The Worst Stats Text eveR

Dan Stich

Contents

The Worst Stats Text eveR

Dan Stich, PhD

Biology Department and Biological Field Station, SUNY Oneonta

Unskillful representation of an alcohol molecule where -OH is the functional group and R is the radical group, or "rest of the molecule", much like it is to modern statistics. This is funny, because R is a "functional" programming language that will drive you to drink (or perhaps undertake some other, healthier stress-reducing activity). Don't worry, I'll explain all of the jokes, and **most** of the code as we go, because this

is **The Worst Stats Text eveR**.

Preface

This book is a compilation of teaching content and lab activities that I have amassed like a digital hoarder during my time teaching BIOL 217 (Quantitative Biology) at SUNY Oneonta. The book started as a collection of R scripts that I eventually converted into web-pages under the former BIOL 217 course website using rmarkdown, and now finally into an e-document (thanks to bookdown!) that is without doubt The Worst Stats Text eveR.

The purpose of this book is to provide a tutorial survey of commonly used statistical tools in R for undergraduate students interested in biology. On any given week, our focus will be to demonstrate one or more techniques in R and show how they might be applied to real-world data, warts and all. My hope is that students take away 1) why we use these tools, 2) how to use them (and how not to!), and 3) how we show what it means. Along the way, we'll incorporate data management and exploration, statistical assumptions, and plotting.

To that end, certain ideas and language within this book are simplified for the target audience - apologies in advance if simplicity or informality jeopardize accuracy in any way. I am happy to receive constructive advice through the GitHub repository for this project located here.

This text and the course assume minimal starting knowledge of statistics or computer programming. We build on both during each chapter, and from one chapter to the next. Throughout the book, we will demonstrate statistical and biological concepts using real and simulated data sets from a variety of sub-disciplines within the biological sciences. My own academic interests are in quantitative aspects of applied ecology and fisheries management. Therefore, many of our examples have a fishy flavor, but I try to incorporate examples from other realms of biology.

The purpose of this book is not to serve as a stand-alone, citable reference document or a comprehensive guide to R even for students enrolled in my own class. It is The Worst Stats Text eveR! Why would you cite a book with that name? The code and generally citation-free ranting contained herein are, however, extensively supplemented by targeted readings on each topic from the

primary literature, published text supplements and discussions. The reader is strongly encouraged to seek out other learning resources appropriate to their comfort level (see Additional Resources on the course website).

About the author

Dr. Dan Stich is Assistant Professor of Biology at the State University of New York College at Oneonta. He teaches undergraduate and graduate courses in organismal biology, ichthyology, ecology, experimental design, lake management, and quantitative biology. He also teaches R workshops for various professional societies to which he belongs. His research focuses on the development and application of quantitative models to answer theoretical and applied questions related to fisheries and aquatic resource management and decision making. You can learn more about his teaching and research through his website.

Dan is not a programmer or a statistician. He is a fish scientist who went rogue with code and stumbled into population modeling as a graduate student. At some point it became as much a hobby as a work endeavor. He is an active user of R and Rstudio and delites in seeing others get hooked on it, too. He maintains and contributes to multiple R packages as part of his research. You can find some of these in his GitHub repositories, where he spends much time talking to himself in his own online issue trackers.

Chapter 1

Introduction to programming in R

Title image. Read about it there.

Welcome to programming in R! This module will serve as a tutorial to help you get acquainted with the R programming environment, and will get you started with some basic tools and information that will help you along your way.

We will use the Rstudio IDE to work with R in this class. It is important to note here that R is the program doing all of the thinking when we write and run code, and RStudio is a software tool that makes it a little easier to work with R - so we're going to need them both (plus a few other tools we'll check out along the way).

1.1 What is R?

Go Google it. This is The Worst Stats Text eveR.

Okay, okay. Briefly, R is a statistical programming language. That language is made up of functions and various objects (R is functional and object-oriented). Objects are things that we do stuff to, or that we create by doing stuff. Functions are the things that do stuff to objects or create objects by doing things. A lot of functions and objects are included in the `base` software distribution (this is the one you just downloaded). Other collections of functions and objects are available through "packages". You could think of these packages like web-extensions, add-ins for Microsoft programs, or mods for Minecraft. These packages may be written in R or built on a variety of other programming languages you may have heard of like C, C++, java, Python, etc. You can see a

YouTube demo of installing packages in RStudio here. We will talk more about this later.

Because R is open-source anybody can write packages (even me). Therefore, there are lots of packages out there and many of them have functions that do the same thing but have slightly different names or behaviors. This framework, and an avid user-community has propelled the capabilities of R and RStudio during recent years, and now R can do everything from basic arithmetic to spatial time-series analysis to searching Amazon. This means that learning R is also a lot like learning the English language because there are about 10 ways to do everything and many of those are based on other programming languages.

1.2 Why should I use R?

For now: because this whole class revolves around your using R. If you don't, you'll fail or look silly at a job interview. I started using R because I needed it to finish my master's thesis. I'd like to think some people start using R "just because" they want to, but usually those people just say they want to start it.

Students are in a unique position to be able to do the things they want to do because they have to do them (somebody write that down). Most of us should probably make it more of a priority.

On that note, hopefully the "why" becomes obvious to you during our time together even if you don't want to be a data scientist or a modeler. If you only ever use R to do t-tests or make descriptive plots it is worth learning. The ability to re-use the same code for a later analysis alone can save you hours. You never lose what you write (and back up!). So, the more and the longer you write R code, the more time you will have to do other things in life that you care more about (as if). If R *is* what you'll love, than hopefully we can help you enjoy that more, too. It's the software that everyone is using because of these things and more, and the development community has continued to grow during the past two decades. That means help is everywhere. Go Google it.

1.3 Where do I start?

If you haven't downloaded and installed the most recent versions of R and RStudio, you should probably go do that now. We'll wait…

Once you have installed both of these, find and open RStudio on your computer so you can work along with the examples below.

It may be helpful to watch a couple of YouTube videos before going much further, especially if you are stuck already (no shame). There are tons of them out there, including some that walk you through how to install and open R and

RStudio. They range from just a couple of minutes to a couple of hours. Here's one example provided by the How To R Channel.

Depending on how long that took, you may or may not be enthused by the following:

> the learning curve for R is steep...like a cliff, not a hill.

But, once you get the hang of it you can learn a lot really quickly. Cheat sheets like these reference cards can help you along the way by serving as miniature reference manuals in the mean-time. There are also *tons* of e-books and websites out there like the one you are reading now. And, there is a huge, active user-community just a Google away. Just searching "how to _____ in R" will return multiple results for most questions, with everything from open-source text books like this to R project websites (e.g. RStan) or programming forums like StackOverflow. You can find links to a few Additional Resources on the course website, but part of learning R is learning how to Google about R.

1.4 Programming conventions

Style and organization

Learning to write code will be easier if you bite the bullet early on and adopt some kind of organization that allows you to interact with it (read, write, run, stare aimlessly, debug) more efficiently.

There are a lot of different ways to write computer code. All of them are intended to increase efficiency and readability. Some rules are more hard-coded and program-specific than others. For example, students in this class will notice that none of my code goes beyond a certain vertical line in the editor. That is to make it so that people don't have to scroll over to the right of the editor to see what I have written when I email them code. When I share code with students I tend to justify everything *really* far to the left because everyone works on tiny laptops with multiple windows open and none of them maximized [shudders].

I suppose there is no "right" way to edit your code, but it will make your life easier if you find a style you like and stick to those conventions. If you are the kind of person who needs order in your life, you can check out the `tidyverse` style guide for tips. You can check code style with the `lintr` package or interactively re-style your code with the `styler` package if you"re thinking that may be a lot of work to remember on the front-end.

Regardless of how you end up styling your code, here are a few helpful hints that ought to help you get comfortable with your keyboard. I guess these are probably generally applicable to programming and not specific to R.

Some handy coding tips

Get to know your keyboard and your speed keys for code execution and completion. Use the mouse to navigate the GUI, not to write code. Here is a fairly comprehensive list of speed-key combinations for all of the major operating systems from the Rstudio website. You don't need to know them all, but it can save you a **ton** of time.

File management is wicked important. This is probably one of the primary struggles folks have with starting to learn R and other languages. At the same time, it is a big part of the secret sauce behind good programming. **For this class, I will assume that you are working out of a single working directory (call it something like "quant_bio" or "biol217". That means I will assume your scripts (`.R` files) for each chapter are in the same folder on your computer as your the folder that contains your data.**

An example of your class folder might look like this:

Save early and often In general, RStudio is really good about keeping track of things for you, and it is more and more foolproof these days. However, there are still times when it will crash and there is nothing you can do to get your work back unless it has been saved to a file. So, whenever you write code, write it in a source file that is saved in a place you know you can find it. It is the first thing I do when I start a script, and the last thing I do before I run any code.

Please go check out the supplemental materials on the course website or check out the YouTube video linked above for more help getting started in R if you have no idea what I am talking about at this point.

Commenting code is helpful And I will require that you do it, at least to start. Comments are a way for you to explain what your code does and why. This is useful for sharing code or just figuring out what you did six months ago. It could also be that critical piece of clarity that makes me say "Oh, I see what you did there, +1" on your homework.

```
# This is a comment.
# We know because it is preceded
# by a hashtag, or "octothorpe".

# R ignores comments so you have
# a way to write down what you have
# done or what you are doing.
```

Section breaks help organization I like to use the built-in heading style. It works really well for code-folding in R and when I"ve written a script that is several hundred lines long, sometimes all I want to see are the section headings. Go ahead and type the code below into a source file (File > New File > Rscript

or `Ctrl+Shift+N`) and save it (File > Save As or `Ctrl+S`). Press the little upside-down triangle to the left of the line to see what it does.

```
# Follow a comment with four dashes or hashes
# to insert a section heading

# Section heading ----

# Also a section heading ####
```

This is really handy for organizing sections in your homework or for breaking code up into smaller sections when you get started. You'll later learn that when you have to do this a lot, there are usually ways you can reduce your code or split it up more efficiently into other files.

Stricter R programming rules

For the next section, open RStudio if it is not already and type the code into a new source file (`Ctrl+Shift+N`).

All code is in R is case sensitive. Run the following lines (with the Run button or `Ctrl+Enter`). If you highlight all of them, they will all be run in sequence from top to bottom. Or, you can manually run each line. Running each line can be helpful for learning how to debug code early on.

```
# Same letter, different case
a <- 1
A <- 2
a == A
```

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

So, what just happened? A few things going on here.

1. We've defined a couple of objects for the first time. If we translate the first line of code, we are saying, "Hey R, assign the value of `1` to an object named `a` for me."
2. Note that the two objects are not the same, and R knows this.
3. The `==` that we typed is a logical test that checks to see if the two objects are identical. If they were, then it would have returned a `TRUE` instead of `FALSE`. This **operator** is very useful, and is more or less ubiquitous in object-oriented languages. We will use it extensively for data queries and conditional indexing (ooooh, I know!).

R will overwrite objects sequentially, so don't name two things the same, unless you don't need the first.

```
a <- 1
a <- 2
a # a takes on the second value here
print(a) # This is another way to look at the value of an object
show(a) # And, here is one more
```

Names should be short and meaningful. `a` is a terrible name, even for a temporary object in most cases.

```
myFirstObject <- 1
```

Cheesy, but better…

Punctuation and special symbols are important And, they are annoying to type in names. Avoid them in object names except for underscores "_" where you can. I try to stick with lowercase for everything I do except built-in data and data from external files because it is a pain to change everything.

```
myobject <- 1 # Illegible
my.Object <- 1 # Annoying to type
myObject <- 1 # Better, but still annoying
my_object <- 1 # Same: maybe find a less annoying name?
```

Importantly, R doesn't really care and would treat all of these as unique, but equivalent objects in all regards. Worth noting that most R style recommendations are moving toward the last example above.

Some symbol combinations are not allowed in object names But, these are usually bad names or temporary objects that create junk in your workspace anyway.

```
# In Rstudio there are nifty
# little markers to show this
# is broken
# 1a <- 1

# This one works (try it by typing
# "a1" in the console)
a1 <- 1
a2 <- a1 + 1
a3 <- a2 + 1
```

We'll see later that sequential operations that require creation of redundant objects (that require memory) are usually better replace by over-writing objects in place or using functions like the pipe `%>%` from the `magrittr` package that help us keep a "tidy" workspace.

