opeRate

Learn to apply R

Adam Rawles

Contents

Chapter 1

opeRate

1.1 Overview

This book is a collection of materials to help users apply their fundamental R knowledge to real programming and analysis. This book is the second in a series of R books I've been working on. The first in the series (teacheR[1]) focuses on the fundamentals of the R language. I would recommend reading teacheR first if you're brand new to the language. It's split into two parts ("For Students" and "For Teachers"). To get the most out of this book, I would suggest that you are at least comfortable with the entirety of the "For Students" section, but it wouldn't hurt to go through the "For Teachers" section while you're at it.

As with the teacheR[2] book, this is a work in progress, so please feel free to make any suggestions or corrections via this book's GitHub repository[3].

1.2 About Me

I began using R in my second year of university, whilst studying psychology. Like so many others before me, I started using R for a particular project - in my case, it was for an analysis of publication bias - before deciding that I wanted to expand my skillset and learn to apply R to lots of different situations. Because I took this approach however, I didn't really develop a fundamental knowledge of how R worked before I started - I just kind of jumped in at the deep end. As a quick analogy, it was a bit like starting with this book without reading the teacheR book first - I kind of knew what was going on, but I was filling in a lot of gaps along the way.

[1] https://teacher.arawles.co.uk
[2] teacher.arawles.co.uk
[3] www.github.com/arawles/operate/issues

And so that is why I decided to develop this series of books - to hopefully help anyone who may find themselves in a similar position that I was in those years ago. If you want to use R but feel as though you don't know where to start, then hopefully this book will give you a good overview of some of the different ways that R can be used or applied.

1.3 Using R

In my primary years, analysis in R took me longer than it would take to do the same analysis in something like Excel. And that's okay. R is a complicated and flexible system, and so your first analysis piece will never be particularly efficient. As you stick with it however, and you get used to the methods of automation and a pipeline of execution, you'll find yourself working much more efficiently, performing analyses in half the time. And that's what I hope I can impart with this book; it'll be slow at first, but you'll notice a turning point when you complete your first analysis project in a decent timescale and you'll never look back. Then, before long, you'll have a repertoire of analysis tools at your disposal that make you a crucial member of any data analysis team.

And so in this book we're going to look at some of the common tasks that one might decide to do in R. Keep in mind though that we can't cover everything, so just because it's not in the book doesn't mean that it can't be done!

Part I

tidyverse

Chapter 2

Introduction

As we learnt in the teacheR[1] book, R is supported by thousands of packages that provide extra functionality to the base R experience. One of the most popular sets of packages developed by Hadley Wickham and the RStudio Team is called the tidyverse[2]. The tidyverse is made up of packages designed for data science work that are all underpinned by a common philosophy and a common syntax. At the core of this philosophy is the concept of "tidiness" in data.

The tidyverse is a set of **opinionated** packages. That means that there's (usually) a right way to do things with the package, and there's a wrong way. There's much debate in the R community as to whether relying heavily on opinionated external packages such as those included in the tidyverse is a good thing. Personally, I think that the packages included in the tidyverse are fantastic, and are a large reason why R is thriving today.

So in this book, we're a tidyverse family, and we'll be using the tidyverse packages throughout.

[1] teacher.arawles.co.uk
[2] https://www.tidyverse.org/

Chapter 3

"Tidy" data

The concept of "tidy" data is something that's important throughout the set of tidyverse packages. There is an in-depth paper[1] published in the Journal of Statistical Software, authored by Hadley Wickham, that describes the concept and requirements of tidy data in full. But for now we're just going to look at the basics.

In order for a dataset to be tidy, it needs to abide by three rules:

1. Every column is a variable
2. Every row is an observation
3. Every cell is a single value

If a dataset doesn't abide by all three of these rules, then it's a "messy" dataset.

The benefit of having your data in a tidy format is fairly simple; it provides a standard for structuring a dataset. This means that tools can be developed with the assumption that the data will be in a specific format, making the development of these data science tools easier. And this is essentially what underpins the entire tidyverse: get your data in a tidy format and you'll have access to everything the tidyverse provides.

[1] https://vita.had.co.nz/papers/tidy-data.html

Chapter 4

The pipe (%>%)

The pipe is a core feature of the tidyverse packages, and has proved so popular in its R implementation that it served as the inspiration for the `|>` function that was introduced into R in version 4.0. For now we're going to look specifically at the tidyverse (or more specifically the `magrittr`) implementation of the pipe but they are very similar.

At the simplest level, the pipe takes the output from whatever is on its left and passes it as the first argument to the expression on the right. Like this:

```
1 %>% print()
```

```
## [1] 1
```

In this example, the 1 is passed as the first argument to the `print()` function, which prints it out.

You can also specify which argument you want the value to be passed as by using the `.` shorthand:

```
1 %>% substr("water", ., 2)
```

```
## [1] "wa"
```

This passes the `1` from the left hand side as the second argument to the `substr()` function (which is the `start` index).

The benefit of the pipe is that it allows you to read code from left to right, rather than from middle outwards. Let's compare two examples with an without the pipe:

```
str <- "water"

print(paste0(substr(str, 1, 1), "ine"))
```

```
## [1] "wine"
```

```
str %>% substr(1,1) %>% paste0("ine") %>% print()
```

```
## [1] "wine"
```

Although the outcome is exactly the same, following the flow of logic is much easier in the bottom example because we can start at the left and more rightwards.

For the purposes of this book that's as much as you need to know. You'll see the pipe used a lot when we're performing multiple data analysis steps, like cleaning then filtering then selecting columns and so on. This is why almost all of the tidyverse functions used for data science will accept the dataset as their first argument - it allows you to chain function calls together using the pipe.

Chapter 5

Quasiquotation

In the teacheR[1] book, we looked at the concept of quasiquotation; selectively quoting and then evaluating parts of an expression. The tidyverse packages rely heavily on this concept, which is why you'll often see (for example) column names passed to the tidyverse packages as variable names rather than character strings, like this:

```
my_df %>%
  dplyr::mutate(new_column = old_column + 1)
```

This might seem odd when you first start using this approach; isn't the function going to look for a variable in the global environment called `old_column` and then fail when it can't find it?

When you pass a column name like this, the tidyverse packages will quote that input and then evaluate the resulting expression in the context of the current data frame. So essentially, R looks for the `old_column` object in the context of `my_df`, rather than in the global environment. Because dataframes are essentially just lists and lists can be treated like environments, it searches that environment for the `old_column` object and finds it, returning the contents of the column.

5.1 tidyselect

Much of this functionality is powered by a package called `{tidyselect}`. The `{tidyselect}` package provides a number of functions for selecting variables from datasets, reducing the time spent laboriously typing out the name of every variable you want to reference.

[1] https://teacher.arawles.co.uk

To better understand the specifics of the `{tidyselect}` dialect and how it interacts with the tidyverse packages, there are a number of vignettes[2] included in the package that explain things well. For now however, if you can understand the basic concept that variables are evaluated in the specific context of the dataset provided to the function, then you should be fine.

[2]https://tidyselect.r-lib.org/reference/language.html

Part II

Data Analysis

Chapter 6

Introduction

In this chapter, we're going to go through some typical data science tasks and learn how to do them in R. Later on, we'll look at how you should apply these skills in your projects, and some tips for making your project easier to read and share with others.

For this chapter, we're going to use a Kaggle dataset[1] that holds information on video game sales. The dataset contains sales in North America, Europe, Japan and globally for the top games each year going back to 1985. The games are broken down by platform, genre, and publisher:

```
## # A tibble: 16,598 x 11
##    Rank Name  Platform Year  Genre Publisher NA_Sales EU_Sales JP_Sales
##   <dbl> <chr> <chr>    <chr> <chr> <chr>        <dbl>    <dbl>    <dbl>
## 1     1 Wii ~ Wii      2006  Spor~ Nintendo      41.5    29.0      3.77
## 2     2 Supe~ NES      1985  Plat~ Nintendo      29.1     3.58     6.81
## 3     3 Mari~ Wii      2008  Raci~ Nintendo      15.8    12.9      3.79
## 4     4 Wii ~ Wii      2009  Spor~ Nintendo      15.8    11.0      3.28
## 5     5 Poke~ GB       1996  Role~ Nintendo      11.3     8.89    10.2
## # ... with 16,593 more rows, and 2 more variables: Other_Sales <dbl>,
## #   Global_Sales <dbl>
```

[1] https://www.kaggle.com/gregorut/videogamesales

Chapter 7

Loading

The first step in any data analysis project you'll undertake is getting at least one dataset. Oftentimes, we have less control over the data we use than we would like; receiving odd Excel spreadsheets or text files or files in a proprietary format or whatever. In this chapter, we'll focus on the more typical data formats (rds, CSV and Excel), but we'll also look at how we might extract data from a web API, which is an increasingly common method for data loading. We'll also look briefly at how you can extract data from a SQL database.

7.1 RDS

If you need to share an R-specific object (like a linear model created with the `lm()` function) or you're certain that the data never needs to be readable in another program, then you utilise the rds format to save and read data.

To load in RDS files, we use the `readRDS()` function, providing the file path of the file we want to read:

```
my_data <- readRDS(file = "path/to/file")
```

To save an object to an .rds file, you just need to provide the object you want to save and a file path to save it to:

```
saveRDS(my_data, file = "path/to/file")
```

7.1.1 Advantages

RDS files are R specific files that contain a serialized version of a specific object. The benefit of R objects is that the object will be preserved in its entirety - you

won't lose the column data types when you save it and then load it in again.

7.1.2 Disadvantages

RDS files cannot be read natively by other programs. This means that if you're trying to share your dataset and someone wants to open it in, say, Excel, they're going to need to convert it to a different format before they can load it. The RDS format therefore isn't ideal for sharing data outside of the R ecosystem.

7.2 CSV

If I have any say in the data format of the files I need to load in, I usually ask for them to be in CSV format. CSV stands for "comma-separated values" and essentially means that the data is stored as one long text string, with each different value or cell separated by a comma (although you will see CSV files with different separators). So for example, a really simple CSV file may look, in its most base format, like this:

```
name,age,
Dave,35,
Simon,60,
Anna,24,
Patricia,75
```

Benefits of the CSV file over something like an Excel file are largely based around simplicity. CSV files are typically smaller and can only have one sheet, meaning that you won't get confused with multiple spreadsheets. Furthermore, values in CSV files are essentially what you see is what you get. With Excel files, sometimes the value that you see in Excel isn't the value that ends up in R (looking at you dates and datetimes). For these reasons, I would suggest using a separated-value file over an Excel file when you can.

7.2.1 Loading CSV files

Loading CSV files in R is relatively simple. There are base* functions that come with R to load CSV files but there's also a popular package called `readr` which can be used so I'll cover both.

* They are technically from the `utils` package which comes bundled with R so we'll call it base R.

7.2.1.1 Base R

To load a CSV file using base R, we'll use the `read.csv()` function:

```
read.csv(file = "path/to/file", header = TRUE, ...)
```

The `file` parameters needs the path to your file as a character string. The `header` parameter is used to tell R whether or not your file has column headers. Our dataset does have headers (i.e. the first row is the column names) so we set that to `TRUE`.

There are lots of other parameters that can be tweaked for the `read.csv()` function, but we won't go through them here.

7.2.1.2 readr

The `readr` package comes with a similar function: `read_csv()`. With the exception of a couple of extra parameters in the `read_csv()` function and potentially some better efficiency, there isn't a massive difference between the two.

Using the `read_csv()` function is simple:

```
readr::read_csv(file = "path/to/file", col_names = TRUE)
```

In this function, the `header` parameter is replaced with the `col_names` parameter. The `col_names` parameter is very similar, you can say whether your dataset has column headings, or you can provide a character vector of names to be used as column headers.

Let's load our Kaggle dataset in using the `readr::read_csv()` function:

```
print(readr::read_csv("./data/vgsales.csv"), n = 5)
```

```
##
## -- Column specification --------------------------------------------------------
## cols(
##   Rank = col_double(),
##   Name = col_character(),
##   Platform = col_character(),
##   Year = col_character(),
##   Genre = col_character(),
##   Publisher = col_character(),
##   NA_Sales = col_double(),
##   EU_Sales = col_double(),
##   JP_Sales = col_double(),
```

```
##   Other_Sales = col_double(),
##   Global_Sales = col_double()
## )
```

```
## # A tibble: 16,598 x 11
##    Rank Name  Platform Year  Genre Publisher NA_Sales EU_Sales JP_Sales
##   <dbl> <chr> <chr>    <chr> <chr> <chr>        <dbl>    <dbl>    <dbl>
## 1     1 Wii ~ Wii      2006  Spor~ Nintendo      41.5    29.0      3.77
## 2     2 Supe~ NES      1985  Plat~ Nintendo      29.1     3.58     6.81
## 3     3 Mari~ Wii      2008  Raci~ Nintendo      15.8    12.9      3.79
## 4     4 Wii ~ Wii      2009  Spor~ Nintendo      15.8    11.0      3.28
## 5     5 Poke~ GB       1996  Role~ Nintendo      11.3     8.89    10.2
## # ... with 16,593 more rows, and 2 more variables: Other_Sales <dbl>,
## #   Global_Sales <dbl>
```

You can see that when we load in a file using `{readr}` without specifying column types, we get an output showing us exactly how each column has been parsed. This is because CSV is a typeless format, and so data isn't always imported as the type you intended it to be.

To override the default types that `{readr}` has assigned, we can use the `col_types` parameter. This can either be provided using the `cols()` helper function like this:

```
readr::read_csv(file = "path/to/file",
                col_names = TRUE,
                col_types = readr::cols(
                  readr::col_character(), readr::col_double(), ... # You need to provi
                ),
                ...
)
```

Or, you can provide a compact string with different letters representing different datatypes:

```
readr::read_csv(file = "path/to/file",
                col_names = TRUE,
                col_types = "cd",
                ...
)
```

The codes for the different datatypes can be found on the documentation page for the `read_csv()` function (type `?read_csv()`).

Let's use the compact string format to load in the video game dataset again but this time, we want the year to be imported as a number:

```
vg_sales <- readr::read_csv("./data/vgsales.csv", col_types = "dccdccddddd")
```

```
## Warning: 271 parsing failures.
## row  col expected actual                file
## 180 Year a double    N/A './data/vgsales.csv'
## 378 Year a double    N/A './data/vgsales.csv'
## 432 Year a double    N/A './data/vgsales.csv'
## 471 Year a double    N/A './data/vgsales.csv'
## 608 Year a double    N/A './data/vgsales.csv'
## ... .... ........ ...... .....................
## See problems(...) for more details.
```

We've got some warnings here because `{readr}` wasn't able to parse some of the values in the Year column as numeric, giving us some NAs. We'll clean up these values in the Cleaning chapter so we can ignore them for now.

7.2.2 Advantages

CSV files can be opened in a number of different software packages, making the CSV format a good candidate for sharing data with people who may not also be using R.

7.2.3 Disadvantages

In the interests of simplicity, CSV files don't store information on the **type** of the data in each column, meaning that you need to be careful when you're loading data from a CSV file that the column types you end up with are the ones you want.

The CSV format also isn't fully standardised, meaning that you might come across some files that say they're CSV, but generate errors when they're parsed. It's relatively rare to see a CSV file that is so different that it can't be parsed at all, but it's something worth remembering.

7.2.4 Other delimited files

Comma-separated value files are just a form of delimited files that use a comma to separate different values. In the wild you might see files separated with all different kinds of symbols, like pipes (|) or tabs. To load in these types of files, use the `readr::read_delim()` function and specify what's being used to separate the values with the `delim` parameter. `readr::read_csv()` basically just wraps `readr::read_delim()` using `delim = ','` anyway, along as you're comfortable loading in CSV files, you should be well equipped to load in any kind of delimited file.

7.3 Excel files

R doesn't have any built-in functions to load Excel files. Instead, you'll need to use a package. One of the more popular packages used to read Excel files is the `readxl` package.

Once you've installed and loaded the `readxl` package. You can use the `read_excel()` function:

```
readxl::read_excel(path = "path/to/file", sheet = NULL, range = NULL, ...)
```

Because Excel files are a little bit more complicated than CSV files, you'll notice that there are some extra parameters. Most notably, the sheet and range parameters can be used to define a subset of the entire Excel file to be loaded. By default, both are set to NULL, which will mean that R will load the entirety of the first sheet.

Like the `readr::read_csv()` function, you can specify column names and types using the `col_names` and `col_types` parameters respectively, and also trim your values using `trim_ws`.

7.3.1 Advantages

I have something of a personal vendetta against storing everything in Excel spreadsheets because of the terrible way Excel displays data, so I personally don't think there are too many advantages in using Excel files.

You can have more than one sheet maybe? That's all I've got.

7.3.2 Disadvantages

The main disadvantage of Excel files for me is that Excel aggressively formats data for the end user. That is, it's difficult to know what value is actually being stored in a cell based on the value that the end user sees in the cell. Dates are a prime example, Excel will show you the date as a date, but will store it as a number with an origin. That alone isn't a sin at all, but combine that with the fact that Excel has multiple origin dates depending on your Excel version and OS[1], that's a strike in my book.

You could also argue that the xlsx format is a software-specific format, and it kind of is. But because Excel is so ubiquitous now, there are multiple ways of opening and converting xlsx files without ever using Excel, so I don't think that's really a disadvantage.

[1] https://support.microsoft.com/en-us/office/date-systems-in-excel-e7fe7167-48a9-4b96-bb53-5612a800b487#ID0EBBH=Windows

Overall, if you can send the data in CSV format instead of an Excel file, do that.

