Data Wrangling and Visualization 101 in R

Florian Jaeger

Fall 2020

Contents

1 Overview

This document provides an introduction to data wrangling and visualization in R. It contains sections marked as exercises. **Please complete these exercises prior to next Monday's class** and submit them on slack. In addition your homework for this Wednesday is to apply the knowledge gain from this tutorial to *your own data*. **By this Wednesday, post at least one plot about your data on slack using the skills learned in this tutorial.** You are encouraged to ask questions (and help others) on slack.

2 Preliminaries

2.1 Version control

RStudio makes version control, data backup, and data sharing easy (e.g., via Github.com). To use it, download and install git on your computer. Get a free github.com or bitbucket.com account. You only have to do this once.

Then, for each project, create a new project in RStudio and link it to the remote repository (select "Create project" > "Version control"). You will have to enter a URL for the remote repository, which you get, for example, at github.com under the repository's main page by clicking the "Clone or download button".

For step by step instructions, follow these links:

- Setting up RStudio for version control
- RStudio help on version control
- Reverting a file to an earlier version

You can clone this repository from Github. In RStudio:

1. From the file menu (top left), select "New project …"
2. Select "Version control"
3. Select "Git"
4. Enter "https://github.com/tfjaeger/BCS206DataWrangling.git" in the top field (for the Repository URL). Select where you want this project to be stored in the last/third field. I usually just go with the default, which for me is the desktop.
5. This should clone the tutorial for Monday from Github.com to your local drive, and open it as a new R project.
6. Once that has happened, check in the Git tab (top right of RStudio window) whether you're on the right branch of the repository.
7. **This tutorial is the 2020-2021 branch of the git repository.** Right next to the "New Branch" button it should say "2020-2021". If it instead says "master", click on "master". This opens a menu from which you can select "2020-2021". Please do so.

2.2 Reproducibility and literate coding

R and RStudio support reproducibility oriented literate coding via Sweave and Knitr: lab books, presentations, and papers can weave/knit together data, code, and text. The document you share contains the code needed to create its outputs (figures, tables, etc.). This is achieved by combining latex or R markdown with R code (or, for that matter, code from other programming languages). For an excellent video-based introduction, see this tutorial on R markdown. *This document is R markdown compiled with RStudio's knitr.* If you downloaded the git repository, you should see both the .Rmd and the .pdf file. The latter is created ("knitted") from the former. The former contains all the R code required to generate the figures, etc. in the PDF file.

3 Data wrangling

The *R* libraries *dplyr* provide us with efficient ways to transform ('wrangle') our data tables. The library *magrittr* let's us concatenate these operations in transparent and easy to read code. Like many of the packages we use in this tutorial, they are part of the *tidyverse*—an awesome and powerful collection of R packages for the data sciences (https://www.tidyverse.org/).

There are many video and interactive tutorials for R, RStudio, and the tidyverse online. I *highly* recommend that you invest a handful of ours to get your inner data scientist started. Your future self will thank you. R

is increasingly used both in academia and in the data sciences, and that includes the libraries we use for this tutorial (which are also available in similar form for Python). I would recommend going through at least the first four Sections of RStudio's R Primers: The basics, working with data, visualizing data, and tidying data.

3.1 An example data set

We will illustrate the use of *dplyr* with the following data from an experiment with a 2AFC task. On each trial, subjects had to fixate on a fixation cross. The fixation cross was then replaced by a small small number, and subjects had to quickly answer whether they saw the number "3" or "8". Half of the trials displayed "3", half displayed "8". Within subject, the experiment manipulated two variables, the size of the number (1, 2, or 4) and whether the number was surrounded by four other numbers above, below, left, and right of it (hard) or not (easy). The presence of other numbers presented close to the target number is known to make recogntion of the target number harder—an effect known as "visual crowding". For each trial, the software recorded whether the subject's response was correct (1 = correct; 0 = incorrect) and the reaction time for the response (RT):

```
summary(d)
```

```
##  size     condition       trial           subject        correct      
##  1:4800   easy:7200   Min.   :  1.00   1      :1440   Min.   :0.0000  
##  2:4800   hard:7200   1st Qu.: 60.75   2      :1440   1st Qu.:0.0000  
##  4:4800               Median :120.50   3      :1440   Median :1.0000  
##                       Mean   :120.50   4      :1440   Mean   :0.6068  
##                       3rd Qu.:180.25   5      :1440   3rd Qu.:1.0000  
##                       Max.   :240.00   6      :1440   Max.   :1.0000  
##                                        (Other):5760                   
##        RT        
##  Min.   :  24.91  
##  1st Qu.: 103.79  
##  Median : 213.15  
##  Mean   : 256.52  
##  3rd Qu.: 351.16  
##  Max.   :1142.95  
##
```

```
glimpse(d)
```

```
## Rows: 14,400
## Columns: 6
## $ size      <fct> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...
## $ condition <fct> easy, easy, easy, easy, easy, easy, easy, easy, easy, eas...
## $ trial     <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, ...
## $ subject   <fct> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 1, 2, 3, 4, 5, 6, 7, 8, 9,...
## $ correct   <int> 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 0, 0, 1, 0, 0, 1, ...
## $ RT        <dbl> 577.883, 253.426, 276.346, 254.985, 282.830, 368.297, 640...
```

3.2 Dplyr's verbs

Dplyr has 'verbs' like filter, select, summarize, mutate, transmute, etc. to let use conduct operations on our data, and reshape the data frame into the format we need. We can use dplyr, for example, to calculate the proportion correct answers in our experiment by using *summarise*.

```r
summarise(d, meanCorrect = mean(correct))
```

```
## # A tibble: 1 x 1
##   meanCorrect
##         <dbl>
## 1       0.607
```

Or just for number size 1:

```r
d.1 = filter(d, size == "1")
summarise(d.1, meanCorrect = mean(correct))
```

```
## # A tibble: 1 x 1
##   meanCorrect
##         <dbl>
## 1       0.502
```

3.3 Maggritr's pipes

Here we will use only of the 'pipes' magrittr provides:

- x %>% f: takes x and hands it to the function f on the right, as f's first argument
- x %<>% f1 %>% f2 %>% etc.: takes x hands it to f1, takes the output of f1 and hands it to f2, etc. And since the first pipe was %<>% (rather than just %>%), the final result will be written back into x.

Figure 1: Magritt's pipe

Figure 2: Magrittr's pipe

3.4 Putting it together: Wrangling through pipes

Remember how we got the mean proportion correct for just number size 1?

```r
d.1 = filter(d, size == "1")
summarise(d.1, meanCorrect = mean(correct))
```

```
## # A tibble: 1 x 1
##   meanCorrect
##         <dbl>
## 1       0.502
```

This is inelegant and hard to read. Pipes let us make this more transparent:

```r
d %>%
  filter(size == "1") %>%
  summarise(meanCorrect = mean(correct))
```

```
## # A tibble: 1 x 1
##   meanCorrect
##         <dbl>
## 1       0.502
```

And this advantage becomes even clearer, the more operationgs we concatenate. For example, *group_by* is an elegant operator that tells the pipes to conduct all subsequent operations for each of the groups (and then put all the separate outcomes back together into a single data frame). So if we want the proportion correct for all groups:

```r
d %>%
  group_by(size) %>%
  summarise(meanCorrect = mean(correct))
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
## # A tibble: 3 x 2
##   size  meanCorrect
##   <fct>       <dbl>
## 1 1           0.502
## 2 2           0.588
## 3 4           0.730
```

3.5 Exercises

Remember dplyr cheatsheet! How can we:

- View the entire data set? (*View*)
- Calculate the by-subject averages for all three number sizes? (*group_by*, *summarise*)
- Calculate the by-subject standard deviations around those averages? (*group_by*, *summarise*)
- Attach this information (the averages and SDs) to each row of the present data.frame? (*group_by*, *mutate*)
- Determine whether RTs were on average faster for correct, as compared to incorrect, trials?
- Add a column for log-transformed RTs to the data set?
- Remove the old column for raw RTs? (*select*)
- Sort the data by log-transformed reaction times? (*arrange*)

Say we further have an additional data frame with information about our subjects:

```
## # A tibble: 10 x 3
## # Rowwise:
##    subject gender   age
##    <fct>   <fct>  <dbl>
##  1 1       female    20
##  2 2       male      19
##  3 3       male      22
##  4 4       female    19
##  5 5       female    19
##  6 6       male      21
##  7 7       female    20
##  8 8       male      21
##  9 9       female    19
## 10 10      male      20
```

- How can we join the information from the two data sources together? (*left_join*)

```
## # A tibble: 10 x 3
## # Rowwise:
##    subject gender   age
##    <fct>   <fct>  <dbl>
##  1 1       female    20
##  2 2       male      19
##  3 3       male      22
##  4 4       female    19
##  5 5       female    19
##  6 6       male      21
##  7 7       female    20
##  8 8       male      21
##  9 9       female    19
## 10 10      male      20
```

```
## Joining, by = "subject"
```

```
## # A tibble: 14,400 x 8
##    size  condition trial subject correct    RT gender   age
##    <fct> <fct>     <int> <fct>     <int> <dbl> <fct>  <dbl>
##  1 1     easy          1 1             1  578. female    20
##  2 1     easy          1 2             0  253. male      19
##  3 1     easy          1 3             1  276. male      22
##  4 1     easy          1 4             0  255. female    19
##  5 1     easy          1 5             1  283. female    19
##  6 1     easy          1 6             0  368. male      21
##  7 1     easy          1 7             1  641. female    20
##  8 1     easy          1 8             0  357. male      21
##  9 1     easy          1 9             1  681. female    19
## 10 1     easy          1 10            1  248. male      20
## # ... with 14,390 more rows
```

