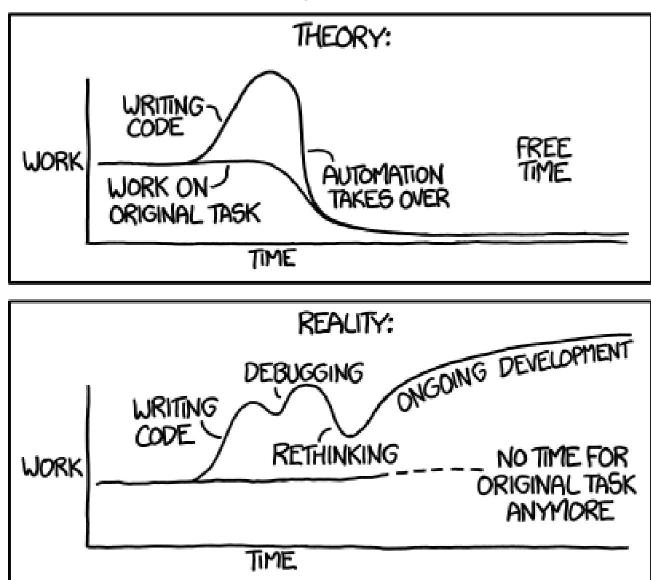


Figure B.2: Automation (taken from XKCD, http://xkcd.com/1319/).

- Related with the above is that if you want to ask somebody else about your R problem, make not only minimal, but *minimal self-contained* example. If your script loads some data, attach it to your question, or use some *embedded* R data (like `trees` or `iris`), or *generate* data with `sample()`, `runif()`, `seq()`, `rep()`, `rnorm()` or other command. Even R experts are unable to answer questions without data.
- Back to the script. In R, many expressions are "Russian dolls" so to understand how they work (or why they do not work), you will need to take them to pieces, "undress", removing parentheses from the outside and going deeper to the core of expression like:

```
> plot(log(trees$Volume), 1:nrow(trees))
> log(trees$Volume)
> trees$Volume
> trees
> 1:nrow(trees)
> nrow(trees)
```

This research could be interleaved with occasional calls to the help like `?log` or `?nrow`.

- To make smaller script, do not remove pieces forever. Use *commenting* instead, both one-line and multi-line. The last is not defined in R directly but one can use:

```
> if(0) {
> ## print here anything _syntactically correct_
> }
```

- If your problem is likely within the large function, especially within the cycle, use some way to “look inside”. For example, with `print()`:

```
> abc <- function(x)
+ {
+ for (i in 1:10) x <- x+1
+ x
+ }
> abc(5)
[1] 15 # why?
> abc <- function(x)
+ {
+ for (i in 1:10) { x <- x+1; print(x) }
+ x
+ }
> abc(5)
[1] 6
[1] 7
[1] 8
...
[1] 14
[1] 15 # OK, now it is more clear
```

  Of course, R has much more advanced functions for debugging but frequently minimization and analysis of `print()`’s (this is called *tracing*) are enough to solve the problem.

- The most common problems are mismatched parentheses or square brackets, and missing commas. Using a text editor with syntax highlighting can eliminate many of these problems. One of useful precautions is always count open and close brackets. These counts should be equal.

- Scripts or command sets downloaded from Internet could suffer from automatic tools which, for example, convert *quotes* into quotes-like (but not read-

able in R) symbols. The only solution is to carefully replace them with the correct R quotes. By the way, this is another reason why not to use office document editors for R.

- Sometimes, your script does not work because *your data changed* and now conflicts with your script.

  This should not happen if your script was made using "paranoid mode", commands which are generally safe for all kinds of data, like `mat.or.vec()` which makes vector if only one column is specified, and matrix otherwise.

  Another useful "paranoid" custom is to make checks like `if(is.matrix) { ... }` everywhere. These precautions allow to avoid situations when you updated data start to be of another type, for example, you had in the past one column, and now you have two.

  Of course, something always should be left to chance, but this means that you should be ready to conflicts of this sort.

- Sometimes, script does not work because there were *changes* in R.

  For example, in the beginning of its history, R used underscore (`_`) for the left assignment, together with `<-`. The story is when S language was in development, on some keyboards underscore was located where on other keyboards there was *left arrow* (as one symbol). These two assignment operators were inherited in R. Later, R team decided to get rid of underscore as an assignment. Therefore, these older scripts might not work in newer R.

  Similar is the recent example when R version 4 and younger introduced new `FALSE` value for the global `stringsAsFactors` option. Consequently, many older scripts (including some of this book commands) which rely on the habit that all character vectors would be converted into factors, failed to work properly.

  Another example was to change clustering `method="ward"` to `method="ward.D"`. This was because initial implementation of Ward's method worked well but did not reflect the original description. Consequently, in older versions of R newer scripts might stop to work.

  Fortunately, in R cases like first and second (broken *backward compatibility*) or third (broken *forward compatibility*) are rare. They are more frequent in R packages though.

- If you downloaded the script and do not understand what it is doing, use minimization and other principles explained above. But even more important is to *play* with a script, change options, change order of commands, feed it with

different data, and so on. Remember that (almost) everything what is made by one human could be deciphered by another one.

B.3 Common pitfalls in R scripting

Patient: Doc, it hurts when I do this.
Doctor: Don't do that.

To those readers who want to dig deeper, this section continues to explain why R scripts do not sometimes work, and how to solve these problems.

B.3.1 Advices

B.3.1.1 Use the Source, Luke!..

The most effective way to know what is going on is to look on the source of R function of interest.

Simplest way to access source is to type function name without parentheses. If the function is buried deeper, then try to use `methods()` and `getAnywhere()`.

In some cases, functions are actually not R code, but C or even Fortran. Download R source ot the source of the R package, open it and find out. This last method (download source) works well for simpler cases too.

B.3.1.2 Keep it simple

Try not to use any external packages, any complicated plots, any custom functions and even some basic functions (like `subset()`) without absolute need. This increases reproducibility and makes your life easier.

Analogously, it is better to avoid running R through any external system. Even macOS R GUI shell has problems (remember history issues?). RStudio and Jupyter are great pieces of software but they bring new layers of complexity along with new problems.

B.3.1.3 Learn to love errors and warnings

They help! If the code issues error or warning, it is a symptom of something wrong. Much worse is when the code does not issue anything but produces unreliable results.

However, warnings sometimes are really boring, especially if you know what is going on and why do you have them. On macOS R GUI it is even worse because they colored in red... So use `suppressWarnings()` function, but again, only when you know what you are doing. You can think of it as of headache pills: useful but potentially dangerous.

B.3.1.4 Subselect by names, not numbers

Selecting columns by numbers (like `trees[, 2:3]`) is convenient but dangerous if you changed your object from the original one. It is always better to use longer approach and select by names, like

```
> trees[, c("Height", "Volume")]
```

When you select by name, be aware of two things. First, selection by unknown name will return `NULL` and can make new column if anything assigned on the right side. This works only for `[[` and `$`:

```
> trees[["aaa"]]
NULL
> trees$aaa
NULL
> trees[, "aaa"]
Error in `[.data.frame`(trees, , "aaa") : undefined columns selected
```

(See also "A Case of Identity" below.)

Second, negative selection works only with numbers:

```
> trees[, -c("Height", "Volume")]
Error in -c("Height", "Volume") : invalid argument to unary operator
> trees[, -which(names(trees) %in% c("Height", "Volume"))]
 [1]  8.3  8.6  8.8 10.5 10.7 10.8 11.0 11.0 11.1 ...
```

But with a help of `shipunov` package you can "minus select" by names:

```
> trees[, trees %-% c("Height", "Volume")] # shipunov
 [1]  8.3  8.6  8.8 10.5 10.7 10.8 11.0 11.0 11.1 ...
```

This package has also a convenient way to select rows:

```
> Pull(trees, Girth < 11) # shipunov
  Girth Height Volume
1   8.3     70   10.3
2   8.6     65   10.3
3   8.8     63   10.2
```

...

B.3.1.5 About reserved words, again

Try to avoid name your objects with reserved words (`?Reserved`). Be especially careful with `T`, `F`, and `return`. If you assign them to any other object, consequences could be unpredictable. This is, by the way, another good reason to write `TRUE` instead of `T` and `FALSE` instead of `F` (you cannot assign anything to `TRUE` and `FALSE`).

It is also a really bad idea to assign anything to `.Last.value`. However, using the default `.Last.value` (it is not a function, see `?.Last.value`) could be a fruitful idea.

If you modified internal data and want to restore it, use something like `data(trees)`.

B.3.2 The Case-book of Advanced R user

B.3.2.1 A Case of Were-objects

When R object undergoes some automatic changes, sooner or later you will see that it changes the type, mode or structure and therefore escapes from your control. Typically, it happens when you make an object smaller:

```
> mode(trees)
[1] "list"
> trees2 <- trees[, 2:3]
> mode(trees2)
[1] "list"
> trees1 <- trees2[, 2]
> mode(trees1)
[1] "numeric"
```

(There is even function `mat.or.vec()`, please **check** how it works.)

Data frames and matrices normally *drop dimensions* after reduction. To prevent this, use `[, , drop=FALSE]` argument. Factors, on other hand, *do not* drop levels after reductions. To prevent, use `[, drop= TRUE]`.

Empty zombie objects appear when you apply malformed selection condition:

```
> trees.new <- trees[trees[, 1] < 0, ]
> str(trees.new)
'data.frame': 0 obs. of  3 variables:
 $ Girth : num
 $ Height: num
 $ Volume: num
```

```
> 1:nrow(trees.new)
[1] 1 0
> seq_len(nrow(trees.new))
integer(0)
```

As you see, even simple 1:nrow() is dangerous in these situations because operator `:` works in both directions. Functions `seq_len()` and `seq_along()` are much safer.

To avoid such situations (there are more pitfalls of this kind), try to use `str()` (or `Str()` from `shipunov` package) each time you create a new object.

B.3.2.2 A Case of Missing Compare

If missing data are present, comparisons should be thought carefully:

```
> aa <- c(1, NA, 3)
> aa[aa != 1] # bad idea
[1] NA  3
> aa[aa != 1 & !is.na(aa)] # good idea
[1] 3
```

Another possibility is to use `which()` and numeric indexing:

```
> trees[2, 2] <- NA
> trees[trees$Height == 80, "Girth"] # bad idea
[1]   NA 11.1 14.2 17.9 18.0 18.0
> trees[which(trees$Height == 80), "Girth"] # good idea
[1] 11.1 14.2 17.9 18.0 18.0
> data(trees) # restores changed data
```

B.3.2.3 A Case of Outlaw Parameters

Consider the following:

```
> mean(trees[, 1])
[1] 13.24839
> mean(trees[, 1], .2)
[1] 12.82632
> mean(trees[, 1], t=.2)
[1] 12.82632
> mean(trees[, 1], tr=.2)
[1] 12.82632
> mean(trees[, 1], tri=.2)
[1] 12.82632
```

```
> mean(trees[, 1], trim=.2)
[1] 12.82632
> mean(trees[, 1], trimm=.2) # why?!
[1] 13.24839
> mean(trees[, 1], anyweirdoption=1) # what?!
[1] 13.24839
```

Problem is that R frequently ignores illegal parameters. In some cases, this makes debugging difficult.

However, not all functions are equal:

```
> IQR(trees[, 1])
[1] 4.2
> IQR(trees[, 1], t=8)
[1] 4.733333
> IQR(trees[, 1], type=8)
[1] 4.733333
> IQR(trees[, 1], types=8)
Error in IQR(trees[, 1], types = 8) : unused argument (types = 8)
> IQR(trees[, 1], anyweirdoption=1)
Error in IQR(trees[, 1], anyweirdoption = 1) :
  unused argument (anyweirdoption = 1)
```

And some functions are even more weird:

```
> bb <- boxplot(1:20, plot=FALSE)
> bxp(bb, horiz=TRUE) # plots OK
> boxplot(1:20, horiz=TRUE) # does not plot horizontally!
> boxplot(1:20, horizontal=TRUE) # this is what you need
```

The general reason of all these different behaviors is that functions above are internally different. The first case is especially harmful because R does not react on your misprints. Be careful.

B.3.2.4 A Case of Identity

Similar by consequences is an example when something was selected from list but the name was mistyped:

```
> prop.test(3, 23)
   1-sample proportions test with continuity correction
data:  3 out of 23, null probability 0.5
X-squared = 11.13, df = 1, p-value = 0.0008492
...
```

```
> pval <- prop.test(3, 23)$pvalue
> pval
NULL # Why?!
> pval <- prop.test(3, 23)$p.value # correct identity!
> pval
[1] 0.0008492268
```

This is not a bug but a *feature* of lists and data frames. For example, it will allow to grow them seamlessly. However, mistypes do not raise any errors and therefore this might be a problem when you debug.

B.3.2.5 The Adventure of the Floating Point

This is well known to all computer scientists but could be new to unexperienced users:

```
> aa <- sqrt(2)
> aa * aa == 2
[1] FALSE # why?!
> aa * aa - 2
[1] 4.440892e-16 # what?!
```

What is going on? Elementary, my dear reader. Computers work only with 0 and 1 and do not know about floating points numbers.

