# Reproducible science with R

Andy P. Field1 and Teresa B. Gibson2

1School of Psychology, University of Sussex

2School of Mathematics and Statistics, Rochester Institute of Technology

# Author Note

Andy P. Field  http://orcid.org/0000-0003-3306-4695

Teresa B. Gibson  http://orcid.org/0000-0002-5853-7447

A github repository of this chapter, including a Quarto document of the example is at <https://github.com/profandyfield/chrmssbs>. The authors have no conflicts of interest to declare.

Correspondence concerning this article should be addressed to Andy P. Field, School of Psychology, University of Sussex, Falmer, Brighton, East Sussex BN1 9QH, Email: andy.field@sussex.ac.uk

# Abstract

This chapter gives a brief overview of R, an open source software environment for statistical computing. We begin by explaining the benfits of using R alongside the RStudio Integrated Development Environment (IDE) and a document production application called Quarto. We provide recommendations about workflow best practice and introduce core concepts within the R eco-system. We then move to an overview of coding by exploring the key concepts relevant to R. Next, we briefly explore how to import data into R. The final section of the chapter brings the fundamental ideas discussed to life by reproducing some reported analyses from a study conducted by Perchtold et al. ([2019](#ref-perchtold2019)). Specifically, we walk through how to produce a table of correlation coefficients and fit a linear model (regression). By the end of this chapter, we hope to have given you a taste of the fundamental principles of using R as part of a reproducible scientific workflow.

*Keywords*: R, RStudio, Quarto, statistical programming, statistical analysis

# Reproducible science with R

R ([R Core Team, 2024](#ref-rcoreteam2024)) is a powerful open source software environment for statistical computing and graphics released in 2000 (after 8 years of development). Much of this strength comes from the fact that the core application (so called ‘base R’) can be extended by installing and loading ‘packages’ developed by individuals and organisations. Most packages are stored on a central repository managed by the R core development team known as CRAN (Comprehensive R Archive Network). However, many authors develop their packages using GitHub making it possible to install packages directly from the GitHub website. At the time of writing this chapter, there are 21,939 packages on CRAN; this demonstrates the vastness of the ecosystem. Whatever you want to do with your data, the chances are that someone has written a package that enables you to do it.

In this chapter we will begin by summarizing the main reasons to use R but also acknowledge some of the struggles that come with learning it. Next, we aim to get you up and running with a quick overview of good workflow practices. In doing so we introduce some of the key concepts in the R ecosystem. R is a code-based so we then introduce some of the fundamental concepts in writing R code before looking at how to import data. We then use open data from Perchtold et al. ([2019](#ref-perchtold2019)) to reproduce some of their published analysis. In doing so, we walk you through the process of fitting some common statistical models and provide tips on presenting information from these models in a report.

## Why use R?

Unlike many popular applications for data processing, visualisation and statistical analysis (e.g., IBM SPSS Statistics, JASP, Jamovi, Minitab), that rely on a point-and-click graphical user interface (GUI), R is script-based. This has some advantages (which we will address later), but also means that R has a relatively steep learning curve for those most comfortable with GUI-based software. Why then, should you climb that educational mountain?

### The pros of R …

For an academic, perhaps the most compelling reason to learn R is that it fits seamlessly with a workflow that promotes open science and reproducibility ([Munafò et al., 2017](#ref-munafò2017)), and these are an increasingly vital part of the scientific process. Point-and-click software for data processing and statistical modelling are inherently not reproducible: there is no obvious record of what has been done and mistakes are not traceable ([Simons & Holmes, 2019](#ref-simons2019)). This is a simplification because, for example, IBM SPSS Statistics stores a journal of activity using a syntax language, yet linking a particular part of that file to a particular analysis is, at best, tricky. More recently IBM SPSS Statistics has introduced a workbook file that translates point and click activity into syntax that is stored along with output. Although this vastly improves reproducibility it is still not possible to produce integrated, formatted reports.

As we shall see when we discuss workflow, using R makes it possible to create academic papers and reports that embed the analysis code — meaning that anyone can reproduce an entire academic paper with a click of a button. R integrates seamlessly with GitHub, which in turn integrates with popular open science repositories such as the Open Science Framework (https://osf.io/), making it convenient to create open, reproducible scientific projects.

A second reason for investing time in R is that by being open source, it has a wider range of tools than most proprietary software and keeps pace with the rapidly changing world of data science and statistics. For example, structural equation models typically require specialist proprietary software such as AMOS or MPLus (see Chapter 3), whereas these models are handled within R by importing packages such as lavaan ([Rosseel, 2012](#ref-lavaan)). Also, there are examples of widely-used statistical methods that are not readily available in proprietary statistics software such as robust estimation, Bayesian estimation, and network analysis. Other applications with point-and-click interfaces (e.g. JASP, jamovi) can fit these kinds of models but use R for the computation. As a result, the statistical models available in these point-and-click open source applications depends upon the developers’ choices about what to implement. If you learn R, you make those decisions.

Finally, for those with secret desires to work outside the academy, R is also a useful and transferable skill. According to a 2020 survey of 579 analytic professionals from 71 countries, R and Python (another programming language) are by far the most widely used statistical programs. Around a quarter of consultants and corporate analysts list R as their primary tool (joint highest with Python), in academia it is 41% (compared to 10% for IBM SPSS Statistics) and for non-profits it is 48% ([Rexer Analytics, 2020](#ref-rexeranalytics2020)). Although the estimate for academia is likely inflated because R-users would more likely be aware of and respond to this sort of survey, the point remains that R is a widely used both inside and outside of academia.

### … and the cons

Every tool has it strengths and weaknesses. As we have just discussed, the great strengths of R are (1) the ability to write reproducible documents that embed statistical analyses; (2) the ability to implement a wide-range of traditional and state-of-the-art statistical procedures; and (3) the open source nature of R means that the skills acquired are transferable to any new working context. We would also argue that using R requires a bit more thought about the statistical models you fit, which is no bad thing.

The main disadvantage compared to point-and-click statistics applications is ease of use. There is a time and emotional investment in learning R that is several magnitudes larger than that of point-and-click applications. The diversity of tools available in R can be overwhelming and there are often multiple ways to achieve the same goal—finding relevant information can be difficult in the beginning. Furthermore, the ecosystem changes rapidly ([Staples, 2023](#ref-staples2023)) meaning that the learning never really stops.

## Using R

### Workflow

R exists as a standalone application into which you type commands at a prompt. For example, if you want to fit a linear model, you type a command that does this and then execute it. However, the interface is sparse and unpleasant. Using R from within the RStudio ([RStudio Team, 2016](#ref-rstudioteam2016)) integrated development environment (IDE) is a much richer experience. The RStudio IDE has several panes (and tabs within them) that help you to keep track of your work. These include panes for file and package management, a pane for writing documents, a pane containing information about your working environment, and one that runs R in its basic form (that is, as a console with a prompt for entering commands). In terms of reproducibility and open science, RStudio also has tools for managing repositories on GitHub. Setting up a project repository on GitHub is a useful way to track project changes and updates using version control and also integrates seamlessly with the Open Science Framework (https://osf.io/). RStudio also integrates with a powerful document production application called Quarto ([Allaire & Dervieux, 2024](#ref-quarto)). Therefore, before you start your R journey, we recommend that you install these three applications:

* R: <https://cran.rstudio.com/>
* RStudio: <https://posit.co/download/rstudio-desktop/>
* Quarto: <https://quarto.org/>

Together these applications allow you to use R to create reproducible scientific and analytic documents and reports. From a user perspective you only ever interact with RStudio because R and Quarto work behind the scenes. [Figure 1](#fig-rrstudioquarto) shows how these three applications interact. In short, you start-up RStudio, navigate the menus to create a new document (*File > New File > Quarto Document …*) and start writing and analyzing data. The document you create is a special kind of document known as a Quarto document (because it will be processed by Quarto). A Quarto document is similar those that you have created in your favorite word processor in that you can type and format text, insert citations (using Zotero), create headings and subheadings and so on. The big difference is that you can also insert so-called *code chunks*. Code chunks are a window to R and should contain any code that you want R to execute. Any code chunk can be executed from within your Quarto document allowing you to work on and check code as you write.

