Intro R for BMS Bachelor programs

CODE staff

version 2023-11-20

Contents

Introduction

This document describes how bachelor students in BMS programs will use the program language R in the courses for Research Methods and Statistics (and beyond). The document will be available at all time when making assignments and when making exams in which R is used. The document will be updated when the staff discovers problems in the way students deal with R (check the version date on top of this document).

How to use this document?

There is no need to learn this document 'by heart'. It is mainly a reference guide. Read until "Installing R and RStudio on your laptop" once before doing anything else with R. Then install R and Rstudio. After that read the rest of "1. Introducing R and RStudio" once. After this make some of the assignments we created for some of the courses. Use the materials in this reference guide, in case you get stuck. After doing three of four R assignments, read the first chapter again, and skim through the remainder of the document. Before you start using R in other courses, read through the document again, to re-familiarize yourself with R.

1. Introducing R and RStudio

What is R?

R is a software environment in which you can analyze data by typing instructions (programming). In the field of behavioral and social sciences, researchers and teachers are increasingly working with R instead of programs like STATA, SAS, SPSS and EXCEL, because R has some advantages compared to these programs:

- it is free, so you can also use it after you finished your studies;
- is is a collective enterprise, and researchers all over the world write code In R, simplifying tasks, which you can use for free too;
- it is open-source and therefore transparent;
- it is extremely flexible, and data analysis and data visualization possibilities in R are almost endless.

R allows you to manage data sets, to change data in order to facilitate analysis, to analyze data and to visualize data. R also allows you to integrate data in text files, which reduces the amount of 'copy-pasting' and thus reduces the number of mistakes made in this process. This improves the reproducibility of research.

What is RStudio?

RStudio is a 'shell' designed to help you to be more productive with R. RStudio is an Integrated Development Environment (IDE) that helps you develop programs and scripts in R. RStudio has a set of integrated tools that are helpful when analyzing and visualizing your data. More information about RStudio can be found here: https://rstudio.com/products/rstudio/features/. You will probably only encounter R via RStudio.

Why R and RStudio at BMS?

At BMS we think students in the behavioral and social sciences should have a clear understanding of 'data' and using R and RStudio will give you both a clear understanding of data and a flexible and transparent way to handle these data. R also allows us to teach statistics in a flexible way, adding new ideas (like 'big data analysis') to our curriculum more easily.

Installing R and RStudio on your laptop

For most purposes, installing R and RStudio on your laptop is straightforward. For R go to

- https://cran.uni-muenster.de

For RStudio go to:

- https://www.rstudio.com/products/rstudio/download/

and select the free version.

How to learn R?

Although we will offer you a lot of materials to learn the R basics, keep in mind that for learning R even better you need to find your way on the web to learn additional procedures and to practice even more. How?

Some suggestions:

- use a search engine (like Google) to ask questions like "how do I enter data into R?";
- use stackoverflow (https://stackoverflow.com) to check out answers to similar questions;
- find your own pet project (data with your running times, for example, or gpx files, or IMDB data);
- read blogs in which data scientists show what can be done with R.

A nice feature in R is the help function. If you need help on a function, just type a "?" in front of a function. If you type "??" in front of the function, you will even get help from all over the web.

```
# For example use this:

?data.frame
??tidyverse
```

After typing and running these commands, the answer can be found in the right side lower corner of the RStudio panel.

Terminology: working directory, console, script and more

Learning R includes learning a lot of new words like 'working directory', 'script' and 'console'. Although it might be hard to understand the terms completely by reading the explanations below while not yet working in R, you will see that in a few weeks you will understand exactly what is meant with the terms. Reread this document a few times after starting working with R.

The *working directory* is the folder on your laptop or pc or in the cloud, where all kinds of data, output (including pictures), and scripts are stored as files, so you can quickly find them back later. It is wise to create a working directory for each separate course or data project and to tell R you will use that directory for this specific project.

```
setwd("[...]")

# The [...] has to be replaced by something applicable to your computer, for
# example.

setwd("/Volumes/myname/Userdata/My Documents/Documenten/Research/2020_Local_elections")

# The current working directory (in case you forget) can be found with the
# command

getwd()
```

Make sure you can find the path of a working directory in your mac or pc: - on Mac, in Finder, the small 'radar' on top allows you to copy the location as 'pathname' - in Windows, in the File Explorer, select View in the toolbar, -> Options, select Change folder and search options, to open the Folder Options dialogue box, Click View to open the View tab. In Advanced settings, add a check mark for Display the full path in the title bar, Click Apply, Click OK to close the dialogue box.

A *script* is a set of commands, for example a command meaning 'read in my data' or 'create a nice a frequency table'. Commands are preferably combined with some *# comments* explaining what happens in the command lines. In order to distinguish between command lines and comments, comments are preceded by the # sign.

Use - - - - - to break up your script into easily readable chunks.

If you want to run a line of the script, you move the cursor to the respective line and you use ctrl+enter (cmd+enter on a Mac). The instruction will then then automatically move to the R Console and the cursor in the script automatically jumps to the next line.

The *console* panel (at the bottom in RStudio, with the prompt ">") can be used to directly type commands. For example, when using R as a simple calculator you can just type in the numbers in the console. However, you are not advised using the console: do as much as possible via the script.

```
# adding 1 + 1
> 1 + 1
[1] 2
```

Objects, including imported data, are stored in the *global environment*. When you create new objects, these objects are stored in the global environment too. For example, if you create a frequency table, you can store

the table (the object) in the global environment. The object can always be called back later in the analysis. The global environment can be saved separately and called later, although for many practical purposes it is sufficient to create a global environment anew, by using the *script* (remember: the set of commands and comments) which created that environment in the first place. The global environment can be found on the right upper side of RStudio.

Data sets, numbers, and output can be stored as *objects* in the global environment, using the *(left) assignment operator* "<-".[1] An object can be a lot of things. For example, if you use R to create a boxplot, the picture of the boxplot can be stored as an object, to be called later when needed.

