

Tutorials for BIOL202: Introduction to Biostatistics

Jason Pither

2025-11-19

Contents

Preface	11
Welcome	11
Author	11
Acknowledgments	11
Copyright	12
UBCO Biology open materials	12
Getting started with R, RStudio & R Markdown	15
1 What are R and RStudio?	15
1.1 Installing R and RStudio	16
2 Start using R & RStudio	17
2.1 The RStudio Interface	18
2.2 Coding basics	18
2.3 R packages	18
2.4 Package installation	19
2.5 Package loading	20
2.6 Intro to R Markdown	20
2.7 Literate programming with R Markdown	21
2.8 Making sure R Markdown knits to PDF	21
2.9 Extra resources	22
2.10 R Resources Online	23

Reproducible Workflows	29
3 Reproducible Research	29
3.1 Computational reproducibility	30
3.2 An example BIOL202 workflow	30
3.3 Microsoft OneDrive	31
3.4 Directory structure	31
3.5 Steps to set up directories	32
3.6 Lecture workflow	32
3.7 Tutorial workflow	34
3.8 Create an RStudio Project	34
3.9 Create subdirectories	34
3.10 Edit an R markdown file	35
3.11 Components of an R Markdown file	36
3.12 Interacting with Tutorial material	36
3.13 Lab assignments workflow	37
4 Preparing and formatting assignments	39
4.1 Open your assignment RStudio project	39
4.2 Download the assignment Rmd file	39
4.3 Open the assignment Rmd file	40
4.4 What to include in your answers	40
4.5 Setting up R Markdown for graphing	44
4.6 Example question / answer	47
4.7 Knitting your assignment to PDF	52
4.8 Submit your assignment	53
5 Preparing and importing Tidy Data	55
5.1 Tidy data	56
5.2 Import a CSV file from a website	56
5.3 Create a CSV file	59
5.4 Import a local CSV file	59
5.5 Get an overview of a dataset	61
5.6 Tutorial practice activities	65

CONTENTS	5
Visualizing and Describing Data	69
6 Visualizing a single variable	69
6.1 Load packages and import data	70
6.2 Get an overview of the data	70
6.3 Create a frequency table	75
6.4 Create a bar graph	78
6.5 Create a histogram	81
6.6 Describing a histogram	86
7 Describing a single variable	89
7.1 Load packages and import data	89
7.2 Describing a categorical variable	90
7.3 Describing a numerical variable	92
7.4 Describing a numerical variable grouped by a categorical variable	97
8 Visualizing associations between two variables	99
8.1 Load packages and import data	100
8.2 Visualizing association between two categorical variables	101
8.3 Visualizing association between two numeric variables	110
8.4 Visualizing association between a numeric and a categorical variable	114
Inferential Statistics	129
9 Sampling, Estimation, & Uncertainty	129
9.1 Load packages and import data	129
9.2 Functions for sampling	132
9.3 Sampling error	135
9.4 Sampling distribution of the mean	137
9.5 Standard error of the mean	144
9.6 Rule of thumb 95% confidence interval	147

10 Hypothesis testing	151
10.1 Load packages and import data	151
10.2 Steps to hypothesis testing	153
10.3 An hypothesis test example	154
11 Analyzing a single categorical variable	163
11.1 Load packages and import data	163
11.2 Estimating proportions	165
11.3 Binomial distribution	173
11.4 Binomial test	179
11.5 Confidence interval approach to hypothesis testing	182
11.6 Goodness-of-fit tests	182
12 Analyzing associations between two categorical variables	183
12.1 Load packages and import data	184
12.2 Fisher's Exact Test	188
12.3 Estimate the Odds of getting sick	193
12.4 Estimate the odds ratio	194
12.5 χ^2 Contingency Test	197
13 Analyzing a single numerical variable	205
13.1 Load packages and import data	205
13.2 One-sample t -test	206
13.3 Confidence intervals for μ	211
14 Comparing means among two groups	217
14.1 Load packages and import data	217
14.2 Paired t -test	218
14.3 Two sample t -test	227
14.4 When assumptions aren't met	238

15 Checking assumptions and data transformations	241
15.1 Load packages and import data	242
15.2 Checking the normality assumption	245
15.3 Checking the equal-variance assumption	251
15.4 Data transformations	252
16 Comparing means among more than two groups	269
16.1 Load packages and import data	270
16.2 Analysis of variance	274
16.3 When assumptions aren't met	290
17 Analyzing associations between two numerical variables	291
17.1 Load packages and import data	291
17.2 Pearson correlation analysis	295
17.3 Rank correlation (Spearman's correlation)	303
18 Least-squares linear regression	309
18.1 Load packages and import data	309
18.2 Least-squares regression analysis	313
18.3 Making predictions	332
18.4 Model-I versus Model-II regression	335
Load all the necessary packages	337
Data summaries with “gtsummary” package	343
Creating tables in R Markdown	347
18.5 Load packages and import data	347
Visual Markdown Editor	355
A more familiar editing environment	355

Common errors and their solutions	357
Google can help	357
Error creating mosaic plot	357
Rosetta error	359
Rtools required during install	359
Could not find function	360
There is no package	360
Trying to use CRAN without setting a mirror	361
PDF Latex is not found	361
Error in parse	362
No such file or directory exists	363
Messy output when loading packages	363
Unused argument	364
Object not found	364
Figure caption doesn't show up below figure in knitted document . . .	365
Figures are placed in weird spots in knitted PDF	365
Installing packages: there is a binary version available	365
Unicode knitting error	366

Preface

Welcome

This is an open source online book that, in one capacity, provides tutorials and other resources for the lab portion of the course BIOL202: Introduction to Biostatistics, at the University of British Columbia, Okanagan campus. Its other role is to serve as a general resource for those wishing to learn how to manage and analyze data using R and R Markdown within the RStudio IDE.

This book is a **living document that is updated intermittently**. If you find errors or wish to provide feedback, please feel free to contact me.

Author

I (Jason Pither) am an Associate Professor in the Department of Biology at the Okanagan campus of the University of British Columbia. I am an ecologist with interests in biogeography, community ecology, and landscape ecology. I have been using “R” (and its predecessor, “S”) in my research for over two decades, and have been teaching Introductory Biostatistics using R since 2014.

Acknowledgments

I am grateful to be able to live, work, and play on the traditional, ancestral, and unceded territory of the Syilx (Okanagan) people. To find out more about Indigenous territories where you live, consult <https://native-land.ca>.

This online book borrows some material generously made openly available by the following keen educators:

- Chester Ismay and Albert Y. Kim (Statistical Inference via Data Science online book). This resource is licensed under a Creative Commons Attribution - NonCommercial-ShareAlike 4.0 International License.
- Mike Whitlock and Dolph Schluter (resources accompanying the text “Analysis of Biological Data”)

It has also benefited from valuable input from the many BIOL202 students and teaching assistants who have helped me over the years. Thank you!

Clerissa Copeland, Jordan Katchen, and Mathew Vis-Dunbar helped with some of the content development and with identifying links to other UBCO Biology material. Thanks!

Copyright

This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International (CC BY-NC-SA 4.0)



Please use the following for citing this document

Pither, J. (2022). *Tutorials for BIOL202: Introduction to Biostatistics*. <https://ubco-biology.github.io/BIOL202/index.html>

All source files are available on github.

UBCO Biology open materials

This resource is part of a larger project to host UBCO Biology lab materials in an open, accessible format.

All BIOL open materials can be found at <https://ubco-biology.github.io/>

Getting started with R, RStudio & R Markdown

Chapter 1

What are R and RStudio?

It is assumed that you are using R via RStudio. First time users often confuse the two. At its simplest:

- R is like a car's engine
- RStudio is like a car's dashboard.



Figure 1.1: R: Engine



Figure 1.2: RStudio: Dashboard

More precisely, R is a programming language that runs computations while RStudio is an *integrated development environment (IDE)* that provides an interface by adding many convenient features and tools. So the way of having access to a speedometer, rearview mirrors, and a navigation system makes driving much easier, using RStudio's interface makes using R much easier as well.

1.1 Installing R and RStudio

At the time of writing this module, the latest version of R was 4.5.1 (“Great Square Root”) (2025-06-13). At the beginning of term, it is recommended that you update to the latest version of R, but once you’ve started into the tutorials, it is recommended that you NOT update again until after term is complete. This is to avoid any unforeseen problems introduced by the update. **However** If you have an older operating system on your computer, you’ll want to check compatibility of both R and RStudio.

RStudio is now owned by “posit”, and its website (posit.co) should detect your operating system and provide the appropriate download option automatically. The website with older versions of R is [here](https://cran.r-project.org/bin/windows/base/), and older versions of RStudio is [here](https://rstudio.com/products/rstudio/download/).

Follow the instructions below, and you are encouraged to view this YouTube video [here](#) prior to starting the steps. It talks about installing for different operating systems and different vintages of Macs (e.g. newer M1 chips versus Intel chips).

- Download and install both R and RStudio (Desktop version) on your computer.

R needs to be installed successfully >prior< to installing RStudio (because the latter depends on the former)

- Figure out what operating system (and version) you have on your computer (e.g. Windows 10; Mac OS X 12.51 “Monterey”)
- Go to this website and click on the appropriate download link at the top of the page (depending on your operating system, Windows / MacOS / Linux)
 - For *Windows* users, download the “base” version; this file will be called something like R-4.1.1-win.exe. Executing this file launches a familiar Windows Setup Wizard that will install R on your computer.
 - For *Mac* users, download the “pkg” file that is appropriate for your version of MacOS; the file will be called something like R-4.1.1.pkg. Download and run this installation package—just accept the default options and you will be ready to go.
- Now to install RStudio: once you have installed “R”, go to this website and click on the “download RStudio desktop” button under the “Install RStudio” heading. The website should detect what operating system you’re using and offer you the appropriate version.

Chapter 2

Start using R & RStudio

Recall our car analogy from a previous tutorial. Much as we don't drive a car by interacting directly with the engine but rather by using elements on the car's dashboard, we won't be using R directly but rather we will use RStudio's interface. After you install R and RStudio on your computer, you'll have two new programs AKA applications you can open. We will always work in RStudio and not R. In other words:



Figure 2.1: R: DO NOT OPEN THIS

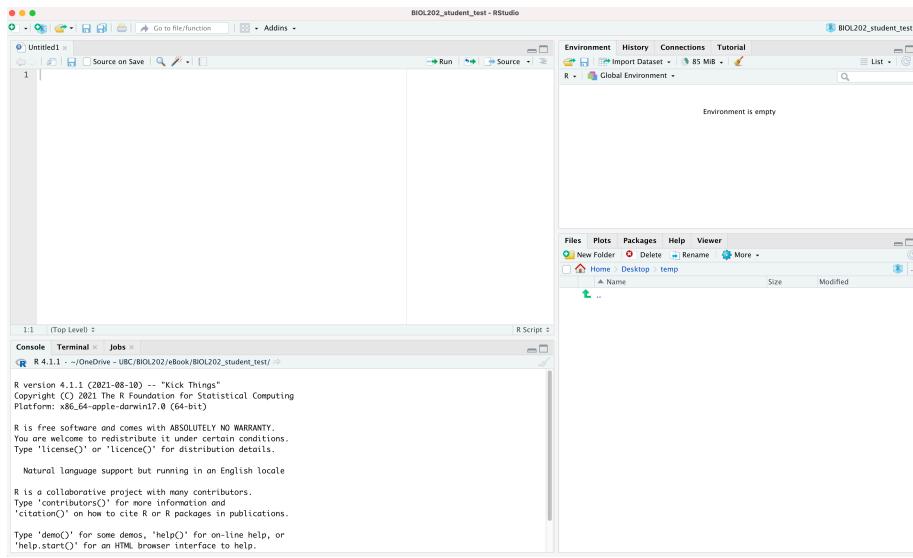


Figure 2.2: RStudio: OPEN THIS

Launch RStudio on your computer to make sure it's working (it loads R for you in the background).

2.1 The RStudio Interface

When you open RStudio, you should see something similar to the following:



Note the four panes which are four panels dividing the screen: the source pane (top left), console pane (bottom left), the files pane (bottom right), and the environment pane (top right). Over the course of this chapter, you'll come to learn what purpose each of these panes serves.

2.2 Coding basics

Please go through section 1.2 of the ModernDive online text called “How do I code in R?”. This should take about **15 minutes**.

You may get overwhelmed by all the terminology on that page... don't worry! You can simply bookmark that page for future reference. But it does cover the coding basics very well!

2.3 R packages

An R package is a collection of functions, data, and documentation that extends the capabilities of R. They are written by a world-wide community of R users. For example, among the most popular packages are:

- `ggplot2` package for data visualization

- `dplyr` package for data wrangling
- `tidyverse`, which is a package that includes a collection of multiple packages (including the preceding two) that are all installed at once. We'll be using this package in the course.

There are two key things to remember about R packages:

- *Installation*: Most packages are not installed by default when you install R and RStudio. You need to install a package before you can use it. Once you've installed it, you likely don't need to install it again unless you want to update it to a newer version of the package.
- *Loading*: Packages are not loaded automatically when you open RStudio. You need to load them everytime you open RStudio.

2.4 Package installation

Let's install the `tidyverse` package.

There are two ways to install an R package:

- In the Files pane:
 - Click on “Packages”
 - Click on “Install”
 - Type the name of the package under “Packages (separate multiple with space or comma):” In this case, type `tidyverse`
 - Click “Install”
- Alternatively, in the Console pane type the following

Later, when you start using R Markdown, never include the following “`install.packages`” code within your R Markdown document; only install packages by typing directly in the Console!

```
install.packages("tidyverse")
```

If you are attempting to install a package (using `install.packages`) and you get this message:

```
There is a binary version available but the source version is later:  
  binary source needs_compilation  
systemfonts  1.0.2  1.0.3          TRUE
```

Do you want to install from sources the package which needs compilation? (Yes/no/cancel)

Respond with “no” (without quotes). Do NOT respond “Yes”.

When working on your own computer, you only need to install a package once, unless you want to update an already installed package to the latest version (something you might do every 6 months or so). **HOWEVER:** If you’re working on a school computer (in a computer lab or in the library), you may need to install packages each session, because local files (on the computer) are automatically deleted daily. If you’re unsure what packages are already installed, consult the “packages” tab in the lower-right RStudio pane when you start up RStudio; installed packages are listed there.

2.5 Package loading

Let’s load the `tidyverse` package.

After you’ve installed a package, you can now load it using the `library()` command. For example, to load the `tidyverse` package, run the following code in the Console pane:

```
library(tidyverse)
```

You have to reload each package you want to use every time you open a new session of RStudio. This is a little annoying to get used to and will be your most common error as you begin. When you see an error such as

```
Error: could not find function
```

remember that this likely comes from you trying to use a function in a package that has not been loaded. Remember to run the `library()` function with the appropriate package to fix this error.

2.6 Intro to R Markdown

As you may have learned already from relevant section in the Biology Procedures and Guidelines resource, R Markdown is a markup language that provides an easy way to produce a rich, fully-documented reproducible analysis. It allows its user to share a single file that contains all of the commentary, R code, and metadata needed to reproduce the analysis from beginning to end. R Markdown allows for “chunks” of R code to be included along with Markdown text to produce a nicely formatted HTML, PDF, or Word file without having to know any complicated programming languages or having to fuss with getting the formatting just right in a Microsoft Word DOCX file.

One R Markdown file can generate a variety of different formats and all of this is done in a single text file with a few bits of formatting. You'll be pleasantly surprised at how easy it is to write an R Markdown document after your first few attempts.

We will be using R Markdown to create reproducible lab reports.

R Markdown is just one flavour of a markup language. RStudio can be used to edit R Markdown. There are many other markdown editors out there, but using RStudio is good for our purposes.

2.7 Literate programming with R Markdown

1. View the following short video:

Why use R Markdown for Lab Reports?

The preceding video described what can be referred to as **literate programming**: authoring a single document that integrates data analysis (executable code) with textual documentation, linking data, code, and text. In R Markdown, the executable R code is placed in “chunks”, and these are embedded throughout sections of regular text.

For an example of a PDF document that illustrates literate programming, see here. It accompanied a lab-based experiment examining the potential for freshwater diatoms to be successfully dispersed over long distances adhered to duck feathers.

2. View the following youtube video on creating an R Markdown document.

If you'd like additional introductory tutorials on R Markdown, see this resource.

2.8 Making sure R Markdown knits to PDF

Now we're going to ensure R Markdown works the way we want. A key functionality we need is being able to “knit” our report to PDF format.

With the newest versions of RStudio, you may be able to knit to PDF without doing anything special first.

Let's give this a try:

Step 1: While in RStudio, select the “+” dropdown icon at top left of RStudio window, and select R Markdown. RStudio may at this point install a bunch of things, and if so that's ok. It may also ask you to install the `rmarkdown` package.. if it does, do so!

Step 2: A window will then appear and you can replace the “Untitled” with something like “Test”, then select OK.

Step 3: This will open an R Markdown document in the top left panel. Don’t worry about all the text in there at this point. What we want to do is test whether it will “knit” (render) the document to PDF format.

Step 4: Select the “Knit” drop-down icon at the top of the RStudio window, and select “Knit to PDF”. RStudio will ask you to first save the markdown file (save it anywhere with any name for now), then it will process the markdown file and render it to PDF.

If this worked, great!! You can ignore the next section here. If it didn’t work, then proceed to this next section:

If the preceding steps did not result in you being able to knit your markdown document to PDF, then do this:

Install the `tinytex` package by typing this code into the command console of RStudio:

```
install.packages("tinytex")
```

Then, once that has installed successfully, type the following:

```
tinytex::install_tinytex()
```

RStudio will take a minute or two to install a bunch of things. Once it’s done, we’re ready to try knitting to PDF.

Recall you only need to install a package once! And this should be the last time you need to deal with the `tinytex` package (you won’t need to “load” it in future), because now that it’s installed, its functionality works in the background with RStudio.

Now go back to the steps 1 through 4 above to try knitting your markdown document to PDF.

In a future tutorial we’ll discuss how to use R Markdown as part of a reproducible workflow.

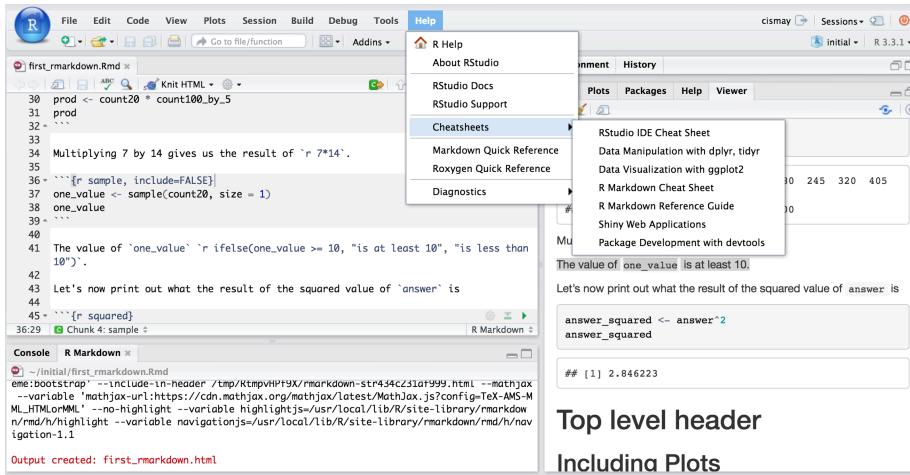
2.9 Extra resources

This page is still under development (Aug 2022)

If you are googling for R code, make sure to also include package names in your search query (if you are using a specific package). For example, instead of googling “scatterplot in R”, google “scatterplot in R with ggplot2”.

Rstudio provides links to several **cheatsheets** that will come in handy throughout the semester.

You can get nice PDF versions of the files by going to **Help -> Cheatsheets** inside RStudio:



The book titled “Getting used to R, RStudio, and R Markdown” by Chester Ismay, which can be freely accessed here, is also a wonderful resource for new users of R, RStudio, and R Markdown. It includes examples showing working with R Markdown files in RStudio recorded as GIFs.

2.10 R Resources Online

Table 2.1: Learning R - Tutorials and workshop materials

URL	Purpose
https://whitlockschluter3e.zoology.ubc.ca.	R resources to accompany Whitlock & Schluter text
https://datacarpentry.org/R-genomics/01-intro-to-R.htm	Data Carpentry Intro R tutorials
https://r-dir.com/learn/tutorials.html	List of useful tutorials including videos
https://www.rstudio.com/resources/webinars/	Various online learning materials at RStudio
https://rmd4sci.njtierney.com	R Markdown for Scientists
http://r4ds.had.co.nz/index.html	Hadley Wickham’s online book

Table 2.2: So many packages - how to find useful ones?

URL	Purpose
http://blog.revolutionanalytics.com/beginner-tips/	Beginner tips
https://mran.microsoft.com/packages/	Tool for exploring packages

Table 2.3: Data analysis in R

URL	Purpose
https://www.zoology.ubc.ca/~schluter/R/	UBC zoology site handy stats reference
http://statmethods.net/	Good general reference

Table 2.4: Data visualization in R

URL	Purpose
http://statmethods.net/	Good general reference for graphing
http://ggplot2.org/book/	Hadley Wickham's ggplot2 book online
http://stats.idre.ucla.edu/r/seminars/ggplot2_intro/	Great tutorial on ggplot2
http://www.cookbook-r.com/Graphs/	Graphing with ggplot2

Table 2.5: BLOGs and general resources

URL	Purpose
https://www.r-bloggers.com/	Popular blog site, lots of info
http://rseek.org/	Search engine for R help

Table 2.6: Resources for reproducible research / open science

URL	Purpose
https://cos.io/	Center for Open Science
https://ropensci.org/	Open science R resources
http://geoscripting-wur.github.io/RProjectManagement/	Lessons about version control
https://nicercode.github.io/	Helping to improve your code

Table 2.7: Geospatial work in R

URL	Purpose
http://spatial.ly/r/	Example maps with R
https://rstudio.github.io/leaflet/	Online mapping with R
https://www.earthdatascience.org/courses/earth-analytics/	Excellent course
https://geoscripting-wur.github.io/	Amazing “geoscripting” tutorial

Reproducible Workflows

Chapter 3

Reproducible Research

A key goal of our Biology program at UBC's Okanagan campus is to foster appreciation for reproducible research, and to equip students (and professors) with skills that will help them undertake reproducible research themselves.

In first year you may have learned from the Biology program's introductory Open Science learning modules that reproducible research studies are not as common as one might assume. (If you did not cover that material, do so now). One key reason for this is insufficient documentation of all the steps taken along the research workflow. Moreover, conducting reproducible research is extremely challenging - more than most scientists appreciate. See, for example, an incredible, recent case concerning ageing experiments with *C. elegans* here.

In the BIOL202 lectures you'll learn more about the various causes of irreproducible research, and about the practices that can help promote reproducibility. In this lab component of the course, you'll learn the basics of how to achieve an acceptable level of *computational reproducibility* (complete computational reproducibility is actually pretty tricky, but we'll get close).

Learning Outcomes

Upon successful completion, students will be able to:

1. Implement a computationally reproducible workflow using R, RStudio, and R Markdown
2. Organize and manage data and project files using versioned, structured formats
3. Create clean, analysis-ready datasets in .csv or .rds formats
4. Use R Markdown to document and explain all analysis steps and results

3.1 Computational reproducibility

Almost all research, whether it's conducted in the field or in a lab, includes a substantial amount of work that's done on the computer. This includes data processing, statistical analyses, data visualization and presentation, and production of research outputs (e.g. publications). Some research is of course exclusively conducted on computers. The bottom line is that computer-based work forms a key and substantive part of all research workflows.

Given that this work is done on computers (which are entirely controllable), it should be able to be reproduced **exactly**. This is known as **computational reproducibility**: an independent researcher should be able to access all data and scripts from a study, run those scripts, and get exactly the same outputs and results as reported in the original study.

In this tutorial you'll start gaining relevant experience and skills by producing a reproducible lab report.

A *workflow* refers to the steps you take when conducting your day-to-day work - say, on a term project, for example. Having a well-designed workflow improves efficiency, and when done right, reproducibility. It includes, for example, how you create, access, and manage files on your computer (or in the cloud).

Following best practices for naming and organizing your files and directories on your computer will help ensure that you can spend more time doing the important work, and less time fiddling and trying to remember what you did and where you saved your work. It will also help your future self, when labs and assignments in upper year courses request that you use R and R Markdown for analyses and reports.

1. Review the Biology department's Procedures and Guidelines webpage description of how to manage files and directories. This should take about **20 minutes**

3.2 An example BIOL202 workflow

Now that you have reviewed the fundamentals of file and directory management, you should decide how to best organize and manage the work you do for BIOL202. Here we'll provide one example approach that most if not all of you will find useful.

3.3 Microsoft OneDrive

Our suggested approach assumes you have set up a Microsoft account through UBC (using your CWL), and have set up the OneDrive application on your own computer, which automatically syncs (when you're online) selected files/directories between your local computer and your OneDrive account in the cloud.

Why OneDrive? As a UBC student, you receive 1TB of free storage! And you also get peace of mind knowing that your files are secure and up-to-date (provided you have an internet connection), and that OneDrive has something called “version control”, which saves old versions of files and allows you to see those versions if you wish, **so long as you maintain the same file name**.

WAIT A SECOND! In the “File Naming” instructions that I just reviewed, I was instructed to create a new file with a new version number in the filename (e.g. with a “V0”, “V1”, “V2” etc...) each time I worked on it!

Those instructions are entirely valid! However, when you have access to a *version control* system, like OneDrive, it is better to **keep the file name the same**, rather than changing it each time you update it. For example, your markdown file (which as a “.Rmd” extension to the name) that you use for your tutorial work should maintain the same name throughout the term, rather than saving a new file each time you do substantive work on it.

Assuming your files are syncing properly between your computer and OneDrive (and this simply requires that you're connected to the internet), you will always be able to see (and if desired, restore) old versions of your files.

If you haven't set up OneDrive yet, follow the instructions provided at this UBC IT website.

Using OneDrive is entirely optional. If you choose not to use OneDrive, please follow the file naming instructions from the Procedures and Guidelines document. And you can still follow the directory structure instructions below, regardless of where you set up your directories (OneDrive or not).

3.4 Directory structure

We anticipate three general categories of work being undertaken:

- Lecture work, including annotating lecture notes (e.g. on PDFs or PowerPoints) and writing study notes of your own (e.g. using Word)
- Tutorial work, including practicing what you learn in tutorials using RStudio and R Markdown, and commenting about tips, or tricky bits
- Lab assignment work, in which you answer questions using R and R Markdown and create a document for submission and grading

Each of these categories of work should have their own directory, and all three of these directories should exist within a “root” directory called “BIOL202”.

3.5 Steps to set up directories

- Having successfully installed OneDrive, you should see a “OneDrive” folder on your computer
- In your OneDrive folder, create a root directory “BIOL202” to host all of your BIOL202 work
- Create a “_README.txt” file for the root BIOL202 directory, as per instructions in the Biology Procedures and Guidelines webpage.

You can create and edit your “_README.txt” file in RStudio! Just click on the “+” drop down at the top left, and select “Text file”. Then type in the information you need, and name it “_README.txt” and save it in the appropriate directory.

- Within the BIOL202 root directory, create three (sub)directories:
 - “BIOL202_lecture”
 - “BIOL202_tutorials”
 - “BIOL202_assignments”
- Create a “_README.txt” file in each of the three sub-directories (again, you can use RStudio to create these!).

An example setup is illustrated below.

The BIOL202 directory, and all its contents, will now sync regularly to your online OneDrive account, so that you can access your up-to-date files from any device upon which OneDrive is installed.

It is possible to work on files stored on your local computer when you’re offline; they just won’t sync to the cloud until you’ve gotten back online.

3.6 Lecture workflow

The directory for your lecture work is now ready to house any lecture-related work that you do. For instance, if you wish to type up study notes in a Word document, you could call that file “Pither_BIOL202_lecture-notes.docx”. Each time you add/edit/update it, OneDrive will keep old versions for you!

**Network drive
or OneDrive**

3.7 Tutorial workflow

How you manage your tutorial workflow comes down to personal preference. The instructions provided below create an RStudio project in the “BIOL202_tutorials” directory (see below), and then create a single R Markdown document, formatted to have sections (headers) for each tutorial you work on. This approach makes it easier to find all of your work in the R Markdown file, so long as it is formatted logically.

I have created an R Markdown file that you can download and use for this purpose, and we’ll download that a bit later.

First, we need to set up an RStudio project.

3.8 Create an RStudio Project

It is best to organize all your R work using RStudio “projects”. For your tutorial work, for example, you will create a new project that will be housed in your “BIOL202_tutorials” directory

To do this, open RStudio then select File > New Project. Then select the option to use an existing directory, and locate and select your tutorial directory. Provide a name for your project file, such as “BIOL202_tutorial”, then select OK. If it asks to open a new RStudio session, you can say yes.

RStudio has now created an RStudio project file that has an “Rproj” extension at the end of the name. This “Rproj” file is the file that you will open (double-click with the mouse) each time you wish to work on your tutorial material. You should see the “Rproj” file in the bottom right files panel of RStudio.

3.9 Create subdirectories

Your tutorial work may involve creating and saving outputs like figures or files, in which case you should have sub-directories for these purposes in your root directory. See the example provided in the Procedures and Guidelines document.

Let’s illustrate the procedure by creating a subdirectory called “figures”, and this time we’ll use R to create the directory.

The following code will create a directory called “figures” in your working directory

```
dir.create("figures")
```

You should see the new folder appear in the files panel on the bottom right of RStudio.

Whenever you wish to generate figures in R, then export them as image files, for example, you can save them to this folder. We'll learn about this later.

Reading and writing files from / to directories requires that we can tell R exactly where to find those directories. That's where the handy package called `here` comes in.

3.9.1 The `here` package

Now we'll install and load an R package called `here` that will help with file and directory management and navigation.

```
install.packages("here")
```

This is a helpful vignette about the `here` package.

Then we load the package:

```
library(here)
```

When you load the package, the `here` function takes stock of where your RStudio project file is located, and establishes that directory as the “working directory”. It should return a file path for you. In future tutorials we'll make use of the `here` package when saving files.

3.10 Edit an R markdown file

In a previous tutorial you learned how to create an R Markdown file, but generally you won't need to do this in this course, because you'll be provided starter documents to work with.

I have created an R Markdown file that you can download and use for starting your tutorial work.

To do this, type the following code into your command console in RStudio (the bottom left panel):

```
download.file("https://raw.githubusercontent.com/ubco-biology/BIOL202/main/more/Example_tutorial.Rmd")
```

This `download.file` command takes a URL address for a file stored online, and then tells R where to save it, and what name to use.

Here we will save the RMD file in your “working directory”, which is the “BIOL202_tutorials” directory. It should show up in the “files” pane in the

bottom right RStudio panel. Click on the “BIOL202_tutorial_markdown.Rmd” file to open it.

This is the R Markdown file that you’ll edit/add to throughout the term. It has some starting instructions / example text already, and you’re welcome to delete / change that.

- Change the “author” information at the top of the document (in the header) to your name.
- Save your file by clicking the save icon in the top left, and be sure to give it an appropriate name as per previous file-naming instructions

For the basics on formatting in R Markdown, consult the R Markdown reference guide

3.11 Components of an R Markdown file

At this point, you should familiarize yourself with the components of a markdown file.

1. **R Markdown basics:** Read through section 4.2 of this online tutorial and watch the videos there too
-

3.12 Interacting with Tutorial material

Students have typically worked through tutorials by reading the material, typing helpful notes in their markdown document, and typing the provided R code into “chunks” within their R Markdown document, then running those chunks. (**NOTE:** it’s better for you to practice typing the code, rather than copying and pasting!)

You can insert as many chunks as you’d like. Chunk insertion is achieved by selecting the green “+” icon at top of the editor panel, and selecting “R”. Or go to the “Code” menu drop down and “Insert Chunk”.

The key advice: keep your work organized by using headings. There’s an option to view your headings all in one navigation pane by clicking on the navigation pane icon at the top right of your editor panel.

3.13 Lab assignments workflow

You can repeat the same steps from earlier for creating a new RStudio project for your lab work. But this time when you create the project, specify your lab assignment directory “BIOL202_assignments” as the location.

You can optionally create subdirectories also, perhaps one for each of the lab assignments.

The key difference in the assignments workflow is that for each assignment you’ll start with a new R Markdown (Rmd) file, which you’ll download from Canvas under the “assignments” link. The document will include the questions you are to answer, and you simply edit / add to the document as you answer them.

Instructions for this will come in a later tutorial.

Next tutorial: Practicing completing a very short lab assignment using R Markdown.

Chapter 4

Preparing and formatting assignments

This material provides instructions on how to prepare and format R Markdown documents for assignments.

4.1 Open your assignment RStudio project

In a previous tutorial it was suggested you set up a working root directory called “BIOL202_assignments” to host all your assignment work. You may have also created sub-directories, one for each of the three assignments. Regardless, in your root “BIOL202_assignments” directory locate the RStudio project file (it has a “Rproj” extension in the filename), and open it by double-clicking it.

4.2 Download the assignment Rmd file

The assignment file will be available to you for download from Canvas under the “Assignments” section. The file is actually an R Markdown file, and therefore has an “Rmd” extension. It is simply a text document (meaning it contains only text), but the syntax used therein is “markdown” syntax, and the file extension (Rmd) specifies it as an R Markdown file. You might recall reading about Markdown in the Biology Procedures and Guidelines document.

Make sure you save the Rmd file into your root “BIOL202_assignments” directory, or alternatively into the appropriate sub-directory.

A small assignment has been set up on Canvas, with the aim of you getting use to the assignment procedure. It is located under “Lab_materials” module.

Download the file, and make sure it's saved into your root directory (where your RStudio project is).

4.3 Open the assignment Rmd file

In RStudio, look in the “files” pane (bottom right) and locate your downloaded assignment Rmd file. Then click it to open.

This Markdown document includes the assignment questions, and serves as a starter document for your assignment.

Once you have opened the document, you'll see the following text at the top:

Complete the following steps:

- keeping the quotation marks, replace the “Practice assignment” text with the current assignment name
- keeping the quotation marks, replace the “Firstname Lastname, Section YY” with your own Firstname Lastname and section number (e.g. ” Jason Pither, Section 04”)
- keeping the quotation marks, replace the “Due date” with the due date of the current assignment, e.g. “September 21, 2025”

Don't alter anything else at the top of the document, including the R Chunk that you see below the header.

- save the Rmd file using a new name (so select file > “save as”), according to appropriate file naming conventions, such as “Pither_BIOL202_lab-assignment-01_V0.Rmd”. Recall that if you're saving this on OneDrive, or in your local directory that is synced to OneDrive, then your file will be version controlled, so you don't need to include a version number (“_V0”) in the filename.

You're now ready to start working on your assignment, and doing literate programming using R Markdown.

In RStudio, under the “Help” menu, you'll find “Cheat Sheets” > R Markdown Cheat Sheet and R Markdown Reference Guide. Have a look at those!

4.4 What to include in your answers

The general approach is to enter your text answers and associated R chunks with code underneath each question.

```
1 ---  
2 title: "Practice Assignment"  
3 author: "Firstname Lastname"  
4 date: "Due date"  
5 output: pdf_document  
6 fig_caption: true  
7 ---  
8  
9 ```{r setup, include = FALSE}  
10 # DO NOT ALTER CODE IN THIS CHUNK  
11 knitr::opts_chunk$set(echo = TRUE)  
12 ````
```

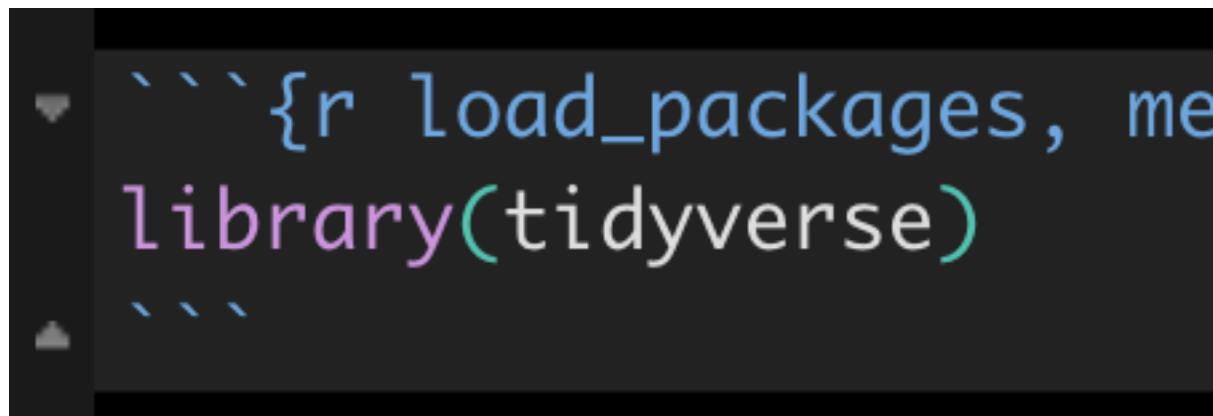
Figure 4.1: Assignment markdown file header

TIP: It is almost always the case that you'll need to load the `tidyverse` package, so it's advisable to include an R chunk right at the beginning of your document that does this.

TIP: To insert a new code chunk, you can either: Use the shortcut: Mac = **Command+Option+I**; Windows = **Ctrl+Alt+I**. Click on the **Insert a new code chunk** icon. You can also specify the coding language of the chunk using the drop-down menu.

4.4.1 Code chunk headers

Here's an example code chunk in which the `tidyverse` package is being loaded. You'll notice some extra text in the header of the chunk.



A screenshot of an R Markdown editor showing a code chunk. The code is:

```
```{r load_packages, message = FALSE}
library(tidyverse)
```
```

The code is highlighted in blue and purple. The first line starts with three backticks followed by the text '{r load_packages, message = FALSE}'. The second line starts with the word 'library' followed by '(tidyverse)'. The third line ends with three backticks. The background of the editor is dark grey.

Figure 4.2: Code chunk header with message suppression

The “load_packages” text is simply giving a name or “tag” to this R chunk. It is good practice to provide a *unique* name to each R chunk.

After the comma is `message = FALSE`. This tells R markdown to NOT display any messages associated with the R commands in this chunk when knitting the R Markdown document. For example, when loading some libraries R will provide a long list of messages in the process. When you “knit” the R Markdown document, R will run each R chunk in your document. By including the `message = FALSE` heading argument, you can avoid including this extra output in your knitted document.

The `message = FALSE` header argument is NOT something to include in all chunks by default. It should only be included if the code within the associated code chunk outputs a lot of unnecessary information that would otherwise clutter your knitted PDF document.

There will be routine steps to take for each question.

4.4.2 Import data

Each assignment question will indicate which dataset(s) you'll need to answer the question, and where to locate / download these. In many instances multiple questions will use the same dataset, in which case you don't need to load data for each question; just load it once.

Here's an example of code for importing data. It uses the `read_csv` function from the `readr` package, which is bundled with the `tidyverse` package:

```
example_data <- read_csv("https://raw.githubusercontent.com/ubco-biology/BIOL202/main/data/exampl
```

The `read_csv` function will take as an argument a URL address for a data file. Here we've provided a URL for the location of a CSV file (comma-separated values file), which is typically what we'll use.

4.4.3 Load packages

Once you've loaded the data, you'll now need to figure out how to answer the question by looking through the tutorials for the answer. And typically, to answer the question you'll need to make sure you've loaded some packages.

If you've forgotten what "packages" are, refresh your memory at this tutorial.

For example, it's likely you've already loaded the `tidyverse` package (as advised above), but if a tutorial indicates you need another package, say the `janitor` package, then insert a code chunk with the following:

```
library(janitor)
```

When working on your own computer, you only need to install a package once (unless you want to update an already installed package to the latest version, which is a rare event). If you're working on a school computer (in a computer lab), you may need to install packages each time you log on to a computer, because the school computers delete local files daily. If you're unsure what packages are already installed, consult the "packages" tab in the lower-right RStudio pane when you start up RStudio; installed packages are listed there.

4.4.4 Answer the questions

After you've imported the required data and loaded required packages, it's time to answer the questions!

You simply insert any required text and R code (in R Chunks) after each question. The tutorials will provide the information required to answer the questions.

TIP: The way to test out your code once it's inserted in the R chunk is to press the green play button at the top right of the code chunk. This will run what's in the chunk. You'll be provided error statements if something goes wrong. If it works, it will provide you the output (and hopefully the answer!), and this is what you use to inform your written (typed) answer.

Here's what you need to include in your answers:

- Answer each question clearly and in full sentences. Make sure you answer specifically the question asked.
- Verify that the code you used to answer the question is included in an R chunk and appears in the knitted document (this should happen automatically)
- Verify that the output from each R chunk is included in your knitted document (this should happen automatically)
- For any figures (graphs) that you produce, you must include a figure caption. See the next section for instructions.
- If you use a graph to answer a question, refer to that graph in your answer

REMINDER: As you work on your document, save frequently, and also knit your document frequently.

TIP: You will often need/want to preview the document you are working on to ensure that everything is being formatted as you expected in the knitting process. The default setting is for the preview to open in a new pop-up window outside of RStudio, although you may find it useful to have it open in the Viewer panel within RStudio itself (particularly if you're only working with one screen!). To change the preview location, click the **Settings** icon in the toolbar and select the option to "Preview in Viewer Pane".

4.5 Setting up R Markdown for graphing

There are some special code chunk header arguments that need to be set in your Markdown document in order to:

- include proper **Figure captions** with your figures (a necessity!)
- set the dimensions of your figures

In each code chunk that includes code to construct a graph, your chunk header options should look like this in the top of the chunk, and there should be one blank line underneath before the main code starts. The following chunk produced the figure below. And recall that in the example below, the “bill_fig” text is simply the name I’ve given to this code chunk.

```
```{r billfig, fig.cap = "Bill depth histogram for Adelie penguins"}  
penguins %>%
 filter(species == "Adelie")
 ggplot(aes(bill_depth_mm))
 geom_histogram(color = "black", binwidth = 1)
 theme_minimal()
 xlab("Bill Depth (mm)")
 ylab("Frequency")
 ggtitle("Bill Depth Histogram for Adelie Penguins")
  ````
```

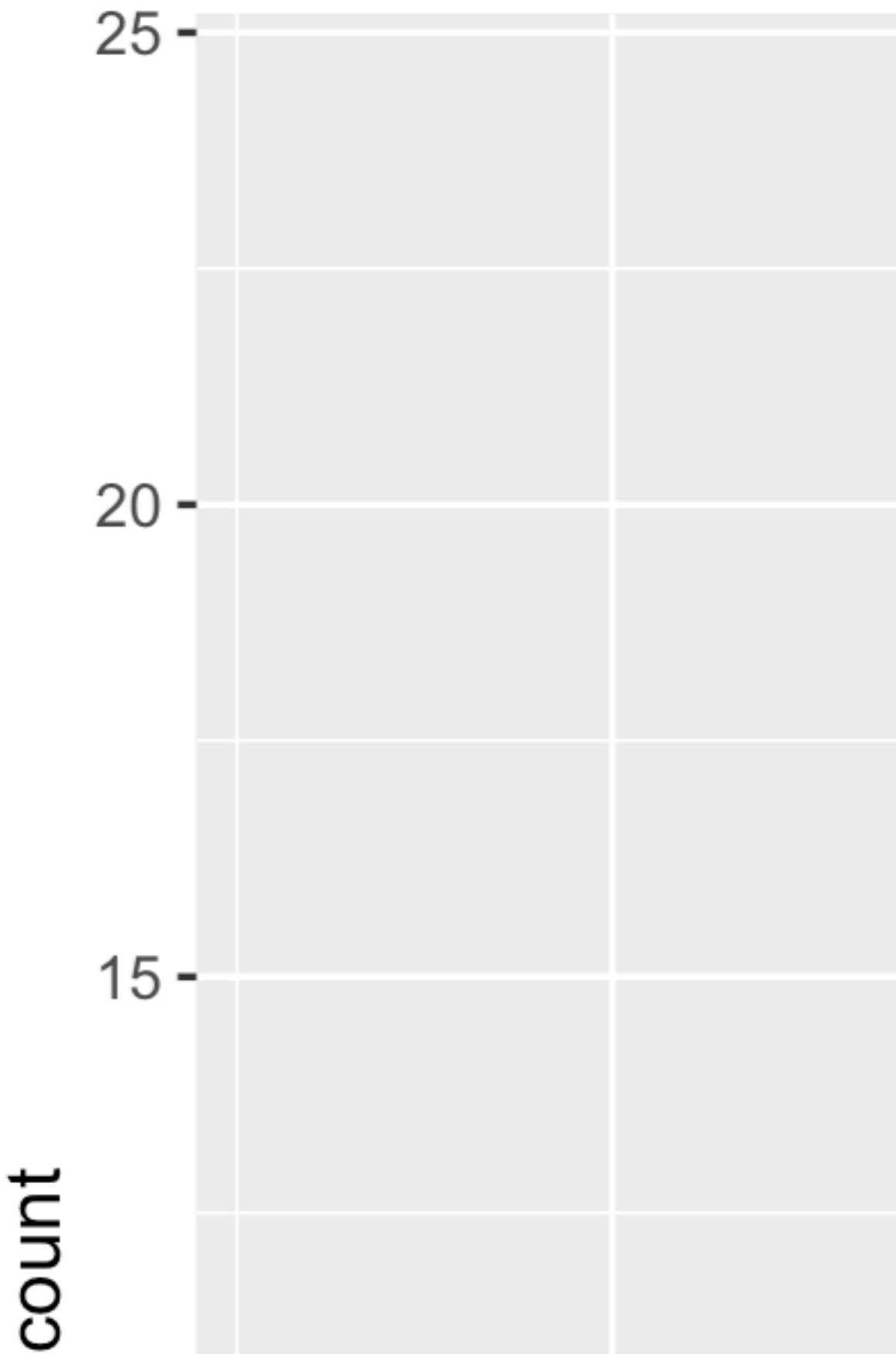


Figure 1: Histogram of bill depth among 151 Adelie penguins

The key parts are the “fig.cap =” argument, to which you provide the text you want to use as the figure caption. Then there’s the “fig.width =” and “fig.height =” arguments, which indicate the size of the figure when knitted. This will likely require some trial-and-error to get a good size, but start with the values in this example (4 for each).

Automated figure numbering

When you knit the document to PDF, you’ll see the “Figure X:” (with a number in place of the “X”) text was appended at the beginning of the caption phrase that was included in the chunk header, “Histogram of bill depth among 151 Adelie penguins”.

This feature - automatic figure numbering - was set within the header of the assignment markdown document:

The line “fig_caption = true” tells R Markdown and the knitting process to automatically include this part of a figure caption, “Figure X”, underneath each figure you produce. Where “X” will automatically be replaced with the appropriate number. For example, it will use “Figure 1:” for your first figure, “Figure 2” for your second, and so on.

The figure captions will only appear correctly once you knit to PDF; captions do not appear in the preview provided within the editor pane.

4.6 Example question / answer

Below is an example of how to answer a question. You haven’t yet learned some of the functions we use here, but follow along for now - it’s just an example!

There are almost *always* multiple coding approaches to get the right answer, some better than others. As long as your code and answer are accurate and make sense, you’ll get the marks!

Question 1. What are the minimum and maximum heights (variable name is “height_cm”) of students in the “students” dataset, which is available at this URL:

<https://raw.githubusercontent.com/ubco-biology/BIOL202/main/data/students2.csv>

As we learned in the [fictitious] “importing and exploring data” tutorial, I can use the `read_csv` function from the `readr` package (loaded with `tidyverse`) to download and import the dataset. It creates a “tibble” object, which here I name “students”:

```
1 ---  
2 title: "Practice Ass  
3 author: "Firstname L  
4 date: "Due date"  
5 output: pdf_document  
6 fig_caption: true  
7 ---  
8  
9 ```{r setup, include  
10 # DO NOT ALTER CODE  
11 knitr::opts_chunk$set(  
12 ````
```

Figure 4.3: Assignment markdown file header with figure caption option

```
students <- read_csv("https://raw.githubusercontent.com/ubco-biology/BIOL202/main/data/students.csv")
```

I also learned that it's a good idea to get an overview of the dataset as a first step after importing data. To do this, use the `skim_without_charts` function from the `skimr` package. I need to load that package first:

```
library(skimr)
```

Now skim:

```
skim_without_charts(students)
```

```
(#tab:skim_data)Data summary
```

```
Name
```

```
students
```

```
Number of rows
```

```
154
```

```
Number of columns
```

```
6
```

```
Column type frequency:
```

```
character
```

```
3
```

```
numeric
```

```
3
```

```
Group variables
```

```
None
```

```
Variable type: character
```

```
skim_variable
```

```
n_missing
```

```
complete_rate
```

```
min
```

```
max
```

```
empty
n_unique
whitespace
Dominant_hand
0
1
1
1
0
2
0
Dominant_foot
0
1
1
1
0
2
0
Dominant_eye
0
1
1
1
0
2
0
Variable type: numeric
skim_variable
n_missing
complete_rate
```

mean

sd

p0

p25

p50

p75

p100

height_cm

0

1

171.97

10.03

150

165.00

171.48

180.0

210.8

head_circum_cm

0

1

56.04

6.41

21

55.52

57.00

58.5

63.0

Number_of_siblings

0

1

1.71

```
1.05
0
1.00
2.00
2.0
6.0
```

This shows we have four character and three numeric variables, with 154 rows (observations) and 7 columns (variables) total.

We can use the `summary` function to get some basic descriptive statistics, including the minimum and maximum of numeric variables. The `summary` function is part of the base R package, so no additional packages need to be loaded.

We also use the `select` function from the `dplyr` package (which is loaded with `tidyverse`) to select which variable in the `students` tibble we wish to summarize.

The use of the “%>%” syntax is described in a later tutorial.

```
summary.height <- students %>%
  select(height_cm) %>%
  summary
summary.height

##    height_cm
##  Min.   :150.0
##  1st Qu.:165.0
##  Median :171.5
##  Mean   :172.0
##  3rd Qu.:180.0
##  Max.   :210.8
```

As shown in the output above, the minimum height was 150.0 cm and the maximum student height was 210.8 cm.

TIP: You'll note that functions and package names above are highlighted in grey. When writing in markdown, it's good practice to encompass function names and package names in single backticks, i.e. `tidyverse`. Backticks are typically located with the tilden (“~”) key on your keyboard.

4.7 Knitting your assignment to PDF

All assignments are to be submitted to Canvas as **PDF documents**.

As you learned in a previous tutorial, simply click on the “knit” button and select PDF. Your PDF file will adopt the same name as your markdown file, but it will have a “pdf” extension rather than an “Rmd” extension.

Once you have knitted your assignment to PDF, open up the PDF document, and verify that it looks correct, and be sure to check all your spelling, and that figures / tables are appropriately formatted etc...

If you need to fix something, close your PDF file, go back to your R Markdown document and do the edits there. Save, then knit again!

4.8 Submit your assignment

Once you’re happy with the PDF you created, it’s time to submit **both your markdown file (.Rmd file) and your PDF document** to the appropriate place in Canvas. Submitting both files enables markers to check where things went wrong if there are any knitting errors.

It’s a good idea also to verify afterwards that your PDF document did indeed upload correctly.

Chapter 5

Preparing and importing Tidy Data

Tutorial learning objectives

In this tutorial you will:

- Review how to format your data
 - Tidy data
 - Do's and Don'ts of naming variables and data entry
 - Wide versus Long format
- Learn how to save a file in CSV (comma-separated values) format
- Learn how to import a CSV file from a website into a `tibble` in R
- Learn how to import a CSV file from a local directory into a `tibble` in R
- Learn how to get an overview of the data and variables in your `tibble`

Importing data should be a straightforward task, but this is not always the case; sometimes data files are not formatted properly, so you need to be careful to check what you import.

Here, you'll learn (or review) how to format your own data files according to best practices, so that you or others will have no problems importing them.

It is assumed that if you are collecting data during a project, you'll likely enter them on your computer using a spreadsheet software program like Excel.

PAUSE: Before starting any data collection and data-entry, ask yourself: how should I organize the spreadsheet for data-entry?

The short answer: according to **TIDY** formatting principles...

5.1 Tidy data

Review the Biology Procedures and Guidelines document chapter on **Tidy data**.

There you'll learn how to arrange and format your data within a spreadsheet. The "Tidy" example provided would look like this in Excel:

If you'd like a longer, more in-depth read about "tidy data", see Hadley Wickham's "R for DataScience" online book, linked here.

If you wish to import and analyze data that have not been formatted according to **tidy** principles, then the most transparent and computationally reproducible way to reformat the data is to do so by coding *within R*, rather than using software like Excel. The process of reformatting / rearranging data is called **data wrangling**, and is mostly beyond the scope of this course.

If you're curious about data wrangling, the **dplyr** package, which is loaded with the **tidyverse** package, provides all the tools for data wrangling. Its associated cheatsheet is available at the package's vignette website.

5.2 Import a CSV file from a website

In previous tutorials we've already seen how to import CSV files from the web. As always, the first step is to load the **tidyverse** library, because it includes many packages and functions that are handy for both data import and data wrangling.

```
library(tidyverse)
```

One package that is loaded with the **tidyverse** package is **readr**, which includes the handy **read_csv** function.

You can view the help file for the **read_csv** function by typing this into your command console pane (bottom left) in RStudio:

```
?read_csv
```

You'll see that the function has many optional "arguments", but in general we can use the default values for these.

If the file you wish to import is located on the web, then we need to provide the "URL" (the web address) to the **read_csv** function. For example, in a previous tutorial you imported the "students.csv" dataset from the course GitHub website, as follows (and note that the URL address is provided in double quotation marks):

| | A | B | C |
|----|--------------|-----|-----|
| 1 | Site | Day | Tro |
| 2 | Mabel-lake | | 1 |
| 3 | Mabel-lake | | 2 |
| 4 | Mabel-lake | | 3 |
| 5 | Postill-lake | | 1 |
| 6 | Postill-lake | | 2 |
| 7 | Postill-lake | | 3 |
| 8 | Ellison-lake | | 1 |
| 9 | Ellison-lake | | 2 |
| 10 | Ellison-lake | | 3 |
| 11 | | | |

Figure 5.1: Tidy data in Excel

```
students <- read_csv("https://raw.githubusercontent.com/ubco-biology/BIOL202/main/data/students.csv")

## Rows: 154 Columns: 6
## -- Column specification -----
## Delimiter: ","
## chr (3): Dominant_hand, Dominant_foot, Dominant_eye
## dbl (3): height_cm, head_circum_cm, Number_of_siblings
##
## 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.
```

PAUSE: Did you get an error like this?

```
Error in read_csv("https://raw.githubusercontent.com/ubco-biology/BIOL202/main/data/students.csv") : could not find function "read_csv"
```

This happens if you forgot to load the tidyverse library!

The default behaviour of the `read_csv` function is to include a printout of the type of variable housed in each column of the dataset. For example, when importing the `students` dataset, the function gave a printout showing that there are three “character” variables (denoted `chr`) (R calls categorical variables character variables), and three `dbl` or double precision, floating point format numeric variables.

1. Take a minute to check out this good overview of how R handles numeric variables.

You can tell R to not provide this information by including the argument `show_col_types = FALSE` within the `read_csv` code. Like so:

```
students <- read_csv("https://raw.githubusercontent.com/ubco-biology/BIOL202/main/data/students.csv",
                      show_col_types = FALSE)
```

We have now imported the data and stored it in a local object called “`students`”. The object is called a “tibble”, which you can think of as a special kind of spreadsheet. More information on “tibbles” can be found [here](#).

Unless otherwise indicated, all CSV data files that we use in this course are stored at the same URL location, specifically: “<https://raw.githubusercontent.com/ubco-biology/BIOL202/main/data/>”. (**note** that clicking on this URL will not load a website). Thus, to import any CSV file you just need to copy that path, then append the appropriate file name to the end of the path. For example, the full path to access a CSV file called `birds.csv` file would be “<https://raw.githubusercontent.com/ubco-biology/BIOL202/main/data/birds.csv>”.

Often you'll need to import data from a locally stored CSV file, rather than from the web. You'll learn how to do this shortly. First: how does one create a CSV file?

5.3 Create a CSV file

Before we create or save any data files on our local computer, we should first create a directory (folder) called “rawdata” to store them.

Let's create the new directory in our “BIOL202_tutorials” working directory. To do this, use the `dir.create` function in R, as follows:

```
dir.create("rawdata")
```

Once you run this code, you'll see the new directory appear in the Files pane in the bottom-right of RStudio. It might look something like this:

The folder is called “rawdata” because the data stored there will be the unedited, raw version of the data, and any files therein should **NOT** be altered. Any changes or edits one makes to the datasets should be saved in new data files that are saved in a different folder called “output”, which we'll create later.

Let's create a data file to work with.

Steps to create and save a CSV file

- Open up Excel or any other spreadsheet software and enter values in the spreadsheet cells exactly as shown in the Excel example in Figure 5.1 from the Tidy Data section above
- You should have one row with the 3 variable names (“Site”, “Day”, “Trout_Caught”), one in each column, then nine rows of data
- Save the file as a CSV file by selecting (MAC) File > Save As > and in the drop down list: CSV UTF-8 (Comma separated), and Windows File > Save as type > CSV UTF-8 (comma separated)
- Name it “trout.csv”, and save it within the newly created “rawdata” folder

Now we're ready to try importing the data into a “tibble”.

5.4 Import a local CSV file

You can find additional help on importing different types of files at the Data Import Vignette, which includes a cheatsheet.

Steps to import a local CSV file

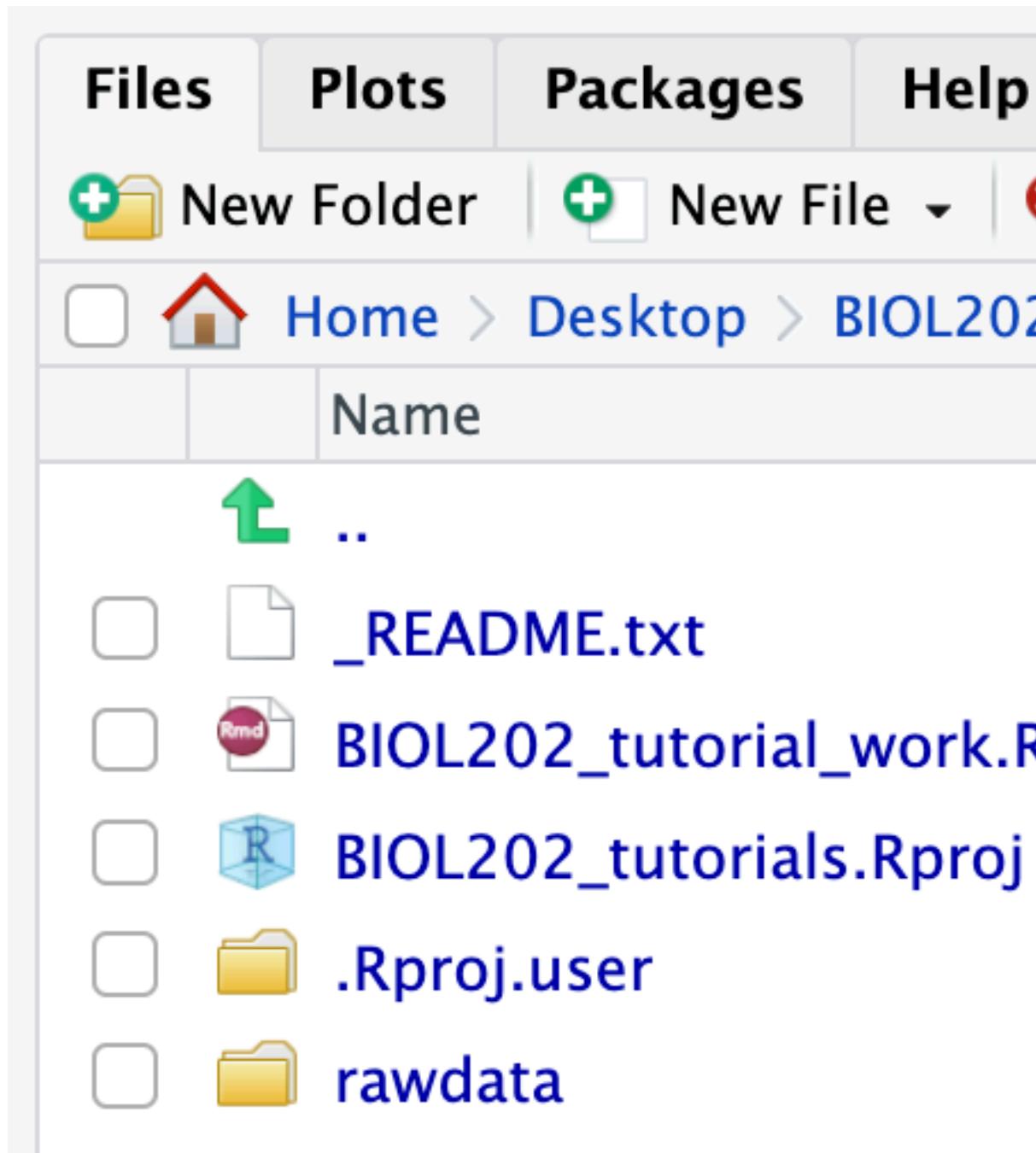


Figure 5.2: View of files with rawdata folder

We'll use the "trout.csv" file that we created previously.

