

<sup>1</sup> Data visualisation using R, for researchers who don't use R

<sup>2</sup> Emily Nordmann<sup>1</sup>, Phil McAleer<sup>1</sup>, Wilhelmiina Toivo<sup>1</sup>, Helena  
<sup>3</sup> Paterson<sup>1</sup>, & Lisa M. DeBruine<sup>2</sup>

<sup>4</sup> <sup>1</sup> School of Psychology, University of Glasgow

<sup>5</sup> <sup>2</sup> Institute of Neuroscience and Psychology, University of Glasgow

<sup>6</sup> Preprint

<sup>7</sup> Abstract

This is a pre-submission manuscript and tutorial and has not yet undergone peer-review. We welcome user feedback which you can provide using this form: <https://forms.office.com/r/ba1UvyykYR>. Please note that this tutorial is likely to undergo changes before it is accepted for publication and we would encourage you to check for updates before citing. All materials associated with this article (including a version that is more accessible than the PDF version) can be accessed at <https://psyteachr.github.io/introdataviz>.

*Keywords:* visualization, ggplot, plots, R

Word count: 11472

<sup>9</sup> **Introduction**

<sup>10</sup> Use of the programming language R (R Core Team, 2021) for data processing and  
<sup>11</sup> statistical analysis by researchers is increasingly common, with an average yearly growth of  
<sup>12</sup> 87% in the number of citations of the R Core Team between 2006-2018 (Barrett, 2019). In  
<sup>13</sup> addition to benefiting reproducibility and transparency, one of the advantages of using R is  
<sup>14</sup> that researchers have a much larger range of fully customisable data visualisations options  
<sup>15</sup> than are typically available in point-and-click software, due to the open-source nature of R.  
<sup>16</sup> These visualisation options not only look attractive, but can increase transparency about  
<sup>17</sup> the distribution of the underlying data rather than relying on commonly used visualisations

---

Emily Nordmann  <https://orcid.org/0000-0002-0806-1081> Phil McAleer  <https://orcid.org/0000-0002-4523-2097> Wilhelmiina Toivo  <https://orcid.org/0000-0002-5688-9537> Helena Paterson  <https://orcid.org/0000-0001-7715-5973> Lisa DeBruine  <https://orcid.org/0000-0002-7523-5539>

Correspondence concerning this article should be addressed to Emily Nordmann, 62 Hillhead Street, Glasgow, G12 8QB. E-mail: [emily.nordmann@glasgow.ac.uk](mailto:emily.nordmann@glasgow.ac.uk)

18 of aggregations such as bar charts of means (Newman & Scholl, 2012).

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

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

### 37 Why R for data visualisation?

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

46 In addition to the benefits of reproducibility, using R for data visualisation gives the  
47 researcher almost total control over each element of the plot. Whilst this flexibility can  
48 seem daunting at first, the ability to write reusable code recipes (and use recipes created  
49 by others) is highly advantageous. The level of customisation and the professional outputs  
50 available using R has, for instance, lead news outlets such as the BBC (Visual & Journalism,  
51 2019) and the New York Times (Bertini & Stefaner, 2015) to adopt R as their preferred  
52 data visualisation tool.

53 **A layered grammar of graphics**

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

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

```
a <- ggplot() + labs(subtitle = "Layer 1")
b <- ggplot(iris, aes(Sepal.Length, Sepal.Width)) + labs(subtitle = "Layer 2")
c <- b + geom_point() + labs(subtitle = "Layer 3")
d <- c + geom_smooth(method = "lm") + labs(subtitle = "Layer 4")
e <- d + labs(x = "Sepal Length (mm)", y = "Sepal Width (mm)") + labs(subtitle = "Layer 5")
f <- e + theme_minimal(base_family = "Times") + labs(subtitle = "Layer 6")

a + b + c + d + e + f + plot_layout(nrow = 2)
```

---

<sup>1</sup>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."

<sup>2</sup>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`."

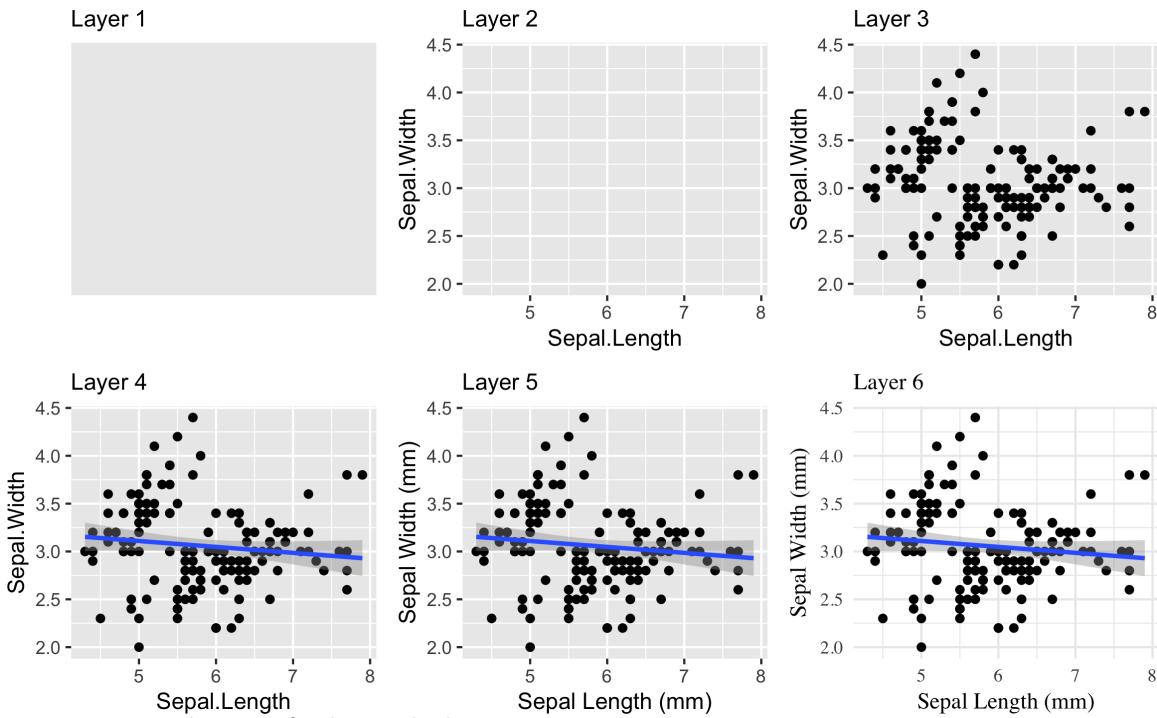


Figure 1. Evolution of a layered plot

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

```
iris %>% ggplot(aes(Sepal.Length, Sepal.Width)) +
  geom_smooth(method = "lm") +
  labs(x = "Sepal Length (mm)", y = "Sepal Width (mm)")+
  theme_minimal()
```

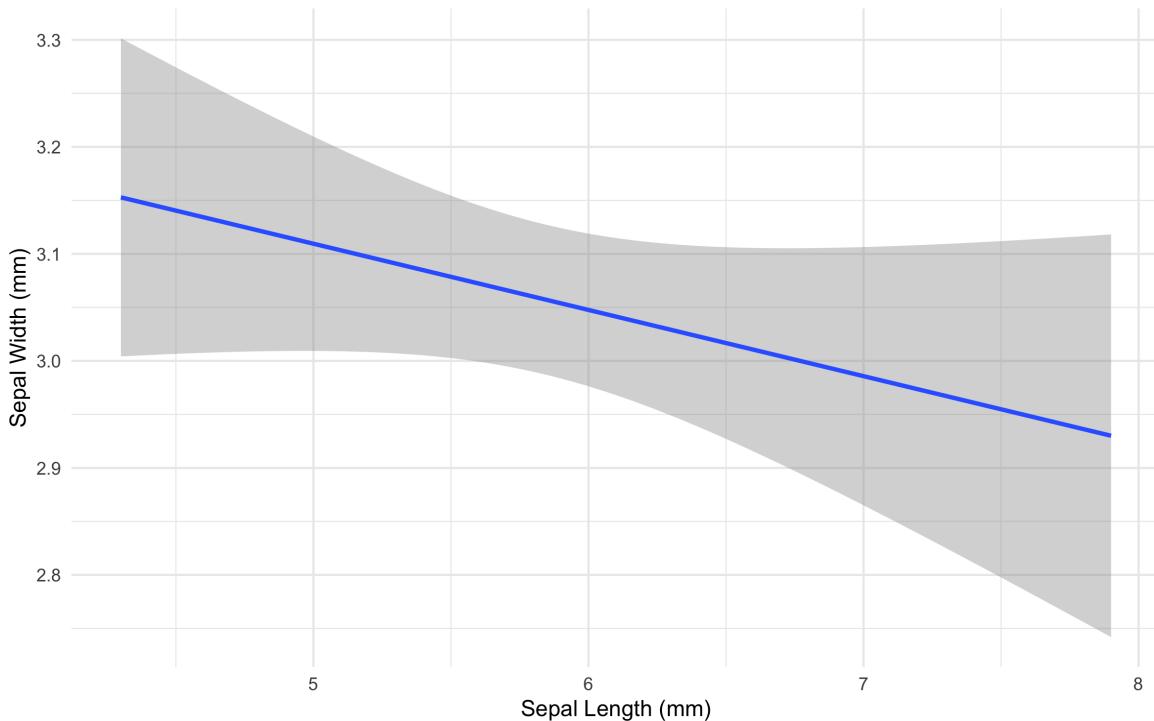


Figure 2. Plot with scatterplot layer removed.

#### 74 Simulated dataset

75 For the purpose of this tutorial, we will use simulated data for a 2 x 2 mixed-design  
 76 lexical decision task in which participants have to decide whether a presented word is a real  
 77 word, or a non-word, with 100 participants. There are 100 rows (1 for each participant) and  
 78 7 variables:

- 79 • Participant information:
  - 80 – **id**: Participant ID
  - 81 – **age**: Age
- 82 • 1 between-subject IV:
  - 83 – **language**: Language group (1 = monolingual, 2 = bilingual)
- 84 • 4 columns for the 2 dependent variables for RT and accuracy, crossed by the within-  
 85 subject IV of condition:
  - 86 – **rt\_word**: Reaction time (ms) for word trials
  - 87 – **rt\_nonword**: Reaction time (ms) for non-word trials
  - 88 – **acc\_word**: Accuracy for word trials
  - 89 – **acc\_nonword**: Accuracy for non-word trials

90 The simulated dataset and tutorial code can be found in the online supplementary  
 91 materials. For newcomers to R, we would suggest working through this tutorial with the

92 simulated dataset, then extending the code to your own datasets with a similar structure,  
93 and finally generalising the code to new structures and problems.

94 **Setting up R and RStudio**

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

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

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

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

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

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

120 **Preparing your data**

121 Before you start visualising your data, you need to get it into an appropriate format.  
122 These preparatory steps can all be dealt with reproducibly using R and the additional  
123 resources section points to extra tutorials for doing so. However, performing these types  
124 of tasks in R can require more sophisticated coding skills and the solutions and tools are  
125 dependent on the idiosyncrasies of each dataset. For this reason, in this tutorial we encourage

126 the reader to complete data preparation steps using the method they are most comfortable  
127 with and to focus on the aim of data visualisation.

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

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

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

151 When working with your own data, you can rename columns in Excel, but the resources  
152 listed in the additional resources point to how to rename columns reproducibly with code.

153 **Data values.** A great benefit to using R is that categorical data can be entered as  
154 text. In the tutorial dataset, language group is entered as 1 or 2, so that we can show you  
155 how to recode numeric values into factors with labels. However, we recommend recording  
156 meaningful labels rather than numbers from the beginning of data collection to avoid  
157 misinterpreting data due to coding errors. Note that values must match *exactly* in order  
158 to be considered in the same category and R is case sensitive, so “mono,” “Mono,” and  
159 “monolingual” would be classified as members of three separate categories.

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

---

<sup>3</sup>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.

163

## Getting Started

164 **Loading packages**

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

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

171 **Loading data**

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

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

176 This code has created an object `dat` into which you have read the data from the file  
177 `ldt_data.csv`. This object will appear in the environment pane in the top right. Note  
178 that the name of the data file must be in quotation marks and the file extension (`.csv`)  
179 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  
181 sensitive), have forgotten to include the file extension `.csv`, or that the data file you want  
182 to load is not stored in your project folder. If you get the error `could not find function`  
183 it means you have either not loaded the correct package (a common beginner error is to  
184 write the code, but not run it), or you have made a typo in the function name.

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

192 **Handling numeric factors**

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

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

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

- `mutate()` makes new columns in a data table, or overwrites a column;
  - `factor()` translates the language column into a factor with the labels “monolingual” and “bilingual.” You can also use `factor()` to set the display order of a column that contains words. Otherwise, they will display in alphabetical order. In this case we are replacing the numeric data (1 and 2) in the `language` column with the equivalent English labels `monolingual` for 1 and `bilingual` for 2. At the same time we will 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
  ))
```

206 Make sure that you always check the output of any code that you run. If after running  
207 this code `language` is full of NA values, it means that you have run the code twice. The  
208 first time would have worked and transformed the values from 1 to `monolingual` and 2 to  
209 `bilingual`. If you run the code again on the same dataset, it will look for the values 1 and  
210 2, and because there are no longer any that match, it will return NA. If this happens, you  
211 will need to reload the dataset from the csv file.

212 A good way to avoid this is never to overwrite data, but to always store the output  
213 of code in new objects (e.g., `dat_recoded`) or new variables (`language_recoded`). For the  
214 purposes of this tutorial, overwriting provides a useful teachable moment so we'll leave it as  
215 it is.

216 Argument names

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

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

228 **Demographic information**

229 You can calculate and plot some basic descriptive information about the demograph-  
 230 ics of our sample using the imported dataset without any additional wrangling (or data  
 231 processing). The code below uses the `%>%` operator, otherwise known as the *pipe*, and can  
 232 be translated as “*and then*”. For example, the below code can be read as:

- 233 • Start with the dataset `dat` *and then*;  
 234 • Group it by the variable `language` *and then*;  
 235 • Count the number of observations in each group

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

236

language	n
monolingual	55
bilingual	45

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

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

243

n
100

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

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

246

mean_age	sd_age	n_values
29.75	8.28	100

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

252     Note that the above code will not save the result of this operation, it will simply  
 253   output the result in the console. If you wish to save it for future use, you can store it in an  
 254   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())
```

255     Finally, the `group_by()` function will work in the same way when calculating summary  
 256   statistics - the output of the function that is called after `group_by()` will be produced for  
 257   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
258	monolingual	27.96	6.78	55
	bilingual	31.93	9.44	45

## 259 Bar chart of counts

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

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

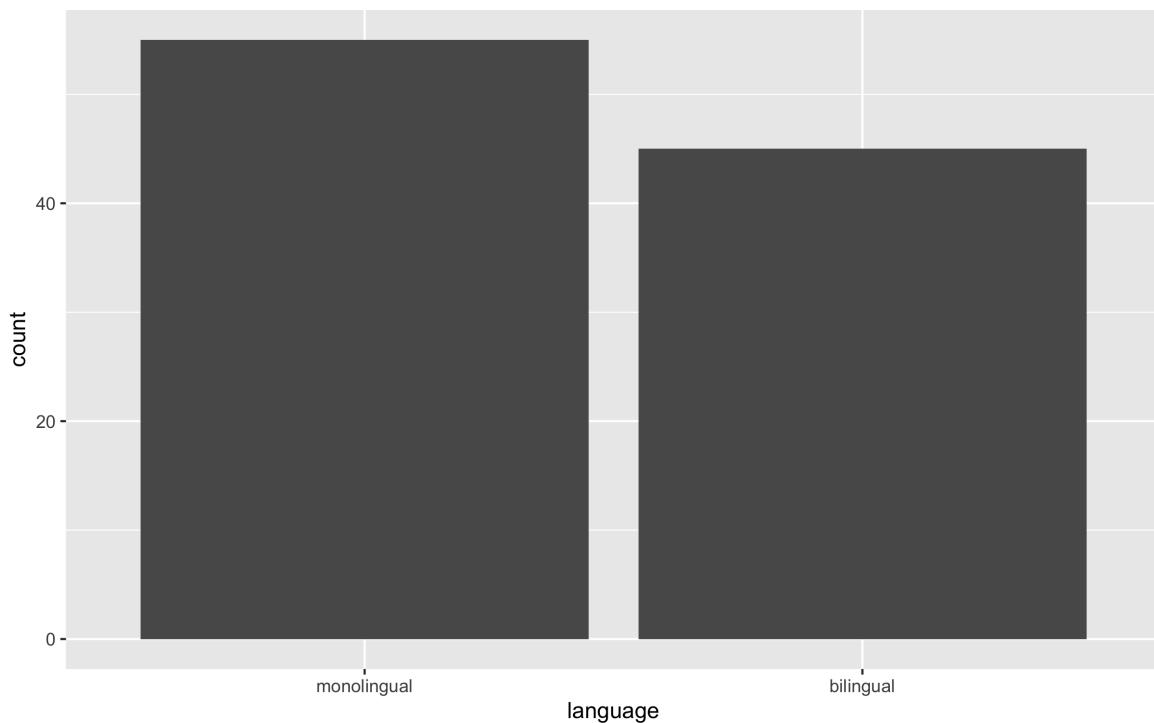


Figure 3. Bar chart of counts.

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

- 263        • `data` specifies which data source to use for the plot
- 264        • `mapping` specifies which variables to map to which aesthetics (`aes`) of the plot. Aesthetic
- 265        mappings describe how variables in the data are mapped to visual properties (aesthetics)
- 266        of geoms.
- 267        • `x` specifies which variable to put on the x-axis

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

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

## 276 Plotting existing aggregates and percent

277        If your data already have the counts that you want to plot, you can set  
 278 `stat="identity"` inside of `geom_bar()` to use that number instead of counting rows. For

279 example, there is currently no function to plot percentages rather than counts within `ggplot`,  
 280 you need to calculate these and store them in an object which is then used as the dataset.