[Figure 1](#fig-rrstudioquarto) shows a Quarto document on the left-hand side. Notice that there is a heading followed by a paragraph of text; under that is a grey box containing some code. Let’s imagine you are creating a report or journal article. You might write (and style) an introduction, methods section and so on. Then, as part of your results section you might use code chunks to import a data file, process the data, fit statistical models, and generate plots and tables of results. To create a finished report, the document must be processed (or *rendered*) by clicking a button. When a Quarto document is rendered the text is processed by Quarto to apply formatting and generate headings, hyperlinks and citations; anything in a code chunk is processed by R. In [Figure 1](#fig-rrstudioquarto) (right hand side) note that, after rendering, the textual paragraph has some hyperlinks and formatting applied by Quarto, and the code chunk has been processed by R to create a table (and the subsequent code chunk creates a plot).

Quarto documents can be rendered into an html file (the most convenient) but for scientific manuscripts you can also render to a pdf (but you need to have LaTeX installed) or Microsoft Word file. There are templates available including one that applies APA style (<https://github.com/wjschne/apaquarto>). Quarto is incredibly powerful. As well as producing documents it can be used to produce websites (without knowing code), presentation slides, books, and interactive dashboards. Once you have the expertise to use these applications, there is very little reason to leave the ecosystem; that’s how powerful it is. It is impossible to provide a step-by-step guide to producing a Quarto document, but the first author has some video guides on YouTube (<http://bit.ly/3WODDkm>) and there is more detail in Field ([in press](#ref-fielddsr22026)).

### Project files

With GUI based applications data and output files are opened and saved manually by the user using dialogue boxes. Part of having a reproducible workflow is automating this process so that files can be loaded without user-intervention. To help with this, the single most useful thing you can do when working with R is to use an RStudio project and organize it sensibly ([Field, in press](#ref-fielddsr22026)). An RStudio project is a folder containing a special file with the extension .Rproj that stores information about the containing folder and your environment. Most importantly, it sets the working directory to be the folder containing the project file; this means that accessing the files becomes a lot more straightforward than if you don’t use an RStudio project. Provided you have stored the files you need for your project within your project folder, you will be able to access it using only its file name (because R will automatically look for the file in your project folder). In terms of organizing the project folder, Field ([in press](#ref-fielddsr22026)) recommends at a minimum having a folder called data for any data files and one called quarto or docs for your Quarto documents. Different projects will have different needs though, so you may need other folders too. Space prevents a detailed explanation of how to set up a project, but there is a video walk through on YouTube (<http://bit.ly/3WODDkm>) and there is more detail in Field ([in press](#ref-fielddsr22026)).

### Packages and functions

Some functions are built into R, but others will be bundled into a package (see earlier). We install packages by executing

install.packages("name\_of\_package")

replacing “name\_of\_package” with the name of the package. This should be done outside of your Quarto document. Installing a package puts a copy of it on your computer so you only need to do it once or until you update R. However, to use a package you have to load it by including

library(name\_of\_package)

in a code chunk (we recommend your first code chunk) within your Quarto document (again replacing “name\_of\_package” with the name of the package). For example, if we want to use the package here ([Müller, 2020](#ref-here)) we’d *go to the R console* within RStudio and execute

install.packages("here")

and to use it *within a Quarto document* we’d place

library(here)

in the first code chunk. (Note that install.packages requires the package name to be in quotes but library does not.) We can now use all of the functions within the parameters package.

### The Tidyverse

The Tidyverse ([Wickham et al., 2019](#ref-tidyverse)) is suite of packages containing a wide array of tools for data science that have been developed using a common underlying philosophy and grammar. It contains the following core packages:

* ggplot2 ([Wickham, 2016](#ref-ggplot2)): a powerful system for plotting data.
* dplyr ([Wickham, François, et al., 2023](#ref-dplyr)): tools for data wrangling.
* tidyr ([Wickham, Vaughan, et al., 2024](#ref-tidyr)): tools for tidying data.
* readr ([Wickham, Hester, et al., 2024](#ref-readr)): tools for importing data.
* purrr ([Wickham & Henry, 2023](#ref-purrr)): tools for functional programming.
* tibble ([Müller & Wickham, 2023](#ref-tibble)): tools for working with data frames.
* stringr ([Wickham, 2023b](#ref-stringr)): tools for working with strings/text.
* forcats ([Wickham, 2023a](#ref-forcats)): tools for working with categorical variables.
* lubridate ([Grolemund & Wickham, 2011](#ref-lubridate)): tools for working with times and dates.

In addition, it contains some smaller packages with more specialized functions, notably readxl ([Wickham & Bryan, 2023](#ref-readxl)) for importing Microsoft Excel files, googlesheets4 ([Bryan, 2023](#ref-googlesheets4)) and googledrive ([D’Agostino McGowan & Bryan, 2023](#ref-googledrive))for working with Google sheets, and haven ([Wickham, Miller, et al., 2023](#ref-haven)) for importing data files from IBM SPSS Statistics, Stata and SAS.

To install all of these packages in one command we would *go to the R console* within RStudio and execute

install.packages("tidyverse")

Having done this, we can either load the ones we want to use individually within our Quarto document, for example

library(dplyr)

or load all of the core packages using

library(tidyverse)

Non-core packages have to be loaded individually. For example, if we want to use readxl then we have to execute

library(readxl)

even if we have already loaded the tidyverse.

### The here() function

The here package is incredibly useful for importing data and more generally locating files within your project. The function here(), from within the package of the same name ([Müller, 2020](#ref-here)) returns the file path of the project folder and will, therefore, make your work reproducible and portable.

Let’s imagine you have a data file called data.csv (imaginative name), stored in a folder called data that is in a folder called my\_project. Imagine that this folder is on your hard drive in your main Documents folder, and that you use Windows. To enable R to access this file you would need to type the following file path (assuming you’ve stolen my name):

C:/Users/andyfield/Documents/my\_project/data/data.csv

If your colleague tries to run your code, it will fail because they will be using their computer, not yours, and the data will be stored somewhere else. If you change computer, your code will fail unless you set it up exactly like your old one. However, by using a project folder we can access the data file with

here("data/data.csv")

This code says “look for a folder called data within my project, and within that look for a file called data.csv”. This code will work on any computer because here() finds the location of the current project, wherever it may be (e.g., “C:/Users/andyfield/Documents/my\_project/”), and then adds the text within the function to the end of it. Wherever you have saved you project folder, here() will find it.

### Housekeeping

Although you can load packages within any code chunk of your Quarto document, we suggest loading them all in your first code chunk. Doing so allows you to easily keep track of the packages that the document depends upon and ensures that all the packages you need are loaded at the start of the rendering process. We’d also recommend loading them in alphabetic order because it’s easier to track which packages you use and avoids the redundancy of accidentally loading packages twice.

To follow our example analysis you will need to install the packages listed below, and then you’re first code chunk should load them as follows:

library(correlation)  
library(here)  
library(knitr)  
library(parameters)  
library(performance)  
library(readxl)  
library(tidyverse)

## Code fundamentals

Now you know how to get started, this section moves on to look at some core concepts when coding R. Let’s assume you have a Quarto document up and running and perhaps you have written some text to remind yourself about what you are about to do and why. Sooner or later you will want to do something using R. To do so you need to insert a code chunk by simultaneously pressing the *Ctrl*, *Alt* and *i* keys (Windows and Linux) or *Cmd*, *Option* and *i* on MacOS. This creates an area with a darker background that will have {r} at the top ([Figure 2](#fig-codechunk)). Anything typed within this area will be processed by R when the document renders. You can put as much code as you want in a single code chunk. At the extreme, you could have a single code chunk in your Quarto document that does everything. However, it is better to break an analysis up into separate code chunks that relate to specific parts of your analysis and to prepare tables and plots in self-contained code chunks.

You can process code while working on a Quarto document; this is known as *executing* or *running* the code. Running code as you write it is useful for seeing the results that you want to report and interpret without needing to render the entire document. Note, in [Figure 2](#fig-codechunk), that code chunks have three icons. From left-to-right, the first opens a dialogue box to set options for the code chunk, the second executes all of the previous code chunks, and the last executes all of the code in the code chunk. The results of the code will be displayed below the code chunk in Quarto. You can also execute specific parts of your code by selecting it and pressing the *Ctrl* (*Cmd* on MacOS) and return keys of your keyboard.