4 Data visualization

The two main libraries in R we will be using for visualization are *ggplot2* and *plotly*. Ggplot2 provides a grammar of graphics approach to plotting. Plotly let's us interact with our data. In particular, *ggplotly()* wrapped around a ggplot2 figure let's us interact with that figure. Before you start, make sure to get the

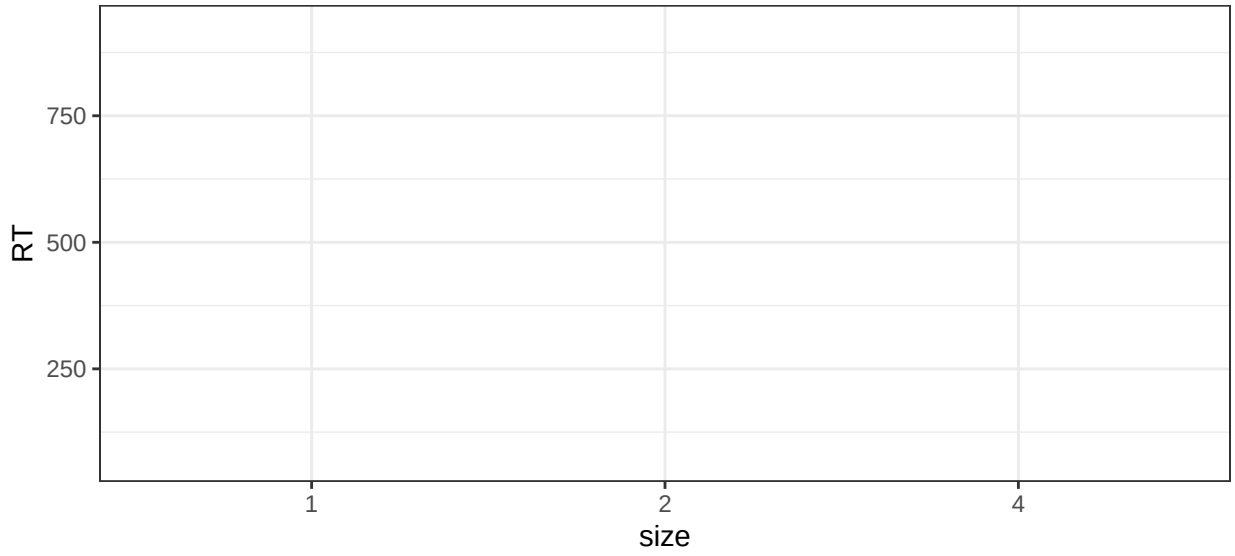
superb ggplot2 cheatsheet.

4.1 Ggplot2's components (aesthetic mappings)

In order to plot in ggplot2, we need to understand the way it thinks about visualization. There are excellent online course that explain all of this, so I focus on the basics. **For reasons that become apparent below, we start by just looking at one subject, subject 1.**

At the heart of a plot is a mapping between properties of your data (i.e., column in your data frame) and abstract properties of the plot (such as x- or y-coordinates, color, fill, transparency (alpha), linetype, shape, or label information). If we call the function ggplot() in order to create a figure, we specify two arguments: the name of the data frame we want to work with, and the mapping. The latter is done through a helpful function called *aes()*—for aesthetics:

```
d.s1 = d %>%
  filter(subject == 1)

ggplot(
  data = d.s1,
  mapping =
    aes(
      x = size,
      y = RT))
```




```
# or equivalently and shorter:
# ggplot(
#   d,
#   aes(
#       x = size,
#       y = RT))
```

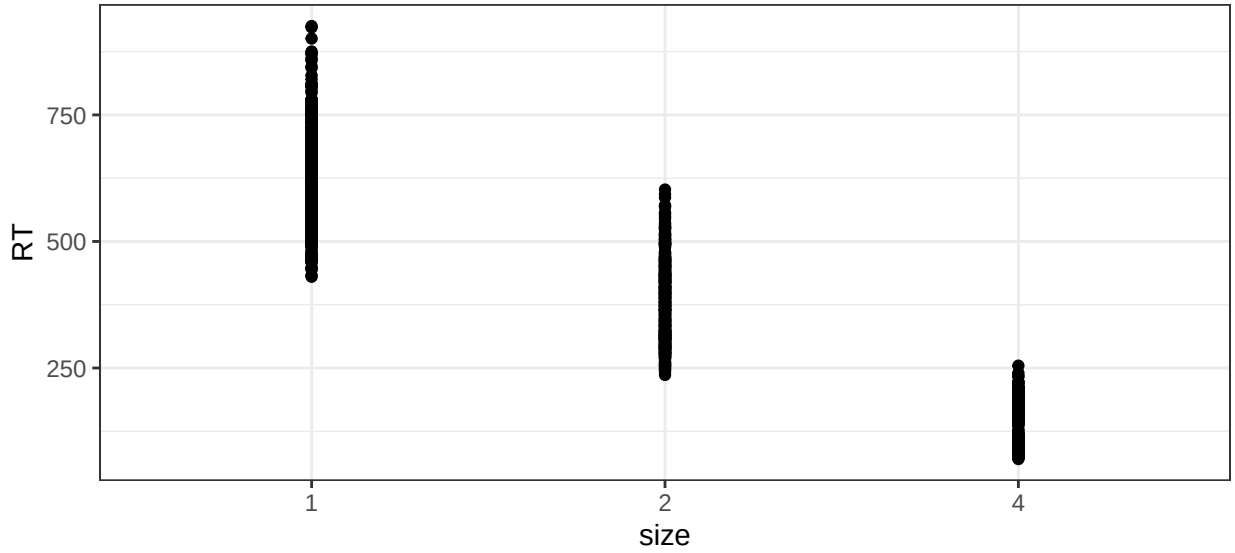
4.1.1 Adding geometric components (geoms)

Notice that this by itself only returns an empty plot. That's the case because we have not yet specified how we want the abstract properties of the graph to be expressed visually. That's achieved by specifying *geom*s (for geometrics), such as points (*geom_point*), lines (*geom_line*), histograms (*geom_histogram*), lineranges

(*geom_linerange*), and many similar functions (I take it you're getting th hang for the naming scheme …). You can find all of them on the ggplot2 cheatsheet.

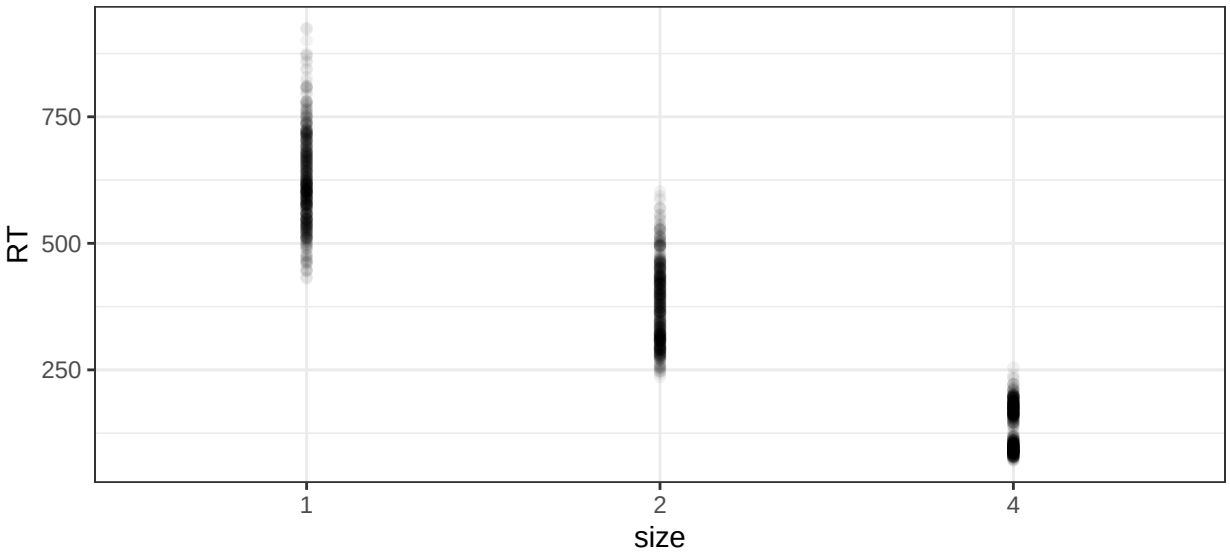
We add such components to a plot with "+".

```
p = ggplot(
  data = d.s1,
  mapping =
    aes(
      x = size,
      y = RT)) +
  geom_point()

plot(p)
```



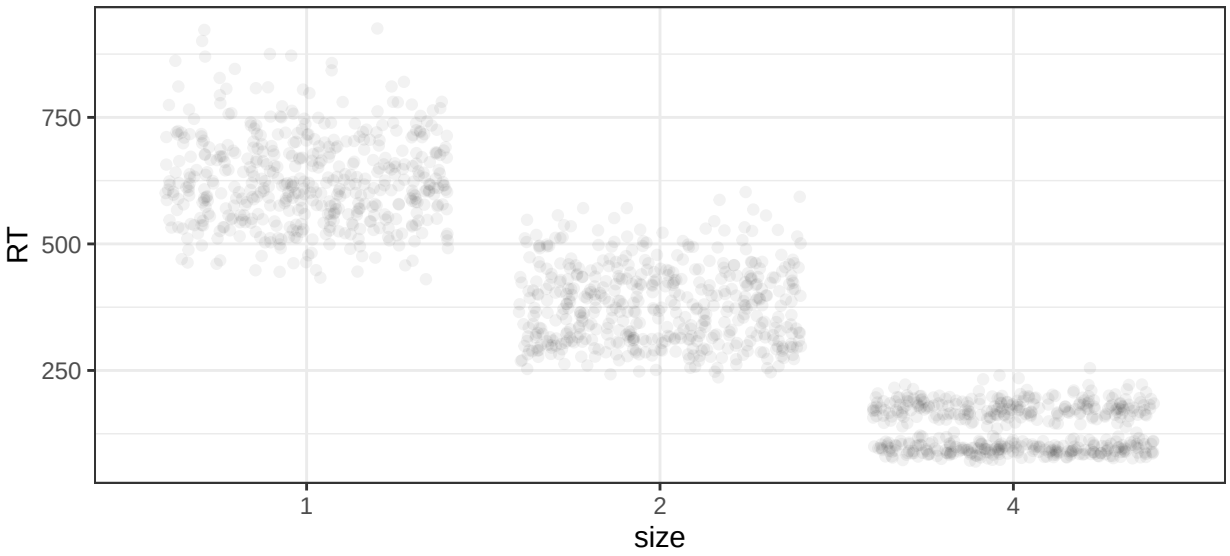
Notice that this plot is not particularly helpful! There is a lot of points and they are all overlaid, making it hard to get a clear idea of the data. One way we can improve this plot is to add transparency to the points. Another way is to jitter the points along the x-axis. Both approaches help convey information about the distribution of RTs at each number size, and sometimes—especially, when we have a lot of data—we might want to combine both approaches. For example, here's the same plot with just transparency added. Since we have quite a bit of data, it still does not convey much information:

```
p = ggplot(
  data = d.s1,
  mapping =
    aes(
      x = size,
      y = RT)) +
  geom_point(alpha = .05)

plot(p)
```



