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

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

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

Preprint

Abstract

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

Keywords: visualization, ggplot, plots, R

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

Introduction

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

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

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

Why R for data visualisation?

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

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

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

A layered grammar of graphics

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

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

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

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

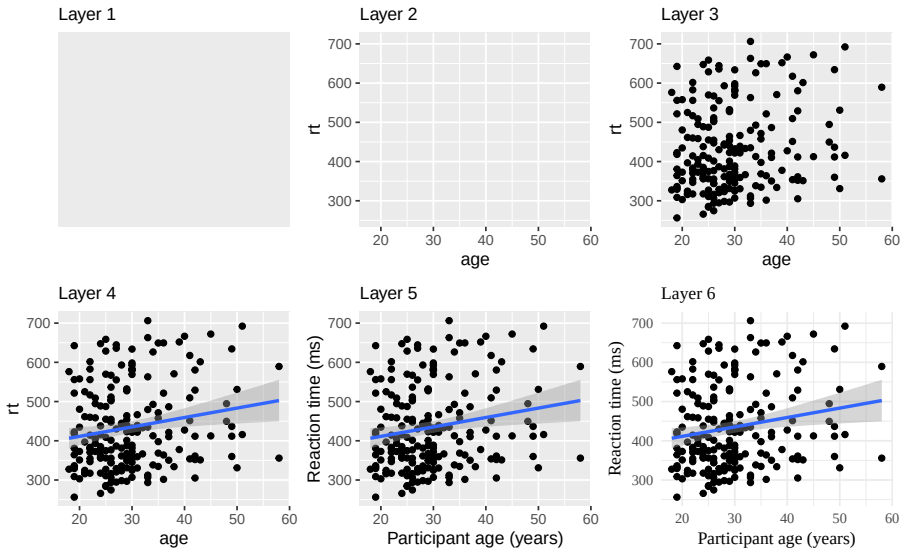


Figure 1. Evolution of a layered plot

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

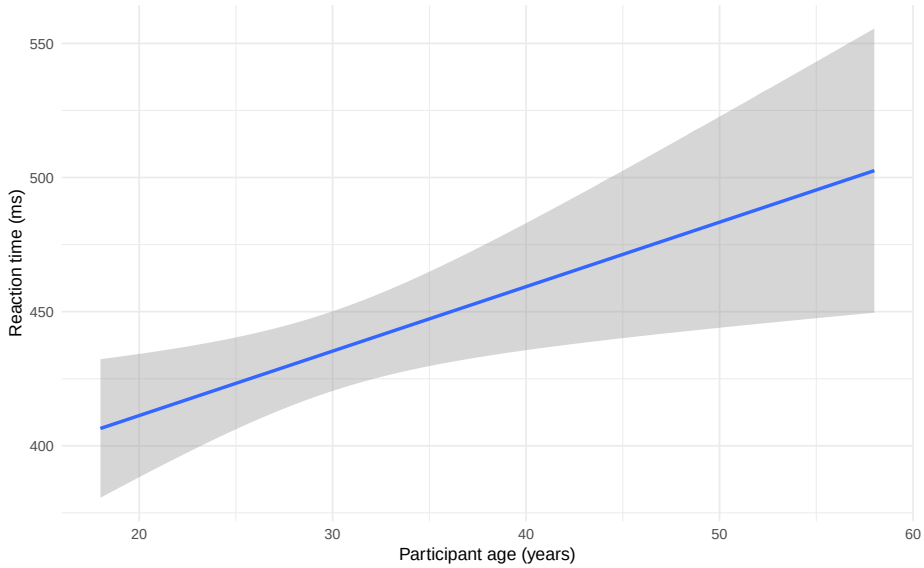


Figure 2. Plot with scatterplot layer removed.

Tutorial components

This tutorial contains three components.

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

Simulated dataset

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

- Participant information:
  - `id`: Participant ID
  - `age`: Age
- 1 between-subject independent variable (IV):
  - `language`: Language group (1 = monolingual, 2 = bilingual)
- 4 columns for the 2 dependent variables (DVs) of RT and accuracy, crossed by the within-subject IV of condition:
  - `rt_word`: Reaction time (ms) for word trials
  - `rt_nonword`: Reaction time (ms) for non-word trials
  - `acc_word`: Accuracy for word trials
  - `acc_nonword`: Accuracy for non-word trials

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

Setting up R and RStudio

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

Projects are a useful way of keeping all your code, data, and output in one place. To create a new project, open RStudio and click `File - New Project - New Directory`

- `New Project`. You will be prompted to give the project a name, and select a location for where to store the project on your computer. Once you have done this, click `Create Project`. Download the simulated dataset and code tutorial Rmd file from the online materials (`ldt_data.csv`, `workbook.Rmd`) and then move them to this folder. The files pane on the bottom right of RStudio should now display this folder and the files it contains - this is known as your *working directory* and it is where R will look for any data you wish to import and where it will save any output you create.

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

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

R Markdown is a dynamic format that allows you to combine text and code into one reproducible document. The R Markdown workbook available in the online materials contains all the code in this tutorial and there is more information and links to additional resources for how to use R Markdown for reproducible reports in the additional resources.

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

For more information on how to use R with RStudio, please see the additional resources in the online appendices.

Preparing your data

Before you start visualising your data, it must be in an appropriate format. These preparatory steps can all be dealt with reproducibly using R and the additional resources section points to extra tutorials for doing so. However, performing these types of tasks in R can require more sophisticated coding skills and the solutions and tools are dependent on the idiosyncrasies of each dataset. For this reason, in this tutorial we encourage the reader to complete data preparation steps using the method they are most comfortable with and to focus on the aim of data visualisation.

Data format. The simulated lexical decision data is provided in a `csv` (comma-separated variable) file. Functions exist in R to read many other types of data files; the `rio` package's `import()` function can read most types of files. However, `csv` files avoids problems like Excel's insistence on mangling anything that even vaguely resembles a date. You may wish to export your data as a `csv` file that contains only the data you want to visualise, rather than a full, larger workbook. It is possible to clean almost any file reproducibly in R, however, as noted above, this can require higher level coding skills. For getting started with visualisation, we suggest removing summary rows or additional notes

from any files you import so the file only contains the rows and columns of data you want to plot.

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

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

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

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

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

Getting Started

Loading packages

To load the packages that have the functions we need, use the `library()` function. Whilst you only need to install packages once, you need to load any packages you want to use with `library()` every time you start R or start a new session. When you load the `tidyverse`, you actually load several separate packages that are all part of the same

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

collection and have been designed to work together. R will produce a message that tells you the names of the packages that have been loaded.

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

Loading data

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

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

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

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

Handling numeric factors

Another useful check is to use the functions `summary()` and `str()` (structure) to check what kind of data R thinks is in each column. Run the below code and look at the output 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 <- mutate(dat, 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
))
```

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

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

Argument names

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

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

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

Summarising data

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

- Start with the dataset `dat` *and then;*
- Group it by the variable `language` *and then;*
- Count the number of observations in each group *and then;*
- Remove the grouping

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

language	n
monolingual	55
bilingual	45

`group_by()` does not result in surface level changes to the dataset, rather, it changes the underlying structure so that if groups are specified, whatever functions called next are performed separately on each level of the grouping variable. This grouping remains in the object that is created so it is important to remove it with `ungroup()` to avoid future operations on the object unknowingly being performed by groups.

The above code therefore counts the number of observations in each group of the variable `language`. If you just need the total number of observations, you could remove the `group_by()` and `ungroup()` lines, which would perform the operation on the whole dataset, rather than by groups:

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

n
100

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

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

mean_age	sd_age	n_values
29.75	8.28	100

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

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

Finally, the `group_by()` function will work in the same way when calculating summary statistics – the output of the function that is called after `group_by()` will be produced for each level of the grouping variable.

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

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

Bar chart of counts

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

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

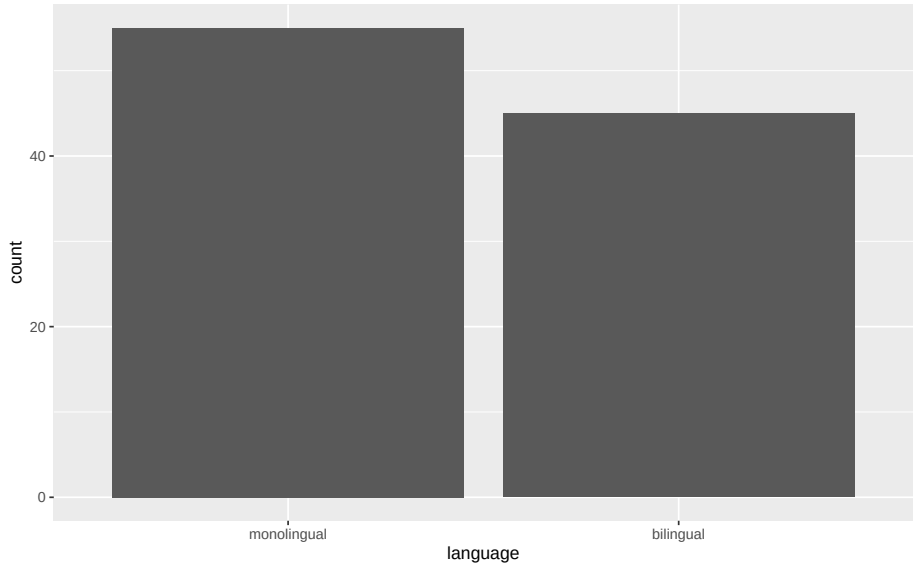


Figure 3. Bar chart of counts.