And we'll make use of the `here` package that we were introduced to in an earlier tutorial.

Let's load the package:

```
library(here)
```

New tool: Pipes or "%>%" are implemented as part of the `magrittr` package, which is loaded with the `tidyverse`. In brief, pipes allow us to string together a series of functions, and we'll use them frequently in tutorials.

Here we'll use a pipe to help import the data file.

Let's see the code first, then explain after:

```
trout <- here("rawdata", "trout.csv") %>%
  read_csv()
```

- First we have the name of the object (a "tibble") that we wish to create, "trout".
- Then you see the assignment operator "<-", which tells R to assign whatever we're doing to the right of the operator to the object "trout".
- Then we have the `here` function, which is taking two inputs: the name of the directory we wish to get something from ("rawdata"), and then the name of the file we wish to do something with, here "trout.csv").
- Then we have a pipe "%>%", which tells R that we're not done coding yet - there's more to come on the next line...
- Lastly, we use the `read_csv` function, whatever came before the pipe is what is fed to the `read_csv` function.

Go ahead and run the chunk of code above to create the "trout" object.

Next we'll learn how to get an overview of the data stored in a tibble object.

5.5 Get an overview of a dataset

When you import data it is always a good idea to immediately get an overview of the data.

Key questions you want to be able to answer are:

- How many variables (columns) are there in the dataset?
- How many observations (rows) are in the dataset?

- Are there variables whose data are categorical? If so, which ones?
- Are there variables whose data are numerical? If so, which ones?
- Are there observations missing anywhere?

As we learned in the “Preparing and formatting assignments” tutorial, the `skimr` package has a handy function called `skim_without_charts` that provides a good overview of a data object. This is a rather long function name, and in fact the main function is called `skim`. However, by default, `skim` includes small charts in its output, and we don’t want that presently, hence the use of `skim_without_charts`.

Let’s load that package now:

```
library(skimr)
```

And get an overview of the `trout` dataset, again using the pipe approach:

```
trout %>%
  skim_without_charts()
```

Data summary

Name

Piped data

Number of rows

9

Number of columns

3

Column type frequency:

character

1

numeric

2

Group variables

None

Variable type: character

```
skim_variable  
n_missing  
complete_rate  
min  
max  
empty  
n_unique  
whitespace  
Site  
0  
1  
10  
12  
0  
3  
0  
Variable type: numeric  
skim_variable  
n_missing  
complete_rate  
mean  
sd  
p0  
p25  
p50  
p75  
p100  
Day  
0  
1  
2.00
```

```

0.87
1
1
2
3
3
Trout_Caught
0
1
2.78
1.79
0
1
3
4
5

```

A lot of information is provided in this summary output, so let's go through it:

- The Data Summary shows the name of the object, the number of rows, and the number of columns
- Column type frequency shows how many columns (variables) are of type “character”, which is equivalent to “categorical”, and how many are “numeric”
- Group variables shows if there are any variables that are specified as “grouping” variables, something we don't cover yet.
- Then it provides summaries of each of the variables, starting with the character or categorical variables, followed by the numeric variables
- Each summary includes a variety of descriptors, described next
- The “n_missing” descriptor tells you how many observations are missing in the given variable. In the “trout” dataset we don't have any missing values
- The “n_unique” descriptor for categorical variables indicates how many unique values (categories) are in that variable; for the “Site” variable in the “trout” dataset there are 3 unique values
- The descriptors for the numeric variables include the mean, standard deviation (sd), and the quantiles

Now you have what you need to answer each of the questions listed above!

One additional function that is useful during the overview stage is `head`. This function just gives you a view of the first 6 rows of the dataset:

```
# we can use head(trout), or the pipe approach:  
trout %>%  
  head()
```

```
## # A tibble: 6 x 3  
##   Site          Day Trout_Caught  
##   <chr>        <dbl>      <dbl>  
## 1 Mabel-lake     1          1  
## 2 Mabel-lake     2          3  
## 3 Mabel-lake     3          3  
## 4 Postill-lake   1          3  
## 5 Postill-lake   2          4  
## 6 Postill-lake   3          5
```

When you use `head` on a “tibble”, like we have here, it outputs another “tibble”, in this case 6 rows by 3 columns. But recall that the full “trout” dataset includes 9 rows and 3 columns.

5.6 Tutorial practice activities

This activity will help reinforce each of the key learning outcomes from this tutorial.

Steps

You are going to take measurements of the lengths (in mm) of your thumb, index finger, and middle finger on each hand; but don’t start measuring yet!

First:

- Create a new R Markdown document for this practice activity. This is where you’ll record the procedures you use for this practice activity
- As we’ve learned in previous tutorials, one of the first steps we should do is include a code chunk in the markdown document in which we load any packages we’ll need.
- Include a code chunk to load the packages used in the present tutorial
- Save the R Markdown document in your root “BIOL202_tutorials” directory, and provide it an appropriate file name.
- Open a new blank spreadsheet in Excel

Before taking the measurements, think about how you can make your measurement procedure reproducible. Where exactly are you measuring from and to on each digit? Are you using a ruler? What's your measurement precision? Whatever approach you take, make sure you type it out clearly in your R Markdown document, so that someone else could repeat it.

- Also before you start measuring, think about how you'll organize the data in the spreadsheet, including how many variables you'll have, what to name those variables, and how many rows or observations you'll have.

HINT: Even before you start measuring, most of your data sheet should be filled with values, and when you type in your 6 measurements, these should be entered in a single column.

- Once you've organized the spreadsheet, and even before you start entering the digit measurements, save it as a CSV file into your “rawdata” folder, remembering to use an appropriate file name
- Once you've typed out the methods in your markdown document, you can start taking measurements and recording them in the spreadsheet
- Once you've finished entering the data, save the spreadsheet again, then quit Excel.

Now would be a good time to create and edit a “`_README.txt`” file for your new “rawdata” folder.

Now you're ready to import the data into R.

- In your R Markdown document, include a code chunk to import the data.

Now you're ready to get an overview of the data.

- In your R Markdown document, include a code chunk to get an overview of the dataset.

Once you've confirmed that each of the code chunks work in your R Markdown document, you're ready to knit!

- Knit your document to PDF.

All done!

Visualizing and Describing Data

Chapter 6

Visualizing a single variable

Tutorial learning objectives

In this tutorial you will:

- Revisit how to import data and get an overview of a “tibble” object
- Learn how to construct a frequency table
- Learn how to include a table caption
- Learn how to visualize the frequency distribution of a single categorical variable using a bar graph
- Learn how to visualize the frequency distribution of a single numerical variable using a histogram
- Learn how to describe a histogram

Background

How to best visualize data depends upon (i) whether the data are **categorical** or **numerical**, and (ii) whether you’re visualizing one variable or associations between two variables (we don’t cover how to visualize associations between more than two variables). This tutorial focuses on visualizing a single variable.

When visualizing a single variable, we aim to visualize a **frequency distribution**. A frequency distribution is the frequency with which unique data values occur in the dataset.

- If the variable is categorical, we can visualize the frequency distribution using a **bar graph**
- If the variable is numeric, we visualize the frequency distribution using a **histogram**

In this tutorial you’ll learn to construct and interpret each of these types of visualization.

6.1 Load packages and import data

In this tutorial we will make use of `tidyverse` and its suite of packages, as well as the `skimr` package. You'll also use the `palmerpenguins` package that provides some penguin-related data to work with (see this website for more info). Lastly, you'll use the `knitr` package for helping create nice tables. The latter package should have come installed with RStudio, so check the “packages” tab in the bottom-right pane of RStudio to see if it's already installed. If it's not, then install it following the instructions you saw earlier.

```
library(tidyverse)
library(palmerpenguins)
library(skimr)
library(knitr)
library(janitor)
```

And we will use the following datasets in this tutorial:

- the `penguins` dataset that is available as part of the `palmerpenguins` package
- the `tigerdeaths.csv` file contains data associated with example 2.2A in the Whitlock and Schluter text
- the `birds.csv` file contains counts of different categories of bird observed at a marsh habitat

Unless otherwise indicated, all CSV data files that we use in this course are stored at the same URL location, specifically: “<https://raw.githubusercontent.com/ubco-biology/BIOL202/main/data/>”. Thus, to import any CSV file you just need to copy that path, then append the appropriate file name to the end of the path. For example, the full path to access the `birds.csv` file would be “<https://raw.githubusercontent.com/ubco-biology/BIOL202/main/data/birds.csv>”. And a previous tutorial showed you how to import using the `read_csv` function.

6.2 Get an overview of the data

The `penguins` object is a *tibble*, with each row representing a *case* and each column representing a *variable*. Tibbles can store a mixture of data types: numeric variables, categorical variables, logical variables etc... all in the same object (as separate columns). This isn't the case with other object types (e.g. matrices).

We'll get an overview of the data using the `skim_without_charts` function, as we learned in the Preparing and Importing Tidy Data tutorial:

```
penguins %>%  
  skim_without_charts()
```

(#tab:vis1_skim) Data summary

Name

Piped data

Number of rows

344

Number of columns

8

Column type frequency:

factor

3

numeric

5

Group variables

None

Variable type: factor

skim_variable

n_missing

complete_rate

ordered

n_unique

top_counts

species

0

1.00

FALSE

3

Ade: 152, Gen: 124, Chi: 68

island

0

1.00

FALSE

3

Bis: 168, Dre: 124, Tor: 52

sex

11

0.97

FALSE

2

mal: 168, fem: 165

Variable type: numeric

skim_variable

n_missing

complete_rate

mean

sd

p0

p25

p50

p75

p100

bill_length_mm

2

0.99

43.92

5.46

32.1

39.23

44.45
48.5
59.6
bill_depth_mm
2
0.99
17.15
1.97
13.1
15.60
17.30
18.7
21.5
flipper_length_mm
2
0.99
200.92
14.06
172.0
190.00
197.00
213.0
231.0
body_mass_g
2
0.99
4201.75
801.95
2700.0
3550.00
4050.00

```
4750.0
6300.0
year
0
1.00
2008.03
0.82
2007.0
2007.00
2008.00
2009.0
2009.0
```

Optionally, we can also get a view of the first handful of rows of a tibble by simply typing the name of the object on its own, and hitting return:

```
penguins
```

```
## # A tibble: 344 x 8
##   species island   bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
##   <fct>   <fct>        <dbl>        <dbl>        <int>        <int>
## 1 Adelie  Torgersen     39.1       18.7        181       3750
## 2 Adelie  Torgersen     39.5       17.4        186       3800
## 3 Adelie  Torgersen     40.3        18         195       3250
## 4 Adelie  Torgersen      NA         NA          NA        NA
## 5 Adelie  Torgersen     36.7       19.3        193       3450
## 6 Adelie  Torgersen     39.3       20.6        190       3650
## 7 Adelie  Torgersen     38.9       17.8        181       3625
## 8 Adelie  Torgersen     39.2       19.6        195       4675
## 9 Adelie  Torgersen     34.1       18.1        193       3475
## 10 Adelie Torgersen      42         20.2        190       4250
## # i 334 more rows
## # i 2 more variables: sex <fct>, year <int>
```

In a previous tutorial you learned the important information to look for when getting an overview of a dataset using the `skim_without_charts` function.

TIP It's important to check whether there are any missing values for any of the variables in your dataset. In the `penguins` dataset, you'll see from the `skim_without_charts` output that there are 344 cases (rows), but (as an example) there are 2 missing values for each of the 4 morphometric variables,

including body mass. **You need to take note of this so that you report the correct sample sizes in any table or figure captions!**

Once you have gotten an overview your dataset's structure and contents, the next order of business is always to *visualize* your data using graphs and sometimes tables.

1. **Import and data overview:** Following the instructions provided in previous tutorials, import the `tigerdeaths.csv` and `birds.csv` datasets, and get an overview of each of those datasets.

6.3 Create a frequency table

Sometimes when the aim is to visualize a single categorical variable, it's useful to present a *frequency table*. If your variable has more than, say, 10 unique categories, then this approach can be messy, and instead one should solely create a **bar graph**, as described in the next section.

Many straightforward operations like tabulation and calculating descriptive statistics can be done using the functionality of the `dplyr` package (see the cheatsheet here), which gets loaded as part of the `tidyverse` suite of packages.

Here, we'll use this functionality to create a *frequency table* for a categorical variable.

We'll demonstrate this using the `tigerdeaths.csv` dataset that you should have imported as part of a suggested activity in the previous section, using code like this:

TIP Some datasets that we use for tutorials need to be imported into an object in your workspace. This is the case with the `tigerdeaths` dataset, and the code for importing the data into a tibble is below. Other datasets, like the `penguins` dataset, exist within packages (`palmerpenguins`), and their objects are already created for you. Most of the time, and particularly for your lab assignments, you need to import a dataset and create an object, as we do below.

```
# here we import the data from a CSV file and put it into a "tibble" object called "tigerdeaths"
tigerdeaths <- read_csv("https://raw.githubusercontent.com/ubco-biology/BIO1L202/main/data/tigerdeaths.csv")
```

You would also have gotten overview of the data as part of the activity, using the `skim_without_charts` function. This would have shown that the `activity` variable is of type “character”, which tells us it is a **categorical** variable, and that it includes 9 unique categories. We also would have seen that there are 88 cases (rows) in the dataset.

Let's provide the code to generate the frequency table, using the pipes “%>%” approach we learned about in an earlier tutorial. We'll assign the output

to a new object that will hold the frequency table. We'll name the object "tiger.table". Note that we won't yet view the table here... we'll do that next.

We'll provide the code first, then explain it step-by-step after.

Here's the code for creating the frequency table and assigning it to a new object named "tiger.table":

```
tiger.table <- tigerdeaths %>%
  count(activity, sort = TRUE) %>%
  mutate(relative_frequency = n / sum(n)) %>%
  adorn_totals()
```

- The first line provides the name of the object (tibble) that we're going to create (here, "tiger.table"), and use the assignment operator ("<-") tell R to put whatever the output of our operation is into that object. The next part of the first line provides the name of the object that we're going to do something with, here "tigerdeaths". The "%>%" tells R that we're not done yet, and there's more lines of code to come.
- The second line uses the `count` function from the `dplyr` package to tally the unique values of a variable, in this case the "activity" variable. It also takes an argument "sort = TRUE", telling it to sort the counts in descending order (the default sort direction). Then another "%>%" to continue the code..
- The last line uses `mutate` function from the `dplyr` package that creates a new variable, and the arguments provided in the parentheses tells R what that variable should be called, here "relative_frequency", and then how to calculate it.
- The `n` in the third line is a function that tallies the sample size or count of all observations in the present category or group, and then the `sum(n)` sums up the total sample size. Thus, `n / sum(n)` calculates the relative frequency (equivalent to the proportion) of all observations that are within the given category
- the `adorn_totals` in the last line is a function from the `janitor` package that enables adding row and / or column totals to tables (see the help for this function for more details)

Try figuring out how you would change the last line of code in the chunk above so that the table showed the *percent* rather than the *relative frequency* of observations in each category

Now that we've created the frequency table, let's have a look at it.

In a supplementary tutorial, you'll find instructions on how to create fancy tables for output. Here, you'll learn the basics.

For our straightforward approach to tables with table headings (or captions), we'll use the `kable` function that comes with the `knitr` package, using the pipe approach:

```
tiger.table %>%  
  kable(caption = "Frequency table showing the activities of 88 people at the time they were attacked  
  and killed by tigers near Chitwan national Park, Nepal, from 1979 to 2006")
```

| activity | n | relative_frequency |
|-----------------------|----|--------------------|
| Grass/fodder | 44 | 0.500 |
| Forest products | 11 | 0.125 |
| Fishing | 8 | 0.091 |
| Hherding | 7 | 0.080 |
| Disturbing tiger kill | 5 | 0.057 |
| Fuelwood/timber | 5 | 0.057 |
| Sleeping in house | 3 | 0.034 |

```
Walking
```

```
3
```

```
0.034
```

```
Toilet
```

```
2
```

```
0.023
```

```
Total
```

```
88
```

```
1.000
```

The key argument to the `kable` function is the table object (which here we provide before the pipe), and the table heading (caption).

Notice that this produces a nicely formatted table with an appropriately worded caption. The argument “`digits = 3`” tells it to return numeric values to 3 digits in the table.

You now know how to create a frequency table for a categorical variable!

Your table caption won’t include a number (e.g. Table 1) until you actually knit to PDF. Be sure to check your PDF to ensure that the table captions show up, and are numbered!

2. **Frequency table:** Try creating a frequency table using the `birds` dataset, which includes data about four types of birds observed at a wetland.

6.4 Create a bar graph

We use a *bar graph* to visualize the frequency distribution for a single categorical variable.

We’ll use the `ggplot` approach with its `geom_bar` function to create a bar graph. The `ggplot` function comes with the `ggplot2` package, which itself is loaded as part of the `tidyverse`.

To produce the bar graph, we use a frequency table as the input. Thus, let’s repeat the creation of the “`tiger.table`” from the preceding section, **but this time we exclude the `adorn_totals` line of code**, because we don’t want the “total” row to be plotted in the bar graph.

```
tiger.table <- tigerdeaths %>%
  count(activity, sort = TRUE) %>%
  mutate(relative_frequency = n / sum(n))
```

Recall that the “tiger.table” is a sort of summary presentation of the “activity” variable:

```
tiger.table
```

```
## # A tibble: 9 x 3
##   activity           n relative_frequency
##   <chr>         <int>             <dbl>
## 1 Grass/fodder     44            0.5
## 2 Forest products  11            0.125
## 3 Fishing          8             0.09091
## 4 Herding          7             0.07955
## 5 Disturbing tiger kill  5             0.05682
## 6 Fuelwood/timber  5             0.05682
## 7 Sleeping in house 3             0.03409
## 8 Walking          3             0.03409
## 9 Toilet           2             0.02273
```

It shows the total counts (frequencies) of individuals in each of the nine “activity” categories.

And although in the code chunk below you’ll see that we provide an “x” and a “y” variable for creating the graph, remember that we’re really only visualizing a *single categorical variable*.

Let’s provide the code first, and explain after.

```
ggplot(data = tiger.table, aes(x = reorder(activity, n), y = n)) +
  geom_bar(stat = "identity") +
  ylab("Frequency") +
  xlab("Activity") +
  coord_flip() +
  theme_bw()
```

All figures produced using the `ggplot2` package start with the `ggplot` function. Then the following arguments:

- The tibble (or dataframe) that holds the data (“`data = tiger.table`”)
- An “aes” argument (which stands for “aesthetics”), within which one specifies the variables to be plotted; here we’re plotting the frequencies from

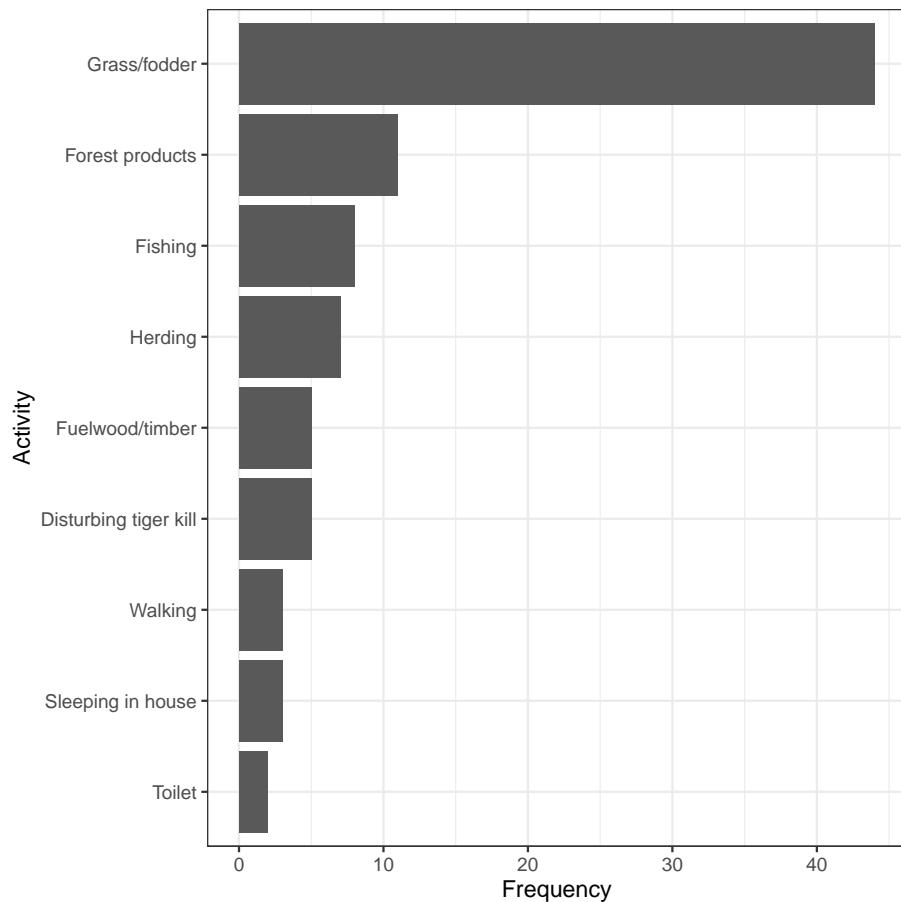


Figure 6.1: Bar graph showing the activities of 88 people at the time they were attached and killed by tigers near Chitwan national Park, Nepal, from 1979 to 2006

the “n” variable in the frequency table as the “y” variable, and the “activity” categorical variable as the “x” variable. To ensure the proper sorting of the bars, we use the `reorder` function, telling R to reorder the `activity` categories according to the frequencies in the `n` variable

- Then there’s a plus sign (“+”) to tell the `ggplot` function we’re not done yet with our graph - there are more lines of code coming (think of it as `ggplot`’s version of the “pipe”)
- Then the type of graph, which uses a function starting with “geom”; here we want a bar graph, hence `geom_bar`
- The `geom_bar` function has its own argument: “`stat = ‘identity’`” tells it just to make the height of the bars equal to the values provided in the “y” variable, here `n`.
- The `ylab` function sets the y-axis label
- The `xlab` function sets the x-axis label
- The `coord_flip` function tells it to rotate the graph horizontally; this makes it easier to fit the activity labels on the graph
- Then the `theme_bw` function indicates we want a simple black-and-white theme

There you have it: a nicely formatted bar graph!

REMINDER Don’t forget to include a good figure caption! Here’s a snapshot of the full code chunk that produced the bar graph above:

3. **Bar graph:** Try creating a bar graph using the `birds` dataset, which includes data about four types of birds observed at a wetland.

6.5 Create a histogram

A **histogram** uses the area of rectangular bars to display the frequency distribution (or relative frequency distribution) of a numerical variable.

We’ll use `ggplot` to create a histogram, and we’ll again use the `penguins` dataset.

We’ll give the code first, then explain below:

```
ggplot(data = penguins, aes(x = body_mass_g)) +
  geom_histogram(colour = "black", fill = "lightgrey") +
  xlab("Body mass (g)") +
  ylab("Frequency") +
  theme_bw()

## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```

```
```{r fig.cap = "Bar graph showing the frequency of different activities at Chitwan National Park, Nepal, from tiger table."}

ggplot(data = tiger.table) +
 geom_bar(stat = "identity") +
 ylab("Frequency") +
 xlab("Activity") +
 coord_flip() +
 theme_bw()

```
```

Figure 6.2: Example code chunk for producing a good bar graph

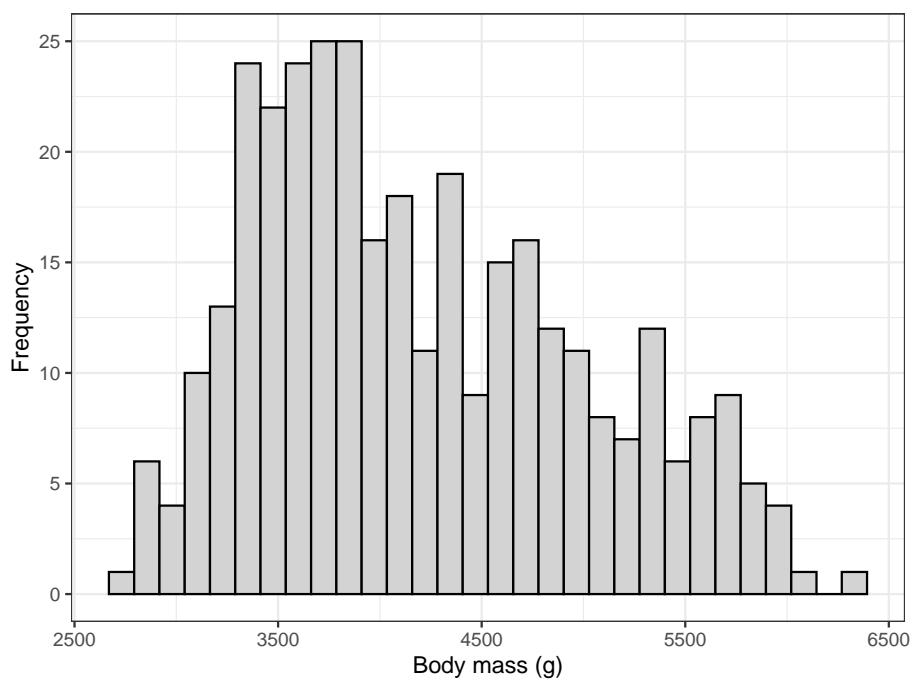


Figure 6.3: Histogram of body mass (g) for 342 penguins

The syntax follows what was seen above when creating a bar graph, but:

- Here we have only a single variable “x” variable, `body_mass_g` to provide the `aes` function.
- We use the `geom_histogram` function, which has its own optional arguments:
 - the “color” we want the outlines of each bar in the histogram to be
 - the “fill” colour we want the bars to be

You can also specify the “bin width” that `geom_histogram` uses when generating the histogram. Notice above that we got a message stating:

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```

It’s telling us that there’s probably a better bin width to use. The trick is to not have too small a bin width, such that you end up with too many bars in your histogram (giving too much detail in the frequency distribution), and to not have too large a bin width such that you have too few bars in your histogram (giving too little detail).

The `hist` function that comes with base R (so no need to load a package) has an algorithm that typically chooses good bin widths. To remove some of the subjectivity from this procedure, let’s leverage that function to figure out the best bin widths.

We’ll provide the code then explain after:

```
penguins.hist.info <- hist(penguins$body_mass_g, plot = FALSE)
```

In the chunk above, we have:

- The “`penguins.hist.info`” is the name we’ll give to the object we’re going to create, and the assignment operator “`<-`” is telling R to put whatever the output from the function is into that new object
- The `hist` function takes the variable you want to generate a histogram function for. And in this case, it’s the `body_mass_g` variable in the `penguins` tibble.
- The dollar sign allows you to specify the tibble name along with the variable name: “`penguins$body_mass_g`”.
- The “`plot = FALSE`” tells the function we don’t wish to produce the actual histogram, and as a consequence the function instead gives us the information that would have gone into creating the histogram, including for example the break points for the histogram bins. It packages this information in the form of a “list”, which is one type of object.

Let's look at the info stored in the list object:

```
penguins.hist.info
```

```
## $breaks
## [1] 2500 3000 3500 4000 4500 5000 5500 6000 6500
##
## $counts
## [1] 11 67 92 57 54 33 26 2
##
## $density
## [1] 6.432749e-05 3.918129e-04 5.380117e-04 3.333333e-04 3.157895e-04
## [6] 1.929825e-04 1.520468e-04 1.169591e-05
##
## $mids
## [1] 2750 3250 3750 4250 4750 5250 5750 6250
##
## $xname
## [1] "penguins$body_mass_g"
##
## $equidist
## [1] TRUE
##
## attr(,"class")
## [1] "histogram"
```

We won't worry about all the information provided here. Instead just notice that the first variable in the list is “breaks”. Specifically, this provides us all the “break points” for the histogram for the given variable; break points are the values that delimit the bins for the histogram bars.

That's the information we can use to get the ideal bin width: the difference between consecutive breaks is our desired bin width!

In this example it's easy to see that the bin width was 500. But lets provide code to calculate it and thus make sure it's reproducible. We simply need to calculate the difference between any two consecutive break points (they will all be equal in magnitude):

```
penguins.hist.info$breaks[2] - penguins.hist.info$breaks[1]
```

```
## [1] 500
```

The above code simply asks R to calculate the difference (using the subtraction sign) between the second element of the “breaks” variable, denoted using the square brackets “breaks[2]”, and the first element “breaks[2]”.

And R returns 500. That's the bin width we want to use!

So let's edit the original histogram code to include the "binwidth" argument in the `geom_histogram` function, as follows:

```
ggplot(data = penguins, aes(x = body_mass_g)) +
  geom_histogram(binwidth = 500, colour = "black", fill = "lightgrey") +
  xlab("Body mass (g)") +
  ylab("Frequency") +
  theme_bw()
```

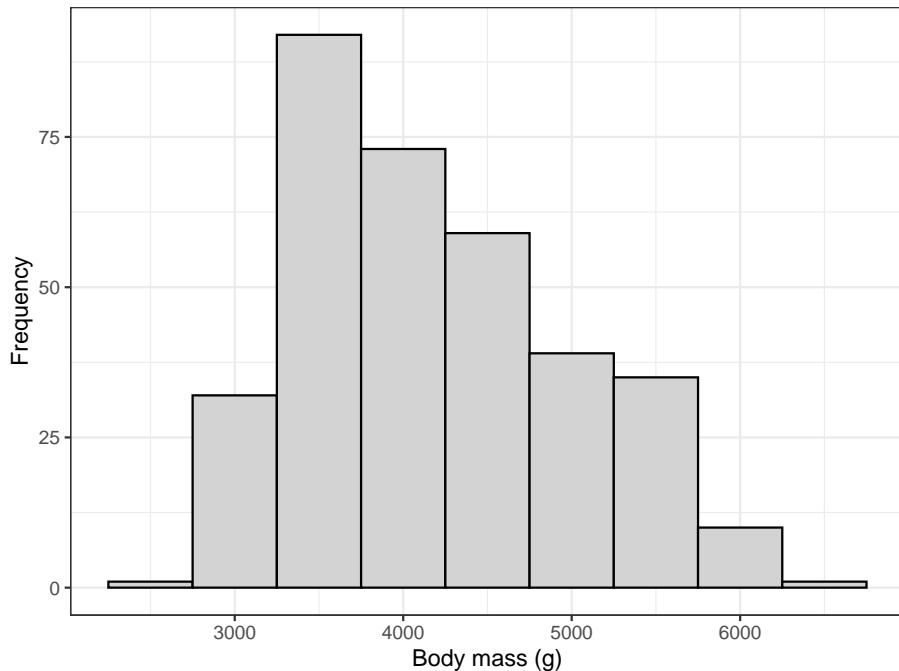


Figure 6.4: Histogram of body mass (g) for 342 penguins

There we go! Now we need to learn how to describe and interpret a histogram...

6.6 Describing a histogram

Things to note in your description of a histogram:

- Is it roughly symmetric or is it negatively or positively skewed?
- Is it roughly bell-shaped?

- Outliers - are there observations (bars) showing up far from the others?
- Are there multiple modes?

So, let's look again at the penguin body mass histogram, and provide a description thereafter:

```
ggplot(data = penguins, aes(x = body_mass_g)) +
  geom_histogram(binwidth = 500, colour = "black", fill = "lightgrey") +
  xlab("Body mass (g)") +
  ylab("Frequency") +
  theme_bw()
```

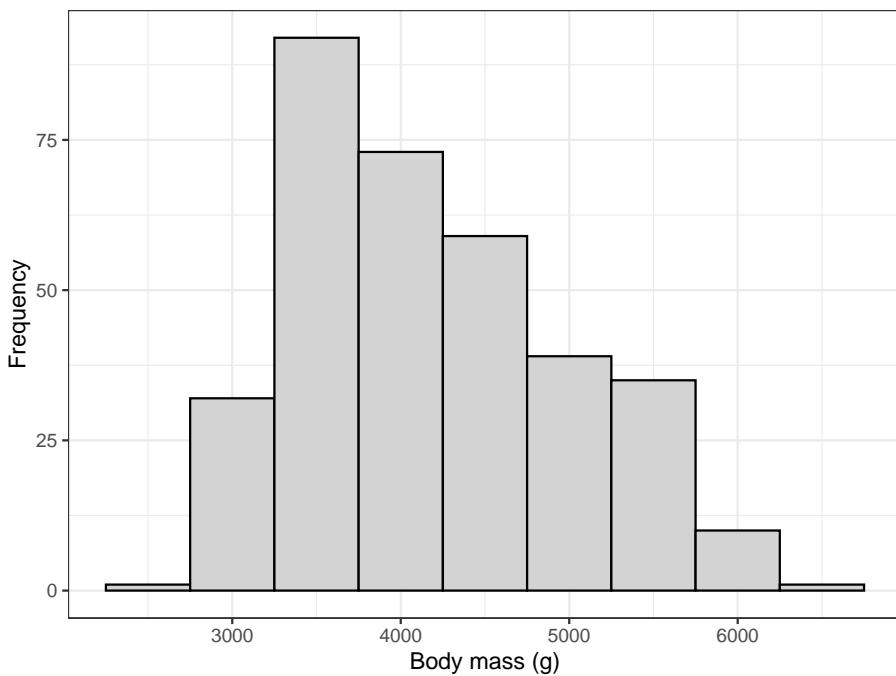


Figure 6.5: Histogram of body mass (g) for 342 penguins

The histogram is moderately positively skewed and thus asymmetric, with a single mode near 3500g. There are no obvious outliers in the distribution.

1. **Histogram:** Create a histogram of the bill lengths of penguins in the `penguins` dataset, and include an appropriate figure caption. Then provide a description of what you see.

Chapter 7

Describing a single variable

Tutorial learning objectives

In this tutorial you will:

- Learn how to calculate the main descriptor of a categorical variable: the **proportion**
- Learn how to calculate measures of centre and spread for a single numerical variable

7.1 Load packages and import data

Let's load some packages first:

```
library(tidyverse)
library(palmerpenguins)
library(skimr)
library(knitr)
library(janitor)
```

And we're introducing a new package called **naniar**, which helps us deal more easily with missing values in datasets.

You may need to install that package (**recall** you only do this once!). Consult a previous tutorial if you forget how.

Once you've installed it, load it:

```
library(naniar)
```

We will use the following datasets in this tutorial:

- the `birds.csv` file contains counts of different categories of bird observed at a marsh habitat
- the `penguins` dataset that is available as part of the `palmerpenguins` package

```
birds <- read_csv("https://raw.githubusercontent.com/ubco-biology/BIO1202/main/data/birds.csv")

## Rows: 86 Columns: 1
## -- Column specification -----
## Delimiter: ","
## chr (1): type
##
## 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.
```

7.2 Describing a categorical variable

The *proportion* is the most important descriptive statistic for a categorical variable. It measures the fraction of observations in a given category within a categorical variable.

For example, the `birds.csv` file has a single variable called `type` that includes tallies (frequencies) of each of four categories of bird observed at a marsh habitat.

```
birds
```

```
## # A tibble: 86 x 1
##   type
##   <chr>
##   1 Waterfowl
##   2 Predatory
##   3 Predatory
##   4 Waterfowl
##   5 Shorebird
##   6 Waterfowl
##   7 Waterfowl
##   8 Songbird
##   9 Predatory
##  10 Waterfowl
## # i 76 more rows
```

The *proportion* of birds belonging to a given category is the same as the *relative frequency* of birds belonging to a given category.

In a previous tutorial, using the `tigerdeaths` dataset, we learned how to create a frequency table that included relative frequencies.

Let's use the same approach for the `birds` dataset. First we create the frequency table, then we display the table with an appropriate heading:

```
birds.table <- birds %>%
  count(type, sort = TRUE) %>%
  mutate(relative_frequency = n / sum(n)) %>%
  adorn_totals()
```

NOTE If there are missing values (“NA”) in the categorical variable, the preceding code will successfully enumerate those and create an “NA” category in the frequency table.

Now display the table:

```
birds.table %>%
  kable(caption = "Frequency table showing the frequencies of each of four types of bird observed")
```

Frequency table showing the frequencies of each of four types of bird observed at a marsh habitat (N = 86)

type

n

relative_frequency

Waterfowl

43

0.500

Predatory

29

0.337

Shorebird

8

0.093

Songbird

6

0.070

Total

86

1.000

We can see, for example, that the proportion (relative frequency) of birds belonging to the “Predatory” category was 0.3372093.

We calculate proportions (relative frequencies) using the simple formula:

$$\hat{p} = \frac{n_i}{N}$$

Where

n_i

is the frequency of observations in the given category of interest i , and N is total number of observations (sample size) across all categories.

Reminder Proportions, and thus relative frequencies, must be between 0 and 1.

7.3 Describing a numerical variable

Numeric variables are described with measures of **centre** and **spread**.

Before calculating descriptive statistics for a numeric variable, it is advisable to visualize its frequency distribution first. Why? Because characteristics of the frequency distribution will govern which measures of centre and spread are more reliable or representative.

- If the frequency distribution is roughly symmetric and does not have any obvious outliers, then the **mean** and the **standard deviation** are the preferred measures of centre and spread, respectively
- If the frequency distribution is asymmetric and / or has outliers, the **median** and the **inter-quartile range** (IQR) are the preferred measures of centre and spread

It is often the case, however, that all four measures are presented together.

New tool Introducing the **summarise** function.

The **dplyr** package, which is loaded with the **tidyverse**, has a handy **summarise** (equivalently **summarize**) function for calculating descriptive statistics.

Check out its help file by copying the following code into your command console:

```
?summarise
```

Let's use the `penguins` dataset for our demonstrations.

The first step is to visualize the frequency distribution. Given that this is a numeric variable, we do this using a histogram, as we learned in a previous tutorial.

```
ggplot(data = penguins, aes(x = body_mass_g)) +
  geom_histogram(binwidth = 500, colour = "black", fill = "lightgrey") +
  xlab("Body mass (g)") +
  ylab("Frequency") +
  theme_bw()
```

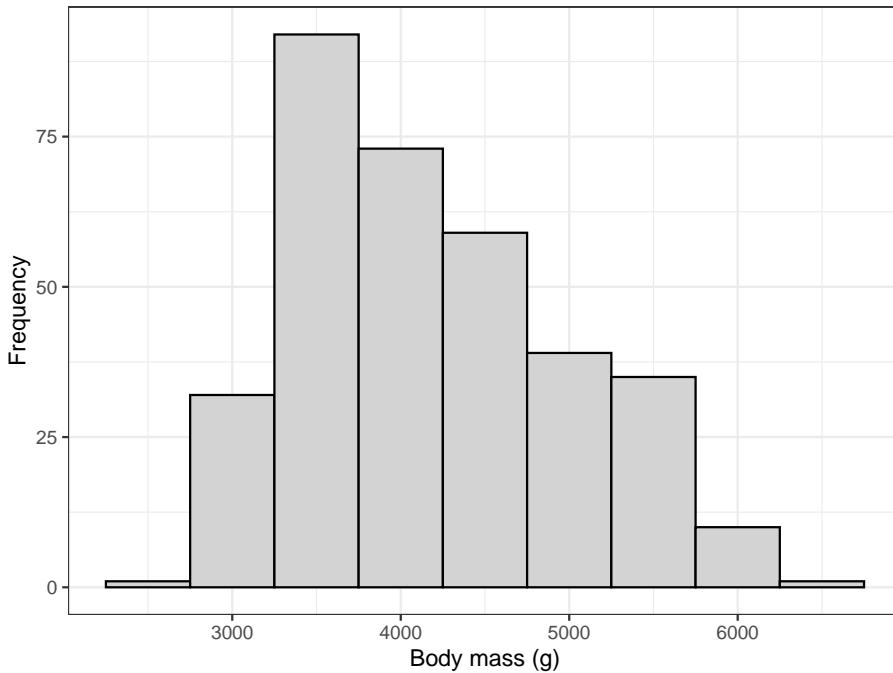


Figure 7.1: Histogram of body mass (g) for 342 penguins

We are reminded that the distribution of body mass is moderately positively skewed and thus asymmetric, with a single mode near 3500g. There are no obvious outliers in the distribution.

This means that the **median** and **IQR** should be the preferred descriptors of centre and spread, respectively.

7.3.1 Calculating the median & IQR

So let's calculate the median and IQR of body mass for all penguins. Let's provide the code, then explain after:

```
penguins %>%
  summarise(
    median_body_mass_g = median(body_mass_g),
    IQR_body_mass_g = IQR(body_mass_g)
  )
```

Uh oh! If you tried to run this code, it would have given you an error:

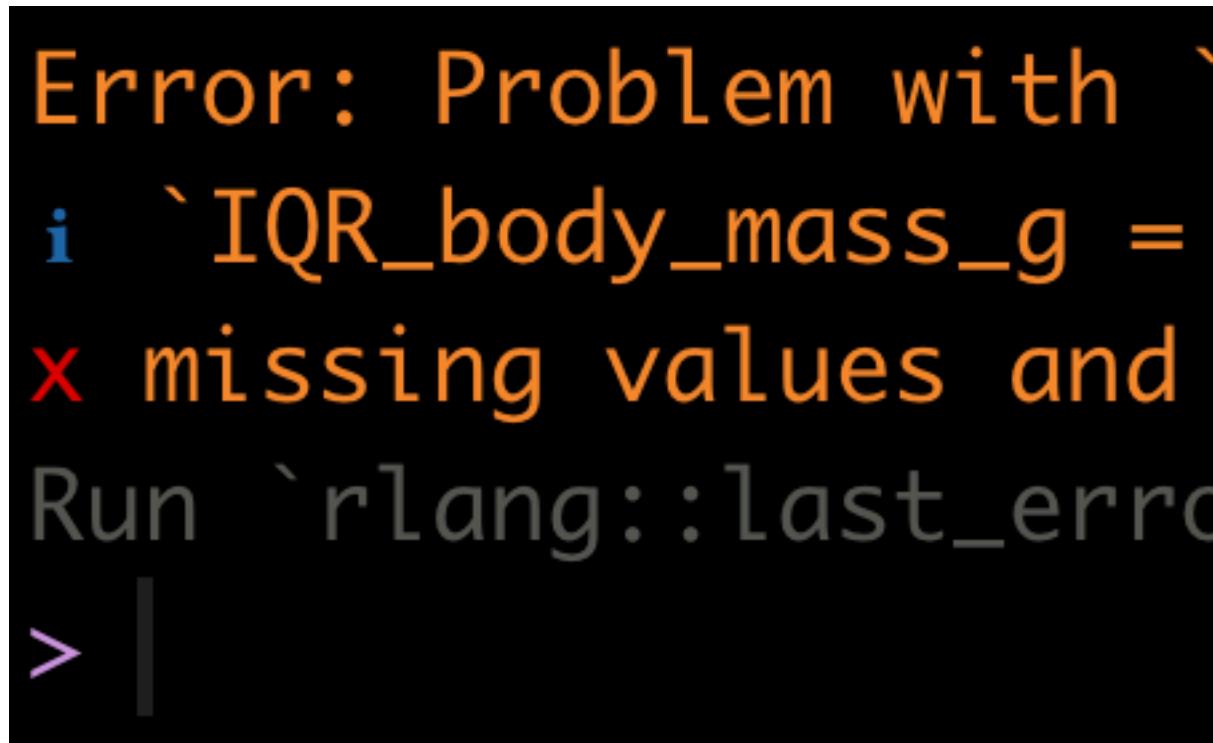


Figure 7.2: Error when functions encounter 'NA' values

We forgot that when we previously got an overview of the `penguins` dataset we discovered there were missing values ("NA" values)!

TIP If there are "NA" values in the variable being analyzed, some R functions, such as the function `median` or `mean`, will simply return "NA". To remedy this, we use the "na.rm = TRUE" argument.

Let's try our code again, adding the “na.rm = TRUE” argument. And note that the key functions called within the `summarise` function are `median` and `IQR` (case sensitive!).

```
penguins %>%
  summarise(
    Median = median(body_mass_g, na.rm = TRUE),
    InterQR = IQR(body_mass_g, na.rm = TRUE)
  )

## # A tibble: 1 x 2
##   Median InterQR
##     <dbl>    <dbl>
## 1     4050     1200
```

In the preceding code chunk, we have:

- The name of the tibble (here `penguins`) being used in the subsequent functions
- A pipe “%>%” to tell R we’re not done coding
- The `summarise` function (`summarize` will work too), telling R we’re going to calculate a new variable
- The name we’ll give to the first variable we’re creating, here we call the variable “Median” (the “M” is capitalized to distinguish this variable name from the function `median`)
- And we define how to calculate the “Median”, here using the `median` function
- We feed the variable of interest from the `penguins` tibble, “body_mass_g”, to the `median` function, along with the argument “na.rm = TRUE”
- We end the line with a comma, telling R that we’re not done providing arguments to the `summarise` function
- We do the same for the inter-quartile range variable we’re creating called “InterQR”, calculating the value using the `IQR` function, and this time no comma at the end of the line, because this is the last argument being provided to the `summarise` function
- We close out the parentheses for the `summarise` function

7.3.2 Calculating the mean & standard deviation

Although the median and IQR are the preferred descriptors for the `body_mass_g` variable, it is nonetheless commonplace to report the mean and standard deviation also.

Let’s do this, and while we’re at it, include even more descriptors to illustrate how they’re calculated.

This time we'll put the output from our `summarise` function into a table, and then present it in a nice format, like we learned how to do for a frequency table.

Let's create the table of descriptive statistics first, a tibble called "penguins.descriptors", and we'll describe what's going on after (NOTE this code chunk was edited slightly on Sept. 30, 2021):

```
penguins.descriptors <- penguins %>%
  summarise(
    Mean = mean(body_mass_g, na.rm = T),
    SD = sd(body_mass_g, na.rm = T),
    Median = median(body_mass_g, na.rm = T),
    InterQR = IQR(body_mass_g, na.rm = T),
    Count = n() - naniar::n_miss(body_mass_g),
    Count_NA = naniar::n_miss(body_mass_g))
```

The first 4 descriptive statistics are self-explanatory based on their variable names.

The last two: "Count" and "Count_NA" are providing the total number of complete observations in the `body_mass_g` variable (thus the number of observations that went into calculating the descriptive statistics), and then the total number of missing values (NAs) in the variable, respectively.

The last two lines of code above require further explanation:

This code: `Count = n() - naniar::n_miss(body_mass_g)`) tells R to first tally the total sample size using the `n()` function, then to subtract from that the total number of missing values, which is calculated using the `n_miss` function from the `naniar` package.

The double colons in `naniar::n_miss(body_mass_g)` indicates that the function `n_miss` comes from the `naniar` package. This syntax, which we have not used previously, provides a failsafe way to run a function even if the package is not presently loaded.

The same coding approach is used in the last line: `Count_NA = naniar::n_miss(body_mass_g)`.

TIP It is important to calculate the total number of complete observations in the variable of interest, because, as described in the Biology Procedures and Guidelines document, this number needs to be reported in figure and table headings.

Now let's show the table of descriptive statistics, using the `kable` function we learned about in a previous tutorial.

```
penguins.descriptors %>%
  kable(caption = "Descriptive statistics of measurements of body mass (g) for 342 penguins", digits = 2)
```

Descriptive statistics of measurements of body mass (g) for 342 penguins

Mean

SD

Median

InterQR

Count

Count_NA

4201.754

801.955

4050

1200

342

2

In another tutorial we'll learn how to present the table following all the guidelines in the Biology Guidelines and Procedures document, including, for example, significant digits. For now, the preceding table is good!

1. **Descriptive statistics:** Create a histogram and table of descriptive statistics for the “flipper_length_mm” variable in the `penguins` dataset.

7.4 Describing a numerical variable grouped by a categorical variable

In this tutorial you'll learn how to calculate descriptive statistics for a numerical variable grouped according to categories of a categorical variable.

For example, a common scenario in biology is to want to calculate and report the mean and standard deviation of a response variable for different “treatment groups” in an experiment. (More commonly we would report the mean and standard error, but that's for a later tutorial!).

It is straightforward to modify the code we used in the preceding tutorial to do what we want.

Specifically, we use the `group_by` function from the `dplyr` package to tell R to do the calculations on the observations within each category of the grouping variable.

For example, let's describe penguin body mass grouped by "species".

We'll create a new tibble object called "penguins.descriptors.byspecies", and we insert one line of code using the `group_by` function, and telling R which categorical variable to use for the grouping (here, "species"):

```
penguins.descriptors.byspecies <- penguins %>%
  group_by(species) %>%
  summarise(
    Mean = mean(body_mass_g, na.rm = T),
    SD = sd(body_mass_g, na.rm = T),
    Median = median(body_mass_g, na.rm = T),
    InterQR = IQR(body_mass_g, na.rm = T),
    Count = n() - naniar::n_miss(body_mass_g),
    Count_NA = naniar::n_miss(body_mass_g))
```

It's that simple!

Let's have a look at the output:

```
penguins.descriptors.byspecies
```

| | species | Mean | SD | Median | InterQR | Count | Count_NA |
|---|-----------|-------|-------|--------|---------|-------|----------|
| 1 | Adelie | 3701. | 458.6 | 3700 | 650 | 151 | 1 |
| 2 | Chinstrap | 3733. | 384.3 | 3700 | 462.5 | 68 | 0 |
| 3 | Gentoo | 5076. | 504.1 | 5000 | 800 | 123 | 1 |

1. Use the `kable` function to output this new tibble in a nice format

Chapter 8

Visualizing associations between two variables

Tutorial learning objectives

In this tutorial you will:

- Learn how to visualize associations between two categorical variables using a contingency table
- Learn how to visualize associations between two categorical variables graphically
- Learn how to visualize associations between two numerical variables
- Learn how to visualize associations between a numerical response variable and a categorical explanatory variable

Background

The type of graph that is most suitable for visualizing an association between two variables depends upon the type of data being visualized:

- If both variables are categorical, we can visualize the association in a table called a **contingency table**, or we can visualize the association graphically using a **grouped bar chart** or a **mosaic plot**
- If both variables are numeric, we visualize the association graphically using a **scatterplot**
- If the response variable is numerical and the explanatory variable is categorical, we visualize the association graphically using a **strip chart**, **boxplot**, or variations on these
- We do not discuss the scenario where the response variable is categorical and the explanatory variable is numerical

In this tutorial you'll learn to construct and interpret each of these types of visualization. In later tutorials you'll learn how to conduct statistical analyses of these associations.

8.1 Load packages and import data

Let's load some familiar packages first:

```
library(tidyverse)
library(palmerpenguins)
library(knitr)
library(skimr)
```

We also need the `janitor` package, the `ggbasicplots` package, and the `ggExtra` packages, and these are likely to be new to you. Check whether these packages are installed under the “`packages`” tab in the bottom-right panel in RStudio. If they are not yet installed, then install them. **Only install them once on your computer!**

Load the packages:

```
library(janitor)  
library(ggmosaic)  
library(ggExtra)
```

Import Data

We'll again make use of the `penguins` dataset, which gets loaded as a “tibble” object with the `palmerpenguins` package.

Import the `locusts.csv` data, which are described in the Whitlock & Schluter text, Figure 2.1-2. We'll create a tibble called `locust`.

Import the `bird.malaria.csv` data, which are described in the Whitlock & Schluter text, Example 2.3A (p. 40). We'll create a tibble called `bird.malaria`.

```
bird.malaria <- read_csv("https://raw.githubusercontent.com/ubco-biology/BIO1202/main/data/bird_ma
## Rows: 65 Columns: 3
## -- Column specification -----
## Delimiter: ","
## chr (2): treatment, response
## dbl (1): bird
##
## 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.
```

1. Get an overview of the `locust` and `bird.malaria` tibbles.

8.2 Visualizing association between two categorical variables

We'll cover three ways to visualize associations between two categorical variables:

- a **contingency table**
- a **grouped bar graph**
- a **mosaic plot**

8.2.1 Constructing a contingency table

New tool The `tabyl` function from the `janitor` package is useful for creating contingency tables, or more generally, cross-tabulating frequencies for multiple categorical variables.

You can check out more about the `tabyl` function at this vignette.

Let's use the `bird.malaria` dataset for our demonstration.

If you got an overview of the dataset, as suggested as part of the activity in the preceding section, you would have seen that the `bird.malaria` tibble includes two categorical variables: `treatment` and `response`, each with 2 categories.

The dataset includes 65 rows. Each row corresponds to an individual (unique) bird. Thirty of the birds were randomly assigned to the “Control” treatment group, and 35 were randomly assigned to the “Egg removal” treatment group.

The `response` variable includes the categories “Malaria” and “No Malaria”, indicating whether the bird contracted Malaria after the treatment.

Our goal is to visualize the frequency of birds that fall into each of the four unique combinations of category:

- Control + No Malaria
- Control + Malaria
- Egg removal + No Malaria
- Egg removal + Malaria

More specifically, we are interested in comparing the incidence of malaria among the Control and Egg removal treatment groups. We'll learn in a later tutorial how to conduct this comparison statistically.

Let's provide the code, then explain after. We'll again make use of the `kable` function from the `knitr` package to help present a nice table. So first we create the table ("bird.malaria.table"), then in a later code chunk we'll output a nice version of the table using the `kable` function.

First create the basic contingency table:

```
bird.malaria.freq <- bird.malaria %>%
  tabyl(treatment, response)
```

Code explanation:

- the first line is telling R to assign any output from our commands to the object called "bird.malaria.freq"
- the first line is also telling R that we're using the `bird.malaria` object as input to our subsequent functions, and the pipe (%>%) tells R there's more to come.
- the second line uses the `tabyl` function, and we provide it with the names of the variables from the `bird.malaria` object that we want to use for tabulating frequencies. Here we provide the variable names "treatment", and "response"

Let's look at the table:

```
bird.malaria.freq
```

```
##      treatment Malaria No Malaria
##      Control      7       28
## Egg removal     15       15
```

It is typically a good idea to also include the row and column totals in a contingency table.

To do this, we use the `adorn_totals` function, from the `janitor` package, as follows, and we'll create a new object called "bird.malaria.freq.totals":

8.2. VISUALIZING ASSOCIATION BETWEEN TWO CATEGORICAL VARIABLES103

```
bird.malaria.freq.totals <- bird.malaria %>%
  tabyl(treatment, response) %>%
  adorn_totals(where = c("row", "col"))
```

- the last line tells the `adorn_totals` function that we want to add the row and column totals to our table

Now let's see what the table looks like before using the `kable` function. To do this, just provide the name of the object:

```
bird.malaria.freq.totals
```

```
##   treatment Malaria No Malaria Total
##   Control      7       28     35
## Egg removal    15       15     30
##   Total        22       43     65
```

Now let's use the `kable` function to improve the look, and add a table heading.

```
bird.malaria.freq.totals %>%
  kable(caption = "Contingency table showing the incidence of malaria in female great tits in relation to experimental treatment")
```

Contingency table showing the incidence of malaria in female great tits in relation to experimental treatment

treatment

Malaria

No Malaria

Total

Control

7

28

35

Egg removal

15

15

30

Total

22

43

65

Relative frequencies

Often it is useful to also present a contingency table that shows the *relative frequencies*. However, it's important to know how to calculate those relative frequencies.

For instance, recall that in this malaria example, we are interested in comparing the incidence of malaria among the Control and Egg removal treatment groups. Thus, we should calculate the relative frequencies using the **row totals**. This will become clear when we show the table.

We can get relative frequencies, which are equivalent to proportions, using the `adorn_percentages` function (the function name is a misnomer, because we're calculating proportions, not percentages!), and telling R to use the row totals for the calculations.

First create the new table object “`bird.malaria.prop`”:

```
bird.malaria.prop <- bird.malaria %>%
  tabyl(treatment, response) %>%
  adorn_percentages("row")
```

Now present it using `kable`:

```
bird.malaria.prop %>%
  kable(caption = "Contingency table showing the relative frequency of malaria in female great tits in relation to experimental treatment")
```

Contingency table showing the relative frequency of malaria in female great tits in relation to experimental treatment

treatment

Malaria

No Malaria

Control

0.2

0.8

Egg removal

0.5

0.5

8.2.2 Constructing a grouped bar graph

To construct a grouped bar graph, we first need *wrangle* (reformat) the data to be in the form of a **frequency table**.

Let's revisit what the `bird.malaria` tibble looks like:

```
bird.malaria
```

```
## # A tibble: 65 x 3
##   bird treatment response
##   <dbl> <chr>     <chr>
## 1     1 Control    Malaria
## 2     2 Control    Malaria
## 3     3 Control    Malaria
## 4     4 Control    Malaria
## 5     5 Control    Malaria
## 6     6 Control    Malaria
## 7     7 Control    Malaria
## 8     8 Egg removal Malaria
## 9     9 Egg removal Malaria
## 10   10 Egg removal Malaria
## # i 55 more rows
```

To wrangle this into the appropriate format, here's the appropriate code:

```
bird.malaria.tidy <- bird.malaria %>%
  group_by(treatment) %>%
  count(response)
```

This is similar to what you learned in a previous tutorial, but here we've added a new function!

New tool The `group_by` function from the `dplyr` package enables one to apply a function to each category of a categorical variable. See more help using “`?group_by`”.

In the preceding code chunk, we're tallying the observations in the two “treatment” variable categories, but also keeping track of which category of “response” the individual belongs to.

Let's have a look at the result:

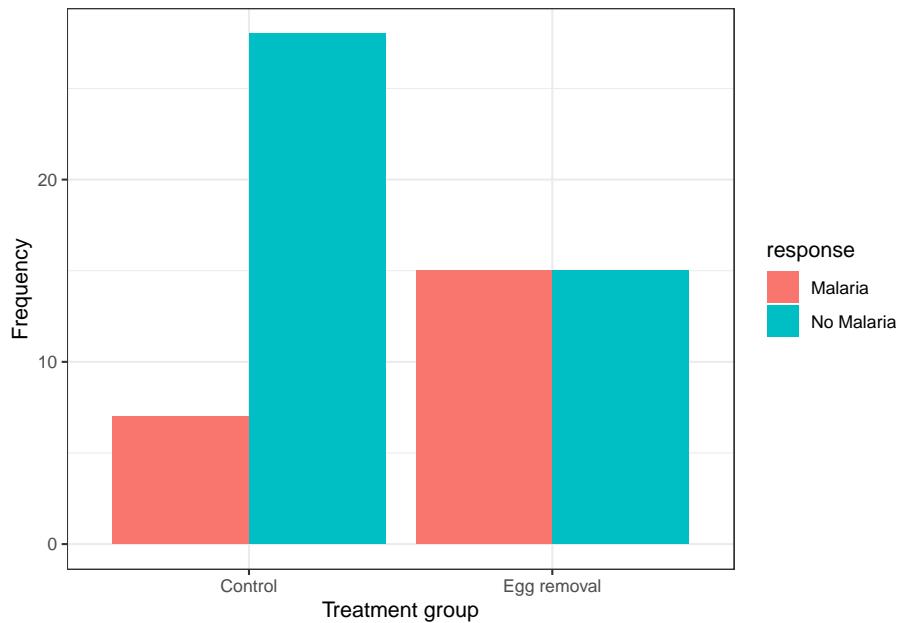
```
bird.malaria.tidy
```

```
## # A tibble: 4 x 3
```

```
## # Groups:  treatment [2]
##   treatment response n
##   <chr>      <chr>    <int>
## 1 Control    Malaria    7
## 2 Control    No Malaria 28
## 3 Egg removal Malaria   15
## 4 Egg removal No Malaria 15
```

We now have what we need for a grouped bar chart, using the `ggplot` function:

```
ggplot(data = bird.malaria.tidy, aes(x = treatment, y = n, fill = response)) +
  geom_bar(stat = "identity", position = position_dodge()) +
  ylab("Frequency") +
  xlab("Treatment group") +
  theme_bw()
```



This code is similar to what we used previously to create a bar graph, but there are two key differences:

- in the first line within the `aes` function, we include a new argument `fill = response`, telling R to use different bar fill colours based on the categories in the “response” variable.
- in the second line, we provide a new argument to the `geom_bar` function: `position = position_dodge()`, which tells R to use separate bars for

each category of the “fill” variable (if we did not include this argument, we’d get a “stacked bar graph” instead)

It is best practice to use the **response variable** as the “fill” variable in a grouped bar graph, as we have done in the malaria example.

If we wished to provide an appropriate figure heading, this would be the code:

```
```{r malaria_barchart, fill = "white",  
relation = control (N = 3)}

gplot(data = bird.malaria.tidy,
 geom_bar(stat = "identity",
 ylab("Frequency") +
 xlab("Treatment group"),
 theme_bw())
```
```

Figure 8.1: Example code chunk for producing a good grouped bar graph

And the result:

```
ggplot(data = bird.malaria.tidy, aes(x = treatment, y = n, fill = response)) +  
  geom_bar(stat = "identity", position = position_dodge()) +
```

```
ylab("Frequency") +  
xlab("Treatment group") +  
theme_bw()
```

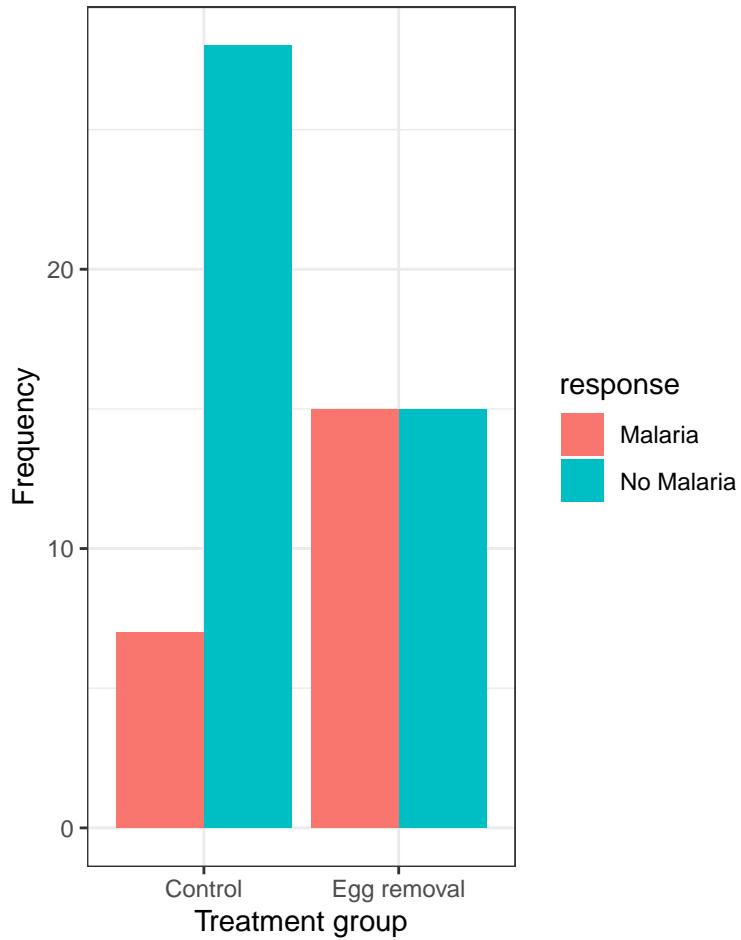


Figure 8.2: Grouped bar graph showing the incidence of malaria in female great tits in relation to control ($N = 35$) and Egg-removal ($N = 30$) treatment groups.

8.2.3 Constructing a mosaic plot

An alternative and often more effective way to visualize the association between two categorical variables is a **mosaic plot**.

For this we use the `geom_mosaic` function, from the `ggbmosaic` package, in conjunction with the `ggplot` function.

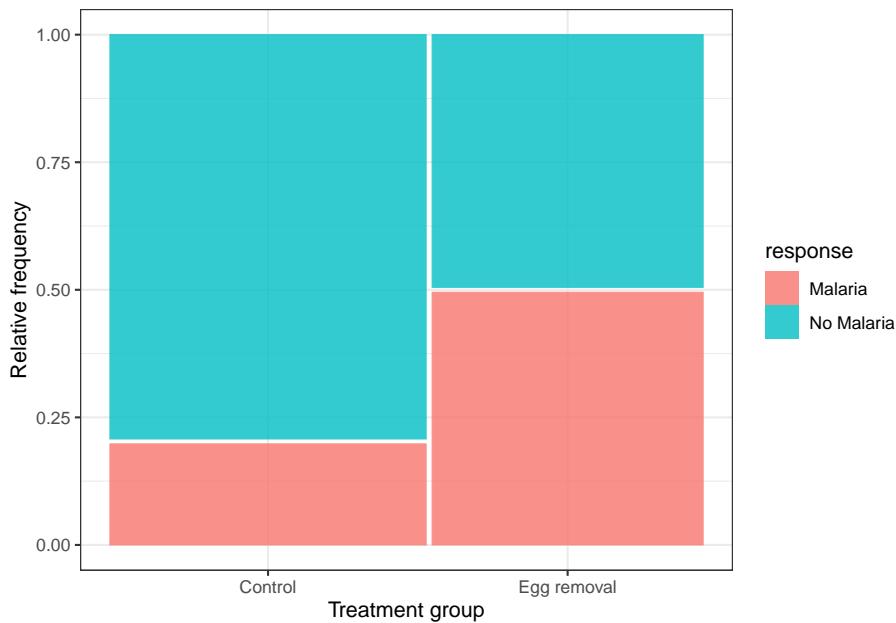
For more information about the `ggbmosaic` package, see this vignette.

For the `geom_mosaic` function, we actually use the original (raw) `bird.malaria` tibble, which has a row for every observation (i.e. it isn't summarized first into a frequency table).

Here's the code, and we'll explain after.

NOTE: Some users might get an error when attempting this; if this happens, consult the relevant section of the Common errors and their solutions page tutorial webpage, which includes an entry on this mosaic plot issue). If you get some “warnings”, don't worry about those, as they won't interfere with getting a proper plot.