Some things can be expressed in multiple ways. Both `T` and `TRUE` can be used to indicate a logical that evaluates as being TRUE. But `t` is used to transpose data.

```
T == TRUE
```

Some names are "reserved", "built-in", or pre-defined. Did you notice that R already knew what `T` and `TRUE` were? We will talk more about this later in the course if we need to.

Other examples include functions like `in, if, else, for, function()` and a mess of others have special uses.

Some *symbols* are also reserved for special use as "operators", like: `+, -, *, % %, &, /, <, (, {, [, "", '', ...`, and a bunch of others. We will use basically all of these in just the first couple of chapters.

1.5 Next steps

These are just some basic guidelines that should help you get started working with R and RStudio. In Chapter 2, we will begin working with objects, talk about how R sees those objects, and then look at things we can do to those objects using functions.

Chapter 2

Data structures

Contrast how you see a fish and how computers see fish. Our job is to bridge the gap. No problem…

In this chapter, we will introduce basic data structures and how to work with them in R. One of our challenges is to understand how R sees our data.

R is what is known as a "high-level" or "interpreted" programming language, in addition to being "functional" and "object-oriented". This means the pieces that make it up are a little more intuitive to the average user than most low-level languages like C or C++. The back-end of R is, in fact, a collection of low-level code that builds up the functionality that we need. This means that R has a broad range of uses, from data management to math, and even GIS and data visualization tools, all of which are conveniently wrapped in an "intuitive", "user-friendly" language.

Part of this flexibility comes from the fact that R is also a "vectorized" language. Holy cow, R is so many things. But, why do you care about this? This will help you wrap your head around how objects are created and stored in R, which will help you understand how to make, access, modify, and combine the data that you will need for any approach to data analysis. It is maybe easiest to see by taking a look at some of the data structures that we'll work with.

We will work exclusively with objects and functions created in base R for this Chapter, so you do not need any of the class data sets to play along.

2.1 Vectors

The vector is the basic unit of information in R. Pretty much everything else we'll concern ourselves with is made of vectors and can be contained within one. Wow, what an existential paradox *that* is.

Let's take a look at how this works and why it matters. Here, we have defined `a` as a variable with the value of `1`.

```
a <- 1
```

...or have we?

```
print(a)
```

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

What is the square bracket in the output here? It's an index. The index is telling us that the first element of `a` is `1`. This means that `a` is actually a "vector", not a "scalar" or singular value as you may have been thinking about it. You can think of a vector as a column in an Excel spreadsheet or an analogous data table. By treating every object (loosely) as a vector, or an element thereof, the language becomes much more general.

So, even if we define something with a single value, it is still just a vector with one element. For us, this is important because of the way that it lets us do math. It makes vector operations so easy that we don't even need to think about them when we start to make statistical models. It makes working through the math a zillion times easier than on paper! In terms of programming, it can make a lot of things easier, too.

An **atomic vector** is a vector that can hold one and only one kind of data. These can include:

- Character
- Numeric
- Integer
- Logical
- Factor
- Date/time

And some others, but none with which we'll concern ourselves here.

If you are ever curious about what kind of object you are working with, you can find out by exposing the data structure with `str()`:

Let's go play with some!

```
str(a)
```

```
##  num 1
```

Examples of atomic vectors follow. Run the code to see what it does:

Integers and numerics

First, we demonstrate one way to make a vector in R. The `c()` function ("combine") is our friend here for the quick-and-dirty approach.

In this case, we are making an object that contains a sequence of whole numbers, or integers.

```
# Make a vector of integers 1-5
a <- c(1, 2, 3, 4, 5)

# One way to look at our vector
print(a)
```

Here is another way to make the same vector, but we need to pay attention to how R sees the data type. A closer look shows that these methods produce a **numeric** vector (`num`) instead of an **integer** vector (`int`). For the most part, this one won't make a huge difference, but it can become important when writing or debugging statistical models.

```
# Define the same vector using a sequence
a <- seq(from = 1, to = 5, by = 1)
str(a)
```

```
##  num [1:5] 1 2 3 4 5
```

We can change this by explicitly telling R how to build our vector:

```
a <- as.vector(x = seq(1, 5, 1), mode = "numeric")
```

Notice that I did not include the argument names in the call to `seq()` because these are commonly used default arguments.

Characters and factors

Characters are anything that is represented as text strings.

```
b <- c("a", "b", "c", "d", "e") # Make a character vector
b # Print it to the console
```

```
## [1] "a" "b" "c" "d" "e"
```

```
str(b) # Now it's a character vector
```

```
##  chr [1:5] "a" "b" "c" "d" "e"
```

They are readily converted (sometimes automatically) to **factors**:

```
b <- as.factor(b) # But we can change if we want
b
```

```
## [1] a b c d e
## Levels: a b c d e
```

```
str(b) # Look at the data structure
```

```
##  Factor w/ 5 levels "a","b","c","d",..: 1 2 3 4 5
```

Factors are a special kind of data type in R that we may run across from time to time. They have **levels** that can be ordered numerically. This is not important except that it becomes useful for coding variables used in statistical models- R does most of this behind the scenes and we won't have to worry about it for the most part. In fact, in a lot of cases we will want to change factors to numerics or characters so they are easier to manipulate.

This is what it looks like when we code a factor as number:

```
as.numeric(b)
```

```
# What did that do?
?as.numeric
```

> Aside: we can ask R what functions mean by adding a question mark as we do above. And not just functions: we can ask it about pretty much any built-in object. The help pages take a little getting used to, but once you get the hang of it… In the mean time, the internet is your friend and you will find a multitude of online groups and forums with a quick search.

Logical vectors

Most of the `logical` vectors we deal with are yes/no or comparisons to determine whether a given piece of information matches a condition. Here, we use a logical check to see if the object `a` we created earlier is the same as object `b`. If we store the results of this check to a new object `c`, we get a new logical vector filled with `TRUE` and `FALSE`, one for each element in `a` and `b`.

```
# The "==" compares the numeric vector to the factor one
c <- a == b
c
```

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

```
str(c)
```

```
##  logi [1:5] FALSE FALSE FALSE FALSE FALSE
```

We now have a logical vector. For the sake of demonstration, we could perform any number of logical checks on a vector using built-in R functions (it does not need to be a logical like `c` above).

We can check for missing values.

```
is.na(a)
```

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

We can make sure that all values are finite.

```
is.finite(a)
```

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

The exclamation `!` point means "not" in to computers.

```
!is.na(a)
```

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

We can see if specific elements meet a criterion.

```
a == 3
```

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

We can just look at unique values.

```
unique(b)
```

```
## [1] a b c d e
## Levels: a b c d e
```

The examples above are all simple vector operations. These form the basis for data manipulation and analysis in R.

2.2 Vector operations

A lot of data manipulation in R is based on logical checks like the ones shown above. We can take these one step further to actually perform what one might think of as a query.

For example, we can reference specific elements of vectors directly. Here, we specify that we want to print the third element of `a`.

```
# This one just prints it
a[3]
```

```
## [1] 3
```

We might want to store that value to a new object `f` that is easier to read and type out.

```
# This one stores it in a new object
f <- a[3]
```

Important

If it is not yet obvious, we have to assign the output of functions to new objects for the values to be usable in the future. In the example above, `a` is never actually *changed*. This is a common source of confusion early on.

Going further, we could select vector elements based on some condition. On the first line of code, we tell R to show us the indices of the elements in vector `b` that match the character string `c`. Out loud, we would say, "`b` where the value of `b` is equal to `c`" in the first example. We can also use built-in R functions to just store the indices for all elements of `b` where `b` is equal to the character string "`c`".

```
b[b == "c"]
```

```
## [1] c
## Levels: a b c d e
```

```
which(b == "c")
```

```
## [1] 3
```

Perhaps more practically speaking, we can do elementwise operations on vectors easily in R. Here are a bunch of different things that you might be interested in doing with the objects that we've created so far. Give a few of these a try. Perhaps more practically speaking, we can do elementwise operations on vectors easily in R. Give a few of these a shot.

```
a * .5 # Multiplication
a + 100 # Addition
a - 3 # Subtraction
a / 2 # Division
a^2 # Exponentiation
exp(a) # Same as "e to the a"
log(a) # Natural logarithm
log10(a) # Log base 10
```

If we change b to `character`, we can do string manipulation, too!

```
# Convert b to character
b <- as.character(b)
```

We can append text. Remember, the examples below will just print the result. We would have to overwrite `b` or save it to a new object if we wanted to be able to use the result somewhere else later.

```
# Paste an arbitrary string on to b
paste(b, "AAAA", sep = "")
```

```
## [1] "aAAAA" "bAAAA" "cAAAA" "dAAAA" "eAAAA"
```

```
# We can do it the other way
paste("AAAA", b, sep = "")
```

```
## [1] "AAAAa" "AAAAb" "AAAAc" "AAAAd" "AAAAe"
```

```
# Add symbols to separate
paste("AAAA", b, sep = "--")
```

```
## [1] "AAAA--a" "AAAA--b" "AAAA--c" "AAAA--d" "AAAA--e"
```

```
# We can replace text
gsub(pattern = "c", replacement = "AAAA", b)
```

```
## [1] "a"    "b"    "AAAA" "d"    "e"
```

```
# Make a new object
e <- paste("AAAA", b, sep = "")

# Print to console
e
```

```
## [1] "AAAAa" "AAAAb" "AAAAc" "AAAAd" "AAAAe"
```

```
# We can strip text
# (or dates, or times, etc.)
substr(e, start = 5, stop = 5)
```

```
## [1] "a" "b" "c" "d" "e"
```

We can check how many elements are in a vector.

```
# A has a length of 5,
# try it and check it
length(a)
```

```
## [1] 5
```

```
# Yup, looks about right
a
```

```
## [1] 1 2 3 4 5
```

And we can do lots of other nifty things like this. We can also bind multiple vectors together into a rectangular `matrix`. Say what?

2.3 Matrices

Matrices are rectangular objects that we can think of as being made up of vectors.

We can make matrices by binding vectors that already exist.

```
cbind(a, e)
```

```
##      a   e
## [1,] "1" "AAAAa"
## [2,] "2" "AAAAb"
## [3,] "3" "AAAAc"
## [4,] "4" "AAAAd"
## [5,] "5" "AAAAe"
```

Or we can make an empty one to fill.

```
matrix(0, nrow = 3, ncol = 4)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    0    0    0    0
## [2,]    0    0    0    0
## [3,]    0    0    0    0
```

Or we can make one from scratch.

```
mat <- matrix(seq(1, 12), ncol = 3, nrow = 4)
```

We can do all of the things we did with vectors to matrices, but now we have more than one column, and official "rows" that we can also use to these ends:

```
ncol(mat) # Number of columns
nrow(mat) # Number of rows
length(mat) # Total number of entries
mat[2, 3] # Value of row 2, column 3
str(mat)
```

See how number of rows and columns is defined in data structure? With rows and columns, we can assign column names and row names.

```r
colnames(mat) <- c("first", "second", "third")
rownames(mat) <- c("This", "is", "a", "matrix")

# Take a look
mat
```

```
##        first second third
## This       1      5     9
## is         2      6    10
## a          3      7    11
## matrix     4      8    12
```

We can also do math on matrices just like vectors, because matrices are just vectors smooshed into two dimensions (it's totally a word).

```r
mat * 2
```

```
##        first second third
## This       2     10    18
## is         4     12    20
## a          6     14    22
## matrix     8     16    24
```

All the same operations we did on vectors above…one example.

More on matrices as we need them. We won't use these a lot in this module, but R relies heavily on matrices to do linear algebra behind the scenes in the models that we will be working with.

2.4 Dataframes

Dataframes are like matrices, only not. They have a row/column structure like matrices and are also rectangular in nature. But, they can hold more than one data type!

Dataframes are made up of atomic vectors.

This is probably the data structure that we will use most in this book, along with atomic vectors.