Instead of exact comparison, use “near exact” `all.equal()` which is aware of this situation:

```
> all.equal(aa * aa, 2)
[1] TRUE
> all.equal(aa * aa - 2, 0)
[1] TRUE
```

B.3.2.6 A Case of Twin Files

Do this small exercise, preferably on two computers and/or virtual machines, one under Windows and another under Linux:

```
> pdf("Ex.pdf")
> plot(1)
> dev.off()
> pdf("ex.pdf")
> plot(1:3)
> dev.off()
```

On Linux, there are two files with proper numbers of dots in each, but on Windows, there is only one file named `Ex.pdf` but with *three* dots! This is even worse on macOS, because typical installation behaves like Windows but there are other variants too.

Do not use uppercase letters in file names. And do not use any other symbols (including spaces) except lowercase ASCII letters, underscore, 0–9 numbers, and dot for extension. This will help to make your work portable.

B.3.2.7 A Case of Bad Grammar

The style of your scripts could be the matter of taste, but not always. Consider the following:

```
> aa<-3
```

This could be interpreted as either

```
> aa <- 3
```

or

```
> aa < -3
```

Always keep spaces around assignments. Spaces after commas are not so important but they will help to read your script.

* * *

Here is also a good place to speak about `<-` and `=` assignments. Second was invented more recently, mainly to save typing and to satisfy users of other programming languages (like Python), and has some shortcomings.

First, equality assignment works only on the top level. Within a function, it has a different meaning and assigns only argument value, not the variable. Consider the following example:

```
> Sum <- function(a, b) a + b
> a <- 3
> Sum(a=5, b=4)
[1] 9
> a
[1] 3 # the same
> Sum(a <- 5, b=4)
[1] 9
> a
[1] 5 # changed
```

Second, equality is the right-to-left assignment whereas the arrow has also left-to-right form, `->`. Third, there is also `==` logical operator and therefore `=` sign might be used in three different contexts.

B.3.2.8 A Case of Double Dipping

Double comparisons do not work! Use logical concatenation instead.

```
> aa <- 3
> 0 < aa < 10
Error: unexpected '<' in "0 < aa <"
> aa > 0 & aa < 10
[1] TRUE
```

B.3.2.9 A Case of Factor Join

There is no `c()` for factors in R, result will be not a factor but numerical codes. This is concerted with a nature of factors.

However, if you really want to concatenate factors and return result as a factor, `?c` help page recommends:

```
> c(factor(LETTERS[1:3]), factor(letters[1:3]))
[1] 1 2 3 1 2 3
> c.factor <- function(..., recursive=TRUE)
+ unlist(list(...), recursive=recursive)
> c(factor(LETTERS[1:3]), factor(letters[1:3]))
[1] A B C a b c
Levels: A B C a b c
```

B.3.2.10 A Case of Bad Font

Here is a particularly nasty error:

```
> ll <- seq(0, 1, 1ength=10)
Error: unexpected input in "ll <- seq(0, 1, 1en"
```

Unfortunately, well-known problem. It is always better to use good, visually discernible monospaced font (like above so you should easily spot the problem). Avoid also lowercase "l", just in case. Use "j" instead, it is much easier to spot.

By the way, error message shows the problem because it stops printing exactly where is something wrong.

B.3.2.11 A Case of Disproportionate Condition

`ifelse()` is a powerful way to use conditions because it uses many of them at once (i.e., it is vectorized). However, it is good to remember that the length of answer determined by *length of condition* argument:

```
> ifelse("a" != "b", 1:10, c(0, "A", "B")) # the first surprise
[1] 1
> ifelse(c(TRUE, FALSE, TRUE), 1, 10) # the second surprise
[1] 1 10 1
```

B.3.3 Good, Bad, and Not-too-bad

This last section is even more practical. Let us discuss several R scripts.

B.3.3.1 Good

This is an example of (almost) ideal R script:

```
1  ### PREPARATION
2
3  library(effsize)
4
5  Normal <- function(x) {
6   ifelse(shapiro.test(x)$p.value > 0.05, "NORMAL", "NON-NORMAL")
7  }
8
9  cc <-
10  read.table("http://ashipunov.info/shipunov/open/ceratophyllum.txt",
11  h=TRUE)
12
13 ### DATA PROCESING
14
15 ## check data
16 str(cc)
17 head(cc)
18 sapply(cc[, 4:5], Normal) # both non-normal
19
20 ## plot it first
21 pdf("plot1.pdf")
22 boxplot(cc[, 4:5], ylab="Position of stem, mm on grid")
23 dev.off()
```

```
24
25 ## we only need the effect size
26 cliff.delta(cc$PLANT1, cc$PLANT2)
```

Its main features:

- clearly separated parts: loading of external material (lines 1–12) and processing of the data itself (lines 13–26)
- package(s) first (line 3), then custom functions (line 5–7), then data (line 9)
- data is checked (lines 16–18) with `str()` and then checked for normality
- after checks, data was plotted first (lines 21–23), then analyzed (line 26)
- acceptable style
- every step is commented

To see how it works, change working directory to where script is located, then load this script into `R` with:

```
> source("good.r", echo=TRUE)
```

Another variant is non-interactive and therefore faster and cleaner. It requires `Rresults` script (included in `shipunov` package, works out of the box on `macOS` and `Linux`):

```
$ Rresults good.r
```

This will print both input and output to the terminal, plus also save it as a text file and save plots in one PDF file with script name. Since this book is the `R` script, you will find examples of `Rresults` output in the on-line book directory[4].

B.3.3.2 Bad

Now consider the following script:

```
1 wiltingdata<-
2  read.table("http://ashipunov.info/shipunov/open/wilting.txt",
3  h=TRUE)
4   url.show("http://ashipunov.info/shipunov/open/wilting_c.txt")
5 sapply(wiltingdata, Normality)
6 willowsdata<-wiltingdata[grep("Salix",wiltingdata$SPECIES),]
```

[4] There is, by the way, a life-hack for lazy reader: all plots which you need to **make** yourself are actually present in the output PDF file.

```
7  Rro.test(willows[,1],willows[,2])
8  summary(K(willows[,1],willows[,2]))
9  library(shipunov)
10 plot(wiltingadta)
```

It is really bad, it simply does not work. Problems start on the first line, and both interactive (with `source()`) and non-interactive (with `Rresults`) ways will show it like:

```
> wiltingdata<-
+ read.table("http://ashipunov.me/shipunov/open/wilting.txt",h=TRUE)
Error in read.table("http://ashipunov.me/shipunov/open/wilting.txt",
  duplicate 'row.names' are not allowed
Calls: source -> withVisible -> eval -> eval -> read.table
Execution halted
```

Something is really wrong and you will need to find and correct (debug) it. And since code was not commented, you have to guess what author(s) actually wanted.

Other negative features:

- no parts, no proper order of loading, checking and plotting
- interactive `url.show()` might block non-interactive applications and therefore is potentially harmful (not a mistake though)
- bad style: in particular, no spaces around assignments and no spaces after commas
- very long object names, they are hard to type

Debugging process will consist of multiple tries until we make the working (preferably in the sensible way), "not-too-bad" script. This could be prettified later, most important is to make it work.

There are many ways to debug. For example, you can open (1) R in the terminal, (2) text editor[5] with your script and probably also some advanced (3) file manager. Run the script first to see the problem. Then copy-paste from R to editor and back again.

Let us go to the first line problem first. Message is cryptic, but likely this is some conflict between `read.table()` and the actual data. Therefore, you need to look on data and if you do, you will find that data contains both spaces and tabs. This is why R was confused. You should tell it to use tabs:

[5] Among text editors, Geany is one of the most universal, fast, free and works on all main operation systems.

```
> read.table("http://ashipunov.me/shipunov/open/wilting.txt",
+ h=TRUE, sep="\t")
```

First line starts to work. This way, step by step, you will come to the next stage.

B.3.3.3 Not too bad

```
library(shipunov)
wilt <- read.table("http://ashipunov.info/shipunov/open/wilting.txt",
 h=TRUE, sep="\t")
## url.show("http://ashipunov.info/shipunov/open/wilting_c.txt")
Str(wilt)
Normality(wilt[, 2])
willowsdata <- wilt[grep("Salix", wilt$SPECIES), ]
willowsdata$SPECIES <- droplevels(willowsdata$SPECIES)
willows <- split(willowsdata$TIME, willowsdata$SPECIES)
pdf("willows.pdf")
boxplot(willows, ylab="Wilting time, minutes")
dev.off()
Rro.test(willows[[1]], willows[[2]])
summary(K(willows[[1]], willows[[2]]))
```

This is result of debugging. It is not yet fully prettified, there are no chapters and comments. However, it works and likely in the way implied by authors.

What have been changed:

- custom commands moved up to line 3 (not to the proper place, better would be line 1, but this position garantrees work)
- `url.show()` commented out
- checks added (lines 5–6)
- names shortened a bit and style improved (not very important but useful)
- plotting now plots to file, not just to screen device
- object `willows` appeared out of nowhere, therefore we had to guess what is it, why it was used, and then somehow recreate it (lines 8–9)

> We were able to recreate `willows` object but it is not the same as in initial script. What is different? Is it possible to make them the same?

B.4 Answers to exercises

Answer to question about book code. If you are really attentive, you might find that some lines of code are preceded by space before greater sign. For example, `q()` in the second chapter. Of course, I do not want `R` to exit so early. This is why this code is not processed.

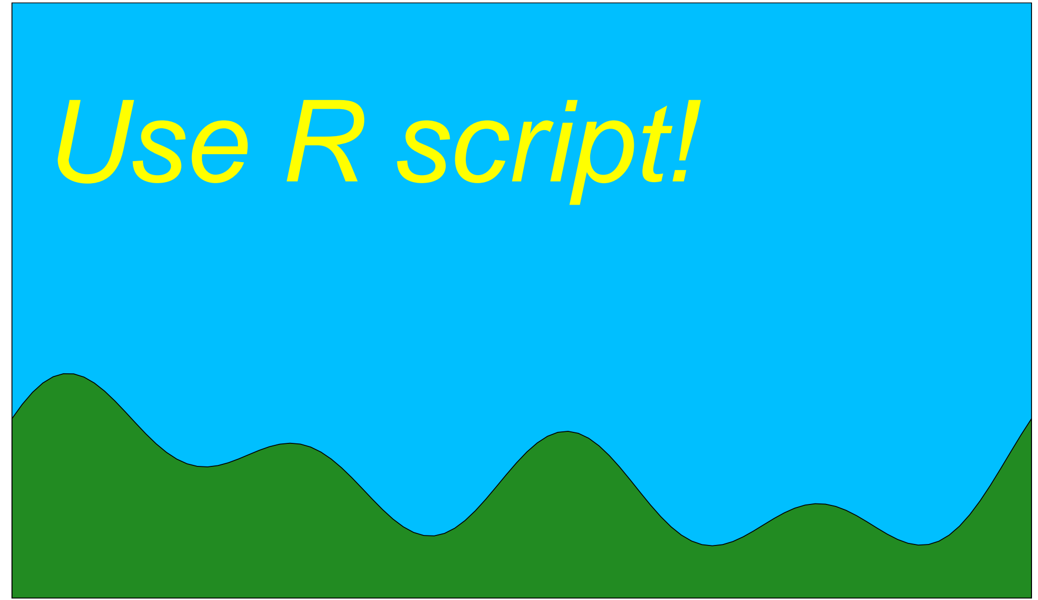
Now you can **find** other examples and think why they do not go trough `R`.

* * *

Answer to question about recreating the object impied in "bad" script. Our new object apparently is a list and requires subsetting with double brackets whereas original object was likely a matrix, with two columns, each representing one species.

We can `stack()` our list and make it the data frame, but this will not help us to subset exaclty like in original version.

The other way is to make both species parts exactly equal lengths and then it is easy to make (e.g., with `cbind()`) a matrix which will consist of two columns-species. However, this will result in loosing some data. Maybe, they did use some different version of data? It is hard to tell. Do not make bad scripts!



(And this concluding image was made with command:)

```
> Gridmoon(Nightsky=FALSE, Moon=FALSE, Stars=FALSE, # shipunov
+ Hillcol="forestgreen", Text="Use R script!", Textcol="yellow",
+ Textpos=c(0.35, 0.85), Textsize=96)
```

Appendix C

R fragments

C.1 R and databases

There are many interfaces which connect R with different database management software and there is even package sqldf which allows to work with R data frames through commands from SQL language. However, the R core also can work in the database-like way, even without serious extension. The table C.1 shows the correspondence between SQL operators and commands of R.

SELECT	`[`, `subset()`
JOIN	`merge()`, `shipunov::Recode()`
GROUP BY	`aggregate()`, `tapply()`
DISTINCT	`unique()`, `duplicated()`
ORDER BY	`order()`, `sort()`, `rev()`
WHERE	`which()`, `%in%`, `==`
LIKE	`grep()`
INSERT	`rbind()`
EXCEPT	`!` and `-`

Table C.1: Approximate correspondence between SQL operators and R functions.

One of the most significant disadvantages there is that some of these R commands are slow. The shipunov package contains ready-to-use Recode() function which is flexible and fast. With Recode(), we can operate with multiple data frames as with one.