When naming *objects*, use only lowercase letters, numbers, and _. Use underscores (_) to separate words within a name. For example, only use names like 'data_dpes_2021', or 'freq_tab_001'.

```
# for example, to use the 10 cases throughout a script, you can define the
# object 'n' to '10'
n <- 10

# or to create a vector of numbers, you can write ...
dataset_1 <- c(3, 2, 1, 4, 9, 6, 7, 100)

# calling this object in a script or in the console gives the contents of the
# thus created object.
dataset_1
```

```
## [1]   3   2   1   4   9   6   7 100
```

```
# this give the following output in the console:
```

Functions in R are 'commands' telling R what to do: they are 'do this' statements. R has a lot of those, and users are adding more every day. Functions end with parentheses: "()".[2] For example:

```
# The command head() means 'Show me the first part (the head) of a data set'.
# In this example the called data set is an object called dataset_1 (see
# above).

head(dataset_1)
```

Data frames and data types

Data frames in R are matrices (numbers and words stored in rows and columns) in which the columns are variables and the rows are units of observation (observations). Sometimes this data frame is called a *data matrix*, but in the R language a data matrix is a data frame with only numbers, not with strings (text) or other types of variables.
One single column (or: a variable) with numbers or with strings is called a *vector*.[3] Sometimes objects are not just data frames, but combinations of data frames and something more, or combinations of different data frames. These objects are called *lists*.

For reasons beyond the scope of this introduction, some functions only work with data frames (even if they contain only numbers), not with matrices. In these cases you have to explicitly tell R your matrix is a data frame.

Variables in a data frame have a name (like 'x' or 'gender'). These can be be called directly using the '$' sign. So 'data$gender' refers to the column/variable 'gender' in the data frame. More generally, the '$' sign refers

[1] To be fair, also "=" will work, but most people now use the "<-" assignment operator

[2] The `magrittr` package (included in the `tidyverse` package) allows you to omit parentheses () on functions that do not have arguments. Avoid this feature, because it may be confusing. Be consistent in that functions always have parentheses, objects do not.

[3] A *vector* can contain numbers (1, 1.5, 3.333 etc.), or strings (words), or factors (to be explained later), or integers (1,2,3,4,5), etc. A single vector cannot have *different* types of data.

to a 'component' of an object. If you store output as an object to the global environment, you can call a specific element of that output using the '$' sign.

There can be different types of variables in a vector/data frame. The most important are variables containing:

- logical values (True or False) (used for some dummy variables);
- a factor consisting of a limited set of attributes ('low', 'middle' and 'high' for example) (used for nominal and ordinal variables). A factor can also be stored as an 'ordered factor' and is than not merely 'nominal' but 'ordinal'.
- integer values (-1, 0, 1, 2, 3 etc) (mainly used for ordinal variables and for count variables);
- real values (1.001, 1.002, 10000) (used for interval and ratio variables);
- characters/ words ('male', 'female') (if these variables are used for dummies (like in 'male' / 'female') and for nominal variables, you have to change them to (ordered) 'factor' variables).

Operators in R

The most frequently used operators used in R, beyond the well known - (Minus), + (Plus), * (Multiplication), / (Division) and <- (the Left assignment operator), are:

operator	meaning
^	Exponentiation
!	Not
>	Greater than
==	Equal to *not simply '='*
>=	Greater than or equal to, binary
<=	Less than or equal to, binary
:	Sequence (in model formulae: interaction)
$	List subset, including a column in a data frame

Some notes on writing readable scripts

Commands can all be put in the console (the one at the bottom-left of RStudio, starting with the ">"). But since you often want to redo the same thing (trying and tweaking), and because you want to ask others for help and show them exactly what you did, we urge you to store all commands in a *script*.

Open a new script in RStudio (file -> new file -> R script). This will appear in the upper-left pane. Type all commands in the script. This script can be stored (extension is .R) and is basically a .txt file. The rule to use the script as much as possible also applies to things you can do in RStudio. Importing your data **can** be done by using RStudio, but we urge you to use commands like `read.csv()` in the script too.

When writing a script, start with a title, preceded with the # (because it is not a command) on top, for example:

```
# This script is for computing the mean maximum temperature for several days

# This vector includes the highest temperatures recorded on several days
temperature <- c(19, 17, 20, 20, 13, 13, 15, 17)

mean(temperature)  # to compute the mean maximum temperature
```

When writing commands in the script, always put a space after a comma, and never before a comma, just like in regular English.[4]

[4]In addition: never put spaces inside or outside parentheses for regular function calls. Most infix operators (==, +, -, <-, etc.) should always be surrounded by spaces. Use " for quoting text. Only use ' when the text already contains double quotes and no single quotes. If the arguments to a function do not all fit on one line (of max 80 characters), put each argument on its own line and indent with two spaces.

Downloading and activating R Packages

R and RStudio can be used more efficiently, by 'add ons' called *packages*. These packages simplify programming in R as they include new functions. After downloading (= installing) a package, there is no need to do that every time you are using R, although it may be wise to sometimes check for updates. After installing and opening R and RStudio, you can install packages using a command in a script.

```
# This script installs the packages 'foreign' and 'tidyverse':

install.packages("foreign", "tidyverse")
```

Downloading (= installing) a package is not the same as 'calling' that package for usage in R: packages are not automatically opened when you are using R and RStudio. And that is a good thing: packages can use conflicting short cuts (functions with the same name, but with very different outcomes). Therefore, each time you are using a specific *package* you have to load that package into the library (aka 'activate'). A *script* therefore often starts with 'calling' the relevant packages for that script.

```
# Loading the packages into the library is done with commands like these:

library(foreign)
library(tidyverse)
```

R packages to be used at BMS

A lot of statistical analyses can be done by using 'base R' and its associated functions, but for some more specialized analyses we will use specialized packages.

