Reproducible Road Safety Research with R

Robin Lovelace

Table of contents

Preamble

About the Author

Robin Lovelace is Professor of Transport Data Science at the Leeds Institute for Transport Studies (ITS) specialising in the analysis of regional transport systems and modelling scenarios of change. Robin is Lead Developer of the Propensity to Cycle Tool (see www.pct.bike) and follow-on projects such as the Network Planning Tool for Scotland (NPT), Principal Investigator of the Department for Transport funded SaferActive project and author of popular open source software packages (such as stplanr) and books (such as Geocomputation with R).

Disclaimer

This report has been prepared for the RAC Foundation by Robin Lovelace at the Leeds Institute for Transport Studies. Any errors or omissions are the author's sole responsibility. The report content reflects the views of the author and not necessarily those of the RAC Foundation.

Foreword: Road Safety GB

Road Safety GB recognises the importance of high quality data analysis as a fundamental element in efforts to reduce road casualties at a local and national level by enabling evidence-based intervention. The use of open source software like R in this way is to be actively encouraged and tools such as this manual provide valuable support for analysts working across the industry in enhancing the analysis they are able to provide to decision-makers. This approach also supports the reproduction of high-quality analysis up and down the country using locally-held data, which I hope in turn will improve the consistency and quality of evidence used in day-to-day road safety activity.

I would like to thank those that have worked hard to pull this manual together and encourage all those working with road safety data to make use of this resource to learn, develop and share their analysis methods with others.

Matt Staton, Director of Research, Road Safety GB

Foreword: Department for Transport

The STATS19 data collection for road traffic collisions has existed in the current form since 1979. It is a well-established source of road safety data which offers great insights into the trends and locations of road traffic collisions for central and local government, the police and the general public. The openness and accessibility of this data is important to DfT, who have launched a data download tool to improve the access to road safety data. As well as working to enhance the use of R and Reproducible Analytical Pipelines to improve the quality of their analysis and publications.

The standard use of packages and code creates transparent and consistent framework for analysis. And the work by the University of Leeds takes new and experienced R users through the process of producing temporal and special analysis using STATS19 data. We commend this book to all those who wish to conduct analysis of road traffic collisions and use analysis to help save lives on our roads.

John Wilkins, Deputy Director, Statistics Travel and Safety Division, Department for Transport

Foreword: RAC Foundation

If there is one thing to notice from the changes in the road safety sector over the past decade it is the rapid development of data and data science. Not too long ago road safety analysis involved mounds of paperwork, file sizes too large for transmission, and geo-coding using old A to Zs. We now have systems with handheld devices taking precise GPS co-ordinates at the scene, online-only systems and automated error checking. The volumes of data we possess are growing rapidly, our abilities to maintain and clean data have become more straightforward and what it is possible to discover from data has become cheaper and easier to obtain. Our expectation is, understandably, that across the sector we should be able to access road safety data and to do more with it, more easily and more readily.

The types of analyses that we used to associate with sectors like pharmaceuticals and insurance, with high-end technology and well-funded research programmes, are increasingly within the grasp of people with normal laptops and pay-by-the-hour cloud-computing, and in sectors that don't always have much money to throw at a problem. The good news is that there is scope for road safety analysis to pick up these methods and approaches adopted in other sectors and work hopefully to bring about the new insights we need to improve road safety.

At the RAC Foundation we want road transport to benefit from this new world of data analysis – where the tools available are getting cheaper and what is possible is growing rapidly. But this can only happen if the opportunities are given to the sector's road safety analysts to learn new skills for the job.

While Great Britain has a history of road crash data recording that is world-leading, our analysis of it is all-too-often locked in to a pattern of labour-intensive and repetitious reporting, with analysts lacking the support they need to improve skills and find the space to do the sorts of analyses that will help us achieve the next step-change decline in casualties on our roads. Which is why we commissioned this work from Robin Lovelace.

We hope that Robin's manual will go some way towards meeting the current need: by giving road safety analysts a self-help training manual to develop their skills in R,, the open-source analytical tool. This manual covers everything one would need for doing the regular tasks of road safety analysis entirely in the R language, designed to be accessible to the newcomer. R allows you to code analysis for reproducible research; reproducible in the sense that others can check and verify it as well as borrow, share and adapt it to their own work. Analysts can also repeat their own work as fresh data becomes available – there's no need to recreate the wheel. The openness, efficiency and power of working in R offers the opportunity, if taken, to improve how road safety analysis gets done. Let's be honest: like learning any new skill, effort is needed upfront to reap the benefits of this new way of working. But we firmly believe the effort it is worth it, and we think this manual is great way to get you started.

Steve Gooding, Director, RAC Foundation

Preface

Many areas of research have real world implications, but few have the ability to *save lives* in the way that road safety research does. Road safety research is a data driven field, underpinned by attribute-rich spatio-temporal event-based datasets representing the grim reality of people who are tragically hurt or killed on the roads. Because of the incessant nature of road casualties, there is a danger that it becomes normalised, an implicitly accepted cost associated with the benefits of personal mobility.

Data analysis in general and 'data science' in particular has great potential to support more evidence-based road safety policies. Data science can be defined as a particular type of data analysis process, in that it is script-based, reproducible and scalable. As such, it has the ability to represent what we know about road casualties in new ways, demonstrate the life-saving impacts of effective policies, and prioritise interventions that are most likely to work.

This manual was not designed to be a static textbook that is read once and accumulates dust. It is meant to be a hand-book, taken out into the field of applied research and referred to frequently in the course of an analysis project. As such, it is applied and exercise based.

There are strong links between data science, open data, open source software and more collaborative ways of working. As such, this book is itself a collaborative and open source project that is designed to be a living document. We encourage any comments, questions or contributions related to its contents, the source code of which can be found at the Reproducible Road Safety Research with R (rrsrr) repo on the ITSLeeds GitHub organisation, via the issue

tracker. More broadly, we hope you enjoy the contents of the book and find the process of converting data science into data driven policy changes and investment rewarding. Get ready for the brave new reproducible world and enjoy the ride!

Robin Lovelace, Leeds, Autumn 2020

Thanks

Many thanks to everyone who made this happen, especially RAC Foundation for funding the project, Malcolm Morgan and Andrea Gilardi for contributing to earlier versions, and the Department for Transport for funding reproducible road safety research through the SaferActive project.

1 Introduction

This book teaches reproducible road safety analysis with R, a popular, free and open source statistical programming language. It was initially developed for a 2 day course, Introduction to R for Road Safety course. Since then, interest in the topic has grown. The RAC Foundation charity in the UK funded the development of this manual as a free and open resource to support their objective of making the roads safer for everyone. The content is based on open access road crash data from the UK, which is provided by the R package **stats19**. However, the content is designed to be general and should be of use to **anyone working with road crash data worldwide,** that has (at a minimum) the following variables (see Table 1.1 for an example crash dataset):

- a timestamp;
- a location (or address that can be geocoded); and
- attribute data, such as severity of crash, and type of vehicles involved.

```
a = stats19::get_stats19(2020)
```

```
Files identified: dft-road-casualty-statistics-collision-2020.csv

   https://data.dft.gov.uk/road-accidents-safety-data/dft-road-casualty-statistics-collision-

Data saved at /tmp/RtmphlCgnU/dft-road-casualty-statistics-collision-2020.csv

Reading in:

/tmp/RtmphlCgnU/dft-road-casualty-statistics-collision-2020.csv

date and time columns present, creating formatted datetime column
```

```
v = c("date", "longitude", "latitude", "accident_severity")
a_sample = a[3:5, v]
knitr::kable(a_sample)
```

Table 1.1: Sample crash dataset. These are real records taken from the open STATS19 database, provided by the UK's Department for Transport, under the terms of the Open Government Licence (OGL).

date	longitude	latitude	accident_severity
2020-01-01	NA	NA	Slight
2020-01-01	NA	NA	Serious
2020-01-01	NA	NA	Slight

Clearly, work is needed to go from the raw data to evidence that can save lives. Although the datasets used in this guide report road casualty data from Britain, the approach knows no borders: R works equally well in China, India, the USA and Zambia. Road safety is a global issue, that can be considered an epidemic and "the leading cause of death for people aged between 5 and 29 years" worldwide, ahead of hunger, disease and war. The urgency and ubiquity of the 'road violence' epidemic is shown in Figure 1.1. Although Britain has relatively safe roads by international standards (with around 3 road traffic deaths per 100,000 people per year, compared with a global average of 17), it still sees over 1000 road deaths each year and unmeasurable costs to families who have lost loved ones and people left with permanent injuries due to poorly designed roads and transport policies.

```
knitr::include_graphics("figures/road-casualties.png")
```

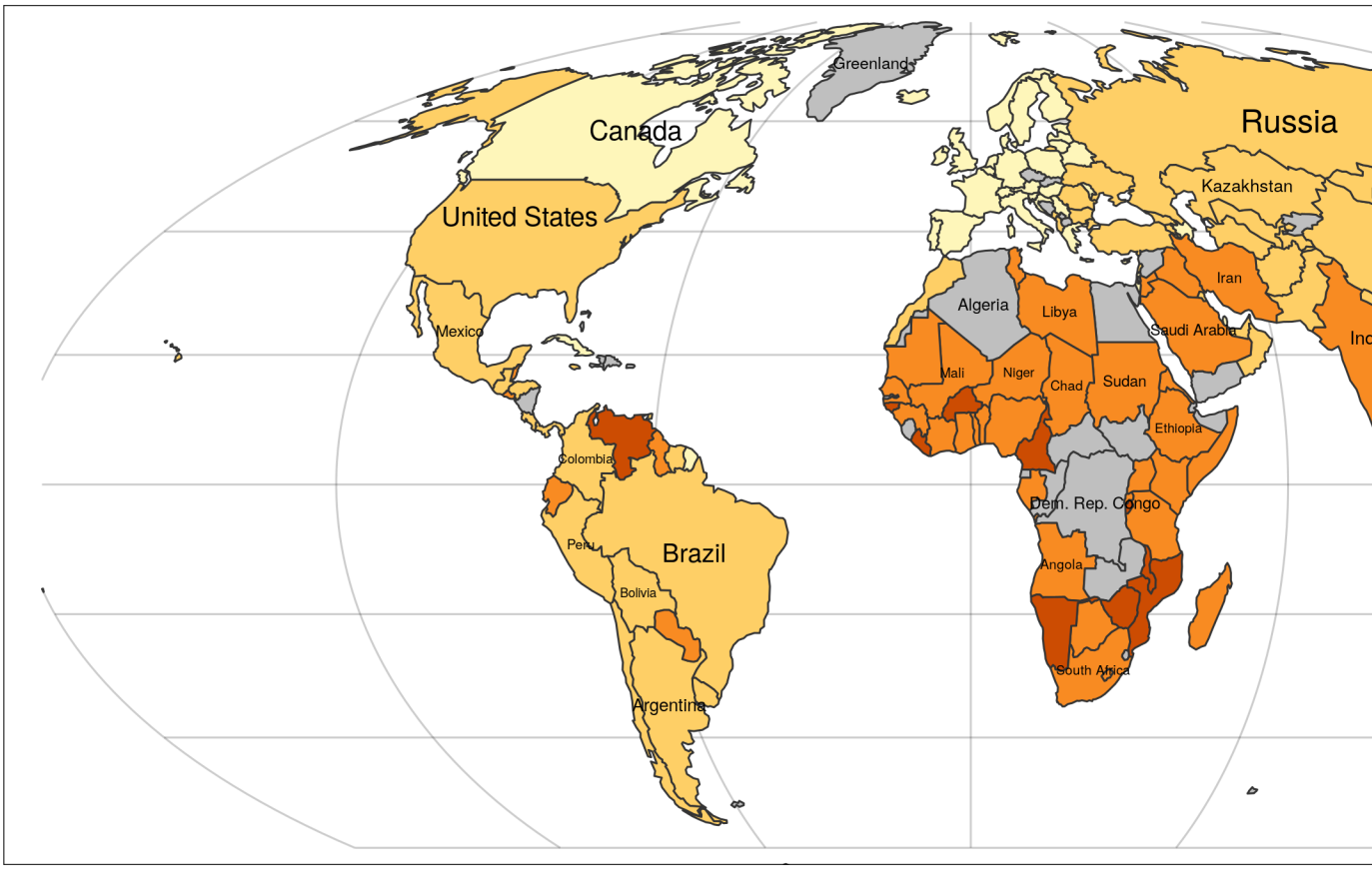


Figure 1.1: Road danger levels worldwide in 2016. Data source: World Bank. Reproducible source code: github.com/itsleeds/rrsrr.

The guide is practical, meaning that you should reproduce the examples that are provided throughout. As with many practical skills, you learn data science by doing data science.

Before getting stuck in with the practical content, which begins in Section 2, the remainder of this chapter:

- introduces the concept reproducibility and its importance for evidence-based policies;
- explains the choice of R as a 'tool of the trade' for road safety research;
- outlines how to install R on your computer or access it through remote servers in the 'cloud'; and
- explains the structure of the manual, outlines the contents of each section and how they should be used for maximum benefit depending on your level of experience and aims (Section 1.5).

1.1 Reproducibility

Reproducible research can be defined as work that generates results that can be regenerated by others using publicly accessible code. By contrast, findings that cannot be repeated are **not reproducible**.

Reproducibility is not a binary concept, but a continuum. At one extreme, there is work that does not report the data source, methods or software. At the other end of that continuum, there are findings that can be reproduced in their entirety, including the production of figures and, as is the case with this manual, the manuscript/medium in which results are presented. Reproducibility can be built into every stage of quantitative research, as shown in Table 1.2.

```
re = readr::read_csv("reproducibility.csv")
```

```
Rows: 4 Columns: 2
-- Column specification --------------------------------------------------------
Delimiter: ","
chr (2): Research component, Requirement

i Use `spec()` to retrieve the full column specification for this data.
i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
knitr::kable(re)
rm(re)
```

Table 1.2: Four elements of reproducibility adapted from Peng, Dominici, and Zeger (2006).

Research component	Requirement
Data	Datasets used are available.
Methods	Computer code underlying figures, tables, etc are available. Software to execute that code is available.
Documentation	Documentation of the computer code, software environment, and methods for others to repeat and build on the analysis.
Distribution	Standard methods of distribution are used for others to access the software, data, and documentation.

The importance of reproducibility in scientific research should be obvious: if findings cannot be repeated, this casts doubt on the validity and truth of the conclusions drawn from them. Reproducibility is vital for *falsifiability*, a cornerstone of science.

In applied policy-relevant research areas, such as road safety research, reproducibility is equally important: policy makers and the public want to have confidence that the evidence underlying key decisions is reliable. Policies based on results that nobody can reproduce are harder to defend than policies that have a clear evidence base open for others to repeat, including members of the public and educators. In transport planning, open and reproducible methods support more transparent and democratically accountable interventions. Reproducibility leads to solid science, which is conducive to effective policies. In the context of road safety research, this means that **reproducibility can save lives**.

1.2 What is R?

R is an open source programming language first developed by award-winning academic statisticians Dr Ross Ihaka and Professor Robert Gentleman. Since its first release in 1995 and the release of version 1.0.0 in 2000, R has seen rapid uptake. As of September 2020, R was ranked as the 9th most used programming language on the TIOBE Index, ahead of other languages for data processing, such as SQL and MATLAB, and behind *general purpose* languages such as C, Java and Python.

An important feature of R is that it was *designed* for data processing and statistical analysis. This means that you can undertake many aspects of road safety research using the core language. R is widely acknowledged to outperform other open languages for data science, such as Julia, Python and Scala, in terms of data visualisation and deployment of web applications for presenting data via the R package `shiny`. Furthermore, recently developed packages `tidyverse` and `sf`, provide a unified and user friendly system for working with attribute-rich and geographic datasets. Because road crash data is commonly attribute-rich and geographic, we will be using these packages in subsequent sections.

1.3 Why R for road safety research?

R is an outstanding language for reproducible research. It is accessible with no licensing restrictions and easy installation procedures on a wide range of computers, including most versions of Windows, Mac and Linux operating systems. Furthermore, R is highly extensible. With 15,000+ packages available, many of which are developed by professional statisticians and domain experts, R provides access to a wide array of statistical, computational and visualisation techniques. Many packages, such as `markdown` and `reprex`, were designed to support more reproducible research.

From a road safety perspective, R is well suited to handling data structures used in road safety research. R excels at the processing, analysis, modelling and visualisation of large spatio-temporal and attribute-rich datasets of the type key to road safety research. R is a mature and growing tool for data science, popular in industry, academia and government,

so it creates multiple opportunities for collaboration within and between organisations and internationally.

1.4 Prerequisites

You do not need to be a professional programmer, data scientist or computer wizard to use R for road safety research. If you have primarily used graphical user interfaces (GUIs), such as Microsoft Excel, it may take some time to get used to the code-based R approach. However, the command-line interface (CLI) of R is no 'harder' than the incessant pointing and clicking demanded by tools such as Excel and web-based GUIs for road safety research. It takes time to adapt to new ways of working, and R has a steep learning curve at the outset. However, persevering can be very rewarding: proficiency with R's CLI is a future-proof and transferable skill that can yield huge productivity gains. Perhaps the most important prerequisite, therefore, is time and a willingness to try new ways of working.

The good news is that it has never been easier to learn and install R, as highlighted in the `stats19-training-setup` that can be found on the **stats19** package website at docs.ropensci.org/stats19.

It is important to have an up-to-date version of R installed before proceeding to the practical sections of this manual. Note that, like any actively developed software, R is evolving so it is worth updating or re-installing R/RStudio every year or so and updating your R packages every month or so to ensure you have the latest software.

1.5 Installing R and RStudio

To complete the exercises in this guide, you will need to install:

- R from cran.r-project.org
- RStudio from rstudio.com
- R packages, by opening RStudio and typing `install.packages("stats19")` in the console to install the `stats19` package, for example (see Section 4 for details)

We recommend using at least the latest stable release of R (4.0.3 at the time of writing in 2020). If you're running on macOS or Linux, you may need to install additional dependencies, as documented in blog posts and documentation pages such as Installation of R 4.0 on Ubuntu 20.04 LTS and tips for spatial packages and the Installing section of the github.com/r-spatial/sf package README page. We recommend running R on a decent computer, with at least 4 GB RAM and ideally 8 GB or more RAM. R is computationally efficient and therefore fast language for data science but, because of the size of some road safety datasets, we recommend using it on a high spec laptop or desktop.