7.4 Web-based APIs

Loading static data from text and Excel files is very common. However, an emerging method of data extraction is via web-based APIs. These web-based APIs allow a user to extract datasets from larger repositories using just an internet connection. This allows for access to larger and more dynamic datasets.

7.4.1 What are APIs?

API stands for application programming interface. APIs are essentially just a set of functions for interacting with an application or service. For instance, many of the packages that you'll use will essentially just be forms of API; they provide you with functions to interact with an underlying system or service.

For data extraction, we're going to focus more specifically on web-based APIs. These APIs use the HTTP protocols to accept requests and then return the data requested. Whilst there are multiple *methods* that can be implemented in an API to perform different actions, we're going to focus on the `GET` method. That is, we're purely *getting* something from the API rather than trying to change anything that's stored on the server. You can think of the `GET` method as being read-only.

To start with, we're going to look at exactly how you would interact with an API, but then we'll look at the BMRSr package, which I wrote to make interacting with the Balancing Mechanism and Reporting Service easier.

7.4.2 Accessing APIs in R

To access a web-based API in R, we're going to need a connection to the internet, something that can use the HTTP protocol (we're going to use the `httr` package) and potentially some log in credentials for the API. In this case, we're going to just use a test API, but in reality, most APIs require that you use some kind of authentication so that they know who's accessing their data.

As previously mentioned, to extract something from the API, you'll be using the `GET` method. The `httr` package makes this super easy by providing a `GET` function. To this function, we'll need to provide a URL. The `GET` function will then send a GET request to that address and return the response. A really simple GET request could be:

```
httr::GET(url = "http://google.com")
```

```
## Response [http://www.google.com/]
##   Date: 2021-09-10 07:02
##   Status: 200
##   Content-Type: text/html; charset=ISO-8859-1
##   Size: 13.9 kB
## <!doctype html><html itemscope="" itemtype="http://schema.org/WebPage" lang="...
## var f=this||self;var h,k=[];function l(a){for(var b;a&&(!a.getAttribute||!(b=...
## function n(a,b,c,d,g){var e="";c||-1!==b.search("&ei=")||(e="&ei="+l(d),-1===...
## google.y={};google.sy=[];google.x=function(a,b){if(a)var c=a.id;else{do c=Mat...
## document.documentElement.addEventListener("submit",function(b){var a;if(a=b.t...
## </style><style>body,td,a,p,.h{font-family:arial,sans-serif}body{margin:0;over...
## var f=this||self;var g,h,k=null!==(g=f.mei)&&void 0!==g?g:1,l=null!==(h=f.sdo...
## if (!iesg){document.f&&document.f.q.focus();document.gbqf&&document.gbqf.q.fo...
## }
## })();</script><div id="mngb"><div id=gbar><nobr><b class=gb1>Search</b> <a cl...
## ...
```

That seems like a really complicated response at first, but when we look at each part, it's quite simple.

- Response
- This is telling us where we got our response from. In this case, we sent a request to Google, so we got a response from Google.
- Date
  - Fairly self-explanatory - the date and time of the response.
- Status
  - Status codes[2] give you an indication of how the handling of the request went. 200 means "Success".
- Content-Type
  - This is telling us what type the response is. In this case, the response is just a HTML page, which is exactly what we expect as that's what you get when you type "google.com" into your browser.
- Size
  - This is the size of the response
- Content
  - Below the size, we see the actual response body. In this case, we've been given the html for the google.com page.

[2] https://en.wikipedia.org/wiki/List_of_HTTP_status_codes

As simple as this example was, it didn't really give us anything interesting back, just the Google homepage. So let's use the GET request to get something more interesting.

We're going to access the jsonplaceholder[3] website, which provides fake APIs for testing. But for now, imagine that this is something like an Instagram database, holding users and their posts and comments.

The first step in accessing an API is to understand that commands the API is expecting. APIs will have what we call **endpoints**. These are paths that we can use to access a certain dataset. For instance, looking at the website, we can see that there are endpoints for lots of different types of data: posts, comments, albums, photos, todos and users. To access an endpoint, we just need to make sure we're using the correct path. So let's try getting a list of users:

```
httr::GET(url = "https://jsonplaceholder.typicode.com/users")
```

```
## Response [https://jsonplaceholder.typicode.com/users]
##   Date: 2021-09-10 07:02
##   Status: 200
##   Content-Type: application/json; charset=utf-8
##   Size: 5.64 kB
## [
##   {
##     "id": 1,
##     "name": "Leanne Graham",
##     "username": "Bret",
##     "email": "Sincere@april.biz",
##     "address": {
##       "street": "Kulas Light",
##       "suite": "Apt. 556",
##       "city": "Gwenborough",
## ...
```

Looking at the content type, we can see that unlike when we sent a request to Google.com, we've got a Content-Type of application/json. JSON is a data structure often used to send data across APIs. We won't go into the structure of it now because R does most of the conversion for us, but if you're interested, there's more info on the JSON structure at www.json.org[4].

Trying to read raw JSON is hard, but `httr` includes functions to help us get it into a better structure for R. Using the `httr::content()` function, `httr` will automatically read the response content and convert it into the format we ask for (via the `as` parameter). For now, we're going to leave the `at` parameter as 'NULL' which guesses the best format for us.

[3] https://jsonplaceholder.typicode.com/
[4] https://www.json.org/json-en.html

```
response <- httr::GET(url = "https://jsonplaceholder.typicode.com/users")
content <- httr::content(response)
head(content, 1) # we'll just look at the first entry for presentation sake
```

```
## [[1]]
## [[1]]$id
## [1] 1
##
## [[1]]$name
## [1] "Leanne Graham"
##
## [[1]]$username
## [1] "Bret"
##
## [[1]]$email
## [1] "Sincere@april.biz"
##
## [[1]]$address
## [[1]]$address$street
## [1] "Kulas Light"
##
## [[1]]$address$suite
## [1] "Apt. 556"
##
## [[1]]$address$city
## [1] "Gwenborough"
##
## [[1]]$address$zipcode
## [1] "92998-3874"
##
## [[1]]$address$geo
## [[1]]$address$geo$lat
## [1] "-37.3159"
##
## [[1]]$address$geo$lng
## [1] "81.1496"
##
##
##
## [[1]]$phone
## [1] "1-770-736-8031 x56442"
##
## [[1]]$website
## [1] "hildegard.org"
##
```

```
## [[1]]$company
## [[1]]$company$name
## [1] "Romaguera-Crona"
##
## [[1]]$company$catchPhrase
## [1] "Multi-layered client-server neural-net"
##
## [[1]]$company$bs
## [1] "harness real-time e-markets"
```

We can see that R has taken the response and turned it into a list for us. From here, we can then start our analysis.

In many cases however, you won't want a complete list. Instead, you'll want to provide some parameters to limit the data you get back from your endpoint. Most APIs will have a way of doing this. For example, reading the jsonplaceholder website, we can see that we can get all the posts for a specific user by appending the url with "?userId=x". This section of the URL (things after a ?) are called the query part of the URL. So let's try getting all of the posts for the user with ID 1:

```
response <- httr::GET(url = "https://jsonplaceholder.typicode.com/posts?userId=1")
content <- httr::content(response)
head(content, 1) # we'll just look at the first entry for presentation sake
```

```
## [[1]]
## [[1]]$userId
## [1] 1
##
## [[1]]$id
## [1] 1
##
## [[1]]$title
## [1] "sunt aut facere repellat provident occaecati excepturi optio reprehenderit"
##
## [[1]]$body
## [1] "quia et suscipit\nsuscipit recusandae consequuntur expedita et cum\nreprehenderit molesti
```

Whilst the parameters here are pretty simple, you will come across APIs that accept multiple parameters, making data extraction from an API a very powerful tool.

7.4.3 BMRSr

As easy as the above was, interacting with APIs that have several parameters and complicated URLs can get confusing. To this end, many people create

packages in R that act as wrappers for various APIs. These packages will then provide you with functions that will automatically create the request, send it and receive and parse the content. You can kind of think about it as an API for an API!

This is what I did for the Balancing Mechanism Reporting Service (BMRS[5]) API. BMRS provides a massive amount of energy-related data, but creating the correct URLs and dealing with the response can be tricky. The BMRSr package that I wrote was designed to help with that.

We'll now go through a quick demo of the BMRSr package. If you're not too bothered about this part, feel free to skip to the next section.

If you're interested, there are a couple of things you'll need:

- The BMRSr package installed
- A free BMRS API key that can be retrieved from the ELEXON portal[6].

Once you've got those two prerequisites, using BMRSr should be quite easy. The main function in the BMRSr package is the `full_request()` function, which will create your URL, send the request, and parse the response depending on your parameters. To do this however, the `full_request()` function needs some parameters:

- `data_item`
  - A data item to retrieve. The BMRS platform holds lots of datasets, and so we need to specify which one we want to retrieve.
- `api_key`
  - Our API_key that we got from the Elexon portal
- parameters
  - Depending on which data_item you chose, you'll need to provide some parameters to filter the data
- `service_type`
  - What format you want the data returned in: values are XML or CSV.

So what parameters do I need? Well, the easiest way to find out is to use the `get_parameters()` function. This will return all of the parameters that can be provided to the `full_request()`.

Let's do an example. Say I want to return data for the B1620 data item, which shows us aggregated generation output per type. So, the first step is to know what parameters I can provide using the `get_parameters()` function:

[5] https://bmreports.com/
[6] https://www.elexonportal.co.uk/

```
BMRSr::get_parameters("B1620")
```

```
## [1] "settlement_date" "period"
```

This tells me that I can provide two parameters in my request - the date and the settlement period. Using this information in my `full_request()` function...

```
bmrs_data <- BMRSr::full_request(data_item = "B1620",
                                 api_key = "put_your_API_key_here",
                                 service_type = "csv",
                                 settlement_date = "01/11/2019",
                                 period = "*") # From reading the API manual,
# I know that this returns all periods
head(bmrs_data, 2)
```

```
## # A tibble: 2 x 13
##   `*Document Type` `Business Type` `Process Type` `Time Series ID` Quantity
##   <chr>            <chr>           <chr>          <chr>                <dbl>
## 1 Actual generati~ Production      Realised       NGET-EMFIP-AGPT~      1636
## 2 Actual generati~ Production      Realised       NGET-EMFIP-AGPT~         0
## # ... with 8 more variables: `Curve Type` <chr>, Resolution <chr>, `Settlement
## #   Date` <date>, `Settlement Period` <dbl>, `Power System Resource
## #   Type` <chr>, `Active Flag` <chr>, `Document ID` <chr>, `Document
## #   RevNum` <dbl>
```

And there we have it, we've retrieved a energy-related dataset from an API using the BMRSr package. There are roughly 101 data items available on BMRS so there's a massive amount of data there for those who want to access it.

7.5 Databases

In the corporate world or when you're dealing with larger systems, you'll often have some form of database that stores all of the system data. This database can function as a great repository of data for your analyses, and by utilising a live connection to the database, we can easily update our analyses in the future.

In this section, we'll really only cover the basics of how R can interact with an SQL database, so don't feel as though you need to be a data engineer expert to understand this section. As long as you know the tiniest bit of SQL, you should be fine.

7.5.1 Database Management Systems and ODBC

There are lots of different types of database managements systems (DBMS) such as SQL Server, MySQL, MariaDB, and so on. These DBMSs allow us to create and maintain databases.

When we connect to our database, we do so through our DBMS, but all of them have slightly different implementations. This means that if we developed a package to connect to one type of database (like MySQL), we'd have to create an entirely different package to do the same thing with a MariaDB database. This is where the Open Database Connectivity (ODBC) standard comes in. The ODBC standard allows is to interface with databases that use different DBMS systems using a common API. This means that we could swap out our SQL Server database with a MySQL one, and we wouldn't need to make too many changes. The ODBC standard is implemented via a driver; this driver functions as an interface layer between our application (R) and the database.

So when we connect to a database with R we go in this order:

R -> ODBC Driver -> DBMS

R tells the ODBC driver to run a query, and then the ODBC driver converts that request to one that can be interpreted by the DBMS that it was built for. This means that as long as we've got the appropriate ODBC driver for our DBMS, we can use (basically) the same R code to interact with any kind of ODBC-compliant database.

So to connect to our database, we're going to need 2 things:

- The `odbc` R package
  - This provides us with the R functions to create our connections and run our queries as so on.
  - Think of the `odbc` package as a set of tools for interacting with any ODBC database; it knows how to interface with the ODBC drivers, not the DBMS.
- The ODBC driver for our database
  - This is the actual driver used by the `odbc` package that communicates with the database.
  - This is the implementation of the ODBC standard for the database we're using.

7.5.2 Connecting with the `odbc` package

For this example, let's say that we're using an SQL Server database, and so we've got the ODBC Driver 17 for SQL Server installed.

First, let's make sure we've got the driver detected:

```
odbc::odbcListDrivers()
```

```
##                             name   attribute
## 1 ODBC Driver 17 for SQL Server Description
## 2 ODBC Driver 17 for SQL Server      Driver
## 3 ODBC Driver 17 for SQL Server  UsageCount
##                                                          value
## 1                    Microsoft ODBC Driver 17 for SQL Server
## 2 /opt/microsoft/msodbcsql17/lib64/libmsodbcsql-17.7.so.2.1
## 3                                                          1
```

We can see our driver has been detected. Now we can use the `odbc::dbConnect()` function to create a connection to our database that we can then use to run queries:

```
my_connection <- odbc::dbConnect(drv = odbc::odbc(),
                                 driver = "ODBC Driver 17 for SQL Server",
                                 # You'll need to change this to your server
                                 server = "SQLDATABASESERVER",
                                 # You'll need to change this to your database
                                 database = "SQLDATABASE")
```

With the `drv` parameter we're specifying the type of driver that we're using. Because we're using an ODBC driver, we can use the `odbc::odbc()` function. The `driver` parameter expects a character string of the actual driver we're going to use (not the type), so that's where we enter the name we saw when we ran the `odbc::odbcListDrivers()` function. The `server` is the server that your database is being on hosted on and the `database` parameter is the database on the server that you want to connect to.

The `odbc` will use this information to create a connection string[7] that it will then use to try and connect to the database. If there are any other values you need to include in the connection string, you can include them as named parameters and they'll be added to the string:

```
my_connection <- odbc::dbConnect(drv = odbc::odbc(),
                                 driver = "ODBC Driver 17 for SQL Server",
                                 # You'll need to change this to your server
                                 server = "SQLDATABASESERVER",
                                 # You'll need to change this to your database
                                 database = "SQLDATABASE",
                                 # This would add an 'extra_parameter' entry to the string
                                 extra_parameter = "extra_parameter_value")
```

[7] https://www.connectionstrings.com/

Note: You'll want to make sure you assign your connection to something because we'll be passing the connection object to some other functions soon.

7.5.2.1 Authentication

You might also have to specify some credentials when you try and connect to your database. To provide a username and password, just use the `uid` and `pwd` parameters.

If your database supports it, you can use Windows Authentication, meaning that you don't need to provide an explicit username and password - the database will use your Windows account instead. To force the driver to try and use this type of authentication, you can add a `Trusted_Connection` parameter to the function call and set the value to `"Yes"` (not `TRUE` or `"TRUE"`). This parameter is then added to the connection string via the `...` argument of the `odbc::dbConnect()` function.

7.5.2.2 DSNs

Instead of specifying this information via the `odbc::dbConnect()` function, you can also create a Data Source Name (DSN) entry. This contains essentially the same information as we provided (the server location, the database, access credentials and so on) but then allows us to just use the `dsn` parameter of the `odbc::dbConnect()` function.

A DSN entry might look like this:

```
[MyDatabase]
Driver = /opt/microsoft/msodbcsql17/lib64/libmsodbcsql-17.7.so.2.1
Server = SQLDATABASESERVER
Database = SQLDATABASE
Port = 1234
```

We can then just provide the DSN name to the `odbc::dbConnect()` function:

```
my_connection <- odbc::dbConnect(drv = odbc::odbc(),
                                 dsn = "MyDatabase")
```

This can be particularly useful if you're working in more than one environment, where you might want to connect to the same database but the connection string is going to be different for each environment. Instead, you can create a Data Source Name entry with the same name but with different specifications, and then use the same `odbc::dbConnect(drv = odbc::odbc(), dsn = "MyDatabase")` call in both.

I won't go into exactly how to add DSNs here because it depends on your driver and your OS, but the process is pretty simple once you've found the right information for your setup.

7.5.3 Querying the database

Now we've got our connection (`my_connection`), we can send queries to the database. To send queries and get back the results in one step, we can use the `odbc::dbGetQuery()` function, passing the connection and then the string containing the SQL we want to execute.

```
odbc::dbGetQuery(my_connection, "select top 1 Id from dbo.ExampleTable")
```

```
##   Id
## 1  1
```

We then get back the data as a normal R data.frame. R will deal with the data type conversion, but the exact conversion is dependent on the driver. Luckily, Microsoft provides a complete breakdown[8] of how data types are mapped between on R and SQL Server on their website.

[8] https://docs.microsoft.com/en-us/sql/machine-learning/r/r-libraries-and-data-types?view=sql-server-ver15

Chapter 8

Cleaning

Once you've got your data into R, you'll likely need to clean it up a bit. Here are some of the more common operations that you'll be doing when it comes to data cleaning:

- Filtering
- Removing/replacing missing values
- Changing column types

Let's look at how we might do these tasks in R using our Kaggle dataset as an example.