```
ggplot(data = bird.malaria) +
  geom_mosaic(aes(x = product(treatment), fill = response)) +
  scale_y_continuous() +
  xlab("Treatment group") +
  ylab("Relative frequency") +
  theme_bw()
```



In the code chunk above we see one key difference from previous uses of the `ggplot` function is that the `aes` function is not provided in the arguments

to `ggplot` in the first line, but is instead provided to the arguments of the `geom_mosaic` function on the second line.

We also see `product(treatment)`, which is difficult to explain, so suffice it to say that it's telling the `geom_mosaic` function to calculate relative frequencies based on the "treatment" variable, and in conjunction with the fill variable "response".

The `scale_y_continuous` function tells `ggplot` to add a continuous scale for the y-axis, and here, this defaults to zero to one for relative frequencies.

We'll learn about interpreting mosaic plots next.

1. Using the `penguins` dataset, try creating a mosaic plot for comparing the relative frequency of penguins belonging to the three different "species" across the three different islands (variable "island").

8.2.4 Interpreting mosaic plots

Let's provide the mosaic plot again, and this time we'll provide an appropriate figure heading in the chunk header, as we learned previously:

```
ggplot(data = bird.malaria) +
  geom_mosaic(aes(x = product(treatment), fill = response)) +
  scale_y_continuous() +
  xlab("Treatment group") +
  ylab("Relative frequency") +
  theme_bw()
```

Here's the code:

When interpreting a mosaic plot, the key is to look how the relative frequency of the categories of the response variable - denoted by the "fill" colours - varies across the explanatory variable, which is arranged on the x-axis.

For example, in the malaria example above:

"The mosaic plot shows that the incidence (or relative frequency) of malaria is comparatively greater among birds in the egg removal treatment group compared to the control group. Only about 20% of birds in the control group contracted malaria, whereas 50% of the birds in the the egg-removal group contracted malaria."

8.3 Visualizing association between two numeric variables

We use a `scatterplot` to show association between two numerical variables.

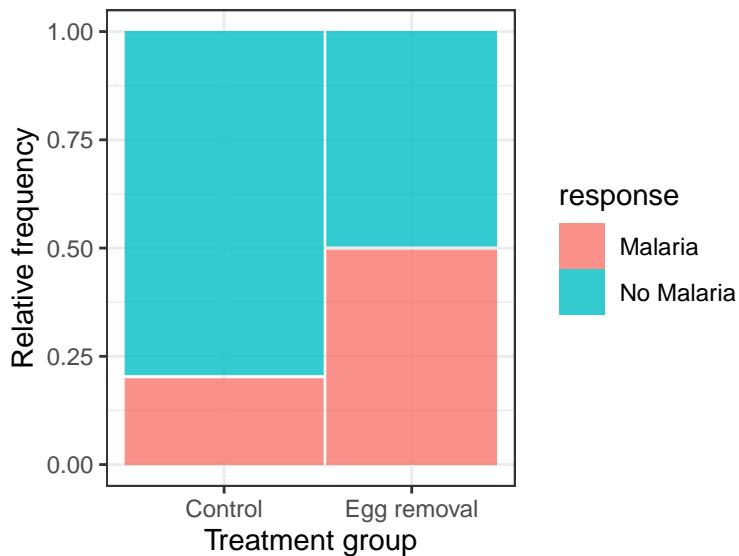


Figure 8.3: Relative frequency of the incidence of malaria in female great tits in relation to control ($N = 35$) and Egg-removal ($N = 30$) treatment groups.

We'll use the `ggplot` function that we've seen before, along with `geom_point` to construct a scatterplot.

We'll provide an example using the `penguins` dataset, examining how bill depth and length are associated among the penguins belonging to the Adelie species.

As shown in the tutorial on preparing and formatting assignments, we can use the `filter` function from the `dplyr` package to easily subset datasets according to some criterion, such as belonging to a specific category.

```
penguins %>%
  filter(species == "Adelie") %>%
  ggplot(aes(x = bill_length_mm, y = bill_depth_mm)) +
  geom_point(shape = 1) +
  xlab("Bill length (mm)") +
  ylab("Bill depth (mm)") +
  theme_bw()
```

In the code chunk above, we have:

- the input tibble `penguins` followed by the pipe (“%>%”)
- the `filter` function with the criterion used for subsetting, specifically any cases in which the “species” categorical variable equals “Adelie”

```
```{r fig.cap = "Relative  
and Egg-removal (N = 30)"

ggplot(data = bird.malaria,
 geom_mosaic(aes(x = pro,
 scale_y_continuous() +
 xlab("Treatment group") +
 ylab("Relative frequency") +
 theme_bw()))
```
```

Figure 8.4: Example code chunk for producing a good mosaic plot

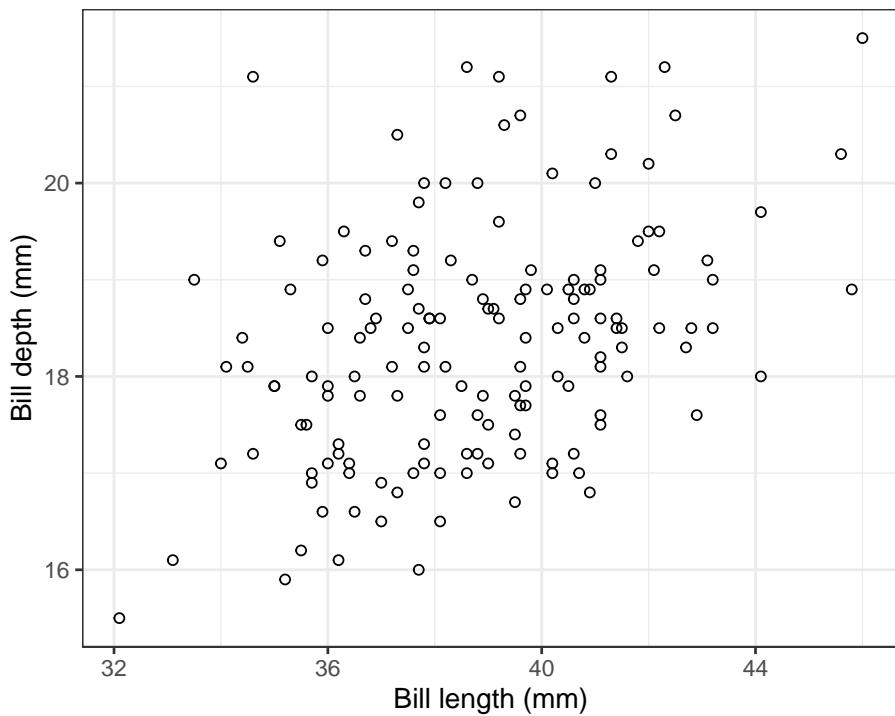


Figure 8.5: Scatterplot of the association between bill length and depth among 151 Adelie penguins

- then we provide the `ggplot` function and its `aes` argument, specifying the x- and y- variables to be used
- then we use `geom_point` to tell R to create a scatterplot using points, and specifically “shape = 1” denotes hollow circles
- then we have x and y labels, followed by the `theme_bw` function telling R to use black and white theme

Notice that the figure caption indicates the number of observations (sample size) used in the plot. In a previous tutorial it was emphasized that one needs to be careful in tallying the actual number of observations being used in a graph or when calculating descriptive statistics. For example, there is one missing value (“NA”) in the bill measurements for the Adelie penguins, hence the sample size of 151 instead of 152.

Recall that you can use the `skim` or `skim_without_charts` functions to get an overview of a dataset or of a single variable in a dataset, and to figure out how many missing values there are for each variable. You can also use the `summarise` function, as described previously.

8.3.1 Interpreting and describing a scatterplot

Things to report when describing a scatterplot:

- is there an association? A “shotgun blast” pattern indicates no. If there is an association, is it *positive* or *negative*?
- if there is an association, is it weak, moderate, or strong?
- is the association *linear*? If not, is there a different pattern like concave down?
- are there any *outlier* observations that lie far from the general trend?

In the scatterplot above, bill length and depth are positively associated, and the association is moderately strong. There are no observations that are strongly inconsistent with the general trend, though one individual with bill length of around 35mm and depth of around 21mm may be somewhat unusual.

1. Using the `penguins` dataset, create a scatterplot of flipper length in relation to body mass, and provide an appropriate figure caption.

8.4 Visualizing association between a numeric and a categorical variable

To visualize association between a numerical response variable and a categorical explanatory variable, we have a variety of options, and the choice depends in part on the sample sizes within the categories being visualized.

- When sample sizes are relatively small in each category, such as 20 or fewer, use a **stripchart**
- When sample sizes are larger (>20), use a **violin plot**, or less ideal, a **boxplot**.

We'll use locust serotonin data set from the text book. Consult figure 2.1-2 in the text for a description.

Always remember to get an overview of the dataset before attempting to create graphs, and not only for establishing sample sizes. If you had gotten an overview of the **locust** dataset, you would see we have a numeric response variable “serotoninLevel”, but the categorical (explanatory) variable “Treatment-Time” is actually coded as a numerical variable, with values of 0, 1, or 2 hours. Although this variable is coded as numeric, we can treat it as though it is an ordinal categorical variable.

We should re-code the “treatmentTime” variable in the **locust** dataset as a “factor” variable with three “levels”: 0, 1, 2. This is not necessary for our graphs to work, but it is good practice to do this when you encounter this situation where a variable that should be treated as an ordinal categorical variable is coded as numerical.

We do this using the **as.factor** function, as follows:

```
locust$treatmentTime <- as.factor(locust$treatmentTime)
```

Before creating a stripchart, it's a good idea to prepare a table of descriptive stats for your numerical response variable grouped by the categorical variable.

1. Using what you learned in a previous tutorial, create a table of descriptive statistics of serotonin levels grouped by the treatment group variable.
-

8.4.1 Create a stripchart

Now we're ready to create a stripchart of the locust experiment data. Note that we're not yet ready to add “error bars” to our strip chart; that will come in a later tutorial.

We'll provide the code, then explain after:

```
locust %>%
  ggplot(aes(x = treatmentTime, y = serotoninLevel)) +
  geom_jitter(colour = "black", size = 3, shape = 1, width = 0.1) +
  xlab("Treatment time (hours)") +
  ylab("Serotonin (pmoles)") +
  ylim(0, 25) +
  theme_bw()
```

- the `ggplot` line of code is familiar
- the new function here is the `geom_jitter` function that simply plots the points in each group such that they are “jittered” or offset from one-another (to make them more visible). Its arguments include ‘colour = “black”’ telling R to use black points, “size = 3” to make the points a little larger than the default (1), “shape = 1” denoting hollow circles, and “width = 0.1” telling R to jitter the points a relatively small amount in the horizontal direction. Feel free to play with this arguments to get a feel for how they work.
- the x- and y-axis labels come next
- then we specify the minimum and maximum limits to the y-axis using the `ylim` function

Notice how all the data are visible! And it’s evident that in the control and 1-hour treatment groups the majority of locusts exhibited comparatively low levels of serotonin (note the clusters of points).

8.4.2 Create a violin plot

Given that violin plots are best suited to when one has larger sample sizes per group, we’ll go back to the `penguins` dataset for this, and evaluate how body mass of male penguins varies among species.

Let’s first find out more about the data for the male penguins, so that we can include sample sizes in our figure captions. Specifically, we’ll tally the number of complete body mass observations for each species, and also the number of missing values (NAs).

We’ll combine the `filter` function with the `group_by` function that we learned about in a previous tutorial:

```
penguins %>%
  filter(sex == "male") %>%
  group_by(species) %>%
  summarise(
  Count = n() - naniar::n_miss(body_mass_g),
  Count_NA = naniar::n_miss(body_mass_g))
```

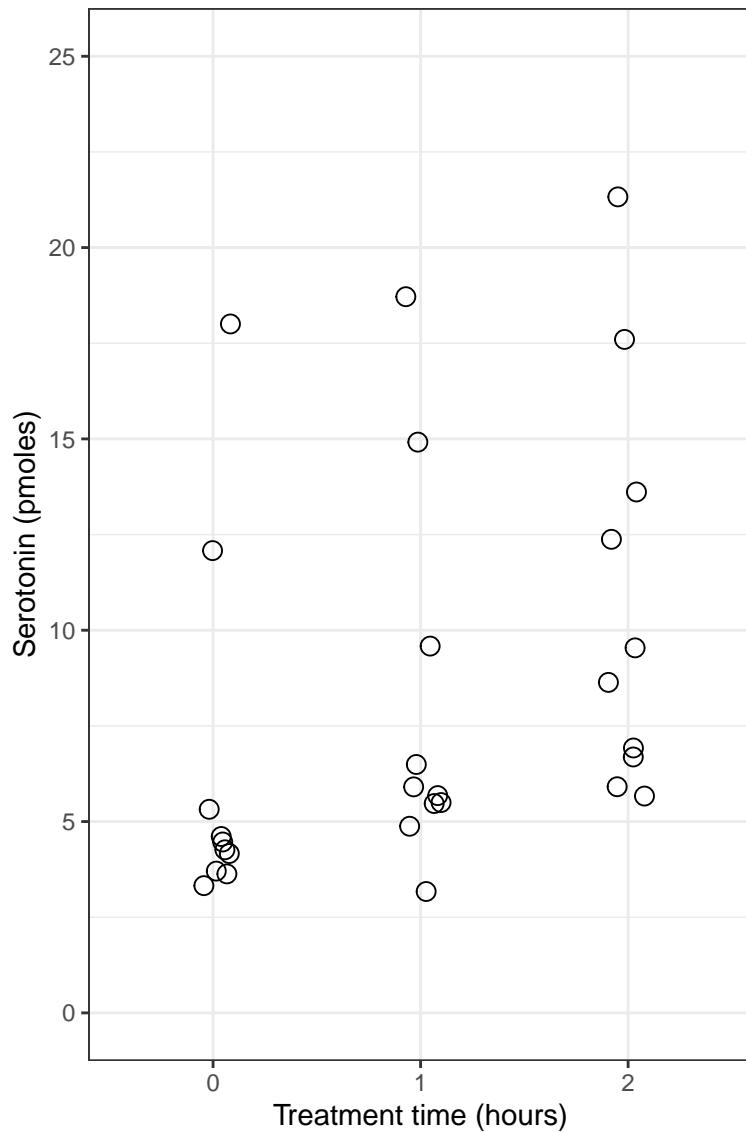


Figure 8.6: Serotonin levels in the central nervous system of desert locusts that were experimentally crowded for 0 (control), 1, and 2 hours. N = 10 per treatment group.

```
## # A tibble: 3 x 3
##   species   Count Count_NA
##   <fct>     <int>    <int>
## 1 Adelie      73       0
## 2 Chinstrap   34       0
## 3 Gentoo      61       0
```

This is the same code we used previously for calculating descriptive statistics using a grouping variable (though we've eliminated some of the descriptive statistics here), but we inserted the `filter` function in the second line to make sure we're only using the male penguin records.

We now have the accurate sample sizes for each species (under the "Count" variable) we need to report in any figure caption.

We use the familiar `ggplot` approach for creating violin plots.

When using the `ggplot` function, we can assign the output to an object. We can then subsequently add features to the plot by adding to the object. We'll demonstrate this here.

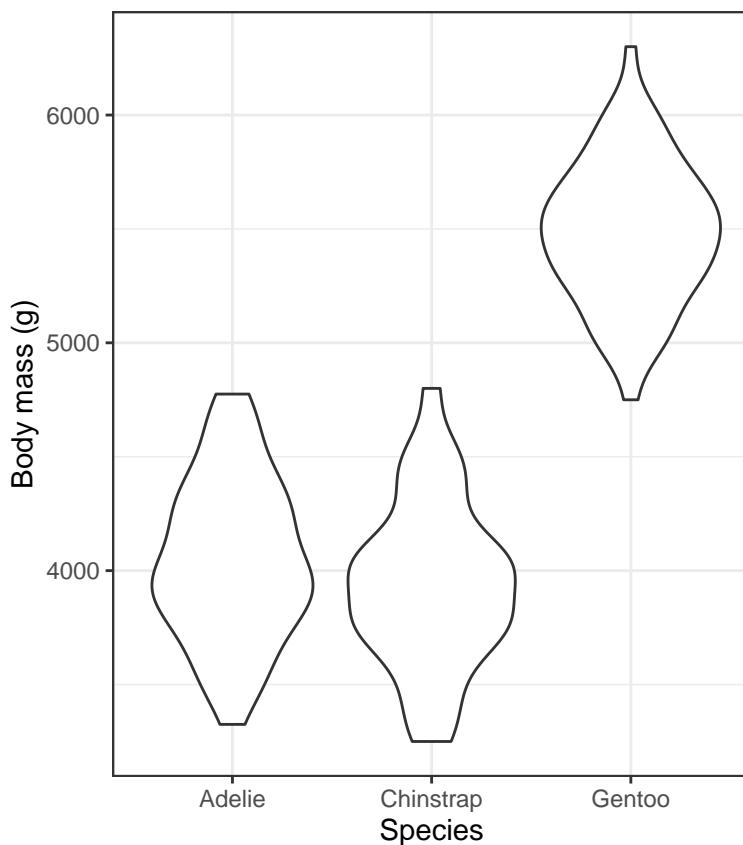
Let's assign the basic violin plot to an object called "bodymass.violin", and we'll explain the rest of the code after:

```
bodymass.violin <- penguins %>%
  filter(sex == "male") %>%
  ggplot(aes(x = species, y = body_mass_g)) +
  geom_violin() +
  xlab("Species") +
  ylab("Body mass (g)") +
  theme_bw()
```

- We assign the output to the object called "bodymass.violin", and tell R which data object we're using (`penguins`)
- We then `filter` the dataset to include only male penguins (`sex == "male"`), and note the two equal signs and the quotations around "male"
- Then the familiar `ggplot` with its `aes` argument
- Now the new `geom_violin` function, and it has optional arguments that we haven't used (see help file for the function)
- Then the familiar labels and theme functions

Let's now have a look at the graph, and to do so, we simply type the name of the graph object we created:

```
bodymass.violin
```



One problem with the above graph is that we don't see the individual data points.

We can add those using the `geom_jitter` function we learned about when creating stripcharts.

Here's how we add features to an existing `ggplot` graph object, and we can again create a new object, or simply replace the old one.

Here, we'll create a new object called "bodymass.violin.points":

```
bodymass.violin.points <- bodymass.violin + geom_jitter(size = 2, shape = 1, width = 0.1)
```

And now show the plot:

```
bodymass.violin.points
```

TIP: If you wish to run all the code at once in a single chunk to create a figure, rather than adding new code to an existing object, here's what you'd include in

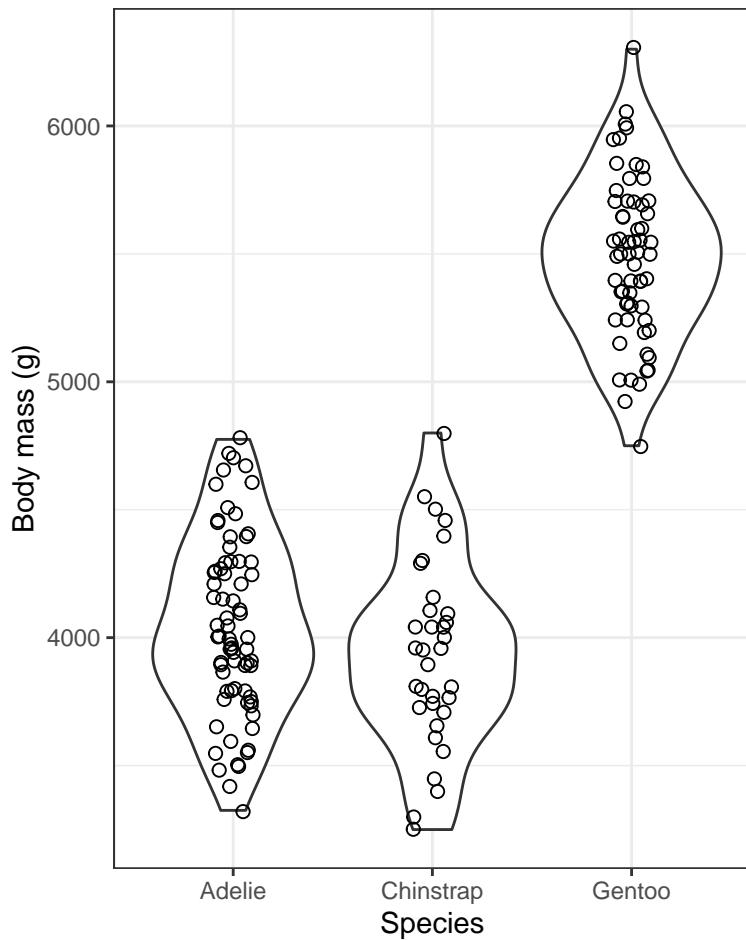


Figure 8.7: Violin plot showing the body mass (g) of male Adelie ($N = 73$), Chinstrap ($N = 34$), and Gentoo ($N = 61$) penguins.

your chunk (but here we don't show the chunk header that would include the caption):

```
penguins %>%
  filter(sex == "male") %>%
  ggplot(aes(x = species, y = body_mass_g)) +
  geom_violin() +
  geom_jitter(size = 2, shape = 1, width = 0.1) +
  xlab("Species") +
  ylab("Body mass (g)") +
  theme_bw()
```

To help understand what the violin plot is showing, we'll provide a new **bonus** graph that adds something called "density plots" to the margin of the violin plot.

Don't worry about replicating this type of graph, but if you can, fantastic!

First we create the main `ggplot` violin plot object:

```
bodymass.violin2 <- penguins %>%
  filter(sex == "male") %>%
  ggplot() +
  geom_violin(aes(x = species, y = body_mass_g, colour = species)) +
  geom_jitter(aes(x = species, y = body_mass_g, colour = species), size = 2, shape = 1, width = 0.1) +
  xlab("Species") +
  ylab("Body mass (g)") +
  ylim(3000, 6300) +
  theme_bw() +
  theme(legend.position = "bottom") +
  guides(colour = guide_legend(title="Species"))
```

Now we add density plots in the margins using the `ggMarginal` function from the `ggExtra` package:

```
ggMarginal(bodymass.violin2, type = "density", alpha = 0.3, groupFill = TRUE)
```

The violin plot is designed to give an idea of the frequency distribution of response variable values within each group. Specifically, the width of the violin reflects the frequency of observations in that range of values. This is evident in the above figure within the "density plots" that are provided in the right-hand margin. Consider these density plots as a smoothed out version of a histogram.

We can see, for example, that for all three species of penguin there is a bulge in the middle indicating that there is a central mode to the body mass values in each group, with fewer values towards lower and higher extremes. The frequency distribution for the Gentoo species approximates a "bell shape" distribution, for example (the blue data).

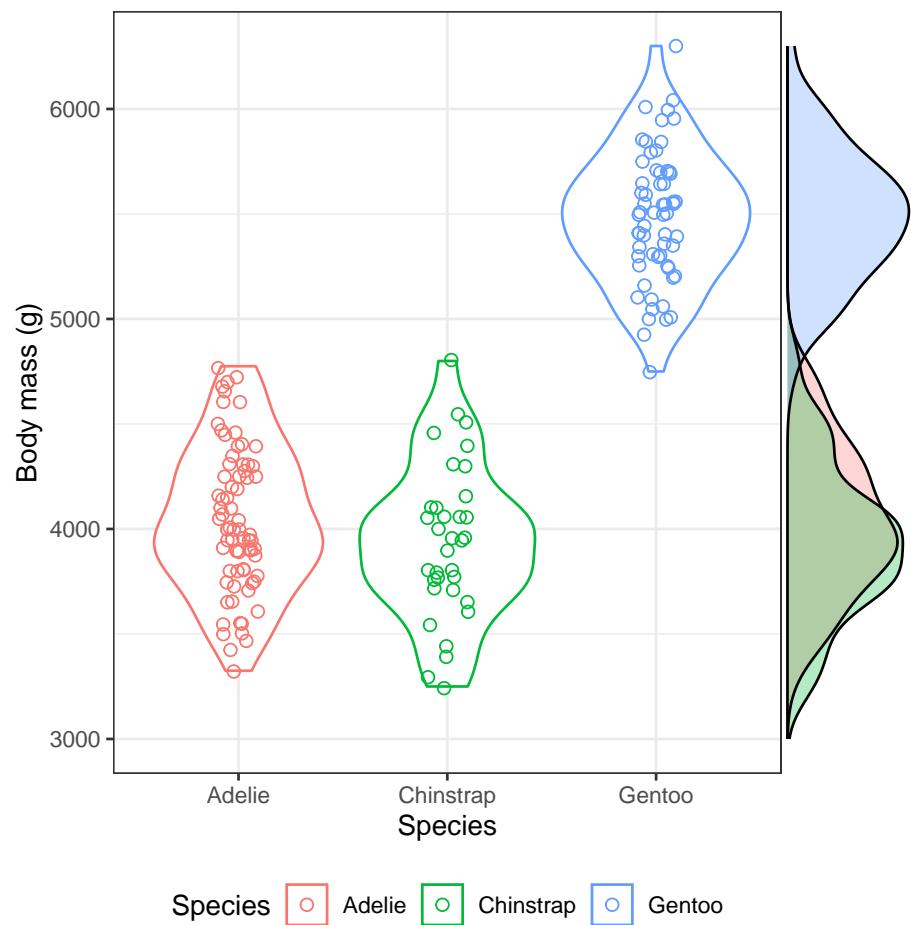


Figure 8.8: Violin plot showing the body mass (g) of male Adelie ($N = 73$), Chinstrap ($N = 34$), and Gentoo ($N = 61$) penguins. Density plots are provided in the margin.

8.4.3 Creating a boxplot

Here we'll learn how to create basic boxplots, and also superimpose boxplots onto violin plots.

In future tutorials we'll learn how to add features to these types of graphs in order to complement statistical comparisons of a numerical response variable among categories (groups) of a categorical explanatory variable.

Here is the code for creating boxplots, again using the penguins body mass data, and this time the `geom_boxplot` function:

```
penguins %>%
  filter(sex == "male") %>%
  ggplot(aes(x = species, y = body_mass_g)) +
  geom_boxplot() +
  xlab("Species") +
  ylab("Body mass (g)") +
  theme_bw()
```

For more information about the features of the boxplot, look at the help file for the `geom_boxplot` function:

```
?geom_boxplot
```

The first time you include a boxplot in a report / lab, be sure to include in the figure caption the details of what is being shown. You only need to do this the first time. Subsequent boxplot figure captions can refer to the first one for details.

When sample sizes are large in each group, like they are for the penguins data we've been visualizing, the most ideal way to visualize the data is to combine violin and boxplots. We'll do this next!

8.4.4 Combining violin and boxplots

Superimposing boxplots onto violin plots (and of course, showing individual points too!) provides for a very informative graph.

We already have a basic violin plot object created, called “bodymass.violin”, so let's start with that, then add the boxplot information, then superimpose the points. We need to do it in that order, so that the points become the “top” layer of information, and aren't hidden behind the boxplot or violins.

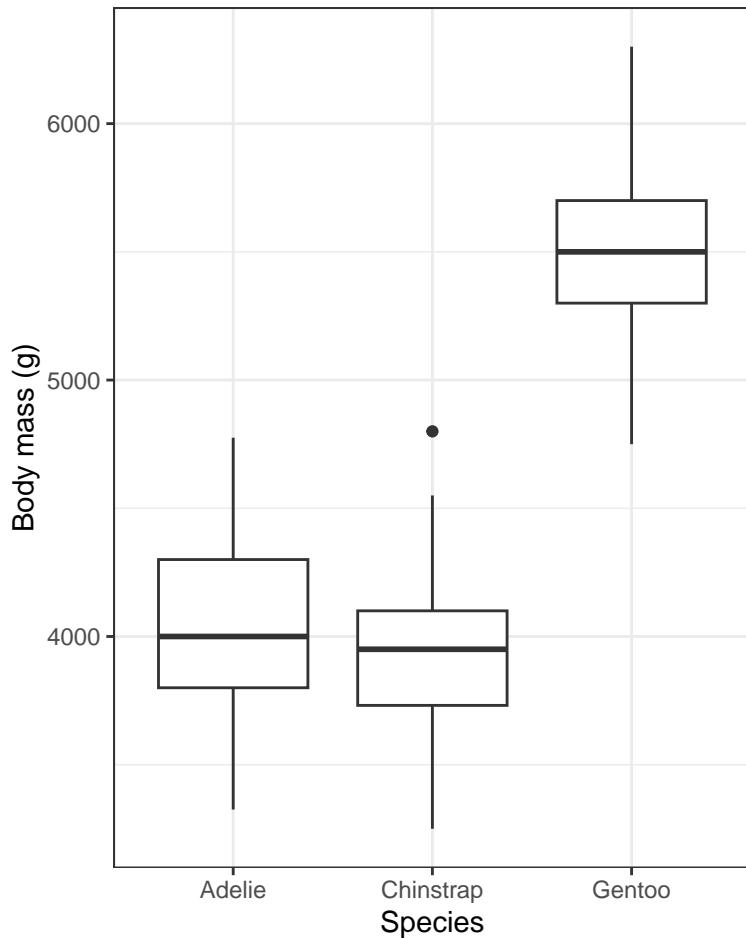


Figure 8.9: Boxplot showing the body mass (g) of male Adelie ($N = 73$), Chinstrap ($N = 34$), and Gentoo ($N = 61$) penguins. Density plots are provided in the margin. Boxes delimit the first and third quartiles, and the middle line depicts the median. Whiskers extend to values beyond the 1st (lower) and 3rd (upper) quartiles, up to a maximum of $1.5 \times \text{IQR}$, and individual black points are outlying values.

```
bodymass.violin +
  geom_boxplot(width = 0.1) +
  geom_jitter(colour = "grey", size = 1, shape = 1, width = 0.15)
```

- We start with the base violin plot object “bodymass.violin”
- We then add the boxplot using `geom_boxplot`, ensuring that the boxes are narrow in width (`width = 0.1`) so they don’t overwhelm the violins
- We then add the individual data points using `geom_jitter`, and this time using the colour “grey” so that they don’t obscure the black boxes underneath, and making them a bit smaller this time (`size = 1`), and keeping them as hollow circles (`shape = 1`), and this time spreading them out horizontally a bit more (`width = 0.15`)

TIP: if you’d rather do all the code in one chunk, without creating objects, here’s what you’d include:

```
penguins %>%
  filter(sex == "male") %>%
  ggplot(aes(x = species, y = body_mass_g)) +
  geom_violin() +
  geom_boxplot(width = 0.1) +
  geom_jitter(colour = "grey", size = 1, shape = 1, width = 0.15) +
  xlab("Species") +
  ylab("Body mass (g)") +
  theme_bw()
```

It often takes some playing around with argument values before one gets the ideal graph. For example, in the above graph, try changing some of the values used in the `geom_jitter` function.

1. Using the `penguins` dataset, and only the records pertaining to female penguins, create a combined violin / boxplot graph showing bill length in relation to species. Include an appropriate figure caption.

8.4.5 Interpreting stripcharts, violin plots and boxplots

In general, stripcharts, violin plots, and boxplots are used to visualize how a numeric variable varies or differs among categories (groups) of a categorical variable. For instance, it’s pretty obvious from the violin plots above that Gentoo penguins have, on average, considerably greater body mass than the other two species. We will wait until a future tutorial to learn more about interpreting violin / boxplots, because there we learn how to add more information to the graphs, such as group means and measures of uncertainty. For now, you should be comfortable interpreting any obvious patterns in the plots.

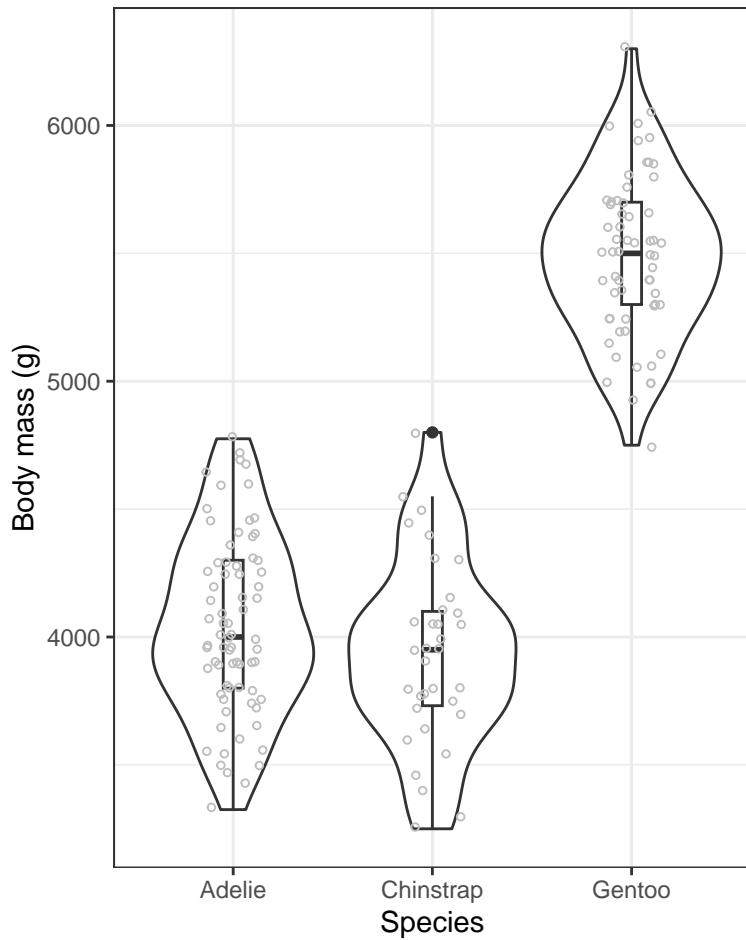


Figure 8.10: Violin plot showing the body mass (g) of male Adelie ($N = 73$), Chinstrap ($N = 34$), and Gentoo ($N = 61$) penguins. Boxplots are superimposed, with boxes delimiting the first and third quartiles, and middle line depicting the median. Whiskers extend to values beyond the 1st (lower) and 3rd (upper) quartiles, up to a maximum of $1.5 \times \text{IQR}$, and individual black points are outlying values.

Inferential Statistics

Chapter 9

Sampling, Estimation, & Uncertainty

Tutorial learning objectives

In this tutorial you will:

- Learn how to take a random sample of observations from a dataset
- Learn through simulation what “sampling error” is
- Learn about sampling distributions through simulation
- Learn how to calculate the standard error of the mean
- Learn how to calculate the “rule of thumb 95% confidence interval”

9.1 Load packages and import data

Let's load some familiar packages first:

```
library(tidyverse)
library(nanar)
library(knitr)
library(skimr)
```

We will also need a new package called `infer`, so install that package using the procedure you previously learned, then load it:

```
library(infer)
```

Import Data

For this tutorial we'll use the human gene length dataset that is used in Chapter 4 of the Whitlock & Schluter text.

The dataset is described in example 4.1 in the text.

Let's import it:

```
genelengths <- read_csv("https://raw.githubusercontent.com/ubco-biology/BIOL202/main/datasets/genelengths.csv")
```

```
## Rows: 22385 Columns: 4
## -- Column specification -----
## Delimiter: ","
## chr (3): gene, name, description
## dbl (1): size
##
## 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.
```

Get an overview of the data

We'll use `skim_without_charts` to get an overview:

```
genelengths %>%
  skim_without_charts()
```

(#tab:est_overview) Data summary

Name

Piped data

Number of rows

22385

Number of columns

4

Column type frequency:

character

3

numeric

1

Group variables

None

Variable type: character

skim_variable

n_missing

complete_rate

min

max

empty

n_unique

whitespace

gene

0

1.00

17

18

0

22385

0

name

0

1.00

2

15

0

19906

0

description

432

0.98

1

```
51
0
4183
0

Variable type: numeric
skim_variable
n_missing
complete_rate
mean
sd
p0
p25
p50
p75
p100
size
0
1
3511.46
2833.29
69
1684
2744
4511
109224
```

The “size” variable is the key one: it includes the gene lengths (number of nucleotides) for each of the 22385 genes in the dataset.

9.2 Functions for sampling

To illustrate the concepts of “sampling error” and the “sampling distribution of the mean”, we’re going to make use of datasets that include measurements for all

individuals in a “population” of interest, and we’re going to take random samples of observations from those datasets. This mimics the process of sampling at random from a population.

There are two functions that are especially helpful for taking random samples from objects.

- The `sample` function in the base R package is easy to use for taking a random sample from a `vector` of values. Look at the help function for it:

```
?sample
```

The second useful function is the `slice_sample` function from the `dplyr` package (loaded with `tidyverse`). This function takes random samples of rows from a `dataframe` or `tibble`.

```
?slice_sample
```

Note that both functions have a `replace` argument that dictates whether the sampling we conduct occurs with or without replacement. For instance, the following code will not work (try copying and pasting into your console):

```
sample(1:6, size = 10, replace = F)
```

Here we’re telling R to sample 10 values from a vector that includes only 6 values (one through six), and with the `replace = F` argument we told R not to put the sampled numbers back in the pile for future sampling.

If instead we specified `replace = T`, as shown below, then we be conducting *sampling with replacement*, and sampled values are placed back in the pool each time, thus enabling the code to work (try it):

```
sample(1:6, size = 10, replace = T)
```

Whether we wish to *sample without replacement* or not depends on the question / context (see page 138 in Whitlock & Schluter). For most of the questions and contexts we deal with in this course, we do wish to sample **with** replacement. Why? Because we are typically assuming (or mimicking a situation) that we’re randomly sampling from a very large population, in which case the probability distribution of possible values in the individuals that remain doesn’t change when we sample. This is not the case if we’re sampling from a small population.

9.2.1 Setting the “seed” for random sampling

Functions such as `sample` and `slice_sample` have a challenge to meet: they must be able to take samples that are as random as possible. On a computer, this is easier said than done; special algorithms are required to meet the challenge. R has some pretty good ones available, so we don’t have to worry about it (though to be honest, they can only generate “pseudorandom” numbers, but that’s good enough for our purposes!).

However, we DO need to worry about *computational reproducibility*.

Imagine we have authored a script for a research project, and the script makes use of functions such as `sample`. Now imagine that someone else wanted to re-run our analyses on their own computer. Any code chunks that included functions such as `sample` would produce different results for them, because the function takes a different random sample each time it’s run!

Thankfully there’s a way to ensure that scripts implementing random sampling can be computationally reproducible: we can use the `set.seed` function.

This assumes you are using R version 4 or later (e.g. 4.1). The `set.seed` function used a different default algorithm in versions of R prior to version 4.0.

Let’s provide the code chunk then explain after:

```
set.seed(12)
```

The `set.seed` function requires an integer value as an argument. You can pick any number you want as the seed number. Here we’ll use 12, and if you do too, you’ll get the same results as in the tutorial.

Let’s test this using the `runif` function, which generates random numbers between user-designated minimum and maximum values (the default is numbers between zero and one).

Let’s include the `set.seed` function right before using the `runif` function, and here we’ll again use a seed number of 12.

```
set.seed(12)
runif(3)
```

```
## [1] 0.06936092 0.81777520 0.94262173
```

Here we told the `runif` function to generate three random numbers, and it used the default minimum and maximum values of zero and one.

Now let’s try a different seed:

```
set.seed(25)
runif(3)

## [1] 0.4161184 0.6947637 0.1488006
```

So long as you used the same seed number, you should have gotten the same three random numbers as shown above.

When you are authoring your own script or markdown document (e.g. for a research project), it is advisable to only set the seed once, at the beginning of your script or markdown document. When you run or knit your completed script/markdown, the seed will be set at the beginning, the code chunks will run in sequence, and each chunk that uses a random number generator will do so in a predictable way (based on the seed number). The script will be computationally reproducible. In contrast, when we're trying out code (e.g. when working on tutorial material), we aren't running a set sequence or number of code chunks, so we can't rely on everyone getting the same output for a particular code chunk. For this reason you'll see in the tutorial material that many of the code chunks that require use of a random number generator will include a `set.seed` statement, to ensure that everyone gets the same output from that specific code chunk (this isn't always required, but often it is).

1. Generate 10 random numbers using the `runif` function, and first set the seed number to 200.

9.3 Sampling error

Sampling error is the chance difference, caused by sampling, between an estimate and the population parameter being estimated

Here we'll get a feel for **sampling error** using the human gene data.

Let's use the `slice_sample` function to randomly sample $n = 20$ rows from the `genelengths` tibble, and store them in a tibble object called "randsamp1":

```
set.seed(12)
randsamp1 <- genelengths %>%
  slice_sample(n = 20, replace = FALSE) %>%
  select(size)
```

In the preceding chunk, we:

- set the seed (here using integer 12), so that everyone gets the same output for this code chunk

- tell R that we want the output from our code to be stored in an object named “randsamp1”
- use the `slice_sample` function to randomly sample 20 rows from the `genelengths` tibble, and to do so without replacement
- use the `select` function to tell R to return only the “size” variable from the newly generated (sampled) tibble

Now let’s use the `mean` and `sd` base functions to calculate the mean and standard deviation using our sample.

We’ll assign the calculations to an object “`randsamp1.mean.sd`”, then we’ll present the output using the `kable` function.

```
randsamp1.mean.sd <- randsamp1 %>%
  summarise(
    Mean_genelength = mean(size, na.rm = TRUE),
    SD_genelength = sd(size, na.rm = TRUE)
  )
```

Now present the output using the `kable` function so we can control the maximum number of digits shown.

Using the `kable` approach to presenting tibble outputs ensures that, when knitted, your output shows a sufficient number of decimal places... something that doesn’t always happen without using the `kable` function.

```
kable(randsamp1.mean.sd, digits = 4)
```

Mean_genelength

SD_genelength

3408.25

1443.698

Do your numbers match those above?

Let’s now draw another random sample of the same size (20), making sure to give the resulting tibble object a different name (“`randsamp2`”). Here, we won’t set the seed again, because all that is required is for everyone to get a different sample from the first one above; we don’t need to have everyone get the *same* sample, but it’s ok if we do.

```
randsamp2 <- genelengths %>%
  slice_sample(n = 20, replace = FALSE) %>%
  select(size)
```

And calculate the mean and sd:

```
randsamp2.mean.sd <- randsamp2 %>%
  summarise(
    Mean_genelength = mean(size, na.rm = TRUE),
    SD_genelength = sd(size, na.rm = TRUE)
  )
```

And show using the `kable` function:

```
kable(randsamp2.mean.sd, digits = 4)
```

Mean_genelength

SD_genelength

4381.35

2081.471

Are they the same as we saw using the first random sample? NO! This reflects **sampling error**.

1. Repeat the process above to get a third sample of size 20 genes, and calculate the mean and standard deviation of gene length for that sample.

9.4 Sampling distribution of the mean

Now let's learn code to conduct our own resampling exercise.

Let's repeat the sampling exercise we did in the preceding section concerning sampling error, but this time repeat it many times, say 10000 times.

We'll also calculate the mean gene length for each sample that we take, and store this value.

For this, we'll use the newly installed `infer` package, and its handy function `rep_slice_sample`, which complements the `slice_sample` function we've used before.

Let's look at the code then explain after:

```
gene.means.n20 <- genelengths %>%
  rep_slice_sample(n = 20, replace = FALSE, reps = 10000) %>%
  summarise(
    mean_length = mean(size, na.rm = T),
    sampsize = as.factor(20)
  )
```

In the above chunk, we:

- Tell R to put the output from our functions into a new tibble object called “gene.means.n20”, with the “n20” on the end denoting the fact that we used a sample size of 20 for these samples
- use the `rep_slice_sample` function to sample $n = 20$ rows at random from the `genelengths` tibble, and repeat this “`reps = 10000`” times
- next we use the familiar `summarise` function to then create a new variable “`mean_length`”, which in the next line we define as the mean of the values in the “`size`” variable (and recall, at this point we have a sample of 20 of these)
- we also create a new categorical (factor) variable “`sampsiz`e” that indicates what sample size we used for this exercise, and we store it as a factor rather than a number (we’re doing this because it will come in handy later)

Let’s look at the first handful of rows in the newly created tibble:

```
gene.means.n20
```

```
## # A tibble: 10,000 x 3
##   replicate mean_length sampsiz
##       <int>      <dbl> <fct>
## 1         1      3094. 20
## 2         2      4705. 20
## 3         3      2576. 20
## 4         4      2957. 20
## 5         5      3060. 20
## 6         6      3768. 20
## 7         7      3412. 20
## 8         8      4258. 20
## 9         9      3242. 20
## 10        10     3134. 20
## # i 9,990 more rows
```

We see that it is a tibble with 10000 rows and three columns: a variable called “`replicate`”, which holds the numbers one through the number of replicates run (here, 10000), a variable “`mean_length`” that we created, which holds the sample mean gene length for each of the replicate samples we took, and a variable “`sampsiz`e” that tells us the sample size used for each sample (here, 20).

Pause: If we were to plot a histogram of these 10000 sample means, what shape do you think it would have? And what value do you think it would be centred on?

9.4.1 Visualize the sampling distribution

Let's plot a histogram of the 10000 means that we calculated.

First, to figure out what axis limits we should set for our histogram, let's get an idea of what the mean gene length values look like:

```
gene.means.n20 %>%
  skim_without_charts(mean_length)
```

(#tab:view_means) Data summary

Name

Piped data

Number of rows

10000

Number of columns

3

Column type frequency:

numeric

1

Group variables

None

Variable type: numeric

skim_variable

n_missing

complete_rate

mean

sd

p0

p25

p50

p75

```
p100
mean_length
0
1
3497.63
621.56
1650.2
3066.16
3439.65
3871.6
9609.4
```

OK, so we see that the minimum and maximum values are 1650.2 (under the “p0” heading) and 9609.4 (under the “p100” heading). So this can inform our x-axis limits when constructing the histogram (note that you typically need to try out different maximum values for the y-axis limits).

Let’s first remind ourselves what the true mean gene length is in the entire population of genes. We’ll assign the output to an object “true.mean.length” and a variable called “Pop_mean”, then show using the `kable` approach.

```
true.mean.length <- genelengths %>%
  summarise(
    Pop_mean = mean(size, na.rm = TRUE)
  )
```

Show using `kable`:

```
kable(true.mean.length, digits = 4)
```

```
Pop_mean
3511.457
```

Now let’s plot the histogram of 10000 sample means:

```
gene.means.n20 %>%
  ggplot(aes(x = mean_length)) +
  geom_histogram(binwidth = 250, color = "black", fill = "lightgrey") +
  ylim(0, 1700) +
  xlim(1500, 10000) +
  xlab("Gene length (nucleotides)") +
```

```
ylab("Number of genes") +
theme_bw()
```

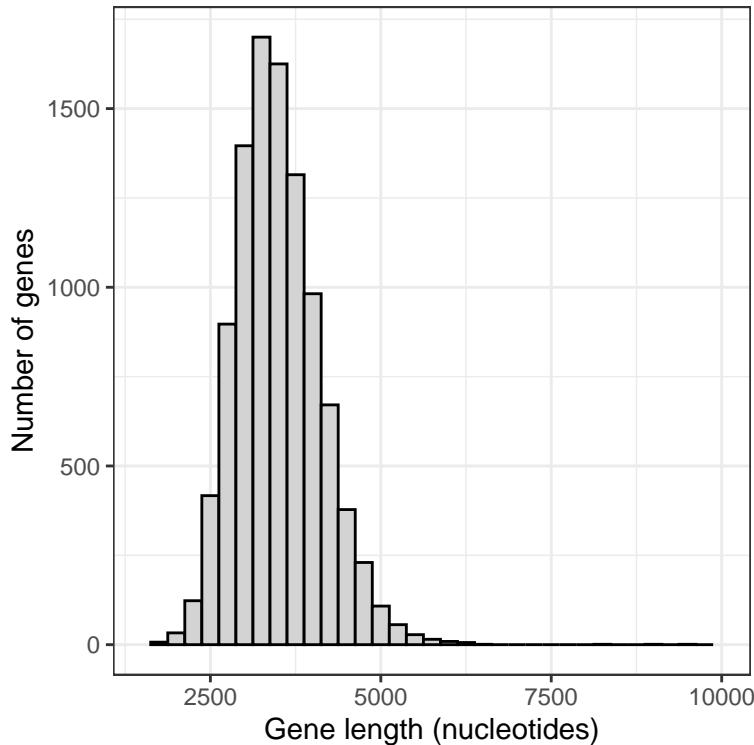


Figure 9.1: Approximate sampling distribution of mean gene lengths from the human genome (10000 replicate samples of size 20).

This histogram is an approximation of the **sampling distribution of the mean** for a sample size of 20 (to construct a real sampling distribution we'd need to take an infinite number of replicate samples).

Before interpreting the histogram, let's first calculate the mean of the 10000 sample means, then display using the **kable** approach:

```
sample.mean.n20 <- gene.means.n20 %>%
  summarise(
    Sample_mean = mean(mean_length, na.rm = TRUE)
  )
```

Show using **kable**:

```
kable(sample.mean.n20, digits = 4)
```

Sample_mean

3497.633

Note that the mean of the sample means is around 3498, which is pretty darn close to the true population mean of 3511.

This reflects the fact that, provided a good random sample, the sample mean is an **unbiased estimate** of the true population mean.

Now, let's interpret the histogram:

- The vast majority of means are centred around the true population mean (around 3500)
- The frequency distribution unimodal, but is right (positively) skewed; it has a handful of sample means that are very large

The latter characteristic reflects the fact that the population of genes includes more than 100 genes that are extremely large. When we take a random sample of 20 genes, it is possible that - just by chance - our sample could include one or more of the extremely large genes in the population, and this would of course inflate the magnitude of our mean gene length in the sample.

Let's repeat this sampling exercise using a larger sample size, say 50.

```
gene.means.n50 <- genelengths %>%
  rep_slice_sample(n = 50, replace = FALSE, reps = 10000) %>%
  summarise(
    mean_length = mean(size, na.rm = T),
    sampsize = as.factor(50)
  )
```

Now plot a histogram, making sure to keep the x- and y-axis limits the same as we used above, so that we can compare the outputs

```
gene.means.n50 %>%
  ggplot(aes(x = mean_length)) +
  geom_histogram(binwidth = 250, color = "black", fill = "lightgrey") +
  xlim(1500, 10000) +
  xlab("Gene length (nucleotides)") +
  ylab("Number of genes") +
  theme_bw()
```

Notice that there is much less positive skew to this frequency distribution.

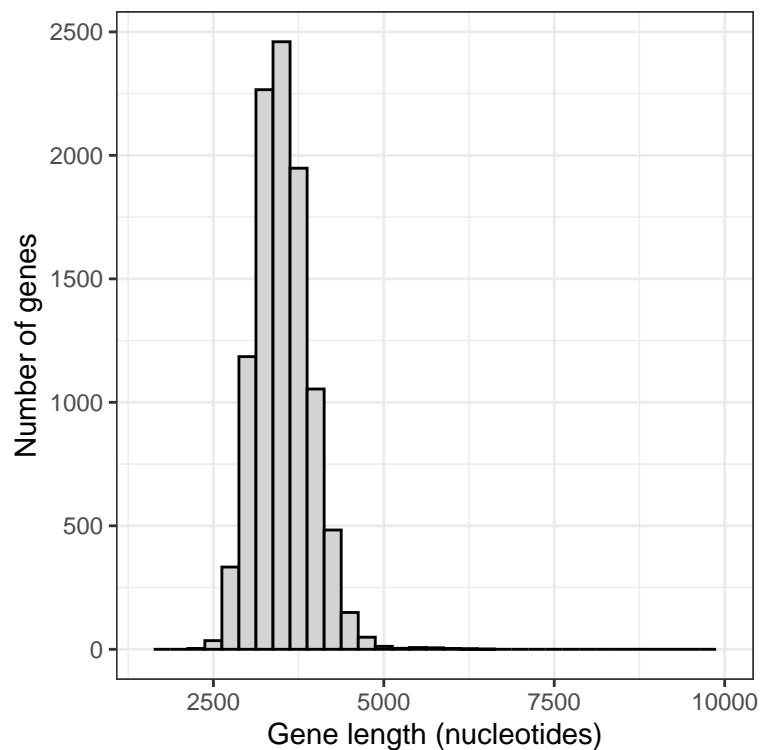


Figure 9.2: Approximate sampling distribution of mean gene lengths from the human genome (10000 replicate samples of size 50).

Let's produce a figure that allows us to directly compare the two sampling distributions.

First we need to combine the data into one tibble.

New tool We can merge two tibbles (or data frames) together by rows using the `bind_rows` function from the `dplyr` package

The `bind_rows` function will simply append one object to another by rows. This requires that the two (or more) objects each have the same variables. In our case, the two objects have the same three variables.

```
gene.means.combined <- bind_rows(gene.means.n20, gene.means.n50)
```

Now we have what we need to produce a graph that enables direct comparison of the two sampling distributions.

This type of graph is not something you'll need to be able to do for assignments!

```
gene.means.combined %>%
  ggplot(aes(x = mean_length, fill = as.factor(sampsizes))) +
  geom_histogram(position = 'identity', colour = "darkgrey", alpha = 0.5, binwidth = 2)
  scale_fill_manual(values=c("#999999", "#E69F00")) +
  xlab("Gene length (nucleotides)") +
  ylab("Number of genes") +
  theme_bw() +
  theme(legend.position = "top") +
  guides(fill = guide_legend(title = "Sample size"))
```

We can see that the smaller sample size (20) yields a broader sampling distribution, and though it's a difficult to see, it also exhibits a longer tail towards larger gene lengths.

With larger sample sizes, we'll get more precise estimates of the true population mean!

1. Follow this online tutorial that helps visualize the construction of a sampling distribution of the mean

9.5 Standard error of the mean

This is the formula for calculating the **standard error of the mean** when the population parameter is unknown (which is usually the case):

$$SE_{\bar{Y}} = \frac{s}{\sqrt{n}}$$

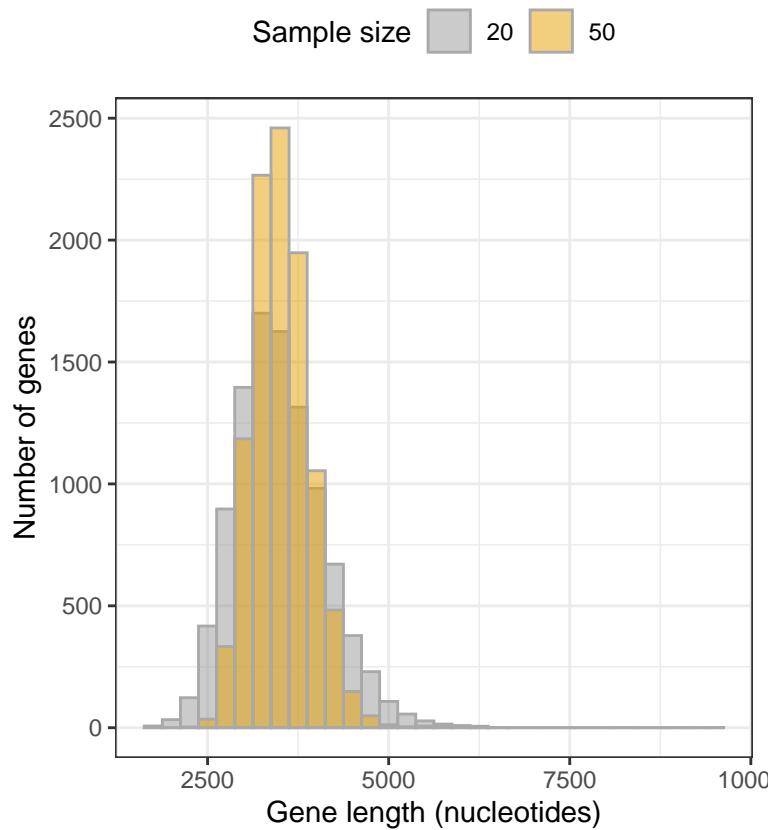


Figure 9.3: Approximate sampling distributions of mean gene lengths from the human genome, `fig.height=`, `message=FALSE`, `warning=FALSE`, using sample sizes of 20 and 50 (10000 replicate samples in each distribution).

This is one measure of uncertainty that we report alongside our sample-based estimate of the population mean.

It represents the standard deviation of the sampling distribution of the mean. Thus, it is a measure of spread for the sampling distribution of the mean.

Calculating the SEM in R

To illustrate how to calculate the SEM, we'll make use of the gene length dataset again.

Let's take a random sample of 20 genes, making sure to set the seed this time so everyone gets the same result. We'll use a seed number of 29, and we'll save our sample of values from the variable "size" in an object (tibble) called "newsamp.n20":

```
set.seed(29)
newsamp.n20 <- genelengths %>%
  slice_sample(n = 20, replace = FALSE) %>%
  select(size)
```

Now let's show the code for calculating the SEM for the mean calculated using values in the "size" variable from our "newsamp.n20" object. We'll output stats to a new object "newsamp.n20.stats" then display the object using `kable`.

