Data visualisation using R, for researchers who don't use R

Emily Nordmann[1], Phil McAleer[1], Wilhelmiina Toivo[1], Helena Paterson[1], & Lisa M. DeBruine[2]

[1] School of Psychology, University of Glasgow
[2] Institute of Neuroscience and Psychology, University of Glasgow

Abstract

In addition to benefiting reproducibility and transparency, one of the advantages of using R is that researchers have a much larger range of fully customisable data visualisations options than are typically available in point-and-click software, due to the open-source nature of R. These visualisation options not only look attractive, but can increase transparency about the distribution of the underlying data rather than relying on commonly used visualisations of aggregations such as bar charts of means. In this tutorial, we provide a practical introduction to data visualisation using R, specifically aimed at researchers who have little to no prior experience of using R. First we detail the rationale for using R for data visualisation and introduce the "grammar of graphics" that underlies data visualisation using the ggplot package. The tutorial then walks the reader through how to replicate plots that are commonly available in point-and-click software such as histograms and boxplots, as well as showing how the code for these "basic" plots can be easily extended to less commonly available options such as violin-boxplots. The dataset and code used in this tutorial as well as an interactive version with activity solutions, additional resources and advanced plotting options is available at https://osf.io/bj83f/.

Keywords: visualization, ggplot, plots, R
Word count: 11472

Introduction

Use of the programming language R (R Core Team, 2021) for data processing and statistical analysis by researchers is increasingly common, with an average yearly growth of 87% in the number of citations of the R Core Team between 2006-2018 (Barrett, 2019). In addition to benefiting reproducibility and transparency, one of the advantages of using R is that researchers have a much larger range of fully customisable data visualisations options than are typically available in point-and-click software, due to the open-source nature of R. These visualisation options not only look attractive, but can increase transparency about the distribution of the underlying data rather than relying on commonly used visualisations of aggregations such as bar charts of means (Newman & Scholl, 2012).

Yet, the benefits of using R are obscured for many researchers by the perception that coding skills are difficult to learn (Robins, Rountree, & Rountree, 2003). Coupled with this, only a minority of psychology programmes currently teach coding skills (Wills, n.d.) with the majority of both undergraduate and postgraduate courses using proprietary point-and-click software such as SAS, SPSS or Microsoft Excel. While the sophisticated use of proprietary software often necessitates the use of computational thinking skills akin to coding (for instance SPSS scripts or formulas in Excel), we have found that many researchers do not perceive that they already have introductory coding skills. In the following tutorial we intend to change that perception by showing how experienced researchers can redevelop their existing computational skills to utilise the powerful data visualisation tools offered by R.

In this tutorial, we aim to provide a practical introduction to data visualisation using R, specifically aimed at researchers who have little to no prior experience of using R. First we detail the rationale for using R for data visualisation and introduce the "grammar of graphics" that underlies data visualisation using the `ggplot2` package. The tutorial then walks the reader through how to replicate plots that are commonly available in point-and-click software such as histograms and boxplots, as well as showing how the code for these "basic" plots can be easily extended to less commonly available options such as violin-boxplots.

Why R for data visualisation?

Data visualisation benefits from the same advantages as statistical analysis when writing code rather than using point-and-click software – reproducibility and transparency. The need for psychological researchers to work in reproducible ways has been well-documented and discussed in response to the replication crisis (e.g. Munafò et al., 2017) and we will not repeat those arguments here. However, there is an additional benefit to reproducibility that is less frequently acknowledged compared to the loftier goals of improving psychological science: if you write code to produce your plots, you can reuse and adapt that code in the future rather than starting from scratch each time.

In addition to the benefits of reproducibility, using R for data visualisation gives the researcher almost total control over each element of the plot. Whilst this flexibility can seem daunting at first, the ability to write reusable code recipes (and use recipes created

₅₀ by others) is highly advantageous. The level of customisation and the professional outputs
₅₁ available using R has, for instance, lead news outlets such as the BBC (Visual & Journalism,
₅₂ 2019) and the New York Times (Bertini & Stefaner, 2015) to adopt R as their preferred
₅₃ data visualisation tool.

₅₄ **A layered grammar of graphics**

₅₅   There are multiple approaches to data visualisation in R; in this paper we use the
₅₆ popular package[1] `ggplot2` (Wickham, 2016a) which is part of the larger `tidyverse`[2] (Wick-
₅₇ ham, 2017) collection of packages that provide functions for data wrangling, descriptives,
₅₈ and visualisation. A grammar of graphics (Wilkinson, Anand, & Grossman, 2005) is a stan-
₅₉ dardised way to describe the components of a graphic. `ggplot2` uses a layered grammar of
₆₀ graphics (Wickham, 2010), in which plots are built up in a series of layers. It may be helpful
₆₁ to think about any picture as having multiple elements that sit semi-transparently over each
₆₂ other. A good analogy is old Disney movies where artists would create a background and
₆₃ then add moveable elements on top of the background via transparencies.

₆₄   Figure 1 displays the evolution of a simple scatterplot using this layered approach.
₆₅ First, the plot space is built (layer 1); the variables are specified (layer 2); the type of
₆₆ visualisation (known as a `geom`) that is desired for these variables is specified (layer 3) - in
₆₇ this case `geom_point()` is called to visualise individual data points; a second geom is added
₆₈ to include a line of best fit (layer 4), the axis labels are edited for readability (layer 5), and
₆₉ finally, a theme is applied to change the overall appearance of the plot (layer 6).

---

[1]The power of R is that it is extendable and open source - put simply, if a function doesn't exist or is difficult to use, anyone can create a new **package** that contains data and code to allow you to perform new tasks. You may find it helpful to think of packages as additional apps that you need to download separately to extend the functionality beyond what comes with "Base R."

[2]Because there are so many different ways to achieve the same thing in R, when Googling for help with R, it is useful to append the name of the package or approach you are using, e.g., "how to make a histogram ggplot2."
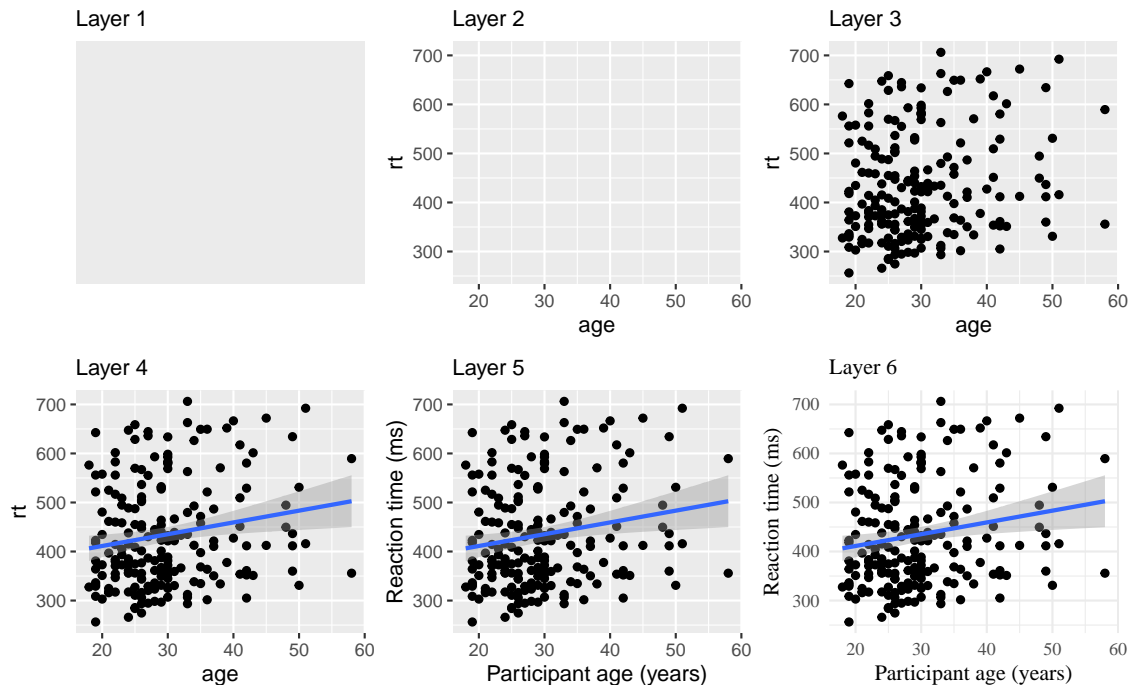
Figure 1. Evolution of a layered plot

⁷⁰   Importantly, each layer is independent and independently customisable. For example,
⁷¹ the size, colour and position of each component can be adjusted, or one could, for example,
⁷² remove the first geom (the data points) to only visualise the line of best fit, simply by
⁷³ removing the layer that draws the data points (Figure 2). The use of layers makes it easy
⁷⁴ to build up complex plots step-by-step, and to adapt or extend plots from existing code.
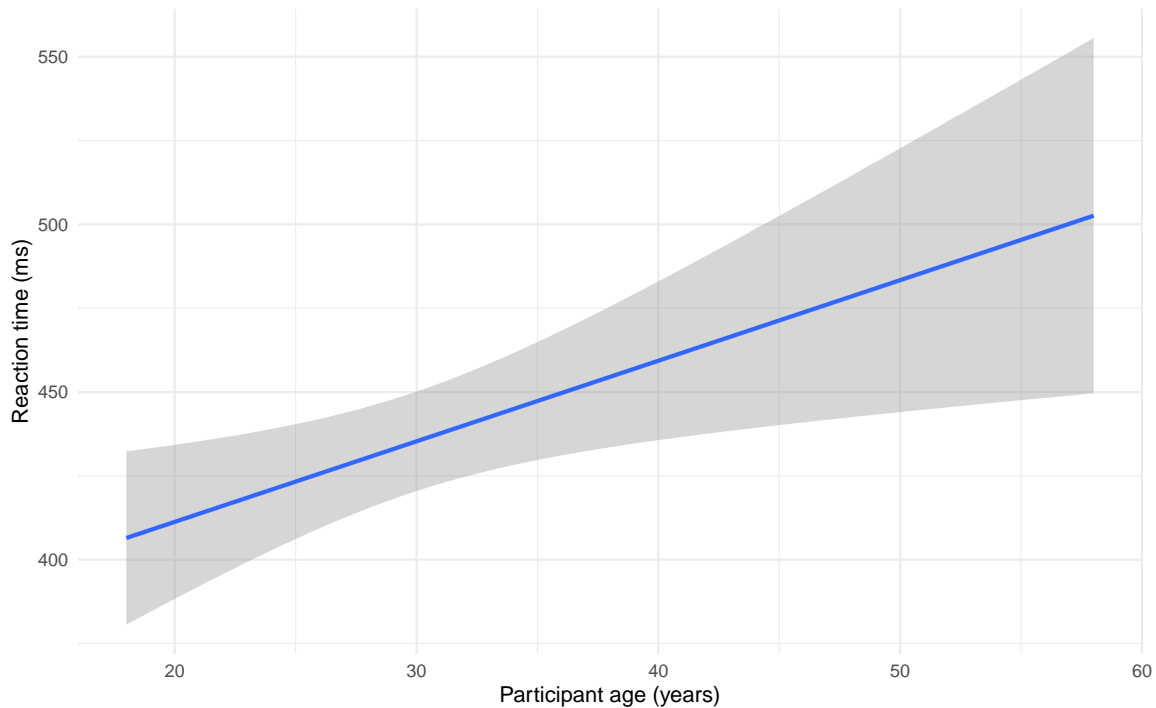
Figure 2. Plot with scatterplot layer removed.

Tutorial components

This tutorial contains three components.

1. A traditional PDF manuscript that can easily be saved, printed, and cited.
2. An online version of the tutorial published at https://psyteachr.github.io/introdataviz/ that may be easier to copy and paste code from and that also provides the optional activity solutions as well as additional appendices, including code tutorials for advanced plots beyond the scope of this paper and links to additional resources.
3. An Open Science Framework repository published at https://osf.io/bj83f/ that contains the simulated dataset (see below), preprint, and R Markdown workbook.

Simulated dataset

For the purpose of this tutorial, we will use simulated data for a 2 x 2 mixed-design lexical decision task in which participants have to decide whether a presented word is a real word, or a non-word, with 100 participants. There are 100 rows (1 for each participant) and 7 variables:

- Participant information:

    - `id`: Participant ID

92   – `age`: Age

93 • 1 between-subject independent variable (VI):

94   – `language`: Language group (1 = monolingual, 2 = bilingual)

95 • 4 columns for the 2 dependent variables (DVs) for RT and accuracy, crossed by the
96   within-subject IV of condition:

97   – `rt_word`: Reaction time (ms) for word trials
98   – `rt_nonword`: Reaction time (ms) for non-word trials
99   – `acc_word`: Accuracy for word trials
100   – `acc_nonword`: Accuracy for non-word trials

101   For newcomers to R, we would suggest working through this tutorial with the simu-
102 lated dataset, then extending the code to your own datasets with a similar structure, and
103 finally generalising the code to new structures and problems.

104 **Setting up R and RStudio**

105   We strongly encourage the use of RStudio (RStudio Team, 2021) to write code in R.
106 R is the programming language whilst RStudio is an *integrated development environment*
107 that makes working with R easier. More information on installing both R and RStudio can
108 be found in the additional resources.

109   Projects are a useful way of keeping all your code, data, and output in one place.
110 To create a new project, open RStudio and click `File - New Project - New Directory`
111 `- New Project`. You will be prompted to give the project a name, and select a location
112 for where to store the project on your computer. Once you have done this, click `Create`
113 `Project`. Download the simulated dataset and code tutorial Rmd file from the online
114 materials and then (`ldt_data.csv`, `workbook.Rmd`) to this folder. The files pane on the
115 bottom right of RStudio should now display this folder and the files it contains - this is
116 known as your *working directory* and it is where R will look for any data you wish to import
117 and where it will save any output you create.

118   This tutorial will require you to use the packages contained with the `tidyverse`
119 collection. Additionally, we will also require use of `patchwork`. To install these packages,
120 copy and paste the below code into the console (the left hand pane) and press enter to
121 execute the code.

```r
# only run in the console, never put this in a script
package_list <- c("tidyverse", "patchwork")
install.packages(package_list)
```

122   The R Markdown workbook available in the online materials contains all the code in
123 this tutorial and there is more information and links to additional resources for how to use
124 R Markdown for reproducible reports in the additional resources.

125   The reason that the above install packages code is not included in the workbook
126 is that every time you run the install command code it will install the latest version of
127 the package. Leaving this code in your script can lead you to unintentionally install a
128 package update you didn't want. For this reason, avoid including install code in any script
129 or Markdown document.

Preparing your data

131   Before you start visualising your data, you need to get it into an appropriate format.
132 These preparatory steps can all be dealt with reproducibly using R and the additional re-
133 sources section points to extra tutorials for doing so. However, performing these types of
134 tasks in R can require more sophisticated coding skills and the solutions and tools are de-
135 pendent on the idiosyncrasies of each dataset. For this reason, in this tutorial we encourage
136 the reader to complete data preparation steps using the method they are most comfortable
137 with and to focus on the aim of data visualisation.

138   **Data format.** The simulated lexical decision data is provided in a `csv` file rather
139 than e.g., `xslx`. Functions exist in R to read many other types of data files, however, we
140 recommend that you convert any `xlsx` spreadsheets to `csv` by using the `Save As` function
141 in Microsoft Excel. The `csv` file format strips all formatting and only stores data in a single
142 sheet and so is simpler for new users to import to R. You may wish to create a `csv` file that
143 contains only the data you want to visualise, rather than a full, larger workbook. When
144 working with your own data, remove summary rows or additional notes from any files you
145 import. All files should only contains the rows and columns of data you want to plot.

146   **Variable names.** Ensuring that your variable names are consistent can make it
147 much easier to work in R. We recommend using short but informative variable names, for
148 example `rt_word` is preferred over `dv1_iv1` or `reaction_time_word_condition` because
149 these are either hard to read or hard to type.

150   It is also helpful to have a consistent naming scheme, particularly for variable names
151 that require more than one word. Two popular options are `CamelCase` where each new word
152 begins with a capital letter, or `snake_case` where all letters are lower case and words are
153 separated by an underscore. For the purposes of naming variables, avoid using any spaces in
154 variable names (e.g., `rt word`) and consider the additional meaning of a separator beyond
155 making the variable names easier to read. For example, `rt_word`, `rt_nonword`, `acc_word`,
156 and `acc_nonword` all have the DV to the left of the separator and the level of the IV to the
157 right. `rt_word_condition` on the other hand has two separators but only one of them is
158 meaningful and it is useful to be able to split variable names consistently. In this paper, we
159 will use `snake_case` and lower case letters for all variable names so that we don't have to
160 remember where to put the capital letters.

161   When working with your own data, you can rename columns in Excel, but the re-
162 sources listed in the online appendices point to how to rename columns reproducibly with
163 code.

164   **Data values.** A great benefit to using R is that categorical data can be entered
165 as text. In the tutorial dataset, language group is entered as 1 or 2, so that we can

show you how to recode numeric values into factors with labels. However, we recommend
recording meaningful labels rather than numbers from the beginning of data collection to
avoid misinterpreting data due to coding errors. Note that values must match *exactly* in
order to be considered in the same category and R is case sensitive, so "mono," "Mono,"
and "monolingual" would be classified as members of three separate categories.

Finally, cells that represent missing data should be left empty rather than containing
values like `NA`, `missing` or `999`[3]. A complementary rule of thumb is that each column should
only contain one type of data, such as words or numbers, not both.

Getting Started

Loading packages

To load the packages that have the functions we need, use the `library()` function.
Whilst you only need to install packages once, you need to load any packages you want
to use with `library()` every time you start R or start a new session. When you load
the `tidyverse`, you actually load several separate packages that are all part of the same
collection and have been designed to work well together. R will produce a message that
tells you the names of all the packages that have been loaded.

```
library(tidyverse)
library(patchwork)
```

Loading data

To load the simulated data we use the function `read_csv()` from the `readr` tidyverse
package. Note that there are many other ways of reading data into R, but the benefit of
this function is that it enters the data into the R environment in such a way that it makes
most sense for other tidyverse packages.

```
dat <- read_csv(file = "ldt_data.csv")
```

This code has created an object `dat` into which you have read the data from the file
`ldt_data.csv`. This object will appear in the environment pane in the top right. Note
that the name of the data file must be in quotation marks and the file extension (`.csv`)
must also be included. If you receive the error `...does not exist in current working`
`directory` it is highly likely that you have made a typo in the file name (remember R is case
sensitive), have forgotten to include the file extension `.csv`, or that the data file you want
to load is not stored in your project folder. If you get the error `could not find function`
it means you have either not loaded the correct package (a common beginner error is to
write the code, but not run it), or you have made a typo in the function name.

---

[3]If your data use a missing value like `NA` or `999`, you can indicate this in the `na` argument of `read_csv()`
when you read in your data. For example, `read_csv("data.csv", na = c("", "NA", 999))` allows you to
use blank cells `""`, the letters `"NA"`, and the number `999` as missing values.

196   To view the dataset, click `dat` in the environment pane or run `View(dat)` in the
197 console. The environment pane also tells us that the object `dat` has 100 observations of 7
198 variables, and this is a useful quick check to ensure one has loaded the right data. Note
199 that the 7 variables have an additional piece of information `chr` and `num`; this specifies the
200 kind of data in the column. Similar to Excel and SPSS, R uses this information (or variable
201 type) to specify allowable manipulations of data. For instance character data such as the
202 `id` cannot be averaged, while it is possible to do this with numerical data such as the `age`.

Handling numeric factors

204   Another useful check is to use the functions `summary()` and `str()` (structure) to check
205 what kind of data R thinks is in each column. Run the below code and look at the output
206 of each, comparing it with what you know about the simulated dataset:

```
summary(dat)
str(dat)
```

207   Because the factor `language` is coded as 1 and 2, R has categorised this column
208 as containing numeric information and unless we correct it, this will cause problems for
209 visualisation and analysis. The code below shows how to recode numeric codes into labels.

210 • `mutate()` makes new columns in a data table, or overwrites a column;
211 • `factor()` translates the language column into a factor with the labels "monolingual"
212   and "bilingual." You can also use `factor()` to set the display order of a column that
213   contains words. Otherwise, they will display in alphabetical order. In this case we
214   are replacing the numeric data (1 and 2) in the `language` column with the equivalent
215   English labels `monolingual` for 1 and `bilingual` for 2. At the same time we will
216   change the column type to be a factor, which is how R defines categorical data.

```
dat <- dat %>%
  mutate(language = factor(
    x = language, # column to translate
    levels = c(1, 2), # values of the original data in preferred order
    labels = c("monolingual", "bilingual") # labels for display
  ))
```

217   Make sure that you always check the output of any code that you run. If after running
218 this code `language` is full of `NA` values, it means that you have run the code twice. The
219 first time would have worked and transformed the values from 1 to `monolingual` and 2 to
220 `bilingual`. If you run the code again on the same dataset, it will look for the values 1 and
221 2, and because there are no longer any that match, it will return NA. If this happens, you
222 will need to reload the dataset from the csv file.

223   A good way to avoid this is never to overwrite data, but to always store the output
224 of code in new objects (e.g., `dat_recoded`) or new variables (`language_recoded`). For the

225 purposes of this tutorial, overwriting provides a useful teachable moment so we'll leave it
226 as it is.

Argument names

228   Each function has a list of arguments it can take, and a default order for those
229 arguments. You can get more information on each function by entering `?function_name`
230 into the console, although be aware that learning to read the help documentation in R is a
231 skill in itself. When you are writing R code, as long as you stick to the default order, you
232 do not have to explicitly call the argument names, for example, the above code could also
233 be written as:

```r
dat <- dat %>%
  mutate(language = factor(language,
                           c(1, 2),
                           c("monolingual", "bilingual")))
```

234   One of the challenges in learning R is that many of the "helpful" examples and
235 solutions you will find online do not include argument names and so for novice learners are
236 completely opaque. In this tutorial, we will include the argument names the first time a
237 function is used, however, we will remove some argument names from subsequent examples
238 to facilitate knowledge transfer to the help available online.

Demographic information

240   You can calculate and plot some basic descriptive information about the demographics
241 of our sample using the imported dataset without any additional wrangling (i.e., data
242 processing). The code below uses the `%>%` operator, otherwise known as the *pipe,* and can
243 be translated as "*and then"*. For example, the below code can be read as:

244 • Start with the dataset `dat` *and then;*

245 • Group it by the variable `language` *and then;*

246 • Count the number of observations in each group

```r
dat %>%
  group_by(language) %>%
  count()
```

language	n
monolingual	55
bilingual	45

248 `group_by()` does not result in surface level changes to the dataset, rather, it changes
249 the underlying structure so that if groups are specified, whatever function is called next
250 is performed separately on each level of the grouping variable. The above code therefore
251 counts the number of observations in each group of the variable `language`. If you just
252 need the total number of observations, you could remove the `group_by()` line which would
253 perform the operation on the whole dataset, rather than by groups:

```
dat %>%
  count()
```

254

n
100

255 Similarly, we may wish to calculate the mean age (and SD) of the sample and we can
256 do so using the function `summarise()` from the `dplyr` tidyverse package.

```
dat %>%
  summarise(mean_age = mean(age),
            sd_age = sd(age),
            n_values = n())
```

257

mean_age	sd_age	n_values
29.75	8.28	100

258 This code produces summary data in the form of a column named `mean_age` that
259 contains the result of calculating the mean of the variable `age`. It then creates `sd_age`
260 which does the same but for standard deviation. Finally, it uses the function `n()` to add
261 the number of values used to calculate the statistic in a column named `n_values` - this is
262 a useful sanity check whenever you make summary statistics.

