The #TidyTuesday Cookbook

A case study approach to learning data visualisation with {ggplot2}

Nicola Rennie

Contents

Preface

#TidyTuesday is a weekly social data project which aims to make learning to work with data easier, by providing real-world datasets. Participants are encouraged to explore the data shared via GitHub each week, create an output such as a data visualization, and share their output alongside their code, with the community.

After three years of weekly contributions, I've worked with around 150 datasets and created over 150 data visualizations. This book will present a subset of these visualizations, and describe the process used to create them. Each chapter will cover a different data visualization, showing: the data exploration process; the choice of data visualization type; the initial design ideas with hand-drawn sketches; the first build of a plot; and the iterative process of styling plots.

Author

Nicola Rennie is a Lecturer in Health Data Science within the Centre for Health Informatics, Computing, and Statistics at Lancaster University. She holds a PhD in Statistics and Operational Research, focusing on analysing and visualising transport demand. Her current research is focused on applications of statistics and machine learning to health-related data, communicating statistics, and the effective teaching of data science. She has experience of teaching at both undergraduate and postgraduate level, in courses covering fundamentals of data science, population health, and statistical programming. Nicola has also previously worked in data science consultancy, and delivered training courses covering topics including advanced data visualisation with R, statistical modelling, and reproducible reporting. She is the author and maintainer of several R packages, including multiple {ggplot2} extension packages. Nicola is a regular speaker at R and data science meetups, and is the current chapter organiser of R-Ladies Lancaster. She is co-author of the Royal Statistical Society's Best Practices for Data Visualisation Guide, and an active member of the Royal Statistical Society. Several of her data visualisations have been long-listed at the Information is Beautiful Awards.

1

Introduction

Welcome to *The #TidyTuesday Cookbook*. Each chapter will cover a different data visualisation, showing: the data exploration process; the choice of data visualisation type; the initial design ideas with hand-drawn sketches; the first build of a plot; and the iterative process of styling plots. Full code will be provided and explained for each step of the creative process.

In this chapter, we'll discuss why you should visualize your data - although by virtue of the fact that you're reading this book, hopefully you're already sold on the general idea! You'll also find out a bit more about where the idea for this book came from and what #TidyTuesday is, alongside a few notes about the R code you'll see in this book.

1.1 Why visualize data?

Data visualization can be a very effective and efficient means of communicating information. Visualizing your data typically serves one of two purposes: (i) as part of exploratory analysis to help uncover discrepancies in data and identify interesting relationships to measure; or (ii) to communicate key insights and messages to a broader audience. The case-study nature of this book means that we'll talk about both of these aspects, though we 'll focus mostly on the second.

Choosing an appropriate type of visualization and making careful choice about design can clarify the message you are trying to convey to a reader. That does not necessarily mean that every chart must follow a set of *rules* and stick to a rigid format. Instead, data visualization is a blend of science and creativity - many of the key *landmark* data visualizations held up as excellent examples don't fit into the standard categories of bar charts, scatter plots, or line graphs.

That being said, the visualizations in this book are not necessarily always the most effective choice of visualization for the data and relationship shown. Rather, this book aims to show you examples of the end-to-end process of creating data visualizations, with a focus on the technical details of building them in R. You'll see some *hacky* solutions and unusual ideas that you can

use to transform your data visualizations.

1.2 Background knowledge

This book is primarily aimed at those who wish to develop their data visualization skills in R. Readers of this book may find a basic knowledge of R, more specifically of the {tidyverse} ecosystem, useful though all code used in examples is fully explained. Readers do not need to be experienced in {ggplot2}, though this book will also be of interest to those who are. Readers of this book will be of interest to those who are already familiar with R (including {ggplot2}), and wish to develop their skills in designing data visualizations further. It will also be of interest to those who already design data visualizations using other tools, and want to learn how to do the equivalent in R.

1.3 A note about R code

In R, the pipe operator *takes the thing on its left and passes it along to the function on its right* (Wickham, Çetinkaya-Rundel, and Grolemund 2023). You can find a full description of the pipe operator in R for Data Science. The pipe (`%>%`) was first introduced to R via the {magrittr} package. Since version 4.1.0 of R, a version of the pipe (`|>`) has existed in base R. The base R version of the pipe is used throughout the book. Although there are some difference between the two version of the pipe, in this book, they can be used interchangeably.

Namespacing (the pre-fixing of functions with the package name and `::`) is used for all packages except {ggplot2} and base R packages. Namespacing is useful for two reasons (i) from a learning perspective, it makes it easier to recognise where functions come from and how they connect together, and (ii) from a programming perspective, it reduces conflicts and errors - something we all want less of!

You may notice that some of the final images differ slightly from those initially created and published on social media. You might also find some small differences in the code used to produce them if you compare the contents of this book to original scripts in the GitHub repository. These differences are likely due to one of four reasons:

- packages have since been updated and code has been changed to use newer syntax. Many of the code changes relate to changes in {ggplot2} version 3.5.0;

- some aspects have been omitted from a visualization to avoid explaining *everything* in Chapter 1 - but those aspects are all covered and linked to in later chapters;
- after many years of practice, there may be more efficient ways of re-writing code from some of the earlier plots. Any changes are clearly labelled and discussed;
- some images may be different due to copyright reasons.

All software requirements, including a complete list of package versions, can be found in the Appendix.

Part I

Common charts don't need to be boring!

2

UK Museums: highlighting line charts with {gghighlight}

In this chapter we'll consider different ways of visualising data that varies over time, aiming to avoid *spaghetti charts*, with the help of the {gghighlight} extension package.

2.1 Data

The Mapping Museums project (Mapping Museums 2021) has collected data relating to over 4,000 museums in the UK, covering museums from 1960 onward. The data can be downloaded from the Mapping Museums website at www.mappingmuseums.org. It was also used as a #TidyTuesday (R4DS Online Learning Community 2023) data set in November 2022, and we'll use that version for this chapter.

Let's start by reading in the data and looking at the definitions of the variables:

```
museums <- readr::read_csv("data/museums.csv")
```

The data contains information for 35 different variables on 4191 museums. After the first `museum_id` columns, the next 8 columns provide information on the location of the museum - including address and co-ordinates. The next set of columns provides information about the museums such as whether it's an accredited museum, how it's governed, what types of items it has, and when it was open. Since this is a collated data set, information is also provided on the original source of the data for some of these variables. The remaining columns provide information on the area in which the museum is located, including information about different deprivation indices and geodemogrqaphic group (type of area e.g. `"University Towns and Cities"`). A full glossary of the terms used can be found on the Mapping Museums website at museweb.dcs.bbk.ac.uk/glossary.

2.2 Exploratory work

Let's start exploring what these variables look like!

> Remember that you can use `View(museums)` to inspect the data in a more human-readable format.

2.2.1 Data exploration

There are many different aspects of this data we could inspect:

- How are museums spread out across the UK? Is there a higher concentration of museums in more affluent areas?
- What types of museums are most prevalent in the UK? Does the vary based on whether the museum is accredited?
- How is the number of museums open changing over time? Are more museums opening than closing?

Some of the variables, such as `DOMUS_Subject_Matter` have quite a lot of missing values which makes

Although these definitely warrant further exploration,

Given the wealth of information around deprivation indices (8 different variables), that's the first aspect of the data that jumps out. How does the number of museums vary per index of deprivation?

```
barplot(
  table(museums$Area_Deprivation_index),
  cex.axis = 0.5,
  cex.names = 0.5
)
```

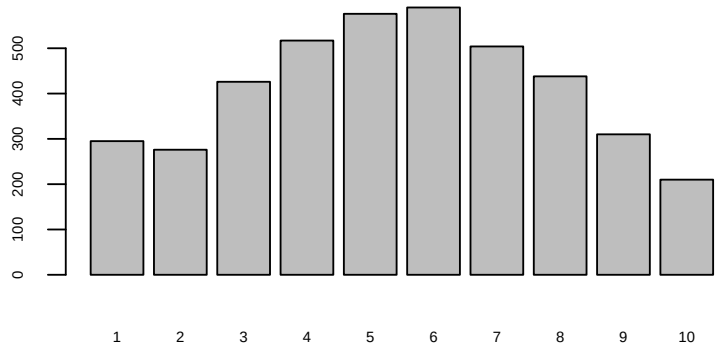


Figure 2.1: Bar chart of number of museums per level of deprivation, with higher numbers of museums shown in levels 5 and 6.

Here, `1` is the most deprived, and `10` is the least deprived. This is quite an interesting relationship - there have been more museums in places with a moderate level of deprivation compared to higher or lower level. This might feel a little bit counter-intuitive, we might expect there to be more museums in more affluent areas. But looking at the total number of museums in the data per index of deprivation doesn't tell us the whole story. How many of these museums were open at once? How many are still open today?

To get a better understanding of the relationship between deprivation and the number of museums, we could look at how the number of museums changed for each index of deprivation has changed since 1960 (the earliest date in the data set). Then we might be able to tell whether more are opening or closing, and how this varies across the different levels of deprivation.

2.2.2 Exploratory sketches

We have 10 deciles of deprivation, so we'll have 10 time series that we want to plot that show the number of museums over time. There are different options for plotting time series like this. We need to think about some different choices:

- What type of *geometry* will we use? Lines, points, shaded areas? Lines are the most common approach for
- Do we plot all 10 lines on the same chart and use color to denote the

different levels of deprivation?
- Or do we use faceting to split it into multiple smaller plots by deprivation index?

If we go with line charts, and faceting by deprivation index, that might look something like this:

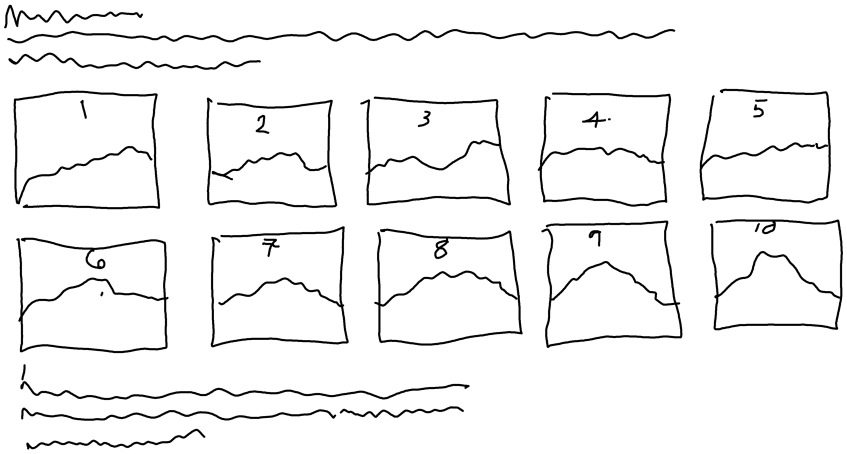


Figure 2.2: Initial sketch of a faceted line chart showing change over time for each level of deprivation.

We could achieve something this type of plot using `geom_line()` and `facet_wrap()` from {ggplot2} . We've talked about faceting before in both Chapter …(1) and Chapter …(2) but we haven't really thought much about the layouts of those facets. In Chapter …(1), we used faceting with only 3 categories, where the layout choice is fairly obvious: either one column or one row. In Chapter …(2), we used `facet_grid()` where the number of levels in two factor variables define the number of rows and columns. Here, we have a little bit more flexibility and we might want to think about controlling that layout.

If we leave it to the default options, {ggplot2} usually tries to coerce it into something *square-ish*. Here, it would likely give us a 3x4 grid filled with 10 plots (for the 10 levels of deprivation) and 2 blank spaces. It's not always possible, but plots without those blank spaces often look a lot cleaner and tidier. We could do either a 5x2 (or 2x5) grid, or a 10x1 (or 1x10) grid to fit our small multiple plots into a perfect rectangle. A 5x2 grid works better here since we don't want to make a very wide but very short plot.

2.3 Preparing a plot

So let's get started on preparing the data to create this plot!

2.3.1 Data wrangling

This is one of those very *real* datasets - the data wrangling is not straightforward for the data. In fact, the two columns relating to opening and closing dates are the two that will require the most attention.

First, let's drop any columns that we don't actually need using the `select()` function from {dplyr} - we only need the `Year_opened`, `Year_closed`, and `Area_Deprivation_index` columns to calculate the number of museums open each year . There are a few `NA` values in the `Area_Deprivation_index` column so we'll drop those rows using `drop_na()` from {tidyr} .

Now let's start dealing with the year columns. If you look at the values, you'll notice that they're not exactly what you'd expect in a *year* column:

```
head(museums$Year_opened, 4)
```

```
[1] "2012:2012" "1971:1971" "1984:1984" "1971:1971"
```

```
head(museums$Year_closed, 4)
```

```
[1] "9999:9999" "2007:2017" "9999:9999" "2012:2012"
```

Each entry is in fact two years, separated by a `:`. According to the Mapping Museums glossary, this is because these are actually date ranges. For some museums, it wasn't possible to establish an exact opening or closing date, and instead a date range is given based on partial information. For example, a value of `"2007:2017"` means the museum opened (or closed) sometime between 2007 and 2017.

Let's separate out these year values into four columns instead of two using the `separate()` function from {tidyr} . We separate based on the `:`, and create two new columns, `opened1` and `opened2`. from the `Year_opened` column. We do the same thing for the `Year_closed` closed column. We also want to make sure that these new columns are numeric rather than character columns, so we use `mutate()` and `across()` from {dplyr} to convert them using `as.numeric()` . We can also convert the `Area_Deprivation_index` to a factor rather than a numeric and make sure the order is correct.

```
museum_subset <- museums |>
  dplyr::select(
    Year_opened, Year_closed, Area_Deprivation_index
```

```
) |>
tidyr::drop_na() |>
tidyr::separate(
  Year_opened,
  into = c("opened1", "opened2"),
  sep = ":"
) |>
tidyr::separate(
  Year_closed,
  into = c("closed1", "closed2"),
  sep = ":"
) |>
dplyr::mutate(
  dplyr::across(
    c(opened1, opened2, closed1, closed2), as.numeric
  ),
  Area_Deprivation_index = factor(Area_Deprivation_index,
   ↪  levels = 1:10)
)
```

> You could alternatively use `separate_wider_delim()` instead of `separate()`.