1.6 R in the cloud

If you do not have access to a suitable computer on which you can install R, or just want to get up-and-running quickly, you can run R in the cloud.

Various organisations manage RStudio Server instances, but by far the most well-known cloud provider is at cloud.rstudio.com. To run R in the cloud, sign-up to cloud.rstudio.com (or cloud instance of your choice) and access RStudio from the browser.

1.7 Recommended packages

Of the thousands of available packages that are available for road safety research, we will use a handful that are mature, well-tested and well-suited to statistical analysis and visualisation of road casualty data. For the first practical steps, in Section 2, all you need is a working version of R and RStudio. In Section 4 we will see how to install and use add-on packages such as `stats19`. If you want to be ahead of the game, you can check that you have the necessary packages installed by running the following commands, which **install and load the packages that we will use for the course**:

```
install.packages("remotes") # installs the remotes package
pkgs = c(
  "stats19",     # downloads and formats open stats19 crash data
  "sf",          # spatial data package
  "tidyverse",   # a 'metapackage' with many functions for data processing
  "tmap",        # for making maps
  "pct",         # access travel data from DfT-funded PCT project
  "stplanr"      # transport planning tools
)

remotes::install_cran(pkgs)
lapply(pkgs, library, character.only = TRUE)
```

1.8 Overview

The rest of the manual is structured as follows.

- Section 2 introduces the basics of the R language. While not essential reading for people who already have experience with R or who just want to get stuck-in to importing datasets, as per Section 5, it is recommended reading even if you already use R. This section introduces key aspects of the R language that may not be needed for basic data

analysis tasks, but which will be vital when 'debugging' your code (the process you go through to remove bugs/mistakes). It provides a strong foundation for subsequent sections.

- Section 3 provides a brief introduction to productive research workflows using RStudio, an advanced integrated development environment for not only writing R code, but also for project management and boosting your productivity with a suite of features that puts Excel to shame.
- Section 4 introduces the `stats19` package and other R packages we will be using in subsequent chapters.
- Section 5 demonstrates key data processing techniques using `tidyverse`.
- Section 6 teaches key functions for working with timestamps.
- Section 7 shows how you can create maps and perform geographic data analysis with road crash data in R.
- Section 8 provides an introduction to joining road crash data, with a focus on casualty and accident tables in STATS19 data (introduced in Section 4).
- Section 9 suggests next steps for road safety researchers looking to take their skills to the next levels and provide the strong evidence needed to save lives.

2 R basics

Learning a programming language is like learning any language. If you're learning French, for example, you could just dive in and start gesticulating to people in central Paris. However, it's worth taking the time to understand the structure and key words of the language first. The same applies to data science: it will help if you first understand a little about the syntax and grammar of the language that we will use to in relation to the 'data' (the statistical programming language R) before diving into using it for road safety research. This section, parts of which were originally developed for a 2 day course on data science, may seem tedious for people who just want to crack on and load-in data. However, working through the examples below is recommended for most people unless you're already an experienced R user, although even experienced R users may learn something about the language's syntax and fundamental features.

The first step is to **start RStudio**, e.g. if you are on Windows, this can be achieved by tapping **Start** and searching for RStudio. You should see an **R console** pane like that which is displayed in Figure 2.1.

```
knitr::include_graphics("figures/console.png")
```

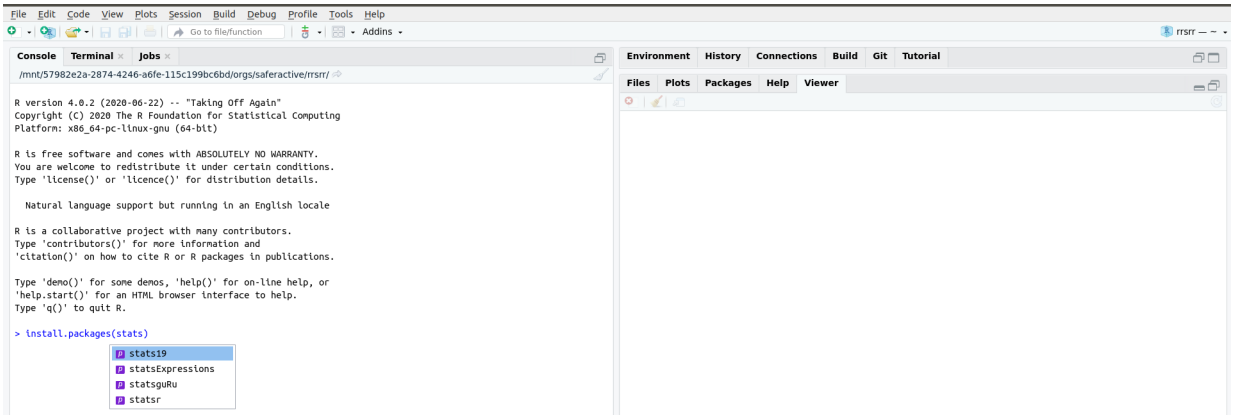


Figure 2.1: The R console pane in RStudio that appears when you first open RStudio. After typing install.packages(stats), you should see the package name auto-complete. Pressing `Enter` at this point will trigger the autocompletion of the command 'install.packages("stats19").

If you saw something like that which is shown in Figure 2.1 congratulations! You are ready to start running R code by entering commands into the console.

2.1 Creating and removing R objects

R can be understood as a giant calculator. If you feed the console arithmetic tasks, it will solve them precisely and instantly. Try typing the following examples (note that `pi` is an inbuilt object) into the R console in RStudio and pressing `Enter` to make R run the code. (`Note:` The output of the code, when shown in this manual, is preceeded by '##'.)

```
2 + 19
```

```
[1] 21
```

```
pi^(19 + 2) / exp(2 + 19)
```

```
[1] 20.89119
```

Use the same approach to find the square route of 361 (answer not shown):

```
sqrt(361)
```

This is all well and good, providing a chance to see how R works with numbers and to get some practice with typing commands into the console. However, the code chunks above do not make use of a key benefit of R: it is *object oriented*, meaning it stores values and complex objects, such as data frames representing road casualties, and processes them *in memory* (meaning that R is both fast and memory hungry when working with large datasets). If you are more familiar with Excel, a data frame may be thought of as fulfilling the purpose of a single worksheet containing a set of data.

The two most common ways of creating objects are using `<-` 'arrow' or `=` 'equals' assignment. These symbols are *assignment operators* because they assign contents, such as numbers, to named objects. We advocate using `snake_case`, a style that avoids upper case characters to ease typing and uses the underscore symbol (`_`) to clearly demarcate spaces between words.

The objects created in the previous code chunk have now served their purpose, which is to demonstrate basic object creation in R. So, based on the wise saying that tidying up is the most important part of a job, we will now remove these objects:

```
x = 2
y = 19
z = x + y
pi^z / exp(z)
```

```
[1] 20.89119
```

```
rm(x, y, z)
```

What just happened? We removed the objects using the R function `rm()`, which stands for 'remove'. A function is an instruction or set of instructions for R to do something with what we give to that function. What we give to the function are known as arguments. Each function has set of arguments we can potentially give to it.

Technically speaking, we *passed the objects to arguments in the `rm()` function call*. In plain English, things that go inside the curved brackets that follow a function name are the arguments. The `rm()` function removes the objects that it is passed (most functions modify objects). A 'nuclear option' for cleaning your workspace is to run the following command, the meaning of which you will learn in the next section. (Can you guess?)

```
rm(list = ls())
```

Next exercise: create objects that more closely approximate road casualty data by typing and running the following lines of code in the console:

```
casualty_type = c("pedestrian", "cyclist", "cat")
casualty_age = seq(from = 20, to = 60, by = 20)
```

2.2 Object types: vectors and data frames

The final stage in the previous section involved creating two objects with sensible names in our R session. After running the previous code chunk the `casualty_*` objects are in the workspace (technically, the 'global environment'). You should be able to see the object names in the Environment tab in the top right of RStudio. Objects can also be listed with the `ls()` command as follows:

```
ls()
```

```
[1] "casualty_age"  "casualty_type"
```

The previous command executed the function `ls()` with no arguments. This helps explain the command `rm(list = ls())`, which removed all objects in the global environment in the previous section. This also makes the wider point that functions can accept arguments (in this case the `list` argument of the `rm()` function) that are themselves function calls.

Two key functions for getting the measure of R objects are `class()` and `length()`.

```
class(casualty_type)
```

```
[1] "character"
```

```
class(casualty_age)
```

```
[1] "numeric"
```

The class of the `casualty_type` and `casualty_type` objects are `character` (meaning text) and `numeric` (meaning numbers), respectively. The objects are *vectors*, a sequence of values of the same type. Next challenge: guess their length and check your guess was correct by running the following commands (results not shown):

```
length(casualty_type)
length(casualty_age)
```

To convert a series of vectors into a data frame with rows and columns (similar to an Excel worksheet), we will use the `data.frame()` function. Create a data frame containing the two casualty vectors as follows:

```
crashes = data.frame(casualty_type, casualty_age)
```

We can see the contents of the new `crashes` object by entering the following line of code. This prints its contents (results not shown, you need to run the command on your own computer to see the result):

```
crashes
```

We can get a handle of data frame objects such as `crashes` as follows:

```
class(crashes)
```

```
[1] "data.frame"
```

```
nrow(crashes)
```

```
[1] 3
```

```
ncol(crashes)
```

```
[1] 2
```

The results of the previous commands tell us that the dataset has 3 rows and 2 columns. We will use larger datasets (with thousands of rows and tens of columns) in later sections, but for now it's good to 'start small' to understand the basics of R.

2.3 Subsetting by index or column name

As we saw above, the most basic type of R object is a *vector*, which is a sequence of values of the same type such as the numbers in the object `casualty_age`. In the earlier examples, `x`, `y` and `z` were all *numeric vectors* with a length of 1; `casualty_type` is a *character vector* (because it contains letters that cannot be added) of length 3; and `casualty_age` is a *numeric vector* of length 3.

Subsetting means 'extracting' only part of a vector or other object, so that only the parts of most interest are returned to us. Subsets of vectors can be returned by providing numbers representing the positions (index) of the elements within the vector (e.g. '2' representing selection of the 2nd element) or with logical (`TRUE` or `FALSE`) values associated with the element. These methods are demonstrated below, to return the 2nd element of the `casualty_age` object is returned:

```
casualty_age
```

```
[1] 20 40 60
```

```
casualty_age[2]
```

```
[1] 40
```

```
casualty_age[c(FALSE, TRUE, FALSE)]
```

```
[1] 40
```

Two dimensional objects such as matrices and data frames can be subset by rows and columns. Subsetting in base R is achieved by using square brackets `[]` after the name of an object. **To practice subsetting, run the following commands to index and column name and verify that you get the same results to those that are shown below.**

```
casualty_age[2:3] # second and third casualty_age
crashes[c(1, 2), ] # first and second row of crashes
crashes[c(1, 2), 1] # first and second row of crashes, first column
crashes$casualty_type # returns just one column
```

The final command used the dollar symbol (`$`) to subset a column. We can use the same symbol to create a new column as follows:

```
vehicle_type = c("car", "bus", "tank")
crashes$vehicle_type = vehicle_type
ncol(crashes)
```

```
[1] 3
```

Notice that the dataset now has three columns after we added one to the right of the previous one. Note also that this would involve copying and pasting cells in Excel, but in R it is instant and happens as fast as you can type the command. To confirm that what we think has happened has indeed happened, print out the object again to see its contents:

```
crashes
```

```
  casualty_type casualty_age vehicle_type
1    pedestrian           20          car
2       cyclist           40          bus
3           cat           60         tank
```

In Section 5 we will use `filter()` and `select()` functions to subset rows and columns. Before we get there, it is worth practicing subsetting using the square brackets to consolidate your understanding of how base R works with vector objects such as `vehicle_type` and data frames such as `crashes`. If you can answer the following questions, congratulations, you are ready to move on. If not, it's worth doing some extra reading and practice on the topic of subsetting in base R.

Exercises

1. Use the `$` operator to print the `vehicle_type` column of `crashes`.

2. Subset the crashes with the `[,]` syntax so that only the first and third columns of `crashes` are returned.
3. Return the 2nd row and the 3rd column of the `crashes` dataset.
4. Return the 2nd row and the columns 2:3 of the `crashes` dataset.
5. **Bonus**: what is the `class()` of the objects created by each of the previous exercises?

2.4 Subsetting by values

It is also possible to subset objects by the values of their elements. This works because the `[` operator accepts logical vectors returned by queries such as 'Is it less than 3?' (`x < 3` in R) and 'Was it light?' (`crashes$dark == FALSE`), as demonstrated below:

```
x[c(TRUE, FALSE, TRUE, FALSE, TRUE)] # 1st, 3rd, and 5th element in x
x[x == 5] # only when x == 5 (notice the use of double equals)
x[x < 3] # less than 3
x[x < 3] = 0 # assign specific elements
casualty_age[casualty_age %% 6 == 0] # just the ages that are a multiple of 6
crashes[crashes$dark == FALSE, ] # just crashes that occured when it wasnt dark
```

Exercises

1. Subset the `casualty_age` object using the inequality (`<`) so that only elements less than 50 are returned.
2. Subset the `crashes` data frame so that only tanks are returned using the `==` operator.
3. **Bonus**: assign the age of all tanks to 61.

2.5 Dealing with NAs and recoding

R objects can have a value of NA. NA is how R represents missing data.

```
z = c(4, 5, NA, 7)
```

NA values are common in real-world data but can cause trouble. For example:

```
sum(z) # result is NA
```

Some functions can be told to ignore NA values.

```
sum(z, na.rm = TRUE) # result is equal to 4 + 5 + 7
```

You can find NAs using the `is.na()` function, and then remove them:

```
is.na(z)
z_no_na = z[!is.na(z)] # note the use of the not operator !
sum(z_no_na)
```

If you remove records with NAs, be warned: the average of a value excluding NAs may not be representative.

2.6 Changing class

Sometimes you may want to change the class of an object. This is called class coercion, and can be done with functions such as `as.logical()`, `as.numeric()` and `as.matrix()`.

Exercises

1. Coerce the `vehicle_type` column of `crashes` to the class `character`.
2. Coerce the `crashes` object into a matrix. What happened to the values?
3. **Bonus:** What is the difference between the output of `summary()` on `character` and `factor` variables?

2.7 Recoding values

Often it is useful to 'recode' values. In the raw STATS19 files, for example, -1 means NA. There are many ways to recode values in R, the simplest and most mature of which is the use of 'factors', which are whole numbers representing characters. Factors are commonly used to manage categorical variables such as sex, ethnicity or, in road traffic research, vehicle type or casualty injury severity.

```
z = c(1, 2, -1, 1, 3)
l = c(NA, "a", "b", "c") # labels in ascending order
z_factor = factor(z, labels = l) # factor z using labels l
z_character = as.character(z_factor) # convert factors to characters
z_character
```

```
[1] "a" "b" NA  "a" "c"
```

Exercises

1. Recode `z` to Slight, Serious and Fatal for 1:3 respectively.
2. **Bonus:** read the help file at `?dplyr::case_when` and try to recode the values using this function.

2.8 Saving R objects

You can also save individual R objects as `.Rds` files. The `.Rds` format is the data format for R, meaning that any R object can be saved as an `Rds` file, equivalent to saving an Excel spreadsheet as a `.xlsx` file. The following command saves the `crashes` dataset into a compressed file called `crashes.Rds`:

```
saveRDS(crashes, "crashes.Rds")
```

Try reading in the data just saved, and checking that the new object is the same as `crashes`, as follows:

```
crashes2 = readRDS("crashes.Rds")
identical(crashes, crashes2)
```

```
[1] TRUE
```

R also supports many other formats, including CSV files, which can be created and imported with the functions `readr::read_csv()` and `readr::write_csv()` (see also the readr package).

```
readr::write_csv(crashes, "crashes.csv") # uses the write_csv function from the readr package
crashes3 = readr::read_csv("crashes.csv")
identical(crashes3, crashes)
```

Notice that `crashes3` and `crashes` are not identical. What has changed? Hint: read the help page associated with `?readr::write_csv`.

2.9 Now you are ready to use RStudio

Bonus: reproduce the following plot by typing the following code into the console.

```r
eyes = c(2.3, 4, 3.7, 4)
eyes = matrix(eyes, ncol = 2, byrow = T)
mouth = c(2, 2, 2.5, 1.3, 3, 1, 3.5, 1.3, 4, 2)
mouth = matrix(mouth, ncol = 2, byrow = T)
plot(eyes, type = "p", main = "RRR!", cex = 2, xlim = c(1, 5), ylim = c(0, 5))
lines(mouth, type = "l", col = "red")
```

3 Using RStudio

The previous section taught the basics of the R language. We entered and ran commands directly in the console. In this section we will learn how to write R scripts in RStudio's source editor. We will also take a step back and considers how R code fits into the wider context of scripts, projects, and getting help in RStudio. RStudio is an integrated development environment (IDE) for R that makes it easy to create and run scripts, explore R objects and functions, plot results and get help.

The first exercise is to open RStudio, take a look around, identify and explore the main components, shown in Figure 3.1. Click on different buttons in RStudio's GUI and try changing the Global Settings (in the Tools menu) and see RStudio's shortcuts by pressing `Alt-Shift-K` (or `Option+Shift+K` on Mac).

```
knitr::include_graphics("figures/rstudio-ui.png")
```

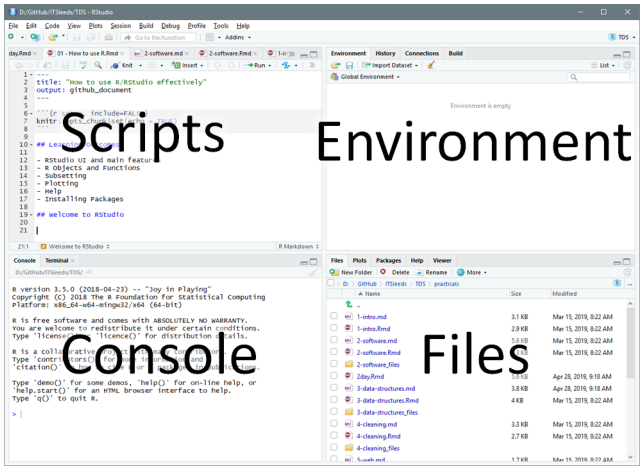


Figure 3.1: The RStudio user interface showing the four main 'panes'.