### Creating objects

Most commands in R have a common structure that can be summed up as:

object <- instructions

The term on the left is something you wish to create that you have decided to name ‘object’ (but you could have named it something else); on the right will be some instructions about how to create it. In the middle is an arrow an arrow (<-) known as the *assignment operator*, so called because it assigns what’s on the right to the name on the left. You can create many different types of ‘object’; some examples are a table of data, a table of results, a statistical model, output from a statistical model, or a plot. For example, if we want to fit a linear model (regression) we can do this using code that has this general structure:

my\_lm <- lm()

The left hand side tells R to store the object we create using the name ‘my\_lm’, and the right hand side tells R to use a function called lm() (the lm stands for linear model) to create it. A *function* is a bit of code that someone has written (and that you can write yourself) that takes the form name\_of\_function(). Some examples of functions are lm(), mutate(), and ggplot(). Within the brackets you specify pre-defined ‘options’, known as arguments, that vary across functions but essentially tell the function what you want it to do. For example, the lm() function, as a bare minimum, needs two arguments to be set. One is labelled ‘formula’ and the other ‘data’, so we could include these arguments in our code above:

my\_lm <- lm(formula, data)

Arguments are always separated by commas and require you to specify a value using =. It is worth noting, however, that some arguments have default values, and you don’t need to include them unless you want to override that default. The lm() function needs a formula specifying the linear model that you want; this takes the form y ~ x for one predictor and y ~ x1 + x2 + ... xn for multiple predictors. In each formula the ‘y’ and all the terms starting with ‘x’ need to be replaced with the names of variables within your data object. You also need to tell the function where to find the data using the data argument. For example, if you wanted to predict a variable called ‘happiness’ from another variable called ‘relationship\_satisfaction’, and these variables were stored in an object called ‘happy\_data’, then you could fit the model as follows:

happy\_lm <- lm(formula = happiness ~ relationship\_satisfaction, data = happy\_data)

Executing this code will fit an OLS linear model that predicts happiness from relationship satisfaction and stores it in an object called happy\_lm. We can extract from that object information such as model parameter estimates, fit statistics, residuals and so on because, once an object has been created, it is stored in R’s memory. For example, if we have loaded the parameters package within our Quarto document

library(parameters)

we could use the model\_parameters function from that package to extract the table of coefficients from our linear model by executing

model\_parameters(happy\_lm)

### Naming objects

R is case sensitive, so it will treat an object called happy\_lm as being completely different to an object called Happy\_lm. Humans are way more fallible to ignoring the case of letters, so this case sensitivity is a common cause of an R error. There are other conventions about which you need to be aware too. As a brief summary of the style recommended in Field ([in press](#ref-fielddsr22026)):

* Always use lower case letters to name things.
* R treats a space as separating objects so you will get errors if you use spaces in object names; instead use an underscore (e.g. relationship\_satisfaction not relationship satisfaction).
* Be concise and meaningful. So, relationship\_satisfaction is probably sufficient, yet we could truncate it to relation\_satis without losing meaning, scores\_on\_the\_meaningfullness\_of\_my\_relationships\_scale is meaningful but too long, and rs is concise but not meaningful.
* Place spaces around all operators (=, +, -, <-) to make code easier to read.
* Use a suffix (consistently) to indicate what an object contains; for example, you might use \_lm for a linear model or \_mod for a model generally, \_tbl for a table, \_out for the output of a model and so on.

### The pipe operator

Often, when writing code, we want to string several operations together. For example, you might want to filter and then sort your data. Sticking with our relationship satisfaction and happiness data (happy\_data), imagine that we had a variable within the data called duration that recorded the length of the relationship in years. We may want to filter out short-term relationships (i.e., those less than a year). We also want to sort the data by this variable.

We can use the filter() function from the dplyr package to filter the data by loading dplyr in our first code chunk using library(dplyr) and later executing:

long\_happy <- filter(.data = happy\_data, duration > 1)

This creates a new data object called long\_happy. Within the filter() function, we specify the original data object that we want to filter (.data = happy\_data), and we include a logical instruction (duration > 1) that says ‘keep the case if the variable duration is greater than 1’. Remembering that duration is the number of years of the relationship, this will retain all cases where the relationship is more than a year old.

We can now sort this new data object using the arrange() function, also from dplyr.

long\_happy\_sort <- arrange(.data = long\_happy, duration)

This creates a new data object called long\_happy\_sort. Within the arrange() function we specify the filtered data object that we want to sort (.data = long\_happy) and we include the variable by which we want to sort the data (duration).

To sum up, we could achieve the filtering and sorting by executing this code:

long\_happy <- filter(.data = happy\_data, duration > 1)  
long\_happy\_sort <- arrange(.data = long\_happy, duration)

This method is fine, but we’ve created multiple objects, and we’re now storing the same data in two objects(long\_happy and long\_happy\_sort), so it might be more efficient to combine these functions:

long\_happy\_sort <- arrange(.data = filter(.data = happy\_data, duration > 1), duration)

Note that within arrange(), we have replaced long\_happy with filter(.data = happy\_data, duration > 1); instead of creating an object long\_happy and using it, we use the code that filters the file directly. We have nested the filter() function within arrange(). The advantage of this method is that we are not unnecessarily storing objects, but the disadvantage that it is difficult to read and understand.

In the interests of writing clear and easily-readable code, we recommend using the pipe operator (|>) to link functions. As its name suggests, the pipe operator creates a pipeline between functions allowing the output of one function to ‘flow’ into the next. In the current example, we can use the pipe operator as follows:

long\_happy\_sort <- filter(.data = happy\_data, duration > 1) |> arrange(duration)

or more explicitly

long\_happy\_sort <- happy\_data |>  
 filter(duration > 1) |>  
 arrange(duration)

We create long\_happy\_sort by starting the pipeline with the original data object (happy\_data) that ‘flows’ into the filter() function. Note that, within this function, we no longer need to specify the data because filter() will assume it needs to use whatever is coming through the pipe. The filter() function does its job and then sends the resulting filtered data through the next pipe into arrange(), where it is sorted. The advantages of the pipe are that you avoid creating lots of intermediate objects and your code becomes very readable (it is clear that you start with some data, filter it and then sort it).

## Working with data

If you have worked with other statistical or data management software you will be familiar with spreadsheets. These are typically interactive grids of columns and rows where you can type values or text and then save them as a data file. R does not have an interactive spreadsheet interface. You can, of course, use a spreadsheet application and then import the results into R; we will explore this method shortly.

However, if you want to enter data directly into R, you do it by creating variables using code and then binding these variables in a data object known as a *data frame*. As you explore the R ecosystem you will come across a special type of data frame known as a *tibble*. Most users can treat tibbles and data frames interchangeably because a tibble *is* a data frame that behaves in 3 subtly different ways. However, you need to be aware that these two different types of data frame exist to know that anything you learn about a data frame probably applies to a tibble and vice versa.

Entering data manually is a lengthy topic, and realistically you would mostly use this method only for fairly small data sets, so we won’t cover how to here — but see Field ([in press](#ref-fielddsr22026)). More likely, you will have collected and stored data using specialist survey (e.g., Qualtrics, Survey Monkey) or experiment generator (e.g., Inquisit, ePrime, and PsychoPy) software. Alternatively, if it does need to be entered manually, you might choose to use an interactive spreadsheet such as Microsoft Excel, numbers, Apache OpenOffice or Google sheets. As mentioned earlier, the Tidyverse has several specialist packages for importing different types of data file, but often importing will go more smoothly if you export the data into a text-based file format, such as CSV or a tab-delimited text, before importing into R.

### Importing data

In our example analysis we will use the data from Perchtold et al. ([2019](#ref-perchtold2019)) (link in the resources), that is available as an Excel file (pone.0211618.s002.xlsx). Fortunately, we can use the read\_excel function from the readxl packages that is installed as part of tidyverse. Let’s assume we have downloaded the data file and saved it in a folder called data within our main project folder. We can import the data using the following code:

perchtold\_df <- here("data/pone.0211618.s002.xlsx") |>   
 read\_excel()

This code uses here() to get the full file path for the data file by appending the text within the function to the file path for the project. This file path is then piped into read\_excel(), that reads in the file and converts it to a data frame; we save this data frame as perchtold\_df (but we could have named it anything).

