Useful things

Paul Schmidt

2023-11-07

A personal collection of useful R packages and more.

Table of contents

This chapter is a collection of things I wish I had known earlier in my years using R and that I hope can be of use to you. Sections are named after R packages or whatever applies and sorted alphabetically.

1 {broom}

In R, results from statistical tests, models etc. are often formatted in a way that may not be optimal for further processing steps. Luckily, {broom} will format the results of the most common functions into tidy data structures.

```r
# Correlation Analysis for built-in example data "mtcars"
mycor <- cor.test(mtcars$mpg, mtcars$disp)
mycor
```

```
	Pearson's product-moment correlation

data:  mtcars$mpg and mtcars$disp
t = -8.7472, df = 30, p-value = 9.38e-10
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 -0.9233594 -0.7081376
sample estimates:
       cor
-0.8475514
```

```r
library(broom)
tidy(mycor)
```

```
# A tibble: 1 x 8
  estimate statistic  p.value parameter conf.low conf.high method    alternative
     <dbl>     <dbl>    <dbl>     <int>    <dbl>     <dbl> <chr>     <chr>
1   -0.848     -8.75 9.38e-10        30   -0.923    -0.708 Pearson'~ two.sided
```

2 `{conflicted}`

Sometimes, different packages have different functions with identical names. A famous example is the function `filter()`, which exists in {stats} and {dplyr}. If both of these packages are loaded, it is not clear which of the two functions should be used. This is called a function conflict and it is especially tricky here since {stats} is always loaded. By default, R will simply pick the package that was loaded later - which is obviously not optimal.

One way of dealing with function conflicts is by using the packagename::functionname() method, because when writing `dplyr::filter()` instead of `filter()` it is no longer ambiguous which function you are referring to.

Another way of dealing with function conflicts more explicitly is by loading the {conflicted} package. Once it is loaded, function conflicts will lead to an `Error` that forces you to deal with the issue:

```
library(conflicted)
library(dplyr)

PlantGrowth %>% filter(weight > 6)
```

```
Error:
! [conflicted] filter found in 2 packages.
Either pick the one you want with `::`:
* dplyr::filter
* stats::filter
Or declare a preference with `conflicts_prefer()`:
* `conflicts_prefer(dplyr::filter)`
* `conflicts_prefer(stats::filter)`
```

As you can see, it first suggests using the packagename::functionname() method mentioned above, but also points to the `conflict_prefer()` function. By running this function once in the beginning of the script, R will always use the function from the package that you declared the "winner":

```
library(conflicted)
library(dplyr)

conflicts_prefer(dplyr::filter)

PlantGrowth %>% filter(weight > 6)
```

```
   weight group
1    6.11   ctrl
2    6.03   trt1
3    6.31   trt2
4    6.15   trt2
```

3 {desplot}

{desplot} makes it easy to plot experimental designs of field trials in agriculture. However, you do need two columns that provide the x and y coordinates of the individual plots on your field.

TO DO

4 {dlookr}

When providing descriptive statistics tables, one may find the number of relevant measures become annoyingly large so that even with the {tidyverse}, several lines of code are necessary. Here are just five measures, and they are not even including the `na.rm = TRUE` argument, which is necessary for data with missing values.

```r
library(tidyverse)

PlantGrowth %>%
  group_by(group) %>%
  summarise(
    mean = mean(weight),
    stddev = sd(weight),
    median = median(weight),
    min = min(weight),
    max = max(weight)
  )
```

```
# A tibble: 3 x 6
  group  mean stddev median   min   max
  <fct> <dbl>  <dbl>  <dbl> <dbl> <dbl>
1 ctrl   5.03  0.583   5.15  4.17  6.11
2 trt1   4.66  0.794   4.55  3.59  6.03
3 trt2   5.53  0.443   5.44  4.92  6.31
```