263 Note that the above code will not save the result of this operation, it will simply
264 output the result in the console. If you wish to save it for future use, you can store it in an
265 object by using the `<-` notation and print it later by typing the object name.

```
age_stats <- dat %>%
  summarise(mean_age = mean(age),
            sd_age = sd(age),
            n_values = n())
```

266 Finally, the `group_by()` function will work in the same way when calculating sum-
267 mary statistics - the output of the function that is called after `group_by()` will be produced
268 for each level of the grouping variable.

```
dat %>%
  group_by(language) %>%
  summarise(mean_age = mean(age),
            sd_age = sd(age),
            n_values = n())
```

language	mean_age	sd_age	n_values
monolingual	27.96	6.78	55
bilingual	31.93	9.44	45

269

Bar chart of counts

270

271   For our first plot, we will make a simple bar chart of counts that shows the number
272 of participants in each `language` group.

```
ggplot(data = dat, mapping = aes(x = language)) +
  geom_bar()
```
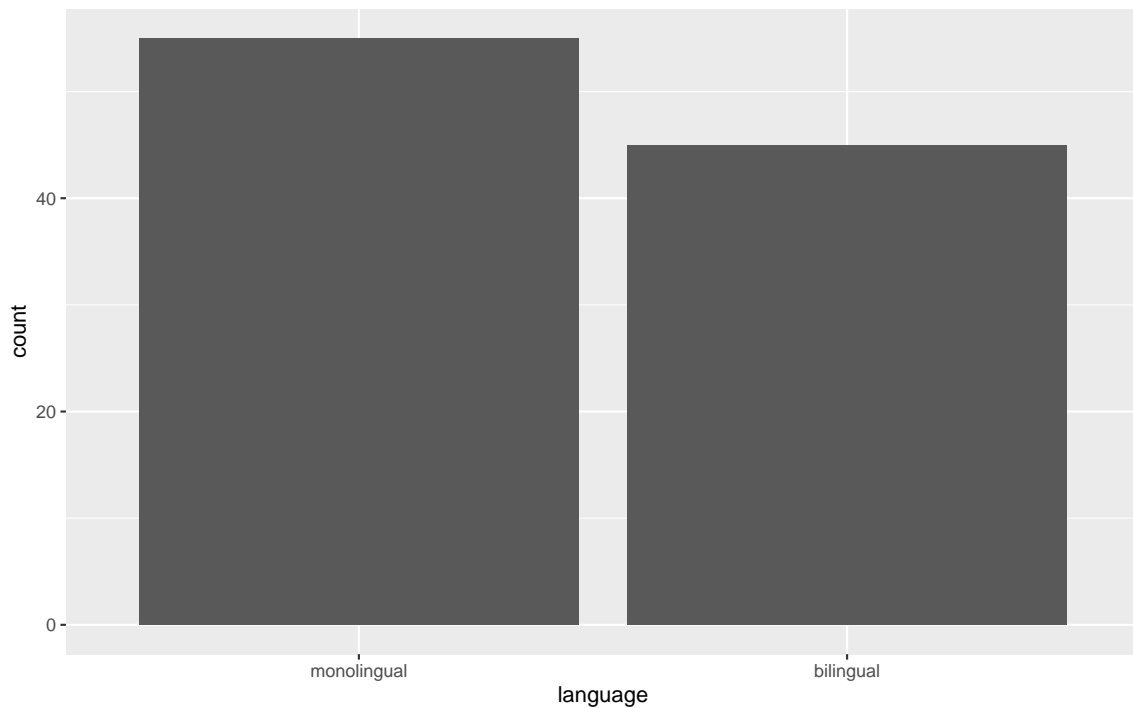

Figure 3. Bar chart of counts.

273   The first line of code sets up the base of the plot.

274 • `data` specifies which data source to use for the plot

275 • `mapping` specifies which variables to map to which aesthetics (`aes`) of the plot. Aes-
276   thetic mappings describe how variables in the data are mapped to visual properties
277   (aesthetics) of geoms.

278 • `x` specifies which variable to put on the x-axis

²⁷⁹   The second line of code adds a `geom`, and is connected to the base code with `+`.
²⁸⁰ In this case, we ask for `geom_bar()`. Each `geom` has an associated default statistic. For
²⁸¹ `geom_bar()`, the default statistic is to count the data passed to it. This means that you
²⁸² do not have to specify a `y` variable when making a bar plot of counts; when given an `x`
²⁸³ variable `geom_bar()` will automatically calculate counts of the groups in that variable. In
²⁸⁴ this example, it counts the number of data points that are in each category of the `language`
²⁸⁵ variable.

²⁸⁶   The base layer and the geoms you add as layers work in symbiosis so it is worthwhile
²⁸⁷ checking the mapping rules as these are related to the default statistic for the plot's geom.

²⁸⁸ **Plotting existing aggregates and percent**

²⁸⁹   If your dataset already has the counts that you want to plot, you can set
²⁹⁰ `stat="identity"` inside of `geom_bar()` to use that number instead of counting rows.
²⁹¹ For example, there is currently no function to plot percentages rather than counts within
²⁹² `ggplot2`, you need to calculate these and store them in a new object that is then used as
²⁹³ the dataset.

²⁹⁴   Notice that we are now omitting the names of the arguments `data` and `mapping` in
²⁹⁵ the `ggplot()` function.

```r
dat_percent <- dat %>%     # start with the data in dat
  count(language) %>%       # count rows per language (makes a new column called n)
  mutate(percent = (n/sum(n)*100)) # make a new column 'percent' equal to
                                   # n divided by the sum of n times 100

ggplot(dat_percent, aes(x = language, y = percent)) +
  geom_bar(stat="identity")
```
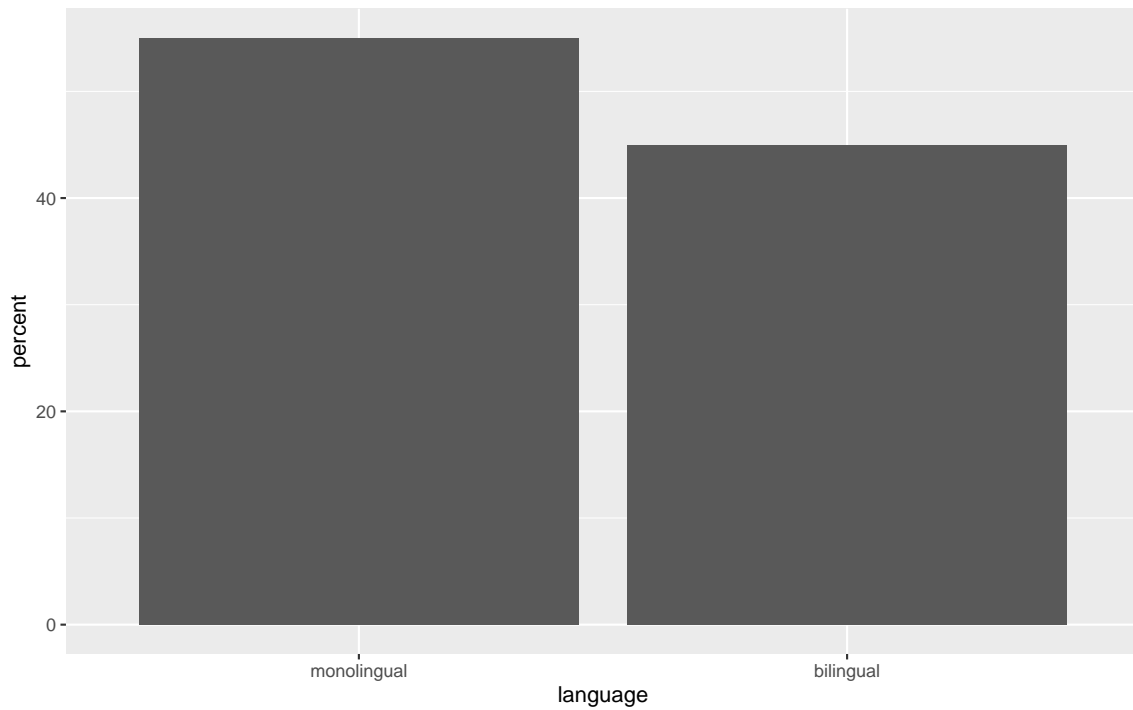
Figure 4. Bar chart of pre-calculated counts.

Histogram

The code to plot a histogram of `age` is very similar to the code used for the bar chart. We start by setting up the plot space, the dataset we want to use, and mapping the variables to the relevant axis. In this case, we want to plot a histogram with `age` on the x-axis:
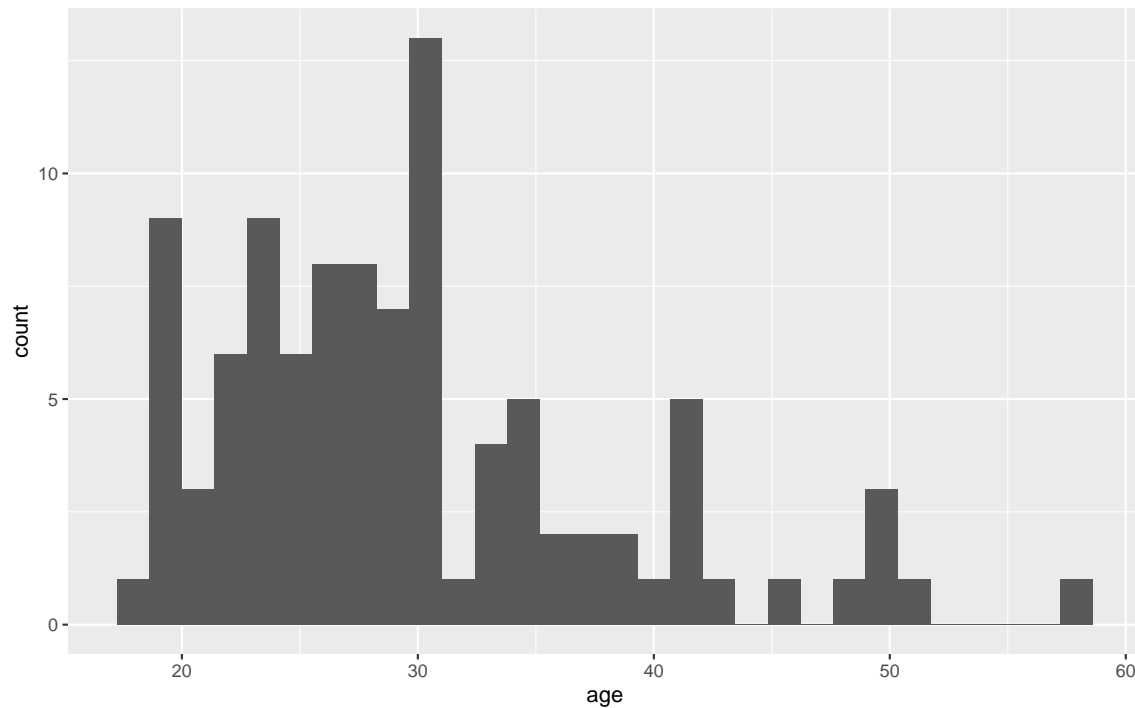
```
ggplot(dat, aes(x = age)) +
  geom_histogram()
```

Figure 5. Histogram of ages.

³⁰¹ The base statistic for `geom_histogram()` is also count, and by default
³⁰² `geom_histogram()` divides the x-axis into "bins" and counts how many observations are in
³⁰³ each bin and so the y-axis does not need to be specified. When you run the code to produce
³⁰⁴ the histogram, you will get the message `stat_bin() using bins = 30. Pick better`
³⁰⁵ `value with binwidth`. This means that the default number of bins `geom_histogram()`
³⁰⁶ divided the x-axis into is 30. For our data that looks appropriate, but for example, if you
³⁰⁷ want one bar to equal 5 years, you can adjust `binwidth = 5`.

```
ggplot(dat, aes(x = age)) +
  geom_histogram(binwidth = 5)
```
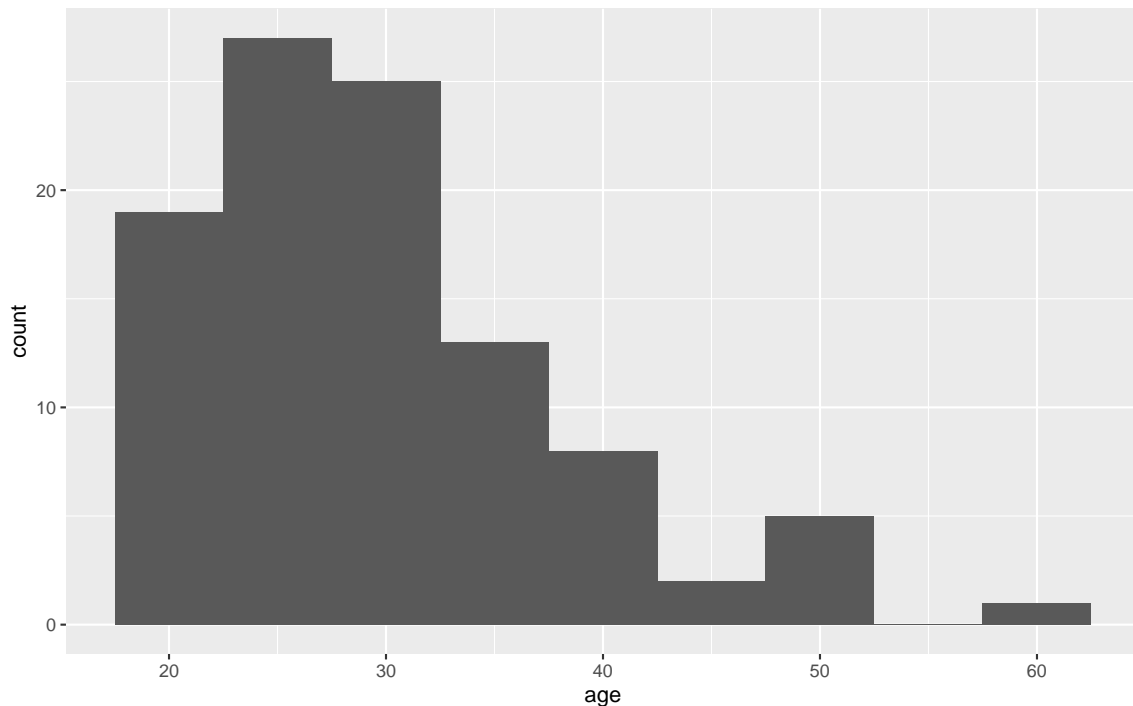
Figure 6. Histogram of ages where each bin covers five years.

308 **Customisation 1**

309   So far we have made basic plots with the default visual appearance. Before we move on
310 to the experimental data we will introduce some simple visual customisation options. There
311 are many ways in which you can control or customise the visual appearance of figures in R.
312 However, once you understand the logic of one, it becomes easier to understand others that
313 you may see in other examples. Visual appearance of elements can be customised within a
314 geom itself, within the aesthetic mapping, or by connecting additional layers with `+`. In this
315 section we look at the simplest and most commonly-used customisations: changing colours,
316 adding axis labels, and adding themes.

317   **Changing colours.**   For our basic bar chart, you can control colours used to display
318 the bars by setting `fill` (internal colour) and `colour` (outline colour) inside the geom
319 function. This methods changes **all** the bars; we will show you later how to set fill or colour
320 separately for different groups.

```
ggplot(dat, aes(age)) +
  geom_histogram(binwidth = 1,
                 fill = "white",
                 colour = "black")
```
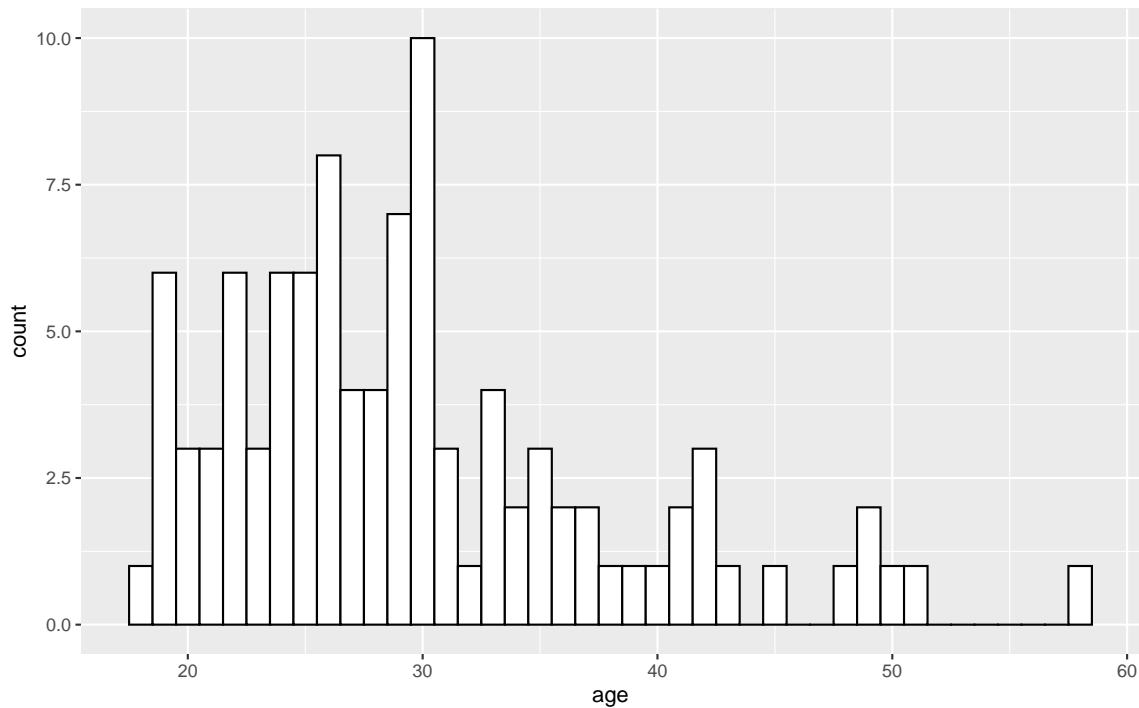
Figure 7. Histogram with custom colors for bar fill and line colors.

³²¹   **Editing axis names and labels.**   To edit axis names and labels you can connect
³²² `scale_*` functions to your plot with `+` to add layers. These functions are part of `ggplot2`
³²³ and the one you use depends on which aesthetic you wish to edit (e.g., x-axis, y-axis, fill,
³²⁴ colour) as well as the type of data it represents (discrete, continuous).

³²⁵   For the bar chart of counts, the x-axis is mapped to a discrete (categorical) variable
³²⁶ whilst the y-axis is continuous. For each of these there is a relevant scale function with
³²⁷ various elements that can be customised. Each axis then has its own function added as a
³²⁸ layer to the basic plot.

```
ggplot(dat, aes(language)) +
  geom_bar() +
  scale_x_discrete(name = "Language group",
                   labels = c("Monolingual", "Bilingual")) +
  scale_y_continuous(name = "Number of participants",
                     breaks = c(0,10,20,30,40,50))
```
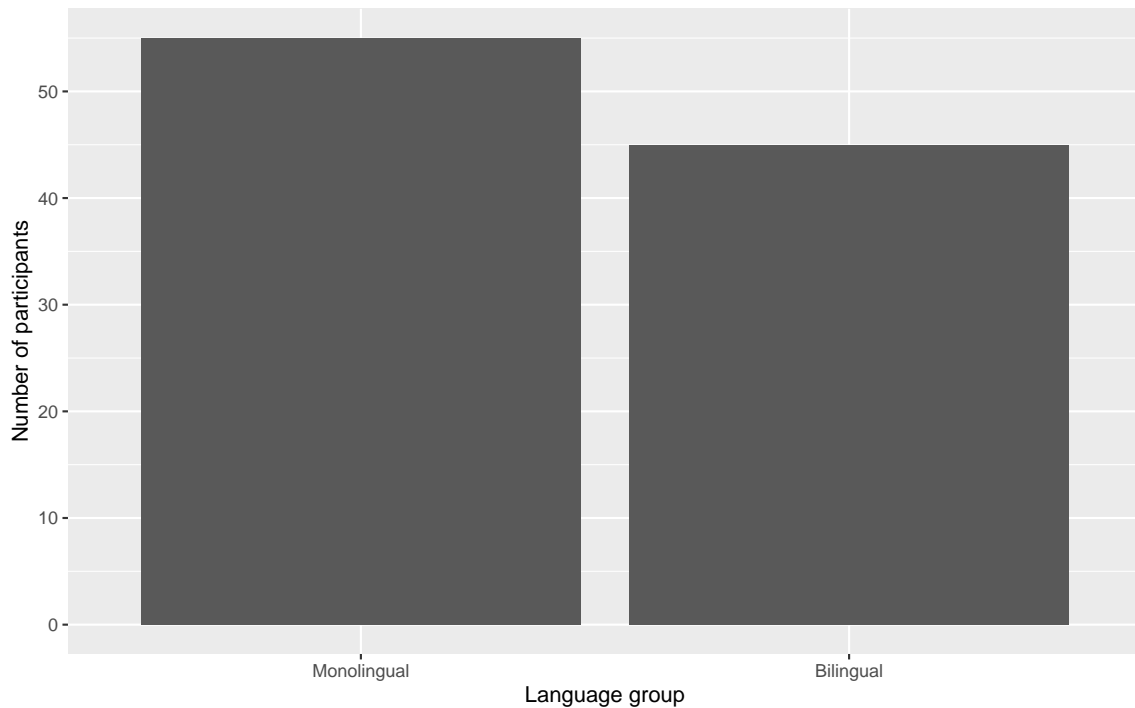
Figure 8. Bar chart with custom axis labels.

329 • `name` controls the overall name of the axis (note the use of quotation marks)

330 • `labels` controls the names of the conditions with a discrete variable.

331 • `c()` is a function that you will see in many different contexts and is used to combine
332   multiple values. In this case, the labels we want to apply are combined within `c()` by
333   enclosing each word within their own parenthesis, and are in the order displayed on
334   the plot. A very common error is to forget to enclose multiple values in `c()`.

335 • `breaks` controls the tick marks on the axis. Again because there are multiple values,
336   they are enclosed within `c()` although because they are numeric and not text, they
337   do not need quotation marks.