This is important if data is organized hierarchically. For example, if we are measuring plants in different regions, we might want to have two tables: the first with regional data, and the second with results of measurements. To connect these two tables, we need a *key*, the same column which presents in both tables:

```
> head(eq_l) # locations
  N.POP WHERE  SPECIES
1     1 Tver   arvense
2     2 Tver   arvense
3     3 Tver   arvense
...
> head(eq_s) # measurements
  N.POP DL.R DIA.ST N.REB N.ZUB DL.OSN.Z DL.TR.V DL.BAZ DL.PER
1     1  424    2.3    13    12      2.0       5    3.0     25
2     1  339    2.0    11    12      1.0       4    2.5     13
3     1  321    2.5    15    14      2.0       5    2.3     13
...
```

As you see, this column is N.POP, number of population. Now we will make the dataset eq8 which contains both measurements and species names:

```
> SP.NEW <- Recode(eq_s$N.POP, eq_l$N.POP, eq_l$SPECIES)
> eq8 <- cbind(SPECIES=SP.NEW, eq_s[, -1])
> head(eq8)
  SPECIES DL.R DIA.ST N.REB N.ZUB DL.OSN.Z DL.TR.V DL.BAZ DL.PER
1 arvense  424    2.3    13    12      2.0       5    3.0     25
2 arvense  339    2.0    11    12      1.0       4    2.5     13
3 arvense  321    2.5    15    14      2.0       5    2.3     13
...
```

* * *

There is another feature related with databasing: quite frequently, there is a need to convert "text to columns". This is especially important when data contains pieces of text instead of single words:

```
> mm <- c("Plantago major", "Littorella uniflora")
> do.call(rbind, strsplit(mm, split=" ")) # exactly one space
```

```
     [,1]         [,2]
[1,] "Plantago"   "major"
[2,] "Littorella" "uniflora"
```

(Vectorized function `do.call()` constructs a function call from its arguments. Function `stsplit()` returns a list, and to convert it into data frame, we run `rbind()` on all list components.)

Another way to do (almost) the same is to use *connection*:

```
> read.table(textConnection(mm), sep=" ")
          V1       V2
1   Plantago    major
2 Littorella uniflora
```

(By the way, there are some differences in results between two examples above. What are these differences? Please **find out**.)

This way is even more powerful because it can handle situations when the new line is present within a text:

```
> nn <- c("a b", "c d\ne f")
> read.table(textConnection(nn), sep=" ")
  V1 V2
1  a  b
2  c  d
3  e  f
```

(Symbol `\n` represents the new line.)

* * *

There is also the *data encoding* operation which converts categorical data into binary (0/1) form. Several ways are possible:

```
> m.o
[1] L   S   XL  XXL S   M   L
Levels: S < M < L < XL < XXL
> model.matrix( ~ m.o - 1, data=data.frame(m.o))
  m.oS m.oM m.oL m.oXL m.oXXL
1    0    0    1     0      0
2    1    0    0     0      0
3    0    0    0     1      0
4    0    0    0     0      1
5    1    0    0     0      0
```

```
6    0    1    0     0      0
7    0    0    1     0      0
...
> Tobin(m.o, convert.names=FALSE) # shipunov
     S M L XL XXL
[1, ] 0 0 1  0   0
[2, ] 1 0 0  0   0
[3, ] 0 0 0  1   0
[4, ] 0 0 0  0   1
[5, ] 1 0 0  0   0
[6, ] 0 1 0  0   0
[7, ] 0 0 1  0   0
```

C.2 R and time

If we measure same object multiple times, especially at regular (sampling) intervals, we will finally have the *time series*, specific type of measurement data. While many common options of data analysis are applicable to time series, there are multiple specific methods and plots.

Time series frequently have two components, non-random and *random*. The first could in turn contain the *seasonal* component which is related with time periodically, like year seasons or day and night. The *trend* is the second part of non-random component, it is both no-random and non-periodical.

If time series has the non-random component, the later values should correlate with earlier values. This is *autocorrelation*. Autocorrelation has lags, intervals of time where correlation is maximal. These lags could be organized hierarchically.

Different time series could be *cross-correlated* if they are related.

If the goal is to analyze the time series and (1) fill the gaps within (*interpolation*) or (2) make forecast (*extrapolation*), then one need to create the time series *model* (for example, with `arima()` function).

But before the start, one will need to convert the ordinary data frame or vector into time series. Conversion of dates is probably most complicated:

```
> dates.df <- data.frame(dates=c("2011-01-01","2011-01-02",
+ "2011-01-03","2011-01-04","2011-01-05"))
> str(dates.df$dates)
 chr [1:5] "2011-01-01" "2011-01-02" "2011-01-03"
 "2011-01-04" "2011-01-05"
```

```
> dates.1 <- as.Date(dates.df$dates, "%Y-%m-%d")
> str(dates.1)
 Date[1:5], format: "2011-01-01" "2011-01-02" "2011-01-03"
 "2011-01-04" ...
```

In that example, we showed how to use `as.Data()` function to convert one type to another. Actually, our recommendation is to use the fully numerical date:

```
> d <- c(20130708, 19990203, 17650101)
> as.Date(as.character(d), "%Y%m%d")
[1] "2013-07-08" "1999-02-03" "1765-01-01"
```

The advantage of this system is that dates here are accessible (for example, for sorting) both as numbers and as dates.

And here is how to create time series of the regular type:

```
> ts(1:10,              # sequence
+ frequency = 4,        # by quartile
+ start = c(1959, 2)) # start in the second quartile 1959

     Qtr1 Qtr2 Qtr3 Qtr4
1959         1    2    3
1960    4    5    6    7
1961    8    9   10
```

(If the time series is irregular, one may want to apply `its()` from the `its` package.)

It is possible to convert the whole matrix. In that case, every column will become the time series:

```
> z <- ts(matrix(rnorm(30), 10, 3),
+ start=c(1961, 1), # start in January 1961
+ frequency=12) # by months
> class(z)
[1] "mts"    "ts"     "matrix" # multivariate ts
```

Generic `plot()` function "knows" how to show the time series (Fig. C.1):

```
> plot(z,
+ plot.type="single", # place all series on one plot
+ lty=1:3)
```

(There is also specialized `ts.plot()` function.)

There are numerous analytical methods applicable to time series. We will show some of them on the example of "non-stop" observations on carnivorous plant—sundew

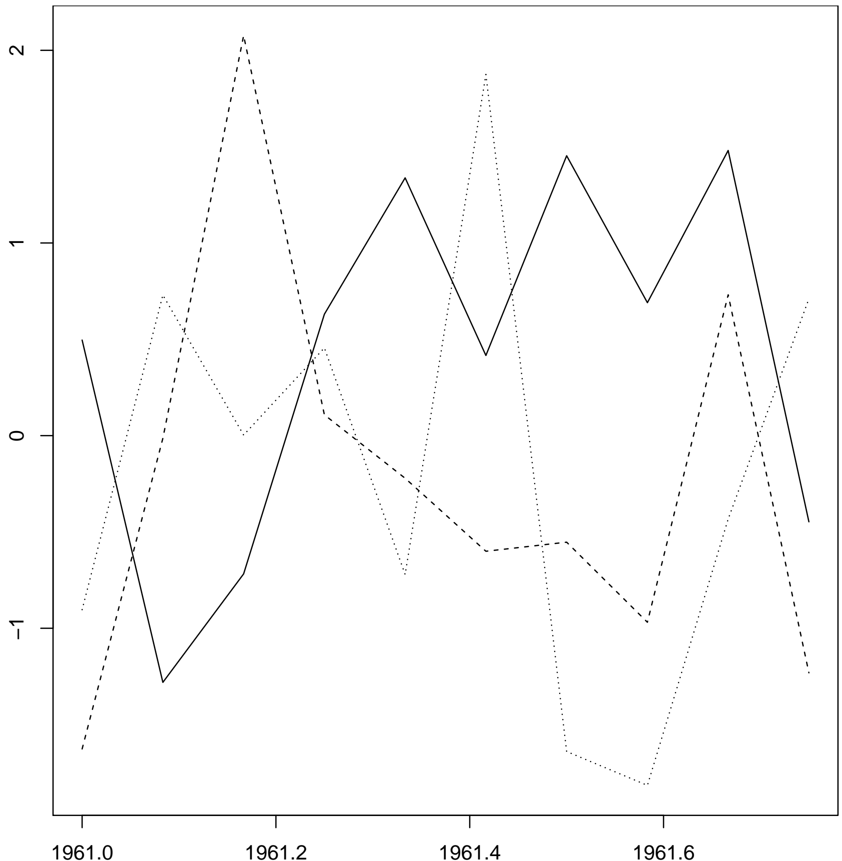


Figure C.1: Three time series with a common time.

(*Drosera rotundifolia*). In nature, leaves of sundew are constantly open and close in hope to catch and then digest the insect prey (Fig. C.2). File `sundew.txt` contains results of observations related with the fourth leaf of the second plant in the group observed. The leaf condition was noted every 40 minutes, and there were 36 observations per 24 hours. We will try to make the time series from `SHAPE` column which encodes the shape of leaf blade (1 flat, 2 concave), it is the ranked data since it is possible to imagine the `SHAPE` = 1.5. Command `file.show()` reveals this structure:

```
WET;SHAPE
2;1
1;1
1;1
...
```

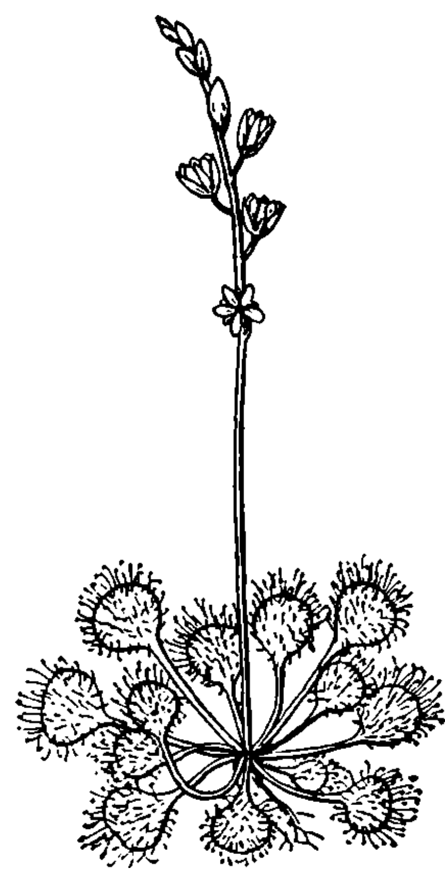


Figure C.2: Sundew, *Drosera rotundifolia*. These carnivorous plants know their time to eat.

Now we can read the file and check it:

```
> leaf <- read.table("data/sundew.txt", h=TRUE, sep=";")
> str(leaf)
'data.frame': 80 obs. of  2 variables:
 $ WET  : int  2 1 1 2 1 1 1 1 1 1 ...
 $ SHAPE   : int  1 1 1 2 2 2 2 2 2 2 ...
> summary(leaf)
     WET             SHAPE
 Min.   :1.000   Min.   :1.0
 1st Qu.:1.000   1st Qu.:1.0
 Median :1.000   Median :2.0
 Mean   :1.325   Mean   :1.7
```

```
 3rd Qu.:2.000   3rd Qu.:2.0
 Max.   :2.000   Max.   :2.0
...
```

Everything looks fine, there are no visible errors or outliers. Now convert the `SHAPE` variable into time series:

```
> shape <- ts(leaf$SHAPE, frequency=36)
```

Let us check it:

```
> str(shape)
 Time-Series [1:80] from 1 to 3.19: 1 1 1 2 2 2 2 2 2 2 ...
```

Looks perfect because our observations lasted for slightly more than 3 days. Now access the periodicity of the time series (seasonal component) and check out the possible trend (Fig. C.3):

```
> (acf(shape, main=expression(italic("Drosera")*" leaf")))

Autocorrelations of series ‘shape’, by lag

0.0000 0.0278 0.0556 0.0833 0.1111 0.1389 0.1667 0.1944 0.2222
...
```

(Please note also how `expression()` was used to make part of the title italic, like it is traditional in biology.)

Command `acf()` (auto-correlation function) outputs coefficients of autocorrelation and also draws the autocorrelation plot. In our case, significant periodicity is absent because almost all pikes lay within the confidence interval. Only first tree pikes are outside, these correspond with lags lower than 0.05 day (about 1 hour or less). It means that within one hour, the leaf shape will stay the same. On larger intervals (we have 24 h period), these predictions are not quite possible.

However, there is a tendency in pikes: they are much smaller to the right. It could be the sign of trend. Check it (Fig. C.4):

```
> plot(stl(shape, s.window="periodic")$time.series, main="")
```

As you see, there is a tendency for decreasing of SHAPE with time. We used `stl()` function (STL—“Seasonal Decomposition of Time Series by Loess” to show that. STL segregates the time series into seasonal (day length in our case), random and trend components.