```
newsamp.n20.stats <- newsamp.n20 %>%
  summarise(
    Count = n() - naniar::n_miss(size),
    Mean_genelength = mean(size, na.rm = TRUE),
    SD_genelength = sd(size, na.rm = TRUE),
    SEM = SD_genelength/sqrt(Count)
  )
```

You've seen all but the last lines of code before! Importantly, as we've seen before, the "Count" variable, calculated using the `n` function in combination with the `n_miss` function from the `naniar` package, tallies the number of non-missing observations in the variable of interest (here, "size"). This is the sample size that we'll need for the calculation of the SEM.

The last line creates a new variable "SEM" that is calculated using the formula shown above, and inputs from the previous lines' calculations.

Specifically, the sample size "n" value comes from the "Count" variable, and the "s" value comes from the "SD_genelength" variable. And lastly, we use the `sqrt` function to take the square-root of the sample size.

Display the output:

```
kable(newsamp.n20.stats, digits = 4)
```

Count
 Mean_genelength
 SD_genelength
 SEM
 20
 3563.7
 3747.552
 837.9782

If you wish to calculate and report only the mean and SEM for a variable (here, the “size” variable from our tibble “newsamp.n20”):

```
newsamp.n20.stats.short <- newsamp.n20 %>%
  summarise(
    Mean_genelength = mean(size, na.rm = TRUE),
    SEM = sd(size, na.rm = TRUE)/sqrt(n() - naniar::n_miss(size))
  )
```

Display the output:

```
kable(newsamp.n20.stats.short, digits = 4)
```

Mean_genelength
 SEM
 3563.7
 837.9782

TIP It is a good idea to use the longer approach that reports the sample size (“Count”), mean, standard deviation, and SEM. Why? Because in this way you’re reporting each value (n and s) that goes into the calculation of the SEM.

9.6 Rule of thumb 95% confidence interval

The **Rule of Thumb** 95% confidence interval is calculated simply as the sample mean +/- two standard errors.

Thus, the **lower confidence limit** is calculated as the sample mean minus 2 times the standard error, and the **upper confidence limit** is calculated as the sample mean plus 2 times the standard error.

We can re-use code from the previous section for the calculations, and we'll assign the output to a new object called "newsamp.n20.allstats":

```
newsamp.n20.allstats <- newsamp.n20 %>%
  summarise(
    Count = n() - na.omit::n_miss(size),
    Mean_genelength = mean(size, na.rm = TRUE),
    SD_genelength = sd(size, na.rm = TRUE),
    SEM = SD_genelength/sqrt(Count),
    Lower_95_CL = Mean_genelength - 2 * SEM,
    Upper_95_CL = Mean_genelength + 2 * SEM
  )
```

We've added two lines of code, one for the lower confidence limit and one for the upper confidence limit.

Notice that we again use calculations completed in preceding lines as input for the subsequent lines. Specifically, we use the "Mean_genelength" value and the "SEM" value in our confidence limit calculations.

The asterisk (*) denotes multiplication.

Let's look at the output:

```
kable(newsamp.n20.allstats, digits = 4)
```

| | |
|-----------------|--|
| Count | |
| Mean_genelength | |
| SD_genelength | |
| SEM | |
| Lower_95_CL | |
| Upper_95_CL | |
| 20 | |
| 3563.7 | |
| 3747.552 | |
| 837.9782 | |
| 1887.744 | |
| 5239.656 | |

If you are asked to report the rule-of-thumb 95% confidence **interval**, then you'd use the output from your calculations above to report:

$$\mathbf{1887.744 < \mu < 5239.656}$$

Note that it is the population parameter being estimated, μ , that appears in between the lower and upper limits.

To produce greek letters such as “mu”, simply precede the “mu” with a back-slash, then enclose the entire term with dollar signs, as shown at this webpage.

NOTE: In a later tutorial you'll learn how to calculate actual confidence intervals, rather than “rule of thumb” confidence intervals

Chapter 10

Hypothesis testing

Tutorial learning objectives

- Learn the steps to conducting a hypothesis test
- Learn how to simulate a null distribution (using a binomial distribution example)
- Learn how to calculate the P -value
- Learn how write concluding statements for hypothesis tests

10.1 Load packages and import data

Load the `tidyverse`, `skimr`, `naniar`, and `infer` packages:

```
library(tidyverse)
library(skimr)
library(infer)
library(naniar)
```

Import the CSV file called “damselfly”:

```
damselfly <- read_csv("https://raw.githubusercontent.com/ubco-biology/BIO1202/main/data/damselfly.csv")
## # A tibble: 20 x 1
##   species
##   <fct>
## 1 damselfly
## 2 damselfly
## 3 damselfly
## 4 damselfly
## 5 damselfly
## 6 damselfly
## 7 damselfly
## 8 damselfly
## 9 damselfly
##10 damselfly
### ... with 10 more rows, and 1 more variable:
###   .rowid. <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
```

Get an overview of the dataset:

```
damselfly %>%  
  skim_without_charts()
```

(#tab:hyp_overview)Data summary

Name

Piped data

Number of rows

20

Number of columns

1

Column type frequency:

character

1

Group variables

None

Variable type: character

skim_variable

n_missing

complete_rate

min

max

empty

n_unique

whitespace

direction

0

1

9

```
17  
0  
2  
0
```

And view the first handful of rows:

```
damselfly
```

```
## # A tibble: 20 x 1  
##   direction  
##   <chr>  
## 1 clockwise  
## 2 counter_clockwise  
## 3 counter_clockwise  
## 4 clockwise  
## 5 counter_clockwise  
## 6 counter_clockwise  
## 7 counter_clockwise  
## 8 counter_clockwise  
## 9 counter_clockwise  
## 10 counter_clockwise  
## 11 counter_clockwise  
## 12 clockwise  
## 13 counter_clockwise  
## 14 counter_clockwise  
## 15 counter_clockwise  
## 16 counter_clockwise  
## 17 counter_clockwise  
## 18 counter_clockwise  
## 19 counter_clockwise  
## 20 counter_clockwise
```

10.2 Steps to hypothesis testing

Below is a list of all the steps required when conducting a hypothesis test.

Several of the steps we will learn about in later tutorials.

- Identify the appropriate statistical test and thus **null distribution** for the test statistic
- State the null (H_0) and alternative (H_A) hypotheses

- Set an α level (usually 0.05)
- Determine whether a one-tailed or two-tailed test is appropriate (almost always the latter)
- Check assumptions of the statistical test
 - state the assumptions of the test
 - use appropriate figures and / or tests to check whether the assumptions of the statistical test are met
 - transform data to meet assumptions if required
 - if assumptions can't be met (e.g. after transformation), use non-parametric test and repeat the first four steps above
- Provide an appropriate figure, including figure caption, to visualize the raw or transformed data
- Provide a line or two interpreting your figure, and this may inform your concluding statement
- Calculate the test statistic value using the observed data
- Use the **null distribution** to determine the **P -value** associated with the observed test statistic
- Draw the appropriate conclusion and communicate it clearly
- Calculate and include a confidence interval (e.g. for t -tests) or R^2 value (e.g. for ANOVA) when appropriate

10.3 An hypothesis test example

When defending territories along streams, males of the damselfly species *Calopteryx maculata* (pictured here) often battle intruders in the air, flying around and around in circles with their foe for minutes at a time. No one knows whether these battles are flown in a consistent direction, i.e. predominantly clockwise or counter-clockwise, as would be expected if the damselflies exhibited a form of “handedness”, like many animals do.

A researcher was curious about this because he had worked with these damselflies for years and witnessed many territorial bouts (see exmaple research here).

The researcher conducted a study (fictional) in which he video-recorded 20 male damselflies defending territories (all randomly sampled from a population), and determined the predominant direction of flight during circular flight battles.

One battle per damselfly was recorded, and each battle was known to involve a unique combattant.

He found that in 17 out of 20 bouts the damselflies flew in the counter-clockwise direction.

Should this result be considered evidence of handedness in this population?

For this example, we'll use the imported `damsel` dataset, which includes the single categorical variable of interest "direction", and there are two categories: "clockwise" and "counter_clockwise".

damsel

```
## # A tibble: 20 x 1
##   direction
##   <chr>
## 1 clockwise
## 2 counter_clockwise
## 3 counter_clockwise
## 4 clockwise
## 5 counter_clockwise
## 6 counter_clockwise
## 7 counter_clockwise
## 8 counter_clockwise
## 9 counter_clockwise
## 10 counter_clockwise
## 11 counter_clockwise
## 12 clockwise
## 13 counter_clockwise
## 14 counter_clockwise
## 15 counter_clockwise
## 16 counter_clockwise
## 17 counter_clockwise
## 18 counter_clockwise
## 19 counter_clockwise
## 20 counter_clockwise
```

10.3.1 Following the steps to hypothesis testing

For the damselfly example, we have $n = 20$ **random trials**, each of which has two possible outcomes: clockwise battle or counter-clockwise battle. This

conforms to the expectations of a **binomial test**, for which the *binomial distribution* is used to generate the appropriate **null distribution**. We'll learn about the binomial test in a later tutorial.

For the present tutorial, we're going to generate a null distribution via simulation.

To do this, we need to consider what our “null expectation” is, i.e. if there was truly no handedness among the damselflies.

Our null expectation is that clockwise and counter-clockwise battles would occur with equal frequency. In other words, if we focus on counter-clockwise battles as the category of interest, then on average - across many, many battles - we'd expect those to make up $p_0 = 0.5$ of the battles. This is just like flipping a fair coin: over the long run (i.e. over many, many coin flips), we'd expect heads to make up half the outcomes, i.e. $p_0 = 0.5$.

We can now specify our H_0 and H_A :

H_0 : The proportion of damselfly battles in the population flown in the counter-clockwise direction is 0.5 ($p_0 = 0.5$)

H_A : The proportion of damselfly battles in the population flown in the counter-clockwise direction is not 0.5 ($p_0 \neq 0.5$)

We'll set $\alpha = 0.05$.

Although our H_0 makes a specific statement about a proportion (p_0), our *test statistic* will actually be the **number of battles that occurred in a counter-clockwise direction**. Of course, we can re-express this number as a proportion, but convention is that we use the number (frequency) instead.

As indicated by the **H_A** statement, we'll use a 2-tailed alternative hypothesis because we have no reason to exclude the possibility that clockwise battles are in fact more predominant.

TIP: The symbol for “not equal” can be done using this syntax: `\ne`. Simply type that as part of your regular markdown text. A subscript can be added by prefacing the character with a “`_`”. Thus, in the alternative hypothesis statement above, we used `$p_0 \neq 0.5$` to create the $p_0 \neq 0.5$.

10.3.2 Simulating a “null distribution”

For our damselfly example, we can easily simulate the random trials (20 territorial flight battles, each yielding one of 2 outcomes) by using what we learned in the previous tutorial about simulations in R.

First, we'll create a tibble object called “null_options” that holds a categorical variable “direction” with two categories, “clockwise” and “counter_clockwise”:

```
null_options <- tibble(direction = c("clockwise", "counter_clockwise"))
```

Have a look at the resulting object:

```
null_options
```

```
## # A tibble: 2 x 1
##   direction
##   <chr>
## 1 clockwise
## 2 counter_clockwise
```

Now we can use the `rep_slice_sample` function from the `infer` package (which we learned about in the last tutorial) to simulate a large number of repetitions of our $n = 20$ random trials. The number of repetitions is governed by the `reps` argument.

We'll set the seed first, to ensure everyone gets the same result for this initial demonstration repetition.

Let's start with a single repetition of $n = 20$ random trials. We'll explain the code after:

```
set.seed(199)
num.reps <- 1
null_options %>%
  rep_slice_sample(n = 20, replace = TRUE, weight_by = c(0.5, 0.5), reps = num.reps) %>%
  count(direction)

## # A tibble: 2 x 3
## # Groups:   replicate [1]
##   replicate direction      n
##   <int>     <chr>    <int>
## 1 1         clockwise     9
## 2 2         counter_clockwise 11
```

- We first set the seed (using integer 199 here; arbitrarily chosen)
- Next we create an object “`num.reps`” that is simply one number: the number of replications we wish to run (here, 1)
- We tell R we’re using the “`null_options`” object (created previously) for input into the subsequent lines, using the pipe (`%>%`) to continue the code
- We use the `rep_slice_sample` function, setting the number of trials n to 20, and ensuring that we use sampling with replacement here (“`replace = T`”) because our “`null_options`” object has only two observations to sample from, and we need 20 total;

- You can think of the “weight_by” argument as a way to control the relative probability of sampling each of the rows in the input tibble (here, the “null_options” tibble). Here we provide two values (0.5, 0.5), specifying that we want each of the two rows in the tibble to have equal likelihood of being sampled at random. This corresponds to what we want for our null expectation, and for the generation of the null distribution
- The last argument to the `rep_slice_sample` function is “reps = num.reps”, using our created object to tell R that we want only one repetition of the 20 trials
- In the last line of code we use the `count` function to tally the number of observations belonging to each of the two categories in the “direction” variable

So you see that the output above is a tibble with 3 variables: “replicate”, “direction”, and “n”.

We’re specifically interested in the number of times (the frequency) out of the 20 trials that the damselfly battle was predominantly “counter_clockwise” (what we consider a “success”). So in our first repetition here, 11 of the trials were predominantly in the counter-clockwise direction.

Now what we need to do is run many thousands of repetitions (we’ll do 10^5), and each time tally the number of battles (out of the 20 trials) in which the direction was counter-clockwise.

This will generate a reasonable approximation of a **null distribution** for our study.

After the setting the seed, we create an object that simply holds the number of replications we wish to run.

```
set.seed(199)
num.reps <- 100000
null_distribution <- null_options %>%
  rep_slice_sample(n = 20, replace = TRUE, weight_by = c(0.5, 0.5), reps = num.reps) %>%
  count(direction) %>%
  filter(direction == "counter_clockwise")
```

In the above code chunk, we:

- assign our output to a new object called “dull_distribution”
- We changed the “reps” argument for the `rep_slice_sample` function to 10^5 (all other arguments stay the same).
- We use the `count` function to tally the number of observations belonging to each of the two categories in the “direction” variable
- We then `filter` the output from our sampling to the rows with the “direction” variable equal to “counter_clockwise”

Let's look at the resulting object:

```
null_distribution
```

```
## # A tibble: 100,000 x 3
## # Groups:   replicate [100,000]
##   replicate direction      n
##   <int>     <chr>     <int>
## 1 1         counter_clockwise 11
## 2 2         counter_clockwise  9
## 3 3         counter_clockwise  8
## 4 4         counter_clockwise 11
## 5 5         counter_clockwise 13
## 6 6         counter_clockwise  9
## 7 7         counter_clockwise  9
## 8 8         counter_clockwise 12
## 9 9         counter_clockwise  6
## 10 10        counter_clockwise  9
## # i 99,990 more rows
```

We have a tibble with the number of rows equal to the number of replications we performed (100000), three variables (“replicate”, “direction”, and “n”), and the key information are the numbers in the “n” variable. These represent the number of battles (out of 20) that were predominantly in the counter-clockwise direction.

We can now create a histogram of these data, thus producing an approximation of a “null distribution”.

The only difference here, compared to previous times we've created histograms, is that we add the `after_stat(density)` argument to the `aes` part of the `ggplot` function. This tells R to display the relative frequency (= probability density) rather than the raw counts on the y-axis. Notice we also use “binwidth = 1”, because we want to see a bar for each potential outcome (0 through 20).

```
null_distribution %>%
  ggplot(aes(x = n, after_stat(density))) +
  geom_histogram(binwidth = 1, fill = "lightgrey", colour = "black") +
  xlim(0, 20) +
  xlab("Number of counter-clockwise battles") +
  ylab("Relative frequency") +
  theme_bw()
```

The null distribution above shows us that if the null hypothesis was true, the most probable outcome would be 10 counter-clockwise battles out of 20. But other outcomes are of course possible, with decreasing probability towards zero and twenty.

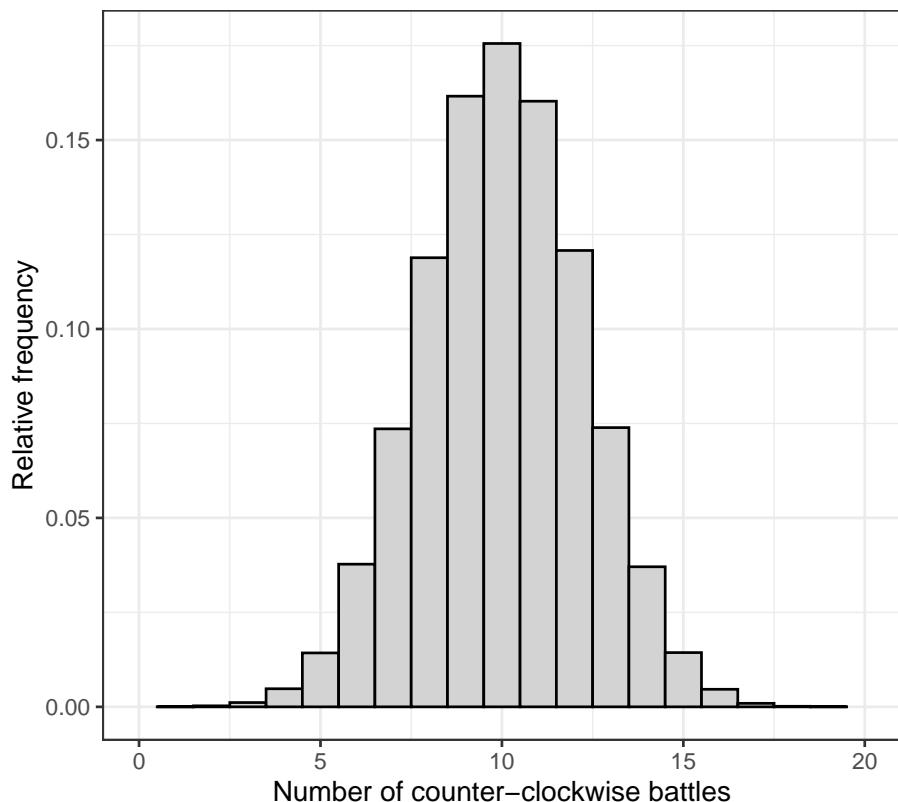


Figure 10.1: Approximate null distribution for the damselfly study, with $n = 20$ trials, and probability of success = 0.5 in each trial. Data are from 100000 repetitions.

10.3.3 Calculating the *P*-value

The ***P*-value** is defined as the probability of observing an outcome as extreme or more extreme as the observed value of the test statistic, if the null hypothesis were true.

In our damselfly study, we observed 17 out of 20 battles to be in the counter-clockwise direction.

We can calculate a ***P*-value** as follows:

- first calculate the fraction of the 10^5 values in our null distribution that are equal to or greater than our observed test statistic value (here, 17)
- if we have a 2-tailed alternative hypothesis, then we also need to calculate the fraction of the 10^5 values in our null distribution that are less than or equal to 3, which is the mirror number to 17 in the opposite side of the distribution.

Although our approximate null distribution that we created looks quite symmetrical, it's best not to assume it is (because recall that we generated this via simulation). Therefore we don't simply multiply the first value we calculate above by two. In future tutorials you'll learn how to calculate ***P*-values** using built-in R functions and theoretical probability distributions.

In the Introductory R Tutorials you learned how to use **logical comparison operators** such as `>` and `<=`. We'll use these here.

Let's find out how many values within the variable `n` in our `null_distribution` tibble are less than or equal to 3, or greater than or equal to 17:

```
null_distribution %>%
  filter(n <= 3 | n >= 17) %>%
  nrow()
```

```
## [1] 254
```

This code uses the `filter` function to return only the rows that meet the condition specified: the “`n`” variable value less than or equal to 3, OR (using the “`|`” symbol) greater than or equal to 17.

The `nrow` function then simply counts the number of rows in the resulting (filtered) tibble.

We see that among the 10^5 replications we ran, 254 yielded outcomes in which 3 or fewer, or 17 or more out of 20 battles were in the counter-clockwise direction.

To calculate the *P*-value, we need to divide the 254 by the total number of reps that we conducted, here 10^5 .

```
null_distribution %>%
  filter(n <= 3 | n >= 17) %>%
  nrow() / num.reps

## [1] 0.00254
```

So the **P-value** associated with our observed test statistic is 0.00254.

10.3.4 Writing a concluding statement

It is important to write a concluding statement that talks about the actual findings of the study. For example:

In their territorial bouts, the damselflies flew in a counter-clockwise direction significantly more than expected (17 counter-clockwise flights out of $n = 20$ trials; P -value = 0.003). This is consistent with the idea that there is “handedness” in this population of *C. maculata*.

It is also crucial that you report the sample size (here, number of random trials), the value of the observed test statistic (here 17), and the associated **P-value**. Note we rounded the **P-value** to three decimal places.

When we learn new types of statistical test, we’ll adjust our concluding statements accordingly.

1. Using this tutorial as a guide, try to repeat all the steps of a hypothesis test for Example 6.2 in the text book (concerning handedness in toads).

Chapter 11

Analyzing a single categorical variable

Tutorial learning objectives

- Learn how to calculate a standard error for a proportion
- Learn how to calculate a confidence interval for a proportion
- Learn about the binomial distribution and the `dbinom` function
- Learn how to test hypotheses about a proportion using the **binomial test**
- Learn an example of the confidence interval approach to hypothesis testing
- Learn how to conduct a χ^2 **goodness of fit test** using a proportional model

11.1 Load packages and import data

Load the `tidyverse`, `skimr`, `naniar`, `knitr`, and `janitor` packages:

```
library(tidyverse)
library(skimr)
library(naniar)
library(knitr)
library(janitor)
```

We will also need a new package called `binom`, so install that package using the procedure you previously learned, then load it:

```
library(binom)
```

Import the `damselify.csv` data set we used in the preceding tutorial, and also the `birds.csv` dataset we used in an earlier tutorial.

```
damselify <- read_csv("https://raw.githubusercontent.com/ubco-biology/BIO1L202/main/data/damselify.csv")

## Rows: 20 Columns: 1
## -- Column specification -----
## Delimiter: ","
## chr (1): direction
##
## 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.

birds <- read_csv("https://raw.githubusercontent.com/ubco-biology/BIO1L202/main/data/birds.csv")

## Rows: 86 Columns: 1
## -- Column specification -----
## Delimiter: ","
## chr (1): type
##
## 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.
```

Recall what the `damselify` dataset looks like:

```
damselify
```

```
## # A tibble: 20 x 1
##   direction
##   <chr>
## 1 clockwise
## 2 counter_clockwise
## 3 counter_clockwise
## 4 clockwise
## 5 counter_clockwise
## 6 counter_clockwise
## 7 counter_clockwise
## 8 counter_clockwise
## 9 counter_clockwise
## 10 counter_clockwise
```

```
## 11 counter_clockwise
## 12 clockwise
## 13 counter_clockwise
## 14 counter_clockwise
## 15 counter_clockwise
## 16 counter_clockwise
## 17 counter_clockwise
## 18 counter_clockwise
## 19 counter_clockwise
## 20 counter_clockwise
```

The data show the predominant direction (either clockwise or counter-clockwise) of 20 circular battles between male damselflies.

And remind yourself what the birds dataset looks like:

```
birds
```

```
## # A tibble: 86 x 1
##   type
##   <chr>
## 1 Waterfowl
## 2 Predatory
## 3 Predatory
## 4 Waterfowl
## 5 Shorebird
## 6 Waterfowl
## 7 Waterfowl
## 8 Songbird
## 9 Predatory
## 10 Waterfowl
## # i 76 more rows
```

These data describe the category of bird (variable “type” that has 4 different categories) for a random sample of 86 birds sampled at a marsh habitat.

11.2 Estimating proportions

Recall that the key descriptor for a categorical variable is a **proportion**. And when we wish to draw inferences about a categorical attribute within a population of interest, we take a random sample from the population to *estimate* the true proportion p of the population with that attribute. Our estimate is denoted \hat{p} .

In a previous tutorial we used the “birds” dataset to practice creating a “frequency table” that showed the *relative frequencies* of birds falling in each category of the categorical variable “type”. These relative frequencies are equivalent to their respective proportions.

So, for the birds dataset, let’s produce a frequency table, but instead of using the term “relative frequency” we’ll use “p_hat” to denote that this is our estimate of the proportion of birds in that category.

We’ll put the output in an object “bird.freq.table” then we’ll display it after:

```
bird.freq.table <- birds %>%
  count(type, sort = TRUE) %>%
  mutate(p_hat = n / sum(n))
```

Now show the table:

```
kable(bird.freq.table, digits = 4)
```

| type | n | p_hat |
|-----------|----|--------|
| Waterfowl | 43 | 0.5000 |
| Predatory | 29 | 0.3372 |
| Shorebird | 8 | 0.0930 |
| Songbird | 6 | 0.0698 |

We can see that the proportion of birds at the marsh that belong to the “Predatory” category is about 0.34. Assuming we had a good random sample, this is our best estimate of the true proportion of marsh birds that are predatory.

Being an estimate, however, we need to attach some measure of uncertainty to it.

In a previous tutorial we learned how to calculate the standard error and rule-of-thumb 95% confidence interval for the **mean**.

Here we'll learn how to calculate these measures of uncertainty for a **proportion**.

11.2.1 Standard error for a proportion

Here's the equation for the standard error for the proportion:

$$SE_{\hat{p}} = \sqrt{\frac{\hat{p}(1 - \hat{p})}{n}}$$

Let's use the "birds" dataset to demonstrate the calculation.

Let's re-create the table again:

```
bird.freq.table <- birds %>%
  count(type, sort = TRUE) %>%
  mutate(p_hat = n / sum(n))
```

Show the table:

```
kable(bird.freq.table, digits = 4)
```

| type | n | p_hat |
|-----------|----|--------|
| Waterfowl | 43 | 0.5000 |
| Predatory | 29 | 0.3372 |
| Shorebird | 8 | 0.0930 |

```
Songbird
```

```
6
```

```
0.0698
```

That table does show us our estimate of the proportion of birds that are predatory. We'll isolate that value later.

But we also need to get the total sample size, i.e. the total number of birds sampled.

To do this, we need to make sure not to count any missing values, and we can use the `n_complete` function from the `naniar` package:

```
birds.sampszie <- birds %>%
  select(type) %>%
  n_complete()
```

In the preceding code:

- We first ensure the output will be put in a new object called “birds.sampszie”
- We provide the tibble `birds`, which is what is fed to the subsequent code
- We then use `select` to select the variable of interest “type”
- We then use the `n_complete` function to get the total number of non-missing observations in the variable.

Now let's see what the outcome was:

```
birds.sampszie
```

```
## [1] 86
```

In previous tutorials we used a bit more cumbersome way to calculate the total number of complete observations in a variable. The `n_complete` function will simplify things.

Now that we have n (above), stored in the object “birds.sampszie”, we need to extract the “`p_hat`” value associated with the “Predatory” category, and we'll store this in an object called “`pred.phat`”:

```
pred.phat <- bird.freq.table %>%
  filter(type == "Predatory") %>%
  select(p_hat)
```

In the preceding chunk we:

- assign our output to an object “pred.phat”
- use the `filter` function to get the rows where the “type” variable is equal to the “Predatory” category
- use the `select` function to return (select) the “p_hat” variable only

Let’s look at what this produced:

```
pred.phat
```

```
## # A tibble: 1 x 1
##   p_hat
##   <dbl>
## 1 0.3372
```

So this produced a tibble called “pred.phat” with a variable “p_hat”, and it has one value - our proportion estimate \hat{p} .

TIP: We could have calculated and isolated the “P-hat” value for predatory birds all in one go, using the following code:

```
pred.phat <- birds %>%
  count(type, sort = TRUE) %>%
  mutate(p_hat = n / sum(n)) %>%
  filter(type == "Predatory") %>%
  select(p_hat)
```

That demonstrates the power of the pipe approach!

OK, now we’re ready to calculate $SE_{\hat{p}} = \sqrt{\frac{\hat{p}(1-\hat{p})}{n}}$

Here’s how we do this using basic R syntax, recalling that our “n” (86 birds total) is stored in the object “birds.sampsiz”.

We’ll store the value in a new object called “SE_phat”:

```
SE_phat <- sqrt(pred.phat * (1 - pred.phat) / birds.sampsiz)
```

And now see what the value is:

```
SE_phat
```

```
##       p_hat
## 1 0.0509787
```

Reporting the proportion and standard error

To properly report a proportion estimate along with its standard error, we need to know how to produce a “plus/minus” symbol in markdown.

Here’s what you type in the regular text area of your markdown document (not in a code chunk).

`\pm`

The `\pm` is the syntax for a plus-minus symbol. More symbols can be found at this website.

And so you insert that text in between your estimate \hat{p} and the $SE_{\hat{p}}$, as such: 0.34 ± 0.051 .

11.2.2 Confidence interval for a proportion

Although a number of options are available for calculating a confidence interval for a proportion, we’ll use the *Agresti-Coull* method, as it has desirable properties:

$$p' \pm 1.96 \cdot \sqrt{\frac{p'(1-p')}{N+4}}$$

where

$$p' = \frac{X+2}{n+4}$$

The **margin of error** from the above equation is this part:

$$1.96 \cdot \sqrt{\frac{p'(1-p')}{N+4}}$$

This margin of error is the value that we *subtract* to our proportion estimate \hat{p} to get the **lower 95% confidence limit**, and we *add* that value to our proportion estimate \hat{p} to get the **upper 95% confidence limit**.

We’ll again use the “birds” data, and the estimated proportion of birds that are predatory.

Let’s again calculate the sample size of birds:

```
birds.sampsize <- birds %>%
  select(type) %>%
  n_complete()
```

And here's the frequency table we constructed:

```
bird.freq.table <- birds %>%
  count(type, sort = TRUE) %>%
  mutate(p_hat = n / sum(n))
```

Show the table:

```
bird.freq.table
```

```
## # A tibble: 4 x 3
##   type      n   p_hat
##   <chr>    <int>   <dbl>
## 1 Waterfowl     43  0.5
## 2 Predatory      29  0.3372
## 3 Shorebird       8  0.09302
## 4 Songbird        6  0.06977
```

Looking at the formula for the *Agresti-Coull* confidence interval, we see that we need X - the number of “successes”, which in our case is the number of birds in the category “Predatory”. This is provided in our table above. So let’s extract that information.

We’ll store our value of “ X ” in a new object called “bird.X”:

```
bird.X <- bird.freq.table %>%
  filter(type == "Predatory") %>%
  select(n)
```

Now let’s see the output:

```
bird.X
```

```
## # A tibble: 1 x 1
##       n
##   <int>
## 1     29
```

Unlike the standard error calculation, for which we did actual math using R, for the confidence interval we’ll make use of the `binom.confint` function from the `binom` package. Have a look at the help page:

```
?binom.confint
```

The function takes a handful of arguments, including the value of X , the number of trials n , the confidence level we wish to use (typically 0.95 corresponding to a 95% confidence interval), and the method one wishes to use.

Let's try it out using our bird data.

The one catch in the code below is that we can't simply provide the name of our "bird.X" object to provide the value of X for the `binom.confint` function; instead we need to specify that the value we want to use is stored in the variable called "n" within the "bird.X" object. Thus, we use `bird.X$n`. The dollar sign allows us to specify a specific variable within the tibble.

Then, we use "ac" to specify the "Agresti-Coull" methods.

We'll assign our output to a new object called "confint.results":

```
confint.results <- binom.confint(x = bird.X$n, n = birds.sampsize, conf.level = 0.95, m
```

Have a look at the output using the `kable` function to make it nicer looking:

```
kable(confint.results, digits = 4)
```

| |
|---------------|
| method |
| x |
| n |
| mean |
| lower |
| upper |
| agresti-coull |
| 29 |
| 86 |
| 0.3372 |
| 0.2459 |
| 0.4424 |

The function provides us with the values of X , n , the proportion estimate (strangely called "mean"), and the lower and upper confidence limits.

Reporting the confidence interval for a proportion

The appropriate way to report the confidence interval is as follows:

The 95% Agresti-Coull confidence interval is: $0.246 < p < 0.442$.

1. What is the best estimate of the proportion of the marsh birds that belong to the “Shorebird” category? What is the standard error of the proportion, and the 95% Agresti-Coull confidence interval?

11.3 Binomial distribution

The binomial distribution provides the probability distribution for the number of “successes” in a fixed number of independent trials, when the probability of success is the same in each trial.

Here's the formula:

$$\Pr[X] = \binom{n}{X} p^X (1-p)^{n-X}$$

where $\Pr[X]$ is the probability of X

$$\binom{n}{X} = \frac{n!}{X!(n-X)!}$$

It turns out there's a handy function `dbinom` (available in the base R package) that will calculate the exact probability associated with any particular outcome for a random trial with a given sample space (set of outcomes) and probability of success. It uses the equation shown above.

Check the help file for the function:

```
?dbinom
```

Dice example

Imagine rolling a fair, 6-sided die $n = 6$ times (six random trials). Let's consider rolling a "4" a "success".

What is the probability of observing two fours (i.e. two successes) in our 6 rolls of the die (random trials)?

We have $X = 2$ (the number of successes), $p = 1/6$ (the probability of a success in each trial), and $n = 6$ (the number of trials).

Here's the code, where "x" represents our " X ", "size" represents the number of trials (" n "), and "prob" is the probability of success in each trial (here, $1/6$).

We'll create an object to hold the number of trials we wish to use first:

```
num.trials <- 6
dbinom(x = 2, size = num.trials, prob = 1/6)
```

```
## [1] 0.2009388
```

Thus, the probability of rolling two fours (i.e. having 2 successes) out of 6 rolls of the dice is about 0.201.

In order to get the probabilities associated with *each* possible outcome (i.e. 0 through 6 successes), we use the code shown in the chunk below.

```
exact.probs.6 <- tibble(
  X = 0:num.trials,
  probs = dbinom(x = 0:num.trials, size = num.trials, prob = 1/6)
)
exact.probs.6
```

```
## # A tibble: 7 x 2
##       X     probs
##   <int>    <dbl>
## 1     0 0.3349
## 2     1 0.4019
## 3     2 0.2009
## 4     3 0.05358
## 5     4 0.008038
## 6     5 0.0006430
## 7     6 0.00002143
```

Above we created a new tibble object (using the function `tibble`) called "exact.probs.6", with a variable "X" that holds each of the possible outcomes (X

= 0 through 6 or the number of trials), and “probs” that holds the probability of each outcome, calculated using the `dbinom` function:

See that the `dbinom` function will accept a vector of values of `x`, for which the associated probabilities are calculated.

Now let’s use these exact probabilities to create a barplot showing an exact, discrete probability distribution, corresponding to the binomial distribution with a sample size (number of trials) of $n = 6$ and a probability of success $p = 1/6$:

```
ggplot(exact.probs.6, aes(y = probs, x = X)) +
  geom_bar(stat = "identity", fill = "lightgrey", colour = "black") +
  xlab("Number of successes (X)") +
  ylab("Pr[X]") +
  theme_bw()
```

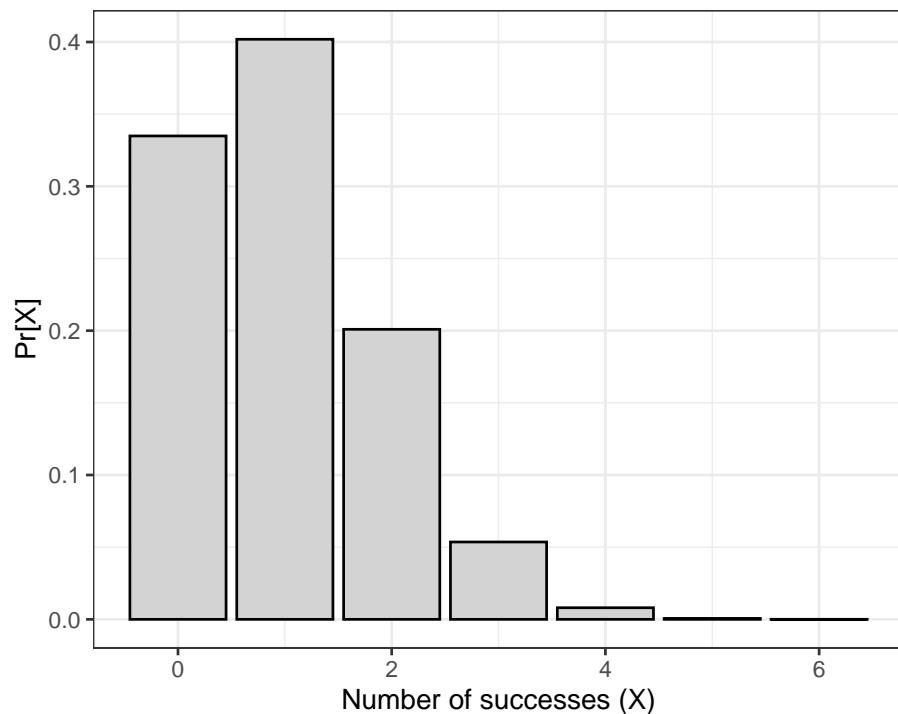


Figure 11.1: Probability of obtaining X successes out of 6 random trials, with probability of success = $1/6$.

What if we wished to calculate the probability of getting *at least* two fours in our 6 rolls of the dice?

Consult the bar chart above. Recall that “rolling a 4” is our definition of a “success” (it could have been “rolling a 1”, or “rolling a 5” - these all result in the same calculation). Thus to calculate the probability of getting at least 2 successes we need to sum up the probabilities associated with getting 2, 3, 4, 5, and 6 successes.

We can do this using the following R code:

```
probs.2_to_6 <- dbinom(x = 2:num.trials, size = num.trials, prob = 1/6)
sum(probs.2_to_6)

## [1] 0.2632245
```

Note that we ask the `dbinom` function to do the calculation for each of the `2:num.trials` outcomes of interest. We store these calculated probabilities in a new object “`probs.2_to_6`”.

Then, we use the `sum` function to sum up the probabilities within that object.

The resulting value of 0.2632245 looks about right based on our bar chart!

Now let’s increase the number of trials to $n = 15$, and compare the distribution to that observed using $n = 6$:

```
num.trials <- 15
exact.probs.15 <- tibble(
  X = 0:num.trials,
  probs = dbinom(x = 0:num.trials, size = num.trials, prob = 1/6)
)
exact.probs.15

## # A tibble: 16 x 2
##       X     probs
##   <int>    <dbl>
## 1     0 6.491e- 2
## 2     1 1.947e- 1
## 3     2 2.726e- 1
## 4     3 2.363e- 1
## 5     4 1.418e- 1
## 6     5 6.237e- 2
## 7     6 2.079e- 2
## 8     7 5.346e- 3
## 9     8 1.069e- 3
```

```
## 10    9 1.663e- 4
## 11   10 1.996e- 5
## 12   11 1.814e- 6
## 13   12 1.210e- 7
## 14   13 5.583e- 9
## 15   14 1.595e-10
## 16   15 2.127e-12
```

Now plot the binomial probability distribution:

```
ggplot(exact.probs.15, aes(y = probs, x = X)) +
  geom_bar(stat = "identity", fill = "lightgrey", colour = "black") +
  xlab("Number of successes (X)") +
  ylab("Pr[X]") +
  theme_bw()
```

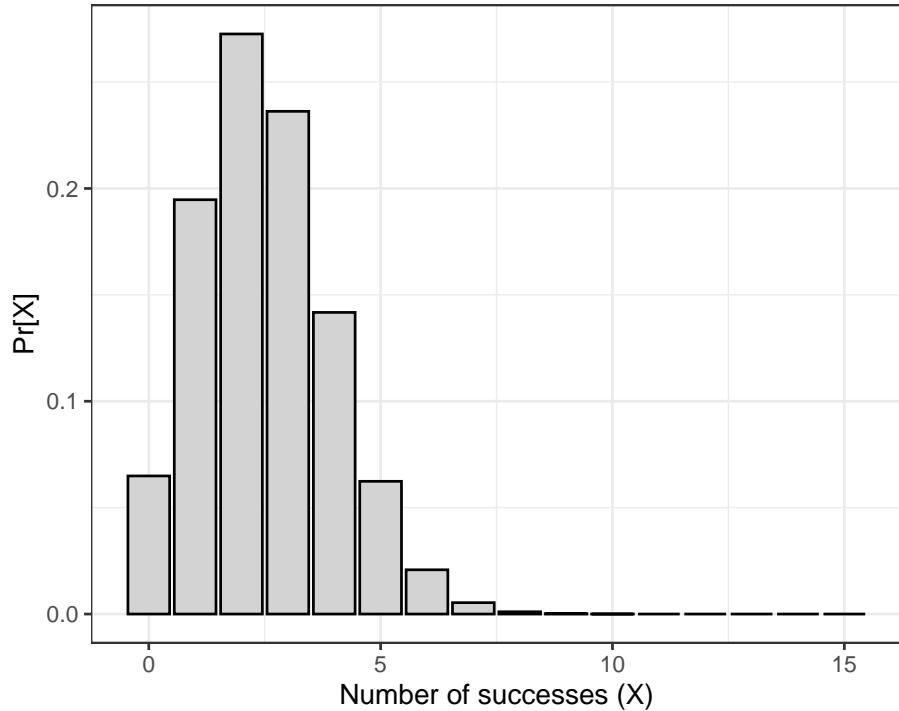


Figure 11.2: Probability of obtaining X successes out of 15 random trials, with probability of success = $1/6$.

1. Challenge: Binomial probabilities

- Use the `dbinom` function to calculate the probability of rolling three “2”s when rolling a fair six-sided die 20 times.
- Produce a graph of a discrete probability distribution for this scenario: $p = 1/4$, and $n = 12$.

11.4 Binomial test

We previously learned about estimating proportions, and calculating measures of uncertainty for those estimates.

Now we'll learn a procedure for testing hypotheses about proportions.

For this example we'll use the `damselfly` dataset we used in the previous tutorial regarding hypothesis tests.

Recall that the researcher found that in 17 out of 20 circular battles (or “bouts”) the damselflies flew in the counter-clockwise direction.

The question was: should this result be considered evidence of handedness in this population?

Take a moment to refresh your memory regarding the steps to hypothesis testing.

We'll explain first how this is a test about a proportion.

Considering the question posed above, we need to first think about what we'd expect if there was truly **no** “handedness” in the population of damselflies. In this case, then we'd expect the circular battles to occur with equal frequency in both clockwise and counter-clockwise directions.

In other words, if there was no handedness, we'd expect the *proportion* of battles that are counter-clockwise (the direction we'll arbitrarily call a “success”) to equal $p = 0.5$.

- We'll use a **binomial test** here, because this is the **most appropriate (and powerful) test** when comparing the observed number of “successes” in a dataset to the number expected under a null hypothesis. Or put another way, we compare an observed *proportion* of successes in a dataset to the proportion expected under a null hypothesis.

Let's devise an appropriate null and alternative hypothesis for this question.

H₀: The proportion of damselfly battles in the population flown in the counter-clockwise direction is 0.5 ($p_0 = 0.5$)

H_A: The proportion of damselfly battles in the population flown in the counter-clockwise direction is not 0.5 ($p_0 \neq 0.5$)

- We'll use an α level of 0.05.
- It is a two-tailed alternative hypothesis; there is no reason to eliminate the possibility that the damselflies exhibit right- or left-“handedness”
- The binomial test assumes that the random trials were independent, and the probability of success was equal in each trial - we'll assume so!
- We don't need a figure for this test
- The test statistic is the number of battles that were flown predominantly in the counter-clockwise direction (the direction we arbitrarily chose as a “success”).
- The binomial test calculates an exact P -value for us (using the binomial equation), and we don't need to rely on an approximate null distribution, like we do for other tests.

Let's conduct the test now.

We use the `binom.test` function that is from the base R package.

```
?binom.test
```

Let's show the code then explain after:

```
binom.test.results <- binom.test(x = 17, n = 20, p = 0.5, alternative = "two.sided")
```

- We create a new object “`binom.test.results`” to hold the results
- the arguments for the `binom.test` function include the number of successes (`x`), the number of trials (`n`), the null hypothesized proportion (`p`), and we specify that the alternative hypothesis is “`two.sided`” - which is almost always the case

Now let's look at the output:

```
binom.test.results

##
##  Exact binomial test
##
## data: 17 and 20
## number of successes = 17, number of trials = 20, p-value = 0.002577
## alternative hypothesis: true probability of success is not equal to 0.5
## 95 percent confidence interval:
##  0.6210732 0.9679291
## sample estimates:
## probability of success
##                         0.85
```

The output from the binomial test includes the number of successes, the number of trials, and the calculated P -value. It also includes a 95% confidence interval for the proportion.

The confidence interval provided by this `binom.test` function is *not* recommended. Rather, you should **ALWAYS** use the Agresti-Coull method to calculate a confidence interval for a proportion, as shown in the previous tutorial.

Let's now calculate the Agresti-Coull 95% confidence interval for the proportion, as we learned previously, using the `binom.confint` function from the `binom` package:

```
damsel.confint.results <- binom.confint(x = 17, n = 20, conf.level = 0.95, methods = "ac")
```

And look at the output:

```
kable(damsel.confint.results, digits = 4)
```

method

x

n

mean

lower

upper

agresti-coull

17

20

0.85

0.6312

0.9561

Now we have all the ingredients for a proper concluding statement.

This is an example of an appropriate concluding statement for a binomial test:

Counter-clockwise battles occurred with significantly greater frequency than expected (17 of 20 battles; observed proportion of counter-clockwise battles = 0.85; Binomial test; P -value = 0.003; Agresti-Coull 95% confidence interval: $0.631 < p < 0.956$).

11.5 Confidence interval approach to hypothesis testing

In the preceding tutorial, we calculated the Agresti-Coull 95% confidence interval for the proportion of damselfly battles flown in the counter-clockwise direction as: $0.631 < p < 0.956$.

Given that the interval excludes (does not encompass) the null hypothesized proportion of $p_0 = 0.5$, we can reject the null hypothesis.

In this case, the appropriate concluding statement would be:

Counter-clockwise battles occurred in a significantly higher proportion than 0.5 (observed proportion of counter-clockwise battles = 0.85; Agresti-Coull 95% confidence interval: $0.631 < p < 0.956$).

1. **Binomial hypothesis test practice:** Using the present tutorial as a guide, use a **binomial test** to address the question posed in Example 6.2 in the text book (concerning handedness in toads). Be sure to include all the steps of a hypothesis test.

11.6 Goodness-of-fit tests

This tutorial is under construction, and will not be covered in 2022

In this tutorial, we continue to learn how to test hypotheses about a categorical variable.

We previously learned how to test a hypothesis about frequencies or proportions when the variable has only two categories of interest, i.e. success and failure (a binary variable). We used a binomial test for this purpose. It is important to note, however, that even in cases where the variable has more than two categories (e.g. hair colour: brown, black, blonde, red), one can define a particular category (e.g. red hair) as a “success”, and the remaining categories as failures, in which case we have simplified our variable to a binary categorical variable.

For testing hypotheses about frequencies or proportions when there are more than two categories, we use goodness of fit (GOF) tests. In general, these types of test evaluate how well an observed discrete frequency (or probability) distribution fits some hypothesized frequency distribution. We'll demonstrate this next.

Chapter 12

Analyzing associations between two categorical variables

Tutorial learning objectives

- Learn about the odds Ratio for a **2 x 2** contingency table
 - Estimate the odds of an outcome
 - Estimate the odds ratio
- Learn about the Fisher's Exact Test for a **2 x 2** contingency table
 - Hypothesis statement
 - Display a Contingency table (review this tutorial)
 - Display a Mosaic plot (review this tutorial)
 - Conduct the Fisher's Exact test
 - Concluding statement
- Learn about the χ^2 Contingency Test on a **m x n** contingency table
 - Hypothesis statement
 - Display a Contingency table (review this tutorial)
 - Display a Mosaic plot (review this tutorial)
 - Check assumptions

- Get results of the test
- Concluding statement

12.1 Load packages and import data

Load the `tidyverse`, `skimr`, `naniar`, `knitr`, `ggbasicplots`, and `janitor` packages:

```
library(tidyverse)
library(skimr)
library(naniar)
library(knitr)
library(ggbasicplots)
library(janitor)
```

We'll also need a new package called `epitools`, so install that now if you haven't done so.

```
library(epitools)

##
## Attaching package: 'epitools'

## The following objects are masked from 'package:binom':
##      binom.exact, binom.wilson
```

We'll use two datasets described in the Whitlock & Schluter text:

- the “cancer.csv” dataset (described in Example 9.2 in the text, page 238)
- the “worm.csv” dataset (described in Example 9.4 in the text, page 246)

```
cancer <- read_csv("https://raw.githubusercontent.com/ubco-biology/BIOL202/main/data/cancer.csv")

## Rows: 39876 Columns: 2
## -- Column specification -----
## Delimiter: ","
## chr (2): aspirinTreatment, response
##
## 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.
```

```
worm <- read_csv("https://raw.githubusercontent.com/ubco-biology/BIOL202/main/data/worm.csv")  
  
## Rows: 141 Columns: 2  
## -- Column specification -----  
## Delimiter: ","  
## chr (2): infection, fate  
##  
## 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.
```

Take a look at the cancer dataset:

```
cancer %>%  
  skim_without_charts()
```

(#tab:cont_datalook_cancer)Data summary

Name

Piped data

Number of rows

39876

Number of columns

2

Column type frequency:

character

2

Group variables

None

Variable type: character

skim_variable

n_missing

complete_rate

min

max

```
empty  
n_unique  
whitespace  
aspirinTreatment  
0  
1  
7  
7  
0  
2  
0
```

response

```
0  
1  
6  
9  
0  
2  
0
```

And the worm dataset:

```
worm %>%  
  skim_without_charts()
```

(#tab:cont_datalook_worm) Data summary

Name

Piped data

Number of rows

141

Number of columns

2

Column type frequency:

character

2

Group variables

None

Variable type: character

skim_variable

n_missing

complete_rate

min

max

empty

n_unique

whitespace

infection

0

1

6

10

0

3

0

fate

0

1

5

9

0

2

0

Both datasets are formatted “tidy” format. For a refresher on this, review the Biology Procedures and Guidelines document chapter on **Tidy data**.

12.2 Fisher's Exact Test

When testing for an association between two categorical variables, the most common test that is used is the χ^2 contingency test, which is described in the next section.

When the two categorical variables *have exactly 2 categories each*, and thus yield a 2 x 2 contingency table, the **Fisher's Exact test** (a type of contingency test) provides an EXACT *P*-value, and is therefore preferred over the χ^2 contingency test (below) when you have a computer to do the calculations.

Often, and especially when the 2 x 2 contingency table deals with a health-related study, one refers to the **Odds Ratio**, which we'll learn about below.

In any case, the most powerful statistical test for a 2 x 2 contingency analysis is a **Fisher's Exact test**.

12.2.1 Hypothesis statement

We'll use the cancer study data again for this example, as described in example 9.2 (Page 235) in the text.

The hypotheses for this test:

H₀: There is no association between the use of aspirin and the probability of developing cancer.

H_A: There is an association between the use of aspirin and the probability of developing cancer.

- We'll use an α level of 0.05.
- It is a two-tailed alternative hypothesis
- We'll use a Fisher's Exact test to test the null hypothesis, because this is the **most powerful test** when analyzing a 2 x 2 contingency table.
- There is no test statistic for the Fisher's Exact test, and nor does it use "degrees of freedom" (the latter you'll learn about soon, and is only required when we use a theoretical distribution for a test statistic)

HOWEVER: it is recommended that you report the "odds ratio" (which you'll learn about below) in your concluding statement, along with its appropriate confidence interval; this is a useful stand-in test statistic for the Fisher's Exact Test

- We also don't need to worry about assumptions for this test, because it is not relying on a theoretical probability distribution

- It is always a good idea to present a figure to accompany your analysis; in the case of a Fisher's Exact test, the figure heading will include information about the sample size / total number of observations, whereas the concluding statement typically does not

12.2.2 Display a contingency table

We'll use the approach we learned in an earlier tutorial to construct a contingency table.

We'll store the table in an object called "cancer.aspirin.table", and we'll make sure to include margin (row and column) totals:

```
cancer.aspirin.table <- cancer %>%
  tabyl(response, aspirinTreatment) %>%
  adorn_totals(where = c("row", "col"))
```

Let's have a look at the result:

```
cancer.aspirin.table
```

```
##   response Aspirin Placebo Total
##   Cancer     1438    1427  2865
##   No cancer  18496   18515 37011
##   Total      19934   19942 39876
```

When dealing with data from studies on human health (e.g. evaluating healthy versus sick subjects), it is convention to organize the contingency table as shown above, with (i) the outcome of interest (here, cancer) in the top row and the alternative outcome on the bottom row, and (ii) the treatment in the first column and placebo (control group) in the second column. When the data are not related to health outcomes, you do not need to worry about the ordering of the rows of data.

Let's use the `kable` function to display a nice looking contingency table:

```
cancer.aspirin.table %>%
  kable(caption = "Contingency table showing the incidence of cancer in relation to experimental treatments")
```

(#tab:cont_cancer_aspirin_table)Contingency table showing the incidence of cancer in relation to experimental treatments

| | Aspirin | Placebo | Total |
|-----------|---------|---------|-------|
| Cancer | 1438 | 1427 | 2865 |
| No cancer | 18496 | 18515 | 37011 |
| Total | 19934 | 19942 | 39876 |

| | |
|-----------|--|
| Placebo | |
| Total | |
| Cancer | |
| 1438 | |
| 1427 | |
| 2865 | |
| No cancer | |
| 18496 | |
| 18515 | |
| 37011 | |
| Total | |
| 19934 | |
| 19942 | |
| 39876 | |

12.2.3 Display a mosaic plot

Let's visualize the data using a mosaic plot, taking note of the frequency of observations falling in each category (from the contingency table produced previously).

Here we'll add a bit of new code to make the mosaic plot more ideally formatted. When we first learned how to create a mosaic plot, we saw that the y-axis was lacking tick-marks and numbers. We'll remedy that here, using the `scale_y_continuous`, which allows us to specify what breaks (ticks) we want on the y-axis.

Here we're showing "relative frequency" on the y-axis, so this should range from 0 to 1. And we'll add breaks at intervals of 0.2. Specifically, we use the base `seq` function to generate a sequence of numbers from 0 to 1, in intervals of 0.2:

```
cancer %>%
  ggplot() +
  geom_mosaic(aes(x = product(aspirinTreatment), fill = response)) +
  scale_y_continuous(breaks = seq(0, 1, by = 0.2)) +
  xlab("Treatment group") +
  ylab("Relative frequency") +
  theme_bw()
```

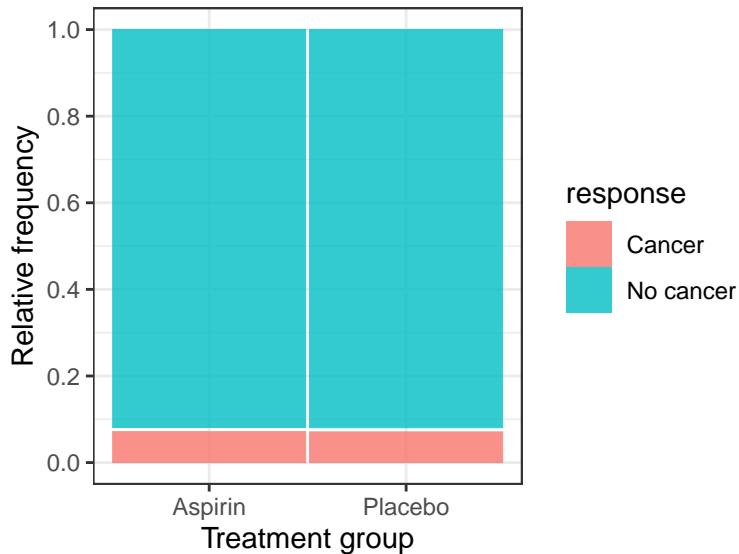


Figure 12.1: Relative frequency of cancer among women randomly assigned to control ($n = 19942$) and aspirin ($n = 19934$) treatment groups.

The mosaic plot shows that the incidence (or relative frequency) of cancer is almost identical in the treatment and control groups.

IMPORTANT It is best practice to display the response variable as the “fill” variable in a mosaic plot.

12.2.4 Conduct the Fisher's Exact Test

To do the Fisher's exact test on the cancer data, it is straightforward, using the `fisher.test` function from the `janitor` package.

There is also `fisher.test` function in the base R `stats` package, but it does not conform to `tidyverse` expectations. Hence our use of the `fisher.test` function from the `janitor` package. When there are multiple packages that use the same name for a function, we can specify the version we want by prefacing the function with the package name and two colons, like this: “`janitor::fisher.test()`”

See the help file for the `janitor` version of the `fisher.test` function:

```
?janitor::fisher.test
```

This function requires a two-way “tabyl” as the input, and we already know how to construct such a table.

We'll put the results in an object called "cancer.fishertest":

```
cancer.fishertest <- cancer %>%
  tabyl(aspirinTreatment, response) %>%
  janitor::fisher.test()
```

Let's look at the results:

```
cancer.fishertest
```

```
## 
## Fisher's Exact Test for Count Data
##
## data: .
## p-value = 0.8311
## alternative hypothesis: true odds ratio is not equal to 1
## 95 percent confidence interval:
## 0.9342128 1.0892376
## sample estimates:
## odds ratio
## 1.008744
```

The P -value associated with the test is 0.831, which is clearly greater than our α of 0.05. We therefore FAIL to reject the null hypothesis.

You'll notice that the output includes the **odds ratio** and its 95% confidence interval. The interval it provides is slightly different from the one we'll learn about below, but **when reporting the results of a Fisher's Exact test it is OK to report the confidence interval provided by the `fisher.test` function**. It is also OK to provide the slightly different one that we learn about below.

Concluding statement

There is no evidence that the probability of developing cancer differs between the control group and the aspirin treatment group (Fisher's Exact Test; P -value = 0.831; odds ratio = 1.01; 95% CI: 0.934 - 1.089).

TIP Report odds ratios to 2 decimal places, and associated measures of uncertainty to 3 decimal places

12.3 Estimate the Odds of getting sick

The **odds** of success (O) are the probability of success (p) divided by the probability of failure ($1-p$):

$$O = \frac{p}{1-p}$$

Curiously, in health-related studies, a “success” is equated with getting ill!!

We’ll use the data stored in the contingency table we produced before, called “cancer.aspirin.table”:

```
cancer.aspirin.table <- cancer %>%
  tabyl(aspirinTreatment, response) %>%
  adorn_totals(where = c("row", "col"))
cancer.aspirin.table

##   aspirinTreatment Cancer No_cancer Total
##       Aspirin        1438     18496 19934
##       Placebo        1427     18515 19942
##           Total       2865     37011 39876
```

And recall that proportions are calculated using frequencies - which is exactly what we have in the table!

Thus, to estimate the “odds” of getting cancer while taking aspirin, we need to:

- first calculate the proportion (= probability) of women who got cancer while taking aspirin (= p)
- then calculate the proportion (= probability) of women who remained healthy while taking aspirin (= $1-p$)
- then calculate the odds as $O = \frac{p}{1-p}$

We’ll do all of this in one go using a series of steps strung together with pipes (“%>%”).

Here’s the code, and we’ll explain each step after:

```
cancer.aspirin.table %>%
  filter(aspirinTreatment == "Aspirin") %>%
  select(Cancer, Total) %>%
  mutate(
    propCancer_aspirin = Cancer / Total,
    propHealthy_aspirin = 1 - propCancer_aspirin,
    oddsCancer_aspirin = propCancer_aspirin/propHealthy_aspirin
  )
```

