Good Statistical Practice (GSP): guidance for using R for scientific publication

Caroline X. Gao

Centre for Youth Mental Health, University of Melbourne

School of Public Health and Preventive Medicine, Monash University

Orygen, Australia

Contents

Preface

This short practice guidance is designed based on a few guidelines and practice principles, books and journal articles, including:

- ASA "Ethical Guidelines for Statistical Practice"
- Reproducible Research with R and R Studio
- Efficient R programming

The aim of this guidance is to promote accountability, reproducibility and integrity of statistical practice in the Health Service and Outcome Research (HSOR) team at Orygen, Centre for Youth Mental Health, University of Melbourne. The guidance is divided into four parts: 1) planning for analysis, 2) data cleaning, 3) analysis, and 4) reporting.

Important note: **the authors do not give permission for you to print this resource on papers, unless you are experimenting with magical ink that disappears after 3 months (you are considered to have net-zero carbon emissions in this case).** If you are thinking about reading this on paper before going to sleep, you are too opportunistic for your study plan : P

Have fun ~~

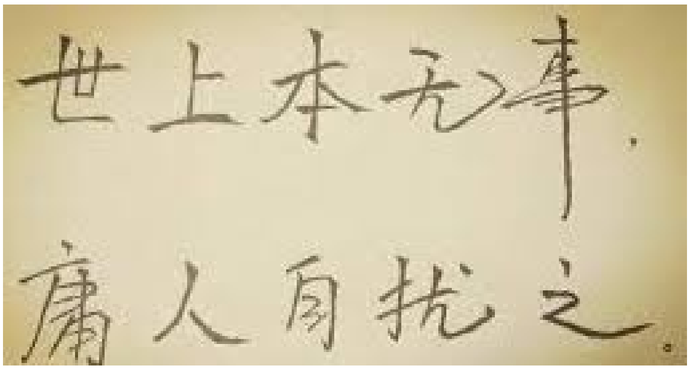
Planning for analysis

Drafting the analysis plan

The analysis will need to be planned before you touch the data. Your analysis may deviate from the analysis plan to some degree, which may relate to data integrity of the variable selected, model fitting factors (e.g.,. multicollinearity, heteroscedasticity etc) and change of research questions. However, unless your aim is purely exploratory (e.g., identify latent clusters) or predictive, your analysis should be guided by the analysis plan to prevent "fishing" results. Remember "If you torture the data long enough, it will confess to anything - Ronald H. Coase", which is really against the scientific principle. The scientific evidence is based on a collection of studies and findings rather than a single paper. If you have a negative finding, you are obligated to publish it!

Less is more

Be aware of the “Complexity Bias”. “Life is really simple, but we insist on making it complicated”.



世上本无事。
庸人自扰之。

The famous quote is not by Confucius (it is from the New Book of Tang published about 1500 years after his death), and it is not well translated, but you get the idea. We often find it easier to face a complex problem than a simple one, and often feel that a complex solution must be better than an easier one. However, this is often not true in applied statistics.

More often than not, ingenious statistical designs are surprisingly simple. A famous example is the Cox model for survival analysis, which simplifies the need to describe the baseline hazard function with proportional hazard assumption. Another example, Dijkstra’s algorithm (algorithm for finding the shortest paths between nodes in a graph), the author, Edsger Dijkstra, once said “One of the reasons that it is so nice was that I designed it without pencil and paper. I learned later that one of the advantages of designing without pencil and paper is that you are almost forced to avoid all avoidable complexities.”

This is the same with your analysis, always force yourself to avoid complexities if you could achieve what you need with a simpler model. There are numerous benefits for simpler models, e.g., less likely to over-fitting with your model, easier to communicate with your audience, less likely to make mistakes etc. If a logistic regression works, there is no need to use a structural equation model.

Officially certified plans

It is important to get your analysis plan approved or reviewed. Some studies have strict protocols on who and when the analysis plan will need to be approved together with other ethical requirements. Other studies will only require you to discuss with your supervisors. Regardless, it will be better to have a written analysis plan with review and/or approval and avoid confusion down the track. Sometimes you might want or need to preregister your study, which is considered as a part of the open science practice.

Use R

Why R?

The essential skill set in the modern analytical community is the ability to use one or more script languages, SPSS doesn't count here. It's often critical for the success of your publications. More often than not, you might face challenges from the reviewer to do something that SPSS is not capable of. If you have already started your analysis in SPSS, please abandoned it ASAP. If R is in its early adolescent years, SPSS is a toddler and it suffers from the Peter Pan Syndrome (it will never grow up).

Source: unknown

Sorry for being a bit offensive, but the reality is the industry and scientific community are gradually moving away from SPSS to other languages for many reasons, cost, capacity, flexibility, reproducibility etc. So to avoid the long term pain, change to R (Stata is also good since this is a practice guide for R, I won't touch on Stata).

Learn to use R

"R being hard to learn" mostly exists in our fears. Once you get on to it the challenges are mostly "feeling a bit confused", there are so many packages, so many ways to do the same thing, so much typing needed, so many materials everywhere… So the best way to learn R is to learn by using it. You can start with a short intensive introduction course (one or two days) to build up your confidence. Start using it in a small project, and then gradually roll it out to more parts of your analysis tasks. You can also start by learning your collaborator's code (communicate with the author and create a separate environment so that you won't break anything).

There are lots of good online training materials for R:

- R for Reproducible Scientific Analysis from Software Carpentry
- R Programming on Coursera by Roger Peng,
- An Introduction to R
- R for Data Science.

Almost all R users are still learning R, so you have no excuse not to.

R setup and project management

Set up R on your computer

One thing that you have to remember: R is a fast evolving-language. It has a new version every few months (sometimes two versions in one month), with funny names.

```r
library(rversions)
tail(r_versions())
```

```
##     version                date             nickname
## 120   4.0.3 2020-10-10 07:05:24 Bunny-Wunnies Freak Out
## 121   4.0.4 2021-02-15 08:05:13        Lost Library Book
## 122   4.0.5 2021-03-31 07:05:15          Shake and Throw
## 123   4.1.0 2021-05-18 07:05:22          Camp Pontanezen
## 124   4.1.1 2021-08-10 07:05:06              Kick Things
## 125   4.1.2 2021-11-01 08:05:12              Bird Hippie
```

Although R kept on updating, you do not need to re-install R all the time. But I tend to update my R half-yearly. It doesn't take a long time to update everything nowadays. The first thing that you need to do after installing R is to install your commonly used packages. A good way to install + load packages is to use the *pacman* package, which is much faster than typing install.package() and library().

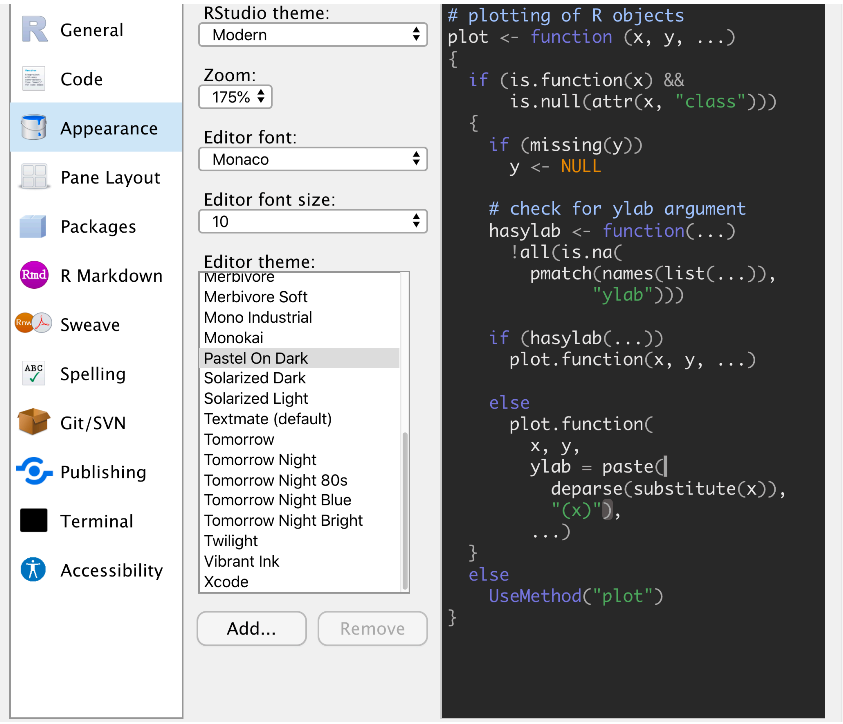
```r
# load libraries
library(pacman)
p_load("dplyr", "tidyr", "ggplot2")
```

install.package() and *pacman* install packages from R CRAN. However, the authors may have additional update on Github which is not loaded to CRAN. If you come across a problem that seems to be a software "bug", you can install the most resent package from Github using *install_github*. You can also check whether your issue was being reported on Github (e.g., https://github.com/rstudio/bookdown/issues).

```r
library(devtools)
install_github("rstudio/bookdown")
```

Customizing RStudio

The modern use of R is almost always via RStudio. After installing the RStudio, you might want to customize it based on your own preferences. See the guide here. I like the "Pastel On Dark" editor theme (green and blue dominated dark theme).



RStudio theme: Modern

Zoom: 175%

Editor font: Monaco

Editor font size: 10

Editor theme: Pastel On Dark

```r
# plotting of R objects
plot <- function (x, y, ...)
{
  if (is.function(x) &&
      is.null(attr(x, "class")))
  {
    if (missing(y))
      y <- NULL

    # check for ylab argument
    hasylab <- function(...)
      !all(is.na(
        pmatch(names(list(...)),
               "ylab")))

    if (hasylab(...))
      plot.function(x, y, ...)

    else
      plot.function(
        x, y,
        ylab = paste(
          deparse(substitute(x)),
          "(x)"),
        ...)
  }
  else
    UseMethod("plot")
}
```

I also like to turn on a few diagnostics including "Check usage of <- in function call" and "Provide R style diagnostics" (make sure you have a good R coding style, see style guide here and a tidyverse style guide here). There is no need to strictly follow these guidelines, but you need to be aware of what is good, acceptable or bad.

Keep tracking of R and package versions

For all your analysis, you need to keep track of the R environment, this includes the R version, platform and all package versions that were loaded in your current analysis environment. R environment information can be easily summarised using *sessionInfo*.

```
sessionInfo()
```

```
## R version 4.0.2 (2020-06-22)
## Platform: x86_64-apple-darwin17.0 (64-bit)
## Running under: macOS  10.16
##
## Matrix products: default
## BLAS:   /Library/Frameworks/R.framework/Versions/4.0/Resources/lib/libRblas.dylib
## LAPACK: /Library/Frameworks/R.framework/Versions/4.0/Resources/lib/libRlapack.dylib
##
## locale:
## [1] en_AU.UTF-8/en_AU.UTF-8/en_AU.UTF-8/C/en_AU.UTF-8/en_AU.UTF-8
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base
##
## other attached packages:
##  [1] rversions_2.0.2  kableExtra_1.3.1 knitr_1.37       forcats_0.5.1
##  [5] stringr_1.4.0    dplyr_1.0.8      purrr_0.3.4      readr_2.1.1
##  [9] tidyr_1.2.0      tibble_3.1.6     ggplot2_3.3.5    tidyverse_1.3.0
## [13] pacman_0.5.1
##
## loaded via a namespace (and not attached):
##  [1] Rcpp_1.0.8        lubridate_1.8.0   here_1.0.1        assertthat_0.2.1
##  [5] rprojroot_2.0.2   digest_0.6.29     conflicted_1.0.4  utf8_1.2.2
##  [9] R6_2.5.1          cellranger_1.1.0  backports_1.4.1   reprex_0.3.0
## [13] evaluate_0.15     httr_1.4.2        pillar_1.7.0      rlang_1.0.2
## [17] curl_4.3.2        readxl_1.3.1      rstudioapi_0.13   rmarkdown_2.11
## [21] webshot_0.5.2     munsell_0.5.0     broom_0.7.12      compiler_4.0.2
## [25] modelr_0.1.8      xfun_0.30         pkgconfig_2.0.3   htmltools_0.5.2
## [29] tidyselect_1.1.2  bookdown_0.21     fansi_1.0.2       viridisLite_0.4.0
## [33] crayon_1.5.0      tzdb_0.2.0        dbplyr_2.0.0      withr_2.5.0
## [37] grid_4.0.2        jsonlite_1.8.0    gtable_0.3.0      lifecycle_1.0.1
## [41] DBI_1.1.0         formatR_1.7       magrittr_2.0.2    scales_1.1.1
## [45] cli_3.2.0         stringi_1.7.6     fs_1.5.2          xml2_1.3.3
## [49] ellipsis_0.3.2    generics_0.1.2    vctrs_0.3.8       tools_4.0.2
## [53] glue_1.6.2        jpeg_0.1-8.1      hms_1.1.1         fastmap_1.1.0
## [57] yaml_2.3.5        colorspace_2.0-3  rvest_1.0.2       memoise_1.1.0
## [61] haven_2.4.3
```

In fact, it might also be a good idea to include an R package version table in the appendix of your paper.

```r
my_session_info <- sessionInfo()
# extract name, version and date
Version <- lapply(my_session_info$otherPkgs, function(x) x$Version)
Version <- unlist(lapply(Version, function(x) ifelse(identical(x,
    character(0)), " ", x)))
Date <- lapply(my_session_info$otherPkgs, function(x) as.character(as.Date(x$Date)))
Date <- unlist(lapply(Date, function(x) ifelse(identical(x,
    character(0)), " ", x)))

# extract session info in to data frame
info_table <- data.frame(Package = names(my_session_info$otherPkgs),
    Version = Version, Date = Date)

# keep in two columns as your list gets too
# long
if (nrow(info_table)%%2 == 0) {
    info_table <- cbind(info_table[c(TRUE, FALSE),
        ], info_table[c(FALSE, TRUE), ])
} else {
    info_table <- cbind(info_table[c(TRUE, FALSE),
        ], rbind(info_table[c(FALSE, TRUE), ],
        c("", "", "")))
}


# display table
rownames(info_table) <- NULL
knitr::kable(info_table, booktabs = TRUE, linesep = "") %>%
    kable_styling(bootstrap_options = "striped")
```

Package	Version	Date	Package	Version	Date
rversions	2.0.2	2020-05-25	kableExtra	1.3.1	2020-10-22
knitr	1.37	2021-12-16	forcats	0.5.1	2021-01-27
stringr	1.4.0	2019-02-10	dplyr	1.0.8	2022-02-08
purrr	0.3.4	2020-04-17	readr	2.1.1	2021-11-30
tidyr	1.2.0	2022-02-01	tibble	3.1.6	2021-11-07
ggplot2	3.3.5	2021-06-25	tidyverse	1.3.0	2019-11-21
pacman	0.5.1	2019-03-11			

This is crucial as packages or R updates may make your code fail or your results differ. This sounds scarier than it really is. All analysis packages will have this issue. It's slightly complected with R because updates of R and affiliated packages are not synchronised. For my 15 years of using R, I haven't found this particularly challenging for a few reasons. The packages update rates were not as fast as we are seeing today. Most of my projects were in isolated environments so I rarely need to re-run the analysis a few years later and will need to produce exactly the same results. If I need to re-run the analysis, I will mostly update the analysis code and make sure it adheres to the current best practice. Occasionally, some of my package updates will cause problems with my current analysis code. As I keep track of my R environment, I will be able to figure out the problem quickly and re-install the older version back.

```
require(devtools)
install_version("bookdown", version = "0.20",
    repos = "http://cran.us.r-project.org")
```

If you are in an area that requires 100% reproducibility all the time, you can use **Microsoft R open** together with *checkpoint* which locks down the set of packages in your project environment, so package updates will not impact your code. There are a few other alternatives, but you can get other benefits from **Microsoft R open** without too much trouble, so it seems to be a wise choice at the current stage. If you need to install lots of things from Github and work on package development, it may not be a good choice.

Project management with R

Over the years of being a programmer and biostatistician, I have learned many things in a painful way. One of them is the importance of good project management with your data, analysis, documentation and drafts. I had to spend a long time to remember what I did, where I save things and also lots of detective work on whether anyone have touched my 'cheese' (with files stored on network drives). When I am lucky, I can request IT to restore a backup at a specific time to find the lost treasure. But in many cases, this doesn't work as either I was stupid enough not to save things on the network drive, or the network drive deletes backups automatically.

Project management became more critical when I had to change my work between computers, laptops and servers as well as swap frequently between Windows and Mac. Managing file versions, backups and file paths became additional burdens. However, all of these things became much easier with RStudio + Github. It's also more user friendly with people who are not familiar with Git.

Essentially the idea is that you can create an independent project environment for isolated task or tasks that share a common data source. This can be a specific analysis for a paper or a range of analyses based on the same dataset. When there are multiple users, it's always best to use a separate project environment for individual analysis, so users do not load the same project file on the network drive.

The method for setting up projects are described here. There are many benefits of using projects:

- No need to type the long path directory in the project environment because it is using a relative file path.
- You can use the here package when working with R Markdown files within the project environment. The default file path in R Markdown is the R Markdown file location, using here package, it always points to the project directory. So you have your path directory consistent when using console and Knit (see R Markdown in the next session).
- Access all files in the project environment easily
- Easy to communicate with Github

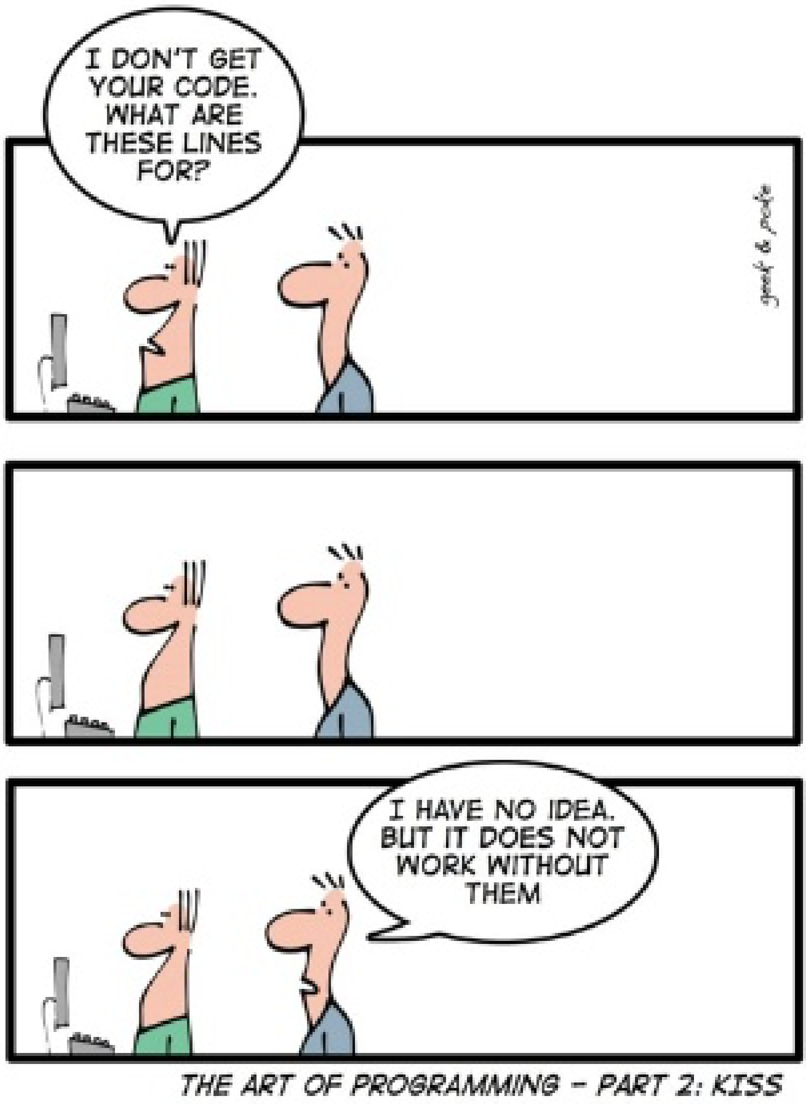
You can set up a rule for organising sub-folders and files. I like to separate code, data and results. However, it is slightly complicated to split analysis code and results when you are using

R Markdown Regardless, it would be better to have a sub-folder structure in place so you can find files easily.