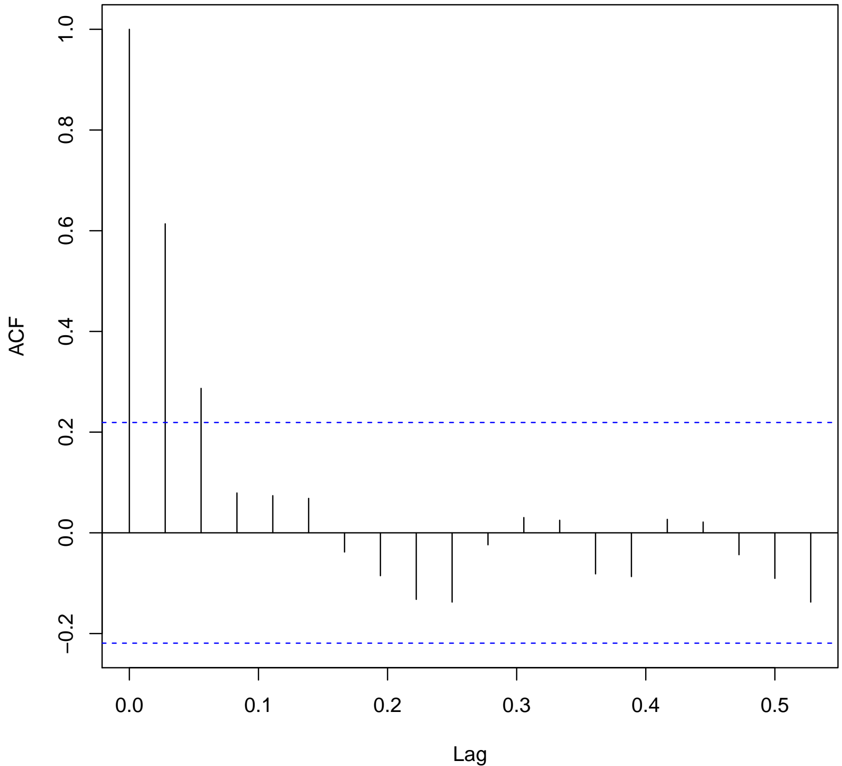


Figure C.3: Autocorrelation plot for the sundew leaf.

> WET is the second character in our sundew dataset. It shows the wetness of the leaf. Does wetness have the same periodicity and trend as the leaf shape?

C.3 R and bootstrap

All generalities like standard deviation and mean are normally taken from sample but meant to represent the whole statistical population. Therefore, it is possible that these estimations could be seriously wrong. Statistical techniques like *bootstrapping* were designed to minimize the risk of these errors. Bootstrap is based only on the given sample but try to estimate the whole population.

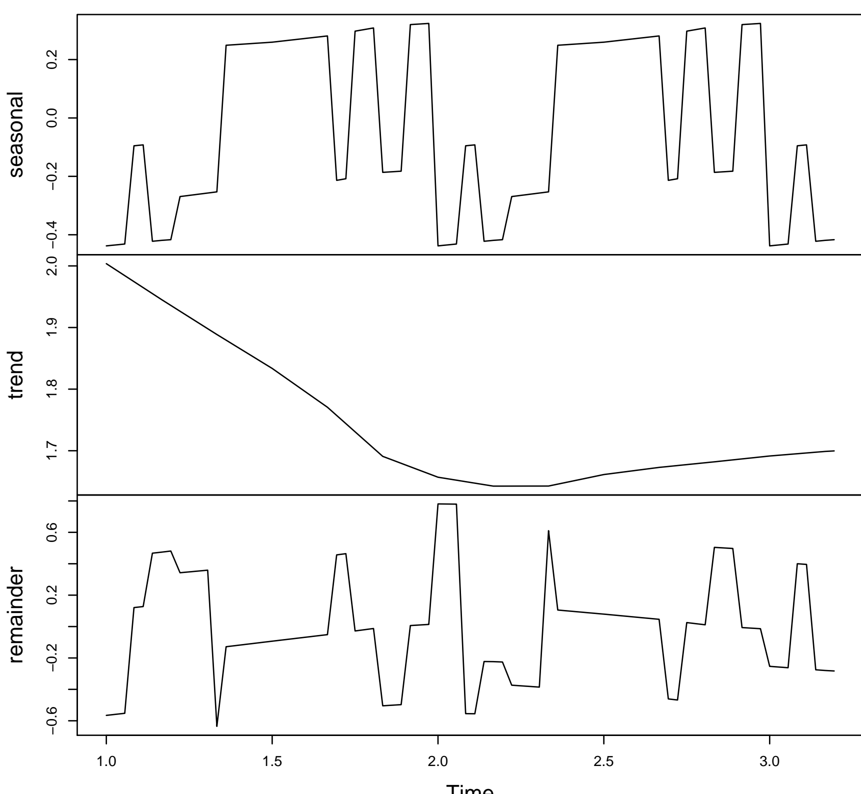


Figure C.4: Seasonal decomposition plot for the leaf of sundew. The possible trend is shown in the middle.

The idea of bootstrap was inspired by from Buerger and Raspe "Baron Munchausen's miraculous adventures", where the main character pulls himself (along with his horse) out of a swamp by his hair (Fig. C.5). Statistical bootstrap was actively promoted by Bradley Efron since 1970s but was not used frequently until 2000s because it is computationally intensive. In essence, *bootstrap* is the re-sampling strategy which replaces part of sample with the subsample of its own. In `R`, we can simply `sample()` our data *with the replacement*.

First, we will bootstrap the mean (Fig. C.6) using the advanced `boot` package:

```
> library(boot)
> ## Statistic to be bootstrapped:
> ave.b <- function (data, indices)
+ {
+ d <- data[indices]
```

Figure C.5: Baron Munchausen pulls himself out of swamp. (*Illustration of Gustave Doré*.)

```
+ return(mean(d))
+ }
>
> (result.b <- boot(data=trees$Height, statistic=ave.b, R=100))

ORDINARY NONPARAMETRIC BOOTSTRAP

Call:
boot(data=trees$Height, statistic=ave.b, R=100)

Bootstrap Statistics :
```

```
    original      bias    std. error
t1*       76 -0.1480645   0.9632468
```

(Note that here and in many other places in this book number of replicates is 100. For the working purposes, however, we recommend it to be at least 1,000.)

```
> plot(result.b)
```

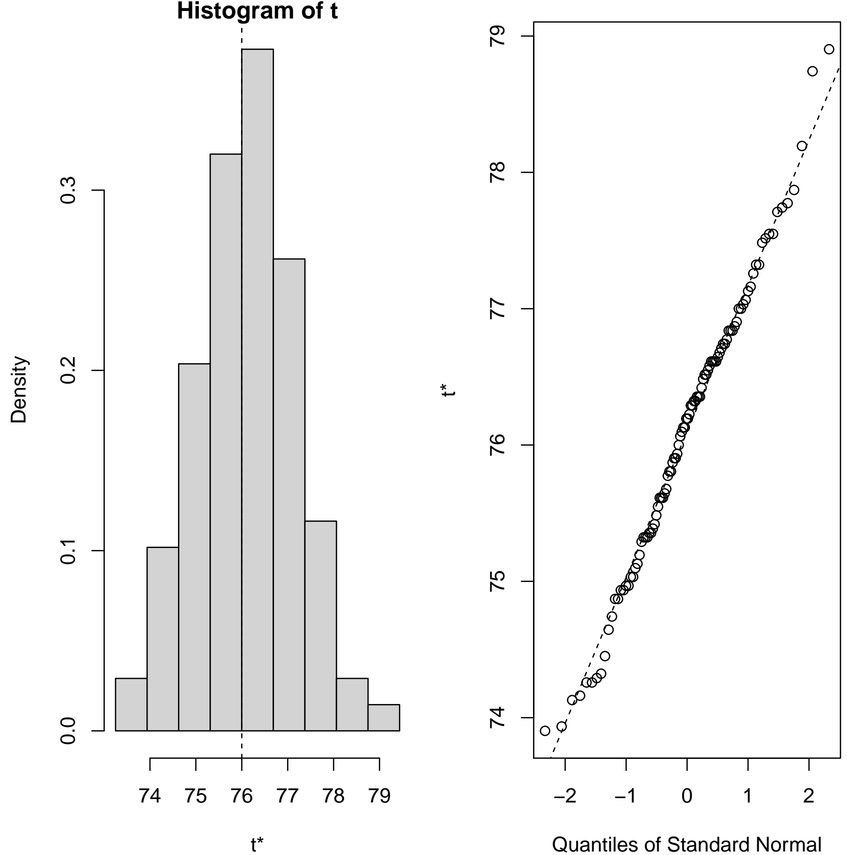


Figure C.6: Graphical representation of bootstrapping sample median.

Package `boot` allows to calculate the 95% confidence interval:

```
> boot.ci(result.b, type="bca")
BOOTSTRAP CONFIDENCE INTERVAL CALCULATIONS
Based on 100 bootstrap replicates

CALL :
```

```
boot.ci(boot.out = result.b, type = "bca")

Intervals :
Level       BCa
95%   (73.99, 77.97 )
Calculations and Intervals on Original Scale
Some BCa intervals may be unstable
```

More basic `bootstrap` package bootstraps in a simpler way. To demonstrate, we will use the `spur.txt` data file. This data is a result of measurements of spur length on 1511 *Dactylorhiza* orchid flowers. The length of spur is important because only pollinators with mouth parts comparable to spur length can successfully pollinate these flowers.

```
> spur <- scan("data/spur.txt")
Read 1511 items
> library(bootstrap)
> result.b2 <- bootstrap(x=spur, 100, function(x) mean(x))
> ## Median of bootstrapped values:
> median(result.b2$thetastar)
[1] 7.52141
> ## Confidence interval:
> quantile(result.b2$thetastar, probs=c(0.025, 0.975))
    2.5%    97.5%
7.446906 7.598170
```

Jackknife is similar to the bootstrap but in that case observations will be taking out of the sample one by one without replacement:

```
> result.j <- jackknife(x=spur, function(x) mean(x))
> ## Median of jackknifed values:
> median(result.j$jack.values)
[1] 7.516424
> ## Standard error:
> result.j$jack.se
[1] 0.04258603
> ## Confidence interval:
> quantile(result.j$jack.values, probs=c(0.025, 0.975))
    2.5%    97.5%
7.513775 7.517748
```

* * *

This is possible to bootstrap standard deviation and mean of this data even without any extra package, with `for` cycle and `sample()`:

```
> boot <- 100
> tt <- matrix(ncol=2, nrow=boot)
> for (n in 1:boot)
> {
> spur.sample <- sample(spur, length(spur), replace=TRUE)
> tt[n, 1] <- mean(spur.sample)
> tt[n, 2] <- sd(spur.sample)
> }
> (result <- data.frame(spur.mean=mean(spur), spur.sd=sd(spur),
+ boot.mean=mean(tt[, 1]), boot.sd=mean(tt[, 2])))
  spur.mean  spur.sd boot.mean  boot.sd
1  7.516082 1.655386  7.513148 1.647674
```

(Alternatively, `tt` could be an empty data frame, but this way takes more computer time which is important for bootstrap. What we did above, is the *pre-allocation*, useful way to save time and memory.)

Actually, spur length distribution does not follow the normal law (**check** it yourself). It is better then to estimate median and median absolute deviation (instead of mean and standard deviation), or median and 95% range:

```
> dact <- scan("data/dact.txt")
Read 48 items
> quantile(dact, c(0.025, 0.5, 0.975))
   2.5%     50%   97.5%
  3.700  33.500 104.825
> apply(replicate(100, quantile(sample(dact, length(dact),
+ replace=TRUE), c(0.025, 0.5, 0.975))), 1, mean)
   2.5%     50%   97.5%
  5.284  38.695 101.672
```

(Note the use of `replicate()` function, this is another member of `apply()` family.)

This approach allows also to bootstrap almost any measures. Let us, for example, bootstrap 95% confidence interval for Lyubishchev's K:

```
> sleep.K.rep <- replicate(100, K(extra ~ group,
+ data=sleep[sample(1:nrow(sleep), replace=TRUE), ]))
> quantile(sleep.K.rep, c(0.025, 0.975))
       2.5%       97.5%
```

```
0.003506551 1.832521405
```

* * *

Bootstrap and jackknife are related with numerous *resampling techniques*. There are multiple R packages (like `coin`) providing resampling tests and related procedures:

```
> library(coin)
> grades$V2 <- as.factor(grades$V2)
> wilcox_test(V1 ~ V2, data=subset(grades, V2 %in% c("A1", "B1")),
+ conf.int=TRUE)
Asymptotic Wilcoxon-Mann-Whitney Test
data:  V1 by V2 (A1, B1)
Z = -1.9938, p-value = 0.04618
alternative hypothesis: true mu is not equal to 0
95 percent confidence interval:
 -9.999284e-01 -1.944661e-05
sample estimates:
difference in location
         -1.053879e-05
```

Bootstrap is also widely used in the machine learning. Above there was an example of `Jclust()` function from the `shipunov` package. There also are `BootA()`, `BootRF()` and `BootKNN()` to bootstrap non-supervised and supervised results.

> In the open repository, data file `cuscuta.txt` (and companion `cuscuta_c.txt`) reflect measurements of the parasitic dodder plant (*Cuscuta epithymum*) infestation on multiple meadow plants. Please find if the infestation is different between lady's mantle (*Alchemilla*) and widow flower (*Knautia*) plants. Use bootstrap and resampling methods.

C.4 R and shape

Analysis of biological shape is a really useful technique. Inspired with highly influential works of D'Arcy Thompson[1], it takes into account not the linear measurements but the *whole shape* of the object: contours of teeth, bones, leaves, flower petals, and even 3D objects like skulls or beaks.