3.1 Projects and scripts

Projects organise files and settings in RStudio into folders. Each project has its own folder and Rproj file. **When using RStudio, always ensure you are working in a named project**

to organise your work. Start a new project with by clicking on `File > New Project` in RStudio's top menu. You create projects either in a new or existing directory (folder). **Make a new project called 'lrrsrr' (short for 'learning reproducible road safety research with R') or a name of your choice and save it in a sensible place on your computer.** The name of the project will appear in the top right of RStudio.

'Scripts' are files where R code are stored, and these can be edited in the **Source Editor** panel (the top left panel in Figure 3.1). **Keeping your code in sensibly named, well organised and reproducible scripts will make your life easier.** We could continue typing all our code into the console, as we did in Section 2. However, that approach is limited when working on anything more complicated than a few simple commands. Code that you want to keep and share should be saved script files, i.e. plain text files that have the `.R` extension.

Make a new script by typing and running this command in the R console:

```
file.edit("section3.R")
```

This will open the Source Editor and place your cursor there. Try jumping between the Source Editor and the Console by pressing `Ctl+1` and `Ctl+2`.

Keeping scripts and other files associated with a project in a single folder per project (in an RStudio project) will help you to locate things you need and develop an efficient workflow. Next, to check that your project is saved, close RStudio.

3.2 Writing and running code

Re-open RStudio and ensure that you have an empty file open in the Source Editor. We will type some basic commands into this file. Type the following lines of code into your new `section3.R` R script and execute the result line-by-line by pressing `Ctrl+Enter` (`Command+Enter` on Mac):

```
x = 1:5
y = c(0, 1, 3, 9, 18)
plot(x, y)
```

When the code has been sent to the console, two objects are created, both of which are vectors of 5 elements (**Bonus:** check their length using the `length()` function). The third line of the code chunk plots them. Save the script by pressing `Ctrl+S`.

There are several ways to run code within a script and it is worth becoming familiar with each. Try running the code you saved in the previous section using each of these methods:

1. Place the cursor in different places on each line of code and press `Ctrl+Enter` to run that line of code.
2. Highlight a block of code or part of a line of code and press `Ctrl+Enter` to run the highlighted code.
3. Press `Ctrl+Shift+Enter` to run all the code in a script.
4. Select `Run` on the toolbar to run all the code in a script.
5. Bonus: Use the function `source()` to run all the code in a script e.g. `source("section3.R")`

Practice alternating between the console and the source editor by pressing `Ctl+1` and `Ctl+2`.

3.3 Viewing Objects

To practice typing code into scripts, rather than into the console, we will re-create the objects we created in Section 2. Create a new script called `objects.R` and type the following commands, character-for-character, including spaces in the right places. Typing rather than copy-pasting will help develop good coding style and speed:

```
vehicle_type = c("car", "bus", "tank")
casualty_type = c("pedestrian", "cyclist", "cat")
casualty_age = seq(from = 20, to = 60, by = 20)
set.seed(1)
dark = sample(x = c(TRUE, FALSE), size = 3, replace = TRUE)
small_matrix = matrix(1:24, nrow = 12)
crashes = data.frame(vehicle_type, casualty_type, casualty_age, dark)
```

Run the code line-by-line by pressing `Ctl+Enter` multiple times, as described in the previous section. Try viewing the objects in the following ways:

1. Type the name of the object into the console, e.g. `crashes` and `small_matrix`, and run that code. Scroll up to see the numbers that didn't fit on the screen.
2. Use the `head()` function to view just the first 6 rows e.g. `head(small_matrix)`
3. Bonus: use the `n` argument in the previous function call to show only the first 2 rows of `small_matrix`
4. Click on the `crashes` object in the environment tab to View it in a spreadsheet.
5. Run the command `View(vehicle_type)`. What just happened?

We can also get an overview of an object using a range of functions, including:

- `summary()`
- `class()`
- `typeof()`
- `dim()`

- `length()`

View a summary of the `casualty_age` variable by running the following line of code (you should see the same output as shown below):

```
summary(casualty_age)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
     20      30      40      40      50      60
```

Exercise: use the functions listed above (`class()` to `length()`) to test the basic R functions and extract key information about the object `vehicle_type`. What does the output tell us about the object?

3.4 Autocompletion

RStudio can help you write code by autocompleting it. RStudio will look for similar objects and functions after typing the first three letters of a name.

```
knitr::include_graphics("figures/autocomplete.jpg")
```

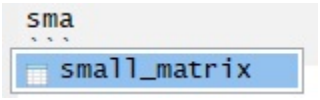


Figure 3.2

When there is more than one option, you can select from the list using the mouse or arrow keys. Within a function, you can get a list of arguments by pressing `Tab`.

```
knitr::include_graphics("figures/functionhelp.jpg")
```

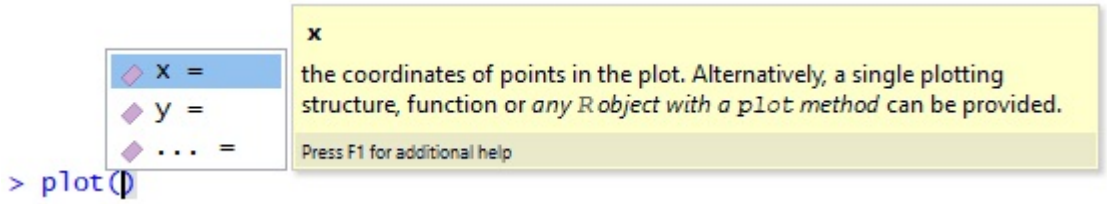


Figure 3.3

Test RStudio's amazing autocompletion capabilities by typing the beginning of object names and functions and pressing `Tab` to see what suggestions pop-up. Try pressing `Up` and `Down` after pressing `Tab` to select different options.

Bonus: try autocompleting file names by typing `""` (the closing quote mark should be added automatically) and pressing `Tab` when your cursor is between the quote marks. What happens when you type `"~/"` and press `Tab` with your cursor just after the tilde (`~`) symbol. What does this mean? (**Hint:** it involves the word 'home' and you can search the web to find a full answer)

3.5 Getting help

Every function in R has a help page. You can view the help using `?`. For example, `?sum` and `?plot`. Many packages also contain 'vignettes', which are long form help documents containing examples and guides. `vignette()` will show a list of all the vignettes available, or you can also view a specific vignette. For example, `vignette(topic = "sf1", package = "sf")`.

Try getting help on the `stats19` package by typing the following and pressing `Tab` when your cursor is just to the left of the closing bracket `)`. Autocompletion works for more than just R objects and files - try making RStudio autocomplete and run the command `vignette("stats19-vehicles")`. For example:

```
vignette(stats19)
```

You can further search and explore R's help files using the **Help** panel in the bottom right window in RStudio.

3.6 Commenting Code

It is good practice to use comments in your code to explain what it does. You can comment code using `#`

For example:

```
# Create vector objects (a whole line comment)
x = 1:5 # a seqence of consecutive integers (inline comment)
y = c(0, 1, 3, 9, 18.1)
```

You can comment/uncomment a whole block of text by selecting it and using `Ctrl+Shift+C`.

Pro tip: You can add a comment section using `Ctrl+Shift+R`.

3.7 The global environment

The 'Environment' tab shows all the objects in your environment. This includes datasets, parameters, and any functions you have created. By default, new objects appear in the Global Environment, but you can see other environments with the dropdown menu. For example, each package has its own environment.

Sometimes you wish to remove things from your environment, perhaps because you no longer need them or because things are getting cluttered.

You can remove an object with the `rm()` function e.g. `rm(x)` or `rm(x, y)`. Alternatively, you can clear your whole environment with the 'broom' button on the 'Environment' Tab.

1. Remove the object `x` that was created in a previous section.
2. What happens when you try to print the `x` by entering it into the console?
3. Try running the following commands in order: `save.image(); rm(list = ls()); load(".RData")`. What happened?
4. How big (how many bytes) is the `.RData` file in your project's folder?
5. Tidy up by removing the `.Rdata` file with `file.remove(".Rdata")`.

3.8 Debugging Code

All the code shown so far is reproducible and, unless you introduced typos, is 'bug free', meaning that it runs without errors. Typos are common though and even experienced R users frequently see error messages as they undertake interactive data analysis. For that reason, learning to fix typos in R code is an important skill. RStudio comes to the rescue here with helpful debugging features. To test them, write some code that fails, as shown in the code chunk and exercises below, then answer the questions below by interacting with RStudio:

```
x = 1:5
y = c(0, 1, 3, 9 18.1) # R code with a typo
```

```
Error in parse(text = input): <text>:2:18: unexpected numeric constant
1: x = 1:5
2: y = c(0, 1, 3, 9 18.1
                     ^
```

```
knitr::include_graphics("figures/rstudio-autocomplete.png")
```


Figure 3.4: Debugging code with RStudio: notice the wavy red line highlighting a typo.

1. Try running the faulty code. How can the error message help debug the code?
2. What is the problem with the code shown in the figure?
3. Create other types of error in the code you have run (e.g. no symetrical brackets and other typos).
4. Does RStudio pick up on the errors? And what happens when you try to run buggy code?

Always address debugging prompts to ensure your code is reproducible

3.9 Productivity boosting features

Finally, we look at functionality in RStudio that goes beyond the features described above. RStudio is an advanced and powerful IDE and is highly customisable in myriad ways, especially since the launch of the RStudio Addins add-on system in 2016. Rather than try to be comprehensive (an impossible task), this section provides a list of additional RStudio features, starting simple, that have been tried and tested, with links to the relevant documentation rather than extended descriptions.

- Zoom levels and appearance settings: it is important for code and other text to be the right size. Too small and it's hard to see, too big and you end up frequently scrolling up and down. The appropriate text size varies: if you're doing a screen share, big text is appropriate; if you're writing copious amounts of text (as I was when writing this prose in RStudio), smaller text will be handy. On Windows and Linux you can zoom with the shortcuts `Ctl+Shift++` and `Ctl+-` for zooming in and out respectively. See 'Tools > Global Options' menu (which can be launched with the shortcut `Alt+T` and then `G`) for more advanced 'Appearance' settings.
- Global search (and replace): in addition to search and replace functionality for single files (accessed in the standard way, by pressing `Ctl+F`), RStudio has a powerful global search feature inbuilt. Launch this feature by pressing `Ctl+Shift+F` and you can search any file types (e.g. only files ending in `.R`) for any string within an entire project. This feature is very handy when working with large, multi-file projects.
- Shortcuts: there are many, many shortcuts built into RStudio. In fact, there is even a shortcut to show the list of shortcuts. Try pressing `Alt+Shift+K` to get the complete list. Nobody I know can remember, let alone use, all of these. However, over time I expect that you will learn to love some of them. My top 5 RStudio-specific shortcuts are:
  - `Ctl+Enter` to send a line of code from the code editor (called the Source Editor in RStudio) to be executed or 'run' in the console. Amazingly, some other prominent IDEs such as Microsoft's VSCode editor, lack this important feature by default.
  - `Ctl+1` and `Ctl+2` to switch between the console (for writing test code and 'run once' commands) and the code editor (for writing code to keep).

  - `Alt+Up/Down` and `Alt+Shift+Up/Down` to move and copy lines of code up and down, handy when you want to re-order your code or make small changes to a copy of a line of code.
  - `Ctl+Shift+M` will create the pipe operator (`%>%`, this pipe was created using the shortcut!), saving time when creating `dplyr` pipelines, as discussed in Section 5.
  - `Ctl+Shift+F10` when you want to restart R, leaving you with a 'blank slate' in which packages are not loaded and objects are removed from the global environment.

- Git integration for collaboration: RStudio provides two mechanisms for sharing your code with others via sites such as GitHub and GitLab, with the 'Git' panel in the top right pane and via the Terminal panel described in the next section.

- Support for Python, C++ and other languages: a joke on Twitter said "What's the best Python editor? RStudio." Although most Python programmers would probably disagree, the joke is true in the sense that R has good support for some other languages: Python and C++ in particular. If you open a Python script in RStudio on a computer that has Python and the `reticulate` R package installed, the R console will magically convert into a Python console when you press `Ctl+Enter` to execute a line of Python code, as described in the article "Reticulated Python" on the RStudio website.

Like R package, an active community of developers is developing a range of extensions and RStudio itself is gradually evolving to meet the evolving needs of 21st Century data scientists. If there are any features that you would like to see, you can always ask others for pointers, e.g. on the RStudio Community forum.

4 R packages

4.1 What are packages?

R has over 15,000 packages published on the official 'CRAN' site and many more published on code sharing sites such as GitHub. Packages are effectively plugins for R that extend it in many ways. Packages are useful because they enhance the range of things you can do with R, providing additional functions, data and documentation that build on the core (known as 'base') R packages. They range from general-purpose packages, such as `tidyverse` and `sf`, to domain-specific packages, such as `stats19`.

This chapter demonstrates the package lifecycle with reference `stats19` and provides a taster of R's visualisation capabilities for general purpose packages `ggplot2` and `dplyr`. The `stats19` package is particularly relevant for reproducible road safety research: its purpose is to download and clean road traffic collision data from the UK's Department for Transport. Domain-specific packages, such as `stats19`, are often written by subject-matter experts, providing tried and tested solutions within a particular specialism. Packages are reviewed by code experts prior to being made available via CRAN.

Regardless of whichever packages you install and use, you will take the following steps:

1. installing the package;
2. loading the package;
3. using the package; and
4. updating the package.

Of these, the third stage takes by far the most amount of time. Stages 1, 2 and 4 are equally important, however; you cannot use a package unless it has been properly installed, loaded and, to get the best performance out of the latest version, updated when new versions are released. We will learn each of these stages of the package lifecycle with the `stats19` package.

4.2 The stats19 R package

Like many packages, `stats19` was developed to meet a real world need. STATS19 data is provided as a free and open resource by the Department for Transport, encouraging evidence-based and accountable road safety research and policy interventions. However, researchers at

the University of Leeds found that repeatedly downloading and formatting open STATS19 data was time-consuming, taking valuable resources away from more valuable (and fun) aspects of the research process. Significantly, manually recoding the data was error prone. By packaging code, we found that we could solve the problem in a free, open and reproducible way for everyone.

By abstracting the process to its fundamental steps (download, read, format), the `stats19` package makes it easy to get the data into appropriate formats (of classes `tbl`, `data.frame` and `sf`), ready for further processing and analysis. The package built upon previous work, with several important improvements, including the conversion of crash data into geographic data in a `sf` data frame for geographic research. It enables creation of geographic representations of crash data, geo-referenced to the correct coordinate reference system, in a single function called `format_sf()`. Part-funded by the RAC Foundation, the package should be of use to academic researchers and professional road safety data analysts working at local authority and national levels in the UK.

The following sections demonstrate how to install, load and use packages with reference to `stats19`. This information can be applied in relation to any package.

4.3 Installing packages

The `stats19` package is available on CRAN. This means that it has a web page on the CRAN website at cran.r-project.org with useful information, including who developed the package, what the latest version is, and when it was last updated (see cran.r-project.org/package=stats19). More importantly, being 'on CRAN' (which technically means 'available on the Comprehensive R Archive Network') means that it can be installed with the command `install.packages()` as follows:

```
install.packages("stats19")
```

You might think that now that the package has been installed we can start using it, but that is not true. This is illustrated in the code below, which tries and fails to run the `find_file_name()` function from the `stats19` package to find the file containing STATS19 casualties data for the year 2020. Check that this function exists by running the following command `?find_file_name`:

```
find_file_name(years = 2020, type = "casualt")
```

```
Error in find_file_name(years = 2020, type = "casualt"): could not find function "find_file_
```

4.4 Loading packages

After you have installed a package the next step is to 'load' it. Load the `stats19` package, that was installed in the previous section, using the following code:

```
library(stats19)
```

```
Data provided under OGL v3.0. Cite the source and link to:
www.nationalarchives.gov.uk/doc/open-government-licence/version/3/
```

What happened? Other than the message telling us about the package's datasets (most packages load silently, so do not worry if nothing happens when you load a package), the command above made the functions and datasets in the package available to us. Now we can use functions from the package without an error message, as follows:

```
find_file_name(years = 2020, type = "casualt")
```

```
[1] "dft-road-casualty-statistics-casualty-2020.csv"
```

This raises the question: how do you know which functions are available in a particular package? You can find out using the autocompletion, i.e. by pressing `Tab` after typing the package's name, followed by two colons. Try typing `stats19::` and then hitting `Tab`, for example. You should see a load of function names appear, which you view by pressing `Up` and `Down` on your keyboard.

The final thing to say about packages is that they can be used without being loaded by typing `package::function()`. We used this before in Section 2.8, where we imported csv data using the `readr` package via `readr::read_csv()`.
So `stats19::find_file_name(years = 2020, type = "casualt")` works even if the package isn't loaded.

You can test this by running the `sf_extSoftVersion()` command from the `sf` package. This command reports the versions of key geographic libraries installed on your system. In the first attempt below, the command fails and reports an error. In the second and third attempts, utilising `::` and `library`, you can see that the command succeeds:

```
# try running a function without loading the sf package first
sf_extSoftVersion()
```

```
Error in sf_extSoftVersion(): could not find function "sf_extSoftVersion"
```

```
# run a function from a package's namespace without loading it but using ::
sf::sf_extSoftVersion()
```

```
          GEOS           GDAL         proj.4 GDAL_with_GEOS     USE_PROJ_H
      "3.12.1"        "3.8.4"        "9.4.0"         "true"         "true"
          PROJ
       "9.4.0"
```

```
# fun a function call after loading the package (the most common way)
library(sf)
```

```
Linking to GEOS 3.12.1, GDAL 3.8.4, PROJ 9.4.0; sf_use_s2() is TRUE
```

```
sf_extSoftVersion()
```

```
          GEOS           GDAL         proj.4 GDAL_with_GEOS     USE_PROJ_H
      "3.12.1"        "3.8.4"        "9.4.0"         "true"         "true"
          PROJ
       "9.4.0"
```

As a bonus, try running the command `sf::sf_extSoftVersion` without the brackets `()`. What does that tell you about the package?

4.5 Using packages

After loading a package, as described in the previous section, you can start using its functions. In the `stats19` package that means the following command `get_stats19()` will now work:

```
crashes_2020 = get_stats19(year = 2020, type = "accidents")
nrow(crashes_2020)
```

```
[1] 91199
```

This command demonstrates the value of packages. It would have been possible to get the same dataset by manually downloading and cleaning the file from the STATS19 website on data.gov.uk. However, by using the package, the process has been achieved much faster and with fewer lines of code than would have been possible using general-purpose base R functions.