Let's make a dataframe to see how it works.

```
# Make a new object 'a' from a sequence
a <- seq(from = .5, to = 10, by = .5)

# Vector math: raise each 'a' to power of 2
b <- a^2

# Replicates values in object a # of times
c <- rep(c("a", "b", "c", "d"), 5)

# Note, we don't use quotes for objects,
# but we do for character variables
d <- data.frame(a, b, c)
```

Now we can look at it:

```
print(d)
```

```
##        a      b c
## 1    0.5   0.25 a
## 2    1.0   1.00 b
## 3    1.5   2.25 c
## 4    2.0   4.00 d
## 5    2.5   6.25 a
## 6    3.0   9.00 b
## 7    3.5  12.25 c
## 8    4.0  16.00 d
## 9    4.5  20.25 a
## 10   5.0  25.00 b
## 11   5.5  30.25 c
## 12   6.0  36.00 d
## 13   6.5  42.25 a
## 14   7.0  49.00 b
## 15   7.5  56.25 c
## 16   8.0  64.00 d
## 17   8.5  72.25 a
## 18   9.0  81.00 b
## 19   9.5  90.25 c
## 20  10.0 100.00 d
```

Notice that R assigns names to dataframes on the fly based on object names that you used to create them unless you specify elements of a data frame like this. They are not `colnames` as with matrices, they are `names`. You can set them when you make the dataframe like this:

```
d <- data.frame(a = a, b = b, c = c)
```

Now can look at the names.

```
# All of the names
names(d)
```

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

```
# One at a time: note indexing, names(d) is a vector!!
names(d)[2]
```

```
## [1] "b"
```

We can change the names.

```
# All at once- note quotes
names(d) <- c("Increment", "Squared", "Class")

# Print it to see what this does
names(d)

# Or, change one at a time..
names(d)[3] <- "Letter"

# Print it again to see what changed
names(d)
```

We can also rename the entire dataframe.

```
e <- d
```

Have a look:

```
# Head shows first six
# rows by default
head(e)
```

```
##     a    b c
## 1 0.5 0.25 a
## 2 1.0 1.00 b
## 3 1.5 2.25 c
## 4 2.0 4.00 d
## 5 2.5 6.25 a
## 6 3.0 9.00 b
```

```
# Or, we can look at any
# other number that we want
head(e, 10)
```

```
##      a     b c
## 1  0.5  0.25 a
## 2  1.0  1.00 b
## 3  1.5  2.25 c
## 4  2.0  4.00 d
## 5  2.5  6.25 a
## 6  3.0  9.00 b
## 7  3.5 12.25 c
## 8  4.0 16.00 d
## 9  4.5 20.25 a
## 10 5.0 25.00 b
```

We can make new columns in data frames like this!

```
# Make a new column with the
# square root of our increment
# column
e$Sqrt <- sqrt(e$Increment)
e
```

Looking at specific elements of a dataframe is similar to a matrix, with some added capabilities. We'll do this with a real data set so it's more fun. There are a whole bunch of built-in data sets that we can use for examples. Let's start by looking at the `iris` data.

```
# This is how you load built-in
# data sets
data("iris")
```

Play with the functions below to explore how this data set is stored in the environment, and how R sees it. This is a good practice to get into in general.

```
# We can use ls() to see
# what is in our environment
ls()

# Look at the first six rows
# of data in the object
head(iris)
```

```
# How many rows does it have?
nrow(iris)

# How many columns?
ncol(iris)

# What are the column names?
names(iris)

# Have a look at the data structure-
# tells us all of the above
str(iris)

# Summarize the variables
# in the dataframe
summary(iris)
```

Now let's look at some specific things.

```
# What is the value in 12th row
# of the 4th column of iris?
iris[12, 4]
```

```
## [1] 0.2
```

```
# What is the mean sepal length
# among all species in iris?
mean(iris$Sepal.Length)
```

```
## [1] 5.843333
```

What about the `mean` of `Sepal.Length` just for `setosa`?

A couple of new things going on here:

1. We can refer to the columns as atomic vectors within the dataframe if we want to. Some times we have to do this…
2. Note the logical check for species

What we are saying here is, "Hey R, show me the mean of the column Sepal.Length in the dataframe iris where the species name is `setosa`"

```
mean(iris$Sepal.Length[iris$Species == "setosa"])
```

```
## [1] 5.006
```

We can write this out longhand to make sure it's correct (it is).

```
logicalCheck <- iris$Species == "setosa"
lengthCheck <- iris$Sepal.Length[iris$Species == "setosa"]
```

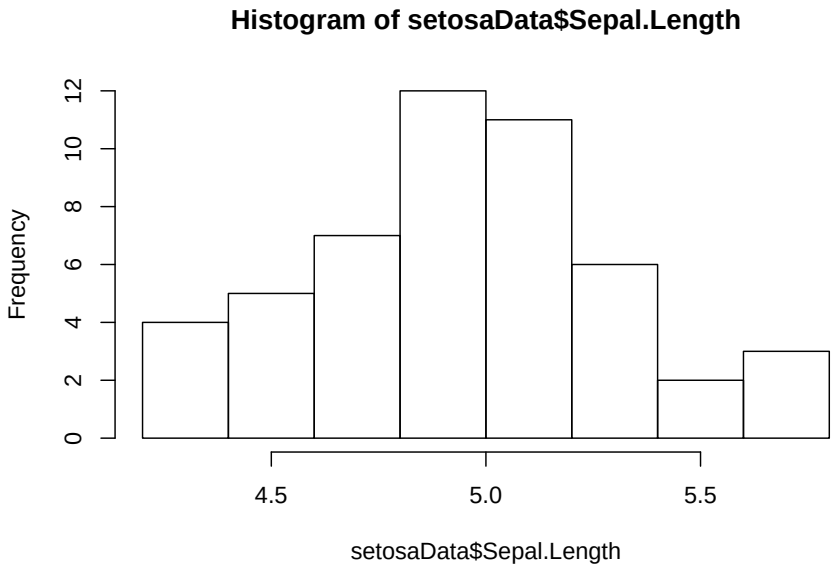
We can also look at the whole data frame just for `setosa`.

```
# Note that the structure of species
# is preserved as a factor with three
# levels even though setosa is the
# only species name in the new df
setosaData <- iris[iris$Species == "setosa", ]

str(setosaData)
```

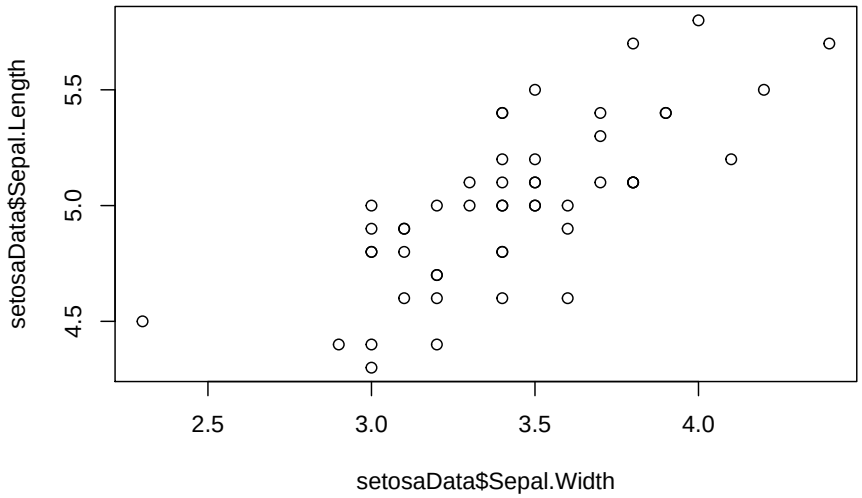
```
## 'data.frame': 50 obs. of  5 variables:
##  $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
##  $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
##  $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
##  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
##  $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
```

Finally, once we are working with dataframes, plotting becomes much easier to understand, and we can ease into some rudimentary, clunky R plots.

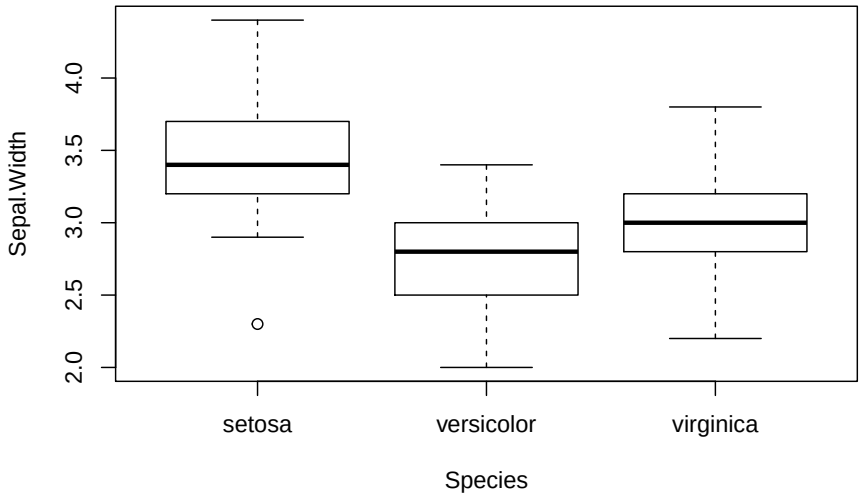
```
# Some quick plotting code

# Once we have a nice dataframe like
# these ones, we can actually step into
# The world of exploratory analyses.

# Make a histogram of sepal lengths
hist(setosaData$Sepal.Length)
```




```
# Bi-plot
plot(setosaData$Sepal.Width, setosaData$Sepal.Length)
```



```r
# Boxplots
boxplot(Sepal.Width ~ Species, data = iris)
```



Much, **MUCH** more of this to come as we continue.

2.5 Lists

Lists are the ultimate data type in R. They are actually a vector that can hold different kinds of data, like a dataframe. In fact, a dataframe is just a spectacularly rectangular list. Each element of a list can be any kind of object (an atomic vector, a matrix, a dataframe, or even another list!!).

Most of the real, filthy R programming relies heavily on lists. We will have to work with them at some point in this class, but we won't take a ton of time on them here.

Let's make a list- just to see how they work. Notice how our index operator has changed from `[ ]` to `[[ ]]`? And, at the highest level of organization, we have only one dimension in our list, but any given element `myList[[i]]` could hold any number of dimensions.

```r
# Create an empty list with four elements
myList <- vector(mode = "list", length = 4)
```

```
# Assign some of our previously
# created objects to the elements
myList[[1]] <- a
myList[[2]] <- c
myList[[3]] <- mat
myList[[4]] <- d
```

Have a look at the list:

```
# Print it
# Cool, huh?
myList
```

```
## [[1]]
##  [1]  0.5  1.0  1.5  2.0  2.5  3.0  3.5  4.0  4.5  5.0  5.5  6.0  6.5  7.0  7.5
## [16]  8.0  8.5  9.0  9.5 10.0
##
## [[2]]
##  [1] "a" "b" "c" "d" "a" "b" "c" "d" "a" "b" "c" "d" "a" "b" "c" "d" "a" "b" "c"
## [20] "d"
##
## [[3]]
##        first second third
## This       1      5     9
## is         2      6    10
## a          3      7    11
## matrix     4      8    12
##
## [[4]]
##       a      b c
## 1   0.5   0.25 a
## 2   1.0   1.00 b
## 3   1.5   2.25 c
## 4   2.0   4.00 d
## 5   2.5   6.25 a
## 6   3.0   9.00 b
## 7   3.5  12.25 c
## 8   4.0  16.00 d
## 9   4.5  20.25 a
## 10  5.0  25.00 b
## 11  5.5  30.25 c
## 12  6.0  36.00 d
## 13  6.5  42.25 a
## 14  7.0  49.00 b
## 15  7.5  56.25 c
```

```
## 16  8.0  64.00 d
## 17  8.5  72.25 a
## 18  9.0  81.00 b
## 19  9.5  90.25 c
## 20 10.0 100.00 d
```

You can assign names when you create the list like we did for dataframes, too. You can do this manually, or R will do it on the fly for you. You can also reassign names to a list that yo've already created.

```
# No names by default
names(myList)

# Give it names like we did with
# a dataframe
names(myList) <- c("a", "c", "mat", "d")

# See how the names work now?
myList

# We reference these differently [[]]
myList[[1]]

# But we can still get into each object
# Play around with the numbers to see what they do!
myList[[2]][5]

# Can also reference it this way!
myList$c[1]
```

Very commonly, model objects and output are stored as lists. In fact, most objects that require a large amount of diverse information in R pack it all together in one place using lists, that way we always know where to find it and how as long as the objects are documented. Conceptually, every object in R, from your workspace on down the line, is a list **AND** an element of a list. It seems like a lot to take in now, but will be very useful in the future.

2.6 Next steps

For more practice with the data structures and R functions we covered here, you can check out this walk-through of basic R commands from the How To R YouTube Channel.

In the Chapter 3(#Chapter3), we will begin using functions from external R packages to read and work with real data.

Chapter 3

Working with data

American shad, the best fish, lost in the data deluge. Let's figure out how to make some sense of it.