[1] Thompson D. W. 1945. On growth and form. Cambridge, New York. 1140 pp.

Naturally, shape is not exactly measurement data so it should be analyzed with special approaches. There are methods based on the analysis of curves (namely, Fourier coefficients) and methods which use *landmarks* and *thin-plate splines* (TPS). The last method allows to visualize aligned shapes with PCA (in so-called tangent space) and plot transformation grids.

In R, several packages capable to perform this statistical analysis of shape, or *geometric morphometry*. Fourier analysis is possible with `momocs`, and landmark analysis used below with `geomorph` package:

```
> library(geomorph)
> TangentSpace2 <- function(A)
+ {
+ x <- two.d.array(A)
+ pc.res <- prcomp(x)
+ pcdata <- pc.res$x
+ list(array=x, pc.summary=summary(pc.res), pc.scores=pcdata)
+ }
```

(One additional function was defined to simplify the workflow.)

Data comes out of leaf measures of alder tree. There are two data files: classic morphometric dataset with multiple linear measurements, and geometric morphometric dataset:

```
> am <- read.table("data/bigaln.txt", sep=";", head=TRUE)
> ag <- readland.tps("data/bigaln.tps", specID="imageID")
```

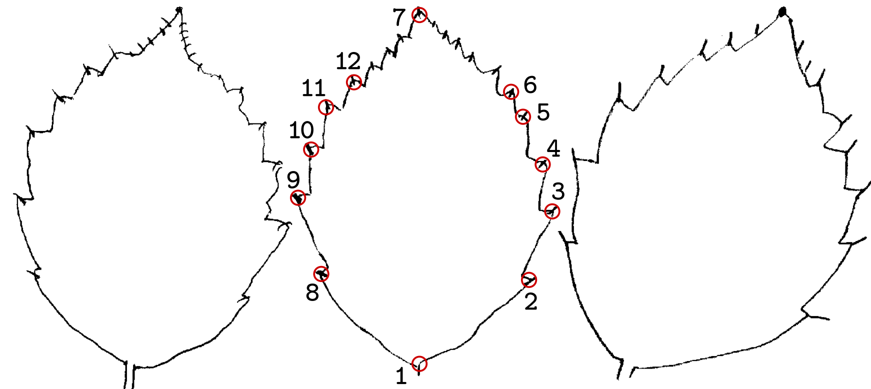


Figure C.7: Example of three alder leaf contours with landmark locations.

Geometric morphometric data was prepared with separate program, tpsDig[2]. In the field, every leaf was contoured with sharp pencil, and then all images were scanned.

Next, PNG images were supplied to tpsDig and went through landmark mapping[3]. In total, there were 12 landmarks: top, base, and endpoints of the first (lower) five pairs of primary leaf veins (Fig. C.7). Note that in geometric morphometry, preferable number of cases should be > 11 times bigger then number of variables.

Next step is the *Generalized Procrustes Analysis* (GPA). The name refers to bandit from Greek mythology who made his victims fit his bed either by stretching their limbs or cutting them off (Fig. C.8). GPA aligns all images together:

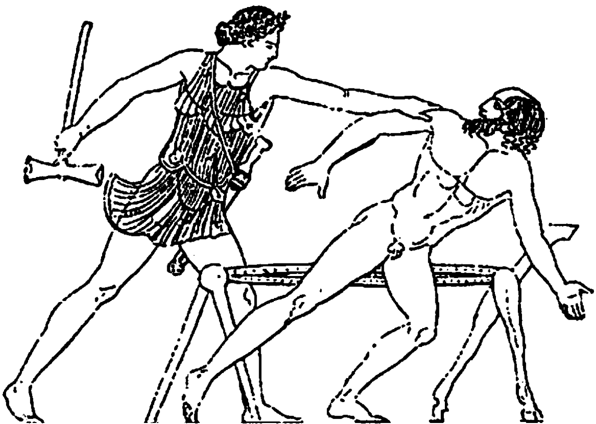


Figure C.8: Procrustes wants to fit Theseus for his bed (from Attic red-figure neck-amphora, 470–460 BC).

```
> gpa.ag <- gpagen(ag)
```

... and next—principal component analysis on GPA results:

```
> ta.ag <- TangentSpace2(gpa.ag$coords)
> screeplot(ta.ag$pc.summary) # importance of principal components
> pca.ag <- ta.ag$pc.summary$x
```

(**Check** the PCA screeplot yourself.)

[2]Rohlf F.J. tpsDig. Department of Ecology and Evolution, State University of New York at Stony Brook. Freely available at `http://life.bio.sunysb.edu/morph/`

[3]Actually, `geomorph` package is capable to digitize images with `digitize2d()` function but it works only with JPEG images.

Now we can plot the results (Fig. C.9). For example, let us check if leaves from top branches (high `P.1` indices) differ in their shape from leaves of lower branches (small `P.1` indices):

```
> pca.ag.ids <- as.numeric(gsub(".png", "", row.names(pca.ag)))
> branch <- cut(am$P.1, 3, labels=c("lower", "middle", "top"))
> b.code <- as.numeric(Recode(pca.ag.ids, am$PIC, branch,
+ char=FALSE)) # shipunov
> plot(pca.ag[, 1:2], xlab="PC1", ylab="PC2", pch=19, col=b.code)
> legend("topright", legend=paste(levels(branch), "branches"),
+ pch=19, col=1:3)
```

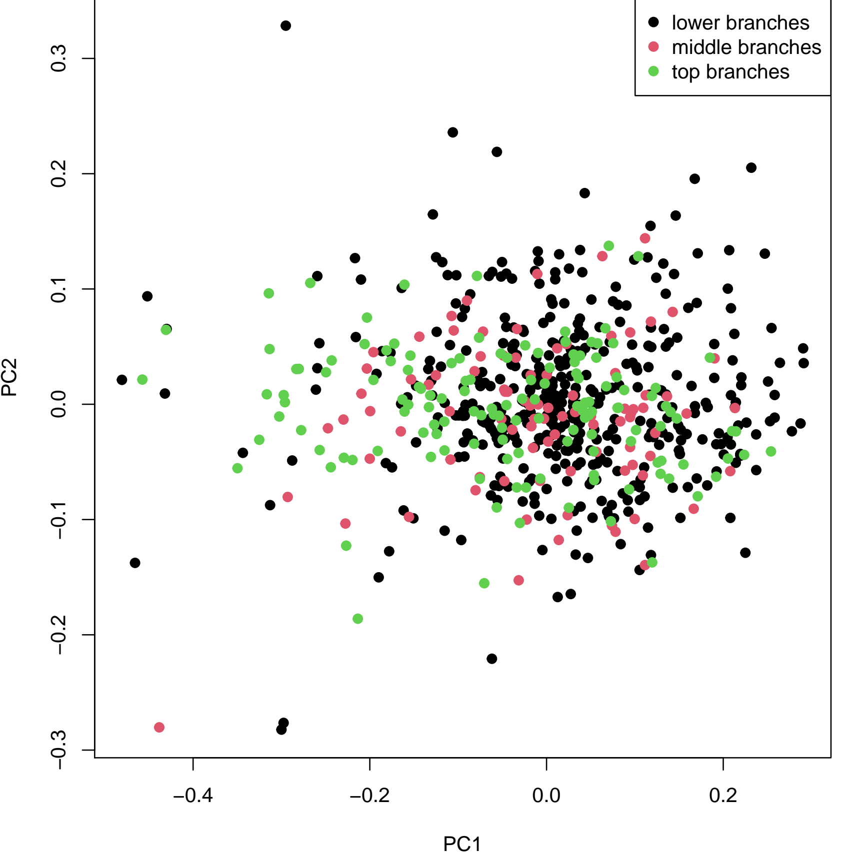


Figure C.9: Alder leaves shapes in two-dimensional tangent space made with Procrustes analysis.

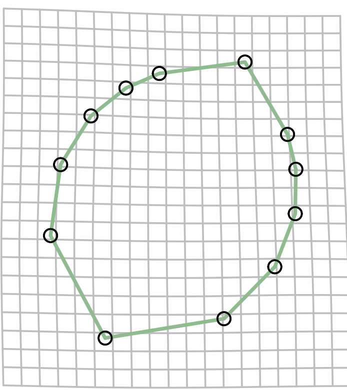
Well, the difference, if even exists, is small.

Now plot *consensus shapes* of top and lower leaves. First, we need *mean shapes* for the whole dataset and separately for lower and top branches, and then *links* to connect landmarks:

```
> c.lower <- mshape(gpa.ag$coords[, , b.code == 1])
> c.top <- mshape(gpa.ag$coords[, , b.code == 3])
> all.mean <- mshape(gpa.ag$coords)
> ag.links <- matrix(c(1, rep(c(2:7, 12:8), each=2), 1),
+ ncol=2, byrow=TRUE)
```

Finally, we plot D'Arcy Thompson's *transformation grids* (Fig. C.10):

```
> old.par <- par(mfrow=c(1, 2))
> GP <- gridPar(grid.col="grey",
+ tar.link.col="darkseagreen", tar.pt.bg=0)
> plotRefToTarget(c.lower, all.mean, links=ag.links, gridPars=GP)
> title(main="lower branches", line=-5, cex=0.8)
> plotRefToTarget(c.top, all.mean, links=ag.links, gridPars=GP)
> title(main="top branches", line=-5, cex=0.8)
> par(old.par)
```



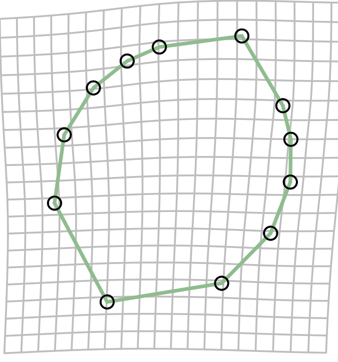


Figure C.10: D'Arcy Thompson's transformation grids (referenced to the overall mean shape) for alder leaves.

Small difference is clearly visible and could be the starting point for the further research.

C.5 R and Bayes

Most of statistical test and many methods use “throwing coin” assumption; however long we throw the coin, probability to see the face is always ½.

There is another approach, “apple bag”. Suppose we have closed, non-transparent bag full of red and green apples. We took the first apple. It was red. We took the second one. It was red again. Third time: red again. And again.

This means that red apples are likely dominate in the bag. It is because the apple bag is not a coin: it is possible to take all apples from bag and leave it empty but it is impossible to spend all coin throws. Coin throwing is unlimited, apple bag is limited.

So if you like to know proportion of red to green apples in a bag after you took several apples out of it, you need to know some *priors*: (1) how many apples you took, (2) how many red apples you took, (3) how many apples are in your bag, and then (4) calculate proportions of everything in accordance with particular formula. This formula is a famous Bayes formula but we do not use formulas in this book (except one, and it is already spent).

All in all, Bayesian algorithms use conditional models like our apple bag above. Note that, as with apple bag we need to take apples first and then calculate proportions, in Bayesian algorithms we always need sampling. This is why these algorithms are complicated and were never developed well in pre-computer era.

* * *

Below, Bayesian approach exemplified with *Bayes factor* which in some way is a replacement to p-value.

Whereas p-value approach allows only to reject or fail-to-reject null, Bayes factors allow to express preference (higher degree of belief) towards one of two hypotheses.

If there are two hypotheses, `M1` and `M2`, then Bayes factor of:

< 0 negative (support `M2`)

0–5 negligible

5–10 substantial

10–15 strong

15–20 very strong

> 20 decisive

So unlike p-value, Bayes factor is also an effect measure, not just a threshold.

To calculate Bayes factor in R, one should be careful because there are plenty of hidden rocks in Bayesian statistics. However, some simple examples will work:

Following is an example of typical two-sample test, traditional and Bayesian:

```
## Restrict to two groups
> chickwts <- chickwts[chickwts$feed %in% c("horsebean", "linseed"), ]
## Drop unused factor levels
> chickwts$feed <- factor(chickwts$feed)
## Plot data
> plot(weight ~ feed, data=chickwts, main="Chick weights")
## traditional t test
> t.test(weight ~ feed, data=chickwts, var.eq=TRUE)
## Compute Bayes factor
> library(BayesFactor)
> bf <- ttestBF(formula = weight ~ feed, data=chickwts)
> bf
Bayes factor analysis
--------------
[1] Alt., r=0.707 : 5.98 ±0%
Against denominator:
Null, mu1-mu2 = 0
---
Bayes factor type: BFindepSample, JZS
```

Many more examples are at http://bayesfactorpcl.r-forge.r-project.org/

C.6 R, DNA and evolution

In biology, majority of research is now related with DNA-based phylogenetic studies. R is aware of these methods, and one of examples (morphological though) was presented above. DNA phylogeny research includes numerous steps, and the scripting power of R could be used to automate procedures by joining them in a sort of workflow which we call Ripeline.

Book supplements contain archived folder `ripeline.zip` which includes R scripts and data illustrating work with DNA tabular database, FASTA operations, DNA alignment, flank removal, gapcoding, concatenation, and examples of how to use internal and external tree estimators.