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

```
dat_percent <- dat %>%
  group_by(language) %>%
  count() %>%
  ungroup() %>%
  mutate(percent = (n/sum(n)*100))

ggplot(dat_percent, aes(x = language, y = percent)) +
  geom_bar(stat="identity")
```

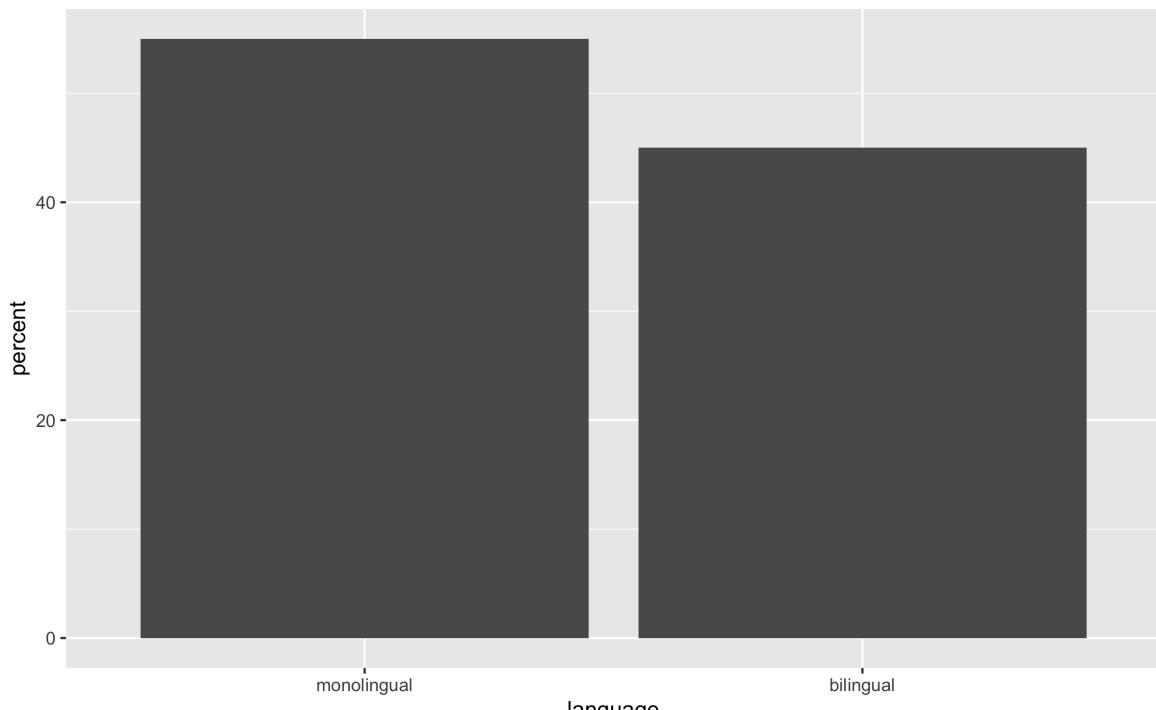


Figure 4. Bar chart of pre-calculated counts.

### 283 Histogram

284 The code to plot a histogram of `age` is very similar to the code used for the bar chart.  
 285 We start by setting up the plot space, the dataset we want to use, and mapping the variables  
 286 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()
```

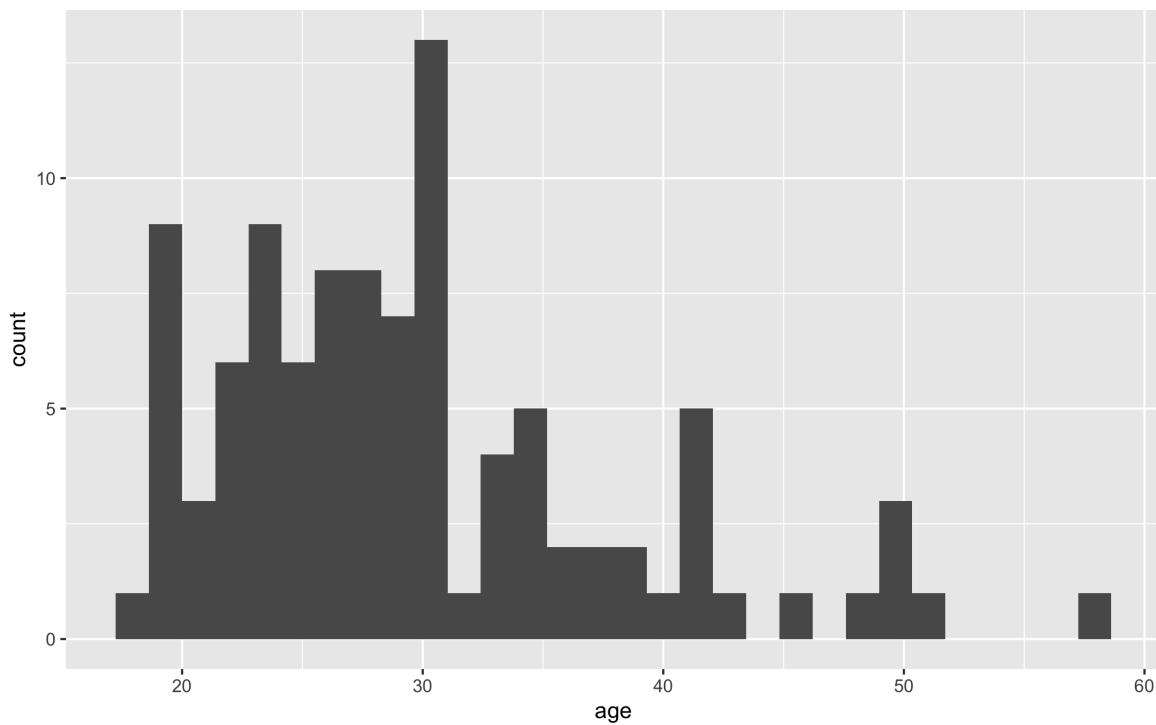


Figure 5. Histogram of ages.

287        The base statistic for `geom_histogram()` is also `count`, and by default  
 288        `geom_histogram()` divides the x-axis into “bins” and counts how many observations are in  
 289        each bin and so the y-axis does not need to be specified. When you run the code to pro-  
 290        duce the histogram, you will get the message `stat_bin() using bins = 30. Pick better`  
 291        `value with binwidth`. This means that the default number of bins `geom_histogram()`  
 292        divided the x-axis into is 30. For our data that looks appropriate, but for example, if you  
 293        want one bar to equal 5 years, you can adjust `binwidth = 5`.

```
ggplot(dat, aes(x = age)) +
  geom_histogram(binwidth = 5)
```

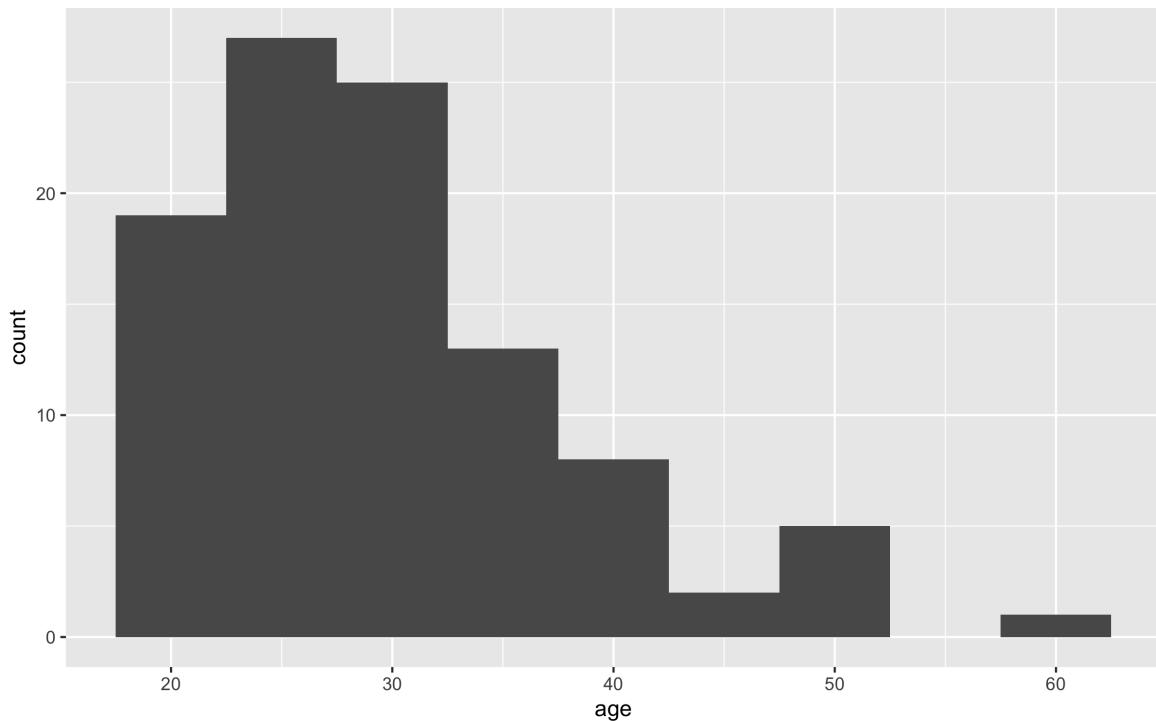


Figure 6. Histogram of ages where each bin covers five years.

294 **Customisation 1**

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

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

```
ggplot(dat, aes(age)) +
  geom_histogram(binwidth = 1,
                 fill = "white",
                 colour = "black")
```

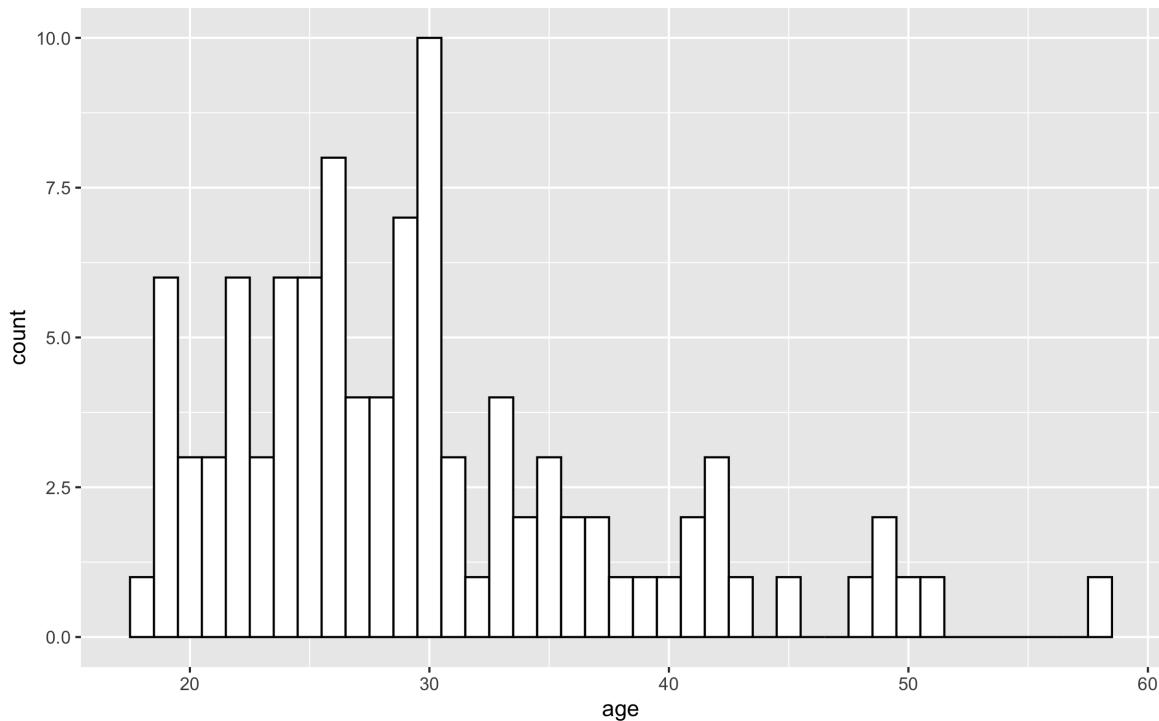


Figure 7. Histogram with custom colors for bar fill and line colors.

307     **Editing axis names and labels.** To edit axis names and labels you can connect  
 308     scale\_\* functions to your plot with + to add layers. These functions are part of ggplot  
 309     and the one you use depends on which aesthetic you wish to edit (e.g., x-axis, y-axis, fill,  
 310     colour) as well as the type of data it represents (discrete, continuous).

311     For the bar chart of counts, the x-axis is mapped to a discrete (categorical) variable  
 312     whilst the y-axis is continuous. For each of these there is a relevant scale function with  
 313     various elements that can be customised. Each axis then has its own function added as a  
 314     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))
```

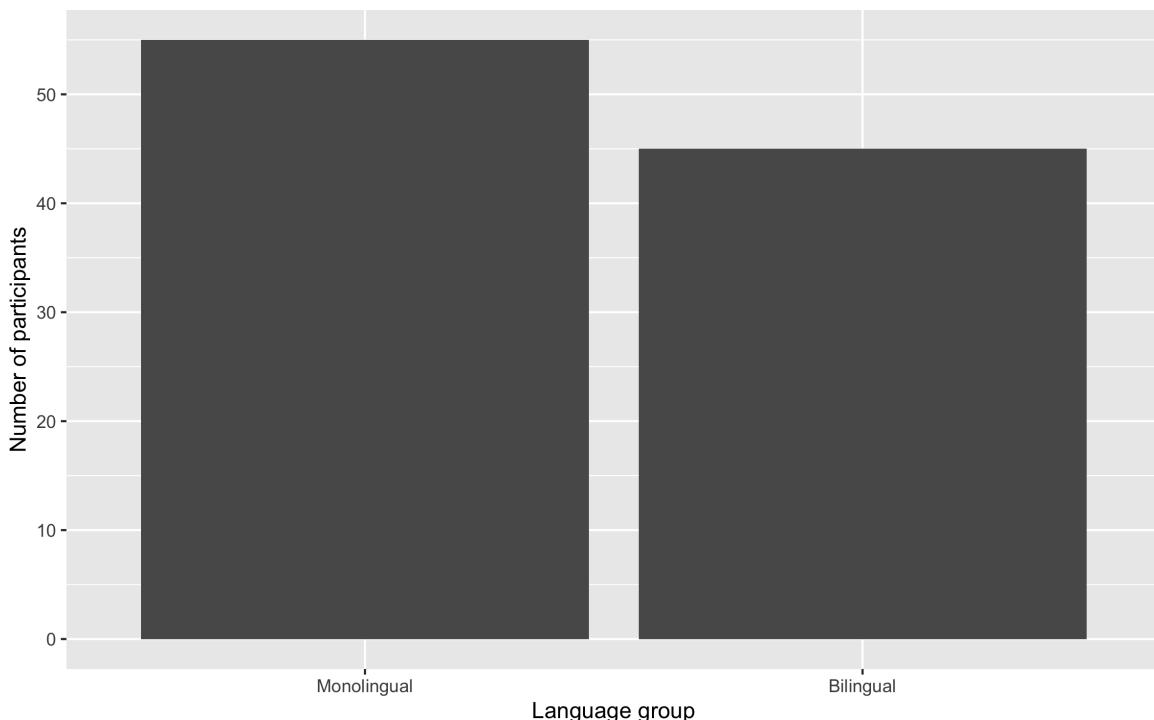


Figure 8. Bar chart with custom axis labels.

- 315 • **name** controls the overall name of the axis (note the use of quotation marks)
- 316 • **labels** controls the names of the conditions with a discrete variable.
- 317 • **c()** is a function that you will see in many different contexts and is used to combine  
318 multiple values. In this case, the labels we want to apply are combined within **c()** by  
319 enclosing each word within their own parenthesis, and are in the order displayed on  
320 the plot. A very common error is to forget to enclose multiple values in **c()**.
- 321 • **breaks** controls the tick marks on the axis. Again because there are multiple values,  
322 they are enclosed within **c()** although because they are numeric and not text, they do  
323 not need quotation marks.

324     **Discrete vs. continuous errors.** Another very common error is to map the wrong  
325 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"))
```

326     This will produce the error **Discrete value supplied to continuous scale** be-  
327 cause we have used a **continuous** scale function, despite the fact that x-axis variable is  
328 discrete. If you get this error (or the reverse), check the type of data on each axis and the  
329 function you have used.

330       **Adding a theme.** `ggplot` has a number of built-in visual themes that you can apply  
 331      as an extra layer. The below code updates the x-axis and y-axis labels to the histogram,  
 332      but also applies `theme_minimal()`. Each part of a theme can be independently customised,  
 333      which may be necessary, for example, if you have journal guidelines on fonts for publication.  
 334      There are further instructions for how to do this in the additional resources.

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

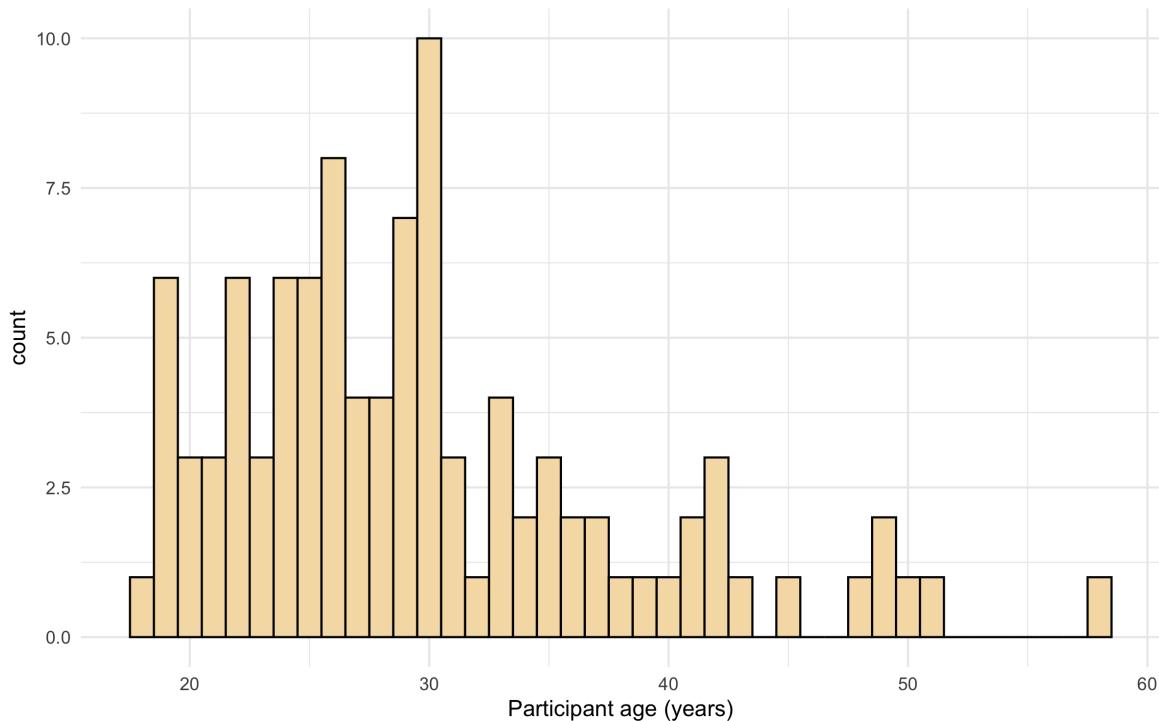


Figure 9. Histogram with a custom theme.

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

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

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

### 337   Activities 1

338       Before you move on try the following:

- 339       1. Add a layer that edits the `name` of the y-axis histogram label to `Number of participants`.
- 340
- 341       2. Change the colour of the bars in the bar chart to red.

Table 1  
*Data in wide format.*

<b>id</b>	<b>age</b>	<b>language</b>	<b>rt_word</b>	<b>rt_nonword</b>	<b>acc_word</b>	<b>acc_nonword</b>
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

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

## Transforming Data

347 Data formats

To visualise the experimental reaction time and accuracy data using `ggplot`, 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 between groups variables, 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<sup>4</sup> 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

<sup>4</sup>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 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

366 less intuitive part is that long data has multiple rows for each participant and a column  
 367 that encodes the level of the IV (**word** or **nonword**). In essence, the long-format encodes  
 368 repeated-measures variable in the same way as a between-group variable in SPSS. Wickham  
 369 (2014) provides a comprehensive overview of the benefits of a similar format known as tidy  
 370 data, which is a standard way of mapping a dataset to its structure, but for the purposes of  
 371 this tutorial there are two important rules: each column should be a *variable* and each row  
 372 should be an *observation*.

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

382 The benefits and flexibility of this format will hopefully become apparent as we  
 383 progress through the tutorial, however, a useful rule of thumb when working with data  
 384 in R for visualisation is that *anything that shares an axis should probably be in the same*  
 385 *column*. For example, a simple bar chart of means for the reaction time DV would display  
 386 the variable **condition** on the x-axis with bars representing both the **word** and **nonword**  
 387 data, therefore, these data should be in one column and not split.

### 388 Transforming data

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

---

<sup>5</sup>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.

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

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

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

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

- 403       • As with the other tidyverse functions, the first argument specifies the dataset to use  
 404       as the base, in this case `dat`. This argument name is often dropped in examples.
- 405       • `cols` specifies all the columns you want to transform. The easiest way to visualise  
 406       this is to think about which columns would be the same in the new long-form dataset  
 407       and which will change. If you refer back to Table 1, you can see that `id`, `age`, and  
 408       `language` all remain, while the columns that contain the measurements of the DVs  
 409       change. The colon notation `first_column:last_column` is used to select all variables  
 410       from the first column specified to the second. In our code, `cols` specifies that the  
 411       columns we want to transform are `rt_word` to `acc_nonword`.
- 412       • `names_to` specifies the name of the new column that will be created.
- 413       • Finally, `values_to` names the new column that will contain the measurements, in this  
 414       case we'll call it `dv`. At this point you may find it helpful to go back and compare `dat`  
 415       and `long` again to see how each argument matches up with the output of the table.

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

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")
```

- The first argument is again the dataset you wish to work from, in this case `long2`. We have removed the argument name `data` in this example.
- `names_from` acts somewhat like the reverse of `names_to` from `pivot_longer()`. It will take the values from the variable specified and use these as variable names, i.e., in this case, the values of `rt` and `acc` that are currently in the `dv_type` column, and turn these into the column names.