Obviously, there are multiple packages who try to address just that. The one I've been using for some time now is {dlookr} with its `describe()` function. It actually provides more measures than I usually need[1], but it has everything I want and I disregard the rest (via `select()`).

```
PlantGrowth %>%
  group_by(group) %>%
  dlookr::describe(weight)
```

```
# A tibble: 3 x 27
  described_variables group     n    na  mean    sd se_mean   IQR skewness
  <chr>               <fct> <int> <int> <dbl> <dbl>   <dbl> <dbl>    <dbl>
1 weight              ctrl     10     0  5.03 0.583   0.184 0.743    0.321
2 weight              trt1     10     0  4.66 0.794   0.251 0.662    0.659
3 weight              trt2     10     0  5.53 0.443   0.140 0.467    0.673
# i 18 more variables: kurtosis <dbl>, p00 <dbl>, p01 <dbl>, p05 <dbl>,
#   p10 <dbl>, p20 <dbl>, p25 <dbl>, p30 <dbl>, p40 <dbl>, p50 <dbl>,
#   p60 <dbl>, p70 <dbl>, p75 <dbl>, p80 <dbl>, p90 <dbl>, p95 <dbl>,
#   p99 <dbl>, p100 <dbl>
```

> **i** Note
>
> It is intentional that I did not actually load the {dlookr} package, but instead used its `describe()` function via the packagename::functionname() method. This is because of a minor bug in the {dlookr} package described here, which is only relevant if you are using the package with knitr/Rmarkdown/quarto. I am using quarto to generate this website and thus I avoid loading the package. This is fine for me, since I usually only need this one function one time during an analysis. It is also fine for you, since the code works the same way in a standard R script.

5 {ggtext}

Adding long text to plots created via {ggplot2} is problematic, since you have to insert line breaks yourself. However, {ggext}'s `geom_textbox()` for data labels and `element_textbox_simple()` for title, caption etc. will automatically add line breaks:
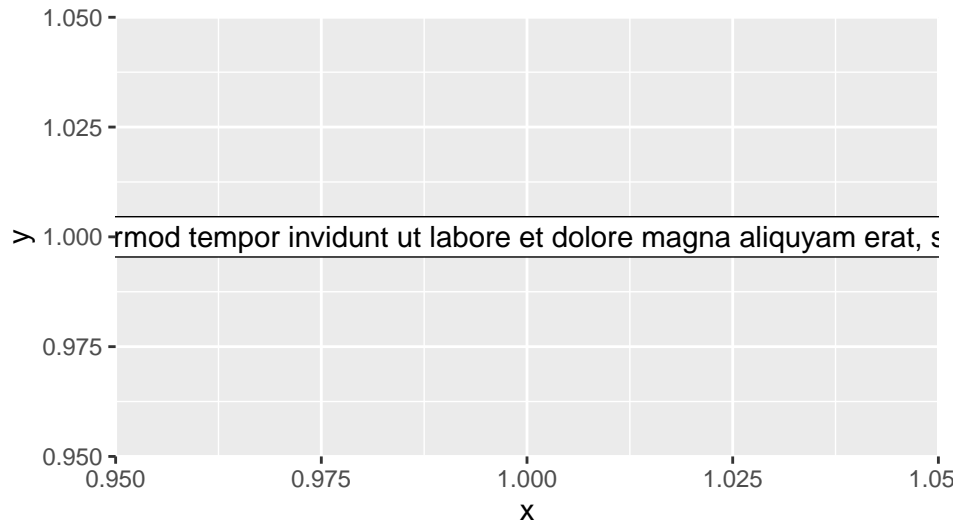
```
longtext <- "Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirm
```