C.7 R and reporting

Literal programming, the idea of famous Donald Knuth, is the way to interleave the code and explanatory comments. Resulted document is the *living report*: when you change your code or your data, it will be immediately reflected in the report. There many ways to create living reports in R using various office document formats but the most natural way is to use LaTeX. Let us create the text file and call it, for example, test_sweave.rnw:

```
\documentclass[b5paper,12pt]{article}
\usepackage{Sweave}

\begin{document} % Body of the document

\textsf{R} as a calculator:

<<echo=TRUE,print=TRUE>>=
1 + 1
1 + pi
sin(pi/2)
@

Image:

<<fig=TRUE>>=
plot(1:20)
@
\end{document}
```

On the next step, this file should be "fed" to the R:

```
> Sweave("test_sweave.rnw")
Writing to file test_sweave.tex
Processing code chunks ...
 1 : echo print term verbatim
 2 : echo term verbatim eps pdf
You can now run LaTeX on 'test_sweave.tex'
```

After that, you will have the new LaTeX file, test_sweave.tex. Finally, with a help of pdfLaTeX you can obtain the PDF which is shown on the Figure C.11.

R as a calculator:

```
> 1 + 1
[1] 2
> 1 + pi
[1] 4.141593
> sin(pi/2)
[1] 1
```

Image:

```
> plot(1:20)
```

1:20

Index

1

Figure C.11: The example of `Sweave()` report.

* * *

R and TEX are friendly software so it is possible to make them work together in order to automate book processing. Such books will be "semi-static" where starting data comes from the regularly updated database, and then R scripts and TEX work to create typographically complicated documents.

Flora and fauna manuals and checklists are perfect candidates for these semi-static manuals. This book supplements contain the Rmanual archived folder rmanual.zip which illustrates how this approach works on example of imaginary "kubricks" (see above).

C.8 R without graphics

If you run R on the remote server, or if you run R on your own Android smartphone (for example, within Termux app), you may face the situation when you cannot view your plots. The most used solution is to produce image plots, PDF or PNG, and then look on them outside R.

However, there is a small family of terminal (ASCII) graphics in R which allow to plot your data without graphical devices. Two commands, stem() and symnum() are present in basic R, and the command Missing.map() is in shipunov package.

There is also the txtplot package which is entirely about terminal plotting. For example, this is how to plot mtcars data:

```
> library(txtplot)
> with(mtcars, txtplot(cyl, mpg))
35 +--+------------+-------------+-------------+------------+--+
   |  *                                                         |
   |                                                            |
30 +  *                                                         +
   |  *                                                         |
25 +  *                                                         +
   |  *                                                         |
   |  *                          *                              |
20 +                             *                              +
   |                             *                          *   |
   |                                                        *   |
15 +                                                        *   +
   |                                                        *   |
   |                                                        *   |
10 +--+------------+-------------+-------------+------------+--+
```

```
      4            5             6             7            8
```

C.9 Answers to exercises

Answer to the sundew wetness question. Let us apply the approach we used for the leaf shape:

```
> wet <- ts(leaf$WET, frequency=36)
> str(wet)
Time-Series [1:80] from 1 to 3.19: 2 1 1 2 1 1 1 1 1 1 ...
> acf(wet)
> plot(stl(wet, s.window="periodic")$time.series)
```

(Plots are not shown, please **make** them yourself.)

There is some periodicity with 0.2 (5 hours) period. However, trend is likely absent.

* * *

Answer to the dodder infestation question. Inspect the data, load and check:

```
> cu <- read.table(
+ "http://ashipunov.me/shipunov/open/cuscuta.txt",
+ h=TRUE, sep="\t")
> str(cu)
'data.frame': 96 obs. of  3 variables:
 $ HOST  : chr  "Alchemilla" "Alchemilla" ...
 $ DEGREE: int  3 2 2 2 1 2 1 0 0 1 ...
 $ WHERE : int  3 3 3 3 3 1 1 0 0 1 ...
```

(Note that two last columns are not true numbers, they make sense only as *ranked* variables. Consequently, only nonparametric methods are applicable here.)

Then we need to select two hosts:

```
> cu2 <- cu[cu$HOST %in% c("Alchemilla","Knautia"), ]
> cu2$HOST <- factor(cu2$HOST)
```

It is better to convert this to the short form:

```
> cu2.s <- split(cu2$DEGREE, cu2$HOST)
```

No look on these samples graphically:

```
> boxplot(cu2.s)
```

There is a prominent difference. Now to numbers:

```
> sapply(cu2.s, median)
Alchemilla    Knautia
         2          0
> cliff.delta(cu2.s$Alchemilla, cu2.s$Knautia)
Cliff's Delta
delta estimate: 0.5185185 (large)
95 percent confidence interval:
       inf        sup
-0.1043896  0.8492326

> wilcox.test(cu2.s$Alchemilla, cu2.s$Knautia)
Wilcoxon rank sum test with continuity correction
data:  cu2.s$Alchemilla and cu2.s$Knautia
W = 41, p-value = 0.09256
alternative hypothesis: true location shift is not equal to 0
...
```

Interesting! Despite on the difference between medians and large effect size, Wilcoxon test failed to support it statistically. Why? Were shapes of distributions similar?

```
> ansari.test(cu2.s$Alchemilla, cu2.s$Knautia)
Ansari-Bradley test
data:  cu2.s$Alchemilla and cu2.s$Knautia
AB = 40, p-value = 0.7127
alternative hypothesis: true ratio of scales is not equal to 1
...

> library(beeswarm)
> la <- layout(matrix(c(1, 3, 2, 3), ncol=2, byrow=TRUE))
> for (i in 1:2) hist(cu2.s[[i]], main=names(cu2.s)[i],
+ xlab="", xlim=range(cu2.s))
> bxplot(cu2.s) ; beeswarm(cu2.s, cex=1.2, add=TRUE)
```

(Please note how to make complex layout with `layout()` command. This commands takes matrix as argument, and then simply place plot number something to the position where this number occurs in the matrix. After layout was created, you can check it with command `layout.show(la)`.)

As both Ansari-Bradley test and plots suggest, shapes of distributions are really different (Fig. C.12). One workaround is to use robust rank order test which is not so sensitive to the differences in variation:

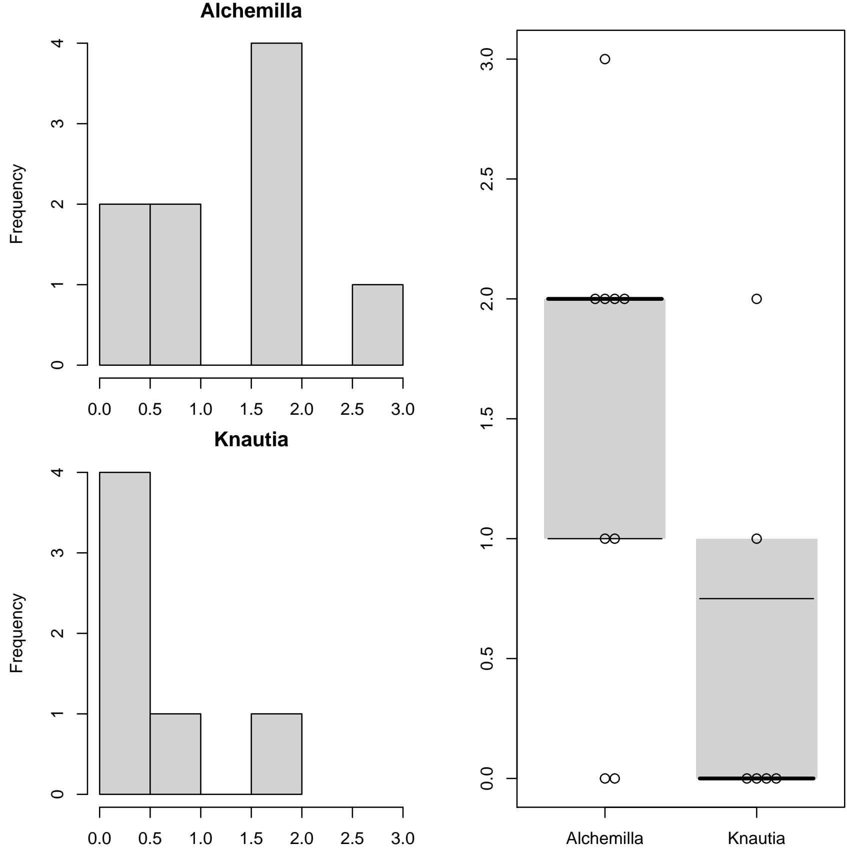


Figure C.12: Histograms of two sample distributions, plus beswarm plot with boxplot lines.

```
> Rro.test(cu2.s$Alchemilla, cu2.s$Knautia) # shipunov
        z   p.value
1.9913639 0.0464409
```

This test found the significance.

Now we will try to bootstrap the difference between medians:

```
> library(boot)
> meddif.b <- function (data, ind) { d <- data[ind];
+ median(d[cu2$HOST == "Alchemilla"]) - median(
+ d[cu2$HOST == "Knautia"]) }
> meddif.boot <- boot(data=cu2$DEGREE, statistic=meddif.b,
```

```
+ strata=cu2$HOST, R=999)
> boot.ci(meddif.boot, type="bca")
BOOTSTRAP CONFIDENCE INTERVAL CALCULATIONS
Based on 999 bootstrap replicates
CALL :
boot.ci(boot.out = meddif.boot, type = "bca")
Intervals :
Level       BCa
95%   ( 0,  2 )
Calculations and Intervals on Original Scale
```

(Please note how strata was applied to avoid mixing of two different hosts.)

This is not dissimilar to what we saw above in the effect size output: large difference but 0 included. This could be described as "prominent but unstable" difference.

That was not asked in assignment but how to analyze whole data in case of so different shapes of distributions. One possibility is the Kruskal test with Monte-Carlo replications. By default, it makes 1000 tries:

```
> library(coin)
> cu$HOST <- as.factor(cu$HOST)
> kruskal_test(DEGREE ~ HOST, data=cu, distribution=approximate())
Approximative Kruskal-Wallis Test
data:  DEGREE by
 HOST (Alchemilla, Briza, Carex flava, Cirsium, Dactylis, ...
chi-squared = 24.063, p-value = 0.1223
```

There is no overall significance. It is not a surprise, ANOVA-like tests could sometimes contradict with individual or pairwise.

Another possibility is a post hoc robust rank order test:

```
> pairwise.Rro.test(cu$DEGREE, cu$HOST) # shipunov
Pairwise comparisons using Robust rank order test
data:  cu$DEGREE and cu$HOST

                 Alchemilla Briza   Carex flava Cirsium Dactylis
Briza            0.0066     -       -           -       -
Carex flava      0.3817     0.9310  -           -       -
Cirsium          0.8392     0.1680  0.3164      -       -
Dactylis         0.3668     0.6628  0.8759      0.4834  -
Equisetum        0.8629     0.7886  0.8629      0.7460  0.9310
Erodium          0.7485     0.0022  0.7485      0.8282  0.7959
Galium           0.9427     0.0615  0.5541      0.8675  0.6320
```

```
Hieracium        0.8715     0.0615  0.6628      0.7859  0.6780
Hypericum        0.7859     < 2e-16 0.2910      0.9276  0.0269
Knautia          0.2647     0.8629  0.9427      0.3164  0.8769
...
P value adjustment method: BH
```

Now it found some significant differences but did not reveal it for our marginal, unstable case of *Alchemilla* and *Knautia*.

Appendix D

Most essential R commands

This is the short collection of the most frequently used R commands based on the analysis of almost 500 scripts (Fig. 3.8). For the longer list, check R reference card attached to this book, or R help and manuals.

? Help

<- Assign right to left

[Select part of object

$ Call list element by name

abline() Add the line from linear regression model

aov() Analysis of variation

as.character() Convert to text

as.numeric() Convert to number

as.matrix() Convert to matrix

boxplot() Boxplot

c() Join into vector

cbind() Join columns into matrix

chisq.test() Chi-squared test

cor() Correlation of multiple variables

colSums() Sum every column

cor.test() Correlation test

data.frame() Make data frame

dev.off() Close current graphic device

dotchart() Replacement for "pie" chart

example() Call example of command

factor() Convert to factor, modify factor

file.show() Show file from disk

function() ... Make new function

head() Show first rows of data frame

help() Help

hist() Histogram

ifelse() Vectorized condition

`legend()` Add legend to the plot

`library()` Load the installed package

`length()` Length (number of items) of variable

`list()` Make list object

`lines()` Add lines to the plot

`lm()` Linear model

`log()` Natural logarithm

`log10()` Decimal logarithm

`max()` Maximal value

`mean()` Mean

`median()` Median

`min()` Minimal value

`NA` Missed value

`na.omit` Skip missing values

`names()` Show names of elements

`nrow()` How many rows?

`order()` Create order of objects

`paste()` Concatenate two strings

`par()` Set graphic parameters

`pdf()` Open PDF device

`plot()` Graph

`points()` Add points (dots) to the plot

`predict()` Predict values

`q("no")` Quit R and do not save workspace

`qqnorm(); qqline()` Visual check for the normality

`rbind()` Join into matrix by rows

`read.table()` Read data file from disk into R

`rep()` Repeat

`sample()` Random selection

`savehistory()` Save history of commands (does not work under macOS GUI)

`scale()` Make all variables comparable

`sd()` Standard deviation

`source()` Run script

`str()` Structure of object

`summary()` Explain the object, e.g., return main description statistics

`t()` Transpose (rotate right)

`t.test()` Student test (t-test)

`table()` Make contingency table

`text()` Add text to the plot

`url.show()` Show the Internet file

`wilcox.test()` Wilcoxon test

`write.table()` Write to disk

Appendix E

The short R glossary

This very short glossary will help to find the corresponding R command for the most widespread statistical terms. This is similar to the "reverse index" which might be useful when you know what to do but do not know which R command to use.