338   **Discrete vs. continuous errors.** Another very common error is to map the wrong
339 type of `scale_` function to a variable. Try running the below code:

```
# produces an error
ggplot(dat, aes(language)) +
  geom_bar() +
  scale_x_continuous(name = "Language group",
                     labels = c("Monolingual", "Bilingual"))
```

340   This will produce the error `Discrete value supplied to continuous scale` be-
341 cause we have used a `continuous` scale function, despite the fact that x-axis variable is

342 discrete. If you get this error (or the reverse), check the type of data on each axis and the
343 function you have used.

344   **Adding a theme.** ggplot2 has a number of built-in visual themes that you can
345 apply as an extra layer. The below code updates the x-axis and y-axis labels to the his-
346 togram, but also applies `theme_minimal()`. Each part of a theme can be independently
347 customised, which may be necessary, for example, if you have journal guidelines on fonts
348 for publication. There are further instructions for how to do this in the online appendices.

```r
ggplot(dat, aes(age)) +
  geom_histogram(binwidth = 1, fill = "wheat", color = "black") +
  scale_x_continuous(name = "Participant age (years)") +
  theme_minimal()
```
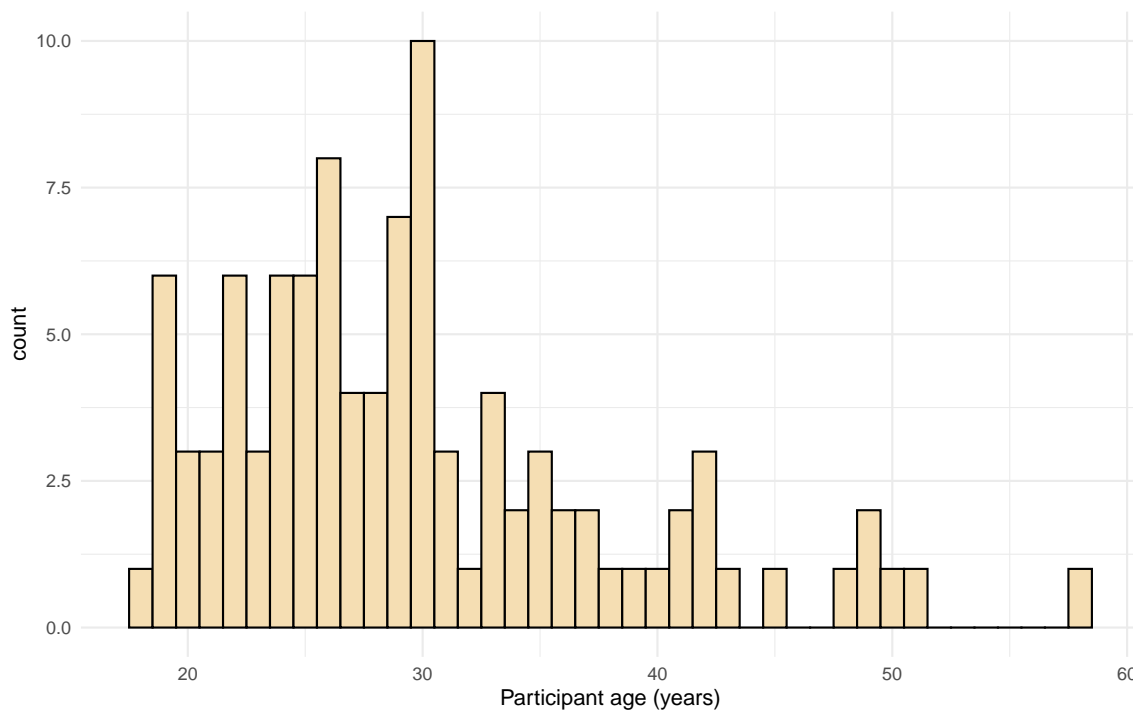

Figure 9. Histogram with a custom theme.

349   You can set the theme globally so that all subsequent plots use a theme.

```r
theme_set(theme_minimal())
```

350   If you wished to return to the default theme, change the above to specify
351 `theme_grey()`.

Activities 1

Before you move on try the following:

1. Add a layer that edits the **name** of the y-axis histogram label to `Number of participants`.

2. Change the colour of the bars in the bar chart to red.

3. Remove `theme_minimal()` from the histogram and instead apply one of the other available themes. To find out about other available themes, start typing `theme_` and the auto-complete will show you the available options - this will only work if you have loaded the `tidyverse` library with `library(tidyverse)`.

Transforming Data

Data formats

To visualise the experimental reaction time and accuracy data using `ggplot2`, we first need to reshape the data from wide-format to long-format and it is this step that can cause friction with novice users of R. Traditionally, psychologists have been taught data skills using wide-format data. Wide-format data typically has one row of data for each participant with separate columns for each score or variable. Where there are repeated-measures variables, the dependent variable is split across different columns with one measurement for each condition and where there is a between groups variable, a separate column is added to encode the group to which a participant or observation belongs.

The simulated lexical decision data is currently in wide-format (see Table 1) where each participant's aggregated[4] reaction time and accuracy for each level of the within-subject variable is split across multiple columns.

Wide-format is popular because it is intuitive to read and easy to enter data into as all the data for one participant is contained within a single row. However, for the purposes of analysis, and particularly for analysis using R, this format is unsuitable. Whilst it is intuitive to read by a human, the same is not true for a computer. Wide-format data concatenates multiple pieces of information in a single column, for example in Table 1, `rt_word` contains information related to both a DV and one level of an IV. In comparison, long-format data separates the DV from the IV's so that each column represents only one variable. The less intuitive part is that long data has multiple rows for each participant and a column that encodes the level of the IV (`word` or `nonword`). In essence, the long-format encodes repeated-measures variable in the same way as a between-group variable

---

[4]In this tutorial we have chosen to gloss over the data processing steps that must occur to get from the raw data to aggregated values. This type of processing requires a more extensive tutorial than we can provide in the current paper. More importantly, it is still possible to use R for data visualisation having done the preparatory steps using existing workflows in Excel and SPSS, so long as the file is saved/exported as a `.csv` file. We bypass these initial steps and focus on tangible outputs that may then encourage further mastery of reproducible methods. Collectively we tend to call the steps for reshaping data and for processing raw data or for getting data ready to use statistical functions "wrangling."

Table 1
Data in wide format.

id	age	language	rt_word	rt_nonword	acc_word	acc_nonword
S001	22	monolingual	379.46	516.82	99	90
S002	33	monolingual	312.45	435.04	94	82
S003	23	monolingual	404.94	458.50	96	87
S004	28	monolingual	298.37	335.89	92	76
S005	26	monolingual	316.42	401.32	91	83
S006	29	monolingual	357.17	367.34	96	78

Table 2
Data in the correct format for visualization.

id	age	language	condition	rt	acc
S001	22	monolingual	word	379.46	99
S001	22	monolingual	nonword	516.82	90
S002	33	monolingual	word	312.45	94
S002	33	monolingual	nonword	435.04	82
S003	23	monolingual	word	404.94	96
S003	23	monolingual	nonword	458.50	87

in SPSS. Wickham (2014) provides a comprehensive overview of the benefits of a similar format known as tidy data, which is a standard way of mapping a dataset to its structure, but for the purposes of this tutorial there are two important rules: each column should be a *variable* and each row should be an *observation.*

Moving from using wide-form to long-form datasets can require a conceptual shift on the part of the researcher and one that usually only comes with practice and repeated exposure[5]. For our example dataset, adhering to these rules for reshaping the data would produce Table 2. Rather than different observations of the same dependent variable being split across columns, there is now a single column for the DV reaction time, and a single column for the DV accuracy. Each participant now has multiple rows of data, one for each observation (i.e., for each participant there will be as many rows as there are levels of the within-subject IV). Although there is some repetition of age and language group, each row is unique when looking at the combination of measures.

The benefits and flexibility of this format will hopefully become apparent as we progress through the tutorial, however, a useful rule of thumb when working with data in R for visualisation is that *anything that shares an axis should probably be in the same column.* For example, a simple boxplot showing reaction time by condition would display the variable `condition` on the x-axis with bars representing both the `word` and `nonword` data, and `rt` on the y-axis. Therefore, all the data relating to `condition` should be in one column, and all the data relating to `rt` should be in a separate single column, rather than being split like in wide-format data.

---

[5]That is to say, if you are new to R, know that many before you have struggled with this conceptual shift - it does get better, it just takes time and your preferred choice of cursing.

Transforming data

⁴⁰⁶ We have chosen a 2 x 2 design with two DVs as we anticipate that this is a design
⁴⁰⁷ many researchers will be familiar with and may also have existing datasets with a similar
⁴⁰⁸ structure. However, it is worth normalising that trial-and-error is part of the process of
⁴⁰⁹ learning how to apply these functions to new datasets and structures. Data visualisation
⁴¹⁰ can be a useful way to scaffold learning these data transformations because they can provide
⁴¹¹ a concrete visual check as to whether you have done what you intended to do with your
⁴¹² data.

⁴¹³ **Step 1: `pivot_longer()`.** The first step is to use the function `pivot_longer()`
⁴¹⁴ to transform the data to long-form. We have purposefully used a more complex dataset
⁴¹⁵ with two DVs for this tutorial to aid researchers applying our code to their own datasets.
⁴¹⁶ Because of this, we will break down the steps involved to help show how the code works.

⁴¹⁷ This first code ignores that the dataset has two DVs, a problem we will fix in step 2.
⁴¹⁸ The pivot functions can be easier to show than tell - you may find it a useful exercise to
⁴¹⁹ run the below code and compare the newly created object `long` (Table 3) with the original
⁴²⁰ `dat` Table 1 before reading on.

```
long <- pivot_longer(data = dat,
                     cols = rt_word:acc_nonword,
                     names_to = "dv_condition",
                     values_to = "dv")
```

⁴²¹ • As with the other tidyverse functions, the first argument specifies the dataset to use
⁴²² as the base, in this case `dat`. This argument name is often dropped in examples.

⁴²³ • `cols` specifies all the columns you want to transform. The easiest way to visualise this
⁴²⁴ is to think about which columns would be the same in the new long-form dataset and
⁴²⁵ which will change. If you refer back to Table 1, you can see that `id`, `age`, and `language`
⁴²⁶ all remain, while the columns that contain the measurements of the DVs change. The
⁴²⁷ colon notation `first_column:last_column` is used to select all variables from the
⁴²⁸ first column specified to the second. In our code, `cols` specifies that the columns we
⁴²⁹ want to transform are `rt_word` to `acc_nonword`.

⁴³⁰ • `names_to` specifies the name of the new column that will be created.

⁴³¹ • Finally, `values_to` names the new column that will contain the measurements, in this
⁴³² case we'll call it `dv`. At this point you may find it helpful to go back and compare `dat`
⁴³³ and `long` again to see how each argument matches up with the output of the table.

⁴³⁴ **Step 2: `pivot_longer()` adjusted.** The problem with the above long-form data-
⁴³⁵ set is that because we have ignored that there are two DVs, `dv_condition` still continues
⁴³⁶ to conflate two variables - it has information about the type of DV and the condition of
⁴³⁷ the IV. To account for this, we include a new argument `names_sep` and adjust `name_to` to
⁴³⁸ specify the creation of two new columns. Note that we are pivoting the same wide-format
⁴³⁹ dataset `dat` as we did in step 1.

Table 3
Data in long format with mixed DVs.

id	age	language	dv_condition	dv
S001	22	monolingual	rt_word	379.46
S001	22	monolingual	rt_nonword	516.82
S001	22	monolingual	acc_word	99.00
S001	22	monolingual	acc_nonword	90.00
S002	33	monolingual	rt_word	312.45
S002	33	monolingual	rt_nonword	435.04

Table 4
Data in long format with dv type and condition in separate columns.

id	age	language	dv_type	condition	dv
S001	22	monolingual	rt	word	379.46
S001	22	monolingual	rt	nonword	516.82
S001	22	monolingual	acc	word	99.00
S001	22	monolingual	acc	nonword	90.00
S002	33	monolingual	rt	word	312.45
S002	33	monolingual	rt	nonword	435.04

- `names_sep` specifies how to split up the variable name in cases where it has multiple components. This is when taking care to name your variables consistently and meaningfully pays off. Because the word to the left of the separator (`_`) is always the DV type and the word to the right is always the condition of the within-subject IV, it is easy to automatically split the columns.

- Note that when specifying more than one column name, they must be combined using `c()` and be enclosed in their own quotation marks.

```
long2 <- pivot_longer(data = dat,
                      cols = rt_word:acc_nonword,
                      names_sep = "_",
                      names_to = c("dv_type", "condition"),
                      values_to = "dv")
```

Step 3: `pivot_wider()`. Although we have now split the columns so that there are separate variables for the DV type and level of condition, because we have two DVs, there is an additional bit of wrangling required to get the data in the right format for plotting.

In the current long-form dataset, the column `dv` contains both reaction time and accuracy measures and keeping in mind the rule of thumb that *anything that shares an axis should probably be in the same column,* this creates a problem because we cannot plot two different units of measurement on the same axis. To fix this we need to use the function `pivot_wider()`. Again, we would encourage you at this point to compare `long2` and `dat_long` with the below code to try and map the connections before reading on.

```
dat_long <- pivot_wider(long2,
                        names_from = "dv_type",
                        values_from = "dv")
```

456   • The first argument is again the dataset you wish to work from, in this case `long2`.
457   We have removed the argument name `data` in this example.

458   • `names_from` acts somewhat like the reverse of `names_to` from `pivot_longer()`. It
459   will take the values from the variable specified and use these as variable names, i.e.,
460   in this case, the values of `rt` and `acc` that are currently in the `dv_type` column, and
461   turn these into the column names.

462   • Finally, `values_from` specifies the values to fill the new columns with. In this case,
463   the new columns `rt` and `acc` will be filled with the values that were in `dv`. Again, it
464   can be helpful to compare each dataset with the code to see how it aligns.

465   This final long-form data should look like Table 2.

466   If you are working with a dataset with only one DV, note that only step 1 of this pro-
467   cess would be necessary. Also, be careful not to calculate demographic descriptive statistics
468   from this long-form dataset. Because the process of transformation has introduced some
469   repetition for these variables, the wide-form dataset where 1 row = 1 participant should be
470   used for demographic information. Finally, the three step process noted above is broken
471   down for teaching purposes, in reality, one would likely do this in a single pipeline of code,
472   for example:

```
dat_long <- pivot_longer(data = dat,
                         cols = rt_word:acc_nonword,
                         names_sep = "_",
                         names_to = c("dv_type", "condition"),
                         values_to = "dv") %>%
  pivot_wider(names_from = "dv_type",
              values_from = "dv")
```

473   **Histogram 2**

474   Now that we have the experimental data in the right form, we can begin to create some
475   useful visualizations. First, to demonstrate how code recipes can be reused and adapted,
476   we will create histograms of reaction time and accuracy. The below code uses the same
477   template as before but changes the dataset (`dat_long`), the bin-widths of the histograms,
478   the `x` variable to display (`rt`/`acc`), and the name of the x-axis.

```
ggplot(dat_long, aes(x = rt)) +
  geom_histogram(binwidth = 10, fill = "white", colour = "black") +
```

```
  scale_x_continuous(name = "Reaction time (ms)")

ggplot(dat_long, aes(x = acc)) +
  geom_histogram(binwidth = 1, fill = "white", colour = "black") +
  scale_x_continuous(name = "Accuracy (0-100)")
```
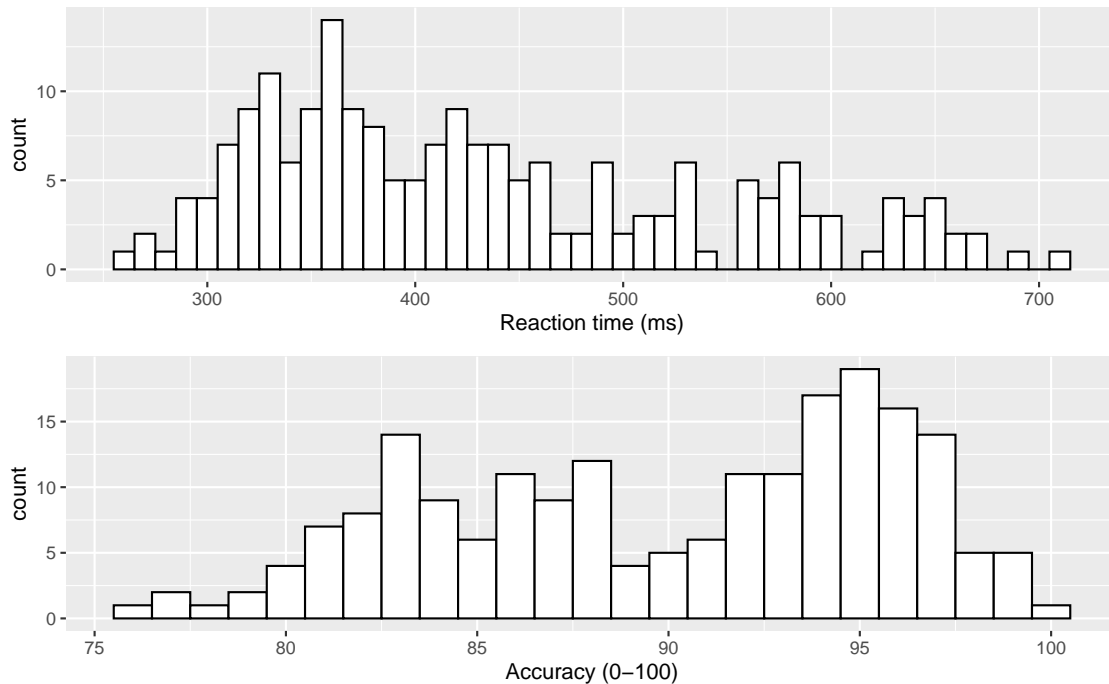

Figure 10. Histograms showing the distribution of reaction time (top) and accuracy (bottom)

479 **Density plots**

480   The layer system makes it easy to create new types of plots by adapting existing
481 recipes. For example, rather than creating a histogram, we can create a smoothed density
482 plot by calling `geom_density()` rather than `geom_histogram()`. The rest of the code
483 remains identical.

```
ggplot(dat_long, aes(x = rt)) +
  geom_density()+
  scale_x_continuous(name = "Reaction time (ms)")
```
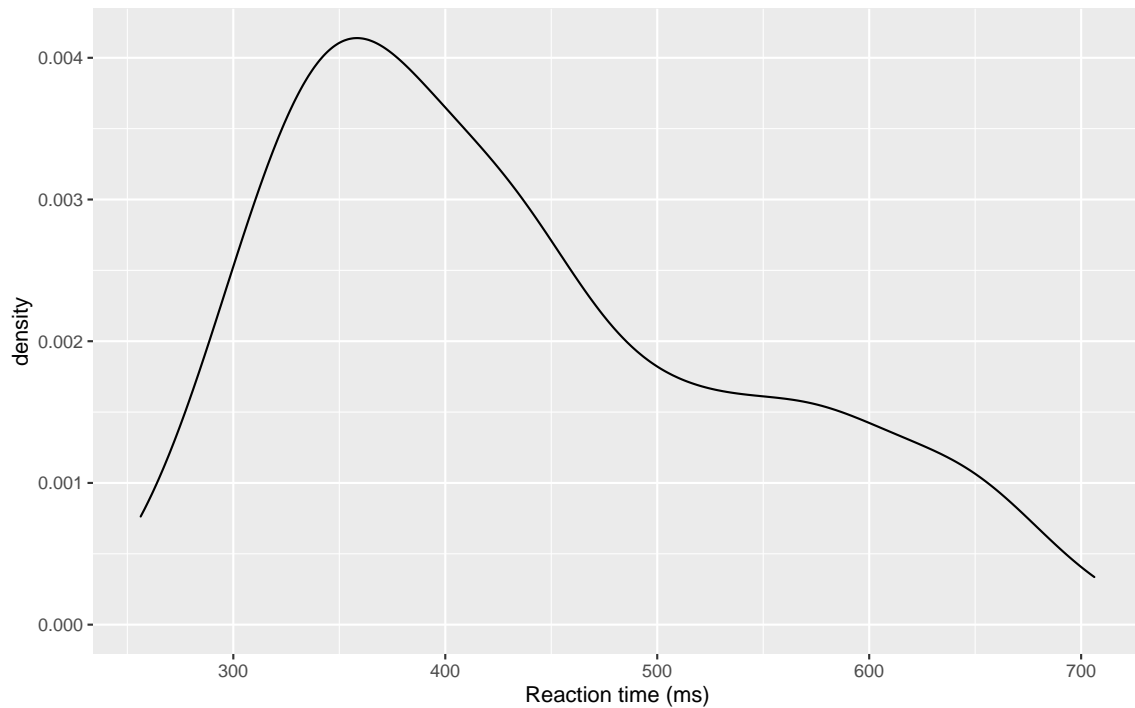
Figure 11. Density plot of reaction time.

484   **Grouped density plots.**   Density plots are most useful for comparing the distri-
485 butions of different groups of data. Because the dataset is now in long format, it makes it
486 easier to map another variable to the plot because each variable is contained within a single
487 column.

488 • In addition to mapping `rt` to the x-axis, we specify the `fill` aesthetic to fill the
489   visualisation of each level of the `condition` variable with different colours.
490 • As with the x and y-axis scale functions, we can edit the names and labels of our fill
491   aesthetic by adding on another `scale_*` layer.
492 • Note that the `fill` here is set inside the `aes()` function, which tells ggplot to set
493   the fill differently for each value in the `condition` column. You cannot specify which
494   colour here (e.g., `fill="red"`), like you could when you set `fill` inside the `geom_*()`
495   function before.

```
ggplot(dat_long, aes(x = rt, fill = condition)) +
  geom_density()+
  scale_x_continuous(name = "Reaction time (ms)") +
  scale_fill_discrete(name = "Condition",
                      labels = c("Word", "Non-word"))
```
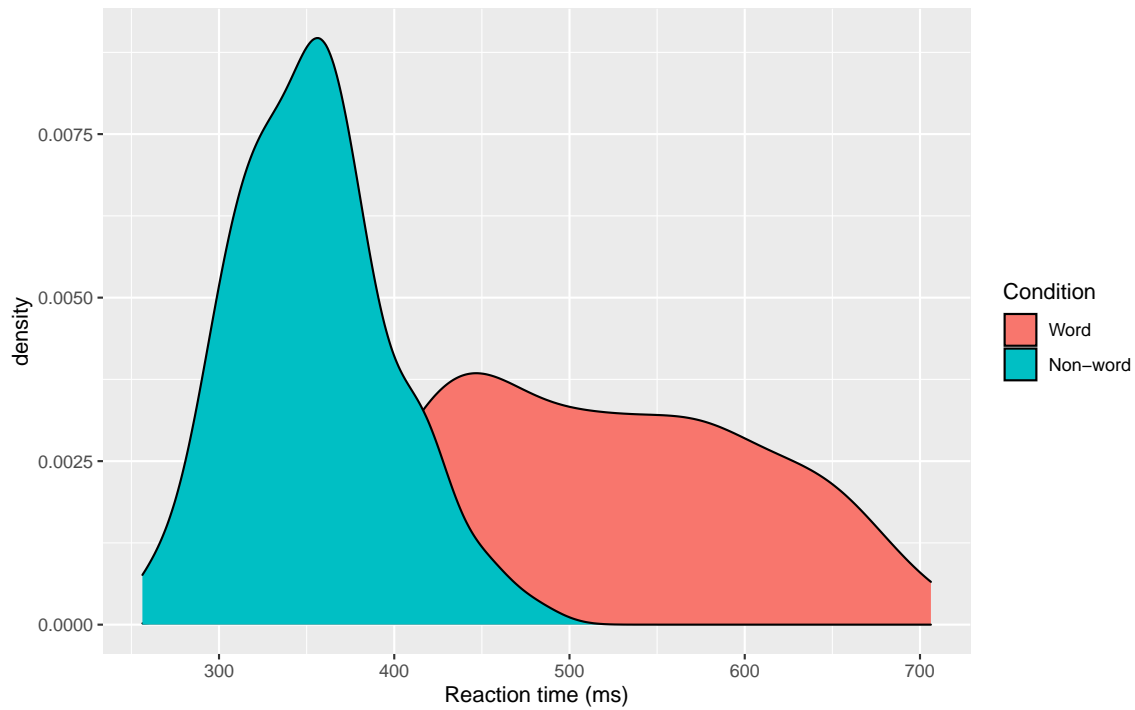
Figure 12. Density plot of reaction times grouped by condition.