Since the number of packages is huge[5], we have decided to use only a limited number of packages at BMS. The use of R at BMS will be based on the *tidyverse*[6]. We will therefore mainly use packages belonging to the *tidyverse* (that are installed and loaded when installing and loading the *tidyverse* package).[7] We will also use some other packages.

Here is an overview of the packages you will be using at BMS:

- `tidyverse` is a set of packages for the most important data manipulation is visualization tasks. This is a 'meta-package' containing (among others):
  - `ggplot2` to visualize your data;
  - `dplyr` and `tidyr` to change data, and to make sure that each variable is in a column, each observation is a row and each value is a cell;
  - `readr` to import .csv data.

These packages are thus loaded in the library when using the function `library("tidyverse")`.

When installing the tidyverse, some other packages are downloaded too, some of which we will use:

- `readxl` to import .xls and .xlsx sheets;
- `haven` to import SPSS, Stata, and SAS data;
- `broom` to use a `tidy()` function, which may come handy when doing statistical analysis. This package is also part of `tidymodels`;
- `modelr` to make plots with the residuals and/or the predicted values (this package is also part of `tidymodels`).

Make sure to explicitly put such a tidyverse related package in the library if you want to use it. Unlike the core packages, these packages are NOT automatically loaded when using the library("tidyverse") command.

Finally, we will use a set of specialized packages, including:

[5] click here to see all available packages

[6] click here to see the tidyverse style guide

[7] What is a bit confusing, is that the tidyverse is both a *set* of packages, and a 'programming philosophy'. There are many more packages adhering to that tidyverse philosophy.

- `foreign` to import SPSS, Stata, and SAS data;
- `janitor` to create frequency tables and cross tabulations;
- `psych` for psychometric analysis, including scale construction and factor analysis;
- `lmerTest` to approximate *p*-values;
- `lmtest` for various diagnostic tests in the context of linear models (Levene's test for example);
- `lme4` for the linear mixed model;
- `car` for some additional diagnostic tests in the context of linear model (including the vif value);
- `CTT` for classical test theory (psychometrics)
- `Lambda4` for Collection of Internal Consistency Reliability Coefficients (psychometrics)
- `mirt` for Item Response Theory (psychometrics).

At the beginning of a meeting or course we will tell you which packages to download (`install.packages()`) and call (`library()`).

Datasets in R-packages

Apart from a set of commands, most packages also contain data sets, to enable easy illustrations of what packages can do.

```
# to see which datasets are available in the packages loaded in the library,
# use:

data()

# to see all 'available' data in installed packages on your computer, use:

data(package = .packages(all.available = TRUE))

# sometimes it makes sense to put a data set in the Global Environment:

gss_cat <- gss_cat  # loaded in the package forcats, which is part of the core tidyverse
```

The 'pipe' operator

Some packages add additional operators to the set of base R operators. The 'pipe' operator (`%>%`) is the most important one. We urge you to use the 'pipe' function as much as possible.

Suppose we wanted to create a new object with some gss_cat data from 2000 after we imported these data as an object into R. In this data set each row is an individual in a specific year. Religion and partyid are variables in this dataset. Suppose we want to select *only* **data** from 2000, and focus *only* on age and religion, and we want to store these data in a smaller object called 'gss_cat_small'. We **could** use:

```
gss_cat <- gss_cat  # storing the data set in the global environment
gss_cat_small <- filter(select(gss_cat, year, age, relig), year == 2000)  # filtering variables and sel
```

This is called a *nested command* (with the functions 'filter' and 'select'). A nested command is often difficult to understand. Much simpler is:

```
# using a 'pipe' to simplify commands
gss_cat_small <- gss_cat %>%
  select(gss_cat, year, age, relig) %>%
  filter(year == 2000)
```

This second command will do the same thing and means: you have a big data frame (gss_cat, which is an easily called object in the package), you select some variables from this data frame, and then you filter those cases (from a specific year).

When writing code, use the pipe `%>%` operator as much as possible. There is special hotkey in RStudio for the pipe operator: Ctrl+Shift+M (Windows & Linux), Cmd+Shift+M (Mac). When indenting after a pipe >%> operator, you need to use two spaces (you can automatically set this in RStudio's preferences (RStudio / Preferences / Code/ Insert spaces for tab / tab width = 2).

This document will be made available as a reference for what we expect you to know in various phases of your study program. The document will be updated regularly. *We want to stick to about 20 to 25 pages.* Suggestions for improvement are welcome.

Please note: some commands below work without additional packages (in 'base R'), however throughout the remainder of this document we assume you have installed the (core) tidyverse packages by using `library("tidyverse")`. For additional commands we will tell which R-packages you need.

2. Handling data files

Creating data frames

Creating your own data set can be done in many ways. We mention two:

(1) typing in data:

```
#creating a data frame by variable

age <- c(44, 30, 20, 67) %>% as.integer()
gender <- c("male", "female", "male", "female") %>% as.factor()
length <- c(1.67, 1.70, 1.80, 1.81) %>% as.numeric()
membership <- c("T", "F", "T", "T") %>% as.logical()

# merge the variables
dataset_2 <- data.frame(age, gender, length, membership)
```

(2) generating random data: If you do not have actual data, but want to show something in R, you can randomly select observations from a distribution.

```
# creating a data frame with a random variable based on the normal
# distribution, assumed mean of the normal distribution is 50, sd is 20, with
# 100 units.

n <- 100
dataset_3 <- rnorm(n, 50, 20) %>%
    as.data.frame()
```

Importing .csv, .sav and .spss files as data frames: read

To import a data set, you first need to activate the correct package. These pacakges are `readr` (for csv files, although this is part of the tidyverse core and is thus installed when using `library("tidyverse")`), `foreign` (for SPSS files), and `readxl` (for excel files, also part of the tidyverse, but not of the core, so it needs to be called separately). Next to that, you need to know the location of the data set on your computer.