The result of the `nrow()` function call shows that we have downloaded a decent amount of data representing over 100k road traffic casualty incidents across Great Britain in 2020.

We will use other functions from the package in subsequent sections of this guide. If you would like to learn more about `stats19` and how it can be used for road safety research, check out its vignettes. The `stats19` vignette, for example, should appear in the **Help** panel in the bottom right panel in RStudio after running the following command:

```
vignette("stats19")
```

4.6 Updating packages

Packages can be updated with the command `update.package()` or in 'Tools > Check for Package Updates' in RStudio. You only need to install a package once but packages can be updated many times. It is important to update packages regularly because updates will offer bug-fixes and other improvements. To update just one package, you can give the function a package name, e.g.:

```
update.packages(oldPkgs = "stats19")
```

Completing the following short exercises will ensure you've got a good understanding of packages and package versions.

1. Take a look in the 'Packages' tab in the 'Files' pane in RStudio (bottom right by default).
2. What version of the `stats19` package is installed on your computer?
3. What happens the second time you run `update.packages()`. Why?

4.7 ggplot2

`ggplot2` is a generic plotting package that is part of the 'tidyverse' meta-package. The `tidyverse` is an 'Opinionated collection of R packages designed for data science'. `ggplot2` is flexible, popular and has dozens of add-on packages which build on it, such as `gganimate`. To plot non-spatial data, it works as follows (the command should generate the image shown in Figure 4.1, showing a bar chart of the number of crashes over time):

```
library(ggplot2)
ggplot(crashes_2020) + geom_bar(aes(date), width = 1)
```

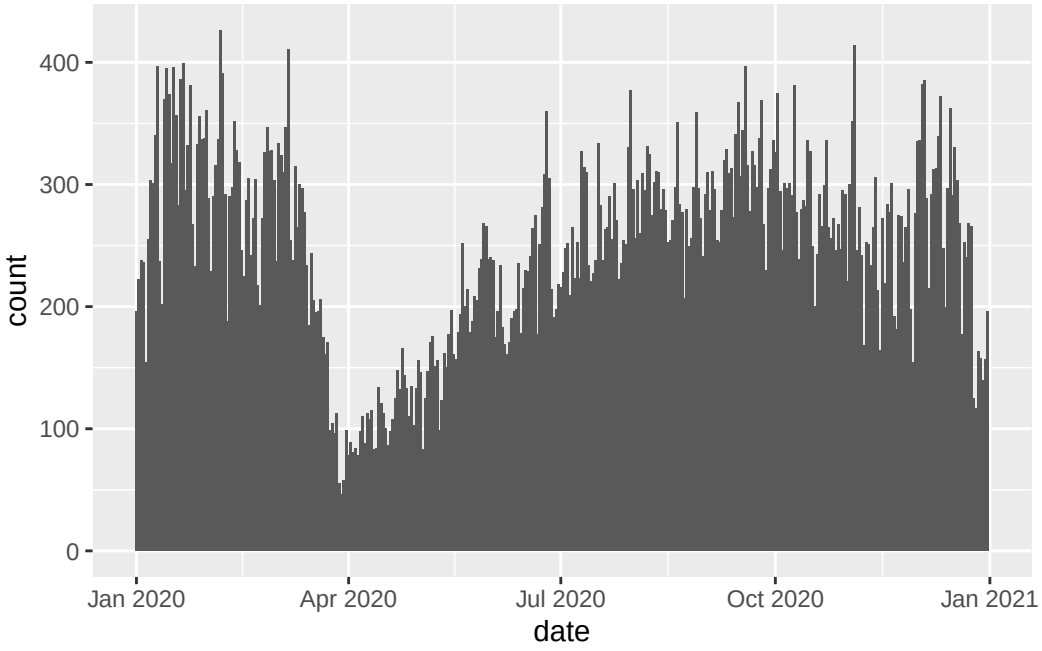


Figure 4.1: A simple ggplot2 graph.

A key feature of the `ggplot2` package is the function `ggplot2()`. This function initiates the creation of a plot by taking a data object as its main argument followed by one or more 'geoms' that represent layers (in this case a bar chart represented by the function `geom_bar()`). Another distinctive feature of `ggplot2()` is the use of `+` operator to add layers.

The package is excellent for generating publication quality figures. Starting from a basic idea, you can make incremental tweaks to a plot to get the output you want. Building on the figure above, we could make the bin width (width of the bars) wider, add colour depending on the crash severity and use count (Figure 4.2) or proportion (Figure 4.3) as our y axis, for example, as follows:

```
ggplot(crashes_2020) + geom_bar(aes(date, fill = accident_severity), width = 1)
ggplot(crashes_2020) +
  geom_bar(aes(date, fill = accident_severity), width = 1, position = "fill") +
  ylab("Proportion of crashes")
```

The package is huge and powerful, with support for a very wide range of plot types and themes, so it is worth taking time to read the documentation associated with the package, starting with the online reference manual and heading towards the online version of the package's official book. As a final taught bit of `ggplot2` code in this section, create a facetted plot showing how the number of crashes per hour varies across the days of the week by typing the following into

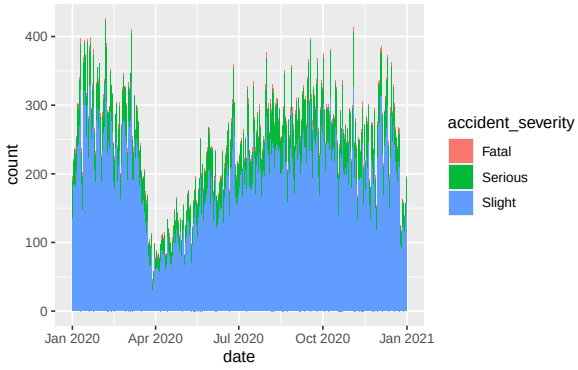


Figure 4.2: Demonstration of fill and position arguments in ggplot2.

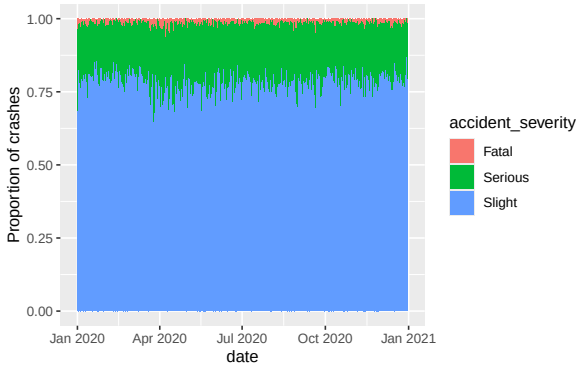


Figure 4.3: Demonstration of fill and position arguments in ggplot2.

the Source Editor and running the chunk line-by-line (the meaning of the commands should become clear by the end of the next section):

```
library(tidyverse)
crashes_2020 %>%
  mutate(hour = lubridate::hour(datetime)) %>%
  mutate(day = lubridate::wday(date)) %>%
  filter(!is.na(hour)) %>%
  ggplot(aes(hour, fill = accident_severity)) +
  geom_bar(width = 1.01) +
  facet_wrap(~day)
```

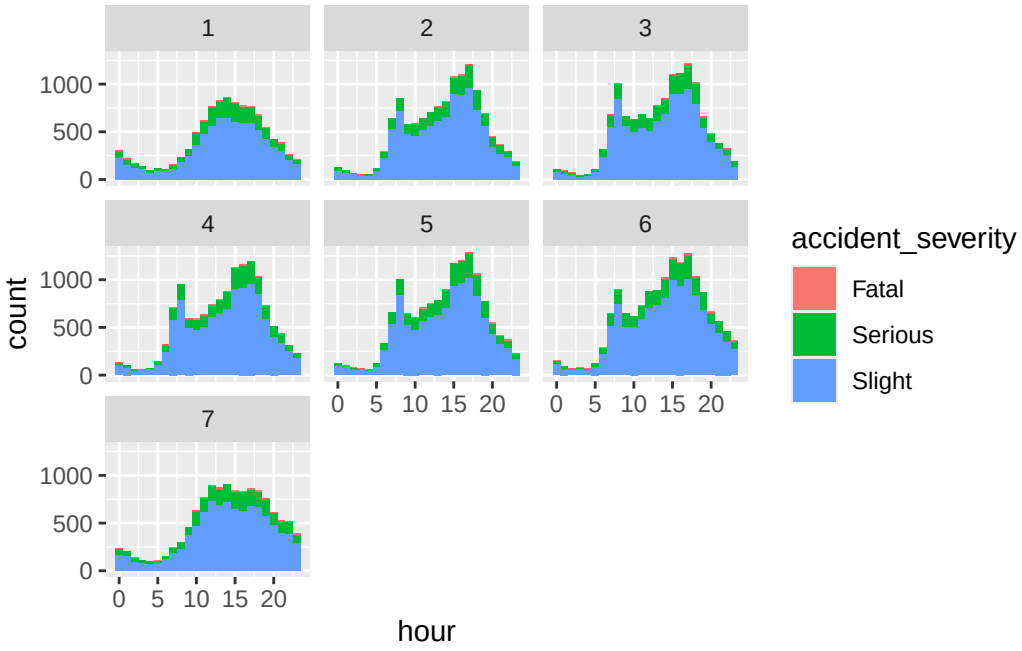


Figure 4.4: A plot showing a facetted time series plot made with ggplot2.

Exercises: 1. Install a package that build on `ggplot2` that begins with with `gg`. Hint: enter `install.packages(gg)` and hit `Tab` when your cursor is between the `g` and the `)`. 2. Open a help page in the newly installed package with the `?package_name::function()` syntax. 3. Load the package. 4. **Bonus:** try using functionality from the new 'gg' package building on the example above to create plots like those shown below (**Hint:** the right plot below uses the economist theme from the `ggthemes` package; try other themes).

4.8 dplyr

Another useful package in the tidyverse is `dplyr`, which stands for 'data pliers', which provides a handy syntax for data manipulation. `dplyr` has many functions for manipulating data frames and using the pipe operator `%>%`. The pipe operator puts the output of one command into the first argument of the next, as shown below (**Note:** the results are the same):

```
library(dplyr)
class(crashes_2020)
```

```
[1] "spec_tbl_df" "tbl_df"      "tbl"         "data.frame"
```

```
crashes_2020 %>% class()
```

```
[1] "spec_tbl_df" "tbl_df"      "tbl"         "data.frame"
```

We will learn more about this package and its other functions in Section 5.

5 Manipulating data

This section is an introduction to manipulating datasets using the `dplyr` package. As outlined in the previous section, `dplyr` and `ggplot2` are part of the `tidyverse`, which aims to provide a user-friendly framework for data science.

Experience of teaching R over the past few years suggests that many people find it easier to get going with data driven research if they learn the 'tidy' workflow presented in this section. However, if you do not like this style of R code or you are simply curious, we encourage you to try alternative approaches for achieving the similar results using base R, the `data.table` R package or other languages such as Python or Julia. If you just want to get going with processing data, the `tidyverse` is a solid and popular starting point.

Before diving into the `tidyverse`, it is worth re-capping where we have got to so far as we have covered a lot of ground. Section 2 introduced R's basic syntax; Section 3 showed how to use the Source Editor and other features of RStudio to support data science; and Section 4 introduced the concept and practicalities of R packages, with reference to `stats19`, `ggplot2` and `dplyr`.

In this section, we will start with a blank slate. In Section 2 we learned that in R having a 'clear desk' means an *empty global environment*. This can be achieved by running the following command, which removes the `list()` of all objects returned by the function `ls()`:

```
rm(list = ls())
```

5.1 tibbles

Although the data processing techniques in R are capable of handling large datasets, such as the `crashes_2023` object that we created in the previous section, representing 100k+ casualties, it makes sense to start small. Let's start by re-creating the `crashes` dataset from Section 2, but this time using the `tidyverse` `tibble()` function. This is the `tidyverse` equivalent of base R's `data.frame`. `tibble` objects can be created, after loading the `tidyverse`, as follows:

```
library(tidyverse)
crashes = tibble(
  casualty_type = c("pedestrian", "cyclist", "cat"),
```

```
  casualty_age = seq(from = 20, to = 60, by = 20),
  vehicle_type = c("car", "bus", "tank"),
  dark = c(TRUE, FALSE, TRUE)
)
```

In the previous code chunk, we passed four vector objects as *named arguments* to the `tibble` function, resulting in columns such as `casualty_type`. A `tibble` is just a fancy way of representing `data.frame` objects, preferred by `tidyverse` users and optimised for data science. It has a few sensible defaults and advantages compared with the `data.frame`, one of which can be seen by printing a `tibble`:

```
class(crashes)
```

```
[1] "tbl_df"     "tbl"        "data.frame"
```

```
crashes
```

```
# A tibble: 3 x 4
  casualty_type casualty_age vehicle_type dark
  <chr>                <dbl> <chr>        <lgl>
1 pedestrian              20 car          TRUE
2 cyclist                 40 bus          FALSE
3 cat                     60 tank         TRUE
```

Note the `<chr>`, `<dbl>` or `<lgl>` text below each column, providing a quick indication of the class of each variable - this is not provided when using `data.frame`.

5.2 filter() and select() rows and columns

In the previous section, we briefly introduced the package `dplyr`, which provides an alternative to base R for manipulating objects. `dplyr` provides different, and some would argue simpler, approaches for subsetting rows and columns than base R.
`dplyr` operations for subsetting rows (with the function `filter()`) and columns (with the function `select()`) are demonstrated below. Here we can also see the use of the pipe operator `%>%` to take the dataset and apply the function to that dataset.

```
crashes %>% filter(casualty_age > 50) # filters rows
```

```
# A tibble: 1 x 4
  casualty_type casualty_age vehicle_type dark
  <chr>                <dbl> <chr>        <lgl>
1 cat                     60 tank         TRUE
```

```
crashes %>% select(casualty_type) # select just one column
```

```
# A tibble: 3 x 1
  casualty_type
  <chr>
1 pedestrian
2 cyclist
3 cat
```

It should be clear what happened: `filter()` returns only rows that match the criteria in the function call, only observations with a `casualty_age` greater than 50 in this case. Likewise, `select()` returns data objects that include only columns named inside the function call, `casualty_type` in this case.

To gain a greater understanding of the functions, type and run the following commands, which also illustrate how the `%>%` can be used more than once to manipulate data (more on this soon):

```
crashes_darkness = crashes %>% filter(dark)
crashes_a = crashes %>% select(contains("a"))
crashes_darkness_a = crashes %>%
  filter(dark) %>%
  select(contains("a"))
```

Can you guess what the dimensions of the resulting objects will be? Write down your guesses for the number of rows and number of columns that the new objects, `crashes_darkness` to `crashes_darkness_a`, have before running the following commands to find out. This also demonstrates the handy function `dim()`, short for dimension (results not shown):

```
dim(crashes)
dim(crashes_darkness)
?contains # get help on contains() to help guess the output of the next line
dim(crashes_a)
dim(crashes_darkness_a)
```

Look at the help pages associated with `filter()`, `select()` and the related function `slice()` as follows and try running the examples that you will find at the bottom of the help pages for each to gain a greater understanding (note you can use the `package::function` notation to get help on functions also):

```
?dplyr::filter
?dplyr::select
?dplyr::slice
```

5.3 Ordering and selecting the 'top n'

Other useful pipe-friendly functions are `arrange()` and `top_n()`. `arrange()` can be used to sort data. Within. the `arrage()` function, optional arguments can be used to define the order in which it is sorted. `top_n()`simply selects the top 'n' number of rows in your data frame. We can use these functions to arrange datasets and take the top most 'n' values, as follows:

```
crashes %>%
  arrange(vehicle_type)
```

```
# A tibble: 3 x 4
  casualty_type casualty_age vehicle_type dark
  <chr>                <dbl> <chr>        <lgl>
1 cyclist                 40 bus          FALSE
2 pedestrian              20 car          TRUE
3 cat                     60 tank         TRUE
```

```
crashes %>%
  top_n(n = 1, wt = casualty_age)
```

```
# A tibble: 1 x 4
  casualty_type casualty_age vehicle_type dark
  <chr>                <dbl> <chr>        <lgl>
1 cat                     60 tank         TRUE
```

5.4 Summarise

A powerful two-function combination is `group_by()` and `summarise()`. Used together, they can provide *grouped summaries* of datasets. In the example below, we find the mean age of casualties in dark and light conditions.

```
crashes %>%
  group_by(dark) %>%
  summarise(mean_age = mean(casualty_age))
```

```
# A tibble: 2 x 2
  dark  mean_age
  <lgl>    <dbl>
1 FALSE       40
2 TRUE        40
```

The example above shows a powerful feature of these pipelines. Many operations can be 'chained' together, whilst keeping readability with subsequent commands stacked below earlier operations. The combination of `group_by()` and `summarise()` can be very useful in preparing data for visualisation with a `ggplot2` function. Another useful feature of the `tidyverse` from a user perspective is the autocompletion of column names mid pipe. If you have not noticed this already, you can test it by typing the following, putting your cursor just before the `)` and pressing `Tab`:

```
crashes %>% select(ca) # press Tab when your cursor is just after the a
```

You should see `casualty_age` and `casualty_type` pop up as options that can be selected by pressing `Up` and `Down`. This may not seem like much, but when analysing large datasets with dozens of variables, it can be a godsend.

Rather than providing a comprehensive introduction to the `tidyverse` suite of packages, this section should have offered enough to get started with using it for road safety data analysis. For further information, check out up-to-date online courses from respected organisations like Data Carpentry and the free online books such as R for Data Science.

5.5 Tidyverse exercises

1. Use `dplyr` to filter rows in which `casualty_age` is less than 18, and then 28.
2. Use the `arrange` function to sort the `crashes` object in descending order of age (**Hint:** see the `?arrange` help page).
3. Read the help page of `dplyr::mutate()`. What does the function do?
4. Use the mutate function to create a new variable, `birth_year`, in the `crashes` data.frame which is defined as the current year minus their age.
5. **Bonus:** Use the `%>%` operator to filter the output from the previous exercise so that only observations with `birth_year` after 1969 are returned.