Similarly, had the file been a CSV file we could have imported it using read\_csv() from the readr package.

perchtold\_df <- here("data/pone.0211618.s002.csv") |>   
 read\_csv()

To view an object, we can select the name of the object in our code chunk and execute it. RStudio will display the object below the code chunk. If we want to view our data, we should select perchtold\_df and press *Ctrl* (*Cmd* on MacOS) and the return key. Note that the data have been read in as a series of columns, each representing a variable, and each row represents a participant ([Figure 3](#fig-perchtolddf)).

### Data types

When a data file is imported the function used takes an educated guess at what each column contains. For example, in [Figure 3](#fig-perchtolddf), note that each column has <dbl> underneath; this label indicates that read\_excel thinks that these columns contain numbers with decimals. R recognises the following data types, that are labelled with abbreviations of the highlighted word:

* int: numbers in *integer* format (i.e., whole numbers).
* dbl: numbers in *double*-precision floating-point format. In layperson terms these are any real number. Doubles have decimal places whereas integers do not.
* chr: values containing strings of *characters*. In other words text.
* fct: a special kind of character variable where the text strings represent categories or groups. This kind of variable is known as a *factor*. When a variable is designated as a factor, each unique text string is stored as a ‘level’ of a categorical variable. An example of a factor would be a variable that takes the text values ‘Treatment’ and ‘Control’ in the data file depending on whether a person was assigned to a treatment condition or a control group. When you want to fit models with categorical variables, you will work with factors.
* date: values that contain dates.
* lgl: values that are either TRUE or FALSE — known as *logical* or boolean values.

Although functions for importing variables typically do a good job of guessing the data type, they do get it wrong (dates in Excel are particularly messy) so you might need to manually specify the data type of each column within the function or change the data type after the data has been imported. It is fair to say there is a lot to learn about processing data, and that we do not have room to explore, so we recommend reading Field ([in press](#ref-fielddsr22026)).

### Accessing variables

Sometimes, we might want to extract a particular column or variable from a data frame or work with a subset of the variables. When working with a single variable, the easiest way to access it is using the $ symbol. In general terms:

dataframe$variable\_name

If we wanted to work with the variable called psyc within the perchtold\_df data frame, we would do this using

perchtold\_df$psyc

If we want a selection of variables we can use select() from dplyr; this function takes the form

select(.data = name\_of\_dataframe, list\_of\_variables)

For example, if we wanted to use only the variables psych, pos3 and pos7 in our data frame, we could achieve this using

select(.data = perchtold\_df, psych, pos3, pos7)

or using the pipe:

perchtold\_df |>   
 select(psych, pos3, pos7)

## Example analysis

To demonstrate R’s extensive analytic capability, we use open data from a study conducted by Perchtold et al. ([2019](#ref-perchtold2019)). In this study 95 qualified participants aged 17 to 35 years completed questionnaires measuring both their ability to generate humorous reappraisals of adverse emotional events (the humorous adaptation of the reappraisal inventiveness test, RIT) and depression (the Center for Epidemiology Studies depression scale, CES-D). In the RIT participants were confronted with 4 anxiety-provoking situations (e.g. *Late at night, you are the only one left working at the office. As you are sitting at your desk, suddenly all the lights on your floor switch off*) and were asked to write as many humorous ways as they could think of to diminish anxiety. Each of the humorous explanations (or reappraisals) was classified into RIT categories; these categories were further classified as either a positive reinterpretation strategy (situation-focused reappraisal) or a de-emphasizing strategy (self-focused reappraisal). The CES-D (variable name ads) resulted in an overall score from 1 to 48 — higher scores indicating a greater number of self-reported depressive experiences ([Radloff, 1977](#ref-radloff1977)). The data are available from <https://doi.org/10.1371/journal.pone.0211618.s002>.

We begin by demonstrating some of R’s statistical functions using a one-sample *t*-test, which is not found in Perchtold et al. ([2019](#ref-perchtold2019)). Thereafter, we use R to replicate some the analyses reported in Perchtold et al. ([2019](#ref-perchtold2019)). Specifically, we replicate some of Table 2 on page 8 of Perchtold et al. ([2019](#ref-perchtold2019)) (available from <https://doi.org/10.1371/journal.pone.0211618.t002>):

* The correlations and *p*-values in column 3 (labelled *r(p)*).
* The unstandardized parameter estimates (regression coefficients) and their associated estimated standard errors in column 5 (labelled *B(se)*).

Remember from earlier sections that our Quarto document should begin with a code chunk that loads the libraries we need and reads in the data file.

library(correlation)  
library(here)  
library(knitr)  
library(parameters)  
library(performance)  
library(readxl)  
library(tidyverse)  
  
perchtold\_df <- here("data/pone.0211618.s002.xlsx") |>   
 read\_excel()

### One-Sample *t*-Test

A first analysis determines whether the participants differ from a sample diagnosed with clinical depression. Depressive symptoms were measured with the CES-D. Based on the original study that describes the scale ([Radloff, 1977](#ref-radloff1977)), a score of 16 or more is interpreted as indicating depression; however, more recent work with clinical samples suggests that a score of 21 or more has better sensitivity and specificity in discriminating samples with clinical depression from general population samples ([Shean & Baldwin, 2008](#ref-shean2008)). Therefore, our hypothesis that participants differ from a sample diagnosed with clinical depression can be tested using a one-sample *t*-test. Let’s assume that before collecting data we set the Type I error rate to be = 0.05.

* H\_0: the true mean CES-D is equal to 21
* H\_1: the true mean CES-D is less than 21

In R, we perform a *t*-test of this hypothesis using the t.test() function that is available in base R. The general form of the t-test() is shown below:

t.test(x, y = NULL,  
 alternative = c("two.sided", "less", "greater"),  
 mu = 0, paired = FALSE, var.equal = FALSE,  
 conf.level = 0.95, …)

Note that the function includes arguments that specify: x—the variable from a data frame (or vector of data), alternative—a text string indicating whether the test is two-sided or one-sided, and the direction of the one-sided test (“less” or “greater”), mu—the value against which you’re testing (by default zero), conf.level—the confidence level, paired—whether the data are paired (e.g. dependent), and, var.equal—whether variances are assumed to be equal (by default they are not assumed equal and we advise not changing this argument). The last two arguments apply only when a two-sample *t*-test is performed.

For our data, we want to test the variable ads in the data. We can access this variable in the perchtold\_df data frame using perchtold\_df$ads (see earlier); we set x = perchtold\_df$ads. We want to compare the sample mean of this variable to a value of 21, so we set mu = 21; we are interested specifically in whether the sample mean is less than 21 (a one-tailed test), so we set alternative = "less". We have saved the results in an object that we’ve chosen to call depression\_ost (we can use whatever name we like, but the \_ost reminds us it is a *o*ne-*s*ample *t*-test). To look at the text output of the *t*-test we execute depression\_ost as on the second line.

depression\_ost <- t.test(x = perchtold\_df$ads, mu = 21, alternative = "less" )   
depression\_ost

The text output looks like this:

One Sample t-test  
  
data: perchtold\_df$ads  
t = -5.0516, df = 94, p-value = 1.073e-06  
alternative hypothesis: true mean is less than 21  
95 percent confidence interval:  
 -Inf 17.44955  
sample estimates:  
mean of x   
 15.70989

On examination of the output ([Table 1](#tbl-ost)) the *p*-value for this test is < 0.001 which is less our criterion = 0.05. Therefore, we have sufficient evidence to reject the null hypothesis and conclude at the 5% level of significance that the true mean CES-D value is less than 21.

The default text output isn’t really suitable for reports or journal articles, but the function model\_parameters() in the parameters package ([Lüdecke et al., 2020](#ref-parameters)) can be used to create consistent tabulated output from many statistical functions including t.test() and lm(). Therefore, having created and stored the object depression\_ost, we can create a nicely formatted table by placing it into this function. Further formatting can be achieved by piping this table into the display() function from the insights package, where, among other things you can set the number of decimal places shown using three arguments. digits (2 by default) sets the number of decimal places for any numeric values except confidence intervals and *p*-values; ci\_digits (also 2 by default) sets the decimal places for confidence intervals; p\_digits (3 by default) sets the number of decimal places for *p*-values. For example, if you wanted one decimal place in general, but 3 for confidence intervals and 4 for *p* values you would use display(digits = 1, ci\_digits = 3, p\_digits = 4). However, if we use the sensible defaults we could display a table using:

model\_parameters(depression\_ost) |>   
 display()

### Correlation Coefficient

Perchtold et al. ([2019](#ref-perchtold2019)) correlate self-reported depressive experiences to each of the three sub-strategies within the positive reinterpretation strategy—general positive aspects (pos3), worst-case comparison (pos7), disadvantage as advantage (pos8)—and each of the three sub-strategies within the de-emphasizing category—alternative explanation (rel10), trivializing the problem (rel11), handing over responsibility (rel14). These correlation coefficients can be found in Table 2 of the Perchtold article in the column labelled ‘r(p)’.