The purpose of this chapter is to get you comfortable working with data in R and give you some tools for summarizing those data in a meaningful way. This is not meant to be a comprehensive treatment of these subjects but rather an introduction to the tools that are available to you (say it with me: "Worst Stats Text eveR"). There are a lot of tools out there and you may come up with something that works better for you once you have some basics under your belt.

Now that you have a handle on the types of data you can expect to run into in R, let's have a look at how we read and work with data that we get from the real world.

We will work with the `ctr_fish.csv` file for Chapter 3, so you will need to download the class data sets that go with this book to play along. We will also need the `tidyverse` package, but instructions for installation are provided below because this is the first time we have downloaded and installed a package.

3.1 Data read

There are few things that will turn someone away from a statistical software program faster than if they can't even figure out how to get the program to read in their data. So, we are going to get it out of the way right up front!

Let's start by reading in a data file - this time we use real data.

The data are stored in a "comma separated values" file (`.csv` extension). This is a fairly universal format, so we read it in using the fairly universal `read.csv()` function. This would change depending on how the data were stored, or how

big the data files were, but that is a topic further investigation for a later date. I probably do 95% of my data reads using `.csv` files. We'll look at a few others later.

Important Remember that I am assuming your scripts are in the same directory (folder) on your computer as where you downloaded and unzipped the class data (see here for reminder).

Before you can read this file you will need to set your working directory. For class, I will ask that you click `Session > Set Working Directory > To Source File Location`. This will set the working directory to wherever you have saved your code so that R can find the folder `data` and the files inside of it. You'll notice that R spits out some code in the console when you click this. You can also use that code to set a working directory in your script but that can cause all kinds of problems, so don't do it.

```
# Start by reading in the data
am_shad <- read.csv("data/ctr_fish.csv")
```

Once you've read your data in, it's always a good idea to look at the first few lines of data to make sure nothing looks 'fishy'. Ha-ha, I couldn't help myself!

These are sex-specific length and age data for American shad (*Alosa sapidissima*) from the Connecticut River, USA. The data are used in models that I maintain with collaborators from NOAA Fisheries, the US Geological Survey, the US Fish and Wildlife Service, and others. The data were provided by CT Department of Energy and Environmental Protection (CTDEEP) and come from adult fish that return to the river from the ocean each year to spawn in fresh water.

You can look at the first few rows of data with the `head()` function:

```
# Look at the first 10 rows
head(am_shad, 10)
```

```
##    Sex Age Length yearCollected backCalculated Mass
## 1    B   1     13          2010           TRUE   NA
## 2    B   1     15          2010           TRUE   NA
## 3    B   1     15          2010           TRUE   NA
## 4    B   1     15          2010           TRUE   NA
## 5    B   1     15          2010           TRUE   NA
## 6    B   1     15          2010           TRUE   NA
## 7    B   1     16          2010           TRUE   NA
## 8    B   1     16          2010           TRUE   NA
## 9    B   1     16          2010           TRUE   NA
## 10   B   1     16          2010           TRUE   NA
```

The `NA` values are supposed to be there. They are missing data.

And, don't forget about your old friend `str()` for a peek at how R sees your data. This can take care of a lot of potential problems later on.

```
# Look at the structure of the data
str(am_shad)
```

```
## 'data.frame': 16946 obs. of  6 variables:
##  $ Sex           : Factor w/ 2 levels "B","R": 1 1 1 1 1 1 1 1 1 ...
##  $ Age           : int  1 1 1 1 1 1 1 1 1 ...
##  $ Length        : num  13 15 15 15 15 15 16 16 16 16 ...
##  $ yearCollected : int  2010 2010 2010 2010 2010 2010 2010 2010 2010 2010 ...
##  $ backCalculated: logi  TRUE TRUE TRUE TRUE TRUE TRUE ...
##  $ Mass          : int  NA NA NA NA NA NA NA NA NA NA ...
```

There are about 17,000 observations (rows) of 6 variables (columns) in this data set. Here is a quick breakdown:

`Sex`: fish gender. `B` stands for 'buck' (males), `R` stands for 'roe' (females).

`Age`: an integer describing fish age.

`Length`: fish length at age (cm).

`yearCollected`: the year in which the fish was caught.

`backCalculated`: a logical indicating whether or not the length of the fish was back-calculated from aging.

`Mass`: the mass of indnividual fish (in grams). Note that this is NA for all ages that were estimated from hard structures (so all cases for which `backCalculated == TRUE`).

3.2 Quick data summaries

There are a number of simple ways to summarize data quickly in base R. We already looked at a few of these in previous chapters. But what about something a little more in-depth?

One quick way to look at your data is using the `summary()` function

```
summary(am_shad)
```

```
##  Sex           Age            Length      yearCollected  backCalculated
##  B:9512   Min.   :1.000   Min.   : 3.00   Min.   :2010   Mode :logical
##  R:7434   1st Qu.:2.000   1st Qu.:31.00   1st Qu.:2011   FALSE:3046
##           Median :3.000   Median :38.00   Median :2012   TRUE :13900
##           Mean   :3.155   Mean   :36.39   Mean   :2012
##           3rd Qu.:4.000   3rd Qu.:43.00   3rd Qu.:2013
##           Max.   :7.000   Max.   :55.00   Max.   :2014
##
##       Mass
##  Min.   :   0
##  1st Qu.: 900
##  Median :1120
##  Mean   :1173
##  3rd Qu.:1440
##  Max.   :3280
##  NA's   :14115
```

This is useful for getting the big-picture. For continuous variables (e.g., `Age` and `Length`) R will report some descriptive statistics like the `mean`, `median`, and quantiles. For discrete variables (e.g. `Sex` and `backCalculated`) we get the mode (if not `factor` or `chr`) and counts of observations within each discrete level (e.g. number of observations of `B` and `R` in the variable `Sex`).

But, this approach doesn't really give us much info.

We can create more meaningful summaries pretty easily if we install and load some packages like we talked about in Chapter 1, and then look at different ways of sub-setting the data with base R and some methods that might be a little more intuitive for you.

3.3 Subsetting and selecting data

Before we can make meaningful data summaries, we will probably need to re-organize our data in a logical way (through sub-setting or selected specific chunks of data). A lot of times, we do this along the way without really thinking about it.

3.3.1 Manual subsets and selections

We talked a little about sub-setting data with logical queries in Chapter 2. Now, let's refresh and take that a little further to see why we might want to do that.

First, we'll select just the data from `am_shad` where `backCalculated` was `FALSE`. This will give us only the measured `Length` and `Mass` for each of the fish, along with their `Sex` and `yearCollected`. I'll call this new object `measured`. Remember, `am_shad` is a data frame, so it has two dimensions when we use `[ ]` for sub-setting and these are separated by a comma, like this: `object[rows, columns]`. When we leave the columns blank, R knows that it should keep all of the columns.

```
measured <- am_shad[am_shad$backCalculated == FALSE, ]
```

We could do this for as many conceivable conditions in our data on which we may wish to subset, but the code can get clunky and hard to manage. For example can you imagine re-writing this if you just want to select age six roes without back-calculated lengths?

```
# Notice how we string together multiple
# conditions with "&". If these were 'or'
# we would use the vertical pipe "|"
age_6_rows_measured <- am_shad[am_shad$backCalculated == FALSE &
                                 am_shad$Sex == "R" &
                                 am_shad$Age == 6, ]
```

3.3.2 Subsetting and summaries in base R

This notation can be really confusing to folks who are just trying to learn a new programming language. Because of that, there are great functions like `subset()` available that are more intuitive (but less clear to programmers). You could also subset the data using the following code:

```
measured <- subset(am_shad, backCalculated == FALSE)
```

We could also get our age-six females from the previous example using this approach, and at least the code is a little cleaner:

```
age_6_roes_measured <- subset(am_shad,
                              backCalculated == FALSE &
                                Sex == "R" &
                                Age == 6
                              )
```

Both do the same thing, but we'll see later that using `subset` is preferable if we plan on chaining together a bunch of data manipulation commands using pipes (%>%).

Next, we might be interested to know how many fish we have represented in each `Sex`. We can find this out using the `table` function in base R:

```
# Here, I use the column name because
# we just want all observations of a single
# variable. Be careful switching between names,
# numbers, and $names!
table(measured['Sex'])
```

```
##
##    B    R
## 1793 1253
```

We see that we have `1793` females and `1253` males.

We can also get tallies of the number of fish in each `Age` for each `Sex` if we would like to see that:

```
table(measured$Sex, measured$Age)
```

```
##
##       3   4   5   6   7
##   B 255 848 579 108   3
##   R   0 361 658 220  14
```

But, what if we wanted to calculate some kind of summary statistic, like a `mean` and report that by group?

For our age-6 females example, it would look like this:

```
age_6_roes_measured <- subset(am_shad,
                              backCalculated == FALSE &
                                Sex == "R" &
                                Age == 6
                              )

age_6_female_mean <- mean(age_6_roes_measured$Length)
```

Again, we could do this manually, but would require a lot of code for a simple calculation if we use the methods above all by themselves to get these for each age group of roes.

We would basically just copy-and-paste the code over and over to force R into making the data summaries we need. Nothing wrong with this approach, and it certainly has its uses for simple summaries, but it can be cumbersome and

redundant. It also fills your workspace up with tons of objects that are hard to keep track of and that will cause your code-completion suggestions to be *wicked* annoying in RStudio.

That usually means there is a better way to write the code…

3.3.3 Subsetting and summaries in the tidyverse

Long ago, when I was still a noOb writing R code with a stand-alone text editor and a console there were not a ton of packages available for the express purpose of cleaning up data manipulation in R. The one I relied on most heavily was the `plyr` package. Since then, R has grown and a lot of these functions have been gathered under the umbrella of the tidyverse, which is a collection of specific R packages designed to make the whole process less painful. These include packages like `dplyr` (which replaced `plyr`) and others that are designed to work together with similar syntax to make data science (for us, data manipulation and presentation) a lot cleaner and better standardized. We will rely heavily on packages in the tidyverse throughout this book.

Before we can work with these packages, however, we need to install them - something we haven't talked about yet! Most of the critical R packages are hosted through the Comprehensive R Archive Network, or CRAN. Still, tons of others are available for installation from hosting services like GitHub and GitLab.

If you haven't seen it yet, here is a three-minute video explaining how to install packages using RStudio. **Watch it. Please.**

It is also easy to install packages by running a line of code in the console. We could install each of the packages in the tidyverse separately. But we can also get all of them at once because they are all packaged together, too.

Follow the instructions in the YouTube link above, or install the package from the command line:

```
install.packages('tidyverse')
```

Once we have installed these packages, we can use the functions in them to clean up our data manipulation pipeline and get some really useful information.

3.4 Better data summaries

Now, we'll look at some slightly more advanced summaries. Start by loading the `dplyr` package into your R session with the following code.

```
library(dplyr)
```

We can use functions from the `dplyr` package to calculate mean `Length` of fish for each combination of `Sex` and `Age` group much more easily than we did for a single group above.

First, we group the data in `measured` data frame that we created previously using the `group_by` function. For this, we just need to give R the data frame and the variables by which we would like to group:

```
g_lengths <- group_by(measured, Sex, Age)
```

This doesn't change how we see the data much (it gets converted to a `tibble`), just how R sees it.

Next, we summarize the variable `Length` by `Sex` and `Age` using the `summarize` function:

```
sum_out <- summarize(g_lengths, avg = mean(Length))

head(sum_out)
```

```
## # A tibble: 6 x 3
## # Groups:   Sex [2]
##   Sex     Age   avg
##   <fct> <int> <dbl>
## 1 B         3  38.1
## 2 B         4  40.5
## 3 B         5  42.0
## 4 B         6  43.4
## 5 B         7  46.8
## 6 R         4  45.0
```

Wow! That was super-easy!