8.1 Filtering

In some cases, you'll only want a subset of the data that's been provided to you. To filter the dataset down, use the `dplyr::filter()` function and provide your criteria:

```
vg_sales %>%
  dplyr::filter(Year > 200) %>%
  print(n = 5)
```

```
## # A tibble: 16,327 x 11
##    Rank Name  Platform  Year Genre Publisher NA_Sales EU_Sales JP_Sales
##   <dbl> <chr> <chr>    <dbl> <chr> <chr>        <dbl>    <dbl>    <dbl>
## 1     1 Wii ~ Wii       2006 Spor~ Nintendo      41.5    29.0      3.77
## 2     2 Supe~ NES       1985 Plat~ Nintendo      29.1     3.58     6.81
## 3     3 Mari~ Wii       2008 Raci~ Nintendo      15.8    12.9      3.79
```

```
## 4     4 Wii ~ Wii       2009 Spor~ Nintendo      15.8    11.0      3.28
## 5     5 Poke~ GB        1996 Role~ Nintendo      11.3     8.89    10.2
## # ... with 16,322 more rows, and 2 more variables: Other_Sales <dbl>,
## #   Global_Sales <dbl>
```

You can utilise the `&` and `|` operators (`and` and `or` respectively) to create more complicated criteria. You can also separate your expressions with commas, which is equivalent to `&`:

```
vg_sales %>%
  dplyr::filter(Year == 1999 | Year == 2000) %>%
  print(n = 2)
```

```
## # A tibble: 687 x 11
##    Rank Name  Platform  Year Genre Publisher NA_Sales EU_Sales JP_Sales
##   <dbl> <chr> <chr>    <dbl> <chr> <chr>        <dbl>    <dbl>    <dbl>
## 1    13 Poke~ GB        1999 Role~ Nintendo      9        6.18     7.2
## 2    70 Gran~ PS        1999 Raci~ Sony Com~     3.88     3.42     1.69
## # ... with 685 more rows, and 2 more variables: Other_Sales <dbl>,
## #   Global_Sales <dbl>
```

```
vg_sales %>%
  dplyr::filter(Year == 1999, Platform == "GB") %>%
  # equivalent to dplyr::filter(Year == 1999 & Platform == "GB")
  print(n = 2)
```

```
## # A tibble: 11 x 11
##    Rank Name  Platform  Year Genre Publisher NA_Sales EU_Sales JP_Sales
##   <dbl> <chr> <chr>    <dbl> <chr> <chr>        <dbl>    <dbl>    <dbl>
## 1    13 Poke~ GB        1999 Role~ Nintendo      9        6.18     7.2
## 2   172 Poke~ GB        1999 Misc  Nintendo      3.02     1.12     1.01
## # ... with 9 more rows, and 2 more variables: Other_Sales <dbl>,
## #   Global_Sales <dbl>
```

8.2 Missing values

Missing values are common in data science - data collection is often imperfect and so you'll end up with observations or data-points missing. Firstly, you need to decide what you're going to do with those.

The easiest approach to just to remove them, and we can do that with the `dplyr::filter()` and the `is.na()` functions. If you remember in the Loading chapter, `{readr}` had trouble parsing some of the values in our Year column to numeric, and so gave us some NAs. Let's remove those rows using this approach:

```
## To remove rows with NA in one column
clean_vg_sales <- vg_sales %>%
  dplyr::filter(!is.na(Year))
print(clean_vg_sales, n = 5)
```

```
## # A tibble: 16,327 x 11
##    Rank Name  Platform  Year Genre Publisher NA_Sales EU_Sales JP_Sales
##   <dbl> <chr> <chr>    <dbl> <chr> <chr>        <dbl>    <dbl>    <dbl>
## 1     1 Wii ~ Wii       2006 Spor~ Nintendo      41.5    29.0      3.77
## 2     2 Supe~ NES       1985 Plat~ Nintendo      29.1     3.58     6.81
## 3     3 Mari~ Wii       2008 Raci~ Nintendo      15.8    12.9      3.79
## 4     4 Wii ~ Wii       2009 Spor~ Nintendo      15.8    11.0      3.28
## 5     5 Poke~ GB        1996 Role~ Nintendo      11.3     8.89    10.2
## # ... with 16,322 more rows, and 2 more variables: Other_Sales <dbl>,
## #   Global_Sales <dbl>
```

```
## To remove rows with NA in any column
vg_sales %>%
  dplyr::filter(dplyr::across(dplyr::everything(), ~!is.na(.x))) %>%
  print(n = 5)
```

```
## # A tibble: 16,327 x 11
##    Rank Name  Platform  Year Genre Publisher NA_Sales EU_Sales JP_Sales
##   <dbl> <chr> <chr>    <dbl> <chr> <chr>        <dbl>    <dbl>    <dbl>
## 1     1 Wii ~ Wii       2006 Spor~ Nintendo      41.5    29.0      3.77
## 2     2 Supe~ NES       1985 Plat~ Nintendo      29.1     3.58     6.81
## 3     3 Mari~ Wii       2008 Raci~ Nintendo      15.8    12.9      3.79
## 4     4 Wii ~ Wii       2009 Spor~ Nintendo      15.8    11.0      3.28
## 5     5 Poke~ GB        1996 Role~ Nintendo      11.3     8.89    10.2
## # ... with 16,322 more rows, and 2 more variables: Other_Sales <dbl>,
## #   Global_Sales <dbl>
```

Another approach to replace them with either the average for that column or with the closest neighbour value (this works better with time series data).

To replace with the nearest value, we can use the `tidyr::fill()` function. That approach wouldn't be appropriate for our video games example so let's use the `datasets::airquality` dataset instead:

```
head(datasets::airquality, 5)
```

```
##   Ozone Solar.R Wind Temp Month Day
## 1    41     190  7.4   67     5   1
## 2    36     118  8.0   72     5   2
```

```
## 3    12     149 12.6   74     5   3
## 4    18     313 11.5   62     5   4
## 5    NA      NA 14.3   56     5   5
```

```
## Fill a single column
datasets::airquality %>%
  tidyr::fill(Ozone, .direction = "downup") %>%
  head(5)
```

```
##   Ozone Solar.R Wind Temp Month Day
## 1    41     190  7.4   67     5   1
## 2    36     118  8.0   72     5   2
## 3    12     149 12.6   74     5   3
## 4    18     313 11.5   62     5   4
## 5    18      NA 14.3   56     5   5
```

```
## Fill all columns
datasets::airquality %>%
  tidyr::fill(dplyr::everything(), .direction = "downup") %>%
  head(5)
```

```
##   Ozone Solar.R Wind Temp Month Day
## 1    41     190  7.4   67     5   1
## 2    36     118  8.0   72     5   2
## 3    12     149 12.6   74     5   3
## 4    18     313 11.5   62     5   4
## 5    18     313 14.3   56     5   5
```

To replace them with the mean, we can either use a package like `zoo`, or we can use the `tidyverse` packages and our own function:

```
replace_with_mean <- function(x) {
  replace(x, is.na(x), mean(x, na.rm = TRUE))
}

datasets::airquality %>%
  # The mutate function creates new columns or overwrites existing ones
  dplyr::mutate(dplyr::across(dplyr::everything(), replace_with_mean)) %>%
  head(5)
```

```
##      Ozone  Solar.R Wind Temp Month Day
## 1 41.00000 190.0000  7.4   67     5   1
## 2 36.00000 118.0000  8.0   72     5   2
## 3 12.00000 149.0000 12.6   74     5   3
## 4 18.00000 313.0000 11.5   62     5   4
## 5 42.12931 185.9315 14.3   56     5   5
```

A word of warning here, however. If you're doing complex modelling or very sensitive analyses, filling values like this can be misleading at best. Always think about the best approach for your specific project and what the repercussions of filling empty values might be.

8.3 Changing column types

When you import data into R, sometimes the type of the column doesn't match what you want it to be. One way of tackling this is to define your column types when you import the data (as we looked at before), but it's also perfectly acceptable to change the column type after the import.

Probably the most common conversion is from a character string to a date. For this example, we're just going to use some test data:

```
bad_tibble <- tibble::tribble(~bad_date, ~value,
                              "2012/01/01", 100,
                              "2014/06/01", 200)

bad_tibble %>%
  dplyr::mutate(good_date = as.Date(bad_date, format = "%Y/%m/%d"))
```

```
## # A tibble: 2 x 3
##   bad_date   value good_date
##   <chr>      <dbl> <date>
## 1 2012/01/01   100 2012-01-01
## 2 2014/06/01   200 2014-06-01
```

Essentially, all you need to do is wrap your conversion function (e.g. `as.Date()`, `as.character()`) in a `dplyr::mutate()` call and you should be able to change your columns to whatever you need. To replace the original column, make sure you give it the same name when using `dplyr::mutate()`:

```
bad_tibble %>%
  # This will replace the bad_date column rather than creating a new column
  dplyr::mutate(bad_date = as.Date(bad_date, format = "%Y/%m/%d"))
```

```
## # A tibble: 2 x 2
##   bad_date   value
##   <date>     <dbl>
## 1 2012-01-01   100
## 2 2014-06-01   200
```

Chapter 9

Tidying

Now that you know your dataset is clean, you'll want to get it into a format that is amenable to your analysis. This is the **tidying** stage. To better understand what "tidy" data is and why we're trying to get our data into this format, make sure you've read the tidyverse chapter.

9.1 Pivoting

The main culprit of untidy data is data that breaks the first rule: Not every column is a variable. Data in this format is sometimes called wide data - as you add more levels, you'll add more columns and so the data will get wider. Conversely, tidy data will get longer as you add more levels.

So there two different forms of data:

- Long
- Wide

Let's look at use our Kaggle dataset as an example to see how the two forms differ in practice:

```
## # A tibble: 16,327 x 7
##    Year Name              NA_Sales EU_Sales JP_Sales Other_Sales Global_Sales
##   <dbl> <chr>                <dbl>    <dbl>    <dbl>       <dbl>        <dbl>
## 1  2006 Wii Sports            41.5    29.0      3.77        8.46         82.7
## 2  1985 Super Mario Bros.     29.1     3.58     6.81        0.77         40.2
## 3  2008 Mario Kart Wii        15.8    12.9      3.79        3.31         35.8
## # ... with 16,324 more rows
```

What would happen if we added data for more countries? We'd be adding more columns, and so our data would get wider. So our `clean_vg_sales` data is currently in the wide format. If we had the country as its own column however, like this:

```
## # A tibble: 81,635 x 3
##    Year Country Sales
##   <dbl> <chr>   <dbl>
## 1  2006 NA      41.5
## 2  2006 EU      29.0
## 3  2006 JP       3.77
## 4  2006 Other    8.46
## 5  2006 Global  82.7
## # ... with 81,630 more rows
```

Now we've got the country as a variable, the more countries we add the longer the data is going to get. So we've converted our data from the wide format to the long format. To do this, we've pivoted the Sales columns. Let's look at the two types of pivot.

Note: One thing to keep in mind is that long data does not always mean tidy, but wide data can never be tidy. It's a subtle distinction but it's important to remember.

9.1.1 Pivoting columns to rows (longer)

To convert wide data to long, we need to pivot the columns to rows. To this using `{tidyr}`, we use the `pivot_longer()` function because we want to pivot from the wide format to the long format.

To do this in the simplest way, we just need to tell the function which columns we want to pivot:

```
tidyr::pivot_longer(clean_vg_sales, cols = c(NA_Sales, EU_Sales, JP_Sales, Other_Sales
  # We'll just look at the columns we're interested in for the moment
  dplyr::select(Name, Year, name, value) %>%
  print(n = 2)
```

```
## # A tibble: 81,635 x 4
##   Name        Year name     value
##   <chr>      <dbl> <chr>    <dbl>
## 1 Wii Sports  2006 NA_Sales  41.5
## 2 Wii Sports  2006 EU_Sales  29.0
## # ... with 81,633 more rows
```

This is a good start. Now we've converted to long format, we're abiding by the three rules and so we've got a tidy dataset! But there's definitely some improvements to be done. Firstly, "name" and "value" aren't the best names we could come up with for these columns, so we should probably use some new ones. To do this, we just need to provide new names to the `names_to` and `values_to` parameters:

```
tidyr::pivot_longer(clean_vg_sales,
                    cols = c(NA_Sales, EU_Sales, JP_Sales, Other_Sales, Global_Sales),
                    names_to = "Country",
                    values_to = "Sales") %>%
  dplyr::select(Name, Year, Country, Sales) %>%
  print(n = 2)
```

```
## # A tibble: 81,635 x 4
##   Name        Year Country  Sales
##   <chr>      <dbl> <chr>    <dbl>
## 1 Wii Sports  2006 NA_Sales  41.5
## 2 Wii Sports  2006 EU_Sales  29.0
## # ... with 81,633 more rows
```

Secondly, our Country column has the country code suffixed with '_Sales' (e.g. "NA_Sales" instead of just "NA"). If the text we wanted to remove was before the country name, we could utilise the `names_prefix` parameter to remove matching text from the start of each variable name:

```
tidyr::pivot_longer(clean_vg_sales,
                    cols = c(NA_Sales, EU_Sales, JP_Sales, Other_Sales, Global_Sales),
                    names_to = "Country",
                    values_to = "Sales",
                    names_prefix = "(_Sales)") # Remove text matching "_Sales" exactly
```

Because it's at the end of the string though, we have to deal with it a bit differently. One way to remove that text would be to split the column name by the "_" and then just exclude the second part:

```
tidy_vg_sales <- clean_vg_sales %>%
  tidyr::pivot_longer(cols = c(NA_Sales, EU_Sales, JP_Sales, Other_Sales, Global_Sales),
                      names_to = c("Country", NA), values_to = "Sales", names_sep = "_"
  )

tidy_vg_sales %>%
  dplyr::select(Year, Country, Sales) %>%
  print(n = 5)
```

```
## # A tibble: 81,635 x 3
##    Year Country Sales
##   <dbl> <chr>   <dbl>
## 1  2006 NA      41.5
## 2  2006 EU      29.0
## 3  2006 JP       3.77
## 4  2006 Other    8.46
## 5  2006 Global  82.7
## # ... with 81,630 more rows
```

Here, we've specified that we want to convert our column names to two separate 'names' columns, but that we want to exclude the second column (that's what the `NA` in the `c("Country", NA)` vector represents). Then, we've specified with the `names_sep` parameter that we want to split the columns by the `_` character.

But what happens now if we get sent the same dataset but with more countries (e.g. a `SA_Sales` column)? We'd have to add those extra columns to our `cols` vector manually. Instead, we can use some `tidyselect` syntax to choose columns based on their features, like how they end:

```
clean_vg_sales %>%
  tidyr::pivot_longer(cols = tidyselect::ends_with("_Sales"),
                      names_to = c("Country", NA), values_to = "Sales", names_sep = "_"
  )
```

Now our code will work with any number of columns as long as they end in '_Sales'.

This is just one example of how we could utilise the `pivot_longer()` function to convert 'messy' data to the tidy format. In the wild you'll get data that violates any of the three rules of tidy data in many different ways, and we just don't have the space to go through it here. If you do need how to convert some of the other forms of messy data, then the `{tidyr}` package has a number of vignettes[1] outlining many different ways to use the tidying tools it provides.

9.1.2 Pivoting rows to columns (wider)

Most of the time you should be going from wide to long, but we'll go through how to do the reverse for the occasions where it's required. To transform the data to the wide format, we use the `pivot_wider()` function. At the simplest level, we just need to provide where the name comes from and where the values come from with the `names_from` and `values_from` parameters respectively:

[1] https://tidyr.tidyverse.org/articles/index.html

```
tidyr::pivot_wider(tidy_vg_sales, names_from = Country, values_from = Sales)
```

```
## # A tibble: 16,327 x 11
##     Rank Name     Platform  Year Genre  Publisher  `NA`    EU    JP Other Global
##    <dbl> <chr>    <chr>    <dbl> <chr>  <chr>     <dbl> <dbl> <dbl> <dbl>  <dbl>
##  1     1 Wii Spo~ Wii       2006 Sports Nintendo   41.5 29.0   3.77 8.46    82.7
##  2     2 Super M~ NES       1985 Platf~ Nintendo   29.1  3.58  6.81 0.77    40.2
##  3     3 Mario K~ Wii       2008 Racing Nintendo   15.8 12.9   3.79 3.31    35.8
##  4     4 Wii Spo~ Wii       2009 Sports Nintendo   15.8 11.0   3.28 2.96    33
##  5     5 Pokemon~ GB        1996 Role-~ Nintendo   11.3  8.89 10.2  1       31.4
##  6     6 Tetris   GB        1989 Puzzle Nintendo   23.2  2.26  4.22 0.580   30.3
##  7     7 New Sup~ DS        2006 Platf~ Nintendo   11.4  9.23  6.5  2.9     30.0
##  8     8 Wii Play Wii       2006 Misc   Nintendo   14.0  9.2   2.93 2.85    29.0
##  9     9 New Sup~ Wii       2009 Platf~ Nintendo   14.6  7.06  4.7  2.26    28.6
## 10    10 Duck Hu~ NES       1984 Shoot~ Nintendo   26.9  0.63  0.28 0.47    28.3
## # ... with 16,317 more rows
```

To get our data more like we started, we can use the `names_glue` to provide a `{glue}`[2] string argument to create the column names from a template:

```
tidyr::pivot_wider(tidy_vg_sales, names_from = Country, values_from = Sales,
                   names_glue = "{Country}_Sales") %>%
  dplyr::select(Year, Name, tidyselect::ends_with("_Sales")) %>%
  print(n = 5)
```

```
## # A tibble: 16,327 x 7
##    Year Name                 NA_Sales EU_Sales JP_Sales Other_Sales Global_Sales
##   <dbl> <chr>                   <dbl>    <dbl>    <dbl>       <dbl>        <dbl>
## 1  2006 Wii Sports               41.5    29.0      3.77        8.46         82.7
## 2  1985 Super Mario Bros.        29.1     3.58     6.81        0.77         40.2
## 3  2008 Mario Kart Wii           15.8    12.9      3.79        3.31         35.8
## 4  2009 Wii Sports Resort        15.8    11.0      3.28        2.96         33
## 5  1996 Pokemon Red/Pokemon~     11.3     8.89    10.2         1            31.4
## # ... with 16,322 more rows
```

9.2 Separating & Uniting

9.2.1 Separating

A less common violation of the tidy data principles occurs when a single column contains more than one variable. Let's look at an example:

[2] https://glue.tidyverse.org/

```
## # A tibble: 4 x 2
##   Species    LeafArea
##   <chr>      <chr>
## 1 Eucalyptus 20x100
## 2 Elm        10x5
## 3 Agave      2x10
## 4 Persea     15x11
```

Here we've got a dataset with different plant species and the area of their leaves. But we've got two variables combined into a single column; we've got both the width and the length of the leaf included in a single 'area' variable. To convert this dataset to the tidy format, we need to split out those values into separate 'width' and 'length' variables.