```
ggplot(data = dat, mapping = aes(x = language)) +
  geom_bar(aes(y = (..count..)/sum(..count..))) +
  scale_y_continuous(name = "Percent", labels=scales::percent)
```

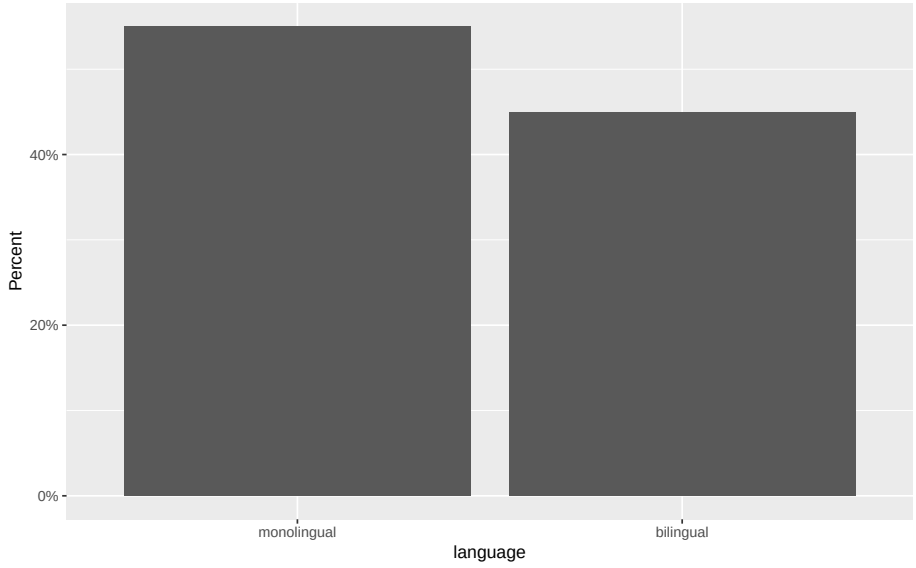


Figure 4

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

- `data` specifies which data source to use for the plot

- `mapping` specifies which variables to map to which aesthetics (`aes`) of the plot. Mappings describe how variables in the data are mapped to visual properties (aesthetics) of geoms.
- `x` specifies which variable to put on the x-axis

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

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

Aggregates and percentages

If your dataset already has the counts that you want to plot, you can set `stat="identity"` inside of `geom_bar()` to use that number instead of counting rows. For example, to plot percentages rather than counts within `ggplot2`, you can calculate these and store them in a new object that is then used as the dataset. You can do this in the software you are most comfortable in, save the new data, and import it as a new table, or you can use code to manipulate the data.

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

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

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

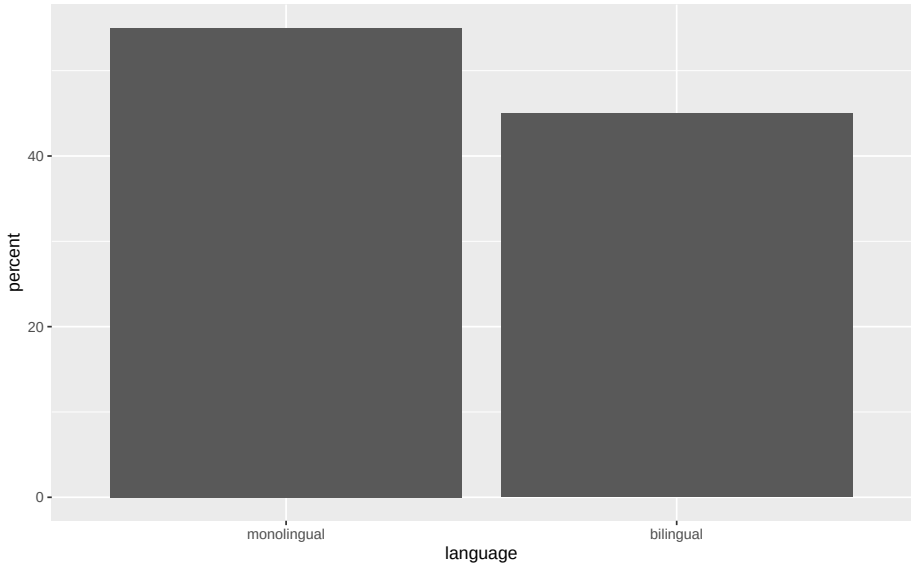


Figure 5. Bar chart of pre-calculated counts.

Histogram

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

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

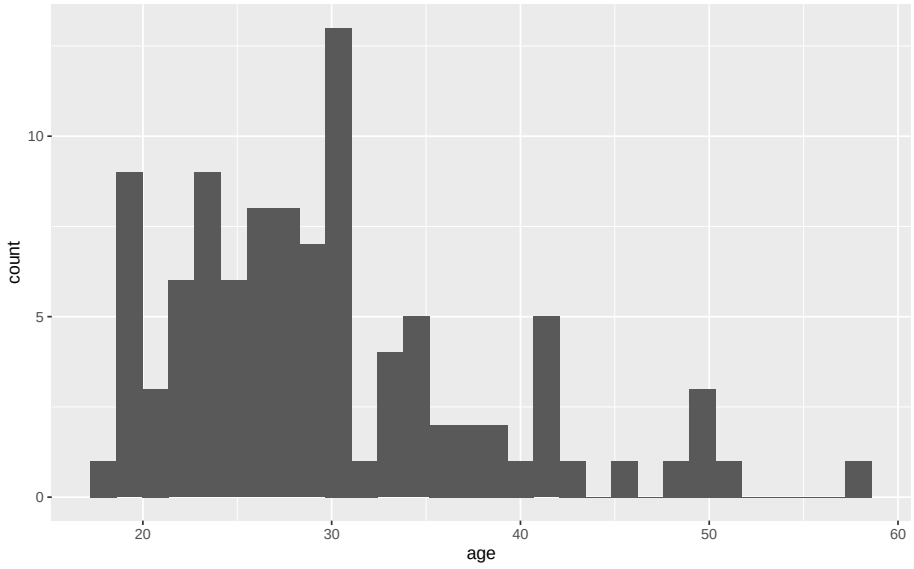


Figure 6. Histogram of ages.

The base statistic for `geom_histogram()` is also count, and by default `geom_histogram()` divides the x-axis into 30 "bins" and counts how many observations are in each bin and so the y-axis does not need to be specified. When you run the code to produce the histogram, you will get the message "stat_bin() using bins = 30. Pick better value with binwidth". You can change this by either setting the number of bins (e.g., `bins = 20`) or the width of each bin (e.g., `binwidth = 5`) as an argument.

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

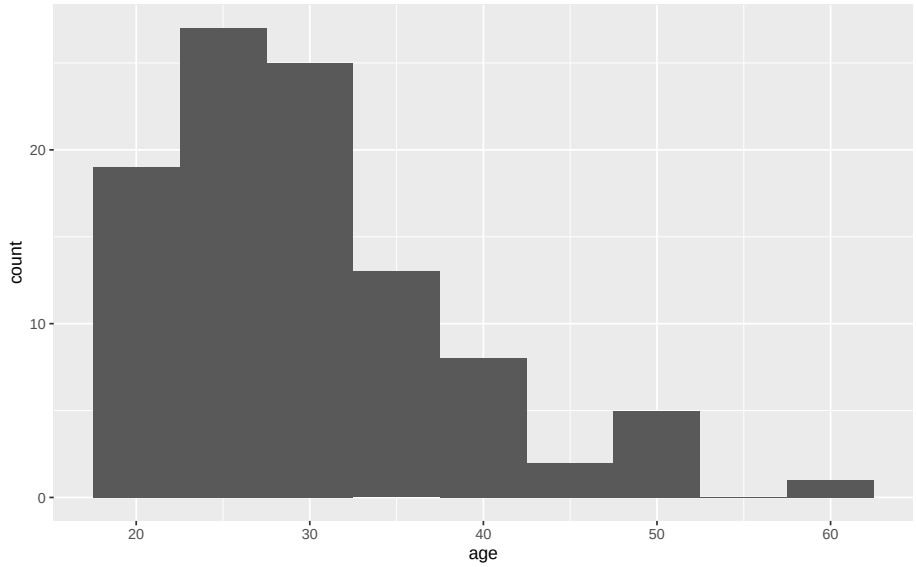


Figure 7. Histogram of ages where each bin covers five years.

Customisation 1

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

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

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

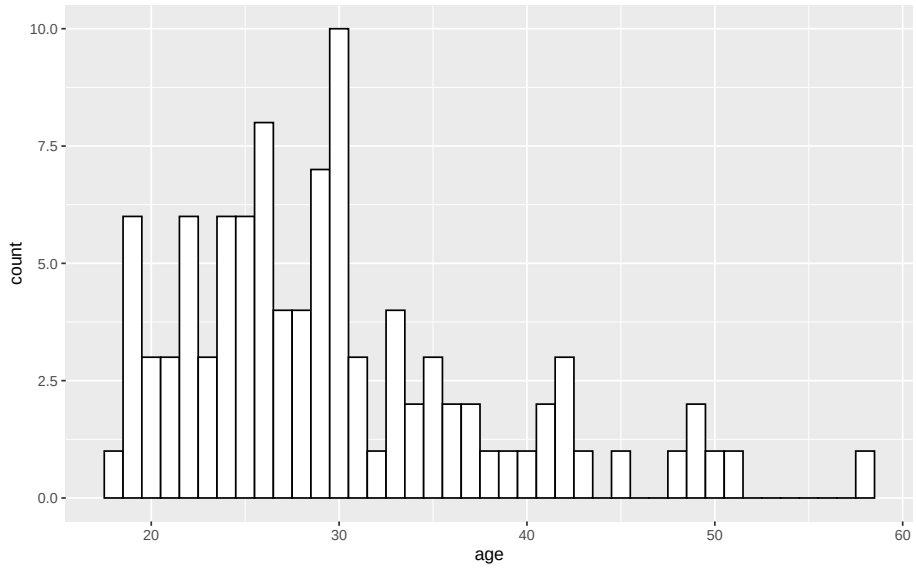


Figure 8. Histogram with custom colors for bar fill and line colors.

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

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

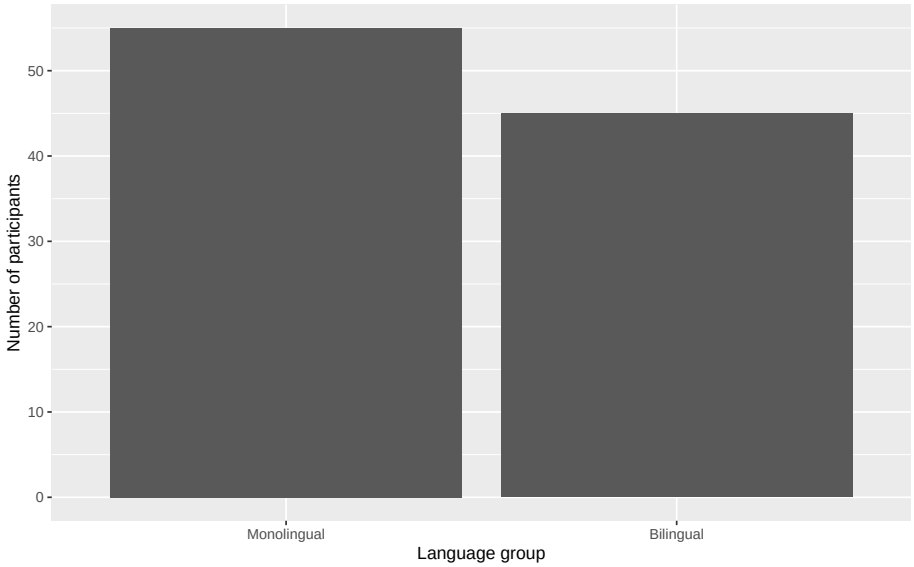


Figure 9. Bar chart with custom axis labels.

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

A common error is to map the wrong 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"))
```

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

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

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

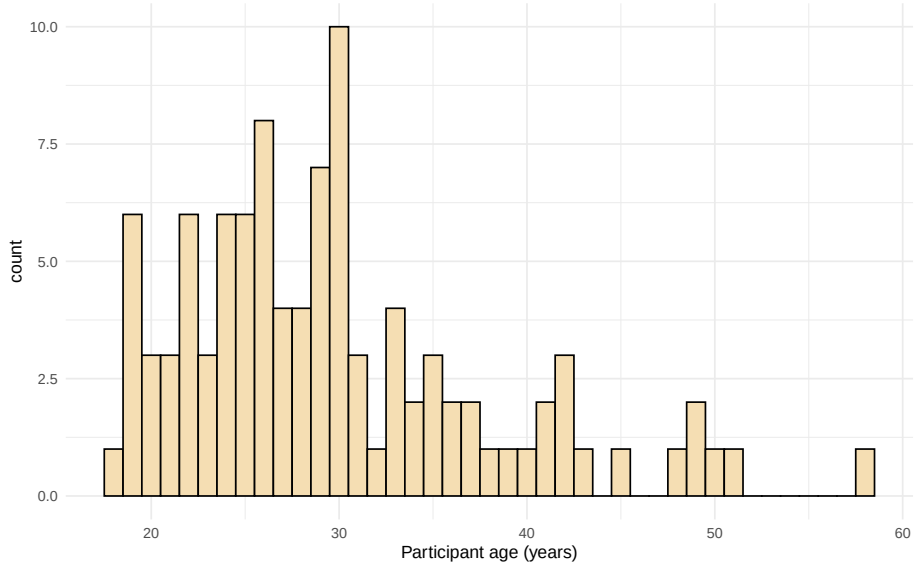


Figure 10. Histogram with a custom theme.

You can set the theme globally so that all subsequent plots use a theme. `theme_set()` is not part of a `ggplot()` object, you should run this code on its own. It may be useful to add this code to the top of your script so that all plots produced subsequently use the same theme.

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

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

Activities 1

Before you move on try the following:

Table 1
Data in wide format.

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

1. Add a layer that edits the **name** of the y-axis histogram label to `Number of participants`.
2. Change the colour of the bars in the bar chart to red.
3. Remove `theme_minimal()` from the histogram and instead apply one of the other available themes. To find out about other available themes, start typing `theme_` and the auto-complete will show you the available options - this will only work if you have loaded the `tidyverse` library with `library(tidyverse)`.

Transforming Data

Data formats

To visualise the experimental reaction time and accuracy data using `ggplot2`, we first need to reshape the data from wide format to long format. This step 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. For repeated-measures variables, the dependent variable is split across different columns. For 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 [4] reaction time and accuracy for each level of the within-subject variable is split across multiple columns for the repeated factor of condition (words versus non-words).

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

[4] In this tutorial we have chosen to gloss over the data processing steps that must occur to get from the raw data to aggregated values. This type of processing requires a more extensive tutorial than we can provide in the current paper. More importantly, it is still possible to use R for data visualisation having done the preparatory steps using existing workflows in Excel and SPSS. 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

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 IVs so that each column represents only one variable. The less intuitive part is that long-format data has multiple rows for each participant (one row for each observation) and a column that encodes the level of the IV (`word` or `nonword`). Wickham (2014) provides a comprehensive overview of the benefits of a similar format known as tidy data, which is a standard way of mapping a dataset to its structure. For the purposes of this tutorial there are two important rules: each column should be a *variable* and each row should be an *observation*.

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

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

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

Wide to long format

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

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

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

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

- As with the other tidyverse functions, the first argument specifies the dataset to use as the base, in this case `dat`. This argument name is often dropped in examples.
- `cols` specifies all the columns you want to transform. The easiest way to visualise this is to think about which columns would be the same in the new long-form dataset and which will change. If you refer back to Table 1, you can see that `id`, `age`, and `language` all remain, while the columns that contain the measurements of the DVs change. The colon notation `first_column:last_column` is used to select all variables from the first column specified to the last In our code, `cols` specifies that the columns we want to transform are `rt_word` to `acc_nonword`.
- `names_to` specifies the name of the new column that will be created. This column will contain the names of the selected existing columns.
- Finally, `values_to` names the new column that will contain the values in the selected columns. In this case we'll call it `dv`.