And here is the plot with some additional jitter added along the x-axis. Note that we intentionally avoid jitter along the y-axis:

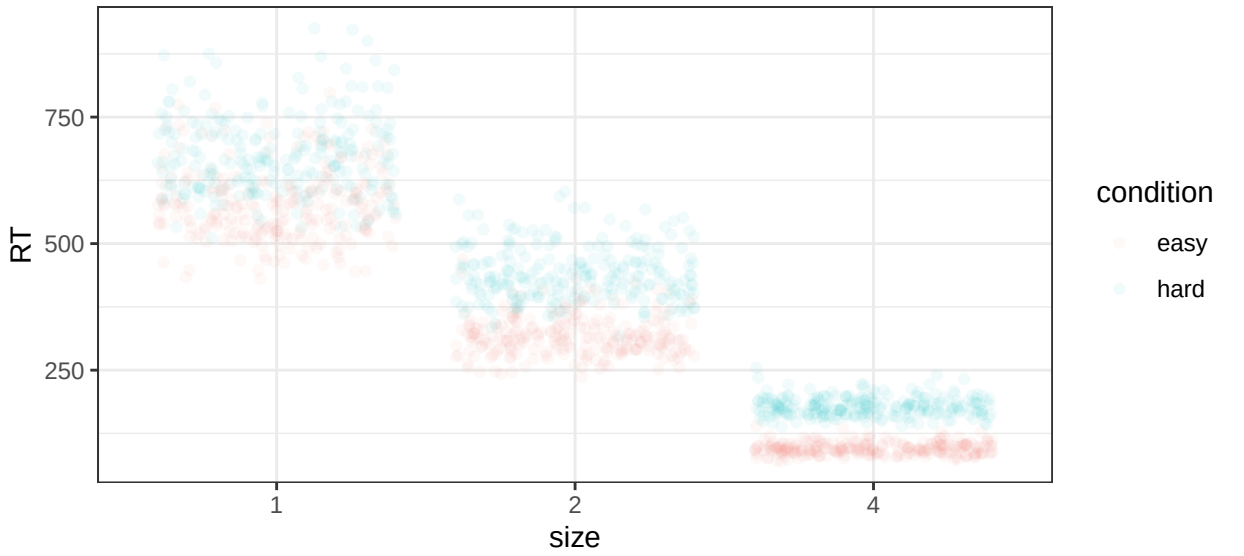
```
p = ggplot(
  data = d.s1,
  mapping =
    aes(
      x = size,
      y = RT)) +
  geom_point(alpha = .05, position = position_jitter(height = 0))

plot(p)
```



That's much better already, giving us a much clearer idea of how the data is distributed! We are now in a position to think about how to include information about condition in this plot. There are a few ways to do this. For example, we could use different point shapes or colors for the easy and hard condition. Here, we do the latter:

```
p = ggplot(
  data = d.s1,
```

```
  mapping =
    aes(
      x = size,
      y = RT,
      color = condition)) +
  geom_point(alpha = .05, position = position_jitter(height = 0))

plot(p)
```

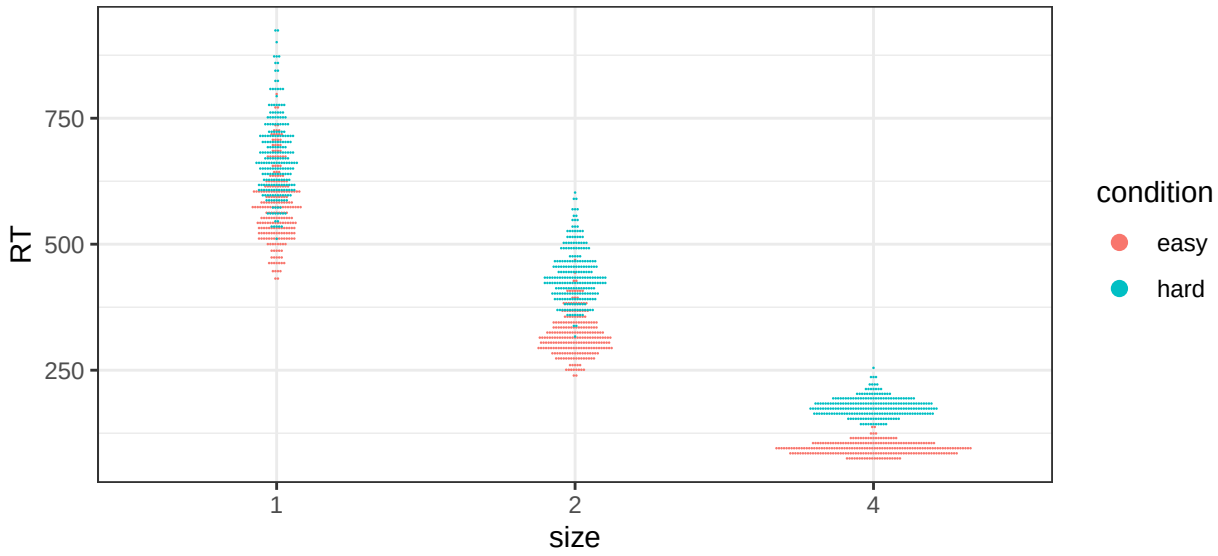


Notice the clear 'stripes' in the data

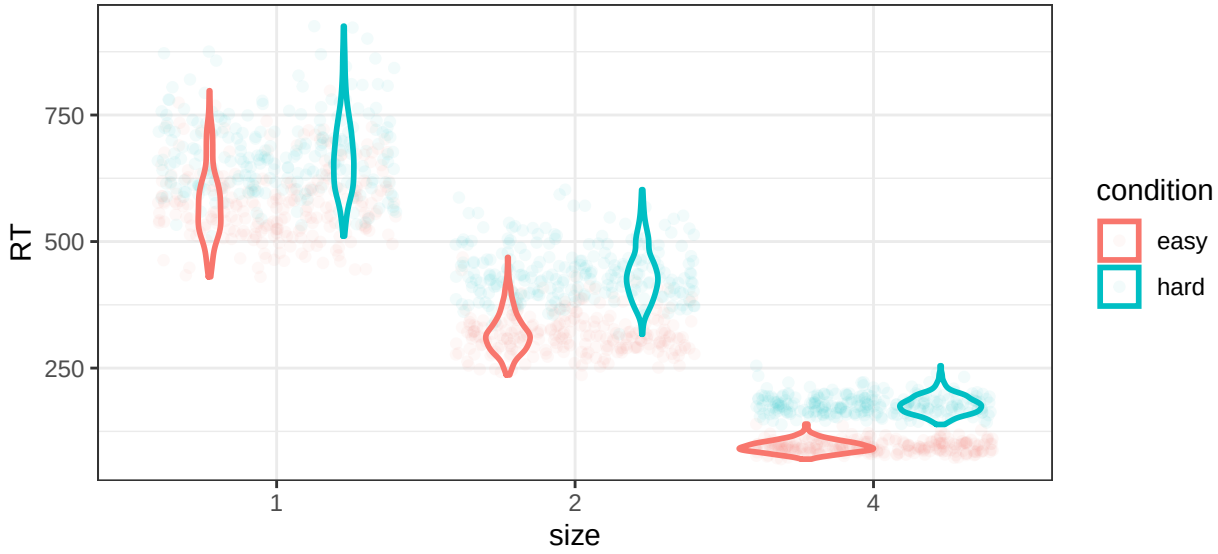
4.1.2 Visualizing data summaries

While it is often useful to see the raw data, this *alone* can be insufficient to communicate important aspects of the data. Specifically, we often are interested in whether the distributions of two conditions (e.g., the number size and easy vs. hard conditions) differ. The plots we have seen so far *suggest* that this might be the case, but we can't be certain. In order to convey information about the distribution of our data, we often provide data *summaries* instead of—or in addition to—the raw data. Whenever possible, I would highly recommend to go with the "in addition to" option. There is a lof of value in seeing the raw data, as long as the overall tendencies we are interested in can also be recognized.