State-of-art R Markdown/R Notebook practice

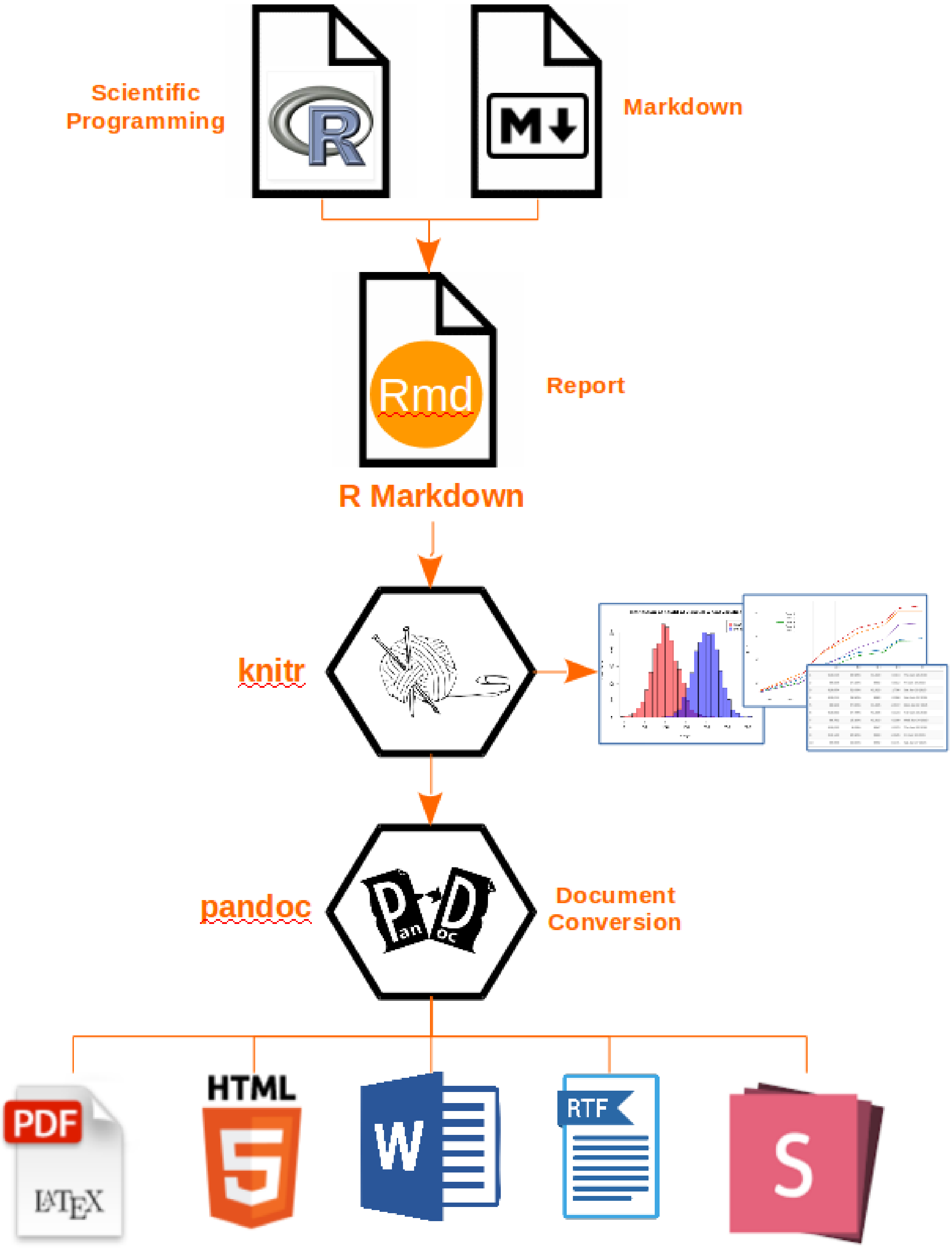
The idea of Markdown practice was originated from Donald Knuth. Donald believed that programmers should think of programs as works of literature and introduced a new programming paradigm called "Literate programming". He then developed TeX to facilitate this paradigm of coding, which is the typesetting system that LaTex is based. The idea behind it is simple, combining the text and code for humans to read not computers. In the end, scientific discoveries are for humans to interpret, replicate and extend, and not for computers.

Although this is a fascinating idea, it had been hard to achieve for many reasons. Academic publications mostly only allow words, tables, figures but not computer code, so there is no obvious momentum from the academic communities. Software developers and computer science engineers mostly focused on the end product and not so much on displaying the process in between. Technically it is also difficult to achieve as this approach will need to be implemented within or closely connected to one or multiple programming environments.


I DON'T GET YOUR CODE. WHAT ARE THESE LINES FOR?
geek & poke
I HAVE NO IDEA. BUT IT DOES NOT WORK WITHOUT THEM
THE ART OF PROGRAMMING - PART 2: KISS

Source: http://geek-and-poke.com/geekandpoke/2009/7/25/the-art-of-programming-part-2.html

However, the sudden boom of data science brought in new challenges. As software engineers move towards data science, the community gradually realised that the traditional coding style (e.g., modular programming, black-box approach) became barriers instead of advantages. Each step of the programming code, from cleaning variables to running algorithm, suddenly became very important. Hence there is an immediate need to communicate regarding every step of the process. This needs drove the birth of many interactive computational notebooks (IPython, SageMath, Beaker, Jupyter, Apache Zeppelin and R Markdown). R Markdown became the most successful computational notebooks for R users. Why? Probably for the same reason as to why R has been successful, simplicity, flexibility, open source etc.



Source: http://applied-r.com/project-reporting-template/

R Markdown is a system that integrates narrative, analysis, code and output to create a production quality document. The code and text were implemented via R in a *.Rmd file, which will be knit to Markdown file and then convert to file type of choice via Pandoc. There are a few important feature that makes it one of R's “killer” feature

- Reproducibility and transparency. All data processes can be integrated in to one central location to allow the work being reproduced and/or published easily. You are no longer required to live in the copying and pasting world. All analyses and results can be easily reproduced with changes in source data, cleaning routine and analyses. There are different degrees of implementing R Markdown (e.g., save data cleaning code, produce interactive tables and plots, write the full paper). Regardless of where you are at, using R Markdown will significantly reduce your workload, avoid copying and pasting error, and improve the reproducibility and transparency. There is no need to panic when the data manage has identified one person is missing from the data set and all analysis will need to be repeated.

- Easy progress tracking and debugging. Before the age of R Markdown, R users will need to store all the code into a plain text file or R file and execute the code one by one to see the results. When the code file is getting very long, it gets harder to maintain the code and changes. Then coders will use a split system to break long code into modules to make the progress tracking and debugging easier. However, this will no longer be the case with R Markdown. All code can be execute when you "Knit" the file and errors will be reported.You can also headers to indexing sections of your code and use hyperlinks to quickly locate to different sections.

- Communication. You can integrate narrative, code and results together. This is quite crucial in the communication in data analysis because it avoids all the efforts of reading segregated code comments, switching between documents to find results and maintaining separate documentation. All these process can be integrated into one system.

- Very nice for equations. If you need to type lots equations, then R Markdown is also your friend. The standard LaTeX equations works with all types of outputs (when your equations are getting complicated, sometimes word doesn't work)

- Extension. R Markdown system provides you will limitless extension capacities, such as integrating multiple languages (I can combine python, Stan and R in the same file now), producing multiple types of documents (word, html, pdf, slides, books, shiny app etc), displaying interactive plots

J.J. Allaire gave an excellent introduction presentation, Notebooks with R Markdown, in the 2016 useR conference. I hope I have convinced you to move to R Markdown.

Here are some practical point.

Learn to use R Markdown

There are many good online introductions on using R markdown:

- R Markdown from R Studio
- R Markdown Basics by Andy Lin from the famous IDRE, UCLA stats resource website.
- R Markdown cheatsheet
- R Markdown Reference Guide
- R Markdown: The Definitive Guide by Yihui Xie
- bookdown: Authoring Books and Technical Documents with R Markdown (for advanced users) by Yihui Xie

YAML header

R Markdown files (*.Rmd) starts with YAML headers, which specify document parameters (or pandoc parameters), such as title, author, type of document, parameters etc.

```
---
title: "Good Statistical Practice (GSP)"
output: html_document
---
```

One thing you have to remember is that indentation has its meaning in YAML header. See the following example, having a table of content (toc), is a sub-option for the HTML document, hence is it needs to be indented.

```
---
title: "Good Statistical Practice (GSP)"
output:
  html_document:
    toc: TRUE
---
```

There is no comprehensive YAML guide exist. However, there is a nice package called ymlthis, which provides an addin for choosing YAML specifications. I normally use the bookdown output styles which makes, references tables and figures citation slightly easier for both pdf and word documents. The YAML header for this document looks like this:

```
---
title: "Good Statistical Practice (GSP)"
```

```
author:
- name: Caroline X. Gao [1,2], Matthew Hamilton [3]
output:
   bookdown::pdf_document2:
     toc: yes
     number_sections: false
     latex_engine: xelatex
     pandoc_args:
      - --template=template.tex
   bookdown::word_document2:
     toc: yes
     number_sections: false
     reference_docx: "template.docx"
geometry: margin=1in
fontsize: 11pt
bibliography: GSP.bib
tables: yes
header-includes:
   \usepackage{float}
   \floatplacement{figure}{H}
   \newcommand{\beginsupplement}{\setcounter{table}{0}  \renewcommand{\thetable}{S\arabic{ta
   \usepackage{lscape}
   \newcommand{\blandscape}{\begin{landscape}}
   \newcommand{\elandscape}{\end{landscape}}
---
```

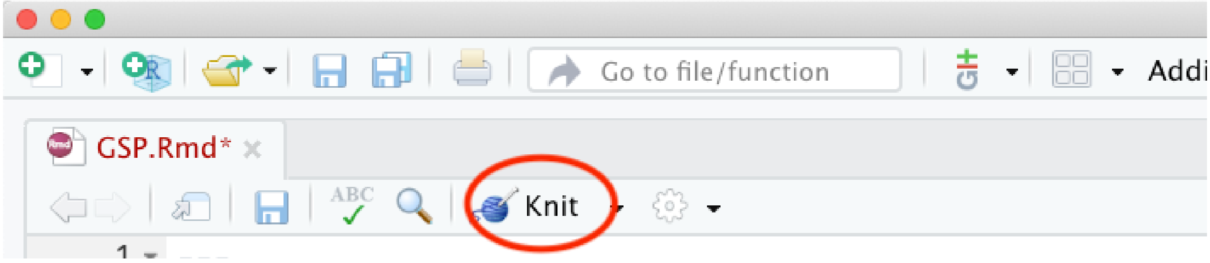
You may notice that I have used template documents, “template.tex” and “template.docx”. The LaTex template was inherited from Prof Rob Hyndman’s MonashEBSTemplates with minor modifications. “template.docx” is simply a Word document with defined fonts. “Header-includes” contains the additional latex command that I want to use in the documents, e.g., “lscape” package is for the landscape pdf page and “beginsupplement” is for setting supplementary table and figure captions (starts from Table S1 and Figure S1) .

Using R Markdown

R Markdown is very easy to use. Code will need to be included as code chunks and everything else outside of code chunks will be treated as formatted text, which includes, headings, lists, standard text and equations, see R Markdown cheatsheet for details. Code chunks include various definitions, such as code type (R, Python, Stan …), chunk name, whether to display code, figure size, table options etc. * R Markdown Reference Guide provides very detailed information about this.

Knit document

To create html, word or PDF file requires to Knit the Rmd document.



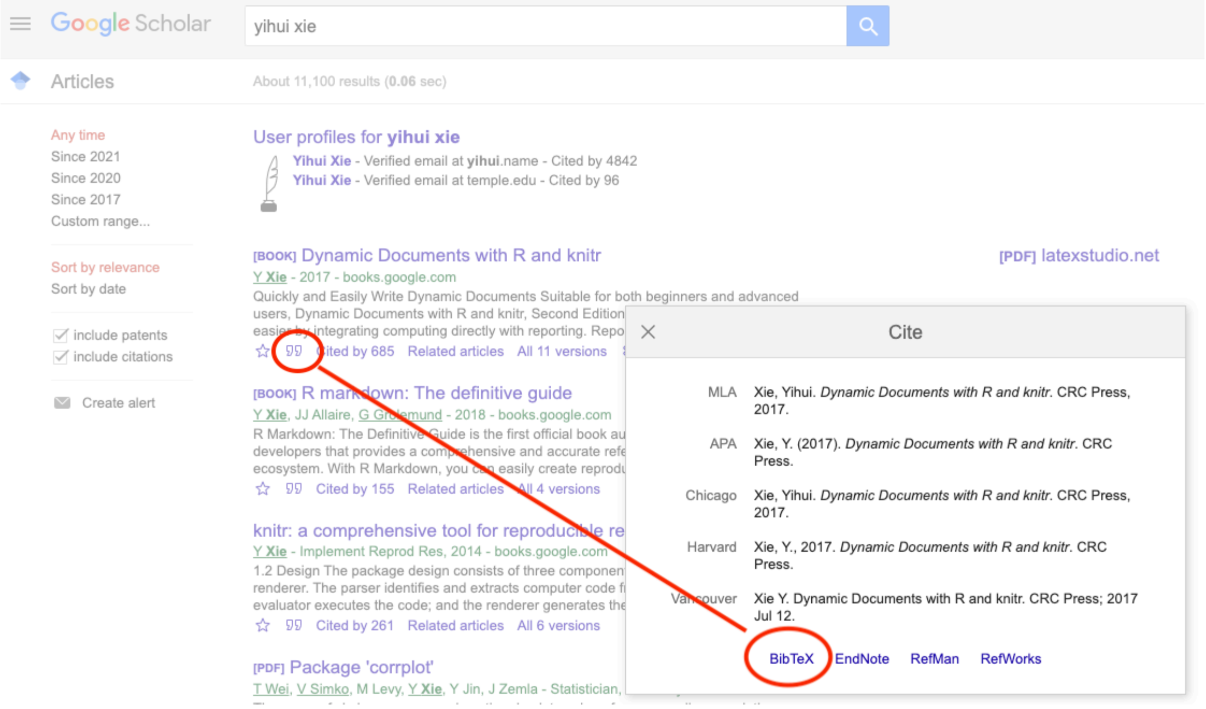
You also need to install LaTex (https://www.latex-project.org/get/) if you would like to Knit to PDF files. However, LaTex is a large monster to work with. So Yihui Xie has developed TinyTeX for R users, which require much low resources. You can install additional LaTex packages easily if needed.

Citation in R Markdown

Another advantage of using R Markdown is that you can finally give up using Endnote (the monster crashes your computer and word all the time). R Markdown can work with the many citation management file types. The commonly used one is the LaTeX bibliography management system BibTeX, in which the references were stored as plain text like this:

```
@article{Xu_2020,
  title={Socioeconomic inequality in vulnerability to
    all-cause and cause-specific hospitalisation associated
    with temperature variability: a time-series study
    in 1814 Brazilian cities},
  author={Xu, Rongbin and Zhao, Qi and Coelho,
    Micheline SZS and Saldiva, Paulo HN and
    Abramson, Michael J and Li, Shanshan and Guo, Yuming},
  journal={The Lancet Planetary Health},
  volume={4},
  number={12},
  pages={e566--e576},
  year={2020},
  publisher={Elsevier}
}
```

Most journals provide direct download of the BibTex of articles. You can also export the Endnote library to a bib file. You can also use Google Scholar to download BibTex citation. However, it does not include the doi of the paper, which is frequesntly required in journal article submission.



About 11,100 results (0.06 sec)

User profiles for yihui xie

Yihui Xie - Verified email at yihui.name - Cited by 4842

Yihui Xie - Verified email at temple.edu - Cited by 96

[BOOK] Dynamic Documents with R and knitr [PDF] latexstudio.net

Y Xie - 2017 - books.google.com

Quickly and Easily Write Dynamic Documents Suitable for both beginners and advanced users, Dynamic Documents with R and knitr, Second Edition easier by integrating computing directly with reporting. Repo

☆ 99 Cited by 685 Related articles All 11 versions

[BOOK] R markdown: The definitive guide

Y Xie, JJ Allaire, G Grolemund - 2018 - books.google.com

R Markdown: The Definitive Guide is the first official book au developers that provides a comprehensive and accurate refe ecosystem. With R Markdown, you can easily create reprodu

☆ 99 Cited by 155 Related articles All 4 versions

knitr: a comprehensive tool for reproducible re

Y Xie - Implement Reprod Res, 2014 - books.google.com

1.2 Design The package design consists of three componen renderer. The parser identifies and extracts computer code f evaluator executes the code; and the renderer generates the

☆ 99 Cited by 261 Related articles All 6 versions

[PDF] Package 'corrplot'

T Wei, V Simko, M Levy, Y Xie, Y Jin, J Zemla - Statistician,

Cite

MLA	Xie, Yihui. *Dynamic Documents with R and knitr*. CRC Press, 2017.
APA	Xie, Y. (2017). *Dynamic Documents with R and knitr*. CRC Press.
Chicago	Xie, Yihui. *Dynamic Documents with R and knitr*. CRC Press, 2017.
Harvard	Xie, Y., 2017. *Dynamic Documents with R and knitr*. CRC Press.
Vancouver	Xie Y. Dynamic Documents with R and knitr. CRC Press; 2017 Jul 12.