Scatterplots

Scatterplots are created by calling `geom_point()` and require both an x and y variable to be specified in the mapping.

```
ggplot(dat_long, aes(x = rt, y = age)) +
  geom_point()
```
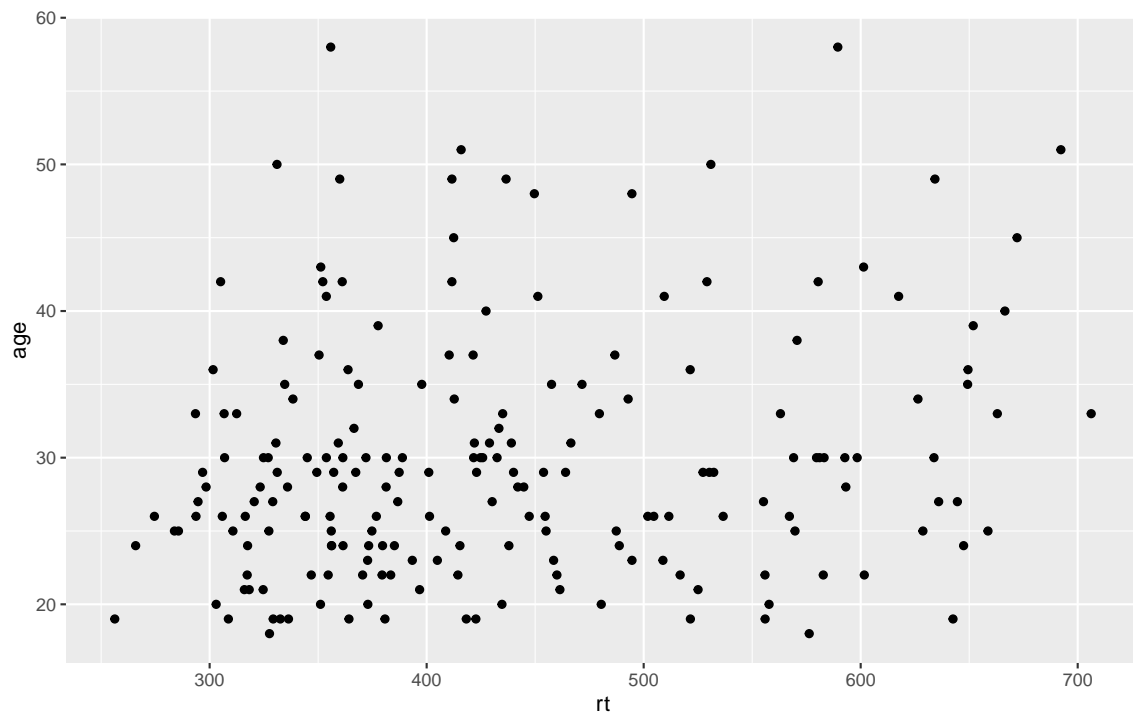
Figure 13. Point plot of reaction time versus age.

A line of best fit can be added with an additional layer that calls the function
`geom_smooth()`. The default is to draw a LOESS or curved regression line, however, a
linear line of best fit can be specified using `method = "lm"`. By default, `geom_smooth()`
will also draw a confidence envelope around the regression line, this can be removed by
adding `se = FALSE` to `geom_smooth()`. A common error is to try and use `geom_line()` to
draw the line of best fit, which whilst a sensible guess, will not work (try it).

```
ggplot(dat_long, aes(x = rt, y = age)) +
  geom_point() +
  geom_smooth(method = "lm")
```

Figure 14. Line of best fit for reaction time versus age.

⁵⁰⁵ **Grouped scatterplots.** Similar to the density plot, the scatterplot can also be
⁵⁰⁶ easily adjusted to display grouped data. For `geom_point()`, the grouping variable is mapped
⁵⁰⁷ to `colour` rather than `fill` and the relevant `scale_` function is added.

```
ggplot(dat_long, aes(x = rt, y = age, colour = condition)) +
  geom_point() +
  geom_smooth(method = "lm") +
  scale_colour_discrete(name = "Condition",
                        labels = c("Word", "Non-word"))
```

Figure 15. Grouped scatter plot of reaction time versus age by condition.

508 **Transforming data 2**

509   Following the rule that *anything that shares an axis should probably be in the same*
510 *column* means that we will frequently need our data in long-form when using `ggplot2`,
511 however, there are some cases when wide-form is necessary. For example, we may wish to
512 visualise the relationship between reaction time in the word and non-word conditions. The
513 easiest way to achieve this in our case would simply be to use the original wide-form data
514 as the input:

```
ggplot(dat, aes(x = rt_word, y = rt_nonword, colour = language)) +
  geom_point() +
  geom_smooth(method = "lm")
```
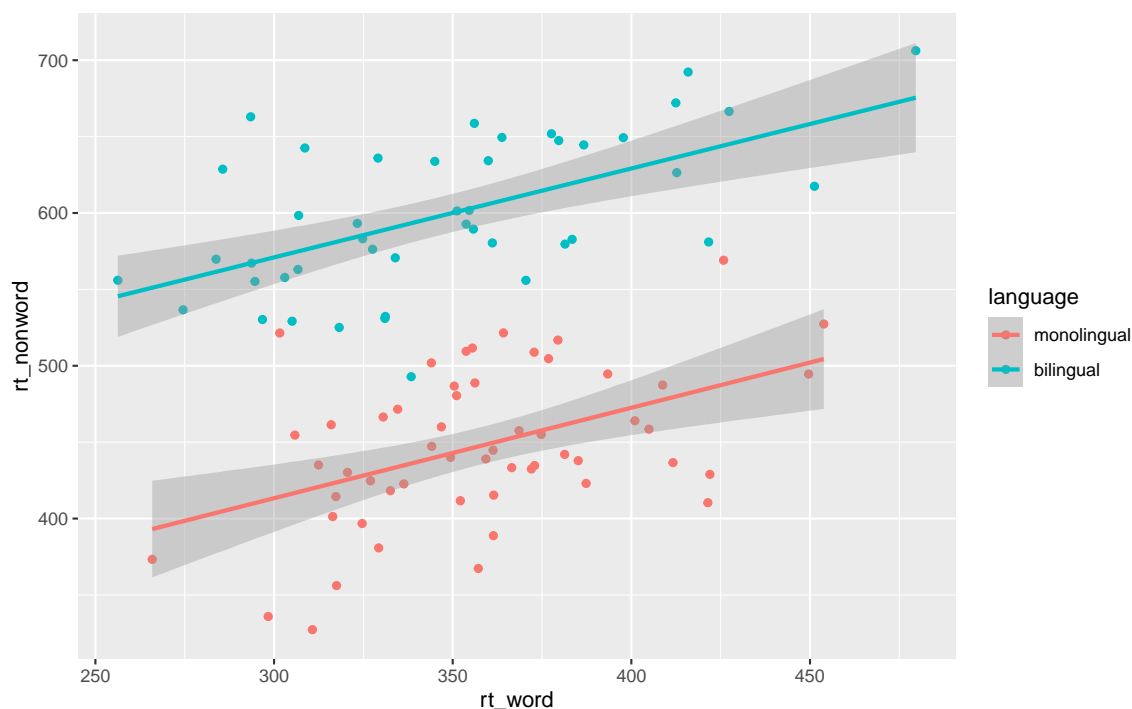
Figure 16. Scatterplot with data grouped by langauge group

⁵¹⁵ However, there may also be cases when you do not have an original wide-form version
⁵¹⁶ and you can use the `pivot_wider()` function to transform from long to wide.

```
dat_wide <- dat_long %>%
  pivot_wider(id_cols = "id",
              names_from = "condition",
              values_from = c(rt,acc))
```

id	rt_word	rt_nonword	acc_word	acc_nonword
S001	379.4585	516.8176	99	90
S002	312.4513	435.0404	94	82
S003	404.9407	458.5022	96	87
S004	298.3734	335.8933	92	76
S005	316.4250	401.3214	91	83
S006	357.1710	367.3355	96	78

⁵¹⁸ Customisation 2

⁵¹⁹ **Accessible colour schemes.** One of the drawbacks of using `ggplot2` for visuali-
⁵²⁰ sation is that the default colour scheme is not accessible (or visually appealing). The red
⁵²¹ and green default palette is difficult for colour-blind people to differentiate, and also does
⁵²² not display well in grey scale. You can specify exact custom colours for your plots, but one

₅₂₃ easy option is to use a colour palette and the `viridis` scale functions call such a palette.
₅₂₄ These take the same arguments as their default `scale` sister functions for updating axis
₅₂₅ names and labels, but display plots in contrasting colours that can be read by colour-blind
₅₂₆ people and that also print well in grey scale. The `viridis` scale functions provide a number
₅₂₇ of different options for the colour - try setting `option` to any letter from A - E to see the
₅₂₈ different sets.

```r
ggplot(dat_long, aes(x = rt, y = age, colour = condition)) +
  geom_point() +
  geom_smooth(method = "lm") +
  # use "viridis_d" instead of "discrete" for better colours
  scale_colour_viridis_d(name = "Condition",
                         labels = c("Word", "Non-word"),
                         option = "E")
```
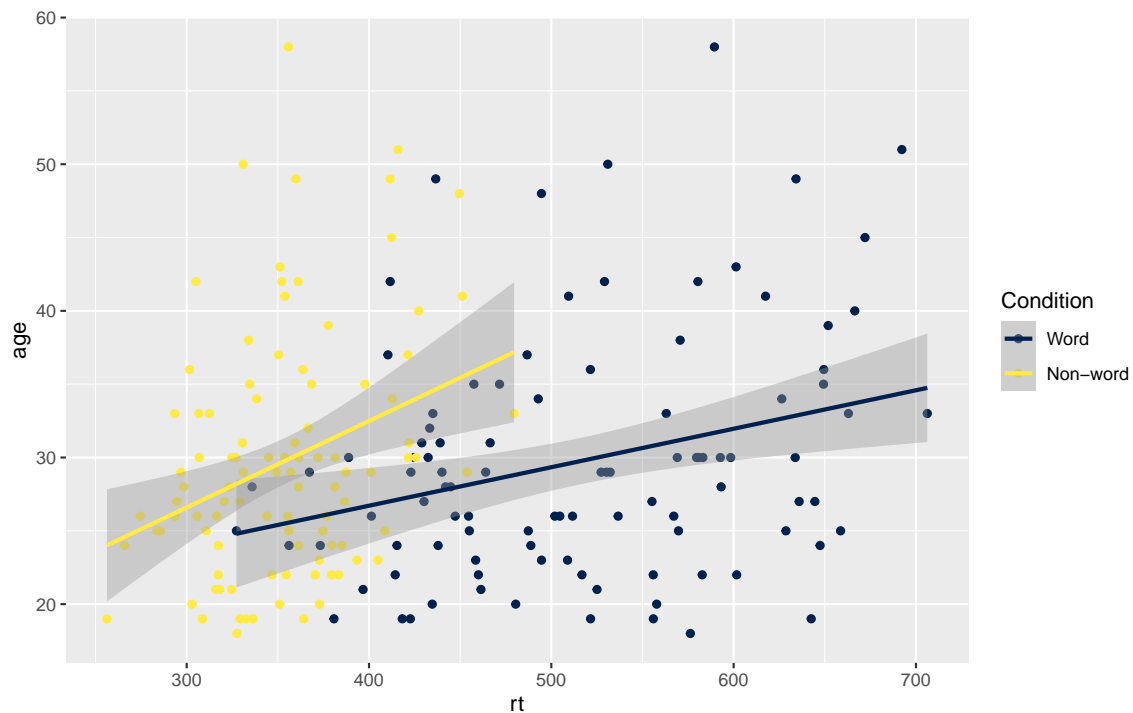

Figure 17. Use the viridis colour scheme for accessibility.

Specifying axis breaks with `seq()`

₅₃₀   Previously, when we have edited the `breaks` on the axis labels, we have done so
₅₃₁ manually, typing out all the values we want to display on the axis. For example, the below
₅₃₂ code edits the y-axis so that `age` is displayed in increments of 5.

```
ggplot(dat_long, aes(x = rt, y = age)) +
  geom_point() +
  scale_y_continuous(breaks = c(20,25,30,35,40,45,50,55,60))
```
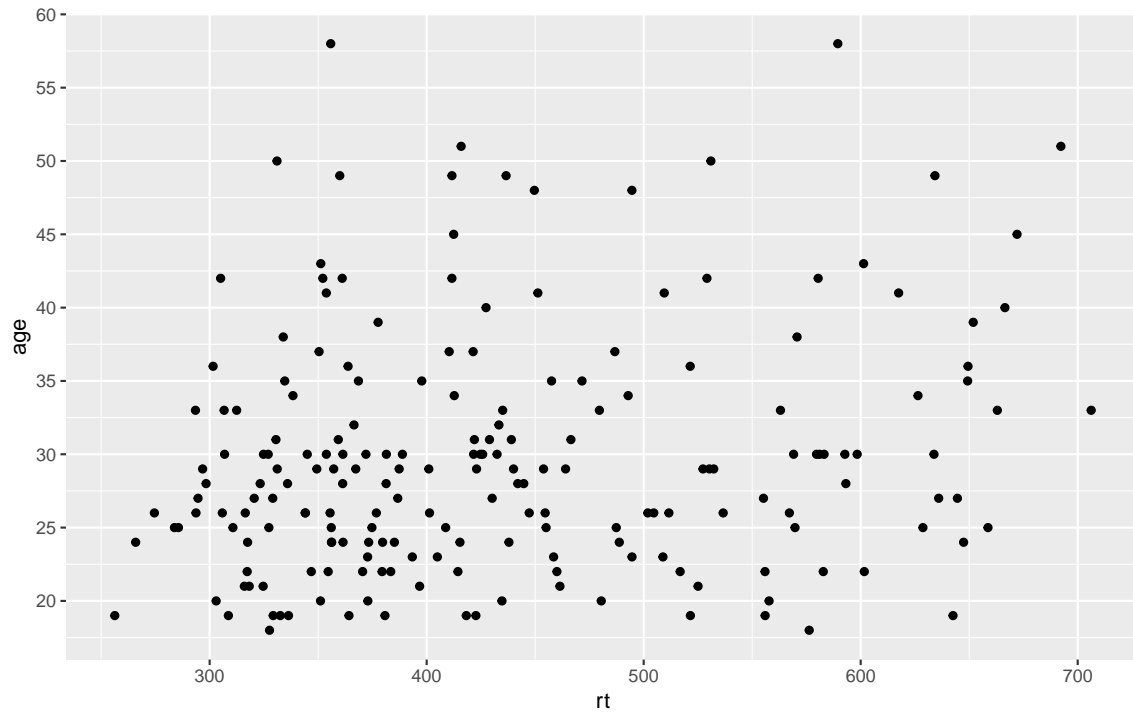

Figure 18

533 However, this is somewhat inefficient. Instead, we can use the function `seq()` to
534 specify the first and last value and the increments `by` which the breaks should display
535 between these two values.

```
ggplot(dat_long, aes(x = rt, y = age)) +
  geom_point() +
  scale_y_continuous(breaks = seq(20,60, by = 5))
```
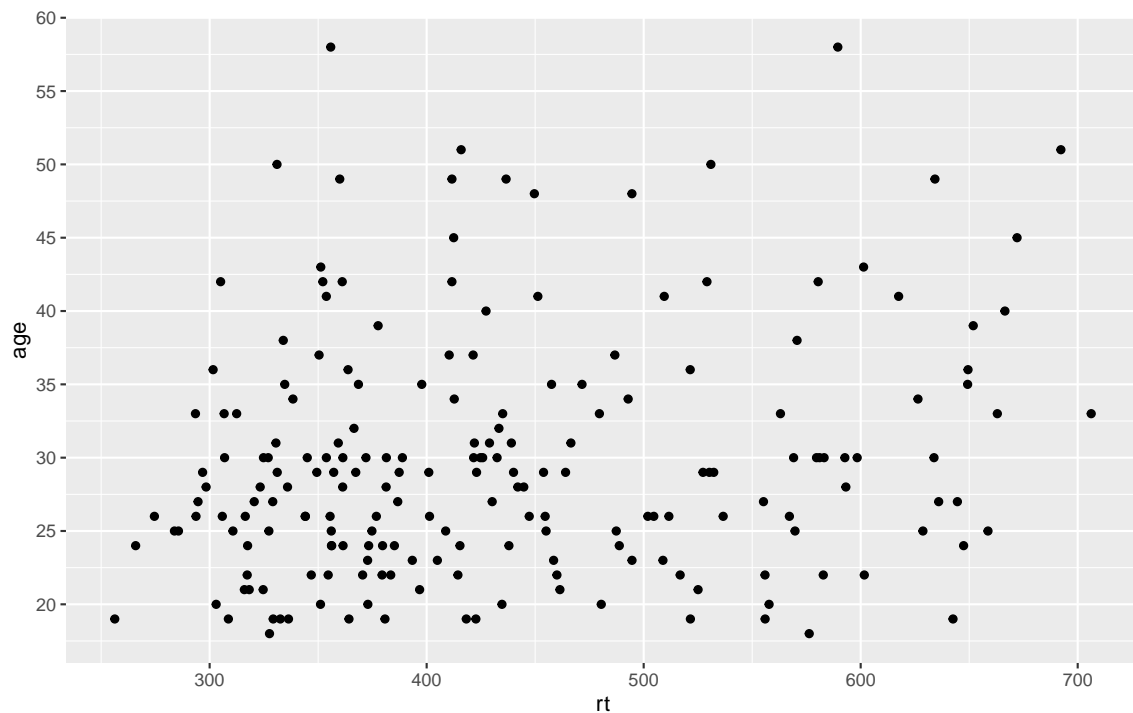
Figure 19

536 **Activities 2**

537 Before you move on try the following:

538 1. Use `fill` to created grouped histograms that display the distributions for `rt` for each
539 `language` group separately and also edit the fill axis labels. Try adding `position =`
540 `"dodge"` to `geom_histogram()` to see what happens.

541 2. Use `scale_*_*()` functions to edit the name of the x and y-axis on the scatterplot

542 3. Use `se = FALSE` to remove the confidence envelope from the scatterplots

543 4. Remove `method = "lm"` from `geom_smooth()` to produce a curved regression line.

544 5. Replace the default `scale_fill_*()` on the grouped density plot with the colour-blind
545 friendly version.

546   **Representing Summary Statistics**

547 The layering approach that is used in `ggplot2` to make figures comes into its own
548 when you want to include information about the distribution and spread of scores. In this
549 section we introduce different ways of including summary statistics on your figures.

⁵⁵⁰ **Boxplots**

⁵⁵¹   As with `geom_point()`, the boxplot geom also require an x and y-variable to be
⁵⁵² specified. In this case, x must be a discrete, or categorical variable, whilst y must be
⁵⁵³ continuous.

```
ggplot(dat_long, aes(x = condition, y = acc)) +
  geom_boxplot()
```
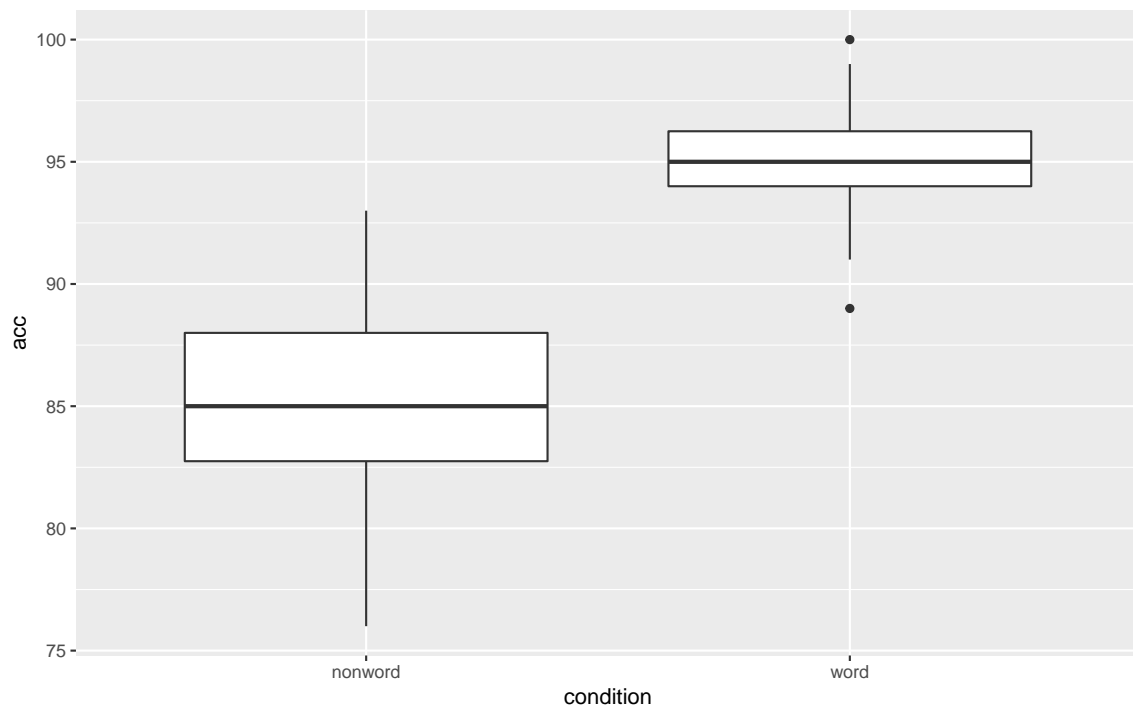

Figure 20. Basic boxplot.