Data can be summarized down to various levels of abstraction. For example, we can *bin* the data, and report the binned 'raw' data:

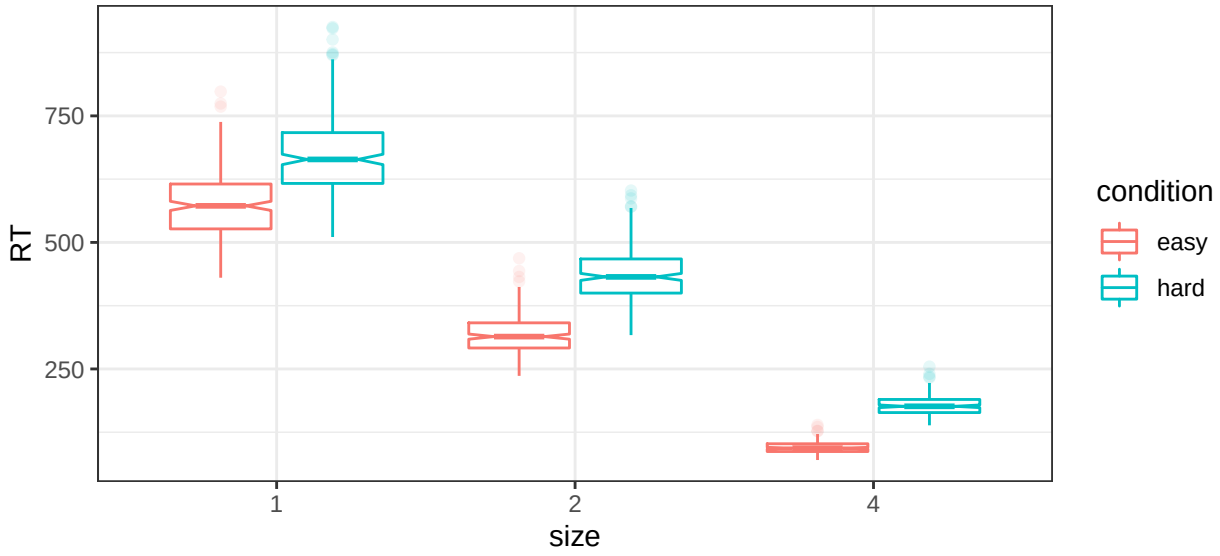


Dotplots can be an effective means for a categorical outcome variable (or the proportions resulting from them). Try it out for plotting the correctness of responses, rather than RTs. For a continuous outcome variables, however, dotplots are difficult to read (as in the RT plot above), and you might be better of with plotting the histogram or density of the distribution. This can elegantly be done by adding so-called violins to the raw data:

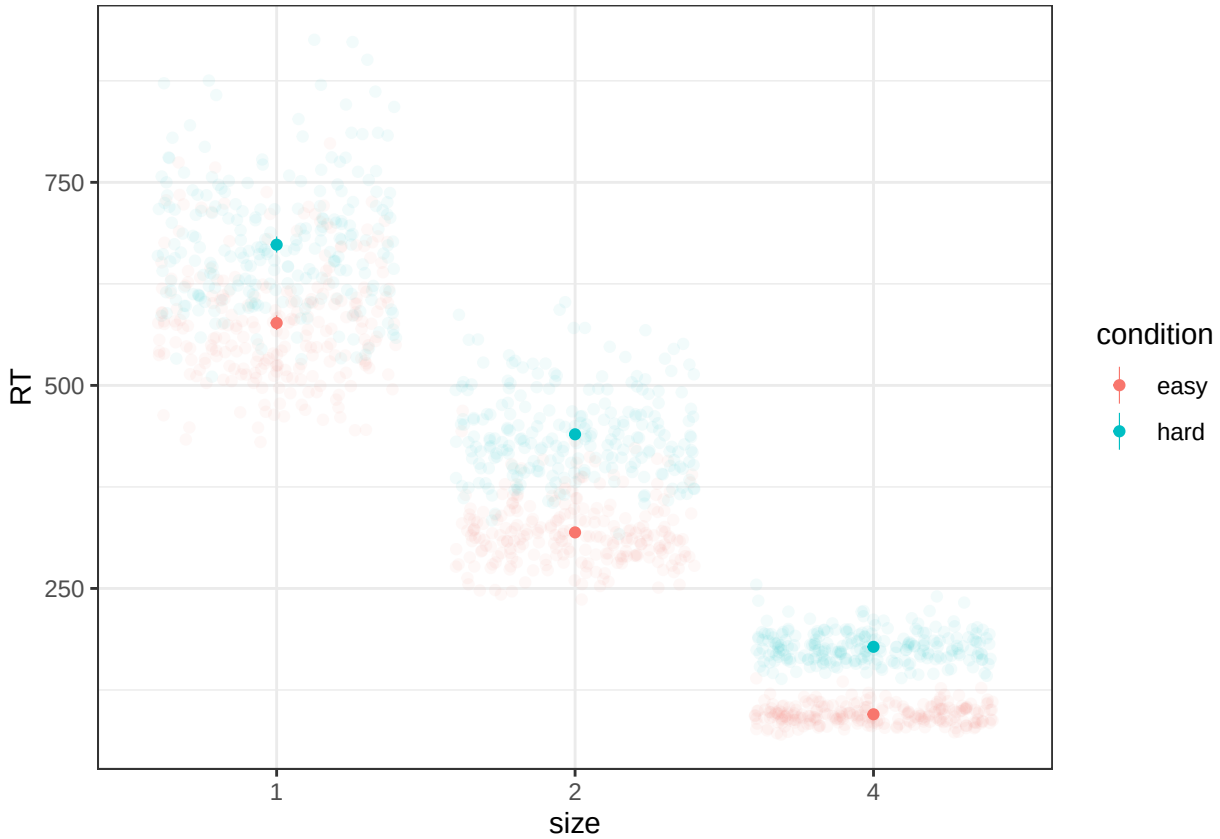


Some researchers also like dotplots, which try to provide a high-level summary of the data while also providing *some* information about the general distribution of the data and any potential outliers. Boxplots tend to be plotted *instead* of the raw data since boxplots already contain any raw data that is not captured well by the summary they provide (the outlier points). The line in the center of the box provides the median of the data in that condition. The upper and lower end of the box indicate the "inter-quartile range (IQR) from the 25th to the 75th quantile of the outcome (you can use the command quantiles() to get the quantiles yourself, if you want to compare them to what the plot provides). The whiskers (the lines below and above the box) extend to the largest and smallest value at most 1.5-times larger/smaller than the end of the box. Just as the height of the box provides a summary of the middle 50% of the data (the IQR), the whiskeys give an idea of the distribution beyond that middle part. Any points outside of that range are plotted and are referred to as"outliers". Notably all outliers in the present case are high RT outliers, not low RT outliers. This is quite common for variables that have highly skewed distributions: RTs have a lower, but no upper bound.

Finally, the notches (indentation on the sides of the box) provide the 95% confidence intervals around the means (CIs. We'll learn later *how CIs are obtained*, and what exactly they mean. For now, it suffices to say that (appropriately obtained) CIs are a critical visual component in plots when we want to know whether the means differ between conditions.



In publications, you will find that the data is often summarized even further into just the central tendency (mean) and its 95% CIs. Here we plot the mean and 95% CIs on top of the raw data. In this case, we have so much data that the CIs are very narrow and not even recognizable in the plot:



Here have used a pointrange to summarize the data. Alternatives include linerange (geom_linerange; no

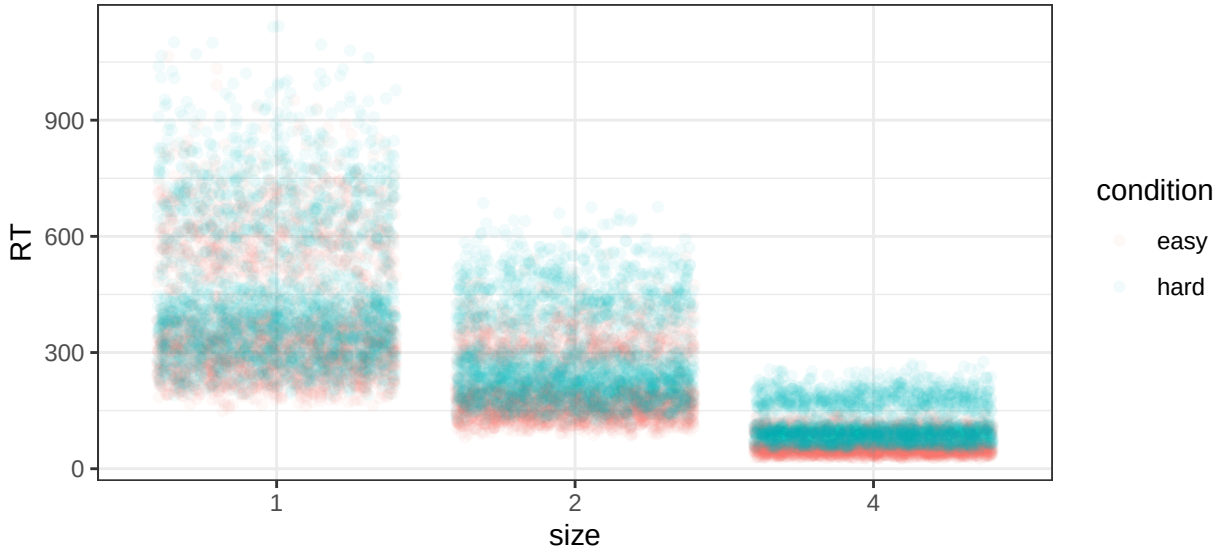
mean shown) or bar graphs with error intervals (geom_bar and geom_errorbar).

It is important that we summarize the data at an appropriate level when obtained CIs. We will cover this in more detail in a later class, but for now remember that what we do here is ok if we're looking at the data from one subject, but that we need to do things a bit differently when we look at the combined data from multiple subjects.

4.1.3 Plotting data from multiple subjects

So far we have plotted just one subject. Notice what happens when we update the scatter plot with the data from *all* subjects. There are now multiple 'stripes' visible in the data. Think about why that is the case. Any ideas?