When using BibTeX, you need to first store the reference in a *.bib file and specify the name of the file in the YAML header. Then you can reference all your code in the text using @ followed by the id of the reference with either @Xu_2020 which generates author (year) or [@Xu_2020] which generates (author, year). By default, Pandoc uses the Chicago author-date CSL format for citations and references. You can specify the style according to the journal's requirements. Most of the journals' csl styles can be found here. All citation features can be automated including citation before or after punctuation.

```
---
title: "Good Statistical Practice (GSP)"
output:
  html_document:
    toc: TRUE
bibliography: GSP.bib
csl: biomed-central.csl
---
```

Data pre-processing

A range of terms were used to refer to the pre-processing stage of your data analysis, e.g., data cleaning, data cleansing, data wrangling, data mungling etc. This is a stage that you organize, validate, and prepare data for further analysis.

Importing data to R

R can import different types of source data (csv, excel, SPSS, Stata, SAS. SQL…). See the comprehensive guide *here*. There are three packages frequently used for processing common external data: *foreign*, *haven*, and *Hmisc*. *foreign* includes most of the importing and exporting functions, however *spss.get* from *Hmisc* provides additional enhanced features including applying proper labels, compress data etc. *haven* is also easy to work with which allows you to use the numeric values instead of directly import as factor variables.

When working with spss, *haven::read_sav()* seems to be the most robust and convenient function. It rarely has any problems and is generally fast enough; it is also part of the *tidyverse*, which allows you to %m% individual functions together.

```
dta <- haven::read_sav(here::here("Data", "testdata.sav"))
dta
```

```
## # A tibble: 5 x 4
##   numeric                    factor_numeric factor_n_coded_mi~ date
##     <dbl>                         <dbl+lbl>          <dbl+lbl> <dttm>
## 1       1   1 [strongly disagree]          1 [strongly disa~ 1983-12-11 00:00:00
## 2       2   2 [disagree]                   2 [disagree]      2018-07-01 00:00:00
## 3       3   3 [neither agree nor disagree] NA                2017-10-23 00:00:00
## 4      NA NA                               5 [strongly agre~ NA
## 5       3 NA                              NA                  NA
```

```
dta <- as_factor(dta)
dta
```

```
## # A tibble: 5 x 4
##   numeric factor_numeric             factor_n_coded_miss date
##     <dbl> <fct>                      <fct>               <dttm>
## 1       1 strongly disagree          strongly disagree   1983-12-11 00:00:00
## 2       2 disagree                   disagree            2018-07-01 00:00:00
## 3       3 neither agree nor disagree <NA>                2017-10-23 00:00:00
## 4      NA <NA>                       strongly agree      NA
## 5       3 <NA>                       <NA>                NA
```

```
dta <- foreign::read.spss(here::here("Data", "testdata.sav"),
    to.data.frame = TRUE)
```

```
## re-encoding from CP1252
```

```
head(dta)
```

```
##   numeric             factor_numeric factor_n_coded_miss        date
## 1       1          strongly disagree   strongly disagree 12659328000
## 2       2                   disagree            disagree 13749782400
## 3       3 neither agree nor disagree                <NA> 13728096000
## 4      NA                       <NA>      strongly agree          NA
## 5       3                       <NA>                <NA>          NA
```

```
dta <- Hmisc::spss.get(here::here("Data", "testdata.sav"),
    datevars = c("date"))
```

```
## re-encoding from CP1252
```

```
dta
```

```
##   numeric             factor.numeric factor.n.coded.miss       date
## 1       1          strongly disagree   strongly disagree 1983-12-11
## 2       2                   disagree            disagree 2018-07-01
## 3       3 neither agree nor disagree                <NA> 2017-10-23
## 4      NA                       <NA>      strongly agree       <NA>
## 5        3                        <NA>                <NA>       <NA>
```

Working with data.frame, tibble and data.table

Data frame is an easier understandable term, representing a flat data file with different columns having different variables with different formats. However, you might also hear tibbles. It was designed as a modern version of data frame, it allows better print visualization, flexible variable names, no row names etc. Normally it is better to use tibble, however sometimes it gets into trouble, then you need to change back to data frame.

```
as_tibble(dta)
```

```
## # A tibble: 5 x 4
##   numeric    factor.numeric             factor.n.coded.miss date
##   <labelled> <fct>                      <fct>               <date>
## 1  1         strongly disagree          strongly disagree   1983-12-11
## 2  2         disagree                   disagree            2018-07-01
## 3  3         neither agree nor disagree <NA>                2017-10-23
## 4 NA         <NA>                       strongly agree      NA
## 5  3         <NA>                       <NA>                NA
```

```
data.frame(dta)
```

```
##   numeric             factor.numeric factor.n.coded.miss       date
## 1       1          strongly disagree   strongly disagree 1983-12-11
## 2       2                   disagree            disagree 2018-07-01
## 3       3 neither agree nor disagree                <NA> 2017-10-23
## 4      NA                       <NA>      strongly agree       <NA>
## 5       3                       <NA>                <NA>       <NA>
```

If you are using large datasets, you can also use data.table. The coding style follows SQL. Although it takes some time to learn the coding style, it is suggested to be over 10 times faster than operating with data.frame.

Name you variables properly!!!!!!!!

Every time when I saw variable names such as “X1”, “V2”, “12345”, “WWF111”, “GGG222”, “Don’t use”, “?”, “Gender”, “Gender1”, “Gender2”, “Gender3”, “Gsp11229371”, "never_ending_name_but_do_not_know_what_it_means", or "theyusenamesthatjustruntogetherwithnocapitalsorunderscores", I would turn into Jeff Atwood.

Variable names have to be proper because they will make both you and your collaborator's work much easier. You will be able to find variables easily and communicate well with your code. So when you import your data, make sure that all variables have proper names (actually variable names have to be proper in the database). There are many common naming systems:

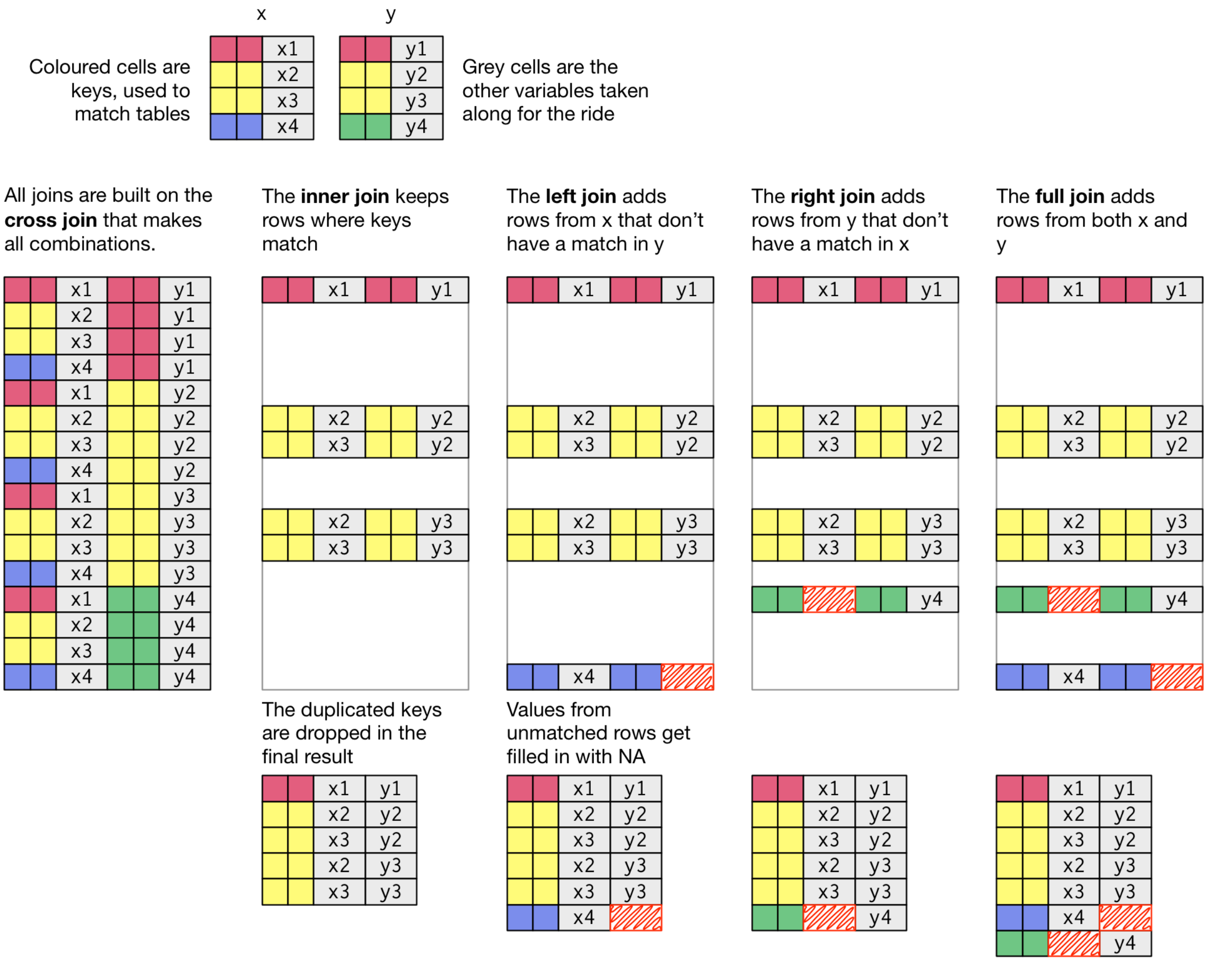
Analysis	Function and package
twoWords	lower camel uase
TwoWords	upper uamel uase
two_words	snake case
TWO_WORDS	screaming snake case
two_Words	camel snake case
Two_Words	pascal snake_case

Although snake case is mostly recommended, you can use any/all systems as long as you use them consistently. R allows you to use other special characters such as "." in variable names, but some other programming language may not allow you to do so (e.g., Stata do not allow "."). So stick with snake, camel or combined systems.

Generally, variable names should be nouns and function names should be verbs. You also need to name clusters of variables with similar yet specific names so they are easier to be referred to using *dplyr::starts_with()* or *dplyr::ends_with()*. For example, If you use "IESR_q1", "IESR_q2"…"IESR_q22" as IESR individual item names and IESR_total as IESR total scores. Then you can use *dplyr::starts_with("IESR_q")* to select individual items without the total items. If you name them as "IESR1", "IESR2"…"IESR22", "IESRTotal", *dplyr::starts_with("IESR")* will select all of them.

Organising data

In this stage, you will need to combine data from different sources into one or more datasets. In many cases, raw data were extracted from a relational database into multiple flat data files, which will need to be merged first. R has many packages for merging the datasets. The most popular functions are mutating joins from *tidyverse*. An important thing to check is whether the *by* variable(s) can uniquely define one record before merging the data. If not, joins will create all possible pairs, so you might double up your records. An easier way of checking is to count the number of records before and other *join*.


x
y
x1
x2
x3
x4
y1
y2
y3
y4
Coloured cells are keys, used to match tables
Grey cells are the other variables taken along for the ride
All joins are built on the **cross join** that makes all combinations.
The **inner join** keeps rows where keys match
The **left join** adds rows from x that don't have a match in y
The **right join** adds rows from y that don't have a match in x
The **full join** adds rows from both x and y
The duplicated keys are dropped in the final result
Values from unmatched rows get filled in with NA

Source: https://twitter.com/hadleywickham/status/684407629259526144

One important benefit of using R is that it allows you to open many datasets and process them simultaneously. This is one of the major differences compared to SPSS and Stata, which allow one data file at a time.

R also allows you to store many datasets into one list. The benefit of list + data frame may not be trivial at this stage. But it is extremely powerful to make your code simple and efficient we you work with repetitive operations with datasets having the same structure.

```r
data1 <- tibble(a = 1:4, b = 2:5)
data2 <- tibble(c = 3:6, d = 4:7)
ManyDatasets <- list(data1, data2)
ManyDatasets
```

```
## [[1]]
## # A tibble: 4 x 2
##         a     b
##   <int> <int>
## 1     1     2
## 2     2     3
## 3     3     4
## 4     4     5
##
## [[2]]
## # A tibble: 4 x 2
##         c     d
##   <int> <int>
## 1     3     4
## 2     4     5
## 3     5     6
## 4     6     7
```

Inspection

An important stage of pre-processing is to have a global understanding of your data:

- Is the number of observations of your data correct? (sometimes, it takes some effort to finalise the total sample size, particularly for longitudinal studies as participants may withdraw the study and request their data to be removed).
- Are factor variables stored as a character or numeric?
- Is the data file long or wide when there are multiple observations for one participant?
- How is missing data coded?

- What's the proportion of missing in different variables? Is there any variable with substantial missing data? If so why is the case?
- Are the nested variables correctly entered? e.g., Drinking frequency vs whether drinks or not.
- Are date variables correct?
- Is the raw data matching the definitions in the data dictionary?

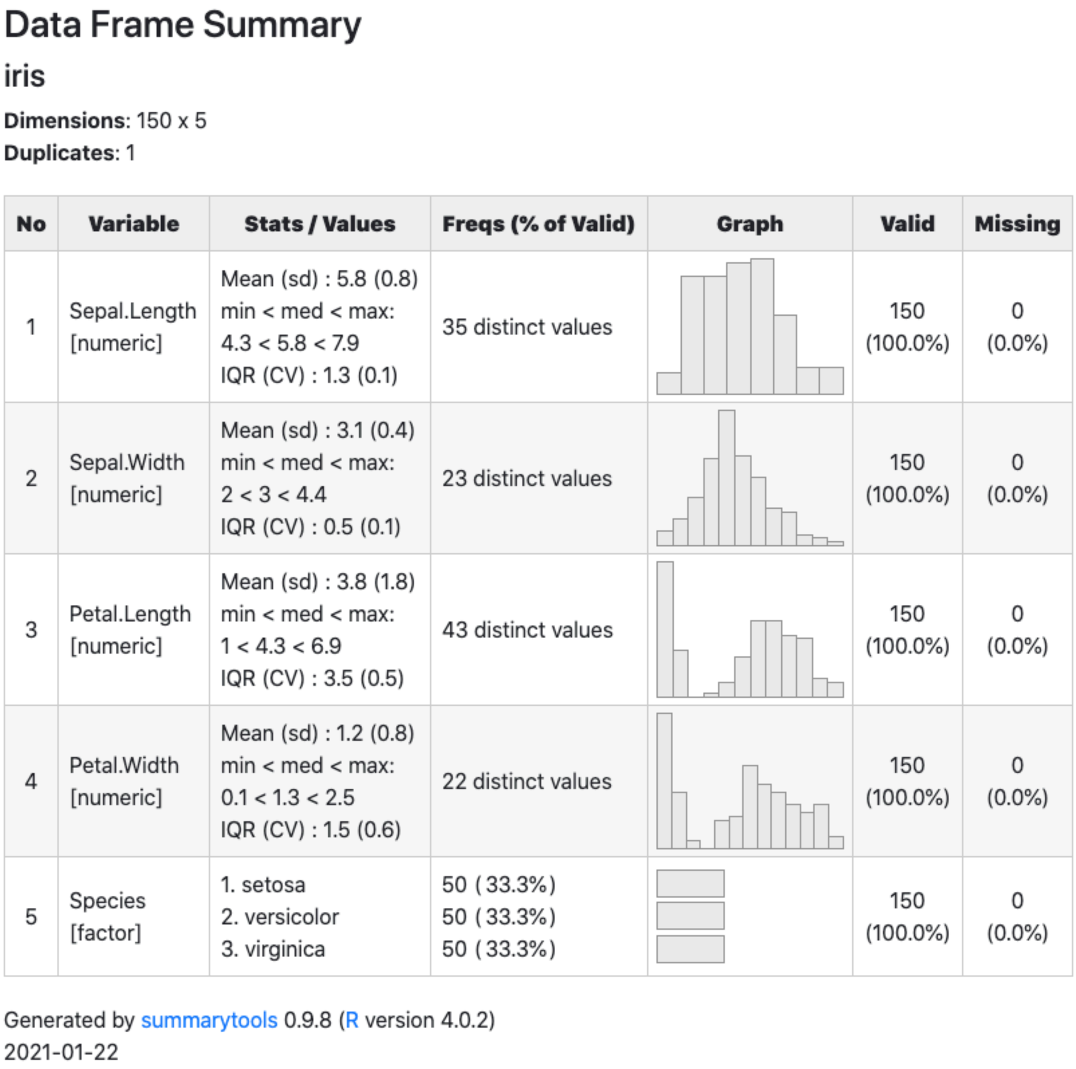
There are a few functions that are quite useful, including *summary()*, *str()*, *psych::describe()*, *tibble::glimpse()* ... The function I used the most is *describe()* from *Hmisc* which is similar with the codebook command in Stata.

```
Hmisc::describe(iris)
```

```
## iris
##
##  5  Variables      150  Observations
## --------------------------------------------------------------------------------
## Sepal.Length
##        n  missing distinct     Info     Mean      Gmd      .05      .10
##      150        0       35    0.998    5.843   0.9462    4.600    4.800
##      .25      .50      .75      .90      .95
##    5.100    5.800    6.400    6.900    7.255
##
## lowest : 4.3 4.4 4.5 4.6 4.7, highest: 7.3 7.4 7.6 7.7 7.9
## --------------------------------------------------------------------------------
## Sepal.Width
##        n  missing distinct     Info     Mean      Gmd      .05      .10
##      150        0       23    0.992    3.057   0.4872    2.345    2.500
##      .25      .50      .75      .90      .95
##    2.800    3.000    3.300    3.610    3.800
##
## lowest : 2.0 2.2 2.3 2.4 2.5, highest: 3.9 4.0 4.1 4.2 4.4
## --------------------------------------------------------------------------------
## Petal.Length
##        n  missing distinct     Info     Mean      Gmd      .05      .10
##      150        0       43    0.998    3.758    1.979     1.30     1.40
##      .25      .50      .75      .90      .95
##     1.60     4.35     5.10     5.80     6.10
##
## lowest : 1.0 1.1 1.2 1.3 1.4, highest: 6.3 6.4 6.6 6.7 6.9
## --------------------------------------------------------------------------------
## Petal.Width
##        n  missing distinct     Info     Mean      Gmd      .05      .10
##      150        0       22     0.99    1.199   0.8676      0.2      0.2
##      .25      .50      .75      .90      .95
##      0.3      1.3      1.8      2.2      2.3
##
## lowest : 0.1 0.2 0.3 0.4 0.5, highest: 2.1 2.2 2.3 2.4 2.5
## --------------------------------------------------------------------------------
## Species
##        n  missing distinct
##      150        0        3
##
## Value          setosa versicolor  virginica
## Frequency          50         50         50
## Proportion      0.333      0.333      0.333
## --------------------------------------------------------------------------------
```

dfSummary from *summarytools* is a new function being developed which provides more comprehensive evaluations including distributions, distinct values and missing data.

```
summarytools::view(summarytools::dfSummary(iris))
```