At this point you may find it helpful to go back and compare `dat` and `long` again to see how each argument matches up with the output of the table.

Table 3
Data in long format with mixed DVs.

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

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

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

Step 2: `pivot_longer()` adjusted. The problem with the above long-format dataset is that `dv_condition` combines two variables - it has information about the type of DV and the condition of the IV. To account for this, we include a new argument `names_sep` and adjust `name_to` to specify the creation of two new columns. Note that we are pivoting the same wide-format dataset `dat` as we did in step 1.

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

- `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.

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 the two DVs are different types of data, there is an additional bit of wrangling required to get the data in the right format for plotting.

In the current long-format dataset, the column `dv` contains both reaction time and accuracy measures. 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` is the reverse of `names_to` from `pivot_longer()`. It will take the values from the variable specified and use these as the new column names. In this case, the values of `rt` and `acc` that are currently in the `dv_type` column will become the new column names.
- `values_from` is the reverse of `values_to` from `pivot_longer()`. It specifies the column that contains the values to fill the new columns with. In this case, the new columns `rt` and `acc` will be filled with the values that were in `dv`.

Again, it can be helpful to compare each dataset with the code to see how it aligns. This final long-form data should look like Table 2.

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

Histogram 2

Now that we have the experimental data in the right form, we can begin to create some useful visualizations. First, to demonstrate how code recipes can be reused and adapted,

we will create histograms of reaction time and accuracy. The below code uses the same template as before but changes the dataset (`dat_long`), the bin-widths of the histograms, 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)")
```

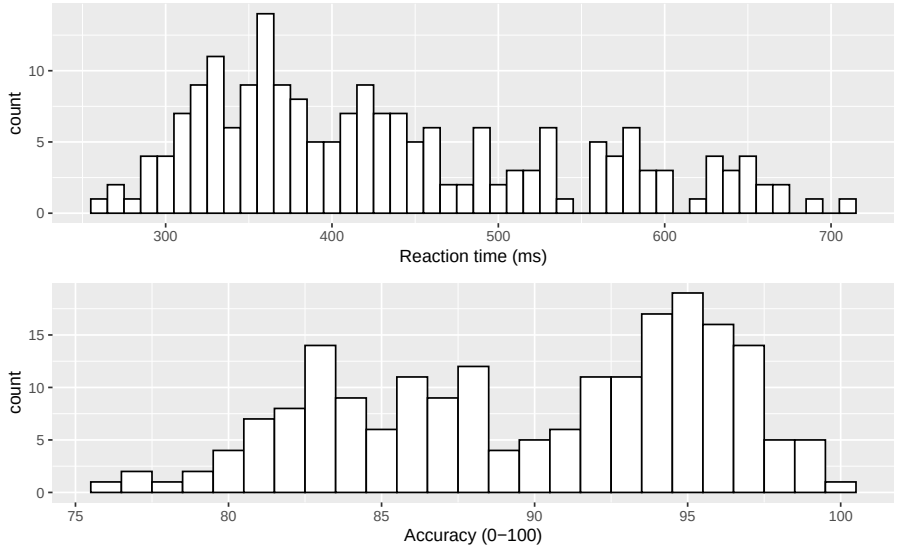


Figure 11. Histograms showing the distribution of reaction time (top) and accuracy (bottom)

Density plots

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

```
ggplot(dat_long, aes(x = rt)) +
  geom_density()+
  scale_x_continuous(name = "Reaction time (ms)")
```

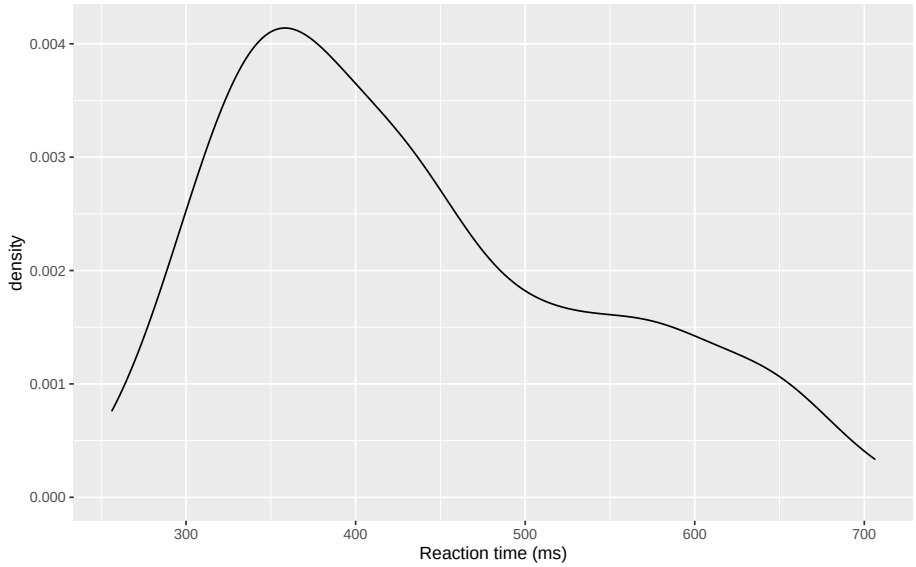


Figure 12. Density plot of reaction time.

Grouped density plots. Density plots are most useful for comparing the distributions of different groups of data. Because the dataset is now in long format, with each variable contained within a single column, we can map `condition` to the plot.

- In addition to mapping `rt` to the x-axis, we specify the `fill` aesthetic to fill the visualisation so that each level of the `condition` variable is represented by a different colour.
- Because the density plots are overlapping, we set `alpha = 0.75` to make the geoms 75% transparent.
- As with the x and y-axis scale functions, we can edit the names and labels of our fill aesthetic by adding on another `scale_*` layer (`scale_fill_discrete()`)[6].
- Note that the `fill` here is set inside the `aes()` function, which tells ggplot to set the fill differently for each value in the `condition` column. You cannot specify which colour here (e.g., `fill="red"`), like you could when you set `fill` inside the `geom_*()` function before.

```
ggplot(dat_long, aes(x = rt, fill = condition)) +
  geom_density(alpha = 0.75)+
  scale_x_continuous(name = "Reaction time (ms)") +
  scale_fill_discrete(name = "Condition",
                      labels = c("Non-Word", "Word"))
```

[6]Please note that the code and figure for this plot has been corrected from the published paper due to the labels "Word" and "Non-word" being incorrectly reversed. This is of course mortifying as authors, although it does provide a useful teachable moment that R will do what you tell it to do, no more, no less, regardless of whether what you tell it to do is wrong.

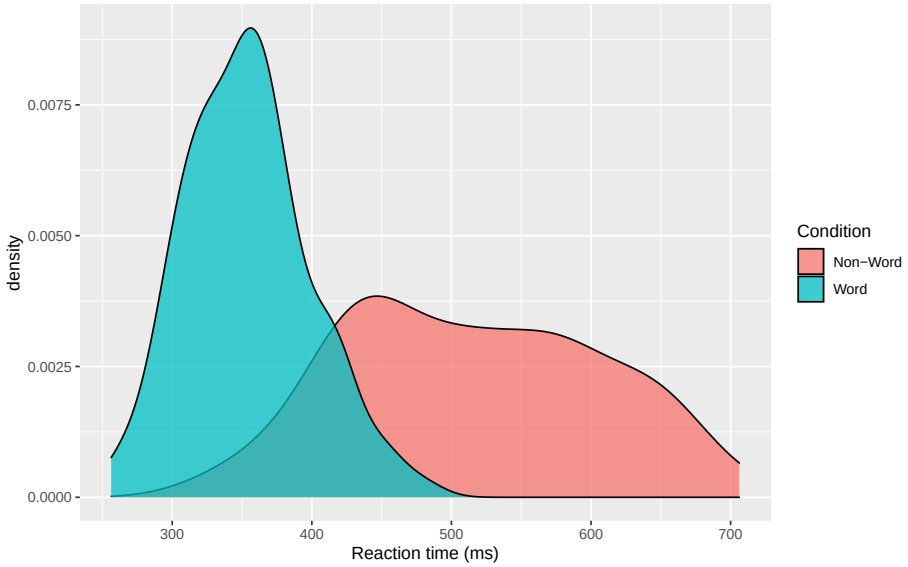


Figure 13. Density plot of reaction times grouped by condition.

Scatterplots

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

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

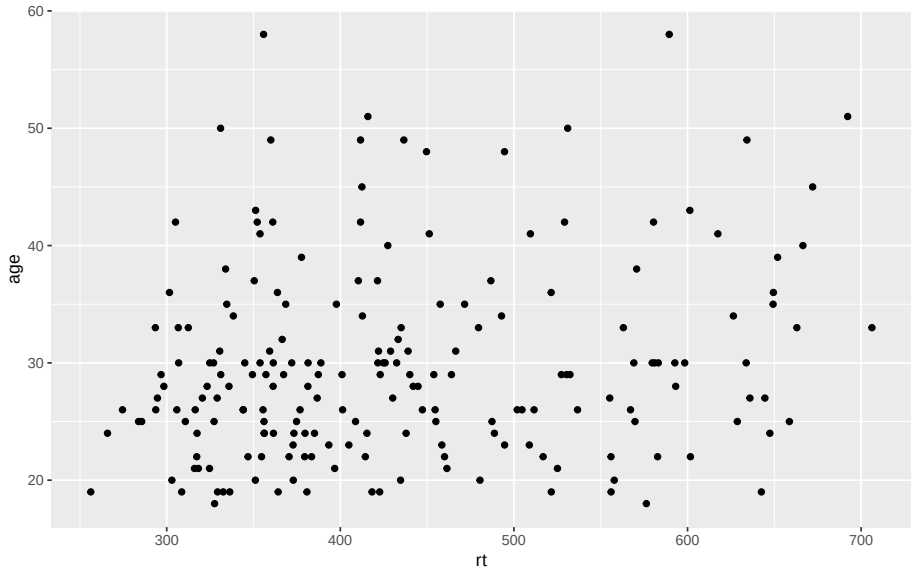


Figure 14. Scatterplot of reaction time versus age.

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

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

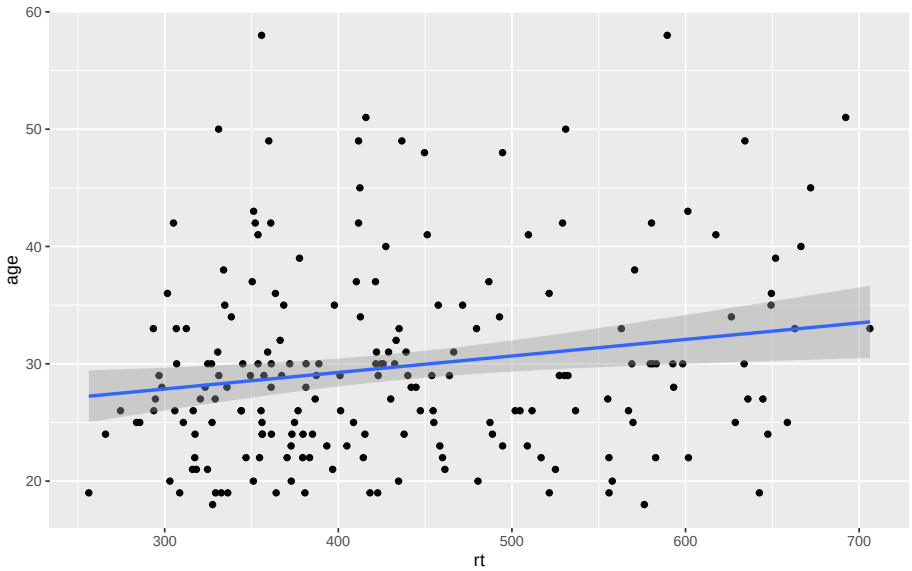


Figure 15. Line of best fit for reaction time versus age.

Grouped scatterplots. Similar to the density plot, the scatterplot can also be easily adjusted to display grouped data. For `geom_point()`, the grouping variable is mapped to `colour` rather than `fill` and the relevant `scale_*` function is added[7].

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

[7]Please note that the code and figure for this plot has been corrected from the published paper due to the labels "Word" and "Non-word" being incorrectly reversed. This is of course mortifying as authors, although it does provide a useful teachable moment that R will do what you tell it to do, no more, no less, regardless of whether what you tell it to do is wrong.

Figure 16. Grouped scatterplot of reaction time versus age by condition.

Long to wide format

Following the rule that *anything that shares an axis should probably be in the same column* means that we will frequently need our data in long-form when using `ggplot2`, However, there are some cases when wide format is necessary. For example, we may wish to visualise the relationship between reaction time in the word and non-word conditions. This requires that the corresponding word and non-word values for each participant be in the same row. The easiest way to achieve this in our case would simply be to use the original wide-format data as the input:

```
ggplot(dat, aes(x = rt_word, y = rt_nonword, colour = language)) +
  geom_point() +
  geom_smooth(method = "lm")
```

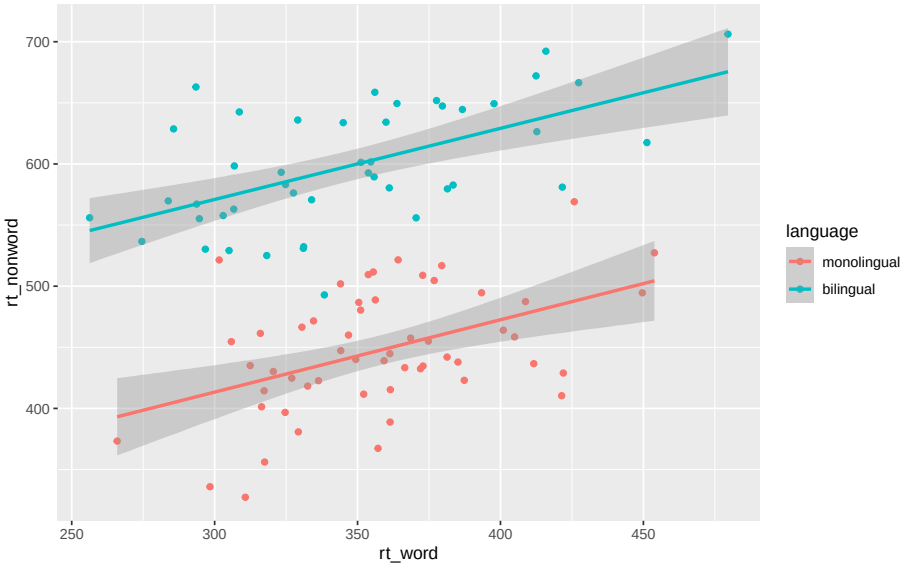


Figure 17. Scatterplot with data grouped by language group

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

Customisation 2

Accessible colour schemes. One of the drawbacks of using `ggplot2` for visualisation is that the default colour scheme is not accessible (or visually appealing). The red and green default palette is difficult for colour-blind people to differentiate, and also does not display well in greyscale. You can specify exact custom colours for your plots, but one easy option is to use a custom colour palette. These take the same arguments as their default `scale` sister functions for updating axis names and labels, but display plots in contrasting colours that can be read by colour-blind people and that also print well in grey scale. For categorical colours, the "Set2", "Dark2" and "Paired" palettes from the `brewer` scale

functions are colourblind-safe (but are hard to distinhuish in greyscale). For continuous colours, such as when colour is representing the magnitude of a correlation in a tile plot, the `viridis` scale functions provide a number of different colourblind and greyscale-safe options[8].

```
ggplot(dat_long, aes(x = rt, y = age, colour = condition)) +
  geom_point() +
  geom_smooth(method = "lm") +
  scale_color_brewer(palette = "Dark2",
                     name = "Condition",
                     labels = c("Non-word", "Word"))
```

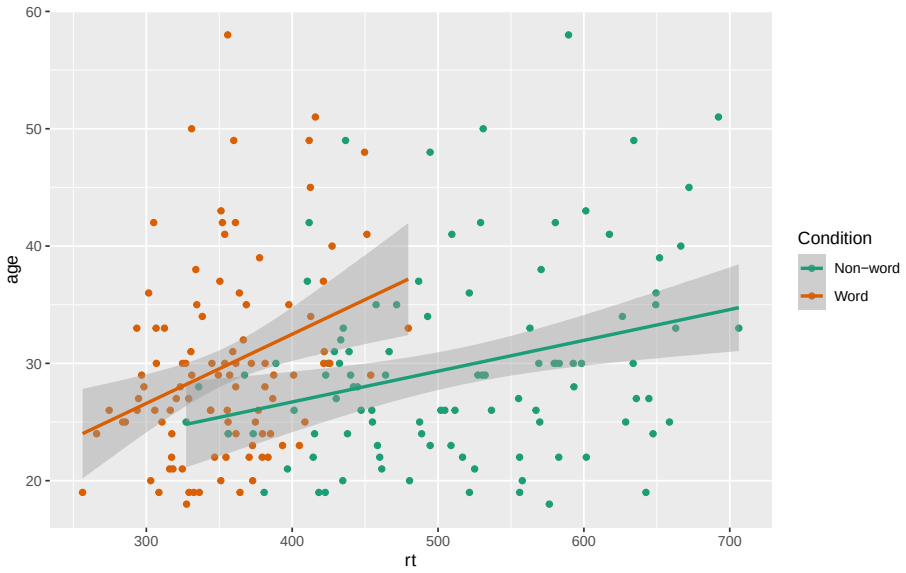


Figure 18. Use the Dark2 brewer colour scheme for accessibility.

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

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

However, this is somewhat inefficient. Instead, we can use the function `seq()` (short

[8]Please note that the code and figure for this plot has been corrected from the published paper due to the labels "Word" and "Non-word" being incorrectly reversed. This is of course mortifying as authors, although it does provide a useful teachable moment that R will do what you tell it to do, no more, no less, regardless of whether what you tell it to do is wrong.

for sequence) to specify the first and last value and the increments `by` which the breaks should display between these two values.

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

Activities 2

Before you move on try the following:

1. Use `fill` to created grouped histograms that display the distributions for `rt` for each `language` group separately and also edit the fill axis labels. Try adding `position = "dodge"` to `geom_histogram()` to see what happens.
2. Use `scale_*` functions to edit the name of the x and y-axis on the scatterplot
3. Use `se = FALSE` to remove the confidence envelope from the scatterplots
4. Remove `method = "lm"` from `geom_smooth()` to produce a curved fit line.
5. Replace the default fill on the grouped density plot with a colour-blind friendly version.

Representing Summary Statistics

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

Boxplots

As with `geom_point()`, boxplots also require an x- and y-variable to be specified. In this case, `x` must be a discrete, or categorical variable[9], whilst `y` must be continuous.

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

[9] If the data in the x-axis column is numeric (e.g., 1 and 2 for the condition), you will see one boxplot and the message "Continuous x aesthetic – did you forget aes(group=...)?" You can fix this by converting the column to a factor like this: `x = factor(condition)`.

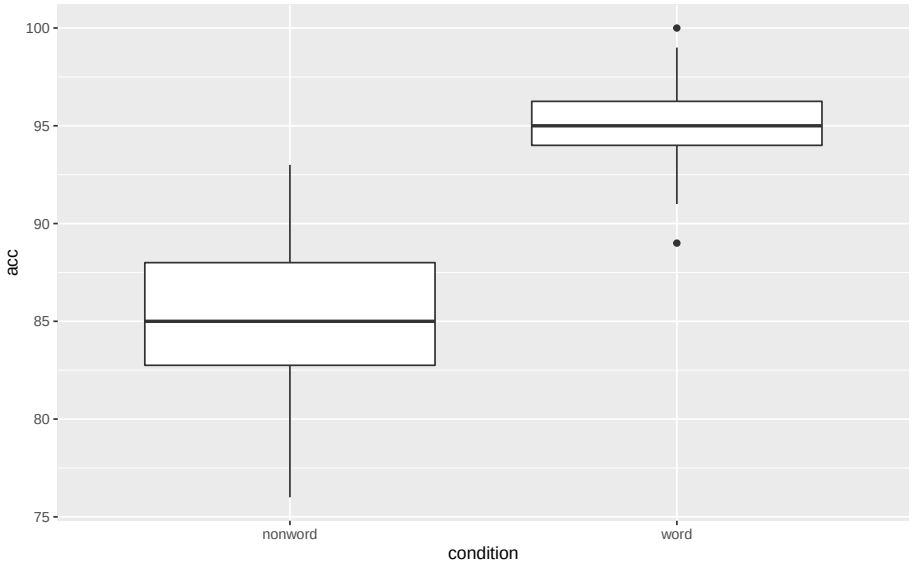


Figure 19. Basic boxplot.

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[10].

```
ggplot(dat_long, aes(x = condition, y = acc, fill = language)) +
  geom_boxplot() +
  scale_fill_brewer(palette = "Dark2",
                    name = "Group",
                    labels = c("Bilingual", "Monolingual")) +
  theme_classic() +
  scale_x_discrete(name = "Condition",
                   labels = c("Non-Word", "Word")) +
  scale_y_continuous(name = "Accuracy")
```

[10]Please note that the code and figure for this plot has been corrected from the published paper due to the labels "Word" and "Non-word" being incorrectly reversed. This is of course mortifying as authors, although it does provide a useful teachable moment that R will do what you tell it to do, no more, no less, regardless of whether what you tell it to do is wrong.

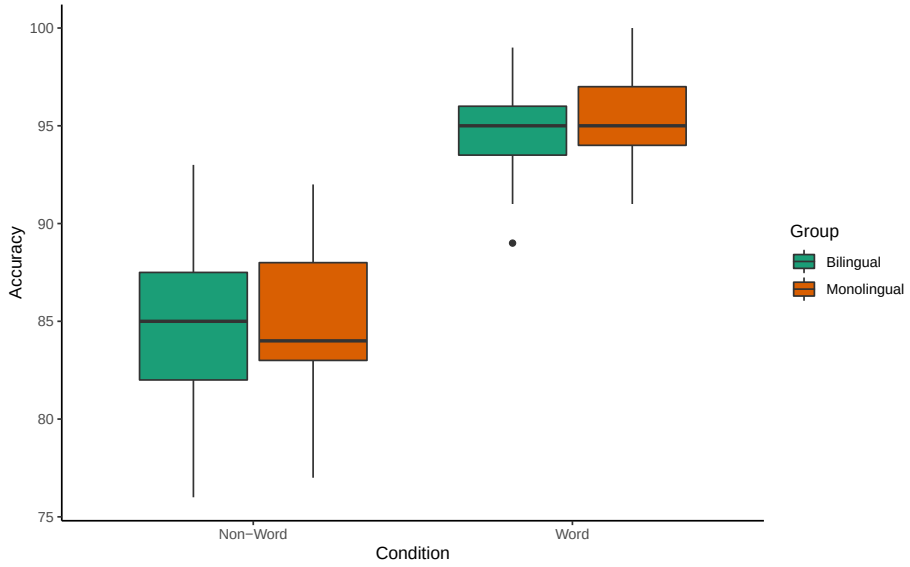


Figure 20. Grouped boxplots

Violin plots

Violin plots display the distribution of a dataset and can be created by calling `geom_violin()`. They are so-called because the shape they make sometimes looks something like a violin. They are essentially sideways, mirrored density plots. Note that the below code is identical to the code used to draw the boxplots above, except for the call to `geom_violin()` rather than `geom_boxplot()`[^ch4-3].