Finally, to make things even more streamlined, we can chain all of these operations together using the `%>%` function from `magrittr`. This really cleans up the code and gives us small chunks of code that are easier to read than the dozens of lines of code it would take to do this manually.

```
# This will do it all at once!
sum_out <- # Front-end object assignment
  measured %>% # Pass measured to the group_by function
  group_by(Sex, Age) %>% # Group by Sex and age and pass to summarize
  summarize(avg = mean(Length))
```

We could also assign the output to a variable at the end, whichever is easier for you to read:

```
  measured %>% # Pass measured to the group_by function
  group_by(Sex, Age) %>% # Group by Sex and age and pass to summarize
  summarize(avg = mean(Length)) -> sim_out # Back-end object assignment
```

And, it is really easy to get multiple summaries out like this at once:

```
sum_out <-
  measured %>%
  group_by(Sex, Age) %>%
  summarize(avg = mean(Length), s.d. = sd(Length))

head(sum_out)
```

```
## # A tibble: 6 x 4
## # Groups:   Sex [2]
##   Sex     Age   avg  s.d.
##   <fct> <int> <dbl> <dbl>
## 1 B         3  38.1  2.75
## 2 B         4  40.5  2.70
## 3 B         5  42.0  2.29
## 4 B         6  43.4  2.09
## 5 B         7  46.8  1.61
## 6 R         4  45.0  2.65
```

Isn't that slick? Just think how long that would have taken most of us in Excel!

This is just one example of how functions in packages can make your life easier and your code more efficient. Now that we have the basics under our belts, lets move on to how we create new variables.

3.5 Creating new variables

There are basically two ways to create new variables: we can modify an existing variable (groups or formulas), or we can simulate new values for that variable (random sampling.)

If we have a formula that relates two variables, we could predict one based on the other deterministically.

For example, I have fit a length-weight regression to explain the relationship between `Length` and `Mass` using the `am_shad` data we've worked with in previous sections.

This relationship looks like your old friend $y = mx + b$, the equation for a line, but we log10-transform both of the variables before fitting the line (more to come later in the class). Using this relationship, we can predict our **independent variable** (`Mass`) from our **dependent variable** (`Length`) if we plug in new values for `Length` and the **parameters** of the line.

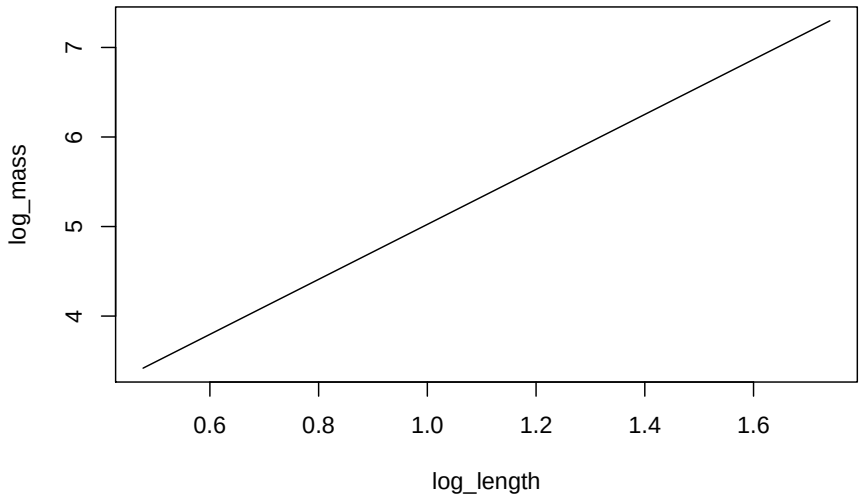
In this case, I know that `m` = 3.0703621, and `b` = -1.9535405.

If I plug these numbers in to the equation above, I can predict `log10(Mass)` for new lengths `log10(Length)`:

$$log_{10}Mass = 3.0703621 \cdot log_{10}Length - 1.9535405$$

In R, this looks like:

```
# Parameters from length-weight regression
m <- 3.0703621
b <- 1.9535405

# Make a sequence of new lengths based on range in data,
# then take the log of the whole thing all at once.
log_length <- log10( seq(min(am_shad$Length), max(am_shad$Length), 1) )

# Calculate a new thing (log10_mass) using parameters for line
# and sequence of new log10_length.
log_mass <- m * log_length + b

# Plot the prediction
plot(x = log_length, y = log_mass, type = "l")
```



3.6 Data simulation

The point of simulation is usually to account for uncertainty in some process (i.e. we could just pick a single value if we knew it). This is almost always done based on probability. There are a number of ways we could do this. One is by drawing from some probability distribution that we have described, and the other is by randomly sampling data that we already have.

3.6.1 Random sub-samples from a dataset

Let's say we want to take random samples from our huge data set so we can fit models to a subset of data and then use the rest of our data for model validation in weeks to come.

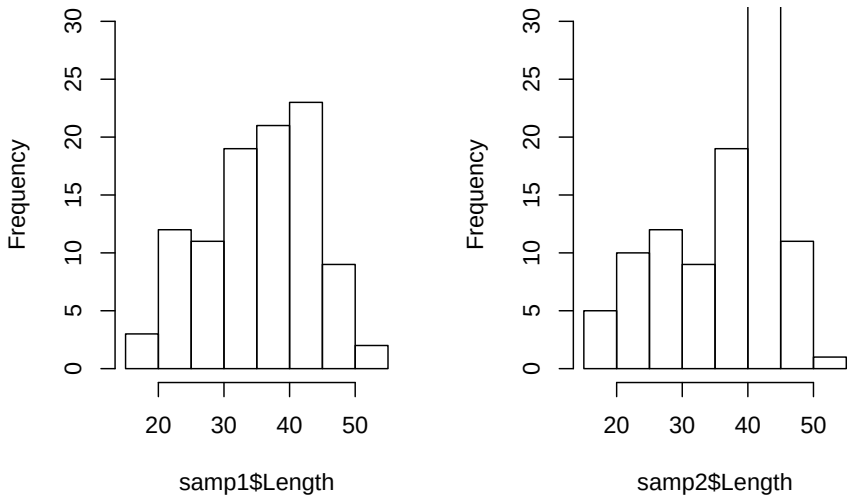
We have around 17,000 observations in the `am_shad` data set. But, what if we wanted to know what it would look like if we only had 100 samples from the same population?

First, tell R how many samples you want.

```
n_samples <- 100
```

Now let's take two samples of 100 fish from our dataframe to see how they compare:

```
# Randomly sample 100 rows of data from our data frame two different
# times to see the differences
samp1 <- am_shad[sample(nrow(am_shad), size = n_samples, replace = FALSE), ]
samp2 <- am_shad[sample(nrow(am_shad), size = n_samples, replace = FALSE), ]

# We can look at them with our histograms
par(mfrow = c(1, 2))
hist(samp1$Length, main = "", ylim = c(0, 30))
hist(samp2$Length, main = "", ylim = c(0, 30))
```



*If you are struggling to get your plotting window back to "normal" after this, you can either click the broom button in your "Plots" window, or you can run the following code for now:

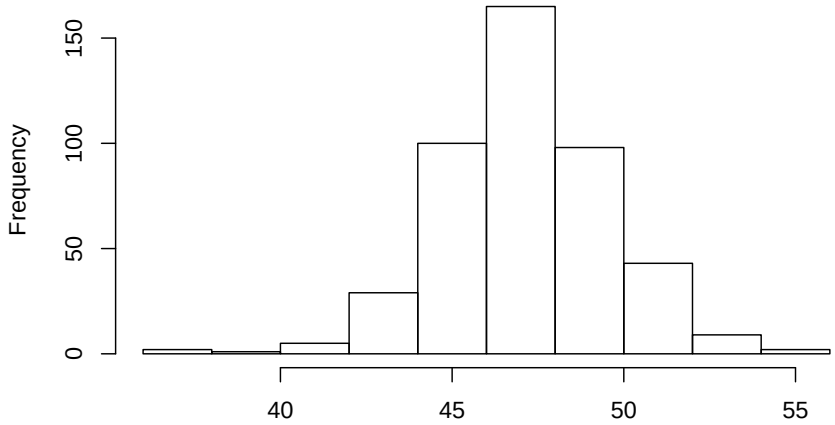
3.6.2 Stochastic simulation

Now, instead of sampling our data let's say we have some distribution from which we would like sample. So, let's make a distribution.

We will start with the normal, and we can move into others when we talk about probability distributions and sample statistics in Chapter 5. For this, we will use the distribution of American shad lengths for age-6 females because it approximates a normal distribution. We will calculate the `mean` and `sd` because those are the parameters of the normal distribution.

Start by looking at the size distribution for age 6 females. We use the tidy workflow here with really awful default graphics (more to come in Chapter 4), but we add two arguments to our `subset` call. We want to select only the variable `Length` from `am_shad`, and we want to drop all other information so we can send the output straight to the `hist()` function as a vector.

```
am_shad %>%
  subset(Age == 6 & Sex == "R", select='Length', drop=TRUE) %>%
   hist(main = "")
```



.

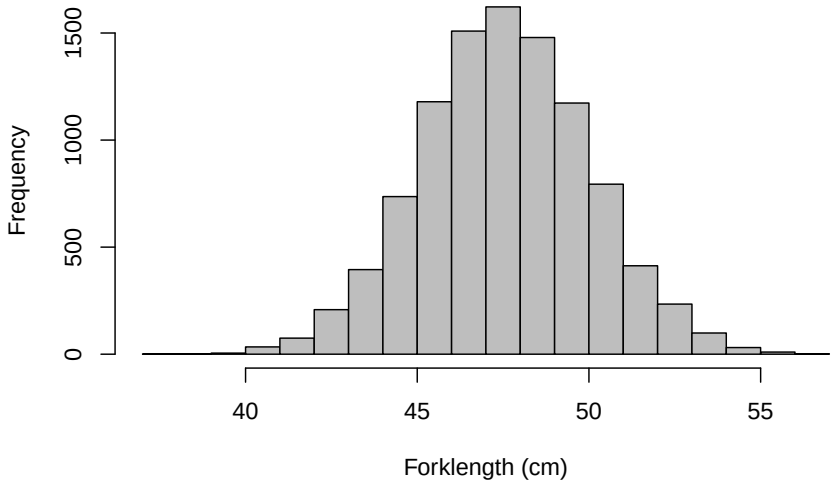
Now, let's calculate the `mean` and `sd` of `Length` for age 6 females.

```
# Calculate the mean Length
x_bar <- am_shad %>%
  subset(Age == 6 & Sex == "R", select='Length', drop=TRUE) %>%
  mean

# Calculate standard deviation of Length
sigma <- am_shad %>%
  subset(Age == 6 & Sex == "R", select='Length', drop=TRUE) %>%
  sd
```

Note that we could also use the `filter()` function from the `dplyr` package for this job, and for big data sets it would be a lot faster for un-grouped data.

Now, we can use the mean and standard deviation to randomly sample our normal distribution of lengths.

```
# Take a random sample from a normal distribution
length_sample <- rnorm(n = 10000, mean = x_bar, sd = sigma)

# Plot the sample to see if it is a normal- YAY it is!
hist(length_sample,
  col = "gray",
  main = "",
  xlab = "Forklength (cm)"
)
```



We've add a couple of new arguments to the histogram call to make it a little less ugly here. In Chapter 4 we are going to ramp it up and play with some plots!

3.7 Next steps

In this chapter, we provided a general overview of our work flow when it comes to reading in data and manipulating them to get useful summaries. In Chapter 4 we will use these processes to help us visualize important trends in these summaries before we begin working with descriptive statistics and sampling distributions in Chapter 5.

Chapter 4

Plotting and graphics

Sweet graphs, right? Want to learn how to make them? Okay, but baby steps here, alright?.

In this Chapter we will walk through plotting in R, both with the base graphic utilities that come with R, and the `ggplot2` package from the `tidyverse` that has taken over the world (er, revolutionized how we write R code). Both of these are actually great to work with for different reasons. The base graphics are built-in, powerful, and give you 100% full control over your graphic environment. The `ggplot2` library (and a million packages that people have written to work with it) takes these to a new level in terms of functionality and 95% of the time it will do exactly what you need. That other 5% of the time it is really great to be able to fall back on the base graphics.

For the examples in this chapter, we'll work with the water quality data contained in `physical.csv`. You will need to download the class data sets that go with this book to play along if you have not already (click here for instructions from the course website). But, you should have downloaded those to complete the examples in Chapter 2.

We will walk through histograms, scatter plots, line graphs, and boxplots in base graphics and `ggplot2` in this Chapter. Later in the book, we will add examples of how to plot predictions from statistical models alongside raw data using these same tools.

If you installed the `tidyverse` successfully in Chapter 3, then you can load all the packages we'll need by including `library(tidyverse)` at the start of your script:

```
# Chapter 4 Lecture module

# Package load
```

```
library(tidyverse)

# 4.2 Plotting with base R ----
# ...
```

4.1 Plots matter as much as stats

Before we get started:

1. There are few statistical tests that hold intuitive meaning to our readers. The ability to present information in a visually meaningful way to the reader can help to make the interpretation of your science crystal clear without having to worry about whether or not your reader has the faculties to interpret the results of some fancy statistical test that *you* think is cool.