We can use the `separate()` function from the `{tidyr}` package to do this. We just need to provide the column we want to split via the `col` parameter, the column names that we want to split the original column into with the `into` parameter, and the separator that's separating the two or more values within the column via the `sep` parameter:

```
separate_example <- tidyr::separate(combined_example, col = LeafArea, into = c("Width"
print(separate_example)
```

```
## # A tibble: 4 x 3
##   Species    Width Length
##   <chr>      <chr> <chr>
## 1 Eucalyptus 20    100
## 2 Elm        10    5
## 3 Agave      2     10
## 4 Persea     15    11
```

9.2.2 Uniting

Just like pivoting wider was essentially the inverse of pivoting longer, uniting columns is the inverse of separating them. Again, our focus in this chapter is to tidy up our data, and so we'll likely always be separating columns rather than uniting them. Regardless, it's good to understand how to unite columns for the occasions that it is required.

To join multiple columns together, we use the `unite()` function from the `{tidyr}` package. We just need to provide the name of the new column we want to create via the `col` parameter, the column names of the columns to unite as unnamed parameters, and then the separating character we want to use via `sep`:

```
tidyr::unite(separate_example, col = "LeafArea", Width, Length, sep = "x")
```

```
## # A tibble: 4 x 2
##   Species    LeafArea
##   <chr>      <chr>
## 1 Eucalyptus 20x100
## 2 Elm        10x5
## 3 Agave      2x10
## 4 Persea     15x11
```

Chapter 10

Grouping

Before we look at adding new columns and summarising our data, we need to understand the concept of grouping our data.

When a data frame has one or more groups, any function that's applied (when summarising or adding a new column) is applied for each combination of those groups. So say we have a dataset like this:

```
head(ungrpd, 5)
```

```
##   Year Month Group Value
## 1 2012     1     A    10
## 2 2012     1     B    20
## 3 2012     2     A    30
## 4 2012     2     B    15
## 5 2012     3     A    25
```

If we grouped by Year or by Month, then any function we applied would be applied separately to each value of those groups. So say we wanted get the total values for each Year, we could group by the `Year` column and then just sum the `Value` column. The same logic applies for if we wanted to get the total for each year *and* month; we could group by the `Year` and `Month` and sum the `Value` column and we'd then get a value for each distinct year and month.

To group a dataset, we use the `dplyr::group_by()` function:

```
grpd <- dplyr::group_by(ungrpd, Year)
```

The `dplyr::group_by()` function returns the same dataset but now grouped by the variables you provided. At first, it might not seem as though anything happened - if we print the dataset we still get basically the same output...

```
print(grpd, n = 5)
```

```
## # A tibble: 12 x 4
## # Groups:   Year [2]
##   Year  Month Group Value
##   <chr> <dbl> <chr> <dbl>
## 1 2012      1 A        10
## 2 2012      1 B        20
## 3 2012      2 A        30
## 4 2012      2 B        15
## 5 2012      3 A        25
## # ... with 7 more rows
```

Except now we have a new entry `Groups:`. This tells us that the dataset has been grouped and by what variables. This means that any subsequent summarisation or mutation we do will be done relative those groups.

To test whether a dataset has been grouped, you can use the `dplyr::is_grouped_df()`, and to ungroup a dataset, just use `dplyr::ungroup()`:

```
dplyr::is_grouped_df(grpd)
```

```
## [1] TRUE
```

```
dplyr::is_grouped_df(ungrpd)
```

```
## [1] FALSE
```

```
print(dplyr::ungroup(grpd), n = 5)
```

```
## # A tibble: 12 x 4
##   Year  Month Group Value
##   <chr> <dbl> <chr> <dbl>
## 1 2012      1 A        10
## 2 2012      1 B        20
## 3 2012      2 A        30
## 4 2012      2 B        15
## 5 2012      3 A        25
## # ... with 7 more rows
```

Chapter 11

Mutating

Sometimes you'll want to add new columns to your data. Most of the time, these will be calculated columns that can be created based on one or more of the other columns in the dataset. To create new columns, we use the `dplyr::mutate()` function.

First off, let's add something simple; the Sales column in our dataset is in millions. Let's convert that to normal units. All we need to do is provide the name of our new column to the `dplyr::mutate()` function and how our new column should be calculated:

```
tidy_vg_sales %>%
  dplyr::mutate(Sales_units = Sales * 1000000) %>%
  dplyr::select(Rank, Name, Platform, Year, Genre, Publisher, Sales_units) %>%
  print(n = 5)
```

```
## # A tibble: 81,635 x 7
##    Rank Name       Platform  Year Genre  Publisher Sales_units
##   <dbl> <chr>      <chr>    <dbl> <chr>  <chr>           <dbl>
## 1     1 Wii Sports Wii       2006 Sports Nintendo     41490000
## 2     1 Wii Sports Wii       2006 Sports Nintendo     29020000
## 3     1 Wii Sports Wii       2006 Sports Nintendo      3770000
## 4     1 Wii Sports Wii       2006 Sports Nintendo      8460000
## 5     1 Wii Sports Wii       2006 Sports Nintendo     82740000
## # ... with 81,630 more rows
```

As you can see, we've created a new column called `EU_Proportion` and we've calculated by dividing the number of EU sales by the total number of sales.

If we wanted to overwrite the Sales column, we do the same but give the new column the same name:

```
tidy_vg_sales %>%
  dplyr::mutate(Sales = Sales * 1000000) %>%
  dplyr::select(Rank, Name, Platform, Year, Genre, Publisher, Sales) %>%
  print(n = 5)
```

```
## # A tibble: 81,635 x 7
##    Rank Name       Platform  Year Genre  Publisher    Sales
##   <dbl> <chr>      <chr>    <dbl> <chr>  <chr>        <dbl>
## 1     1 Wii Sports Wii       2006 Sports Nintendo  41490000
## 2     1 Wii Sports Wii       2006 Sports Nintendo  29020000
## 3     1 Wii Sports Wii       2006 Sports Nintendo   3770000
## 4     1 Wii Sports Wii       2006 Sports Nintendo   8460000
## 5     1 Wii Sports Wii       2006 Sports Nintendo  82740000
## # ... with 81,630 more rows
```

While this is certainly a powerful tool, we're not really changing the world here. However, we can create more complicated columns by leveraging the concept of applying our function to groups.

When our dataset is grouped, our `mutate()` call will be evaluated for each permutation of the provided groups. Let's look at an example of creating a cumulative sum for each Publisher and Country:

```
tidy_vg_sales %>%
  dplyr::group_by(Publisher, Country) %>%
  dplyr::arrange(Year) %>% # We need to order from earliest date to latest
  dplyr::mutate(Cumulative_Sales = cumsum(Sales)) %>%
  dplyr::filter(Publisher %in% c("Atari", "Activision")) %>% # Let's just look at Atar
  print(n = 5)
```

```
## # A tibble: 6,565 x 9
## # Groups:   Publisher, Country [10]
##    Rank Name      Platform  Year Genre  Publisher Country Sales Cumulative_Sales
##   <dbl> <chr>     <chr>    <dbl> <chr>  <chr>     <chr>   <dbl>            <dbl>
## 1   259 Asteroids 2600      1980 Shoot~ Atari     NA       4                4
## 2   259 Asteroids 2600      1980 Shoot~ Atari     EU       0.26             0.26
## 3   259 Asteroids 2600      1980 Shoot~ Atari     JP       0                0
## 4   259 Asteroids 2600      1980 Shoot~ Atari     Other    0.05             0.05
## 5   259 Asteroids 2600      1980 Shoot~ Atari     Global   4.31             4.31
## # ... with 6,560 more rows
```

Now we've got a new column showing the total number of sales for each Publisher up to each year. What `{dplyr}` has done is essentially filtered the data down a specific publisher and country, created a cumulative sum of the `Sales` column, and then done that for each publisher and country and stitched it back together.

Chapter 12

Summarising

As we saw previously, the base videa game sales dataset has nearly 17,000 entries! That's a lot of data, and we're not really going to able to extract or show any real insights from this data without summarising it to some degree. What we want to do is summarise the data to some degree. We lose some of the granularity of the data but it allows is to understand the data a bit better and identify patterns.

Summarisation uses exactly the same grouping concept as mutatation does; by grouping the data we change the scope of each application of the function to be limited to that group.

Let's go through an example. Let's say that we want to get the global sales for each publisher in each year. For now, we don't care about genre or platform. To do this, we'll need to group by Publisher, Year and Country:

```
tidy_vg_sales %>%
  dplyr::group_by(Publisher, Year, Country)
```

And then we'll want to sum up the `Sales` column to get our totals:

```
tidy_vg_sales %>%
  dplyr::group_by(Publisher, Year, Country) %>%
  dplyr::summarise(Total_Sales = sum(Sales), .groups = "drop") %>%
  # The .groups = 'drop' parameter will automatically remove the grouping from the dataset
  print(n = 5)
```

```
## # A tibble: 11,645 x 4
##   Publisher        Year Country Total_Sales
##   <chr>           <dbl> <chr>         <dbl>
```

```
## 1 10TACLE Studios  2006 EU             0.01
## 2 10TACLE Studios  2006 Global         0.02
## 3 10TACLE Studios  2006 JP             0
## 4 10TACLE Studios  2006 NA             0.01
## 5 10TACLE Studios  2006 Other          0
## # ... with 11,640 more rows
```

Now we can see that the number of rows has reduced down to just under 2,000, and we've got the total global sales for each of our publishers, and for each year. `{dplyr}` has looked through each publisher and each year that publisher has at least 1 game, and added up the total number of sales, then moved onto the next Publisher/Year combination.

While we're here, let's lump together some of the less popular publishers using the `forcats::fct_lump_n()` function:

```
smmrsd_vg_sales <- tidy_vg_sales %>%
  dplyr::group_by(Publisher, Year, Country) %>%
  dplyr::summarise(Total_Sales = sum(Sales), .groups = "drop") %>%
  dplyr::mutate(Publisher = forcats::fct_lump_n(Publisher, n = 5, w = Total_Sales)) %>%
  dplyr::group_by(Publisher, Year, Country) %>%
  dplyr::summarise(Total_Sales = sum(Total_Sales), .groups = "drop")
# The .groups = 'drop' parameter will automatically remove the grouping from the datas
print(smmrsd_vg_sales, n = 5)
```

```
## # A tibble: 870 x 4
##   Publisher   Year Country Total_Sales
##   <fct>      <dbl> <chr>         <dbl>
## 1 Activision  1980 EU             0.18
## 2 Activision  1980 Global         3.02
## 3 Activision  1980 JP             0
## 4 Activision  1980 NA             2.82
## 5 Activision  1980 Other          0.03
## # ... with 865 more rows
```

In this case we've used the `sum()` function, but any kind of summary function can be used:

```
tidy_vg_sales %>%
  dplyr::group_by(Publisher, Year, Country) %>%
  dplyr::summarise(Total_Sales = sum(Sales),
                   Average_Sales = mean(Sales),
                   .groups = "drop")
```

```
## # A tibble: 11,645 x 5
```

```
##    Publisher        Year Country Total_Sales Average_Sales
##    <chr>           <dbl> <chr>         <dbl>         <dbl>
##  1 10TACLE Studios  2006 EU             0.01         0.01
##  2 10TACLE Studios  2006 Global         0.02         0.02
##  3 10TACLE Studios  2006 JP             0            0
##  4 10TACLE Studios  2006 NA             0.01         0.01
##  5 10TACLE Studios  2006 Other          0            0
##  6 10TACLE Studios  2007 EU             0.03         0.015
##  7 10TACLE Studios  2007 Global         0.09         0.045
##  8 10TACLE Studios  2007 JP             0            0
##  9 10TACLE Studios  2007 NA             0.06         0.03
## 10 10TACLE Studios  2007 Other          0            0
## # ... with 11,635 more rows
```

Because `dplyr::mutate()` and `dplyr::summarise()` both use a similar syntax and both function on grouped datasets, using the correct function when starting out can be tough. The best way to remember which one to use is to ask "How many rows am I expecting back from this?". If the answer is fewer than you've got now, you'll want the `summarise()` function. Otherwise you're looking at `mutate()`.

Chapter 13

Plotting

Now that you've got your data summarised to a suitable level, you might want to create some graphics to help gain insight into the trends and patterns in the data. For this section, we're going to rely on the `{ggplot2}` package from the tidyverse. This is arguably the most advanced plotting package available for R. R does provide plotting functions without the use of packages, but we're going to focus on `{ggplot2}` here.

13.1 ggplot2

`{ggplot2}` is based on the concept of a Grammar of Graphics[1]. This is the concept that, like language, graphics have a grammar that allow us to describe their components.

This concept was originally proposed by Leland Wilkinson, but has been adopted heavily in the `{ggplot2}` philosophy, with Hadley Wickham publishing a paper titled 'A Layered Grammer of Graphics'[2] that outlines his proposal for a layered grammar and how it's implemented in `{ggplot2}`. We'll cover the real basics of `{ggplot2}` here but I'd recommend reading the paper and reading the `{ggplot2}` documentation[3] if you're interested in gaining a deeper understanding.

13.1.1 Components

At the core of the Grammar of Graphics philosophy is the idea that plots are defined by their components. For example, two scatterplots could be extremely

[1] https://www.springer.com/in/book/9780387245447
[2] http://vita.had.co.nz/papers/layered-grammar.html
[3] https://ggplot2.tidyverse.org/

different, even though we'd both call them scatterplots. Instead, the two graphics are better defined by the components that make them up.

When we create a plot using `{ggplot2}` we build it up by creating and combining these components.

The main components that make up a plot are:

- Data and aesthetics mappings
- Geometrics objects (or geoms)
- Statistical transformations (or stats)
- Scales
- Facets

These sound really scary, but they're actually super simple. Let's look at each one.

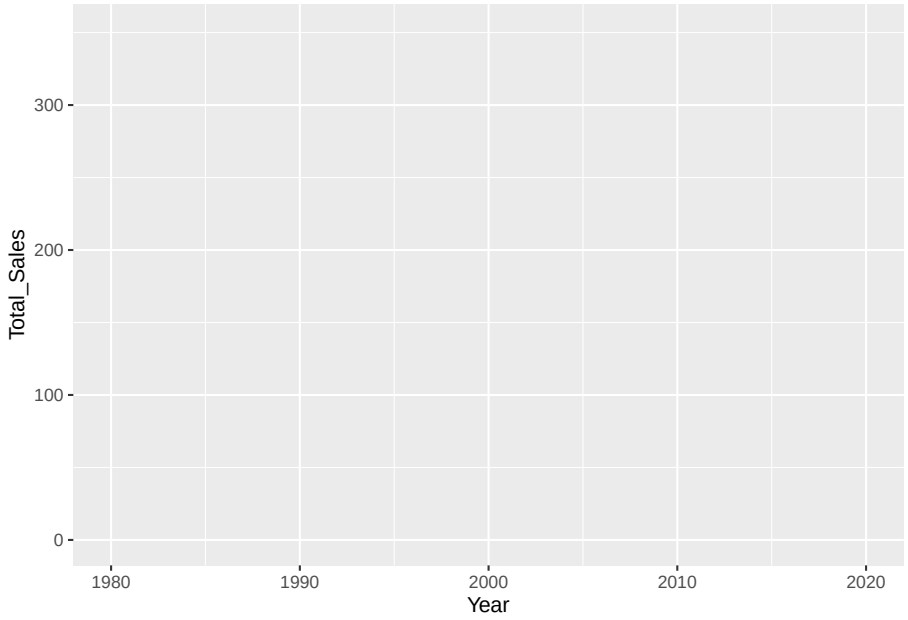
13.1.1.1 Data and aesthetic mappings

Every plot is a representation of some data. Therefore, to have a graphic, you need some data. That's simple enough.

But which parts of the data should be shown on the plot and where? This is what our aesthetic mappings represent. We might have a dataset with two variables, `z` and `w`. Our mapping may be then that we want the `z` variable on the x axis of the plot, and the `w` variable on the y axis. We can also utilise other mappings like colour and size.

To define our dataset and our aesthetic mappings using `{ggplot2}`, we use the `ggplot()` and `aes()` functions. Let's see an example using our video game sales experiment from the previous chapters:

```
library(ggplot2)

smmrsd_vg_sales %>%
  # Just plot global sales for now
  dplyr::filter(Country == "Global") %>%
  ggplot(mapping = aes(x = Year, y = Total_Sales, colour = Publisher))
```



Here we've defined that we want to show the `Year` on the X axis, the `Total_Sales` on the Y, and we want to map the `Publisher` to the colour of whatever we show on the plot.