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

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

448     If you are working with a dataset with only one DV, note that only step 1 of this  
 449       process would be necessary. Also, be careful not to calculate demographic descriptive  
 450       statistics from this long-form dataset. Because the process of transformation has introduced  
 451       some repetition for these variables, the wide-form dataset where 1 row = 1 participant  
 452       should be used for demographic information. Finally, the three step process noted above is  
 453       broken down for teaching purposes, in reality, one would likely do this in a single pipeline of  
 454       code, 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")
```

## 455 Histogram 2

456     Now that we have the experimental data in the right form, we can begin to create some  
 457       useful visualizations. First, to demonstrate how code recipes can be reused and adapted,  
 458       we will create histograms of reaction time and accuracy. The below code uses the same  
 459       template as before but changes the dataset (`dat_long`), the bin-widths of the histograms,  
 460       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)")
```

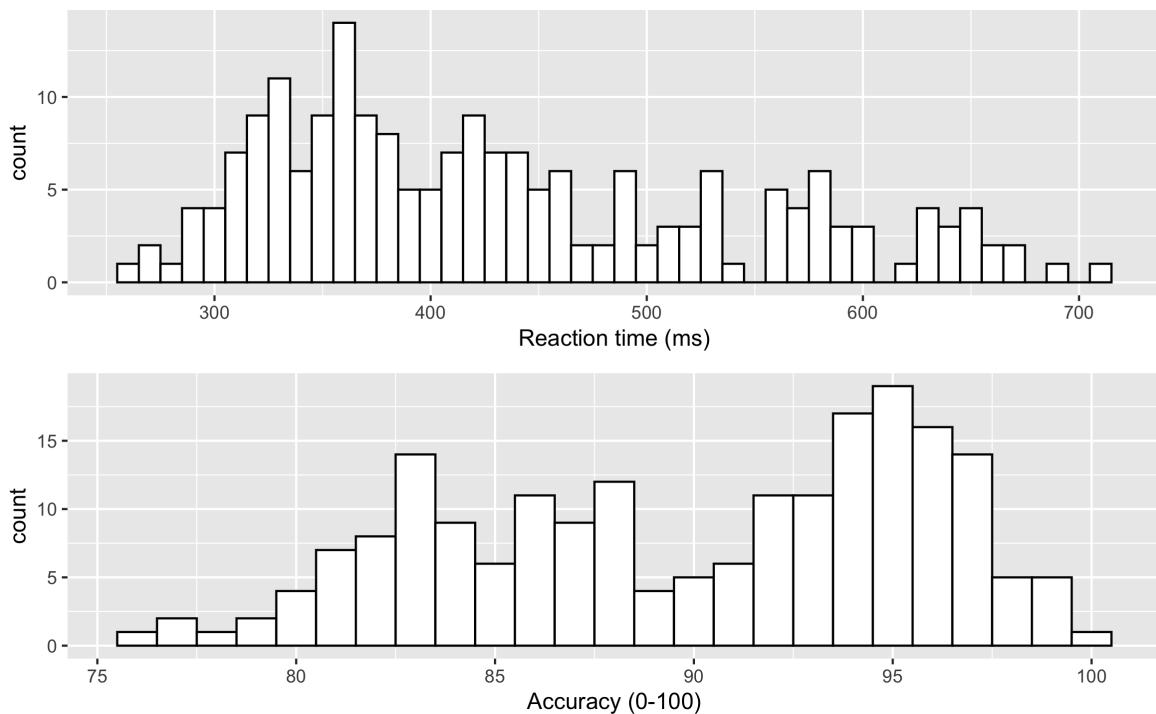


Figure 10. Histograms showing the distribution of reaction time (top) and accuracy (bottom)

#### 461 Density plots

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

```
462 ggplot(dat_long, aes(x = rt)) +  

  463   geom_density() +  

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

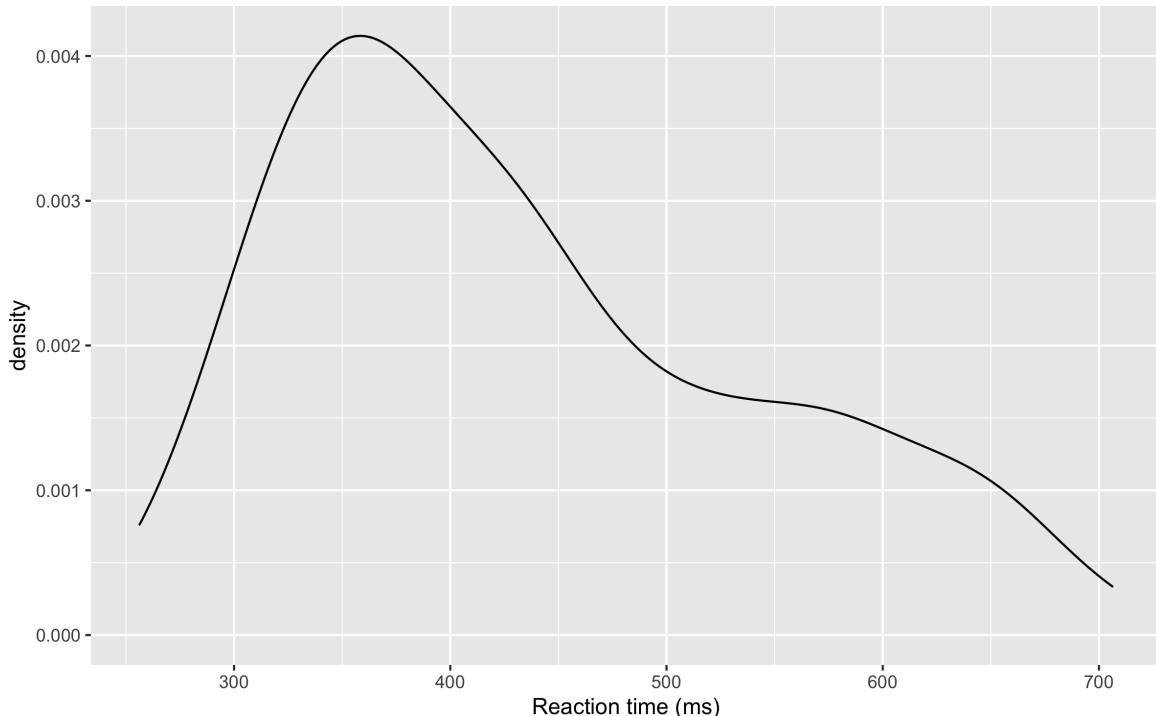


Figure 11. Density plot of reaction time.

466       **Grouped density plots.** Density plots are most useful for comparing the distributions  
 467       of different groups of data. Because the dataset is now in long format, it makes it  
 468       easier to map another variable to the plot because each variable is contained within a single  
 469       column.

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

```
477 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"))
```

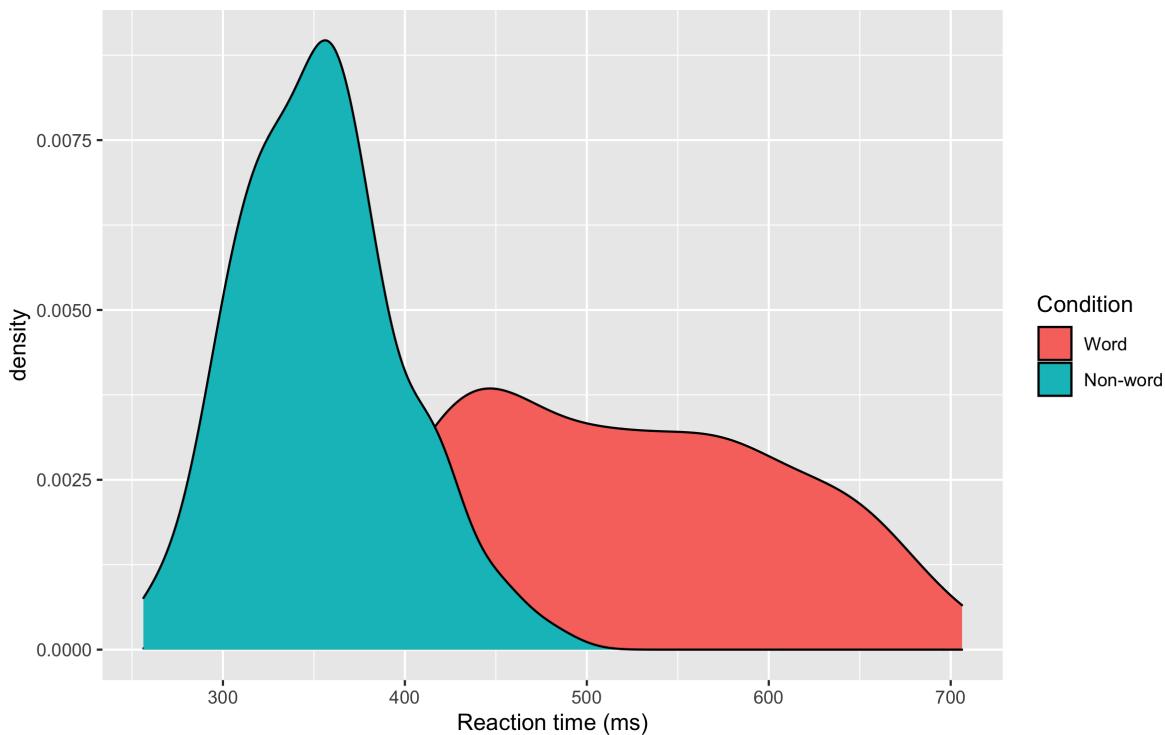


Figure 12. Density plot of reaction times grouped by condition.

478 **Scatterplots**

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

```
ggplot(dat_long, aes(x = rt, y = age)) +  
  geom_point()
```

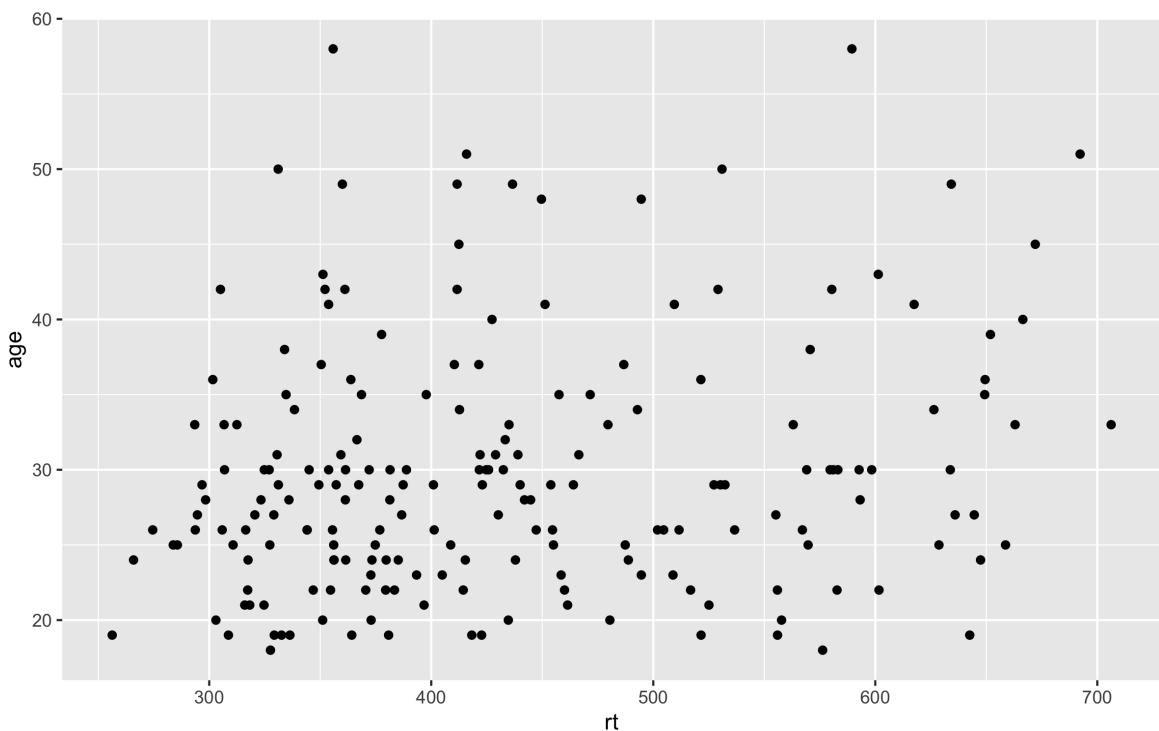


Figure 13. Point plot of reaction time versus age.

481 A line of best fit can be added with an additional layer that calls the function  
 482 `geom_smooth()`. The default is to draw a LOESS or curved regression line, however, a linear  
 483 line of best fit can be specified using `method = "lm"`. By default, `geom_smooth()` will also  
 484 draw a confidence envelope around the regression line, this can be removed by adding `se =`  
 485 `FALSE` to `geom_smooth()`. A common error is to try and use `geom_line()` to draw the line  
 486 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")
```

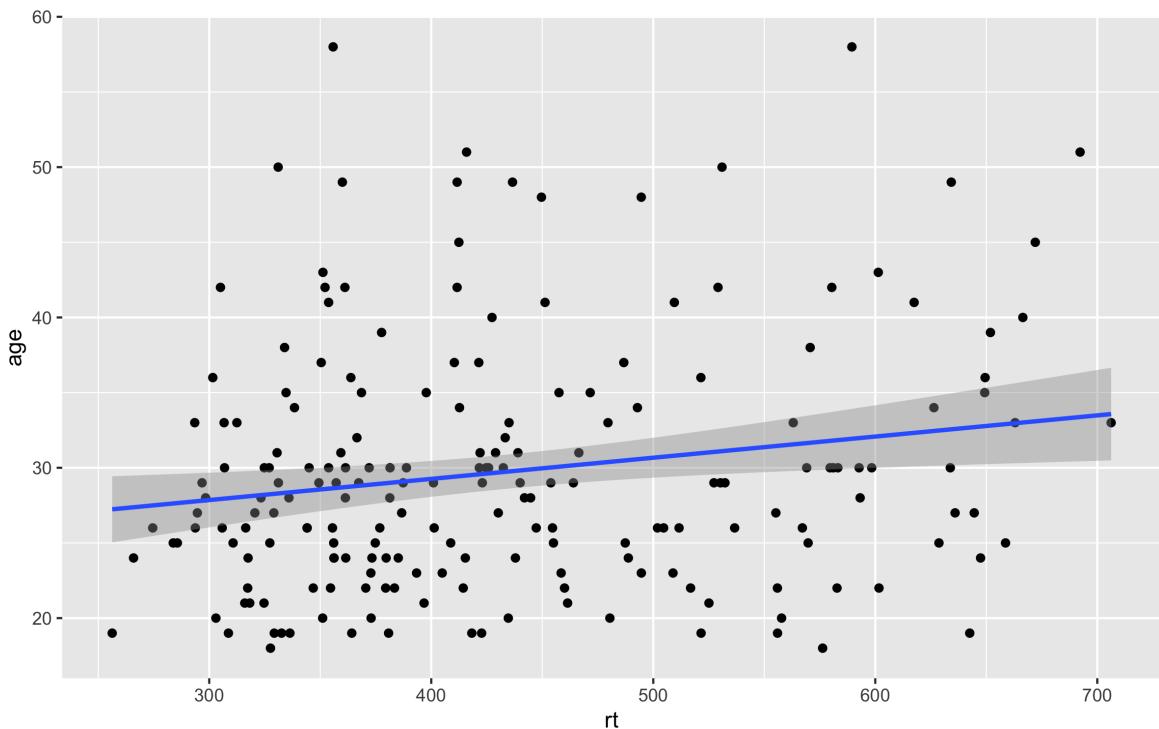


Figure 14. Line of best fit for reaction time versus age.