Now we need to think about how to deal with all of these *year* columns:

- If the date before the `:` and the date after the `:` are the same, we want to treat it as an exact year and only keep one value.
- If the dates do not match, we need to decide a way of choosing which year to use. The simplest approach is to take the midpoint of the date range.
- If the value in the `Year_closed` column is `"9999:9999"`, this means that the museum is still open.

Let's start with the last of these issues first. For the `closed1` and `closed2` columns, if the value is `"9999"`, we'll convert it to an `NA_real_` value and otherwise leave it as it is. We can do this using a combination of `mutate()`, `across()`, and `if_else()` from {dplyr} .

The remaining two issues can be dealt with at the same time using `case_when()` from {dplyr} . We can create a new column called `closed` which:

- if `closed1` and `closed2` are equal, takes this value;
- if `closed1` and `closed2` are not equal, takes the value in the middle (rounded so that we work only with whole year values).

The same approach is then applied to create another new column called `opened`. There are no instances in the data where one of the closing dates

is `NA` but the other is not, so we don't need to worry about that. We can also tidy up the output by dropping any columns we don't need, renaming the `Area_Deprivation_index` to something a little bit shorter and easier to work with, and arranging the data by level of deprivation.

```
museum_data <- museum_subset |>
  dplyr::mutate(dplyr::across(
    c(closed1, closed2),
    ~ dplyr::if_else(.x == 9999, NA_real_, .x)
  )) |>
  dplyr::mutate(closed = dplyr::case_when(
    closed1 == closed2 ~ closed1,
    closed1 != closed2 ~ round((closed2 + closed1) / 2)
  )) |>
  dplyr::mutate(opened = dplyr::case_when(
    opened1 == opened2 ~ opened1,
    opened1 != opened2 ~ round((opened2 + opened1) / 2)
  )) |>
  dplyr::select(Area_Deprivation_index, opened, closed) |>
  dplyr::rename(deprivation = Area_Deprivation_index) |>
  dplyr::arrange(deprivation)
```

This gives us data that looks like this:

```
head(museum_data)
```

```
# A tibble: 6 x 3
  deprivation opened closed
  <fct>        <dbl>  <dbl>
1 1             1988     NA
2 1             1984   2007
3 1             1983     NA
4 1             1969     NA
5 1             1989     NA
6 1             1995     NA
```

What we want to know is how many museums were open in a given year for each level of deprivation. Let's create a function that takes three inputs: a year of interest, a level of deprivation, and the `museum_data`. Inside the function, we can then `filter()` the data to only the level of deprivation we're interested in. Then we can count how many museums opened before or in the year we're interested in, and then same for the how many museums closed (remembering to deal with the `NA` values for museums that are still open). The difference between the two will be how many were open in that year.

```
num_year <- function(year, dep, data = museum_data) {
  df <- dplyr::filter(data, deprivation == dep)
```

```
  num_open <- sum(df$opened <= year)
  num_closed <- sum(df$closed <= year, na.rm = TRUE)
  diff <- num_open - num_closed
  return(diff)
}
```

Let's test it works:

```
num_year(1980, 3)
```

```
[1] 186
```

There were 186 museums open in 1980. Now, we need to run this function for every combination of year (from `1960` to `2021`) and deprivation level (from `1` to `10`). There are many different ways of doing this in R, and we're going to use the {purrr} package (Wickham and Henry 2023). We start by creating to variables with sequences of years and levels of deprivation - this is useful for testing if the code works because we can easily change it to a smaller number of years. Then we use `expand.grid` to create a `data.frame` with all of the combinations and pass this in as the first argument to `pmap_vec` from {purrr}. The second argument to `pmap_vec` is the function we want to apply (`num_year()`) that we defined above), where the first argument of `num_year()` comes from the first column of the grid of values, and the second argument from the second column. We can then convert the output into a matrix where each column is a level of deprivation, and each row is a year.

```
all_years <- 1960:2021
deps <- 1:10
output <- purrr::pmap_vec(
  expand.grid(all_years, deps),
  ~ num_year(year = .x, dep = .y)
)
results <- matrix(output,
  nrow = length(all_years),
  byrow = FALSE
)
colnames(results) <- 1:10
```

Now, our data looks like this:

```
head(results)
```

```
       1  2   3   4   5   6   7   8  9 10
[1,] 103 78 110 125 125 124 122 110 76 51
[2,] 103 78 112 132 125 131 126 113 77 53
[3,] 104 78 113 138 129 135 129 117 83 58
```

```
[4,] 105 79 118 140 136 140 130 123 85 60
[5,] 105 79 122 144 139 145 135 126 86 64
[6,] 106 80 123 148 143 147 137 128 89 67
```

We're almost ready to start plotting our data! We just need to convert this into a `tibble()` (or `data.frame`), add the `year` column using `mutate()`, and put the data into a long format using `pivot_longer` from {tidyr} . We want to end up with a 3 column data set, where the 3 columns are: `year`, `deprivation`, and `museums` (number of open museums).

```
plot_data <- results |>
  tibble::as_tibble() |>
  dplyr::mutate(year = all_years) |>
  tidyr::pivot_longer(
    -year,
    names_to = "deprivation",
    values_to = "museums"
  ) |>
  dplyr::mutate(
    deprivation = factor(deprivation, levels = 1:10)
  )
```

2.3.2 The first plot

One of most common approaches to visualising multiple time series, is to plot multiple lines on the same plot. Although we're already thinking about facets, it's still worth seeing what those line charts look like. It can help us to understand the overall variability of the data. We can create the initial plot using `ggplot()` to define the data and the aesthetic mapping with the `aes()` function. We have `year` on the x-axis, `museums` on the y-axis, and each line will have a different color based on `deprivation`. The actual lines are then added with `geom_line()`.

```
library(ggplot2)
ggplot(
  data = plot_data,
  mapping = aes(x = year, y = museums, color = deprivation)
) +
  geom_line()
```

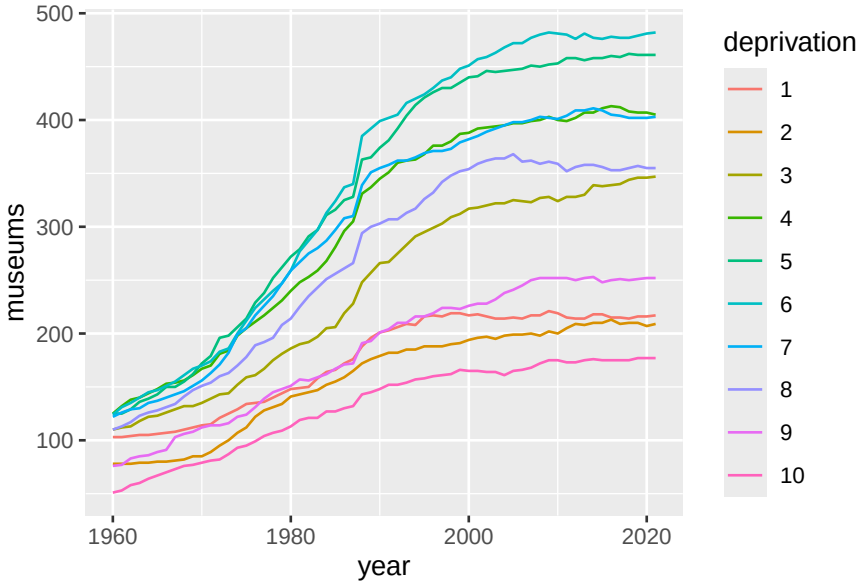


Figure 2.3: A line chart of total number of museums open per year for each level of deprivation, with the overlapping lines resembling spaghetti!

There are a few things that jump out immediately about this plot:

- The variability is increasing over time: in 1960 the gap between the highest and lowest values is much smaller than the gap between the highest and lowest values in 2020.
- It's difficult to tell which line belongs to which level of deprivation: some of the colors are quite similar, many of the lines intersect, and the order of the legend is generally in reverse to the order of the lines.
- It's what we might call a *spaghetti plot*: it shows the overall trend across all levels, but it's hard to tell the difference between different levels of deprivation.

Let's go back to our previous idea, and try separating out the deprivation levels into different facets. We can do this using `facet_wrap(~deprivation, nrow = 2)` , with the `nrow` argument used to create that 5x2 grid we talked about earlier. Let's also try changing `geom_line()` to `geom_area()`. Although line charts and area charts can both show the same data, line charts often cause us to focus more on the trend over time. In contrast, area charts often cause us to focus on the total volume over time - helping to highlight differences in total number of open museums over time rather than just increases and decreases. Remember to also change `color` to `fill` in the aesthetic mapping!

```
ggplot(
  data = plot_data,
  mapping = aes(x = year, y = museums, fill = deprivation)
) +
  geom_area() +
  facet_wrap(~deprivation, nrow = 2)
```

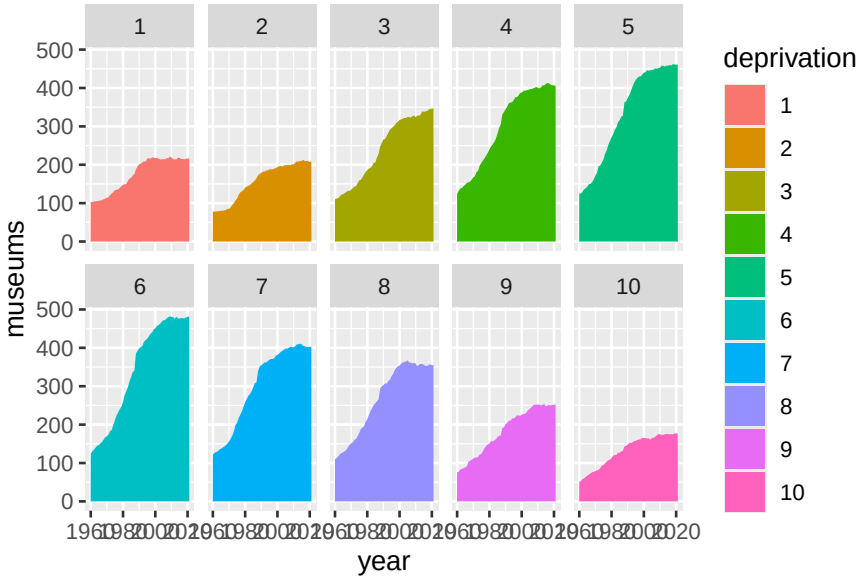


Figure 2.4: An area chart of total number of museums open per year for each level of deprivation, with the each deprivation level in a different facet.

This tells a similar story to the initial bar chart in Figure 2.1 - there are more museums open in the areas with moderate levels of deprivation. This plot still isn't ideal, for a couple of reasons:

- The number of open museums in 1960 is different in each faceted plot: since they all start at different levels, it makes it more difficult to compare the relative increases or decreases.
- It's hard to directly compare one facet to another since each facet only contains one trend line: we have to imagine overlaying the lines in our head to compare one trend line to another.

We could solve these problems by making two changes:

- Rescale the data based on the number of museums open in 1960; and
- Show all 10 lines on each faceted plot but highlight only one relating to each level of deprivation.

2.3.3 Highlighting with {gghighlight}

The second change can be implemented easily using the {gghighlight} package (Yutani 2023). {gghighlight} is a {ggplot2} extension package, specifically designed for highlighting points and lines based on some conditions. Let's switch back to `geom_line()` and add `gghighlight::gghighlight()`. We set `use_direct_label = FALSE` because {gghighlight} will otherwise add a label to each highlighted line - unnecessary since each line is labelled by it's facet label already.

```
ggplot(
  data = plot_data,
  mapping = aes(x = year, y = museums, color = deprivation)
) +
  geom_line() +
  facet_wrap(~deprivation, nrow = 2) +
  gghighlight::gghighlight(use_direct_label = FALSE)
```

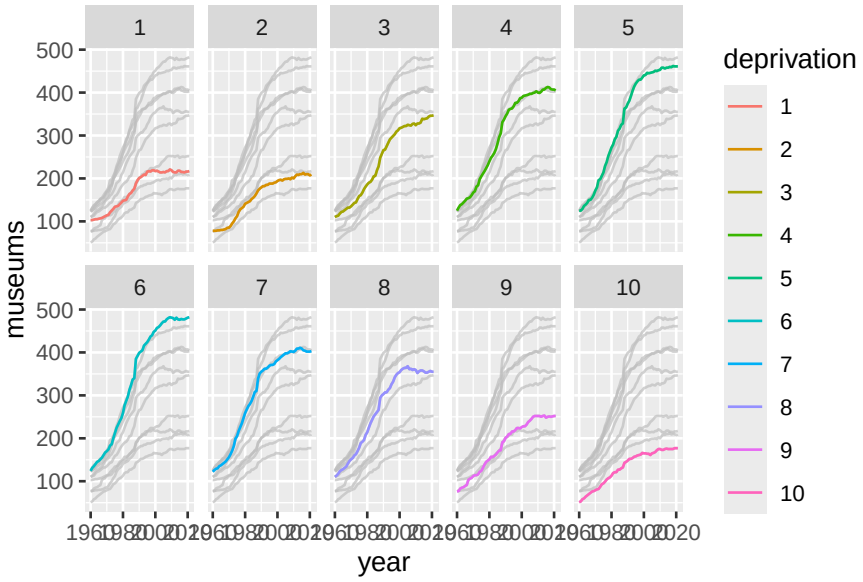


Figure 2.5: A line chart of total number of museums open per year for each level of deprivation, with the each deprivation level in a different facet and individual lines highlighted.