6 Temporal data

Time is ubiquitous in road safety data, since collisions and road safety implementations always happen at some point in time. This section will show how you can analyse the temporal dimensions of the real world `crashes_2021` object we created in Section 4, and then demonstrate how to handle time series data in base R, as well as with `hms` and `lubridate` packages. The aim is to get you up-to-speed with how data analysis with time data 'feels' before learning the details in subsequent sections. If you are the kind of person who likes to know the details first, feel free to skip this section and return to it later.

```
library(tidyverse)
```

```
-- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
v dplyr     1.1.4     v readr     2.1.5
v forcats   1.0.0     v stringr   1.5.1
v ggplot2   3.5.2     v tibble    3.3.0
v lubridate 1.9.4     v tidyr     1.3.1
v purrr     1.1.0
-- Conflicts ------------------------------------------ tidyverse_conflicts() --
x dplyr::filter() masks stats::filter()
x dplyr::lag()    masks stats::lag()
i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to beco
```

```
crashes = tibble(
  casualty_type = c("pedestrian", "cyclist", "cat"),
  casualty_age = seq(from = 20, to = 60, by = 20),
  vehicle_type = c("car", "bus", "tank"),
  dark = c(TRUE, FALSE, TRUE)
)
```

6.1 Temporal analysis of crash data

To get a feel for temporal data analysis in R, let's start by reading-in crash data for 2021 with the `stats19` package by typing the following into the Source Editor and running the code, line-by-line, as taught in Section 3:

```
library(stats19)
crashes_2021 = get_stats19(2021)
```

Note that, unlike the longer `crashes_2021 = get_stats19(year = 2021, type = "accidents")` used in Section 4, we did not use *named arguments* in this code chunk. Instead of `year = 2021`, we simply typed 2021. That is possible because R functions can be specified by name matching or order: the first argument of `get_stats()` is `year`, so the function is expecting a year value. Also, although we didn't explicitly specify the `accidents` table, `type = "accidents"` is the default value, so `type` only needs to be specified when importing casualty and vehicle datasets.

With that educational aside out of the way, we will now take a look at the time variables that are actually in our newly read-in dataset:

```
library(tidyverse)
crashes_2021 %>%
  select(matches("time|date")) %>%
  names()
```

```
[1] "date"     "time"     "datetime"
```

Building on the previous section and a bit of guesswork, it should be clear what just happened: we selected variables that *match* (with the `matches()` function) the character strings `"time"` or (as indicated by the `|` vertical pipe symbol) `"date"` and returned the matching variable names. This shows that the `stats19` package gives you not one, not two, but three temporal variables.

Exercises:

1. Print the first 6 and then the first 10 elements of each of the three temporal variables in `crashes_2021`.
2. What is the class of each variable (technically, of each vector)?
3. **Bonus:** Extract the weekday from the variable called `date`.
4. **Bonus:** How many crashes happened on Monday?

Of the three time variables, it should be clear from the outcome of previous exercises that `datetime` contains the most useful information. To consolidate the plotting know-how learned in Section 4, we shall start by simply plotting the `datetime` object (Figure 6.1). Plotting data is a good way of understanding new datasets and the variables they contain. Create the following three plots to show how `date` and `time` vary as a function of `datetime`:

```
library(ggplot2)
ggplot(crashes_2021) + geom_point(aes(datetime, date))
ggplot(crashes_2021) + geom_point(aes(datetime, time))
b = c("07:00", "09:00", "12:00", "17:00", "19:00")
ggplot(crashes_2021) + geom_point(aes(datetime, time), alpha = 0.01) +
  scale_y_discrete(breaks = b)
```

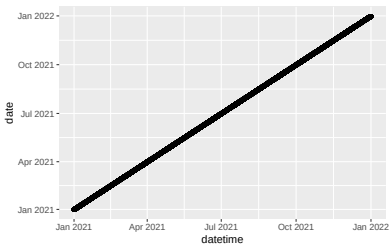


Figure 6.1: Three plots of the datetime (x axis) in relation to the date and time axis.

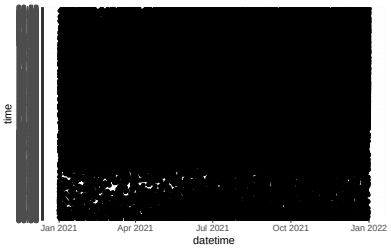


Figure 6.2: Three plots of the datetime (x axis) in relation to the date and time axis.

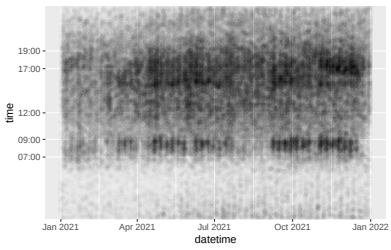


Figure 6.3: Three plots of the datetime (x axis) in relation to the date and time axis.

The three figures above tell us many things about the contents of the three temporal variables. It evenprovides insight into the temporal distribution of road casualties in Great Britain. The first two plots (Figure 6.1 and 6.2) show: 1) that the `date` variable is identical to the `datetime` variable (at least on the daily resolution than can be seen on the graph); and 2) that `time` values repeat regularly for the range of dates in `datetime` (from the start of Jan 2021 to end of Dec 2021). Figure 6.3 makes use of `ggplot2`'s functionality to show only certain labels on the Y axis and reduced opacity, so that overlapping points are not completely black. This by far is the most useful of the three plots, showing that most crashes happen between around 7am and 7pm, with a 'long tail' of crashes in the evening, and that for most of the year there is a clear weekly cycle, reflecting the uptick in crashes during the rush hour commute on weekdays, a pattern that is greatly diminished during several weeks in summer (perhaps corresponding with summer holidays). The 52 weeks of the year can be distinguished even in this small and simple plot, highlighting the ability of visualisation to help understand data. Next, let's look at how the time-of-day that crashes occur varies as a function of season, severity and day of week.

From the `datetime` object of class `POSIXct`, any type of time information can be extracted. This includes the minute, hour, day of week and month of the crash (or other) event that the object records.

Building on the time series plot we created in Section 4, let's create a graph showing how the hourly distribution of crash numbers changes during the course of a working week. We will

do this first by *preprocessing* the data, creating a new object called `crashes_dow`, containing `hour` and `day` columns, then filtering out weekend, and plotting the results, as shown in the code chunk below and Figure 6.4:

```
# days of the week:
dow = c("Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday")
crashes_dow = crashes_2021 %>%
  mutate(hour = lubridate::hour(datetime)) %>%
  mutate(day = factor(weekdays(date), levels = dow))

crashes_dow %>%
  filter(!is.na(hour) & !day %in% c("Saturday", "Sunday")) %>%
  ggplot(aes(hour)) +
  geom_bar(width = 1.01) +
  facet_wrap(~day, nrow = 1)
```

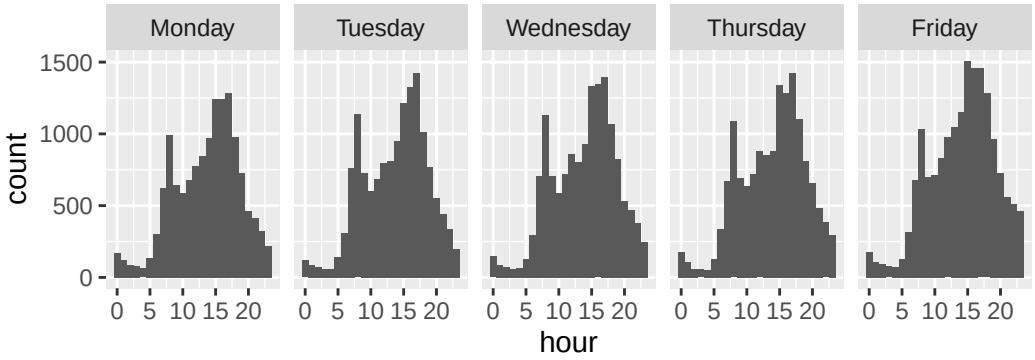


Figure 6.4: Facetted time series showing how the number of crashes increases during the working week.

The result in Figure 6.4 is useful, but if we're interested in the number of crashes per hour on different days of the week *relative to the average*, we need to undertake more preprocessing steps. We will count the number of crashes per hour for all 5 working days and then divide by 5 to get the average number of crashes per hour during weekdays. Then we will count the number of crashes per hour/week combination. Finally we will divide the latter by the former. These steps are shown in the code chunk below, which results in Figure 6.5.

```
crashes_day_rel = crashes_dow %>%  # create 'day of week relative' object
  filter(!is.na(hour) & !day %in% c("Saturday", "Sunday")) %>% # none on weekends
  select(day, hour) %>%            # keep only time columns
  group_by(hour) %>%               # group by hour
```

```
  mutate(n_per_hour = n()/5) %>%   # number per hour (divide by 5 for n. days)
  group_by(day, hour) %>%          # group by day and hour
  summarise(n_hday = n(), n_h = first(n_per_hour)) %>% # summarise results
  mutate(hday_relative = n_hday / n_h) # calculate relative n. crashes per hour/day
summary(crashes_day_rel)
```

```
       day          hour           n_hday           n_h        
 Sunday   : 0   Min.   : 0.00   Min.   :  54.0   Min.   :  63.6  
 Monday   :24   1st Qu.: 5.75   1st Qu.: 195.5   1st Qu.: 252.2  
 Tuesday  :24   Median :11.50   Median : 662.5   Median : 657.3  
 Wednesday:24   Mean   :11.50   Mean   : 630.2   Mean   : 630.2  
 Thursday :24   3rd Qu.:17.25   3rd Qu.: 952.8   3rd Qu.: 899.1  
 Friday   :24   Max.   :23.00   Max.   :1510.0   Max.   :1397.2  
 Saturday : 0                                                    
 hday_relative   
 Min.   :0.7087  
 1st Qu.:0.9309  
 Median :0.9869  
 Mean   :1.0000  
 3rd Qu.:1.0440  
 Max.   :1.6220  
```

```
crashes_day_rel %>%
  ggplot() +
  geom_col(aes(hour, hday_relative)) +
  facet_wrap(~day, nrow = 1)
```

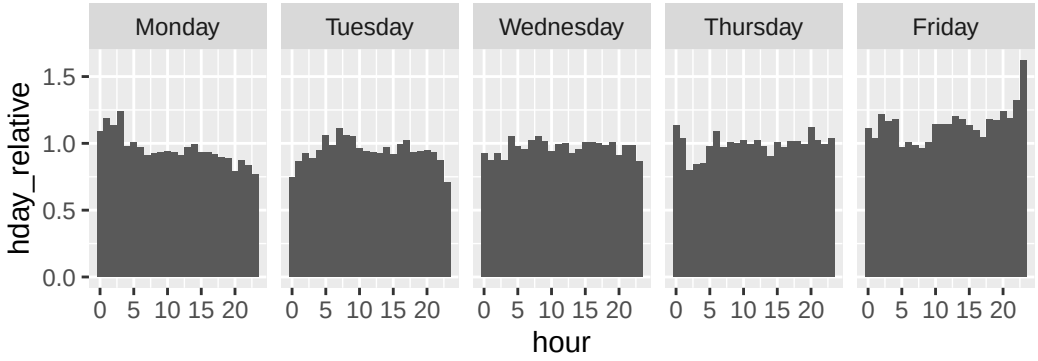


Figure 6.5: Facetted time series showing relative number of crashes per hour by day in the working week.

The results clearly show that Friday is a dangerous day as many of the columns are above 1 (*NB as this is a relative calculation, columns that are less than 1 indicate that there are less crashes per hour on that day than average whereas those above 1 indicate that there are more crashes per hour on that day than average*). The extent to which the high relative number of crashes in the most anomalous hours (Friday evening) is due increased exposure vs increased risk per km travelled cannot be ascertained by this plot but it certainly suggests that Friday afternoon and evening is a worthy focus of road safety research.

Exercises:

1. Building on the code above, show the absolute and relative number of crashes per hour on Saturday and Sunday.
2. Filter the dataset so it contains only data from two police forces of your choice (e.g. `West Yorkshire` and `Metropolitan Police`).
3. Try creating plots similar to those shown above but facetted by police force rather than by day of the week.

6.2 Handling dates and date-times

It is worth remembering that base R already has decent support for dates and `datetimes`, although the base R functions are not particularly intuitive. This is shown in the code chunk below, which creates objects representing the date and time of a fictitious crash event on a cold winter's morning, 1st January 2021, and a subsequent road safety intervention on the 20th October 2021:

```
crash_datetime_character = "2021-01-01 08:35" # creates date/time as a character
crash_datetime = as.POSIXct(crash_datetime_character) # converts date/time to a object of the
class(crash_datetime)
```

```
[1] "POSIXct" "POSIXt"
```

```
intervention_date_character = "2021-10-20"
intervention_date = as.Date(intervention_date_character) # converts date/time to a object of
class(intervention_date)
```

```
[1] "Date"
```

```
# see ?as.POSIXct for more examples
```

'POSIXct', 'POSIXt' and 'Date' are data types for dates and time that enable easy manipulation of such data. Fortunately for most users, there are easier ways to work with time series data, starting with the `hms` package.

6.3 Hours, minutes seconds with hms

The `hms` library in the `tidyverse` can be used to process hours, minutes and seconds, as shown below. See a very basic demo of the package and links to the package's help pages with the following commands in which we use the package without loading it with the `library()` function, demonstrating the `package::function()` syntax taught in Section 4:

```
library(tidyverse)
```

```
crash_time_character = "08:35:00"
crash_time_hms = hms::as_hms(crash_time_character)
class(crash_time_hms)
```

```
[1] "hms"      "difftime"
```

```
?hms::`hms-package`
```

As the package's name suggests, it is used for dealing with hours, minutes and seconds. It can round time objects of class `hms` to the nearest second (or any multiple of a second):

```
hms::round_hms(crash_time_hms, 1)           # time to the nearest second
```

```
08:35:00
```

```
hms::round_hms(crash_time_hms, 1 * 60 * 60) # time to the nearest hour
```

```
09:00:00
```

```
hms::round_hms(crash_time_hms, 1 * 30 * 60) # time to the nearest half hour
```

```
08:30:00
```

It can also convert simple text strings into time objects, e.g. as follows (**Note:** we do not need to include the `:00`):

```
hms::parse_hm("08:35")
```

```
08:35:00
```

6.4 The lubridate package

In many cases the most useful and easy to use package when working with temporal data is `lubridate`. Having installed it, load it as follows:

```
library(lubridate)
```

The simplest example of a Date object that we can analyze is just the current date, i.e.:

```
today()
```

```
[1] "2025-08-30"
```

We can manipulate this object using several `lubridate` functions to extract the current day, month, year, weekday and so on…

```
x = today()
day(x)
wday(x)
wday(x) %in% c(1, 6) # is it the weekend?
month(x)
year(x)
```

Base R can also be used to extract data e.g. `# Base R function to get the day of week weekdays(x)`.

Exercises:

1. Look at the help page of the `lubridate` function `month` to see how it is possible to extract the current month as a character vector.
2. Look at other functions in `lubridate` to extract the current weekday as a number, the week of year and the day of the year.

Date variables are often stored simply as character vectors. This is a problem, since R is not always smart enough to distinguish between character vectors representing Dates. `lubridate` provides functions that can translate a wide range of date encodings such as `ymd()`, which extracts the Year, Month and Day from a character string, as demonstrated below.

```
as.Date("2021-10-17") # works
as.Date("2021 10 17") # fails
ymd("2021 10 17")     # works
dmy("17/10/2021")     # works
```

Import functions, such as `read_csv`, try to recognize the Date variables. Sometimes this fails. You can manually create Date objects, as shown below:

```
x = c("2009-01-01", "2009-02-02", "2009-03-03")
x_date = ymd(x)
x_date
```

```
[1] "2009-01-01" "2009-02-02" "2009-03-03"
```

Exercises:

1. Extract the day, the year-day, the month and the weekday (as a non-abbreviated character vector) of each element of `x_date`.
2. Convert `"09/09/93"` into a date object and extract its weekday.
3. **Bonus:** Read the help page of `as.Date` and `strptime` for further details on base R functions for dates.
4. **Bonus:** Read the Chapter 16 of R for Data Science book for further details on `lubridate` package.

6.5 Dates in a data frame

We can use Dates for subsetting events in a dataframe. For example, if we define `x_date` as before and add it to the `crash` dataset, i.e.:

```
crashes$casualty_day = x_date
```

Then we can subset events using Dates. For example:

```
filter(crashes, day(casualty_day) < 7) # the events that ocurred in the first week of the mo
```

```
# A tibble: 3 x 5
  casualty_type casualty_age vehicle_type dark  casualty_day
  <chr>                <dbl> <chr>        <lgl> <date>
1 pedestrian              20 car          TRUE  2009-01-01
2 cyclist                 40 bus          FALSE 2009-02-02
3 cat                     60 tank         TRUE  2009-03-03
```

```
filter(crashes, weekdays(casualty_day) == "Monday") # the events occurred on monday
```

```
# A tibble: 1 x 5
  casualty_type casualty_age vehicle_type dark  casualty_day
  <chr>                <dbl> <chr>        <lgl> <date>
1 cyclist                 40 bus          FALSE 2009-02-02
```

Exercises:

1. Select only the events (rows in `crashes`) that occurred in January.
2. Select only the events that ocurred in an odd year-day.
3. Select only the events that ocurred in a leap-year (**Hint:** check the function `leap_year`).
4. Select only the events that ocurred during the weekend or in June.
5. Select only the events that ocurred during the weekend and in June.
6. Count how many events ocurred during each day of the week.

6.6 Components of time objects

Now we'll take a look at the time components of a Date. Using the function `hms` (acronym for Hour, Minutes, Seconds) and its subfunctions such as `hm` or `ms`, we can parse a character vector representing several times into an Hour object (which is technically called a 'period object').

```
x = c("18:23:35", "00:00:01", "12:34:56")
x_hour = hms(x)
x_hour
```

```
[1] "18H 23M 35S" "1S"          "12H 34M 56S"
```

We can manipulate these objects using several `lubridate` functions to extract the hour component, the minutes, and so on:

```
hour(x_hour)
```

```
[1] 18  0 12
```

```
minute(x_hour)
```

```
[1] 23  0 34
```

```
second(x_hour)
```

```
[1] 35  1 56
```

If the Hour data does not specify the seconds, we just use a subfunction of `hms`, namely `hm`, to get the hours and minutes, rather than hours, minutes and seconds.

```
x = c("18:23", "00:00", "12:34")
(x_hour = hm(x))
```

```
[1] "18H 23M 0S" "0S"         "12H 34M 0S"
```

We can use Hour data also for subsetting events, like we did for Dates. Let's add a new column for hour to the crashes data:

```
crashes$casualty_hms = hms(c("18:23:35", "00:00:01", "12:34:56"))
crashes$casualty_hour = hour(crashes$casualty_hms)
```

Exercises:

1. Filter only the events that occurred after midday (i.e. the PM events). **Hint:** your answer may include `>= 12`.
2. Filter only the events that ocurred between 15:00 and 19:00.