2. Effects and effect sizes are often (if not always) more important than the ability to detect 'significant' differences. If you can present clear evidence that some treatment or manipulation confers a biologically meaningful change in a visual way alongside these tests, you can provide a much stronger body of evidence with which to argue your case.

3. There are a few graphical tools that are very useful for basic data exploration, diagnostics, etc., that can make your life a lot easier for data analysis and interpretation. They can also help you decide whether something has gone terribly wrong.

The takeaway here is: *don't make shitty graphs.*

4.2 Plotting with base R

Let's look at a few simple types of plots in R. The default graphics in R are not much to look at. But, there are a **ton** of ways to modify these plots, and the user (that's you!) can build plots from the ground up if needed.

One of the great things about base graphics is that many of the plot types take the same, or similar arguments that are all based on shared graphical parameters.

You can access the help file for these shared graphical parameters by running `?pars` in the console. We will use many of these in the sections that follow.

4.2.1 Histograms

Let's start with the histogram function that we began playing with at the end of Chapter 3.

The `hist()` function plots a histogram but it actually does a whole lot more than just that. Like other plotting utilities, it can take a wide variety of arguments and it actually does some basic data analysis behind the scenes. All of the arguments are optional or have default values with the exception of the data that we want to plot (a numeric variable). This is the case for most plotting functions in the base graphics for R.

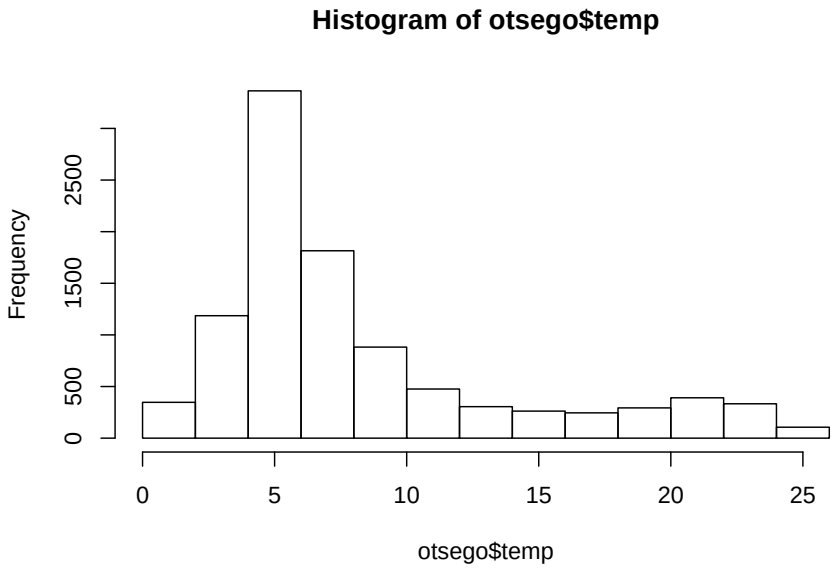
Start by reading in the data contained within the `physical.csv` file from the class data folder. Remember, I am assuming that your code is inside of a folder that also contains your class data folder that you named `data`.

```
# I added stringsAsFactors = FALSE to read in all
# text strings as `chr`.
 otsego <- read.csv("data/physical.csv", stringsAsFactors = FALSE)
```

These are data collected each year from Otsego Lake by students, staff, and faculty at the SUNY Oneonta Biological Field Station in Cooperstown, NY, USA. The data set includes temperature (°C), pH, dissolved oxygen, and specific conductance measurements from a period of about 40 years. There are all kinds of cool spatial and seasonal patterns in the data that we can look at. We will use `temperature` for the examples that follow.

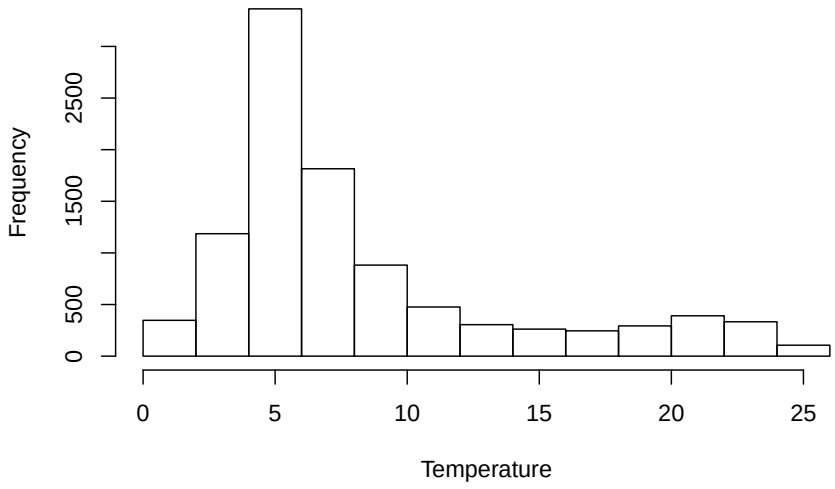
Make a histogram of temperature across all depths and dates just to see what we are working with here:

```
hist(otsego$temp)
```



The default histogram in base graphics leaves much to be desired. Thankfully, it is easy to modify the look of these figures. For example, we can add labels to the x and y-axis using `xlab` and `ylab`, and we can give the histogram a meaningful title by adding `main = ...` to the `hist()` call or remove it completely by saying `main = ""`.

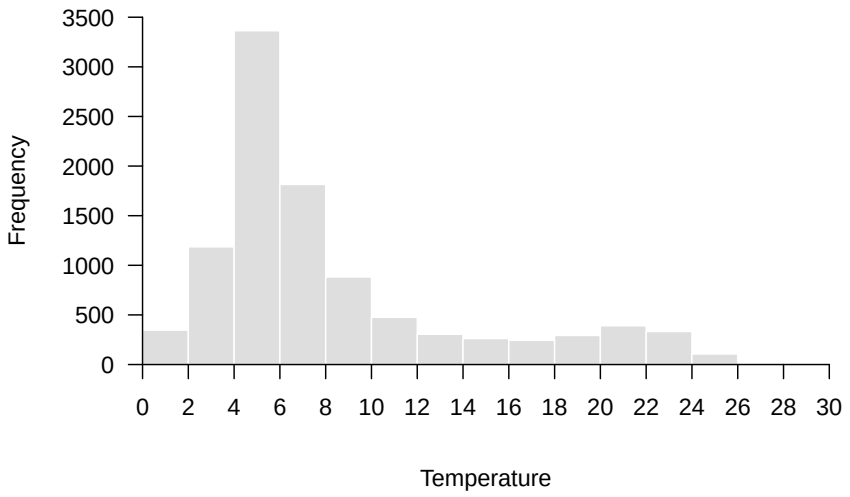
```
hist(otsego$temp, xlab = "Temperature", ylab = "Frequency", main="")
```



We can make the axes cross at zero if we are concerned about that. We need to do this by specifying `yaxt = "n"`, `xaxt = "n"` in the `hist()` call and then follow up by telling R exactly where to start each of the axes. In this example, I also add some changes to the color of the bars (`col`) and the color of the borders (`border`). Finally, I fix the x- and y-axis scales so I know where they'll start.

```
# Make the histogram
hist(otsego$temp,
     xlab = "Temperature",
     ylab = "Frequency",
     main = "",
     xaxt = "n",
     yaxt = "n",
     col = "gray87",
     border = "white",
     xlim = c(0, 30),
     ylim = c(0, 3500)
     )

# Add an x-axis going from zero to thirty degrees
# in increments of 2 degrees and start it at zero
axis(side = 1, at = seq(from = 0, to = 30, by = 2), pos = 0)

# Add a rotated y-axis with default scale and
# start it at zero
```

```
axis(side = 2, las = 2, pos = 0)
```



Colors!!!

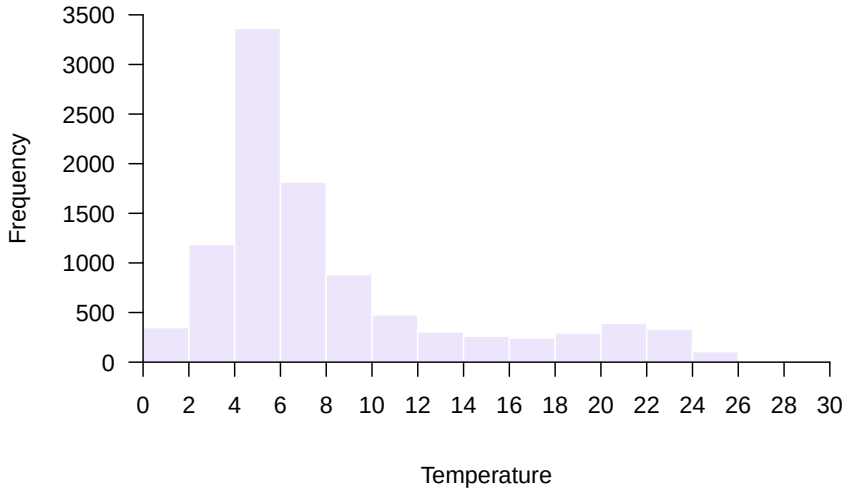
If `gray87` is not your style (whatevs), there are another 656 pre-named colors in R. You can see their names by running `colors()` in the console like this:

```
colors()
```

If you are a little more adventurous, you might try the `rgb()` color specification or hex values. I really like the `rgb()` specification because you can include an `alpha` channel to make your colors transparent (oooooh!). For example, if I change my code above to use the following:

```
col = rgb(red = 0.90, green = 0, blue = 0.30, alpha = 0.10)
```

I get a transparent, purple histogram.



So purply.

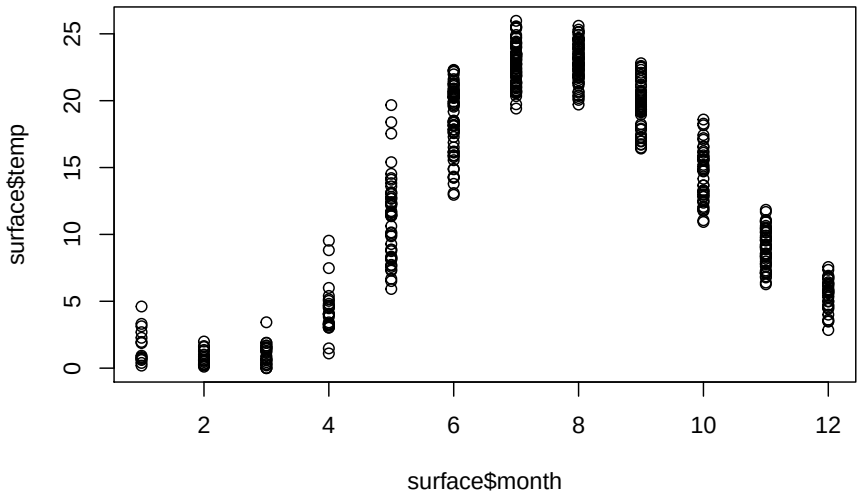
There are tons of great blogs and eBooks with whole chapters devoted to colors and color palletes in R. There are even whole packages we'll work with dedicated to colors. By all means, check them out! We will work with a few as we continue to increase complexity.

4.2.2 Scatterplots

Scatter plots are a great starting point for doing exploratory data analysis or for displaying raw data along with summary graphics. They are also the default behavior for the `plot()` function for continuous variables in base R.