We can also perform an additional bit of data wrangling to rescale the values by their 1960 levels. We start by filtering the data to only include data from the year 1960 and keeping only the `deprivation` and `museums` columns . Then, we join this baseline data back to our original data based on the `deprivation`

level, and for each year calculate the percentage change since 1960 and save it in a new column called `change`.

```
lookup <- plot_data |>
  dplyr::filter(year == 1960) |>
  dplyr::select(deprivation, museums)

new_plot_data <- plot_data |>
  dplyr::left_join(lookup, by = "deprivation") |>
  dplyr::rename(
    museums = museums.x,
    museums_1960 = museums.y
  ) |>
  dplyr::mutate(
    change = (100 * (museums - museums_1960) / museums_1960)
  ) |>
  dplyr::select(year, deprivation, change)
```

We can then re-do our line chart using the rescaled `change` data on the y-axis instead:

```
base_plot <- ggplot(
  data = new_plot_data,
  mapping = aes(x = year, y = change, color = deprivation)
) +
  geom_line() +
  facet_wrap(~deprivation, nrow = 2) +
  gghighlight::gghighlight(use_direct_label = FALSE)
base_plot
```

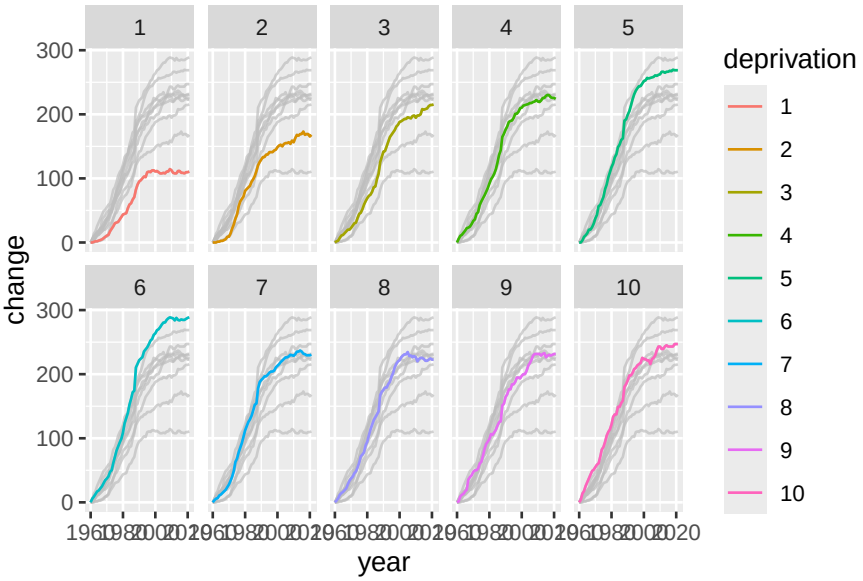


Figure 2.6: A line chart of percentage increase in museums (compared to 1960) for each level of deprivation, with the each deprivation level in a different facet and individual lines highlighted.

This is starting to look promising, but there are still ways that we can style our chart to improve it.

2.4 Advanced styling

So what can we do to make this chart better?

- We can get rid of the legend. The colors are based on the levels of deprivation, which are already labelled on the facet titles.
- The default choice of color isn't great - they're not grayscale printing friendly, and their not colorblind friendly either.
- It's not immediately clear what this chart shows: it could do with smoe text to explain what's going on.

2.4.1 Colors

Let's get started with choosing some colors. As discussed in Chapter ...(1), we'll save color codes as variables: either using hex codes or color names. Let's use

`"black"` for text, and `"#fafafa"` for the background color. The use of a light gray rather than white for the background is primarily personal preference - it's less glaringly bright on a screen. Although, you do have to be a bit more careful around contrast of text against the background - hence the black text.

```
bg_col <- "#fafafa"
text_col <- "black"
```

For the colors of the lines, we can use the {viridis} package (Garnier et al. 2024). The {viridis} package provides multiple different color palettes which are designed to be visually pleasing, perceptually-uniform ,and colorblind friendly. The default viridis palette includes purples, blues, greens, then yellows. Viridis color palettes are most commonly used with continuous data, rather than categorical data. However, since the categories that we wish to color (levels of deprivation) are ordered, they will work well here too. The {viridis} package includes functions to add the colors to plots made with {ggplot2}. Here, we use the `scale_color_viridis()` function, specify that we want a discrete palette and that the lowest values should have the yellow colors with `direction = -1` .

```
col_plot <- base_plot +
  viridis::scale_color_viridis(discrete = TRUE, direction = -1)
col_plot
```

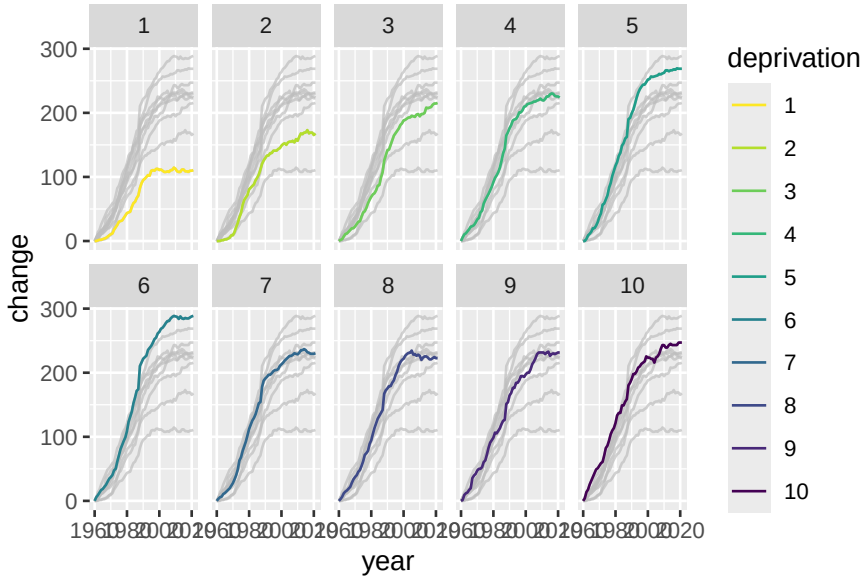


Figure 2.7: An updated version of the previous chart with colors chosen from the {viridis} R package.

2.4.2 Text and fonts

Let's start by loading in a different font using `font_add_google()` from {sysfonts} and setting font options with {showtext} . The *Raleway* font is a minimalist, sans serif font, and we'll use it for both body text and title text. We can save it as a single variable, `body_font`.

```
sysfonts::font_add_google("Raleway", "raleway")
showtext::showtext_auto()
showtext::showtext_opts(dpi = 300)
body_font <- "raleway"
```

We also need to define a title, subtitle, and caption. Adding a question as a title can help guide readers towards what you want them to see, but also force them to look for themselves. Here, we ask *Are there fewer museums opening in more deprived areas?* in the title - telling readers to look at changes across the levels of deprivation, but not giving them the answer straight away.

The subtitle then goes onto explain what the answer is, and exactly what is shown in the chart. The caption is more extended here than you may see in many charts because it explains more about the source of the data and what the variables actually are. Understanding definitions of variables isn't something we should ever take for granted. Unless you are already familiar with indices of deprivation, it may not be clear that `1` means higher levels of deprivation.

```
title <- "Are there fewer museums opening in more deprived
 ↪  areas?"
st <- "The change in the estimated number of open museums since
 ↪  1960 is significantly lower in areas with higher levels of
 ↪  deprivation. *Since around 2000, the number of open museums
 ↪  has stagnated across all areas, regardless of deprivation
 ↪  index. However, the rate of growth prior to this stagnation
 ↪  is lower in more deprived areas."
cap <- "*The Index of Multiple Deprivation (IMD) measures the
 ↪  relative deprivation of geographic areas in the UK,
 ↪  aggregating different dimensions (income, employment,
 ↪  education, health, crime, housing, and living environment).
 ↪  The index ranges from 1 (most deprived) to 10 (least
 ↪  deprived).<br><br>**In some instances it has been impossible
 ↪  to establish an exact opening or closing date for a museum.
 ↪  In these cases, museums' opening and closing dates are taken
 ↪  to be the mid point of a specified range of possible
 ↪  dates.<br><br>N. Rennie | Data: museweb.dcs.bbk.ac.uk"
```

> The caption includes HTML line breaks, `<br>`, since we'll be using `element_textbox_simple()` from {ggtext} for processing the text elements as we've done in previous chapters.

Let's add the title, subtitle, caption, and a y-axis label in using the `labs()` function. We can also remove the default column name label on the x-axis as it's very clear that the x-axis shows years.

```
text_plot <- col_plot +
  labs(
    title = title, subtitle = st, caption = cap,
    x = "",
    y = "% change in estimated number of\nopen museums since
     ↪ 1960**"
  )
```

2.4.3 Adjusting themes

Now we need to edit the `theme()` elements to apply the text fonts and styles, edit the background colors, and remove the legend.

In {ggplot2}, the axis limits are chosen automatically based on the range of the data. It's often useful to chose limits (and breaks) that are *nice* - it makes it easier to calculate where other values are. We can set `scale_y_continuous(limits = c(0, 300))` to make the range of the y-axis between 0 and 300. We can also set `coord_cartesian(expand = FALSE)` to remove the extra space around the plot area that is added by default - giving a slightly cleaner look. Using `theme_minimal()` as a base, we can set the default font size and family using `base_size = 7` and `base_family = body_font`.

Setting `legend.position = "none"` removes the legend on the right hand side. The `plot.title.position = "plot"` and `plot.caption.position = "plot"` arguments make sure that the title, subtitle, and caption text are all aligned with the left side of the plot area. The default is to align with the edge of the panel area (the area shaded grey by default), which doesn't look good when you have long axis labels, or a multi-line axis title on the y-axis.

The `panel.spacing` argument controls how close the facets are to each other - this can help to stop the year labels on side-by-side plots from overlapping. Setting `plot.margin = margin(10, 15, 10, 10)` adds some extra space around the outside of the plot, with the higher value on the right hand side compensating and balancing out the space from the axis title on the left hand side. The `plot.background` and `panel.background` arguments set the background color of the plot and panel areas to be the `bg_col` variable previously defined.

For the `plot.title`, `plot.subtitle`, and `plot.caption`, arguments, we use `element_textbox_simple()` from {ggtext} to make sure that any Markdown or HTML syntax is processed, and to automatically wrap the text in the subtitle and caption. The text is all left-aligned, with a lineheight of `0.5`, and set to use the `text_col` variable for the color. The title font is made slightly larger and in bold. The `axis.text` is also set to be `text_col` colored with with a lineheight of `0.5` using `element_text()`.

Now, we have our final plot:

```
library(ggtext)
text_plot +
  scale_y_continuous(limits = c(0, 300)) +
  coord_cartesian(expand = FALSE) +
  theme_minimal(base_size = 7, base_family = body_font) +
  theme(
    legend.position = "none",
    plot.title.position = "plot",
    plot.caption.position = "plot",
    panel.spacing = unit(1, "lines"),
    plot.margin = margin(10, 15, 10, 10),
    plot.background = element_rect(
      fill = bg_col, color = bg_col
    ),
    panel.background = element_rect(
      fill = bg_col, color = bg_col
    ),
    plot.title = element_textbox_simple(
      color = text_col,
      lineheight = 0.5,
      size = rel(1.2),
      face = "bold",
      margin = margin(b = 5)
    ),
    plot.subtitle = element_textbox_simple(
      color = text_col,
      lineheight = 0.5
    ),
    plot.caption = element_textbox_simple(
      hjust = 0,
      color = text_col,
      lineheight = 0.5
    ),
    axis.text = element_text(
      color = text_col,
      lineheight = 0.5
```

```
    )
)
```

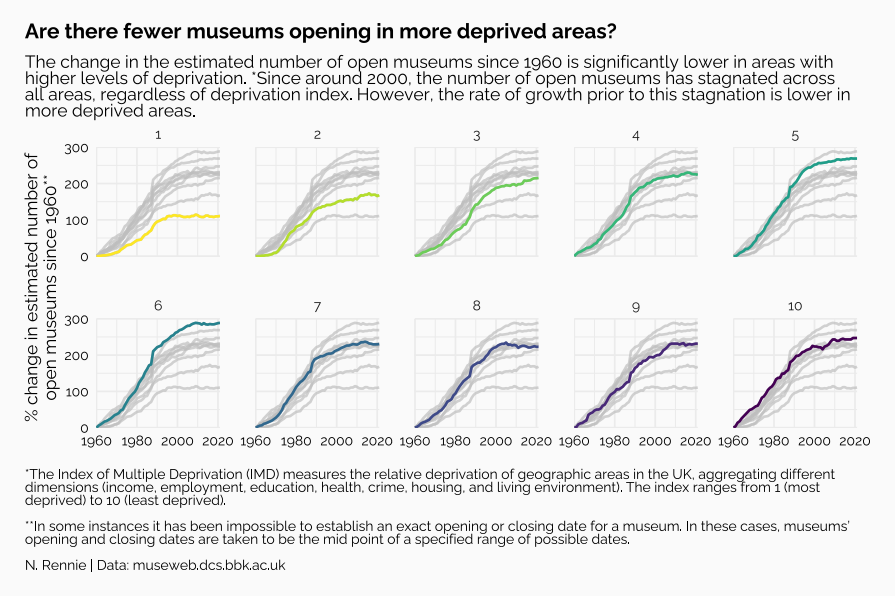


Figure 2.8: A styled version of the previous plot - with a custom font, colored background, and better spacing.

2.5 Reflection

Is there anything that could still be improved about this plot? There are two main elements that jump out:

- The axis text denoting the years only appears on the bottom row of faceted plots. This makes it a little bit tricky to see what's going on in the first row without first looking at the years in the row below. It's a fairly minor point, but adding year labels to the top row would just make it easier for a reader.
- We removed the legend from the plot with the reason that *the colors are based on the levels of deprivation, which are already labelled on the facet titles*. This same argument could be used to remove the colors altogether. There's no need to use color here to denote the deprivation levels when the deprivation levels are given by the facet titles. Especially since the yellow color is harder to see against the pale background color. Using a stronger

color e.g. red for all highlighted lines, would be better for making the lines stand out, and reduce confusion about what the colors mean.

Part II

Visualising spatial data

3

Doctors in an ageing population: making maps with {ggplot2}

In this chapter we'll learn how to identify open data sources, make maps with {ggplot2} using data from the {maps} package, and create title panels with an unorthodox use of facets.

3.1 Data

There's often a *Bring Your Own Data* week each year of #TidyTuesday (R4DS Online Learning Community 2023), where participants are encouraged to source their own data. Some use their own data - visualising how many times they've gone for a run over the past year, or recreating GitHub contributions graphs. Others choose to find and visualise other sources of data. So where do you find publicly available data?

There are many open sources of data, covering a wide range of topics, time frames, and regions across the world. Some government organisations have data portals, some compaies have APIs you can access, some academic papers have accompanying data, or the Google dataset search engine (datasetsearch.research.google.com) might also help you to identify data you're interested in.

One fantastic source of data is the Our World in Data website (ourworldindata.org). The aim of Our World in Data, according to their website, is to *publish the research and data to make progress against the world's largest problems*. There are datasets on everything from energy and environment, to poverty and education, to name a few. Their website also has many examples of beautiful, effective data visualisations if you're ever looking for inspiration.