---

[1]Keep in mind that `p00` is the 0th percentile and thus the minimum. Analogously, `p50` is the median and `p100` the maximum.
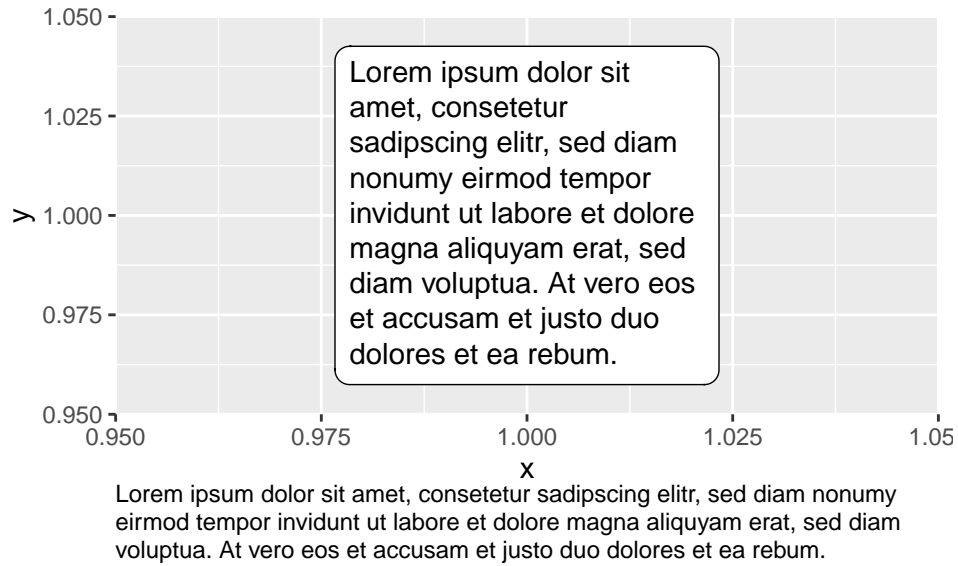
```r
library(ggplot2)

ggplot() +

  aes(y = 1, x = 1, label = longtext) +
  geom_label() +
  labs(caption = longtext)
```


quyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum.

```r
library(ggtext)

ggplot() +
  theme(plot.caption =
          element_textbox_simple()) +
  aes(y = 1, x = 1, label = longtext) +
  geom_textbox() +
  labs(caption = longtext)
```

Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum.

6 {here}

TO DO

7 {insight}

TO DO

8 {janitor}

TO DO

9 Keyboard shortcuts

Here are shortcuts I actually use regularly in RStudio:

Shortcut	Description
CTRL+ENTER	Run selected lines of code
CTRL+C	Convert all selected lines to comment
CTRL+SHIFT+M	Insert %>%

Shortcut	Description
CTRL+SHIFT+R	Insert code section header
CTRL+LEFT/RIGHT	Jump to Word
CTRL+SHIFT+LEFT/RIGHT	Select Word
ALT+LEFT/RIGHT	Jump to Line Start/End
ALT+SHIFT+LEFT/RIGHT	Select to Line Start/End
CTRL+A	Highlight everything (to run the entire code)
CTRL+Z	Undo

Keyboard shortcuts can be customized in RStudio as described here.

10 `{modelbased}`

TO DO

11 `{openxlsx}`

TO DO

12 `{pacman}`

You now know how to install and load R packages the standard way. However, over the years I switched to using the function `p_load()` from the {pacman} package instead of `library()` and `install.packages()`. The reason is simple: Usually R-scripts start with multiple lines of `library()` statements that load the necessary packages. However, when this code is run on a different computer, the user may not have all these packages installed and will therefore get an error message. This can be avoided by using the `p_load()`, because it

- loads all packages that are installed and
- installs and loads all packages that are not installed.

Obviously, {pacman} itself must first be installed (the standard way). Moreover, you may now think that in order to use `p_load()` we do need a single `library(pacman)` first. However, we can avoid this by writing `pacman::p_load()` instead. Simply put, writing `package_name::function_name()` makes sure that this explicit function from this explicit package is being used. Additionally, R actually lets you use this function without loading the corresponding package. Thus, we now arrived at the way I handle packages at the beginning of all my R-scripts:

```
pacman::p_load(
  package_name_1,
  package_name_2,
  package_name_3
)
```

13 {patchwork}

TO DO

14 {performance}

TO DO

15 {readxl}

TO DO

16 {reprex}

TO DO

17 {scales}

TO DO

18 %in% and %not_in%

R has the built-in function `%in%` which checks whether something is present in a vector.

```
treatments <- c("Ctrl", "A", "B")
```

Not only can we checke which treatments are present in our `treatment` vector (left), but we can also easily keep only those that are (right).

```
c("A", "D") %in% treatments
```

```
[1]  TRUE FALSE
```

```
c("A", "D") %>% .[. %in% treatments]
```

```
[1] "A"
```

Not built-in, for some reason, is the opposite of that function - checking whether something is **not** present. Yet, we can quickly built our own function that does exactly that:

```
`%not_in%` <- Negate(`%in%`)
```

```
c("A", "D") %not_in% treatments
```

```
[1] FALSE  TRUE
```

```
c("A", "D") %>% .[. %not_in% treatments]
```

```
[1] "D"
```

19 system('open "file.png"')

TO DO