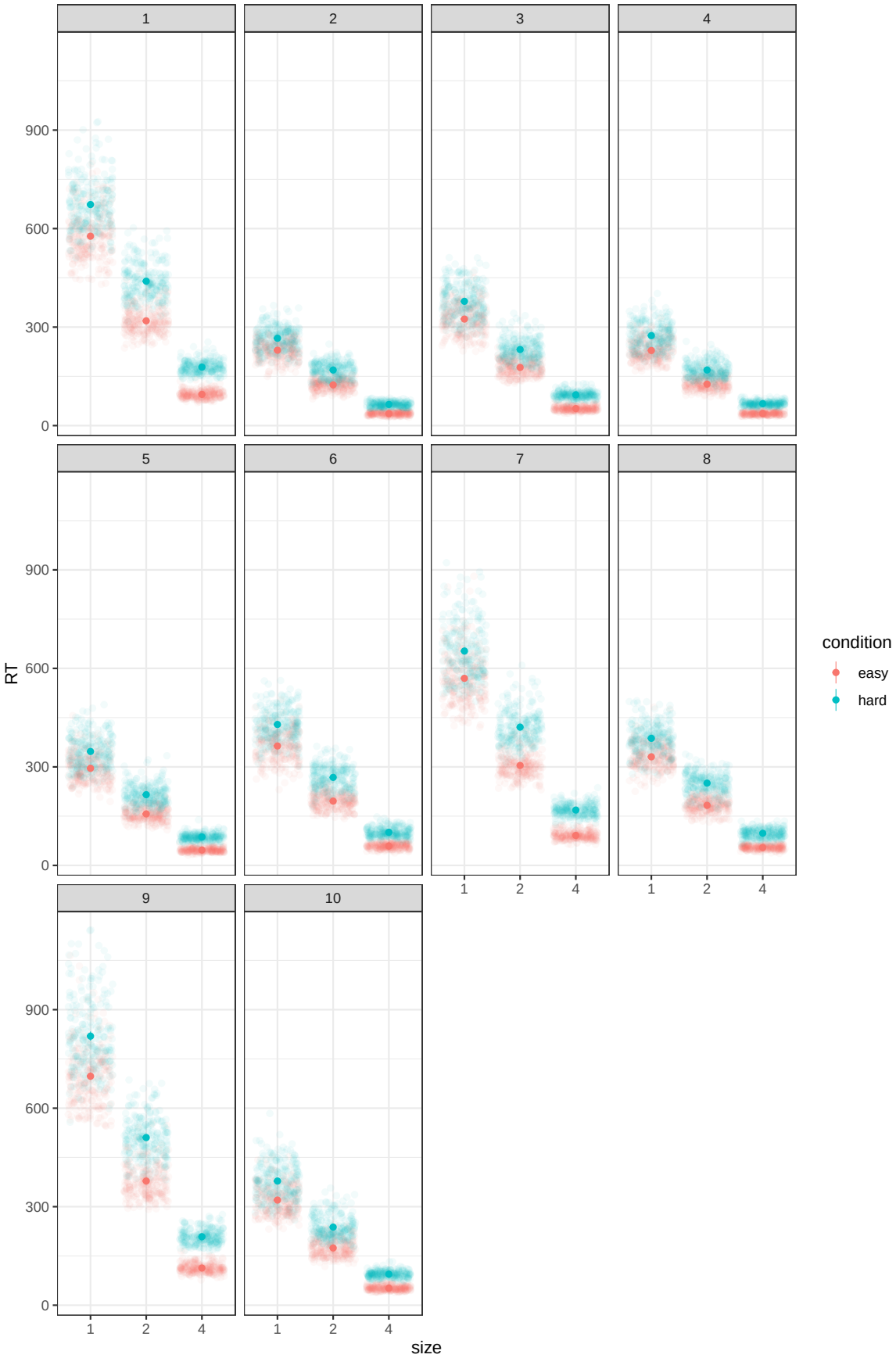
```
# We can update a plot with new data using %+%
p = p %+%
  d

plot(p)
```



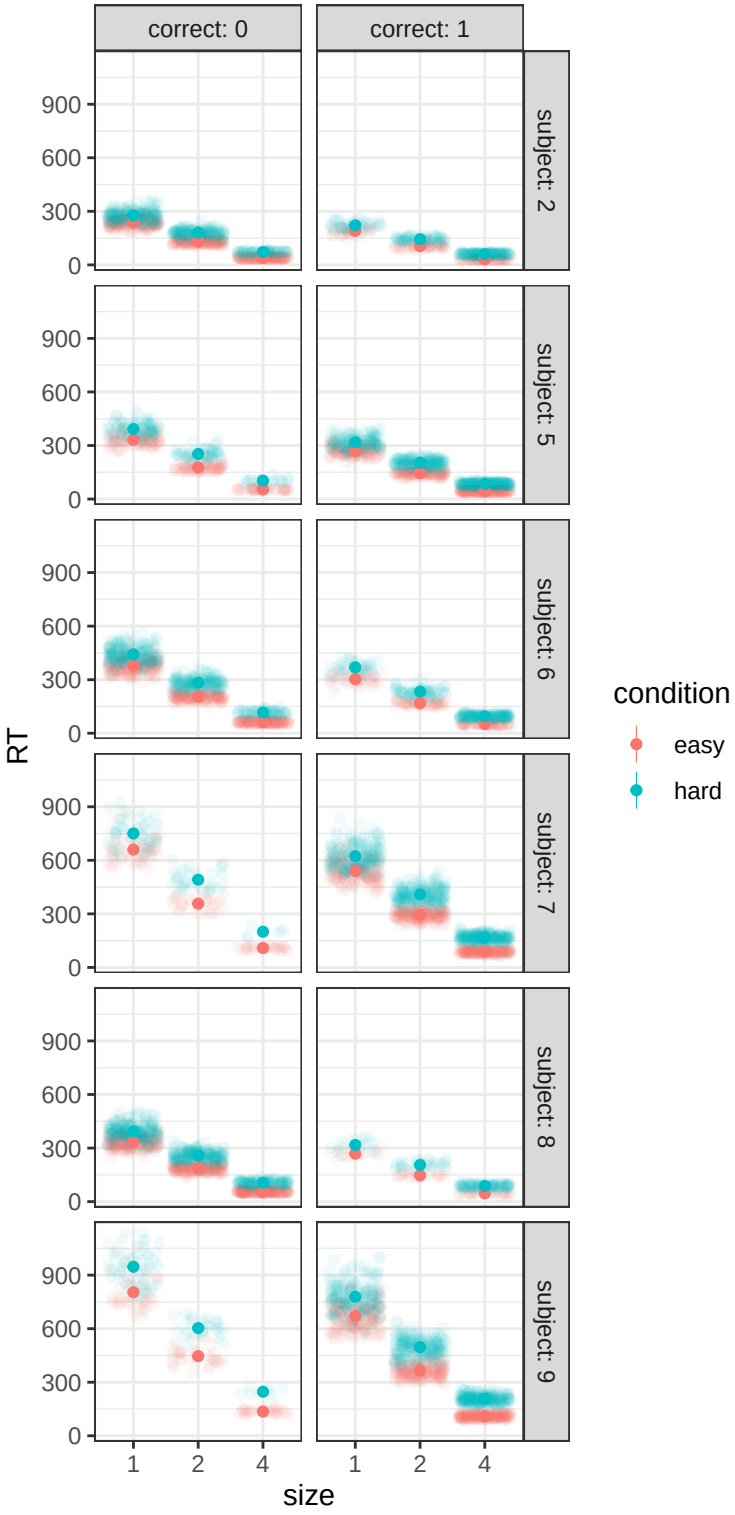
Correct: subjects differ from each other. This includes how fast they are overall, and might also include differences in how they are affected by the experimental conditions. This also means that the individual observations we are *not* independent of each other because the data points from the same subject resemble each other. We talk about "repeated measures" from the subject and call such data "repeated measures data". Another term that is used to refer to such data is *hierarchical* data or *hierarchically organized() data. Whenever we deal with such data, we need to be careful, for example, when we calculate CIs. We will return to that issue in our class on confidence intervals. For now, the next section focuses on how we can effectively visualize the data for all subjects without having to make many individual plots.

4.1.4 Facets

If we want to have separate panels conditional on another variable, we can do so through *facet*s. There are two major facet functions, *facet_wrap* (to have panels conditional on one variable) and *facet_grid* (conditional on two variables). For example, we can have separate panels for each subject:

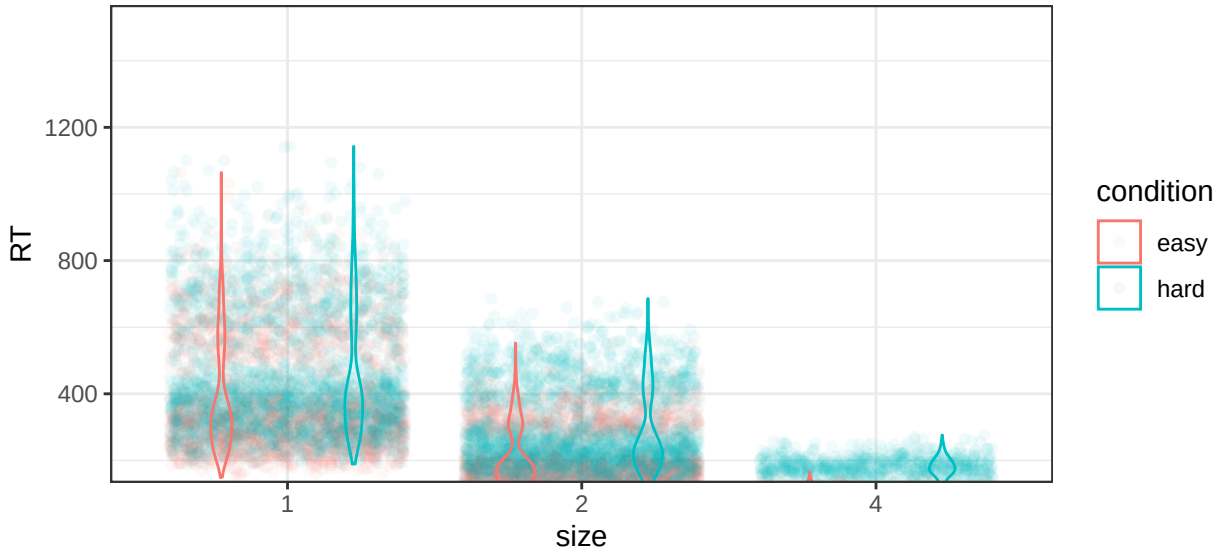

1
2
3
4
5
6
7
8
9
10
RT
900
600
300
0
1
2
4
size
condition
easy
hard

Or we could show by-subject RTs in two columns, separately for false and correct answers. Here, we do so after first sampling 6 random subjects (since the plot would otherwise be rather large).