The correlation() function in the correlation package ([Makowski et al., 2022](#ref-correlation)) can be used to generate correlation coefficients between all columns in a data frame, as well as *p*-values and confidence intervals. This function has the general form:

correlation::correlation(data = my\_dataframe,  
 method = "pearson",  
 p\_adjust = "holm",  
 ci = 0.95)

Only the main arguments and their defaults are shown. Use data to specify the name of the data frame containing the variables that you wish to correlate and method to set the correlation coefficient. the default method is Pearson but you can use Kendall or Spearman’s method with method = "kendall" and method = "spearman" respectively. Furthermore, p\_adjust defines a method for correcting *p*-values for the number of tests conducted (by default the Holm method is used but others are available), and ci sets the confidence interval width as a proportion (by default it is 0.95 for a 95% confidence interval).

To get the correlations we need, we can pipe our data frame into the select() function, that we met earlier, to select the variables we need, and then pipe this into correlation(). Perchtold and colleagues did not adjust their *p*-values for the number of tests, so we need to over-ride the default of p\_adjust = "holm" to p\_adjust = "none" to replicate their results exactly

perchtold\_df |>  
 select(ads, pos3, pos7, pos8, rel10, rel11, rel14) |>  
 correlation(p\_adjust = "none")

If you run this code, you will see, for example, the unadjusted *p*-value of the ads-pos3 correlation = , matching the value reported in Perchtold Table 2. Other things being equal, it’s good practice to adjust the *p*-values when using multiple tests, so let’s do that by leaving the defaults as they are (note the brackets of the correlation function are now empty). We will also save the results as perchtold\_cor so that we have more options for printing the output as a table.

perchtold\_cor <- perchtold\_df |>  
 select(ads, pos3, pos7, pos8, rel10, rel11, rel14) |>  
 correlation()

For the purpose of report-writing, we could pipe perchtold\_cor into display() to produce a tabulated output ([Table 2](#tbl-cor)). Note that the correlation coefficients are reported in the r column; for example, the correlation coefficient for the ads-pos3 pair is . The *p*-values have now been adjusted for multiple comparisons, so note that the adjusted *p*-value of the ads-pos3 correlation coefficient has changed to 0.434.

perchtold\_cor |>   
 display()

### Multiple Linear Regression

#### Fitting the model.

Perchtold et al. ([2019](#ref-perchtold2019)) also used multivariate linear regression to test whether three sub-strategies within the positive reinterpretation strategy—general positive aspects (pos3), worst-case comparison (pos7), disadvantage as advantage (pos8)—and three sub-strategies within the de-emphasizing category—alternative explanation (rel10), trivializing the problem (rel11), handing over responsibility (rel14) are predictive of self-reported depressive experiences measured via the CES-D depression scale (ads). These results are reported in Table 2 of Perchtold et al. ([2019](#ref-perchtold2019)).

Several of the research questions that can be answered using multiple linear regression:

1. Do *any* of the six reappraisal strategies have a linear relationship to self-reported depressive experience? This question is addressed by looking at an overall test of the fit of the model.
2. Which of the six strategies explain variation in self-reported depressive experience? This question is answered by testing the estimated parameter for each strategy against a value of 0.
3. What is the likely size of the relationship between a specific strategy and self-reported depressive experience? This question is answered through interpreting the confidence interval for a parameter under certain important assumptions.
4. Is the model biased? Although this is not a primary research question, any statistical model we fit makes certain assumptions; the answers to your substantive research questions are only a good as the extent to which your model assumptions hold.

In R, we answer these questions by estimating a linear regression using the lm() function. We designate the data frame stored in the object perchtold\_df and specify the model using a formula with the dependent variable on the left side of a tilde (~) and the explanatory variables on the right side of the tilde (separated by + signs).

ads ~ pos3 + pos7 + pos8 + rel10 + rel11 + rel14

The estimated model can be written to an object; we have chosen to call it ads\_reg in this case. A text summary of the model results can be viewed using the summary() function.

ads\_reg <- lm(data = perchtold\_df, ads ~ pos3 + pos7 + pos8 + rel10 + rel11 + rel14)   
summary(ads\_reg)

The text output looks like this:

Call:  
lm(formula = ads ~ pos3 + pos7 + pos8 + rel10 + rel11 + rel14,   
 data = perchtold\_df)  
  
Residuals:  
 Min 1Q Median 3Q Max   
-15.5765 -7.0530 -0.9251 6.7948 29.2241   
  
Coefficients:  
 Estimate Std. Error t value Pr(>|t|)   
(Intercept) 15.1549 2.4245 6.251 1.42e-08 \*\*\*  
pos3 -4.3719 1.3359 -3.273 0.00152 \*\*   
pos7 4.7280 1.5149 3.121 0.00244 \*\*   
pos8 0.3791 0.8455 0.448 0.65496   
rel10 0.1369 0.3862 0.354 0.72386   
rel11 -0.0746 0.8142 -0.092 0.92721   
rel14 -0.6992 2.9514 -0.237 0.81329   
---  
Signif. codes: 0 '\*\*\*' 0.001 '\*\*' 0.01 '\*' 0.05 '.' 0.1 ' ' 1  
  
Residual standard error: 9.715 on 88 degrees of freedom  
Multiple R-squared: 0.152, Adjusted R-squared: 0.09414   
F-statistic: 2.628 on 6 and 88 DF, p-value: 0.02164

#### Classes and Lists.

The text summary is fine, but for reports we might want to extract specific information from the model. When R fits any model, it produces useful data elements that are unseen in the output; these are not printed by using a function such as summary(). For example, an object created by the lm() function contains useful data such as all of the residuals and fitted values. These do not need to be calculated; you just need to learn how to extract them.

The first step is to write your output to an object similar to what we did when we created ads\_reg. The object will have standardized names for the information it contains and will store this information in a *list*. A list object is like a bucket—it contains things, and these things can be different from one another. Whereas a data frame or tibble must have a fixed number of rows across all columns, a list is not limited to that structure. A list can contain a character vector of length 1 and also a numeric array with dimensions .

In R, each object created in a particular way is designated as a class. For example, an object created by lm() has the same structure and is designated as a class “lm” object. Classes allow the creation of a template to store information, and just as importantly, to extract information. For example, every “lm” object is stored as class-lm and objects of this class will contain residuals, fitted values, etc. Users can write functions or packages to manipulate class objects such as a class-lm object. Most of us will not write our own functions to extract information from a class. Instead, we use existing packages and functions to do that for us. For example, when we put our ads\_reg model, that we created with lm(), into the function model\_parameters(), it checks its class (and complains if it is a class that it doesn’t recognise); having identified its class as class-lm it knows how to navigate the list of objects contained within it and find the parameter estimates. It does so before processing this information into a table for us.

#### Extracting information from the model and reporting it.

To create tabulated reports of our model, we can again use model\_parameters() to extract parameter estimates from ads\_reg and tabulate them for us. Similarly, to obtain fit statistics, we can use the model\_performance() function; to test the fit, we can use test\_wald() both from the performance package ([Lüdecke et al., 2021](#ref-performance)). The performance package also has the function check\_model() to produce assumption checks.

##### Question 1: Do any of the six reappraisal strategies have a linear relationship to self-reported depressive experience?

To get overall fit statistics of the model and test it, we can place the model we created ads\_reg into model\_performance() and test\_wald() respectively. When only one model is provided to test\_wald(), it will compare it to a model with no predictors. The output tells us that the model fit of the data well (i.e., including predictors significantly improves the fit), *F*(6, 88) = 2.63, *p* = 0.022. The fit statistics tell us that the model accounts for 9.4% of the depression scores when adjusting for the number of predictors ( = 0.094).

model\_performance(ads\_reg)  
test\_wald(ads\_reg)

The text output looks like this:

# Indices of model performance  
  
AIC | AICc | BIC | R2 | R2 (adj.) | RMSE | Sigma  
---------------------------------------------------------------  
710.318 | 711.993 | 730.749 | 0.152 | 0.094 | 9.350 | 9.715

Name | Model | df | df\_diff | F | p  
------------------------------------------------  
Null model | lm | 94 | | |   
Full model | lm | 88 | 6.00 | 2.63 | 0.022  
Models were detected as nested (in terms of fixed parameters) and are compared in sequential order.