Data Frame Summary

iris

Dimensions: 150 x 5

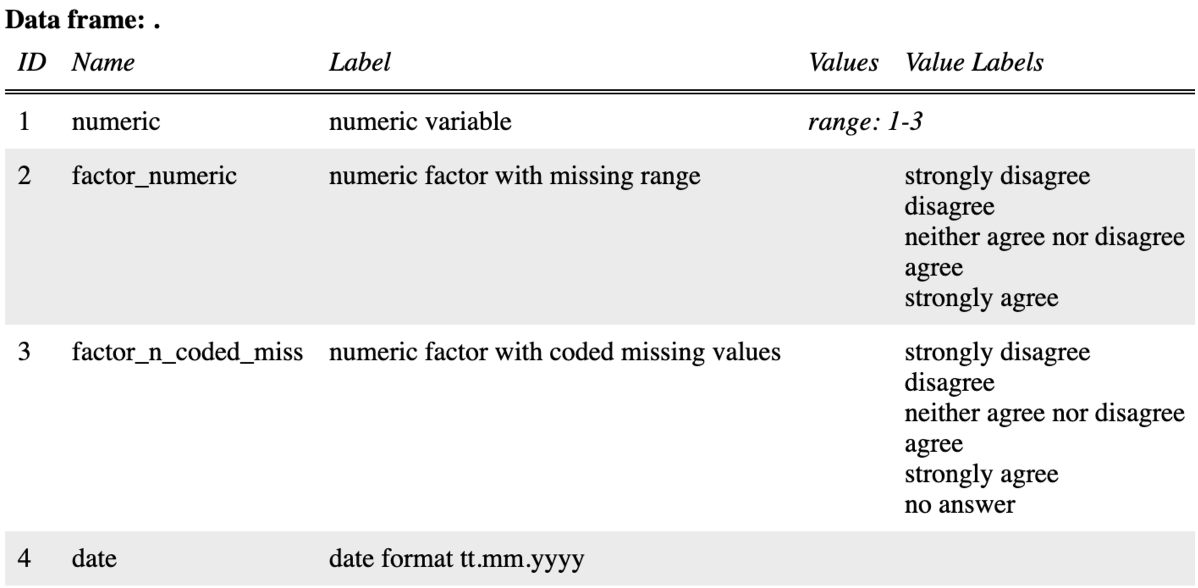
Duplicates: 1

No	Variable	Stats / Values	Freqs (% of Valid)	Graph	Valid	Missing
1	Sepal.Length [numeric]	Mean (sd) : 5.8 (0.8) min < med < max: 4.3 < 5.8 < 7.9 IQR (CV) : 1.3 (0.1)	35 distinct values		150 (100.0%)	0 (0.0%)
2	Sepal.Width [numeric]	Mean (sd) : 3.1 (0.4) min < med < max: 2 < 3 < 4.4 IQR (CV) : 0.5 (0.1)	23 distinct values		150 (100.0%)	0 (0.0%)
3	Petal.Length [numeric]	Mean (sd) : 3.8 (1.8) min < med < max: 1 < 4.3 < 6.9 IQR (CV) : 3.5 (0.5)	43 distinct values		150 (100.0%)	0 (0.0%)
4	Petal.Width [numeric]	Mean (sd) : 1.2 (0.8) min < med < max: 0.1 < 1.3 < 2.5 IQR (CV) : 1.5 (0.6)	22 distinct values		150 (100.0%)	0 (0.0%)
5	Species [factor]	1. setosa 2. versicolor 3. virginica	50 (33.3%) 50 (33.3%) 50 (33.3%)		150 (100.0%)	0 (0.0%)

Generated by summarytools 0.9.8 (R version 4.0.2)
2021-01-22

view_df from *sjPlot* also provide data summary, but more related to labels and factor choices.

```
haven::read_sav(here::here("Data", "testdata.sav")) %>%
    as_factor() %>% sjPlot::view_df()
```

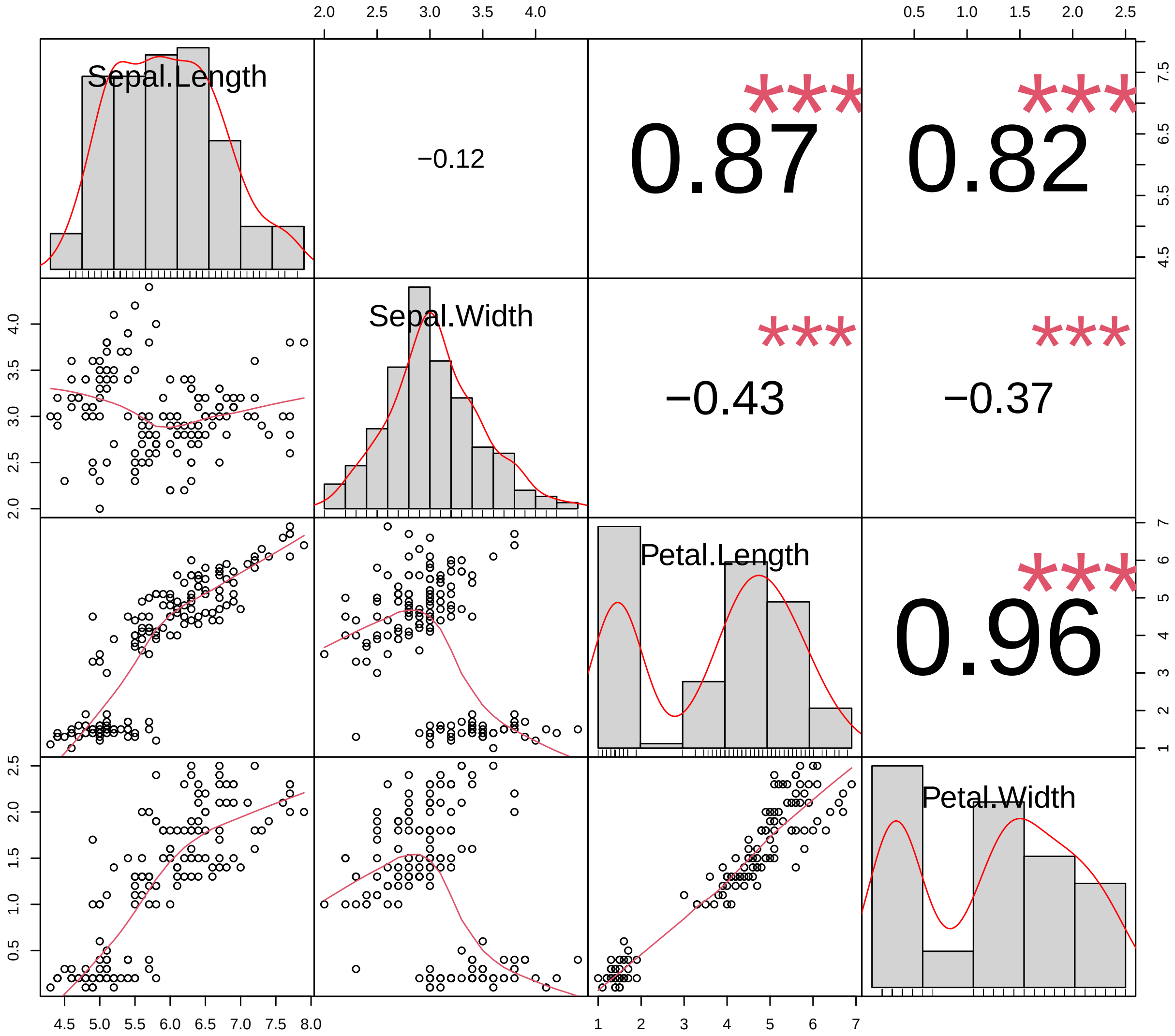


Data frame: .

ID	*Name*	*Label*	*Values*	*Value Labels*
1	numeric	numeric variable	*range: 1-3*	
2	factor_numeric	numeric factor with missing range		strongly disagree disagree neither agree nor disagree agree strongly agree
3	factor_n_coded_miss	numeric factor with coded missing values		strongly disagree disagree neither agree nor disagree agree strongly agree no answer
4	date	date format tt.mm.yyyy		

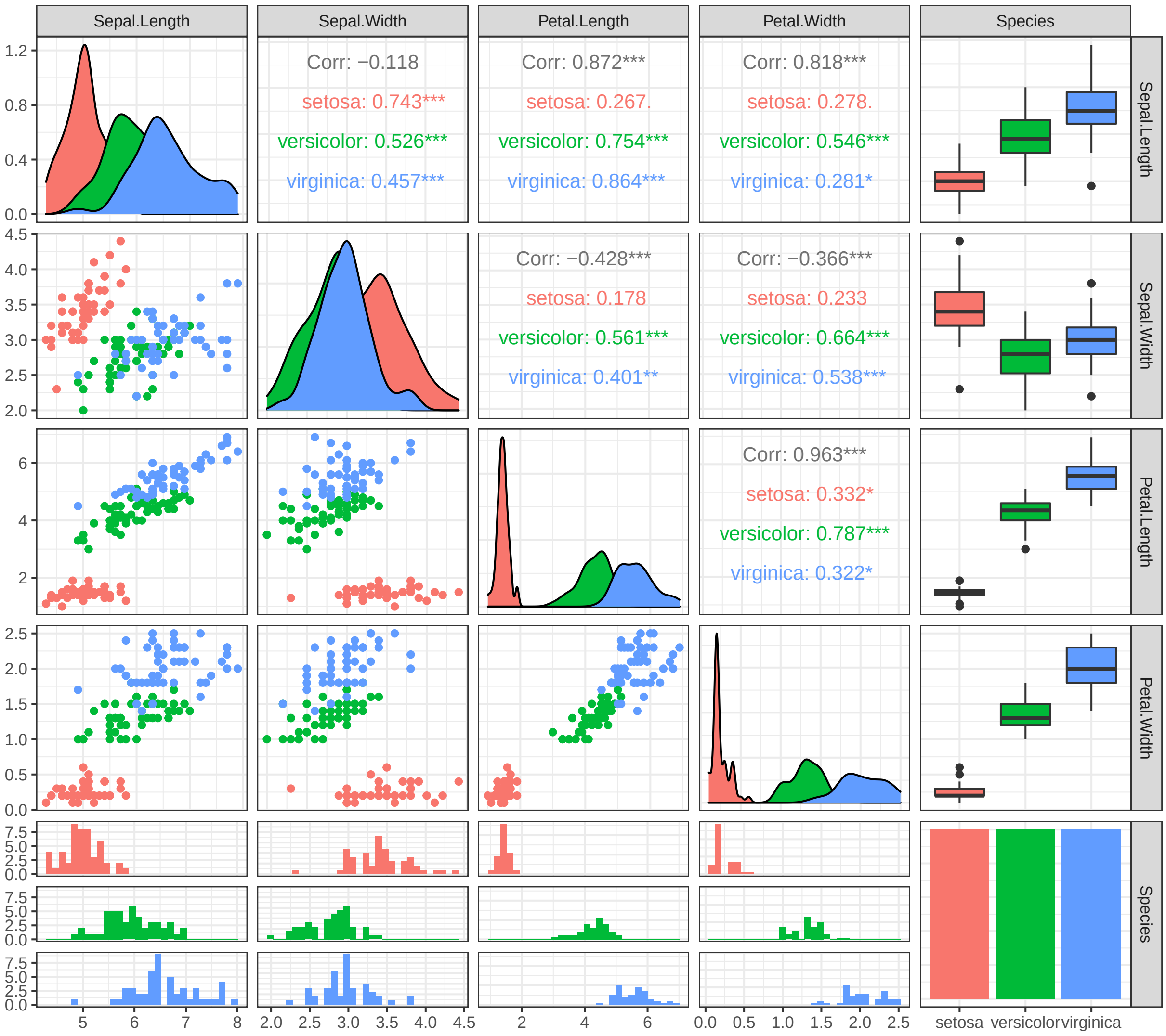
You might also need to check multivariate associations briefly to check if there were any variables were coded in the wrong direction (e.g., some psychological instruments requires reverse coding). This can be achieved via simply checking, scatter plots and correlations.

```r
library("PerformanceAnalytics")
PerformanceAnalytics::chart.Correlation(iris[1:4],
    histogram = TRUE, pch = 19)
```



ggpairs is an interesting library I found recently, which provides combined multivariate plots.

```
library(GGally)
GGally::ggpairs(iris, aes(color = Species)) +
    theme_bw()
```



There are many other types of plots displaying multivariate associations, such as heatmaps, mosaic plots, flow diagrams etc. Interested readers can find more examples in Data Visualization with R by Rob Kabacoff.

Data cleaning

After the data is properly organised, you will need to start the data cleaning phase. The total workload depends on the type of source data received. If you are lucky that the data is collected using a more advanced tool such as Redcap (designed by a competent data manager), the process may not be substantially tedious. Still, generally data scientists and statisticians spend 60%-80% of their time in data cleaning and pre-processing. So be prepared!!

Another issue with data cleaning is that there is no clear definition of what is a “cleaned” data. Hadley Wickham provided a more general definition of tidy vs messy data https://vita.had.co.nz/papers/tidy-data.pdf. This is my definition of cleaned data related to health studies:

A cleaned data is one dataset with (i) variables validated, ordered, clearly named and labelled, (ii) categorical variables properly categorized with clear labels, (iii) derived and transformed variables properly included, (iv) and the required format (by the analysis) established.

"Validated" refers to a range of aspect including: consistency (male participants should not report menstrual cramps in PHQ15), validity (poor quality tests sometimes needed to be excluded, should be coded as missing, data out of range should not be included), variable type (numeric variables should not include strings e.g., <5), uniformity (using the same (unit of measure)

It will be better to store related variables closer in the dataset so it is easier to check. Sometimes variable naming can be problematic and confusing. For example, when individual items in K10 is named as K1, K2... K10, then the last item name is overlapping with the general understanding of K10, which represents the summary score of K10.

Categorical variables are sometimes stored as numeric variables (1,2,3,999), which would need to be labelled as factor variables. Sometimes there is an "other" category that affiliate with a free-text field. Often what was entered will need to be re-evaluated and coded back to existing categories. Categorical variables from source data often includes much more detailed categories than needed for analysis. It's better to create consistent categorisations. For example, gender identity can sometimes be collected as male, female, transgender, gender neutral, agender, pangender, genderqueer etc. Most of these categories would include very small numbers that cannot be used in the analysis. Then it will be a good practice to generate another shorter version of the gender identity variable in the cleaned data (e.g., male, female and non-binary). Then this variable can be used consistently across different analyses.

In psychology, lots of measures were conducted using instruments with multiple items. Hence derived variables (e.g., total scores, weighted scores) will need to be generated in the cleaned data.

The "format of the data" refers to the format of data required for different types of analysis, e.g., cross-sectional analysis requires a wide version of data (one row per person); longitudinal analysis requires a long version of data (multiple rows per person with each row representing one observation with unique identifiers for each person or a higher level of clustering group); survival analysis requires censored data (one or multiple rows per person depending on whether evaluating single outcome or recurrent outcomes with time scale and censoring status defined). Sometimes you will need to have both the wide and long version of cleaned data stored.

More often your final “cleaned” data will change over time for most of larger studies due to issues identified when the study evolves, e.g., additional errors identified, more data collected etc. Hence you will need to create a version control system to capture these changes. A common practice is to store the “cleaned” versions of data with a timestamp, e.g., “A_brilliant_study_cleaned_V1_03032012.rds”.

Tidyverse data cleaning routine

The data cleaning code pre Hadley Wickham era normally look like this:

```
data$group <- as.factor(data$group)
data$time <- as.factor(data$time)
data$gender <- as.factor(data$gender)
data$work_status <- as.factor(data$work_status)
data$diagnosis <- as.factor(data$diagnosis)
data$therapy <- as.factor(data$therapy)
```

The operation this chunk of code was trying to achieve was simply converting a few variables from string to factor. When there are only a few variables to edit, this type of code is still readable. As the number of variables and operations increases, you code will soon be too hard to read and amend. Hadley introduced a game-changing new philosophy: **tidyverse**. This new philosophy defines not only what the tidy data should look like, but also what tidy code should look like. Using tidyverse grammar (using *dplyr* package), the above data cleaning code became:

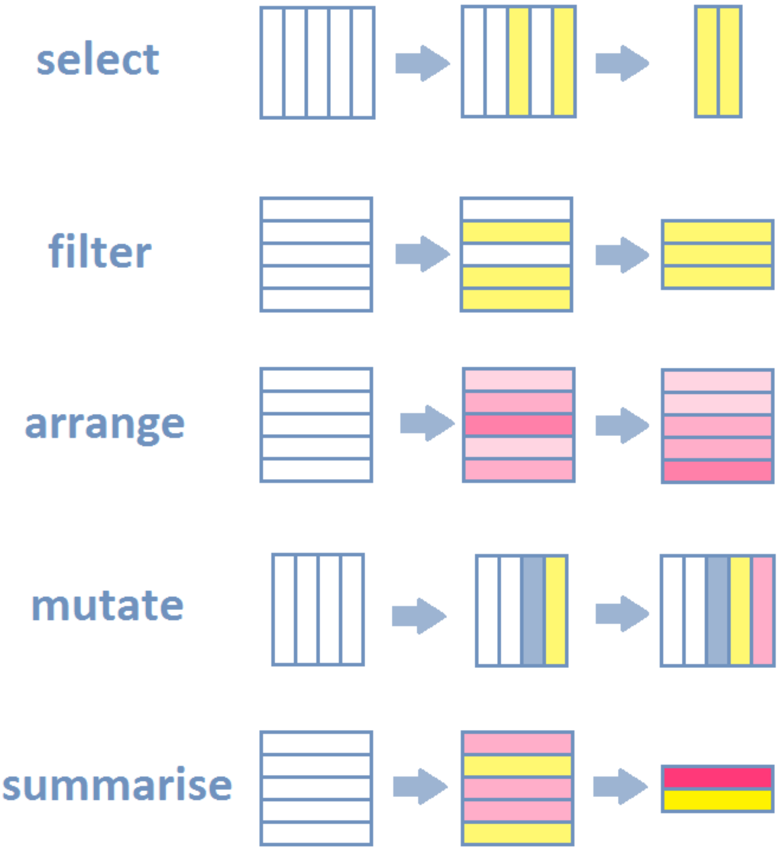
```
data1 <- data %>% mutate_at(vars(group, time,
    gender, work_status, diagnosis, therapy),
    as.factor)
```

How to read this is simple, I am manipulating 6 variables (changing them to factor variables) in the *data* dataset and store the new dataset in *data1*. The code is now much more concise and readable. The beauty of this new style also lie on the use of pipe operation %>% in building a series of operations on the same dataset.

```
data1 <- data %>% mutate_at(vars(group, time,
    gender, work_status, diagnosis, therapy),
    as.factor) %>% mutate_at(vars(starts_with("k10_q")),
    as.numeric) %>% mutate(k10_total = rowSums(select(.,
    k10_q1:k10_q10)))
```

The above code change 6 variables to factor variables, change individual K10 items to numeric and then calculate the total score of K10.