For this chapter, we'll visualise data on Medical doctors per 1,000 people, a dataset which comes from the Our World in Data website (Our World in Data 2019).

```
doctors <- readr::read_csv("data/doctors.csv")
```

3.2 Exploratory work

Let's start by exploring the data to see if there are interesting patterns that can be visualised.

3.2.1 Data exploration

The data is reasonably small, containing only 4 columns: `entity` (denoting the country or a larger region), `code` (the country code), `year` (year the data relates to), and `Physicians (per 1,000 people)`. The 4682 rows of data cover 221 different regions (some aggregates of others), with data covering rlength(unique(doctors$year))‘ years.

```
head(doctors)
```

```
# A tibble: 6 x 4
  entity      code   year `Physicians (per 1,000 people)`
  <chr>       <chr> <dbl>                           <dbl>
1 Afghanistan AFG    1960                           0.035
2 Afghanistan AFG    1965                           0.063
3 Afghanistan AFG    1970                           0.065
4 Afghanistan AFG    1981                           0.077
5 Afghanistan AFG    1986                           0.183
6 Afghanistan AFG    1987                           0.179
```

Often when there's a time component to data, one of the most obvious patterns to consider is how other variables change over time. Although line charts are probably most common for visualising time series data, a simple scatter plot can also indicate if there's a general trend in the data. Sometimes scatter plots also look cleaner than line charts - a line for each region in this chart would very much look like a *spaghetti chart* as discussed in Chapter 2.

```
plot(
  x = doctors$year,
  y = doctors$`Physicians (per 1,000 people)`,
  xlab = "Year", ylab = "Physicians per 1,000 people")
```

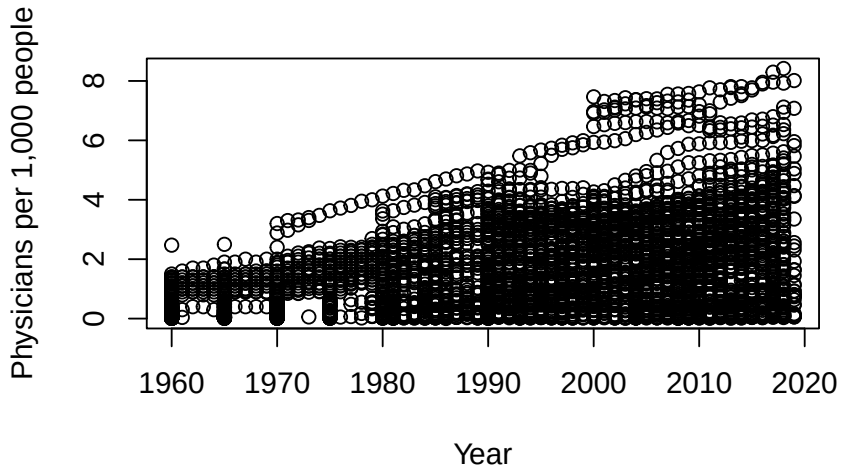


Figure 3.1: Simple non-styled scatterplot of number of physicians per 1,000 people over time showing an increasing trend, created in base R.

There seems to be a general increasing trend between 1960 and 2019. The other important component of this data that we may want to explore is the spatial aspect - is there a pattern over space as well as over time? The most common approach to visualising spatial data is, of course, to plot it on a map. If the aim is to show how a variable changes across different countries (or other defined regions), it's very common to color the country based on the value of the variable. These are often termed *choropleth maps*.

3.2.2 Exploratory sketches

At this point, it's also often a good time to start thinking about the orientation and aspect ratio of the plot you'll create. This will depend a lot of where the plot is going to end up - for example, plots in a single column academic article will typically be landscape graphs. The choice of orientation and aspect ratio can also affect how clearly your data is displayed - choosing a very wide plot for time series data can stretch the series and obliterate any appearance of trend. With maps, you're a little bit more constrained because there is already an underlying aspect ratio in the plot you're creating. For this map, a landscape orientation with a 6x4 aspect ratio should work reasonably well.

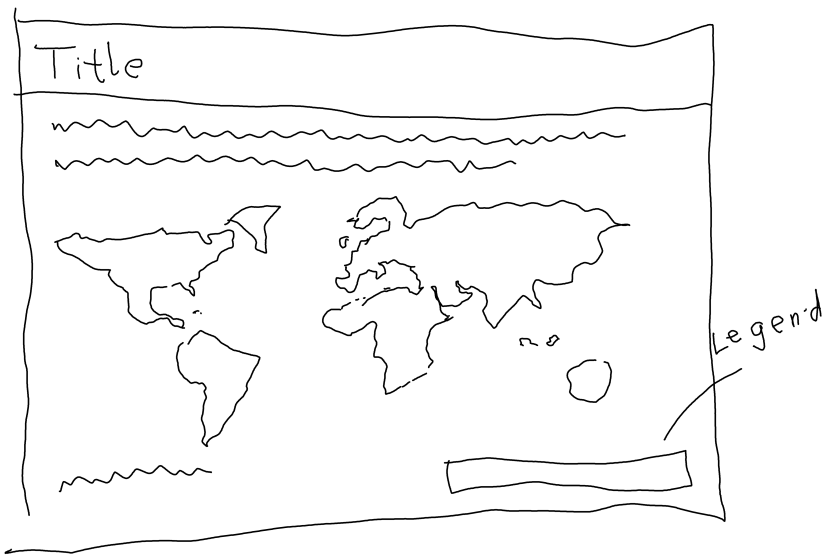


Figure 3.2: Initial sketch of a map of the world, showing title enclosed in a box and legend positioned horizontally in the bottom right

3.3 Preparing a plot

To create the map sketched out in Figure 3.2, we need to do two things (i) decide which data to plot: which regions, and which years; and (ii) source some spatial data beyond just region names and country codes.

3.3.1 Data wrangling

Since this data is already fairly tidy, there isn't too much data wrangling to be done. The only processing we really need to do is getting rid of the data we don't need, and renaming a couple of columns to make them easier to work with. We can use the `rename()` function from {dplyr} to rename the `entity` column to `region` (for reasons that will become clear in the next paragraph!). We also rename the `Physicians (per 1,000 people)` column to `doctors` to make it easier to work with. The data has multiple entries for each country, spanning different years. We *could* make an animated map to show how the number of doctors is changing over time, but for now we'll keep it simple with a static map showing a snapshot at one point in time. However, there's a bit of a problem. If you inspect the data, you'll see that not every country has an entry for each year - let's use the most recent data available for each country.

For each `region`, we keep the row with only the most recent year using a combination of `group_by()` and `slice_max()` from {dplyr}.

```
doctors <- doctors |>
  dplyr::rename(
    region = entity,
    doctors = `Physicians (per 1,000 people)`) |>
  dplyr::select(region, year, doctors) |>
  dplyr::group_by(region) |>
  dplyr::slice_max(year) |>
  dplyr::ungroup()
```

To plot this data on a world map, we also need data for the country borders. Luckily, the `map_data()` function built into {ggplot2} can help us with that! This function takes data from the {maps} package and turns it into an object you can plot directly with {ggplot2}.

```
library(ggplot2)
world <- map_data("world")
```

Of course, it's never quite that straightforward. We need to join the `world` map data to our `doctors` data, and to do that we need a column in each data sets to join by - we'll use the `region` column. If you try to join these two data sets using the `region` column, you'll notice that you end up with some unexpected `NA` values. So what's going on?

> You don't need to rename columns in your data to be able to join them, but for this example, I found it a little bit easier to work with the data after renaming `entity` to `region`.

There are two issues here. Firstly, there are more regions in the `world` data than there are in the `doctors` data:

```
length(unique(world$region))
```

```
[1] 252
```

```
length(unique(doctors$region))
```

```
[1] 221
```

This is partly due to the fact that the `doctors` data has implicitly missing values - if no data is available for a region, no rows exists in the data for that region. It isn't listed with `NA` values. Note that there are also some *regions* in `doctors` which do not exist in `world` - for example, the entity `"Upper-middle-income countries"` is listed within `doctors`.

Secondly, if you inspect the region names, you'll see that for some countries, their names are encoded differently. For example, in the `world` data, the

United States is listed as `"USA"` whilst in the `doctors` data, it's listed as `"United States"`. Here, the easiest thing to do is manually rename the values that differ in one of the datasets. We can use the `recode()` function from {dplyr} to do that. Note that `recode()` has the rather unusual (for the {tidyverse}) syntax of `old_name = new_name`:

```
plot_data <- doctors |>
  dplyr::mutate(
    region =
      dplyr::recode(region,
        "United Kingdom" = "UK",
        "United States" = "USA",
        "Democratic Republic of Congo" = "Democratic Republic of
        ↪  the Congo",
        "Cote d'Ivoire" = "Ivory Coast",
        "Congo" = "Republic of Congo",
        "Czechia" = "Czech Republic"
      )
  )
```

The entries in `region` column of `doctors` that don't correspond to countries e.g. `"Upper-middle-income countries"` are not values that are required for the map. Therefore a `left_join()` can be performed, with `world` on the left - keeping all the countries listed in `world` and joining only those with a corresponding value in `doctors`. The remaining countries in `world` with no match in `doctors` are listed with `NA` values. The rows for `"Antarctica"` are filtered out - Antarctica is often given a disproportionate amount of space on world maps (at least those not centered on Antarctica) in the process of projecting a sphere onto a rectangle.

```
map_data <- dplyr::left_join(world, plot_data, by = "region") |>
  dplyr::filter(region != "Antarctica")
```

Now, we have everything we need to create a simple map.

3.3.2 The first plot

We start, as almost always, with the `ggplot()` function, and pass in the data and aesthetic mappings that will apply to the whole plot. The longitude (`long`) and latitude (`lat`) are passed to the `x` and `y` axes; and we specify that the `fill` color of each country should be based on the `doctors` column. We also specify `map_id` - aesthetic mapping that isn't seen as often as the others. This is used to tell `geom_map()` which column defines (not entirely unlike the `group` aesthetic discussed in previous chapters).

Both `geom_sf()` and `geom_map()` are used for creating maps within {ggplot2}.

However, they expect different formats of data: `geom_sf()` expects an `sf` object, whereas `geom_map()` works with coordinates as columns in a `data.frame` or `tibble`. Here, we'll use `geom_map()`. For examples of using `geom_sf()`, see Chapter ??.

> Note: you could also use `geom_polygon()` to plot `map_data` instead of `geom_map()`.

```
base_plot <- ggplot(
  data = map_data,
  mapping = aes(
    x = long,
    y = lat,
    map_id = region,
    fill = doctors
  )
) +
  geom_map(map = map_data)
base_plot
```

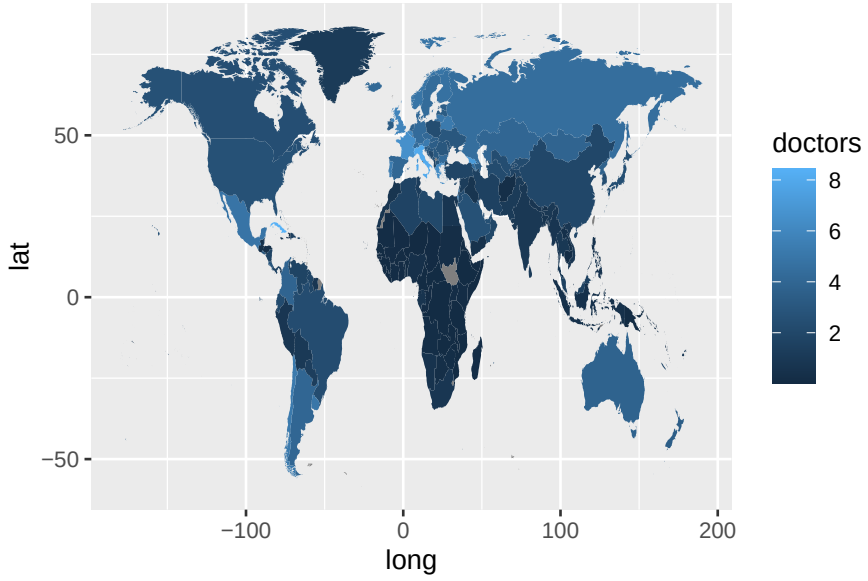


Figure 3.3: Map of the world with countries colored based on number of doctors per 1,000 people. The map looks stretched.

So we have a simple map that shows our data, but there are several problems with it:

- The map looks as if someone was stretched in vertically, since there's no map projection specified. Countries are still recognisable, but not quite the right shape.
- The color palette is not ideal. It's more intuitive for brighter or lighter colors to represent smaller values, and for darker colors to represent higher values - at least for light colored backgrounds (Schloss et al. 2019). The default gradient color scale in {ggplot2} is the opposite way around.
- There are labels that don't need to be there (`lat` and `long`), and missing labels that should be there (`title` and `subtitle`, for example).

So let's fix those elements of the initial plot.

3.4 Advanced styling

We'll start by considering alternative color palettes, then think about text that should be added, before finalising the layout.

3.4.1 Colors

There are many, many color palette R packages in existence, and even more outwith the R ecosystem. In fact, the {paletteer} package is designed give a common interface to a comprehensive collection of color palettes in R. One of my favourite is color palette R packages is (Mills 2022) - a collection of color palettes inspired by works of art at the Metropolitan Museum of Art in New York. It has many beautiful palettes, and many that work in traditional data visualisations. You can view all available palettes using `MetBrewer::display_all(colorblind_only = TRUE)`. Since `doctors` is a continuous variable, we'll look at the sequential palettes only.