```
ggplot(dat_long, aes(x = condition, y = acc, fill = language)) +
  geom_violin() +
  scale_fill_brewer(palette = "Dark2",
                    name = "Group",
                    labels = c("Bilingual", "Monolingual")) +
  theme_classic() +
  scale_x_discrete(name = "Condition",
                   labels = c("Non-word", "Word")) +
  scale_y_continuous(name = "Accuracy")
```

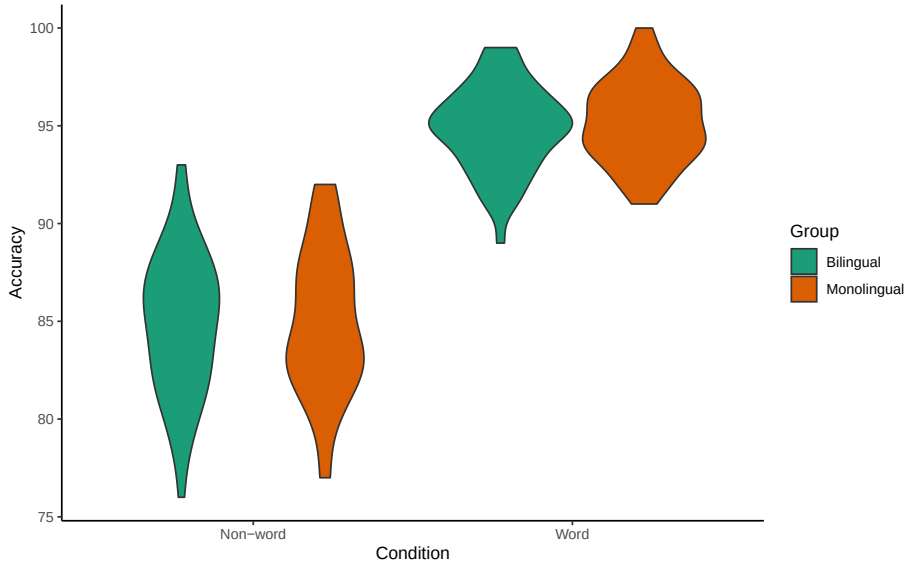


Figure 21. Violin plot.

Bar chart of means

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

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

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

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

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

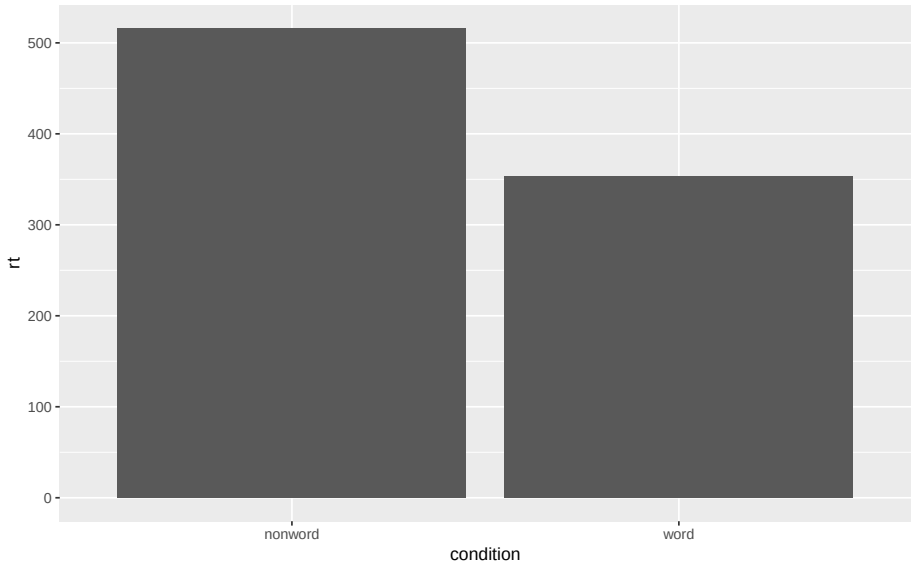


Figure 22. Bar plot of means.

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

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

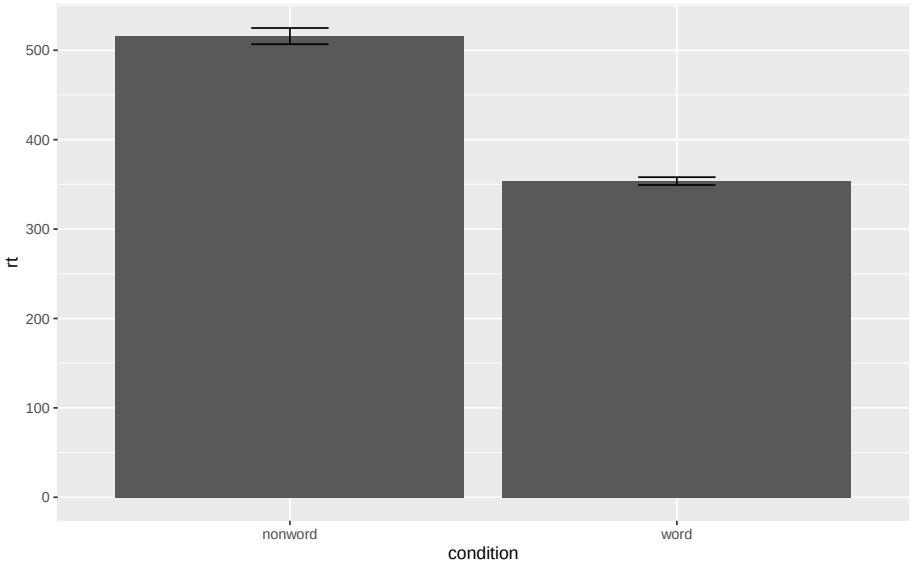


Figure 23. Bar plot of means with error bars representing SE.

Violin-boxplot

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

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

```
ggplot(dat_long, aes(x = condition, y= rt)) +
  geom_violin() +
  # remove the median line with fatten = NULL
  geom_boxplot(width = .2,
               fatten = NULL) +
  stat_summary(fun = "mean", geom = "point") +
  stat_summary(fun.data = "mean_se",
               geom = "errorbar",
               width = .1)
```

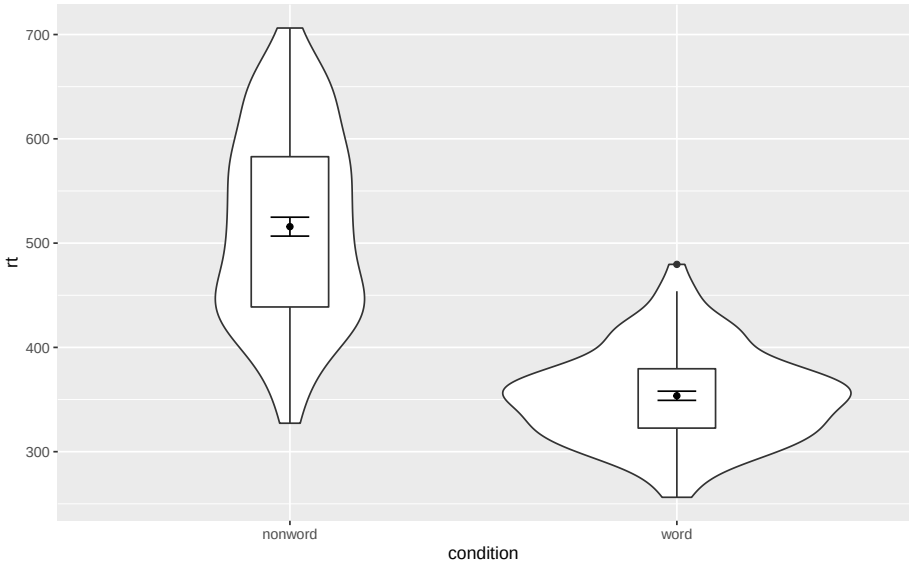


Figure 24. Violin-boxplot with mean dot and standard error bars.

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

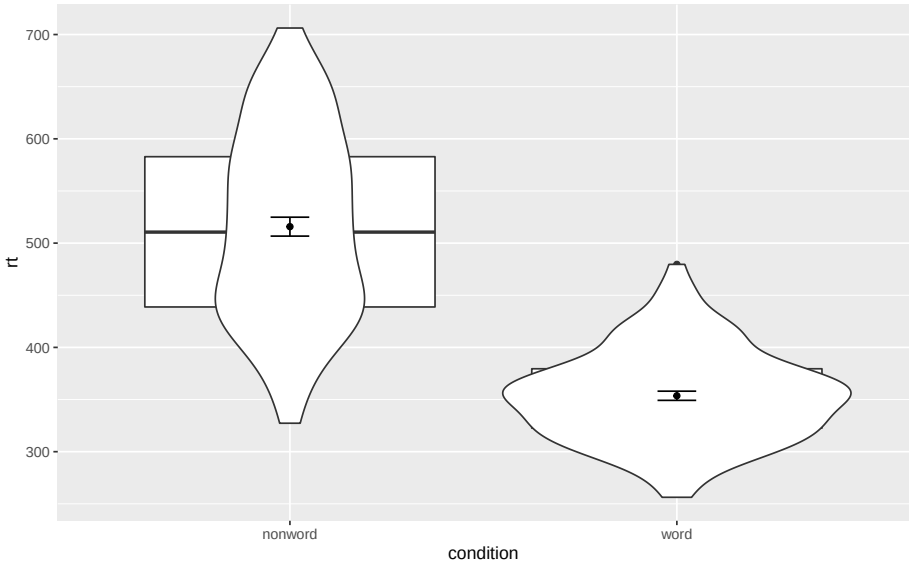


Figure 25. Plot with the geoms in the wrong order.

Grouped violin-boxplots. As with previous plots, another variable can be mapped to `fill` for the violin-boxplot. (Remember to add a colourblind-safe palette.) However, simply adding `fill` to the mapping causes the different components of the plot to become misaligned because they have different default positions:

```
ggplot(dat_long, aes(x = condition, y= rt, fill = language)) +
  geom_violin() +
  geom_boxplot(width = .2,
               fatten = NULL) +
  stat_summary(fun = "mean",  geom = "point") +
  stat_summary(fun.data = "mean_se",
               geom = "errorbar",
               width = .1) +
  scale_fill_brewer(palette = "Dark2")
```

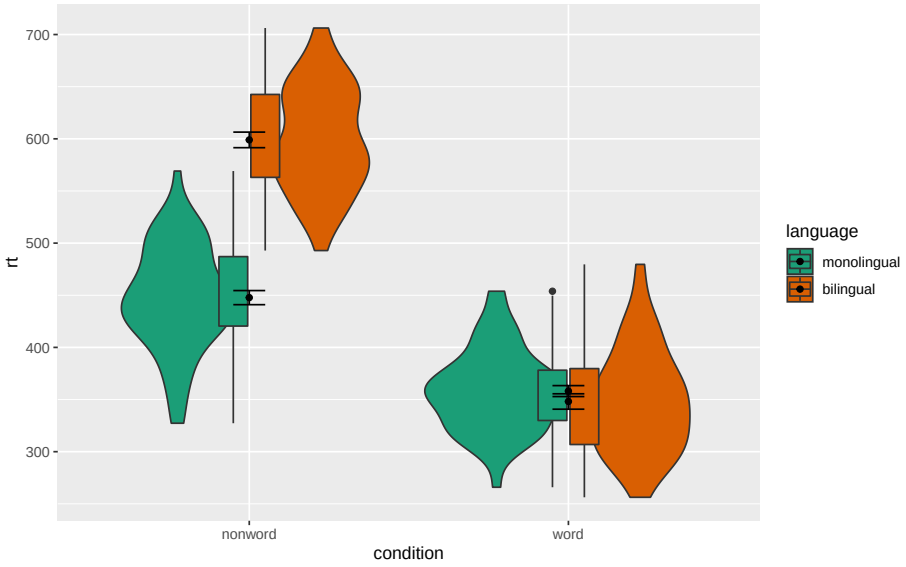


Figure 26. Grouped violin-boxplots without repositioning.

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

```
# set the offset position of the geoms
pos <- position_dodge(0.9)

ggplot(dat_long, aes(x = condition, y= rt, fill = language)) +
  geom_violin(position = pos) +
  geom_boxplot(width = .2,
               fatten = NULL,
               position = pos) +
  stat_summary(fun = "mean",
               geom = "point",
               position = pos) +
  stat_summary(fun.data = "mean_se",
               geom = "errorbar",
               width = .1,
               position = pos) +
  scale_fill_brewer(palette = "Dark2")
```

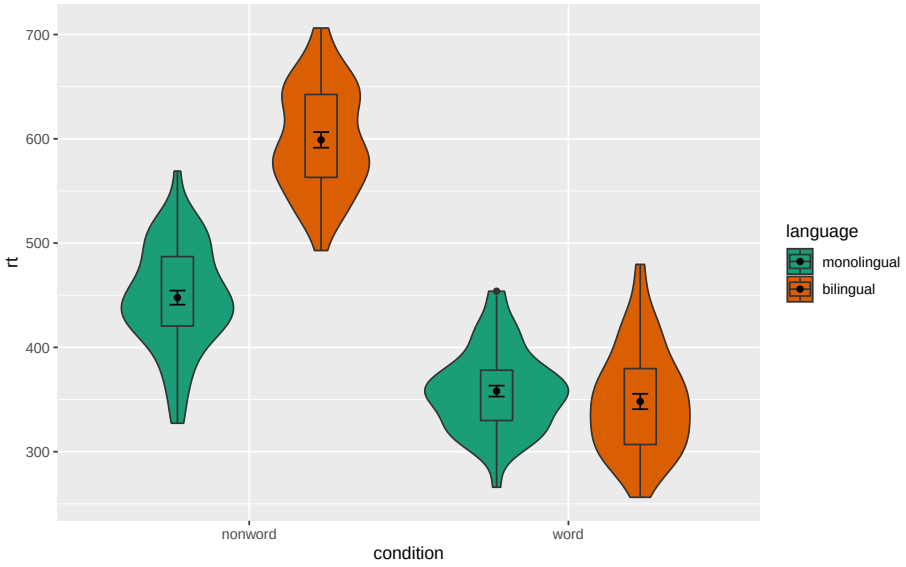


Figure 27. Grouped violin-boxplots with repositioning.

Customisation part 3

Combining multiple type of plots can present an issue with the colours, particularly when the fill and line colours are similar. For example, it is hard to make out the boxplot against the violin plot above.

There are a number of solutions to this problem. One solution is to adjust the transparency of each layer using `alpha`. The exact values needed can take trial and error:

```
ggplot(dat_long, aes(x = condition, y= rt, fill = language,
                     group = paste(condition, language))) +
  geom_violin(alpha = 0.25, position = pos) +
  geom_boxplot(width = .2,
               fatten = NULL,
               alpha = 0.75,
               position = pos) +
  stat_summary(fun = "mean",
               geom = "point",
               position = pos) +
  stat_summary(fun.data = "mean_se",
               geom = "errorbar",
               width = .1,
               position = pos) +
  scale_fill_brewer(palette = "Dark2")
```

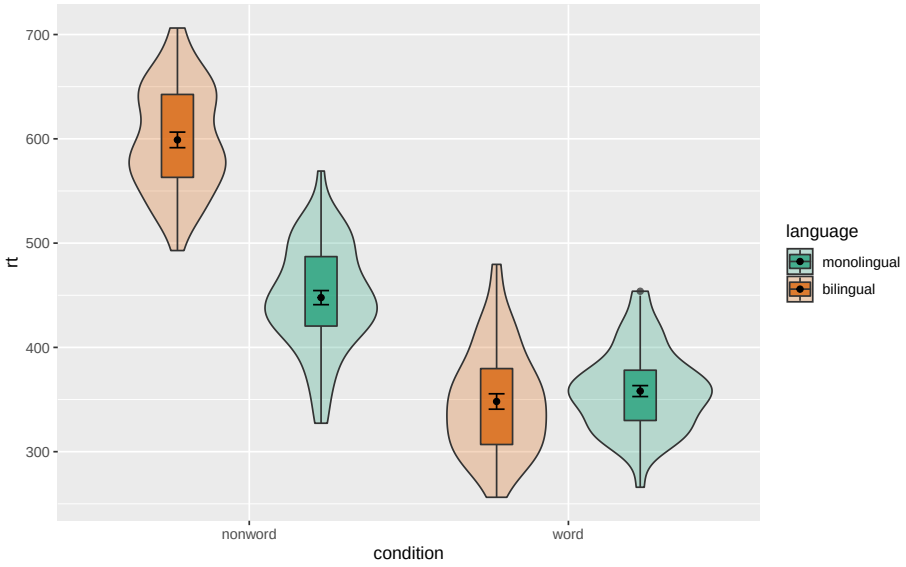


Figure 28. Using transparency on the fill color.

Alternatively, we can change the fill of individual geoms by adding `fill = "colour"` to each relevant geom. In the example below, we fill the boxplots with white. Since all of the boxplots are no longer being filled according to language, but you still want a four separate boxplots, you have to add an extra mapping to `geom_boxplot()` to specify that you want the output grouped by the interaction of condition and language.

```
ggplot(dat_long, aes(x = condition, y= rt, fill = language)) +
  geom_violin(position = pos) +
  geom_boxplot(width = .2, fatten = NULL,
               mapping = aes(group = interaction(condition, language)),
               fill = "white",
               position = pos) +
  stat_summary(fun = "mean",
               geom = "point",
               position = pos) +
  stat_summary(fun.data = "mean_se",
               geom = "errorbar",
               width = .1,
               position = pos) +
  scale_fill_brewer(palette = "Dark2")
```

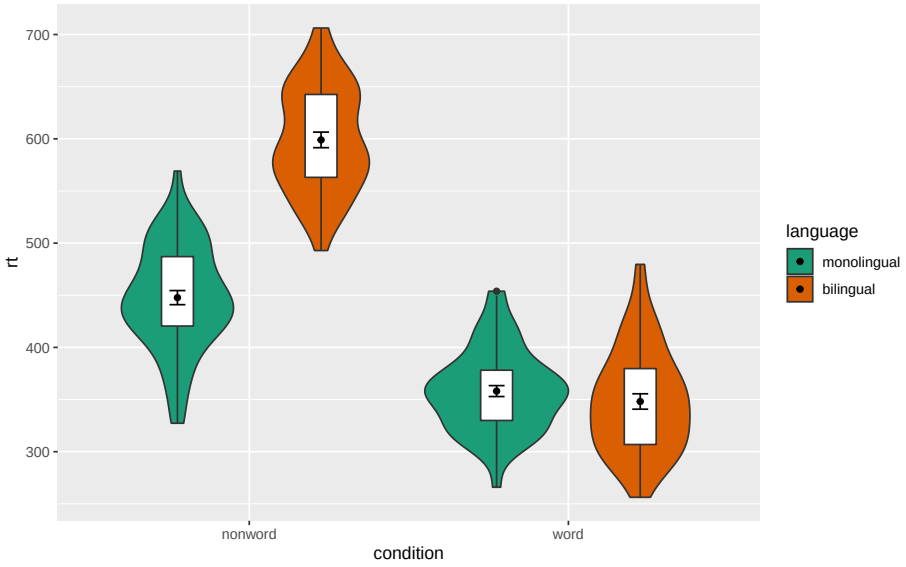


Figure 29. Manually changing the fill color.

Activities 3

Before you go on, do the following:

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

Multi-part Plots

Interaction plots

Interaction plots are commonly used to help display or interpret a factorial design. Just as with the bar chart of means, interaction plots represent data summaries and so they 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. If you want different groups to be different sizes (for example, the sample size of each study when showing the results of a meta-analysis or population of a city on a map), set this inside the `aes()` function; if you want to change the size for all groups, set it inside the relevant `geom_*()` 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 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()
```

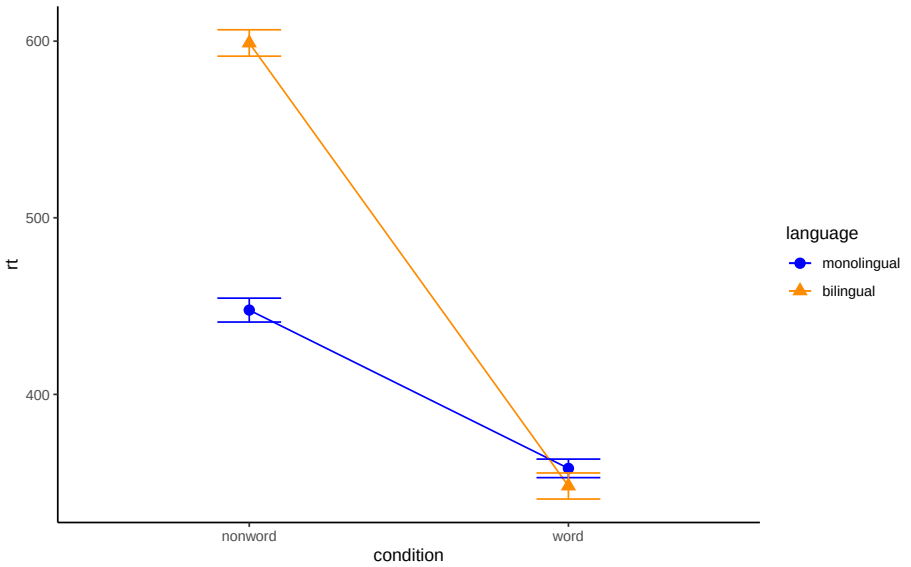


Figure 30. Interaction plot.

You can use redundant aesthetics, such as indicating the language groups using both colour and shape, in order to increase accessibility for colourblind readers or when images are printed in greyscale.

Combined interaction plots

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

This code is more complex than all prior code because it does not use a universal mapping of the plot aesthetics. In our code so far, the aesthetic mapping (`aes`) of the plot has been specified in the first line of code because all layers used the same mapping. However, is is also possible for each layer to use a different mapping – we encourage you to build up the plot by running each line of code sequentially to see how it all combines.

- The first call to `ggplot()` sets up the default mappings of the plot that will be used unless otherwise specified - the `x`, `y` and `group` variable. Note the addition of `shape`, which will vary the shape of the geom according to the language variable.
- `geom_point()` overrides the default mapping by setting its own `colour` to draw the data points from each language group in a different colour. `alpha` is set to a low value to aid readability.
- Similarly, `geom_line()` overrides the default grouping variable so that a line is drawn to connect the individual data points for each *participant* (`group = id`) rather than each language group, and also sets the colours.

- Finally, the calls to `stat_summary()` remain largely as they were, with the exception of setting `colour = "black"` and `size = 2` so that the overall means and error bars can be more easily distinguished from the individual data points. Because they do not specify an individual mapping, they use the defaults (e.g., the lines are connected by language group). For the error bars, the lines are again made solid.

```
ggplot(dat_long, aes(x = condition, y = rt,
                     group = language, shape = language)) +
  # adds raw data points in each condition
  geom_point(aes(colour = language),alpha = .2) +
  # add lines to connect each participant's data points across conditions
  geom_line(aes(group = id, colour = language), alpha = .2) +
  # add data points representing cell means
  stat_summary(fun = "mean", geom = "point", size = 2, colour = "black") +
  # add lines connecting cell means by condition
  stat_summary(fun = "mean", geom = "line", colour = "black") +
  # add errorbars to cell means
  stat_summary(fun.data = "mean_se", geom = "errorbar",
               width = .2, colour = "black") +
  # change colours and theme
  scale_color_brewer(palette = "Dark2") +
  theme_minimal()
```

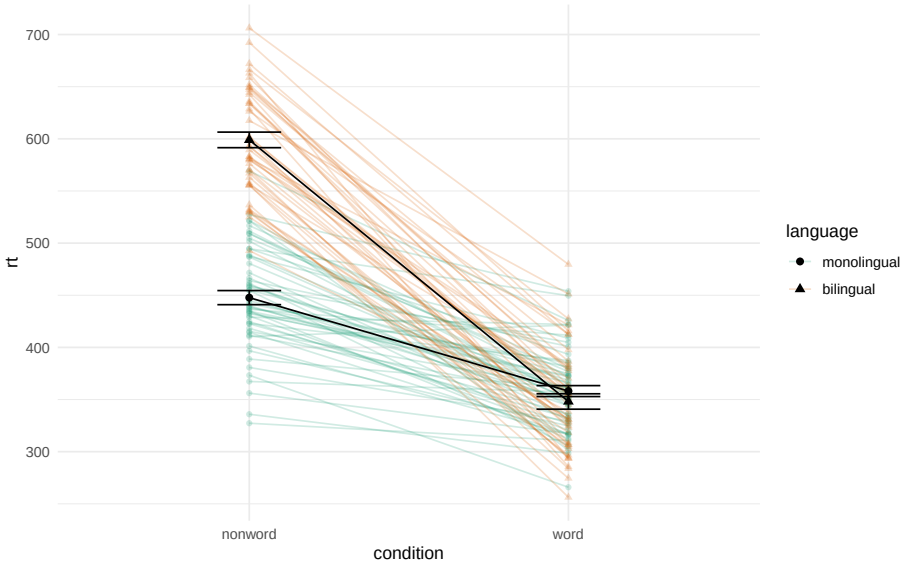


Figure 31. Interaction plot with by-participant data.

Facets

So far we have produced single plots that display all the desired variables. However, there are situations in which it may be useful to create separate plots for each level of a variable. This can also help with accessibility when used instead of or in addition to group colours. The below code is an adaptation of the code used to produce the grouped scatterplot (see Figure 26) in which it may be easier to see how the relationship changes when the data are not overlaid.