⁵⁵⁴   **Grouped boxplots.**   As with histograms and density plots, `fill` can be used to
⁵⁵⁵ create grouped boxplots. This looks like a lot of complicated code at first glance, but most
⁵⁵⁶ of it is just editing the axis labels.

```
ggplot(dat_long, aes(x = condition, y = acc, fill = language)) +
  geom_boxplot() +
  scale_fill_viridis_d(option = "E",
                       name = "Group",
                       labels = c("Bilingual", "Monolingual")) +
  theme_classic() +
  scale_x_discrete(name = "Condition",
                   labels = c("Word", "Non-word")) +
  scale_y_continuous(name = "Accuracy")
```
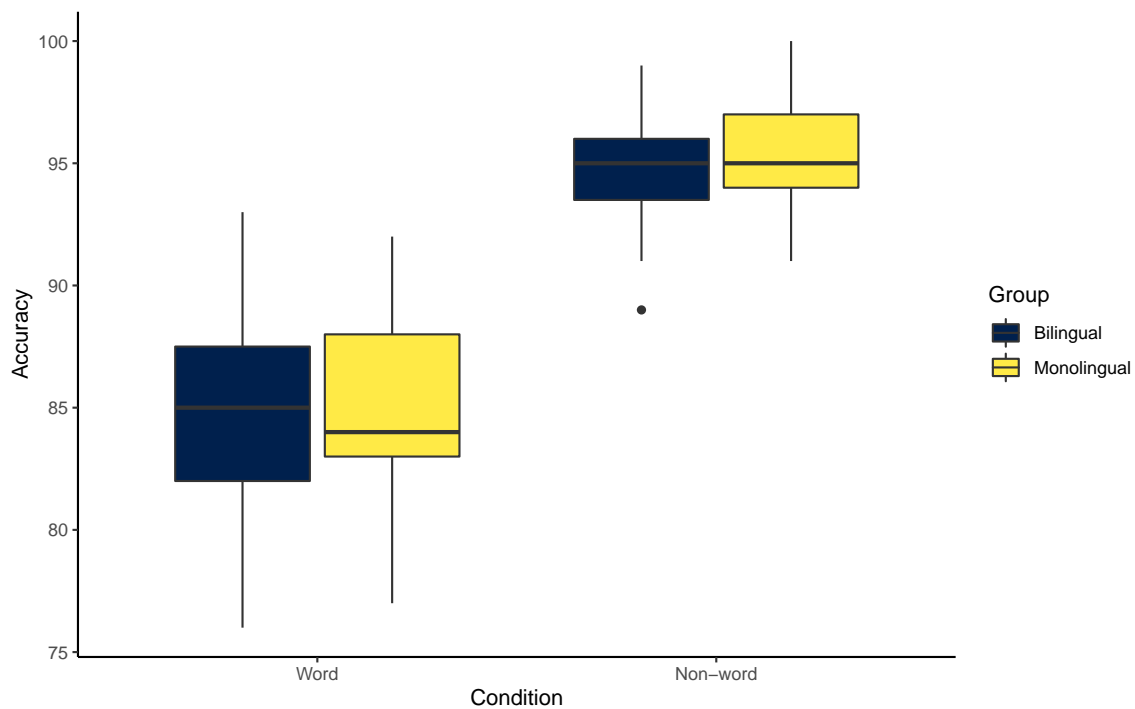
Figure 21. Grouped boxplots

Violin plots

Violin plots display the distribution of a dataset and can be created by calling `geom_violin()`. They are so-called because the shape they make sometimes looks something like a violin. They are essentially a mirrored density plot on its side. Note that the below code is identical to the code used to draw the boxplots above, except for the call to `geom_violin()` rather than `geom_boxplot()`.

```
ggplot(dat_long, aes(x = condition, y = acc, fill = language)) +
  geom_violin() +
  scale_fill_viridis_d(option = "D",
                       name = "Group",
                       labels = c("Bilingual", "Monolingual")) +
  theme_classic() +
  scale_x_discrete(name = "Condition",
                   labels = c("Word", "Non-word")) +
  scale_y_continuous(name = "Accuracy")
```
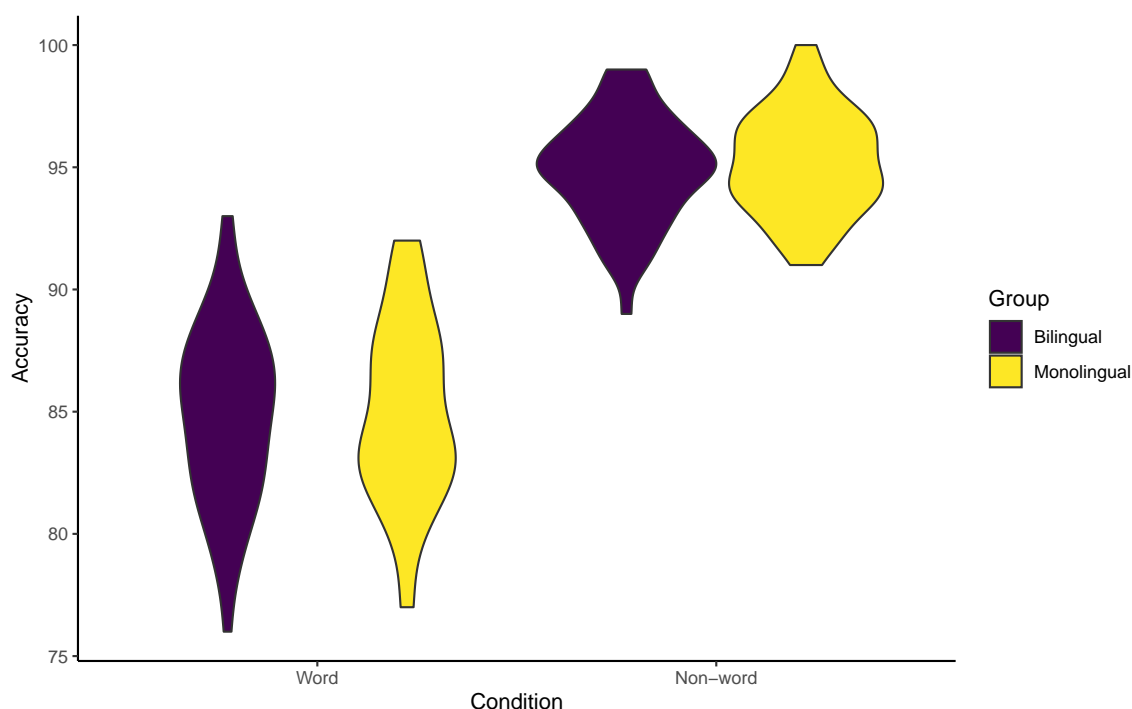
Figure 22. Violin plot.

Bar chart of means

Commonly, rather than visualising distributions of raw data researchers will wish to visualise means using a bar chart with error bars. As with SPSS and Excel, `ggplot2` requires you to calculate the summary statistics and then plot the summary. There are at least two ways to do this, in the first you make a table of summary statistics as we did earlier when calculating the participant demographics and then plot that table. The second approach is to calculate the statistics within a layer of the plot. That is the approach we will use below.

First we present code for making a bar chart. The code for bar charts is here because it is a common visualisation that is familiar to most researchers, however, we would urge you to use a visualisation that provides more transparency about the distribution of the raw data, such as the violin-boxplots we will present in the next section.

To summarise the data into means we use a new function `stat_summary`. Rather than calling a `geom_*` function, we call `stat_summary()` and specify how we want to summarise the data and how we want to present that summary in our figure.

- `fun` specifies the summary function that gives us the y-value we want to plot, in this case, `mean`.

- `geom` specifies what shape or plot we want to use to display the summary. For the first layer we will specify `bar`. As with the other geom-type functions we have shown

582   you, this part of the `stat_summary()` function is tied to the aesthetic mapping in the
583   first line of code. The underlying statistics for a bar chart means that we must specify
584   and IV (x-axis) as well as the DV (y-axis).

```
ggplot(dat_long, aes(x = condition, y = rt)) +
  stat_summary(fun = "mean", geom = "bar")
```
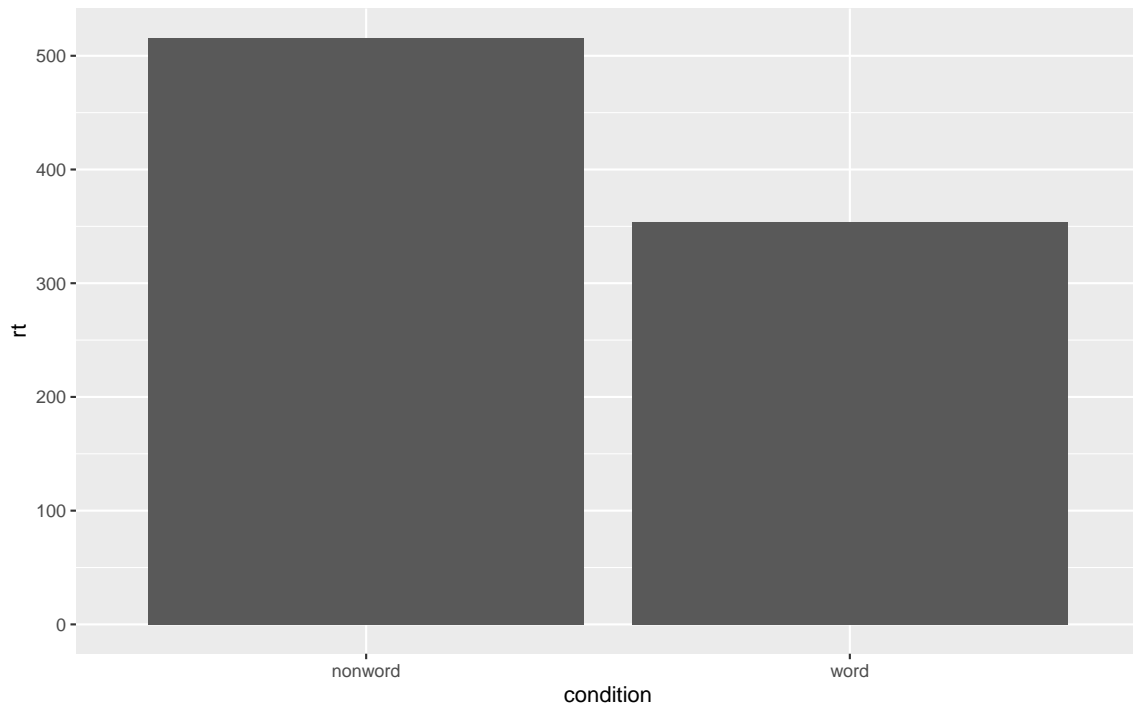

Figure 23. Bar plot of means.

585   To add the error bars, another layer is added with a second call to `stat_summary`.
586   This time, the function represents the type of error bars we wish to draw, you can choose
587   from `mean_se` for standard error, `mean_cl_normal` for confidence intervals, or `mean_sdl` for
588   standard deviation. `width` controls the width of the error bars - try changing the value to
589   see what happens.

590   • Whilst `fun` returns a single value (y) per condition, `fun.data` returns the y-values we
591   want to plot plus their minimum and maximum values, in this case, `mean_se`

```
ggplot(dat_long, aes(x = condition, y = rt)) +
  stat_summary(fun = "mean", geom = "bar") +
  stat_summary(fun.data = "mean_se", geom = "errorbar", width = .2)
```
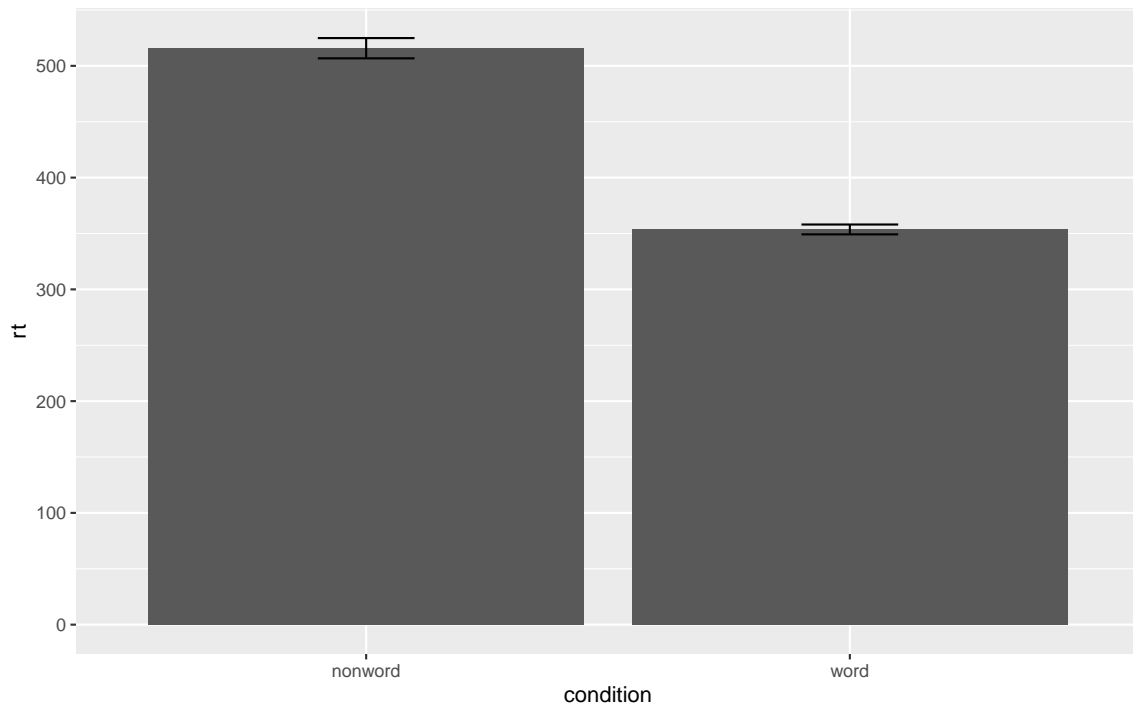
*Figure 24*. Bar plot of means with error bars representing SE.

Violin-boxplot

The power of the layered system for making figures is further highlighted by the ability to combine different types of plots. For example, rather than using a bar chart with error bars, one can easily create a single plot that includes density of the distribution, confidence intervals, means and standard errors. In the below code we first draw a violin plot, then layer on a boxplot, a point for the mean (note `geom = "point"` instead of `"bar"`) and standard error bars (`geom = "errorbar"`). This plot does not require much additional code to produce than the bar plot with error bars, yet the amount of information displayed is vastly superior.

- `fatten = NULL` in the boxplot geom removes the median line, which can make it easier to see the mean and error bars. Including this argument will result in the warning message `Removed 1 rows containing missing values (geom_segment)` and is not a cause for concern. Removing this argument will reinstate the median line.

```
ggplot(dat_long, aes(x = condition, y= rt)) +
  geom_violin() +
  # remove the median line with fatten = NULL
  geom_boxplot(width = .2, fatten = NULL) +
  stat_summary(fun = "mean", geom = "point") +
  stat_summary(fun.data = "mean_se", geom = "errorbar", width = .1)
```
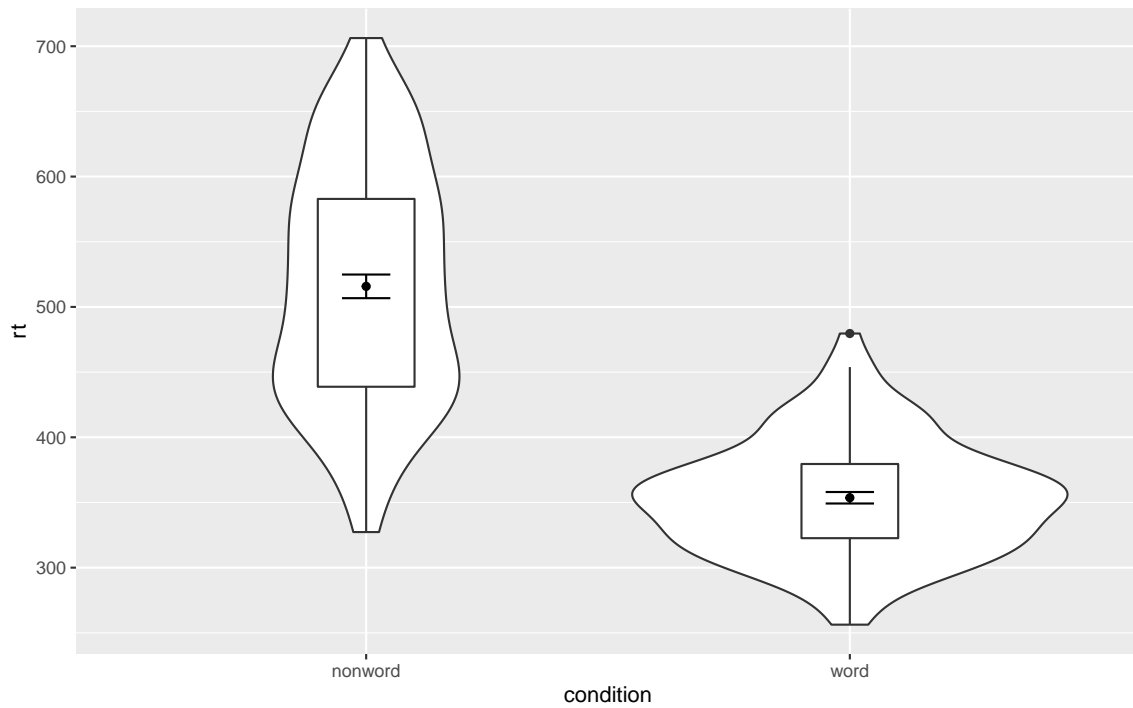
Figure 25. Violin-boxplot with mean dot and standard error bars.

₆₀₅   It is important to note that the order of the layers matters and it is worth experiment-
₆₀₆ ing with the order to see where the order matters. For example, if we call `geom_boxplot()`
₆₀₇ followed by `geom_violin()`, we get the following mess:

```
ggplot(dat_long, aes(x = condition, y= rt)) +
  geom_boxplot() +
  geom_violin() +
  stat_summary(fun = "mean", geom = "point") +
  stat_summary(fun.data = "mean_se", geom = "errorbar", width = .1)
```
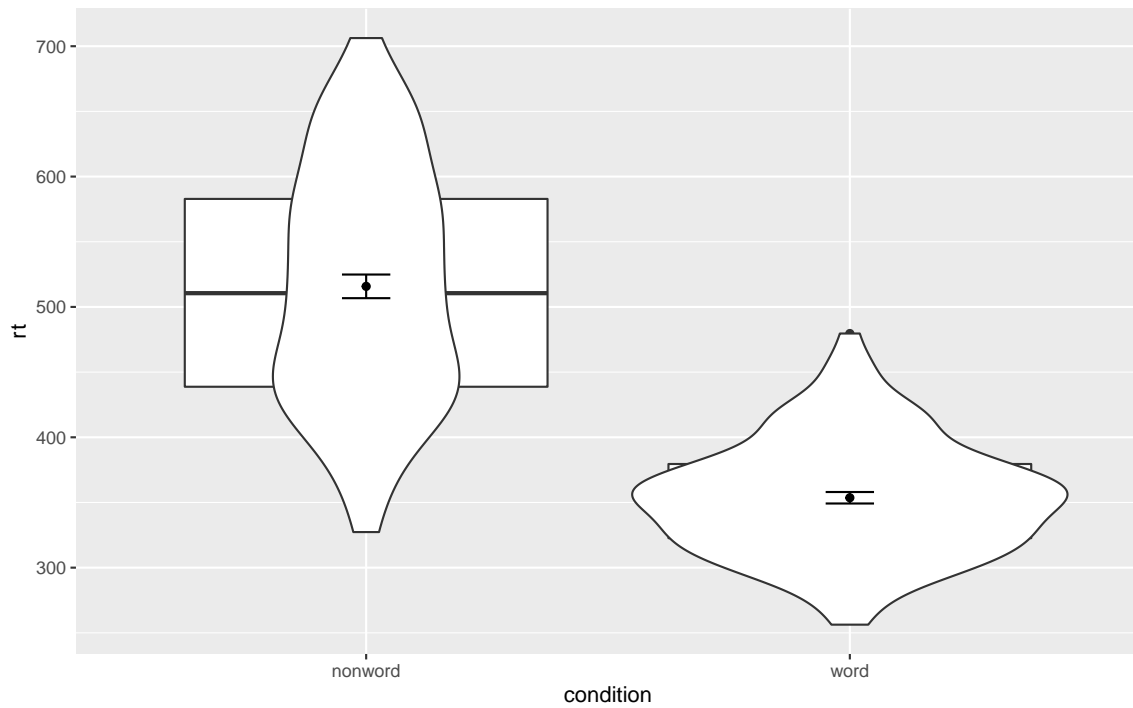
Figure 26. Plot with the geoms in the wrong order.

⁶⁰⁸   **Grouped violin-boxplots.** As with previous plots, another variable can be
⁶⁰⁹ mapped to `fill` for the violin-boxplot. However, simply adding `fill` to the mapping
⁶¹⁰ causes the different components of the plot to become misaligned because they have differ-
⁶¹¹ ent default positions:

```
ggplot(dat_long, aes(x = condition, y= rt, fill = language)) +
  geom_violin() +
  geom_boxplot(width = .2, fatten = NULL) +
  stat_summary(fun = "mean", geom = "point") +
  stat_summary(fun.data = "mean_se", geom = "errorbar", width = .1)
```
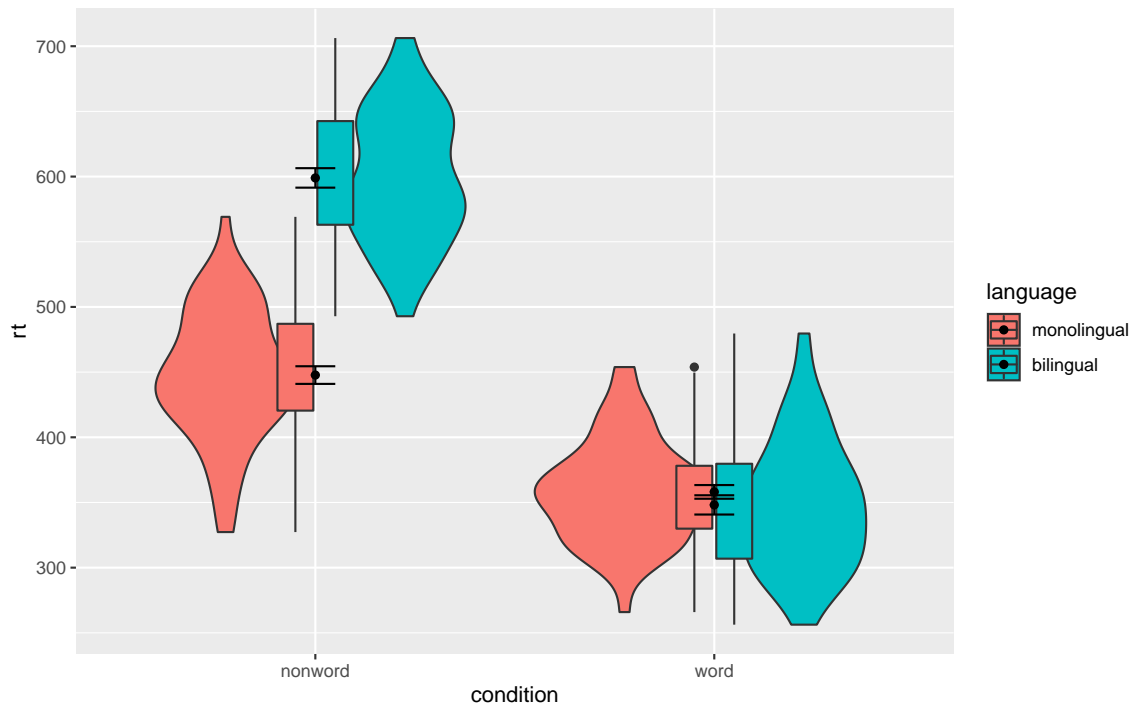
Figure 27. Grouped violin-boxplots without repositioning.