487       **Grouped scatterplots.** Similar to the density plot, the scatterplot can also be  
 488 easily adjusted to display grouped data. For `geom_point()`, the grouping variable is mapped  
 489 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"))
```

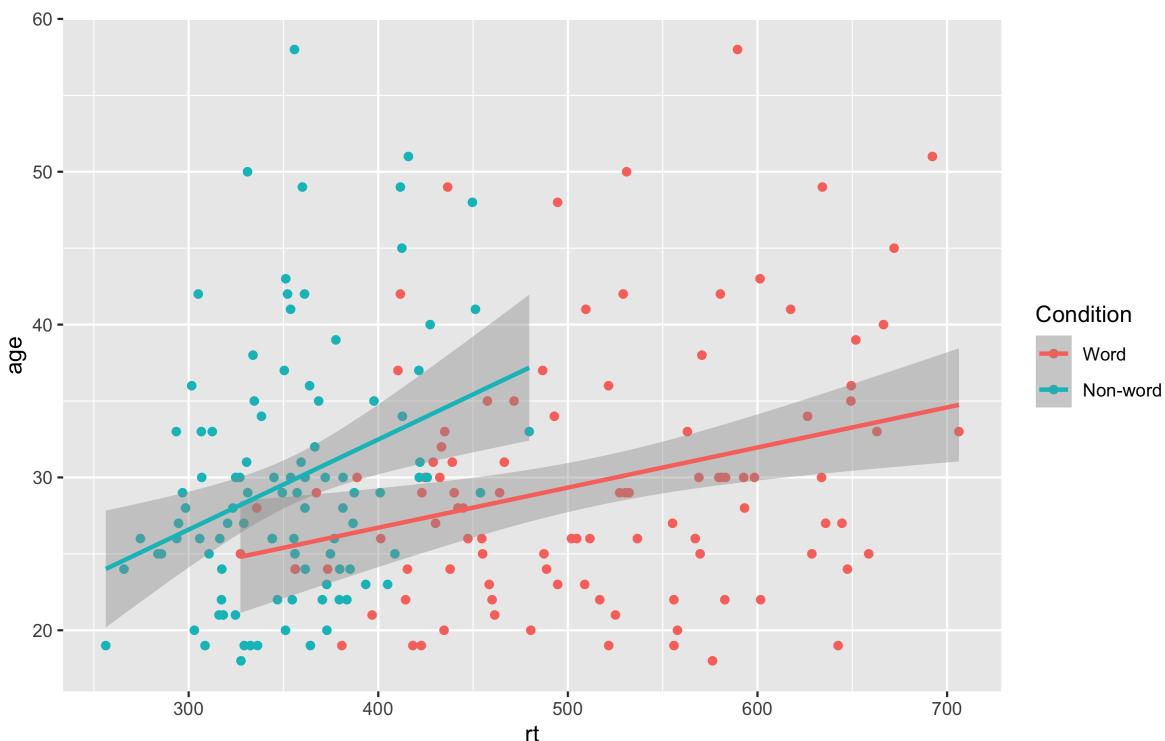


Figure 15. Grouped scatter plot of reaction time versus age by condition.

#### 490 Transforming data 2

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

```
497 ggplot(dat, aes(x = rt_word, y = rt_nonword, colour = language)) +  

  498   geom_point() +  

  499   geom_smooth(method = "lm")
```

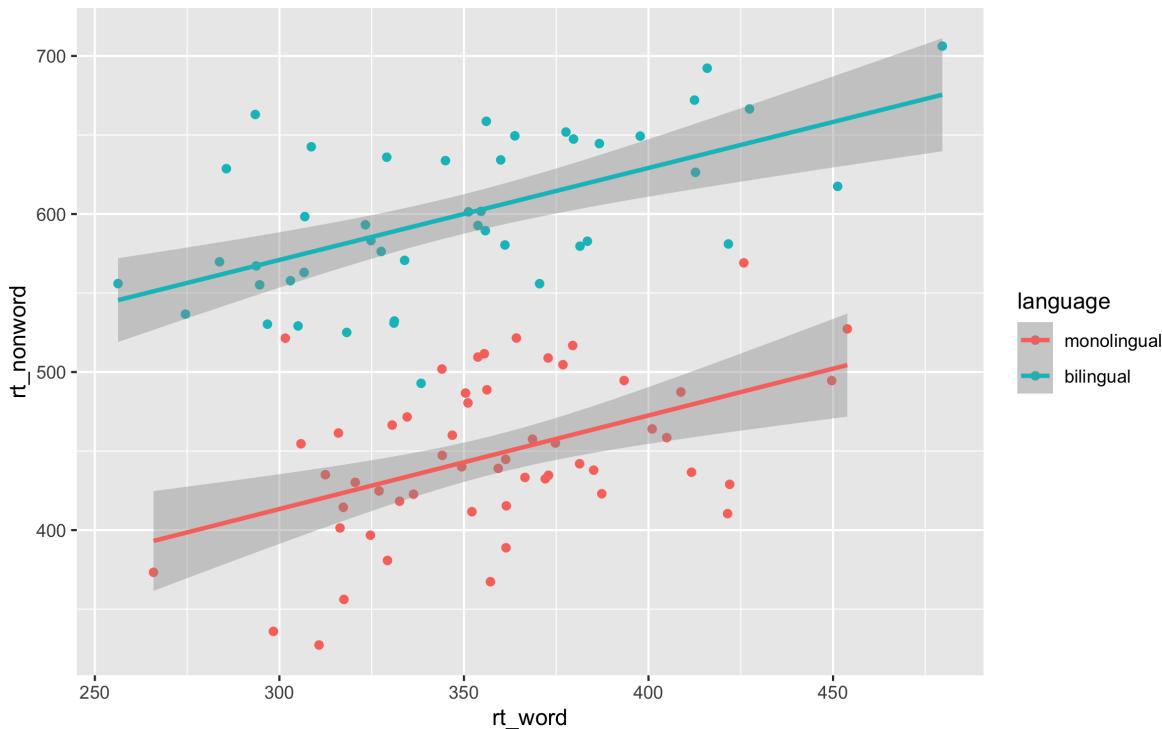


Figure 16. Scatterplot with data grouped by langauge group

497 However, there may also be cases when you do not have an original wide-form version  
 498 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

500 **Customisation 2**

501 **Accessible colour schemes.** One of the drawbacks of using `ggplot` for visualisation  
 502 is that the default colour scheme is not accessible (or visually appealing). The red and green  
 503 default palette is difficult for colour-blind people to differentiate, and also does not display  
 504 well in grey scale. You can specify exact custom colours for your plots, but one easy option  
 505 is to use a colour palette and the `viridis` scale functions call such a palette. These take

506 the same arguments as their default `scale` sister functions for updating axis names and  
 507 labels, but display plots in contrasting colours that can be read by colour-blind people and  
 508 that also print well in grey scale. The `viridis` scale functions provide a number of different  
 509 options for the colour - try setting `option` to any letter from A - E to see the different sets.

```
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")
```

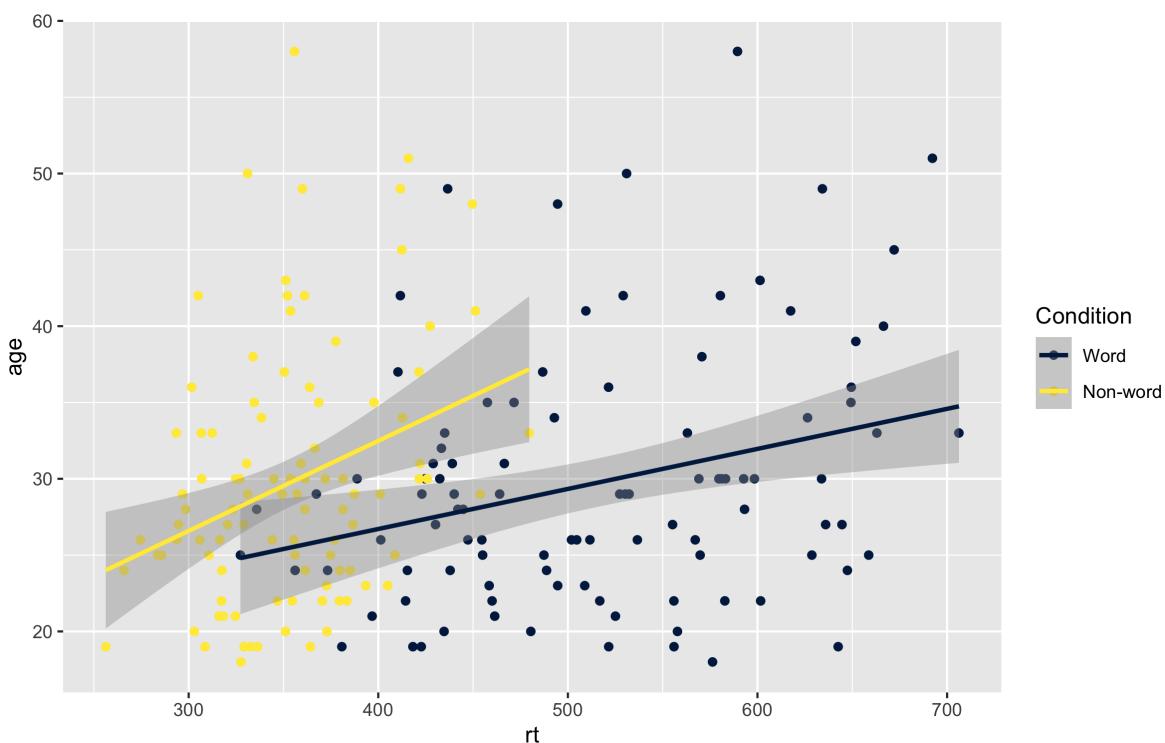


Figure 17. Use the viridis colour scheme for accessibility.

510 **Activities 2**

511 Before you move on try the following:

- 512 1. Use `fill` to created grouped histograms that display the distributions for `rt` for each  
 513 language group separately and also edit the fill axis labels. Try adding `position =  
 514 "dodge"` to `geom_histogram()` to see what happens.
- 515 2. Use `scale_*_*` functions to edit the name of the x and y-axis on the scatterplot
- 516 3. Use `se = FALSE` to remove the confidence envelope from the scatterplots

517     4. Remove `method = "lm"` from `geom_smooth()` to produce a curved regression line.  
518     5. Replace the default `scale_fill_*`() on the grouped density plot with the colour-blind  
519       friendly version.

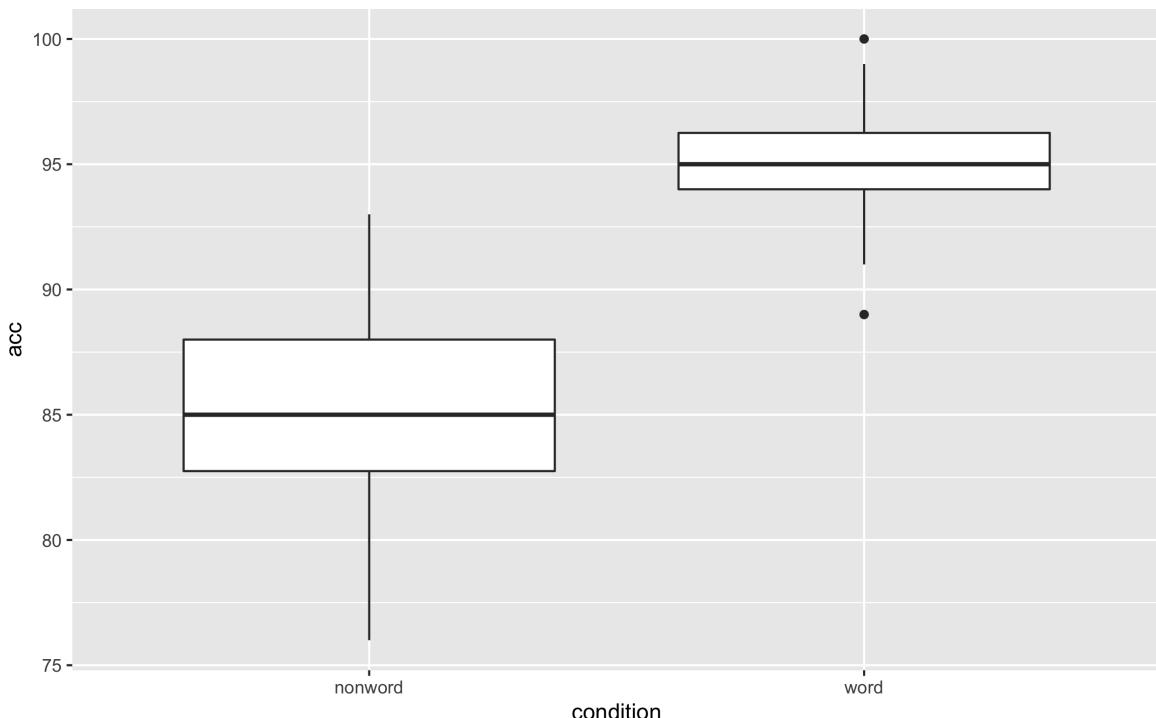
520 Representing Summary Statistics

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

524 Boxplots

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

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



*Figure 18.* Basic boxplot.

**Grouped boxplots.** As with histograms and density plots, `fill` can be used to create grouped boxplots. This looks like a lot of complicated code at first glance, but most 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")
```

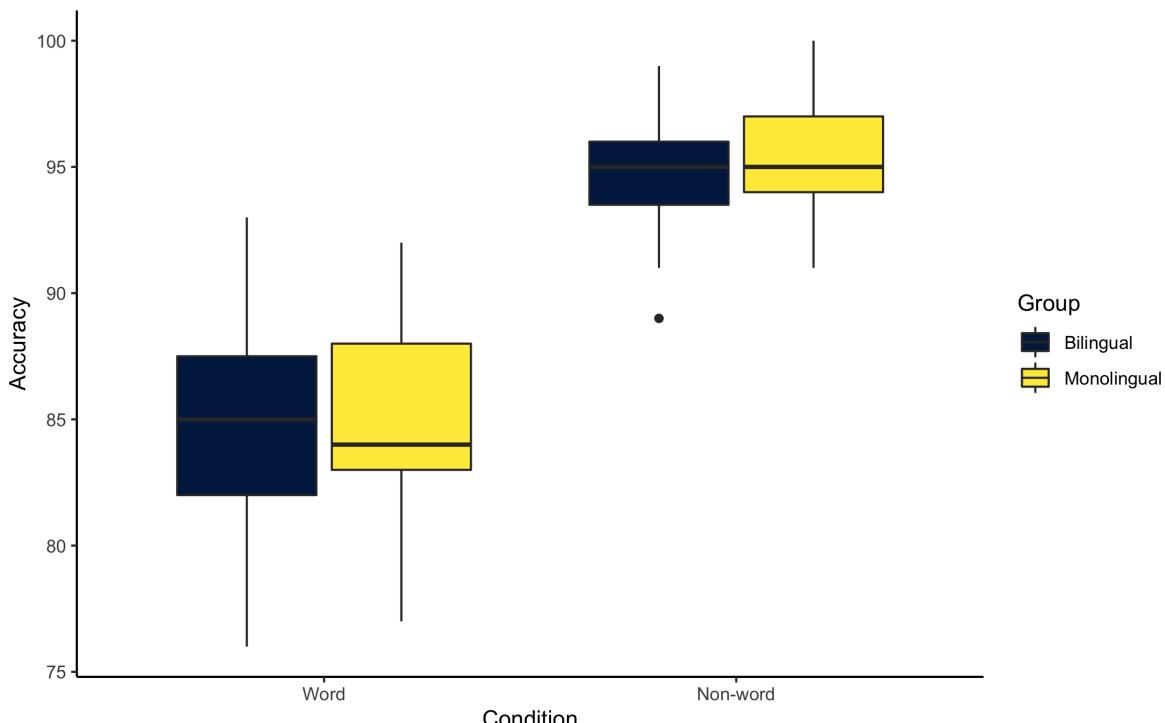


Figure 19. Grouped boxplots

### 531 Violin plots

532 Violin plots display the distribution of a dataset and can be created by calling  
 533 `geom_violin()`. They are so-called because the shape they make sometimes looks something  
 534 like a violin. They are essentially a mirrored density plot on its side. Note that the below code  
 535 is identical to the code used to draw the boxplots above, except for the call to `geom_violin()`  
 536 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")
```

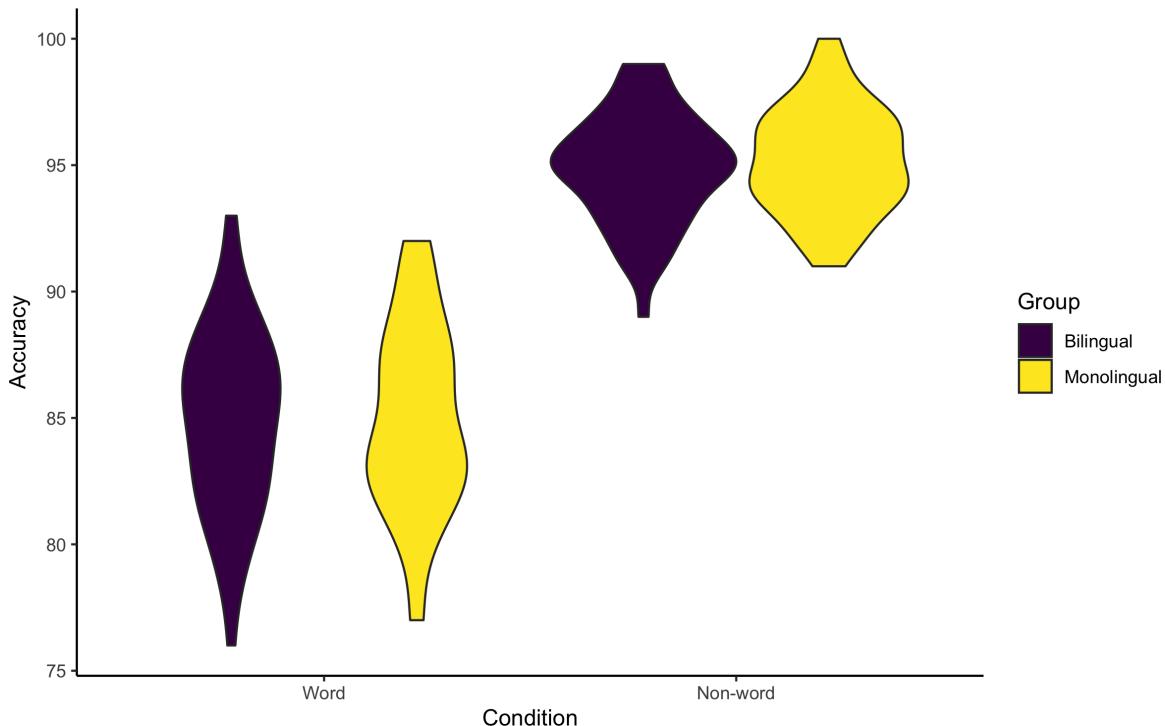


Figure 20. Violin plot.

### 537 Bar chart of means

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

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

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

- 551     • `fun` specifies the summary function that gives us the y-value we want to plot, in this  
 552       case, `mean`.
- 553     • `geom` specifies what shape or plot we want to use to display the summary. For the first  
 554       layer we will specify `bar`. As with the other geom-type functions we have shown you,  
 555       this part of the `stat_summary()` function is tied to the aesthetic mapping in the first  
 556       line of code. The underlying statistics for a bar chart means that we must specify an  
 557       IV (x-axis) as well as the DV (y-axis).

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

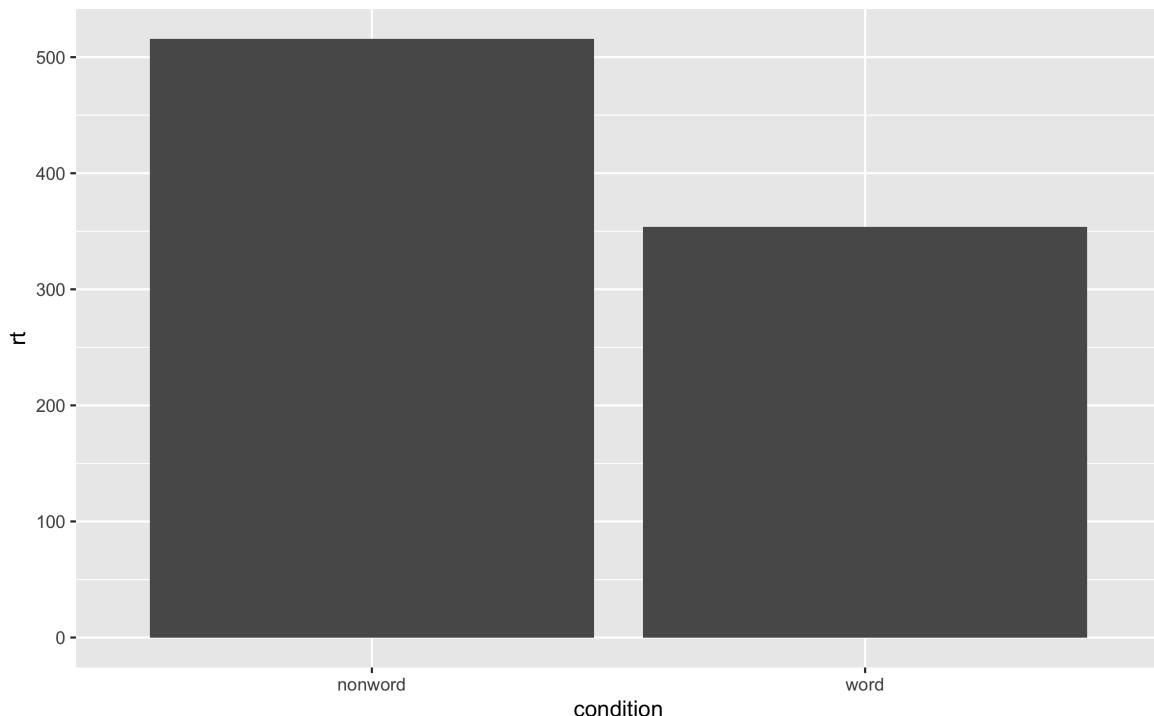


Figure 21. Bar plot of means.

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

- 563     • Whilst `fun` returns a single value (y) per condition, `fun.data` returns the y-values we  
 564       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)
```