- Rather than using `colour = condition` to produce different colours for each level of `condition`, this variable is instead passed to `facet_wrap()`.
- Set the number of rows with `nrow` or the number of columns with `ncol`. If you don't specify this, `facet_wrap()` will make a best guess.

```
ggplot(dat_long, aes(x = rt, y = age)) +
  geom_point() +
  geom_smooth(method = "lm") +
  facet_wrap(facets = vars(condition), nrow = 2)
```

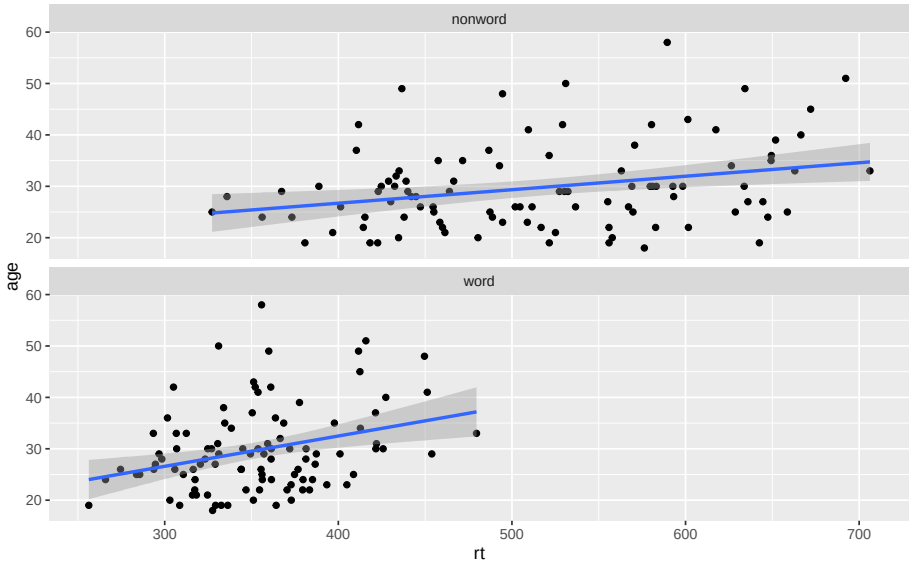


Figure 32. Faceted scatterplot

As another example, we can use `facet_wrap()` as an alternative to the grouped violin-boxplot (see Figure 27) in which the variable `language` is passed to `facet_wrap()` rather than `fill`. Using the tilde (`~`) to specify which factor is faceted is an alternative to using `facets = vars(factor)` like above. You may find it helpful to translate `~` as **by**, e.g., facet the plot by language.

```
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()
```

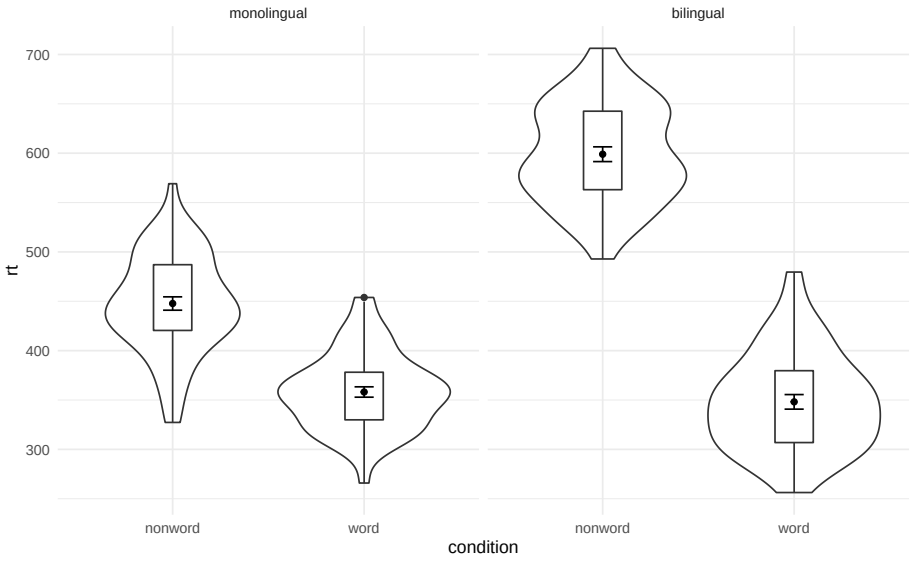


Figure 33. Facted violin-boxplot

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

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

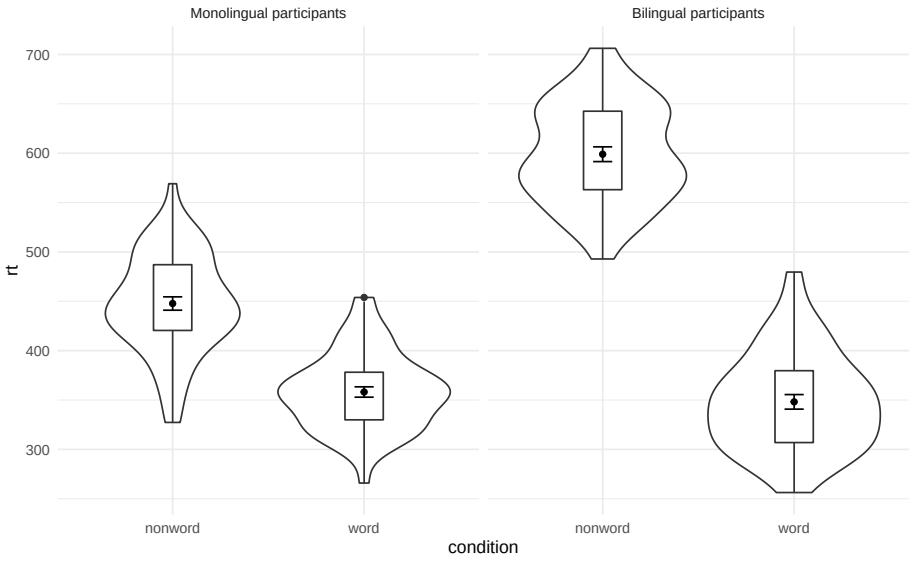


Figure 34. Faceted violin-boxplot with updated labels

Storing plots

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

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

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

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

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

Saving plots as images

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

`ggsave()` will save the last plot displayed. However, you can also specify a specific plot object if you have one saved.

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

The width, height and resolution of the image can all be manually adjusted. Fonts will scale with these sizes, and may look different to the preview images you see in the Viewer tab. The help documentation is useful here (type `?ggsave` in the console to access the help).

Multiple plots

As well as creating separate plots for each level of a variable using `facet_wrap()`, you may also wish to display multiple different plots together. The `patchwork` package provides an intuitive way to do this. Once it is loaded with `library(patchwork)`, you simply need to save the plots you wish to combine to objects as above and use the operators `+`, `/` `()` and `|` to specify the layout of the final figure.

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

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

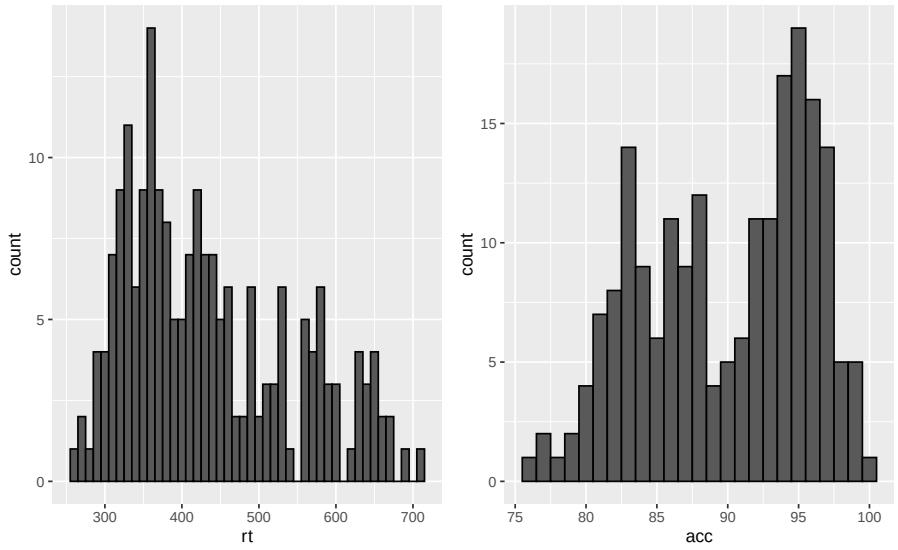


Figure 35. Side-by-side plots with patchwork

```
p1 / p2 # stacked
```

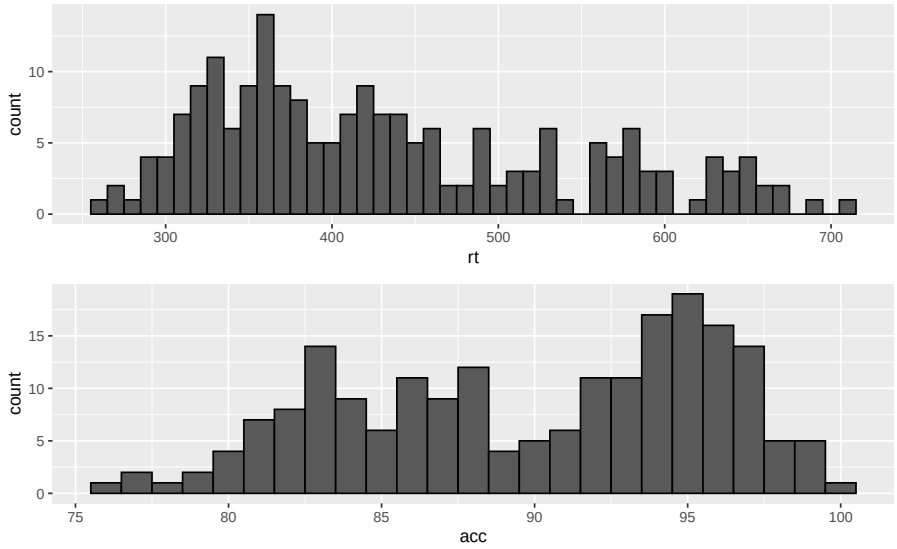


Figure 36. Stacked plots with patchwork

Combining three or more plots. Three or more plots can be combined in a number of ways. The `patchwork` syntax is relatively easy to grasp with a few examples and a bit of trial and error. The exact layout of your plots will depend upon a number of factors. Create three plots names `p1`, `p2` and `p3` and try running the examples below. Adjust the use of the operators to see how they change the layout. Each line of code will draw a different figure.

```
p1 / p2 / p3
(p1 + p2) / p3
p2 | p2 / p3
```

Customisation part 4

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

```
p1 + labs(x = "Mean reaction time (ms)",
          y = "Number of participants",
          title = "Distribution of reaction times",
          subtitle = "for 100 participants")
```

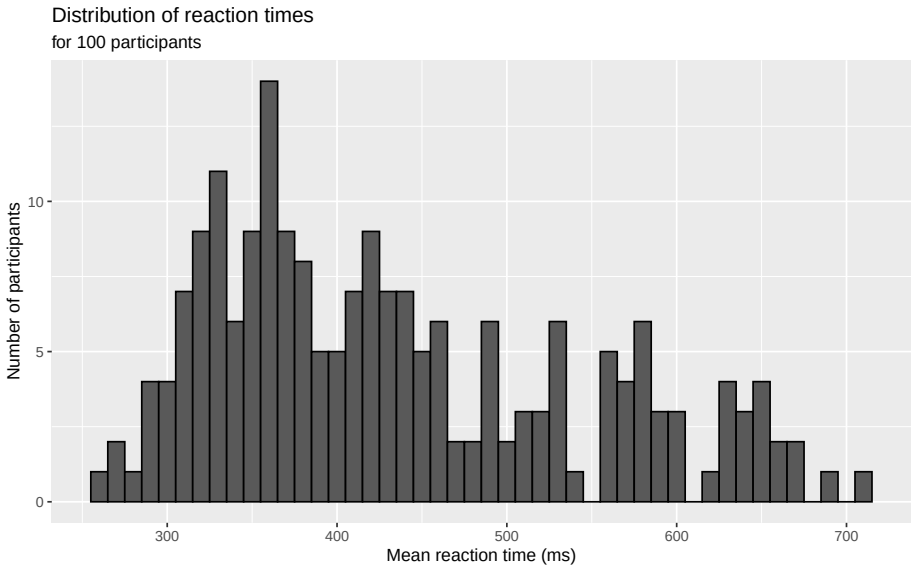


Figure 37. Plot with edited labels and title

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

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

The code below adds `fill = language` to violin-boxplots that are also faceted by language. We adjust `alpha` and use the brewer colour palette to customise the colours. Specifying a `fill` variable means that by default, R produces a legend for that variable. However, the use of colour is redundant with the facet labels, so you can remove this legend with the `guides` function.

```
ggplot(dat_long, aes(x = condition, y= rt, fill = language)) +
  geom_violin(alpha = .4) +
  geom_boxplot(width = .2, fatten = NULL, alpha = .6) +
  stat_summary(fun = "mean", geom = "point") +
  stat_summary(fun.data = "mean_se", geom = "errorbar", width = .1) +
  facet_wrap(~factor(language,
                     levels = c("monolingual", "bilingual"),
                     labels = c("Monolingual participants",
```

```
                                  "Bilingual participants"))) +
  theme_minimal() +
  scale_fill_brewer(palette = "Dark2") +
  guides(fill = "none")
```

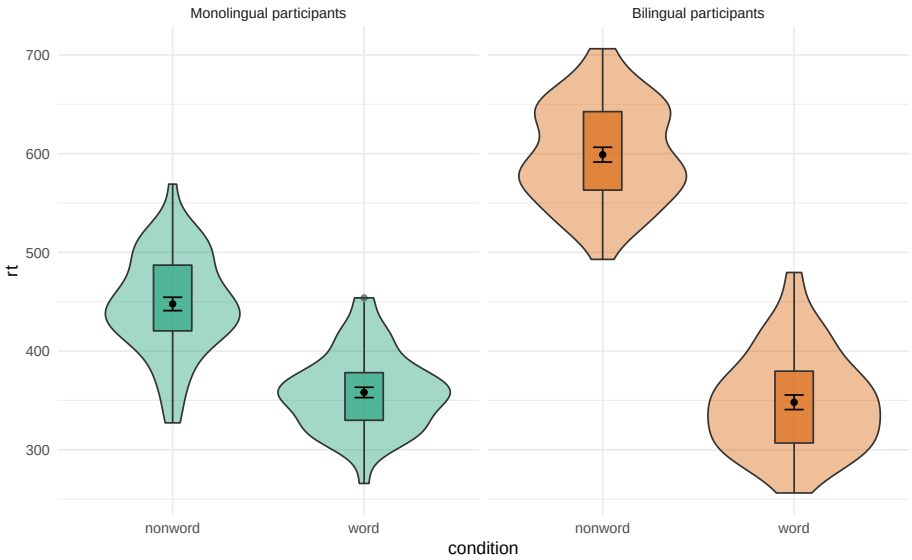


Figure 38. Violin-boxplot with redundant facets and fill.