4.1.5 Scales and coordinate systems

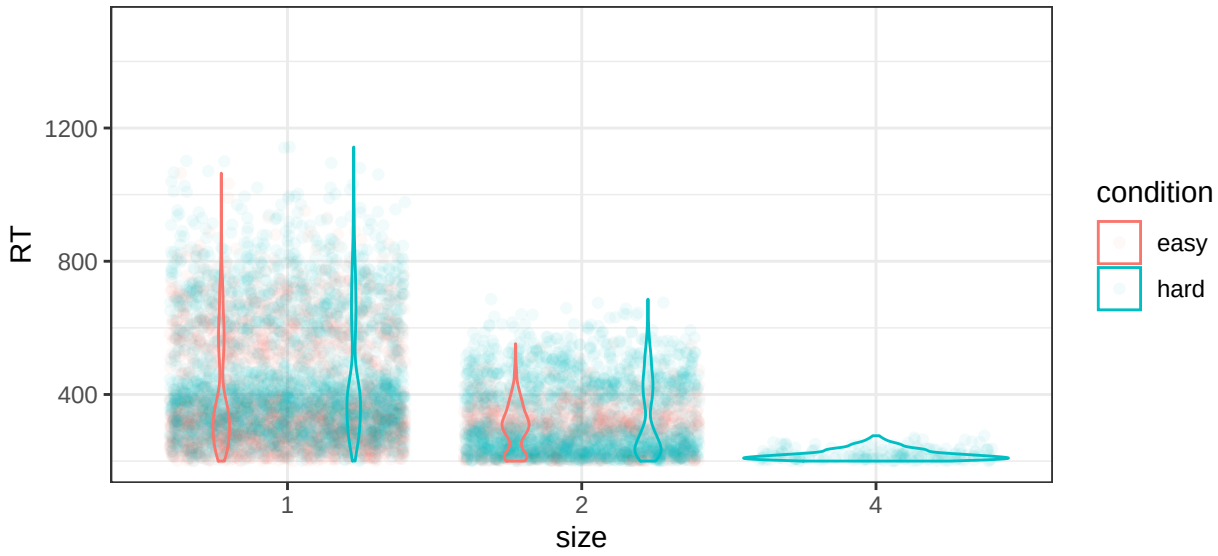
Sometimes we don't want to see all of the data, or we want to zoom into some ranges of our data. We can do so by explicitly specifying the x- and y-limits of our coordinate system:



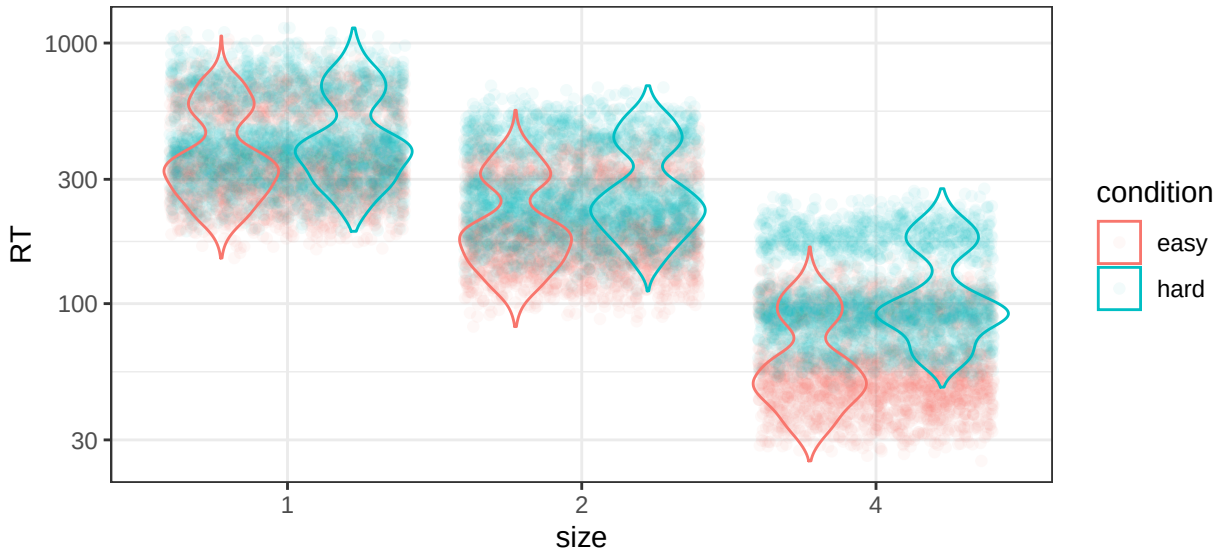
Note that this zooms into parts of our data without excluding any data (e.g., from the calculation of the violins, which have the same shape as above). If we want to exclude data, transform data or in other ways change the way the aesthetical mappings are interpreted, this is achieved through *scales*. For example, the following *ex*cludes all RTs below 300 and above 2000. Note how that changes the violin plots (as it should: they estimate the ditribution of RTs):

```
## Warning: Removed 6740 rows containing non-finite values (stat_ydensity).
```

```
## Warning: Removed 6740 rows containing missing values (geom_point).
```



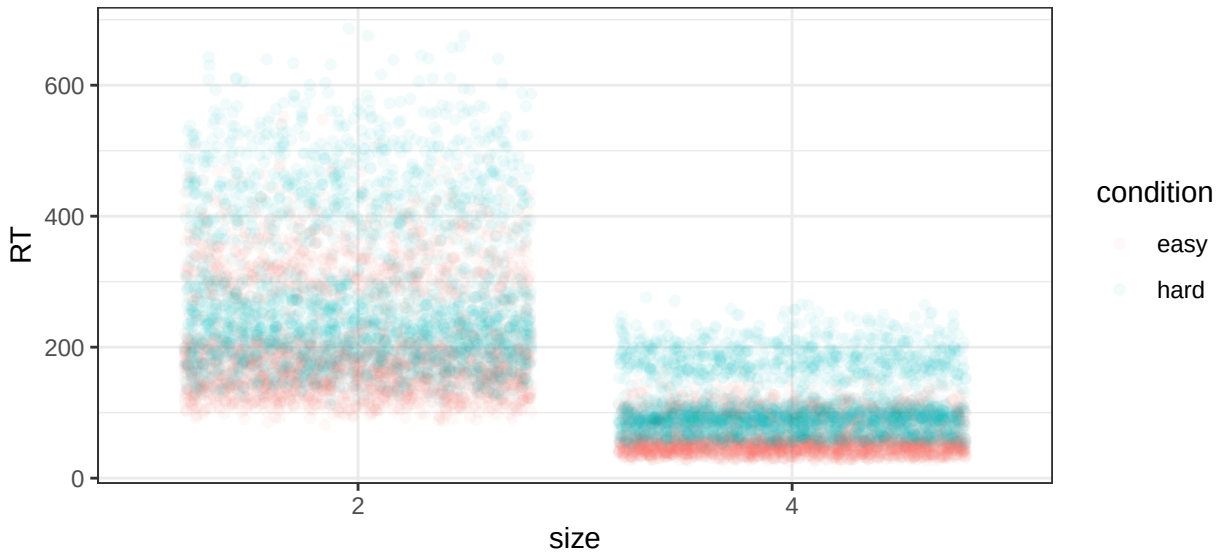
More generally, **scales are applied before *ggplot2* applies any statistics, whereas coordinate systems do only affect how we see the data**. This is important to remember. For example, since reaction times often have distributions that are more lognormal, rather than normal, let's update our original plot to use a log-transformed y-axis. This does not only affect how we *see* the data, it also affects how teh densities that are plotted by the violin geom are estimated:



4.2 Pipes (again)

Of course, we can use pipes to pipe the data frame into the plotting function, optionally after first piping the data through some additional *dplyr* operations (since the output of that entire pipe is again a data frame):

```
p = d %>%
  filter(size != "1") %>%
  ggplot(
    aes(
      x = size,
      y = RT,
      color = condition)) +
  geom_point(alpha = .05, position = position_jitter())

plot(p)
```



4.3 Exercises

Remember ggplot2 cheatsheet!

- Plot a histogram of the RTs by number size. Make on version where you plots the histograms in different facets, and another version where you have only one facet and use fill color to distinguish between number sizes. (*geom_histogram*)
- Plot the average proportion of correct answers by number size as a pointrange.
- Do the same, but first average by subject and number size, and then plot the average (and confidence interval) of those by-subject averages of correct responses.
- Try to make a pie chart that shows the proportion correct for the three number sizes. (*coord_polar*)

5 Working out the details of your visualizations (Next week)

- making x-axis have right units
- color choices
- axis labels
- legends
- panel grid, etc.