{MetBrewer} does have functions that interface directly with {ggplot2} (such as `scale_fill_met_c()`) but we're going to use some of the colors in the palette to also define variables for the highlight and text colors. To get a good range of colors, we extract 20 colors from the `"Hokusai2"` palette. The `text_col` is the 18th color and `highlight_col` is the 15th color. A variable containing the background color, `bg_col`, is also defined.

```
library(MetBrewer)
col_palette <- met.brewer("Hokusai2", n = 20)
text_col <- col_palette[18]
highlight_col <- col_palette[15]
bg_col <- "#EADEDA"
```

These colors can then be passed into `scale_fill_gradientn()` from {gg-

plot2}. The limits of the color scale can also be set. Rather than adding labels for values on the legend, we can add text labels for *Fewer doctors* and *More doctors*. These are position 0.8 in from the limits of the color scale.

```
col_plot <- base_plot +
  scale_fill_gradientn(
    colors = col_palette,
    limits = c(0, 10),
    breaks = c(0.8, 9.2),
    labels = c("Fewer doctors", "More doctors"))
col_plot
```

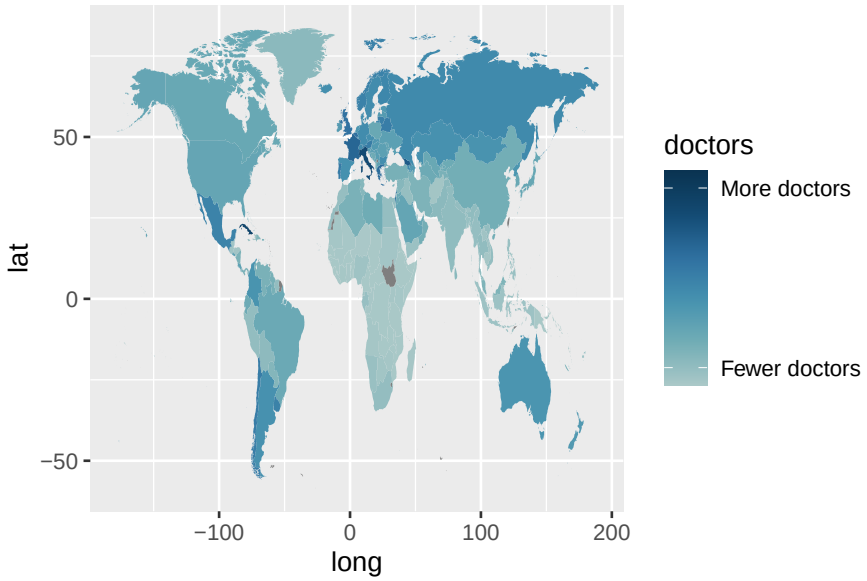


Figure 3.4: Map of the world with countries colored based on number of doctors per 1,000 people, showing a different color scheme.

3.4.2 Text and fonts

Now, we can define some text for the title, subtitle, and caption. As in previous chapters, we'll be using {ggtext} for formatting which means we can use markdown syntax to add bold font and line breaks.

```
title <- "Doctors in an ageing population"
st <- "This map show the number of doctors per thousand people,
 ↪  revealing which countries* may be more likely to struggle in
 ↪  providing care for a population.<br><br>*using the most
 ↪  recent available data for each country."
```

```
cap <- "**Data**: Our World in Data | **Graphic**: N. Rennie"
```

These text variables can then be passed into the `labs()` function in {ggplot2}:

```
text_plot <- col_plot +
  labs(title = title, subtitle = st, caption = cap)
```

Fonts can also be defined using the {sysfonts} and {showtext} packages. Here, the *Roboto* font is loaded through Google Fonts for the main font used, and *Roboto Slab* is loaded for use in the title.

```
sysfonts::font_add_google(name = "Roboto", family = "roboto")
sysfonts::font_add_google(name = "Roboto Slab", family =
 ↪  "roboto_slab")
showtext::showtext_auto()
showtext::showtext_opts(dpi = 300)
body_font <- "roboto"
title_font <- "roboto_slab"
```

To apply these fonts to the plot, the `theme` elements need to be adjusted.

3.4.3 Adjusting themes

We start by adding `theme_void()` from {ggplot2}. The `theme_void` function removes all `theme` elements - including grid lines, axis labels, and the background. The legend and specified titles and subtitles remain. This theme is especially useful for maps where it's more common for axis lines, axis titles, and grid lines not to be displayed. Like other built-in theme options, we can still set the `base_size` and `base_family` to set the default size and font family for any text that is displayed.

We also set the `plot.title`, `plot.subtitle`, and `plot.caption` to use `element_textbox_simple` from {ggtext} to allow the markdown syntax and automatically wrap long subtitles as we've seen in previous chapters.

```
library(ggtext)
text_plot +
  theme_void(base_size = 8, base_family = body_font) +
  theme(
    plot.margin = margin(10, 10, 10, 10),
    plot.background = element_rect(
      fill = bg_col, color = bg_col
    ),
    panel.background = element_rect(
      fill = bg_col, color = bg_col
    ),
```

```
    plot.title = element_textbox_simple(
      color = text_col,
      family = title_font
    ),
    plot.subtitle = element_textbox_simple(
      color = text_col
    ),
    plot.caption = element_textbox_simple(
      hjust = 0,
      color = text_col
    )
  )
```

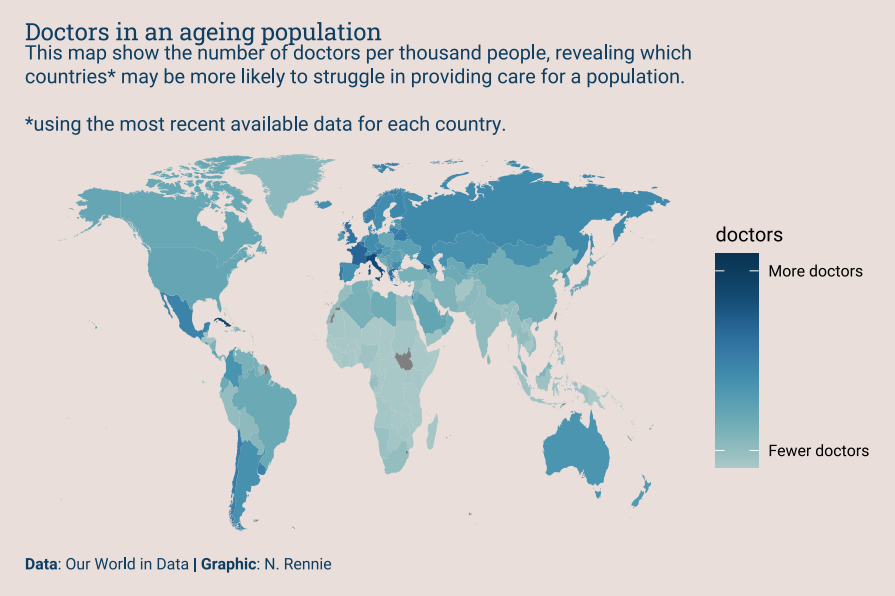


Figure 3.5: Map of the world with countries colored based on number of doctors per 1,000 people. The map looks stretched, and the fonts in the title and subtitle are close together.

One of the aesthetic design choices we might want to make here, is to include the title within a banner with a different colored background. Although this might seem like a fairly straightforward thing to want, it's actually not that easy with {ggplot2}. There are some solutions to the problem using packages like {cowplot} or {grid} to draw rectangles and text. But we can do this within {ggplot2}, but using facets in a way they were not designed to be used. Let's start by adding an additional column called `label` to `map_data` that contains

the title (for every row in the data):

```
map_data$label <- title
```

> Note: this column does not need to be called `label`, you can use any name you choose as long as it's not an existing column.

Then we can use `facet_wrap()` and facet across the `label` column. Since there's only one value of `label` in the data, this just adds the title as strip text at the top of the plot. While we're here, let's make the country outlines in the map the same color as the background, and make the lines a little bit thicker.

```
styled_plot <- ggplot(
  data = map_data,
  mapping = aes(
    long,
    lat,
    map_id = region,
    fill = doctors
  )
) +
  geom_map(
    map = map_data,
    color = bg_col,
    linewidth = 0.3
  ) +
  scale_fill_gradientn(
    colors = col_palette,
    limits = c(0, 10),
    breaks = c(0.8, 9.2),
    labels = c("Fewer doctors", "More doctors")
  ) +
  labs(title = title, subtitle = st, caption = cap) +
  facet_wrap(~label) +
  theme_void(base_size = 7, base_family = body_font) +
  theme(
    plot.margin = margin(10, 10, 10, 10),
    plot.background = element_rect(
      fill = bg_col, color = bg_col
    ),
    panel.background = element_rect(
      fill = bg_col, color = bg_col
    ),
    plot.title = element_textbox_simple(
      color = text_col,
```

```
      family = title_font,
      lineheight = 0.5
    ),
    plot.subtitle = element_textbox_simple(
      color = text_col,
      lineheight = 0.5
    ),
    plot.caption = element_textbox_simple(
      hjust = 0,
      color = text_col,
      lineheight = 0.5
    ),
    strip.background = element_rect(
      fill = highlight_col, color = highlight_col
    )
  )
styled_plot
```

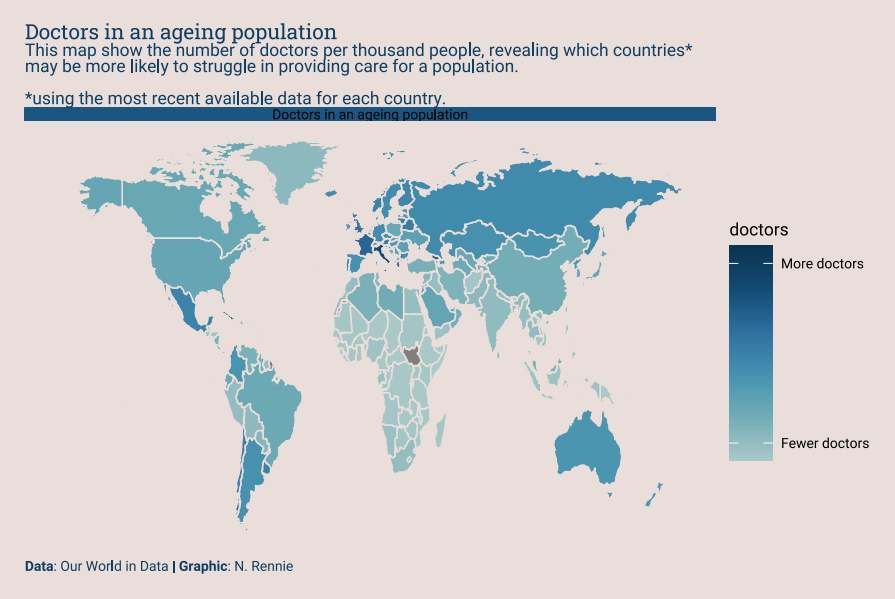


Figure 3.6: Map of the world with duplicated title in the facet strip text, with dark text against a dark blue background.

It's obvious that there are some issues with this plot now that the strip text contains the title:

- There is a duplicate title set using `labs()`, and the `strip.text` title is

too dark to read against the blue background.
- The subtitle is above the title.
- The map still appears stretched.
- The legend is taking up quite a lot of space, and the white ticks in the colorbar are distracting.

To solve the first two problems, the `title` and `subtitle` arguments of `labs()` can be set to `NULL` to remove them from the plot. Instead, the `tag` argument in `labs()` can be used to set the subtitle. The nice thing about using `tag` is the `plot.tag.position` argument withing `theme()` which allows you to position the text anywhere on the plot. The `strip.text` and `plot.tag` arguments of theme should also be set using `element_textbox_simple()` from {ggtext} to allow the text to be styled as we wish. The top, left, and right margins of the plot should be set to `0` using `plot.margin` to make sure that the strip text banner goes to the edge of the plot.

To solve the third problem of the map looking stretched, we can apply `coord_sf()` which applies the World Geodetic System 1984 (WGS84) CRS (coordinate reference system). The upper limit of the `y` axis can also be extended beyond the range of the data to make room for the subtitle added using `tag`.

```
styled_plot2 <- styled_plot +
  labs(
    title = NULL, subtitle = NULL, tag = st
  ) +
  # add space for the tag (subtitle) text
  coord_sf(ylim = c(-60, 140)) +
  theme(
    # move and format the tag (subtitle) text
    plot.margin = margin(0, 0, 5, 0),
    plot.tag.position = c(0.015, 0.8),
    plot.tag = element_textbox_simple(
      color = text_col,
      lineheight = 0.6,
      hjust = 0,
      maxwidth = 0.98
    ),
    # add margin for caption
    plot.caption = element_textbox_simple(
      hjust = 0,
      color = text_col,
      margin = margin(l = 5),
      lineheight = 0.6
    ),
    # change title text color
```

```
    strip.text = element_textbox_simple(
      color = bg_col,
      family = title_font,
      margin = margin(7, 5, 7, 5),
      lineheight = 0.6,
      size = rel(1.7)
    )
  )
```

> Note: the values used in `plot.tag.position = c(0.015, 0.8)` and the `strip.text` argument `margin = margin(7, 5, 7, 5)` took a lot of trial and error to get *just right*. There's no magic involved in choosing these values!

To solve the final problem of the legend appearance, we can edit the style elements in `theme`. The `legend.title` is removed by setting it to a blank element with `element_blank()`, and the legend text labels are styled with `element_text()`.

The placement of the legend is set through the `legend.position`, `legend.justification.bottom`, `legend.margin`, and `legend.direction` arguments.

> Since version 3.5.0 of {ggplot2}, you can also style the `theme` of individual legends from inside the `guide_*()` functions. Prior to version 3.5.0, `legend.position` was used to position the legend inside the plot before `legend.position.inside` and `legend.justification.bottom` were introduced to allow custom legend positions more easily. The other difference is that `legend.ticks = element_blank()` can be used to remove the white tick marks inside the colorbar legend. In older versions of {ggplot2}, `guides(fill = guide_colorbar(ticks = FALSE))` could be used instead.