Akaike's Information Criterion, AIC – `AIC()` – criterion of the model optimality; the best model usually corresponds with minimal AIC.

analysis of variance, ANOVA – `aov()` – the family of parametric tests, used to compare multiple samples.

analysis of covariance, ANCOVA – `lm(response ~ influence*factor)` – just another variant of linear models, compares several regression lines.

"apply family" – `aggregate()`, `apply()`, `lapply()`, `sapply()`, `tapply()` and others — R functions which help to avoid *loops*, repeats of the same sequence of commands. Differences between most frequently used functions from this family (applied on data frame) are shown on Fig. E.1.

arithmetic mean, mean, average – `mean()` – sum of all sample values divides to their number.

bar plot – `barplot()` – the diagram to represent several numeric values (e.g., counts).

Bartlett test – `bartlett.test()` – checks the null if variances of samples are equal (ANOVA assumption).

bootstrap – `sample()` and many others – technique of sample sub-sampling to estimate population statistics.

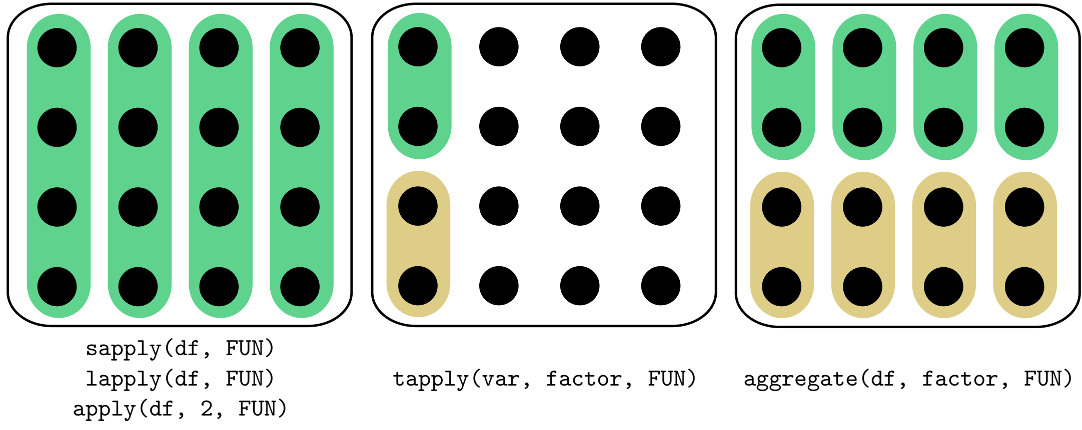


Figure E.1: Five frequently used finctions from “apply family”.

boxplot – `boxplot()` – the diagram to represent main features of one or several samples.

Chi-squared test – `chisq.test()` – helps to check if there is a association between rows and columns in the contingency table.

cluster analisys, hierarchical – `hclust()` – visualization of objects’ dissimilarities as dendrogram (tree).

confidence interval – the range where some population value (mean, median *etc.*) might be located with given probability.

correlation analysis – `cor.test()` – group of methods which allow to describe the determination between several samples.

correlation matrix – `cor()` – returns correlation coefficients for all pairs of samples.

data types – there is a list (with synonyms):

measurement:

continuous;

meristic, discrete, discontinuous;

ranked, ordinal;

categorical, nominal.

distance matrix – `dist()`, `daisy()`, `vegdist()` – calculates distance (dissimilarity) between objects.

distribution – the "layout", the "shape" of data; *theoretical distribution* shows how data should look whereas *sample distribution* shows how data looks in reality.

F-test – `var.test()` – parametric test used to compare variations in two samples.

Fisher's exact test – `fisher.test()` – similar to chi-squared but calculates (not estimates) p-value; recommended for small data.

generalized linear models – `glm()` – extension of linear models allowing (for example) the binary response; the latter is the logistic regression.

histogram – `hist()` – diagram to show frequencies of different values in the sample.

interquartile range – `IQR()` – the distance between second and fourth quartile, the robust method to show variability.

Kolmogorov-Smirnov test – `ks.test()` – used to compare two distributions, including comparison between sample distribution and normal distribution.

Kruskal-Wallis test – `kruskal.test()` – used to compare multiple samples, this is nonparametric replacement of ANOVA.

linear discriminant analysis – `lda()` – multivariate method, allows to create classification based on the training sample.

linear regression – `lm()` – researches linear relationship (linear regression) between objects.

long form – `stack(); unstack()` – the variant of data representation where group (feature) IDs and data are both vertical, in columns:

```
SEX SIZE
M   1
M   1
F   2
F   1
```

LOESS – `loess.smooth()` – Locally wEighted Scatterplot Smoothing.

McNemar's test – `mcnemar.test()` – similar to chi-squared but allows to check association in case of paired observations.

Mann-Whitney test – `wilcox.test()` – see the Wilcoxon test.

median – `median()` – the value splitting sample in two halves.

model formulas – `formula()` – the way to describe the statistical model briefly:

`response ~ influence`: analysis of the regression;

`response ~ influence1 + influence2`: analysis of multiple regression, additive model;

response ~ factor: one-factor ANOVA;

response ~ factor1 + factor2: multi-factor ANOVA;

response ~ influence * factor: analysis of covariation, model with interactions, expands into "response ~ influence + influence : factor".

Operators used in formulas:

. all predictors (influences and factors) from the previous model (used together with update());

+ adds factor or influence;

- removes factor or influence;

: interaction;

* all logical combinations of factors and influences;

/ inclusion, so "factor1 / factor2" means that factor2 is embedded within factor1 (like street is "embedded" in district, and district in city);

| condition, "factor1 | factor2" means "split factor1 by the levels of factor2";

1 intercept, so response ~ influence - 1 means linear model without intercept;

I() returns arithmetical values for everything in parentheses. It is also used in data.frame() command to skip conversion into factor for character columns.

multidimensional scaling, MDS – cmdscale() – builds something like a map from the distance matrix.

multiple comparisons – p.adjust() – see XKCD comic for the best explanation (Fig. E.2).

nonparametric – not related with a specific theoretical distribution, useful for the analysis of arbitrary data.

normal distribution plot – plot(density(rnorm(1000000))) – "bell", "hat" (Fig. E.3).

normal distribution – rnorm() – the most important theoretical distribution, the basement of parametric methods; appears, for example if one will shot into the target for a long time and then measure all distances to the center (Fig. E.4):

```
> library(plotrix)
> plot(c(-1, 1), c(-1, 1), type="n", xlab="", ylab="", axes=FALSE)
```

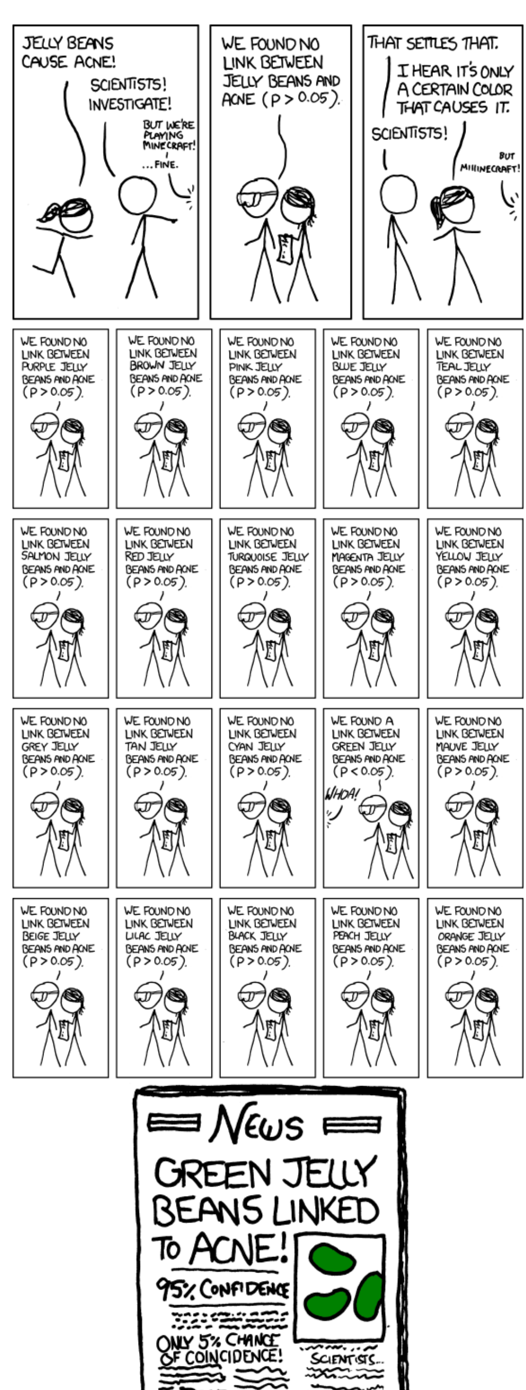


Figure E.2: Multiple comparisons (taken from XKCD, http://xkcd.com/882/).

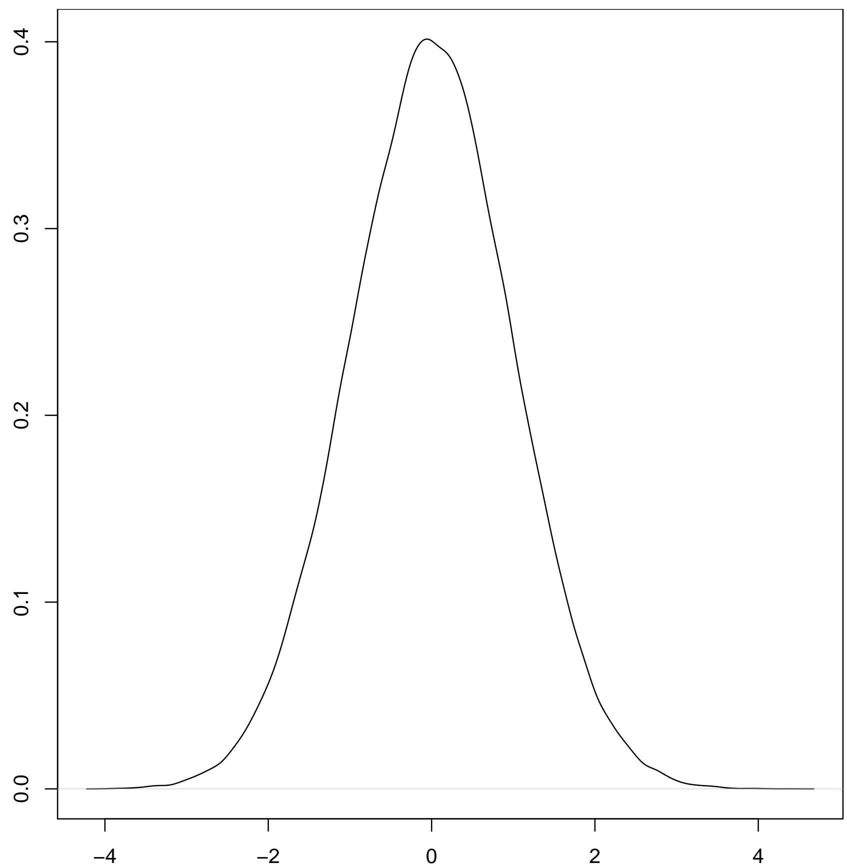


Figure E.3: Normal distribution plot.

```
> for(n in seq(0.1, 0.9, 0.1)) draw.circle(0, 0, n)
> set.seed(11); x <- rnorm(100, sd=.28); y <- rnorm(100, sd=.28)
> points(x, y, pch=19)
```

one-way test – `oneway.test()` – similar to simple ANOVA but omits the homogeneity of variances assumption.

pairwise t-test – `pairwise.t.test()` – parametric *post hoc* test with adjustment for multiple comparisons.

pairwise Wilcoxon test – `pairwise.wilcox.test()` – nonparametric *post hoc* test with adjustment for multiple comparisons.

parametric – corresponding with the known (in this book: normal, see) distribution, suitable to the analysis of the normally distributed data.