##### Question 2: Which of the six strategies explain variation in self-reported depressive experience?

To answer this question, we need the model parameters and tests of whether they differ from 0. We can obtain these in tabulated form by placing our model in model\_parameters() and using display() to print it nicely (and optionally set the number of decimal places). [Table 3](#tbl-pe) shows that pos3 and pos7 significantly predicted depression (their *p*-values are less than our a priori alpha of 0.05). The estimates themselves tell us that as general positive aspects increase by one unit, depression on the ADS falls by 4.37 units; that is, as worst-case comparison increases by one unit, depression on the ADS increases by 4.73 units.

model\_parameters(ads\_reg) |>   
 display()

##### Question 3: What is the likely size of the relationship between a specific strategy and self-reported depressive experience?

To address this question, we need to look at the confidence intervals for each parameter estimate and interpret them under certain important assumptions (see [Field, in press](#ref-fielddsr22026)). A 95% confidence interval is constructed such that the population or true value of the estimate will fall within the interval for 95% of samples *in the long run*. It does not tell us the interval within which the true population falls unless we *assume that our sample is one of the 95% that contains the population value* and accept that this assumption will be wrong 5% of the time. Under this (possibly incorrect) assumption, [Table 3](#tbl-pe) suggests, for example, the likely size of effect for general positive aspects is between and . In other words, the best case scenario is that as general positive aspects increase by one unit, depression on the ADS falls by 7.03 units. However, the worst case scenario is that ADS falls by only 1.72 units. You can interpret the other confidence intervals in a similar way.

##### Question 4: Is the model biased?

For linear models estimated using ordinary least squares, the model assumptions, in order of importance, are ([Field, in press](#ref-fielddsr22026)):

* Linearity and additivity
* The expected value of errors is zero
* Spherical errors
  + Homoscedasticity (homogeneity of variance)
  + Independence of errors
* Normally distributed errors and a normally distributed sampling distribution

We also need to consider whether there are influential cases that have biased the model. Although the assumptions relate to the (unobservable) model errors we test them using the (observable) residuals of the fitted model.

Space prevents a detailed dive into the impact of violating each assumption, but broadly speaking they have the potential to bias the parameter estimates and the standard error. Biased standard errors result in biased confidence intervals and significance tests of parameter estimates. In terms of parameter estimates, linearity and additivity and influential cases matter, in terms of standard errors all of these assumptions (potentially) matter—see Field ([in press](#ref-fielddsr22026)) for a thorough review.

In R, we can test all of these assumption visually by placing our model into the check\_model() function from the performance package ([Lüdecke et al., 2021](#ref-performance)) and using the default arguments.

check\_model(ads\_reg)

[Figure 4](#fig-ads_reg) shows the resulting plots. Let’s look at what these plots tell us about each assumption:

* Linearity and additivity: the top-right plot suggests a lack of linearity in the residuals. Linearity would be indicated by a flat reference line whereas the observed line is not. It is also noteworthy that there are two large fitted values that appear to contribute a lot to the lack of linearity.
* The expected value of errors is zero: the top-right plot suggests that residuals are approximately symmetrical around 0 suggesting that model errors might reasonably have an expected value of 0.
* Homoscedasticity: the middle-left plot suggests heteroscedasticity because again the reference line is not flat.
* Normally distributed errors: the bottom-right plot suggests that residuals (and, therefore, model errors) are not normally distributed because the observed residuals do not fall on the horizontal line.
* Normally distributed sampling distribution: with a sample of 95 we might reasonable assume (based on the central limit theorem) that this assumption holds.
* Influential cases: the middle-right plot suggests that there are no influential cases because all of the standardized residuals fall within the contour lines.

The lack of linearity in the residuals suggests that a linear model is not the most appropriate model for the data. Also, the fact that the model residuals suggest that we cannot assume homoscedasticity or normality point towards fitting a robust model that uses robust standard errors ([Field & Wilcox, 2017](#ref-field2017)). Fortunately, this is easily done by using the vcov = "HC4" in model\_parameters(). We could, for example, revise our earlier code to get [Table 4](#tbl-robpe). Compare this table with [Table 3](#tbl-pe), note that the standard errors, significance tests and confidence intervals have changed. Notably, pos7 no longer significantly predicts depression and its confidence interval crosses zero suggesting that under the usual assumptions the effect of the predictor could plausibly be zero. This example highlights the importance of checking model assumptions and adjusting for bias.

model\_parameters(ads\_reg, vcov = "HC4") |>   
 display()

## Summary

R is a powerful tool in the workflow of any scientist. In conjunction with RStudio and Quarto, it enables the user to produce production-ready reports and articles that embed the statistical analysis within. This makes documents portable and reproducible. R has a steep learning curve, but the key is to establish good workflow practices (use RStudio projects, be organised) and to not be afraid to try and fail. We have shown that with relatively little code (and experience), it is possible to fit statistical models and create summary tables from them that are embedded within a single document. The rest, is up to you.

Table 1

One Sample t-test

| Parameter | Mean | mu | Difference | 95% CI | t(94) | p |
| --- | --- | --- | --- | --- | --- | --- |
| perchtold\_df$ads | 15.71 | 21.00 | -5.29 | (-Inf, 17.45) | -5.05 | < .001 |

Alternative hypothesis: true mean is less than 21

Table 2

Correlation coefficients from Perchtold et al. (Table 2)

Correlation Matrix (pearson-method)

| Parameter1 | Parameter2 | r | 95% CI | t(93) | p |
| --- | --- | --- | --- | --- | --- |
| ads | pos3 | -0.23 | (-0.42, -0.03) | -2.31 | 0.434 |
| ads | pos7 | 0.21 | (0.01, 0.40) | 2.09 | 0.706 |
| ads | pos8 | 0.03 | (-0.18, 0.23) | 0.24 | > .999 |
| ads | rel10 | 0.04 | (-0.17, 0.24) | 0.35 | > .999 |
| ads | rel11 | 0.04 | (-0.16, 0.24) | 0.43 | > .999 |
| ads | rel14 | -5.47e-03 | (-0.21, 0.20) | -0.05 | > .999 |
| pos3 | pos7 | 0.33 | (0.14, 0.50) | 3.37 | 0.023\* |
| pos3 | pos8 | 0.05 | (-0.15, 0.25) | 0.53 | > .999 |
| pos3 | rel10 | -0.02 | (-0.22, 0.18) | -0.19 | > .999 |
| pos3 | rel11 | -0.14 | (-0.33, 0.07) | -1.34 | > .999 |
| pos3 | rel14 | -0.09 | (-0.28, 0.12) | -0.86 | > .999 |
| pos7 | pos8 | -0.01 | (-0.21, 0.19) | -0.12 | > .999 |
| pos7 | rel10 | 1.77e-03 | (-0.20, 0.20) | 0.02 | > .999 |
| pos7 | rel11 | 0.03 | (-0.17, 0.23) | 0.28 | > .999 |
| pos7 | rel14 | -0.05 | (-0.25, 0.15) | -0.47 | > .999 |
| pos8 | rel10 | -0.02 | (-0.22, 0.19) | -0.16 | > .999 |
| pos8 | rel11 | -0.15 | (-0.34, 0.05) | -1.49 | > .999 |
| pos8 | rel14 | -0.11 | (-0.31, 0.09) | -1.08 | > .999 |
| rel10 | rel11 | 0.05 | (-0.15, 0.25) | 0.49 | > .999 |
| rel10 | rel14 | 0.24 | (0.04, 0.42) | 2.35 | 0.422 |
| rel11 | rel14 | -0.06 | (-0.26, 0.14) | -0.60 | > .999 |

p-value adjustment method: Holm (1979) Observations: 95

Table 3

Parameter estimates from Perchtold et al. (Table 2)

| Parameter | Coefficient | SE | 95% CI | t(88) | p |
| --- | --- | --- | --- | --- | --- |
| (Intercept) | 15.15 | 2.42 | (10.34, 19.97) | 6.25 | < .001 |
| pos3 | -4.37 | 1.34 | (-7.03, -1.72) | -3.27 | 0.002 |
| pos7 | 4.73 | 1.51 | (1.72, 7.74) | 3.12 | 0.002 |
| pos8 | 0.38 | 0.85 | (-1.30, 2.06) | 0.45 | 0.655 |
| rel10 | 0.14 | 0.39 | (-0.63, 0.90) | 0.35 | 0.724 |
| rel11 | -0.07 | 0.81 | (-1.69, 1.54) | -0.09 | 0.927 |
| rel14 | -0.70 | 2.95 | (-6.56, 5.17) | -0.24 | 0.813 |