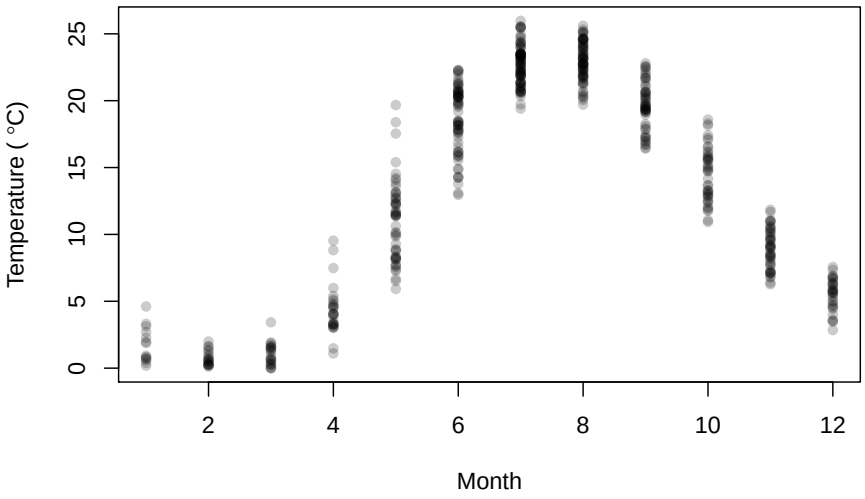
Let's demonstrate by by plotting surface temperature (`depth` = 0.1 m) by month across years. We'll use the data management skills we picked up in Chapter 3 to filter the data first.

```
# Filter to get July surface temperatures
surface <- otsego %>% filter(depth == 0.1)

# Default scatter plot
plot(x = surface$month, y = surface$temp)
```



As with the `hist()` function, the default here is underwhelming. We can use many of the same arguments that we specified in `hist()` to dress this up a bit. This time, we will specify a plotting character `pch` that corresponds to a filled circle. Then, we tell R to give it an `rgb()` background (`bg`) with no color for lines that go around each point. That way the data points are darker where there is overlap between them. Finally, we use `expression()` to include the degree symbol in the y-axis label.

```
# Better scatter plot
plot(x = surface$month,
     y = surface$temp,
     pch = 21,
     bg = rgb(0, 0, 0, 0.2),
     col = NA,
     xlab = "Month",
     ylab = expression(paste("Temperature ( ", degree, "C)"))
     )
```



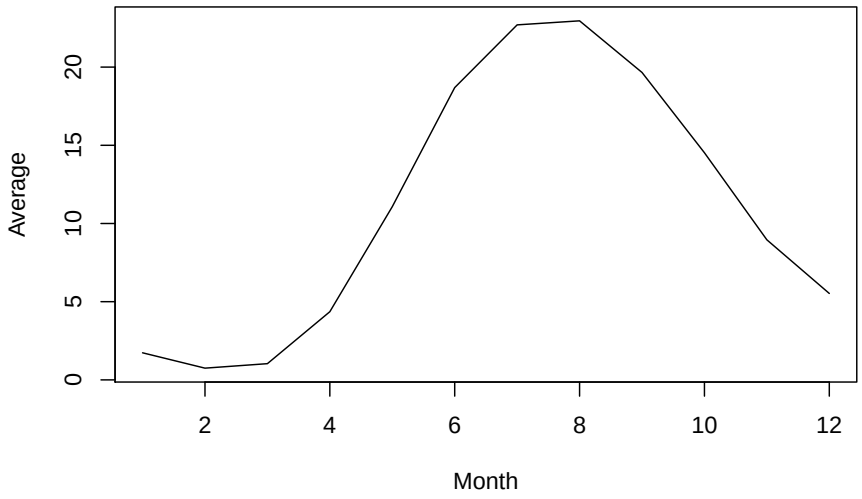
This is a lot more informative because it shows us where most of the observations fall within a given month, and how much variability there is. But, it would be nice to have some summary.

4.2.3 Lines

We can plot lines in a few different ways in the base graphics of R. We can create stand-alone line graphs with data in R pretty easily with the `plot()` function we used for scatter plots in the preceding section.

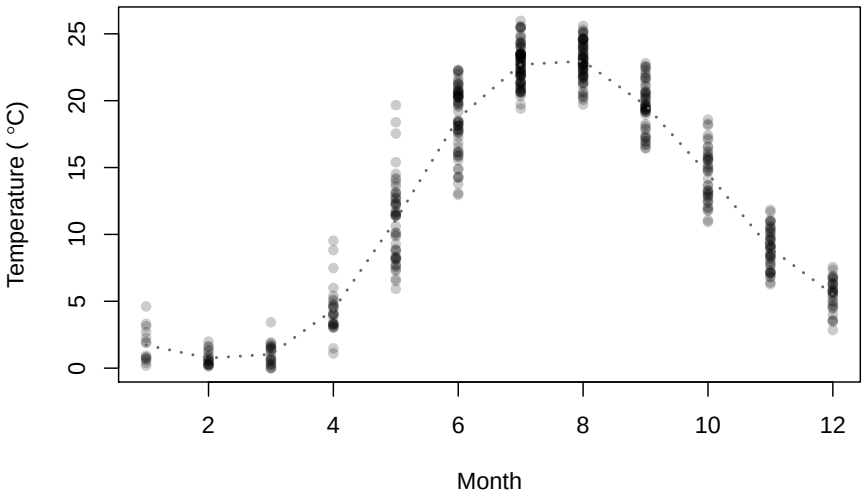
For example, let's say that we want to just plot average surface temperature in each month as a line graph. We can summarize the data quickly and then plot those:

```
mids <- surface %>%
  group_by(month) %>%
  summarize(avg = mean(temp))

plot(mids$month, mids$avg, type = "l", xlab = "Month", ylab = "Average")
```



We could even add these to the scatter plot of our raw data using the `lines()` function. Play around with `lty` and `lwd` to see if you can figure out what they do. If you get stuck, don't forget to Google it! (Worst Stats Text eveR.)

```
# Same scatter plot
plot(x = surface$month,
     y = surface$temp,
     pch = 21,
     bg = rgb(0, 0, 0, 0.2),
     col = NA,
     xlab = "Month",
     ylab = expression(paste("Temperature ( ", degree, "C)"))
     )
# Add a thick, dotted line that is gray (this is a gray40 job)
lines(mids$month, mids$avg, lty = 3, lwd = 2, col = "gray40")
```



We could also add the means to the main plot with `points()` and choose a different size or color than the raw data. We'll look at these options and more as we step up complexity.

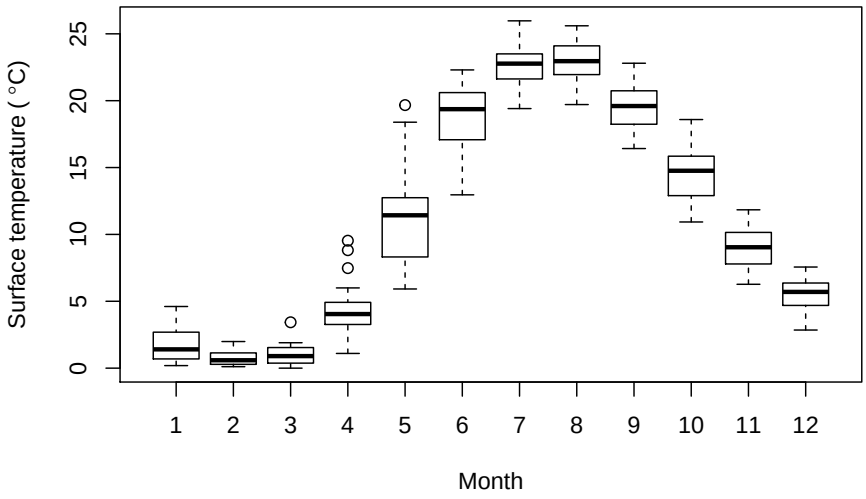
For raw data like these, though, we are better off using a box plot to show those types of summaries.

4.2.4 Boxplots

The basic box plot is straightforward to create, but can be a real pain to modify because the syntax is slightly different from the other plotting functions we've worked with so far.

Let's try to summarize surface temperature by month using a box plot to see how these work in the base R graphics. Notice that we are specifying the variables as a formula here, and explicitly telling R what the data set is:

```
boxplot(temp~month, data = surface,
        xlab = "Month",
        ylab = expression(paste("Surface temperature ( ", degree, "C)")))
```



Wow, that was waaaaay to easy! Have you ever tried to make one of those in Excel? Forget about it. It would take you half a day, and then when you realized you forgot ten data points you would have to do it all again.

But, it is still much ugly. Maybe there is a way we can change that?

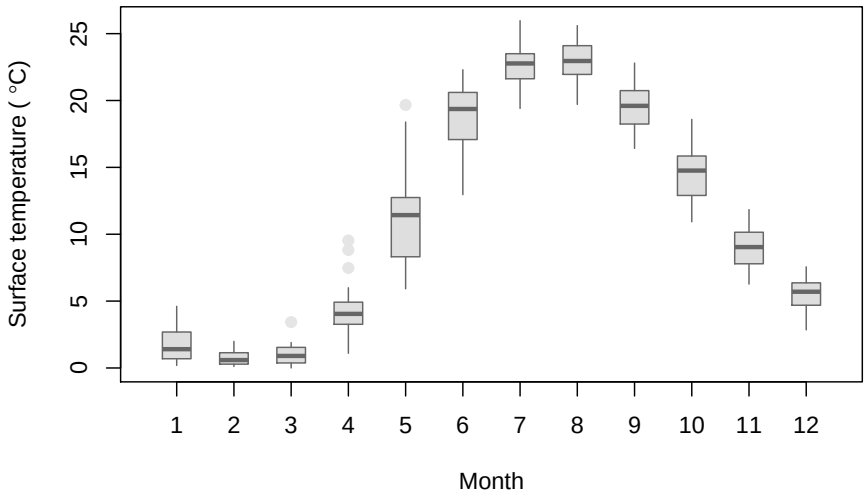
Of course there is!

Let's add a little color and tweak some options. For a full set of optional arguments you can change, run `?bxp` in the console. (Ooh, that one is sneaky: `bxp()` is the function inside of `boxplot()` that actually draws the plots).

Options are named consistently by the part of the plot. For example, `boxwex`, `boxcol` and `boxfill` all control the look of the box. Likewise, the options `boxcol`, `whiskcol` and `staplecol` control the colors of the box, whiskers, and staples, respectively. Nifty, right? Play with the settings below to see what each one does. Then, go explore some more options.It is the easiest way to learn when you are learning from The Worst Stats Text eveR.

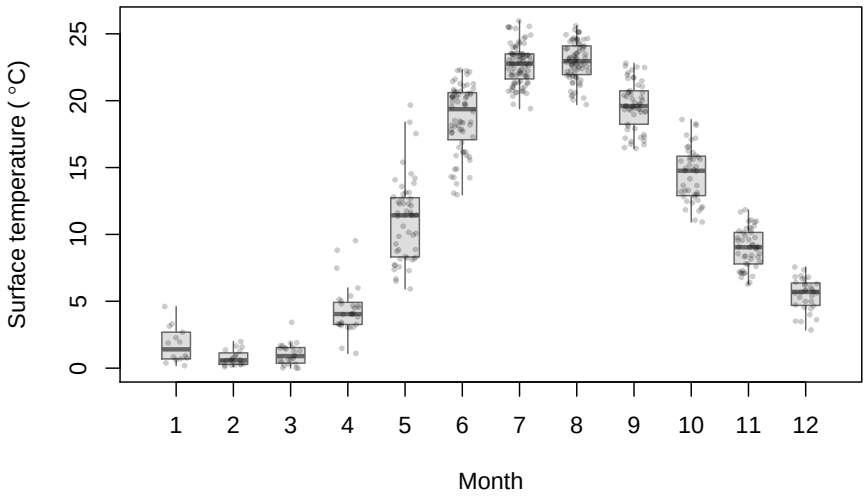
```
boxplot(temp~month,
        data = surface,
        xlab = "Month",
        ylab = expression(paste("Surface temperature ( ", degree, "C)")),
        border = "gray40",
        boxwex = 0.50, boxcol = "gray40", boxfill = "gray87",
        whisklty = 1, whisklwd=1, whiskcol = "gray40",
        staplewex = 0, staplecol = NA,
```

```
        outpch = 21, outbg = "gray90", outcol = "gray90"
        )
```



Finally, we can combine this with a scatter plot to `jitter` our raw data over the top of the boxes in each month:

```
boxplot(temp~month,
        data = surface,
        xlab = "Month",
        ylab = expression(paste("Surface temperature ( ", degree, "C)")),
        border = "gray40",
        boxwex = 0.50, boxcol = "gray40", boxfill = "gray87",
        whisklty = 1, whisklwd=1, whiskcol = "gray40",
        staplewex = 0, staplecol = NA,
        outpch = 21, outbg = NA, outcol = NA
        )
points(jitter(surface$month), surface$temp, cex=.4, pch=19, col=rgb(0,0,0,0.2))
```



That is actually starting to look pretty snazzy! We'll continue to work to improve our base graphics as we move forward. For now, let's have a look at how to do these things in `ggplot2` next.

4.3 Plotting with `ggplot2`

Plotting with `ggplot2` and the dozens of packages that use it is a bit different than plotting with base graphics in R. Part of the reason for this is that it uses a work flow that is similar to the data manipulation we have looked at so far. In general, you could think of this as creating a canvas, applying some routine aesthetics based on the data structure (e.g. grouping), and then adding layers on to the canvas like we did with base graphics.