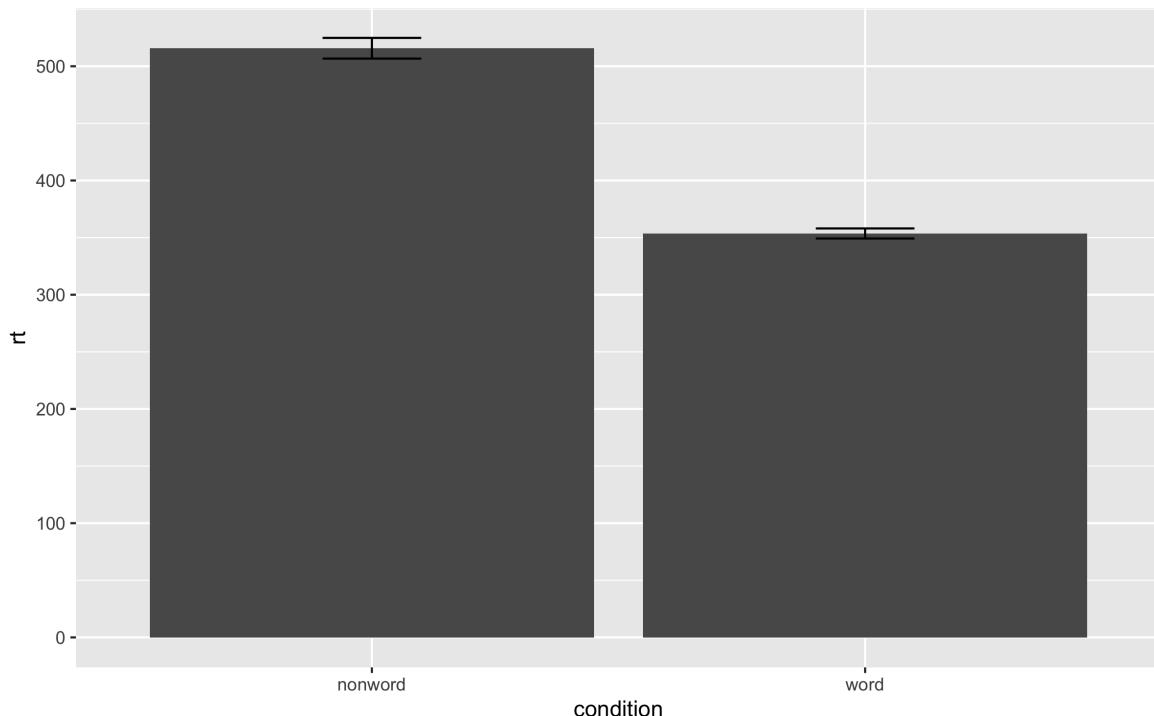


Figure 22. Bar plot of means with error bars representing SE.

## 565 Violin-boxplot

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

- 574
  - `fatten = NULL` in the boxplot geom removes the median line, which can make it easier  
 575       to see the mean and error bars. Including this argument will result in the warning  
 576       message `Removed 1 rows containing missing values (geom_segment)` and is not  
 577       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)
```

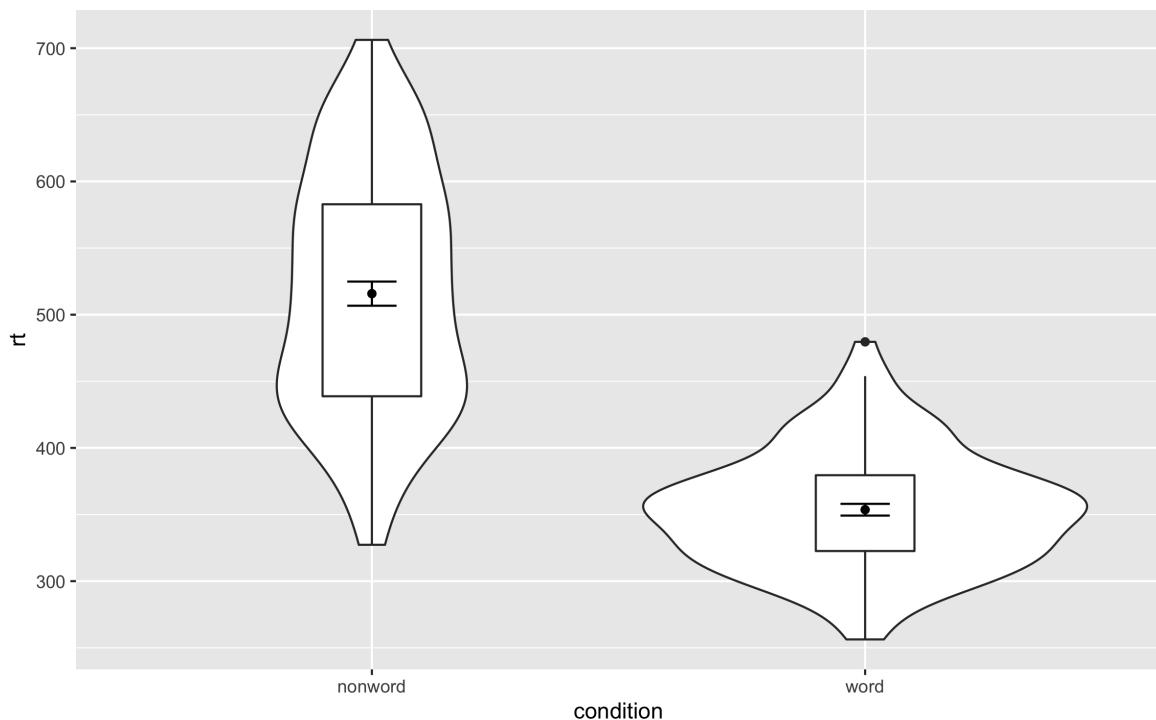


Figure 23. Violin-boxplot with mean dot and standard error bars.

578 It is important to note that the order of the layers matters and it is worth experimenting  
 579 with the order to see where the order matters. For example, if we call `geom_boxplot()`  
 580 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)
```

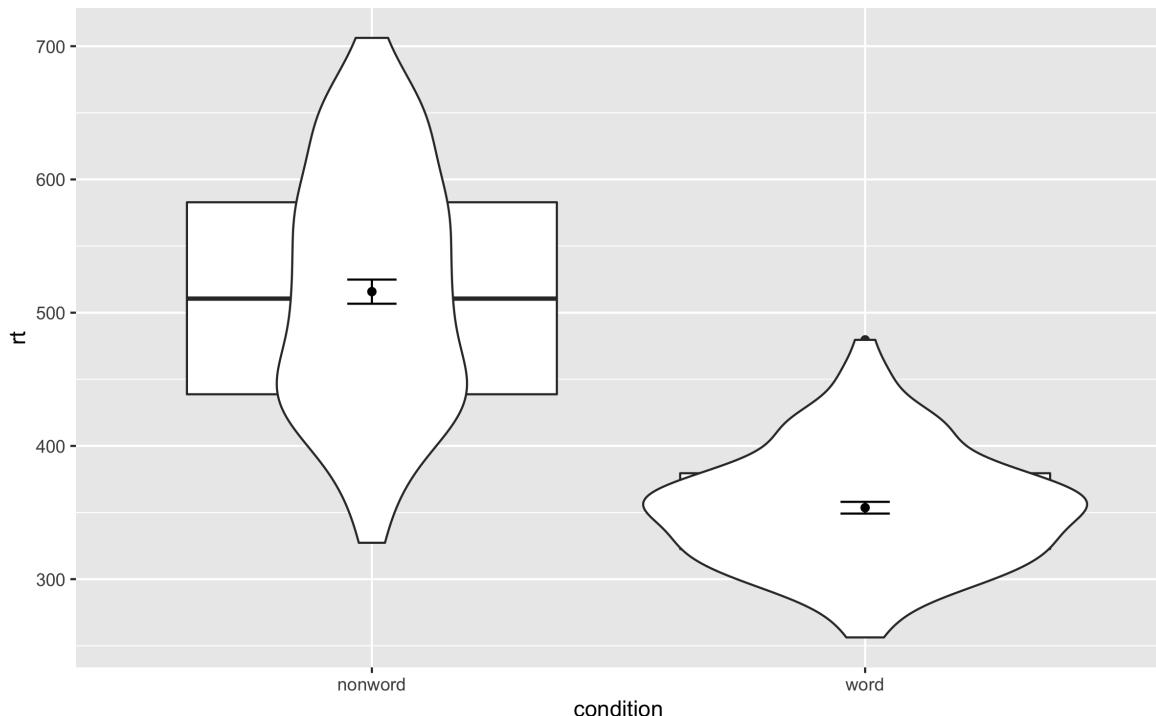


Figure 24. Plot with the geoms in the wrong order.

581     **Grouped violin-boxplots.** As with previous plots, another variable can be mapped  
 582     to `fill` for the violin-boxplot. However, simply adding `fill` to the mapping causes the  
 583     different components of the plot to become misaligned because they have different default  
 584     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)
```

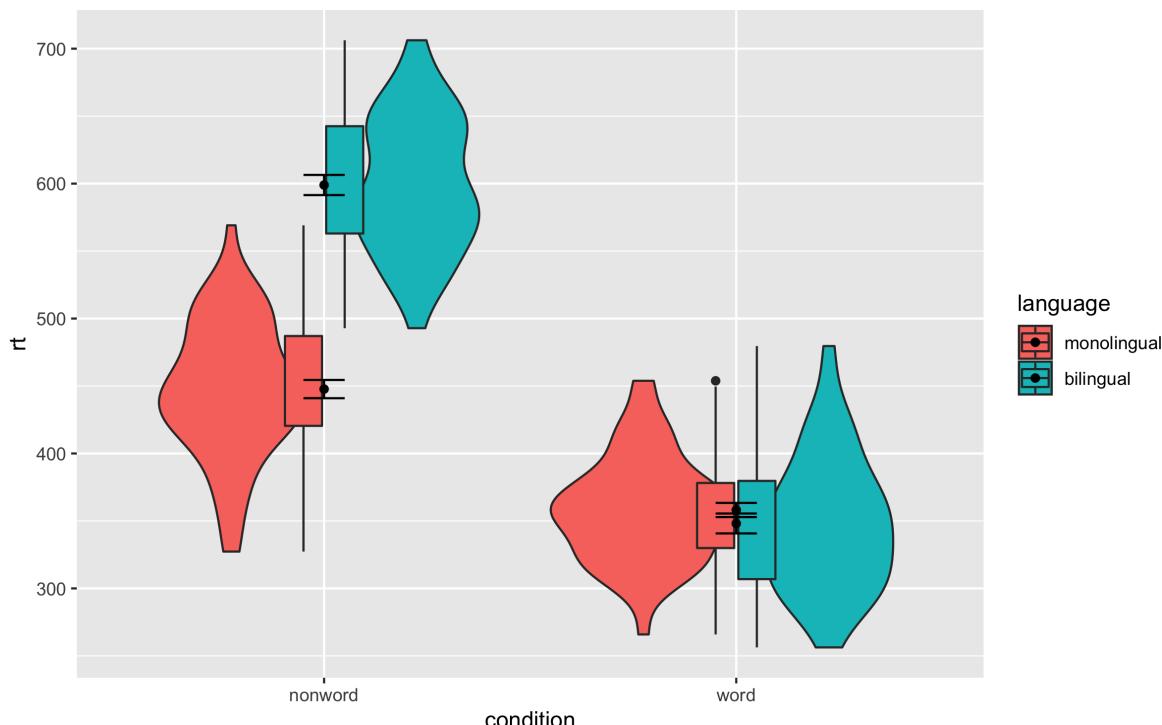


Figure 25. Grouped violin-boxplots without repositioning.

585 To rectify this we need to adjust the argument `position` for each of the misaligned  
 586 layers. `position_dodge()` instructs R to move (dodge) the position of the plot component  
 587 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))
```

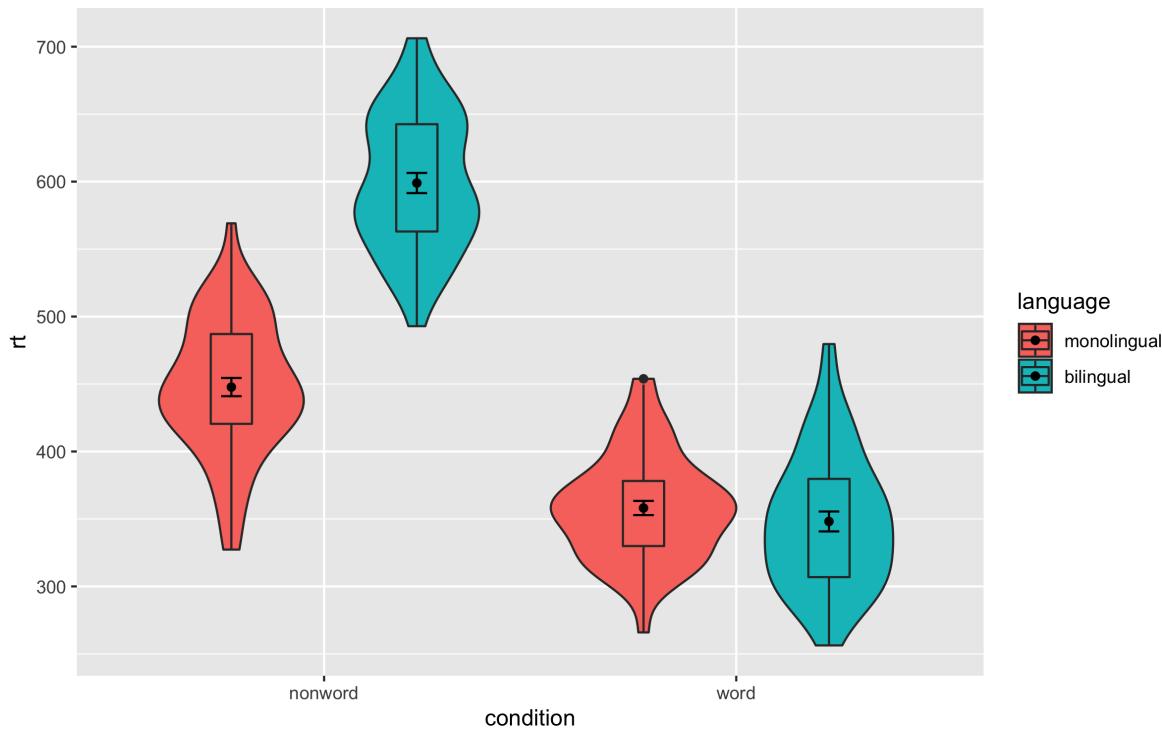


Figure 26. Grouped violin-boxplots with repositioning.

### 588 Customisation part 3

589 Combining multiple type of plots can present an issue with the colours, particularly  
 590 when the viridis scheme is used - in the below example it is hard to make out the black lines  
 591 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()
```

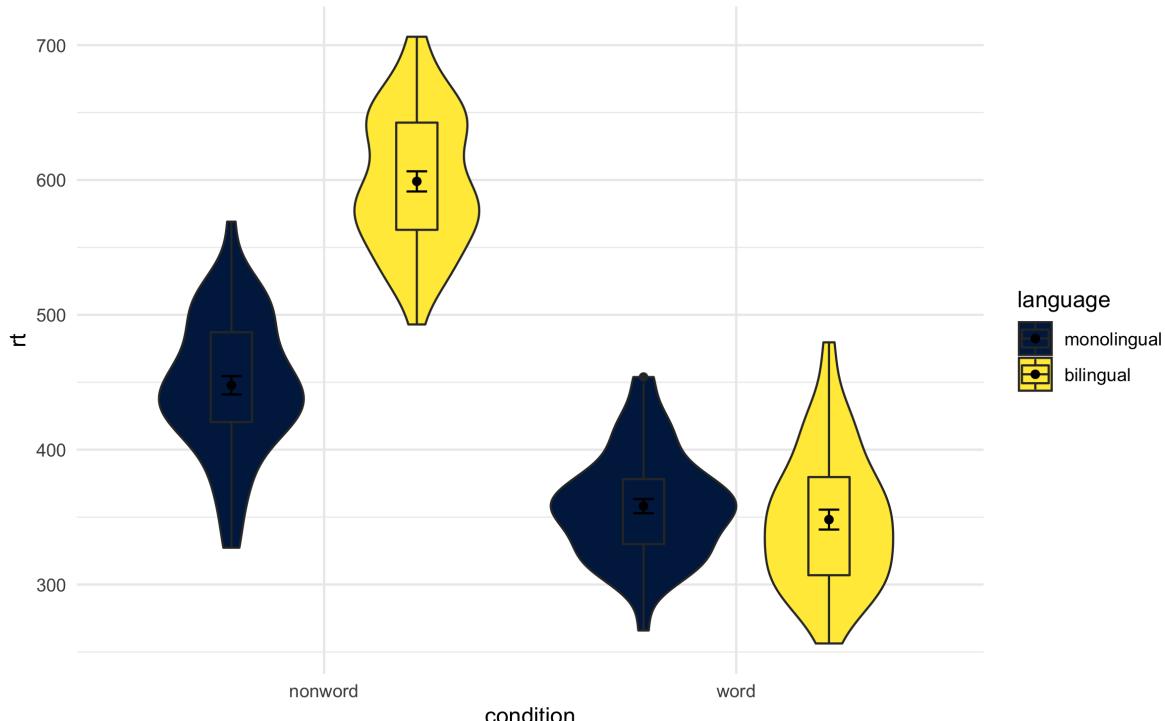


Figure 27. A color scheme that makes lines difficult to see.

592 There are a number of solutions to this problem. First, we can change the colour of  
 593 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()
```

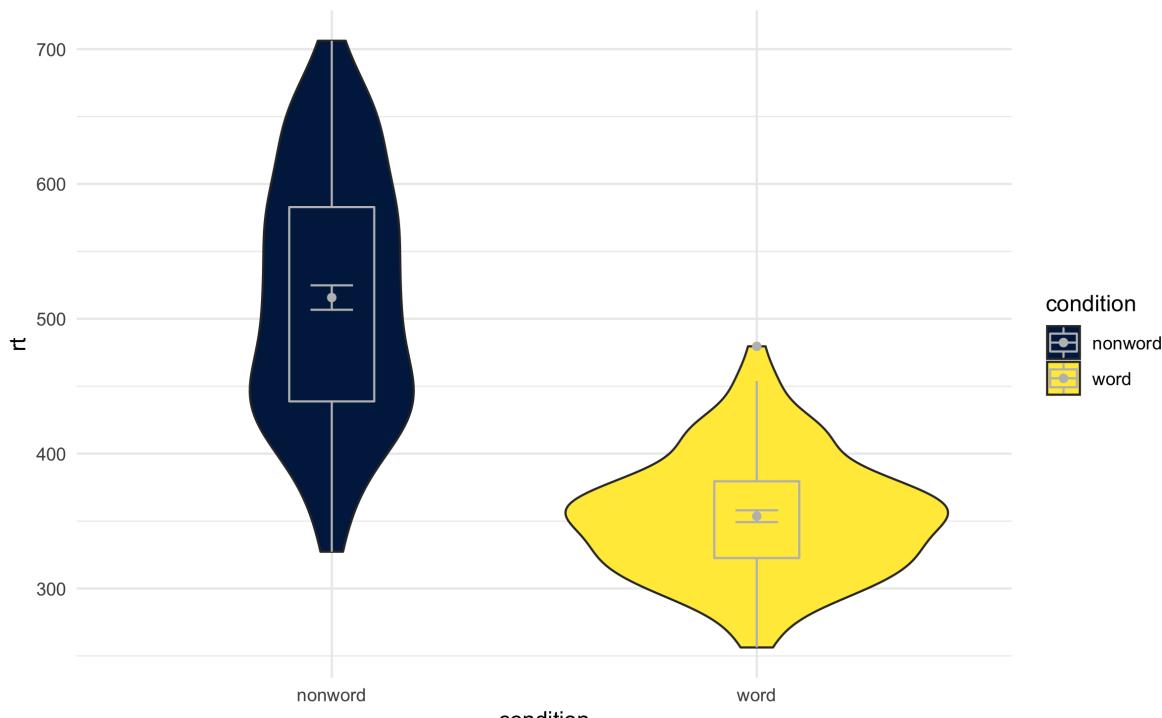


Figure 28. Manually changing the line colors.