The size of the legend is controlled through the `legend.key.width` and `legend.key.height` arguments.

```
styled_plot2 +
  theme(
    # legend text
    legend.title = element_blank(),
    legend.text = element_text(
      color = text_col,
      lineheight = 0.5,
      hjust = 0.5
    ),
    # legend size
```

```
    legend.key.width = unit(1.5, "cm"),
    legend.key.height = unit(0.3, "cm"),
    # legend position
    legend.position = "bottom",
    legend.justification.bottom = "right",
    legend.margin = margin(0, 5, -5, 0),
    legend.direction = "horizontal",
    legend.ticks = element_blank()
  )
```

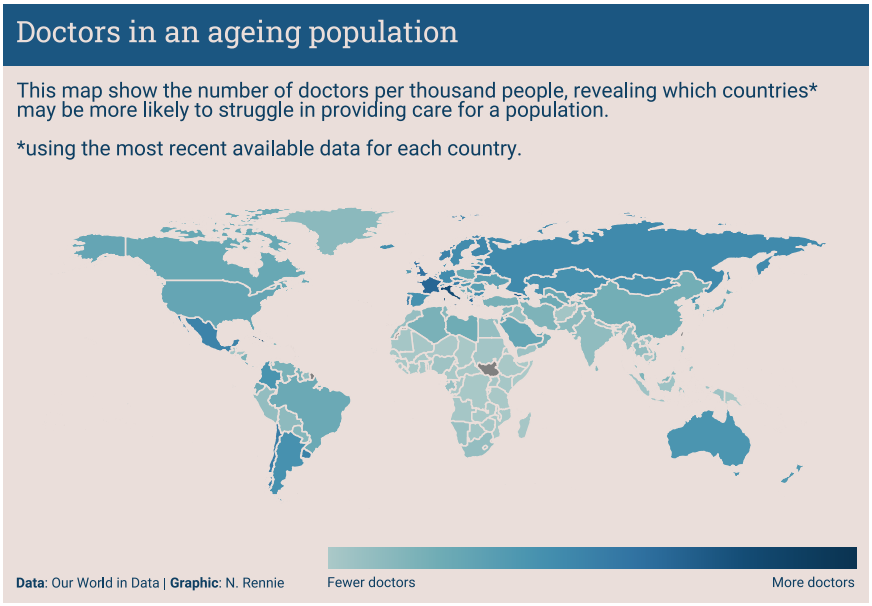


Figure 3.7: Final styled plot showing a map of the world, with title inside a blue banner and a horizontal colorbar legend at the bottom right of the page.

3.5 Reflection

When the original plot of this data was created, it plotted the number of doctors per 1,000 people over the age of 70 and the colors were based on the log of this value - compare Figure 3.7 and Figure 3.8. The raw values on the color scale were hard to interpret, so the choice was made to use *Fewer doctors* and *More doctors* labels instead. For this version, where the underlying data is simpler and easier to interpret, having the values on the legend would add

more useful information. The original title was also kept, but could probably be changed to something more informative.

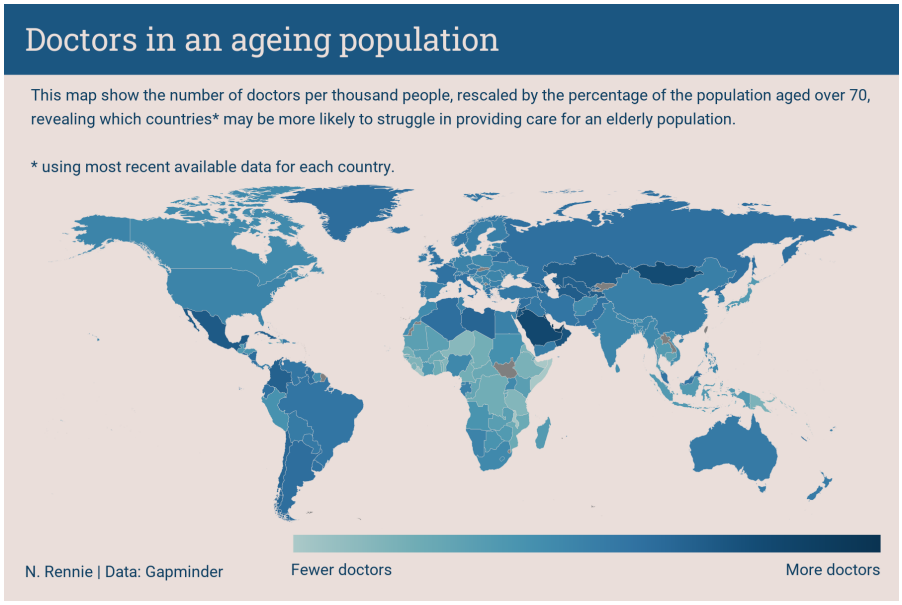


Figure 3.8

When the data was processed, the choice was made to plot a map showing the values for the most recently available data. This means that for some countries the data is more recent (and therefore perhaps more reliable), whilst for others it's much older. In fact, running `range(doctors$year)` shows that the most recent data in the plot is from 2019, whilst the oldest is from 1980 - a gap of almost 40 years! That makes it much harder to accurately compare between countries, and there's no indication for each country on this map how recent the data is. Readers might end up drawing conclusions that show differences between countries, when actually the different is between times. Showing uncertainty on maps is tricky, and there's no straightforward solution here. Perhaps setting the colors to a binary scale showing whether he value is above or below average, with the intensity of the color denoting the recentness of the data, is one approach. Or at least a more detailed explanation about the range of time the data relates to could be included within the subtitle.

One improvement to this map over the original is the use of `coord_sf()` to *correctly* scale the aspect ratio of the map. In the original map, using `coord_map()` (now deprecated in favour of `coord_sf()`) produced some odd looking results, and in the end, after applying the rest of the styling changes, the end result didn't look too squashed. But `coord_sf()` made it much easier to achieve.

Part III

Weird and wonderful: completely custom charts

4

Technology adoption: making gauge charts with {ggforce}

In this chapter we'll discover how to create gauge charts, a type of chart not native to {ggplot2}, with the help of the {ggforce} extension package.

4.1 Data

The *technology adoption* (Comin and B. 2004) data comes from

```
technology <- readr::read_csv("data/technology.csv")
```

The `technology` data is reasonably large with 491636 rows and 7 columns. The data is in long format, and

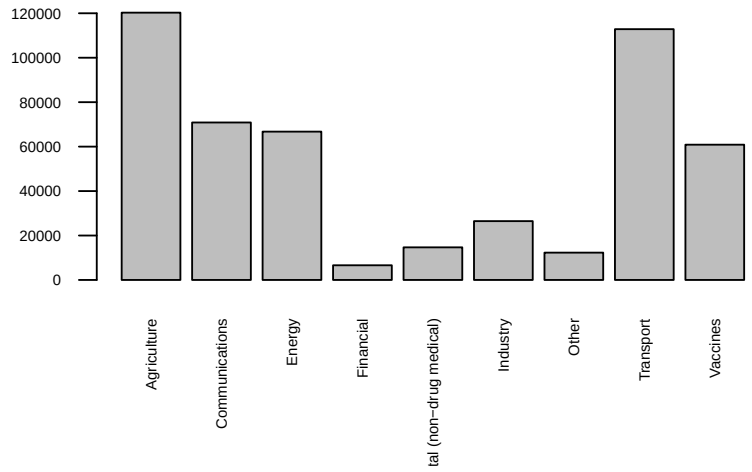
4.2 Exploratory work

What might be an interesting aspect of this data to visualize?

4.2.1 Data exploration

As in other chapters in this book, we'll start with some basic exploratory plots in base R. For example, we may look at the distribution of variables in each `category` using the `barplot` function:

```
barplot(
  table(technology$category),
  las = 2,
  cex.axis = 0.5,
  cex.names = 0.5
)
```



some initial base R exploratory plots

Let's look at something more specific

A list of all ISO3 country codes can be obtained by running `unique(technology$iso3c)`, and from there we can choose a subset of countries we want to look at in more detail. Let's look at Great Britain, USA, Sweden, Brazil, New Zealand, and Venezuela, and store these choices in a vector called `countries`. We also need to narrow down the data we want to consider - there are 194 different questions contained in the data. You can check by running `length(unique(technology$label))`. One variable we will consider further is the percentage of children who received a measles immunization - indicated by the `"pctimmunizmeas"` level in the `variable` column.

Although including all years of data would better allow us to consider trends in the values, sometimes looking at a only a few snapshots can be more effective. For example, by considering only the years 1980 and 2010 as we'll do here, readers get a *Wow, look how much things have changed!* message rather than the perhaps less impactful visual of a gradual trend. We can use the `filter` function from {dplyr} to filter our `technology` data set to consider only the rows showing data about percentage of children who received a measles immunization, in the years 1980 or 2010, and relating to countries in our specified vector of `countries`.

We no longer need the `group`, `category`, `variable`, or `label` columns, as these

are constant for our data so we can remove these columns using `select()` from {dplyr}.

```
# subset of countries to look at further
countries <- c("GBR", "USA", "SWE", "BRA", "NZL", "VEN")

# subset data for specific topic, years, and countries
measles_data <- technology |>
  dplyr::filter(
    label == "% children who received a measles immunization",
    year %in% c(1980, 2010),
    iso3c %in% countries
  ) |>
  dplyr::select(-c(group, category, variable, label))
head(measles_data)
```

```
# A tibble: 6 x 3
  iso3c  year value
  <chr> <dbl> <dbl>
1 BRA    1980    57
2 BRA    2010    99
3 GBR    1980    53
4 GBR    2010    89
5 NZL    1980    80
6 NZL    2010    91
```

Our tidier data now shows just the percentage (`value`) of children who received a measles immunization in each country (`iso3c`), in each of 1980 and 2010 (`year`), How might we visualize this data?

There are a couple of obvious options that come to mind: a simple grouped bar chart, a slope chart, or indeed the (not often popular) pie chart Our choice of data visualization will depend on which aspects of the data we want to show. Do we want to compare 1980 to 2010? Do we want to compare countries to each other? Do we just want to show the range of values in the data? In this data, the most interesting example is a comparison between 1980 and 2010. Although a slope chart would likely work well for this data, we're going a little bit more experimental with a gauge chart.

At the time of writing, there isn't a built-in function in {ggplot2} to create gauge charts. If you've never heard of a gauge chart, this initial sketch might give you an idea of what we're aiming for.

4.2.2 Exploratory sketches

Before ...

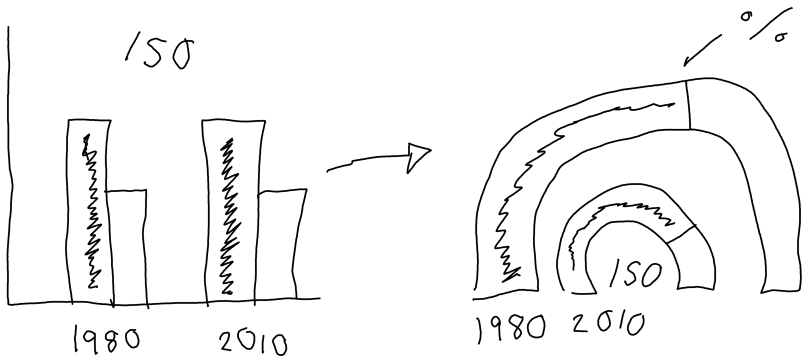


Figure 4.1

4.3 Preparing a plot

4.3.1 Data wrangling

```
plot_data <- measles_data |>
  dplyr::mutate(no_value = 100 - value) |>
  tidyr::pivot_longer(
    cols = c(value, no_value),
    names_to = "YN",
    values_to = "perc"
  ) |>
  tidyr::pivot_wider(names_from = "year", values_from = "perc")
  ↪ |>
  dplyr::mutate(YN = factor(YN)) |>
  dplyr::mutate(
    perc_1980 = `1980` / 100,
    perc_2010 = `2010` / 100
  ) |>
  dplyr::select(-c(`1980`, `2010`)) |>
  dplyr::group_by(iso3c) |>
  dplyr::mutate(
    ymax_1980 = cumsum(perc_1980),
    ymax_2010 = cumsum(perc_2010)
  )
head(plot_data)
```

```
# A tibble: 6 x 6
# Groups:   iso3c [3]
  iso3c YN       perc_1980 perc_2010 ymax_1980 ymax_2010
  <chr> <fct>        <dbl>     <dbl>     <dbl>     <dbl>
1 BRA   value         0.57      0.99      0.57      0.99
2 BRA   no_value      0.43      0.01      1         1
3 GBR   value         0.53      0.89      0.53      0.89
4 GBR   no_value      0.47      0.11      1         1
5 NZL   value         0.8       0.91      0.8       0.91
6 NZL   no_value      0.2       0.09      1         1
```

4.3.2 The {ggforce} extension package

The {ggforce} extension package (Pedersen 2022) contains many useful functions which extend the behavior of {ggplot2}, many of them aimed at exploratory data visualisation. We won't cover many of it's function in this chapter, and instead we'll focus on how to use it to create gauge charts.

{ggforce} is available on CRAN and can be installed with the usual `install.packages("ggforce")` command.

4.3.3 Gauge charts with {ggforce}

`geom_arc_bar()`

4.3.4 Reformatting data

gauge data

this changed from original mutate_at

```
gauge_data <- plot_data |>
  dplyr::ungroup() |>
  dplyr::mutate(
    ymin_1980 = c(rbind(
      rep(0, length(countries)),
      (dplyr::slice_head(plot_data, n = -1) |>
        dplyr::pull(ymax_1980))
    ))
  ) |>
  dplyr::mutate(
    ymin_2010 = c(rbind(
      rep(0, length(countries)),
      (dplyr::slice_head(plot_data, n = -1) |>
        dplyr::pull(ymax_2010))
    ))
```

```
  ) |>
  dplyr::group_by(iso3c) |>
  dplyr::mutate(
    dplyr::across(
      dplyr::starts_with("y", ignore.case = FALSE),
      ~ scales::rescale(.,
        to = pi * c(-0.5, 0.5),
        from = 0:1
      )
    )
  )
```

4.3.5 The first plot

basic plot

```
library(ggplot2)
basic_plot <- ggplot(data = gauge_data) +
  ggforce::geom_arc_bar(
    mapping = aes(
      x0 = 0, y0 = 0,
      r0 = 0.7, r = 1,
      start = ymin_2010, end = ymax_2010,
      fill = YN
    )
  ) +
  ggforce::geom_arc_bar(
    mapping = aes(
      x0 = 0, y0 = 0,
      r0 = 0.2, r = 0.5,
      start = ymin_1980, end = ymax_1980,
      fill = YN
    )
  ) +
  facet_wrap(~iso3c, nrow = 2)
basic_plot
```

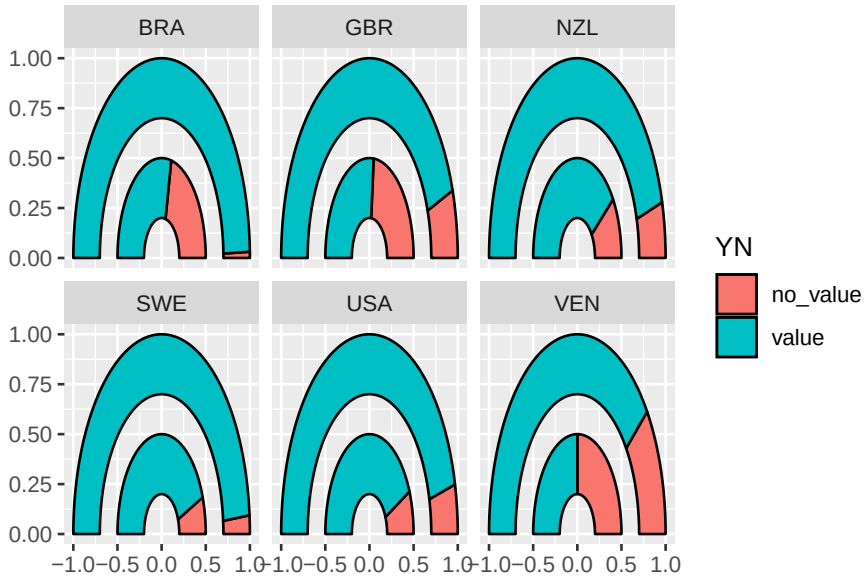


Figure 4.2: Initial plot created using `geom_arc_bar()` from {ggforce}, facetted by different countries.