7 Spatial data

From displaying simple point data to examining collision density along routes or between areas, the geographic property of STATS19 data is one of its most useful attributes. Mapping is a hugely useful and powerful aspect of R and has many applications in road safety, both in understanding geographic trends and presenting insight to colleagues. This aspect of R is covered in detail in the book Geocomputation With R. By mapping collision data in R, you can add layers containing other geographic datasets to further understand the reasons for certain trends. This can lead to new opportunities for intervention and collaboration with other parties who may have mutually compatible solutions for reaching their goals. We will use the following packages in this section:

```
library(sf)        # load the sf package for working with spatial data
library(tidyverse) # load the tidyverse as before
```

7.1 sf objects

All road crashes happen somewhere and, in the UK at least, all collisions recorded by the police are given geographic coordinates. These can help in prioritising interventions to save lives by focusing on 'crash hotspots.' R has strong geographic data capabilities with the `sf` package, providing a generic class for spatial vector data. Points, lines and polygons are represented in `sf` as objects in a special 'geometry column', typically called 'geom' or 'geometry', extending the data frame class we've already seen in `crashes`, created in Section 5 (repeated here to consolidate data frame creation):

```
crashes = tibble(
  casualty_type = c("pedestrian", "cyclist", "cat"),
  casualty_age = seq(from = 20, to = 60, by = 20),
  vehicle_type = c("car", "bus", "tank"),
  dark = c(TRUE, FALSE, TRUE)
)
```

Create an `sf` data frame called `crashes_sf` that expands the `crashes` data frame to include a geometry column based on the `crashes` longitude and latitude data as follows:

```
crashes_sf = crashes # create copy of crashes dataset
crashes_sf$longitude = c(-1.3, -1.2, -1.1)
crashes_sf$latitude = c(50.7, 50.7, 50.68)
crashes_sf = st_as_sf(crashes_sf, coords = c("longitude", "latitude"), crs = 4326) # st_as_s
```

Exercises:

1. Plot only the geometry column of `crashes_sf` (**Hint:** the solution may contain `$geometry`). If the result is like that in Figure 7.1, congratulations, it worked!
2. Plot `crashes_sf`, only showing the age variable.
3. Plot the 2nd and 3rd crashes, showing which happened in the dark.
4. **Bonus:** How far apart are the points? (**Hint:** `sf` functions begin with `st_`)
5. **Bonus:** Near which settlement did the tank runover the cat?

```
plot(crashes_sf$geometry)
plot(crashes_sf["casualty_age"])
plot(crashes_sf[2:3, "dark"])
```

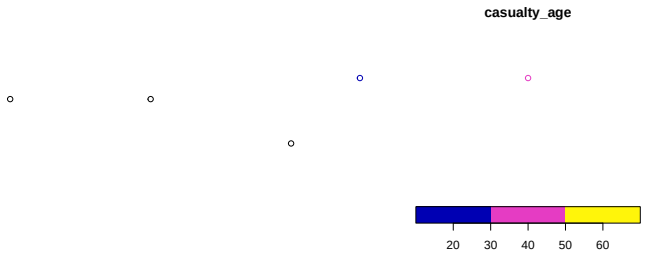


Figure 7.1: The `crashes_sf` dataset shown in map form with the function `plot()`.

Figure 7.2: The `crashes_sf` dataset shown in map form with the function `plot()`.

Figure 7.3: The `crashes_sf` dataset shown in map form with the function `plot()`.

7.2 Reading and writing spatial data

You can read and write spatial data with `read_sf()` and `write_sf()`, as shown below (see `?read_sf`):

First, let's create the `zones` object that we'll use for demonstration:

```
zones = pct::get_pct_zones("isle-of-wight")[1:9]
```

Now we can read and write spatial data:

```
write_sf(zones, "zones.geojson") # saves the spatial dataset called zones as geojson file ty
write_sf(zones, "zmapinfo", driver = "MapInfo file") # saves the dataset as a MapInfo file
read_sf("zmapinfo") # reads-in mapinfo file
```

See Chapter 6 of *Geocomputation with R* for further information.

7.3 sf polygons

`sf` objects can also represent administrative zones. This is illustrated below with reference to `zones`, a spatial object representing the Isle of Wight, that we created in the previous section.
Exercises:

1. What is the class of the `zones` object?
2. What are its column names?
3. Print its first 2 rows and columns 6:8 (the result is below).

```
Simple feature collection with 2 features and 5 fields
Geometry type: MULTIPOLYGON
Dimension:     XY
Bounding box:  xmin: -1.301131 ymin: 50.69052 xmax: -1.28837 ymax: 50.70547
Geodetic CRS:  WGS 84
# A tibble: 2 x 6
  geo_code    all bicycle  foot car_driver                              geometry
  <chr>     <dbl>   <dbl> <dbl>      <dbl>                      <MULTIPOLYGON [°]>
1 E01017326   698      23   285        286 (((-1.289993 50.69766, -1.290177 50.~
2 E01017327   720      25   225        374 (((-1.295712 50.69383, -1.29873 50.6~
```

7.4 Spatial subsetting and sf plotting

Like index and value subsetting, spatial subsetting can be done with the `[]` notation. We can identify the crashes (`crashes_sf`) that occur in the Isle of Wight (`zones`) by subsetting as follows (i.e. subset `zones` by whether it contains data in `crashes_sf`):

```
zones_containing_crashes = zones[crashes_sf, ]
```

To plot a new layer on top of an existing `sf` plot, use the `add = TRUE` argument, e.g. as follows:

```
plot(zones$geometry) # plot just the geometry of one layer
plot(zones_containing_crashes$geometry, col = "grey", add = TRUE)
```

Remember to plot only the `geometry` column of objects to avoid multiple maps. Colours can be set with the `col` argument.

Exercises:

1. Plot the geometry of the zones, with the zones containing crashes overlaid on top in red (see Figure 7.4).
2. Plot the zone containing the 2nd crash in blue (see Figure 7.5).
3. **Bonus:** Plot all zones that intersect with a zone containing crashes, with the actual crash points plotted in black (see Figure 7.6).

```
plot(zones$geometry)
plot(zones_containing_crashes$geometry, col = "red", add = TRUE)
plot(zones$geometry)
plot(zones[crashes_sf[2, ], ], col = "blue", add = TRUE)
plot(zones$geometry)
plot(zones[zones_containing_crashes, ], col = "yellow", add = TRUE)
plot(crashes_sf$geometry, pch = 20, add = TRUE)
```

Figure 7.4: Illustration of the results of spatial subsetting.

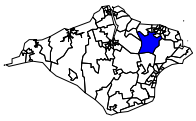


Figure 7.5: Illustration of the results of spatial subsetting.

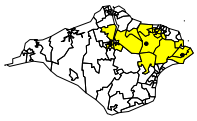


Figure 7.6: Illustration of the results of spatial subsetting.

7.5 Geographic joins

Geographic joins involve assigning values from one object to a new column in another, based on the geographic relationship between them. With `sf` objects, the data from the `crashes_sf` dataset is joined onto the 'target' `zones` dataset, to create a new object called `zones_joined`:

```
zones_joined = st_join(zones[1], crashes_sf)
```

The above code takes the `geo_code` column data from `zones`, matches it to the `geometry` column in `crashes_sf` and then joins it to the crashes that have occurred in those geo_codes. The matched, joined `geo_code` is a new column in the `zone_joined` dataset. We now know the administrative geo_code in which each crash occured.

Exercises:

1. Plot the `casualty_age` variable of the new `zones_joined` object (see Figure 7.7, to verify the result).
2. How many zones are returned in the previous command?
3. Select only the `geo_code` column from the `zones` and the `dark` column from `crashes_sf` and use the `left = FALSE` argument to return only zones in which crashes occurred. Plot the result. (**Hint:** it should look like the map shown in Figure 7.8)

See Chapter 4 of *Geocomputation with R* for further information on geographic joins.

```
plot(zones_joined["casualty_age"])
zjd = st_join(zones[1], crashes_sf["dark"], left = FALSE)
plot(zjd)
```

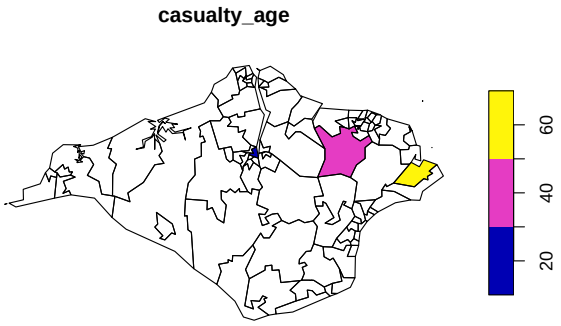


Figure 7.7: Illustration of geographic joins.

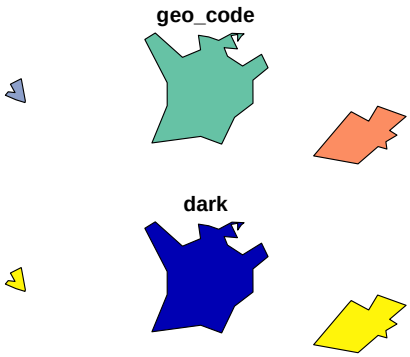


Figure 7.8: Illustration of geographic joins.

7.6 Coordinate Reference Systems

A Coordinate Reference Systems (CRS) is used for plotting data on maps. There are many systems in use but they can generally be classified into two groups; 'projected' and 'geographic'. A projected system, such as Eastings/Northings, plots locations onto a flat 2D projection of the Earth's surface. A geographic system, such as Longitude/Latitude, refers to locations on the 3D surface of the globe. Distance and direction calculations work differently between geographic and projected CRSs, so it is often necessary to convert from one to another. Fortunately, R makes this very easy, and every CRS has its own unique reference number. For example, 27700 for the British National Grid system.

CRSs define how two-dimensional points (such as longitude and latitude) are actually represented in the real world. A CRS value is needed to interpret and give true meaning to coordinates. You can get and set CRSs with the command `st_crs()`. Transform CRSs with the command `st_transform()`, as demonstrated in the code chunk below, which converts the 'lon/lat' geographic CRS of `crashes_sf` into the projected CRS of the British National Grid:

```
crashes_osgb = st_transform(crashes_sf, 27700)
```

Exercises:

1. Try to subset the zones with the `crashes_osgb`. What does the error message say?
2. Create `zones_osgb` by transforming the `zones` object.
3. **Bonus:** Use `st_crs()` to find out the units measurement of the British National Grid.

For more information on CRSs see Chapter 6 of *Geocompuation with R*.

7.7 Buffers

Buffers are polygons surrounding geometries, usually with fixed distance. For example, in road safety research a 30m buffer can be created around crash locations to identify crashes that happened in close proximity to a particular junction or road segment.

Exercises:

1. Find out and read the help page of `sf`'s buffer function.
2. Create an object called `crashes_1km_buffer` representing the area within 1 km of the `crashes_osgb` object and plot the result using the command: `plot(crashes_1km_buffer)`. As a fun bonus, try: `mapview::mapview(crashes_1km_buffer)`.
3. **Bonus:** Try creating buffers on the geographic version of the `crashes_sf` object. What happens?

7.8 Attribute operations on sf objects

We can do non-spatial operations on `sf` objects because they are `data.frames`. Try the following attribute operations on the `zones` data:

```
# use the zones dataset we created earlier
sel = zones$all > 3000  # create a subsetting object
zones_large = zones[sel, ] # subset areas with a population over 3,000
zones_2 = zones[zones$geo_name == "Isle of Wight 002",] # subset based on 'equality' query
zones_first_and_third_column = zones[c(1, 3)]
zones_just_all = zones["all"]
```

Exercises:

1. Practice the subsetting techniques you have learned on the `sf data.frame` object `zones`:
    - Create an object called `zones_small`, which contains only regions with less than 3000 people in the `all` column.
    - Create a selection object called `sel_high_car` which is `TRUE` for regions with above median numbers of people who travel by car and `FALSE` otherwise.
    - Create an object called `zones_foot` which contains only the foot attribute from `zones`.
    - **Bonus 1:** plot `zones_foot` using the function `plot` to show where walking is a popular mode of travel to work.
    - **Bonus 2:** building on your answers to previous questions, use `filter()` from the `dplyr` package to subset small regions where car use is high.
2. **Bonus:** What is the population density of each region (**Hint:** you may need to use the functions `st_area()`, `as.numeric()` and use the 'all' column)?
3. **Bonus:** Which zone has the highest percentage of people who cycle?

7.9 Mapping road crash data

So far we have used the `plot()` function to make maps. That's fine for basic visualisation, but for publication-quality maps we recommend using `tmap`. See Chapter 8 of *Geocomputation with R* for further explanation and alternatives. After installation, load the package as follows:

```
library(tmap)
tmap_mode("plot") # this sets the tmap mode to plotting as opposed to interactive
```

```
i tmap mode set to "plot".
```

Exercises:

1. Create the plots of the `zones` object using `plot()` and `tm_shape() + tm_polygons()` functions (see Figure 7.9).
2. Create an interactive version of the `tmap` plot by setting `tmap_mode("view")` and re-running the plotting commands.
3. Add an additional layer to the interactive map showing the location of crashes, using marker and dot symbols.
4. **Bonus:** Change the default basemap (**Hint:** you may need to search in the package documentation or online for the solution).

```
plot(zones[c("all", "bicycle")])
tm_shape(zones) +
  tm_polygons(c("all", "bicycle"))
```

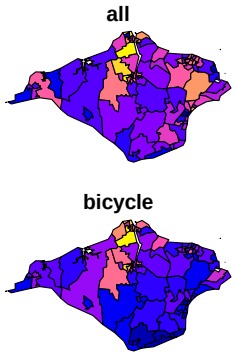


Figure 7.9: Illustration of the plot and tmap approaches for creating maps.

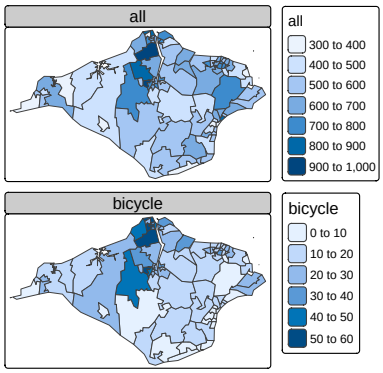


Figure 7.10: Illustration of the plot and tmap approaches for creating maps.

7.10 Analysing point data

Based on the saying, "Don't run before you can walk," we've learned the vital foundations of R before tackling a real dataset. Temporal and spatial attributes are key to road crash data, hence the emphasis on `lubridate` and `sf`. Visualisation is central to understanding data and influencing policy, which is where `tmap` comes in. With these solid foundations, plus knowledge of how to ask for help (we recommend reading R's internal help, asking colleagues, searching the internet and creating new questions or comments on online forums such as StackOverflow or GitHub and we suggest you follow this order of resources to get help), you are ready to test the methods on some real data.

Before doing so, take a read of the `stats19` vignette, which can be launched as follows:

```
vignette(package = "stats19") # view all vignettes available on stats19
vignette("stats19") # view the introductory vignette
```

This should now be sufficient to tackle the following exercises:

1. Download and plot all crashes reported in Great Britain in 2023. (**Hint:** see the stats19 vignette)
2. Find the function in the `stats19` package that converts a `data.frame` object into an `sf` data frame. Use this function to convert the road crashes into an `sf` object, called `crashes_sf`, for example.
3. Filter crashes that happened in the Isle of Wight based on attribute data. (**Hint:** the relevant column contains the word `local`)
4. Filter crashes happened in the Isle of Wight using geographic subsetting. (**Hint:** remember `st_crs()`?)
5. **Bonus:** Which type of subsetting yielded more results and why?
6. **Bonus:** How many crashes happened in each zone?
7. Create a new column called `month` in the crash data using the function `lubridate::month()` and the `date` column.
8. Create an object called `a_zones_may` representing all the crashes that happened in the Isle of Wight in the month of May.
9. **Bonus:** Calculate the average (`mean`) speed limit associated with each crash that happened in May across the zones of the Isle of Wight (the result is shown in the map).

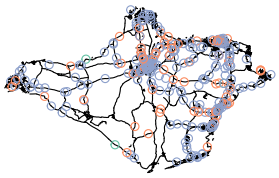


Figure 7.11: Maps of the Isle of Wight.