As part of our mapping, we can see that `{ggplot2}` has also provided appropriate scales for our variables. It's provided a numeric scale with an appropriate range for the X and Y axis. We can't see it yet, but it's also assigned a scale to our colour aesthetic, mapping colours to values in the Publisher column. We'll look at manually specifying and customising the scales later.

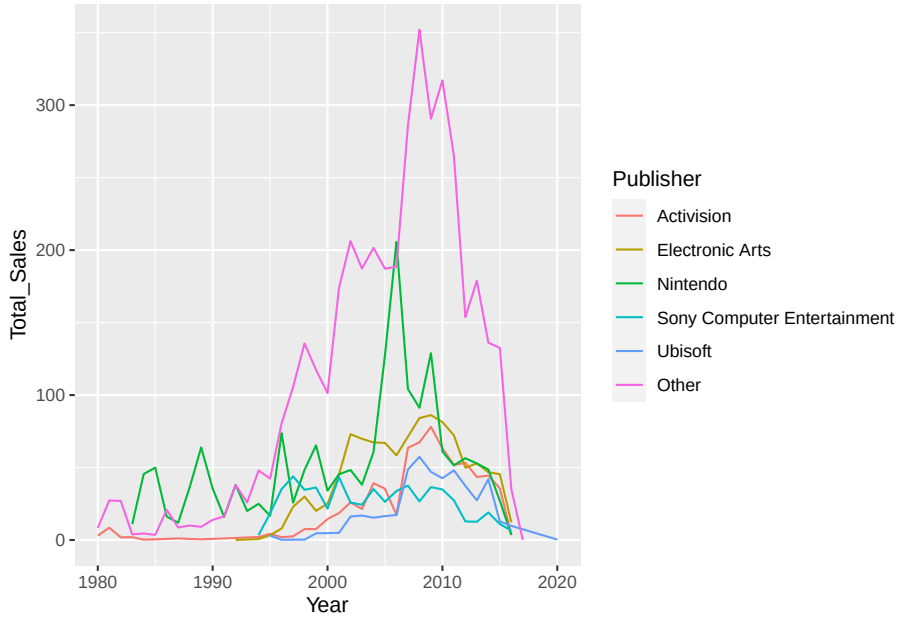
Although we've got our scales, and we can see that `{ggplot2}` clearly knows which variable we want on each axis, but there's nothing on the graph at the moment... This is because we now need to define our geoms.

13.1.1.2 Geometric objects (geoms)

Our geometric objects will be the object that's actually shown on the plot (e.g. bars, points, lines, and so on). To add our geometric object, we use the `geom_...` functions. Our geom will then inherit the mapping from our `ggplot()` call. If we wanted to add additional mappings, the `geom_...` functions also allow for a `mapping` parameter where additional mappings can be provided.

We want lines for our example, so we use the `geom_line()` function. To add our geom component, we just need to add it to our plot so far using the `+` operator:

```
smmrsd_vg_sales %>%
  dplyr::filter(Country == "Global") %>%
  ggplot(mapping = aes(x = Year, y = Total_Sales, colour = Publisher)) +
  geom_line()
```



This is equivalent to:

```
smmrsd_vg_sales %>%
  dplyr::filter(Country == "Global") %>%
  geom_line(mapping = aes(x = Year, y = Total_Sales, colour = Publisher))
```

Different geoms also allow for different mappings. For example, the line geom allows you to specify the `linetype` aesthetic that changes the way the line is drawn (solid, dashed, etc.) depending on the value of your variable. This aesthetic would mean nothing for the point geom however, which instead has an aesthetic called `shape`. To see which aesthetics are supported by which geoms, look at the documentation for the geom function you want to use (e.g `?geom_line`).

13.1.1.3 Statistical transformations (stats)

Before the data is plotted, it can go through a statistical transformation or 'stat'. This will take the value or values from the dataset and transform them.

For example, we might want to plot the sum of all of the values in a group rather than each value individually. We could use a statistical transformation that sums up all the values to do this for us. That way, we can plot the data in different ways with the same base dataset.

We can see that when we call `geom_line()` the default value for the `stat` parameter is 'identity'. This means that `{ggplot2}` performs no stat transformation on the data before it plots it - if the value of the variable on the x axis is 10, then the value that's plotted will be 10.

There are many predefined stats that `{ggplot2}` provides that we can to plot the data in different ways. To change the stat transformation for a geom, we can change the `stat` parameter to one of a set of predefined character strings that represent certain stats. For example, if we set the `stat` parameter for our `geom_line()` to `'smooth'`, the values are transformed by a smoothing function (the smoothing method can be changed via the `method` parameter):

```
smmrsd_vg_sales %>%
  dplyr::filter(Country == "Global") %>%
  geom_line(mapping = aes(x = Year, y = Total_Sales, colour = Publisher), stat = "smooth")
```

```
## mapping: x = ~Year, y = ~Total_Sales, colour = ~Publisher
## geom_line: na.rm = FALSE, orientation = NA
## stat_smooth: na.rm = FALSE, orientation = NA
## position_identity
```

Some geoms use a stat different to 'identity' by default. For example, using `geom_smooth()` uses the 'smooth' stat to essentially do the same as we just did with `geom_line()`. Similary, `geom_boxplot()` will automatically calculate the median and the whiskers and hinges from your data via the `"boxplot"` stat.

Stats are particularly useful because they allow us to plot the same data at different levels of summary. If we didn't utilise stats, we would have to transform the dataset before we started plotting. When we look at layers in one of the coming sections, this should help demonstrate the power of stat transformations in building meaningful graphics.

13.1.1.4 Facets

In our example, we've filtered the dataset down to just show the global sales, but instead we could also use faceting to show the data for each region in a different subplot. Faceting splits the original dataset, plots the data separately and then combines it into a single panel. To facet our plot, we use the `facet_wrap()` or `facet_grid()` functions:

```
ggplot(smmrsd_vg_sales, mapping = aes(x = Year, y = Total_Sales, colour = Publisher)) +
  geom_line() +
  facet_wrap(~Country, scales = "free_y", ncol = 2)
```

We use the ~ shorthand to say that we want to use the Publisher variable. We could also use the `dplyr::vars()` function (i.e. `facet_wrap(dplyr::vars(Publisher))`).

Then we specify that we want each subplot to have a free Y axis (rather than all using the same scale range) with `scales = "free_y"`, and that we want to have 2 columns of subplots with `ncol = 2`.

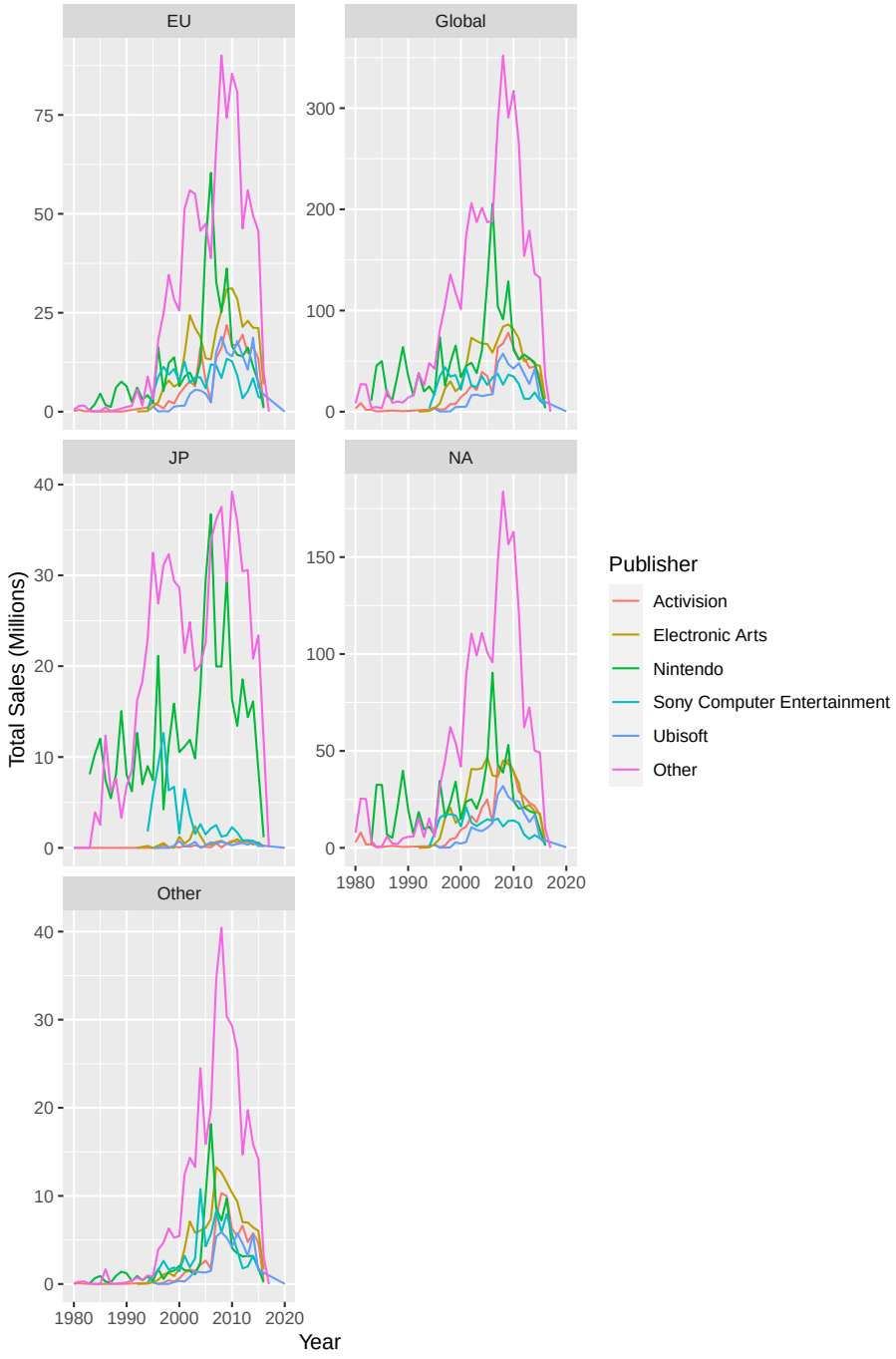
Facets are not features of layers, but instead are universal to the plot. So we couldn't facet our coloured lines by Publisher but then not facet our regression line.

13.1.1.5 Scales

Scales are the domains of our mappings. Each variable that we map via `aes` will have a scale associated with it. For example, when we map our Year variable to the x axis, `{ggplot2}` creates an automatic numeric scale, with values between 1980 and 2020 (the minimum and maximum values in our range). Similarly, when we map the Publisher to the colour aesthetic, `{ggplot2}` automatically creates a colour scale, mapping colours to specific Publishers.

`{ggplot2}` will create default scales for the variables in our mapping, but we can alter them manually. To do so, we just use the accompanying `scale_{aesthetic}_{type}` function. So for our x axis, we're scaling our x mapping and we want a continuous, numeric scale so we would use the `scale_x_continous()` function. Similarly, for our y axis we'd use `scale_y_continuous()`:

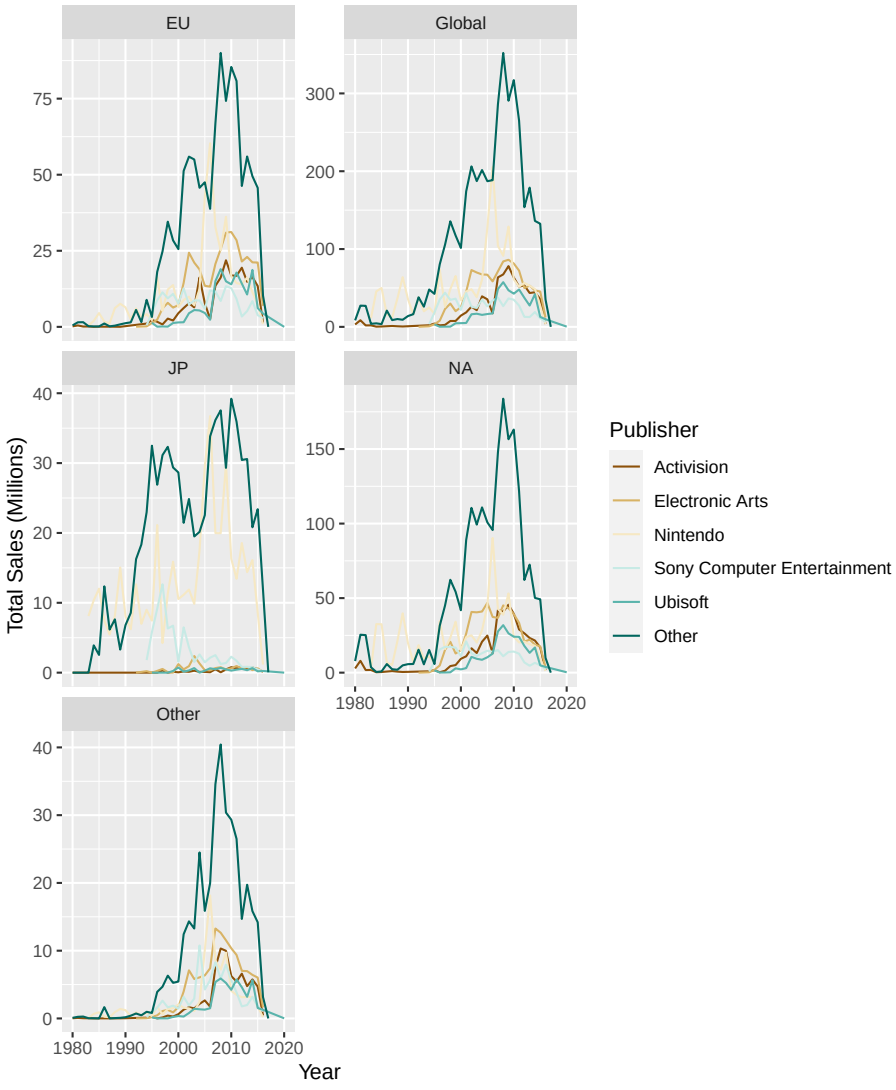
```
ggplot(smmrsd_vg_sales, mapping = aes(x = Year, y = Total_Sales)) +
  geom_line(mapping = aes(colour = Publisher)) +
  scale_x_continuous(name = "Year") +
  # Let's change the limits to stop at 0
  scale_y_continuous(name = "Total Sales (Millions)", limits = c(0, NA)) +
  facet_wrap(~Country, scales = "free_y", ncol = 2)
```



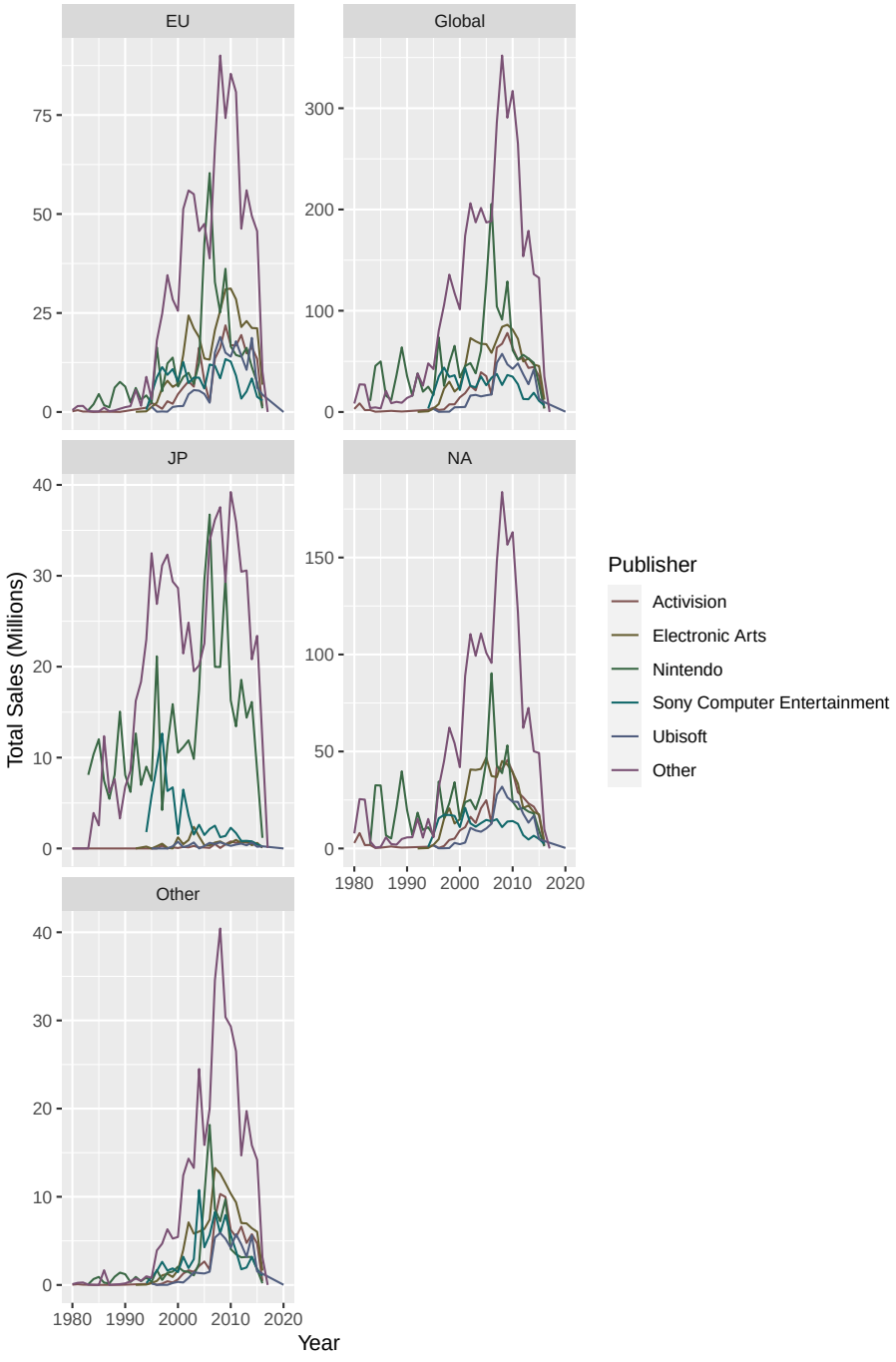
To change the colour scale, we would use the `scale_colour_brewer/hue()` function. These two functions take a slightly different approach in how they

assign colours to the levels of the variable.

```
ggplot(smmrsd_vg_sales, mapping = aes(x = Year, y = Total_Sales)) +
  geom_line(mapping = aes(colour = Publisher)) +
  scale_x_continuous(name = "Year") +
  scale_y_continuous(name = "Total Sales (Millions)", limits = c(0, NA)) +
  scale_colour_brewer(name = "Publisher", palette = "BrBG") +
  facet_wrap(~Country, scales = "free_y", ncol = 2)
```



```
ggplot(smmrsd_vg_sales, mapping = aes(x = Year, y = Total_Sales)) +
  geom_line(mapping = aes(colour = Publisher)) +
  scale_x_continuous(name = "Year") +
  scale_y_continuous(name = "Total Sales (Millions)", limits = c(0, NA)) +
  scale_colour_hue(name = "Publisher", l = 40, c = 30) +
  facet_wrap(~Country, scales = "free_y", ncol = 2)
```