Suppose you have two version of a data file called 'dataset_4' both located in "C:/My Documents". The data sets can be loaded into the *global environment* with the following commands:

```
# set working directory to the correct folder and put all files to be imported
# in that folder

install.packages("tidyverse", "foreign")  # you do this only once every few months or so
library("tidyverse", "foreign", "readxl")  # you need to do that every time after (re)starting R

setwd("C:/My Documents")  # setting the working directory

# load a csv data file, in this case separated by comma's, with the package
# 'readr'
data_1 <- read.csv("dataset_4.csv", sep = ",")

# load an spss file with the packave foreign
data_2 <- read.spss("dataset_4.sav", to.data.frame = TRUE)
```

Working with labelled data (mainly SPSS and SAV files): attr

Sometimes imported datasets are 'labeled'. For example, the variable 'gender' is stored as a series of 1's and 2's and a label is added for the variable ('gender as derived from the sampling frame') and for the values (1 means 'woman' and 2 means 'man'). The labels are attributes of a data frame.

To inspect the labels in a dataframe:

```
# finding the variable label
data$variable %>%
    attr("label")

# finding the value labels
data$variable %>%
    attr("labels")
```

View data frames: view, str and colnames

To illustrate the commands used below, we will use the data set *gss_cat* (from the tidyverse package forcats). This is a sample of data from the general social survey in the US. It contains the variables: year (year of survey, 2000–2014); age (Maximum age truncated to 89); marital (marital status); race (race); rincome (reported income); partyid (party affiliation); relig (religion); denom (denomination); tvhours (hours per day watching tv).

Suppose we have a data frame and you want to inspect it.

```
# View the data from the data set dataset4 in a spreadsheet.
gss_cat %>%
    View()  # mind the capital V

# Get a quick overview of the types of data in your matrix and their names
gss_cat %>%
    str()

# Only get the column names of a data set
gss_cat %>%
    colnames()
```

Subsets of a data frame: select, filter and the assignment operator <-

```
# Viewing a subset of the data:
gss_cat %>%
    select(marital, age, race) %>%
    View()

gss_cat_2000 <- gss_cat %>%
    filter(year == 2000) %>%
    View()
```

Selected variables can also be stored as a separate object in the global environment.

```
# Make a new data frame with only the items (variables) that you want:
gss_cat_social <- gss_cat %>%
    select(marital, age, race)

# Using column number to select variables:
gss_cat_social <- gss_cat %>%
```

```
    select(1:5, 7, 8)

# Make a new data frame with only observations from 2000:
gss_cat_2000 <- gss_cat %>%
    filter(year == 2000)

# Make a new data frame with observations from 2002 and later:
gss_cat_2002plus <- gss_cat %>%
    filter(year >= 2002)
```

Adding a new variable to an existing data frame: mutate

Adding a variable to the data set can be done with mutate().

```
# Adding a standardized variable (a z-score) to a data frame
gss_cat <- gss_cat %>%
    mutate(ztvhours = (tvhours - mean(tvhours))/sd(tvhours))

# Or use the function `scale()` to get the same standardized variable
gss_cat <- gss_cat %>%
    mutate(ztvhours2 = scale(tvhours))


# Creating an index and adding that index to the data frame
dataset5 <- dataset5 %>%
    mutate(index = item3 + item4 + item4)

# Creating a logged version of an existing variable (the index in this example)
dataset5 <- dataset5 %>%
    mutate(log_version_index = log(index))

# If you want to sum values of a lot of columns and ignore the missings, use
dataset5 <- dataset5 %>%
    mutate(sum = rowSums(.[c(1:4, 7:20, 22)], na.rm = TRUE))
# The 'c' in this command stands for 'combine' and is part of base R.
```

Renaming a variable in a data frame: rename

```
# Change the old name 'relig' to the new variable name 'religion' (new = old)

gss_cat <- gss_cat %>%
  rename(religion = relig)
```

Dealing with missing data: na_if, is.na, na.rm, na.omit

R uses only one way of declaring a specific observation as missing: NA (Not Available). If your data set includes missings, but the missings are coded with a number (for example: 99), you need to replace these values before analyzing the data.

```
# To change other values (here: the word 'Don't know)' to NA, use na_if()

gss_cat <- gss_cat %>%
  mutate(relig = na_if(relig, "Don't know"))
# This means: in the data set gss_cat, change the existing variable relig (which contains contains case
```

There are several ways to find out how many missings are included in the data. Also, there are multiple ways to deal with these missings (pairwise deletion versus listwise deletion).

```
# To see which variables in the data file called 'mydata' have missings use:
summary(gss_cat)

# Or use 'is.na' to detect the number of missing values in a specific column
# (the 'religion' variable).
gss_cat$relig %>%
    is.na() %>%
    sum()

# Pairwise deletion of cases (exclude cases that have a missing value on a
# variable, but keep them when working with other variables): 'na.rm'
mean(gss_cat$tvhours, na.rm = TRUE)

# Listwise deletion of cases (drop cases that have a missing value on any of
# the variables used): 'na.omit'
gss_cat_no_missings <- na.omit(gss_cat)

# Make a new data frame only containing units without a missing value on the
# variable 'relig'. Please note that you now drop many cases ONLY because this
# variable is missing.
gss_cat_no_missings_on_relig <- gss_cat %>%
    filter(!is.na(relig))

# Please note that in R we can use the '!' sign, to say 'not'.
```

Changing the type of variable: from character to factor and from factor to ordered factor

If you have a character variable with the 'words' 'man' and 'women' you probably want to treat this variable as a factor, not merely as a column of words. And if the factor in your data frame has three 'values'/'attributes' (low, medium and high), you have to make sure this variable is stored as an 'ordered factor'.

```
# because gss_cat only has numerical variables and factors, we first change a
# factor to a character variable
gss_cat <- gss_cat %>%
    mutate(marital_char = as.character(marital))

# changing text back to factor
gss_cat <- gss_cat %>%
    mutate(marital_factor = as.factor(marital_char))

# changing factor to ordered factor
gss_cat <- gss_cat %>%
    mutate(marital_ord = factor(marital_factor, order = TRUE, levels = c("Never married",
        "Married", "Divorced", "Separated", "Widowed", "No answer")))
# In this example, we have put the brackets in a way that avoids omitting a
# bracket (which happens a LOT!)
```