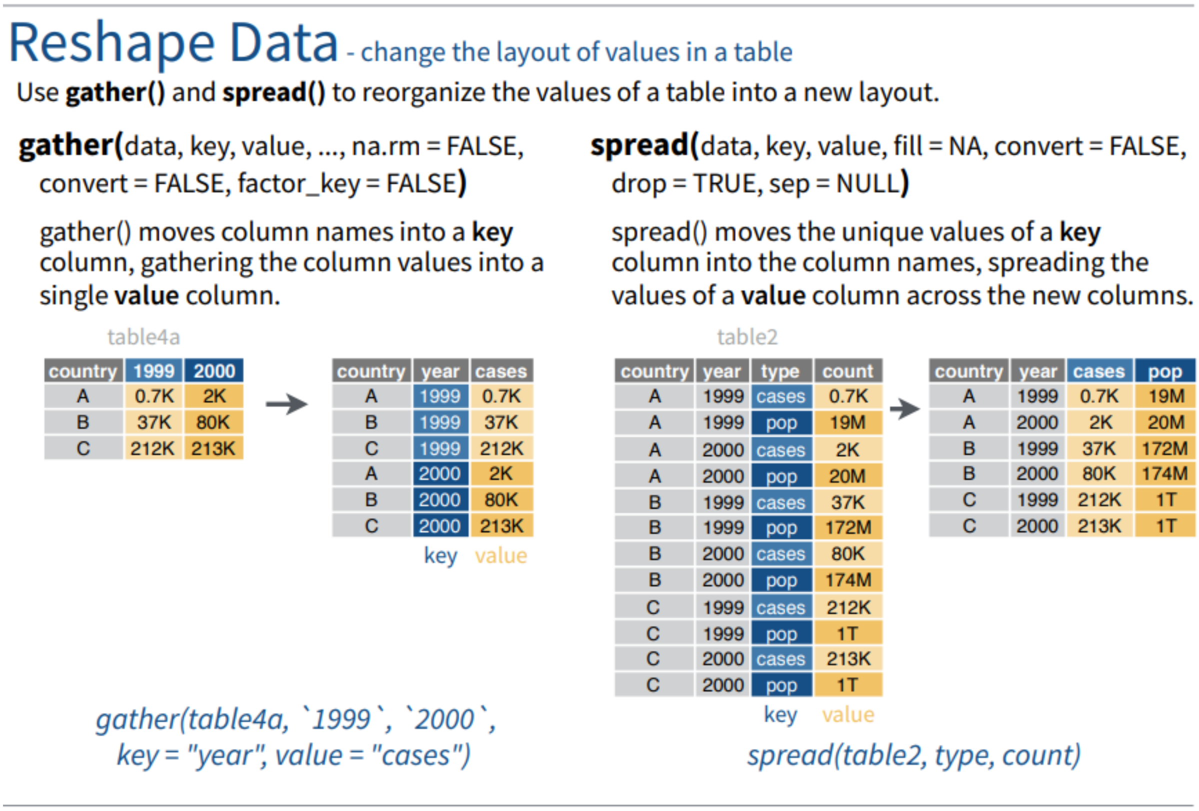
There are a range of useful functions in dplyr such as, group_by(), filter(), select(),and summarize(). The detailed introduction on *dplyr* can be found in [Chapter 4 of Introduction to Data Science] (https://rafalab.github.io/dsbook/tidyverse.html) by Hadley and *dplyr* cheat sheet If you are still using the traditional "hard" coding style, it is time to upgrade!



source: http://perso.ens-lyon.fr/lise.vaudor/dplyr/

Re-shaping the data

It is common to have to work with both wide(or wider) and long(or longer) formats of the data during your data cleaning process. You are also frequently required to change data between these two types, see an example of reshaping using *gather()* and *spread()* from *tidyr* package below:



Reshape Data - change the layout of values in a table

Use **gather()** and **spread()** to reorganize the values of a table into a new layout.

gather(data, key, value, ..., na.rm = FALSE, convert = FALSE, factor_key = FALSE**)**

gather() moves column names into a **key** column, gathering the column values into a single **value** column.

table4a

country	1999	2000
A	0.7K	2K
B	37K	80K
C	212K	213K

→

country	year	cases
A	1999	0.7K
B	1999	37K
C	1999	212K
A	2000	2K
B	2000	80K
C	2000	213K

key value

gather(table4a, `1999`, `2000`, key = "year", value = "cases")

spread(data, key, value, fill = NA, convert = FALSE, drop = TRUE, sep = NULL**)**

spread() moves the unique values of a **key** column into the column names, spreading the values of a **value** column across the new columns.

table2

country	year	type	count
A	1999	cases	0.7K
A	1999	pop	19M
A	2000	cases	2K
A	2000	pop	20M
B	1999	cases	37K
B	1999	pop	172M
B	2000	cases	80K
B	2000	pop	174M
C	1999	cases	212K
C	1999	pop	1T
C	2000	cases	213K
C	2000	pop	1T

key value

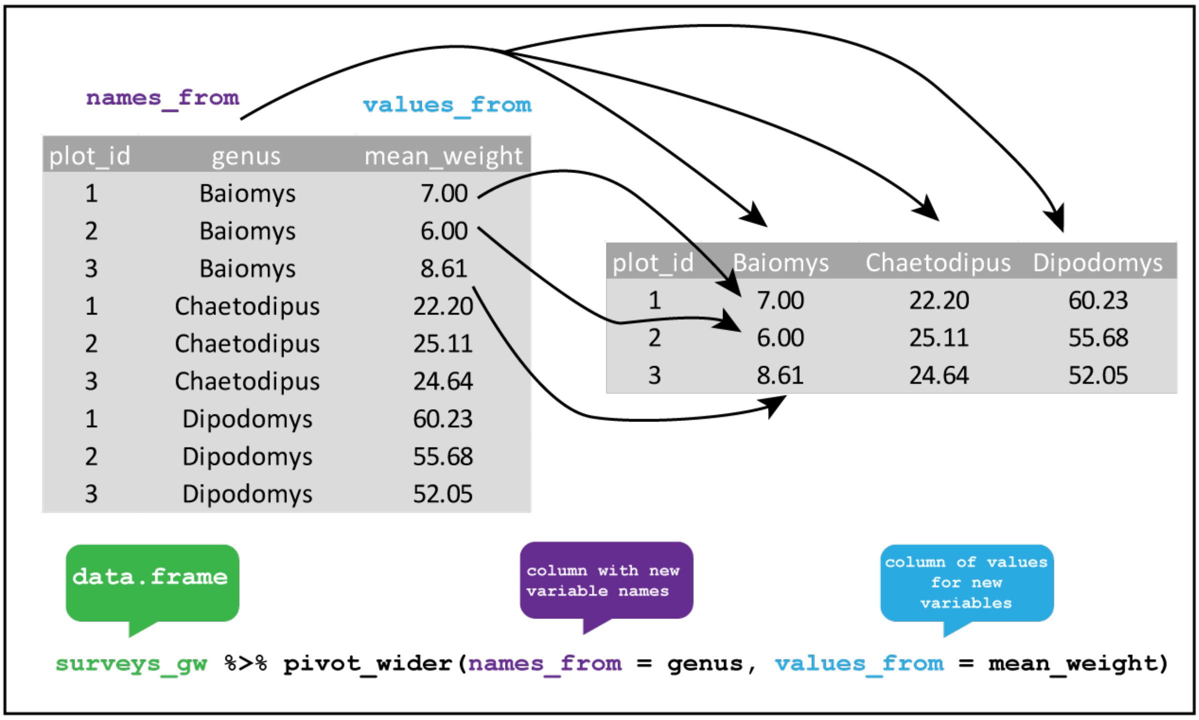
→

country	year	cases	pop
A	1999	0.7K	19M
A	2000	2K	20M
B	1999	37K	172M
B	2000	80K	174M
C	1999	212K	1T
C	2000	213K	1T

spread(table2, type, count)

source: https://dannytabach.com/connecting-r-and-google-sheets.html

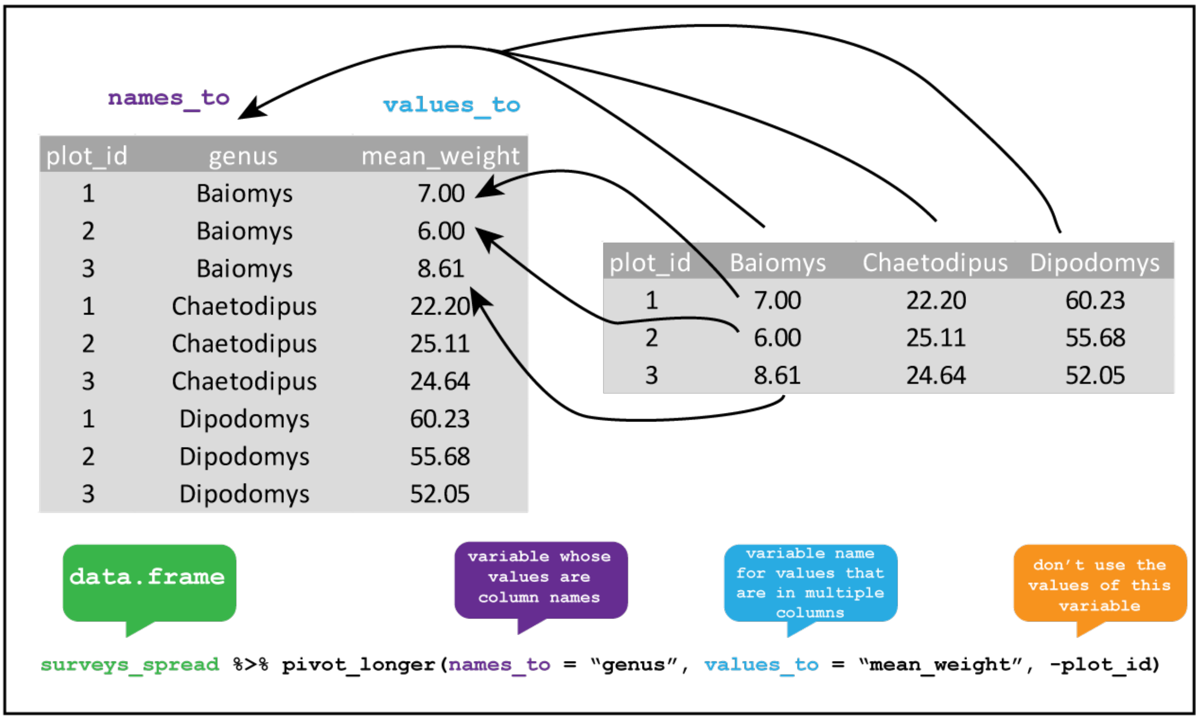
You can also use the *reshape* function from the *stats* package which can work with multiple columns. Two new functions in *tidyr* can also work with multiple columns and rows.


names_from
values_from
data.frame
column with new variable names
column of values for new variables

plot_id	genus	mean_weight
1	Baiomys	7.00
2	Baiomys	6.00
3	Baiomys	8.61
1	Chaetodipus	22.20
2	Chaetodipus	25.11
3	Chaetodipus	24.64
1	Dipodomys	60.23
2	Dipodomys	55.68
3	Dipodomys	52.05

plot_id	Baiomys	Chaetodipus	Dipodomys
1	7.00	22.20	60.23
2	6.00	25.11	55.68
3	8.61	24.64	52.05

```
surveys_gw %>% pivot_wider(names_from = genus, values_from = mean_weight)
```


names_to
values_to
data.frame
variable whose values are column names
variable name for values that are in multiple columns
don't use the values of this variable

plot_id	genus	mean_weight
1	Baiomys	7.00
2	Baiomys	6.00
3	Baiomys	8.61
1	Chaetodipus	22.20
2	Chaetodipus	25.11
3	Chaetodipus	24.64
1	Dipodomys	60.23
2	Dipodomys	55.68
3	Dipodomys	52.05

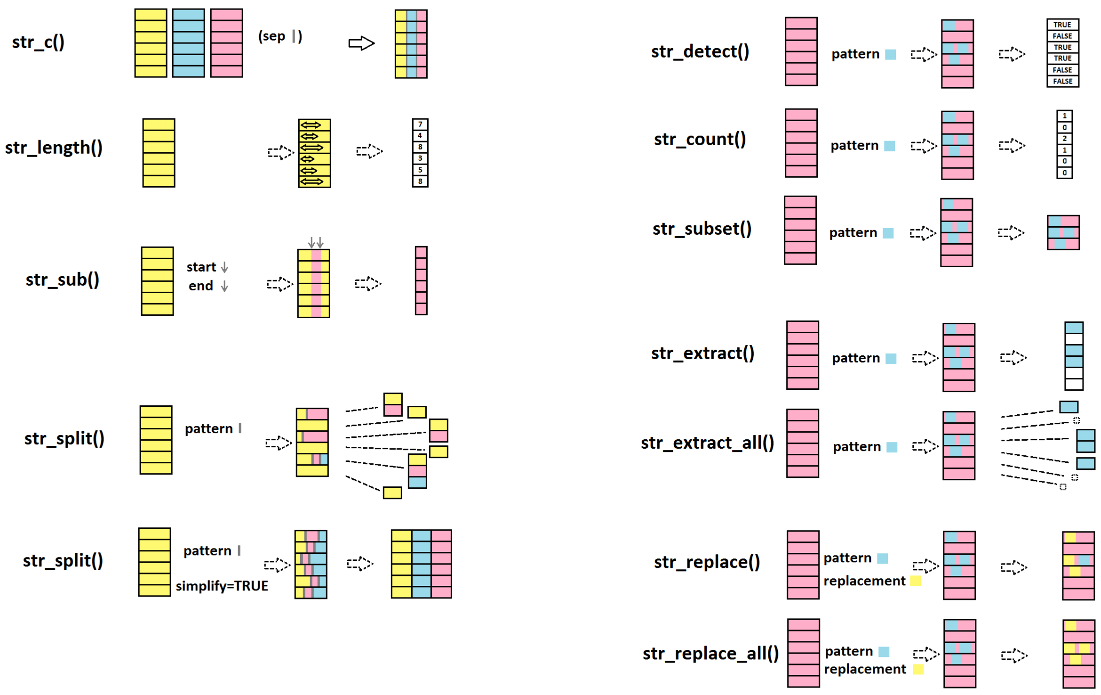
plot_id	Baiomys	Chaetodipus	Dipodomys
1	7.00	22.20	60.23
2	6.00	25.11	55.68
3	8.61	24.64	52.05

```
surveys_spread %>% pivot_longer(names_to = "genus", values_to = "mean_weight", -plot_id)
```

source: https://ab604.github.io/docs/coding-together-2019/data-wrangle-2.html

Other useful packages

String operation package *stringr*



Source: Manipuler des strings avec R

Package working with date and time variables *lubridate*. See tutoria here

Apply family, see tutorial here

purrr for functional programming

janitor package assist with data cleaning

Notes for yourself and others

There is a famous quote about writing programming code

"When I wrote this code, only God and I understood what it did. Now only God knows"

The quote was originally from *The Barretts of Wimpole Street* when Robert Browning was trying to work out what he meant in a poem. However it is the reality for a vast majority of programmers, data scientists and statisticians.

I have developed a respiratory infection transmission model during my PhD time as an early R user. It was only published in my PhD thesis not in scientific publications due to lack of interest in airborne disease transmission. Then COVID19 happened, and we thought it will be important to publish this work in a reproducible format. However, my code at that time was so terrible that it took me half a year to figure out what I did in 3 months. I had excuses back then, e.g., stress to finish the work on time, difficulties in including equations in code comments… Now my code is slightly better with my following rules:

- Always use R Markdown (or R Notebook) file for data cleaning
- Document decisions in text field of the R Markdown file
- Include specific code comments alongside the code

The modern version of my data cleaning code looks like this:

Calculate total scores and obtain OASIS categories using cut-off value of >=8 following Moore et al 2015.

```
OASIS <- OASIS %>% # calculate total score
mutate(oasis_total = rowSums(select(., OASISq1:OASISq5))) %>%
    # define groups
mutate(oasis_total_group = cut(oasis_total, breaks = c(-1,
    7, Inf), labels = c("Absence of anxiety disorder ",
    "Presence of anxiety disorder"))) %>% # remove useless columns
select(-c(AxDate, status)) %>% # rename variables for consistency
```

```
setnames(old = paste0("OASISq", 1:5), new = paste0("oasis_q",
    1:5))

# check
table(OASIS$oasis_total, OASIS$oasis_total_group,
    exclude = NULL)
```

Code checking

It's important to cross-check your data cleaning code. The common self-checking points are listed below:

- When a new variable is generated based on an old variable, check whether it is correct via cross-tabulate it with the original variable.
- When performing merging datasets, always check if you have correctly matched records, whether the total number of rows is correct
- When calculating total scores, choose to eyeball a few records to see if it is correct.
- Cross check you final data with raw data (look at the Hmisc::describe() of the raw data and the updated data side by side)
- When changing the data between long and wide data types, manually check if the number of rows is as expected (e.g., 1000 participants completed a baseline survey and 500 completed a follow-up, the long data should have 1500 rows).
- Whether nested variables are coded correctly. If you have a drinking quantity variable that was only asked when people reported drinking. Then you should code the quantity as 0 if they did not drink (unless you have a good reason not to). This is because sometimes you might accidentally exclude observations if these nested variables were coded as missing. Also if you use multiple imputation, coding them to missing (intentionally missing) will also cause problems.