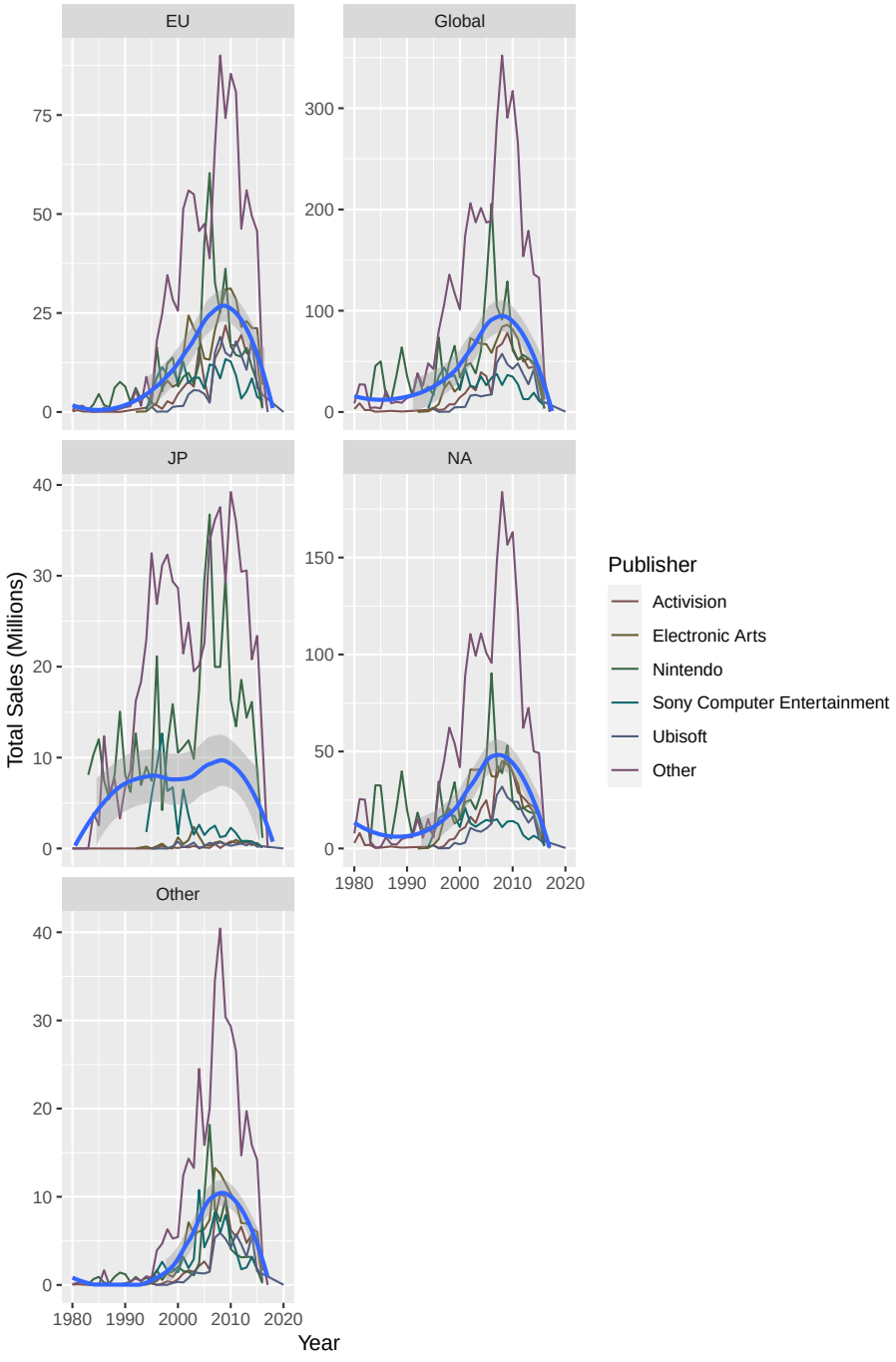

EU
Global
JP
NA
Other
Total Sales (Millions)
Year
Publisher
Activision
Electronic Arts
Nintendo
Sony Computer Entertainment
Ubisoft
Other

13.1.2 Layers

Together, the data, mapping, stat and geom components form a **layer**. A plot can be made up of multiple layers. For example, let's show the same data but overlay a regression line that uses the smooth stat and a different mapping:

```
ggplot(smmrsd_vg_sales, mapping = aes(x = Year, y = Total_Sales)) +
  geom_line(mapping = aes(colour = Publisher)) +
  geom_smooth(method = "loess", formula = y ~ x) +
  scale_x_continuous(name = "Year") +
  scale_y_continuous(name = "Total Sales (Millions)", limits = c(0, NA)) +
  scale_colour_hue(name = "Publisher", l = 40, c = 30) +
  facet_wrap(~Country, scales = "free_y", ncol = 2)
```

```
## Warning: Removed 42 rows containing missing values (geom_smooth).
```



We've now got two layers to this plot, with slightly different mappings and stats. The coloured lines are split by Publisher, but the regression line does not use

that colour mapping, meaning that we have a single line for all the Publishers. The coloured lines also use the 'identity' stat transformation (essentially no stat), where the regression line uses the 'smooth' stat transformation - it calculates a smooth conditional mean of y given x (`y ~ x`).

In summary, a layer is made up of:

- A dataset and aesthetic mapping (aes)
- A statistical transformation (stat)
- A geometric object (geom)

And plots can be made of up one or more layers.

Facets and scales are not components of layers - they are universal to the plot. You couldn't have one layer that used a completely different scale (e.g. one showing numbers on the x axis and the other showing groups on the same axis) because you wouldn't be able to plot them on the same graph!

Chapter 14

Modelling

This chapter is inspired by the simple and concise explanation presented in Hadley Wickham and Garrett Grolemund's R for Data Science book[1].

A model helps us quantify the relationships between variables in a dataset. By understanding these relationships, we can use these models to better understand the data we have, and to make predictions for unseen data.

A relationship can be quantified by two influences:

- A true pattern
- Random error

The goal of the model is to identify the true patterns from the random error and noise. As an example, let's say you're testing the impact of a new drug on cancer treatment. You have half of your participants take the drug, and the other half take placebo. Then, you want to model the impact of your treatment to see if it actually works. In your data though, you're going to see changes between the groups that are due to your intervention (your drug), but also some changes that are just due to random influences, like individual differences, measurement errors, whatever it may be. The goal of a model is to strip away these random influences and quantify in its purest from, the relationship between your variables of interest.

In reality, you will never perfectly capture a relationship for two reasons:

1. We can never truly strip away the noise.
2. Perfect relationships don't really exist.

But that's not really the point. We will never get things spot on, but models can help us with useful approximations that can inform our decision making and further analysis.

[1] https://r4ds.had.co.nz/model-basics.html#model-basics

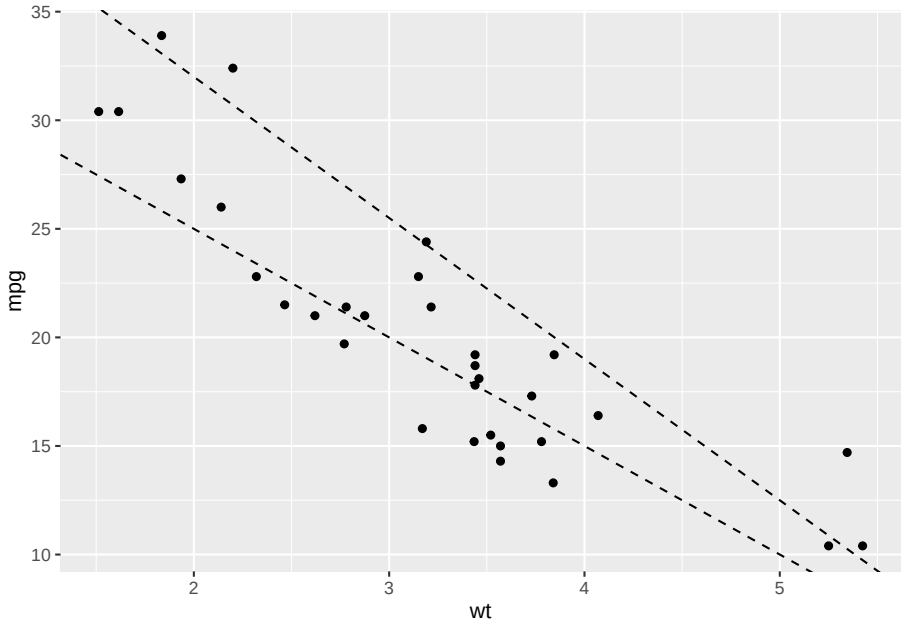
14.1 Types of model

Models types usually fall into "families". These will be a set of models that are underpinned by some common statistics or philosophy. The most common type of model that is taught at beginner level is the linear model. A linear model assumes a relationship that can be represented in the form `y = a + a1x...`, where `a` and `a{n}` represent coefficients (values that will change between models depending on your data). Somewhat confusingly, linear models also extend to what we might call quadratic or exponential models that are represented as `y = a + a1*x + a2*x^2...`. This is because the 'linear' in linear model refers to the coefficients being linear, not our regression line being linear.

14.2 Coefficients

In linear modelling, our coefficients are the values that change in our formula to describe our data. In our standard linear model formula (`y = a + a1*x`), these two coefficients represent the intercept (`a`) and the slope (`a1`). In other words, the `a` represents the value of `y` when `x` is 0 (so where the line would start on the y axis), and `a1` defines how quickly that line goes up or down (is gradient). There will be values for those two coefficients that will produce a better model that others. For example, let's create two models to predict the miles per gallon a car will get based on its weight from the `mtcars` dataset and compare them.

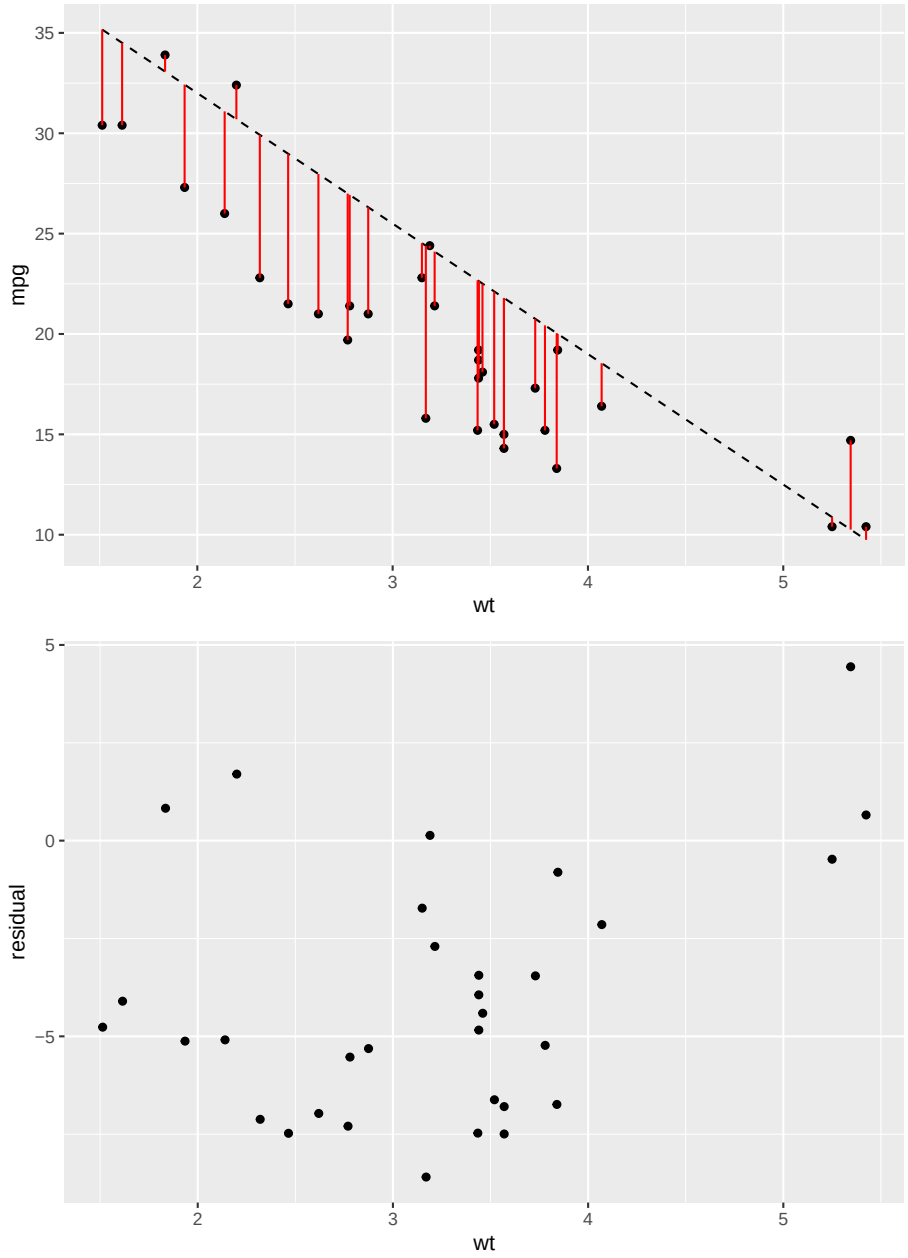
```
mtcars %>%
  ggplot(aes(x = wt, y = mpg)) +
  geom_point() +
  geom_abline(intercept = 45, slope = -6.5, linetype = "dashed") +
  geom_abline(intercept = 35, slope = -5, linetype = "dashed")
```



Here we've created two models with different coefficient values; one where `a` is 45 and `a1` is -6.5, and other where `a` is 35 and `a1` is -5. Both seem to represent the data pretty well, but how can we know which of the coefficients is the 'better' fit? Similarly, we've tried just two sets of values here, how can we compare different combinations of values to determine which model is better?