612   To rectify this we need to adjust the argument `position` for each of the misaligned
613 layers. `position_dodge()` instructs R to move (dodge) the position of the plot component
614 by the specified value - finding what value you need can sometimes take trial and error.

```
ggplot(dat_long, aes(x = condition, y= rt, fill = language)) +
  geom_violin() +
  geom_boxplot(width = .2, fatten = NULL, position = position_dodge(.9)) +
  stat_summary(fun = "mean", geom = "point",
               position = position_dodge(.9)) +
  stat_summary(fun.data = "mean_se", geom = "errorbar", width = .1,
               position = position_dodge(.9))
```
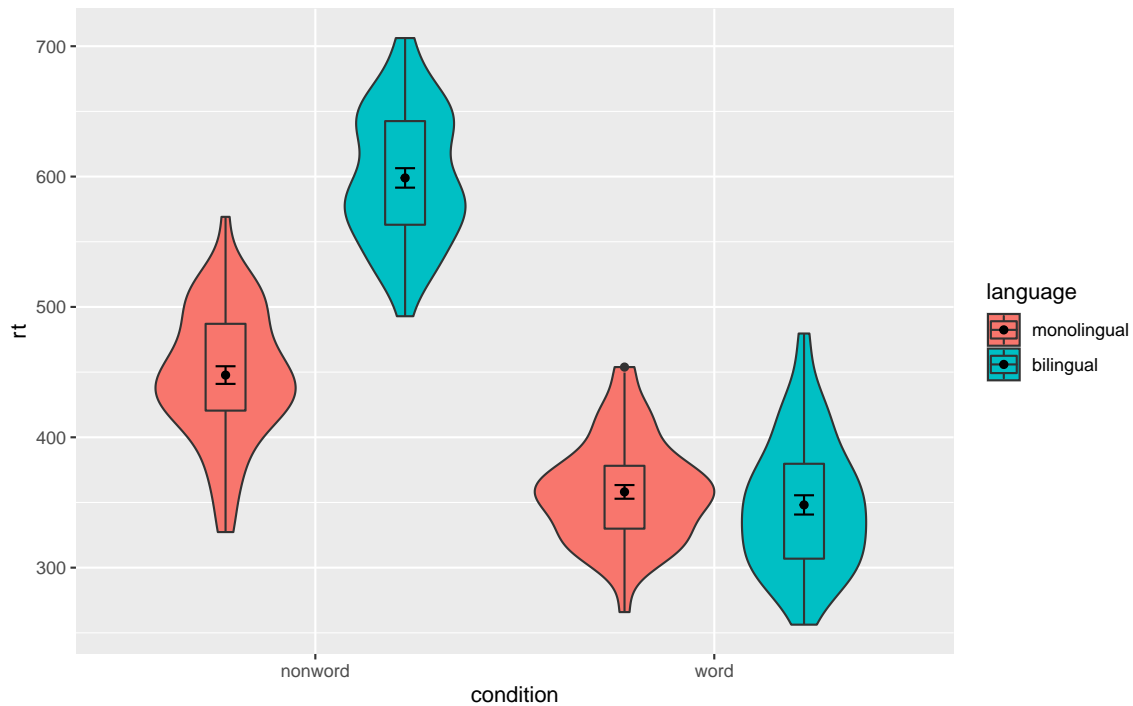
Figure 28. Grouped violin-boxplots with repositioning.

Customisation part 3

Combining multiple type of plots can present an issue with the colours, particularly when the viridis scheme is used - in the below example it is hard to make out the black lines of the boxplot and the mean/error bars.

```
ggplot(dat_long, aes(x = condition, y= rt, fill = language)) +
  geom_violin() +
  geom_boxplot(width = .2, fatten = NULL, position = position_dodge(.9)) +
  stat_summary(fun = "mean", geom = "point",
               position = position_dodge(.9)) +
  stat_summary(fun.data = "mean_se", geom = "errorbar", width = .1,
               position = position_dodge(.9)) +
  scale_fill_viridis_d(option = "E") +
  theme_minimal()
```
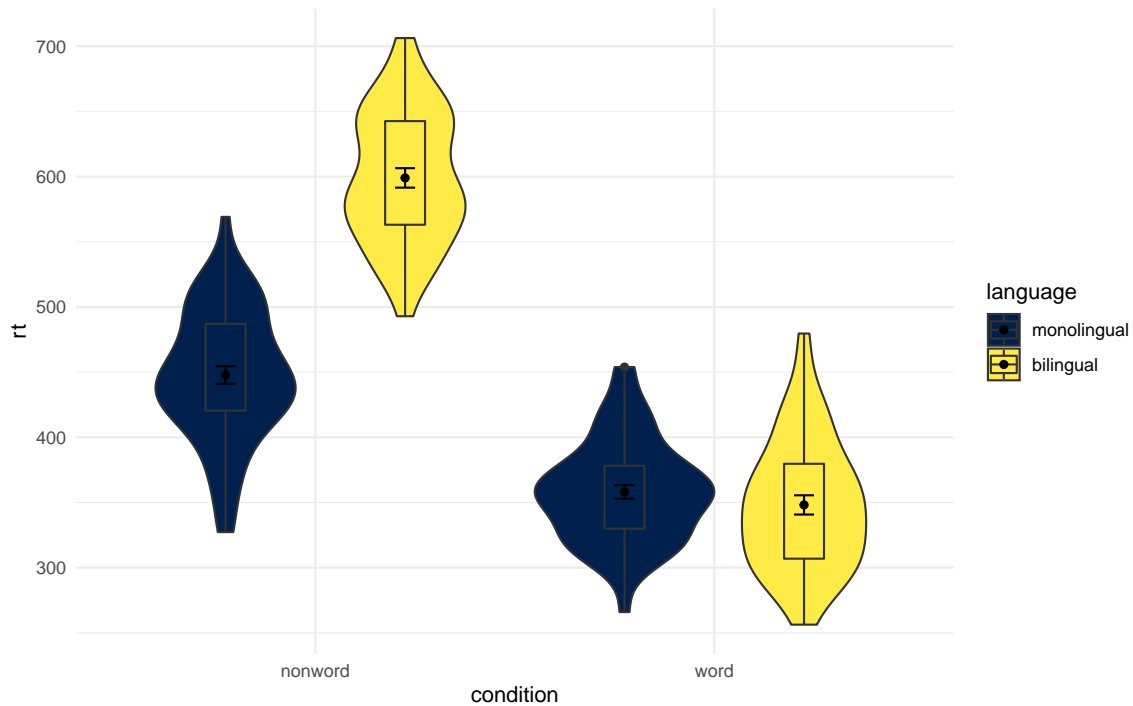
Figure 29. A color scheme that makes lines difficult to see.

619   There are a number of solutions to this problem. First, we can change the colour of
620 individual geoms by adding `colour = "colour"` to each relevant geom:

```
ggplot(dat_long, aes(x = condition, y= rt, fill = condition)) +
  geom_violin() +
  geom_boxplot(width = .2, fatten = NULL, colour = "grey") +
  stat_summary(fun = "mean", geom = "point", colour = "grey") +
  stat_summary(fun.data = "mean_se", geom = "errorbar", width = .1, colour = "grey") +
  scale_fill_viridis_d(option = "E") +
  theme_minimal()
```
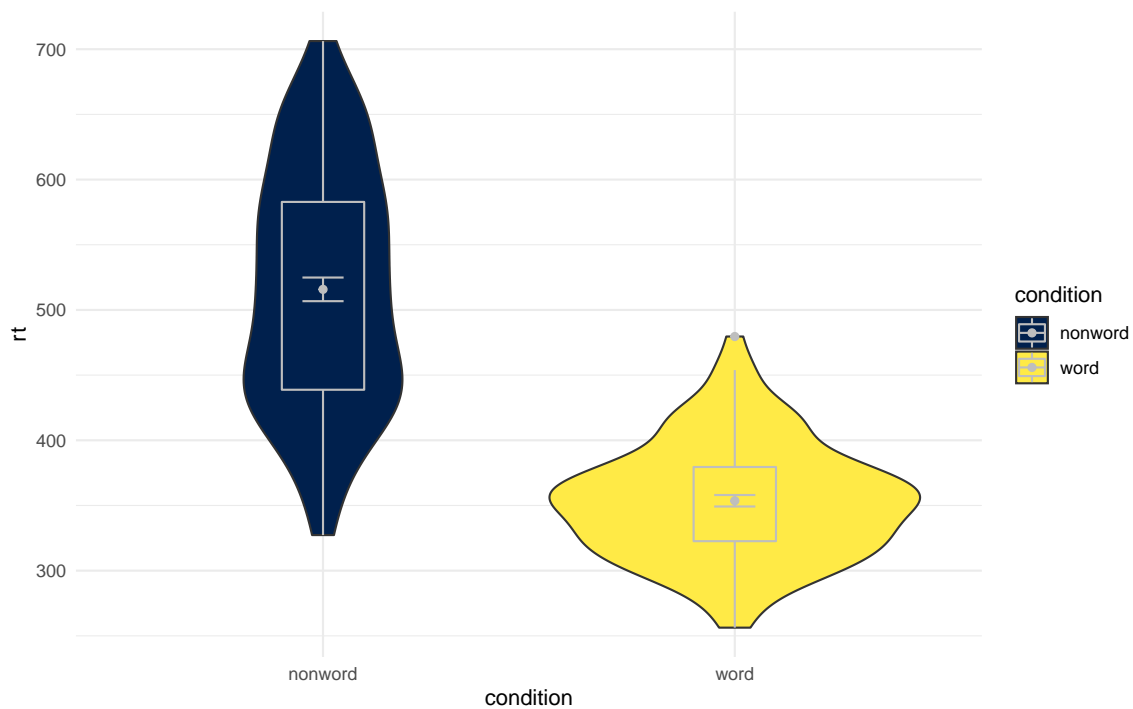
Figure 30. Manually changing the line colors.

621   We can also keep the original colours but adjust the transparency of each layer using
622   `alpha`. Again, the exact values needed can take trial and error:

```
ggplot(dat_long, aes(x = condition, y= rt, fill = condition)) +
  geom_violin(alpha = .4) +
  geom_boxplot(width = .2, fatten = NULL, alpha = .5) +
  stat_summary(fun = "mean", geom = "point") +
  stat_summary(fun.data = "mean_se", geom = "errorbar", width = .1) +
  scale_fill_viridis_d(option = "E") +
  theme_minimal()
```
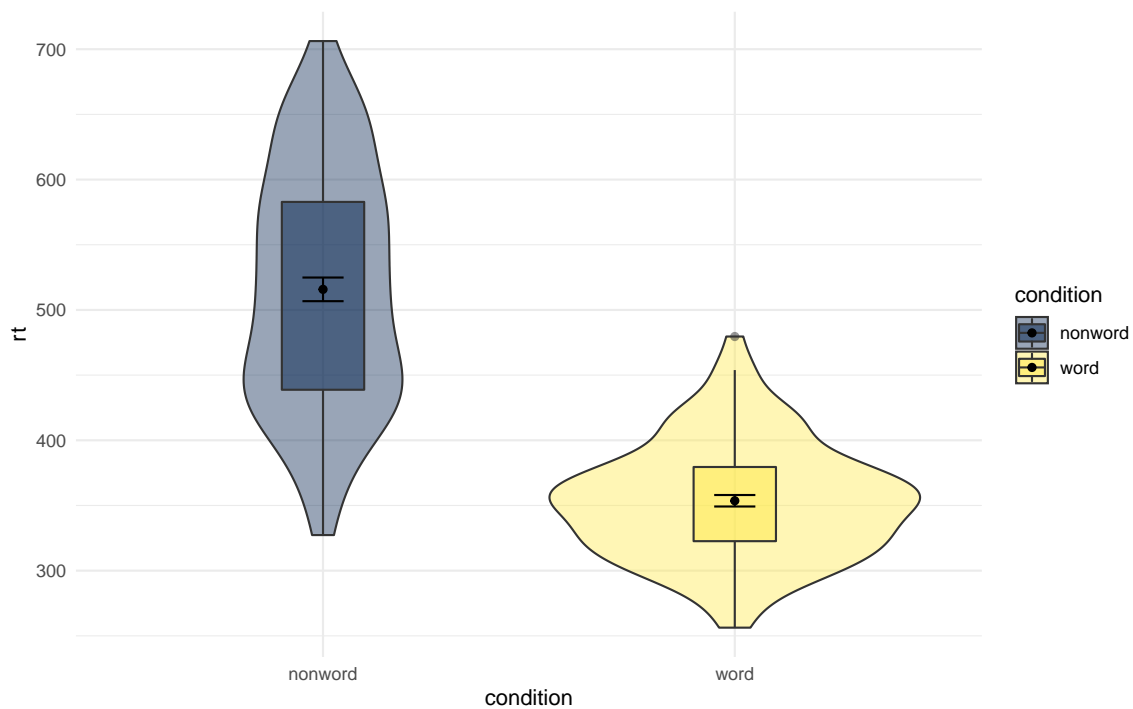
Figure 31. Using transparency on the fill color.

623 **Activities 3**

624   Before you go on, do the following:

625 1. Review all the code you have run so far. Try to identify the commonalities between
626   each plot's code and the bits of the code you might change if you were using a different
627   dataset.

628 2. Take a moment to recognise the complexity of the code you are now able to read.

629 3. For the violin-boxplot, for `geom = "point"`, try changing `fun` to `median`

630 4. For the violin-boxplot, for `geom = "errorbar"`, try changing `fun.data` to
631   `mean_cl_normal` (for 95% CI)

632 5. Go back to the grouped density plots and try changing the transparency with `alpha`.

633   **Multi-part Plots**

634 **Interaction plots**

635   Interaction plots are commonly used to help display or interpret a factorial design.
636 Just as with the bar chart of means, interaction plots represent data summaries and so they
637 are built up with a series of calls to `stat_summary()`.

₆₃₈ • `shape` acts much like `fill` in previous plots, except that rather than producing dif-
₆₃₉   ferent colour fills for each level of the IV, the data points are given different shapes.

₆₄₀ • `size` lets you change the size of lines and points. You usually don't want different
₆₄₁   groups to be different sizes, so this option is set inside the relevant `geom_*()` function,
₆₄₂   not inside the `aes()` function.

₆₄₃ • `scale_color_manual()` works much like `scale_color_discrete()` except that it
₆₄₄   lets you specify the colour values manually, instead of then being automatically ap-
₆₄₅   plied based on the palette you choose/default to. You can specify RGB colour values
₆₄₆   or a list of predefined colour names - all available options can be found by run-
₆₄₇   ning `colours()` in the console. Other manual scales are also available, for example,
₆₄₈   `scale_fill_manual`.

```r
ggplot(dat_long, aes(x = condition, y = rt,
                     shape = language,
                     group = language,
                     color = language)) +
  stat_summary(fun = "mean", geom = "point", size = 3) +
  stat_summary(fun = "mean", geom = "line") +
  stat_summary(fun.data = "mean_se", geom = "errorbar", width = .2) +
  scale_color_manual(values = c("blue", "darkorange")) +
  theme_classic()
```
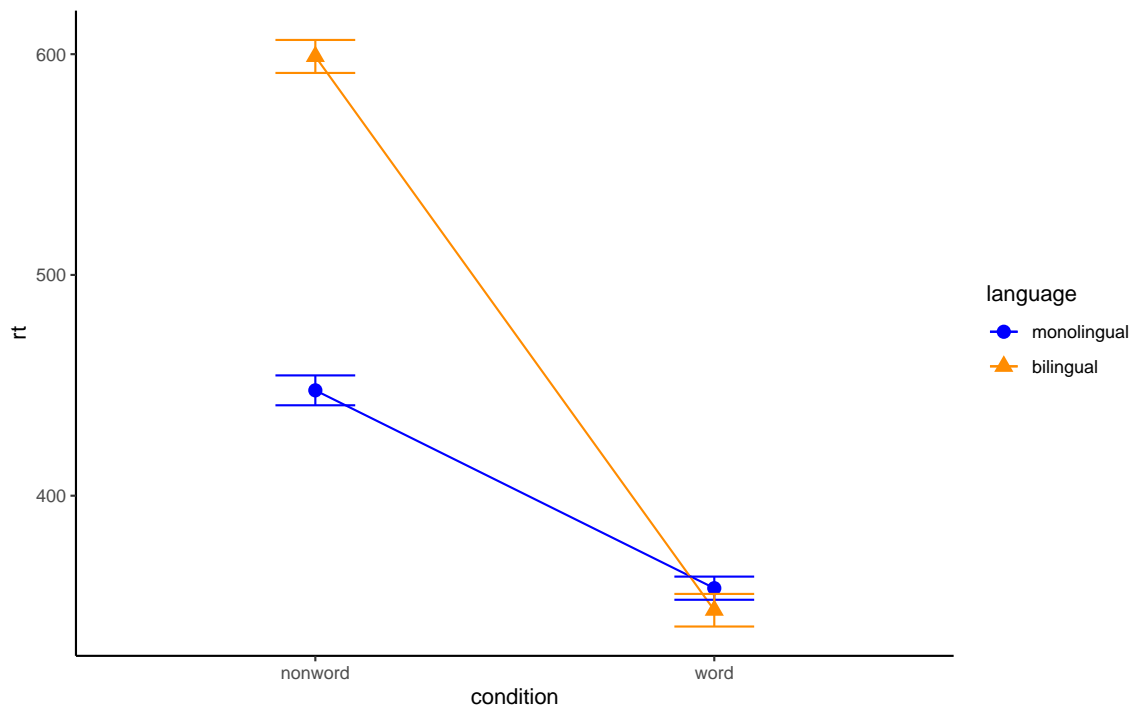

Figure 32. Interaction plot.

Combined interaction plots

A more complex interaction plot can be produced that takes advantage of the layers to visualise not only the overall interaction, but the change across conditions for each participant.

This code is more complex than all prior code because it does not use a universal mapping of the plot aesthetics. In our code so far, the aesthetic mapping (`aes`) of the plot has been specified in the first line of code as all layers have used the same mapping, however, is is also possible for each layer to use a different mapping.

- The first call to `ggplot()` sets up the default mappings of the plot that will be used unless otherwise specified - the `x`, `y` and `group` variable. Note two additions are `shape` and `linetype` that will vary those elements according to the language variable.
- `geom_point()` overrides the default mapping by setting its own `colour` to draw the data points from each language group in a different colour. `alpha` is set to a low value to aid readability. Note that because the aesthetic override was defined within the geom function, the colours are not represented in the legend.
- Similarly, `geom_line()` overrides the default grouping variable so that a line is drawn to connect the individual data points for each *participant* (`group = id`) rather than each language group, and also sets the colours. The default line type is also overridden and set for all lines to be solid.
- Finally, the calls to `stat_summary()` remain largely as they were, with the exception of setting `colour = "black"` and `size = 2` so that the overall means and error bars can be more easily distinguished from the individual data points. Because they do not specify an individual mapping, they use the defaults (e.g., the lines are connected by language group). For the error bars the lines are again made solid.

```
ggplot(dat_long, aes(x = condition, y = rt,
                     group = language, shape = language)) +
  geom_point(aes(colour = language),alpha = .2) +
  geom_line(aes(group = id, colour = language), alpha = .2) +
   stat_summary(fun = "mean", geom = "point", size = 2, colour = "black") +
  stat_summary(fun = "mean", geom = "line", colour = "black") +
  stat_summary(fun.data = "mean_se", geom = "errorbar",
               width = .2, colour = "black") +
  theme_minimal()
```
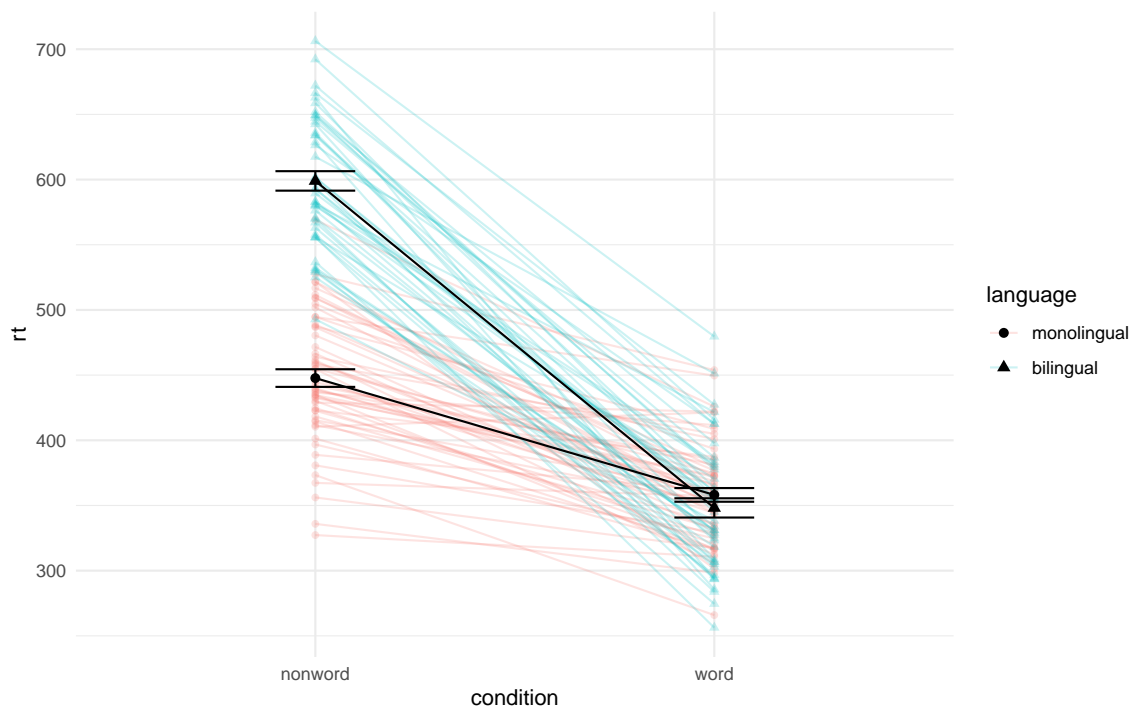
Figure 33. Interaction plot with by-participant data

⁶⁷³ **Facets**

⁶⁷⁴   So far we have produced single plots that display all the desired variables in one,
⁶⁷⁵ however, there are situations in which it may be useful to create separate plots for each
⁶⁷⁶ level of a variable. The below code is an adaptation of the code used to produce the grouped
⁶⁷⁷ scatterplot (see Figure 27) in which it may be easier to see how the relationship changes
⁶⁷⁸ when the data are not overlaid.