```
##   Cancer Total propCancer_aspirin propHealthy_aspirin oddsCancer_aspirin
##   1438 19934          0.07213806          0.9278619        0.07774654
```

- we first **filter** the table to return only the rows pertaining to the “Aspirin” treatment group; the frequencies that we need for the calculations are in this row
- then we **select** the columns from that row with names “Cancer” and “Total”, which include the frequency of women who got cancer while on Aspirin (under the “Cancer” column), and the total frequency of women in the Aspiring treatment group (in the “Total”)
- we then use the **mutate** function to create three new variables:
 - “propCancer_aspirin” is calculated at the Cancer frequency divided by the Total frequency (within the Aspirin group)
 - “propHealth_aspirin” is calculated simply as 1 minus prop-Cancer_aspirin
 - “oddsCancer_aspirin” is calculated last as “propCancer_aspirin/propHealthy_aspirin”

Thus, the odds of getting cancer while on aspirin are about 0.08:1, or equivalently, approximately 1:13 (which you get from dividing 0.0777 into 1).

Alternatively, “the odds are 13 to 1 that a women who took aspirin would **not** get cancer in the next 10 years”.

1. Estimate odds

- Estimate the odds that a woman in the placebo group would get cancer
-

12.4 Estimate the odds ratio

We'll use the **oddsratio** function from the **epitools** package to calculate the odds ratio (\hat{OR}) and its 95% confidence interval.

Check out the help file for the function:

```
?oddsratio
```

The **oddsratio** function expects the contingency table to be arranged exactly like this:

```
#           treatment control
# sick          a          b
# healthy       c          d
```

If you were calculating the odds ratio by hand, using the letters shown in the table above, the shortcut formula is:

$$\hat{OR} = \frac{a/c}{b/d}$$

Here's the code for producing the appropriately formatted 2 x 2 table as so:

```
cancer %>%
  tabyl(aspirinTreatment, response) %>%
  select(Cancer, "No cancer")

##   Cancer No cancer
##   1438     18496
##   1427     18515
```

So that's what the `oddsratio` function is expecting as input.

However, it's also expecting it in the form of a "matrix" object.

Here's all the code at once, and we'll store the output (which comes in the form of a "list") in an object called "cancer.odds":

```
cancer.odds <- cancer %>%
  tabyl(aspirinTreatment, response) %>%
  select(Cancer, "No cancer") %>%
  as.matrix() %>%
  oddsratio(method = "wald")
```

We've seen the first two lines before. Then:

- we `select` the two columns associated with the "Cancer" and "No cancer" data. **NOTE** that because there's a space in the variable name "No cancer", we need to use quotation marks around it
- Then we use the base `as.matrix` function to coerce the resulting 2 x 2 table that we've created into a matrix type object, which is what the `oddsratio` function is expecting.
- lastly we run the `oddsratio` function, with the argument "method = 'wald'" (don't worry about why)

Let's have a look at the rather verbose output:

```
cancer.odds
```

```
## $data
##      Cancer No_cancer Total
## row1     1438     18496 19934
## row2     1427     18515 19942
## Total    2865     37011 39876
##
## $measure
##                NA
## odds_ratio with 95% C.I. estimate    lower    upper
##                  [1,] 1.000000      NA      NA
##                  [2,] 1.008744 0.9349043 1.088415
##
## $p.value
##            NA
## two-sided midp.exact fisher.exact chi.square
##      [1,]      NA      NA      NA
##      [2,] 0.8224348 0.8310911 0.8223986
##
## $correction
## [1] FALSE
##
## attr(),"method")
## [1] "Unconditional MLE & normal approximation (Wald) CI"
```

This is more information than we need.

What we're interested in is the information under the “\$measure” part, and specifically the “odds ratio with 95% C.I.”.

To limit the output to the relevant information, use this code:

```
cancer.odds$measure[2,]
```

```
## estimate    lower    upper
## 1.0087436 0.9349043 1.0884148
```

This isolates the actual estimate of the odds ratio (\hat{OR}) with its 95% confidence interval.

The estimate of the odds ratio is around 1.009, and notice the 95% confidence interval encompasses one.

Given that the calculated 95% confidence interval encompasses 1 (representing equal odds among treatment and control groups), there is presently no evidence

that the odds of developing cancer differ among control and aspirin treatment groups.

IMPORTANT The odds ratio and its 95% confidence interval are useful to report in any analysis of a 2 x 2 contingency table that deals with health outcomes data like those used here.

12.5 χ^2 Contingency Test

When the contingency table is of dimensions greater than 2 x 2, the most commonly applied test is the χ^2 Contingency Test.

For this activity we're using the "worm" data associated with **Example 9.4 on page 244** of the test. Please read the example!

12.5.1 Hypothesis statement

As shown in the text example, we have a 2 x 3 contingency table, and we're testing for an association between two categorical variables.

Here are the null and alternative hypotheses (compare these to what's written in the text):

H₀: There is no association between the level of trematode parasitism and the frequency (or probability) of being eaten.

H_A: There is an association between the level of trematode parasitism and the frequency (or probability) of being eaten.

- We use $\alpha = 0.05$.
- It is a two-tailed alternative hypothesis
- We'll use a contingency test to test the null hypothesis, because this is appropriate for analyzing for association between two categorical variables, and when the resulting contingency table has dimension greater than 2 x 2.
- We will use the χ^2 test statistic, with degrees of freedom equal to $(r-1)(c-1)$, where "r" is the number of rows, and "c" is the number of columns, so $(2-1)(3-1) = 2$.
- We must check assumptions of the χ^2 contingency test
- It is always a good idea to present a figure to accompany your analysis; in the case of a contingency test, the figure heading will include information about the sample size / total number of observations

12.5.2 Display the contingency table

Let's generate a contingency table:

```
worm %>%
  tabyl(fate, infection) %>%
  adorn_totals(where = c("row", "col"))

##      fate highly lightly uninfected Total
##   eaten      37      10       1    48
## not eaten     9      35      49    93
##   Total      46      45      50   141
```

Hmm, the ordering of the categories of the categorical (factor) variable “infection” is the opposite to what is displayed in the text.

TIP The ordering of the categories is not actually crucial to this type of analysis, but it's certainly better practice to show them in appropriate order!

In order to change the order of categories in a character variable, here's what you do (consult this resource for additional info).

First, convert the character variable to a “factor” variable as follows:

```
worm$infection <- as.factor(worm$infection)
```

Now check the default ordering of the levels using the `levels` function (it should be alphabetical):

```
levels(worm$infection)

## [1] "highly"     "lightly"     "uninfected"
```

Now change the ordering as follows:

```
worm$infection <- factor(worm$infection, levels = c("uninfected", "lightly", "highly"))
levels(worm$infection)

## [1] "uninfected" "lightly"     "highly"
```

Now re-display the contingency table, first storing it in an object “worm.table”:

```
worm.table <- worm %>%
  tabyl(fate, infection) %>%
  adorn_totals(where = c("row", "col"))

##      fate uninfected lightly highly Total
##   eaten          1       10      37     48
## not eaten       49       35       9     93
##   Total         50       45      46    141
```

That's better!

Visualize it nicely with `kable`:

```
kable(worm.table)
```

| fate | uninfected | lightly | highly | Total |
|-----------|------------|---------|--------|-------|
| eaten | 1 | 10 | 37 | 48 |
| not eaten | 49 | 35 | 9 | 93 |
| Total | 50 | 45 | 46 | 141 |

12.5.3 Visualize a mosaic plot

Here's a mosaic plot with an ideal figure caption included:

```
worm %>%
  ggplot() +
  geom_mosaic(aes(x = product(infection), fill = fate)) +
  scale_fill_manual(values=c("darkred", "gold")) +
  scale_y_continuous(breaks = seq(0, 1, by = 0.2)) +
  xlab("Level of infection") +
  ylab("Relative frequency") +
  theme_bw()
```

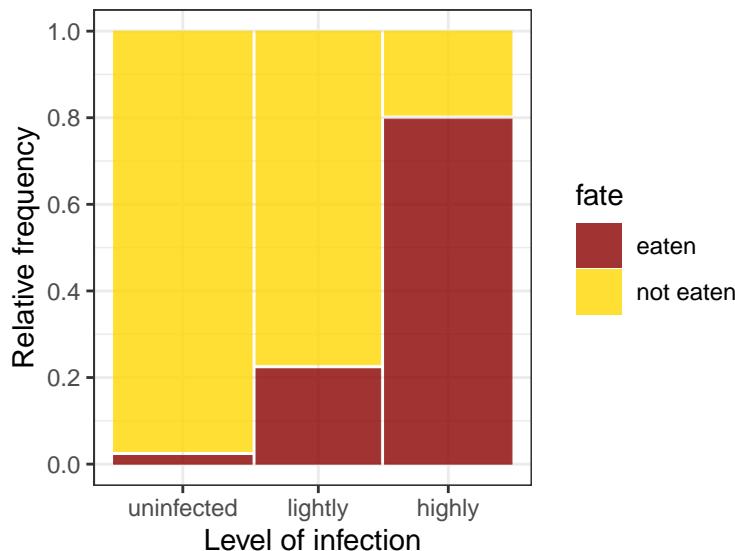


Figure 12.2: Mosaic plot of bird predation on killifish having different levels of trematode parasitism. A total of 50, 45, and 46 fish were in the uninfected, lightly infected, and highly infected groups.

In the code above we manually specified the two fill colours using the `scale_fill_manual` function.

12.5.4 Check the assumptions

The χ^2 contingency test (also known as association test) has assumptions that **must be checked** prior to proceeding:

- none of the categories should have an expected frequency of less than one

- no more than 20% of the categories should have expected frequencies less than five

To test these assumptions, we need to actually conduct the test, because in doing so R calculates the **expected frequencies** for us.

Conduct the test using the `chisq.test` function from `janitor` package. **NOTE** this again overlaps with a function name from the base R package, so we'll need to specify that we want the “janitor” version of the function.

We need our contingency table as input to the function, but this time without margin totals.

We'll assign the results to an object “`worm.chisq.results`”:

```
worm.chisq.results <- worm %>%
  tabyl(fate, infection) %>%
  janitor::chisq.test()
```

Have a look at the output:

```
worm.chisq.results
```

```
## 
## Pearson's Chi-squared test
## 
## data: .
## X-squared = 69.756, df = 2, p-value = 7.124e-16
```

Although only a few bits of information are provided here, the object actually contains a lot more information.

Don't try getting an overview of the object using our usual `skim_without_charts` approach.. that won't work.

Instead, simply use this code:

```
names(worm.chisq.results)
```

```
## [1] "statistic" "parameter" "p.value"    "method"      "data.name"   "observed"
## [7] "expected"  "residuals"  "stdres"
```

As you can see, one of the names is `expected`. This is what holds our expected frequencies (and note that these values do not need to be whole numbers, unlike the observed frequencies):

```
kable(worm.chisq.results$expected)
```

| | fate |
|------------|----------|
| uninfected | 17.02128 |
| lightly | 15.31915 |
| highly | 15.65957 |
| eaten | 32.97872 |
| eaten | 29.68085 |
| | 30.34043 |
| not eaten | 32.97872 |
| not eaten | 29.68085 |
| | 30.34043 |

We see that all our assumptions are met: none of the cells (cross-classified categories) in the table have an expected frequency of less than one, and no more than 20% of the cells have expected frequencies less than five.

12.5.5 Get the results of the test

We can see the results of the χ^2 test by simply typing the name of the results object:

```
worm.chisq.results
```

```
##  
## Pearson's Chi-squared test  
##  
## data: .  
## X-squared = 69.756, df = 2, p-value = 7.124e-16
```

This shows a very large value of χ^2 (69.76) and a very small P -value - much smaller than our stated α . So we reject the null hypothesis.

Concluding statement

The probability of being eaten is significantly associated with the level of trematode parasitism (χ^2 contingency test; $df = 2$; $\chi^2 = 69.76$; $P < 0.001$). Based on our mosaic plot (Fig. X), the probability of being eaten increases substantially with increasing intensity of parasitism.

Chapter 13

Analyzing a single numerical variable

Tutorial learning objectives

- Learn about the one-sample t -test, which compares the mean from a sample of individuals to a value for the population mean (μ_0) proposed in the null hypothesis
 - Learn the assumptions of a one-sample t -test
 - Learn how to calculate an exact confidence interval for a population mean (μ)
 - Learn the confidence interval approach for testing the plausibility of a hypothesized value for μ
-

13.1 Load packages and import data

Load the `tidyverse`, `knitr`, `naniar`, and `janitor` packages:

```
library(tidyverse)
library(knitr)
library(naniar)
library(janitor)
```

We'll use the following datasets:

- the “bodytemp” dataset. These are the data associated with Example 11.3 in the text (page 310)
- the “stalkies” dataset. These are the data associated with Example 11.2 in the text (page 307)

```
bodytemp <- read_csv("https://raw.githubusercontent.com/ubco-biology/BIOL202/main/data/
stalkies <- read_csv("https://raw.githubusercontent.com/ubco-biology/BIOL202/main/data/
```

Reminder: Before proceeding further, remember to get an overview of each of these datasets.

13.2 One-sample t -test

We previously learned statistical tests for testing hypotheses about categorical response variables. For instance, we learned how to conduct a χ^2 contingency test to test the null hypothesis that there is no association between two categorical variables.

Here we are going to learn our first statistical test for testing hypotheses about a numeric response variable, specifically one whose probability distribution in the population is normally distributed.

13.2.1 Hypothesis statement

Review the earlier tutorial that lists the steps to hypothesis testing.

We’ll use the body temperature data for this example, as described in example 11.3 in the text.

Americans are taught as kids that the normal human body temperature is 98.6 degrees Farenheit.

Are the data consistent with this assertion?

- We’ll use a one-sample t -test test to test the null hypothesis, because we’re dealing with a single numerical response variable, and we’re using a sample of individuals to draw inferences about a hypothesized (population) mean μ_0

The hypotheses for this test:

H₀: The mean human body temperature is 98.6°F ($\mu_0 = 98.6^\circ\text{F}$).

H_A: The mean human body temperature is not 98.6°F ($\mu_0 \neq 98.6^\circ\text{F}$).

TIP You can add a degree symbol using this syntax in markdown: `$^\circ\text{C}$`, so degrees Celsius would be `$^\circ\text{C}`

- We'll use an α level of 0.05.
 - It is a two-tailed alternative hypothesis
-

13.2.2 Assumptions of one-sample *t*-test

The **assumptions** of the one-sample *t*-test are as follows:

- the sampling units are randomly sampled from the population (a standard assumption)
- the variable is normally distributed in the population

Consult the “checking assumptions” tutorial for how to formally check the second assumption. For now we’ll assume both assumptions are met.

If the normal distribution assumption is not met, and no data transformation helps, then one can conduct a “non-parametric” test in lieu of the one-sample *t*-test, including a “Sign test” or a “Wilcoxon signed-rank test”. A tutorial on non-parametric tests is under development, but will not be deployed until 2023. Consult chapter 13 in the Whitlock & Schluter text, and this website for some R examples.

13.2.3 A graph to accompany a one-sample *t*-test

Let’s create a histogram of the body temperatures, because this is the most appropriate way to visualize the frequency distribution of a single numeric variable.

Histograms may look a little wonky if you have small sample sizes. This is OK!

If you forget how to create a histogram, consult the previous tutorial on visualizing a single numeric variable.

Here we’ll learn a few more tricks for producing a high-quality histogram.

The first step is to figure out the minimum and maximum values of your numeric variable, and you should have already done this when using the `skim_without_charts` function to get an overview of the dataset (when you first imported the data).

For the `temperature` variable, our minimum and maximum values were 97.4 and 100, respectively.

We'll use this information to ensure that the histogram spans the appropriate range along the x-axis, and to decide on the appropriate “bin widths”:

```
bodytemp %>%
  ggplot(aes(x = temperature)) +
  geom_histogram(binwidth = 0.5, boundary = 97,
                 color = "black", fill = "lightgrey",) +
  xlab("Body temperature (degrees F)") +
  ylab("Frequency") +
  theme_bw()
```

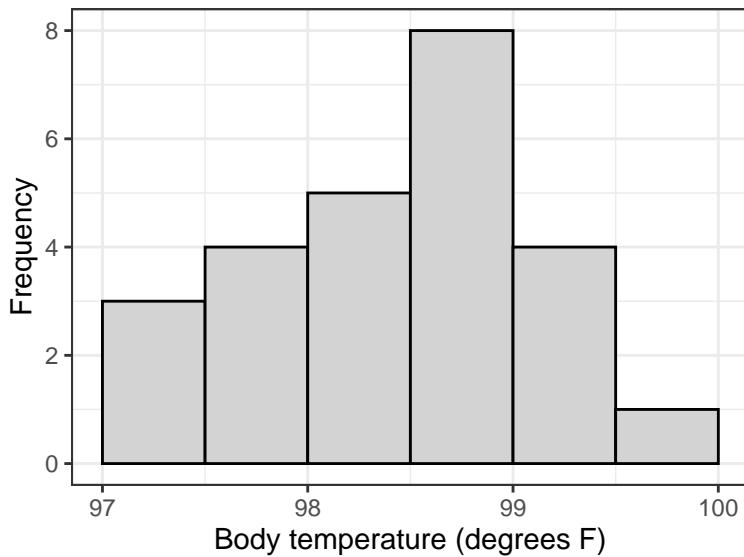


Figure 13.1: Frequency distribution of body temperature (degrees Farenheit) for 25 randomly chosen healthy people.

Notice the new argument to the `geom_histogram` function: “boundary”. Why include this? The minimum value in our dataset is 97.4. Based on our overview of the data, we decided to use a “binwidth” of 0.5. Therefore, it made most sense to have our bins (bars) start at 97, the first whole number preceding our minimum value, then have breaks every 0.5 units thereafter. Constructing the histogram this way makes it easier to interpret.

Recall that it can take a few times playing with different values of “binwidth” to make the histogram have the right number of bins.

Interpreting the histogram:

We can see in the histogram above that most of the individuals had temperatures between 98 and 99°F, which is consistent with conventional wisdom, but there are 7 people with temperature below 98°F, and 5 with temperatures above 99°F. The frequency distribution is unimodal but not especially symmetrical.

13.2.4 Conduct the one-sample *t*-test

We use the `t.test` function (it comes with the `base` package loaded with R) to conduct a one-sample *t*-test.

```
?t.test
```

This function is used for both one-sample and two-sample *t*-tests (covered later), and for calculating 95% confidence intervals for a mean (later in this tutorial).

Because this function has multiple purposes, be sure to pay attention to the arguments.

Here's the code, which we'll explain after:

```
body.ttest <- bodytemp %>%
  select(temperature) %>%
  t.test(mu = 98.6, alternative = "two.sided", conf.level = 0.95)
```

- We first assign our results to a new object called “body.ttest”
- We then `select` the variable “temperature”, as this is the one being analyzed
- We then conduct the test using the function `t.test`, specifying the null hypothesized value as “mu = 98.6”
- We specify “two.sided” for the “alternative” argument
- We provide the “conf.level” of 0.95, which is equal to $1 - \alpha$

Let's look at the results:

```
body.ttest
```

```
##  
## One Sample t-test  
##  
## data: .  
## t = -0.56065, df = 24, p-value = 0.5802  
## alternative hypothesis: true mean is not equal to 98.6  
## 95 percent confidence interval:  
## 98.24422 98.80378  
## sample estimates:  
## mean of x  
## 98.524
```

The output includes:

- The calculated test statistic t
- The degrees of freedom
- The P -value for the test
- A 95% confidence interval for μ
- The sample-based estimate, denoted “mean of x”, but is our (\bar{Y})

The observed P -value for our test is larger than our α level of 0.05. We therefore fail to reject the null hypothesis.

13.2.5 Concluding statement for the one-sample t -test

Here's an example of an appropriately worded concluding statement, including all the relevant information:

We have no reason to reject the null hypothesis that the mean body temperature of a healthy human is 98.6°F (one-sample t -test; $t = -0.56$; $n = 25$ or $df = 24$; $P = 0.58$; 95% confidence interval: $98.244 < \mu < 98.804$).

TIP The confidence interval provided by the `t.test` function is accurate and good to report with your concluding statement for a one-sample t -test. However, below we learn a different way to calculate the confidence interval.

13.3 Confidence intervals for μ

In an earlier tutorial we learned about the **rule of thumb 95% confidence interval**.

Now we will learn how to calculate confidence intervals more precisely.

There are two typical uses of confidence intervals:

- When we are *estimating* a population parameter (such as μ) based on a random sample, in which case the confidence interval is an ideal measure of precision to accompany our estimate
- As an alternative to a formal hypothesis test, in which case we determine whether the hypothesized value of the population parameter (e.g. μ_0) is a plausible value for μ , based on the confidence interval calculated using our sample data

We'll explore the latter use of confidence intervals later in this tutorial. For now, let's demonstrate the first use of confidence intervals: as a measure of precision for an estimate.

13.3.1 Confidence interval as a measure of precision for an estimate

A straightforward way to calculate the confidence interval for the mean of a numeric variable is using the `t.test` function. For an example, see this tutorial section.

The formula for a 95% confidence interval for μ is:

$$\bar{Y} - t_{0.05(2),df}SE_{\bar{Y}} < \mu < \bar{Y} + t_{0.05(2),df}SE_{\bar{Y}}$$

The $t_{0.05(2),df}$ represents the critical value of t for a two-tailed test with $\alpha = 0.05$, and *degrees of freedom* (df), which is calculated from our sample size as $df = n - 1$.

$SE_{\bar{Y}}$ is the familiar standard error of the mean, calculated as:

$$SE_{\bar{Y}} = \frac{s}{\sqrt{n}}$$

The lower 95 confidence limit is the value to the left of the μ in the equation above, and the upper 95% confidence limit is the value to the right of the μ in the equation.

So now we need to figure out the critical value $t_{0.05(2), df}$.

For the body temperature example, we have $n = 25$ and thus $df = 25 - 1 = 24$.

Optionally, we could assign all these values to objects first, for use later:

```
alpha <- 0.05
n <- 25
d_f = n - 1
```

The function we use to find the critical value of t is the base function `qt`:

```
?qt
```

The `qt` function only deals with one tail of the distribution. Thus, if we have a two-sided alternative hypothesis, we need to divide our α level by two in order to calculate the appropriate critical value of t .

The following graph illustrates this for the t distribution associated with $df = 24$ and $\alpha(2) = 0.05$. Specifically, the t_{crit} values are shown by the vertical lines, and delimit the points beyond which the area under the curve towards the tail is equal to $\alpha/2$ (and thus the total area of both red zones= α).

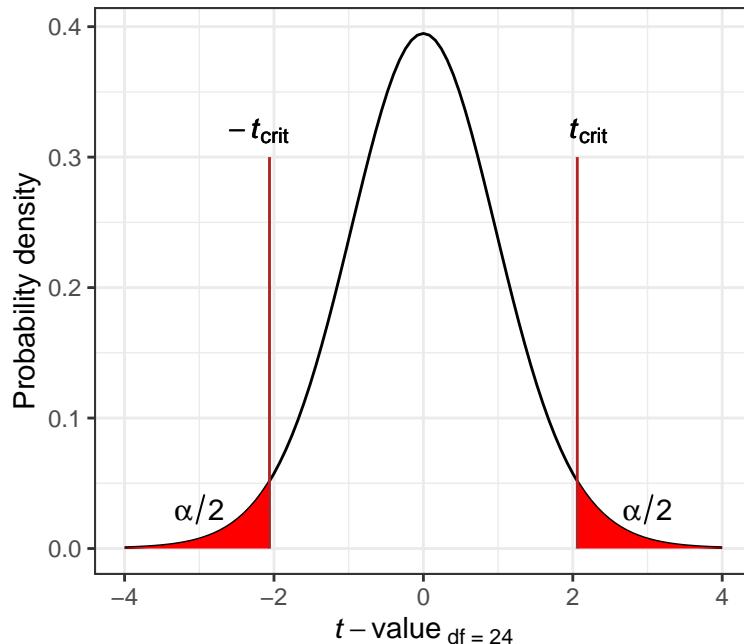


Figure 13.2: The t distribution for $df = 24$.

Here's the code to calculate the critical value of t , and recall that we've defined `alpha` and `d_f` above:

```
tcrit <- qt(p = alpha/2, df = d_f, lower.tail = FALSE)
```

In the code above:

- We assign the result to a new object called `tcrit`
- The $\alpha(2) = 0.05$ is **divided by 2**, because we have a two-sided alternative hypothesis
- The degrees of freedom (which we stored in an object called `d_f`)
- We specified `lower.tail = FALSE` to tell R that we'd like to focus on calculating the critical value for the right-hand (upper) tail, and thus the positive critical value

Alternatively, we could simply provide the values of `alpha` and `degrees of freedom` directly to the function arguments:

```
tcrit <- qt(0.05/2, df = 24, lower.tail = FALSE)
```

TIP If instead we wanted to calculate a 99% confidence interval, then $\alpha(2) = 0.01$, and we'd use 0.01 instead of 0.05 in the `qt` function above. Try it out!

Now let's see what the output is:

```
tcrit
```

```
## [1] 2.063899
```

So to recap, this procedure used R code to give us what we'd otherwise need to look up in a statistical table, like the one provided in the text book.

Now that we've determined the value for $t_{0.05(2),24}$, we need to calculate $SE_{\bar{Y}}$, which we already learned how to do. So let's improve on what we learned in the estimation tutorial, where we learned how to calculate the rule-of-thumb confidence interval.

There, we generated a table (actually a “tibble”) of summary statistics that included the rule-of-thumb confidence limits; now we can include the precisely calculated lower and upper 95% confidence limits.

Recall that we've already created an object “`tcrit`” that holds our required critical value of t .

Here we'll assign the output to a tibble called “`bodytemp.stats`”:

```
bodytemp.stats <- bodytemp %>%
  summarise(
    Count = n() - na.omit(n_miss(temperature)),
    Mean_temp = mean(temperature, na.rm = TRUE),
    SD_temp = sd(temperature, na.rm = TRUE),
    SEM = SD_temp/sqrt(Count),
    Lower_95_CL = Mean_temp - tcrit * SEM,
    Upper_95_CL = Mean_temp + tcrit * SEM
  )
```

The `summarise` function is used to create new summary variables, as we've learned before.

The new part here is that we're calculating precisely the lower and upper 95% confidence limits, using the combination of the `tcrit` value that we calculated and the "SEM" (standard error).

Now let's use the `kable` function to produce a nice table, and here we'll use "digits = 3" because that's appropriate for the confidence limits (noting that it will report three decimal places for all our calculated values):

```
kable(bodytemp.stats, digits = 3)
```

| |
|-------------|
| Count |
| Mean_temp |
| SD_temp |
| SEM |
| Lower_95_CL |
| Upper_95_CL |
| 25 |
| 98.524 |
| 0.678 |
| 0.136 |
| 98.244 |
| 98.804 |

If we wanted to report the confidence interval on its own, we can get the necessary information from our newly created tibble "bodytemp.stats", and type the following inline code in our markdown text (NOT in a code chunk):

Which will provide:

$98.244 < \mu < 98.804$

```
`r round(bodytemp.stats
```

Figure 13.3: Example of inline markdown and R code for a confidence interval

13.3.2 Confidence interval approach to hypothesis testing

Recall our null and alternative hypotheses for the body temperature example:

H_0 : The mean human body temperature is 98.6°F ($\mu_0 = 98.6^\circ\text{F}$).

H_A : The mean human body temperature is not 98.6°F ($\mu_0 \neq 98.6^\circ\text{F}$).

The null hypothesis proposed $\mu_0 = 98.6^\circ\text{F}$.

The confidence interval approach to testing a null hypothesis involves:

- specifying a level of confidence, $100\% \times (1-\alpha)$, such as 95%
- calculating the confidence interval for μ using the sample data
- determining whether or not μ_0 lies within the calculated confidence interval

If it does, then the proposed value of μ_0 is plausible, and there's no reason to reject the null hypothesis.

If it does not, then we reject the null hypothesis and conclude that plausible values of μ are between our lower and upper confidence limits.

For example, for the body temperature example, we calculated a 95% confidence interval as:

$$98.244 < \mu < 98.804$$

The hypothesized value of μ_0 was 98.6°F, which is encompassed by our confidence interval. Thus, it is a plausible value, and there is no reason to reject the null hypothesis.

1. Practice confidence intervals

- First, using the `bodytemp` dataset, calculate the 99% confidence interval for the mean body temperature in the population.

- Then, using the “stalkies” dataset, use the confidence interval approach (95% confidence) to test the hypothesis that the average eye span in the population is $\mu_0 = 8.1\text{mm}$
-

Chapter 14

Comparing means among two groups

Tutorial learning objectives

- Learn about paired and two-sample study designs for comparing the mean of a numeric response variable among two categories of a categorical variable (i.e. two groups)
- Learn about the paired t -test for testing a hypothesis about the difference between means in a paired design (d)
- Learn about the two-sample t -test for testing a hypothesis about the difference between two population means ($\mu_1 - \mu_2$)
- Learn the assumptions of the paired and two-sample t -tests
- Learn how to calculate an exact confidence interval for the difference between two means
- Learn the confidence interval approach for testing the plausibility of a hypothesized value for $\mu_1 - \mu_2$

14.1 Load packages and import data

Load the `tidyverse`, `knitr`, `naniar`, `car`, `skimr`, and `janitor` packages:

```
library(tidyverse)
library(knitr)
library(naniar)
library(janitor)
library(skimr)
library(car)
```

And we also need the `broom` package, which may be new to you.

```
library(broom)
```

The following datasets are required:

- the “blackbird” dataset. These are the data associated with Example 12.2 in the text (page 330)
- the “students” dataset, describing characteristics of students from BIOL202 from several years back

```
blackbird <- read_csv("https://raw.githubusercontent.com/ubco-biology/BIOL202/main/data/blackbird.csv")
students <- read_csv("https://raw.githubusercontent.com/ubco-biology/BIOL202/main/data/students.csv")
```

14.2 Paired *t*-test

We’ll use the `blackbird` dataset for this example.

For 13 red-winged blackbirds, measurements of antibodies were taken before and after implantation with testosterone. Thus, the same bird was measured twice. Clearly, these measurements are not independent, hence the need for a “paired” *t*-test.

Let’s first have a look at the `blackbird` dataset:

```
blackbird
```

```
## # A tibble: 26 x 3
##   blackbird time   Antibody
##       <dbl> <chr>    <dbl>
## 1 1     Before 4.654
## 2 2     Before 3.912
## 3 3     Before 4.913
## 4 4     Before 4.500
## 5 5     Before 4.804
## 6 6     Before 4.883
## 7 7     Before 4.875
## 8 8     Before 4.779
## 9 9     Before 4.977
## 10 10    Before 4.868
## # i 16 more rows
```

The data frame has 26 rows, and includes 3 variables, the first of which “blackbird” simply keeps track of the individual ID of blackbirds.

The response variable of interest, “Antibody” represents antibody production rate measured in units of natural logarithm (\ln) 10^{-3} optical density per minute ($\ln[mOD/min]$).

The factor variable `time` that has two levels: “After” and “Before”.

These data are stored in **tidy format**, which, as you’ve learned, is the ideal format for storing data.

Sometimes you may get data in **wide format**, in which case, for instance, we would have a column for the “Before” antibody measurements and another column for the “After” measurements.

It is always preferable to work with long-format (tidy) data.

Consult the following webpage for instructions on using the `tidyverse` package for converting between wide and long data formats.

With our data in the preferred long format, we can proceed with our hypothesis test, but because the hypothesis focuses on the *differences* in the paired measurements, we need to calculate those first!

14.2.1 Calculate differences

Let’s remind ourselves how the data are stored:

```
blackbird
```

```
## # A tibble: 26 x 3
##   blackbird time   Antibody
##   <dbl> <chr>    <dbl>
## 1 1     Before  4.654
## 2 2     Before  3.912
## 3 3     Before  4.913
## 4 4     Before  4.500
## 5 5     Before  4.804
## 6 6     Before  4.883
## 7 7     Before  4.875
## 8 8     Before  4.779
## 9 9     Before  4.977
## 10 10    Before  4.868
## # i 16 more rows
```

We’ll use, for the first time, the `pivot_wider` function from the `dplyr` package (which is loaded as part of the `tidyverse`).

The `pivot_wider` function essentially takes data stored in long format and converts it to wide format.

The following code requires that there be at least one variable in the tibble that provides a unique identifier for each individual. In the “blackbird” tibble, this variable is “blackbird”. I have added an argument “`id_cols = blackbird`” to the code below to underscore the need for this type of identifier variable. The code will not work if such a variable does not exist in the tibble.

Here’s the code, then we’ll explain after:

```
blackbird.diffs <- blackbird %>%
  pivot_wider(id_cols = blackbird, names_from = time, values_from = Antibody) %>%
  mutate(diffs = After - Before)
```

In the preceding chunk, we:

- create a new object “`blackbird.diffs`” to hold our data
- the `pivot_wider` function takes the following arguments:
 - “`id_cols = blackbird`”, which tells the function which variable in the tibble is used to keep track of the unique individuals (here, the “blackbird” variable)
 - A categorical (grouping) variable “`names_from`” and creates new columns, one for each unique category
 - A “`values_from`” variable; thus, in our case, we get 2 new columns (because there are 2 categories to the “time” variable: Before and After), and the values placed in those columns are the corresponding values of “Antibody”.
- we then create a new variable “`diff`” that equals the values in the newly created “After” variable minus the values in the “Before” variable.

TIP: In the blackbird example we have “Before” and “After” measurements of a variable, and we calculated the *difference* as “*After – Before*”, as this is a logical way to do it. It doesn’t really matter which direction you calculate the difference, but just be aware that you need to make clear how it was calculated, so that your interpretation is correct.

Let’s have a look at the result:

```
blackbird.diffs
```

```
## # A tibble: 13 x 4
##   blackbird Before After    diffs
##       <dbl>  <dbl> <dbl>    <dbl>
## 1        1    4.654 4.443 -0.2113
## 2        2    3.912 4.304  0.3920
## 3        3    4.913 4.977  0.06408
```

```
## 4      4 4.500 4.454 -0.04546
## 5      5 4.804 4.997  0.1932
## 6      6 4.883 4.997  0.1144
## 7      7 4.875 5.011  0.1354
## 8      8 4.779 4.956  0.1767
## 9      9 4.977 5.017  0.04055
## 10    10 4.868 4.727 -0.1401
## 11    11 4.754 4.771  0.01709
## 12    12 4.700 4.595 -0.1054
## 13    13 4.927 5.011  0.08338
```

We can see that some of the `diffs` values are negative, and some are positive. These would of course be switched in sign if we had calculated the differences as “*Before – After*”.

In any case, this is the new tibble and variable “`diffs`” that we’ll use for our hypothesis test!

14.2.2 Hypothesis statement

The hypotheses for this paired *t*-test focus on the mean of the *differences* between the paired measurements, denoted by μ_d :

H_0 : The mean change in antibody production after testosterone implants was zero ($\mu_d = 0$).

H_A : The mean change in antibody production after testosterone implants was not zero ($\mu_d \neq 0$).

Steps to a hypothesis test:

- We’ll use an α level of 0.05.
- It is a two-tailed alternative hypothesis
- We’ll visualize the data, and interpret the output
- We’ll use a paired *t*-test test to test the null hypothesis, because we’re dealing with “before and after” measurements taken on the same individuals, and drawing inferences about a population mean μ_d using sample data
- We’ll check the assumptions of the test (see below)
- We’ll calculate our test statistic
- We’ll calculate the *P*-value associated with our test statistic
- We’ll calculate a 95% confidence interval for the mean difference
- We’ll provide a good concluding statement that includes a 95% confidence interval for the mean difference

14.2.3 A graph to accompany a paired t -test

The best way to visualize the data for a paired t -test is to create a **histogram** of the calculated *differences* between the paired observations.

```
blackbird.diffs %>%
  ggplot(aes(x = diffs)) +
  geom_histogram(binwidth = 0.1, boundary = -0.3,
                 color = "black", fill = "lightgrey",) +
  xlab("Difference in antibody production rate (after - before) (ln[mOD/min]) 10^-3") +
  ylab("Frequency") +
  theme_bw()
```

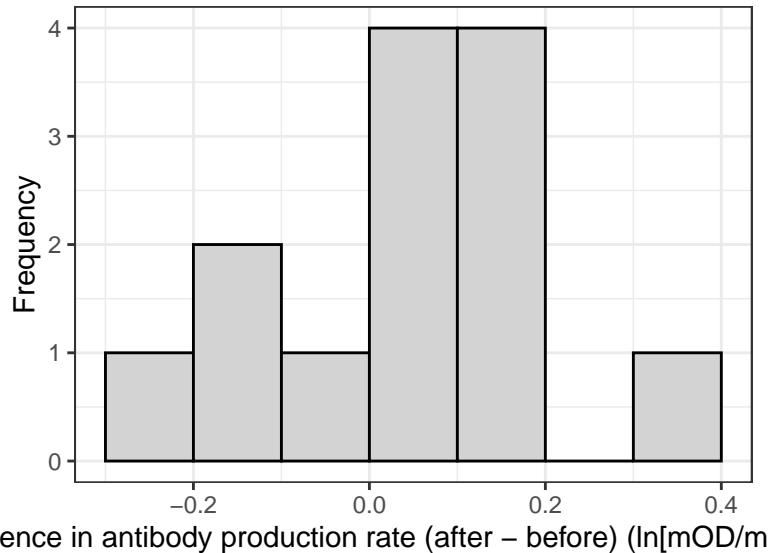


Figure 14.1: Histogram of the differences in antibody production rate before and after the testosterone treatment

With such a small sample size (13), the histogram is not particularly informative. But we do see most observations are just above zero.

OPTIONAL

Another optional but nice way to visualize paired data is using a paired plot.

```
blackbird %>%
  ggplot(aes(x = time, y = Antibody)) +
  geom_point(shape = 1, size = 1.5) +
```

```
geom_line(aes(group = blackbird), color = "grey") +
theme_bw()
```

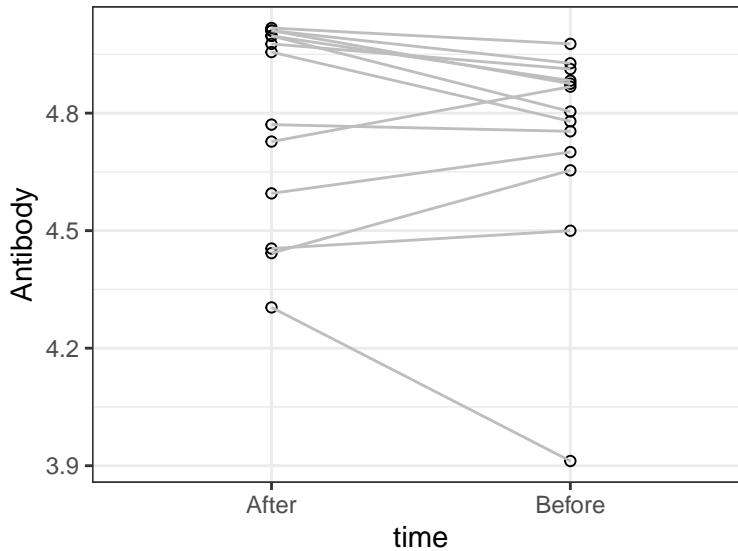


Figure 14.2: Paired plot of antibody production rate before and after the testosterone treatment

OPTIONAL

Notice that the “After” group is plotted on the left, which is a bit counter-intuitive. We could optionally change that by changing how R recognizes the “order” of the “time” variable:

```
blackbird$time <- ordered(blackbird$time, levels = c("Before", "After"))
```

Then repeat the code above to create the paired plot.

14.2.4 Assumptions of the paired t -test

(Updated October 31, 2024)

The assumptions of the paired t -test are the same as the assumptions for the one-sample t -test, except they pertain to the *difference*:

- the sampling units are randomly sampled from the population

- the *mean difference* has a normal distribution in the population (each group of measurements need not be normally distributed)

As instructed in the checking assumptions tutorial, we should use a normal quantile plot to visually check the normal distribution assumption, using the calculated differences.

```
blackbird.diffs %>%
  ggplot(aes(sample = diffs)) +
  stat_qq(shape = 1, size = 2) +
  stat_qq_line() +
  xlab("Normal quantile") +
  ylab("Antibody production (ln[mOD/min]) 10^-3") +
  theme_bw()
```

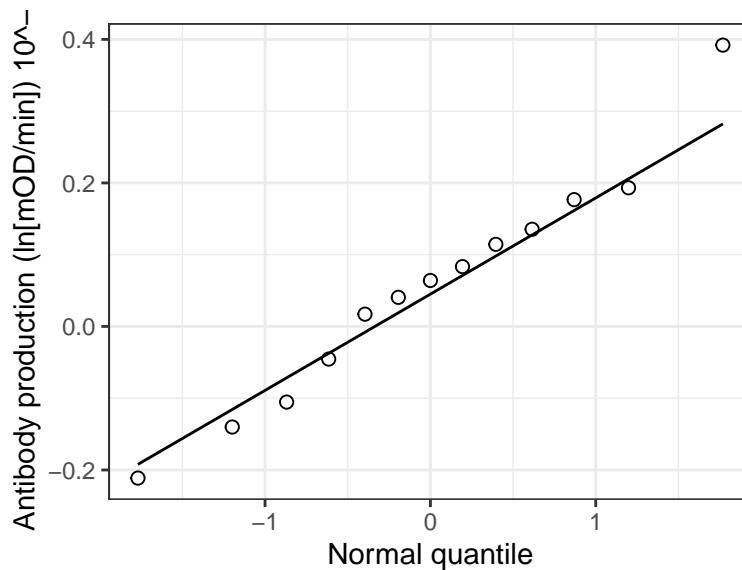


Figure 14.3: Normal quantile plot of the differences in antibody production rate before and after the testosterone treatment ($\ln[\text{mOD}/\text{min}] 10^{-3}$).

We see that most of the lines are close to the line, with one point near the top right that is a bit off...

A reasonable statement would be:

“The normal quantile plot shows that the data generally fall close to the line (except perhaps the highest value), indicating that the normality assumption is reasonably met.”

But if you're feeling uncertain, we can follow this with a **Shapiro-Wilk Normality Test**, which tests the null hypothesis that the data are sampled from a normal distribution.

```
shapiro.result <- shapiro.test(blackbird.diffs$diffs)
shapiro.result.tidy <- tidy(shapiro.result)
shapiro.result.tidy

## # A tibble: 1 x 3
##   statistic p.value method
##       <dbl>    <dbl> <chr>
## 1      0.9781  0.9688 Shapiro-Wilk normality test
```

Given that the P -value is large (and much greater than 0.05), there is no reason to reject the null hypothesis. Thus, our normality assumption is met.

When testing the normality assumption using the Shapiro-Wilk test, there is no need to conduct all the steps associated with a hypothesis test. Simply report the results of the test (the test statistic value and the associated P -value).

For instance:

“A Shapiro-Wilk test revealed no evidence against the assumption that the data are drawn from a normal distribution ($W = 0.98$, P -value = 0.969).”

14.2.5 Conduct the test

We can conduct a **paired t-test** in two different ways:

- conduct a one-sample t -test on the *differences* using the `t.test` function and methods you learned in a previous tutorial.
- conduct a paired t -test using the `t.test` function and the argument `paired = TRUE`.

(1) One-sample t -test on the differences

Let's proceed with the test as we've previously learned.

Here we make sure to set the null hypothesized value of “mu” to zero in the argument for the `t.test` function:

```
blackbird.ttest <- blackbird.diffs %>%
  select(diffs) %>%
  t.test(mu = 0, alternative = "two.sided", conf.level = 0.95)
```

Now have a look at the result:

```
blackbird.ttest
```

```
##  
##  One Sample t-test  
##  
## data: .  
## t = 1.2435, df = 12, p-value = 0.2374  
## alternative hypothesis: true mean is not equal to 0  
## 95 percent confidence interval:  
## -0.04134676 0.15128638  
## sample estimates:  
## mean of x  
## 0.05496981
```

The observed P -value for our test is larger than our α level of 0.05. We therefore fail to reject the null hypothesis.

The values of t and of the lower and upper confidence limits may be reversed in sign, if you conducted your calculation of differences in the alternative way. Specifically, you may get $t = -1.2434925$, and confidence limits of -0.1512864 and 0.0413468. This is totally fine!

(2) Paired t -test

(Updated October 31, 2024)

Here again we'll use the wide-format tibble `blackbird.diffs`, using this approach:

```
blackbird.paired.ttest <- t.test(x = blackbird.diffs$Before, y = blackbird.diffs$After  
                                 paired = TRUE, alternative = 'two.sided', conf.level = 0.95)
```

Here's an explanation:

- we create a new object “`blackbird.paired.ttest`” to store our results in
- we run the `t.test` function with the arguments as follows:
 - we specify `x` equal to the “Before” variable, and `y` equal to the “After” variable
 - we have the “`paired = TRUE`” argument, telling the function that this is a paired design
 - we specify that this is a two-sided test with “`alternative = 'two.sided'`
 - finally we use “`conf.level = 0.95`” which corresponds to an $\alpha = 0.05$

Let's look at the result:

```
blackbird.paired.ttest
```

```
##  
## Paired t-test  
##  
## data: blackbird.diffs$Before and blackbird.diffs$After  
## t = -1.2435, df = 12, p-value = 0.2374  
## alternative hypothesis: true mean difference is not equal to 0  
## 95 percent confidence interval:  
## -0.15128638 0.04134676  
## sample estimates:  
## mean difference  
## -0.05496981
```

The output is identical to what we got when we applied a 1-sample *t*-test on the differences!

The values of *t* and of the lower and upper confidence limits may be reversed in sign, if you conducted your calculation of differences in the alternative way. Specifically, you may get *t* = -1.2434925, and confidence limits of -0.1512864 and 0.0413468. This is totally fine!

14.2.6 Concluding statement

Here's an example of a reasonable concluding statement, and this can apply for either of the two methods used above (note that in either case we call the test a "paired *t*-test, even if we used the one-sample *t*-test on the differences):

We have no reason to reject the null hypothesis that the mean change in antibody production after testosterone implants was zero (paired *t*-test; *t* = -1.24; *df* = 12; *P* = 0.237; 95% confidence interval for the difference: $-0.151 < \mu_d < 0.041$).

14.3 Two sample *t*-test

Have a look at the `students` dataset:

```
students %>%  
  skim_without_charts()
```

Data summary

Name

Piped data

Number of rows

154

Number of columns

6

Column type frequency:

character

3

numeric

3

Group variables

None

Variable type: character

skim_variable

n_missing

complete_rate

min

max

empty

n_unique

whitespace

Dominant_hand

0

1

1

0

2

0

Dominant_foot

0
1
1
1
0
2
0

Dominant_eye

0
1
1
1
0
2
0

Variable type: numeric

skim_variable

n_missing

complete_rate

mean

sd

p0

p25

p50

p75

p100

height_cm

0

1

171.97

10.03
150
165.00
171.48
180.0
210.8
head_circum_cm
0
1
56.04
6.41
21
55.52
57.00
58.5
63.0
Number_of_siblings
0
1
1.71
1.05
0
1.00
2.00
2.0
6.0

These data include measurements taken on 154 students in BIOL202 a few years ago.

We'll use the "height" and "Dominant_eye" variables for this section.

OPTIONAL

Note that the categories in the Dominant_eye variable are "l" and "r", denoting "left" and "right".

We can use the `unique` function to tell us all unique values of a categorical variable:

```
students %>%
  select(Dominant_eye) %>%
  unique()

## # A tibble: 2 x 1
##   Dominant_eye
##   <chr>
## 1 r
## 2 l
```

Let's change these to be more informative, "Left" and "Right".

We can do this with the `recode_factor` function from the `dplyr` package (part of the `tidyverse`), in conjunction with the familiar `mutate` function used to create a new variable (though here we're just over-writing an existing variable):

```
students <- students %>%
  mutate(Dominant_eye = recode_factor(Dominant_eye, r = "Right", l = "Left"))
```

14.3.1 Hypothesis statement

H_0 : Mean height is the same among students with left dominant eyes and right dominant eyes ($\mu_L = \mu_R$).

H_A : Mean height is not the same among students with left dominant eyes and right dominant eyes ($\mu_L \neq \mu_R$).

Steps to a hypothesis test:

- We'll use an α level of 0.05.
- It is a two-tailed alternative hypothesis
- We'll provide a table of descriptive statistics for each group
- We'll visualize the data, and interpret the output
- We'll use a 2-sample t -test to test the null hypothesis, because we're dealing with numerical measurements taken on independent within two independent groups, and drawing inferences about population means μ using sample data
- We'll check the assumptions of the test (see below)
- We'll calculate our test statistic
- We'll calculate the P -value associated with our test statistic

- We'll calculate a 95% confidence interval for the difference ($\mu_L - \mu_R$)
- We'll provide a good concluding statement that includes a 95% confidence interval for the mean difference ($\mu_L - \mu_R$)

14.3.2 A table of descriptive statistics

When we are analyzing a numeric response variable in relation to a categorical variable with two or more categories, it's good practice to provide a table of summary (or "descriptive") statistics (including confidence intervals for the mean) for the numeric variable grouped by the categories.

In a previous tutorial we learned how to calculate descriptive statistics for a numeric variable grouped by a categorical variable. In another tutorial we also learned how to calculate confidence intervals for a numeric variable.

We've also learned that the `t.test` function returns a confidence interval for us.

Let's use all these skills to generate a table of summary statistics for the "height_cm" variable, grouped by "Dominant_eye":

```
height.stats <- students %>%
  group_by(Dominant_eye) %>%
  summarise(
    Count = n() - naniar::n_miss(height_cm),
    Count_NA = naniar::n_miss(height_cm),
    Mean = mean(height_cm, na.rm = TRUE),
    SD = sd(height_cm, na.rm = TRUE),
    SEM = SD/sqrt(Count),
    Low_95_CL = t.test(height_cm, conf.level = 0.95)$conf.int[1],
    Up_95_CL = t.test(height_cm, conf.level = 0.95)$conf.int[2]
  )
```

The only unfamiliar code in the preceding chunk is the last two lines:

- we use the `t.test` function to calculate the lower and upper confidence limits. Specifically:
 - after the closing parenthesis to the `t.test` function, we include "`$conf.int[1]`", and this simply extracts the first value (lower limit) of the calculated confidence limits from the `t.test` output
 - we do the same for the upper confidence limit, but this time we include "`$conf.int[2]`"

Now let's have a look at the table, using the `kable` function to produce a nice table.

NOTE here I am rotating the table so that it fits on the page. To do this, use the `t` function as follows:

```
kable(t(height.stats), digits = 4)
```

Dominant_eye

Right

Left

Count

106

48

Count_NA

0

0

Mean

172.8368

170.0598

SD

10.397638

8.964659

SEM

1.009908

1.293937

Low_95_CL

170.8343

167.4567

Up_95_CL

174.8393

172.6629

It is best to **NOT rotate the table**, but it is fine to do so if your table goes off the page!

We'll learn a better way to get around this later.

14.3.3 A graph to accompany a 2-sample t -test

We learned in an earlier tutorial that we can use a stripchart, violin plot, or boxplot to visualize the association between a numerical response variable and a categorical explanatory variable. Better yet, we can do the combined violin & boxplot.

Here we want to visualize height in relation to dominant eye (Left or Right). We can use the information provided in our descriptive stats table to get the sample sizes for the groups (which we need to report in the figure heading).

```
students %>%
  ggplot(aes(x = Dominant_eye, y = height_cm)) +
  geom_violin() +
  geom_boxplot(width = 0.1) +
  geom_jitter(colour = "grey", size = 1, shape = 1, width = 0.15) +
  xlab("Dominant eye") +
  ylab("Height (cm)") +
  theme_bw()
```

Interpretation

We can see in the preceding figure that heights are generally similar between students with left dominant eyes and those with right dominant eyes. However, it appears that the spread of the heights is greater among students with right dominant eyes. We will need to be careful about the equal-variance assumption for the 2-sample t -test.

14.3.4 Assumptions of the 2-sample t -test

The assumptions of the 2-sample t -test are as follows:

- each of the two samples is a random sample from its population
- the numerical variable is normally distributed in each population
- the variance (and thus standard deviation) of the numerical variable is the same in both populations

Test for normality

Now let's check the normality assumption by plotting a normal quantile plot for each group.

We'll introduce the `facet_grid` function that enables plotting of side-by-side panels according to a grouping variable.

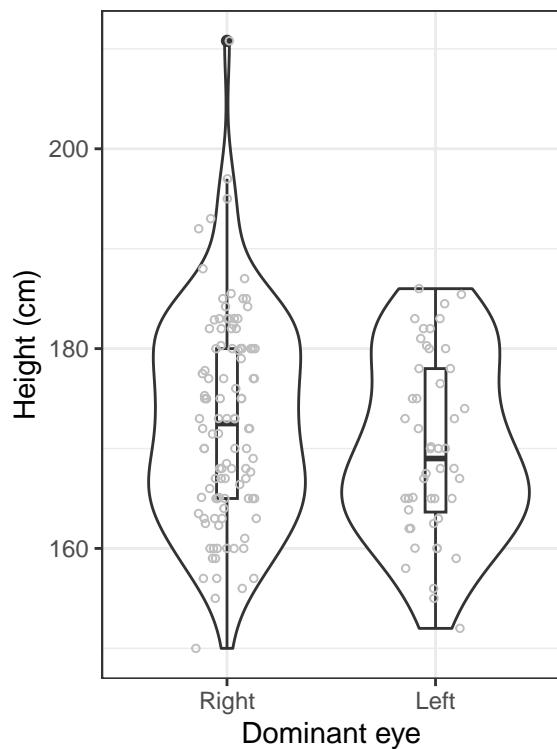


Figure 14.4: Violin and boxplot of the heights of students with right ($n = 106$) and left ($n = 48$) dominant eyes. Boxes delimit the first to third quartiles, bold lines represent the group medians, and whiskers extend to 1.5 times the IQR. Points beyond whiskers are extreme observations.

```
students %>%
  ggplot(aes(sample = height_cm)) +
  stat_qq(shape = 1, size = 2) +
  stat_qq_line() +
  facet_grid(~ Dominant_eye) +
  xlab("Normal quantile") +
  ylab("Height (cm)") +
  theme_bw()
```

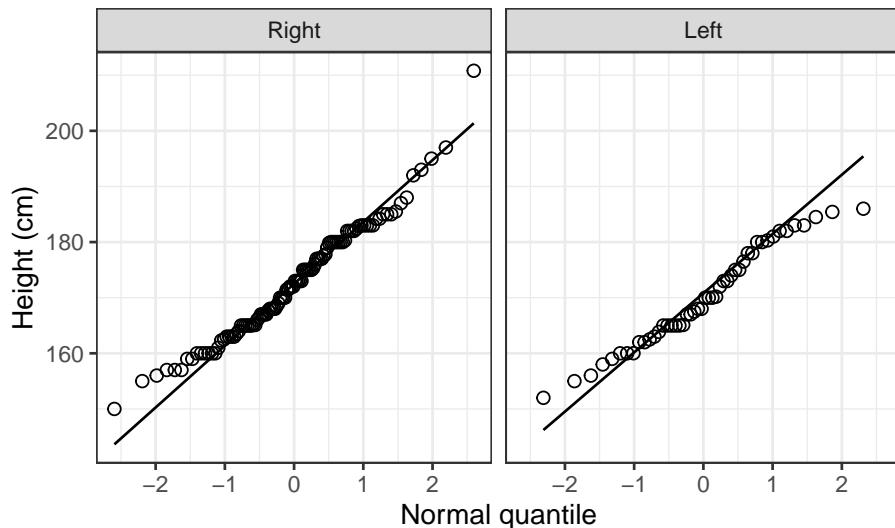


Figure 14.5: Normal quantile plots of height for students with right ($n = 106$) or left ($n = 48$) dominant eyes.

A reasonable statement would be:

“The normal quantile plots show that student height is generally normally distributed for students with left or right dominant eyes. There is one observation among the right-dominant eye students that is a bit off the line.”

Test for equal variances

Now we need to test the assumption of equal variance among the groups, using Levene’s Test as we learned in the checking assumptions tutorial.

```
height.vartest <- leveneTest(height_cm ~ Dominant_eye, data = students)
height.vartest
```

```
## Levene's Test for Homogeneity of Variance (center = median)
##          Df F value Pr(>F)
## group     1  0.907 0.3424
##          152
```

It uses a test statistic “F”, and we see here that the P -value associated with the test statistic is larger than 0.05, so we don’t reject the implied null hypothesis that the variances are equal.

We state “A Levene’s test showed no evidence against the assumption of equal variance ($F = 0.91$; P -value = 0.342).”

Thus, we’ll proceed with conducting the 2-sample t -test.

14.3.5 Conduct the 2-sample t -test

We use the `t.test` function again for this test.

```
height.ttest <- students %>%
  t.test(height_cm ~ Dominant_eye,
        data = ., var.equal = TRUE, conf.level = 0.95)
```

The only difference from the implementation used in the paired t -test is:

- we include the “`var.equal = TRUE`” argument

We again include the argument “`data = .`”, which tells the `t.test` function that whatever data was passed to it from the preceding line is what will be used

We’ll learn a bit later what to do when the equal variance assumption is not met.

Let’s look at the result:

```
height.ttest

##
## Two Sample t-test
##
## data: height_cm by Dominant_eye
## t = 1.6, df = 152, p-value = 0.1117
## alternative hypothesis: true difference in means between group Right and group Left is not equal to zero
## 95 percent confidence interval:
## -0.6521529 6.2061545
## sample estimates:
## mean in group Right   mean in group Left
##           172.8368          170.0598
```

We see that the test produced a P -value greater than α , so we fail to reject the null hypothesis.

Note also that the output includes a **confidence interval** for the *difference* in group means. We need to include this in our concluding statement.

14.3.6 Concluding statement

Here's an example of a concluding statement:

On average, students with left dominant eyes are similar in height to students with right dominant eyes (Figure 19.4) (2-sample t -test; $t = 1.6$; $df = 152$; P -value = 0.112; 95% confidence interval for the difference in height $-0.652 < \mu_d < 6.206$).

14.4 When assumptions aren't met

If the **normal distribution assumption** is violated, and you are unable to find a transformation that works (see the Checking assumptions and data transformations tutorial), then you can try a non-parametric test.

A tutorial on non-parametric tests is forthcoming, but not available yet. Consult chapter 13 in the Whitlock & Schluter text, and this website for some R examples.

If the **equal-variance assumption** is violated for the 2-sample t -test, then you can set the "var.equal" argument to "FALSE" in the `t.test` function, in which case the function implements a "Welch's t -test".

For instance, let's pretend that the height data did not exhibit equal variance among left- and right- dominant eye students, here's the appropriate code:

```
height.ttest.unequal.var <- students %>%
  t.test(height_cm ~ Dominant_eye,
        data = ., var.equal = FALSE, conf.level = 0.95)
height.ttest.unequal.var

##
## Welch Two Sample t-test
##
## data: height_cm by Dominant_eye
## t = 1.6919, df = 104.37, p-value = 0.09366
## alternative hypothesis: true difference in means between group Right and group Left
## 95 percent confidence interval:
```

```
## -0.4778181 6.0318197
## sample estimates:
## mean in group Right mean in group Left
## 172.8368 170.0598
```


Chapter 15

Checking assumptions and data transformations

Tutorial learning objectives

- Learn how to check the normality assumption
 - Normal quantile plots
 - Shapiro-Wilk test for normality
- Learn how to check the equal variance assumption
 - Levene's Test
- Learn how to transform the response variable to help meet assumptions
 - log-transform
 - Dealing with zeroes
 - log bases
 - back-transforming log data
 - logit transform
 - back-transforming logit data
 - when to back-transform?

Most statistical tests, such as the χ^2 goodness of fit test, the χ^2 contingency test, t -test, ANOVA, Pearson correlation, and least-squares regression, have assumptions that must be met. For example, the one-sample t -test requires that the variable is normally distributed in the population, and least-squares regression requires that the residuals from the regression be normally distributed. In this tutorial we'll learn ways to check the assumption that the variable is normally distributed in the population.

We'll also learn how transforming a variable can sometimes help satisfy assumptions, in which case the analysis is conducted on the transformed variable.

15.1 Load packages and import data

Load the usual packages, and `broom`, which has been used in some tutorials:

```
library(tidyverse)
library(skimr)
library(broom)
```

And we need these two packages also: `car`, `boot`. Install these if you don't have them (as per instructions in a previous tutorial), then load them:

```
library(car)
library(boot)
```

The “marine.csv” dataset is discussed in example 13.1 in the text book. The “flowers.csv” dataset is described below. The “students.csv” data include data about BIOL202 students from a few years ago.

Let's make sure to treat any categorical variables as factor variables, using the “stringsAsFactors = T” argument:

```
marine <- read_csv("https://raw.githubusercontent.com/ubco-biology/BIOL202/main/data/marine.csv",
  stringsAsFactors = TRUE)

## # A tibble: 32 x 1
##   biomassRatio
##       <dbl>
## 1      0.000
## 2      0.000
## 3      0.000
## 4      0.000
## 5      0.000
## 6      0.000
## 7      0.000
## 8      0.000
## 9      0.000
## 10     0.000
## # ... with 22 more rows, and 1 more variable:
## #   .rowid. <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32
```



```
flowers <- read_csv("https://raw.githubusercontent.com/ubco-biology/BIOL202/main/data/flowers.csv",
  stringsAsFactors = TRUE)

## # A tibble: 30 x 1
##   propFertile
##       <dbl>
## 1      0.000
## 2      0.000
## 3      0.000
## 4      0.000
## 5      0.000
## 6      0.000
## 7      0.000
## 8      0.000
## 9      0.000
## 10     0.000
## # ... with 20 more rows, and 1 more variable:
## #   .rowid. <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30
```

```
students <- read_csv("https://raw.githubusercontent.com/ubco-biology/BIOL202/main/data/students.csv")  
  
## Rows: 154 Columns: 6  
## -- Column specification -----  
## Delimiter: ","  
## chr (3): Dominant_hand, Dominant_foot, Dominant_eye  
## dbl (3): height_cm, head_circum_cm, Number_of_siblings  
##  
## 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.
```

Explore the `marine` and `flowers` datasets:

```
marine %>%  
  skim_without_charts()
```

(#tab:skim_marine) Data summary

Name

Piped data

Number of rows

32

Number of columns

1

Column type frequency:

numeric

1

Group variables

None

Variable type: numeric

skim_variable

n_missing

complete_rate

mean

244CHAPTER 15. CHECKING ASSUMPTIONS AND DATA TRANSFORMATIONS

```
sd  
p0  
p25  
p50  
p75  
p100  
biomassRatio  
0  
1  
1.73  
0.75  
0.83  
1.27  
1.49  
1.85  
4.25
```

```
flowers %>%  
  skim_without_charts()
```

(#tab:skim_flowers)Data summary

Name
Piped data
Number of rows
30
Number of columns
1

Column type frequency:

numeric
1

Group variables

None

Variable type: numeric

skim_variable

n_missing

complete_rate

mean

sd

p0

p25

p50

p75

p100

propFertile

0

1

0.47

0.33

0.01

0.13

0.49

0.75

0.99

15.2 Checking the normality assumption

Statistical tests such as the one-sample t test assume that the response variable of interest is normally distributed in the population.

Many biological variables are known to be normally distributed in the population, but for some variables we can't be sure. Given a proper random sample from the population, of sufficient sample size, we can assume that the frequency distribution of our sample data will, to reasonable degree, reflect the frequency distribution of the variable in the population.

Importantly, tests such as the one-sample t test are somewhat **robust** to minor violations of this assumption. Nevertheless, it is best practice to be transparent in testing the assumption, i.e. showing how it was tested and exactly what was found.

15.2.1 Normal quantile plots

The most straightforward way to check the normality assumption is to visualize the data using a **normal quantile plot**.

The `ggplot2` package (loaded with the `tidyverse` package) has plotting functions for this, called `stat_qq` and `stat_qq_line`:

```
?stat_qq
?stat_qq_line
```

For details about what Normal Quantile Plots are, and how they're constructed, consult this informative link.

If the frequency distribution were normally distributed, points would fall close to the straight line in the normal quantile plot.

Check out this example showing simulated data drawn from a normal distribution:

Now we'll use the `marine` dataset and its variable called `biomassRatio` to illustrate.

We'll first construct a histogram as you've learned previously, just to see how the shape of the frequency distribution relates to the pattern seen in the normal quantile plot.