4.4 Advanced styling

4.4.1 Colors

```
highlight_col <- "#990c58"
second_col <- "#949398"
bg_col <- "#dedede"
```

add new colors to basic plot

```
basic_plot <- ggplot(data = gauge_data) +
  ggforce::geom_arc_bar(
    mapping = aes(
      x0 = 0, y0 = 0,
      r0 = 0.7, r = 1,
      start = ymin_2010, end = ymax_2010,
      fill = YN
    ),
    color = second_col
```

```
  ) +
  ggforce::geom_arc_bar(
    mapping = aes(
      x0 = 0, y0 = 0,
      r0 = 0.2, r = 0.5,
      start = ymin_1980, end = ymax_1980,
      fill = YN
    ),
    color = second_col
  ) +
  facet_wrap(~iso3c, nrow = 2)
```

add scale fill

```
color_plot <- basic_plot +
  scale_fill_manual(
    breaks = c("value", "no_value"),
    labels = c("Immunised", "Not Immunised"),
    values = c(highlight_col, second_col)
  )
color_plot
```

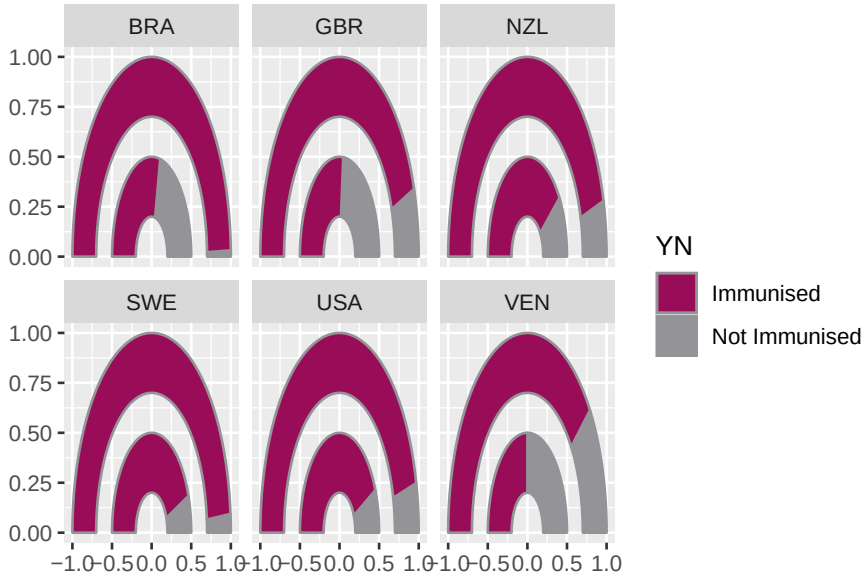


Figure 4.3: Edited version of the previous plot with colors changed from defaults to grey and dark pink.

4.4.2 Text and fonts

As we've seen in previous chapters, we can load in Google fonts using the {sysfonts} and {showtext} packages. Here, we'll keep it clean and minimal by using the `"Ubuntu"` font for both the title and the body font.

```
sysfonts::font_add_google(name = "Ubuntu", family = "ubuntu")
showtext::showtext_auto()
showtext::showtext_opts(dpi = 300)
body_font <- "ubuntu"
```

write text social to add

```
title <- "Measles Vaccinations"
subtitle <- "The inner bar represents the percentage of children
 ↪  who received a measles immunisation in 1980, whilst the
 ↪  outer bar represents the percentage in 2010. An increase in
 ↪  immunisation levels between 1980 and 2010 is seen across all
 ↪  countries.\n\nN. Rennie | Data: data.nber.org
 ↪  (10.3386/w15319)"
```

Since axis labels tend not to make too much sense for `geom-arc_bar()` plots, we'll remove them later when using the `theme` functions. Instead, we can add our own labels using `geom_text()` to the end of the gauges. To make it easier, we can construct a small `data.frame` specifically for adding text labels. This includes the `x`, and `y` coordinates where the text should be positioned (you can read these of from the graph we already have since we haven't yet deleted the axis label), as well as the `label` that should appear.

```
text_df <- data.frame(
  x = c(0.35, 0.85),
  y = c(-0.1, -0.1),
  label = c(1980, 2010)
)
```

We can then add this to the existing plot by adding a layer with `geom_text()`, noting that we need to specify the `data` argument as using the text dataframe we just created. We also need to specify the font family and size directly within the `geom_text()` function, and can add the title and subtitle text created earlier using the `labs()` function from {ggplot2}.

```
text_plot <- color_plot +
  geom_text(
    data = text_df,
    mapping = aes(x = x, y = y, label = label),
    family = body_font,
    size = 3
```

```
  ) +
  labs(
    title = title,
    subtitle = subtitle
  )
text_plot
```

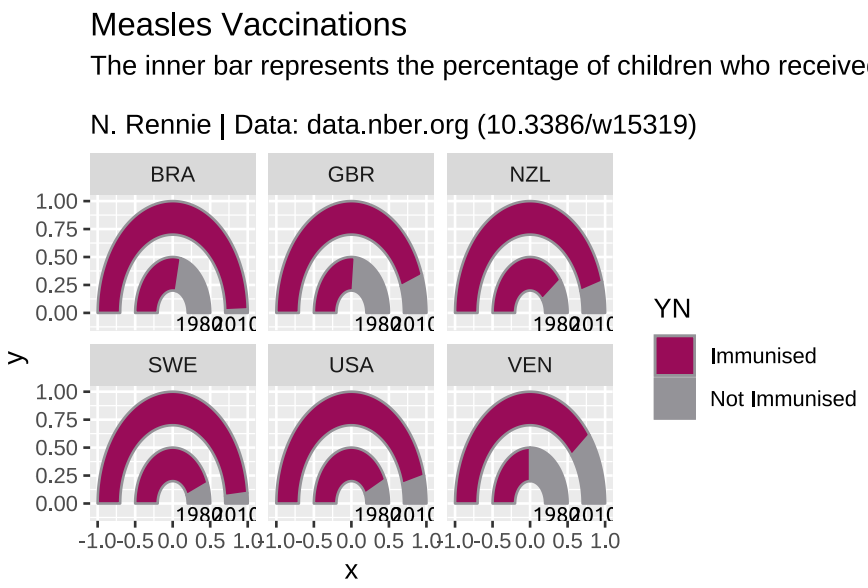


Figure 4.4: Previous plot with additional labels indicating the year on each gauge chart, as well as an added title and subtitle.

You'll notice that the subtitle text runs off the page here. Let's fix that using {ggplot2} themes and {ggtext} functions.

4.4.3 Adjusting themes

We'll start by removing all of the theme element such as the grey background, grid lines, axis labels. The easiest way to do this is using `theme_void()`. We can use the `base_family` argument of `theme_void()` to set the font family that will be used by default for any non-geom text elements that remain.

You may have noticed that the current gauge plots look a bit squashed and not exactly semi-circular. We can fix this by adding `coord_fixed()` which forces a 1:1 aspect ratio on the plot panel.

```
theme_plot1 <- text_plot +
  coord_fixed() +
  theme_void(base_size = 8, base_family = body_font)
theme_plot1
```

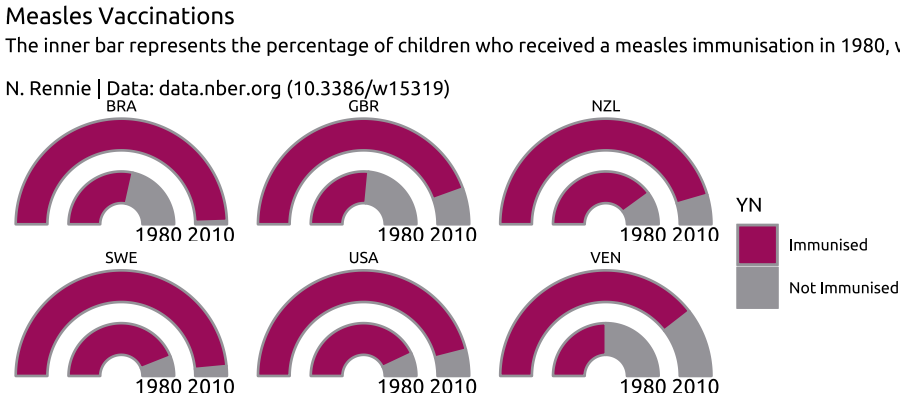


Figure 4.5: Edited version of previous plot with fixed coordinate system to prevent squashing, and all theme elements removed.

This looks better but it's still not great. What do we still need to improve with styling? The title text doesn't stand out and blends in to easily with the subtitle, similarly for the facet text. Perhaps a **bold** font would help? The subtitle text doesn't fit onto the page but we can fix that with the help of the hopefully now familiar `element_textbox_simple()` function from {ggtext} .

bg col remove legend

Let's fix the first three of these issues.

```
main_plot <- theme_plot1 +
  theme(
    legend.position = "none",
    plot.background = element_rect(
      fill = bg_col, color = bg_col
    ),
    panel.background = element_rect(
```

```
      fill = bg_col, color = bg_col
    ),
    strip.text = element_text(
      face = "bold", size = rel(1.2)
    ),
    plot.title = element_text(
      margin = margin(t = 10, b = 10),
      face = "bold",
      size = rel(1.5)
    ),
    plot.subtitle = ggtext::element_textbox_simple(
      maxwidth = 0.8,
      lineheight = 0.5,
      hjust = 0.5,
    ),
    plot.margin = margin(5, 5, 5, 5)
  )
main_plot
```

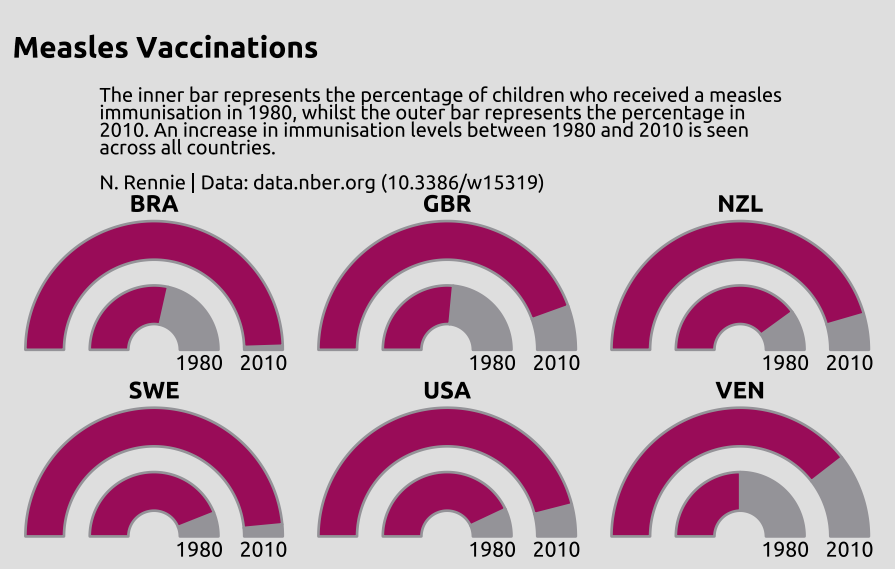


Figure 4.6: Further styling of gauge chart to change background color, prevent overlapping in the subtitle text, and increase the size of the title.

4.4.4 Adding a better legend

As we already saw in chapter ...,

```
legend_plot <- ggplot(
  data = dplyr::filter(gauge_data, iso3c == "USA")
) +
  ggforce::geom_arc_bar(
    mapping = aes(
      x0 = 0, y0 = 0,
      r0 = 0.7, r = 1,
      start = ymin_2010, end = ymax_2010,
      fill = YN
    ),
    color = second_col
  ) +
  ggforce::geom_arc_bar(
    mapping = aes(
      x0 = 0, y0 = 0,
      r0 = 0.2, r = 0.5,
      start = ymin_1980, end = ymax_1980,
      fill = YN
    ),
    color = second_col
  ) +
  geom_text(
    data = text_df,
    mapping = aes(x = x, y = y, label = label),
    family = body_font, size = 6
  ) +
  facet_wrap(~iso3c) +
  scale_fill_manual(
    breaks = c("value", "no_value"),
    labels = c("Immunised", "Not Immunised"),
    values = c(highlight_col, second_col)
  ) +
  labs(title = "How do I read this plot?") +
  coord_fixed() +
  theme_void(base_size = 8, base_family = body_font) +
  theme(
    legend.position = "bottom",
    legend.title = element_blank(),
    plot.background = element_rect(
      fill = "transparent", color = "transparent"
    ),
    panel.background = element_rect(
```

```
      fill = "transparent", color = "transparent"
    ),
    plot.title = element_text(
      hjust = 0.5,
      face = "italic",
      margin = margin(t = 10, b = 10)
    ),
    plot.margin = margin(5, 5, 5, 5)
  )
legend_plot
```

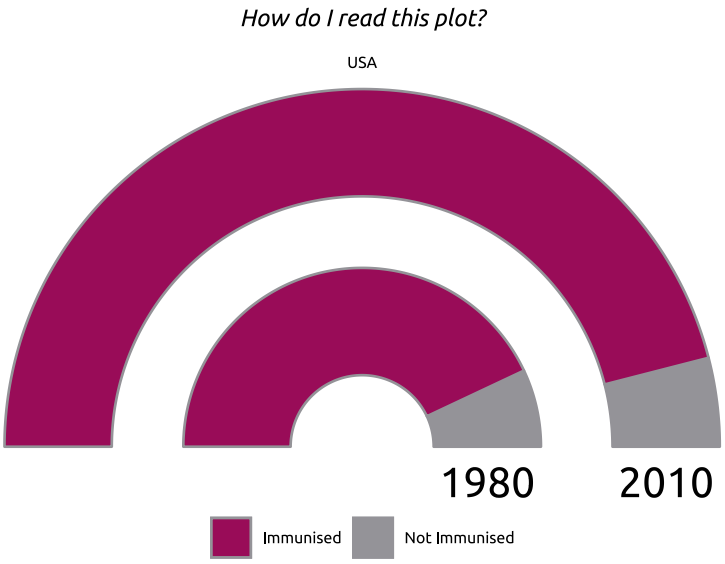


Figure 4.7: Version of the gauge chart shown only for USA which will act as a legend.