⁶⁷⁹   • Rather than using `colour = condition` to produce different colours for each level of
⁶⁸⁰   condition, this variable is instead passed to `facet_wrap()`.

```
ggplot(dat_long, aes(x = rt, y = age)) +
  geom_point() +
  geom_smooth(method = "lm") +
  facet_wrap(~condition)
```
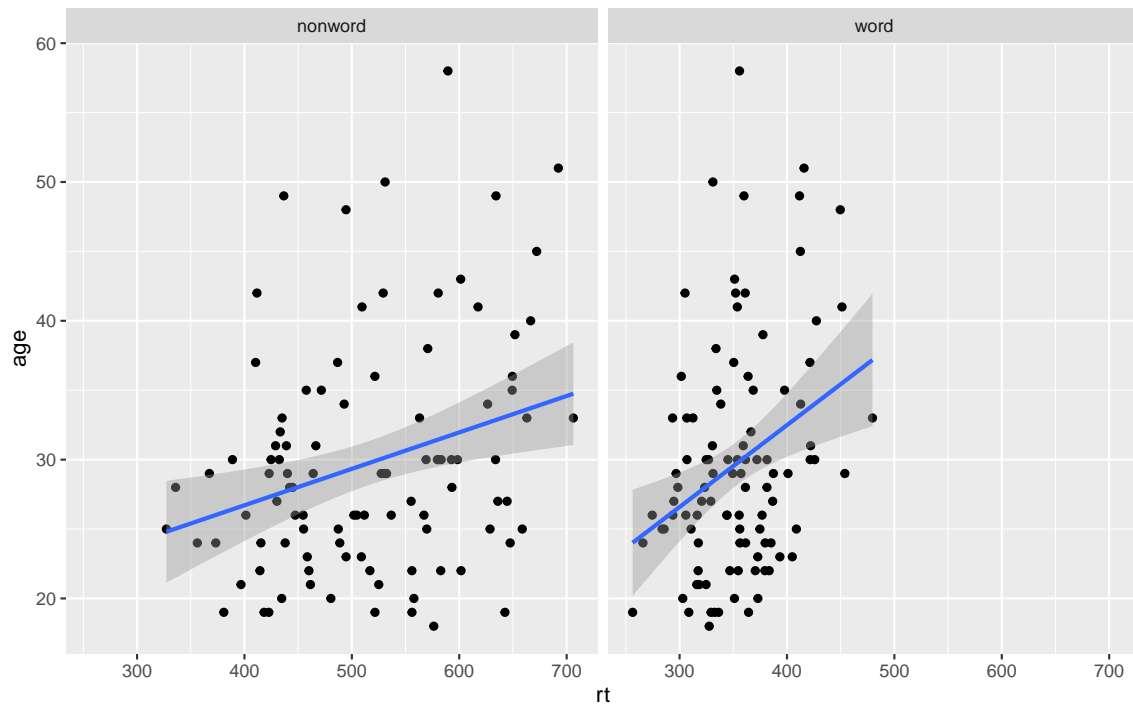
Figure 34. Faceted scatterplot

681   As another example, we can use `facet_wrap()` as an alternative to the grouped
682 violin-boxplot (see Figure 28) in which the variable `language` is passed to `facet_wrap()`
683 rather than `fill`.

```
ggplot(dat_long, aes(x = condition, y= rt)) +
  geom_violin() +
  geom_boxplot(width = .2, fatten = NULL) +
  stat_summary(fun = "mean", geom = "point") +
  stat_summary(fun.data = "mean_se", geom = "errorbar", width = .1) +
  facet_wrap(~language) +
  theme_minimal()
```
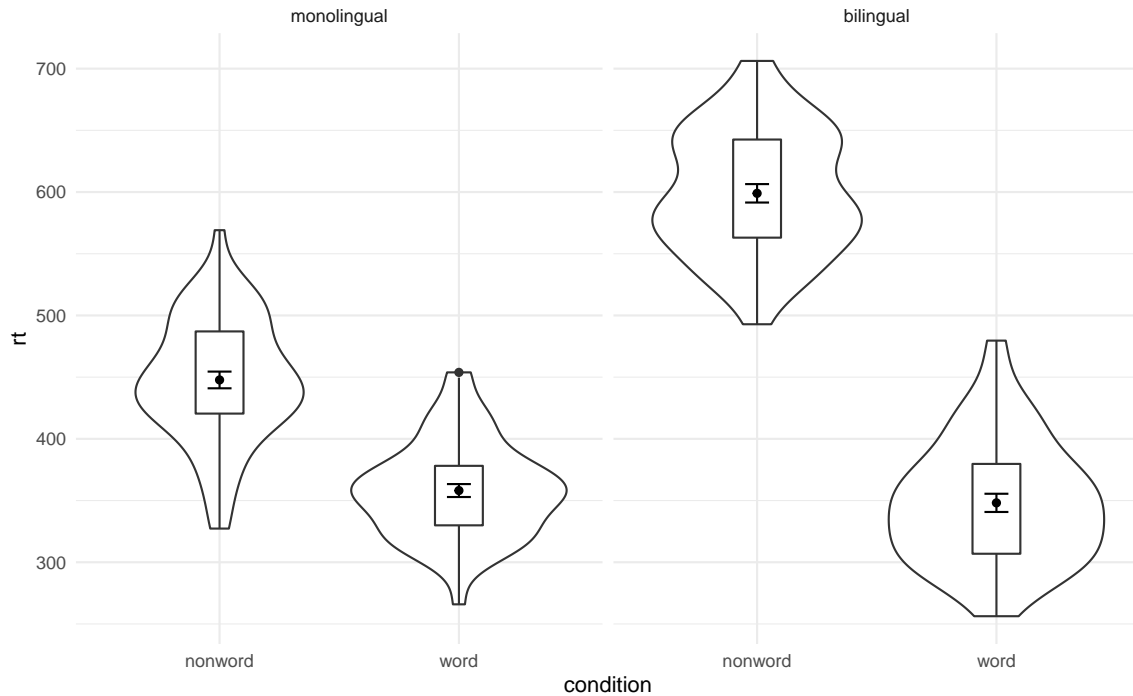
Figure 35. Facted violin-boxplot

Finally, note that editing the labels for faceted variables involves converting the `language` column into a factor. This allows you to set the order of the `levels` and the `labels` to display.

```
ggplot(dat_long, aes(x = condition, y= rt)) +
  geom_violin() +
  geom_boxplot(width = .2, fatten = NULL) +
  stat_summary(fun = "mean", geom = "point") +
  stat_summary(fun.data = "mean_se", geom = "errorbar", width = .1) +
  facet_wrap(~factor(language,
                     levels = c("monolingual", "bilingual"),
                     labels = c("Monolingual participants",
                                "Bilingual participants"))) +
  theme_minimal()
```
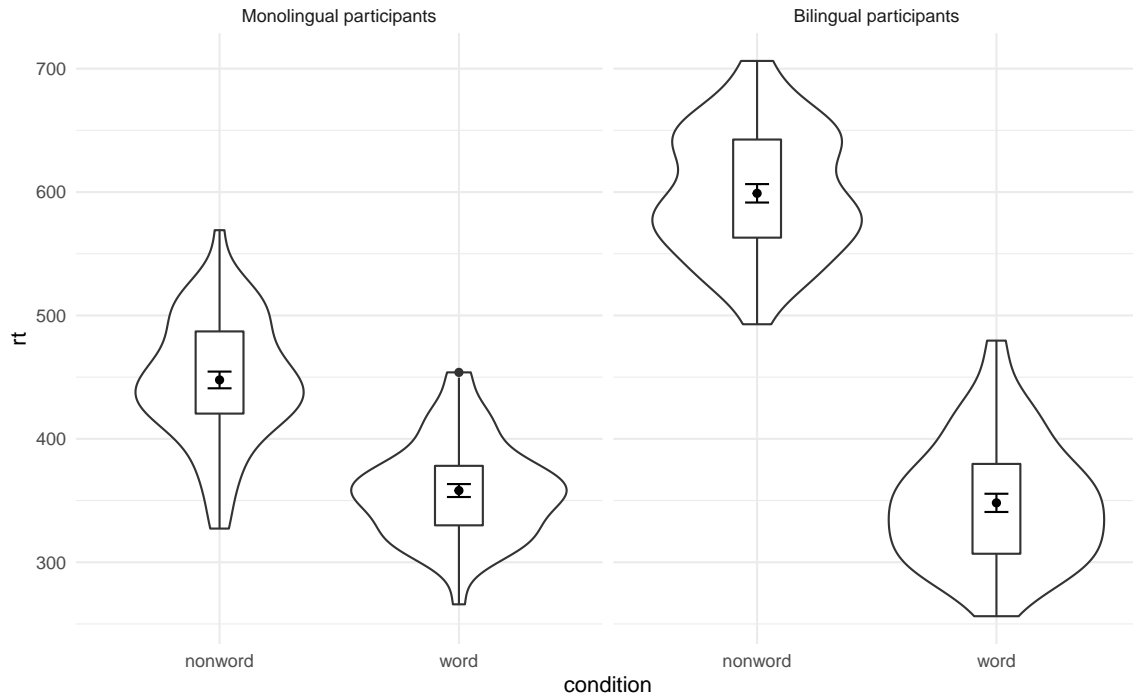
Figure 36. Faceted violin-boxplot with updated labels

Storing plots

Just like with datasets, plots can be saved to objects. The below code saves the histograms we produced for reaction time and accuracy to objects named `p1` and `p2`. These plots can then be viewed by calling the object name in the console.

```
p1 <- ggplot(dat_long, aes(x = rt)) +
  geom_histogram(binwidth = 10, color = "black")

p2 <- ggplot(dat_long, aes(x = acc)) +
  geom_histogram(binwidth = 1, color = "black")
```

Importantly, layers can then be added to these saved objects. For example, the below code adds a theme to the plot saved in `p1` and saves it as a new object `p3`. This is important because many of the examples of `ggplot2` code you will find in online help forums use the `p +` format to build up plots but fail to explain what this means, which can be confusing to beginners.

```
p3 <- p1 + theme_minimal()
```

Saving plots as images

In addition to saving plots to objects for further use in R, the function `ggsave()` can be used to save plots as images on your hard drive. The only required argument for `ggsave` is the file name of the image file you will create, complete with file extension (this can be "eps," "ps," "tex," "pdf," "jpeg," "tiff," "png," "bmp," "svg" or "wmf"). By default, `ggsave()` will save the last plot displayed, however, you can also specify a specific plot object if you have one saved.

```
ggsave(filename = "my_plot.png") # save last displayed plot
ggsave(filename = "my_plot.png", plot = p3) # save plot p3
```

The width, height and resolution of the image can all be manually adjusted and the help documentation for is useful here (type `?ggsave` in the console to access the help).

Multiple plots

As well as creating separate plots for each level of a variable using `facet_wrap()`, you may also wish to display multiple different plots together and the `patchwork` package provides an intuitive way to do this. `patchwork` does not require the use of any functions once it is loaded with `library(patchwork)`, you simply need to save the plots you wish to combine to objects as above and use the operators `+`, `/ ()` and `|` to specify the look of the final figure.

Combining two plots. Two plots can be combined side-by-side or stacked on top of each other. These combined plots could also be saved to an object and then passed to `ggsave`.
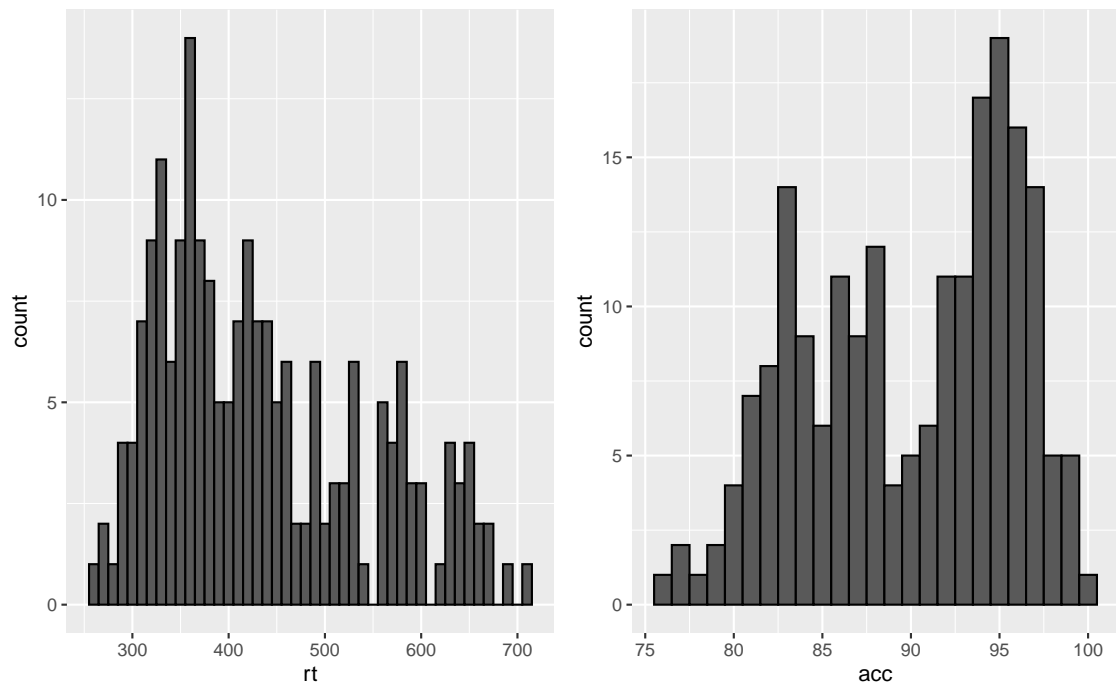
```
p1 + p2 # side-by-side
```

Figure 37. Side-by-side plots with patchwork
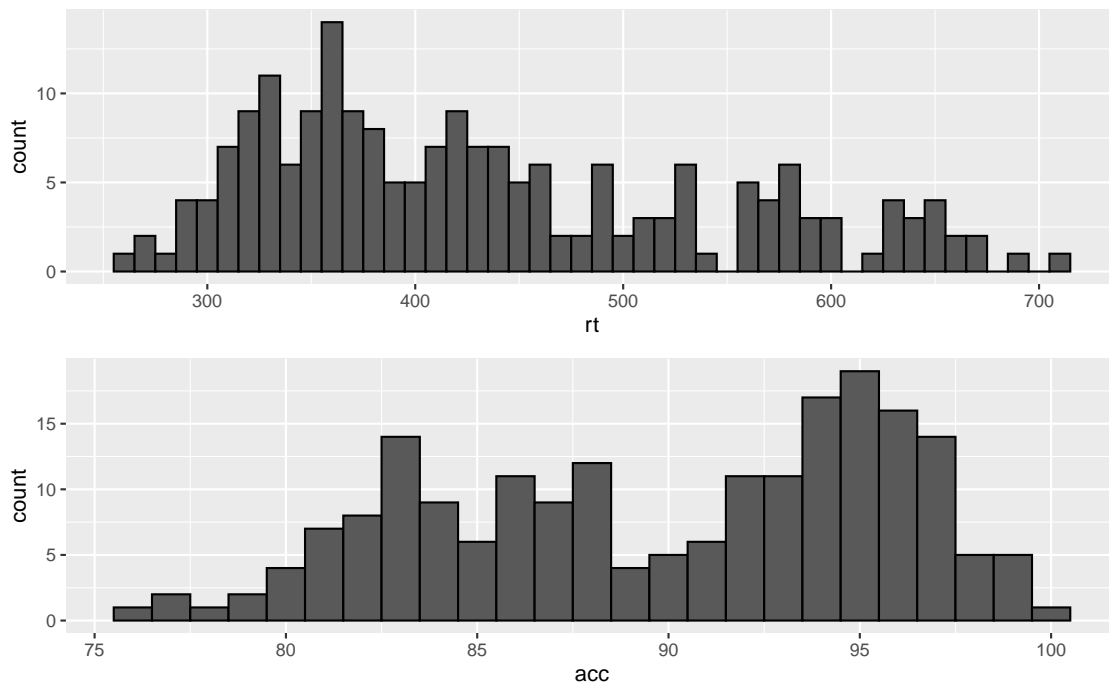
```
p1 / p2 # stacked
```

Figure 38. Stacked plots with patchwork

715   **Combining three or more plots.**   Three or more plots can be combined in a
716 number of ways and the `patchwork` syntax is relatively easy to grasp with a few examples
717 and a bit of trial and error. First, we save the complex interaction plot and faceted violin-
718 boxplot to objects named `p5` and `p6`.

```r
p5 <- ggplot(dat_long, aes(x = condition, y = rt,
                           group = language,
                           shape = language)) +
  geom_point(aes(colour = language),
             alpha = .2) +
  geom_line(aes(group = id, colour = language),
            alpha = .2) +
  stat_summary(fun = "mean",
               geom = "point",
               size = 2,
               colour = "black") +
  stat_summary(fun = "mean",
               geom = "line",
               colour = "black") +
  stat_summary(fun.data = "mean_se",
               geom = "errorbar",
               width = .2,
               colour = "black") +
```

```
  theme_minimal()

p6 <- ggplot(dat_long, aes(x = condition, y= rt)) +
  geom_violin() +
  geom_boxplot(width = .2, fatten = NULL) +
  stat_summary(fun = "mean", geom = "point") +
  stat_summary(fun.data = "mean_se", geom = "errorbar", width = .1) +
  facet_wrap(~factor(language,
                     levels = c("monolingual", "bilingual"),
                     labels = c("Monolingual participants",
                                "Bilingual participants"))) +
  theme_minimal()
```

719   The exact layout of your plots will depend upon a number of factors. Try running
720 the below examples and adjust the use of the operators to see how they change the layout.
721 Each line of code will draw a different figure.

```
p1 /p5 / p6
(p1 + p6) / p5
p6 | p1 / p5
```

722 **Customisation part 4**

723   **Axis labels.**   Previously when we edited the main axis labels we used the `scale_`
724 functions to do so. These functions are useful to know because they allow you to customise
725 each aspect of the scale, for example, the breaks and limits. However, if you only need to
726 change the main axis `name`, there is a quicker way to do so using `labs()`. The below code
727 adds a layer to the plot that changes the axis labels for the histogram saved in `p1` and adds
728 a title and subtitle. The title and subtitle do not conform to APA standards (more on APA
729 formatting in the additional resources), however, for presentations and social media they
730 can be useful.

```
p5 + labs(x = "Type of word",
          y = "Reaction time (ms)",
          title = "Language group by word type interaction plot",
          subtitle = "Reaction time data")
```
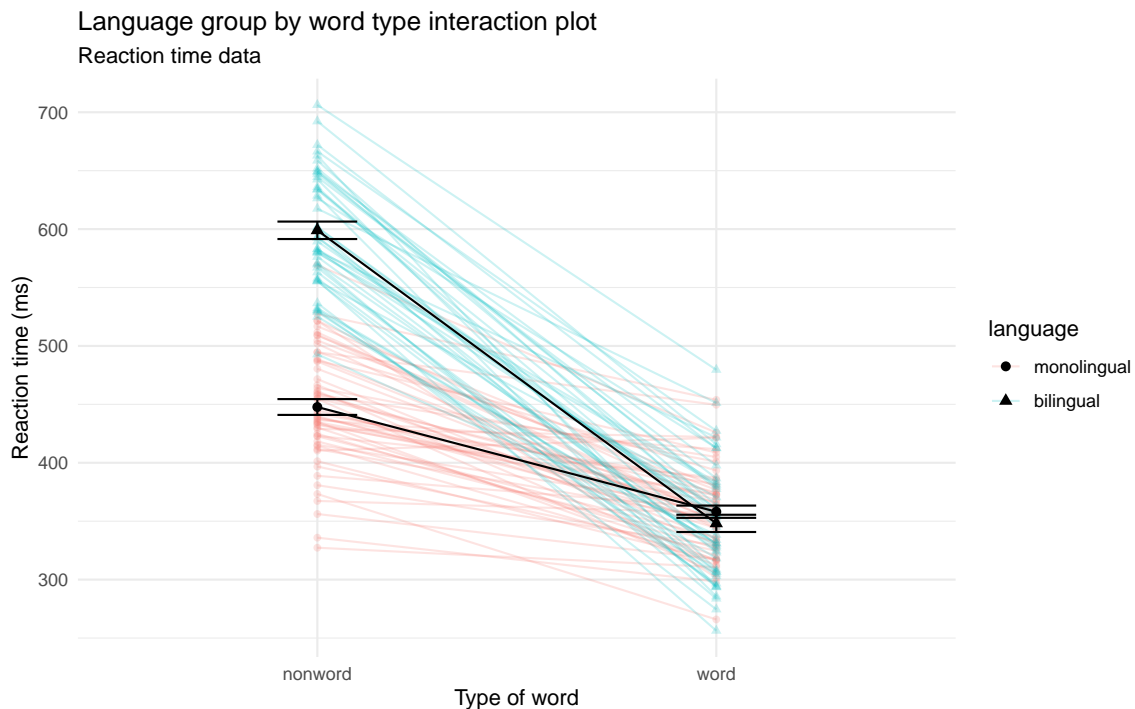
Language group by word type interaction plot
Reaction time data

Figure 39. Plot with edited labels and title

⁷³¹   You can also use `labs()` to remove axis labels, for example, try adjusting the above
⁷³² code to `x = NULL`.

⁷³³   **Redundant aesthetics.** So far when we have produced plots with colours, the
⁷³⁴ colours were the only way that different levels of a variable were indicated, but it is some-
⁷³⁵ times preferable to indicate levels with both colour and other means, such as facets or x-axis
⁷³⁶ categories.