```
marine %>%
  ggplot(aes(x = biomassRatio)) +
  geom_histogram(binwidth = 0.5, color = "black", fill = "lightgrey",
                 boundary = 0, closed = "left") +
  xlab("Biomass ratio") +
  ylab("Frequency") +
  theme_bw()
```

Notice that the distribution is quite right-skewed (or “positively skewed”).

Now the quantile plot:

```
marine %>%
  ggplot(aes(sample = biomassRatio)) +
  stat_qq(shape = 1, size = 2) +
```

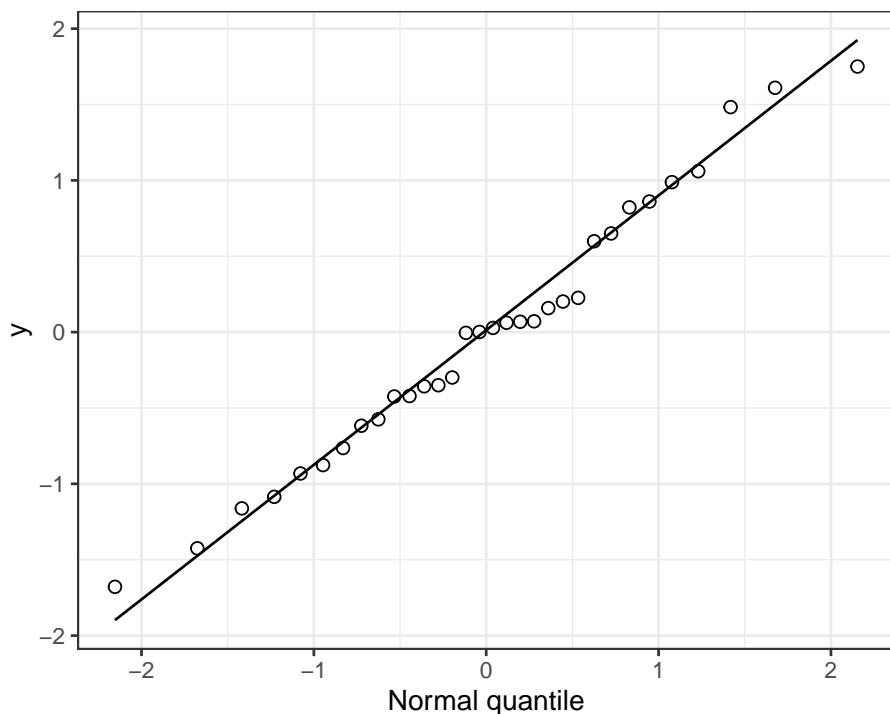


Figure 15.1: Example of a normal quantile plot for a variable that is normally distributed.

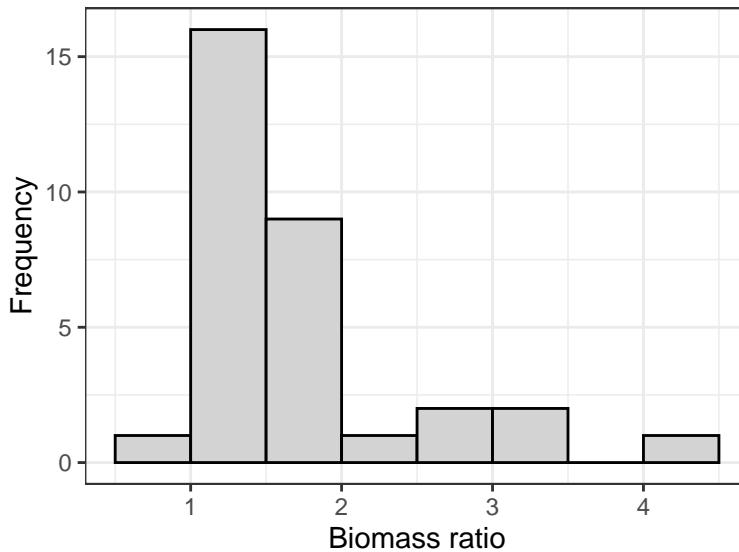


Figure 15.2: The frequency distribution of the ‘biomass ratio’ of 32 marine reserves.

```
stat_qq_line() +
ylab("Biomass ratio") +
xlab("Normal quantile") +
theme_bw()
```

Notice that in the “aes” argument we use “sample = biomassRatio”. This is new, and is only required for the normal quantile plot, specifically the subsequent `stat_qq` and `stat_qq_line` functions.

Notice that normal quantile plot shows points deviating substantially from the straight line in the top-right part of the plot, and this corresponds to the right-skew in the histogram.

Clearly, the frequency distribution of the `biomassRatio` variable does not conform to a normal distribution.

Here’s an example statement one could make when checking this assumption:

The assumption of normality was checked visually using a normal quantile plot, which showed that the data were clearly not normally distributed.

Important: Egregious deviations from the normality assumption will be clearly evident in normal quantile plots (as in the example above). If it is

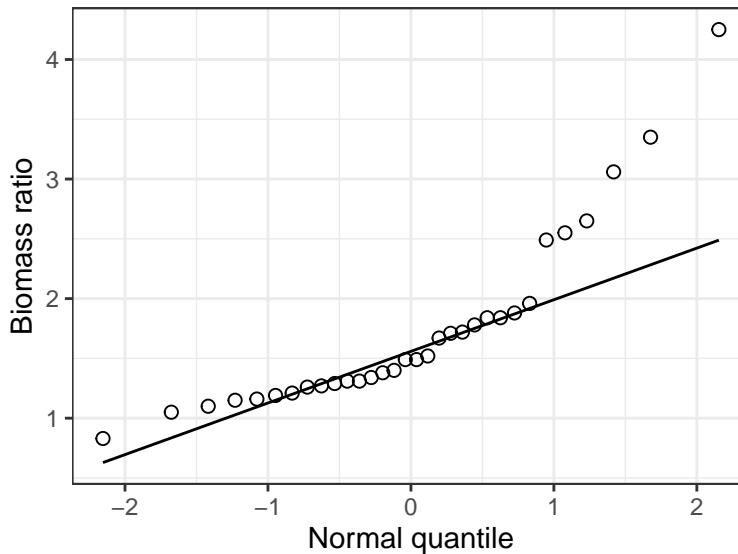


Figure 15.3: Normal quantile plot of the ‘biomass ratio’ of 32 marine reserves.

difficult to tell whether the data are normally distributed, then they probably are OK (at least sufficiently with respect to the assumption).

15.2.2 Shapiro-Wilk test for normality

Although graphical assessments are usually sufficient for checking the normality assumption, one can conduct a formal statistical test of the null hypothesis that the data are sampled from a population having a normal distribution. The test is called the **Shapiro-Wilk test**.

The Shapiro-Wilk test is a type of goodness-of-fit test.

Sometimes the **Shapiro-Wilk test** is applied in a hypothesis testing framework, but when it is applied as part of checking assumptions for another statistical test (like we’re doing here), one does not need to present it in a hypothesis test framework. However, the implied null hypothesis is that “The data are sampled from a population having a normal distribution”, and one does interpret the resulting P -value in the same way as usual, i.e. in relation to an α level (see below).

We’ll make use of the `shapiro.test` function from base R stats package:

```
shapiro.test
```

Let's use this on the `biomassRatio` variable in the “marine” dataset. We'll assign the output to an object called “shapiro.result”, then we'll have a look at the results.

The output from the `shapiro.test` function is not “tidy”, so we will use the `tidy` function from the `broom` package to make it tidy.

Here we go: we provide the function with the tibble name (“marine”) and the variable of interest after a “\$”:

```
shapiro.result <- shapiro.test(marine$biomassRatio)
```

Now tidy the output:

```
shapiro.result.tidy <- tidy(shapiro.result)
```

Now look at the results:

```
shapiro.result.tidy
```

```
## # A tibble: 1 x 3
##   statistic    p.value method
##       <dbl>     <dbl> <chr>
## 1      0.8175 0.00008851 Shapiro-Wilk normality test
```

The tidy object includes:

- The value of the test statistic for the Shapiro-Wilk test (although it is not shown, this test statistic is indicated with a “W”)
- The P -value associated with the test (“`p.value`”)
- The name of the test used (“`method`”)

Given that the P -value is less than a conventional α level of 0.05, the test is telling us that the data do not conform to a normal distribution. Of course we already knew that from our visual assessments!

Here's an example statement:

The assumption of normality was checked visually using a normal quantile plot, and a Shapiro-Wilk test, which revealed evidence of non-normality (Shapiro-Wilk test, $W = 0.82$, P -value < 0.001).

Important: Visual assessments of normality are preferred, because the outcome of the Shapiro-Wilk test is sensitive to sample size: a small sample size will often yield a false negative, whereas very large sample sizes could yield false positives more than it should.

15.3 Checking the equal-variance assumption

Some statistical tests, such as the 2-sample *t*-test and ANOVA, assume that the variance (σ) of the numeric response variable is the same among the populations being compared.

For this we use the **Levene's test**, which we implement using the `leveneTest` function from the `car` package:

```
?leveneTest
```

Like the Shapiro-Wilk test, the Levene's test can be applied in a hypothesis testing framework, but when it is used to evaluate the equal-variance assumption, it need not be.

Nevertheless, the *implied* null hypothesis is that the variance of the numeric response variable is the same among the populations being compared. So in the case where a numeric variable is being compared among two groups, then the implied null hypothesis is that $(\sigma(1) = \sigma(2))$. The usual α level of 0.05 can be used.

We'll use the “students” dataset, and check whether `height_cm` exhibits equal variance among students with different dominant eyes (left or right).

Let's look at the code, and explain after:

```
height.vartest <- leveneTest(height_cm ~ Dominant_eye, data = students)
```

If you get a warning about a variable being “coerced to factor”, that's OK! It is simply telling you that it took the categorical variable and treated it as a ‘factor’ variable.

In the code chunk above we:

- assign the results to a new object called “height.vartest”
- use the `leveneTest` function, in which the arguments are:
 - the numeric response variable (“`height_cm`”)
 - then the “`~`” symbol
 - then the categorical variable “`dominant_eye`”
 - then the “`data = students`” specifies the data object name

TIP: The argument to the `leveneTest` function that is in the form $Y \sim X$ is one we'll use several times.

Let's look at the results:

```
height.vartest
```

```
## Levene's Test for Homogeneity of Variance (center = median)
##          Df F value Pr(>F)
## group     1  0.907 0.3424
##           152
```

The results include:

- the degrees of freedom for the test
- the value of the test statistic “F”
- the P -value associated with the test statistic

For the student height example, the P -value is greater than the standard α of 0.05, so there’s no evidence against the assumption of equal variance.

A reasonable statement would be:

“A Levene’s test showed no evidence against the assumption of equal variance ($F = 0.91$; P -value = 0.342).”

TIP: As part of your main statistical test, such as a two-sample t -test or an ANOVA, you would typically provide a graph such as a stripchart or violin plot to visualize how the numeric response variable varies among the groups of the categorical explanatory variable. Such plots may often reveal that the spread of values of the response variable varies considerably among the groups, underscoring the need to check the equal variance assumption.

15.4 Data transformations

Here we learn how to transform numeric variables using two common methods:

- log-transform
- logit-transform

There are many other types of transformations that can be performed, some of which are described in **Chapter 13** of the course text book.

Contrary to what is suggested in the text, it is better to use the “logit” transformation rather than the “arcsin square-root” transformation for proportion or percentage data, as described in this article by Warton and Hui (2011).

15.4.1 Log-transform

When one observes a right-skewed frequency distribution, as seen here in the marine biomass ratio data, a log-transformation often helps.

```
marine %>%
  ggplot(aes(x = biomassRatio)) +
  geom_histogram(binwidth = 0.5, color = "black", fill = "lightgrey",
                 boundary = 0, closed = "left") +
  xlab("Biomass ratio") +
  ylab("Frequency") +
  theme_bw()
```

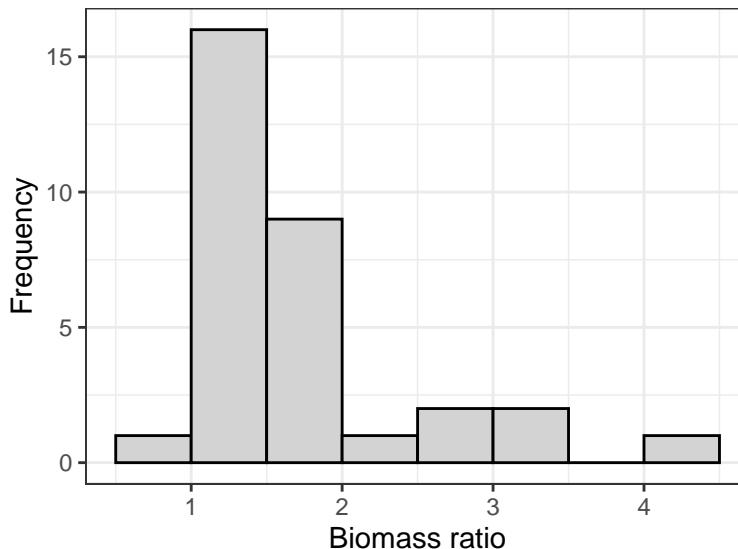


Figure 15.4: The frequency distribution of the ‘biomass ratio’ of 32 marine reserves.

To log-transform the data, simply create a new variable in the dataset using the `mutate` function (from the `dplyr` package) that we’ve seen before. Here we’ll call our new variable `logbiomass`, and use the `log` function to take the **natural log** of the “biomassRatio” variable.

We’ll assign the output to the same, original “tibble” called “marine”:

```
marine <- marine %>%
  mutate(logbiomass = log(biomassRatio))
```

Alternatively, you could use this (less tidy) code to get the same result:

```
marine$logbiomass <- log(marine$biomassRatio)
```

If your variable includes zeros, then you'll need to take extra steps, as described in the next section.

Let's look at the tibble now:

```
marine
```

```
## # A tibble: 32 x 2
##   biomassRatio logbiomass
##       <dbl>      <dbl>
## 1     1.34    0.2927
## 2     1.96    0.6729
## 3     2.49    0.9123
## 4     1.27    0.2390
## 5     1.19    0.1740
## 6     1.15    0.1398
## 7     1.29    0.2546
## 8     1.05    0.04879
## 9     1.1     0.09531
## 10    1.21    0.1906
## # i 22 more rows
```

Now let's look at the histogram of the log-transformed data:

```
marine %>%
  ggplot(aes(x = logbiomass)) +
  geom_histogram(binwidth = 0.25, color = "black", fill = "lightgrey",
                 boundary = -0.25) +
  xlab("Biomass ratio (log-transformed)") +
  ylab("Frequency") +
  theme_bw()
```

Now the quantile plot:

```
marine %>%
  ggplot(aes(sample = logbiomass)) +
  stat_qq(shape = 1, size = 2) +
  stat_qq_line() +
  ylab("Biomass ratio (log)") +
  xlab("Normal quantile") +
  theme_bw()
```

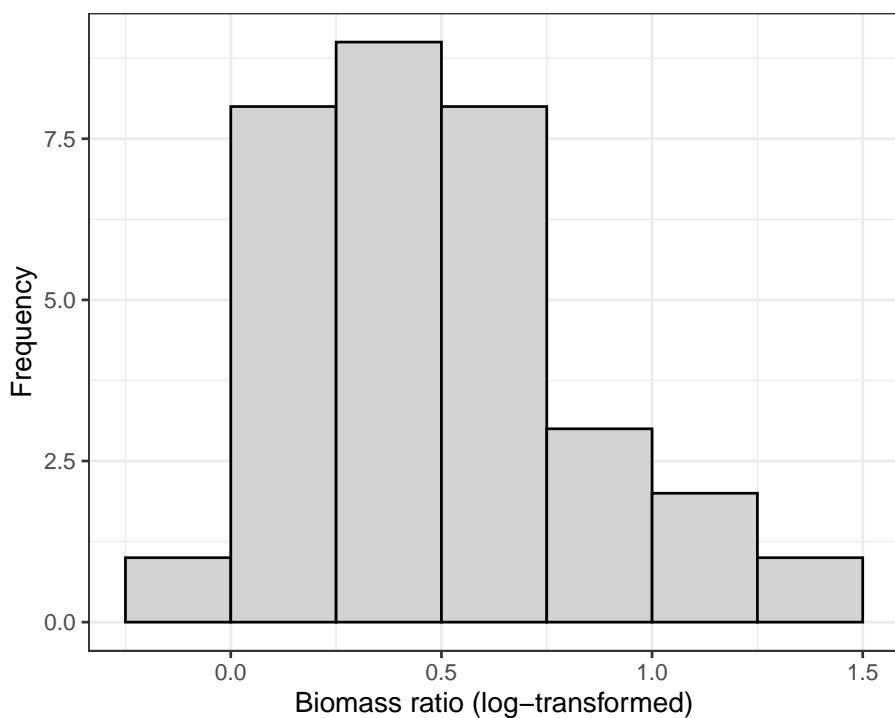


Figure 15.5: The frequency distribution of the ‘biomass ratio’ of 32 marine reserves (log-transformed).

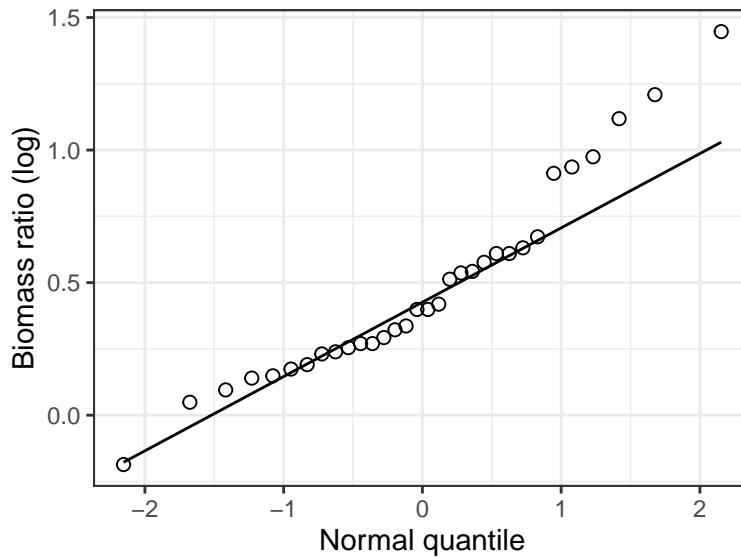


Figure 15.6: Figure: Normal quantile plot of the ‘biomass ratio’ of 32 marine reserves (log-transformed).

The log-transform definitely helped, but the distribution still looks a bit wonky: several of the points are quite far from the line.

Just to be sure, let’s conduct a Shapiro-Wilk test, using an α level of 0.05, and remembering to tidy the output:

```
shapiro.log.result <- shapiro.test(marine$logbiomass)
shapiro.log.result.tidy <- tidy(shapiro.log.result)
shapiro.log.result.tidy
```

```
## # A tibble: 1 x 3
##   statistic p.value method
##       <dbl>   <dbl> <chr>
## 1     0.938 0.06551 Shapiro-Wilk normality test
```

The P -value is greater than 0.05, so we’d conclude that there’s no evidence against the assumption that these data come from a normal distribution.

For this example a reasonable statement would be:

Based on the normal quantile plot (Fig. 18.9), and a Shapiro-Wilk test, we found no evidence against the normality assumption (Shapiro-Wilk test, $W = 0.94$, P -value = 0.066).

You could now proceed with the statistical test (e.g. one-sample t -test) using the **transformed** variable.

15.4.2 Dealing with zeroes

If you try to log-transform a value of zero, R will return a `-Inf` value.

In this case, you'll need to add a constant (value) to each observation, and convention is to simply add 1 to each value prior to log-transforming.

In fact, you can add any constant that makes the data conform best to the assumptions once log-transformed. The key is that you must add the same constant to every value in the variable.

You then conduct the analyses using these newly transformed data (which had 1 added prior to log-transform), remembering that after back-transformation (see below), you need to subtract 1 to get back to the original scale.

Example code showing how to check for zeroes

We'll create a dataset to work with called "apples".

Don't worry about learning this code...

```
set.seed(345)
apples <- as_tibble(data.frame(biomass = rlnorm(n = 14, meanlog = 1, sdlog = 0.7)))
apples$biomass[4] <- 0
```

Here's the resulting dataset, which includes a variable "biomass" that would benefit from log-transform:

```
apples
```

```
## # A tibble: 14 x 1
##   biomass
##       <dbl>
## 1     1.569
## 2     2.235
## 3     2.428
## 4      0
## 5     2.593
## 6     1.745
## 7     1.420
## 8     9.003
## 9     8.657
## 10    9.654
## 11    10.04
```

```
## 12    1.020
## 13    1.500
## 14    3.397
```

Here's a quantile plot of the biomass variable:

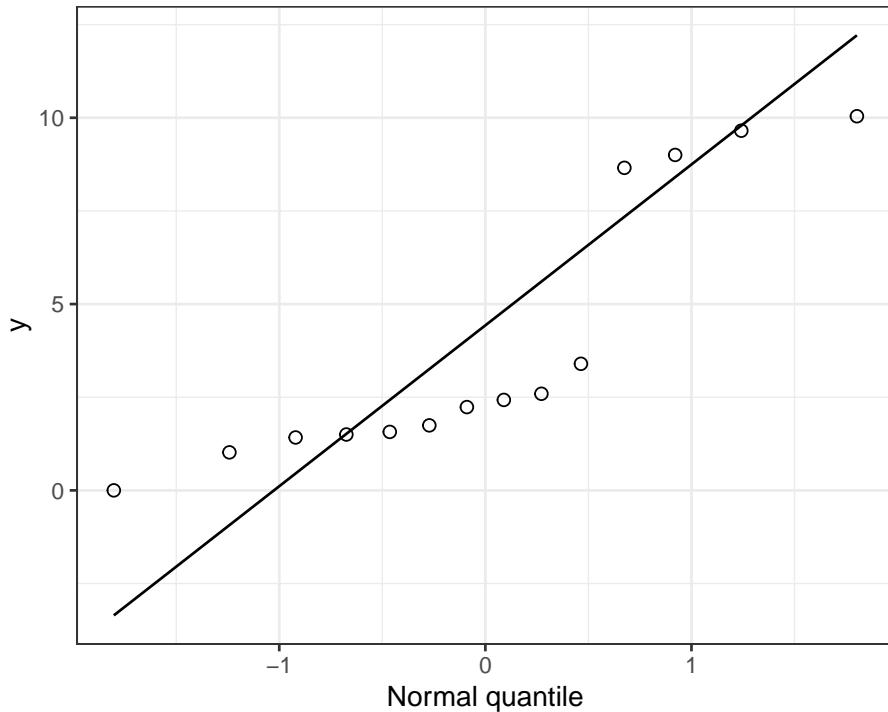


Figure 15.7: Normal quantile plot of made-up biomass data.

Let's first see what happens when we try to log-transform the “biomass” variable:

```
log(apples$biomass)
```

```
## [1] 0.45056428 0.80433995 0.88697947      -Inf 0.95272789 0.55653572
## [7] 0.35059322 2.19753971 2.15833805 2.26733776 2.30674058 0.02011719
## [13] 0.40525957 1.22291473
```

Notice we get a “-Inf” value.

The following code tallies the number of observations in the “biomass” variable that equal zero.

If this sum is greater than zero, then you'll need to add a constant to all observations when transforming.

```
sum(apples$biomass == 0)
```

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

So we have one value that equals zero.

So let's add a 1 to each observation during the process of log-transforming:

```
apples <- apples %>%
  mutate(logbiomass_plus1 = log(biomass + 1))
```

Notice that we name the new variable “logbiomass_plus1” in a way that indicates we've added 1 prior to log-transforming, and that in the `log` calculation we've used “biomass + 1”.

Let's see the result:

```
apples
```

```
## # A tibble: 14 x 2
##   biomass logbiomass_plus1
##       <dbl>          <dbl>
## 1     1.569      0.9436
## 2     2.235      1.174
## 3     2.428      1.232
## 4       0          0
## 5     2.593      1.279
## 6     1.745      1.010
## 7     1.420      0.8837
## 8     9.003      2.303
## 9     8.657      2.268
## 10    9.654      2.366
## 11   10.04       2.402
## 12   1.020       0.7033
## 13   1.500       0.9162
## 14   3.397      1.481
```

Notice that we still have a zero in the newly created variable, **AFTER** having transformed, because for that value we calculated the log of “1” (which equals zero). That's OK!

That's a bit better. We would now use this new variable in our analyses (assuming it meets the normality assumption).

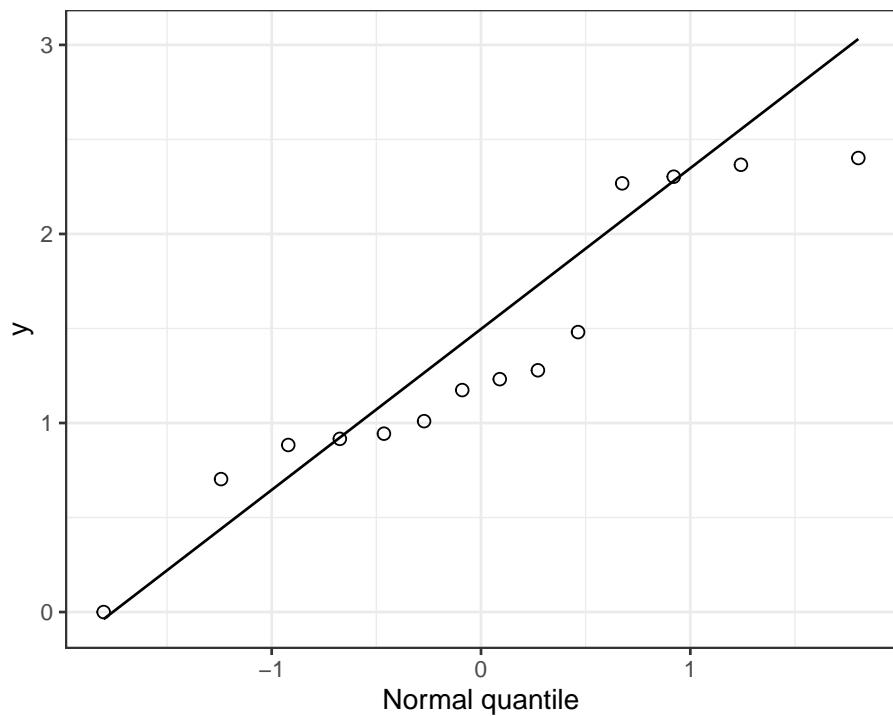


Figure 15.8: Normal quantile plot of made-up biomass data, log-transformed.

15.4.3 Log bases

The `log` function calculates the natural logarithm (base e), but related functions permit any base:

```
?log
```

For instance, `log10` uses log base 10:

```
marine <- marine %>%
  mutate(log10biomass = log10(biomassRatio))
marine
```

```
## # A tibble: 32 x 3
##   biomassRatio logbiomass log10biomass
##       <dbl>      <dbl>        <dbl>
## 1       1.34     0.2927     0.1271
## 2       1.96     0.6729     0.2923
## 3       2.49     0.9123     0.3962
## 4       1.27     0.2390     0.1038
## 5       1.19     0.1740     0.07555
## 6       1.15     0.1398     0.06070
## 7       1.29     0.2546     0.1106
## 8       1.05     0.04879    0.02119
## 9       1.1      0.09531    0.04139
## 10      1.21     0.1906     0.08279
## # i 22 more rows
```

Or the alternative code:

```
marine$log10biomass <- log10(marine$biomassRatio)
```

15.4.4 Back-transforming log data

In order to back-transform data that were transformed using the natural logarithm (`log`), you make use of the `exp` function:

```
?exp
```

Let's try it, creating a new variable in the “marine” dataset so we can compare to the original “`biomassRatio`” variable:

First, back-transform the data and store the results in a new variable within the data frame:

```
marine <- marine %>%
  mutate(back_biomass = exp(logbiomass))
```

Now have a look at the first few lines of the tibble (selecting the original “biomassRatio” and new “back_biomass” variables) to see if the data values are identical, as they should be:

```
marine %>%
  select(biomassRatio, back_biomass)

## # A tibble: 32 x 2
##   biomassRatio back_biomass
##       <dbl>      <dbl>
## 1        1.34      1.34
## 2        1.96      1.96
## 3        2.49      2.49
## 4        1.27      1.27
## 5        1.19      1.19
## 6        1.15      1.15
## 7        1.29      1.29
## 8        1.05      1.05
## 9        1.1       1.1
## 10       1.21      1.21
## # i 22 more rows
```

Yup, it worked!

If you had added a 1 to your variable prior to log-transforming, then the code would be:

```
marine <- marine %>%
  mutate(back_biomass = exp(logbiomass) - 1)
```

Notice the minus 1 comes after the `exp` function is executed.

If you had used the log base 10 transformation, then the code to back-transform is as follows:

```
10^(marine$log10biomass)
```

```
## [1] 1.34 1.96 2.49 1.27 1.19 1.15 1.15 1.29 1.05 1.05 1.10 1.21 1.31 1.26 1.38 1.49 1.84
## [16] 1.84 3.06 2.65 4.25 3.35 2.55 1.72 1.52 1.49 1.49 1.67 1.78 1.71 1.88 0.83 1.16
## [31] 1.31 1.40
```

The \wedge symbol stands for “exponent”. So here we’re calculating 10 to the exponent x , where x is each value in the dataset.

15.4.5 Logit transform

Variables whose data represent proportions or percentages are, by definition, not drawn from a normal distribution: they are bound by 0 and 1 (or 0 and 100%). They should therefore be **logit-transformed**.

The `boot` package includes both the `logit` function and the `inv.logit` function, the latter for back-transforming.

However, the `logit` function that is in the `car` package is better, because it accommodates the possibility that your dataset includes a zero and / or a one (equivalently, a zero or 100 percent), and has a mechanism to deal with this properly.

The `logit` function in the `boot` package does not deal with this possibility for you.

However, the `car` package does not have a function that will back-transform logit-transformed data.

This is why we'll use the **logit function from the car package, and the inv.logit function from the boot package!**

Let's see how it works with the `flowers` dataset, which includes a variable `propFertile` that describes the proportion of seeds produced by individual plants that were fertilized.

Let's visualize the data with a normal quantile plot:

```
flowers %>%
  ggplot(aes(sample = propFertile)) +
  stat_qq(shape = 1, size = 2) +
  stat_qq_line() +
  ylab("Proportion of seeds fertilized") +
  xlab("Normal quantile") +
  theme_bw()
```

Clearly not normal!

Now let's logit-transform the data.

To ensure that we're using the correct `logit` function, i.e. the one from the `car` package and NOT from the `boot` package, we can use the `::` syntax, with the package name preceding the double-colons, which tells R the correct package to use.

```
flowers <- flowers %>%
  mutate(logitfertile = car::logit(propFertile))
```

Or the alternative code:

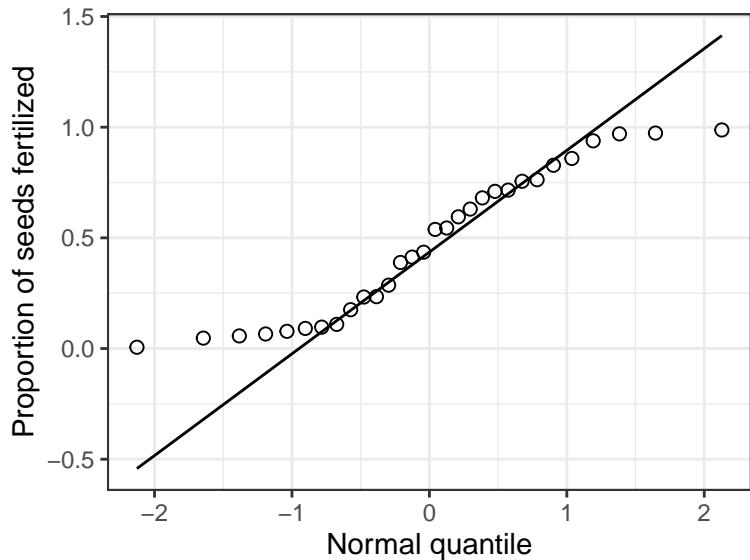


Figure 15.9: Normal quantile plot of the proportion of seeds fertilized on 30 plants (left) and the corresponding normal quantile plot (right)

```
flowers$logitfertile <- car::logit(flowers$propFertile)
```

Now let's visualize the transformed data:

```
flowers %>%
  ggplot(aes(sample = logitfertile)) +
  stat_qq(shape = 1, size = 2) +
  stat_qq_line() +
  ylab("Proportion of seeds fertilized (logit-transformed") +
  xlab("Normal quantile") +
  theme_bw()
```

That's much better!

Next we learn how to back-transform logit data.

15.4.6 Back-transforming logit data

We'll use the `inv.logit` function from the `boot` package:

```
?boot::inv.logit
```

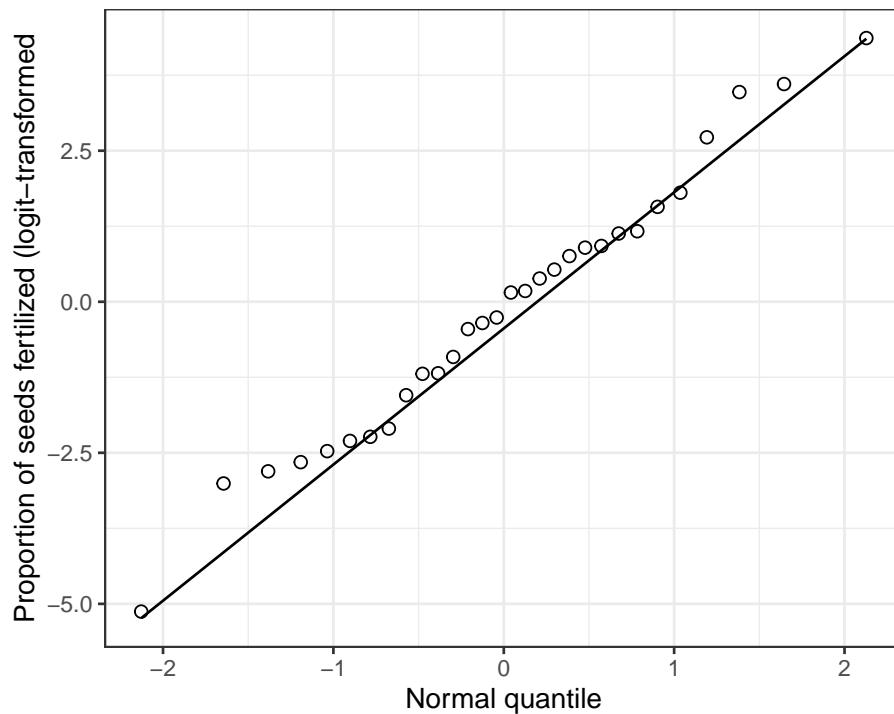


Figure 15.10: Normal quantile plot of the proportion of seeds fertilized (logit transformed) on 30 plants

First do the back-transform:

```
flowers <- flowers %>%
  mutate(flower_backtransformed = boot::inv.logit(logitfertile))
```

Or alternative code:

```
flowers$flower_backtransformed <- boot::inv.logit(flowers$logitfertile)
```

Let's have a look at the original “propFertile” variable and the “flower_backtransformed” variable to check that they're identical:

```
flowers %>%
  select(propFertile, flower_backtransformed)
```

```
## # A tibble: 30 x 2
##   propFertile flower_backtransformed
##       <dbl>             <dbl>
## 1     0.06571        0.06571
## 2     0.9874         0.9874
## 3     0.3888         0.3888
## 4     0.6805         0.6805
## 5     0.07781        0.07781
## 6     0.9735         0.9735
## 7     0.1755         0.1755
## 8     0.1092         0.1092
## 9     0.7158         0.7158
## 10    0.04708        0.04708
## # i 20 more rows
```

Yup, it worked!

15.4.7 When to back-transform?

You should back-transform your data when it makes sense to communicate findings on the original measurement scale.

The most common example is **reporting confidence intervals for a mean or difference in means**.

For example, imagine you had calculated a confidence interval for the log-transformed marine biomass ratio data, and your limits were as follows:

$$0.347 < \ln(\mu) < 0.611$$

These are the log-transformed limits! So we need to back-transform them to get them in the original scale:

```
lower.limit <- exp(0.347)
upper.limit <- exp(0.611)
```

So now the back-transformed interval is:

$$1.415 < \mu < 1.842$$

Voila!

Chapter 16

Comparing means among more than two groups

Tutorial learning objectives

- Learn how to analyze a numeric response variable in relation to a single categorical explanatory variable that has more than two groups. In other words, we will learn how to compare the means of more than 2 groups.
- Learn how to implement all the steps of an **Analysis of Variance** (ANOVA), which is often the most suitable method for testing the null hypothesis of no difference between the means of more than two groups (i.e. $\mu_1 = \mu_2 = \mu_3 = \mu_i..$)

It might seem strange that a test designed to compare means among groups is called “Analysis of Variance”. The reason lies in how the test actually works, as described in Chapter 15 of the text.

- Here we learn **fixed-effects ANOVA** (also called Model-1 ANOVA), in which the different categories of the explanatory variable are predetermined, of direct interest, and repeatable. These methods therefore typically apply to **experimental studies**.
- When the groups are sampled at random from a larger population of groups, as in most **observational studies**, one should typically use a **random-effects ANOVA** (also called Model-2 ANOVA). Consult the following webpage for tutorials on how to conduct various types of ANOVA.
- In this tutorial we’re learning about a **One-way ANOVA**, in which there is only one explanatory (categorical) variable

- Learn the assumptions of ANOVA
- Learn appropriate tables and graphs to accompany ANOVA
- Learn how to conduct a **post-hoc** comparison among all pairs of group means: the **Tukey-Kramer post-hoc test**

16.1 Load packages and import data

Load the `tidyverse`, `knitr`, `naniar`, `car`, `skimr`, and `broom` packages:

```
library(tidyverse)
library(knitr)
library(naniar)
library(skimr)
library(broom)
library(car)
```

For this tutorial we'll use the “circadian.csv” dataset. These data are associated with Example 15.1 in the text.

The `circadian` data describe melatonin production in 22 people randomly assigned to one of three light treatments.

```
circadian <- read_csv("https://raw.githubusercontent.com/ubco-biology/BIO1202/main/data/circadian.csv")

## Rows: 22 Columns: 2
## -- Column specification -----
## Delimiter: ","
## chr (1): treatment
## dbl (1): shift
##
## 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.
```

TIP: When analyzing a numeric response variable in relation to a categorical variable, it is good practice to ensure that the categorical variable is treated as a “factor” type variable by R, and that the categories or “levels” of the factor are ordered in the correct way for the purpose of graphs and / or tables of descriptive statistics. For example, there is often a “control” group (category), and if so, this group should come first in a graph or table.

Let's follow this practice for the “circadian” dataset.

If we look at example 15.1 in the text, and the corresponding figure (Fig. 15.1), we see that there should be three treatment groups “Control”, “Knees”, and “Eyes”, in that order.

Let's have a look at the data.

```
circadian %>%
  skim_without_charts()

(#tab:anova_lookdata)Data summary

Name
Piped data
Number of rows
22
Number of columns
2



---


Column type frequency:
character
1
numeric
1



---


Group variables
None

Variable type: character
skim_variable
n_missing
complete_rate
min
max
empty
n_unique
whitespace
treatment
0
```

```
1  
4  
7  
0  
3  
0  
Variable type: numeric  
skim_variable  
n_missing  
complete_rate  
mean  
sd  
p0  
p25  
p50  
p75  
p100  
shift  
0  
1  
-0.71  
0.89  
-2.83  
-1.33  
-0.66  
-0.05  
0.73
```

And also have a look at the tibble itself:

```
circadian
```

```
## # A tibble: 22 x 2
##   treatment shift
##   <chr>     <dbl>
## 1 control    0.53
## 2 control    0.36
## 3 control    0.2 
## 4 control   -0.37
## 5 control   -0.6 
## 6 control   -0.64
## 7 control   -0.68
## 8 control   -1.27
## 9 knee       0.73
## 10 knee      0.31
## # i 12 more rows
```

The data are stored in tidy (long) format - good!

We can see the unique values of the categorical variable as follows:

```
circadian %>%
  select(treatment) %>%
  unique()
```

```
## # A tibble: 3 x 1
##   treatment
##   <chr>
## 1 control
## 2 knee
## 3 eyes
```

Two important things to notice here:

- The categorical variable “treatment” is recognized as a “character” variable (“chr”) instead of a “factor” variable,
- The category names are not capitalized

Let’s recode this using the method we previously learned in the 2-sample *t*-test tutorial, using the `recode_factor` function from the `dplyr` package (loaded as part of the `tidyverse`).

Be sure to provide the variables in the order that you want them to appear in any tables, figures, and analyses.

```
circadian <- circadian %>%
  mutate(treatment = recode_factor(treatment, control = "Control", knee = "Knee", eyes
```

Have a look:

```
circadian$treatment
```

```
## [1] Control Control Control Control Control Control Control Control Control Knee
## [10] Knee     Knee     Knee     Knee     Knee     Knee     Eyes    Eyes    Eyes
## [19] Eyes    Eyes    Eyes    Eyes
## Levels: Control Knee Eyes
```

OK, so not only have we successfully changed the type of variable to factor, and capitalized the category names, but it turns out that the ordering of the factor “levels” is in the correct order, i.e. consistent with how it’s displayed in the text figure in example 15.1.

Now we’re ready to proceed with ANOVA!

16.2 Analysis of variance

Follow these steps when conducting a hypothesis test:

- State the null and alternative hypotheses
- Set an α level
- Identify the appropriate test & provide rationale
- Prepare an appropriately formatted table of descriptive statistics for the response variable grouped by the categories of the explanatory variable.
- Visualize the data with a good graph and interpret the graph
- Assumptions
 - state the assumptions of the test
 - use appropriate figures and / or tests to check whether the assumptions of the statistical test are met
 - transform data to meet assumptions if required
 - if assumptions can’t be met (e.g. after transformation), use non-parametric test and repeat steps above
- Conduct the test, and report the test statistic and associated P -value

- Calculate and include a confidence interval (e.g. for *t*-tests) or R^2 value (e.g. for ANOVA) when appropriate
- Optionally, for ANOVA, conduct a “post-hoc” comparison of means using “Tukey-Kramer post-hoc test”
- Draw the appropriate conclusion and communicate it clearly
- Your concluding statement should refer to an ANOVA table (if required), a good figure, and the R^2 value

Additional, OPTIONAL steps for ANOVA tests

These steps may be warranted depending on the context.

- An appropriately formatted ANOVA results table (definitely include this for any research projects you’re writing up)
- A good figure with results of **post-hoc tests** included (if these tests were conducted)

16.2.1 Hypothesis statements

The null hypothesis of ANOVA is that the population means μ_i are the same for all treatments. Thus, for our current example in which melatonin production was measured among treatment groups:

H_0 : Mean melatonin production is equal among all treatment groups ($\mu_1 = \mu_2 = \mu_3$).

H_A : At least one treatment group’s mean is different from the others

OR

H_A : Mean melatonin production is not equal among all treatment groups.

We’ll set $\alpha = 0.05$.

We’ll use a fixed-effects ANOVA, which uses the *F* test statistic.

16.2.2 A table of descriptive statistics

In a previous tutorial we learned how to calculate descriptive statistics for a numeric variable grouped by a categorical variable. In the comparing 2 means tutorial we also learned how to use the `t.test` function to calculate confidence intervals for a numeric variable.

Let’s use all these skills to generate a table of summary statistics for the “shift” variable, grouped by “treatment”:

```

circadian.stats <- circadian %>%
  group_by(treatment) %>%
  summarise(
    Count = n() - naniar::n_miss(shift),
    Count_NA = naniar::n_miss(shift),
    Mean = mean(shift, na.rm = TRUE),
    SD = sd(shift, na.rm = TRUE),
    SEM = SD/sqrt(Count),
    Low_95_CL = t.test(shift, conf.level = 0.95)$conf.int[1],
    Up_95_CL = t.test(shift, conf.level = 0.95)$conf.int[2]
  )

```

Let's have a look at the result, and we'll use the `kable` function to make the table nicer:

```
kable(circadian.stats, digits = 4)
```

treatment

Count

Count_NA

Mean

SD

SEM

Low_95_CL

Up_95_CL

Control

8

0

-0.3088

0.6176

0.2183

-0.8250

0.2075

Knee

7

0

```
-0.3357
```

```
0.7908
```

```
0.2989
```

```
-1.0671
```

```
0.3957
```

```
Eyes
```

```
7
```

```
0
```

```
-1.5514
```

```
0.7063
```

```
0.2670
```

```
-2.2047
```

```
-0.8982
```

16.2.3 Visualize the data

We are interested in visualizing a numeric response variable in relation to a categorical explanatory variable.

We learned in an earlier tutorial that we can use a stripchart, violin plot, or boxplot to visualize a numerical response variable in relation to a categorical variable.

It is commonplace to use a stripchart for small sample sizes in each group.

We'll use code we learned in an earlier tutorial.

However, because we're now not only describing and visualizing data, but also testing a hypothesis about differences in group means, it is best to add group means and error bars to our stripchart.

The `stat_summary` function from the `ggplot2` package is what provides the error bars and group means:

```
?stat_summary
```

Here's the code:

```

circadian %>%
  ggplot(aes(x = treatment, y = shift)) +
  geom_jitter(colour = "darkgrey", size = 3, shape = 1, width = 0.1) +
  stat_summary(fun.data = mean_se, geom = "errorbar",
               colour = "black", width = 0.1,
               position = position_nudge(x = 0.15)) +
  stat_summary(fun = mean, geom = "point",
               colour = "black", size = 3,
               position = position_nudge(x = 0.15)) +
  xlab("Light treatment") +
  ylab("Shift in circadian rhythm (h)") +
  theme_classic()

```

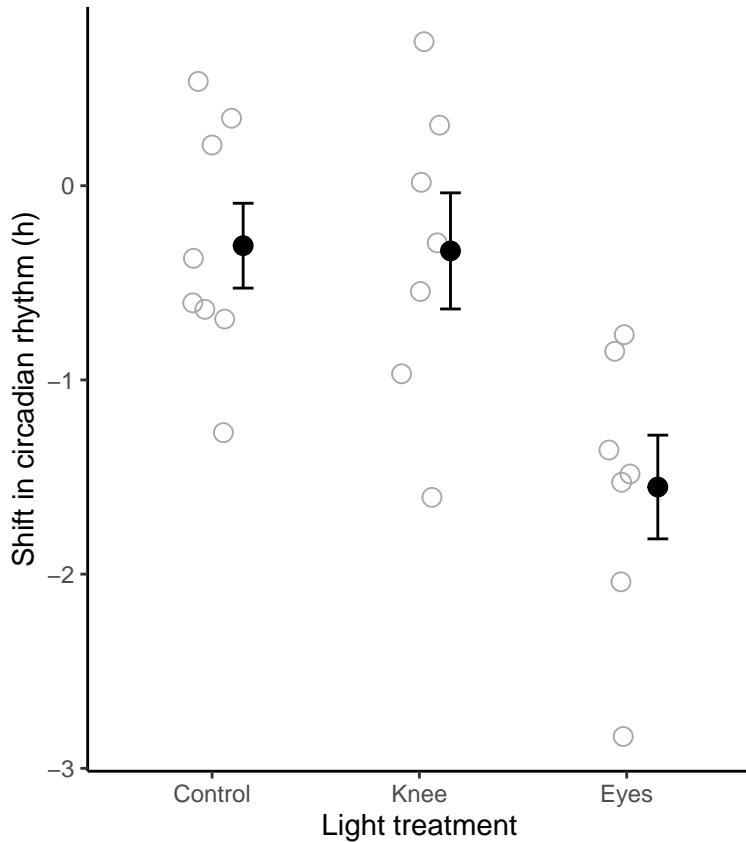


Figure 16.1: Stripchart of phase shifts in the circadian rhythm of melatonin production in 22 participants of an experiment. Solid circles denote group means, and bars +/- one standard error

We've added two sets of `stat_summary` functions, one for adding the group mean, and one for adding the error bars.

- The argument “`fun = mean_se`” adds the standard error of the group means
- the “`geom = ‘errorbar’`” tells ggplot to use an error bar format
- the “`position = position_nudge(x = 0.15)`” tells ggplot to nudge the error bar off to the side a bit, so it doesn't obscure the data points
- the next `stat_summary` function is the same, but this time for the group means.

Clearly there is no difference between the control and knee treatment groups, but we'll have to await the results of the ANOVA to see whether the mean of the “eyes” group is different.

16.2.4 Assumptions of ANOVA

The assumptions of ANOVA are:

- The measurements in every group represent a random sample from the corresponding population (**NOTE**: for an experimental study, the assumption is that subjects are randomly assigned to treatments)
 - The response variable has a normal distribution in each population
 - The variance of the response variable is the same in all populations (called the “homogeneity of variance” assumption)
-

Test for normality

The sample sizes per group are rather small, so graphical aids such as histograms or normal quantile plots will not be particularly helpful.

Instead, let's rely on the fact that (i) the strip chart in Figure 2 does not show any obvious outliers in any of the groups, and (ii) the central limit theorem makes ANOVAs quite robust to deviations in normality, especially with larger sample sizes, but even with relatively small sample sizes such as these. So, we'll proceed under the assumption that the measurements are normally distributed within the three populations.

Test for equal variances

Now we need to test the assumption of equal variance among the groups, using Levene's Test as we learned in the checking assumptions tutorial.

```
variance.check <- leveneTest(shift ~ treatment, data = circadian)
variance.check

## Levene's Test for Homogeneity of Variance (center = median)
##          Df F value Pr(>F)
## group    2  0.1586 0.8545
##          19
```

We see that the P -value for the test is much greater than 0.05, so the null hypothesis of equal variance is **not** rejected. There is no evidence against the assumption that the variances are equal.

A reasonable statement:

“A Levene's test showed no evidence against the assumption of equal variance ($F = 0.16$; P -value = 0.854).”

IF assumptions are not met

A later section discusses what to do if assumptions are NOT met.

16.2.5 Conduct the ANOVA test

There are two steps to conducting an ANOVA in R:

- Use the `lm` function (from the base package) (it stands for “linear model”) to create an object that holds the key output from the test
- Use the `anova` function (from the base package) on the preceding `lm` output object to generate the appropriate output for interpreting the results.

The `lm` function uses the syntax: $Y \sim X$.

Let's do this step first:

```
circadian.lm <- lm(shift ~ treatment, data = circadian)
```

Next the `anova` step:

```
circadian.lm.anova <- anova(circadian.lm)
```

Let's have a look at what this produced:

```
circadian.lm.anova
```

```
## Analysis of Variance Table
##
## Response: shift
##           Df Sum Sq Mean Sq F value    Pr(>F)
## treatment   2 7.2245  3.6122  7.2894 0.004472 ***
## Residuals  19 9.4153  0.4955
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

This is not in the format of a traditional ANOVA table (as described in Chapter 15 of the text): it is missing the “Total” row, and it puts the degrees of freedom (df) column before the Sums of Squares. Also, the column headers could be more informative.

Nor is the object in “tidy” format.

Optionally, we can use the `tidy` function from the `broom` package to tidy the output as follows:

```
circadian.lm.anova.tidy <- circadian.lm.anova %>%
  broom::tidy()
```

Have a look at the tidier (but still not ideal) output:

```
circadian.lm.anova.tidy
```

```
## # A tibble: 2 x 6
##   term        df  sumsq  meansq statistic   p.value
##   <chr>     <int> <dbl>   <dbl>     <dbl>      <dbl>
## 1 treatment     2 7.224  3.612     7.289  0.004472
## 2 Residuals    19 9.415  0.4955    NA       NA
```

16.2.5.1 OPTIONAL: Generate nice ANOVA table

Let's look again at the raw output from the `anova` function:

```
circadian.lm.anova
```

```
## Analysis of Variance Table
##
## Response: shift
##             Df Sum Sq Mean Sq F value    Pr(>F)
## treatment   2 7.2245  3.6122  7.2894 0.004472 ***
## Residuals 19 9.4153  0.4955
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

In the code chunk below we create a new function called “create.anova.table”, which we can use to convert an “anova” object (like the one above) to an appropriately formatted ANOVA table.

When you run this code chunk, it creates the function in your workspace. Once it is created, you can use it.

You will need to run this chunk to create the function each time you start a new R session. **AND** this is a quick-and-dirty function that does **NOT** follow all the best practices in coding!

```
create.anova.table <- function(intable){
  # first check that input table is of class "anova"
  if(class(intable)[1] != "anova") stop("Input object not of class 'anova'")
  require(tidyverse) # ensure tidyverse is loaded
  require(broom) # ensures broom is loaded
  tidy.intable <- tidy(intable)
  # "intable" is a the object from the `anova` output
  temp.anova <- tidy.intable %>%
    select("term", "sumsq", "df", "meansq", "statistic", "p.value") # reorder columns
  temp.anova <- temp.anova %>%
    rename(Source = term, SS = sumsq, MS = meansq, F = statistic, P_val = p.value) # rename
  totals <- data.frame(
    Source = "Total",
    df = sum(temp.anova$df),
    SS = sum(temp.anova$SS),
    MS = NA,
    F = NA,
    P_val = NA
  ) # calculate totals
  nice.anova.table <- as_tibble(rbind(temp.anova, totals)) # generate table
  return(nice.anova.table)
}
```

Let’s try out the function, using our original anova table:

```
 nice.anova.table <- create.anova.table(circadian.lm.anova)
```

And present it:

```
kable(nice.anova.table, digits = 4,  
      caption = "ANOVA table for the circadian rythm experiment.",  
      lable = "niceanovapdf")
```

ANOVA table for the circadian rythm experiment.

Source

SS

df

MS

F

P_val

treatment

7.2245

2

3.6122

7.2894

0.0045

Residuals

9.4153

19

0.4955

NA

NA

Total

16.6398

21

NA

NA

NA

Nice!

16.2.6 Calculate R^2 for the ANOVA

One measure that is typically reported with any “linear model” like ANOVA is the “variance explained” or **coefficient of determination**, denoted R^2 .

This value indicates the fraction of the variation in Y that is explained by groups.

The remainder of the variation ($1 - R^2$) is “error”, or variation that is left unexplained by the groups.

To calculate this we use two steps, and we go back to our original “lm” object we created earlier:

```
circadian.lm.summary <- summary(circadian.lm)
circadian.lm.summary$r.squared
```

```
## [1] 0.4341684
```

We’ll refer to this value in our concluding statement.

Before we get there, we need to figure out **which group mean(s) differ??**.

16.2.7 Tukey-Kramer post-hoc test

As described in Chapter 15 in the text, **planned comparisons** (aka planned contrasts) are ideal and most powerful, but unfortunately we often need to conduct **unplanned comparisons** to assess which groups differ in our ANOVA test. This is what we’ll learn here.

We can guess from our stripchart (Figure 16.1) that it’s the “Eyes” treatment group that differs from the others, but we need a formal test.

We could simply conduct three 2-sample t -tests on each of the three pair-wise comparisons, but then we would inflate our **Type-I error rate**, due to multiple-testing.

The Tukey-Kramer “post-hoc” (unplanned) test adjusts our P -values correctly to account for multiple tests.

For this test we use the **TukeyHSD** function in the base stats package (HSD stands for “Honestly Significant Difference”).

```
?TukeyHSD
```

The one catch in using this function is that we need to re-do our ANOVA analysis using a different function... the **aov** function in lieu of the **lm** function.

This is necessary because the `TukeyHSD` function is expecting a particular object format, which only the `aov` function provides.

First, let's re-do the ANOVA analysis using the `aov` function, creating a new object. **NOTE** This is only necessary if you are wishing to conduct the post-hoc Tukey HSD analysis (which we often are).

```
circadian.aov <- aov(shift ~ treatment, data = circadian)
```

Now we can pass this object to the `TukeyHSD` function, and use the appropriate confidence level (corresponding to 1 minus alpha):

```
circadianTukey <- TukeyHSD(circadian.aov, conf.level = 0.95)
```

And let's make the output tidy:

```
circadianTukey.tidy <- circadianTukey %>%
  broom::tidy()
```

Have a look at the output:

```
circadianTukey.tidy
```

```
## # A tibble: 3 x 7
##   term      contrast    null.value estimate conf.low conf.high adj.p.value
##   <chr>     <chr>          <dbl>     <dbl>     <dbl>     <dbl>     <dbl>
## 1 treatment Knee-Control     0 -0.02696 -0.9525    0.8986    0.9970
## 2 treatment Eyes-Control    0 -1.243     -2.168    -0.3171   0.007866
## 3 treatment Eyes-Knee      0 -1.216     -2.172    -0.2598   0.01168
```

Now let's reformat this table for producing a nice output.