Adding a new variable from another datafile to an existing data frame: left_join

Suppose you have a data file with a large number of countries over a large number of years (every 'country x year' is one observation). You want to add the continents of these countries to the data set. You have another data set with the countries (stored as the same words as in the other data set) and the continents.

```
# gapm1945to2020 is the original dataset gapmcountries is the set with
# countries and continents Both datasets are loaded in the global environment
# country in both data sets is called 'geo' and both have the same values (the
# same words)

gapm1945joined <- gapm1945to2020 %>%
    left_join(gapmcountries, by = "geo")
```

Adding observations to a data file: add_row

Sometimes, you want to add data to an existing data file. For this, you can use the function `add_row()`. With `.before` and `.after`, you can specify where new cases should be added.

```
# to use the add_row() function, the tidyverse packages should be installed and loaded
# all variable names should be included in the command

# data will be added after the last case
dataset1 <- dataset1 %>%
   add_row(., Variable1 = 202, Variable2 = 3, Variable3 = 1)

# Note the  very confusing ".,", which is sometimes used when combining base R commands with tidyverse

# You can also specify where to add the new case (for example: before case 51)
dataset1 <- dataset1 %>%
   add_row(., Variable1 = 202, Variable2 = 3, Variable3 = 1, .before = 51)
```

Recoding and changing variables in an existing data frame: recode and case_when

Recoding single values of a variable into different values in one new variable

Sometimes you want to change the values of a variable. Usually it is best to simply make a new variable using *mutate* (see above). Let us say you have items **x10** and **x11** of type integer (meaning, only the numbers -1, 0, 1, 2 .... etc.) that are scored from 1 to 3 and you'd like to change number 3 into integers 1, and number 1 into integer 3 (reverse coding, 2 will stay 2). We then make new variables of type integer **x10_R** and **x11_R** in the following way. Note that if you use the **L**, your new data type will be *integer* (which saves memory). You leave them out if you want the data type to be *numeric*.

```
# keep in mind R uses in most cases (but NOT with rename, which is very
# confusing) the 'OLD is now NEW' order of values.

df_psychology <- df_psychology %>%
    mutate(x10_R = recode(x10, `1` = 3L, `2` = 2L, `3` = 1L), x11_R = recode(x11,
        `1` = 3L, `2` = 2L, `3` = 1L))

# it is often simpler to temporarily ignore the fact a variable is an integer,
# and to simply add the 'as.integer()' command later.

# It is also possible to recode character/factor variables.
```

```
data <- data %>%
    mutate(var1b = recode(var1a, word = "newword"), var2b = recode(var1b, word2 = "anothernewword"))
```

The code above does not seem to work for all data files. When you for example imported an SPSS data file with labels, another method should be used.

```
# for variable x3: 1 -> 0, 2 -> 1
# use: value - 1

data_new <- data_new %>%
  mutate(x3_R = x3 -1)

# for variable x1 and x2: 1 -> 3, 2 -> 2, 3 -> 1
# use: value * (-1) + 4
data_new <- data_new %>%
  mutate(
    x1_R = x1 * (-1) + 4,
    x2_R = x2 * (-1) + 4
  )

# again, make sure the data are now stored as integers.
```

Recoding a range of values of a variable into different values in a new variable: case_when

When you would like to recode an entire range of values of a variable into the same different value in a different variable, `case_when()` can be used.

```
# recode values of the variable "old_var":
# lower than 20 into 1, 20 - 39 into 2, 40 or higher into 3

dataset1 <- dataset1 %>%
      mutate(new_var = case_when(
        old_var < 20 ~ 1,
          old_var >= 20 & old_var < 40 ~ 2,
        old_var >= 40 ~ 3
  )
)
```

Creating dummy variables

A simple way to create three dummy variables (dummy1 etc..) out of one nominal variable with three values is:

```
data$dummy1 <- ifelse(data$nom == 1, 1, 0)
data$dummy2 <- ifelse(data$nom == 2, 1, 0)
data$dummy3 <- ifelse(data$nom == 3, 1, 0)

# when the nominal variable is stored as words use:
data$england <- ifelse(data$country == "england", 1, 0)
```

This can be used for nominal variables with more values too.

Creating longer or broader data frames using pivot_longer and pivot_wider

Sometimes data are stored in a relatively wide format (a lot of variables). For example, all the countries are rows and there is a variable 'unemployment_2000' and a variable 'unemployment_2001' etc... In order to

change wide format data into long format data, in which there are three variables only: country_name, year, level of unemployment, use `pivot_longer()`.

```
# To create 1 new variable containing the names of 4 variables (Sepal.Length,
# Sepal.Width, Petal.Length, Petal.Width) and 1 variable with the scores:

iris %>%
    pivot_longer(cols = c(Sepal.Length, Sepal.Width, Petal.Length, Petal.Width),
        names_to = "variable", values_to = "score", values_drop_na = TRUE)

# If you want to restructure variables beginning with the same name (for
# example variables for each week, starting with 'wk'), you can use
# starts_with()

billboardlong %>%
    pivot_longer(cols = starts_with("wk"), names_to = "week", values_to = "rank",
        values_drop_na = TRUE)
```

And sometimes you want to go from long format to wide format. Use `pivot_wider()`.

```
#using the same example
billboardwide %>%
  pivot_wider(names_from = week,
              values_from = rank)
```

3. Univariate data analysis and data visualization

A variable can be described with statistics like the mode, mean, median, variance and standard deviation. Also, it is interesting to visualize a variable using a plot.