Common pitfalls

Missing value

Missing values are coded as *NA* in R data. See instructions on dealing with R missing values here and here When tabulating data always use *exclude=NULL*, or you can use tidyverse style function *tabyl* from *janitor* package which can also store cross-tabulate results in a data frame.

```
data <- tibble(a = c(1, 2, 3, 4), b = c(4, NA,
    5, 6), c = c(7, 8, 9, NA))
table(data$a, data$b, exclude = NULL)
```

```
## 
##      4 5 6 <NA>
##    1 1 0 0    0
##    2 0 0 0    1
##    3 0 1 0    0
##    4 0 0 1    0
```

```
library(janitor)
data %>% tabyl(a, b)
```

```
##  a 4 5 6 NA_
##  1 1 0 0   0
##  2 0 0 0   1
##  3 0 1 0   0
##  4 0 0 1   0
```

When using rowSums and rowMeans functions, you have to be mindfull of missing data, see following example.

```
data %>% mutate(sum = rowSums(., na.rm = FALSE),
    mean = rowMeans(., na.rm = FALSE))
```

```
## # A tibble: 4 x 5
##       a     b     c   sum  mean
##   <dbl> <dbl> <dbl> <dbl> <dbl>
## 1     1     4     7    12  4
## 2     2    NA     8    NA NA
## 3     3     5     9    17  5.67
## 4     4     6    NA    NA NA
```

```
data %>% mutate(sum = rowSums(., na.rm = TRUE),
    mean = rowMeans(., na.rm = TRUE))
```

```
## # A tibble: 4 x 5
##       a     b     c   sum  mean
##   <dbl> <dbl> <dbl> <dbl> <dbl>
## 1     1     4     7    12  4
## 2     2    NA     8    10  5
## 3     3     5     9    17  5.67
## 4     4     6    NA    10  5
```

Duplication

Sometimes there are duplicated records in the database or being created in the data pre-processing stage. This is particularly problematic when you are working with large datasets from a secondary source. You have to be mindful to keep track of the number of observations throughout the data cleaning code.

The trap of white space

R string fields sometimes include leading and trailing white spaces, which you cannot see when *View()* the data. However, this will cause issues when you are trying to categorise or merge the data, see following example

```
data1 <- tibble(key = c("a", "b", "c ", "d"),
    number_1 = c(1, 2, 3, 4))
data2 <- tibble(key = c("a", "b", "c"), number_2 = c(5,
    6, 7))
data1 %>% left_join(data2)
```

```
## Joining, by = "key"
```

```
## # A tibble: 4 x 3
##   key   number_1 number_2
##   <chr>    <dbl>    <dbl>
## 1 "a"          1        5
## 2 "b"          2        6
## 3 "c "         3       NA
## 4 "d"          4       NA
```

The *str_trim* function removes the leading and trailing white space, which will make sure the data can be joined correctly.

```
data1 %>% mutate(key = str_trim(key)) %>% left_join(data2)
```

```
## Joining, by = "key"
```

```
## # A tibble: 4 x 3
##   key   number_1 number_2
##   <chr>    <dbl>    <dbl>
## 1 a            1        5
## 2 b            2        6
## 3 c            3        7
## 4 d            4       NA
```

Be mindfull of which function you are calling

One of the issue with R is that a few functions can have the same name. Here are the list of conflicts in my attached libraries.

```
library(conflicted)
conflict_scout()
```

```
## 11 conflicts:
## * `as.Date`        : [zoo]
## * `as.Date.numeric`: [zoo]
## * `chisq.test`     : janitor, stats
## * `filter`         : [dplyr]
## * `first`          : xts, dplyr
## * `fisher.test`    : janitor, stats
## * `group_rows`     : kableExtra, dplyr
## * `lag`            : dplyr, stats
## * `last`           : xts, dplyr
## * `legend`         : PerformanceAnalytics, graphics
## * `Position`       : ggplot2, base
```

R's default conflict resolution system gives precedence to the most recently loaded package. So when calling the functions, it will be better to refer to the package name (*dplyr::summarize()* vs *Hmisc::summarize()*).

Type stability

This is referred as the "WAT" moment with R programming

```
# combining factors makes integers
c(factor("a"), factor("b"))
```

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

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

Be careful with R's automatic coercion.

Data integrity checking

Data integrity refers to the reliability and trustworthiness of data. There are many reasons that could cause data integrity issues, such as human errors in data entry, issues with data collection platform, and errors in data cleaning process. It's important to check the data validity and integrity before analysing the data.

Documentation for the never-ending data cleaning process

Documentation is the key to good data quality. Historically, studies are small and isolated. Once data cleaning is completed (mostly via eyeballing), you will conduct the analysis and the data files (on paper) will be sealed in a draw and no one will be looking at them. Nowadays, we often work with large and evolving datasets, e.g., longitudinal, administrative and registry data. Hence it is important to keep good documentation that summarises important decisions, data errors identified, changes in the data cleaning process, version history, and associated locations where datasets and historical data cleaning files are stored. I often use a separate word document to keep track of these issues. When there are multiple users using the "cleaned" data, it is also important to keep a data dictionary of the cleaned data, in which you note down, the time of the variable introduced to the dataset, any changes in coding and cleaning to the variable as well as other important notice that you would like your users to know.

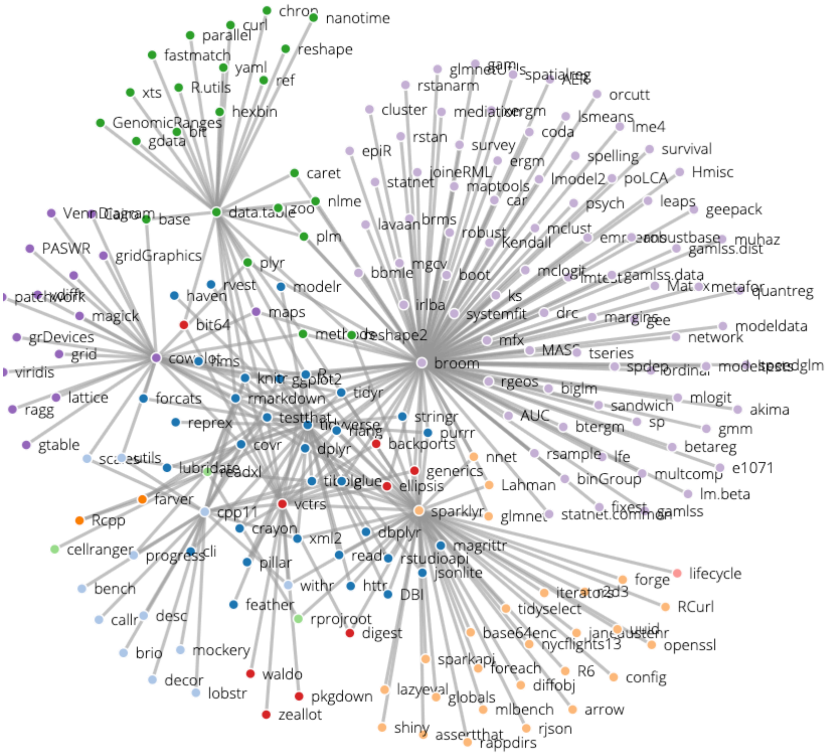
Final remarks

There are a few additional key points that need to be noted for data pre-processing:

- Store backup files (backup original data, intermediate "cleaned" versions of the data and cleaning files) frequently in reliable place(s) so you can uncover historical issues easily. You can zip the files to prevent people from accidentally changing the files or store the backup files in locations with limited and authorised access.
- Establish a code auditing workflow. If possible, have all the code and final data reviewed by a second person.

Statistical analysis with R

R has over 20K indexed packages, over 3M indexed functions and over 3m users worldwide. So R has an explosive level of capacity and flexibility as well as a tremendous workforce. R is now equipped with all or most of the functionalities of many other statistical/ML packages: Stata, SPSS, SAS, Mplus etc (Python is still irreplaceable in ML/deep learning).



Source: https://www.rdocumentation.org/trends

A few common analyses and packages are listed here

Analysis	Function and package
Linear regression	stats::lm
Generlised linear regression	stats::glm
Generalized additive model	mgcv::gam
ARIMA (time series)	forecast::arima
Cox proportional hazards (survival analysis)	survival::coxph
Linear mixed effect	lme4::lmer
Generalised linear mixed effect	lme4::glmer
CFA/SEM	lavaan::cfa, lavaan::sem
Meta-regression	metafor::rma , meta::metareg
Clustering	stats::hclust, stats::kmeans, cluster::pam
Principal component analysis	stats::princomp

One of the best books for learning statistical methods using R is An Introduction to Statistical Learning by Gareth James, Daniela Witten, Trevor Hastie and Robert Tibshirani. The advanced version of this book is The Elements of Statistical Learning by Trevor Hastie, Robert Tibshirani and Jerome Friedman. However, both of the books are from more machine learning (ML) perspectives. A similar free book for psychology students is Advanced Statistics Using R by Zhiyong Zhang and Lijuan Wang. Discovering Statistics Using R by Andy Field is also one of the best R statistical handbooks. There are many content-specific books that are quite interesting and useful:

- An Introduction to Generalized Linear Models by Annette J. Dobson, Adrian G. Barnett. R code were given in some examples
- Forecasting: Principles and Practice by Rob J Hyndman and George Athanasopoulos for time-series analysis
- Applied Survival Analysis Using R by Dirk F. Moore
- Statistical Rethinking A Bayesian Course with Examples in R and Stan by Richard McElreath. Associated lectures can be found here
- Geocomputation with R by Robin Lovelace, Jakub Nowosad, Jannes Muenchow
- Modern Psychometrics with R by Patrick Mair
- Multilevel Modeling Using R by W. Holmes Finch, Jocelyn E. Bolin, Ken Kelley

Exploratory data analysis (EDA)

Once your data is cleaned, you will normally start with the "exploratory" analysis phase. The "exploratory" here refers to evaluating variable properties, such as distribution, bivariate association and dimensionality, rather than exploring "significant" results. More often exploratory results, such as distributions, will also need to be included in your report/thesis/paper. This process provides you with general ideas about your dataset, clues on appropriateness of models (Poisson regression vs negative binomial regression), check outliers, identify any inconsistencies with your understanding (I call this debugging your data).

Exploratory Data Analysis with R by Roger Peng includes lots of useful examples. Chapter 7 of R for Data Science also includes a nice intro to EDA. Importantly, you should not use EDA to generate research questions. Your research questions should be pre-defined and your analysis will need to be planned ahead. EDA can help you refine models, but you should **not** look too hard into it, which might turn your years of work into another piece of "fished" or "hacked" results out of null association (winner's curse). See a few interesting publications here: The Extent and Consequences of P-Hacking in Science and The garden of forking paths: Why multiple comparisons can be a problem, even when there is no "fishing expedition" or "p-hacking" and the research hypothesis was posited ahead of time.

Table 1

A simple descriptive table, normally referred to as "Table 1", is where your data story start. In this table, you will need to provide enough information about the data that was used in then analyses so your readers can have an overview of its profiles. For most of the health analysis, your reporting unit is participants and you would generally be required to report demographic characteristics and risk factors in this table. R has lots of existing packages that automatically generate table 1. *tableby* from *arsenal* is very easy to use when you directly display your results. However, it is difficult to be extracted to tibble/data frame for further manipulations.

```r
library(arsenal)
my_controls <- tableby.control(test = T, total = F,
    digits = 1, digits.pct = 1, digits.p = 3,
    numeric.test = "anova", cat.test = "chisq",
    numeric.stats = c("meansd", "medianq1q3",
        "range", "Nmiss2"), cat.stats = c("countpct",
        "Nmiss2"), stats.labels = list(meansd = "Mean (SD)",
        medianq1q3 = "Median (Q1, Q3)", range = "Min - Max",
        Nmiss2 = "Missing"))
table <- tableby(Species ~ Sepal.Length + Sepal.Width +
    Petal.Length + Petal.Width, data = iris, control = my_controls)

summary(table, title = "Characteristics by species in %>% data")
```

Table 1: *Characteristics by species in %>% data*

	setosa (N=50)	versicolor (N=50)	virginica (N=50)	p value
Sepal.Length				< 0.001
Mean (SD)	5.0 (0.4)	5.9 (0.5)	6.6 (0.6)	
Median (Q1, Q3)	5.0 (4.8, 5.2)	5.9 (5.6, 6.3)	6.5 (6.2, 6.9)	
Min - Max	4.3 - 5.8	4.9 - 7.0	4.9 - 7.9	
Missing	0	0	0	
Sepal.Width				< 0.001
Mean (SD)	3.4 (0.4)	2.8 (0.3)	3.0 (0.3)	
Median (Q1, Q3)	3.4 (3.2, 3.7)	2.8 (2.5, 3.0)	3.0 (2.8, 3.2)	
Min - Max	2.3 - 4.4	2.0 - 3.4	2.2 - 3.8	
Missing	0	0	0	
Petal.Length				< 0.001
Mean (SD)	1.5 (0.2)	4.3 (0.5)	5.6 (0.6)	
Median (Q1, Q3)	1.5 (1.4, 1.6)	4.3 (4.0, 4.6)	5.5 (5.1, 5.9)	
Min - Max	1.0 - 1.9	3.0 - 5.1	4.5 - 6.9	
Missing	0	0	0	
Petal.Width				< 0.001
Mean (SD)	0.2 (0.1)	1.3 (0.2)	2.0 (0.3)	
Median (Q1, Q3)	0.2 (0.2, 0.3)	1.3 (1.2, 1.5)	2.0 (1.8, 2.3)	
Min - Max	0.1 - 0.6	1.0 - 1.8	1.4 - 2.5	
Missing	0	0	0	

Another useful Table 1 function is *tbl_summary* from *gtsummary* , which you can extract results as tibble.

```
library(gtsummary)
```

```
## Warning: package 'gtsummary' was built under R version 4.0.5
```

```
## #BlackLivesMatter
```

```
tbl <- iris %>% tbl_summary(missing = "ifany",
    statistic = where(is.numeric) ~ "{mean} ({sd})",
    by = Species) %>% bold_labels() %>% add_p(test = list(all_numeric() ~
    "aov"), pvalue_fun = ~style_pvalue(.x, digits = 3))
```

```
## Warning: `all_numeric()` was deprecated in gtsummary 1.3.6.
## The {tidyselect} and {dplyr} packages have implemented functions to select variables by
##
## Use `where(is.numeric)` instead.
## This warning is displayed once every 8 hours.
## Call `lifecycle::last_lifecycle_warnings()` to see where this warning was generated.
```

```
tbl
```

```
## Table printed with `knitr::kable()`, not {gt}. Learn why at
## https://www.danieldsjoberg.com/gtsummary/articles/rmarkdown.html
## To suppress this message, include `message = FALSE` in code chunk header.
```

Characteristic	**setosa**, N = 50[1]	**versicolor**, N = 50[1]	**virginica**, N = 50[1]	**p-value**[2]
Sepal.Length	5.01 (0.35)	5.94 (0.52)	6.59 (0.64)	<0.001
Sepal.Width	3.43 (0.38)	2.77 (0.31)	2.97 (0.32)	<0.001
Petal.Length	1.46 (0.17)	4.26 (0.47)	5.55 (0.55)	<0.001
Petal.Width	0.25 (0.11)	1.33 (0.20)	2.03 (0.27)	<0.001

[1] Mean (SD)

[2] One-way ANOVA

gtsummary object can be extracted as tibble and further edited.

```
tbl <- tbl %>% as_tibble()


# remove style
names(tbl) <- str_replace_all(names(tbl), "\\**",
    "")
```

```
indent <- which(str_count(tbl$Characteristic,
    "__") == 0)
tbl$Characteristic <- str_replace_all(tbl$Characteristic,
    "__", "")

# change first row
tbl$Characteristic <- str_remove(tbl$Characteristic,
    "Sepal.")
tbl$Characteristic <- str_remove(tbl$Characteristic,
    "Petal.")
tbl <- tbl %>% add_row(Characteristic = "Sepal",
    .before = 1) %>% add_row(Characteristic = "Petal",
    .before = 4)

options(knitr.kable.NA = "")
knitr::kable(tbl, booktabs = TRUE, linesep = "") %>%
    kable_styling(bootstrap_options = "striped") %>%
    add_indent(c(2, 3, 5, 6))
```

Characteristic	setosa, N = 50	versicolor, N = 50	virginica, N = 50	p-value
Sepal				
Length	5.01 (0.35)	5.94 (0.52)	6.59 (0.64)	<0.001
Width	3.43 (0.38)	2.77 (0.31)	2.97 (0.32)	<0.001
Petal				
Length	1.46 (0.17)	4.26 (0.47)	5.55 (0.55)	<0.001
Width	0.25 (0.11)	1.33 (0.20)	2.03 (0.27)	<0.001

Regression models

Regression model follows very similar style of coding:

function(y~ x1 + x2 + x3, data=data, additional specifications)

```
mod1 <- lm(Petal.Length ~ Species + Sepal.Width *
    Sepal.Length, data = iris)
summary(mod1)
```

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

```
## lm(formula = Petal.Length ~ Species + Sepal.Width * Sepal.Length,
##     data = iris)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -0.74506 -0.18607  0.00386  0.16942  0.79286
##
## Coefficients:
##                          Estimate Std. Error t value Pr(>|t|)
## (Intercept)              -2.14865    1.44426  -1.488  0.13901
## Speciesversicolor         2.18252    0.11214  19.462  < 2e-16 ***
## Speciesvirginica          3.06260    0.12854  23.825  < 2e-16 ***
## Sepal.Width               0.13127    0.48106   0.273  0.78535
## Sepal.Length              0.73177    0.24183   3.026  0.00294 **
## Sepal.Width:Sepal.Length -0.02912    0.08035  -0.362  0.71754
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.2841 on 144 degrees of freedom
## Multiple R-squared:  0.975,  Adjusted R-squared:  0.9741
## F-statistic:  1121 on 5 and 144 DF,  p-value: < 2.2e-16
```

```
# 'Sepal.Width*Sepal.Length' is the
# interaction term.
```

Most of the regression models are easy to fit. The following lists are some of known issues and concerns for common models (some are not specific to R):

- Some GLMs, particularly Relative Risk regression also known as log-binomial regression (GLM with log link and binomial distribution model), have convergence problems (Williamson, Eliasziw, and Fick 2013). This is related to parameter constraints at boundaries. For log-binomial regression, a package *logbin* was developed with advanced convergence ability (use combinatorial EM to cycles through parameter space). For other GLMs, Bayesian models (e.g., using *brms*) can be used to overcome some of the convergence problems.