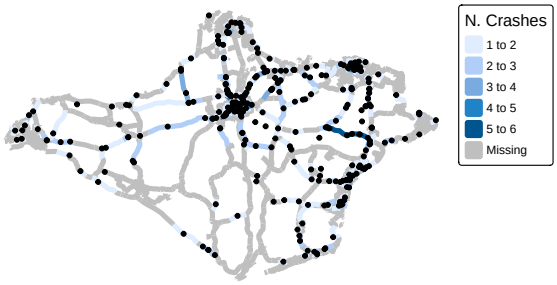


Figure 7.12: Maps of the Isle of Wight.

```
remotes::install_github("itsleeds/osmextract") # install github package for osm data
library(osmextract)
region_name = "essex" # "essex" can be changed to another area name as required
```

```r
osm_data = oe_get(region_name, extra_tags = c("maxspeed", "ref"))
table(osm_data$highway)
#filter osm_data to show only major roads
roads = osm_data %>%
  filter(str_detect(highway, pattern = "moto|prim|seco|tert|trunk"))
# transform geometry and save
roads = st_transform(roads, 27700) #converts to projected BNG system for later use
# plot(roads$geometry) # basic plot of roads
tm_shape(roads) + tm_lines("maxspeed", showNA = T, lwd = 2)
saveRDS(roads, file = "roads.Rds") # Saves road dataset for future use
```

```
-- tmap v3 code detected -------------------------------------------------------

[v3->v4] `tm_lines()`: migrate the argument(s) related to the legend of the
visual variable `col` namely 'showNA' to 'col.legend = tm_legend(<HERE>)'
```

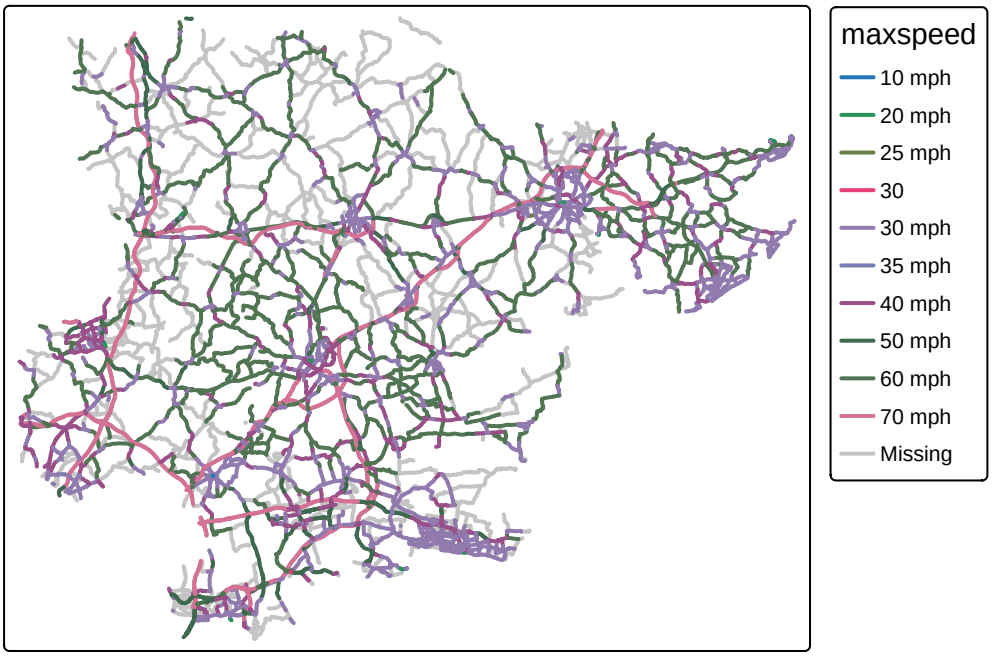


Figure 7.13: Roads in Essex downloaded with the code shown above.

Bonus exercises

Identify a region and zonal units of interest from http://geoportal.statistics.gov.uk/ or from the object `police_boundaries` in the `stats19` package.

1. Read them into R as an `sf` object.
2. Create a map showing the number of crashes in each zone.
3. Identify the average speed limit associated with crashes in each zone.
4. Identify an interesting question you can ask to the data and use exploratory data analysis to find answers.
5. Check another related project for further information on smoothing techniques of counts on a linear network.

8 Joining road crash tables

8.1 STATS19 tables

Thus far, we have been working primarily with 'accident' level data, but there is much useful data in other tables. As outlined in the `stats19` vignette — which you can view by entering the command `vignette("stats19")` to get extended help pages about R packages — there are three main tables that contain STATS19 data.

Let's read-in data from 2023 to take a look:

```
library(stats19)
library(dplyr)
library(ggplot2)
ac = get_stats19(year = 2023, type = "acciden")
ca = get_stats19(year = 2023, type = "casualt")
```

```
Warning: The following named parsers don't match the column names:
accident_severity, carriageway_hazards, collision_index, collision_reference,
collision_year, date, day_of_week, did_police_officer_attend_scene_of_accident,
did_police_officer_attend_scene_of_collision, enhanced_collision_severity,
first_road_class, first_road_number, junction_control, junction_detail,
latitude, legacy_collision_severity, light_conditions,
local_authority_district, local_authority_highway,
local_authority_ons_district, location_easting_osgr, location_northing_osgr,
longitude, lsoa_of_accident_location, lsoa_of_collision_location,
number_of_casualties, number_of_vehicles, pedestrian_crossing_human_control,
pedestrian_crossing_physical_facilities, police_force, road_surface_conditions,
road_type, second_road_class, second_road_number, special_conditions_at_site,
speed_limit, time, trunk_road_flag, urban_or_rural_area, weather_conditions,
adjusted_serious, adjusted_slight, injury_based, accident_ref_no,
effective_date_of_change, previously_published_value, replacement_value,
variable, age_band_of_driver, age_of_driver, age_of_vehicle, dir_from_e,
dir_from_n, dir_to_e, dir_to_n, driver_distance_banding, driver_home_area_type,
driver_imd_decile, engine_capacity_cc, escooter_flag, first_point_of_impact,
generic_make_model, hit_object_in_carriageway, hit_object_off_carriageway,
```

```
journey_purpose_of_driver, junction_location, lsoa_of_driver, propulsion_code,
sex_of_driver, skidding_and_overturning, towing_and_articulation,
vehicle_direction_from, vehicle_direction_to, vehicle_leaving_carriageway,
vehicle_left_hand_drive, vehicle_location_restricted_lane, vehicle_manoeuvre,
vehicle_type

Warning in asMethod(object): NAs introduced by coercion
```

```
ve = get_stats19(year = 2023, type = "vehicle")
```

```
Warning: The following named parsers don't match the column names: accident_severity, carriag
NAs introduced by coercion
NAs introduced by coercion
```

The three objects read-in above correspond to the main types of entity that are recorded by the police:

- Crashes: The 'crash event' table contains general data about crashes, including where and when they happened and the conditions in which the crash occurred (e.g. light levels in the column `light_conditions` in the `ac` object). For historical reasons, crash level data is stored in tables called 'Accidents' (a term that has fallen out of favour because it implies that nobody was at fault). See names for all 33 variables in the crashes table by running the command `names(ac)`. Crashes range from collisions involving only one vehicle and another entity (e.g. a person on foot, bicycle or a car) causing only 'slight' injuries such as a graze, to multi-vehicle pile-ups involving multiple deaths and dozens of slight and serious injuries.
- Casualties: The casualties table, assigned to an object called `ca` in the code above, contains data at the casualty level. As you will see by running the command `names(ca)`, the STATS19 casualties table has 16 variables including `age_of_casualty`, `casualty_severity` and `casualty_type`, reporting the mode of transport in which the person was travelling when they were hit.
- Vehicles: The vehicles table, assigned to `ve` above, contains information about the vehicles and their drivers involved in each collision. As you will see by running the command `names(ve)`, the 23 variables in this table includes `vehicle_type`, `hit_object_off_carriageway` and `first_point_of_impact`. Information about the driver of vehicles involved is contained in variables such as `age_of_driver`, `engine_capacity_cc` and `age_of_vehicle`.

Each table represents the same phenomena: road casualties in Great Britain in 2023. Therefore, you may expect they would have the same number of rows, but this is not the case:

```
nrow(ac)
```

```
[1] 104258
```

```
nrow(ca)
```

```
[1] 132977
```

```
nrow(ve)
```

```
[1] 189815
```

The reason for this is that there are, on average, more than one casualty per crash (e.g. when a car hits two people), and more than one vehicle, including bicycles, per crash. We can find the average number of casualties and vehicles per crash as follows:

```
nrow(ca) / nrow(ac)
```

```
[1] 1.275461
```

```
nrow(ve) / nrow(ac)
```

```
[1] 1.820628
```

The output of the commands above show that there are around 1.3 casualties and 1.8 vehicles involved in each crash record in the STATS19 dataset for 2023. Each table contains a different number of columns, reporting the characteristics of each casualty and each driver/vehicle for the `ca` and `ve` datasets respectively.

```
ncol(ac)
```

```
[1] 38
```

```
ncol(ca)
```

```
[1] 21
```

```
ncol(ve)
```

```
[1] 34
```

The output of the previous code chunk shows that we have more variables in the 'accidents' table than the others but the others, but the other tables are data rich with 16 columns on the casualties and 23 on the vehicles. To check that the datasets are consistent, we can check that the number of casualties reported in the crashes table is equal to the number of rows in the casualties table, and the same for the vehicles table:

```
ac$number_of_casualties = as.numeric(ac$number_of_casualties)
ac$number_of_vehicles = as.numeric(ac$number_of_vehicles)
sum(ac$number_of_casualties) == nrow(ca)
```

```
[1] TRUE
```

```
sum(ac$number_of_vehicles) == nrow(ve)
```

```
[1] TRUE
```

8.2 Joining casualty data

To join casualty (or vehicle) data onto the `ac` object above, the `inner_join()` function from `dplyr` can be used as follows:

```
ac_cas_joined = inner_join(ac, ca)
```

```
Joining with `by = join_by(accident_index, accident_year, accident_reference)`
```

The above command worked because the two datasets have a shared variable name: `accident_index`. Note that the command worked by duplicating accident records for multiple casualties. We can see this finding the accident that had the most crashes and printing the results in the `ac` and new joined dataset, as follows:

```
id_with_most_crashes = ac %>%
  top_n(n = 1, wt = number_of_casualties) %>%
  pull(accident_index)
id_with_most_crashes
```

```
[1] "2023520300610"
```

```
ac %>% filter(accident_index == id_with_most_crashes) %>%
  select(accident_index, accident_severity, number_of_vehicles, number_of_casualties)
```

```
# A tibble: 1 x 4
  accident_index accident_severity number_of_vehicles number_of_casualties
  <chr>          <chr>                          <dbl>                <dbl>
1 2023520300610  Serious                            3                   70
```

```
ac_cas_joined %>% filter(accident_index == id_with_most_crashes) %>%
  select(accident_index, accident_severity, number_of_vehicles, number_of_casualties, casualt
```

```
# A tibble: 70 x 5
   accident_index accident_severity number_of_vehicles number_of_casualties
   <chr>          <chr>                          <dbl>                <dbl>
 1 2023520300610  Serious                            3                   70
 2 2023520300610  Serious                            3                   70
 3 2023520300610  Serious                            3                   70
 4 2023520300610  Serious                            3                   70
 5 2023520300610  Serious                            3                   70
 6 2023520300610  Serious                            3                   70
 7 2023520300610  Serious                            3                   70
 8 2023520300610  Serious                            3                   70
 9 2023520300610  Serious                            3                   70
10 2023520300610  Serious                            3                   70
# i 60 more rows
# i 1 more variable: casualty_reference <chr>
```

8.3 Joining vehicle data

The same approach can be used to join vehicle data onto the crash record data:

```
ac_veh_joined = inner_join(ac, ve)
```

```
Joining with `by = join_by(accident_index, accident_year, accident_reference)`
```

This information can be used as the basis of who-hit-who visualisation, in this case looking at vehicles involved in the most common type of casualties (see the `trafficalmr` package for functions to recode stats19 data):

```
casualty_types_df = ac_cas_joined |>
  count(casualty_type) |>
  arrange(desc(n))
# top_casualty_types
#  1 Car occupant                                              71145
#  2 Pedestrian                                                19263
#  3 Cyclist                                                   14999
#  4 Motorcycle 125cc and under rider or passenger              9115
#  5 Motorcycle over 500cc rider or passenger                   4385
#  6 Van / Goods vehicle (3.5 tonnes mgw or under) occupant     3815
#  7 Bus or coach occupant (17 or more pass seats)              2399
#  8 Motorcycle over 125cc and up to 500cc rider or  passenger  1924
#  9 Other vehicle occupant                                     1531
# 10 Taxi/Private hire car occupant                             1530
recode_casualties = function(x) {
  x_updated = dplyr::case_when(
    stringr::str_detect(x, "Moto") ~ "Motorcyclist",
    stringr::str_detect(x, "Car|Pedestrian|Cyclist") ~ x,
    TRUE ~ "Other"
  )
}
ac_cas_joined$cas_type = recode_casualties(ac_cas_joined$casualty_type)
barplot(table(ac_cas_joined$cas_type))
```

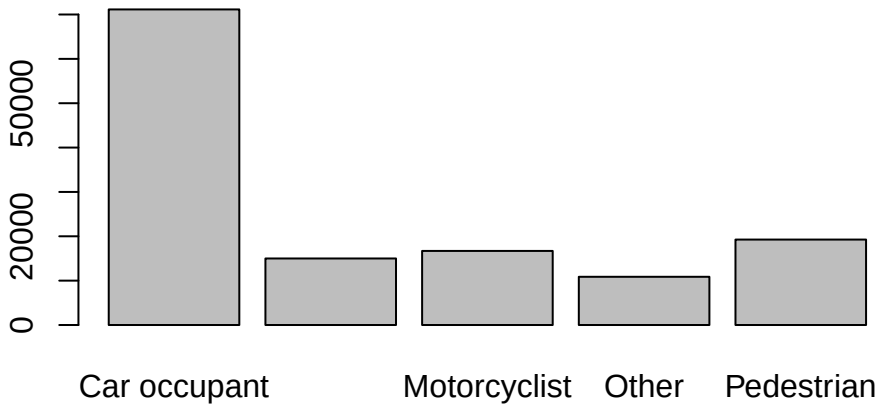


Figure 8.1: Barplot of recoded casualty type frequencies

To find the largest vehicle involved in each casualty, we can similarly pre-process the vehicle data as follows:

```
vehicle_types_df = ac_veh_joined |>
  count(vehicle_type) |>
  arrange(desc(n))
#  1 Car                                  130164
#  2 Pedal cycle                           15667
#  3 Van / Goods 3.5 tonnes mgw or under   11727
#  4 Motorcycle 125cc and under             9710
#  5 Motorcycle over 500cc                  4473
#  6 Bus or coach (17 or more pass seats)   3171
#  7 Taxi/Private hire car                  3073
#  8 Goods 7.5 tonnes mgw and over          2647
#  9 Other vehicle                          2200
# 10 Motorcycle over 125cc and up to 500cc  1975
recode_vehicles = function(x) {
  x_updated = dplyr::case_when(
    stringr::str_detect(x, "Pedal") ~ "Bicycle",
    stringr::str_detect(x, "Car") ~ "Car",
    stringr::str_detect(x, "Van") ~ "Van",
```

```
    stringr::str_detect(x, "HGV|over") ~ "HGV",
    TRUE ~ "Other"
  )
  return(x_updated)
}
ac_veh_joined$veh_type = recode_vehicles(ac_veh_joined$vehicle_type)
levels = c("Bicycle", "Car", "Other", "Van", "HGV")
ac_veh_joined$vehicle = factor(ac_veh_joined$veh_type, levels = levels, ordered = TRUE)
summary(ac_veh_joined$vehicle)
```

```
Bicycle     Car   Other     Van     HGV
  15667  130164   22371   11727    9886
```

```
ac_veh_largest = ac_veh_joined %>%
  group_by(accident_index) %>%
  summarise(largest_vehicle = max(vehicle))
```

```
ac_cas_veh_largest = inner_join(ac_cas_joined, ac_veh_largest)
```

```
Joining with `by = join_by(accident_index)`
```

```
cas_veh_table = table(ac_cas_veh_largest$cas_type, ac_cas_veh_largest$largest_vehicle)
cvt_df = as.data.frame(cas_veh_table)
ggplot(cvt_df) +
  geom_bar(aes(Var2, Freq, fill = Var1), stat = "identity") +
  scale_fill_discrete("Casualty type") +
  xlab("Largest vehicle involved") +
  ylab("Number of casualties")
```

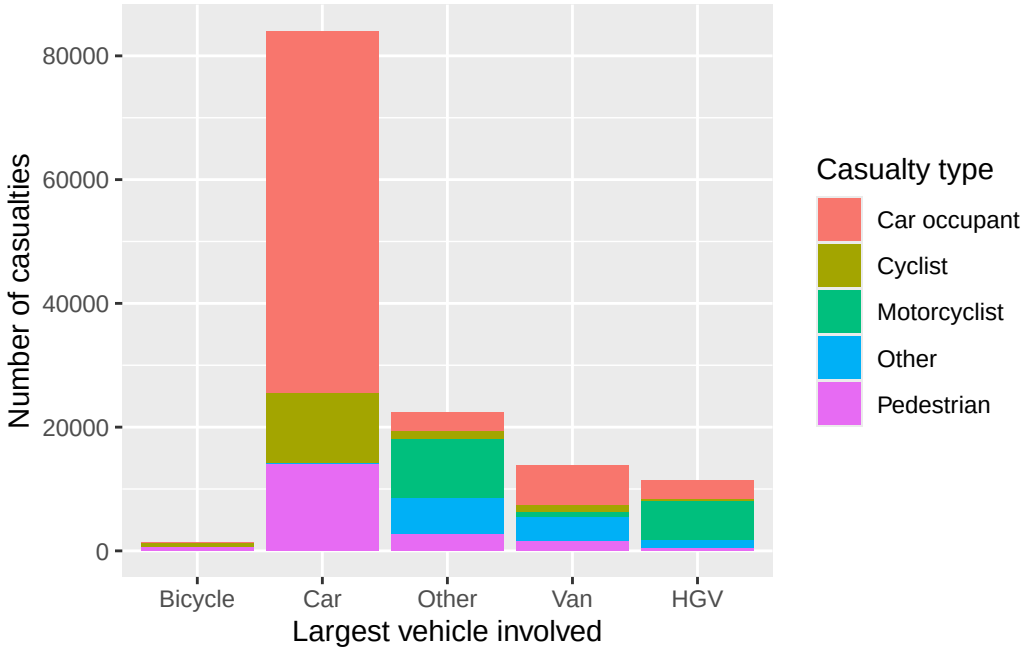


Figure 8.2: 'Who hit who' visualisation of number of casualties (y axis) hurt in crashes involving different vehicle types (largest vehicle in each crash on Y axis).