Summarizing numerical variables: mean, median, variation and standard deviation

Below you will see several examples of obtaining descriptive statistics for items in a data set called `data_new`.

```
# To summarise some main characteristics of one numerical item (called Item1):
data_new %>%
    summarise(mean = mean(Item1), sd = sd(Item1), var = var(Item1), minimum = min(Item1),
        maximum = max(Item1))
# You can also ask for only one of the statistics.

# To summarize the main characteristics (mean, median, minimum, maximum, Q1, Q3
# and number of missings) of multiple variables (in this example: the first 10
# variables) in the data set:
data_new %>%
    select(1:10) %>%
    summary()

# To compute a certain statistic for the first 10 variables in the data set,
# you can also use the map() function:

data_new %>%
    select(1:10) %>%
    map(var)

# instead of var, you can also use mean, sd, or other functions
```

Frequency tables: tabyl

The tabyl() function from the `janitor` package allows you to create frequency tables.

```
# Make sure 'janitor' in loaded into the library

# Create one frequency table
gss_cat %>%
    tabyl(race)

# Create frequency tables for all variables of a data set:
gss_cat %>%
    map(tabyl)

# If the values are in alphabetical, rather than in a meaningful order, make
# sure the variable is stored as an 'ordered factor' (see chapter 2).

# The frequency table can be made nicer using the adorn commands, for example:
gss_cat %>%
    tabyl(marital) %>%
    adorn_totals("row") %>%
    adorn_pct_formatting()
```

Univariate graphs: bar charts, box plots and histograms

We will use the package `ggplot2` (part of the core tidyverse) for creating visualizations. The function `ggplot()` is extremely flexible. Below we present some basics only.

The basic idea of ggplot commands is that you
(1) have a data frame, pipe (`%>%`) that into a
(2) ggplot(), add a
(3) geom_...() to select the type of display you want to have (a bar chart, histogram or a scatterplot), and
(4) use aesthetics (aes()) to select the variable(s) you will use from that data frame.

There are more 'layers' in a ggplot, but these are the most important.

```
# creating a bar chart using the dataset gss_cat with the nominal variable marital
gss_cat %>%
  ggplot(aes(x = marital)) +
  geom_bar()

# to create a box plot for the ratio variable tvhours in the dataset gss_cat
# (please note that x = and y = in the aesthetics tilt the picture)
gss_cat %>%
  ggplot(aes(y = tvhours)) +
  geom_boxplot()

# to create a histogram for the ratio variable tvhours in gss_cat
gss_cat %>%
  ggplot(aes(x = tvhours)) +
  geom_histogram()

# The aesthetics are often put in ggplot command itself. This is efficient if you combine different geo
gss_cat %>%
  ggplot() +
  geom_histogram(aes(x = tvhours))

## when creating QQ plots to assess deviations from normality use:
gss_cat %>%
  ggplot(aes(sample = tvhours)) +
  geom_qq() +
  geom_qq_line()
```

Univariate inferential statistics

Confidence intervals and tests for proportions, medians or means of one variable can be constructed for dichotomies (proportions), nominal variables (goodness of fit) and interval/ratio variables (means testing).

```
# For a proportion, use
n_total <- 1000 # total number of (valid) cases
n_pos <- 690 # number of positives
binom.test(n_pos, n_total, alternative = "two.sided", conf.level = 0.95)
# for confidence interval and two sided test

# For a goodness of fit test, use
n_observed <- c(166, 142, 80, 160)
p_expected <- c(0.25, 0.25, 0.25, 0.25) # these have to add to 1 (proportions)
g_o_f <- chisq.test(n_observed, p = p_expected)
g_o_f
```

```r
# For a test of the median, use
# assuming the data contains a vector with a difference, called data$t2_t1
wilcox.test(data$t2_t1, alternative = "two.sided")
# this can be done using the idea of a signed rank and the linear model too.
# assuming you constructed the signed ranks yourself, use ...
data %>% lm(signed_rank_t2_t1 ~ 1, .)

# For means testing use either t.test or the lm command
# assuming the data are stored in a data frame as data$var
mu_1 <- 100 # if you want to check whether it is different from 100
t.test(data$var, mu = mu_1)

# or in the linear model framework:
data %>%
  lm((var - mu_1) ~ 1, .) %>%
  summary
```

Sometimes you calculate a t-statistic or a chi-square statistic by hand and you want to find the associated p-value, OR you want to know which critical (t or chi-square) value is associated with a specific alpha. These commands allow you to not use 'tables'.

```r
# For finding the p-value for a specific t-value
q <- 1.96 # the t-value
df <- 499 # the degrees of freedom
pt(q, df, lower.tail = FALSE)
# NOTE: this gives the percentage on ONE side

# For finding the critical t-value
p <- 0.025 # the p value (half)
df <- 499 # the degrees of freedom
qt(p, df, lower.tail = FALSE)

chi <- 3.84 # the chi square value
df <- 1 # the degrees of freedom
pchisq(chi, df, lower.tail = FALSE)

# finding the critical value in a chi square distribution
p <- 0.05 # the chi square value
df <- 1 # the degrees of freedom
qchisq(p, df, lower.tail = FALSE)
```

Testing whether data are coming from a normally distributed population.

```r
# For the Shapiro-Wilk test use:
shapiro.test(mtcars$residuals)
```

4. Bivariate data analysis and bivariate data vizualisation

Contingency tables (a.k.a. crosstabs): tabyl

Use the `tabyl()` function from the `janitor` package. The first line creates the table. The rest formats the table (giving it a title, adding totals etc. . .).

```
library(janitor)
mtcars %>%
  tabyl(am, gear, show_na = FALSE) %>%
  adorn_title("combined") %>% #both var names in the title
  adorn_totals("col") %>% #column totals
  adorn_totals("row") %>% #row totals
  adorn_percentages("col") %>% #columnwise, rowwise (row), or total percentages (all)
  adorn_pct_formatting(digits = 1) %>%
  adorn_ns() #show the numbers of cases
```