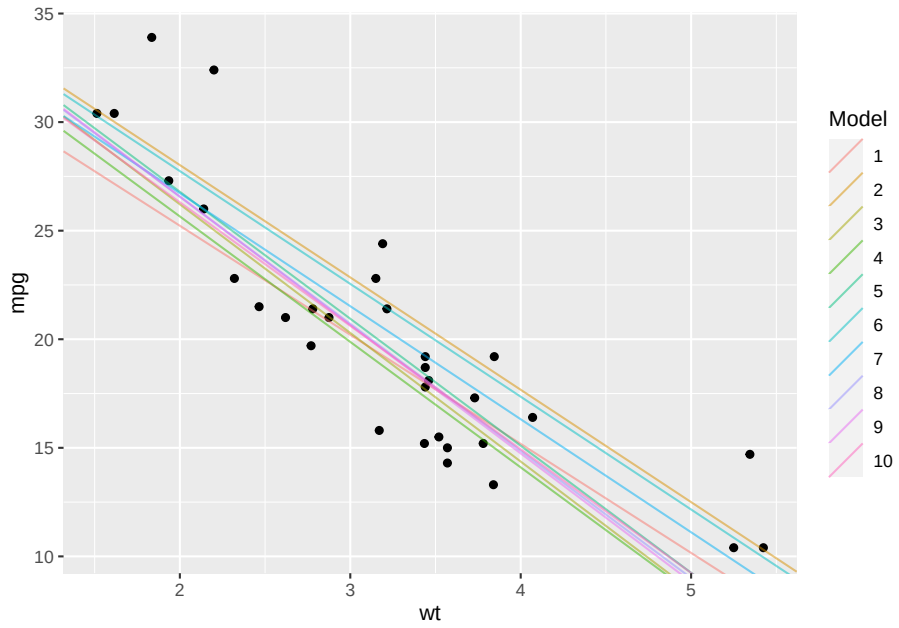
14.3 Residuals

When we create a model, there will be a difference between the model's predicted value and each data point. These are the **model residuals**. The smaller the total residual difference, the more accurate the model is. Let's visualise how we would calculate our residuals:



To compare different models though, we need to boil down this residuals into a single metric. For this we use the root-mean-square-error or RMSE. The calculation for this metric is quite easy - calculate the difference, square it, average it and then take the root. Let's write some functions to produce some models and get some predictions, and then compare the RMSE:

```
# Create some models with different coefficients
coefficients <- data.frame(
  id = 1:10,
  a = runif(10, min = 35, max = 40),
  a1 = runif(10, min = -6, max = -5)
)


# Plot the models
ggplot(mtcars, aes(x = wt, y = mpg)) +
  geom_point() +
  geom_abline(data = coefficients, aes(intercept = a, slope = a1, colour = as.factor(id)), alpha
  scale_colour_discrete(name = "Model")
```




```
# Create a function to represent our model
model <- function(a, a1, wt) {
  a + (wt * a1)
}

# A function to work out the rmse of our model
rmse <- function(a, a1, wt, mpg) {
  diff <- model(a, a1, wt) - mpg
  sqrt(mean(diff ^ 2))
}
```

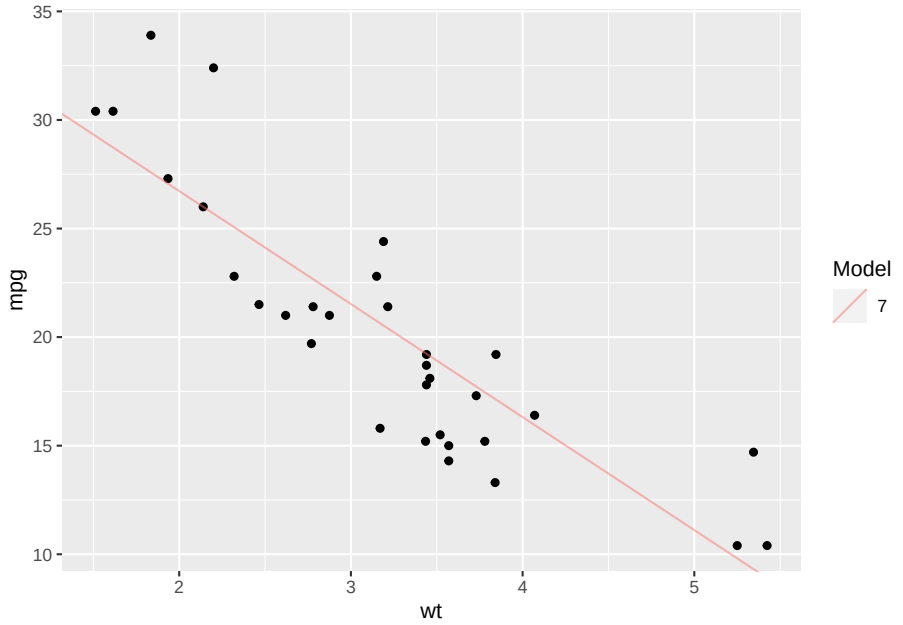
Now we now how to calculate our RMSE, let's apply calculate which of our models has the lowest error:

```
best_model <- coefficients %>%
  # Here we calculate the rmse of each model
  dplyr::mutate(rmse = unlist(purrr::map2(a, a1, rmse, wt = mtcars$wt, mpg = mtcars$mpg
  dplyr::filter(rmse == min(rmse))
print(best_model)
```

```
##   id        a        a1     rmse
## 1  7 37.13843 -5.205525 2.967434
```

According to our analysis, model 7 has the lowest residual error. Let's see what that model looks like:

```
ggplot(mtcars, aes(x = wt, y = mpg)) +
  geom_point() +
  geom_abline(data = best_model, aes(intercept = a, slope = a1, colour = as.factor(id)
  scale_colour_discrete(name = "Model")
```



That looks pretty good, but it's not perfect. We've just tried a number of different values and then chosen the best one. This doesn't mean that we've found the absolute best coefficients for our model. In reality, we'd need to work out the best values by **optimising** our RMSE function - finding the values that produce the global minimum value from that function.

We're not going to bother doing that manually, but that's essentially the process that the `lm()` function in R uses; it automatically finds the best coefficient values by calculating the RMSE for lots of different values and then finding the best one.

14.4 lm() function

Now that we've understood the basics behind a linear model, I can finally reveal that R has a function that will do all of this for us. The `lm()` function in R just needs the dataset and the formula for our model, and works out the coefficients for us, giving us back a model:

```
lm_model <- lm(data = mtcars, mpg ~ wt)
lm_model
```

```
##
## Call:
## lm(formula = mpg ~ wt, data = mtcars)
##
## Coefficients:
## (Intercept)           wt
##      37.285       -5.344
```

We can see that the `lm()` function has calculated coefficient values of 37.285 for the intercept, and -5.344 for the slope. We weren't too far off!

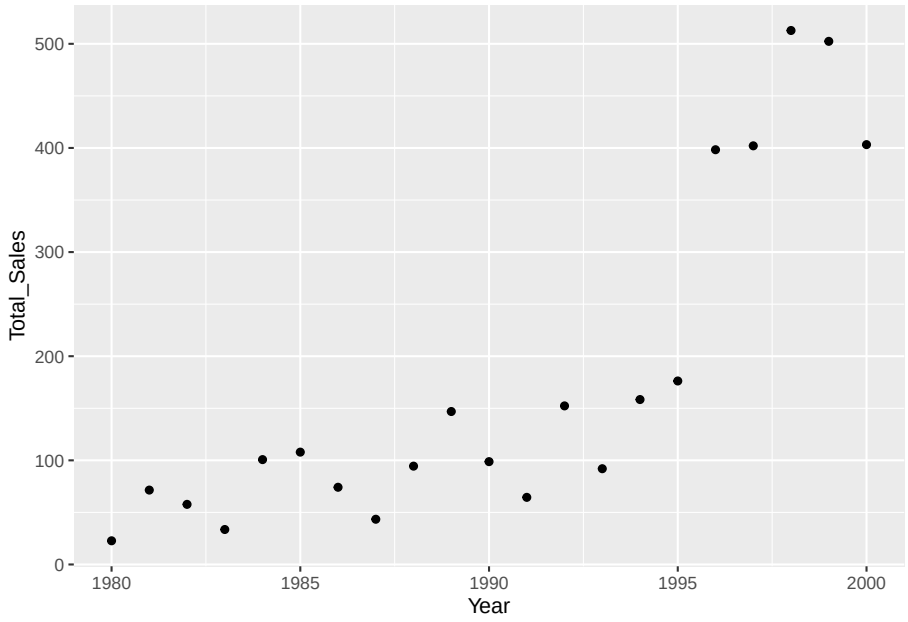
14.5 Example

Now we've created our first model, let's apply what we've learned to our video games dataset. First off, let's model the effect of year on video game sales between 1980 and 2000. Let's start with a plot:

```
vg_sales_to_model <- smmrsd_vg_sales %>%
  dplyr::filter(Year >= 1980 & Year <= 2000) %>%
  dplyr::group_by(Year) %>%
  dplyr::summarise(Total_Sales =  sum(Total_Sales))
```

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

```
vg_sales_to_model %>%
  ggplot(aes(x = Year, y = Total_Sales)) +
  geom_point()
```



There does seem to be a relationship, but it doesn't seem to be quite as linear as our `mtcars` example. Instead, we seem to increase more slowly in the earlier years, before increasing dramatically toward 2000.

To model this relationship then, let's use a quadratic equation. This allows for us to model non-linear relationships between variables.

Quadratic equations are written in the form `y = a + a1*x + a2*x^2`, where `a`, `a1` and `a2` are all coefficients. Although we've introduced a new coefficient here compared to our previous examples, the logic behind the modelling is exactly the same.

Let's create this model using the `lm()` function:

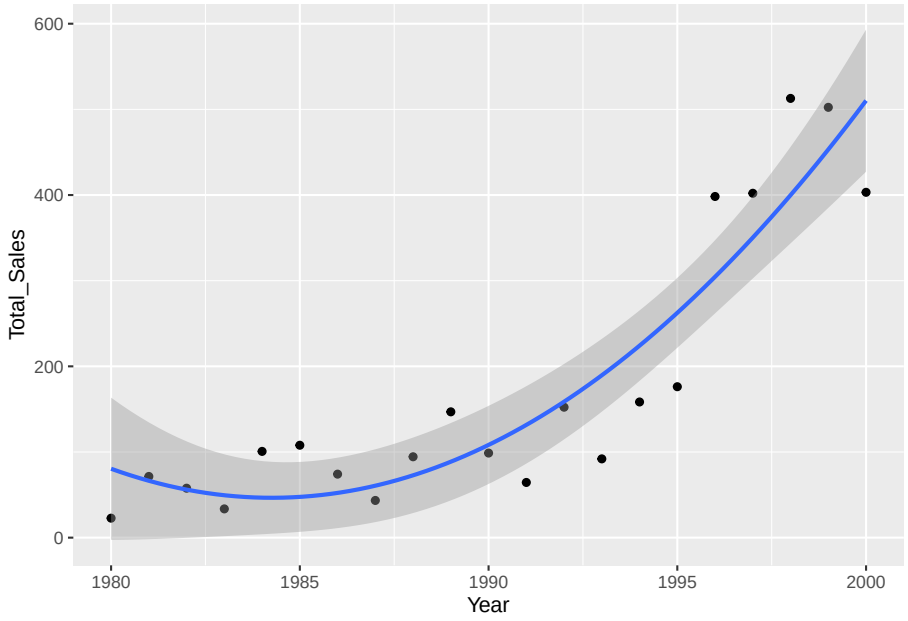
```
vg_model <- lm(data = vg_sales_to_model, formula = Total_Sales ~ Year + I(Year^2))
vg_model
```

```
##
## Call:
## lm(formula = Total_Sales ~ Year + I(Year^2), data = vg_sales_to_model)
##
## Coefficients:
```

```
## (Intercept)         Year    I(Year^2)
##   7.366e+06   -7.424e+03    1.871e+00
```

Let's plot this model using {ggplot2}:

```
vg_sales_to_model %>%
  ggplot(aes(x = Year, y = Total_Sales)) +
  geom_point() +
  # This will produce the same model as we did with the `lm()` model
  geom_smooth(formula = y ~ x + I(x^2), method = "lm")
```



That doesn't look too bad at all.

14.6 Evaluating Models

Jumping straight into modelling a dataset you don't understand or creating a model that looks good and then declaring that your work is done can be a dangerous thing. Modelling should be a weapon in your data science arsenal, but it should be handled with care. It's very easy to make an error and end up with a model that is pointless at best and misleading at worst.

To demonstrate this, let's produce 4 models for the each of the sub-datasets in the `anscombe` dataset:

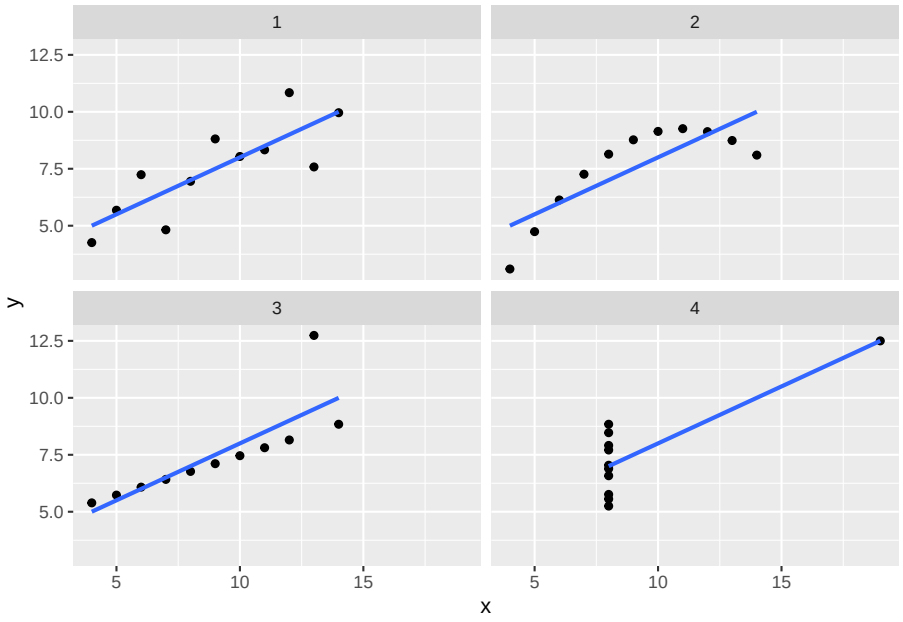
```
tidy_anscombe <- anscombe %>%
  tidyr::pivot_longer(everything(),
                      names_to = c(".value", "dataset"),
                      names_pattern = "(.)(.)"
  )

models <- purrr::map(1:4, ~tidy_anscombe %>% dplyr::filter(dataset == .x) %>% lm(data =
models
```

```
## [[1]]
##
## Call:
## lm(formula = y ~ x, data = .)
##
## Coefficients:
## (Intercept)            x
##      3.0001       0.5001
##
##
## [[2]]
##
## Call:
## lm(formula = y ~ x, data = .)
##
## Coefficients:
## (Intercept)            x
##       3.001        0.500
##
##
## [[3]]
##
## Call:
## lm(formula = y ~ x, data = .)
##
## Coefficients:
## (Intercept)            x
##      3.0025       0.4997
##
##
## [[4]]
##
## Call:
## lm(formula = y ~ x, data = .)
##
## Coefficients:
```

```
## (Intercept)           x
##      3.0017      0.4999
```

According to our output, we've got 4 very similar models. From that output we might assume that the datasets must be very similar. Let's plot them now:

```
tidy_anscombe %>%
  ggplot(aes(x = x, y  = y)) +
  geom_point() +
  geom_smooth(method = "lm", formula = y ~ x, se = FALSE) +
  facet_wrap(~dataset)
```



Now that we've looked a little deeper, we can clearly see that these models are inappropriate for 3 of the datasets (2, 3 and 4). The lesson here is that there are multiple tools at your disposal when analysing data, and overeliance on a particular tool without due care and attention can cause issues.

With that in mind, let's look at some ways of evaluating the strength of your model.

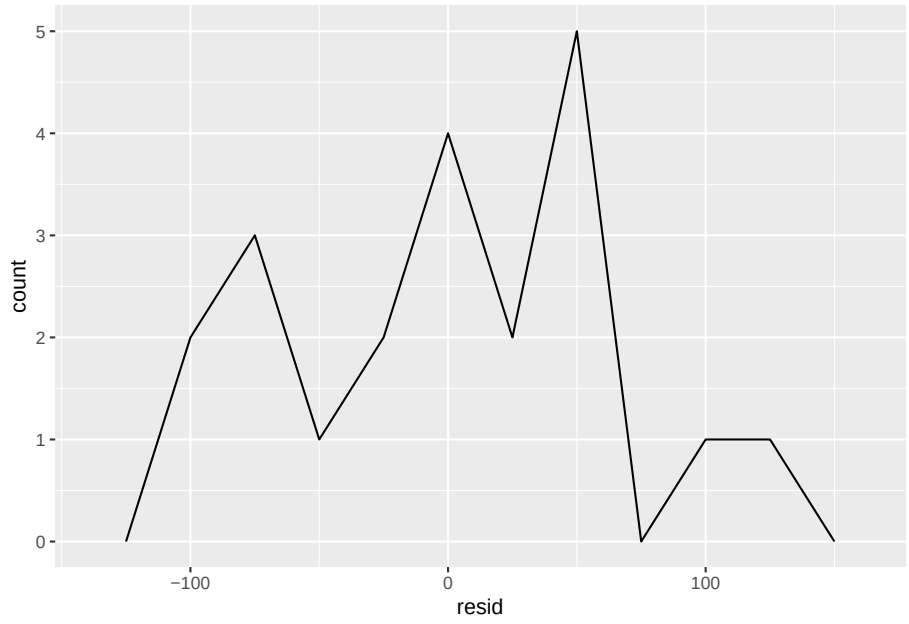
14.6.1 Residuals

As we've learnt previously, the goal of our model is to reduce the total residual difference between our model and the data. But blindly reducing this value

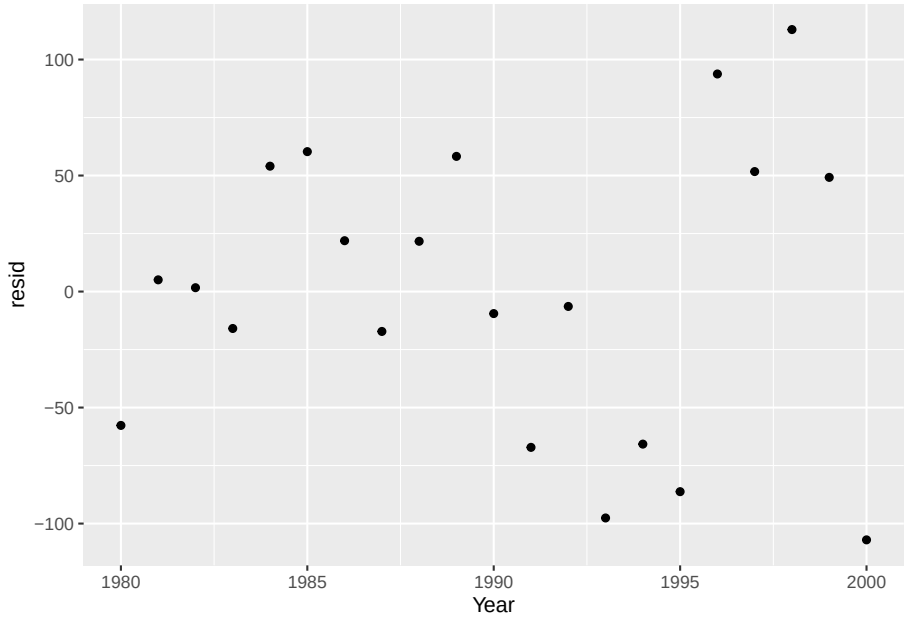
without then inspecting the result can be dangerous. When you're using the `lm()` function, your residuals should be **normally distributed**. That means that (essentially) your model should be overestmating as much as it's underestimating for every value of y.