```
circadianTukey.tidy.table <- circadianTukey.tidy %>%
  select(-c(term, null.value)) %>% # take away these two columns
  rename(Contrast = "contrast", Difference = "estimate",
         Conf_low = "conf.low", Conf_high = "conf.high",
         Adj_Pval = "adj.p.value")
```

Let's see what we made:

```
circadianTukey.tidy.table
```

```
## # A tibble: 3 x 5
##   Contrast      Difference Conf_low Conf_high Adj_Pval
##   <chr>          <dbl>     <dbl>     <dbl>     <dbl>
## 1 Knee-Control -0.02696  -0.9525    0.8986  0.9970
## 2 Eyes-Control  -1.243    -2.168    -0.3171  0.007866
## 3 Eyes-Knee     -1.216    -2.172    -0.2598  0.01168
```

The output clearly shows the pairwise comparisons and associated P -values, **adjusted** for multiple comparisons. It also shows the difference in means, and the lower and upper 95% confidence interval for the differences.

We can see that the mean for the “Eyes” treatment group differs significantly (at $\alpha = 0.05$) from each of the other group means.

One typically shows these results visually on the same figure used to display the data (here, Figure 16.1).

16.2.8 Visualizing post-hoc test results

Here is our original figure, to which we’ll add the results of the post-hoc test:

```
circadian %>%
  ggplot(aes(x = treatment, y = shift)) +
  geom_jitter(colour = "darkgrey", size = 3, shape = 1, width = 0.1) +
  stat_summary(fun.data = mean_se, geom = "errorbar",
               colour = "black", width = 0.1,
               position = position_nudge(x = 0.15)) +
  stat_summary(fun = mean, geom = "point",
               colour = "black", size = 3,
               position=position_nudge(x = 0.15)) +
  xlab("Light treatment") +
  ylab("Shift in circadian rhythm (h)") +
  theme_classic()
```

To visualize the outcomes of the Tukey-Kramer post-hoc test, we superimpose a text letter alongside (or above) each group in the figure, such that groups sharing the same letter are not significantly different according to the Tukey-Kramer post-hoc test.

For this we use the `annotate` function, and this requires that we know the coordinates on the graph where we wish to place our annotations.

Based on our stripchart above, it looks like we need to use Y-value coordinates of around 1.1 to place our letters above each group. **It may take some trial-and-error** to figure this out.

Here’s the code, with the `annotate` function added:

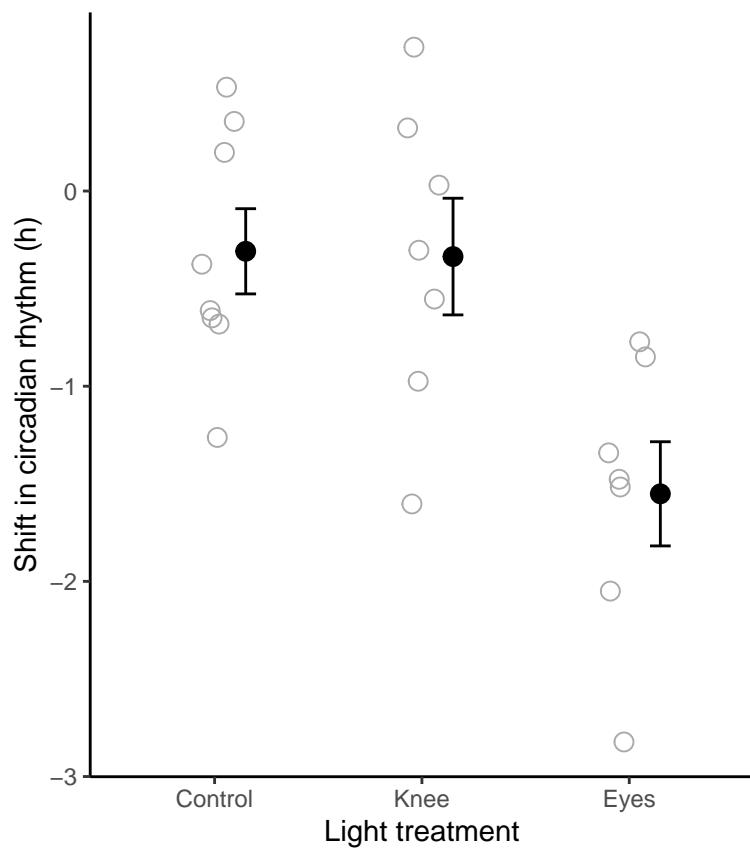


Figure 16.2: Stripchart of phase shifts in the circadian rhythm of melatonin production in 22 participants of an experiment. Solid circles denote group means, and bars +/- one standard error

```
ggplot(circadian, aes(x = treatment, y = shift)) +
  geom_jitter(colour = "darkgrey", size = 3, shape = 1, width = 0.1) +
  stat_summary(fun.data = mean_se, geom = "errorbar",
    colour = "black", width = 0.1,
    position=position_nudge(x = 0.15)) +
  stat_summary(fun = mean, geom = "point",
    colour = "black", size = 3,
    position=position_nudge(x = 0.15)) +
  ylim(c(-3.1, 1.2)) +
  annotate("text", x = 1:3, y = 1.1, label = c("a", "a", "b")) +
  labs(x = "Light treatment", y = "Shift in circadian rhythm (h)") +
  theme_classic()
```

Notice that we have added a line including the `ylim` function, so that we can specify the min and max y-axis limits, ensuring that our annotation letters fit on the graph.

NOTE: The chunk option I included to get the figure heading shown above is as follows:

```
{r fig.cap = "Stripchart showing the phase shift in the circadian rhythm of melatonin p"}
```

The “family-wise” α statement means that the Tukey-Kramer test uses our initial α level, but ensures that the probability of making at least one Type-1 error throughout the course of testing all pairs of means is no greater than the originally stated α level.

TIP: Figure 16.3 is the kind of figure that should be referenced, typically along with ANOVA table 16.2.5.1, when communicating your results of an ANOVA. For instance, with the results of the Tukey-Kramer post-hoc tests superimposed on the figure, you can not only state that the null hypothesis is rejected, but you can also state which group(s) differ from which others.

We are now able to write our truly final concluding statement, below.

1. Annotating stripcharts

Pretend that our post-hoc test showed that the “Knee” group was not different from either the control or the Eyes group. Re-do the figure above, but this time place an “ab” above the Knee group on the figure. This is how you would indicate that the control and eyes grouped differed significantly from one-another, but neither differed significantly from the Knee group.

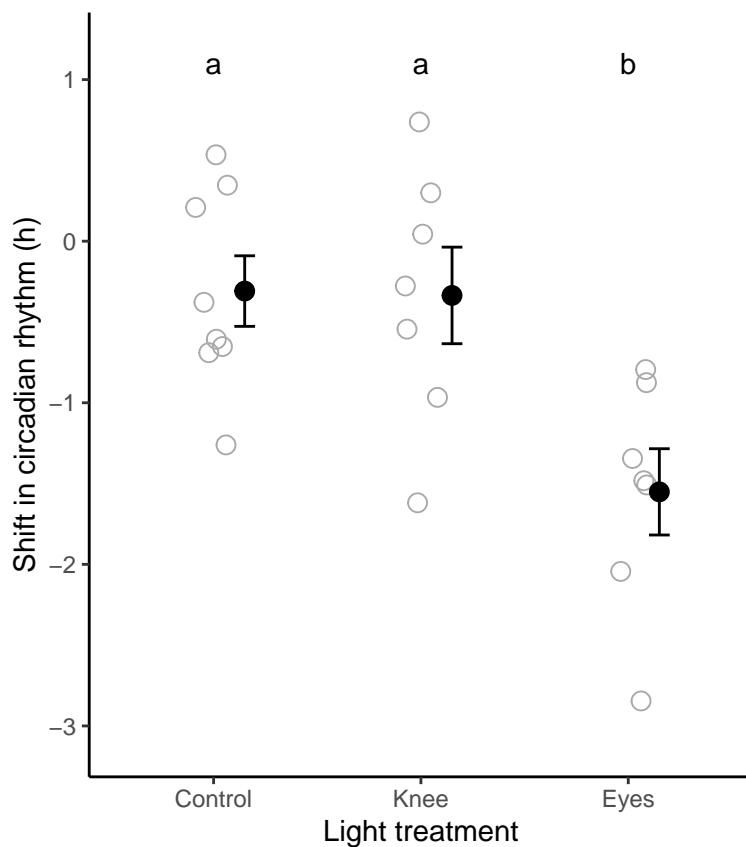


Figure 16.3: Stripchart showing the phase shift in the circadian rhythm of melatonin production in 22 experimental participants given alternative light treatments. Solid circles represent group means, and bars represent \pm one SE. Group means sharing the same letter are not significantly different according to the Tukey-Kramer post-hoc test (family-wise $\alpha = 0.05$).

16.2.9 Concluding statement

With out ANOVA table and final figure in hand, we are ready for the concluding statement:

Shifts in circadian rhythm differ significantly among treatment groups (ANOVA; Table 16.2.5.1; $R^2 = 0.43$). As shown in Figure 16.3, the mean shift among the “Eyes” subjects was significantly lower than both of the other treatment groups.

16.3 When assumptions aren’t met

If the **normal distribution assumption** is violated, and you are unable to find a transformation that works (see the Checking assumptions and data transformations tutorial), then you can try a non-parametric test.

A tutorial on non-parametric tests is under development, but will not be deployed until 2023. Consult chapter 13 in the Whitlock & Schluter text, and this website for some R examples.

In general, you might find that the “Kruskal-Wallis” rank sum test is a good non-parametric alternative to ANOVA. This can be implemented using the `kruskal.test` function:

```
?kruskal.test
```

Chapter 17

Analyzing associations between two numerical variables

Tutorial learning objectives

- Learn about using correlation analyses to test hypotheses about associations between two numerical variables
- Learn that the **Pearson correlation coefficient** measures the strength and direction of the association between two numerical variables
- Learn about the assumptions of correlation analysis
- Learn parametric and non-parametric methods for testing association between two numerical variables

17.1 Load packages and import data

Load the `tidyverse`, `skimr`, `broom`, `knitr`, and `janitor` packages:

```
library(tidyverse)
library(skimr)
library(broom)
library(knitr)
library(janitor)
```

292CHAPTER 17. ANALYZING ASSOCIATIONS BETWEEN TWO NUMERICAL VARIABLES

We'll use the "wolf.csv" and "trick.csv" datasets (discussed in examples 16.2 and 16.5 in the text, respectively).

```
wolf <- read_csv("https://raw.githubusercontent.com/ubco-biology/BIOL202/main/data/wolf.csv")

## Rows: 24 Columns: 2
## -- Column specification -----
## Delimiter: ","
## dbl (2): inbreedCoef, nPups
##
## 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.

trick <- read_csv("https://raw.githubusercontent.com/ubco-biology/BIOL202/main/data/trick.csv")

## Rows: 21 Columns: 2
## -- Column specification -----
## Delimiter: ","
## dbl (2): years, impressivenessScore
##
## 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.
```

The `wolf` dataset includes inbreeding coefficients for wolf pairs, along with the number of the pairs' pups surviving the first winter.

Explore the data:

```
wolf %>% skim_without_charts()
```

(#tab:corr_seedata1) Data summary

Name

Piped data

Number of rows

24

Number of columns

2

Column type frequency:

numeric

2

Group variables

None

Variable type: numeric

skim_variable

n_missing

complete_rate

mean

sd

p0

p25

p50

p75

p100

inbreedCoef

0

1

0.23

0.10

0

0.19

0.24

0.30

0.4

nPups

0

1

3.96

1.88

1

3.00

3.00

5.25

8.0

We see that there are 24 observations for each of the two variables, and no missing values. If there WERE missing values, be sure to report the correct sample size in your results!

Now let's explore the `trick` dataset:

```
trick %>% skim_without_charts()
```

(#tab:corr_seedata2) Data summary

Name

Piped data

Number of rows

21

Number of columns

2

Column type frequency:

numeric

2

Group variables

None

Variable type: numeric

skim_variable

n_missing

complete_rate

mean

sd

p0

p25

```
p50  
p75  
p100  
years  
0  
1  
27.29  
15.21  
2  
17  
28  
39  
50  
impressivenessScore  
0  
1  
3.43  
1.36  
1  
2  
4  
4  
5
```

It includes 21 observations, no missing values, and two integer variables: “years”, and “impressivenessScore”. Reading example 16.5 from the text, we see that the latter variable is a form of ranking variable.

17.2 Pearson correlation analysis

It is commonplace in biology to wish to quantify the strength and direction of a linear association between two numerical variables.

For example, in an earlier tutorial we visualized the association between bill depth and bill length among Adelie penguins, using the “penguins” dataset.

Here we learn how to quantify the strength and direction of this type of association by calculating the **Pearson correlation coefficient**.

When drawing inferences about associations between numerical variables in a population, the true correlation coefficient is referred to as “rho” or ρ .

The sample-based correlation coefficient, which we use to estimate ρ , is referred to as r :

$$r = \frac{\sum (X_i - \bar{X})(Y_i - \bar{Y})}{\sqrt{\sum (X_i - \bar{X})^2} \sqrt{\sum (Y_i - \bar{Y})^2}}$$

In the calculation of r it does not matter which variable is treated as the X and which as the Y . However, in many instances there may good reason to choose which serves as the X (explanatory) and which as the Y (response).

17.2.1 Hypothesis statements

As a refresher, first consult the steps to hypothesis testing.

Researchers were interested in whether inbreeding coefficients of the wolf litters were associated with the number of pups surviving their first winter.

Both variables are numerical, and so the first choice is to conduct a **Pearson correlation analysis**. This analysis yields a sample-based measure called Pearson’s correlation coefficient, or r . This provides an estimate of ρ - the true correlation between the two variables in the population. The absolute magnitude of r (and ρ) reflects the strength of the linear association between two numeric variables in the *population*, and the sign of the coefficient indicates the direction of the association.

The hypothesis statements should be framed in the context of the question, and should include the hypothesized value of the population parameter.

H₀: Inbreeding coefficients are not associated with the number of pups surviving the first winter ($\rho = 0$). **H_A**: Inbreeding coefficients are associated with the number of pups surviving the first winter ($\rho \neq 0$).

We’ll set $\alpha = 0.05$.

17.2.2 Visualize the data

We learned in an earlier tutorial that the best way to visualize an association between two numeric variables is with a scatterplot, and that we can create a scatterplot using the `geom_point` function from the `ggplot2` package:

```
wolf %>%
  ggplot(aes(x = nPups, y = inbreedCoef)) +
  geom_point(shape = 1) +
  xlab("Number of pups") +
  ylab("Inbreeding coefficient") +
  theme_bw()
```

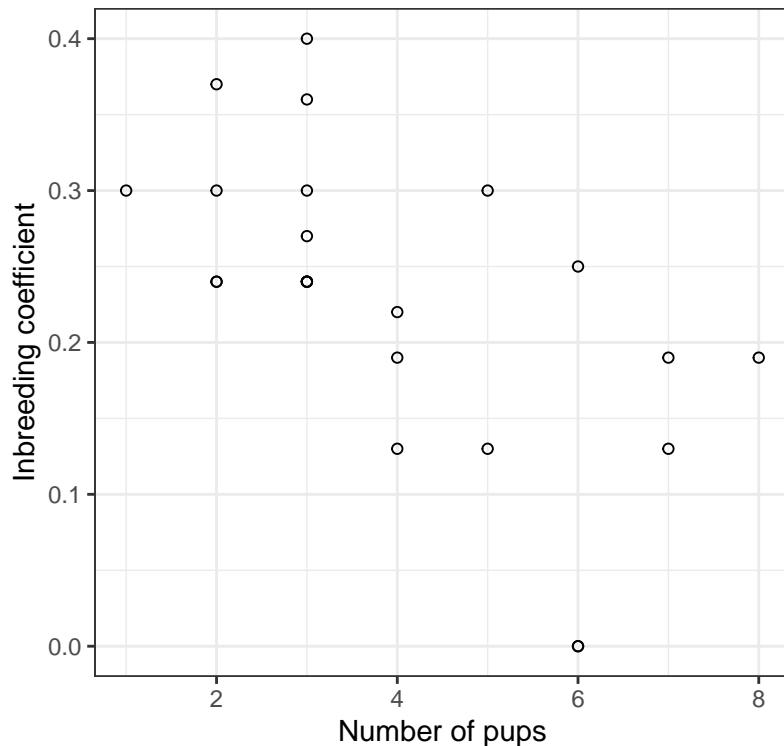


Figure 17.1: The association between inbreeding coefficient and number of surviving wolf pups ($n = 24$).

We notice that there doesn't appear to be the correct number of points (24) in the scatterplot, so there must be some overlapping.

To remedy this, we use the `geom_jitter` function instead of the `geom_point` function.

```
wolf %>%
  ggplot(aes(x = nPups, y = inbreedCoef)) +
  geom_jitter(shape = 1) +
  xlab("Number of pups") +
```

```
ylab("Inbreeding coefficient") +
  theme_bw()
```

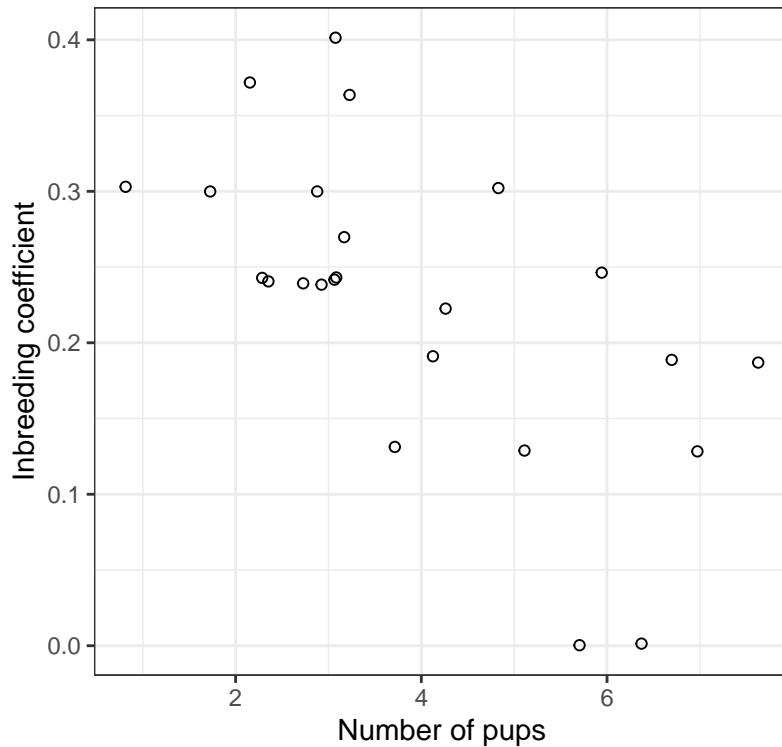


Figure 17.2: The association between inbreeding coefficient and number of surviving wolf pups ($n = 24$). Values have been jittered slightly to improve legibility.

That's better!

Interpreting a scatterplot

In an earlier tutorial, we learned how to properly interpret a scatterplot, and **what information should to include in your interpretation**. Be sure to consult that tutorial.

We see in Figure 17.2 that the association between the inbreeding coefficient and number of surviving pups is negative, linear, and moderately strong. There are no apparent outliers to the association.

17.2.3 Assumptions of correlation analysis

Correlation analysis assumes that:

- the sample of individuals is a random sample from the population
 - the measurements have a **bivariate normal distribution**, which includes the following properties:
 - the relationship between the two variables (X and Y) is linear
 - the cloud of points in a scatterplot of X and Y has a circular or elliptical shape
 - the frequency distributions of X and Y separately are normal
-

Checking the assumptions of correlation analysis

The assumptions are most easily checked using the scatterplot of X and Y .

What to look for as potential problems in the scatterplot:

- a “funnel” shape
- outliers to the general trend
- non-linear association

If any of these patterns are evident, then one should opt for a non-parametric analysis (see below).

(See Figure 16.3-2 in the text for examples of non-conforming scatterplots)

Based on Figure 17.2, there doesn't seem to be any indications that the assumptions are not met, so we'll proceed with testing the null hypothesis.

Be careful with “count” type variables such as “number of pups”, as these may not adhere to the “bivariate normality” assumption. If the variable is restricted to a limited range of possible counts, say zero to 5 or 6, then the association should probably be analyzed using a non-parametric test (see below). The variable “number of pups” in this example is borderline OK...

17.2.4 Conduct the correlation analysis

Conducting a correlation analysis is done using the `cor.test` function that comes with R.

This function does not produce “tidy” output, so we’ll make use of the `tidy` function from the `broom` package to tidy up the correlation output (like we did in for ANOVA output).

Notice that the function expects the “x” and “y” variables as separate arguments.

And here’s how to implement it. First run the `cor.test` function, and notice we provide the “x” and “y” variables

```
wolf.cor <- cor.test(x = wolf$inbreedCoef, y = wolf$nPups,
                      method = "pearson", conf.level = 0.95,
                      alternative = "two.sided")
```

Let’s have a look at the untidy output:

```
wolf.cor
```

```
## 
## Pearson's product-moment correlation
##
## data: wolf$inbreedCoef and wolf$nPups
## t = -3.5893, df = 22, p-value = 0.001633
## alternative hypothesis: true correlation is not equal to 0
## 95 percent confidence interval:
## -0.8120418 -0.2706791
## sample estimates:
##      cor
## -0.6077184
```

Now let’s tidy it up and have a look at the resulting tidy output:

```
wolf.cor.tidy <- wolf.cor %>%
  broom::tidy()
```

Show the output:

```
wolf.cor.tidy
```

```
## # A tibble: 1 x 8
##   estimate statistic p.value parameter conf.low conf.high method alternative
##     <dbl>      <dbl>    <dbl>      <int>     <dbl>     <dbl> <chr>      <chr>
## 1 -0.6077    -3.589  0.001633       22    -0.8120   -0.2707 Pearson'~ two.sided
```

- The “estimate” value represents the value of Pearson’s correlation coefficient r
- The “statistic” value is actually the value for t , which is used to test the significance of r
- The “p.value” associated with the observed value of t
- The “parameter” value is, strangely, referring to the degrees of freedom for the test, $df = n - 2$
- The output also includes the confidence interval for r (“conf.low” and “conf.high”)
- The “method” refers to the type of test conducted
- The “alternative” indicates whether the alternative hypothesis was one- or two-sided (the latter is the default)

Despite the reporting of the t test statistic, **we do not report t in our concluding statement** (see below).

17.2.5 Concluding statement

It is advisable to always refer to a scatterplot when authoring a concluding statement for correlation analysis.

Let’s re-do the scatterplot here.

```
wolf %>%
  ggplot(aes(x = nPups, y = inbreedCoef)) +
  geom_jitter(shape = 1) +
  xlab("Number of pups") +
  ylab("Inbreeding coefficient") +
  theme_bw()
```

Here is an example of a good concluding statement:

Litter size is significantly negatively correlated with the inbreeding coefficient of the parents (Figure 17.3; Pearson $r = -0.61$; 95% confidence limits: -0.812 , -0.271 ; $df = 22$; $P = 0.002$).

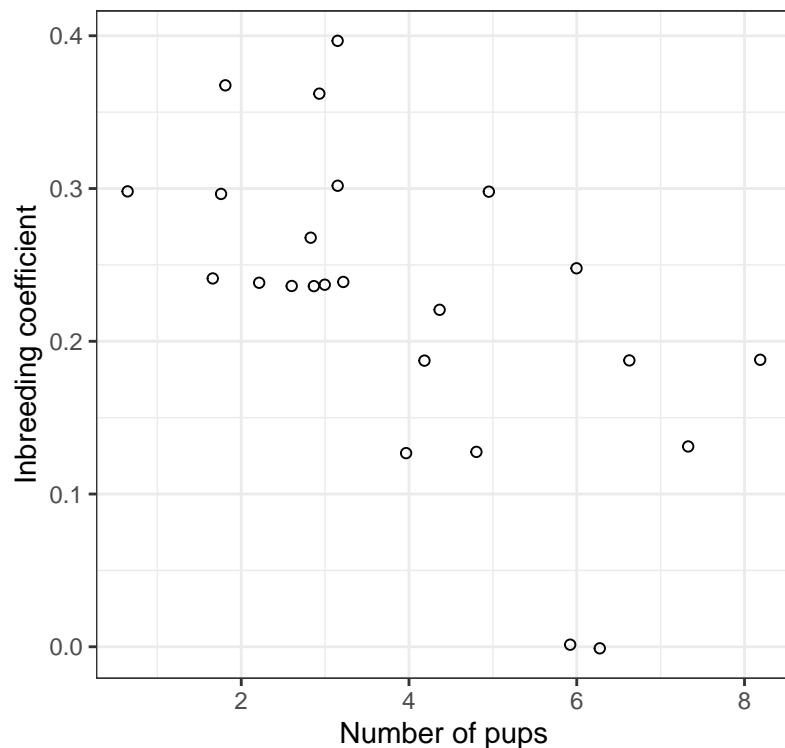


Figure 17.3: The association between inbreeding coefficient and number of surviving wolf pups ($n = 24$). Values have been jittered slightly to improve legibility.

Tip Remember to double-check if you had any missing values in your dataset, do that you don't report the wrong sample size in your figure caption and / or concluding statement.

1. Using the `penguins` dataset that loads with the `palmerpenguins` package, test the null hypothesis that there is no linear association between bill length and bill depth among Gentoo penguins. **HINT:** Before using the `cor.test` function, you'll first need to create a new tibble that includes only the "Gentoo" species data.

17.3 Rank correlation (Spearman's correlation)

If the assumption of bivariate normality is not met for Pearson correlation analysis, then we use Spearman rank correlation.

For example, if one or both of your numerical variables (X and / or Y) is actually a discrete, ordinal numerical variable to begin with (e.g. an attractiveness score that ranges from 1 to 5), then this automatically necessitates the use of Spearman rank correlation, because it does not meet the assumptions of bivariate normality. (This is why one needs to be careful with count data).

We'll use the `trick` dataset for this example, and the data are described in example 16.5 in the text.

17.3.1 Hypothesis statements

The null and alternative hypotheses are:

H₀: There is no linear correlation between the ranks of the impressiveness scores and time elapsed until the writing of the description ($\rho_S = 0$).

H_A: There is a linear correlation between the ranks of the impressiveness scores and time elapsed until the writing of the description ($\rho_S \neq 0$).

Let's use $\alpha = 0.05$.

As shown in the hypothesis statements above, we are interested in ρ_S , which is the true correlation between the *ranks* of the variables in the population. We estimate this using r_S , Spearman's correlation coefficient.

Unlike in the Pearson correlation case (above), which uses t as a test statistic, the rank correlation analysis simply uses the actual Spearman correlation coefficient as the test statistic.

17.3.2 Visualize the data

Let's visualize the association, again using the `geom_jitter` function to help see overlapping values:

```
trick %>%
  ggplot(aes(x = years, y = impressivenessScore)) +
  geom_jitter(shape = 1) +
  xlab("Years elapsed") +
  ylab("Impressiveness score") +
  theme_bw()
```

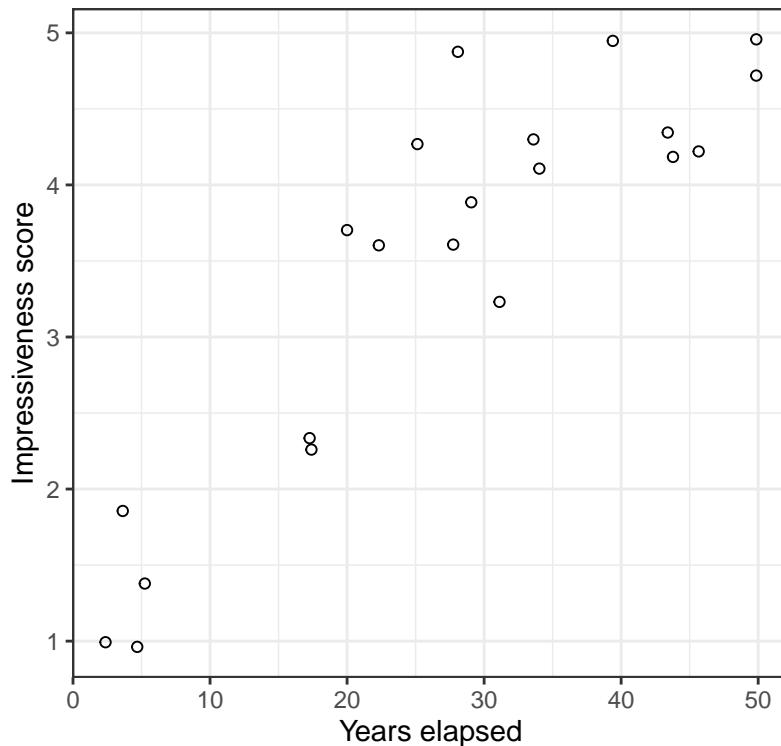


Figure 17.4: Scatterplot of the impressiveness of written accounts of the Indian rope trick by firsthand observers and the number of years elapsed between witnessing the event and writing the account ($n = 21$). Values have jittered slightly to improve legibility.

In Figure 17.4 we see a positive and moderately strong association between the impressiveness of written accounts of the Indian rope

trick by firsthand observers and the number of years elapsed between witnessing the event and writing the account.

17.3.3 Assumptions of Spearman rank correlation

Spearman rank correlation assumes that:

- the observations are a random sample from the population
- the relationship between the two variables is monotonic; in other words it assumes that the relationship between the **ranks** of the two numerical variables is linear.

Checking assumptions

As in the Pearson correlation analysis, we use the scatterplot to check the assumptions.

As shown in Figure 17.4, there is a monotonic relationship between the two variables.

17.3.4 Conduct the test

We use the same `cor.test` function to conduct the test, but change the “method” argument accordingly:

```
trick.cor <- cor.test(x = trick$years, y = trick$impressivenessScore,
                      method = "spearman", conf.level = 0.95,
                      alternative = "two.sided")
```

You may get a warning message, simply saying that it can't compute exact P -values when there are ties in the ranked data. Don't worry about this.

```
trick.cor.tidy <- trick.cor %>%
  broom::tidy()
trick.cor.tidy
```

```
## # A tibble: 1 x 5
##   estimate statistic    p.value method      alternative
##       <dbl>     <dbl>      <dbl> <chr>        <chr>
## 1    0.7843    332.1 0.00002571 Spearman's rank correlation rho two.sided
```

- The “estimate” value represents the value of Spearman’s correlation coefficient r_S ; this is the value you report.
- The “statistic” value is **NOT NEEDED** so ignore
- The “p.value” associated with the observed Spearman’s correlation coefficient
- The “method” refers to the type of test conducted
- The “alternative” indicates whether the alternative hypothesis was one- or two-sided (the latter is the default)

There is no confidence interval reported with Spearman correlation analysis, so there is no need to report one in the concluding statement for a rank correlation. Nor is the degrees of freedom reported, so be sure to have figured out the appropriate degrees of freedom (or sample size “n”) to report in your concluding statement.

17.3.5 Concluding statement

As in the preceding Pearson correlation example, we can refer to the Figure in the parentheses of our concluding statement. Note also that we report n rather than degrees of freedom.

```
trick %>%
  ggplot(aes(x = years, y = impressivenessScore)) +
  geom_jitter(shape = 1) +
  xlab("Years elapsed") +
  ylab("Impressiveness score") +
  theme_bw()
```

Concluding statement:

The rank of impressiveness scores of written accounts of the Indian rope trick by firsthand observers is significantly positively correlated with the rank of number of years elapsed between witnessing the event and writing the account (Figure 17.5; Spearman $r_S = 0.78$; $n = 21$; $P < 0.001$).

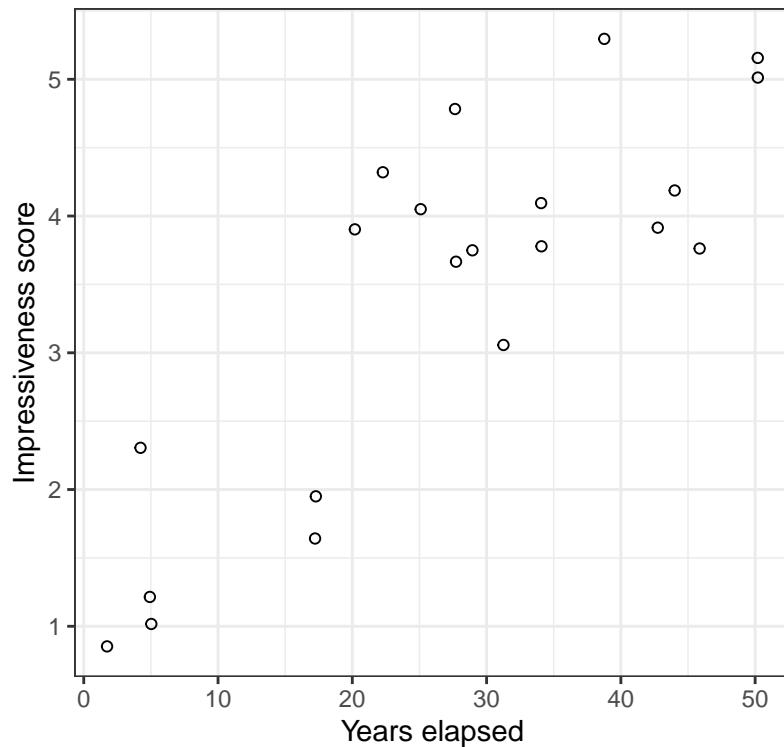


Figure 17.5: Scatterplot of the impressiveness of written accounts of the Indian rope trick by firsthand observers and the number of years elapsed between witnessing the event and writing the account ($n = 21$). Values have jittered slightly to improve legibility.

Chapter 18

Least-squares linear regression

Tutorial learning objectives

- Learn about using least-squares linear regression (“LSR”, also called “Model-1 regression”) to model the relationship between two numeric variables, and to make predictions
- Learn the assumptions of LSR, and how to test whether these assumptions are met
- Learn that the implied statistical null hypothesis for a LSR is that the slope of the relationship is equal to zero
- Learn how to interpret regression analysis output
- Learn about the “coefficient of determination”, R^2 , represents the fraction of variation in the response variable that is accounted for by the regression model
- Learn how to make predictions using LRS
- Learn how to implement and report the findings of LSR

18.1 Load packages and import data

Load the `tidyverse`, `skimr`, `broom`, and `knitr`.

```
library(tidyverse)
library(skimr)
library(broom)
library(knitr)
```

We'll use the "plantbiomass" dataset:

```
plantbiomass <- read_csv("https://raw.githubusercontent.com/ubco-biology/BIOL202/main/datasets/plantbiomass.csv")
```

```
## Rows: 161 Columns: 2
## -- Column specification -----
## Delimiter: ","
## dbl (2): nSpecies, biomassStability
##
## 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.
```

The `plantbiomass` dataset (Example 17.3 in the text book) includes data describing plant biomass measured after 10 years within each of 5 experimental “species richness” treatments, wherein plants were grown in groups of 1, 2, 4, 8, or 16 species. The treatment variable is called `nSpecies`, and the response variable `biomassStability` is a measure of ecosystem stability. The research hypothesis was that increasing diversity (species richness) would lead to increased ecosystem stability.

Below is a visualization of the data.

When one visualizes the data, one might ask why an ANOVA isn't the analysis method of choice. If we were simply interested in testing the null hypothesis that “there is no difference in mean ecosystem stability among the species richness treatment groups”, then an ANOVA could be used, and we would simply treat the “species richness” variable as a categorical variable. However, here we are interested not simply in testing for differences among treatment groups, but more specifically in quantifying if and how ecosystem stability varies with variation in species richness, AND if so, whether we can reliably predict ecosystem stability based on species richness. For this, we need to construct a regression model.

Let's have a look at the data:

```
plantbiomass %>% skim_without_charts()
```

```
(#tab:view_data_lsr)Data summary
```

Name

Piped data

Number of rows

161

Number of columns

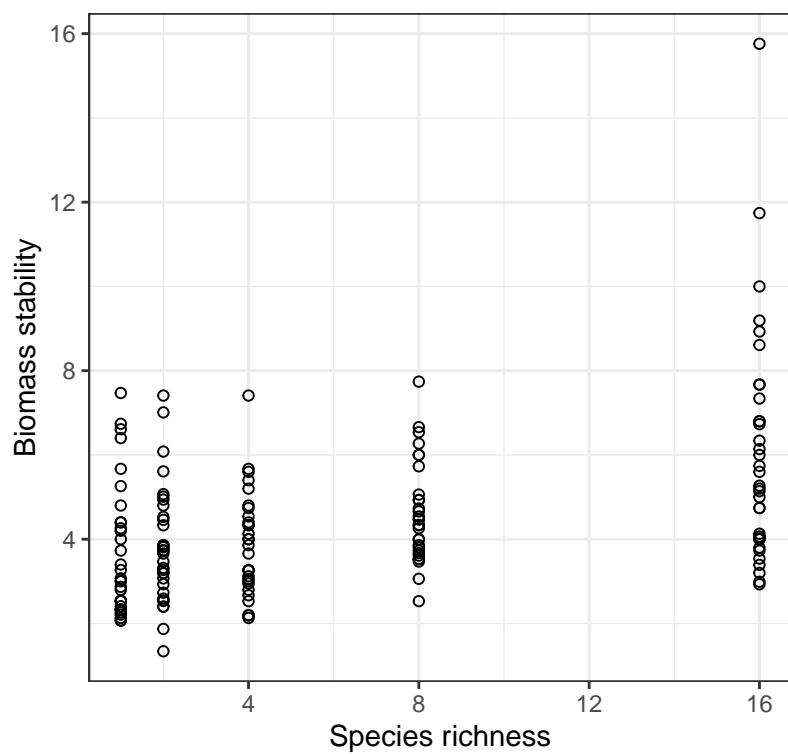


Figure 18.1: Stability of biomass production over 10 years in 161 plots and the initial number of plant species assigned to plots.

2

Column type frequency:

numeric

2

Group variables

None

Variable type: numeric

skim_variable

n_missing

complete_rate

mean

sd

p0

p25

p50

p75

p100

nSpecies

0

1

6.32

5.64

1.00

2.00

4

8.0

16.00

biomassStability

0

1
4.42
1.95
1.34
3.07
4
5.2
15.76

We see that both variables are numeric, and that there are 161 observations overall.

18.2 Least-squares regression analysis

When we are only interested in the strength of a linear association between two numerical variables, we use a correlation analysis.

When we are interest in making *predictions*, we use regression analysis.

Often in the literature you see authors reporting regression results when in fact there was no rationale for using regression; a correlation analysis would have been more appropriate.

The two analyses are mathematically related.

Regression analysis is extremely common in biology, and unfortunately it is also common to see incorrect implementation of regression analysis. In particular, one often sees scatterplots that clearly reveal violations of the assumptions of regression analysis (see below).

Failure to appropriately check assumptions can lead to misleading and incorrect conclusions

18.2.1 Equation of a line and “least-squares line”

Recall from your high-school training that the equation for a straight line is:

$$Y = a + bX$$

In regression analysis, the “least-squares line” is the line for which the sum of all the *squared deviations in Y* is smallest.

In a regression context, a is the Y-intercept and b is the slope of the regression line.

Regression analysis falls under the heading of **inferential statistics**, which means we use it to draw inferences about a linear relationship between Y and X within a *population* of interest. So in the actual population, the true least-squares line is represented as follows:

$$Y = \alpha + \beta X$$

In practice, we estimate the “parameters” α and β using a random sample from the population, and by calculating a and b using regression analysis.

The α in the regression equation has no relation to the α that sets the significance level for a test!

The equation for the true least-squares line shown above uses the equation shown in the course text book. However, an alternative way to write the equation is as follows, and includes an error term “ ϵ ”:

$$Y = \beta_0 + \beta_1 x + \epsilon$$

18.2.2 Hypothesis testing or prediction?

In general, there are two common scenarios in which least-squares regression analysis is appropriate.

- We wish to know whether some numeric variable Y can be reliably predicted using a regression model of Y on some predictor X . For example, “Can vehicle mileage be reliably predicted by the weight of the vehicle?”
- We wish to test a specific hypothesis about the slope of the regression model of Y on X . For example, across species, basal metabolic rate (BMR) relates to body mass (M) according to $BMR = \alpha M^\beta$. Here, α is a constant. If we log-transform both sides of the equation, we get the following relationship, for which the parameters can be estimated via linear regression analysis:

$$\log(BMR) = \log(\alpha) + \beta \cdot \log(M)$$

Some biological hypotheses propose that the slope β should equal $2/3$, whereas others propose $3/4$. This represents an example of when one would propose a specific, hypothesized value for β .

In the most common application of regression analysis we don't propose a specific value for the slope β . Instead, we proceed with the "default" and **unstated** null and alternative hypotheses as follows:

$$\mathbf{H}_0: \beta = 0$$
$$\mathbf{H}_A: \beta \neq 0$$

In this case, **there is no need to explicitly state these hypotheses for regression analysis.**

You must, however, know what is being tested: the null hypothesis that the slope is equal to zero.

18.2.3 Steps to conducting regression analysis

In regression analysis one must implement the analysis before one can check the assumptions. Thus, the order of operations is a bit different from that of other tests.

Follow these steps when conducting regression analysis:

- Set an α level (default is 0.05)
- Provide an appropriate figure, including figure caption, to visualize the association between the two numeric variables
- Provide a sentence or two interpreting your figure, and this may inform your concluding statement
- Conduct the regression analysis, save the output, and use the **residuals** from the analysis to check assumptions
- Assumptions
 - state the assumptions of the test
 - use appropriate figures and / or tests to check whether the assumptions of the statistical test are met
 - transform data to meet assumptions if required
 - if assumptions can't be met (e.g. after transformation), use alternative methods (**these alternative methods are not covered in this course**)
 - if data transformation is required, then do the transformation and provide another appropriate figure, including figure caption, to visualize the transformed data, and a sentence or two interpreting this new figure

- Calculate the confidence interval for the slope and the R^2 value, report these in concluding statement
 - If the slope of the least-squares regression line differs significantly from zero (i.e. the regression is significant), provide a scatterplot that includes the regression line and **confidence bands**, with an appropriate figure caption
 - Use the output from the regression analysis (above) draw an appropriate conclusion and communicate it clearly
-

18.2.4 State question and set the α level

The `plantbiomass` dataset (Example 17.3 in the text book) includes data describing plant biomass measured after 10 years within each of 5 experimental “species richness” treatments, wherein plants were grown in groups of 1, 2, 4, 8, or 16 species. The treatment variable is called `nSpecies`, and the response variable `biomassStability` is a measure of ecosystem stability. The research hypothesis was that increasing diversity (species richness) would lead to increased ecosystem stability. If the data support this hypothesis, it would be valuable to be able to predict ecosystem stability based on species richness.

We will use the conventional α level of 0.05.

18.2.5 Visualize the data

Using the plant biomass data, let’s evaluate whether ecosystem stability can be reliably predicted from plant species richness.

We learned in an earlier tutorial that the best way to visualize an association between two numeric variables is with a scatterplot, and that we can create a scatterplot using the `ggplot` and `geom_point` functions:

```
plantbiomass %>%
  ggplot(aes(x = nSpecies, y = biomassStability)) +
  geom_point(shape = 1) +
  xlab("Species richness") +
  ylab("Biomass stability") +
  theme_bw()
```

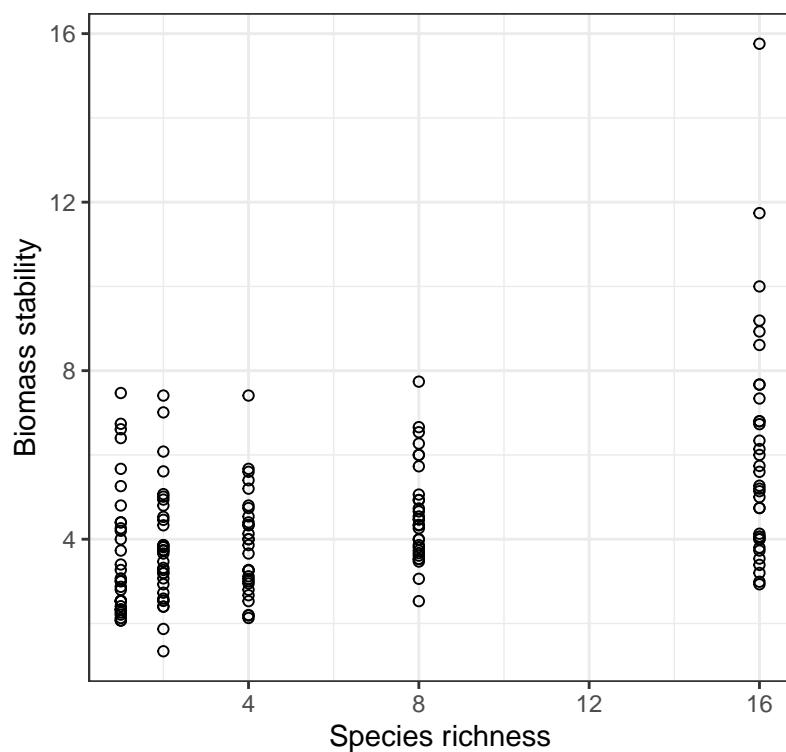


Figure 18.2: Stability of biomass production over 10 years in 161 plots and the initial number of plant species assigned to plots.

Note that in the above figure we do not report units for the response variable, because in this example the variable does not have units. But often you do need to provide units!

This figure that you first produce for visualizing the data may or may not be necessary to present with your results. If your least-squares regression does end up being statistically significant, i.e. having a slope that differs significantly from zero, then you'll create a new figure to reference in your concluding statement (see below). If your regression analysis is non-significant, then you should present and refer to this simple scatterplot (which does not include a best-fit regression line).

18.2.6 Interpreting a scatterplot

In an earlier tutorial, we learned how to properly interpret a scatterplot, and **what information should to include in your interpretation**. Be sure to consult that tutorial.

We see in Figure 18.2 that there is a weak, positive, and somewhat linear association between biomass stability and Species richness. There appears to be increasing spread of the response variable with increasing values of the predictor (x) variable, and perhaps outliers in the treatment group with greatest species richness. We should keep this in mind.

18.2.7 Checking assumptions of regression analysis

Regression analysis assumes that:

- the true relationship between X and Y is linear
- for every value of X the corresponding values of Y are normally distributed
- the variance of Y -values is the same at all values of X
- at each value of X , the Y measurements represent a random sample from the population of possible Y values

Strangely, we must implement the regression analysis before checking the assumptions. This is because we check the assumptions using the **residuals** from the regression analysis.

Let's implement the regression analysis and be sure to store the output for future use.

The function to use is the `lm` function, which we saw in a previous tutorial, but here our predictor (independent) variable is a numeric variable rather than a categorical variable.

Let's run the regression analysis and assign the output to an object as follows:

```
biomass.lm <- lm(biomassStability ~ nSpecies, data = plantbiomass)
```

Let's have a look at the untidy output:

```
biomass.lm
```

```
## 
## Call:
## lm(formula = biomassStability ~ nSpecies, data = plantbiomass)
## 
## Coefficients:
## (Intercept)    nSpecies
##           3.4081      0.1605
```

We'll learn a bit later how to tidy up this output...

Before we do anything else (e.g. explore the results of the regression), we need to first check the assumptions!

To get the residuals, which are required to check the assumptions, you first need to run the `augment` function from the `broom` package on the untidy regression object, as this will allow us to easily access the residuals.

```
biomass.lm.augment <- biomass.lm %>%
  broom::augment()
```

This produces a tibble, in which one of the variables, called “.resid”, houses the residuals that we're after.

Let's have a look:

```
biomass.lm.augment
```

```
## # A tibble: 161 x 8
##   biomassStability nSpecies .fitted .resid     .hat .sigma .cooksD .std.resid
##             <dbl>     <dbl>     <dbl>     <dbl>     <dbl>     <dbl>     <dbl>
## 1           7.47      1     3.569  3.901  0.01177  1.707  0.03063     2.268
```

```

## 2      6.74      1  3.569 3.171  0.01177  1.717 0.02024 1.844
## 3      6.61      1  3.569 3.041  0.01177  1.718 0.01862 1.768
## 4      6.4       1  3.569 2.831  0.01177  1.721 0.01613 1.646
## 5      5.67      1  3.569 2.101  0.01177  1.727 0.008887 1.222
## 6      5.26      1  3.569 1.691  0.01177  1.730 0.005758 0.9835
## 7      4.8       1  3.569 1.231  0.01177  1.733 0.003052 0.7160
## 8      4.4       1  3.569 0.8314 0.01177  1.734 0.001391 0.4834
## 9      4.4       1  3.569 0.8314 0.01177  1.734 0.001391 0.4834
## 10     4.26      1  3.569 0.6914 0.01177  1.735 0.0009622 0.4020
## # i 151 more rows

```

“Residuals” represent the vertical difference between each observed value of Y (here, biomass stability) the least-squares line (the predicted value of Y , “.fitted” in the tibble above) at each value of X .

Don’t worry about the other variables in the tibble for now.

Let’s revisit the assumptions:

- the true relationship between X and Y is linear
- for every value of X the corresponding values of Y are normally distributed
- the variance of Y -values is the same at all values of X
- at each value of X , the Y measurements represent a random sample from the population of possible Y values

With respect to the last assumption regarding random sampling, we can assume that it is met for the plant experiment example. In experiments, the key experimental design feature is that individuals (here, plant seedlings) are randomly assigned to treatments, and that the individuals themselves were randomly drawn from the population prior to randomizing to experimental treatments. If your study is observational, then it is key that the individuals upon which measurements were made were randomly drawn from the population.

For the first three assumptions, we start by looking at the original scatterplot 18.2, but this time we add what’s called a “locally weighted smoother” line, using the `ggplot2` function `geom_smooth`:

```

plantbiomass %>%
  ggplot(aes(x = nSpecies, y = biomassStability)) +
  geom_point(shape = 1) +
  geom_smooth(method = "loess", se = FALSE, formula = y ~ x) +
  xlab("Species richness") +
  ylab("Biomass stability") +
  theme_bw()

```

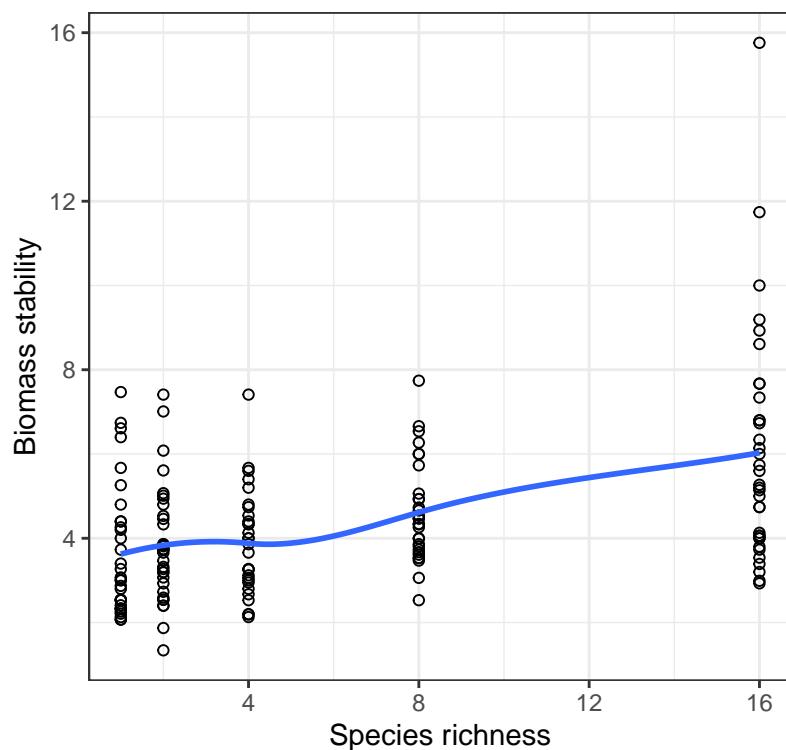


Figure 18.3: Stability of biomass production over 10 years in 161 plots and the initial number of plant species assigned to plots. Also shown is a locally weighted smoothing line.

Note the arguments provided for the `geom_smooth` function.

The smoothing line will make it easier to assess whether the relationship is linear or not. Sometimes it is obvious from the scatterplot that the association is non-linear (curved) (violating the first assumption), and sometimes it is obvious that the spread of the Y values appears to change systematically (e.g. get bigger) with increasing values of X (violating the third assumption).

The second assumption is better assessed using the residuals (see below).

In either case, one can try transforming the response variable, re-running the regression analysis using this transformed variable as the new “ Y ” variable, then checking the assumptions again.

18.2.7.1 Outliers

Outliers will cause the first three regression assumptions to be violated.

Viewing the scatterplot, keep an eye out for **outliers**, i.e. observations that look very unusual relative to the other observations. In regression, observations that are far away from the general linear association can have undue influence over the best-fit line. This sounds a bit fuzzy to judge, and it is. There are more formal approaches to evaluating “outliers”, but for now use this approach: If your eye is drawn to an observation because it seems unusual compared to the rest, then you’re probably on to something. In this case, the most transparent and thorough approach is to report your regression results first using the full dataset, and then second, excluding the outlier observations from the analysis.

Two other types of graphs are used to help evaluate assumptions.

We can check the normality of residuals assumption with a normal quantile plot, which you’ve seen previously.

For this, we use the “augmented” output we created above.