- Linear mixed-effect models sometimes have convergence issues too. You can test the performance of different optimisers, see the example here.
- Multinomial logistic regression model *nnet::multinom* requires predictors to be roughly scaled to [0,1] or the fit will be slow or may not converge at all. Normally this is fine, however, when you have some large values (e.g., IRSAD scores which is around 1000), the model will stop working. Hence it will be better to standardise those predictors.
- Sometimes there will be an error message "Error: no valid set of coefficients has been found: please supply starting values", this means the model is having trouble initialising the optimization and require you to supply a set of starting values for intercept and predictors. I normally run a model with a reduced number of predictors and use the coefs from the reduced model as starting values.
- Set seed for models requires random sampling, e.g., bootstrap.
- It will be better to use splines (e.g., *ns*) instead of square and cube terms to evaluate non-linear associations, see a review by Perperoglou et al. (2019).
- Different packages will not necessarily produce identical results, but they should generally agree with each other. If not something is wrong: either a bug in your code or in the existing code libraries.

```r
library(logbin)
iris$Versicolor <- as.numeric(iris$Species ==
    "versicolor")
mod1 <- glm(Versicolor ~ Sepal.Length, family = binomial(link = "log"),
    data = iris)
mod2 <- logbin::logbin(formula(mod1), data = iris,
    method = "cem", trace = TRUE)
```

```
## logbin parameterisation 1/2
## Deviance = 190.9543 Iterations - 432
## logbin parameterisation 2/2
## Deviance = 190.2446 Iterations - 613
```

```r
coefficients(summary(mod1))
```

```
##                Estimate Std. Error    z value   Pr(>|z|)
## (Intercept)  -1.6925062  0.8234556 -2.0553703 0.03984325
## Sepal.Length  0.1008402  0.1370450  0.7358181 0.46184139
```

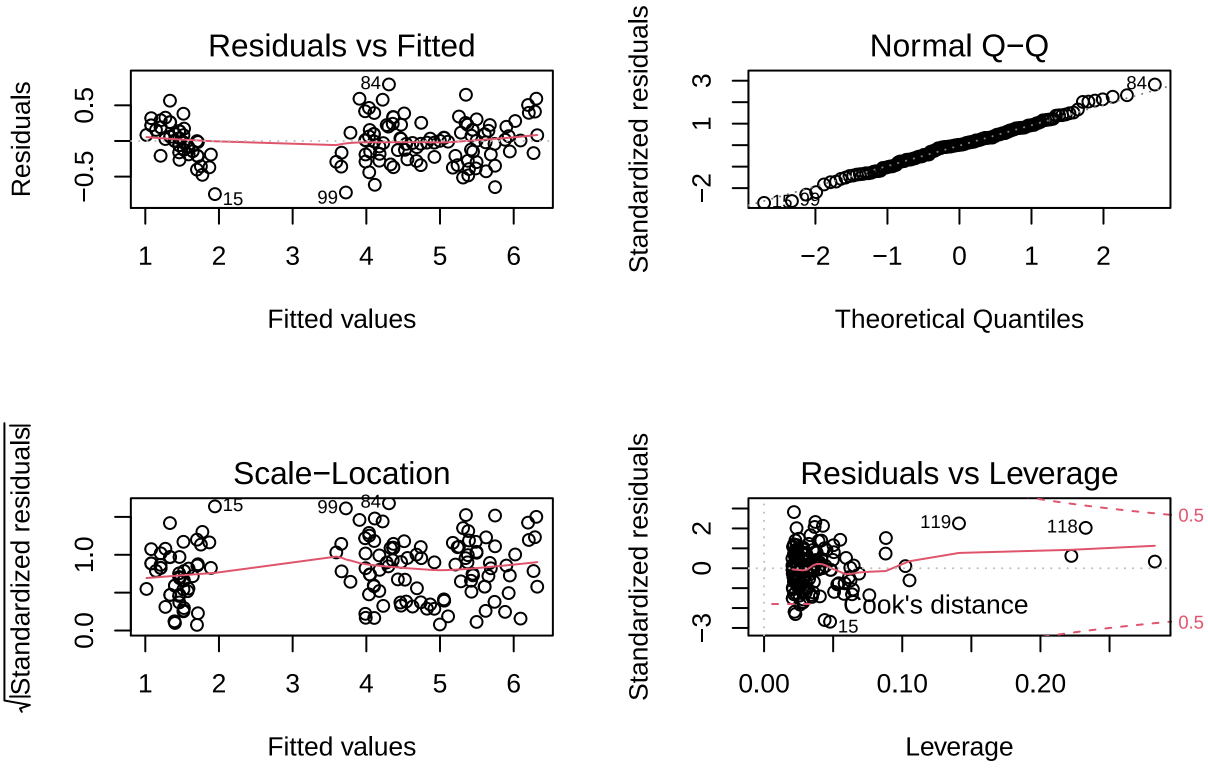
```
coefficients(summary(mod2))
```

```
##                  Estimate Std. Error    z value   Pr(>|z|)
## (Intercept)  -1.692716  0.8234646 -2.0556024 0.03982085
## Sepal.Length  0.100876  0.1370491  0.7360579 0.46169549
```

Check assumptions

All models have different types of assumptions and it is the user's responsibility to identify whether their model is suitable or not. For linear regression, the modeling assumptions can be evaluated using diagnostic plots.

```
mod1 <- lm(Petal.Length ~ Species + Sepal.Width *
    Sepal.Length, data = iris)
par(mfrow = c(2, 2))
plot(mod1)
```

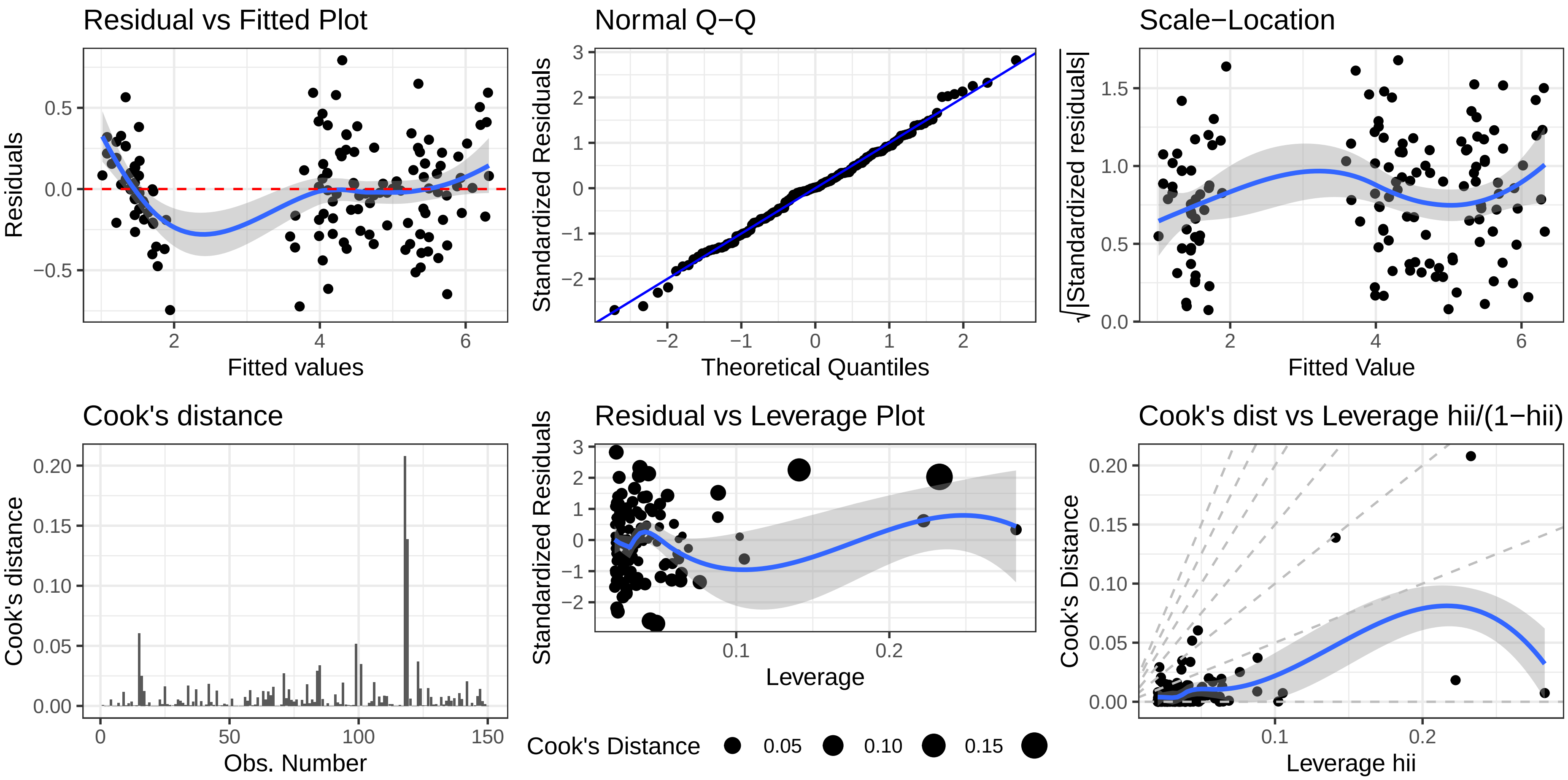


A slightly more attractive version can be implemented using the user defined function *diagPlot* below.

```
library(ggpubr)
```

```r
diagPlot <- function(model) {

    p1 <- ggplot(model, aes(.fitted, .resid)) +
        geom_point() + stat_smooth(method = "loess") +
        geom_hline(yintercept = 0, col = "red",
            linetype = "dashed") + xlab("Fitted values") +
        ylab("Residuals") + ggtitle("Residual vs Fitted Plot") +
        theme_bw()

    p2 <- ggplot(model, aes(qqnorm(.stdresid)[[1]],
        .stdresid)) + geom_point(na.rm = TRUE) +
        geom_abline(intercept = 0, colour = "blue") +
        xlab("Theoretical Quantiles") + ylab("Standardized Residuals") +
        ggtitle("Normal Q-Q") + theme_bw()

    p3 <- ggplot(model, aes(.fitted, sqrt(abs(.stdresid)))) +
        geom_point(na.rm = TRUE) + stat_smooth(method = "loess",
        na.rm = TRUE) + xlab("Fitted Value") +
        ylab(expression(sqrt("|Standardized residuals|"))) +
        ggtitle("Scale-Location") + theme_bw()

    p4 <- ggplot(model, aes(seq_along(.cooksd),
        .cooksd)) + geom_bar(stat = "identity",
        position = "identity") + xlab("Obs. Number") +
        ylab("Cook's distance") + ggtitle("Cook's distance") +
        theme_bw()

    p5 <- ggplot(model, aes(.hat, .stdresid)) +
        geom_point(aes(size = .cooksd), na.rm = TRUE) +
        stat_smooth(method = "loess", na.rm = TRUE) +
        xlab("Leverage") + ylab("Standardized Residuals") +
        ggtitle("Residual vs Leverage Plot") +
        scale_size_continuous("Cook's Distance",
            range = c(1, 5)) + theme_bw() + theme(legend.position = "bottom")
    p6 <- ggplot(model, aes(.hat, .cooksd)) +
        geom_point(na.rm = TRUE) + stat_smooth(method = "loess",
        na.rm = TRUE) + xlab("Leverage hii") +
        ylab("Cook's Distance") + ggtitle("Cook's dist vs Leverage hii/(1-hii)") +
        geom_abline(slope = seq(0, 3, 0.5), color = "gray",
            linetype = "dashed") + theme_bw()

    combine_plot <- ggarrange(p1, p2, p3, p4,
        p5, p6, ncol = 3, nrow = 2)
    return(combine_plot)
}
# source:https://rpubs.com/therimalaya/43190
```

```
plot <- diagPlot(mod1)

plot
```



Extract results

Although R can print analysis results, it often will not present results that can be directly reportable. You are often required to program your own code to store and process results and then assemble proper tables and figures. Luckily there are many third-party packages available to extract results from common models. *sjPlot* is a useful package that can process many common model results.

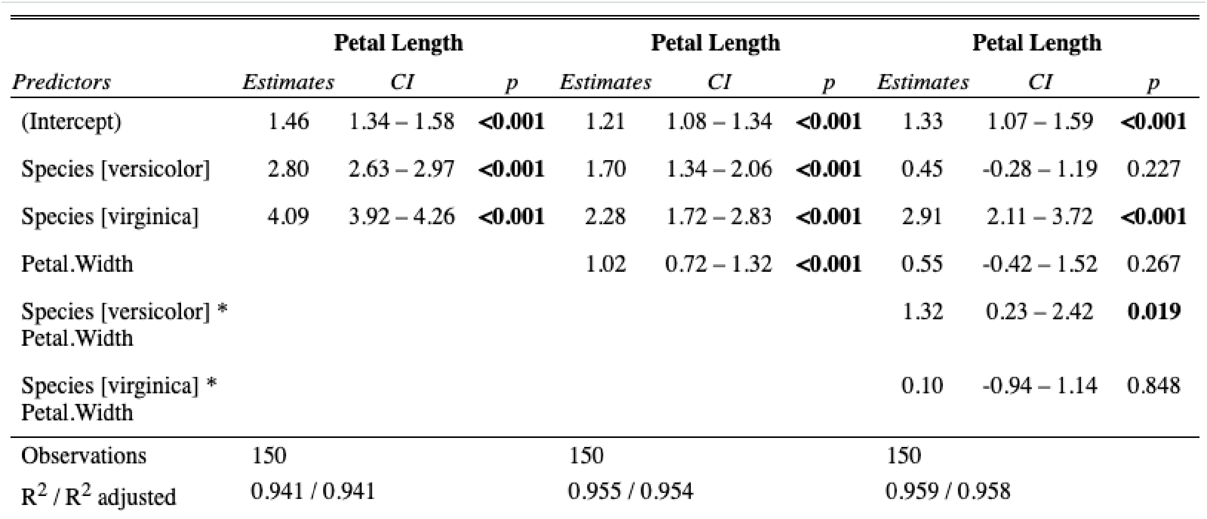
```
library(sjPlot)
```

```
## Registered S3 methods overwritten by 'lme4':
##   method                          from
##   cooks.distance.influence.merMod car
##   influence.merMod                car
##   dfbeta.influence.merMod         car
##   dfbetas.influence.merMod        car
```

```
## Install package "strengejacke" from GitHub (`devtools::install_github("strengejacke/stren
```

```
mod1 <- lm(Petal.Length ~ Species, data = iris)
mod2 <- lm(Petal.Length ~ Species + Petal.Width,
    data = iris)
mod3 <- lm(Petal.Length ~ Species + Petal.Width +
    Species:Petal.Width, data = iris)
```

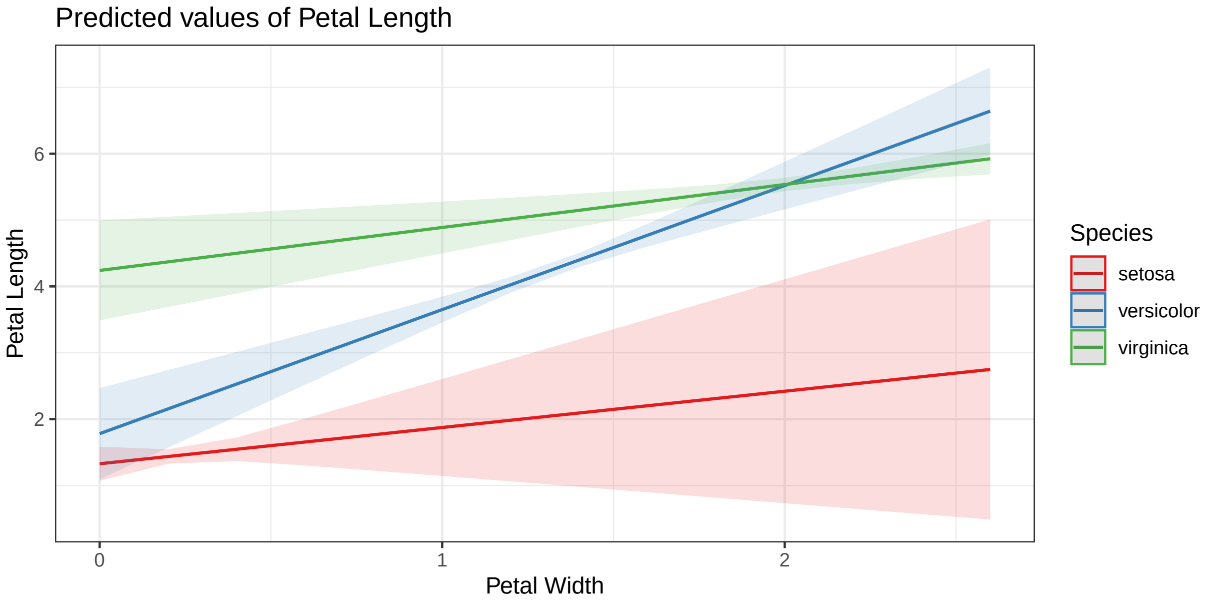
```
tab_model(mod1, mod2, mod3)
```



	Petal Length			Petal Length			Petal Length		
Predictors	*Estimates*	*CI*	*p*	*Estimates*	*CI*	*p*	*Estimates*	*CI*	*p*
(Intercept)	1.46	1.34 – 1.58	**<0.001**	1.21	1.08 – 1.34	**<0.001**	1.33	1.07 – 1.59	**<0.001**
Species [versicolor]	2.80	2.63 – 2.97	**<0.001**	1.70	1.34 – 2.06	**<0.001**	0.45	-0.28 – 1.19	0.227
Species [virginica]	4.09	3.92 – 4.26	**<0.001**	2.28	1.72 – 2.83	**<0.001**	2.91	2.11 – 3.72	**<0.001**
Petal.Width				1.02	0.72 – 1.32	**<0.001**	0.55	-0.42 – 1.52	0.267
Species [versicolor] * Petal.Width							1.32	0.23 – 2.42	**0.019**
Species [virginica] * Petal.Width							0.10	-0.94 – 1.14	0.848
Observations	150			150			150		
R^2 / R^2 adjusted	0.941 / 0.941			0.955 / 0.954			0.959 / 0.958		

Currently *tab_model* only support html output, which hopefully can be extended to latex in the future. *plot_model* from *jsplot* can be use to plot marginal effects of interaction models. More examples can be found here.

```
plot_model(mod3, type = "pred", terms = c("Petal.Width",
    "Species")) + theme_bw()
```



There are many factors you need to consider when displaying results, a few of them were listed here:

(1) Include point estimate, 95% CI and p-value (not starts) for regression models. The new recommendation is to report 3 digits if p-value < 0.05 otherwise 2 digits.