Table 4

Parameter estimates with robust standard errors from Perchtold et al. (Table 2)

| Parameter | Coefficient | SE | 95% CI | t(88) | p |
| --- | --- | --- | --- | --- | --- |
| (Intercept) | 15.15 | 2.49 | (10.21, 20.10) | 6.09 | < .001 |
| pos3 | -4.37 | 1.20 | (-6.75, -1.99) | -3.65 | < .001 |
| pos7 | 4.73 | 2.66 | (-0.56, 10.01) | 1.78 | 0.079 |
| pos8 | 0.38 | 0.90 | (-1.41, 2.17) | 0.42 | 0.675 |
| rel10 | 0.14 | 0.39 | (-0.65, 0.92) | 0.35 | 0.729 |
| rel11 | -0.07 | 0.65 | (-1.36, 1.21) | -0.12 | 0.908 |
| rel14 | -0.70 | 3.89 | (-8.42, 7.03) | -0.18 | 0.858 |

Figure 1

How R, RStudio and Quarto interact

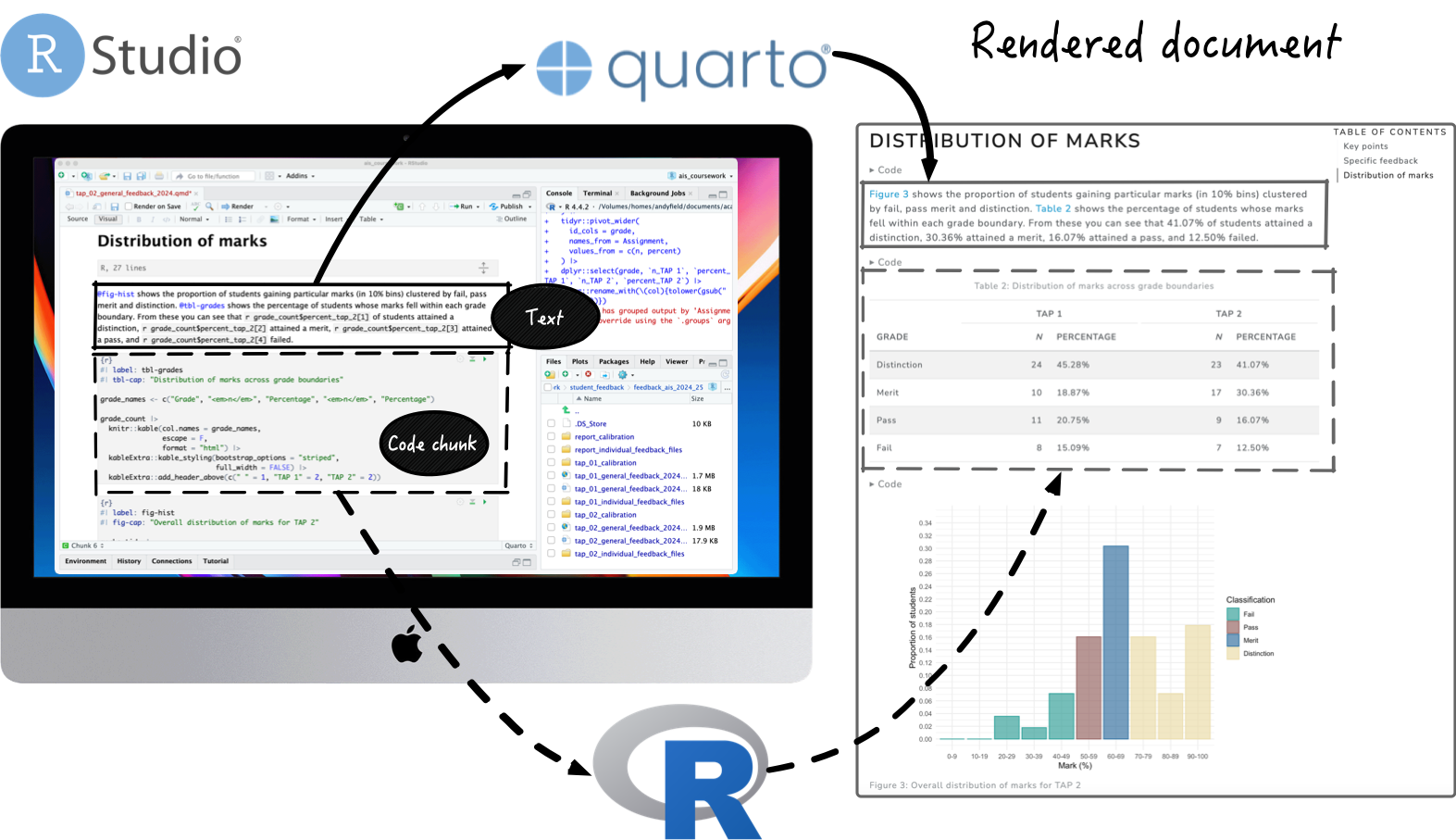


Figure 2

Example of a code chunk



Figure 3

Data from Perchtold et al. (2019)

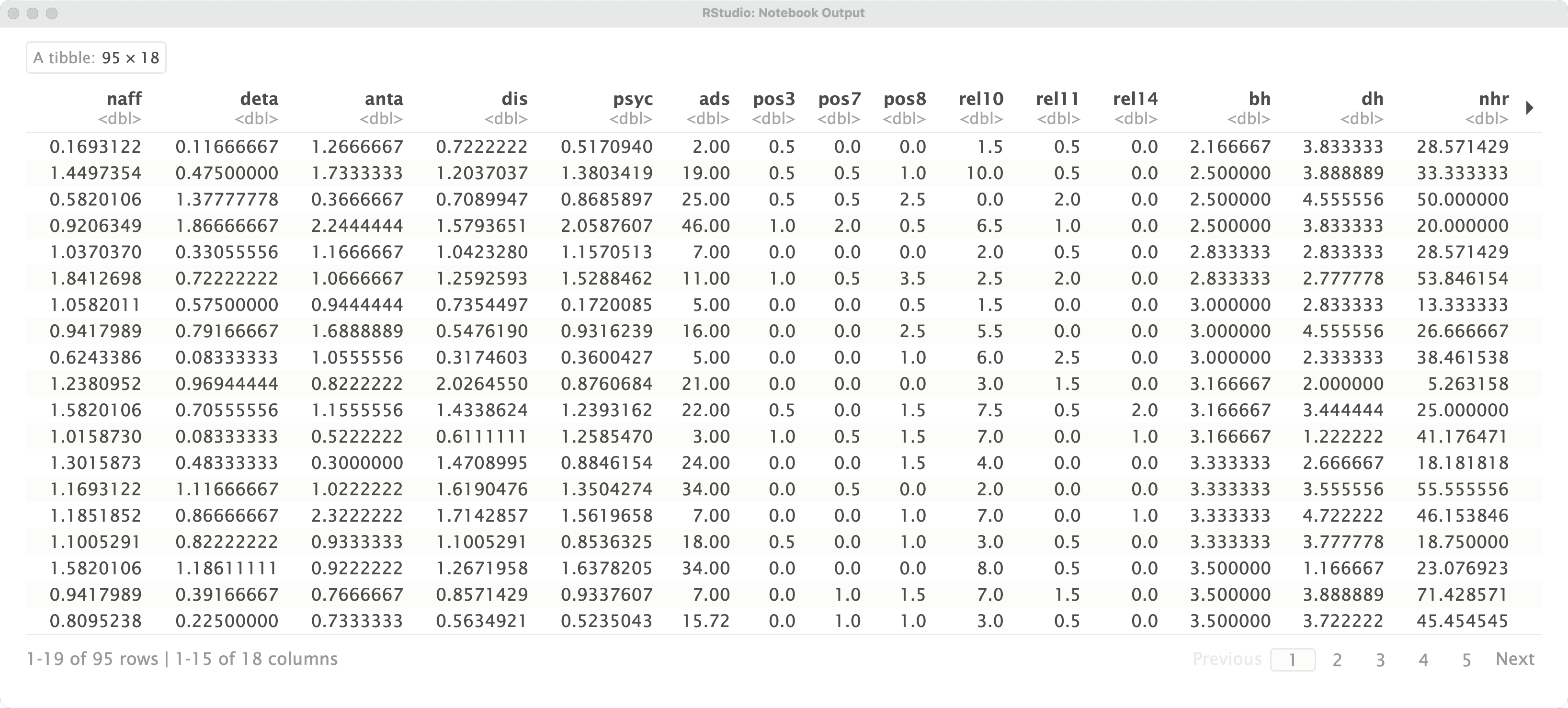
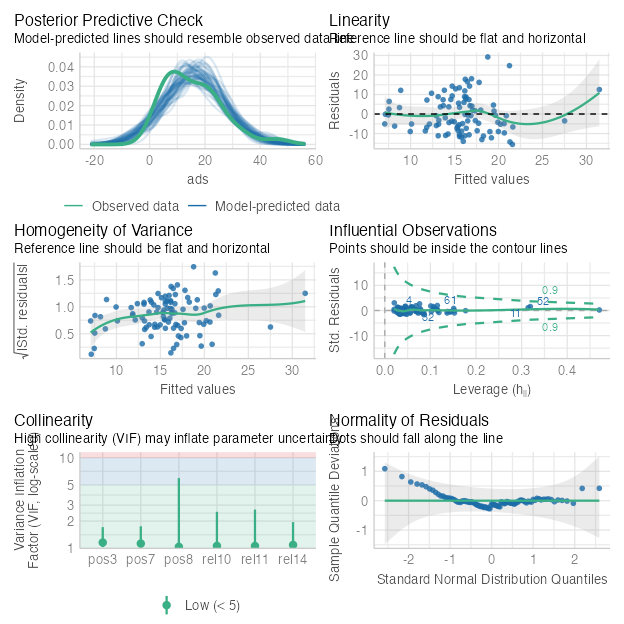