594 We can also keep the original colours but adjust the transparency of each layer using  
 595 `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()
```

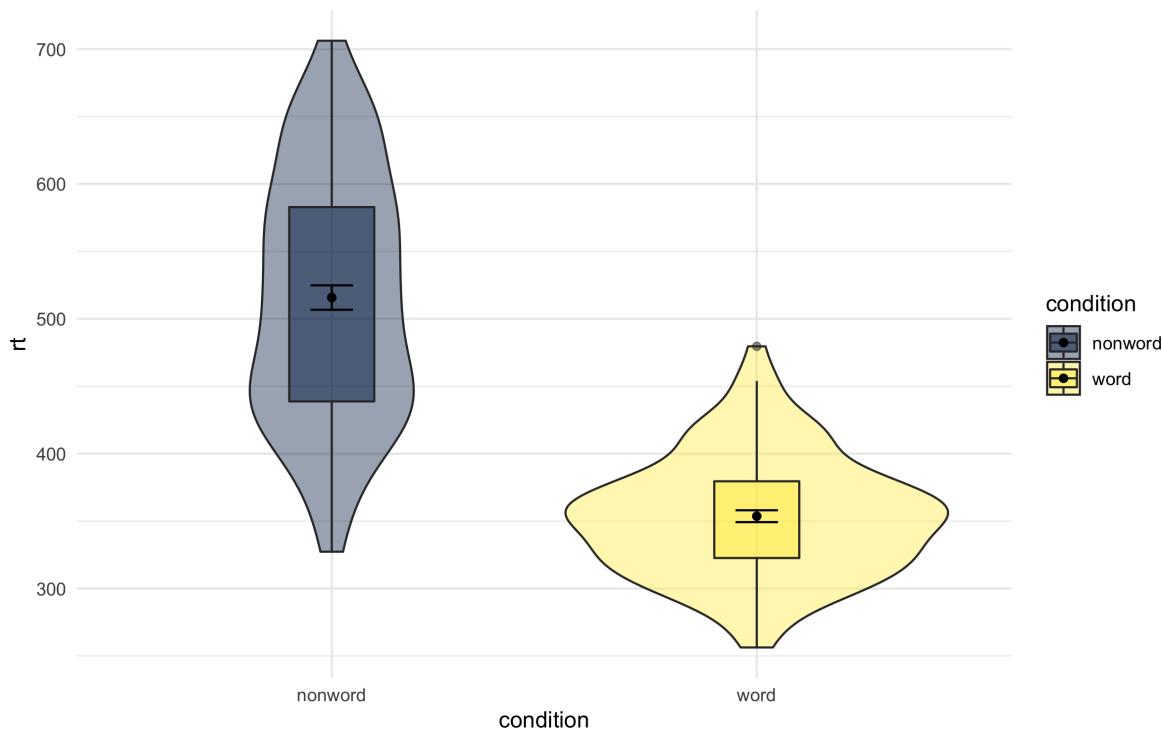


Figure 29. Using transparency on the fill color.

### 596 Activities 3

597 Before you go on, do the following:

- 598 1. Review all the code you have run so far. Try to identify the commonalities between  
599 each plot's code and the bits of the code you might change if you were using a different  
600 dataset.
- 601 2. Take a moment to recognise the complexity of the code you are now able to read.
- 602 3. For the violin-boxplot, for `geom = "point"`, try changing `fun` to `median`
- 603 4. For the violin-boxplot, for `geom = "errorbar"`, try changing `fun.data` to  
604 `mean_cl_normal` (for 95% CI)
- 605 5. Go back to the grouped density plots and try changing the transparency with `alpha`.

### 606 Multi-part Plots

#### 607 Interaction plots

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

- **shape** acts much like **fill** in previous plots, except that rather than producing different colour fills for each level of the IV, the data points are given different shapes.
- **size** lets you change the size of lines and points. You usually don't want different groups to be different sizes, so this option is set inside the relevant **geom\_\***() function, not inside the **aes()** function.
- **scale\_color\_manual()** works much like **scale\_color\_discrete()** except that it lets you specify the colour values manually, instead of them being automatically applied based on the palette you choose/default to. You can specify RGB colour values or a list of predefined colour names - all available options can be found by running **colours()** in the console. Other manual scales are also available, for example, **scale\_fill\_manual**.

```
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()
```

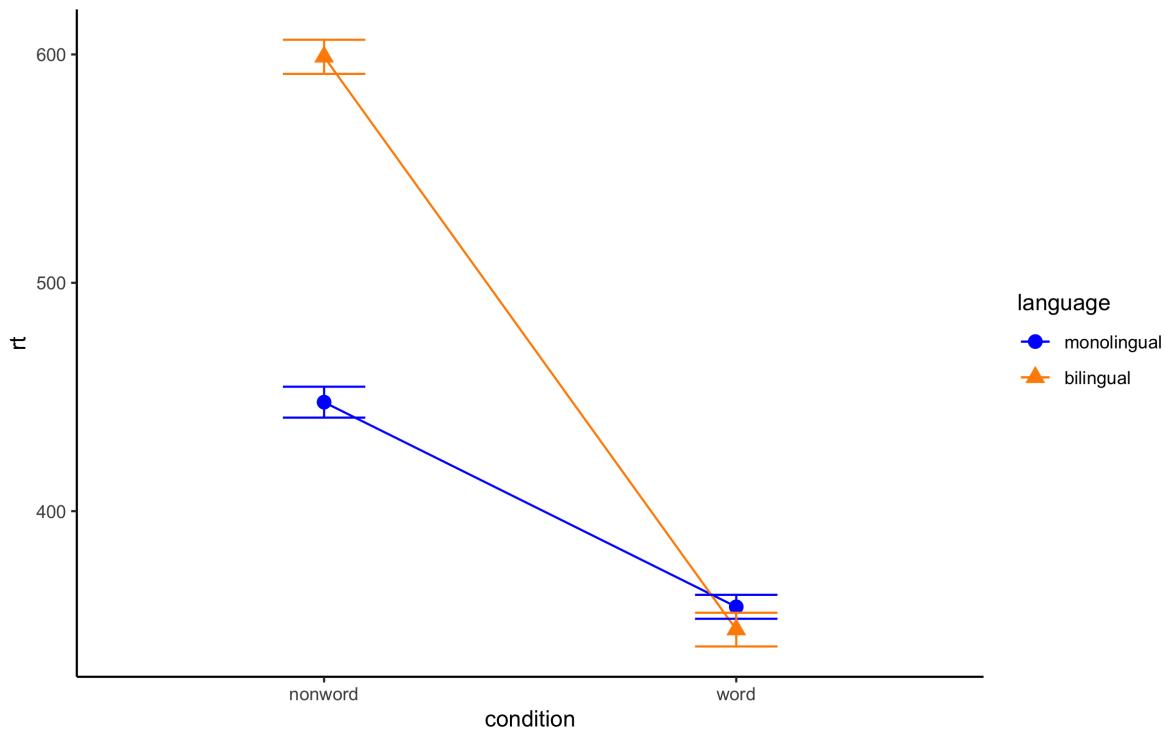


Figure 30. Interaction plot.

621    **Combined interaction plots**

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

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

- 629       • The first call to `ggplot()` sets up the default mappings of the plot that will be used  
 630           unless otherwise specified - the `x`, `y` and `group` variable. Note two additions are `shape`  
 631           and `linetype` that will vary those elements according to the language variable.
- 632       • `geom_point()` overrides the default mapping by setting its own `colour` to draw the  
 633           data points from each language group in a different colour. `alpha` is set to a low value  
 634           to aid readability. Note that because the aesthetic override was defined within the  
 635           geom function, the colours are not represented in the legend.
- 636       • Similarly, `geom_line()` overrides the default grouping variable so that a line is drawn  
 637           to connect the individual data points for each *participant* (`group = id`) rather than  
 638           each language group, and also sets the colours. The default line type is also overridden  
 639           and set for all lines to be solid.
- 640       • Finally, the calls to `stat_summary()` remain largely as they were, with the exception  
 641           of setting `colour = "black"` and `size = 2` so that the overall means and error bars  
 642           can be more easily distinguished from the individual data points. Because they do not  
 643           specify an individual mapping, they use the defaults (e.g., the lines are connected by  
 644           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()
```

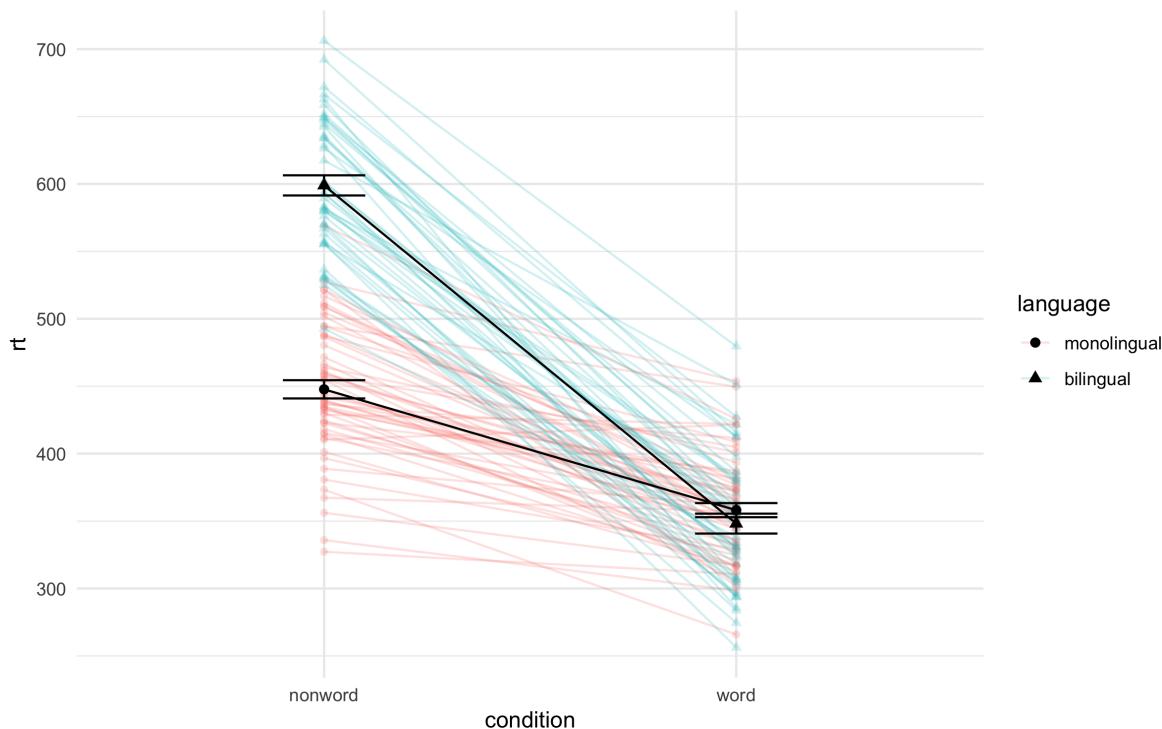


Figure 31. Interaction plot with by-participant data

## 645 Facets

646 So far we have produced single plots that display all the desired variables in one,  
 647 however, there are situations in which it may be useful to create separate plots for each level  
 648 of a variable. The below code is an adaptation of the code used to produce the grouped  
 649 scatterplot (see Figure 25) in which it may be easier to see how the relationship changes  
 650 when the data are not overlaid.

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

```
653 ggplot(dat_long, aes(x = rt, y = age)) +  

  654   geom_point() +  

  655   geom_smooth(method = "lm") +  

  656   facet_wrap(~condition)
```

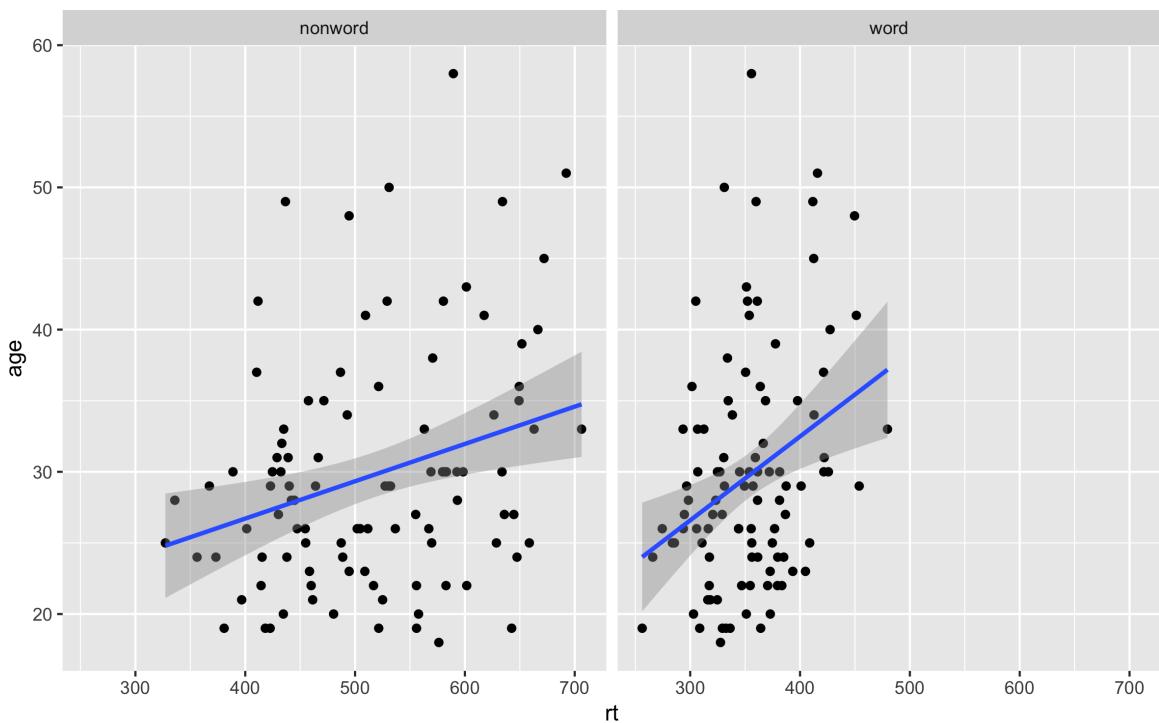


Figure 32. Faceted scatterplot

653 As another example, we can use `facet_wrap()` as an alternative to the grouped  
 654 violin-boxplot (see Figure 26) in which the variable `language` is passed to `facet_wrap()`  
 655 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()
```

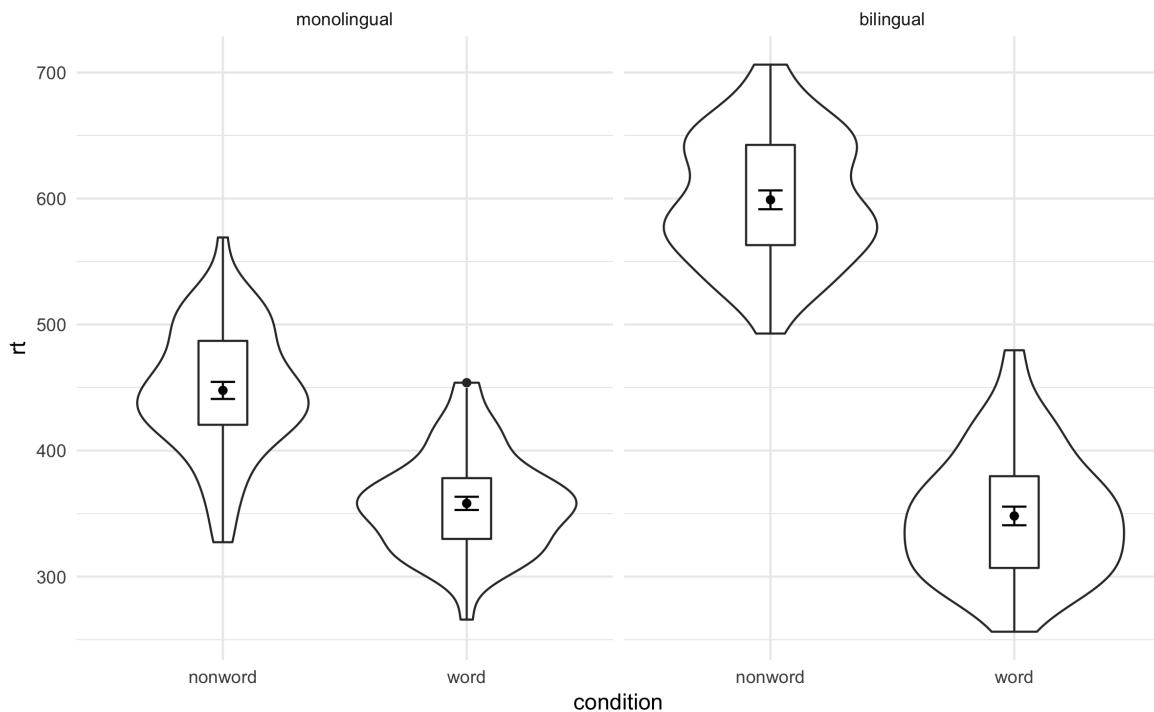


Figure 33. Facted violin-boxplot

656 Finally, note that editing the labels for faceted variables uses the `labeller` function,  
 657 calling it within `facet_wrap`. `labeller` uses syntax that is different from what we have  
 658 shown you up to now. Syntax simply refers to the rules about the words that a function  
 659 uses. In the exercise about looking for the underlying rules to the code from the previous  
 660 chapter you likely tapped into the rules for the various `ggplot` functions. However, once in  
 661 a while you will come across something that just does not follow the rules you have been  
 662 building on and `labeller` is an example of that.

663 Before we explain it, read the below code and see if you can figure out the syntactic  
 664 rules for the function.