(2) Do not report the p-value when hypothesis testing is meaningless. A perfect example is a correlation coefficient $\rho = 0.08$ with a p-value< 0.001 (see example below).

```
set.seed(123)
a <- rnorm(1000)
b <- rnorm(1000)
cor.test(a, b)
```

```
##
##  Pearson's product-moment correlation
##
## data:  a and b
## t = 2.7423, df = 998, p-value = 0.006211
## alternative hypothesis: true correlation is not equal to 0
## 95 percent confidence interval:
##  0.02461834 0.14768079
## sample estimates:
##        cor
## 0.08647944
```

(3) Include reference values. *tab_model* is currently having trouble printing out reference values. It is also difficult to configure custom settings and print LaTex table. The function below prints out linear regression model results and can be configured for different types of models and output requirements.

```
reg_table <- function(model, dta) {
    # Summarise coef
    Coef <- data.frame(coefficients(summary(model))) %>%
        mutate(Variables = rownames(.)) %>% as_tibble() %>%
        filter(Variables != "(Intercept)") %>%
        mutate(Coef = format(round(Estimate, digits = 2),
            nsmall = 2)) %>% mutate(`95% CI` = paste0(format(round(Estimate -
        1.96 * Std..Error, digits = 2), nsmall = 2),
        ", ", format(round(Estimate + 1.96 * Std..Error,
            digits = 2), nsmall = 2))) %>% mutate(`p-value` = Pr...t..) %>%
        mutate(`p-value` = ifelse(`p-value` <
            0.001, "<0.001", ifelse(`p-value` <
            0.05, format(round(`p-value`, digits = 3),
            nsmall = 3), format(round(`p-value`,
            digits = 2), nsmall = 2)))) %>% select(Variables,
        Coef, `95% CI`, `p-value`)

    # categorical variables for left col
    if (length(model$xlevels) > 0) {
        levels <- reshape2::melt(model$xlevels)
        levels <- levels %>% mutate(Variables = paste0(L1,
            value)) %>% group_by(L1) %>% mutate(level = seq(n()))
    } else {
        levels <- data.frame(value = NA, L1 = "",
            Variables = "", level = NA)
```

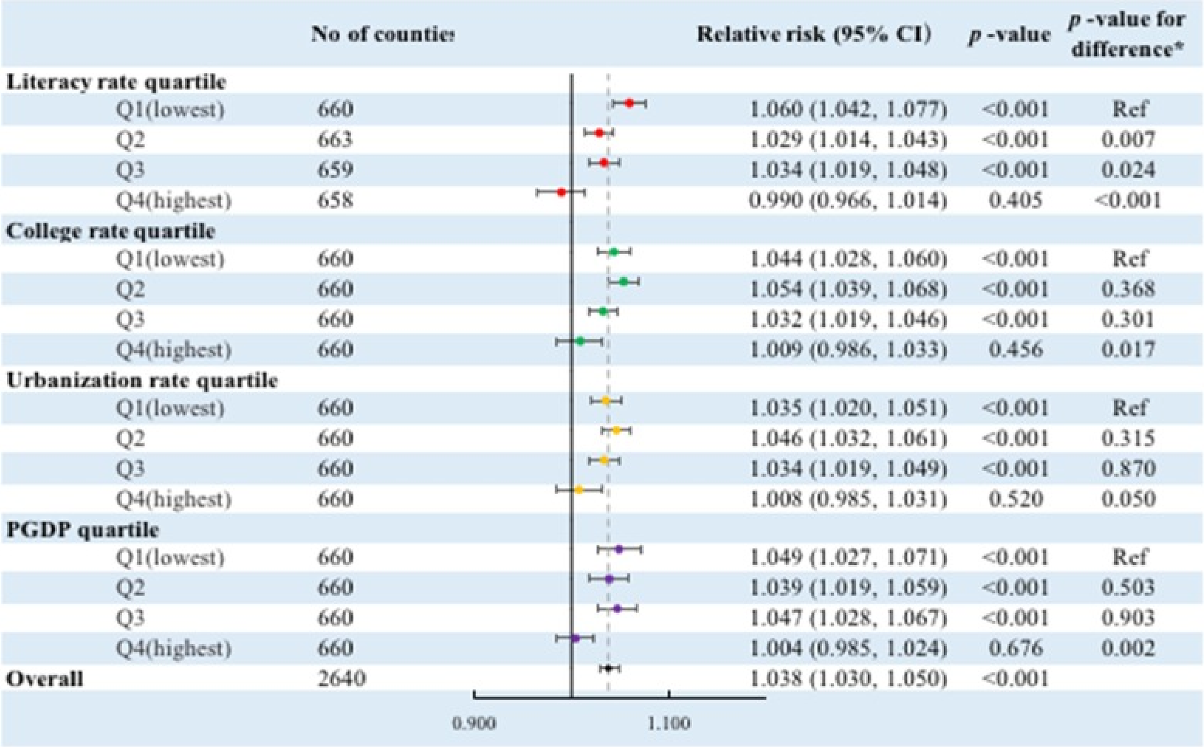
```r
}

# extract variable labels
varlabels <- as.data.frame(Hmisc::label(dta[,
    attr(model$terms, "term.labels")]))
names(varlabels) <- "label"
varlabels$L1 <- rownames(varlabels)
varlabels <- varlabels %>% mutate(label = ifelse(is.na(label) |
    as.character(label) == "", L1, as.character(label))) %>%
    mutate(listvar = seq(n()))

# prepare first column
tablelabel <- data.frame(Variables = names(model$coefficients)) %>%
    filter(Variables != "(Intercept)") %>%
    mutate(Variables = as.character(Variables)) %>%
    full_join(levels) %>% mutate(L1 = ifelse(is.na(L1),
    as.character(Variables), as.character(L1))) %>%
    left_join(varlabels) %>% arrange(listvar,
    level) %>% dplyr::select(-listvar, -level,
    -L1) %>% group_by(label) %>% mutate(total = n(),
    order = seq(n())) %>% mutate(expend = ifelse(order ==
    1 & total > 1, 2, 1)) %>% uncount(expend) %>%
    mutate(order = seq(n())) %>% mutate(value = ifelse(is.na(value) |
    (total > 1 & order == 1), as.character(label),
    as.character(value))) %>% mutate(Variables = ifelse(total >
    1 & order == 1, NA, Variables)) %>% ungroup() %>%
    dplyr::select(Variables, value, total)

# extract final table
finaltable <- tablelabel %>% full_join(Coef) %>%
    mutate(Coef = ifelse(!is.na(Variables) &
        is.na(Coef), "Ref", Coef)) %>% replace(is.na(.),
    "")
# extract which row for indentation and top
```

```
    # row name
    indent <- which(finaltable$Variables != "" &
        finaltable$total > 1)

    # print
    print(finaltable %>% select(-Variables, -total) %>%
        rename(Predictors = value) %>% kable(align = c("l",
        rep("c", 3)), booktabs = T, linesep = "") %>%
        kable_styling(bootstrap_options = "striped",
            full_width = F) %>% add_indent(indent))
}
```

```
Hmisc::label(iris$Petal.Length) <- "Petal length"
Hmisc::label(iris$Sepal.Width) <- "Sepal width"
Hmisc::label(iris$Sepal.Length) <- "Sepal length"
iris$Sepal.Length.Width <- iris$Sepal.Width *
    iris$Sepal.Length
Hmisc::label(iris$Sepal.Length.Width) <- "Sepal length * width"
mod <- lm(Petal.Width ~ Species + Petal.Length +
    Sepal.Width + Sepal.Length + Sepal.Length.Width,
    data = iris)
reg_table(mod, iris)
```

Predictors	Coef	95% CI	p-value
Species			
setosa	Ref		
versicolor	0.64	0.39, 0.88	<0.001
virginica	1.03	0.70, 1.36	<0.001
Petal length	0.24	0.15, 0.34	<0.001
Sepal width	0.10	-0.46, 0.65	0.73
Sepal length	-0.17	-0.45, 0.12	0.26
Sepal length * width	0.02	-0.07, 0.12	0.61

(3) Use table + Figure representation if you need/can



	No of countie:	Relative risk (95% CI)	*p* -value	*p* -value for difference*
Literacy rate quartile				
Q1(lowest)	660	1.060 (1.042, 1.077)	<0.001	Ref
Q2	663	1.029 (1.014, 1.043)	<0.001	0.007
Q3	659	1.034 (1.019, 1.048)	<0.001	0.024
Q4(highest)	658	0.990 (0.966, 1.014)	0.405	<0.001
College rate quartile				
Q1(lowest)	660	1.044 (1.028, 1.060)	<0.001	Ref
Q2	660	1.054 (1.039, 1.068)	<0.001	0.368
Q3	660	1.032 (1.019, 1.046)	<0.001	0.301
Q4(highest)	660	1.009 (0.986, 1.033)	0.456	0.017
Urbanization rate quartile				
Q1(lowest)	660	1.035 (1.020, 1.051)	<0.001	Ref
Q2	660	1.046 (1.032, 1.061)	<0.001	0.315
Q3	660	1.034 (1.019, 1.049)	<0.001	0.870
Q4(highest)	660	1.008 (0.985, 1.031)	0.520	0.050
PGDP quartile				
Q1(lowest)	660	1.049 (1.027, 1.071)	<0.001	Ref
Q2	660	1.039 (1.019, 1.059)	<0.001	0.503
Q3	660	1.047 (1.028, 1.067)	<0.001	0.903
Q4(highest)	660	1.004 (0.985, 1.024)	0.676	0.002
Overall	2640	1.038 (1.030, 1.050)	<0.001	

Source:https://www.sciencedirect.com/science/article/pii/S0160412020321966#f0015

Advanced topics

The benefit of using a full programming language instead of a statistical package for analysis is that you have unlimited potential to extend your analysis, e.g., develop new methods, web scraping, generate interactive graphics, establish a website, draft full paper etc. There are some basic skills that could facilitate these advanced techniques, which require some attention.

Control flow

The control flow in R is similar to other programming languages including choices (e.g.,. if/else, switch) and loops (e.g., for loop). The for loop will probably be one of your most frequently used control flow in the analysis as we frequently need to repeat operations. You can loop over a vector or a list.

```
list <- list(2, 3)


# loop version 1
for (p in list) {
    print(p)
}
```

```
## [1] 2
## [1] 3
```

```
for (i in 1:length(list)) {
    print(list[[i]])
}
```

```
## [1] 2
## [1] 3
```

One thing to notice is that compared with vectorized operations, using loops can down your code. This is a more severe issue for Python compared with R. You probably will not notice any differences when using R with simple operations However, when your work involve sampling, simulation or large datasets, the reduction in speed can be substantial.

```
system.time(print(apply(iris[1:4], 2, median)))
```

```
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width
##         5.80         3.00         4.35         1.30
```

```
##    user  system elapsed
##   0.001   0.000   0.000
```

```
system.time(for (i in 1:4) {
    print(median(iris[, i]))
})
```

```
## [1] 5.8
## [1] 3
## [1] 4.35
## [1] 1.3
```

```
##    user  system elapsed
##   0.004   0.000   0.004
```

This doesn't mean that you are not recommended to use loops. Actually compared with repetitively typing analysis code, using a function + loop is much more reliable and easy to maintain. A few rules to remember when using the loop are listed here:

- Use vectorized operations as much as possible
- Always pre-define and fill the object and avoid growing the object (R automatically create copies of an object when you try to add undefined elements, e.g., a new row).
- Avoid creating new objects within the loop.

Functions

Advance programming in R always requires you to be familiar with functions, as R essentially is a functional programming language. The motivation behind applying functions is that your code needs to be flexible, reusable and easily maintainable. If you find yourself having to copy-and-paste code, it's time to update your code with functions. A function is normally specified as follow:

```
function_name <- function(arg_1, arg_2, ...) {
   Function body
}
```

A function can be named or anonymous. The general rule of function is that it should be small in general programming (less than 100 lines of code). However in practice when your aim was not to distribute your function, you may use slightly longer functions. The easiest example of a function is provided below.

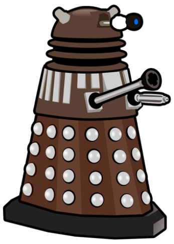
```
Dalek <- function(text = "") {
    if (text == "Exterminate") {
        knitr::include_graphics(here::here("Graphics/Dalek.png"))
    } else {
        print("Exterminate")
    }
}
Dalek("Hello")
```

```
## [1] "Exterminate"
```

```
Dalek()
```

```
## [1] "Exterminate"
```

```
Dalek("Exterminate")
```



Environments

The operating environment is a slightly complicated concept to explain to non-programmers. Detailed description can be found in the Advanced R by Hardley Wickham. You are using the environment every time you use R even you do not realize it. Essentially you can think of the environment as a name list of all the data, variables, libraries, functions and other objects. The name list links a name with a part of memory on your computer. Every name must be unique in an Environment.

```
a <- 1
a <- 1:5
a
```

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

Sometimes there can be confusion. See the example below:

```
mean
```

```
## function (x, ...)
## UseMethod("mean")
## <bytecode: 0x7fec4dba5c78>
## <environment: namespace:base>
```

```
mean <- 1
mean
```

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

```
mean(1:3)
```

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

Now, what is the object 'mean'? Is it a variable with a value of 1 or a function that calculates the mean? The answer is both. This relates to the current(local) and global environment. The variable 'mean' belongs to the current or local environment and the function 'mean' belongs the global environment. R program will search for the object first in the current environments and then the global environment to find matching names. When you call the function mean, R will fail to find any function in the current environment and move up to the global environment to find a matching name.

```r
mean <- function(x) 1
mean(1:5)
```

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

```r
mean(1:10)
```

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

Now I have defined a local function 'mean' and R will use my local function, unless I specify the environment that the 'mean' function that I want to call from. Here the 'mean' function is a part of the base library.

```r
base::mean(1:5)
```

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

Scoping Rules

Scoping rules are closely related to environment, it determines how values are assigned to free variables. An advanced introduction is provided in R Programming for Data Science by Roger Peng. R uses lexical scoping which means that it will search the values of free variables or objects are searched for in the environment in which the function was defined. The order of the R search starts from the local environment first, next to the parent environments and then to the global environment. Although R does this searching all the time, you may not notice any issues unless the objects with the same name were defined from different environments, e.g., same variable names in the nested functions. The recommendation is try to avoid using the same name for different variables/objects.

Funcitonal programming

R is not just a functional programming tool, it has many extended toolkit that can facilitate functional programming. An important one is the *purrr* package. *purrr* provides a range of functions (e.g., *map*) that functionalise for loops (similar to apply family but with greater flexibility), see details in Chapter 9 of Advanced R.

The map family functions provide extended tools to allow better implementation of functional operations with lists, vectors and tibbles.

See the following example that we wanted to regress the "Species" (fifth variable) on each of the petal and sepal length and width variables (first four variables). Results were saved in a list.

```
regess_on <- function(y, x, data) {
    lm(as.formula(paste(names(data)[y], " ~ ",
        names(data)[x])), data = data)
}
results <- map(1:4, regess_on, x = 5, data = iris)
summary(results[[1]])
```

```
##
## Call:
## lm(formula = as.formula(paste(names(data)[y], " ~ ", names(data)[x])),
##     data = data)
##
## Residuals:
## Sepal length
##     Min      1Q  Median      3Q     Max
## -1.6880 -0.3285 -0.0060  0.3120  1.3120
##
## Coefficients:
##                   Estimate Std. Error t value Pr(>|t|)
## (Intercept)         5.0060     0.0728  68.762  < 2e-16 ***
## Speciesversicolor   0.9300     0.1030   9.033 8.77e-16 ***
## Speciesvirginica    1.5820     0.1030  15.366  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.5148 on 147 degrees of freedom
## Multiple R-squared:  0.6187, Adjusted R-squared:  0.6135
## F-statistic: 119.3 on 2 and 147 DF,  p-value: < 2.2e-16
```

In another example, here we first split the data by species, fit a model to each species, compute the summary, then extract the R2.

```
iris %>% split(.$Species) %>% map(~lm(Sepal.Length ~
    Petal.Length, data = .)) %>% map(summary) %>%
    map_dbl("r.squared")
```

```
##     setosa versicolor  virginica
## 0.07138289 0.56858983 0.74688439
```

Data visulisation

Cleveland's five principles of graphical excellence.

Principle 1: Show the data clearly Principle 2: Use simplicity in design Principle 3: Use good alignment on a common scale for quantities to be compared. Principle 4: Keep the visual encoding transparent. Principle 5: Use graphical forms consistent with Principles 1 to 4.

Reporting

One-stop shop

Rmarkdown is a great tool for integrating all data procedures in to isolated environments to allows for better reproducibility, higher level of transparency and easier project management. Here we recommend a two-stage process:

- Stage 1: generic data cleaning. Conduct data cleaning that is associated with one specific project using one R Markdown document. Store all cleaned data in a data file(s).
- Stage 2: conduct paper or report specific data cleaning and data analysis. All analysis should be included into one R Markdown per paper/report.

Write up of the analysis results

- Report the nature and source of the data, validity of instrument and data collection process (e.g., response rate and any possible bias).
- Report any data editing procedures, including any imputation and missing data mechanisms

- When reporting analyses of volunteer data or other data that may not be representative of a defined population, includes appropriate disclaimers and, if used, appropriate weighting.
- Include the complete picture of the analysis results, which may require presenting tables and figures in appendix tables. For example, when reporting a series of multivariate regression models between an exposure and different outcomes, you can choose to include a summary table of adjust coef between exposure and different outcomes in the main text and include all the individual regression model results in the Appendix. The reader can use the appendix tables to understand the impact of confounding variables in the model.
- Report prevalence of outcomes or weighted prevalence of outcomes for representative samples.
- Report point estimate, 95% confidence interval and p-value in results
- Use graphical representations for reporting interaction effects (marginal plot)
- Acknowledges statistical and substantive assumptions made in the execution and interpretation of any analysis.
- Reports the limitations of statistical inference and possible sources of error.
- Where appropriate, addresses potential confounding variables not included in the study.
- Conveys the findings in ways that are meaningful and visually apparent and to the user/reader. This includes properly formatted tables and meaningful graphics (use guidelines by Gordon and Finch (2015)).
- To aid peer review and replication, shares the data (or synthetically generated data) used in the analyses whenever possible/allowable
- Provide all analysis code either as an Appendix or in open repositories such as Github

Version control

Gordon, Ian, and Sue Finch. 2015. "Statistician Heal Thyself: Have We Lost the Plot?" *Journal of Computational and Graphical Statistics* 24 (4): 1210–29. https://minerva-access.unimelb.edu.au/bitstream/handle/11343/55491/Gordon%20and%20Finch%202014%20DOI%20version.pdf;jsessionid=557C01A51F9EC550925E94F564ED970A?sequence=1.

Perperoglou, Aris, Willi Sauerbrei, Michal Abrahamowicz, and Matthias Schmid. 2019. "A Review of Spline Function Procedures in R." *BMC Medical Research Methodology* 19 (1): 1–16.

Williamson, Tyler, Misha Eliasziw, and Gordon Hilton Fick. 2013. "Log-Binomial Models: Exploring Failed Convergence." *Emerging Themes in Epidemiology* 10 (1): 1–10. https://doi.org/https://doi.org/10.1186/1742-7622-10-14.