⁷³⁷   The code below adds `fill = language` to the violin-boxplots that are also faceted
⁷³⁸ by language. We adjust `alpha` and use the viridis colour palette to customise the colours.

```
ggplot(dat_long, aes(x = condition, y= rt, fill = language)) +
  geom_violin(alpha = .4) +
  geom_boxplot(width = .2, fatten = NULL, alpha = .6) +
  stat_summary(fun = "mean", geom = "point") +
  stat_summary(fun.data = "mean_se", geom = "errorbar", width = .1) +
  facet_wrap(~factor(language,
                     levels = c("monolingual", "bilingual"),
                     labels = c("Monolingual participants",
                                "Bilingual participants"))) +
  theme_minimal() +
  scale_fill_viridis_d(option = "E")
```
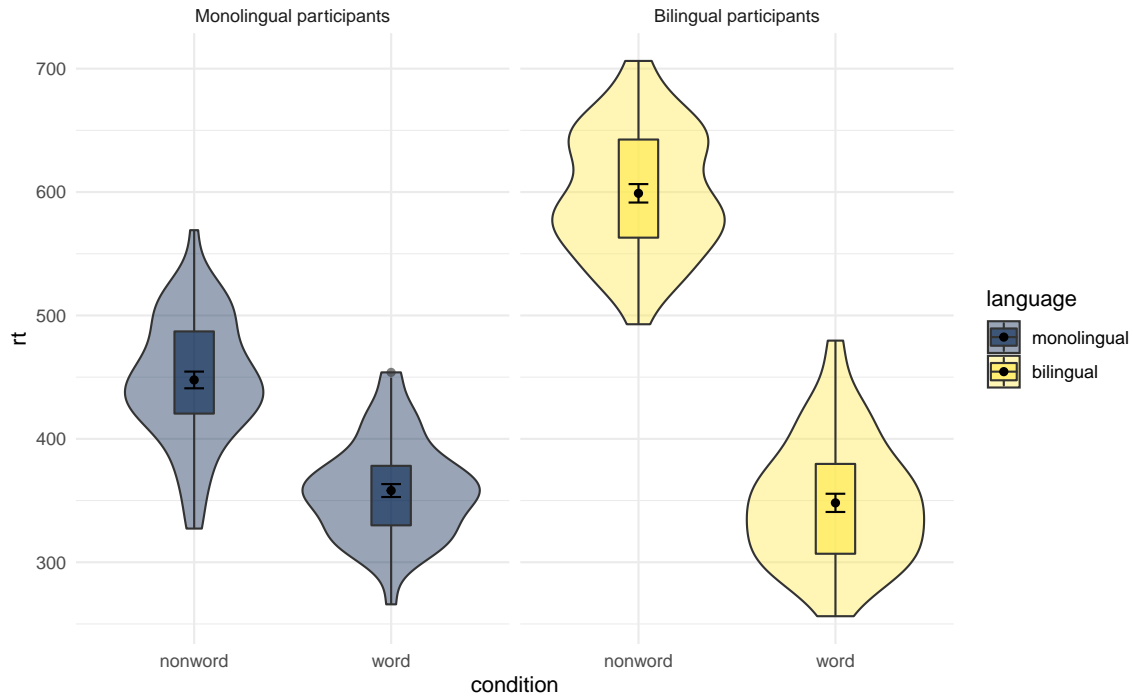
Figure 40. Violin-boxplot with redundant legend

739   Specifying a `fill` variable means that by default, R produces a legend for that vari-
740 able. However, the use of colour is redundant with the facet labels, so you can remove this
741 legend with the `guides` function.

```
ggplot(dat_long, aes(x = condition, y= rt, fill = language)) +
  geom_violin(alpha = .4) +
  geom_boxplot(width = .2, fatten = NULL, alpha = .6) +
  stat_summary(fun = "mean", geom = "point") +
  stat_summary(fun.data = "mean_se", geom = "errorbar", width = .1) +
  facet_wrap(~factor(language,
                     levels = c("monolingual", "bilingual"),
                     labels = c("Monolingual participants",
                                "Bilingual participants"))) +
  theme_minimal() +
  scale_fill_viridis_d(option = "E") +
  guides(fill = FALSE)
```
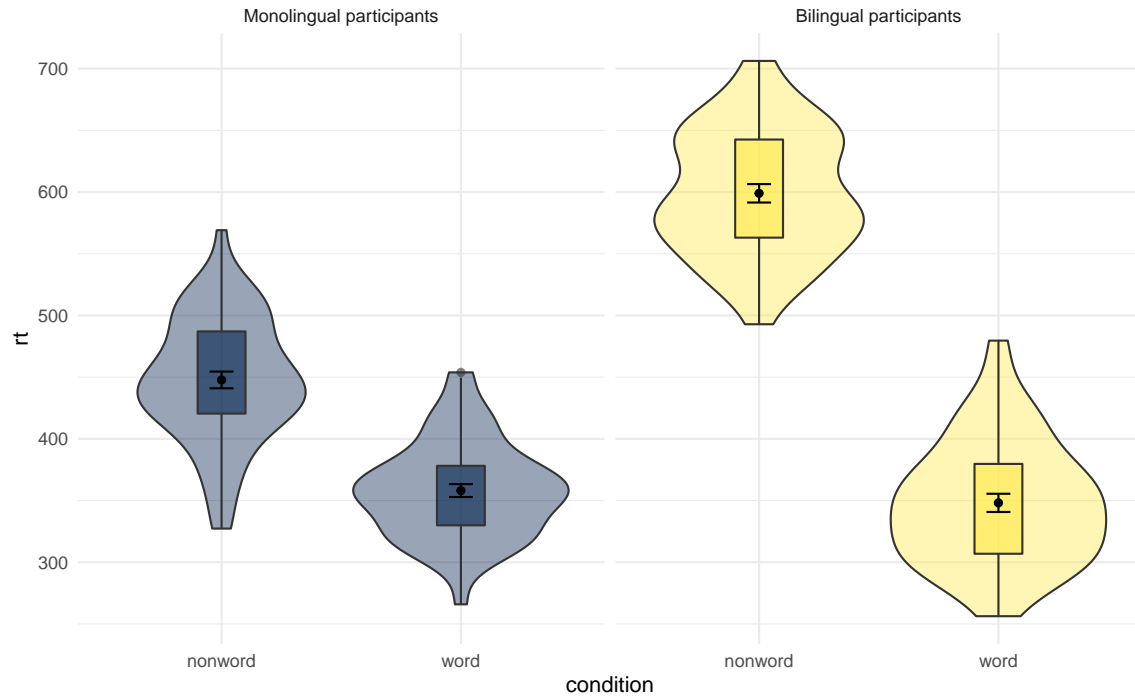
Figure 41. Plot with suppressed redundant legend

742 **Activities 4**

743   Before you go on, do the following:

744   1. Rather than mapping both variables (`condition` and `language`) to a single interaction
745   plot with individual participant data, instead produce a faceted plot that separates the
746   monolingual and bilingual data. All visual elements should remain the same (colours
747   and shapes) and you should also take care not to have any redundant legends.

748   2. Choose your favourite three plots you've produced so far in this tutorial, tidy them
749   up with axis labels, your preferred colour scheme, and any necessary titles, and then
750   combine them using `patchwork`. If you're feeling particularly proud of them, post
751   them on Twitter using #PsyTeachR.

752   **Advanced Plots**

753   This tutorial has but scratched the surface of the visualisation options available using
754   R - in the additional online resources we provide some further advanced plots and cus-
755   tomisation options for those readers who are feeling confident with the content covered in
756   this tutorial, however, the below plots give an idea of what is possible, and represent the
757   favourite plots of the authorship team.

758   We will use some custom functions: `geom_split_violin()` and
759 `geom_flat_violin()`, which you can access through the `introdataviz` package.
760 These functions are modified from (Allen et al., 2021).

```
# how to install the introdataviz package to get split and half violin plots
devtools::install_github("psyteachr/introdataviz")
```

Split-violin plots

762   Split-violin plots remove the redundancy of mirrored violin plots and make it easier
763 to compare the distributions between multiple conditions.
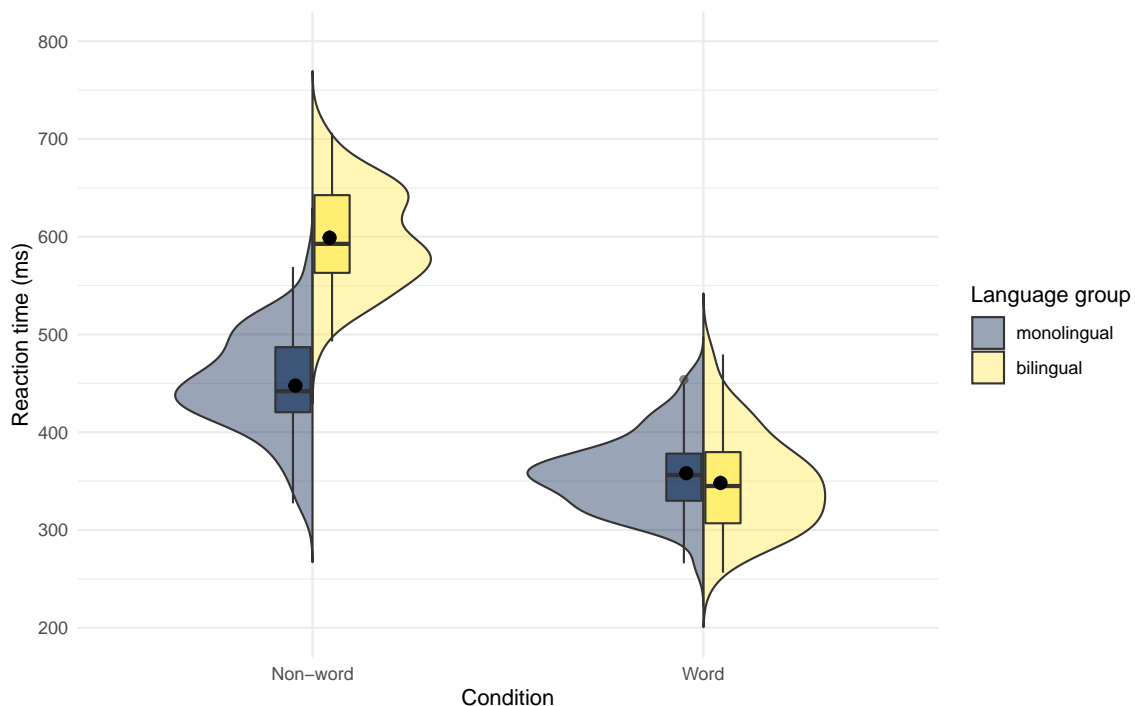
Figure 42. Split-violin plot

Raincloud plots

765   Raincloud plots combine a density plot, boxplot, raw data points, and any desired
766 summary statistics for a complete visualisation of the data. They are so called because the
767 density plot plus raw data is reminiscent of a rain cloud.
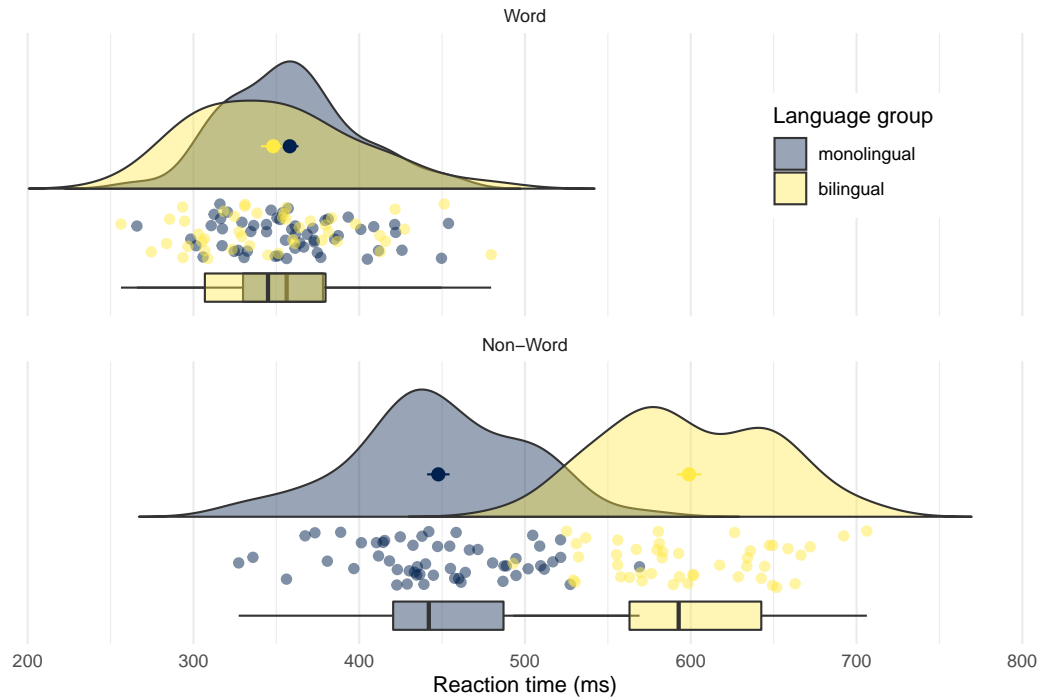
Figure 43. Raincloud plot

Ridge plots

Ridge plots are a series of density plots and show the distribution of numeric values for several groups. Figure 44 shows data from (Nation, 2017) and demonstrates how effective this type of visualisation can be to convey a lot of information very intuitively whilst being visually attractive.
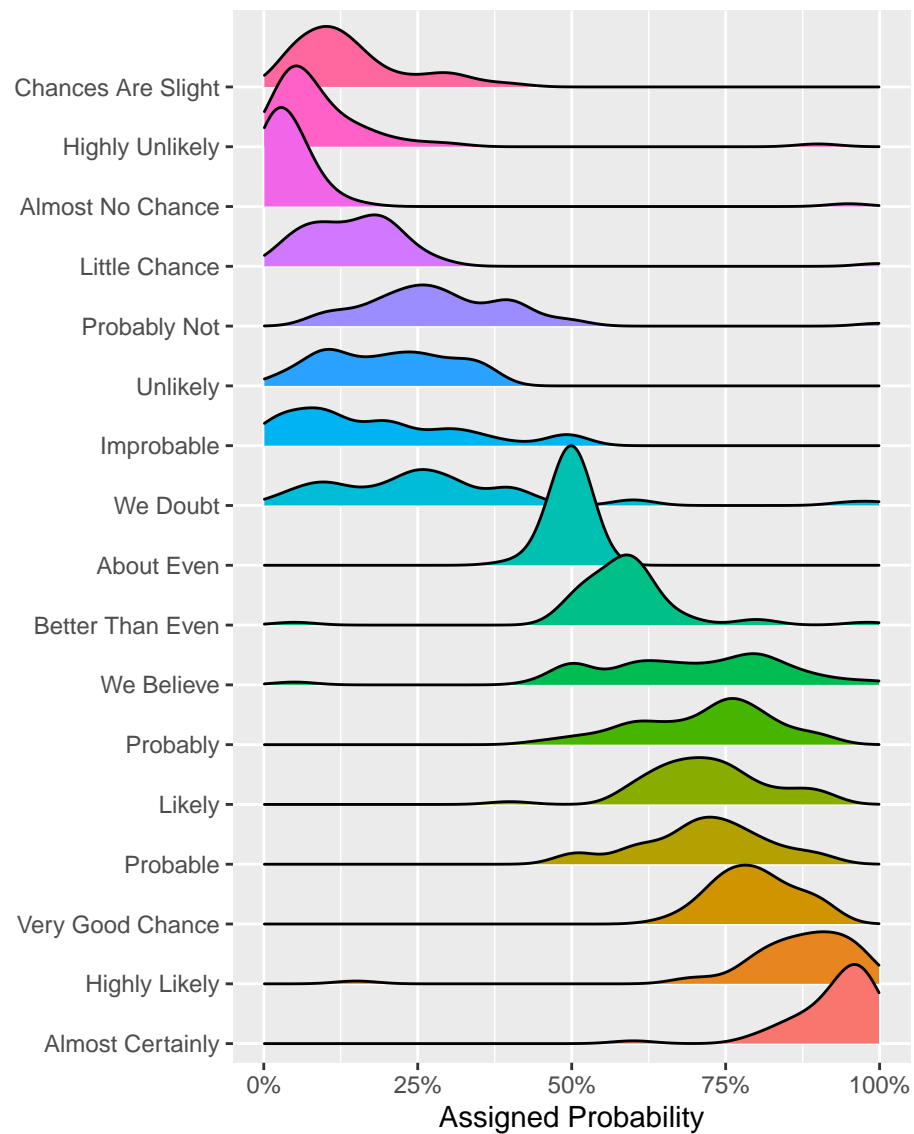
Figure 44. A ridge plot.

Alluvial plots

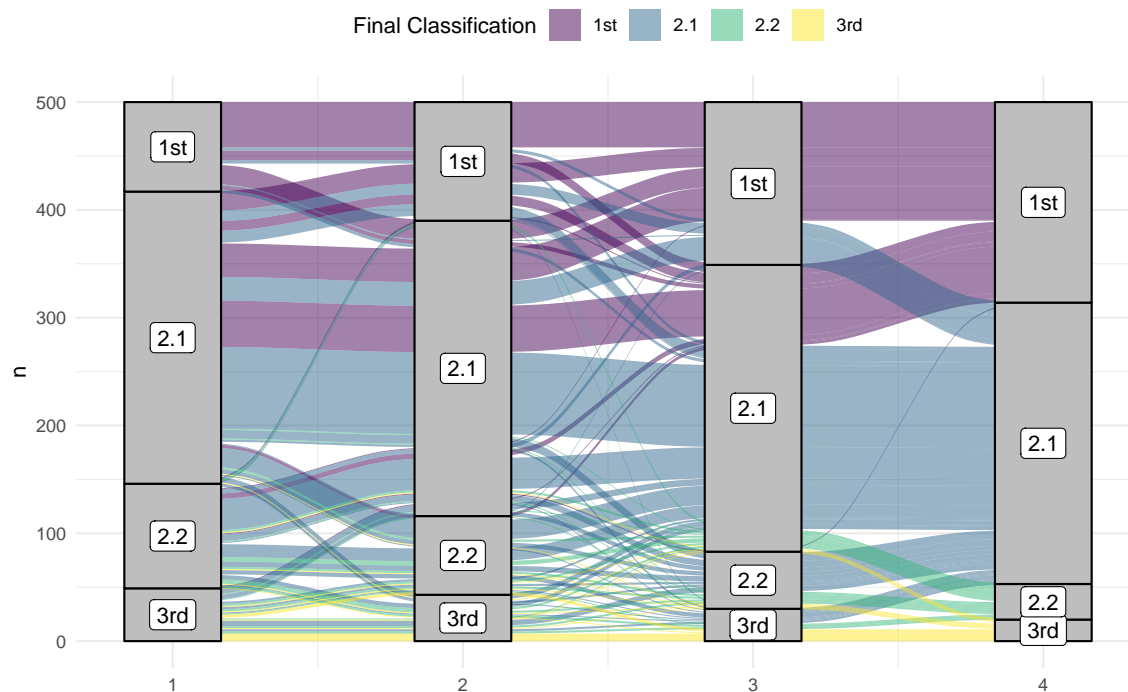Alluvial plots visualise multi-level categorical data through flows that can easily be traced in the diagram.

Figure 45. An alluvial plot showing the progression of student grades through the years.

Conclusion

In this tutorial we aimed to provide a practical introduction to common data visualisation techniques using R. Whilst a number of the plots produced in this tutorial can be created in point-and-click software, the underlying skill-set developed by making these visualisations is as powerful as it is extendable.

We hope that this tutorial serves as a jumping off point to encourage more researchers to adopt reproducible workflows and open-access software, in addition to beautiful data visualisations.

Acknowledgements

References

Allen, M., Poggiali, D., Whitaker, K., Marshall, T. R., van Langen, J., & Kievit, R. A. (2021). Raincloud plots: A multi-platform tool for robust data visualization [version 2; peer review: 2 approved]. *Wellcome Open Research*, *4*. https://doi.org/10.12688/wellcomeopenres.15191.2

Aust, F., & Barth, M. (2020). *papaja: Create APA manuscripts with R Markdown*. Retrieved from https://github.com/crsh/papaja

Barrett, T. S. (2019). Six reasons to consider using r in psychological research.

Bertini, E., & Stefaner, M. (2015). Amanda cox on working with r, NYT projects, favorite data [podcast]. *Data Stories*. Retrieved from https://datastori.es/ds-56-amanda-cox-nyt/

Brunson, J. C. (2020). ggalluvial: Layered grammar for alluvial plots. *Journal of Open Source Software*, *5*(49), 2017. https://doi.org/10.21105/joss.02017

DeBruine, L. (2021). *Faux: Simulation for factorial designs*. Zenodo. https://doi.org/10.5281/zenodo.2669586

Munafò, M. R., Nosek, B. A., Bishop, D. V., Button, K. S., Chambers, C. D., Du Sert, N. P., . . . Ioannidis, J. P. (2017). A manifesto for reproducible science. *Nature Human Behaviour*, *1*(1), 1–9.

Nation, Z. (2017). Perceptions. *GitHub repository*. https://github.com/zonination/perceptions%20%20; GitHub.

Newman, G. E., & Scholl, B. J. (2012). Bar graphs depicting averages are perceptually misinterpreted: The within-the-bar bias. *Psychonomic Bulletin & Review*, *19*(4), 601–607.

Pedersen, T. L. (2020). *Patchwork: The composer of plots*. Retrieved from https://CRAN.R-project.org/package=patchwork

R Core Team. (2021). *R: A language and environment for statistical computing*. Vienna, Austria: R Foundation for Statistical Computing. Retrieved from https://www.R-project.org/

Robins, A., Rountree, J., & Rountree, N. (2003). Learning and teaching programming: A review and discussion. *Computer Science Education*, *13*(2), 137–172.

RStudio Team. (2021). *RStudio: Integrated development environment for r*. Boston, MA: RStudio, PBC. Retrieved from http://www.rstudio.com/

Visual, B., & Journalism, D. (2019). How the BBC visual and data journalism team works with graphics in r. *Medium*. Retrieved from https://medium.com/bbc-visual-and-data-journalism/how-the-bbc-visual-and-data-journalism-team-works-with-graphics-in-r-ed0b35693535

Wickham, H. (2010). A layered grammar of graphics. *Journal of Computational and Graphical Statistics*, *19*(1), 3–28.

Wickham, H. (2016a). *ggplot2: Elegant graphics for data analysis.* Springer-Verlag New York. Retrieved from https://ggplot2.tidyverse.org

Wickham, H. (2016b). *ggplot2: Elegant graphics for data analysis.* Springer-Verlag New York. Retrieved from https://ggplot2.tidyverse.org

Wickham, H. (2017). *Tidyverse: Easily install and load the 'tidyverse'.* Retrieved from https://CRAN.R-project.org/package=tidyverse

Wickham, H., Averick, M., Bryan, J., Chang, W., McGowan, L. D., François, R., . . . Yutani, H. (2019). Welcome to the tidyverse. *Journal of Open Source Software*, *4*(43), 1686. https://doi.org/10.21105/joss.01686

Wickham, H., & others. (2014). Tidy data. *Journal of Statistical Software*, *59*(10), 1–23.

Wilke, C. O. (2021). *Ggridges: Ridgeline plots in 'ggplot2'.* Retrieved from https://CRAN.R-project.org/package=ggridges

Wilkinson, L., Anand, A., & Grossman, R. (2005). Graph-theoretic scagnostics. In *IEEE symposium on information visualization (InfoVis 05)* (pp. 157–158). IEEE Computer Society.

Wills, A. (n.d.). Teaching research methods in r. *rminr*. Retrieved from https://www.andywills.info/rminr/rminrinpsy.html