```
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,  

            labeller = labeller(  

              language = c(monolingual = "Monolingual participants",  

                           bilingual = "Bilingual participants")))) +  

  theme_minimal()
```

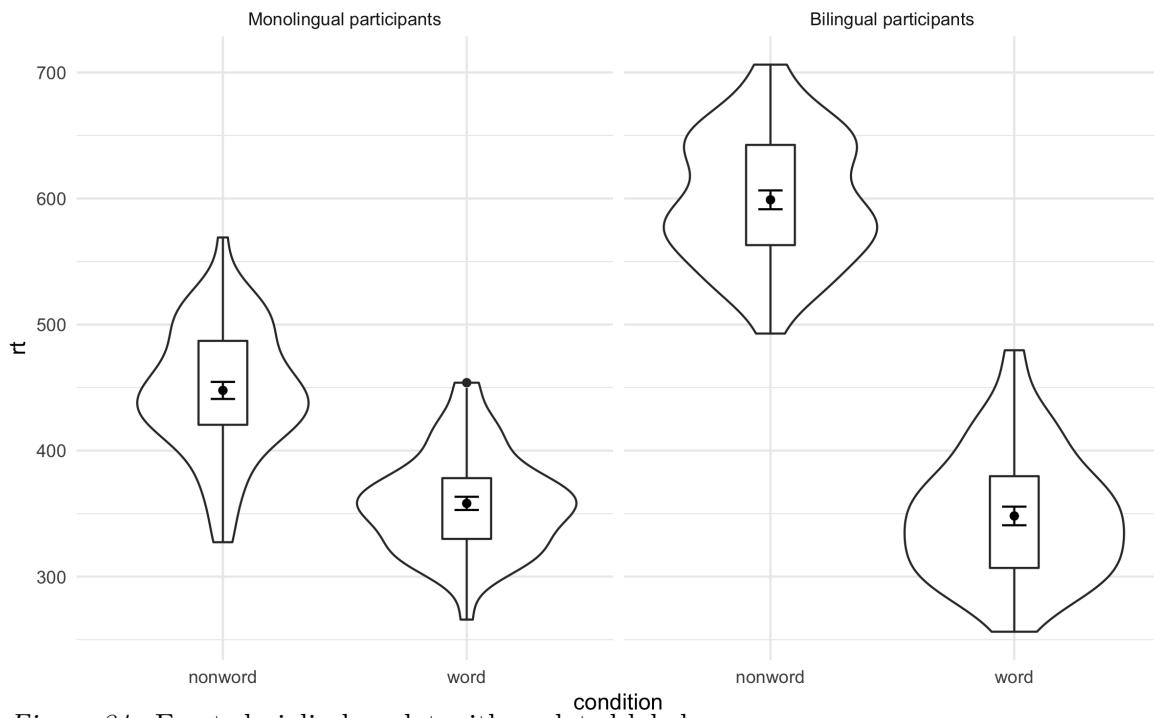


Figure 34. Faceted violin-boxplot with updated labels

Even though the `labeller` syntax is new, it is still fairly logical, for each level of the variable you want to facet by, you need to specify a label, so you call the variable by naming it: `labeller = labeller(language =` then you make a list of the levels (monolingual and bilingual) and for each you tell the function its new name `c(monolingual = "Monolingual participants", bilingual = "Bilingual participants")`.

## 670 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 `ggplot` 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()
```

## 679 Saving plots as images

680 In addition to saving plots to objects for further use in R, the function `ggsave()`  
681 can be used to save plots as images on your hard drive. The only required argument for  
682 `ggsave` is the file name of the image file you will create, complete with file extension (this  
683 can be “eps,” “ps,” “tex,” “pdf,” “jpeg,” “tiff,” “png,” “bmp,” “svg” or “wmf”). By default,  
684 `ggsave()` will save the last plot displayed, however, you can also specify a specific plot  
685 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
```

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

## 688 Multiple plots

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

695 **Combining two plots.** Two plots can be combined side-by-side or stacked on top  
696 of each other. These combined plots could also be saved to an object and then passed to  
697 `ggsave`.

```
p1 + p2 # side-by-side
```

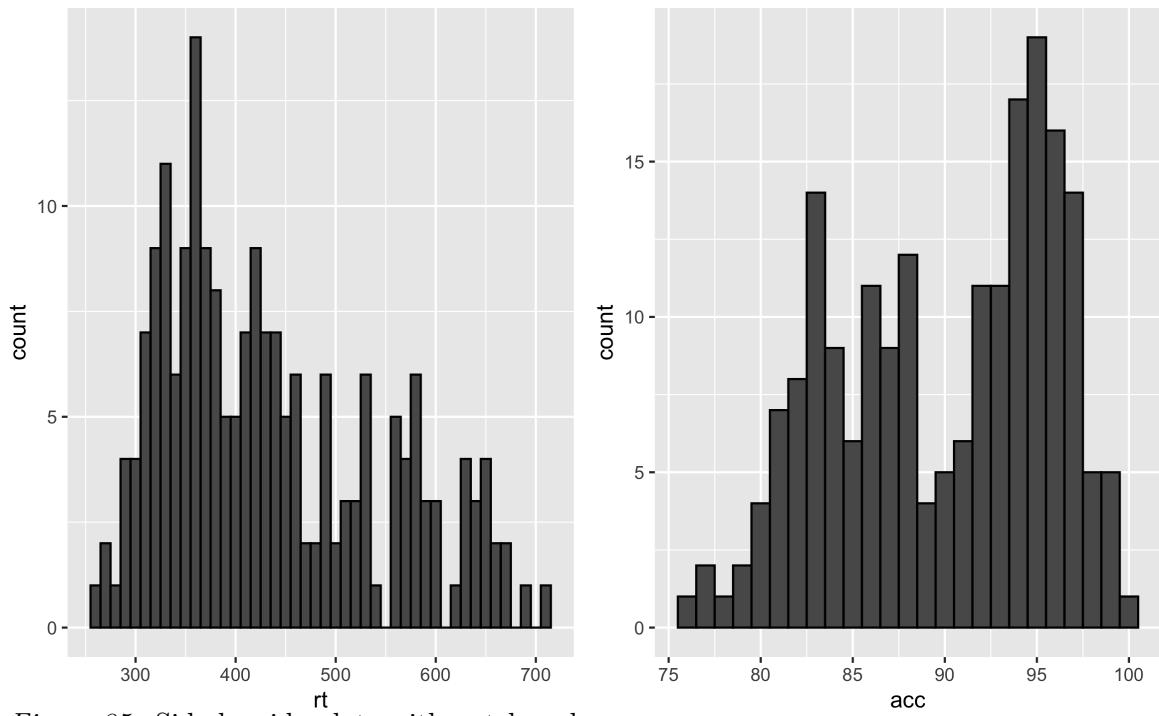


Figure 35. Side-by-side plots with patchwork

p1 / p2 # stacked

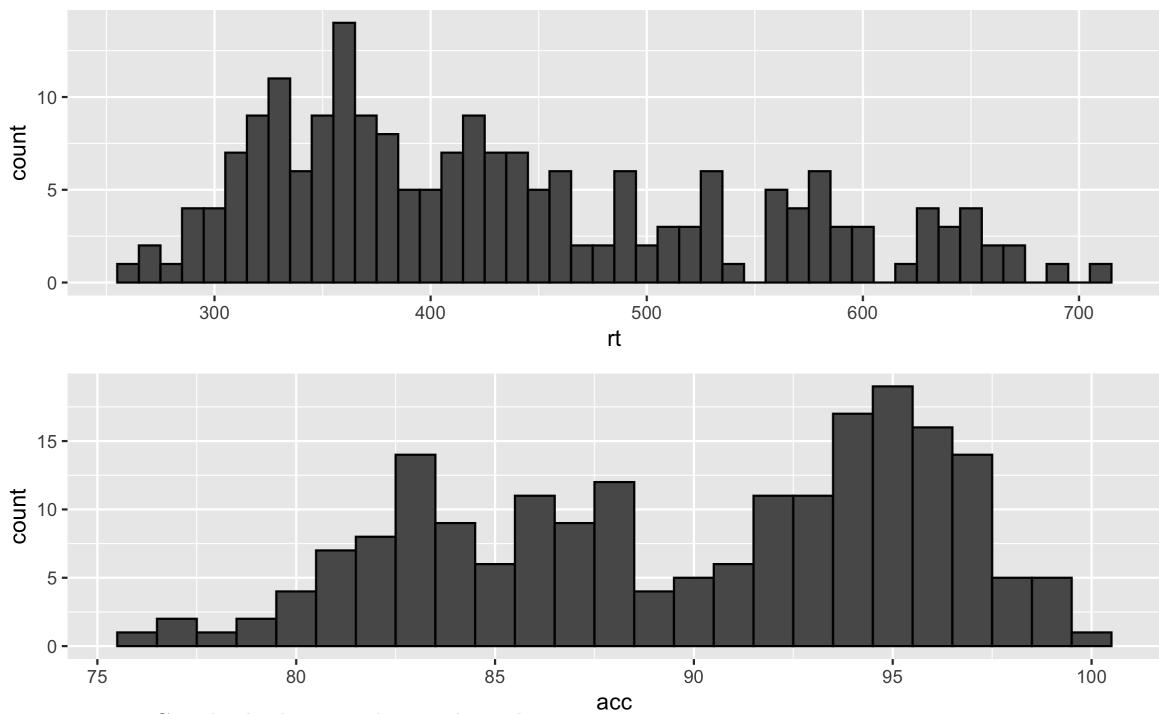


Figure 36. Stacked plots with patchwork

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

```
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(~language,  

            labeller = labeller(  

              language = (c(monolingual = "Monolingual participants",  

                            bilingual = "Bilingual participants")))) +  

  theme_minimal()
```

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

```
p1 / p5 / p6  

(p1 + p6) / p5  

p6 | p1 / p5
```

## 705 Customisation part 4

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

```
p5 + labs(x = "Type of word",  

          y = "Reaction time (ms)",  

          title = "Language group by word type interaction plot",
```

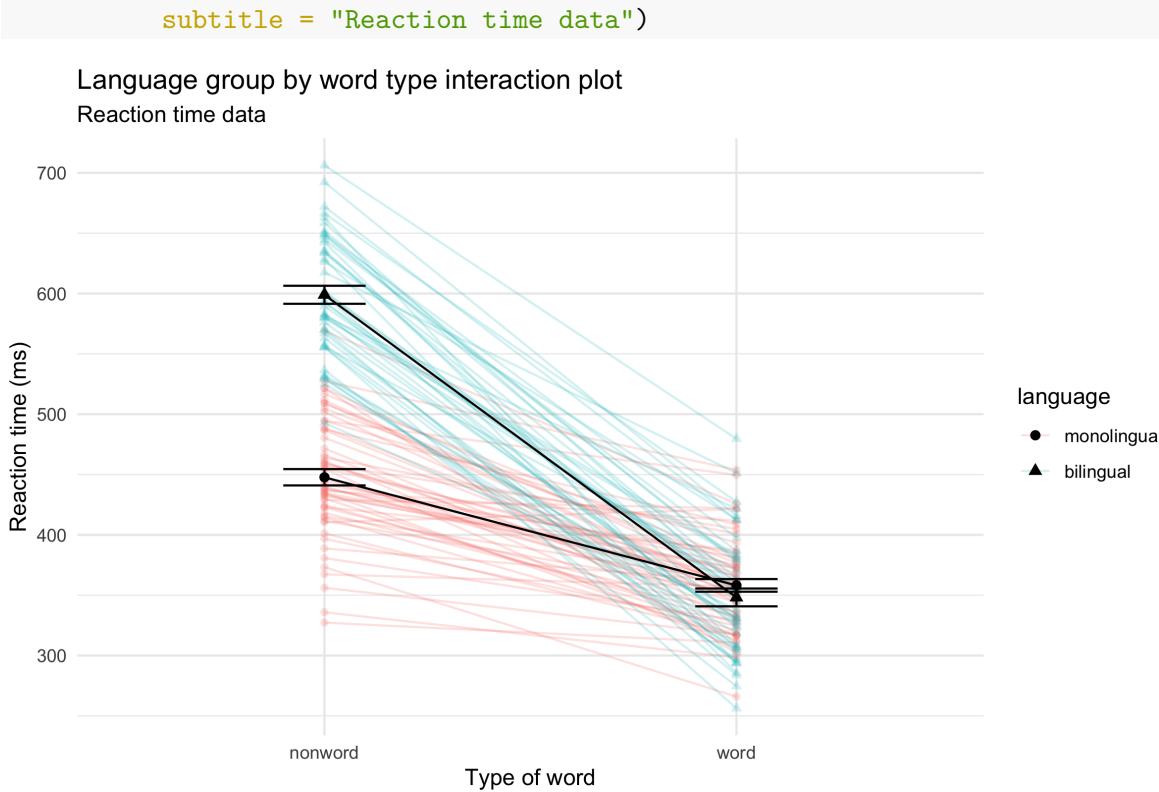


Figure 37. Plot with edited labels and title

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

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

720 The code below adds `fill = language` to the violin-boxplots that are also faceted  
721 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(~language,  
            labeller = labeller(  
              language = (c(monolingual = "Monolingual participants",  
                            bilingual = "Bilingual participants")))) +  
  theme_minimal() +  
  scale_fill_viridis_d(option = "E")
```