Figure 4

Residual plots



Allaire, J., & Dervieux, C. (2024). *quarto: R Interface to “Quarto” Markdown Publishing System*. <https://CRAN.R-project.org/package=quarto>

Bryan, J. (2023). *googlesheets4: Access Google Sheets using the Sheets API V4*. <https://CRAN.R-project.org/package=googlesheets4>

D’Agostino McGowan, L., & Bryan, J. (2023). *googledrive: An Interface to Google Drive*. <https://CRAN.R-project.org/package=googledrive>

Field, A. P. (in press). *Discovering statistics using R and RStudio* (2nd ed.). Sage.

Field, A. P., & Wilcox, R. R. (2017). Robust statistical methods: A primer for clinical psychology and experimental psychopathology researchers. *Behaviour Research and Therapy*, *98*, 19–38. <https://doi.org/10.1016/j.brat.2017.05.013>

Grolemund, G., & Wickham, H. (2011). *Dates and Times Made Easy with {lubridate}* (Vol. 40, pp. 1--25). <https://www.jstatsoft.org/v40/i03/>

Lüdecke, D., Ben-Shachar, M. S., Patil, I., & Makowski, D. (2020). *Extracting, Computing and Exploring the Parameters of Statistical Models using {R}.* (Vol. 5, p. 2445). <https://doi.org/10.21105/joss.02445>

Lüdecke, D., Ben-Shachar, M. S., Patil, I., Waggoner, P., & Makowski, D. (2021). *{performance}: An {R} Package for Assessment, Comparison and Testing of Statistical Models* (Vol. 6, p. 3139). <https://doi.org/10.21105/joss.03139>

Makowski, D., Wiernik, B. M., Patil, I., Lüdecke, D., & Ben-Shachar, M. S. (2022). *{{correlation}}: Methods for Correlation Analysis*. <https://CRAN.R-project.org/package=correlation>

Müller, K. (2020). *here: A Simpler Way to Find Your Files*. <https://CRAN.R-project.org/package=here>

Müller, K., & Wickham, H. (2023). *tibble: Simple Data Frames*. <https://CRAN.R-project.org/package=tibble>

Munafò, M. R., Nosek, B. A., Bishop, D. V. M., Button, K. S., Chambers, C. D., Percie Du Sert, N., Simonsohn, U., Wagenmakers, E.-J., Ware, J. J., & Ioannidis, J. P. A. (2017). A manifesto for reproducible science. *Nature Human Behaviour*, *1*(1), 21. <https://doi.org/10.1038/s41562-016-0021>

Perchtold, C. M., Weiss, E. M., Rominger, C., Feyaerts, K., Ruch, W., Fink, A., & Papousek, I. (2019). Humorous cognitive reappraisal: More benign humour and less “dark” humour is affiliated with more adaptive cognitive reappraisal strategies. *PLOS ONE*, *14*(1), e0211618. <https://doi.org/10.1371/journal.pone.0211618>

R Core Team. (2024). *R: A language and environment for statistical computing*. R Foundation for Statistical Computing.

Radloff, L. S. (1977). The CES-D scale: a self-report depression scale for research in the general population. *Applied Psychological Measurement*, *1*(3), 385–401. <https://doi.org/10.1177/014662167700100306>

Rexer Analytics. (2020). *2020 data science survey highlight*. <https://www.rexeranalytics.com/data-science-survey>

Rosseel, Y. (2012). *{lavaan}: An {R} Package for Structural Equation Modeling* (Vol. 48, pp. 1--36). <https://doi.org/10.18637/jss.v048.i02>

RStudio Team. (2016). *RStudio: Integrated Development for R*. RStudio, Inc. <http://www.rstudio.com/.>

Shean, G., & Baldwin, G. (2008). Sensitivity and specificity of depression questionnaires in a college-age sample. *The Journal of Genetic Psychology*, *169*(3), 281–292. <https://doi.org/10.3200/GNTP.169.3.281-292>

Simons, J. E., & Holmes, D. T. (2019). Reproducible research and reports with R. *The Journal of Applied Laboratory Medicine*, *4*(3), 471–473. <https://doi.org/10.1373/jalm.2018.028373>

Staples, T. L. (2023). Expansion and evolution of the R programming language. *Royal Society Open Science*, *10*(4), 221550. <https://doi.org/10.1098/rsos.221550>

Wickham, H. (2016). *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York. <https://ggplot2.tidyverse.org>

Wickham, H. (2023a). *forcats: Tools for Working with Categorical Variables (Factors)*. <https://CRAN.R-project.org/package=forcats>

Wickham, H. (2023b). *stringr: Simple, Consistent Wrappers for Common String Operations*. <https://CRAN.R-project.org/package=stringr>

Wickham, H., Averick, M., Bryan, J., Chang, W., McGowan, L. D., François, R., Grolemund, G., Hayes, A., Henry, L., Hester, J., Kuhn, M., Pedersen, T. L., Miller, E., Bache, S. M., Müller, K., Ooms, J., Robinson, D., Seidel, D. P., Spinu, V., … Yutani, H. (2019). *Welcome to the {tidyverse}* (Vol. 4, p. 1686). <https://doi.org/10.21105/joss.01686>

Wickham, H., & Bryan, J. (2023). *readxl: Read Excel Files*. <https://CRAN.R-project.org/package=readxl>

Wickham, H., François, R., Henry, L., Müller, K., & Vaughan, D. (2023). *dplyr: A Grammar of Data Manipulation*. <https://CRAN.R-project.org/package=dplyr>

Wickham, H., & Henry, L. (2023). *purrr: Functional Programming Tools*. <https://CRAN.R-project.org/package=purrr>

Wickham, H., Hester, J., & Bryan, J. (2024). *readr: Read Rectangular Text Data*. <https://CRAN.R-project.org/package=readr>

Wickham, H., Miller, E., & Smith, D. (2023). *haven: Import and Export “SPSS,” “Stata” and “SAS” Files*. <https://CRAN.R-project.org/package=haven>

Wickham, H., Vaughan, D., & Girlich, M. (2024). *tidyr: Tidy Messy Data*. <https://CRAN.R-project.org/package=tidyr>