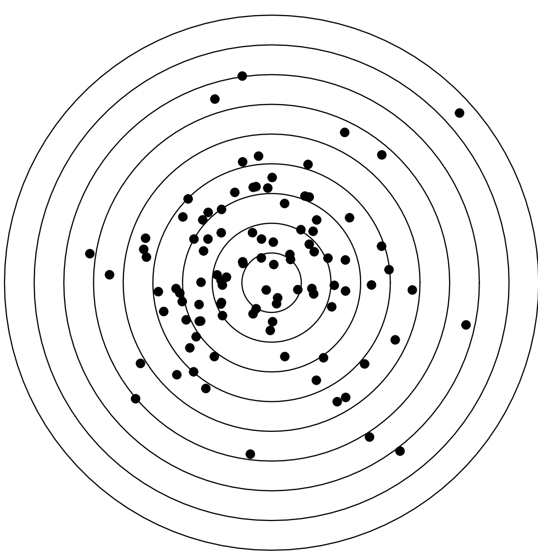


Figure E.4: Similar to shooting practice results? But this is made in R using two *normal* distributions (see the code above)!

post hoc – tests which check all groups pairwise; contrary to the name, it is not necessary to run them after something else.

principal component analysis – `princomp()`, `prcomp()` – multivariate method "projected" multivariate cloud onto the plane of principal components.

proportion test – `prop.test()` – checks if proportions are equal.

p-value – probability to obtain the estimated value if the null hypothesis is true; if p-value is below the threshold then null hypothesis should be rejected (see the "two-dimensional data" chapter for the explanation about statistical hypotheses).

robust – not so sensitive to outliers, many robust methods are also nonparametric.

quantile – `quantile()` – returns values of quantiles (by default, values which cut off 0, 25, 50, 75 and 100% of the sample).

scatterplot – `plot(x, y)` – plot showing the correspondence between two variables.

Shapiro-Wilk test – `shapiro.test()` – test for checking the normality of the sample.

short form – `stack(); unstack()` – the variant of data representation where group IDs are horizontal (they are columns):

```
M.SIZE F.SIZE
```

```
1      2
1      1
```

standard deviation – `sd()` – square root of the variance.

standard error, SE – `sd(x)/sqrt(length(x))` – normalized variance.

stem-and-leaf plot – `stem()` – textual plot showing frequencies of values in the sample, alternative for histogram.

t-test – `t.test()` – the family of parametric tests which are used to estimate and/or compare mean values from one or two samples.

Tukey HSD – `TukeyHSD()` – parametric *post hoc* test for multiple comparisons which calculates Tukey Honest Significant Differences (confidence intervals).

Tukey's line – `line()` – linear relation fit robustly, with medians of subgroups.

uniform distribution – `runif()` – distribution where every value has the same probability.

variance – `var()` – the averaged difference between mean and all other sample values.

Wilcoxon test – `wilcox.test()` – used to estimate and/or compare medians from one or two samples, this is the nonparametric replacement of the t-test.

Appendix F

References

There are oceans of literature about statistics, about R and about both. Below is a small selection of publications which are either mentioned in the text, or could be really useful (as we think) to readers of this book.

And just a reminder: if you use R and like it, do not forget to *cite it*. Run `citation()` command to see how.

Barnard C. J., Gilbert F. S., McGregor P. K. 2017. Asking Questions in Biology: A Guide to Hypothesis-testing, Analysis and Presentation in Practical Work and Research. Pearson Education.

Bookstein F. L. 2018. A Course in Morphometrics for Biologists: Geometry and Statistics for Studies of Organismal Form. Cambridge University Press.

Cann A. J. 2003. Maths from scratch for biologists. John Wiley & Sons.

Cleveland W. S. 1985. The elements of graphing data. Wandsworth Advanced Books and Software. 323 p.

Crawley M. 2007. R Book. John Whiley & Sons. 942 p.

Cumming G. 2014. The new statistics: Why and how. Psychological Science. 25: 7–29.

Dalgaard P. 2008. Introductory statistics with R. 2 ed. Springer Science Business Media. 363 p.

Efron B. 1979. Bootstrap Methods: Another Look at the Jackknife. Annals of Statistics. 7: 1–26.

Glassner A. 2018. Deep learning: from basics to practice. Seattle.

Gonick L., Smith W. 1993. The cartoon guide to statistics. HarperCollins. 230 p.

Goodman S. 2008. A dirty dozen: twelve p-value misconceptions. Seminars in Hematology. 45: 135–140.

Greenland S. 2019. Valid P-values behave exactly as they should: Some misleading criticisms of P-values and their resolution with S-values. The American Statistician. 73: 106–114.

Hevre M. 2014 onward. Aide-mémoire de statistique appliquée à la biologie — Construire son étude et analyser les résultats à l'aide du logiciel R. URL: https://www.maximeherve.com/r-et-statistiques

Kaufman L., Rousseeuw P. J. 1990. Finding groups in data: an introduction to cluster analysis. Wiley-Interscience. 355 p.

Kimble G. A. 1978. How to use (and misuse) statistics. Prentice Hall. 290 p.

Kondrashov D. A. 2016. Quantifying Life: A Symbiosis of Computation, Mathematics, and Biology. University of Chicago Press.

Li R. Top 10 data mining algorithms in plain English. URL: http://rayli.net/blog/data/top-10-data-mining-algorithms-in-plain-english/

Li R. Top 10 data mining algorithms in plain R. URL: http://rayli.net/blog/data/top-10-data-mining-algorithms-in-plain-r/

Marriott F. H. C. 1974. The interpretation of multiple observations. Academic Press. 117 p.

McKillup S. 2011. Statistics explained. An introductory guide for life scientists. Cambridge University Press. 403 p.

Murrell P. 2006. R Graphics. Chapman & Hall/CRC. 293 p.

Petrie A., Sabin C. 2005. Medical statistics at a glance. John Wiley & Sons. 157 p.

R Development Core Team. R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria.

Reinhart A. 2015. Statistics done wrong: The woefully complete guide. No Starch Press.

Rowntree D. 2000. Statistics without tears. Clays. 195 p.

Sokal R. R., Rolf F. J. 2012. Biometry. The principles and practice of statistics in biological research. W.H. Freeman and Company. 937 p.

Sprent P. 1977. Statistics in Action. Penguin Books. 240 p.

Tukey J. W. 1977. Exploratory Data Analysis. Pearson. 688 p.

Venables W. N., Ripley B. D. 2002. Modern applied statistics with S. 4th ed. Springer. 495 p.

Watt T. A. 1997. Introductory statistics for biology students. CRC Press.

Wu S. and others. 2011. Misuse of statistical methods in 10 leading Chinese medical journals in 1998 and 2008. The Scientific World Journal. 11: 2106–2114.

One Page R Reference Card

Alexey Shipunov, January 7, 2020

`c(a, b)`: concatenate, vectorize   `#` comment char
`install.packages("package1")`: install package `package1`
`is.na(x1)`: `TRUE` if `x1 == NA`
`library(package1)`: load package `package1`
`NA`: missing value   `3e3`: $3 \times 10^3 = 3000$ (e-notation)
`options(scipen=100)`: get rid of e-notation
`<-` or `assign()`: assign
`q("no")`: quit R, do not save environment
`history(Inf)`; `savehistory(file1)`: look on command history; save history (not on macOS GUI)
`;` used to separate commands   `::` used as `package::command()`
`Tab`; `Up`; `Ctrl+U`: complete; repeat command; delete command (not on macOS GUI)

Help

`example(com1)`: run examples for command `com1`
`help(com1)` or `?com1`: help about command `com1`
`help(package=rpart)`: help for the package, e.g. `rpart`
`function1`; `methods(function2)`; `getAnywhere(method2)`: look on the `function1` and 2 codes
`??"topic1"`: finds `topic1` in all help files (slow!)

Entering and saving data

`dir(...)` and `setwd()`: list files in directory, go to another
`read.table("file1", h=T, sep=";", as.is=T)`: read data into data frame from `file1` which has header and semicolon as separator; do not convert variables into factors
`scan("file1", what="char")`: read one series of character codes from disk into variable
`sink("file1", split=TRUE)`: output to `file1` and to the terminal until `sink()`
`source("file1.r")`: run commands from file `file1.r`
`write.table(x1, "file1")`: write object `x1` to the file `file1`

Manage variables and objects

`1:3` or `c(1, 2, 3)`: **c**oncatenate `1, 2, 3` into vector
`as.data.frame(x1)`, `as.matrix(x1)`: conversion
`cbind(a1, b1, c1)` or `rbind(a1, b1, c1)`: join columns or rows into matrix
`cut(v1, 2, labels=c("small", "big"))`: split vector `v1` in two intervals
`data.frame(v1, v2)`: list from same-length vectors `v1` and `v2`
`df1$a1`: variable (column) named `a1` from data frame `df1`
`dimnames(mat1)`, or `names(df1)` and `row.names(df1)`: names of rows and columns of `mat1` or `df1`
`droplevels(factor1)`: drop unused factor levels
`grep("str1", x1)`: search `str1` in `x1`
`gsub("str1", "str2", x1)`: replace `str1` to the `str2` in `x1`
`head(df1)`: first rows of data frame
`length(v1)`, `nrow(mat1)`, `ncol(df1)`: sizes
`list1[[-5]]`: all list elements except 5th
`ls()`: list all active objects
`mat1[, 2:5]` or `mat1[, c(2, 3, 4, 5)]`: columns from 2nd to 5th
`matrix(vector1, r1, c1)`: transform `vector1` into matrix with `r1` rows and `c1` columns, columnwise
`merge(df1, df2)`: merge two data frames
`paste("cow", "boy", sep="")`: outputs `"cowboy"`
`rep(x1, n1)`: repeat vector `x1` `n1` times
`sample(x1, n1)`: sample `n1` elements from `x1` without replacement
`seq(n1, n2, n3)`: sequence from `n1` to `n2` by `n3` steps
`stack()` and `unstack()`: convert from short to long form and back again
`str(obj1)`: structure of object `obj1`
`t(mat1)`: rotate 90° matrix or data frame
`with(x1, ...)`: do something within `x1`

Cycles, conditions and functions

`for (i1 in sequence1) dosomething` : cycle
`fun1 <- function(args1) dosomething` : define function
`if (condition1) ...else ...`: single condition
`ifelse(condition1, yes, no)`: vectorized condition

Logic and math

`is.factor(obj1)`, `is.atomic(obj1)`, `is.data.frame(obj1)`: check the type of object `obj1`
`mat1[mat1 > 0]`: elements of `mat1` which are positive
`!<`, `&`, `|`, `==`: "not less", "and", "or", "equal"
`cumsum(x1)`; `diff(x1)`; `prod(x1)`; `sum(x1)`: vector math
`round(x1)`: round
`unique(x1)`: list unique elements of `x1` (could be sparse)
`*`, `^`, `sqrt(pi)`, `abs(-3)`, `log(1)`: multiplication, degree, $\sqrt{\pi}$, 3, natural logarithm
`x1 %in% x2`: which elements of `x1` are in `x2`
`which(logic1)`: indexes of all TRUE's

Descriptive statistics

`aggregate(...)`: pivot table
`apply(x1, n1, function)`: apply function to all rows (if `n1` = 1) or columns (`n1` = 2), output matrix
`colSums(mat1)`: calculate sums of every column
`rev(x1)`, `order(x1)`, `scale(x1)`, `sort(x1)`: reverse, sorting indexes, scale and center, (ascending) sort
`sapply()`; `lapply()`; `do.call()`; `replicate()`: vectorize
`summary(x1)`; `IQR(x1)`; `fivenum(x1)`; `mad(x1)`; `max(x1)`; `mean(x1)`; `median(x1)`; `min(x1)`; `sd(x1)`; `var(x1)`: descriptive statistics
`table(x1, x2)`: cross-tabulation
`tapply(x1, list1, f1)`: apply function `f1` to `x1` grouping by `list1`

Inferential statistics

`chisq.test(tab1)`: χ^2-test for table `tab1`
`cor(df1)`: (Pearson) correlations between all columns of the data frame
`cor.test(x1, x2)`: (Pearson) correlation test
`ks.test(...)`; `t.test(...)`, `wilcox.test(...)`: other tests
`lm(...)`; `glm(...)`; `aov(...)`; `anova(...)`: linear and nonlinear models, analyses of variation (ANOVA)
`predict(model1)`: predict from model
`lm(y ~ x + z, data=...)`: formula interface to the additive linear model, `y` responses on two variables, `x` and `z`

Multivariate statistics

`dist(...)`: distance calculation
`cmdscale(...)`: metric multidimensional scaling (MDA)
`hclust(...)`: hierarchical cluster analysis
`princomp(...)`; `prcomp(...)`: principal component analyses (PCA)

Plots

`boxplot(...)`, `dotchart(...)`, `hist(...)`: useful plots
`identify(...)`: reveal information from points using mouse
`legend("topleft", legend="...")`: add legend to the top left corner
`lines(...)`; `points(...)`; `text(...)`: add lines, then points, then text
`pdf("file1.pdf")`: draw into `file1.pdf` until `dev.off()`
`oldpar <- par(mfrow=c(2,1))`: plots will be stacked until `par(oldpar)`
`oldpar <- par(mar=c(0,0,0,0))`: all plot margins set to zero until `par(oldpar)`
`plot(.., cex=1|2)`: normal dot size, double dot size
`plot(.., col=0|1|2|3)`: white, black, red, green color
`plot(.., lty=0|1|2)`: no lines, straight line, dashed line
`plot(.., type="p|l|s|n")`: **p**oints, **l**ines, **s**tairs and **n**o plot
`qqnorm(vec1)`; `qqline(vec1)`: check normality

`plot(.., pch=..)`: 1 ○ 2 △ 3 + 4 × 5 ◇ 6 ▽ 7 ⊠ 8 ✳ 9 ⊕ 10 ⊕ 11 ✡ 12 ⊞ 13 ⊠ 14 ⧄ 15 ■ 16 ● 17 ▲ 18 ◆ 19 ● 20 • 21 ○ 22 □ 23 ◇ 24 △ 25 ▽ * * . · a a ? ? 0 □