5.1 Themes

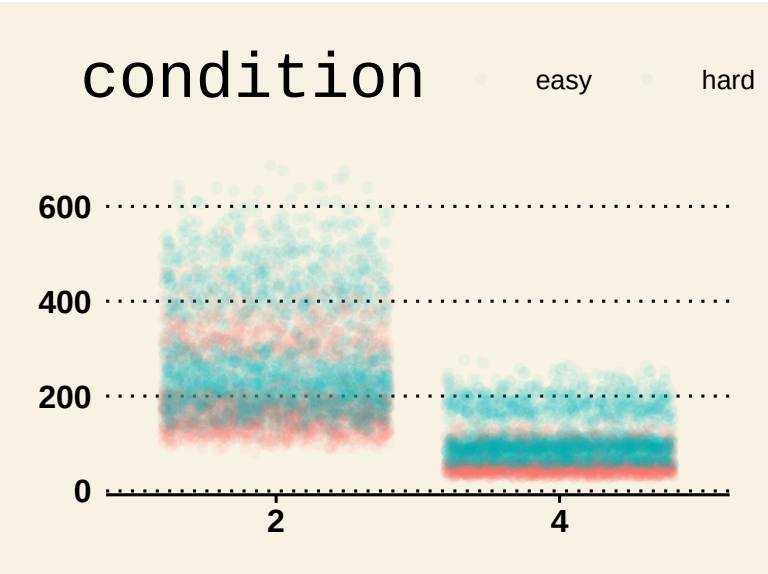
Themes can be used to customize any detail of a plot. There are many customized themes, including in additional libraries (e.g., *ggthemes*). Here a few examples:

```
library(ggthemes)
```

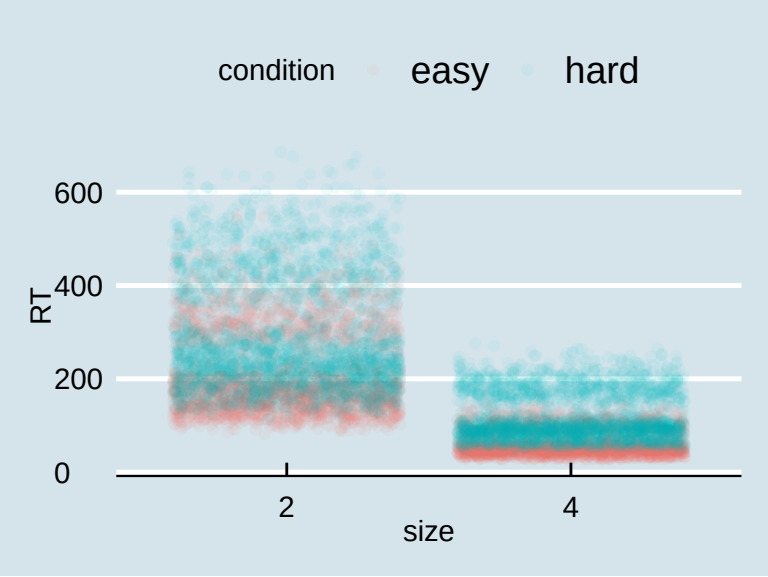
```
##
## Attaching package: 'ggthemes'
```

```
## The following object is masked from 'package:cowplot':
##
##     theme_map
```

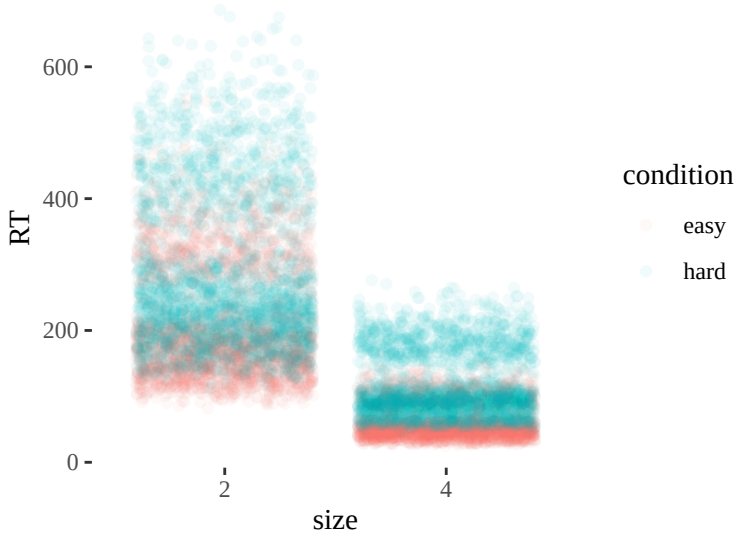
```
p + theme_wsj()
```



```
p + theme_economist()
```



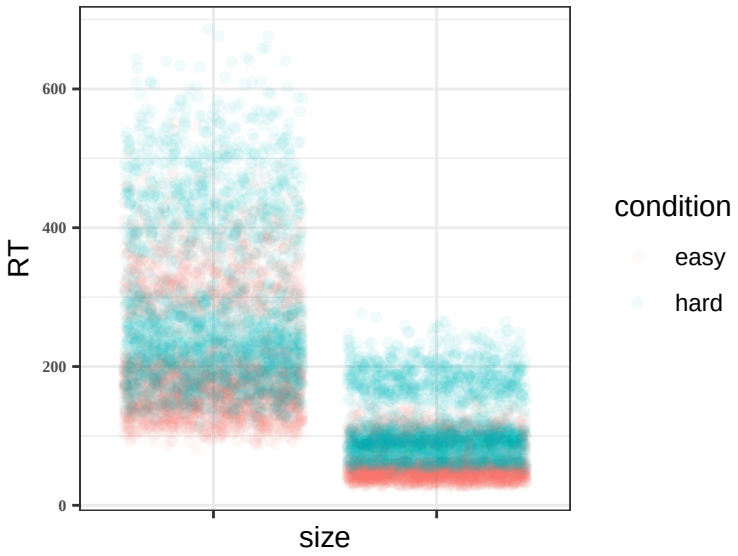

```
p + theme_tufte()
```




```
# etc.
```

If fact, any ggplot has a theme, even if you don't specify one. In that case, *ggplot2* uses the default theme. You can also create your own themes, or modify any aspect of the present theme. Check out *?theme for more information. For example, to remove the x-axis labels, and to change the font of the y-axis labels:

```
p + theme(
  axis.text.x = element_blank(),
  axis.text.y = element_text(family = "Times", face = "bold", size = 6))
```



Elements that can be modified or removed include any text elements in the axis or legend (alignment, angle, size, font, emphasis, etc.), the fill or border lines of any element (incl. the background color of the panel or the entire plot area), the padding, margins, and other spacing between the elements, any tick marks on the axis, any grid lines, any aspects of the legend its orientation, shape, etc. It's worth looking into the helpfile or a *ggplot2* overview, as the themes also contain many elements that are not even visible in the default.

5.2 Other libraries

There's a huge number of additional libraries that build on *ggplot2*, and they keep changing. Many of the libraries introduce additional geoms, incl. single-line commands for complex plots (e.g., k-by-k correlation plots; trees diagrams; field-specific plot types like map types, dentrograms, ridge plots, 3D plots). Other packages extend what *ggplot2* can do or make it easier to use.

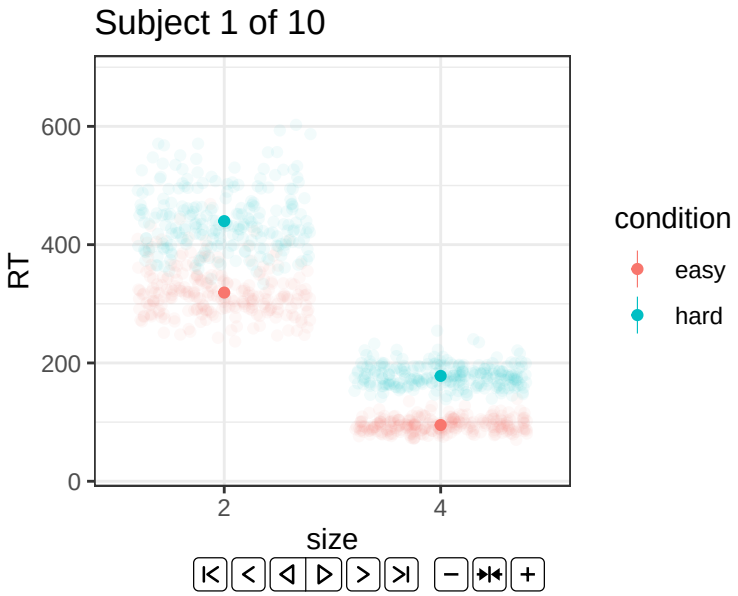
Here I just note a few:

- Extend functionality:
- gganimate for animations, movies, gifs, etc.
- plotly to turn any ggplot into an interactive graph
- cowplot to combine multiple ggplots into a single figure
- Make it easier to do things that ggplot can already do:
  - ggforce to add powerful functionality
  - ggthemes to add themes
  - gggraph to visualize graphs
  - ggeffects to help with plotting model summaries
  - ggsignif to add significance brackets (seen in many publications)
  - ggpubr to make ggplots publication ready
  - ggsci to add scientifically themed palettes
  - viridis to add highly contrastive color schemes (beautiful)
- Shiny to make web-based apps in which users can interact with your plots.