Visualizing relationships between variables: ggplot

For a bivariate visualizations:

```
# A boxplot comparing different species (nominal variable)
iris %>%
  ggplot(aes(x = Species, y = Petal.Width)) +
  geom_boxplot()

# two scale variables, creating a scatterplot
mtcars %>%
  ggplot(aes(x = cyl, y = mpg)) +
  geom_point()

# adding a regression line
mtcars %>%
  ggplot(aes(x = cyl, y = mpg)) +
  geom_point() +
  geom_smooth(method = "lm", se = F)
```

Descriptive and inferential statistics for associations (parametric)

For Pearson's r, we use

```
cor.test(x = mtcars$mpg,
    y = mtcars$disp,
    method = "pearson")
```

For an independent sample t-test (a linear model with a dummy as independent variable), in which both groups can be assumed to have an equal variance, we use:

```
# vs is a dummy and disp is a ratio variable in the data set mtcars
t.test(disp ~ vs, mtcars, var.equal = TRUE)
# or use
lm(disp ~ vs, mtcars) %>% summary()
```

For an independent sample t-test in which both groups cannot be assumed to have equal variances, we use:

```
# vs is a dummy and disp is a ratio variable in the data set mtcars
t.test(disp ~ vs, mtcars, var.equal = FALSE)
```

When studying the effect of nominal variable on a ratio variable (ANOVA), use:

```
# spray is a nominal variable (stored as a factor) and
# count can be treated as a ratio variable
# in the data set InsectSprays

oneway.test(count ~ spray, InsectSprays, var.equal = TRUE)

# or use
lm(count ~ spray, InsectSprays) %>% summary()
```

For a test in which groups cannot be assumed to have equal variances, use:

```
oneway.test(count ~ spray, InsectSprays, var.equal = FALSE)
```

Testing whether variances are the same: Levene's test

Sometimes it may be wise to actually TEST whether two different variances of two (or more) different groups may come from the same population. One of the tests use for this is *Levene's Test*. In R, you use the `leveneTest()` function from the `car` package. You define the dependent variable, here *mpg*, and the independent variable, here *am*.

```
# library(car)
mtcars %>% leveneTest(mpg ~ factor( am ),
                      data = .)
```

When *Levene's Test* is NOT significant, you can assume the groups have equal variances.

Descriptive and inferential statistics for associations (non parametric)

For Cramer's V, we use

```
# install and load the package "rcompanion" first, this contains the command cramerV.
# this function works on tables, so create a table first

library("rcompanion")
my_table <- table(mtcars$vs, mtcars$am)
cramerV(my_table, ci = TRUE)
```

For Kendall's tau b, we use

```
cor.test(x = mtcars$mpg,
    y = mtcars$gear,
    method = "kendall")
```

For Spearman's rho, we use

```
cor.test(x = mtcars$mpg,
    y = mtcars$disp,
    method = "spearman")
```

For a chi-square test, we use

```
# assuming the data are in vector x and vector y2
csq <- chisq.test(data$x, data$y2)
csq

# after calculating chi square by hand,
# you can find the associated p-value by using:
```

```r
chisq <- 3.86 # put the number you find here
df <- 1 # put the degrees of freedom here
pchisq(chisq, df, lower.tail = FALSE) # this gives the p-value
```

For a non-parametric test for comparing the medians of two groups, we use

```r
mtcars %>%
  wilcox.test(mpg ~ am,
              data = .,
              exact = FALSE)
```

For a non-parametric test for comparing the medians of three or more groups, we use

```r
airquality %>%
  kruskal.test(Ozone ~ Month, data = .)
```

5. Multivariate data analysis

3-way contingency tables: tabyl

Use the `tabyl()` function from the `janitor` package. With several adorning functions you can adjust the table to your liking. You can force the table displaying the values 'none', 'some' and 'many' in the correct order, by making sure the variable is stored as an 'ordered factor' (see chapter 2).

```
library(janitor)

mtcars %>%
  tabyl(am, gear, cyl, show_na = FALSE) %>%
  adorn_title("combined") %>% #both var names in the title
  adorn_totals("col") %>% #column totals
  adorn_totals("row") %>% #row totals
  adorn_percentages("col") %>% #columnwise, rowwise (row), or total percentages (all)
  adorn_pct_formatting(digits = 1) %>%
  adorn_ns() #show the numbers of cases
```

Groups in a scatterplot and facetting: ggplot

If you want to visualize different groups in the scatterplot, you can add extra variables to the aesthetics of a ggplot. In addition to x and y, you can use color (the "outside" color of points), fill (the "inside" color of the points), shape (of the points) and size.

```
mtcars %>%
  ggplot(aes(x = cyl,
             y = mpg,
             color = factor(gear)
             )
         ) +
  geom_point()
```

For different subplots, use the layer `facet_wrap(~ )` as default. This is another way to introduce a third variable in the context of data visualizations.

```
mpg %>%
  ggplot(aes(x = factor(cyl), y = hwy)) +
  geom_boxplot() +
  facet_wrap(~ year)
```

Linear models

For the ordinary linear model, we use `lm()`. For a clear presentation of the regression table, we use the `tidy()` function from the `broom` package.

```
library(broom)

model <- mtcars %>%
  lm(qsec ~ wt + cyl, data = .)

# the regression table
model %>%
  tidy()

# the ANOVA table
```

```r
model %>%
  anova() %>%
  tidy()

# overall output, including R squared, is provided by
model %>%
  summary()
```