4.4.5 Combining legend with {patchwork}

Join with {patchwork}

```
library(patchwork)
final_plot <- main_plot +
  inset_element(legend_plot, 0.5, 0.9, 1.1, 1.4) &
  theme(
    plot.background = element_rect(
      fill = bg_col, color = bg_col
    ),
```

```
    panel.background = element_rect(
      fill = bg_col, color = bg_col
    )
  )
final_plot
```

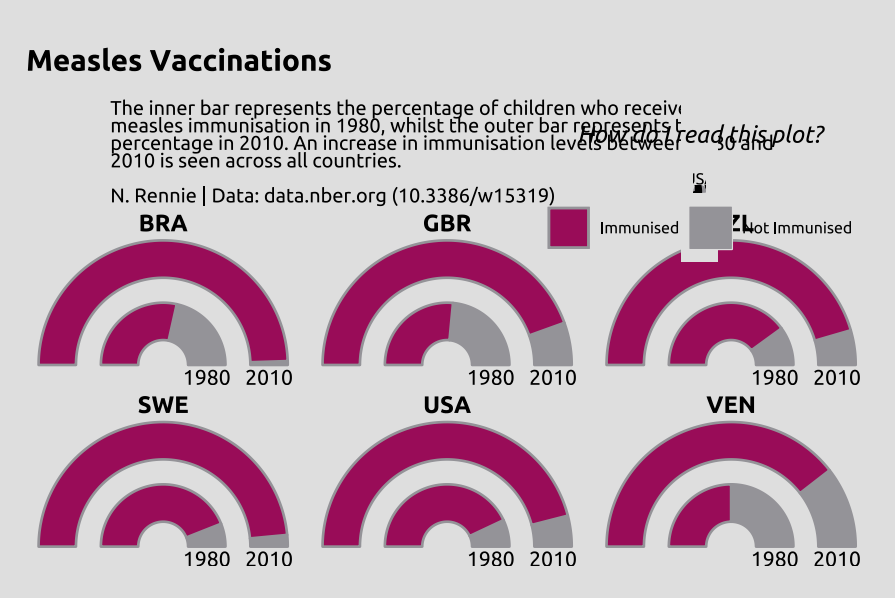


Figure 4.8: Final gauge chart with overlaid legend in top right corner.

As with previous examples, don't be fooled into thinking that the size of the legend and it's positioning within the main plot is something that happened perfectly this first. The values are often picked through a series of trial and error, and with practice you'll get better at choosing starting values.

4.5 Reflection

Are gauge charts the most effective method of visualising this data? No. Gauge charts have their own problems, some of which you can see here. Since the ring representing 2010 is on the outside, the radius is larger, and therefore the area . If you measure the change on arc length between 1980 and 2010, you'll get different answers to if you measured the proportional change in area for the two.

5

Conclusion: other tips and tricks

5.1 Template files for #TidyTuesday

You may notice that

5.2 Helper functions

5.3 {camcorder} for gifs

5.4 Other packages I didn't mention but use

ggsankey and ggalluvial

Bibliography

Comin, D., and Hohijn B. 2004. "Cross-Country Technological Adoption: Making the Theories Face the Facts." *Journal of Monetary Economics* January 2004: 39–83.

contributors, PLDB. 2022. "PLDB: A Programming Language Database." *PLDB*.

Garnier, Simon, Ross, Noam, Rudis, Robert, Camargo, et al. 2024. *viridis(Lite) - Colorblind-Friendly Color Maps for r*. https://doi.org/10.5281/zenodo.4679423.

Hughes, Ellis. 2022a. *Camcorder: Record Your Plot History*. https://CRAN.R-project.org/package=camcorder.

———. 2022b. *tidytuesdayR: Access the Weekly 'TidyTuesday' Project Dataset*. https://CRAN.R-project.org/package=tidytuesdayR.

Hvitfeldt, Emil. 2021. *Paletteer: Comprehensive Collection of Color Palettes*. https://github.com/EmilHvitfeldt/paletteer.

Mapping Museums. 2021. "Mapping Museums Data." www.mappingmuseums.org.

Mills, Blake Robert. 2022. *MetBrewer: Color Palettes Inspired by Works at the Metropolitan Museum of Art*. https://CRAN.R-project.org/package=MetBrewer.

Müller, Kirill, and Hadley Wickham. 2023. *Tibble: Simple Data Frames*. https://CRAN.R-project.org/package=tibble.

Our World in Data. 2019. "Medical Doctors Per 1,000 People, 2019." https://ourworldindata.org/grapher/physicians-per-1000-people.

Pedersen, Thomas Lin. 2022. *Ggforce: Accelerating 'Ggplot2'*. https://CRAN.R-project.org/package=ggforce.

———. 2024. *Patchwork: The Composer of Plots*. https://CRAN.R-project.org/package=patchwork.

Qiu, Yixuan, and authors/contributors of the included fonts. See file AUTHORS for details. 2022. *Sysfonts: Loading Fonts into r*. https://CRAN.R-project.org/package=sysfonts.

Qiu, Yixuan, and authors/contributors of the included software. See file AUTHORS for details. 2023. *Showtext: Using Fonts More Easily in r Graphs*. https://CRAN.R-project.org/package=showtext.

R4DS Online Learning Community. 2023. "Tidy Tuesday: A Weekly Social Data Project." https://github.com/rfordatascience/tidytuesday.

Schloss, Karen B., Connor C. Gramazio, Allison T. Silverman, Madeline

L. Parker, and Audrey S. Wang. 2019. "Mapping Color to Meaning in Colormap Data Visualizations." *IEEE Transactions on Visualization and Computer Graphics* 25 (1): 810–19. https://doi.org/10.1109/TVCG.2018.2865147.

Wickham, Hadley. 2016. *Ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York. https://ggplot2.tidyverse.org.

Wickham, Hadley, Mine Çetinkaya-Rundel, and Garrett Grolemund. 2023. *R for Data Science*. O-Reilly. https://r4ds.hadley.nz.

Wickham, Hadley, Romain François, Lionel Henry, Kirill Müller, and Davis Vaughan. 2023. *Dplyr: A Grammar of Data Manipulation*. https://CRAN.R-project.org/package=dplyr.

Wickham, Hadley, and Lionel Henry. 2023. *Purrr: Functional Programming Tools*. https://CRAN.R-project.org/package=purrr.

Wickham, Hadley, Jim Hester, and Jennifer Bryan. 2024. *Readr: Read Rectangular Text Data*. https://CRAN.R-project.org/package=readr.

Wickham, Hadley, Davis Vaughan, and Maximilian Girlich. 2024. *Tidyr: Tidy Messy Data*. https://CRAN.R-project.org/package=tidyr.

Wilke, Claus O., and Brenton M. Wiernik. 2022. *Ggtext: Improved Text Rendering Support for 'Ggplot2'*. https://CRAN.R-project.org/package=ggtext.

Yutani, Hiroaki. 2023. *Gghighlight: Highlight Lines and Points in 'Ggplot2'*. https://CRAN.R-project.org/package=gghighlight.

Appendix

Software requirements

This book was built using R version 4.3.3. This book is built with Quarto, using version number 1.4.551. All R packages required to build this book can be found in the following table. Note that this table contains all packages required to create the book, not just those required for the examples.

Table 5.1: R packages and version numbers

Package	Version	Package	Version	Package	Version
BH	1.84.0-0	gert	2.0.1	rd2list	0.1.0
DBI	1.2.2	ggforce	0.4.2	readr	2.1.5
KernSmooth	2.23-22	gghighlight	0.4.1	readxl	1.4.3
MASS	7.3-60.0.1	ggplot2	3.5.0	rematch	2.0.0
Matrix	1.6-5	ggrepel	0.9.5	rematch2	2.1.2
MetBrewer	0.2.0	ggtext	0.1.2	renv	1.0.5
R.cache	0.16.0	gh	1.4.0	reprex	2.1.0
R.methodsS3	1.8.2	gifski	1.12.0-2	rlang	1.1.3
R.oo	1.26.0	gitcreds	0.1.2	rmarkdown	2.25
R.utils	2.12.3	glue	1.7.0	rnaturalearth	1.0.1
R6	2.5.1	goftest	1.2-3	rnaturalearthdata	1.0.0
RColorBrewer	1.1-3	googledrive	2.1.1	rpart	4.1.23
RCurl	1.98-1.14	googlesheets4	1.1.1	rprojroot	2.0.4
Rcpp	1.0.12	gridExtra	2.3	rstudioapi	0.15.0
RcppArmadillo	0.12.8.0.0	gridtext	0.1.5	rsvg	2.6.0
RcppEigen	0.3.4.0.0	gtable	0.3.4	rvest	1.0.4
Rttf2pt1	1.3.12	haven	2.5.4	s2	1.1.6
VoronoiPlus	0.1.0	highr	0.10	sass	0.4.8
abind	1.4-5	hms	1.1.3	scales	1.3.0
anytime	0.3.9	htmltools	0.5.7	selectr	0.4-2
askpass	1.2.0	httr	1.4.7	sf	1.0-15
backports	1.4.1	httr2	1.0.0	showtext	0.9-7
base64enc	0.1-3	ids	1.0.1	showtextdb	3.0
bit	4.0.5	imguR	1.0.3	sp	2.1-3
bit64	4.0.5	ini	0.3.1	spatstat	3.0-7
bitops	1.0-7	isoband	0.2.7	spatstat.data	3.0-4
blob	1.2.4	jcolors	0.0.5	spatstat.explore	3.2-6
brio	1.1.4	jpeg	0.1-10	spatstat.geom	3.2-9
broom	1.0.5	jquerylib	0.1.4	spatstat.linnet	3.1-4
bslib	0.6.1	jsonlite	1.8.8	spatstat.model	3.2-10

cachem	1.0.8	kableExtra	1.4.0	spatstat.random	3.2-3
callr	3.7.5	knitr	1.45	spatstat.sparse	3.0-3
camcorder	0.1.0	labeling	0.4.3	spatstat.utils	3.0-4
cellranger	1.1.0	later	1.3.2	stringi	1.8.3
class	7.3-22	lattice	0.22-5	stringr	1.5.1
classInt	0.4-10	lifecycle	1.0.4	styler	1.10.2
cli	3.6.2	lubridate	1.9.3	svglite	2.1.3
clipr	0.8.0	magick	2.8.3	sys	3.4.2
colorspace	2.1-0	magrittr	2.0.3	sysfonts	0.8.9
commonmark	1.9.1	maps	3.4.2	systemfonts	1.0.5
conflicted	1.2.0	markdown	1.12	tensor	1.5
cowplot	1.1.3	memoise	2.0.1	terra	1.7-71
cpp11	0.4.7	mgcv	1.9-1	textshaping	0.3.7
crayon	1.5.2	mime	0.12	tibble	3.2.1
credentials	2.0.1	modelr	0.1.11	tidyr	1.3.1
curl	5.2.1	monochromeR	0.2.0	tidyselect	1.2.0
data.table	1.15.2	munsell	0.5.0	tidyterra	0.5.2
dbplyr	2.4.0	nlme	3.1-164	tidytuesdayR	1.0.3
deldir	2.0-4	nrBrand	0.0.13	tidyverse	2.0.0
desc	1.4.3	openssl	2.1.1	timechange	0.3.0
digest	0.6.34	owidR	1.4.2	tinytex	0.49
downlit	0.4.3	patchwork	1.2.0	tsibble	1.1.4
dplyr	1.1.4	pillar	1.9.0	tweenr	2.0.3
dtplyr	1.3.1	pkgconfig	2.0.3	tzdb	0.4.0
e1071	1.7-14	plotwidgets	0.5.1	units	0.8-5
ellipsis	0.3.2	png	0.1-8	usefunc	1.1.3
emojifont	0.5.5	poissoned	0.1.2	usethis	2.2.3
evaluate	0.23	polyclip	1.10-6	utf8	1.2.4
extrafont	0.19	prettyunits	1.2.0	uuid	1.2-0
extrafontdb	1.0	processx	3.8.3	vctrs	0.6.5
fansi	1.0.6	progress	1.2.3	viridis	0.6.5
farver	2.1.1	proto	1.0.0	viridisLite	0.4.2
fastmap	1.1.1	proxy	0.4-27	vroom	1.6.5
fontawesome	0.5.2	ps	1.7.6	whisker	0.4.1
forcats	1.0.0	purrr	1.0.2	withr	3.0.0
fs	1.6.3	quarto	1.4	wk	0.9.1
gargle	1.5.2	ragg	1.2.7	xfun	0.42
generics	0.1.3	rappdirs	0.3.3	xml2	1.3.6
geofacet	0.2.1	raster	3.6-26	yaml	2.3.8
geogrid	0.1.2	rcartocolor	2.1.1	zip	2.3.1

Data

All data sets used in this book, and links to the relevant licenses:

Table

Chapter	Data	Source URL

1	Programming Languages Database	https://pldb.com/
2	Honey Bee Colonies	https://usda.library.cornell.edu/
3	Mapping Museums	https://museweb.dcs.bbk.ac.uk/
4		
5	Trash Wheels	https://docs.google.com/spreads
6		
7	The small home ranges and large local ecological impacts of pet cats [United Kingdom]	https://www.datarepository.mov
8	Medical doctors per 1,000 people, 2019	https://ourworldindata.org/grap
9		
10		
11	Lemurs	https://lemur.duke.edu/duke-le
12	R Vignettes	https://github.com/rmflight/vig
13	RAM Legacy Stock Assessment Database	https://zenodo.org/records/482
14	Historical Cross Country Technology Adoption Dataset	https://www.nber.org/research/
15		
16		

Index