```
biomass.lm.augment %>%
  ggplot(aes(sample = .resid)) +
  stat_qq(shape = 1, size = 2) +
  stat_qq_line() +
  theme_bw()
```

Figure 18.4 shows that the residuals don’t really fall consistently near the line in the normal quantile plot; there is curvature, and increasing values tend to fall further from the line. This suggests the need to **log-transform the data**.

Before trying a transformation, let’s also check the assumptions that (i) the variance of Y is the same at all values of X , and (ii) the assumption that the

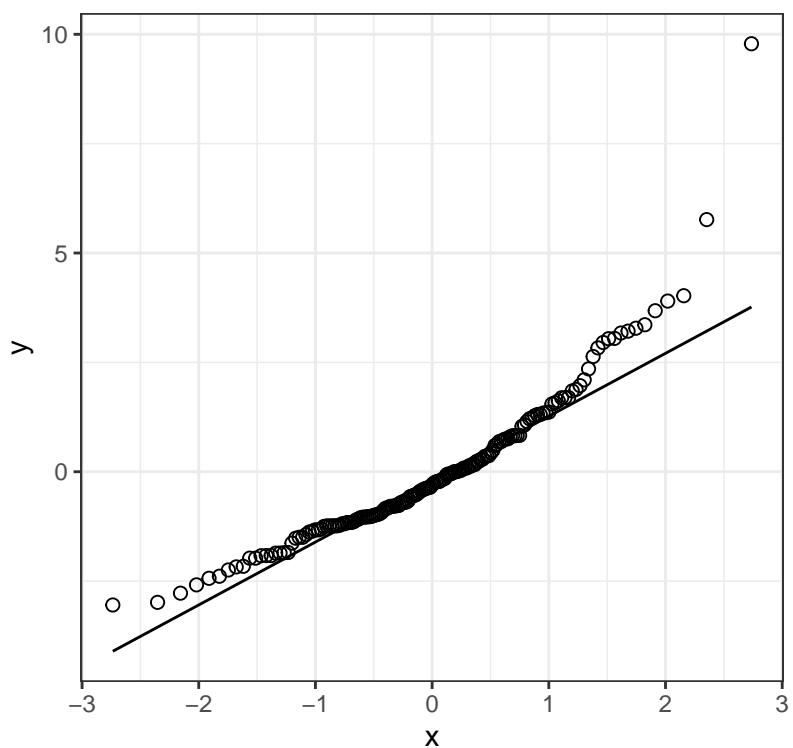


Figure 18.4: Normal quantile plot of the residuals from a regression of biomass stability on Species richness for 161 plots

association is linear. To do this, we plot the residuals against the original “ X ” variable, which here is the number of species initially used in the treatment. Note that we’ll add a horizontal line (“segment”) at zero for reference:

```
biomass.lm.augment %>%
  ggplot(aes(x = nSpecies, y = .resid)) +
  geom_point(shape = 1) +
  geom_abline(slope = 0, linetype = "dashed") +
  xlab("Species richness") +
  ylab("Residual") +
  theme_bw()
```

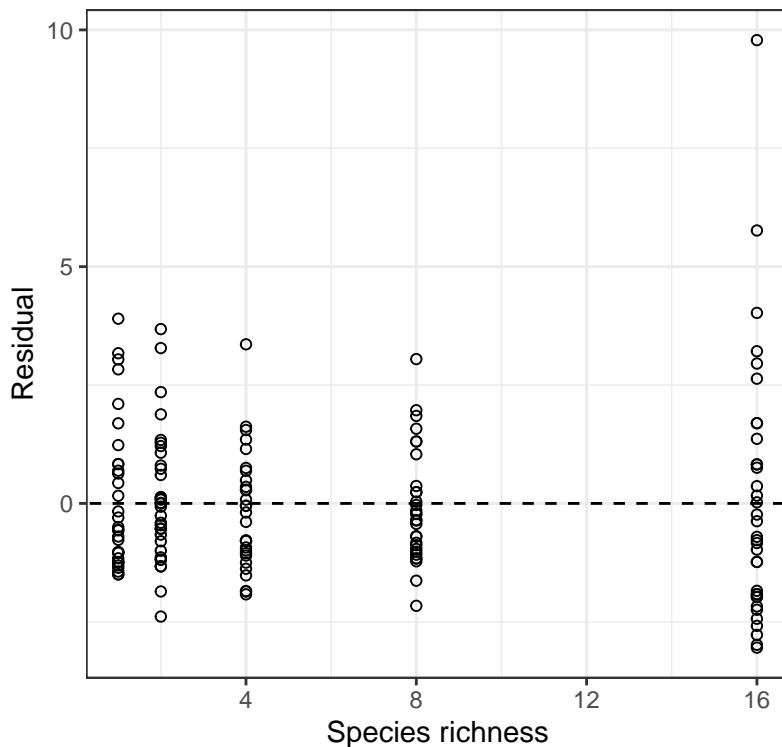


Figure 18.5: Residual plot from a regression of biomass stability on Species richness for 161 plots.

Here we are watching out for:

- a “funnel” shape, or substantial differences in the spread of the residuals at each value of X

- outliers to the general trend
- a curved pattern

Figure 18.5 shows that the variance in biomass stability is greatest within the largest species richness treatment. This again suggests the need for a log transformation.

REMINDER: If ever you see a clear outlier when checking assumptions, then a safe approach is to report the regression with and without the outlier point(s) included. If the exclusion of the outlier changes your conclusions, then you should make this clear.

18.2.8 Residual plots when you have missing values

If there are missing values in either the X or Y variables used in the regression, then simply `filter` those NA rows out before plotting the residuals.

For example, imagine one of the rows in our biomass dataset included a missing value (“NA”) in the “nSpecies” variable, then we could use this code for producing the residual plot:

```
biomass.lm.augment %>%
  filter(!is.na(nSpecies)) %>%
  ggplot(aes(x = nSpecies, y = .resid)) +
  geom_point(shape = 1) +
  geom_abline(slope = 0, linetype = "dashed") +
  xlab("Species richness") +
  ylab("Residual") +
  theme_bw()
```

18.2.9 Transform the data

Let’s log-transform the response variable, creating a new variable `log.biomass` using the `mutate` function from the `dplyr` package:

```
plantbiomass <- plantbiomass %>%
  mutate(log.biomass = log(biomassStability))
```

Now let’s re-run the regression analysis:

```
log.biomass.lm <- lm(log.biomass ~ nSpecies, data = plantbiomass)
```

And augment the output:

```
log.biomass.lm.augment <- log.biomass.lm %>%
  broom::augment()
```

And re-plot the residual diagnostic plots:

```
log.biomass.lm.augment %>%
  ggplot(aes(sample = .resid)) +
  stat_qq(shape = 1, size = 2) +
  stat_qq_line() +
  theme_bw()
```

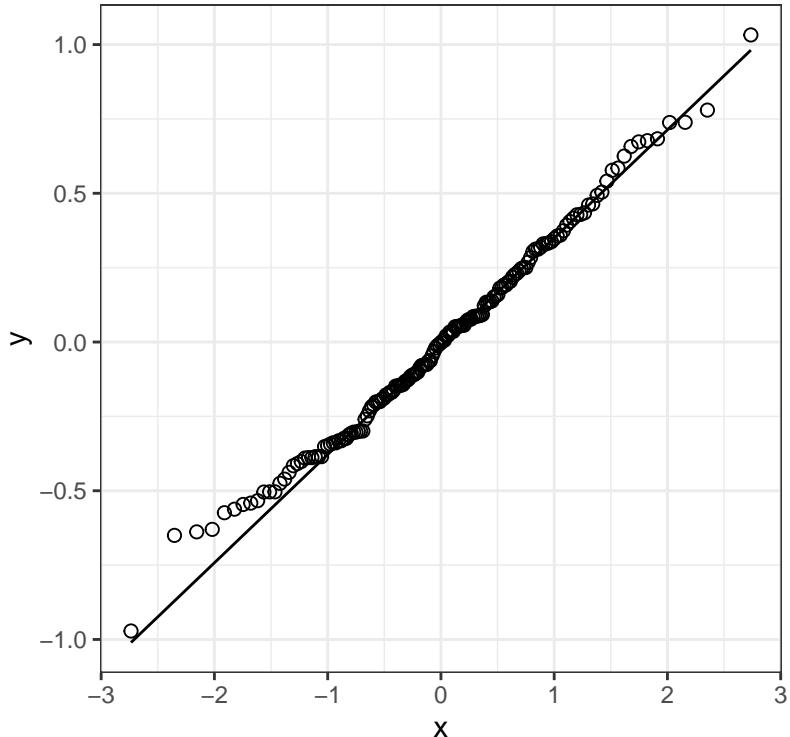


Figure 18.6: Normal quantile plot of the residuals from a regression of biomass stability (log transformed) on Species richness for 161 plots

```
log.biomass.lm.augment %>%
  ggplot(aes(x = nSpecies, y = .resid)) +
  geom_point(shape = 1) +
  geom_abline(slope = 0, linetype = "dashed") +
  xlab("Species richness") +
  ylab("Residual") +
  theme_bw()
```

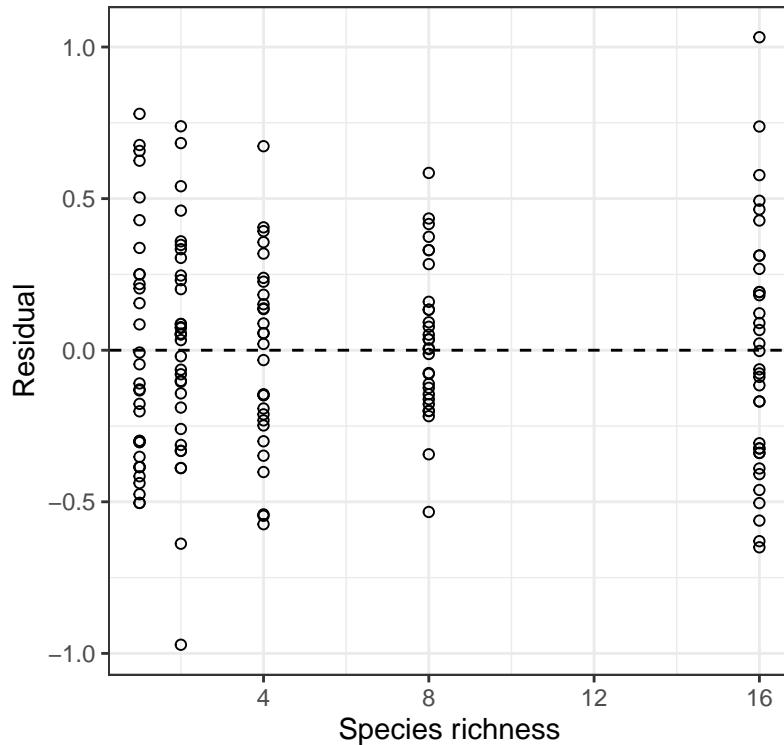


Figure 18.7: Residual plot from a regression of biomass stability (log transformed) on Species richness for 161 plots.

Figure 18.6 shows that the residuals are reasonably normally distributed, and 18.7 shows no strong pattern of changing variance in the residuals along X , and nor is there an obvious curved pattern to the residuals (which would indicate a non-linear association). We therefore proceed with the analyses using the log-transformed response variable.

18.2.10 Conduct the regression analysis

We have already conducted the main regression analysis above, using the `lm` function. We stored the output in the `log.biomass.lm` object. Now, we use the `summary` function to view the entire output from the regression analysis:

```
summary(log.biomass.lm)

##
## Call:
## lm(formula = log.biomass ~ nSpecies, data = plantbiomass)
##
## Residuals:
##    Min     1Q   Median     3Q    Max
## -0.97148 -0.25984 -0.00234  0.23100  1.03237
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 1.198294   0.041298  29.016 < 2e-16 ***
## nSpecies    0.032926   0.004884   6.742 2.73e-10 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.3484 on 159 degrees of freedom
## Multiple R-squared:  0.2223, Adjusted R-squared:  0.2174
## F-statistic: 45.45 on 1 and 159 DF, p-value: 2.733e-10
```

What do we need to focus on in this output?

Under the “Estimate” heading, you see the estimated value of the Intercept (a) and of the slope (b), which is reported to the right of the predictor variable name (“`nSpecies`”).

There are two values of t reported in the table; one associated with the intercept, and one with the slope. These are testing the implied hypotheses that (i) the intercept equals zero and (ii) the slope equals zero. We are typically only interested in the slope.

At the bottom of the output you’ll see a value for the F statistic, just like you saw in the ANOVA tutorial. **It is this value of F that you will report in statements about regression (see below).**

We also see the P -value associated with the test of the zero slope null hypothesis, which is identical to the P -value reported for the overall regression (at the bottom of the output).

It is important to recognize that an overall significant regression (i.e. slope significantly different from zero) does not necessarily mean that the regression

will yield accurate predictions. For this we need to assess the “coefficient of determination”, or R^2 value (called “multiple R-squared” in the output), which here is 0.22, and **represents the fraction of the variation in Y that is accounted for by the linear regression model**. A value of this magnitude is indicative of a comparatively weak regression, i.e. one that does has predictive power, but not particularly good predictive power (provided assumptions are met, and we’re not extrapolating beyond the range of our X values).

Before getting to our concluding statement, we need to additionally calculate the 95% confidence interval for the true slope, β .

18.2.11 Confidence interval for the slope

To calculate the confidence interval for the true slope, β , we use the `confint` function from the base stats package:

```
?confint
```

We run this function on the `lm` model output:

```
confint(log.biomass.lm)
```

```
##              2.5 %      97.5 %
## (Intercept) 1.11673087 1.27985782
## nSpecies    0.02328063 0.04257117
```

This provides the lower and upper limits of the 95% confidence interval for the intercept (top row) and slope (bottom row).

18.2.12 Scatterplot with regression confidence bands

If your regression is significant, then it is recommended to accompany your concluding statement with a scatterplot that includes so-called **confidence bands** around the regression line.

If your regression is not significant, **do not** include a regression line in your scatterplot!

We can calculate two types of confidence intervals to show on the scatterplot:

- the 95% “confidence bands”
- the 95% “prediction intervals”

Most of the time we’re interested in showing **confidence bands**, which show the 95% confidence limits for the *mean* value of Y in the population at each value of X . In other words, we’re more interested in predicting an average value in the population (easier), rather than the value of an individual in the population (much more difficult).

To display the confidence bands, we use the `geom_smooth` function from the `ggplot2` package:

```
?geom_smooth
```

Let’s give it a try, making sure to spruce it up a bit with some additional options, and including a good figure caption, shown here for your reference (this is the code you’d put after the “r” in the header of the R chunk:

```
fig.cap = "Stability of biomass production (log transformed) over 10 years in 161 plots"
```

```
plantbiomass %>%
  ggplot(aes(x = nSpecies, y = log.biomass)) +
  geom_point(shape = 1) +
  geom_smooth(method = "lm", colour = "black",
              se = TRUE, level = 0.95) +
  xlab("Species richness") +
  ylab("Biomass stability (log-transformed)") +
  theme_bw()
```

```
## `geom_smooth()` using formula = 'y ~ x'
```

There are a few key arguments for the `geom_smooth` function:

- the “method” argument tells ggplot to use a least-squares linear model
- the “`se = TRUE`” argument tells ggplot to add confidence bands
- the “`level = 0.95`” tells it to use 95% confidence level
- the “`colour`” argument tells it what colour to make the best-fit regression line

Here we can interpret the figure as follows:

Figure 18.8 shows that biomass stability (log-transformed) is positively and linearly related to the initial number of plant species assigned to experimental plots, but there is considerable variation that remains unexplained by the least-square regression model.

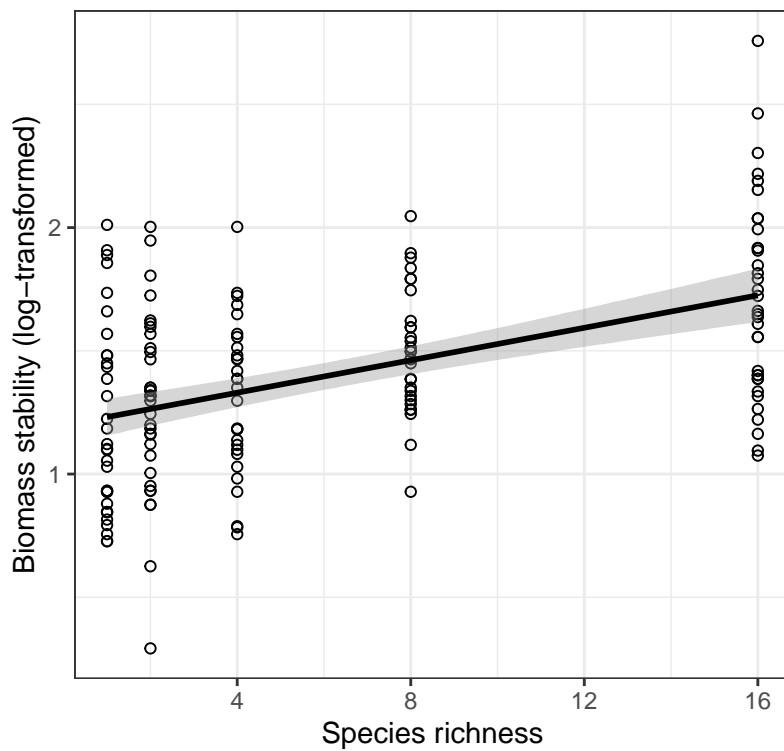


Figure 18.8: Stability of biomass production (log transformed) over 10 years in 161 plots and the initial number of plant species assigned to plots. Also shown is the significant least-square regression line (black solid line; see text for details) and the 95% confidence bands (grey shading).

18.2.13 Concluding statement

Now that we have our new figure including confidence bands (because our regression was significant), and all our regression output (including confidence intervals for the slope), we're ready to provide a concluding statement.

We simply report the sample size rather than the degrees of freedom in the parentheses.

As seen in Figure 18.8, Species richness is a significant predictor of biomass stability: $Biomass\ stability\ (log) = 1.2 + 0.03(Species\ richness)$; $F = 45.45$; $n = 161$; 95% confidence limits for the slope: 0.023, 0.043; $R^2 = 0.22$; $P < 0.001$). Given the relatively low R^2 value, predictions from our regression model will not be particularly accurate.

It is OK to report the concluding statement and associated regression output using the log-transformed data. However, if you wanted to make predictions using the regression model, it is often desirable to report the back-transformed predicted values (see below).

18.3 Making predictions

Even though the R^2 value from our regression was rather low, we'll proceed with using the model to make predictions.

It is not advisable to make predictions beyond the range of X-values upon which the regression was built. So in our case (see Figure 18.8), we would not wish to make predictions using species richness values beyond 16. Such “extrapolations” are inadvisable.

To make a prediction using our regression model, use the `predict.lm` function from the base stats package:

```
?predict.lm
```

If you do not provide new values of `nSpecies` (note that the variable name must be the same as was used when building the model), then it will simply use the old values that were supplied when building the regression model.

Here, let's create a new tibble called “new.data” that includes new values of `nSpecies` (we'll use 7 and 13 as example values) with which to make predictions:

```
new.data <- tibble(nSpecies = c(7, 13))
```

Now let's make the predictions, and **remember** that these predicted values of biomass stability will be in the log scale:

```
predicted.values <- predict.lm(log.biomass.lm, newdata = new.data)
predicted.values
```

```
##      1      2
## 1.428776 1.626331
```

Let's superimpose these predicted values over the original scatterplot, using the `annotate` function from the `ggplot2` package.

We re-use the code from our original `geom_smooth` plot above, and simply add the `annotate` line of code:

```
plantbiomass %>%
  ggplot(aes(x = nSpecies, y = log.biomass)) +
  geom_point(shape = 1) +
  geom_smooth(method = "lm", colour = "black",
              se = TRUE, level = 0.95) +
  annotate("point", x = new.data$nSpecies, y = predicted.values, shape = 2, size = 4) +
  xlab("Species richness") +
  ylab("Biomass stability") +
  theme_bw()

## `geom_smooth()` using formula = 'y ~ x'
```

As expected, Figure 18.9 shows the predicted values exactly where the regression line would be! We generally don't present such a figure... here we're just doing it for illustration.

18.3.1 Back-transforming regression predictions

The regression model we have calculated above is as follows:

$$\ln(\text{biomass stability}) \sim 1.2 + 0.03(\text{species richness})$$

Thus, any predictions we make are predictions of $\ln(y)$. It is sometimes desirable to report predicted values back in their original non-transformed state. To do this, follow the instructions in another tutorial dealing with data transformations.

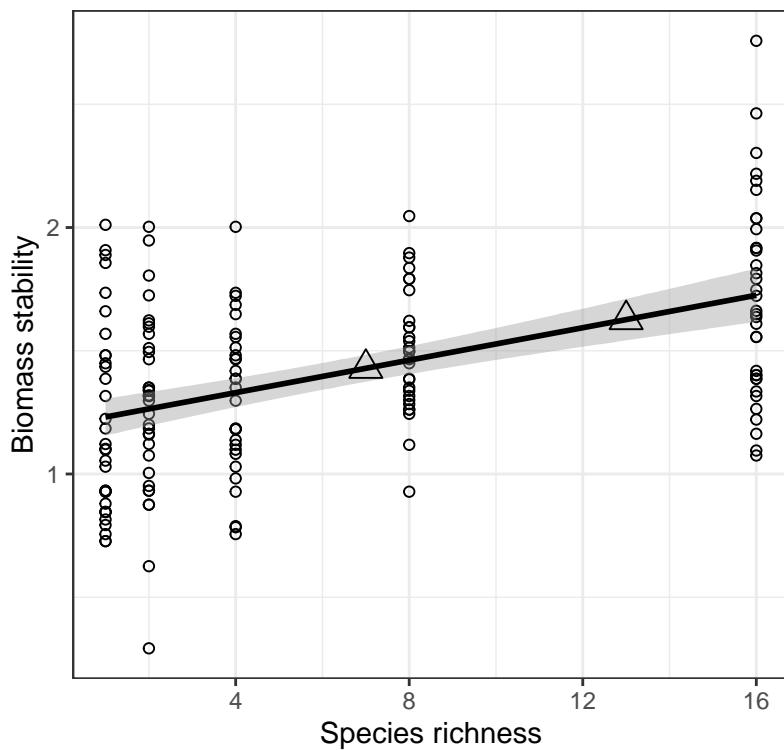


Figure 18.9: Stability of biomass production (log transformed) over 10 years in 161 plots and the initial number of plant species assigned to plots. Also shown are predicted values of stability across values of species richness.

18.4 Model-I versus Model-II regression

18.4.1 Definitions

Model-I regression is the type we just learned - ordinary least-squares regression. We fit a regression line that minimizes the squared residuals with respect to the “Y” variable.

Model-II regression or “reduced major axis regression” fits a line that minimizes the squared residuals in both the “Y” and “X” direction.

18.4.2 Which one do I use?

The biomass experiment example that we’ve used in this tutorial involved direct manipulation of species richness (number of species), and the response variable that was measured was biomass stability.

For two reasons, this experimental study lends itself to Model-I regression. First, there is good theoretical reasons to expect that manipulation of plant richness can cause changes in plant biomass stability. Thus, there is a causal direction that makes X a natural “independent” variable and Y a “dependent” variable. Second, the number of plant species was directly controlled in the experiment, and thus had zero uncertainty associated with its values (biomass stability would have had at least some uncertainty in its estimation). Under either of these conditions, Model-I regression is most appropriate.

Now imagine an observational study in which we randomly sample 40 vegetation plots, and in each, measure plant species richness (number of plant species) and insect species richness. We are interested in predicting insect richness from plant richness, but is Model-I regression appropriate?

This study arguably lends itself more to Model-II regression. Why? First, because it is conceivable that either plant species richness OR insect species richness could be a “dependent” variable or “independent” variable... the causal direction is not clear. Second, both variables were not directly controlled, and therefore there is necessarily some uncertainty involved.

There is some debate about which method should be used in what contexts. If you’re interested in learning more, see this publication and this one.

For this course, you can assume Model-I regression is appropriate for quiz/exam questions, unless otherwise indicated.

Load all the necessary packages

This shows you how to load all the packages required for all the tutorials.

If the package is not yet installed, the function will install it for you.

This list of packages avoids the use of the `tidyverse` package, which has been problematic for some.

Each time you start an R session, can copy and paste this code into your console and run it.

Alternatively, if you're working on an assignment, you can copy and past this code into a code chunk at the beginning of your markdown document, and run it.

```
all.package.names <- c("dplyr", "readr", "ggplot2", "tinytex", "skimr", "palmerpenguins", "knitr",  
  
# Now let's load all the packages, first checking if the given package needs installing  
  
# Load the packages using lapply  
  
lapply(all.package.names, function(package) {  
  if (!requireNamespace(package, quietly = TRUE)) {  
    install.packages(package, dependencies = TRUE)  
  }  
  library(package, character.only = TRUE)  
})  
  
## [[1]]  
## [1] "boot"          "broom"         "car"           "carData"  
## [5] "epitools"       "binom"        "infer"        "ggExtra"  
## [9] "ggmosaic"       "naniar"       "janitor"      "knitr"  
## [13] "palmerpenguins" "here"         "skimr"        "lubridate"  
## [17] "forcats"        "stringr"      "dplyr"        "purrr"
```

```

## [21] "readr"          "tidyverse"       "tibble"        "ggplot2"
## [25] "tidyverse"       "stats"          "graphics"      "grDevices"
## [29] "utils"           "datasets"       "methods"       "base"
##
## [[2]]
## [1] "boot"            "broom"          "car"           "carData"
## [5] "epitools"         "binom"          "infer"         "ggExtra"
## [9] "ggbmosaic"        "naniar"         "janitor"       "knitr"
## [13] "palmerpenguins"   "here"           "skimr"         "lubridate"
## [17] "forcats"          "stringr"        "dplyr"         "purrr"
## [21] "readr"            "tidyverse"      "tibble"        "ggplot2"
## [25] "tidyverse"        "stats"          "graphics"      "grDevices"
## [29] "utils"            "datasets"       "methods"       "base"
##
## [[3]]
## [1] "boot"            "broom"          "car"           "carData"
## [5] "epitools"         "binom"          "infer"         "ggExtra"
## [9] "ggbmosaic"        "naniar"         "janitor"       "knitr"
## [13] "palmerpenguins"   "here"           "skimr"         "lubridate"
## [17] "forcats"          "stringr"        "dplyr"         "purrr"
## [21] "readr"            "tidyverse"      "tibble"        "ggplot2"
## [25] "tidyverse"        "stats"          "graphics"      "grDevices"
## [29] "utils"            "datasets"       "methods"       "base"
##
## [[4]]
## [1] "tinytex"          "boot"           "broom"          "car"
## [5] "carData"          "epitools"        "binom"          "infer"
## [9] "ggExtra"           "ggbmosaic"      "naniar"         "janitor"
## [13] "knitr"            "palmerpenguins" "here"           "skimr"
## [17] "lubridate"        "forcats"        "stringr"        "dplyr"
## [21] "purrr"            "readr"           "tidyverse"      "tibble"
## [25] "ggplot2"          "tidyverse"      "stats"          "graphics"
## [29] "grDevices"        "utils"          "datasets"       "methods"
## [33] "base"
##
## [[5]]
## [1] "tinytex"          "boot"           "broom"          "car"
## [5] "carData"          "epitools"        "binom"          "infer"
## [9] "ggExtra"           "ggbmosaic"      "naniar"         "janitor"
## [13] "knitr"            "palmerpenguins" "here"           "skimr"
## [17] "lubridate"        "forcats"        "stringr"        "dplyr"
## [21] "purrr"            "readr"           "tidyverse"      "tibble"
## [25] "ggplot2"          "tidyverse"      "stats"          "graphics"
## [29] "grDevices"        "utils"          "datasets"       "methods"
## [33] "base"
##

```

```

## [[6]]
## [1] "tinytex"      "boot"        "broom"       "car"
## [5] "carData"       "epitools"     "binom"       "infer"
## [9] "ggExtra"        "ggmosaic"    "naniar"      "janitor"
## [13] "knitr"         "palmerpenguins" "here"        "skimr"
## [17] "lubridate"     "forcats"     "stringr"     "dplyr"
## [21] "purrr"          "readr"        "tidyR"       "tibble"
## [25] "ggplot2"        "tidyverse"    "stats"       "graphics"
## [29] "grDevices"      "utils"        "datasets"    "methods"
## [33] "base"
##
## [[7]]
## [1] "tinytex"      "boot"        "broom"       "car"
## [5] "carData"       "epitools"     "binom"       "infer"
## [9] "ggExtra"        "ggmosaic"    "naniar"      "janitor"
## [13] "knitr"         "palmerpenguins" "here"        "skimr"
## [17] "lubridate"     "forcats"     "stringr"     "dplyr"
## [21] "purrr"          "readr"        "tidyR"       "tibble"
## [25] "ggplot2"        "tidyverse"    "stats"       "graphics"
## [29] "grDevices"      "utils"        "datasets"    "methods"
## [33] "base"
##
## [[8]]
## [1] "tinytex"      "boot"        "broom"       "car"
## [5] "carData"       "epitools"     "binom"       "infer"
## [9] "ggExtra"        "ggmosaic"    "naniar"      "janitor"
## [13] "knitr"         "palmerpenguins" "here"        "skimr"
## [17] "lubridate"     "forcats"     "stringr"     "dplyr"
## [21] "purrr"          "readr"        "tidyR"       "tibble"
## [25] "ggplot2"        "tidyverse"    "stats"       "graphics"
## [29] "grDevices"      "utils"        "datasets"    "methods"
## [33] "base"
##
## [[9]]
## [1] "tinytex"      "boot"        "broom"       "car"
## [5] "carData"       "epitools"     "binom"       "infer"
## [9] "ggExtra"        "ggmosaic"    "naniar"      "janitor"
## [13] "knitr"         "palmerpenguins" "here"        "skimr"
## [17] "lubridate"     "forcats"     "stringr"     "dplyr"
## [21] "purrr"          "readr"        "tidyR"       "tibble"
## [25] "ggplot2"        "tidyverse"    "stats"       "graphics"
## [29] "grDevices"      "utils"        "datasets"    "methods"
## [33] "base"
##
## [[10]]
## [1] "tinytex"      "boot"        "broom"       "car"

```

```

## [5] "carData"      "epitools"      "binom"        "infer"
## [9] "ggExtra"       "ggmosaic"     "naniar"       "janitor"
## [13] "knitr"        "palmerpenguins" "here"         "skimr"
## [17] "lubridate"    "forcats"      "stringr"      "dplyr"
## [21] "purrr"        "readr"        "tidyverse"    "tibble"
## [25] "ggplot2"       "tidyverse"    "stats"        "graphics"
## [29] "grDevices"    "utils"        "datasets"    "methods"
## [33] "base"          ""

## [[11]]
## [1] "tinytex"       "boot"        "broom"        "car"
## [5] "carData"        "epitools"    "binom"        "infer"
## [9] "ggExtra"        "ggmosaic"   "naniar"       "janitor"
## [13] "knitr"         "palmerpenguins" "here"         "skimr"
## [17] "lubridate"     "forcats"     "stringr"      "dplyr"
## [21] "purrr"         "readr"       "tidyverse"    "tibble"
## [25] "ggplot2"        "tidyverse"   "stats"        "graphics"
## [29] "grDevices"     "utils"       "datasets"    "methods"
## [33] "base"          ""

## [[12]]
## [1] "tinytex"       "boot"        "broom"        "car"
## [5] "carData"        "epitools"    "binom"        "infer"
## [9] "ggExtra"        "ggmosaic"   "naniar"       "janitor"
## [13] "knitr"         "palmerpenguins" "here"         "skimr"
## [17] "lubridate"     "forcats"     "stringr"      "dplyr"
## [21] "purrr"         "readr"       "tidyverse"    "tibble"
## [25] "ggplot2"        "tidyverse"   "stats"        "graphics"
## [29] "grDevices"     "utils"       "datasets"    "methods"
## [33] "base"          ""

## [[13]]
## [1] "tinytex"       "boot"        "broom"        "car"
## [5] "carData"        "epitools"    "binom"        "infer"
## [9] "ggExtra"        "ggmosaic"   "naniar"       "janitor"
## [13] "knitr"         "palmerpenguins" "here"         "skimr"
## [17] "lubridate"     "forcats"     "stringr"      "dplyr"
## [21] "purrr"         "readr"       "tidyverse"    "tibble"
## [25] "ggplot2"        "tidyverse"   "stats"        "graphics"
## [29] "grDevices"     "utils"       "datasets"    "methods"
## [33] "base"          ""

## [[14]]
## [1] "tinytex"       "boot"        "broom"        "car"
## [5] "carData"        "epitools"    "binom"        "infer"
## [9] "ggExtra"        "ggmosaic"   "naniar"       "janitor"

```

```

## [13] "knitr"           "palmerpenguins" "here"          "skimr"
## [17] "lubridate"        "forcats"       "stringr"        "dplyr"
## [21] "purrr"            "readr"         "tidyverse"      "tibble"
## [25] "ggplot2"          "tidyverse"      "stats"         "graphics"
## [29] "grDevices"         "utils"         "datasets"      "methods"
## [33] "base"              "base"          "base"          "base"

## [[15]]
## [1] "tinytex"          "boot"          "broom"          "car"
## [5] "carData"           "epitools"       "binom"          "infer"
## [9] "ggExtra"           "ggbmosaic"     "naniar"         "janitor"
## [13] "knitr"             "palmerpenguins" "here"          "skimr"
## [17] "lubridate"         "forcats"       "stringr"        "dplyr"
## [21] "purrr"             "readr"         "tidyverse"      "tibble"
## [25] "ggplot2"           "tidyverse"      "stats"         "graphics"
## [29] "grDevices"          "utils"         "datasets"      "methods"
## [33] "base"              "base"          "base"          "base"

## [[16]]
## [1] "tinytex"          "boot"          "broom"          "car"
## [5] "carData"           "epitools"       "binom"          "infer"
## [9] "ggExtra"           "ggbmosaic"     "naniar"         "janitor"
## [13] "knitr"             "palmerpenguins" "here"          "skimr"
## [17] "lubridate"         "forcats"       "stringr"        "dplyr"
## [21] "purrr"             "readr"         "tidyverse"      "tibble"
## [25] "ggplot2"           "tidyverse"      "stats"         "graphics"
## [29] "grDevices"          "utils"         "datasets"      "methods"
## [33] "base"              "base"          "base"          "base"

## [[17]]
## [1] "kableExtra"        "tinytex"        "boot"          "broom"
## [5] "car"                "carData"        "epitools"       "binom"
## [9] "infer"              "ggExtra"        "ggbmosaic"     "naniar"
## [13] "janitor"           "knitr"          "palmerpenguins" "here"
## [17] "skimr"              "lubridate"      "forcats"       "stringr"
## [21] "dplyr"              "purrr"          "readr"         "tidyverse"
## [25] "tibble"             "ggplot2"        "tidyverse"      "stats"
## [29] "graphics"           "grDevices"      "utils"         "datasets"
## [33] "methods"            "base"          "base"          "base"

## [[18]]
## [1] "kableExtra"        "tinytex"        "boot"          "broom"
## [5] "car"                "carData"        "epitools"       "binom"
## [9] "infer"              "ggExtra"        "ggbmosaic"     "naniar"
## [13] "janitor"           "knitr"          "palmerpenguins" "here"
## [17] "skimr"              "lubridate"      "forcats"       "stringr"

```

```
## [21] "dplyr"          "purrr"           "readr"            "tidyverse"
## [25] "tibble"          "ggplot2"          "tidyverse"        "stats"
## [29] "graphics"         "grDevices"        "utils"           "datasets"
## [33] "methods"          "base"             "base"             "base"
##
## [[19]]
## [1] "kableExtra"      "tinytex"          "boot"             "broom"
## [5] "car"              "carData"          "epitools"         "binom"
## [9] "infer"            "ggExtra"          "ggmosaic"        "naniar"
## [13] "janitor"          "knitr"            "palmerpenguins"  "here"
## [17] "skimr"            "lubridate"        "forcats"          "stringr"
## [21] "dplyr"            "purrr"            "readr"            "tidyverse"
## [25] "tibble"           "ggplot2"          "tidyverse"        "stats"
## [29] "graphics"         "grDevices"        "utils"           "datasets"
## [33] "methods"          "base"             "base"             "base"
##
## [[20]]
## [1] "gtsummary"        "kableExtra"        "tinytex"          "boot"
## [5] "broom"             "car"               "carData"          "epitools"
## [9] "binom"             "infer"            "ggExtra"          "ggmosaic"
## [13] "naniar"            "janitor"          "knitr"            "palmerpenguins"
## [17] "here"              "skimr"             "lubridate"        "forcats"
## [21] "stringr"          "dplyr"             "purrr"            "readr"
## [25] "tidyverse"         "tibble"            "ggplot2"          "tidyverse"
## [29] "stats"             "graphics"          "grDevices"        "utils"
## [33] "datasets"          "methods"          "base"             "base"
```

Data summaries with “gtsummary” package

This tutorial introduces an alternative to the `skimr` package for getting overviews of datasets.

The `skimr` package and its `skim_without_charts` function can cause issues when knitting to PDF.

The `gtsummary` package appears to have fewer such issues.

If you haven’t already, install the `gtsummary` package by typing this in your console (do this only once):

```
install.packages("gtsummary")
```

Let’s load packages ...

```
library(tidyverse)
library(palmerpenguins)
library(gtsummary)
library(knitr)
```

The key function in the `gtsummary` package is the `tbl_summary` function:

```
?tbl_summary
```

Take note of the default settings for the “statistic” argument... by default, the function will return the median and IQR for numeric variables, and the sample size and relative frequency (expressed as percentage) for categorical variables.

For details on this function, along with a tutorial, see this webpage.

We’ll get an overview of the data using the `tbl_summary` function.

```
penguins %>%
 tbl_summary()
```

We could select just some numeric variables, and ask for the mean and standard deviation. Note the syntax for the “statistic” argument... we have to provide a “list”, as follows:

```
penguins %>%
  select(bill_length_mm, bill_depth_mm) %>%
  tbl_summary(statistic = list(all_continuous() ~ "{mean} ({sd})"))
```

So, if you find yourself running into issues with the `skimr` package, feel free to use the `gtsummary` package instead!

| Characteristic | N = 344¹ |
|-----------------------|----------------------------|
| species | |
| Adelie | 152 (44%) |
| Chinstrap | 68 (20%) |
| Gentoo | 124 (36%) |
| island | |
| Biscoe | 168 (49%) |
| Dream | 124 (36%) |
| Torgersen | 52 (15%) |
| bill_length_mm | 44.5 (39.2, 48.5) |
| Unknown | 2 |
| bill_depth_mm | 17.30 (15.60, 18.70) |
| Unknown | 2 |
| flipper_length_mm | 197 (190, 213) |
| Unknown | 2 |
| body_mass_g | 4,050 (3,550, 4,750) |
| Unknown | 2 |
| sex | |
| female | 165 (50%) |
| male | 168 (50%) |
| Unknown | 11 |
| year | |
| 2007 | 110 (32%) |
| 2008 | 114 (33%) |
| 2009 | 120 (35%) |

¹n (%); Median (Q1, Q3)

| Characteristic | N = 344¹ |
|-----------------------|----------------------------|
| bill_length_mm | 43.9 (5.5) |
| Unknown | 2 |
| bill_depth_mm | 17.15 (1.97) |
| Unknown | 2 |

¹Mean (SD)

Creating tables in R Markdown

Tutorial learning objectives

- Learn how to produce nice tables in R Markdown for PDF output

IMPORTANT: This tutorial aims to help you produce nice tables when knitting to PDF. However, because this tutorial is presented in HTML format, some of the output does not look the same as it will in PDF format. These instances are noted.

EXTRA: there is a new package called `flextable` that, if you're keen, you are welcome to check out! We don't cover it here.

18.5 Load packages and import data

Load the following packages, including `palmerpenguins` and `kableExtra`:

```
library(tidyverse)
library(knitr)
library(naniar)
library(skimr)
library(palmerpenguins)
library(kableExtra)
```

We'll use the “circadian.csv” dataset, which is also used in the “comparing more than 2 means tutorial”. These data are associated with Example 15.1 in the text.

The `circadian` data describe melatonin production in 22 people randomly assigned to one of three light treatments.

```
circadian <- read_csv("https://raw.githubusercontent.com/ubco-biology/BIOL202/main/data/circadian.csv")
```

```
## Rows: 22 Columns: 2
## -- Column specification -----
## Delimiter: ","
## chr (1): treatment
## dbl (1): shift
##
## 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.
```

18.5.1 Formatting output from the `skimr` package

It is always good practice to get an overview of a dataset before proceeding with analyses.

For this we have been using the `skim_without_charts` function from the `skimr` package.

You may have noticed that when you knit assignments to PDF, any output from the `skim_without_charts` function tends to run off the edge of the page. Here we'll learn how to avoid this.

The key is to assign the output to an object first, as such:

```
skim.out <- circadian %>%
  skim_without_charts()
```

We will also “transpose” (rotate) the table using the `t` base function:, so that it'll fit on a page:

```
skim.out.transposed <- t(skim.out)
```

Now we'll use the `kbl` function with various arguments to ensure the output looks good and includes a table caption:

NOTE: This table will look different when knitted to PDF (rather than HTML)

IMPORTANT: Notice that, unlike figure captions, which must be provided in the chunk header, the caption for a table is provided as an argument to the `kbl` function within the actual code!

```
kbl(skim.out.transposed, caption = "Overview of the 'circadian' dataset",
  booktabs = TRUE) %>%
  kable_styling(latex_options = "hold_position")
```

In the preceding code we:

- use the `kbl` function and provide 3 arguments:
 - provide the name of the table-like object we wish to format for output
 - provide a table caption
 - provide an argument “`booktabs = TRUE`” (this provides nice formatting)

Then we include pipes (“`%>%`”), and follow with:

- the `kable_styling` function with a few arguments:
 - “`latex_options = ‘hold_position’`” which forces the table to appear where the code chunk comes

For lots of great examples of how the `kableExtra` package can be used, see this vignette for PDF output, and this one for HTML output.

IMPORTANT: If you attempt to use the `skim_without_charts` function without wrapping the about in a `kbl` function, you will likely get an error when you attempt to knit to PDF.

Table 18.1: Overview of the ‘circadian’ dataset

| | 1 | 2 |
|----------------------|-----------|------------|
| skim_type | character | numeric |
| skim_variable | treatment | shift |
| n_missing | 0 | 0 |
| complete_rate | 1 | 1 |
| character.min | 4 | NA |
| character.max | 7 | NA |
| character.empty | 0 | NA |
| character.n_unique | 3 | NA |
| character.whitespace | 0 | NA |
| numeric.mean | NA | -0.7127273 |
| numeric.sd | NA | 0.8901534 |
| numeric.p0 | NA | -2.83 |
| numeric.p25 | NA | -1.33 |
| numeric.p50 | NA | -0.66 |
| numeric.p75 | NA | -0.05 |
| numeric.p100 | NA | 0.73 |

Let’s check to see if this approach works on a larger dataset (one with more variables).

Let's try it on the "penguins" dataset:

```
skim.penguins <- penguins %>%
  skim_without_charts()
```

Transpose:

```
skim.penguins.transposed <- t(skim.penguins)
```

Now let's try the `kbl` output:

NOTE: This table will look different when knitted to PDF (rather than HTML). Specifically, it will print off page...

```
kbl(skim.penguins.transposed, caption = "Overview of the 'penguins' dataset",
     booktabs = TRUE) %>%
  kable_styling(latex_options = "hold_position")
```

Table 18.2: Overview of the 'penguins' dataset

| | 1 | 2 | 3 |
|-------------------|-----------------------------|-----------------------------|--------------------|
| skim_type | factor | factor | factor |
| skim_variable | species | island | sex |
| n_missing | 0 | 0 | 11 |
| complete_rate | 1.0000000 | 1.0000000 | 0.9680233 |
| factor.ordered | FALSE | FALSE | FALSE |
| factor.n_unique | 3 | 3 | 2 |
| factor.top_counts | Ade: 152, Gen: 124, Chi: 68 | Bis: 168, Dre: 124, Tor: 52 | mal: 168, fem: 165 |
| numeric.mean | NA | NA | NA |
| numeric.sd | NA | NA | NA |
| numeric.p0 | NA | NA | NA |
| numeric.p25 | NA | NA | NA |
| numeric.p50 | NA | NA | NA |
| numeric.p75 | NA | NA | NA |
| numeric.p100 | NA | NA | NA |

Argh, our output went off the page!

If this happens, then we add another argument to the `kable_styling` function, the "scale_down" option:

```
kbl(skim.penguins.transposed, caption = "Overview of the 'penguins' dataset",
     booktabs = TRUE) %>%
  kable_styling(latex_options = c("scale_down", "hold_position"))
```

Table 18.3: Overview of the 'penguins' dataset

| | 1 | 2 | 3 | 4 |
|-------------------|-----------------------------|-----------------------------|--------------------|-------------|
| skim_type | factor | factor | factor | numeric |
| skim_variable | species | island | sex | bill_length |
| n_missing | 0 | 0 | 11 | 2 |
| complete_rate | 1.0000000 | 1.0000000 | 0.9680233 | 0.9941860 |
| factor.ordered | FALSE | FALSE | FALSE | NA |
| factor.n_unique | 3 | 3 | 2 | NA |
| factor.top_counts | Ade: 152, Gen: 124, Chi: 68 | Bis: 168, Dre: 124, Tor: 52 | mal: 168, fem: 165 | NA |
| numeric.mean | NA | NA | NA | 43.92193 |
| numeric.sd | NA | NA | NA | 5.4595837 |
| numeric.p0 | NA | NA | NA | 32.1 |
| numeric.p25 | NA | NA | NA | 39.225 |
| numeric.p50 | NA | NA | NA | 44.45 |
| numeric.p75 | NA | NA | NA | 48.5 |
| numeric.p100 | NA | NA | NA | 59.6 |

NOTE: On this HTML page the table above still goes off the page, but the PDF version will work!

Here's an image of the PDF output:

| | |
|-------------------|-----------------------|
| | 1 |
| skim_type | factor |
| skim_variable | species |
| n_missing | 0 |
| complete_rate | 1.0000000 |
| factor.ordered | FALSE |
| factor.n_unique | 3 |
| factor.top_counts | Ade: 152, Gen: 124, C |
| numeric.mean | NA |
| numeric.sd | NA |
| numeric.p0 | NA |
| numeric.p25 | NA |
| numeric.p50 | NA |
| numeric.p75 | NA |
| numeric.p100 | NA |

For very large datasets you may find this approach causes the font to be too small.

18.5.2 A nicely formatted table of descriptive statistics

Here is the code (which you've already learned) to create a good table of descriptive statistics for a numeric response variable grouped by categories in a categorical explanatory variable.

We'll use the “penguins” dataset again, and calculate descriptive statistics for bill lengths of male penguins, grouped by species:

```
penguins.stats <- penguins %>%
  filter(sex == "male") %>%
  group_by(species) %>%
  summarise(
    Count = n() - naniar::n_miss(bill_length_mm),
    Count_NA = naniar::n_miss(bill_length_mm),
    Mean = mean(bill_length_mm, na.rm = TRUE),
    SD = sd(bill_length_mm, na.rm = TRUE),
    SEM = SD/sqrt(Count),
    Low_95_CL = t.test(bill_length_mm, conf.level = 0.95)$conf.int[1],
    Up_95_CL = t.test(bill_length_mm, conf.level = 0.95)$conf.int[2]
  )
```

Here's what the raw table looks like:

```
penguins.stats

## # A tibble: 3 x 8
##   species     Count Count_NA   Mean     SD     SEM Low_95_CL Up_95_CL
##   <fct>      <int>    <int> <dbl>  <dbl>  <dbl>    <dbl>    <dbl>
## 1 Adelie       73        0 40.39  2.277  0.2665    39.86   40.92
## 2 Chinstrap    34        0 51.09  1.565  0.2683    50.55   51.64
## 3 Gentoo       61        0 49.47  2.721  0.3483    48.78   50.17
```

Now let's format it for PDF output:

```
tbl(penguins.stats, caption = "Descriptive statistics for bill length among male penguins.",
  booktabs = TRUE, digits = c(0, 0, 0, 2, 3, 3, 3, 3)) %>%
  kable_styling(latex_options = c("scale_down", "hold_position"), position = "center")
```

The **key difference** from the previous example done with the `skim_without_charts` output is that here we specify the number of decimal places we want each descriptive statistic to be reported to.

Table 18.4: Descriptive statistics for bill length among male penguins.

| species | Count | Count_NA | Mean | SD | SEM | Low_95_CL | Up_95_CL |
|-----------|-------|----------|-------|-------|-------|-----------|----------|
| Adelie | 73 | 0 | 40.39 | 2.277 | 0.267 | 39.859 | 40.922 |
| Chinstrap | 34 | 0 | 51.09 | 1.565 | 0.268 | 50.548 | 51.640 |
| Gentoo | 61 | 0 | 49.47 | 2.721 | 0.348 | 48.777 | 50.171 |

Specifically, the “digits” argument accepts a vector of numbers, whose length is equal to the number of columns being reported in the table, and these numbers indicate the number of decimal places to include for that specific variable.

At this point it would be a good idea to revisit the Biology department’s guidelines for reporting descriptive statistics.

For example, we can see that the first three numbers in the “digits” argument are zeroes, and these correspond to the first three columns of the table: “species”, “Count”, “Count_NA”. These are columns whose values don’t require decimal places.

For the “Mean” column we report the values to 1 more decimal place than was used in the measurement (which you find out by looking at the raw data in the “penguins” object), so here, 2 decimal places.

For measures of spread (like the standard deviation) and measures of uncertainty (including SEM and confidence limits), report the numbers to 2 more decimal places than was used in the measurement, so here, 3 decimal places.

You now know how to produce nicely formatted tables in your knitted PDF output!

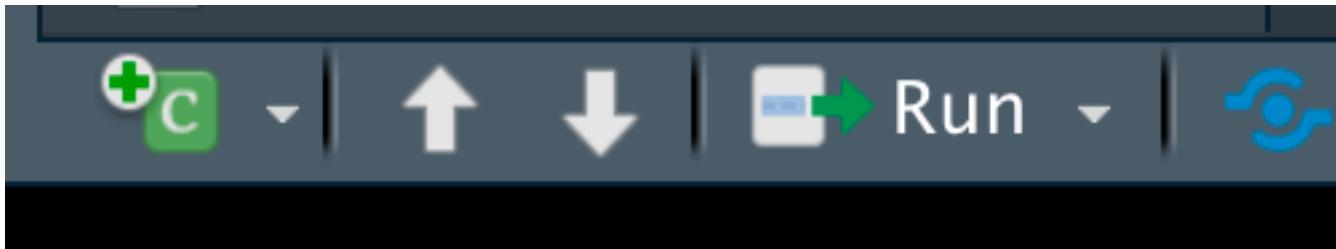
IMPORTANT: Be sure to try knitting to PDF as soon as you’ve used any of the `kable` or `kableExtra` package functions in a code chunk, as this will help you trouble-shoot if you encounter problems.

Visual Markdown Editor

This is an optional tutorial describing a new feature included in the latest version (1.4) of RStudio: a “Visual Markdown Editor”.

A more familiar editing environment

Markdown documents can be edited either in **source** mode (what we’re looking at right now) or in **visual** mode. To switch between these modes, click on the button at the top right of the toolbar on the Source panel in RStudio (when viewing a .Rmd file), or use the shortcut [Command/Control]+Shift+F4.



Toolbar above the editing pane in RStudio.

The tool you want is the one on the far-right in the picture above.

NOTE The visual editor will reformat your source code using Pandoc’s default markdown writer, so it may look slightly different when you toggle back to source view.

For many, the VME will feel much more familiar and user-friendly than the traditional Markdown source editing. Highlights of this new feature include:

- Real-time spell-checking and outline navigation.
- Tight integration with source editing (editing location and undo/redo state are preserved when switching between modes).

- Keyboard support. In addition to traditional shortcuts, you can use markdown expressions (e.g. `##`, **bold**, etc.) for formatting. If you don't remember all of the keyboard shortcuts, you can also use the catch-all `/` shortcut to insert anything.
- Extensive support for citations, including integration with Zotero and the ability to insert citations from DOIs or searches of PubMed, Crossref, and DataCite.
- Scientific and technical writing features, including cross-references, footnotes, equations, code execution, and embedded LaTeX.

NOTE If you opt to use the Visual Markdown Editor for assignments, be sure that all required code chunks are visible in your knitted document.

Common errors and their solutions

Google can help

- R will tell you specifically which line in your code (e.g. in your R Markdown document) the error is coming from. Make sure you read the error and take a look at the line where the error is if indicated
 - Try googling the specific error to see if you can find a solution online; use the search phrase “R-help error statement” (where “error statement” is what R told you)
-

Error creating mosaic plot

When attempting to create a mosaic plot (e.g. Tutorial 8.2.3) using `ggplot2` and `ggridges` packages, this error arises:

```
"Error in make_title(..., self = self) :  
  unused arguments (list(), "Treatment group")"
```

This arises due to a conflict introduced by the update to the `ggplot2` package from version 3.5.2 to version 4.0.0, which occurred on September 11, 2025.

First check the version of `ggplot2` that you have installed by typing this in the console:

```
packageVersion("ggplot2")
```

If it says anything *older* than version 4.0, then **you are fine** and you do NOT need to follow the instructions below (just make sure NOT to update your version of `ggplot2`).

If it tells you that version 4.0 is installed, then follow these instructions:

Copy and paste each of the following lines of code (one chunk at a time) into your console and run them. Note that the hashtag lines are annotations that explain each of the separate code chunks.

```
# remove current install of `ggplot2`
remove.packages("ggplot2")

# install the package `remotes` if not already installed:
if (!requireNamespace("remotes", quietly = TRUE)) {
  install.packages("remotes", dependencies = TRUE)
}

# load the `remotes` package:
library("remotes", character.only = TRUE)

# install the older, correct version of `ggplot2`:
install_version("ggplot2", version = "3.5.2", repos = "http://cran.us.r-project.org")
# if, in response, it asks to install newer versions of other packages, select "None"

# then restart R:
.rs.restartR()

# then load ggplot2
library(ggplot2)

# then check version:
packageVersion("ggplot2")
```

Hopefully at the last command it should say “‘3.5.2’”

You should not encounter the mosaic plot error now.

Rosetta error

```
rosetta error: /var/db/oah/6d4920434a69e36bfc56cad50f9f0c842a3a6392cd76b489aa238e0ee981cb99/863f93286750af2df2dc7f0313342f4d985db
ed63618967443f0e50016f949bf/pandoc.aot: attachment of code signature supplement failed: 1
Error in strsplit(info, "\n")[[1]] : subscript out of bounds
Calls: <Anonymous> ... pandoc_available -> find_pandoc -> lapply -> FUN -> get_pandoc_version
Execution halted
```

This error seems to arise for those using MacOS Ventura.

Solution: This solution is derived from this webpage, and is also described in this Piazza post.

- Close RStudio.
- Open a “terminal” window on your Mac (you can use the search tool to do this).
- Install “brew” by copying and pasting this in your terminal window:

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

- then install “pandoc” by typing this in your terminal window:

```
brew install pandoc
```

- if that’s successful, then open RStudio and type this in the command console:

```
Sys.setenv(RSTUDIO_PANDOC="/opt/homebrew/bin")
```

Now you should be able to knit an RMD file to PDF!

Rtools required during install

```
> install.packages("rlang", dependencies=TRUE)
WARNING: Rtools is required to build R packages but is not currently installed. Please download and install the appropriate version of Rtools before proceeding:
https://cran.rstudio.com/bin/windows/Rtools/
Installing package into 'C:/Users/Linda/Documents/R/win-library/4.0'
(as 'lib' is unspecified)
Warning in install.packages :
  'lib = "C:/Users/Linda/Documents/R/win-library/4.0"' is not writable
Error in install.packages : unable to install packages
```

Solution: Head to this website and follow the instructions to download Rtools onto your computer. Then try re-installing the package again.

Could not find function



```

15
16 + ````{r}
17 | skim(cars)
18 |

```

Error in skim(cars) : could not find function "skim"

Solution: This error arises when you are trying to run a package, but you haven't loaded the package yet. Therefore, R can't find the function you are trying to use. To fix this error, load the package in an R chunk near the top of your RMarkdown document and then re-run your code. Do not load packages in the console! See a screenshot below.



```

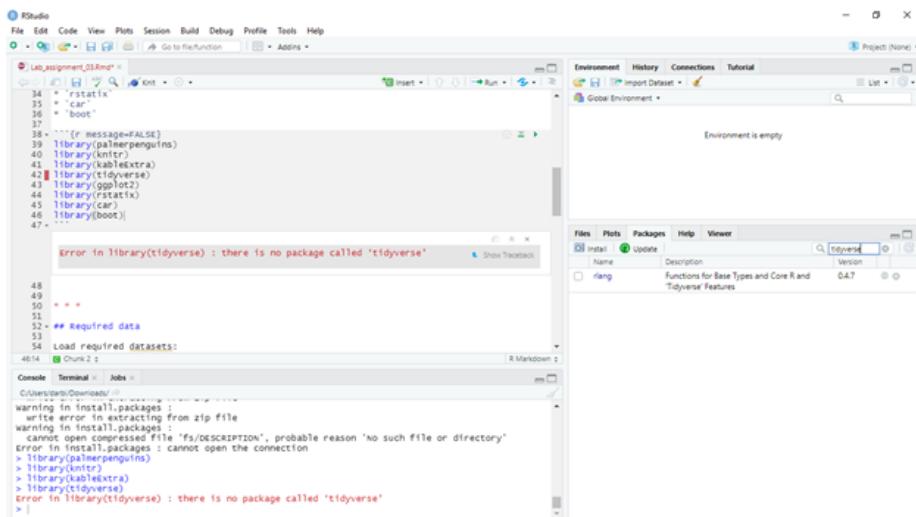
10 - ## Loading packages
11
12 - ````{r}
13 | library(skimr)
14 |

15
16 - ## Re-run the code
17 |
18 - ````{r}
19 | skim(cars)
20 |

```

-- Data Summary -----

There is no package



```

34 |   "gridExtra"
35 |   "car"
36 |   "boot"
37 |
38 - ````{r message=FALSE}
39 | library(palmerpenguins)
40 | library(knitr)
41 | library(cableExtra)
42 | library(tidyverse)
43 | library(ggplot2)
44 | library(rstatix)
45 | library(car)
46 | library(boot)
47 |

```

Error in library(tidyverse) : there is no package called "tidyverse"

```

48
49
50
51
52 - ## Required data
53
54 Load required datasets:
40/44  Chunk 2 R Markdown

```

Console Terminal Jobs

```

C:\Users\Beta\Downloads>install.packages("tidyverse")
Warning in install.packages : "tidyverse" is not available for this version of R
warning in install.packages :
  cannot open compressed file 'fs/DESCRIPTION', probable reason 'no such file or directory'
error in install.packages : cannot open the connection
> install.packages("tidyverse")
> library(palmerpenguins)
> library(knitr)
> library(cabaleExtra)
> library(tidyverse)
> Error in library(tidyverse) : there is no package called "tidyverse"
>

```

Solution: This error happens because the package has not yet been installed. To fix this error run the `install.packages()` function in the R console (NOT in the RMarkdown document!!). Then re-load the package using the `library()` function in the RMarkdown document (NOT in the console!!).

Trying to use CRAN without setting a mirror

The screenshot shows the RStudio interface with an R Markdown document open. The code in the editor is:

```

3 output:
4   word_document: default
5   html_document: default
6   ---
7   ````{r setup, include=FALSE}
8     knitr::opts_chunk$set(echo = TRUE)
9   ````{r}
10  ## Loading packages
11  ````{r}
12  install.packages("skimr", dependencies = TRUE)
13  library(skimr)
14  ````{r}
15  skim(cars)
16  ````{r}
17  ````{r}
18  ````{r}
19  ## Using functions from the package
20  ````{r}
21  skim(cars)
22  ````{r}
23  ````{r}
24  ````{r}

```

In the RStudio status bar, it says "C:/Users/Clerissa/Desktop/blah.Rmd". In the bottom right corner, there is an "Output" tab with an error message:

Line 15 Error in contrib.url(repos, "source") : trying to use CRAN without setting a mirror calls: <Anonymous> ...
withVisible -> eval -> eval -> install.packages -> contrib.url Execution halted

Solution: When we install packages in R, we need to do it in the console and not in the RMarkdown document because it causes this error when knitting. To fix this problem:

- Remove the `install.packages()` code from the RMarkdown document
- Install the package in the console

```

> install.packages("boot", dependencies = TRUE)
trying URL 'https://mug.ca/mirror/cran/bin/macosx/big-sur-x86_64/contrib/4.5/boot_1.3-32.tgz'
Content type 'application/octet-stream' length 646597 bytes (631 KB)
=====
downloaded 631 KB

The downloaded binary packages are in
  /var/folders/w0/r33v6khn0yd8700kjx2ts8300000gs/T//Rtmp1l76yq downloaded_packages
>

```

- Try re-knitting the RMarkdown document

PDF Latex is not found

```

Error: LaTeX failed to compile Practice_assignment.tex. See
https://yihui.org/tinytex/r/#debugging for debugging tips.      In
addition: Warning message: In system2(..., stdout = if (use_file_stdout())
f1 else FALSE, stderr = f2) :      '"pdflatex"' not found  Execution
halted

```

No LaTeX installation detected (LaTeX is required to create PDF output). You should install a LaTeX distribution for your platform: <https://www.latex-project.org/get/>

If you are not sure, you may install TinyTeX in R: `tinytex::install_tinytex()`

Otherwise consider MiKTeX on Windows - <http://miktex.org>

MacTeX on macOS - <https://tug.org/mactex/> (NOTE: Download with Safari rather than Chrome _strongly_ recommended)

Linux: Use system package manager

Solutions:

Option 1: Run these commands in the R console (in this specific order!):

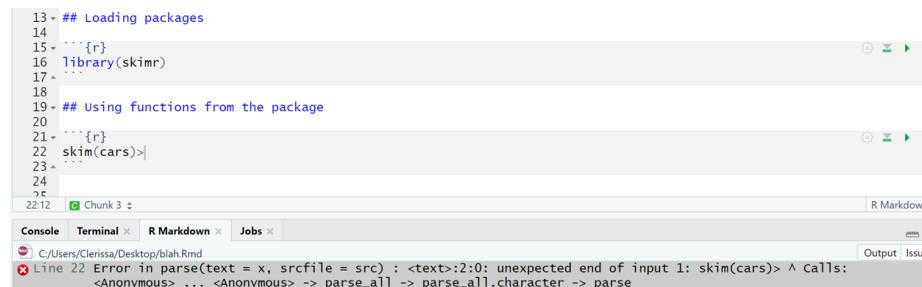
- `install.packages("tinytex", dependencies = TRUE)`
- `library("tinytex")`
- `install_tinytex()`

Option 2: If you are using a Windows computer and option 1 didn't work, try downloading MikTeX. Then try knitting again.

Option 3: If you are using a Mac computer and option 1 didn't work, try downloading MacTeX. Then try knitting again.

Option 4: If none of the above options work, knit your RMarkdown file to a Word document and then save it as a pdf.

Error in parse



The screenshot shows the RStudio interface with the following details:

- Code Editor:** Shows R code in a code editor window. Lines 13-25 are visible, including code for loading packages and using functions from the `skimr` package.
- Console Tab:** Shows the R console tab with the error message: "Error in parse(text = x, srcfile = src) : <text>:2:0: unexpected end of input 1: skim(cars)> ^ Calls: <Anonymous> ... <Anonymous> -> parse_all -> parse_all.character -> parse".
- Output Tab:** Shows the output tab with the same error message.
- Status Bar:** Shows the path "C:/Users/Clerissa/Desktop/blah.Rmd" and the status "R Markdown".

Solution: This type of error usually occurs when there is an unexpected symbol or character (ie. comma, semicolon, colon, bracket, etc.) in your code. In this case, I erroneously included a greater than sign (`>`) at the end of line. To fix this error, simply remove the unexpected symbol.

No such file or directory exists

```
pandoc: assignment_style_template.docx: openBinaryFile: does not exist (No such file or directory)
Error: pandoc document conversion failed with error 1
Execution halted
```

This error happens because in your RMarkdown document you refer to a file/folder, but that file/folder but R can't find the file.

Solution: TBD

Messy output when loading packages

```
Loading libraries
library(dplyr)

##
## Attaching package: 'dplyr'

## The following objects are masked from 'package:stats':
##
##     filter, lag

## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union

Creating the object
students <- 
read.csv(url("https://people.ok.ubc.ca/jpither/datasets/students.csv"),
header = TRUE)

Object is now loaded
nrow(students)

## [1] 154
```

Solution: When you load packages in R, the knitted document will automatically show associated messages as output in your knitted document. This can make your assignment look messy when you are loading a bunch of packages. To hide these messages, add `message = FALSE` into the R chunk where you load packages. See the example below.

```
12
13 #> ## Loading libraries
14
15 ````{r, messages=FALSE}
16 library(dplyr)
17
18
```

Unused argument

The screenshot shows the RStudio interface with the 'Console' tab selected. An error message is displayed: 'Line 110 Error in sd(~age, data = titanic, na.rm = T) : unused argument (data = titanic) calls: <Anonymous> ... handle -> withCallingHandlers -> withVisible -> eval -> eval'. The file path is 'C:/Users/kathy/Downloads/Lab_Assignment_01--1.Rmd'.

Solution: This error sometimes happens because the `library()` function is located below the line of code where a function is called on from that package. To fix this error, the `library()` function needs to be moved above the chunk where you use functions from that package. Best practice is to put it an R chunk at the top of your assignment (below the header) where you load all of the required packages for your assignment. See below for an example screenshot.

```

2 title: Practice Assignment
3 author: "Firstname Lastname, Section YY"
4 date: "Due date"
5 output:
6   word_document:
7     fig_caption: yes
8     reference_docx: assignment_style_template.docx
9     pdf_document: default
10    ---
11
12  ````{r setup, include = FALSE}
13  # DO NOT ALTER CODE IN THIS CHUNK
14  knitr::opts_chunk$set(echo = TRUE)
15
16  ***
17  * * *
18
19  ## Loading Required Packages
20
21  ````{r}
22  library(dyplr)
23  library(palmerpenguins)
24  ***

```

Object not found

```

20  ````{r}
21  nrow(students)
22  |

```

Error in nrow(students) : object 'students' not found

Solution: This error occurs when you're ‘calling’ on an object in your code, but you haven’t actually created that object yet. As such, your computer can’t find the object. To fix this error, you should insert a line of code creating your object. Make sure the object is created BEFORE (on an earlier line of code) you try to use the function. See screenshot below.

```

13  ## Creating the object
14
15  ````{r}
16  students <- read.csv(url("https://people.ok.ubc.ca/jpithier/datasets/students.csv"), header = TRUE)
17
18
19  ## Object is now loaded
20
21  ````{r}
22  nrow(students)
23  ***

```

[1] 154

Figure caption doesn't show up below figure in knitted document

```

87 ~ ## Question 1:
88
89
90 ~ ``{r, fig.cap="Scatterplot of the association between head circumference and height among 154 students."}
91 ggplot(data = students, aes(x = height_cm, y = head_circum_cm)) +
92 geom_point(shape = 1) +
93 xlab("Height (cm)") +
94 ylab("Head circumference (cm)") +
95 theme_bw()
96 ...
97 I created the above scatterplot using the `ggplot()` function.
98

```

This error happens when you knit to PDF and your figure caption doesn't show up despite having the correct code within your R chunk.

Solution: One potential cause for this error is that you have text immediately below your R chunk. To fix this error, make sure there is a blank line between the end of your R chunk with the code for the figure and any subsequent text. See the screenshot below for an example with the blank line inserted.

```

199 Create a bar graph
200
201 #> # Figure caption: This is my figure.
202 ggplot(data = my_table, aes(x = reorder(my_status, n), y = n)) +
203 geom_bar(stat = "identity") +
204 ylab("Frequency") +
205 xlab("my_status") +
206 coord_flip() +
207 theme_bw()
208 ...
209 I created the bar graph by...

```

Notice the blank line at line 208 in the above screenshot.

Figures are placed in weird spots in knitted PDF

This has to do with how the conversion process works, and can't easily be helped. And nor does it really matter - you won't lose marks for this. If you're curious as to why this happens, consult this webpage.

Installing packages: there is a binary version available

If you are attempting to install a package (using `install.packages`) and you get this message:

```

There is a binary version available but the source version is later:
  binary source needs_compilation
systemfonts  1.0.2  1.0.3          TRUE

```

Do you want to install from sources the package which needs compilation? (Yes/no/cancel)

Respond with “no” (without quotes). Do NOT respond “Yes”.

Unicode knitting error

When you try to knit your markdown file, and you get an error that refers to a “Unicode character” like this:

```
! LaTeX Error: Unicode character ^^[ (U+001B)
not set up for use with LaTeX.
```

This means there’s a strange, non-simple text character somewhere in your RMD script. Mostly likely it is the result of copying and pasting text directly from the web into your RMD document. Common examples of troublesome characters include curly quotation marks and special symbols.

To troubleshoot, first go carefully through your RMD text to look for obviously strange characters.

If that doesn’t reveal anything, you’ll need to trysystematically knitting one section of your RMD document at a time. For example:

Open a blank text document alongside your current RMD file, and cut out one question/answer at a time, pasting into blank text document temporarily. Then try knitting your reduced RMD document, and if it works, paste the cut code back in to your RMD document. Then go to the next question and do the same thing. This will help isolate problem.