8.4 Case study: London

The three main tables we have just read-in can be joined by the `accident_index` variable and then filtered using other variables. This is demonstrated in the code chunk below, which subsets all casualties that took place in London, and counts the number of casualties by severity for each crash:

```
library(tidyr)
library(dplyr)
ac_sf = format_sf(ac)
lnd_police = c("City of London", "Metropolitan Police")
ac_lnd = ac_sf %>%
  filter(police_force %in% lnd_police)
ca_lnd = ca %>%
  filter(accident_index %in% ac_lnd$accident_index)
cas_types = ca_lnd %>%
  select(accident_index, casualty_type) %>%
  group_by(accident_index) %>%
  summarise(
```

```
    Total = n(),
    walking = sum(casualty_type == "Pedestrian"),
    cycling = sum(casualty_type == "Cyclist"),
    passenger = sum(casualty_type == "Car occupant")
    )
cj = left_join(ac_lnd, cas_types)
```

What just happened? We found the subset of casualties that took place in London with reference to the `accident_index` variable. Then we used the **dplyr** function, `summarise()`, to find the number of people who were in a car, cycling, and walking when they were injured. This new casualty dataset is joined onto the `crashes_lnd` dataset. The result is a spatial (`sf`) data frame of `ac` in London, with columns counting how many road users of different types were hurt. The joined data has additional variables:

```
base::setdiff(names(cj), names(ac_lnd))
```

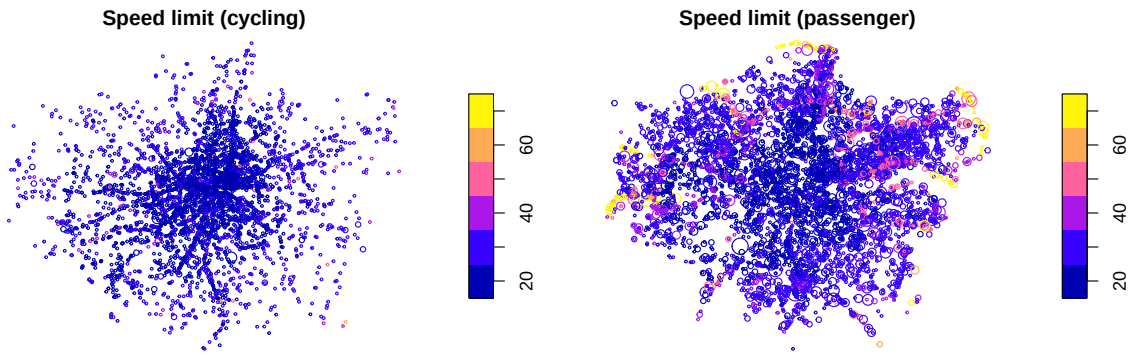
```
[1] "Total"     "walking"   "cycling"   "passenger"
```

As a simple spatial plot, we can map all crashes that occurred in London in 2023, with the colour related to the total number of people hurt in each crash. Placing this plot next to a map of London provides context:

```
cj$speed_limit = as.numeric(cj$speed_limit)
plot(
  cj[cj$cycling > 0, "speed_limit", ],
  cex = cj$Total[cj$cycling > 0] / 3,
  main = "Speed limit (cycling)"
  )
plot(
  cj[cj$passenger > 0, "speed_limit", ],
  cex = cj$Total[cj$passenger > 0] / 3,
  main = "Speed limit (passenger)"
  )
```

The spatial distribution of crashes in London clearly relates to the region's geography. Car crashes tend to happen on fast roads, including busy dual carriageway roads, displayed in yellow in Figure 8.2 above. Cycling is as an urban activity, and the most bike crashes can be found in or near the centre of London, which has a comparatively high level of cycling (compared to the low baseline of 3%).

In addition to the `Total` number of people hurt/killed, `cj` contains a column for each type of casualty (cyclist, car occupant, etc.), and a number corresponding to casualties in crashes



involving each type of vehicle. It also contains the `geometry` column from `ac_sf`. In other words, joins allow the casualties and vehicles tables to be geo-referenced. We can then explore the spatial distribution of different casualty types. For example, Figure 8.3 shows the spatial distribution of pedestrians and car passengers hurt in car crashes across London in 2023, via the following code:

```
library(ggplot2)
ac_types = cj %>%
  filter(accident_severity != "Slight") %>%
  mutate(type = case_when(
    walking > 0 ~ "Walking",
    cycling > 0 ~ "Cycling",
    passenger > 0 ~ "Passenger",
    TRUE ~ "Other"
  ))
ac_types$speed_limit = as.numeric(ac_types$speed_limit)
ggplot(ac_types, aes(size = Total, colour = speed_limit)) +
  geom_sf(show.legend = "point", alpha = 0.3) +
  facet_grid(vars(type), vars(accident_severity)) +
  scale_size(
    breaks = c(1:3, 12),
    labels = c(1:2, "3+", 12)
    ) +
  scale_color_gradientn(colours = c("blue", "yellow", "red")) +
  theme(axis.text = element_blank(), axis.ticks = element_blank())
```

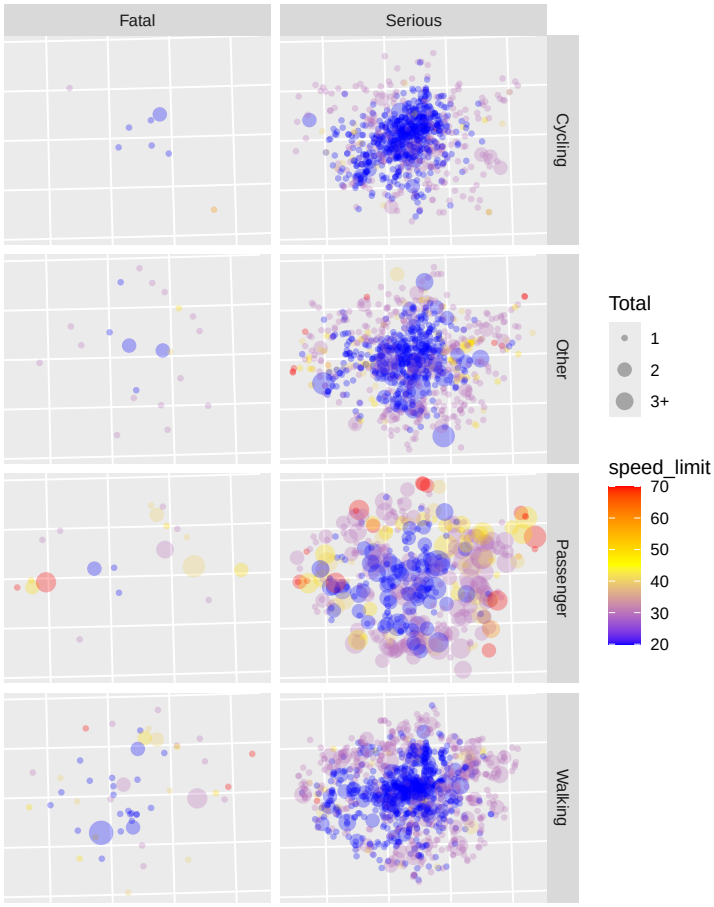


Figure 8.3: Spatial distribution of serious and fatal crashes in London, for cycling, walking, being a car passenger and other modes of travel. Colour is related to the speed limit where the crash happened (red is faster) and size is proportional to the total number of people hurt in each crash (legend not shown).

Exercises:

1. There is a lot going on in the code in this chapter, the most advanced of the guide. With reference to online help, work through the code line-by-line and look-up any aspects of the code that you do not fully understand to help figure out what is going on.
2. Reproduce the final figures for a different city of your choice (not London).
3. **Bonus:** Create more attractive interactive maps to show the spatial distribution of different casualty types in the city of your choice.

9 Next steps

You have reached the end of this short guide on reproducible road safety research with R. Armed with the knowledge of what R and RStudio can do, and how add-on packages provide powerful tools for a wide range of data analysis, visualisation and statistical modelling tasks, you should have a much better understanding of the language's capabilities.

I hope that in the process of working through the exercises, you have learned not only the technicalities of data science with a powerful tool of the trade, but also a way of working that puts reproducibility centre stage. Learning any new skill takes time and effort. However, in my experience, once you get past a critical threshold, the amount of time saved using the new approach starts to outweigh the amount of time involved in becoming fluent. The same concept applies to other 'tools of the trade' that are available, such as the open source geographic information system (GIS) software, QGIS and other languages for data science, such as Python and Julia.

Rather than go off and learn such additional tools, we encourage you to stick with R. It is preferable to know one language in-depth and then branch out to learn other approaches than to learn many approaches superficially or, as put it in the excellent R for Data Science (R4DS) book: "You will get better faster if you dive deep, rather than spreading yourself thinly over many topics". In terms of next steps, you cannot go wrong with checking-out the R4DS website which, like this book, has worked examples and exercises in abundance on a much wider range of data science topics. As you will see by visiting r4ds.had.co.nz many of these topics, including workflow and modelling, will be of use from a road safety research perspective.

A strength of R is its flexibility. It can be used as a calculator one minute and a statistical modelling toolbox the next. R can be used as a web application development framework the next, as illustrated by major shiny apps such as the Propensity to Cycle Tool (try it at www.pct.bike) and tools developed by road safety consultancy Agilysis, described in Section 9.1 below. Indeed, within the R ecosystem there are many sub-ecosystems, each of which has excellence free and open resources for people who want to learn more in a particular domain. If you are particularly in the geographic analysis of road crash data, the book *Geocomputation with R* by yours truly and which has already been mentioned in Chapter 7, is highly recommended. If you are looking for methods of analysing trends and forecasting with time series data, is highly recommended. Indeed, there is a whole library's worth of open resources to be found on any area of data-driven research online, from web development and visualisation to text analysis. A recommended next step for learning more in regards to any of these areas is

the website bookdown.org, which links to books that can also be bought as physical items if you, like many people, prefer learning with a paper resource.

In fact, with the size and rapidly evolving nature of the R ecosystem, one of the hardest things for a beginner is knowing which packages, functions or workflow options to choose from out of a wide array of options. The internet is there to help you, but it can also hinder your progress by serving-up out-of-date solutions and providing 'quick fixes' at the expense of a deep understanding. Therefore, instead of trying to be comprehensive (focussed web searches prioritising tried-and-tested solutions documented in authoritative sources can help with that), the rest of this final section provides pointers on a few particularly useful aspects of R from a road safety analysis perspective. Most people who learn R (or any computer language) will at become frustrated due to tricky-to-fix error messages. As outlined in Section 9.1, written by people who have navigated R's at first daunting learning curve, it can take only a few weeks of learning to get to the point where saves more time than it consumes and takes your work "to the next level".

9.1 Automated reporting with RMarkdown

An advantage of R is that it has many packages dedicated to the communication and publication of results, vital for policy impact. Perhaps the most important important package with regards to the communication of results is `rmarkdown`. This is more of a framework than a package, providing a powerful system for generating reports, web pages and even books (this book was written with the bookdown package, which builds on`rmarkdown`).

Here is not the place to explain how to use RMarkdown and their associated .Rmd files. The framework is explained in detail in a free and open book. To get started with the framework, however, you can try the following code example, which shows the creation of an Rmd file:

```
file.edit("test-document.Rmd")
```

Try adding some code chunks and text by following the guidance in the Rmd cheatsheet, which you can get from the Help > Cheatsheets menu in RStudio.

9.2 Sharing code

Another way to increase the impact of your code is to share it. This can help collaborate with colleagues, getting feedback from others, and generating interest in your work as part of collaborative research processes that have been in operation for hundreds of years, as summarised by the phrase, 'Building on the shoulders of giants.' A more prosaic, but perhaps more important, corollary to that is, 'Do not reinvent the wheel.' By getting your code 'out there' you will be able to ensure that others can use your code and, because publishing your code encourages

searching for other code bases, help you to find components written by others to improve your work. Sharing code can therefore save many hours of time, provided you are happy to read and re-use, and of course give due credit and reference to code and ideas from other people.

The easiest way to share code in 2023 (and likely for the foreseeable future) is GitHub, an online code sharing, project management and file hosting platform. A great way to get started with GitHub, after you have signed up and created a user name at github.com, is to contribute to an existing project. **Challenge:** suggest a change to the code repository on GitHub that contains the source code of this book.

9.3 Asking questions

A final thing to say on R before the testimonials below is how to ask questions. There are many places to ask for help online, including:

- The question and answer site https://stackoverflow.com/. You will get quick answers here but be warned, answers may not always be particularly friendly if you ask a question that doesn't make sense or which has already been answered in the documentation - that should not put you off though, sometimes it's a case of, 'Don't ask, don't get.'
- The https://community.rstudio.com/ forum, where you may get more detailed and friendlier responses, especially if the question relates to RStudio, although the answer may be slower.
- Special interest groups such as https://gis.stackexchange.com/ (for GIS related questions), and the Slack group RSGB Analyst Network for road safety data analysis questions

Perhaps better than all of the above, is to ask a colleague who has slightly more experience than you. That way you will build 'collaboration networks.' The final thing to say on asking questions is 'use reprex!' To see what I mean by this try typing the following:

```
# example of creating a good reprex:
reprex::reprex({
  x = 1
  y = "2"
  # why does this fail?
  x + y
  # but this succeeds?
  x + as.numeric(y)
})
# after running the code above you can share the copied output to help ask questions
```

Note, you can turn any bit of code into a 'reprex' by selecting it and by running the 'Reprex selection' addin in RStudio, as described on the Tidyverse website.

9.4 Testimonials

This final section provides insight not only into how R can be used for road safety research from a range of perspectives, but also navigating R's at times steep learning curve.

9.4.1 R for professional road safety analysts

Will Cubbin, Road Safety Strategy Analyst, Safer Essex Roads Partnership

When I attended the two-day course 'introduction to R' I had little confidence in my natural ability to learn coding. Although familiar with many functions in Excel and having dabbled in VBA, my two previous attempts at any kind of computer language both ended in literal failure. At university I failed a module on C++ and in a previous job I failed a training course on SQL!

As expected the course was a steep learning curve but after two days I had definitely learned a few tricks. However I was still concerned about the amount of material covered by the course that I hadn't understood. It turned out this was actually a good thing because the breadth of the course showed me what R was capable of, and how it could be useful. The next part of my journey with R was to use the course materials and build on the basics I had learned, to achieve what the course showed was possible.

I began with the aim of using the geospatial analysis capability in R to visualise collision data in new ways, to give more detailed insight and present it in a way that would inform meaningful action for front line resources. Having a clear goal of "This is what I want to achieve with my first R project" was crucial. The course had given me an idea of the sort of processes I needed to undertake in order to achieve this goal. The post-course support through GitHub was very good, I also learned a lot by finding examples of code on places like GitHub and stackoverflow through Google searches. The other crucial element in getting my first success with R was having time dedicated to working on the project immediately after the course. I spent two weeks working almost exclusively on this project, starting the week after the R course.

The result was well worth the effort. After the initial two weeks of intensive learning with R, I spent 4 to 6 weeks working on the project a couple of days per week. By the end of this period I had working versions of two interactive mapping tools, comprising: 1) A multi layered leaflet map showing collision locations, collision density along main roads and a "heatmap" (Kernel Density Estimation raster) layer. I made multiple versions of each layer for different modes of transport and behaviours such as drink driving. 2) A 'shiny' mapping app showing collision locations and basic details with date filter. I was able to embed this on our website for public use. It can be viewed here SERP website data page under the heading 'Interactive Map'.

I soon added a second R script to the first of the two projects described above. This script produced and exported a range of standardised infographics showing various breakdowns of the data contained in the map. This allowed me to almost fully-automate the process for updating a proactive Roads-Policing tasking document. It turned this monthly process, which previously took 1 working day to complete, into one taking just 45 minutes. It also added more useful insight to the monthly tasking product.

My next steps are to continue another project using an API to access vehicle telematics data. This project extracts driving events, such as harsh braking and harsh cornering, and plots them on an interactive map. I will also be using R for some statistical analysis as part of a research project I have recently started. Thinking of myself as an "Excel native", I would say R hasn't replaced Excel, but has been a powerful addition to my toolbox so I can do more interesting and in-depth work than ever before.

9.4.2 R in a road safety research consultancy

Dr Craig Smith, Data Scientist, Agilysis Ltd

After attending my first R course, I immediately saw how useful the language could be to our team. At Agilysis, we have integrated R into almost everything our analytics team does, pushing for our work to be as robust and reproducible as possible. The incredible integration R has with SQL database engines, cloud-based infrastructure like AWS, and proprietary GIS software like ArcGIS, has given us a huge scope to automate a lot of our regular data processing tasks. The added ability to automate the production of reports, charts and maps has allowed us to gain quick insights into our data, speeding up our exploratory analysis. The huge Shiny ecosystem has allowed us to produce interactive applications for sharing our tools and visualisations with stakeholders. There is also a wide range of open-source statistical and machine learning packages available which, when combined with the added capability to translate and use tools from python, has allowed us to innovate and take full advantage of what artificial intelligence has to offer. All of this, embedded in a supportive community of R users that is continually sharing its knowledge and helping spread these skills, means that anything is possible for users, whatever their level of experience. The fact that this community includes a growing number of road safety analysts and transport-focused data scientists (thanks, in part, to this book and the previous R for Road Safety courses) means that there are plenty of like-minded R users all over the country that you can share ideas (and code) with.

9.4.3 Using R in a road safety charity

Emily Nagler, Data Analyst, RAC Foundation

I was first introduced to R in my master's program by a professor who was very enthusiastic as to how much better it was than ArcGIS for spatial analysis. Whilst he did project his views very strongly on us as students, I still tended towards ArcGIS when given a choice between

the two. I had never written code before this point, and to me the GUI of ArcGIS was simply easier to work with. However, over the course of my program I incorporated R more and more as our coursework became increasingly complex. Datasets with tens of thousands of rows quickly became a pain in Excel, and the inefficiency of working between two programs rather than one was tedious. Like many of my classmates, I soon realised the point our professor had been trying to make all along: in order to push your analytical capabilities to the next level you need to use the right tool. I can agree that there is a steep learning curve to R, but once you've become comfortable with the language, it's harder to go back than it is to go forward. Meaning, once you reach a point in your proficiency, it's easier to build on your skills and knowledge of the language than it is to revert back to Excel knowing there is a smarter alternative.

I found myself face to face with R again in my role at the RAC Foundation a year later, and since I'd already become an R convert, it was easy to pick up again. However, now I saw new benefits from a different perspective, relating to collaboration and reproducibility. Before this point, the only eyes on my scripts were my own, and there was no need to refer back to them once an assignment was done. However, in my role as a data analyst, it is important and necessary to create something that can be shared and understood by others. The ability to have a full account written out line by line makes it much easier to work off someone else's code and, importantly, to quality check one another's work. By having a traceable, repeatable analysis, there is a level of accountability with the method that you simply cannot get in Excel.

Another great aspect of R is the helpful online community presence, thanks to its open source nature. Anyone can access the software, and anyone can contribute to the discussion, which I often find more helpful than the FAQs of a paid for software. I would say that this aspect alone has helped hugely in my work, both academically and professionally. I'm never far from an answer when stuck, no matter how specific the problem gets. In terms of its use in my work on road safety and transport analysis it has become a great tool for combining data with maps, something that I think highlights results in a more insightful way. That being said, visualisation is just as important as the analysis, and I've been pleased with the array of packages that can create eye-catching maps, interactive dashboards, and much more outside of the traditional data wrangling capabilities. Looking to the future of my career I see myself continuing to use R and I'm confident that its capabilities will only get better the more people engage with it.

9.4.4 Using R in other areas of road safety research

Do you have a use case of reproducible research? Please get in touch on the issue tracker.