For further inspiritation, check out these galaries:

- Gallery of R plots
- List of ggplot2 extensions

For example, instead of faceting our data by subject, we might animate it by subject. Gganimate can create movies, but here I am embedding the animation directly into the PDF. This will only work if you work the document in Acrobat Reader. Try it out:



6 Session info

```
## - Session info ---------------------------------------------------------------
##  setting  value
##  version  R version 4.0.2 (2020-06-22)
##  os       macOS High Sierra 10.13.6
##  system   x86_64, darwin17.0
##  ui       X11
##  language (EN)
##  collate  en_US.UTF-8
##  ctype    en_US.UTF-8
##  tz       America/New_York
##  date     2020-11-01
##
## - Packages -------------------------------------------------------------------
##  package      * version    date       lib source
##  assertthat     0.2.1      2019-03-21 [1] CRAN (R 4.0.2)
##  backports      1.1.9      2020-08-24 [1] CRAN (R 4.0.2)
##  base64enc      0.1-3      2015-07-28 [1] CRAN (R 4.0.2)
##  blob           1.2.1      2020-01-20 [1] CRAN (R 4.0.2)
##  broom          0.7.0      2020-07-09 [1] CRAN (R 4.0.2)
##  callr          3.4.4      2020-09-07 [1] CRAN (R 4.0.2)
##  cellranger     1.1.0      2016-07-27 [1] CRAN (R 4.0.2)
##  checkmate      2.0.0      2020-02-06 [1] CRAN (R 4.0.2)
##  cli            2.0.2      2020-02-28 [1] CRAN (R 4.0.2)
##  cluster        2.1.0      2019-06-19 [1] CRAN (R 4.0.2)
##  colorspace     1.4-1      2019-03-18 [1] CRAN (R 4.0.2)
##  cowplot      * 1.1.0      2020-09-08 [1] CRAN (R 4.0.2)
##  crayon         1.3.4      2017-09-16 [1] CRAN (R 4.0.2)
##  data.table     1.13.0     2020-07-24 [1] CRAN (R 4.0.2)
```

```
##  DBI            1.1.0      2019-12-15 [1] CRAN (R 4.0.2)
##  dbplyr         1.4.4      2020-05-27 [1] CRAN (R 4.0.2)
##  desc           1.2.0      2018-05-01 [1] CRAN (R 4.0.2)
##  devtools       2.3.1      2020-07-21 [1] CRAN (R 4.0.2)
##  digest         0.6.25     2020-02-23 [1] CRAN (R 4.0.2)
##  dplyr        * 1.0.2      2020-08-18 [1] CRAN (R 4.0.2)
##  ellipsis       0.3.1      2020-05-15 [1] CRAN (R 4.0.2)
##  evaluate       0.14       2019-05-28 [1] CRAN (R 4.0.1)
##  fansi          0.4.1      2020-01-08 [1] CRAN (R 4.0.2)
##  farver         2.0.3      2020-01-16 [1] CRAN (R 4.0.2)
##  forcats      * 0.5.0      2020-03-01 [1] CRAN (R 4.0.2)
##  foreign        0.8-80     2020-05-24 [1] CRAN (R 4.0.2)
##  Formula        1.2-3      2018-05-03 [1] CRAN (R 4.0.2)
##  fs             1.5.0      2020-07-31 [1] CRAN (R 4.0.2)
##  generics       0.0.2      2018-11-29 [1] CRAN (R 4.0.2)
##  gganimate    * 1.0.6      2020-07-08 [1] CRAN (R 4.0.2)
##  ggplot2      * 3.3.2      2020-06-19 [1] CRAN (R 4.0.2)
##  ggthemes     * 4.2.0      2019-05-13 [1] CRAN (R 4.0.2)
##  gifski         0.8.6      2018-09-28 [1] CRAN (R 4.0.2)
##  glue           1.4.2      2020-08-27 [1] CRAN (R 4.0.2)
##  gridExtra      2.3        2017-09-09 [1] CRAN (R 4.0.2)
##  gtable         0.3.0      2019-03-25 [1] CRAN (R 4.0.2)
##  haven          2.3.1      2020-06-01 [1] CRAN (R 4.0.2)
##  Hmisc          4.4-1      2020-08-10 [1] CRAN (R 4.0.2)
##  hms            0.5.3      2020-01-08 [1] CRAN (R 4.0.2)
##  htmlTable      2.0.1      2020-07-05 [1] CRAN (R 4.0.2)
##  htmltools      0.5.0      2020-06-16 [1] CRAN (R 4.0.2)
##  htmlwidgets    1.5.1      2019-10-08 [1] CRAN (R 4.0.2)
##  httr           1.4.2      2020-07-20 [1] CRAN (R 4.0.2)
##  jpeg           0.1-8.1    2019-10-24 [1] CRAN (R 4.0.2)
##  jsonlite       1.7.1      2020-09-07 [1] CRAN (R 4.0.2)
##  knitr          1.29.4     2020-08-23 [1] Github (yihui/knitr@bd64f2e)
##  labeling       0.3        2014-08-23 [1] CRAN (R 4.0.2)
##  lattice        0.20-41    2020-04-02 [1] CRAN (R 4.0.2)
##  latticeExtra   0.6-29     2019-12-19 [1] CRAN (R 4.0.2)
##  lazyeval       0.2.2      2019-03-15 [1] CRAN (R 4.0.2)
##  lifecycle      0.2.0      2020-03-06 [1] CRAN (R 4.0.2)
##  lubridate      1.7.9      2020-06-08 [1] CRAN (R 4.0.2)
##  magrittr     * 1.5        2014-11-22 [1] CRAN (R 4.0.2)
##  Matrix         1.2-18     2019-11-27 [1] CRAN (R 4.0.2)
##  memoise        1.1.0      2017-04-21 [1] CRAN (R 4.0.2)
##  modelr         0.1.8      2020-05-19 [1] CRAN (R 4.0.2)
##  munsell        0.5.0      2018-06-12 [1] CRAN (R 4.0.2)
##  nnet           7.3-14     2020-04-26 [1] CRAN (R 4.0.2)
##  pillar         1.4.6      2020-07-10 [1] CRAN (R 4.0.2)
##  pkgbuild       1.1.0.9000 2020-08-06 [1] Github (r-lib/pkgbuild@3a87bd9)
##  pkgconfig      2.0.3      2019-09-22 [1] CRAN (R 4.0.2)
##  pkgload        1.1.0      2020-05-29 [1] CRAN (R 4.0.2)
##  plotly       * 4.9.2.1    2020-04-04 [1] CRAN (R 4.0.2)
##  plyr           1.8.6      2020-03-03 [1] CRAN (R 4.0.2)
##  png            0.1-7      2013-12-03 [1] CRAN (R 4.0.2)
##  prettyunits    1.1.1      2020-01-24 [1] CRAN (R 4.0.2)
##  processx       3.4.4      2020-09-03 [1] CRAN (R 4.0.2)
##  progress       1.2.2      2019-05-16 [1] CRAN (R 4.0.2)
```

```
##  ps              1.3.4      2020-08-11 [1] CRAN (R 4.0.2)
##  purrr         * 0.3.4      2020-04-17 [1] CRAN (R 4.0.2)
##  R.matlab      * 3.6.2      2018-09-27 [1] CRAN (R 4.0.2)
##  R.methodsS3     1.8.1      2020-08-26 [1] CRAN (R 4.0.2)
##  R.oo            1.24.0     2020-08-26 [1] CRAN (R 4.0.2)
##  R.utils         2.10.1     2020-08-26 [1] CRAN (R 4.0.2)
##  R6              2.4.1      2019-11-12 [1] CRAN (R 4.0.2)
##  RColorBrewer    1.1-2      2014-12-07 [1] CRAN (R 4.0.2)
##  Rcpp            1.0.5      2020-07-06 [1] CRAN (R 4.0.2)
##  readr         * 1.3.1      2018-12-21 [1] CRAN (R 4.0.2)
##  readxl        * 1.3.1      2019-03-13 [1] CRAN (R 4.0.2)
##  remotes         2.2.0      2020-07-21 [1] CRAN (R 4.0.2)
##  reprex          0.3.0      2019-05-16 [1] CRAN (R 4.0.2)
##  rlang           0.4.7      2020-07-09 [1] CRAN (R 4.0.2)
##  rmarkdown       2.3        2020-06-18 [1] CRAN (R 4.0.2)
##  rpart           4.1-15     2019-04-12 [1] CRAN (R 4.0.2)
##  rprojroot       1.3-2      2018-01-03 [1] CRAN (R 4.0.2)
##  rstudioapi      0.11       2020-02-07 [1] CRAN (R 4.0.2)
##  rvest           0.3.6      2020-07-25 [1] CRAN (R 4.0.2)
##  scales          1.1.1      2020-05-11 [1] CRAN (R 4.0.2)
##  sessioninfo     1.1.1      2018-11-05 [1] CRAN (R 4.0.2)
##  stringi         1.4.6      2020-02-17 [1] CRAN (R 4.0.2)
##  stringr       * 1.4.0      2019-02-10 [1] CRAN (R 4.0.2)
##  survival        3.2-3      2020-06-13 [1] CRAN (R 4.0.2)
##  testthat        2.3.2      2020-03-02 [1] CRAN (R 4.0.2)
##  tibble        * 3.0.3      2020-07-10 [1] CRAN (R 4.0.2)
##  tidyr         * 1.1.2      2020-08-27 [1] CRAN (R 4.0.2)
##  tidyselect      1.1.0      2020-05-11 [1] CRAN (R 4.0.2)
##  tidyverse     * 1.3.0      2019-11-21 [1] CRAN (R 4.0.2)
##  tweenr          1.0.1      2018-12-14 [1] CRAN (R 4.0.2)
##  usethis         1.6.1      2020-04-29 [1] CRAN (R 4.0.2)
##  utf8            1.1.4      2018-05-24 [1] CRAN (R 4.0.2)
##  vctrs           0.3.4      2020-08-29 [1] CRAN (R 4.0.2)
##  viridisLite     0.3.0      2018-02-01 [1] CRAN (R 4.0.1)
##  withr           2.2.0      2020-04-20 [1] CRAN (R 4.0.2)
##  xfun            0.17       2020-09-09 [1] CRAN (R 4.0.2)
##  xml2            1.3.2      2020-04-23 [1] CRAN (R 4.0.2)
##  yaml            2.2.1      2020-02-01 [1] CRAN (R 4.0.2)
##
## [1] /Library/Frameworks/R.framework/Versions/4.0/Resources/library
```