Activities 4

Before you go on, do the following:

1. 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.

2. 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 from (Allen et al., 2021).

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

Split-violin plots

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

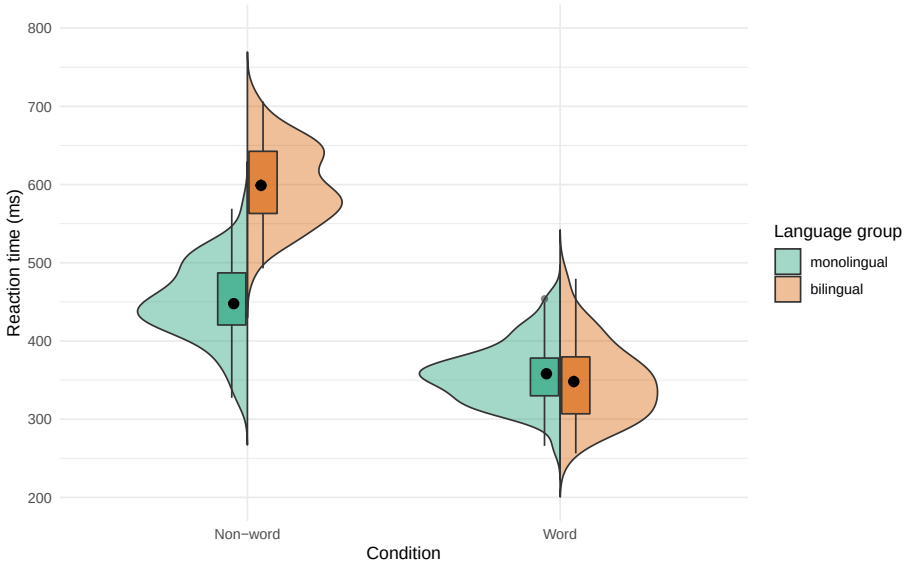
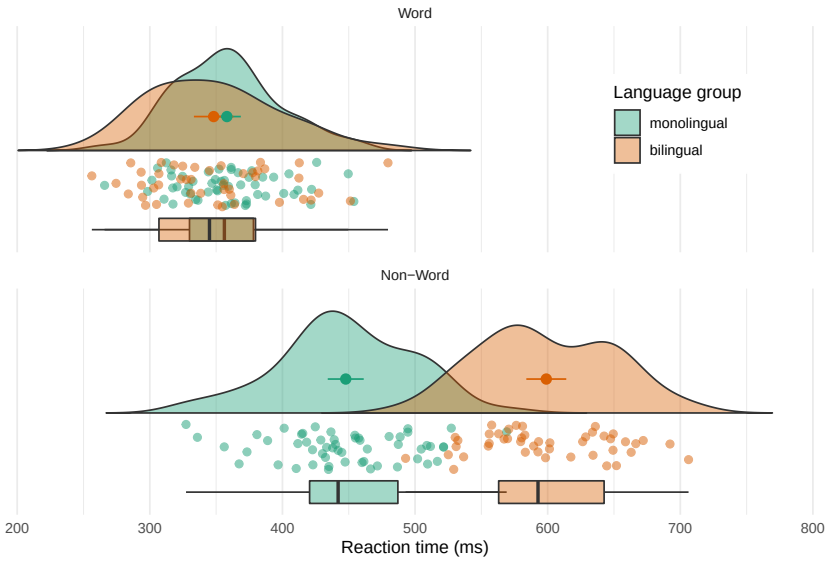


Figure 39. Split-violin plot

Raincloud plots

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

\begin{figure}



797 {

798 }

799 \caption{Raincloud plot. The point and line in the centre of each cloud represents its
800 mean and 95% CI. The rain respresents individual data points.} \end{figure}

801 ## Ridge plots

802 Ridge plots are a series of density plots that show the distribution of values for several
803 groups. Figure 40 shows data from (Nation, 2017) and demonstrates how effective this
804 type of visualisation can be to convey a lot of information very intuitively whilst being
805 visually attractive.

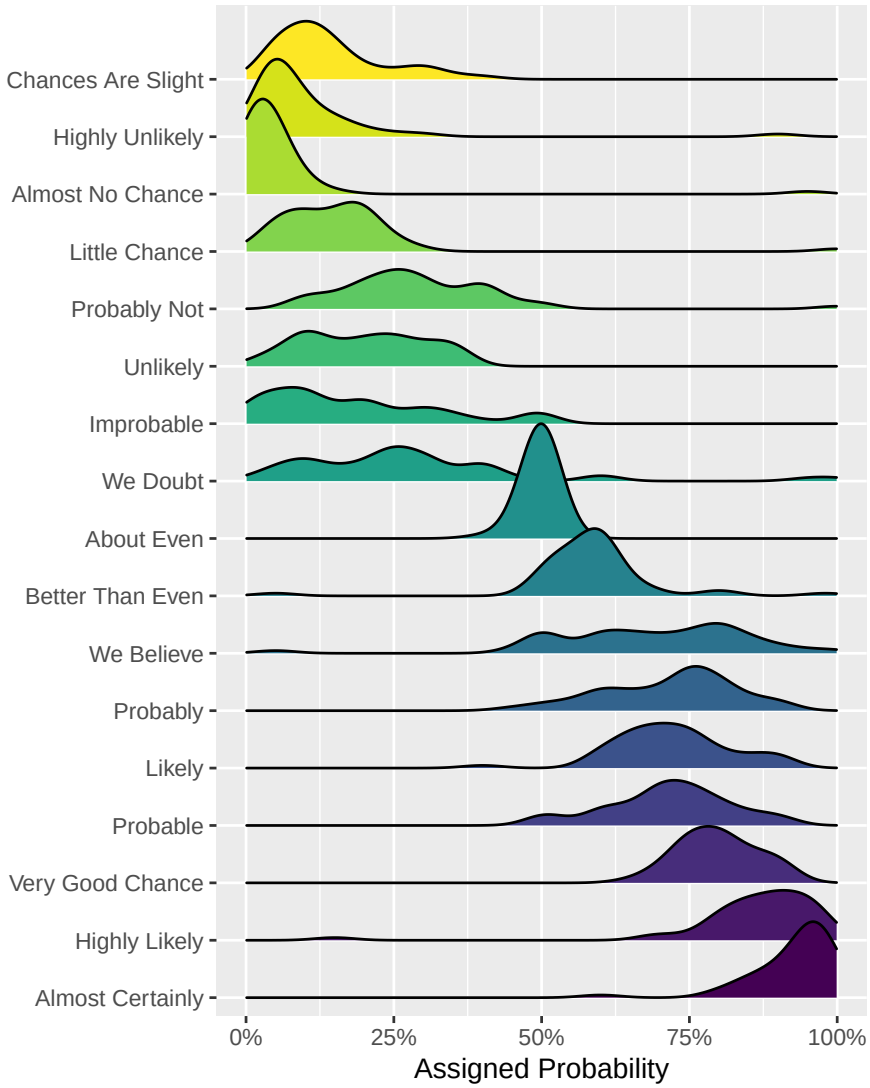


Figure 40. A ridge plot.

Alluvial plots

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

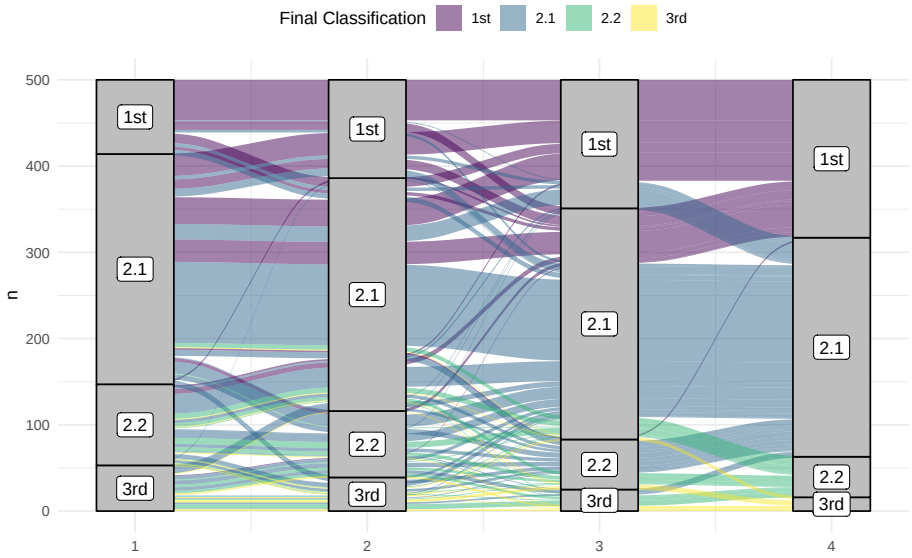


Figure 41. An alluvial plot showing the progression of student grades through the years.

Maps

Working with maps can be tricky. The `sf` package provides functions that work with ggplot2, such as `geom_sf()`. The `rnaturalearth` package provides high-quality mapping coordinates.

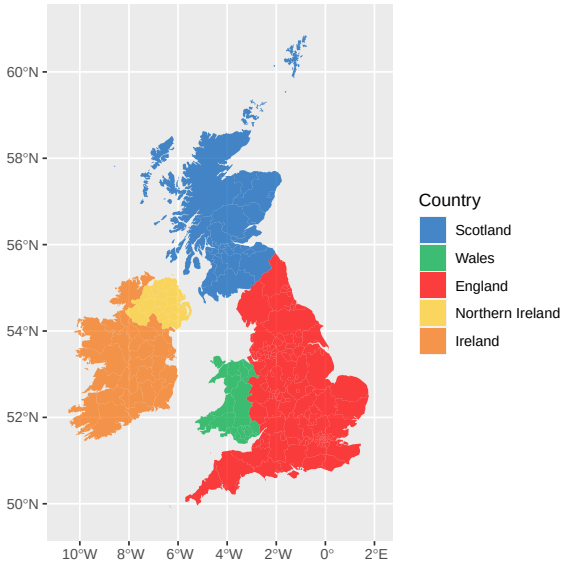


Figure 42. Map coloured by country.

Conclusion

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

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

Acknowledgements

Author Contributions.

- EN: Conceptualization; Visualization; Writing - original draft
- PM: Visualization; Writing - original draft
- WT: Visualization; Writing - original draft
- HP: Visualization; Writing - original draft
- LD: Software; Visualization; Writing - review & editing

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

Funding. LMD was supported by European Research Council grant #647910.

Research Software. This tutorial uses the following open-source research software: R Core Team (2021), Wickham et al. (2019), DeBruine (2021), Aust and Barth (2020), Wickham (2016b), Pedersen (2020), Brunson (2020), Wilke (2021), Pebesma (2018), South (2017).

References

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

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

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

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

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

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

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

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

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

Pebesma, E. (2018). Simple Features for R: Standardized Support for Spatial Vector Data. *The R Journal*, *10*(1), 439–446. https://doi.org/10.32614/RJ-2018-009

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

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

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

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

South, A. (2017). *Rnaturalearth: World map data from natural earth*. Retrieved from https://CRAN.R-project.org/package=rnaturalearth

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

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

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

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

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

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

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

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

Wilkinson, L., Anand, A., & Grossman, R. (2005). Graph-theoretic scagnostics. *IEEE Symposium on Information Visualization (InfoVis 05)*, 157–158. IEEE Computer Society.

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