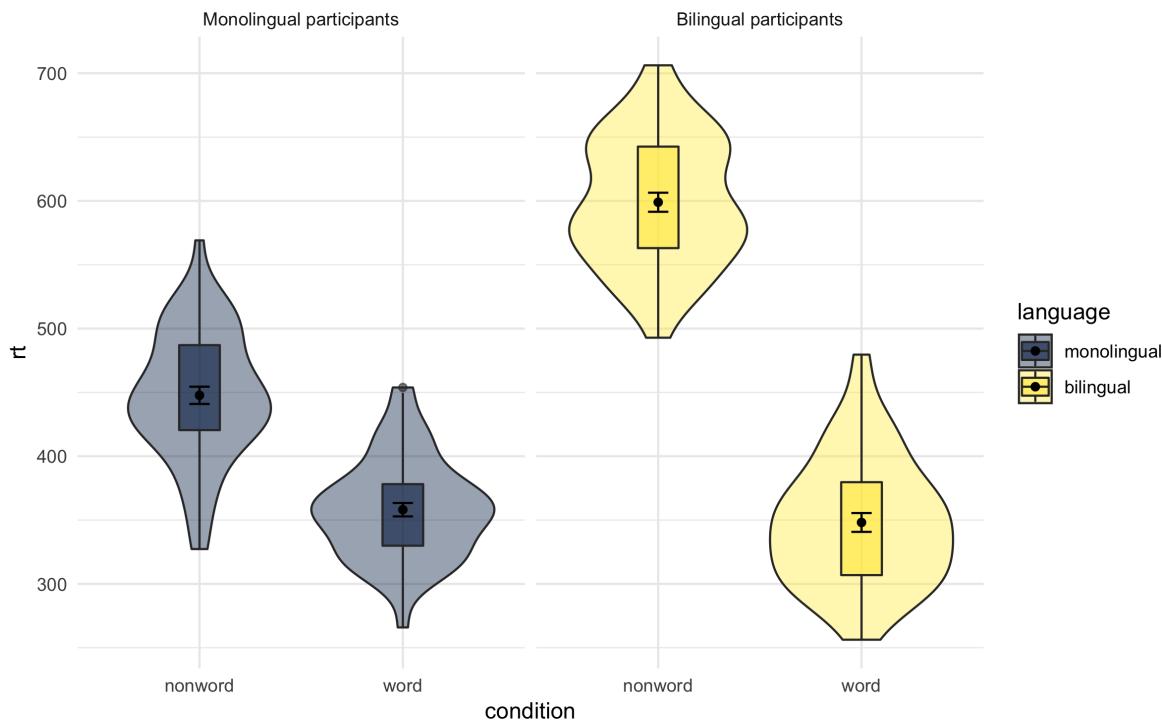


Figure 38. Violin-boxplot with redundant legend

722 Specifying a `fill` variable means that by default, R produces a legend for that variable.  
 723 However, the use of colour is redundant with the facet labels, so you can remove this legend  
 724 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(~language,
             labeller = labeller(language = c(monolingual = "Monolingual participants", bi
```

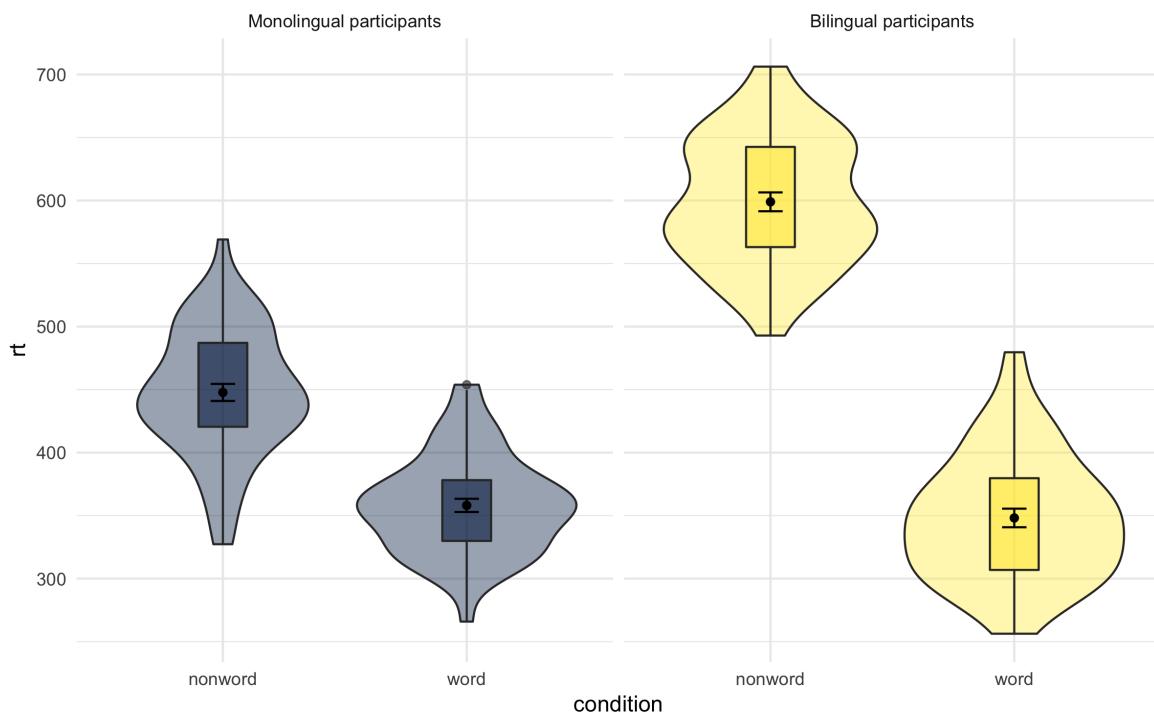


Figure 39. Plot with suppressed redundant legend

## Activities 4

Before you go on, do the following:

- Rather than mapping both variables (`condition` and `language`) to a single interaction plot with individual participant data, instead produce a faceted plot that separates the monolingual and bilingual data. All visual elements should remain the same (colours and shapes) and you should also take care not to have any redundant legends.
- Choose your favourite three plots you've produced so far in this tutorial, tidy them up with axis labels, your preferred colour scheme, and any necessary titles, and then combine them using `patchwork`. If you're feeling particularly proud of them, post them on Twitter using `#PsyTeachR`.

## Advanced Plots

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

We will use some custom functions: `geom_split_violin()` and `geom_flat_violin()`, which you can access through the `introdataviz` package. These functions are modified

743 from (Allen et al., 2021).

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

744 **Split-violin plots**

745 Split-violin plots remove the redundancy of mirrored violin plots and make it easier to  
 746 compare the distributions between multiple conditions.

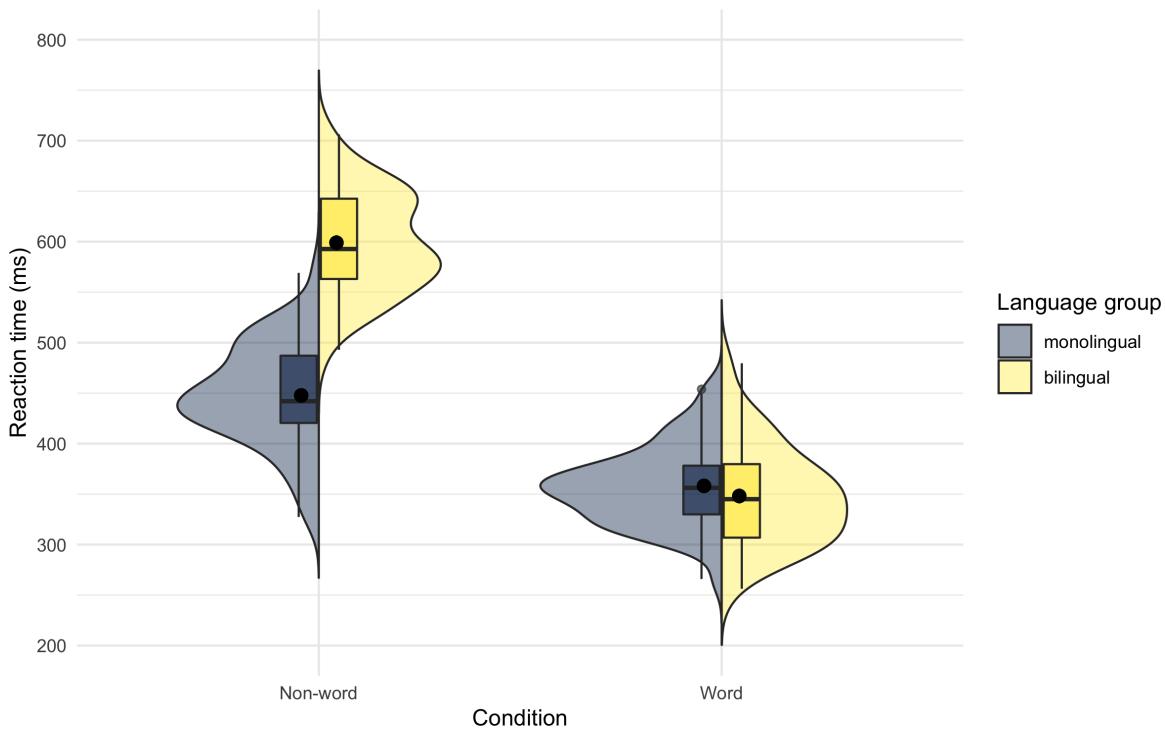


Figure 40. Split-violin plot

747 **Raincloud plots**

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

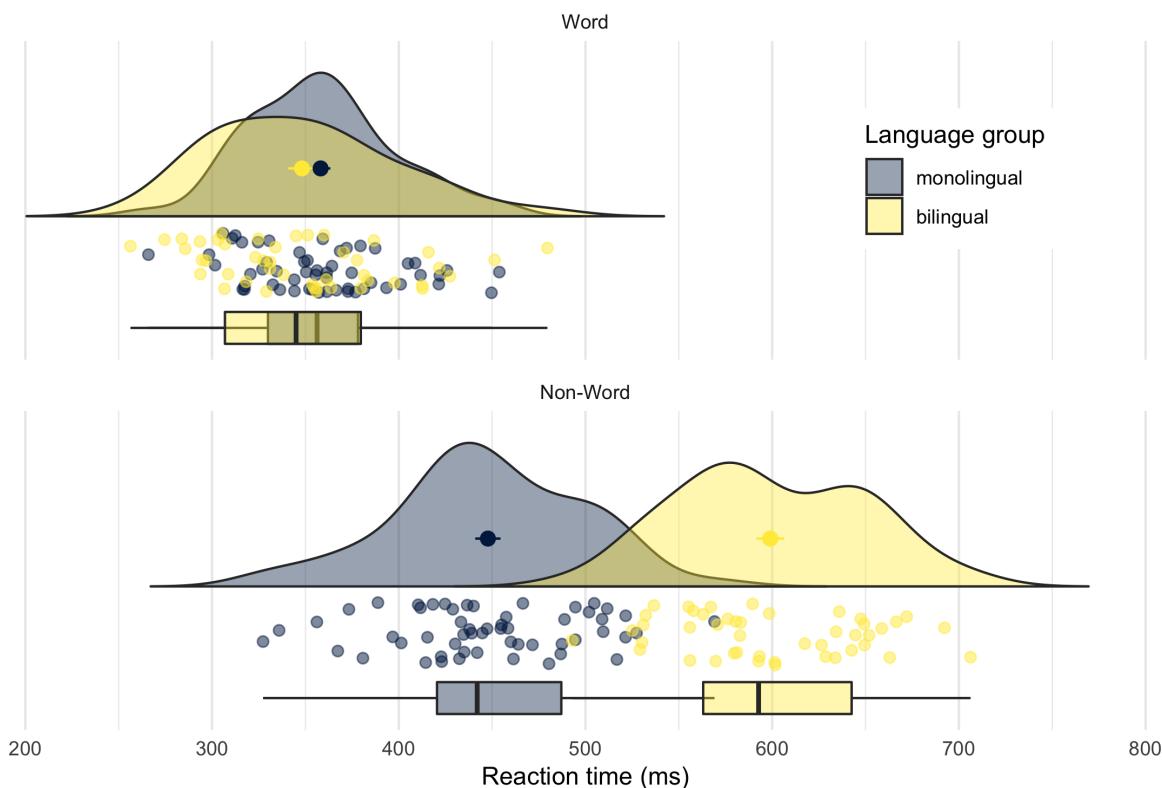


Figure 41. Raincloud plot

### 751 Ridge plots

752 Ridge plots are a series of density plots and show the distribution of numeric values for  
 753 several groups. Figure 42 shows data from (Nation, 2017) and demonstrates how effective  
 754 this type of visualisation can be to convey a lot of information very intuitively whilst being  
 755 visually attractive.

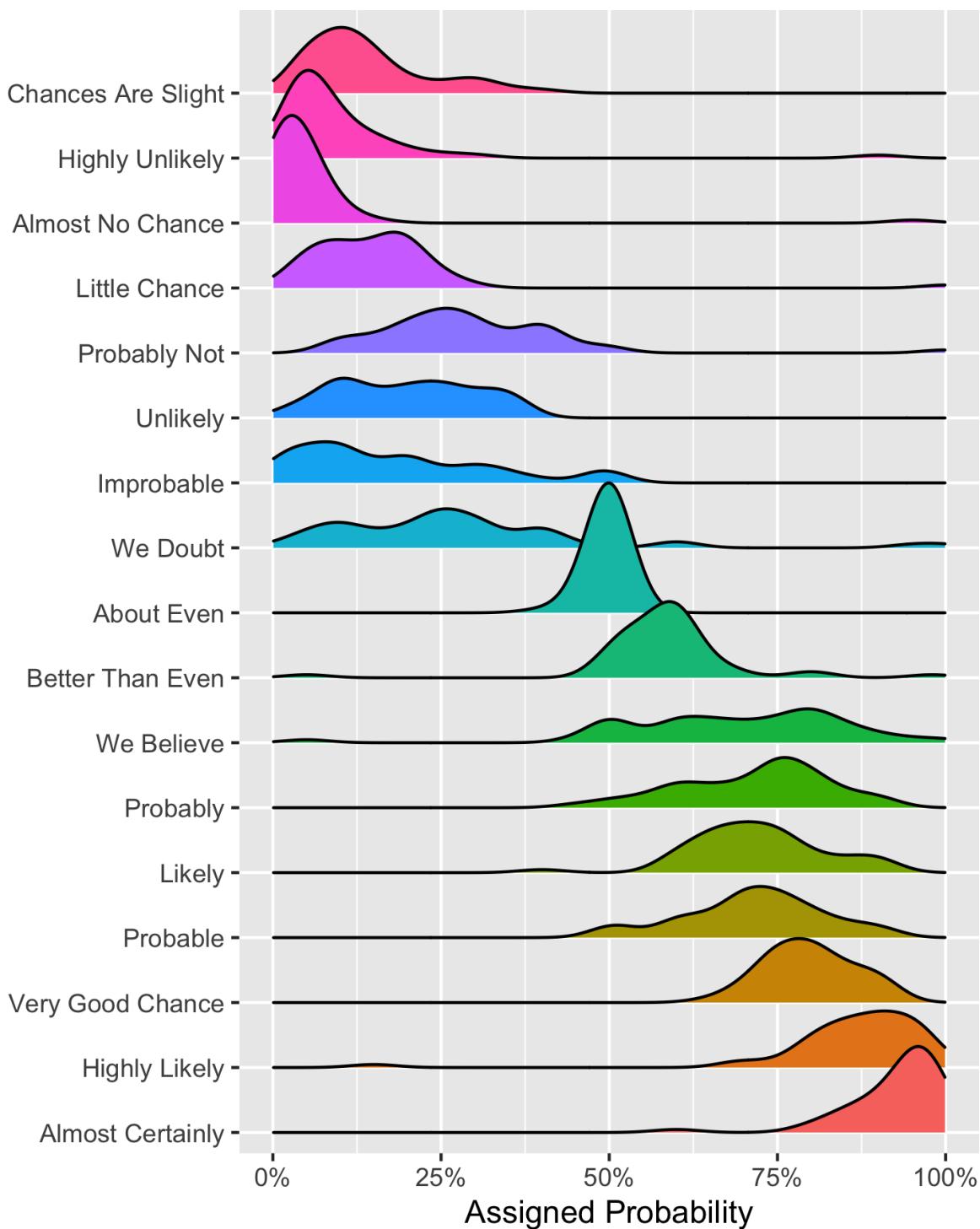


Figure 42. A ridge plot.

756 **Alluvial plots**

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

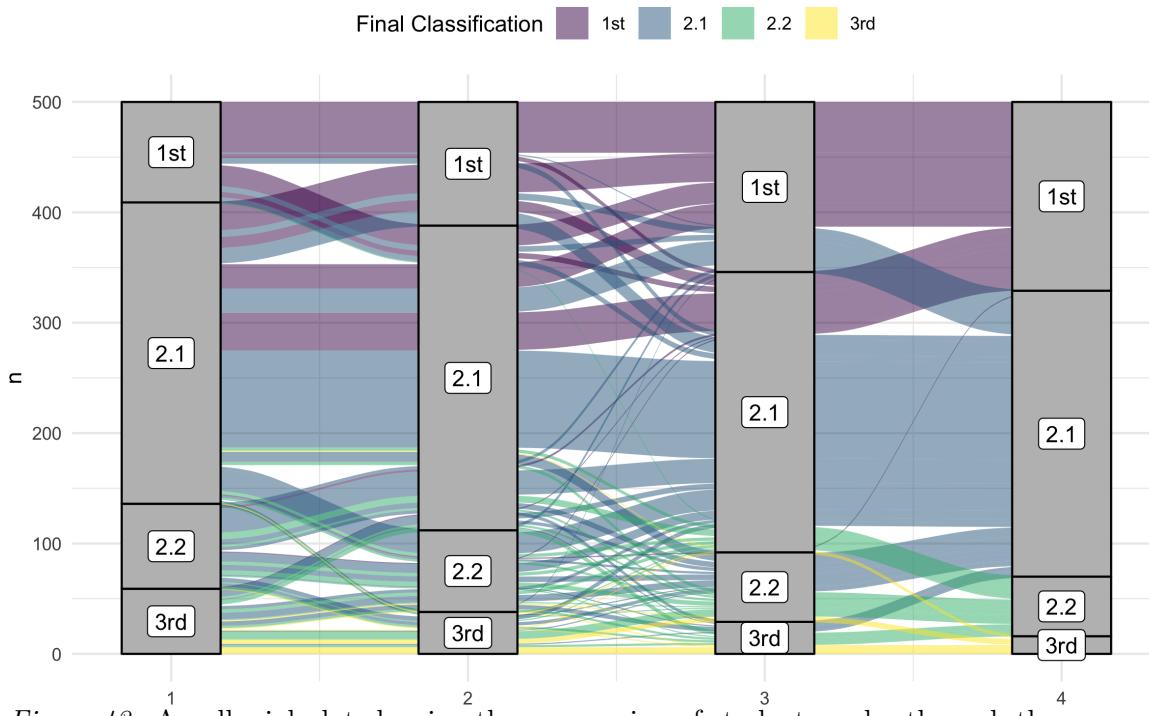


Figure 43. An alluvial plot showing the progression of student grades through the years.

759

## Conclusion

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

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

767

## Acknowledgements

768

### Author Contributions.

- 769 • EN: Conceptualization; Visualization; Writing - original draft
- 770 • PM: Visualization; Writing - original draft
- 771 • WT: Visualization; Writing - original draft
- 772 • HP: Visualization; Writing - original draft

- 773 • LD: Software; Visualization; Writing - review & editing

774      **Declaration of Conflicting Interests.** The author(s) declared that there were no  
775      conflicts of interest with respect to the authorship or the publication of this article.

776      **Funding.** LMD is supported by European Research Council grant #647910.

777      **Research Software.** This tutorial uses the following open-source research software:  
778      R Core Team (2021); Wickham et al. (2019); DeBruine (2021); Aust and Barth (2020),  
779      Wickham (2016b), Pedersen (2020), Brunson (2020), Wilke (2021).

780

## References

- 781 Allen, M., Poggiali, D., Whitaker, K., Marshall, T. R., van Langen, J., & Kievit, R.  
782 A. (2021). Raincloud plots: A multi-platform tool for robust data visualization  
783 [version 2; peer review: 2 approved]. *Wellcome Open Research*, 4. <https://doi.org/10.12688/wellcomeopenres.15191.2>
- 785 Aust, F., & Barth, M. (2020). *papaja: Create APA manuscripts with R Markdown*.  
786 Retrieved from <https://github.com/crsh/papaja>
- 787 Barrett, T. S. (2019). Six reasons to consider using r in psychological research.
- 788 Bertini, E., & Stefaner, M. (2015). Amanda cox on working with r, NYT projects,  
789 favorite data [podcast]. *Data Stories*. Retrieved from <https://datastori.es/ds-56-amanda-cox-nyt/>
- 791 Brunson, J. C. (2020). ggalluvial: Layered grammar for alluvial plots. *Journal of  
792 Open Source Software*, 5(49), 2017. <https://doi.org/10.21105/joss.02017>
- 793 DeBruine, L. (2021). *Faux: Simulation for factorial designs*. Zenodo. <https://doi.org/10.5281/zenodo.2669586>
- 795 Munafò, M. R., Nosek, B. A., Bishop, D. V., Button, K. S., Chambers, C. D., Du  
796 Sert, N. P., ... Ioannidis, J. P. (2017). A manifesto for reproducible science.  
797 *Nature Human Behaviour*, 1(1), 1–9.
- 798 Nation, Z. (2017). Perceptions. *GitHub repository*. <https://github.com/zonation/perceptions%20%20>; GitHub.
- 800 Newman, G. E., & Scholl, B. J. (2012). Bar graphs depicting averages are perceptually  
801 misinterpreted: The within-the-bar bias. *Psychonomic Bulletin & Review*, 19(4),  
802 601–607.
- 803 Pedersen, T. L. (2020). *Patchwork: The composer of plots*. Retrieved from <https://CRAN.R-project.org/package=patchwork>
- 805 R Core Team. (2021). *R: A language and environment for statistical computing*.  
806 Vienna, Austria: R Foundation for Statistical Computing. Retrieved from  
807 <https://www.R-project.org/>
- 808 Robins, A., Rountree, J., & Rountree, N. (2003). Learning and teaching programming:  
809 A review and discussion. *Computer Science Education*, 13(2), 137–172.
- 810 RStudio Team. (2021). *RStudio: Integrated development environment for r*. Boston,  
811 MA: RStudio, PBC. Retrieved from <http://www.rstudio.com/>
- 812 Visual, B., & Journalism, D. (2019). How the BBC visual and data  
813 journalism team works with graphics in r. *Medium*. Retrieved  
814 from <https://medium.com/bbc-visual-and-data-journalism/how-the-bbc-visual-and-data-journalism-team-works-with-graphics-in-r-ed0b35693535>

- 816 Wickham, H. (2010). A layered grammar of graphics. *Journal of Computational and*  
817 *Graphical Statistics*, 19(1), 3–28.
- 818 Wickham, H. (2016a). *ggplot2: Elegant graphics for data analysis*. Springer-Verlag  
819 New York. Retrieved from <https://ggplot2.tidyverse.org>
- 820 Wickham, H. (2016b). *ggplot2: Elegant graphics for data analysis*. Springer-Verlag  
821 New York. Retrieved from <https://ggplot2.tidyverse.org>
- 822 Wickham, H. (2017). *Tidyverse: Easily install and load the 'tidyverse'*. Retrieved  
823 from <https://CRAN.R-project.org/package=tidyverse>
- 824 Wickham, H., Averick, M., Bryan, J., Chang, W., McGowan, L. D., François, R., ...  
825 Yutani, H. (2019). Welcome to the tidyverse. *Journal of Open Source Software*,  
826 4(43), 1686. <https://doi.org/10.21105/joss.01686>
- 827 Wickham, H., & others. (2014). Tidy data. *Journal of Statistical Software*, 59(10),  
828 1–23.
- 829 Wilke, C. O. (2021). *Ggridges: Ridgeline plots in 'ggplot2'*. Retrieved from <https://CRAN.R-project.org/package=ggridges>
- 830 Wilkinson, L., Anand, A., & Grossman, R. (2005). Graph-theoretic scagnostics. In  
831 *IEEE symposium on information visualization (InfoVis 05)* (pp. 157–158). IEEE  
832 Computer Society.
- 833 Wills, A. (n.d.). Teaching research methods in r. *rminr*. Retrieved from <https://www.andywills.info/rminr/rminrinpsy.html>