Diagnostics: residuals, equal variances, outliers and influence

There are various ways to create plots with the residuals and/or the predicted values.

```r
# always create a model you want to diagnose first

model <- mtcars %>%
  lm(qsec ~ wt + cyl, data = .)

# the object 'model' now contains the residuals and the predicted values.
# residuals and predicted values can  be added to the data frame in two ways:

# directly
mtcars$residuals <- model$residuals
mtcars$predictions <- model$fitted.values

# or by using the modelr package:
library(modelr)
mtcars <- mtcars %>%
  add_predictions(model) %>%
  add_residuals(model)

# after adding the residuals and the predicted values displaying residuals plots is done in a ggplot
mtcars %>%
  ggplot(aes(x = pred, y = resid)) +
  geom_point()
```

There are various ways to test for equal variances in a model. These are available in various packages, including `car` and in `lmtest`. Levene's test is introduced in chapter 4.

```r
# Breusch-Pagan Test
library(lmtest)

# estimate the model first and then:
bptest(model)
```

For outliers and influential cases, use:

```r
# After estimating and storing a model, leverage as measured by the hat values can be stored to the ori
data$leverage_cases_model_1 <- hatvalues(model_1)

# Also, one of the plots of the model contains info about Cook's distances
plot(model_1)

# After estimating and storing the model, influence as measured by cooks distances can be stored to the
data$influential_cases_model_1 <- cooks.distance(model_1)
```

```r
# Also, one of the plots of the model contains info about Cook's distances
plot(model_1)
```

After estimating the model, detecting multicollinearity can be done using the `car` library.

```r
library(car)
vif(model)
```

linear mixed models

For the linear mixed model, we use the `lmer()` function from the `lme4` package. Note that the output does not show p-values, nor residual degrees of freedom for fixed effects. This is for a good reason.

```r
library(lme4)

mtcars %>%
  lmer(qsec ~ wt + (1|gear), data = .) %>%
  tidy()

# When we have factors as fixed variables:
mtcars %>%
  lmer(qsec ~ wt + factor(cyl) + (1|gear), data = .) %>%
  anova() %>%
  tidy()
```

If you want approximate p-values, you can use Satterthwaite's degrees of freedom method, implemented in the `lmerTest` package. The same method is used in SPSS.

```r
library(lmerTest)
mtcars %>%
  lmer(qsec ~ wt + (1|gear), data = .) %>%
  summary()
```

For a residual plot, we can use similar syntax as for the linear model, using `modelr`:

```r
model <- mtcars %>%
  lmer(qsec ~ wt + (1|gear), data = .)

mtcars %>%
  add_predictions(model) %>%
  add_residuals(model) %>%
  ggplot(aes(x = pred, y = resid)) +
  geom_point()
```

logistic regression

For a logistic regression model, we use the `glm()` function.

```r
mtcars %>%
  glm(am ~ wt, family = binomial, data = .) %>%
  tidy()
```

Similarly for a Poisson regression.

```r
mtcars %>%
  glm(carb ~ wt, family = poisson, data = .) %>%
  tidy()
```

Nonparametric tests for repeated measures

For a non-parametric test for repeated measures, we use

```
iris %>%
  select(Sepal.Length, Petal.Length) %>%
  as.matrix() %>%
  friedman.test()
```

6. Psychometric analyses

This chapter is about factor analysis, Classical Test Theory and Item Response Theory models in R. For psychometric analyses we use the packages: `psych`, `CTT`, `Lambda4` and `mirt`.

Factor analysis

```
library(psych)

## Before running a factor analyses, explore your data and check the following:

data_new %>%
    KMO()  # Kaiser-Meyer-Olkin measure

data_new %>%
    cortest.bartlett()  # Barlett's sphericity test

data_new %>%
    cor()  # correlation matrix

## Next, determine the number of factors.  For this, you need to run a
## Principal Component Analysis

# eigenvalues for Kiaser's criterion

pca <- data_new %>%
    cor() %>%
    eigen()

eigenvalues <- pca$values

# screeplot based on principal component analysis

# using a ggplot:
tibble(component = 1:length(eigenvalues), eigenvalues) %>%
    ggplot(aes(x = component, y = eigenvalues)) + geom_line() + scale_x_continuous(breaks = 1:length(ei

# OR use:
data_new %>%
    scree(, factors = FALSE)

## to run a factor analysis with two factors:
model_2f <- factanal(data_new, factors = 2, rotation = "varimax")
```

Running the previous code on a real data set, will not show you the results of the factor analysis. You need to type in `model_2f` in the R console to see the results.

Please note that in the output of a factor analysis, a Chi square statistic is shown. This Chi square statistic belongs to a test for the model fit. The null hypothesis for this test is: the model that is used (in the example: a model with two factors) fits. A non-significant result is thus preferable, as we do not reject the null hypothesis then. But, large sample sizes easily give significant results. Because of that, we would like you to ignore this part of the output completely and focus on the interpretation of the factors instead.

Classical Test Theory analyses

```
library(CTT)
library(Lambda4)

# analyze the data
results <- dataset %>%
  as.matrix() %>%
  itemAnalysis()

# show Cronbach's alpha
results$alpha

# show Cronbach's alpha if item removed (deleted),
# p-values (=ItemMean), and item rest correlations (=pBis)
results$itemReport

# Compute lambda 2
dataset %>% guttman() # lambda 3 = Cronbach's alpha
```

Item Response Theory models

```
library(mirt)

# a 1-dimensional model with 2 parameters per item
out2 <- dataset %>%
  mirt(., model = 1, itemtype = "2PL")

out2 # to see Akaike's Information Criterion

# a 1-dimensional model with 1 parameter per item
out1 <- dataset %>%
  mirt(., model = 1, itemtype = "Rasch")

# to extract the estimated parameters of a model:
par2 <- out2 %>%
  coef(, IRTpars=T, simplify=T)

par2$items #to see the parameters

# information plots
out2 %>%
  itemplot(, item = 1, type = "info") #item information plot for item 1

out2 %>%
  plot(, type = "info")               #test information plot

# assessing the item fit
out2 %>%
  itemfit()                           #for all items

out2 %>%
  itemfit(,empirical.plot = 6)        #plot for item 6
```