Let's take a look at the residual values for our video game sales model:

```
vg_residuals <- vg_sales_to_model %>%
  modelr::add_residuals(model = vg_model)

vg_residuals %>%
  ggplot(aes(x = resid)) +
  geom_freqpoly(binwidth = 25)
```



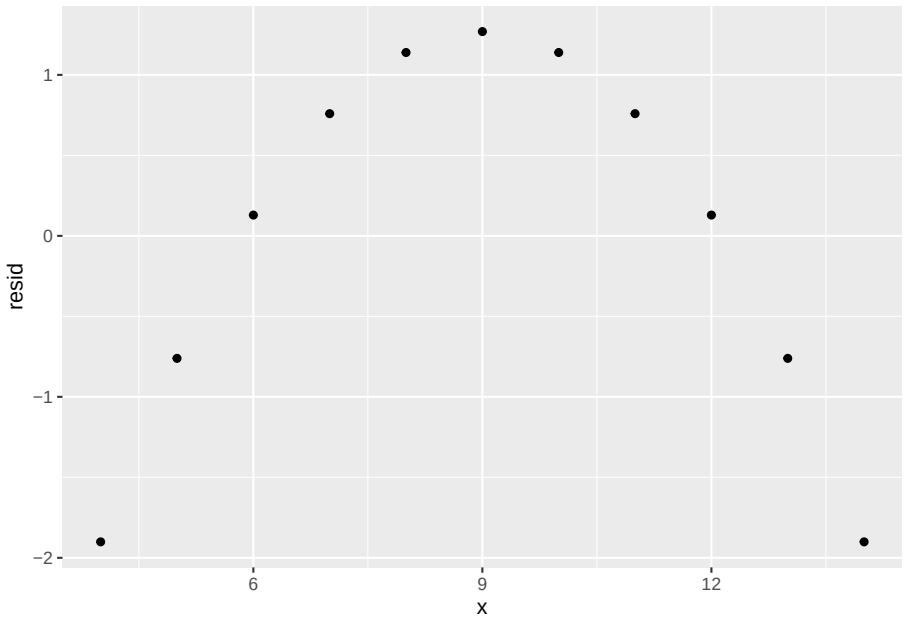

```
vg_residuals %>%
  ggplot(aes(x = Year, y = resid)) +
  geom_point()
```



Looking just at the frequency polygon doesn't really tell us enough but when we then look at the residuals by Year, we can see what looks like random noise. This is good because it means that our model isn't better at predicted one year or range of years better than any other.

Alternatively, let's compare this plot with a plot of one of the `anscombe` datasets:

```
tidy_anscombe %>%
  dplyr::filter(dataset == 2) %>%
  modelr::add_residuals(model = models[[2]]) %>%
  ggplot(aes(x = x, y = resid)) +
  geom_point()
```



Here we can see a very clear trend, meaning that our model is not representing the data very well. Instead, it is overestimating and underestimating in a predictable pattern for different values of x.

- The goal of a model
- Quantify relationships and make predictions
- A model is always wrong, but it can be helpful
- How a model is produced
- Minimising residual error
- Generalised linear modeling
- `lm()`
- Neural networks
-

Part III

Projects

Chapter 15

Introduction

Now we've learnt some basic data science skills, we're going to look at the best way to plan and structure your project. Not all of your analyses will be of sufficient size to warrant a big planning stage, but learning to use a common, separate structure for all of your different projects can really help keep your work clean. This becomes even more important when you begin to combine multiple projects and you want to make sure that they don't slowly start to creep into one. For example, imagine you've previously worked on a project that relied heavily on API data. Then, in your next project, you need to use much of the same data but to a very different end. By utilising this project structure (and more specifically, the idea of "Projects as packages"), you'll be able to easily utilise work from previous projects without duplicating or merging code.

We're going to look at things more conceptually for the next few chapters, but then we're then going to apply all of these concepts to create a full data science project. We'll then use everything we've learned in the Data Analysis chapters along with what we've learned about structuring our project to create an end-to-end example. The project is available to view and download on GitHub[1].

[1] https://github.com/ARawles/operate.package.example

Chapter 16

Workflows

Often the best way to start a data analysis project is to decide what you're workflow should be, breaking your project into distinct stages that are relevant for your particular project.

For instance, if you know that the data you're going to be working on is likely to be littered with mistakes and errors, then you should preemptively allocate a decent amount of your time to the "importing" and "cleaning" steps of your workflow. Similarly, if your end goal is to produce a predictive model at the end, then work in a feedback loop where you inspect and evaluate your model before improving it in the next iteration.

The tidyverse packages we discussed previously are designed to make data science easier, and the workflows we'll look at fit into this philosophy quite nicely. Each stage of these workflows is usually handled by a different package but with a common syntax and data structure underpinning them all.

16.1 Basic Workflow

For now, let's look at a basic workflow and some likely additions or changes you might make depending on your goals.

For these basic workflows, we're only going to look at a subset of all of the stages of analysis that you might identify. They are going to be:

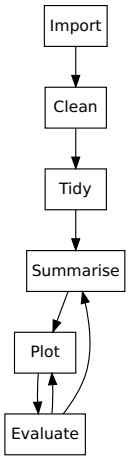
- Importing
- Cleaning
- Tidying
- Collating
- Summarising

- Plotting
- Modelling
- Evaluation

16.1.1 Example 1: Reporting

In this example, imagine someone has come to you and they want a bit more insight into the data they have. It's currently in a messy spreadsheet, and they want you to load it into R and produce some nice looking graphics to help them understand trends and correlations in their data. They're not bothered about any modelling for now, they just want some pretty graphs.

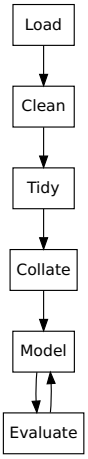
If we mapped out an appropriate workflow for this project, it might look something like this:



We import the data and clean it and tidy it (more on that later), then we summarise it and create our plots. We can then use feedback from our colleague to change how we're summarising or how we're plotting the data to improve our plots until we're happy.

16.1.2 Example 2: Modelling

For this example, imagine a colleague has come to you about a modelling project. They have a number of relatively clean datasets that they want to combine and use to produce a predictive model. They're not too worried about plots or graphics, they're more interested in the model itself. If we mapped out a workflow for this project, it might look a bit more like this:



As you can see, we still end with the loop of modelling and then evaluating our model and using the output to update the model, but the steps we take to get there are a bit different.

16.2 Separating Stages

16.2.1 Functions

16.2.2 Files

16.3 Summary

Hopefully this section has given you an idea of what a typical project workflow could look like. The benefit of splitting your project into these distinct steps is that it can often help you compartmentalise your functions and scripts into distinct stages. We'll look later at structuring your project like an R package, and so splitting your analysis into separate stages can help you construct your project in a portable and replicable way.

Chapter 17

Replicability

The key to a good, robust analysis is replicability. When we say 'replicability', we mean two things:

- That your results can be replicated in other populations and in other settings (i.e. typical 'scientific' replicability)
- That you can easily share your code and method with others who can verify your outputs

If your project is replicable, then it's likely to have fewer issues, make fewer dodgy assumptions, and rely less on the idiosyncracies of your environment or coding practice. That doesn't mean that everything that you do that can be replicated is immediately correct, but it's a useful credential to have.

For now, we're going to focus on how you can make your *code* more replicable. By that I mean, how you can share your results and your code with others either in your business or instution or even in the open source community in general.

17.1 Good practice

There are a few things you can do to make your work immediately more readable for others:

17.1.1 Use a consistent naming convention

Take your pick. Everyone has their preference and that's okay. If you like camelCase then go with camelCase. If you like the _ approach, then go for that. The most important part of your naming convention however is that it's stable.

Don't name some of your functions `like_this` and the rest `likeThis`. Not only does it make it harder to read, but every time you do it, a kitten dies. So be consistent.

17.1.2 Name your variables as nouns and your functions as verbs

Functions do and variables and objects are. Almost all languages share this distinction with verbs and nouns, so utilise that natural divide to improve how you name your functions and variables. Aim to give your functions names that suggest that they `do` something; and bonus points if you can give it a name that's a verb and also gives a decent description of what the function does. When you name your variables, give them meaningful noun-like names. Say you're doing some climate analysis, a good name for the variable that holds the average rainfall in a year might be `avg_yearly_rainfall` whilst a function that converts a temperature in Celsius to Fahrenheit might be `convert_degrees()`.

17.1.3 Use functions where you can

R is a functional language and so using and creating functions is at the very heart of programming in R. Rather than relying on R scripts that need to be run in their entirety to produce your output, by creating functions and using them in your analysis, you can more easily scale your project. For example, imagine your scope is initially to produce a report for the year. Then, after your done, your manager is so impressed with your report, they want you to do the same thing for the last 10 years. If you've used a couple of R scripts and you haven't written any functions, then this is likely going to involve copying and pasting a whole lot of code and changing the years. Instead, if you use functions to perform your analysis, then creating reports for multiple years can be as simple as changing your dataset.

As a rough rule of thumb, if you find yourself copying and pasting your R code more than twice to do very similar things, you should probably be using a function.

17.1.4 State your dependencies

One of the great things about R is the number of packages that are available that let you do all sorts of weird and wonderful things. Unfortunately, because there are so many great packages, it's unlikely that the person you're sharing your code with will have them all installed. If you're sending someone a script, then best practice is to include all the packages you use at the top of your script like this:

```
library(ggplot2)
library(dplyr)
```

An extra step which few people do but can be very helpful is to still prepend your functions with which package they came from like this `dplyr::mutate()`. Not only does this avoid any namespace conflicts where two packages might have functions with the same name and you don't end up using the one you think you are because of the order of your `library()` calls, but it also makes it infinitely easier for anyone reading your code to find the package that a function comes from. Admittedly, this is overkill in a sense because we've already told R which packages we're using with our `library()` calls and R has loaded in the objects from that package, but this practice can really improve the readability of your code.

Later we'll look at designing our project as a package and packages have a different way of stating dependencies for the user, so this is primarily for the case where you're just sending a script or two to someone.

17.2 Documentation

When you're working on a project, never think that you'll remember everything when you come back to it. You won't. Instead, imagine you're always writing for code for someone who's never seen it before, because, trust me, when you come back to a project after 6 months you'll have absolutely no idea what you're looking at.

At the heart of writing code that's easy to understand is documentation. Here we'll talk about the two main types of documentation.

17.2.1 Function documentation

To get a package on CRAN, all the functions that your package exports needs to be documented. So if you type, say, `?dplyr::mutate()` into the console and hit Enter, you'll be taken to the Help page for the `mutate` function. This ensures that the package users are not left guessing what a parameter is supposed to be, or what they're going to get returned from the function.

Even if you're not ever planning on submitting your work to CRAN, function documentation is extremely useful. The `roxygen` package makes this documentation as easy as possible for package developers, and you can use the same principles in your analysis.

At the very least, you should be documentating the input variables (the `@param` tag in `roxygen`), your return value(s) (the `@return` tag) and maybe an example

or two (the `@examples` tag). This will make explaining your code to someone else or relearning it yourself infinitely easier.

17.2.2 Long-form documentation

Whilst understanding what each of your functions does is important, it's also important to document how all of the little pieces fit together. To acheive this, it's a good idea to write some long-form documentation, like a README or a vignette.

Long form-documentation is often written in something called RMarkdown. RMarkdown is a spin on markdown[1] that allows you to embed and run R code when generating a document. This can be really useful for explaining your process and sharing your workings.

17.2.2.1 READMEs

READMEs act as an entrypoint for anyone that stumbles across your package. It should be in the root of your project directory, and should be written in markdown or plain text. Anyone should be able to read your README and understand what the project is doing. I won't go into exactly what should be included because it will depend on the type of project your doing, but you know a good README when you see it.

If you also use a repository system like GitHub, your README will act as the homepage and so should always be present.

17.2.2.2 Vignettes

Vignettes are less defined that READMEs. Vignettes should be an in-depth form of documentation for a concept in your project. For example, say that your project is looking at the level of data quality of various open source APIs. You might have a few different vignettes covering different important concepts in your project:

1. Background
    - This would cover why the how the project came to be, and what the ultimate goal is.
2. Selection of APIs
    - Why were certain APIs chosen and others excluded? That would be explained here.

[1] https://en.wikipedia.org/wiki/Markdown

3. Methodology
    - How are you assessing data quality? Here you would explain your different tests and standards.

Whilst a README serves as a general introduction to the project, vignettes more often serve as in-depth descriptions of certain aspects of the project. Like READMEs, vignettes should be written in markdown (or more likely RMarkdown).

Chapter 18

Projects as Packages

With the last two chapters in mind, to make a good, replicable and easy to understand project, you need a few things:

1. A clear structure
2. Easily replicable code (e.g. consistent functions and clear dependencies)
3. Good documentation (functional and long-form)
4. Maybe a test or two if you want to be confident in your functions and findings

While you could easily fulfill all of these criteria with a standard RStudio project, there's already a type of project that conforms to these standards: *Packages*.

While packages are primarily meant for collating and distributing new functionality, we can utilise the package structure and some of the tools that come along with package development in our projects.

To better understand what a package is and how they are developed and maintained, I would recommend reading Hadley and Jenny Bryan's R Packages book[1] book. Although that book is focused directly on developing packages for submission to CRAN, hopefully you'll be able to identify the parts that are going to be applicable to the way we're using packages.

18.1 Structure

18.1.1 R code

The structure of R files in a package is pretty simple. Everything goes in the root of the R folder. That means no subfolders. There are also some filenames

[1] https://r-pkgs.org/

that should be avoided for normal packages, but we won't worry too much about that for now.

When you then run the `devtools::load_all()` command, `{devtools}` checks that you've got the dependencies listed in the DESCRIPTION file loaded and then loads all of the files in the R folder are then loaded. This means you can make quick changes to your code and then run `devtools::load_all()` to bring those changes into your current environment. This helps prevent working on data or functions that are out of date (which often happens when you're manually sourcing R files).

18.1.2 Data

For some projects, you'll want to include a static dataset or datasets that you're basing your analysis on. Packages also have a way of including data in a standard way. Use the `usethis::use_data()` function to bundle an R object with your package. You can make sure that the data needed for your project is included with it.

For other projects, you'll want to use the latest data whenever the analysis is done. For this, I would recommend creating a set of functions to get your data and including those in the package. That will ensure that getting the data needed for your analysis is as simple as possible for people reading your code.

18.2 Documentation

18.2.1 Functional

To document your functions, I would highly recommend the `roxygen` package. It's by far the easiest way to document your functions. Once you've documentated your functions, you can run `devtools::document()` to automatically generate the documentation that will be seen when someone visits the help page for your function (e.g. via `?your_function`).

18.2.2 Long-form

For your long-form documentation, the `usethis::use_readme()` function will provide you with a template for you to build your README.

For vignettes, the `usethis::use_vignette()` function will create the appropriate file and folder for you, so you can just focus on writing.

18.3 DESCRIPTION

Every package you download will have a DESCRIPTION file. This file has a number of fields, like who the author is, what license the content is under, and so on. We can utilize many of the fields to help document our project. For example, the Description and Title field are just as relevant for a package-project. Equally, we can use the Version field to keep track of different iterations of our analysis. We can also use this file to state our dependencies.

18.3.1 Dependencies

Every package will have an entry called Imports in its DESCRIPTION file. Here, the author is stating every other package that's needed for this package to run. So we can use that same field to state all of the packages that are required for our analysis. That way, when someone installs our package to replicate or check our analysis, all the appropriate dependencies can be installed at the same time.

18.4 Abstraction

When I was younger, I wrote a package to create a kind of financial stability report. The report essentially used a number of APIs to pull in macroeconomic data and then create an RMarkdown report displaying the data. Then, a few months later, I started another analysis that used very much the same kind of data but for a different purpose. Now, there were three things I could do:

1. Copy all the code used to get the API data from the original project into the new one
2. Put the original project as a dependency of the new project, importing all of the API data but also all of the other functions
3. Abstract the functions used to get the data from the original project into a new package, and then have both projects use that as dependency.

Given that the title of this section is "Abstraction", what do you think I went with?

Structuring your analysis in this way helps you re-use or repurpose code in the future without having to copy and paste or duplicate any of your previous work.

It can be quite hard to understand the concept of structuring your project as a package without actually giving it a go, so let's go through an example.

Chapter 19

Git & GitHub

- Version control
- Collaboration

Chapter 20

Project Example

- Creating GitHub repo
- Creating project
- Filling out description (including dependencies)
- Importing data
  - Store the file in the 'data-raw' folder, using the `usethis::use_data_raw()` function
  - Write the script to load the csv file and save it with `usethis::use_data()`
- File structure
  - Separate files for each stage (e.g. clean, tidy, summarise, plot)
  - analysis.R file that has the overarching analysis
- Documentation
  - README
  - Function & dataset
    - `{roxygen2}` and markdown
  - Vignettes