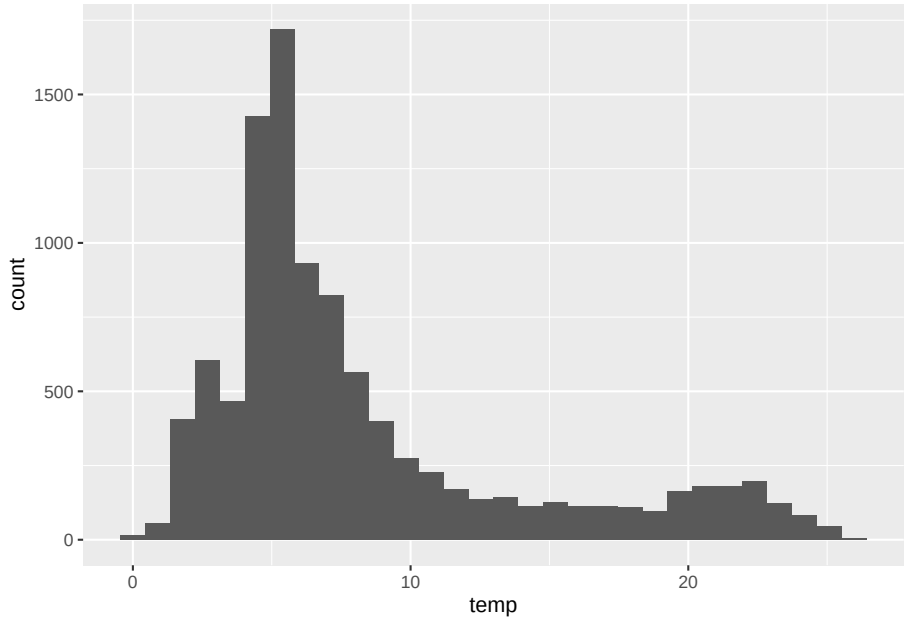
It takes a little getting used to, but you'll see how powerful it can be for multi-faceted plots when we get to later chapters. We'll walk through the same plots that we did in base graphics, but this time we'll use the `ggplot()` function and layer on the pieces.

4.3.1 Histograms

The default histogram is easy to create with `ggplot()` and a geometry layer. We start with the `ggplot` call, and then add the histogram geometry, `geom_histogram()`, like this. I usually save the plot to an object with an

arbitrary name I don't use for anything, like `p` or `s` or `v`, and then print it explicitly.

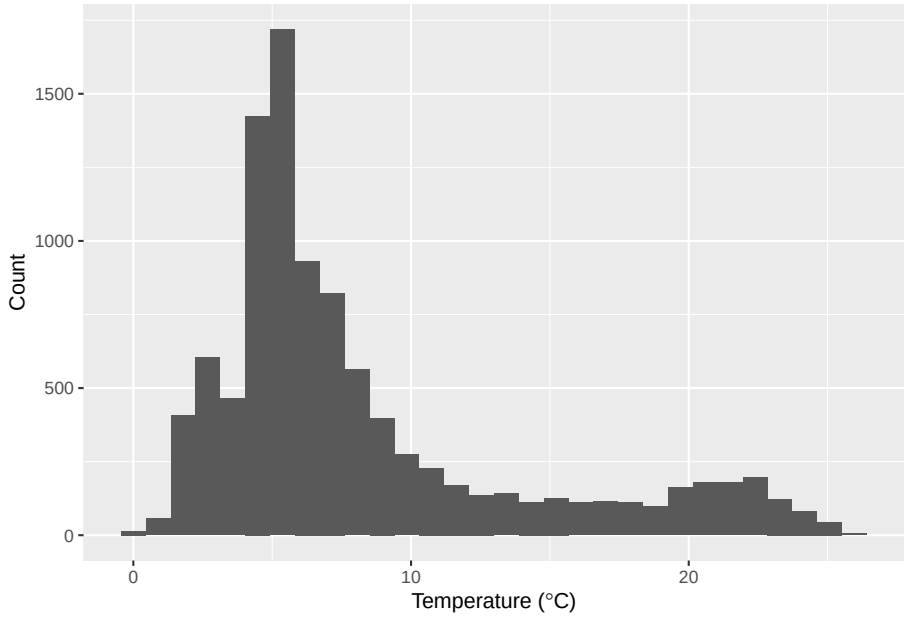
```
# Histogram of water temperature across
# all dates and depths
p <- ggplot(otsego, aes(x=temp)) + geom_histogram(bins=30)

print(p)
```



Right away this looks a lot prettier than the default histogram that we produced with base graphics. Of course, we can customize just like we did before.

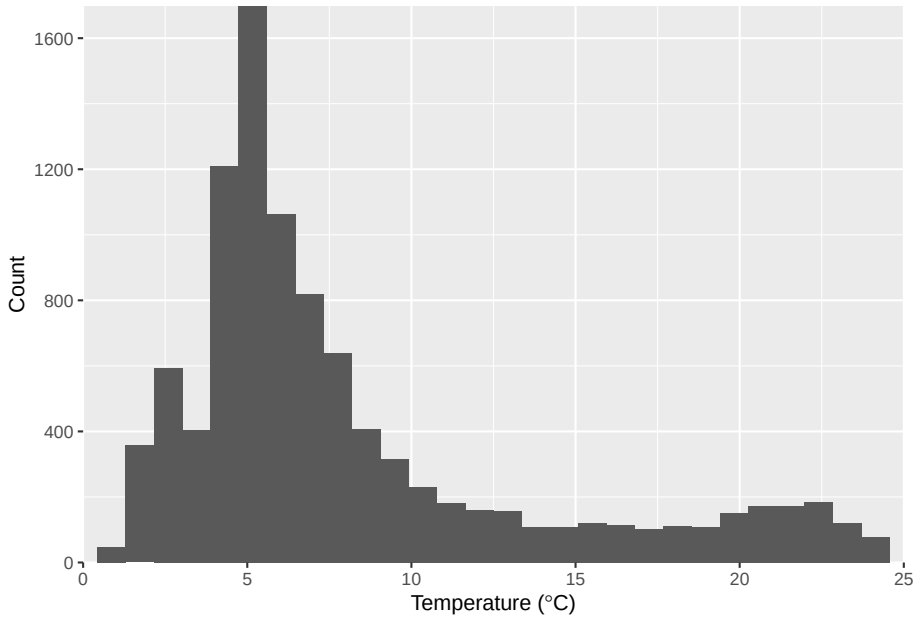
Let's add labels for the x and y-axis next:

```
# Histogram of water temperature across
# all dates and depths
p <- ggplot(otsego, aes(x=temp)) +
  geom_histogram(bins=30) +
  xlab(expression(paste("Temperature (", degree, "C)"))) +
  ylab("Count")

print(p)
```



We can also scale the `x-axis` and the `y-axis` like we did in the base graphics example.

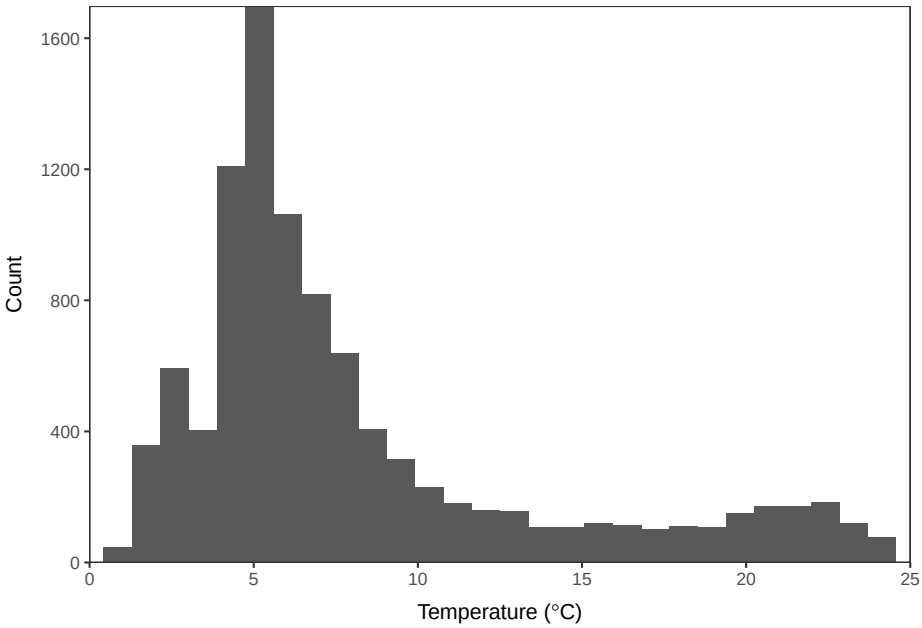
```
# Histogram of water temperature across
# all dates and depths
p <- ggplot(otsego, aes(x=temp)) +
  geom_histogram(bins=30) +
  xlab(expression(paste("Temperature (", degree, "C)"))) +
  ylab("Count") +
  scale_x_continuous(limits=c(0, 25), expand = c(0, 0)) +
  scale_y_continuous(expand = c(0, 0))
print(p)
```



We can modify each of other layers individually, all at once using preset ggplot themes or by modifying a pre-defined theme.

Let's have a look at how a theme changes the appearance. I am going to add `theme_bw()` here, but check out the others linked above. I also add a few adjust the position of the x- and y-axis labels and removed the panel grid in the `theme()` function after applying a theme.

```
# Histogram of water temperature across
# all dates and depths
p <- ggplot(otsego, aes(x=temp)) +
  geom_histogram(bins=30) +
  scale_x_continuous(limits=c(0, 25), expand = c(0, 0)) +
  scale_y_continuous(expand = c(0, 0)) +
  xlab(expression(paste("Temperature (", degree, "C)"))) +
  ylab("Count") +
  theme_bw() +
  theme(
    axis.title.x = element_text(vjust = -1),
    axis.title.y = element_text(vjust = 3),
    panel.grid = element_blank()
  )
print(p)
```



Spend some time practicing this and changing options to see what you can come up with. Be sure to check out the descriptions of the options you can pass to theme by running `?theme` to get the help file.

4.3.2 Scatter plots

Now that we have a basic feel for the `ggplot2` work flow, changing plot types is really easy because all of the parts of our plots work together in the same way.

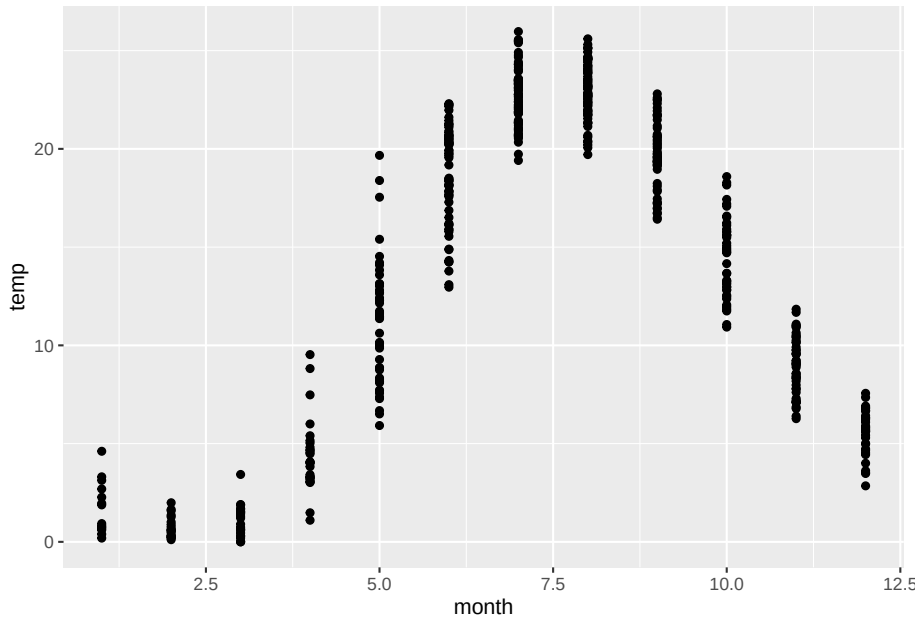
As a reminder, we previously built scatter plots of surface temperature in Otsego Lake, NY by month using base graphics.

Go ahead and subset the data again:

```
surface <- otsego %>% filter(depth == 0.10)
```

Now, we can make the default scatterplot:

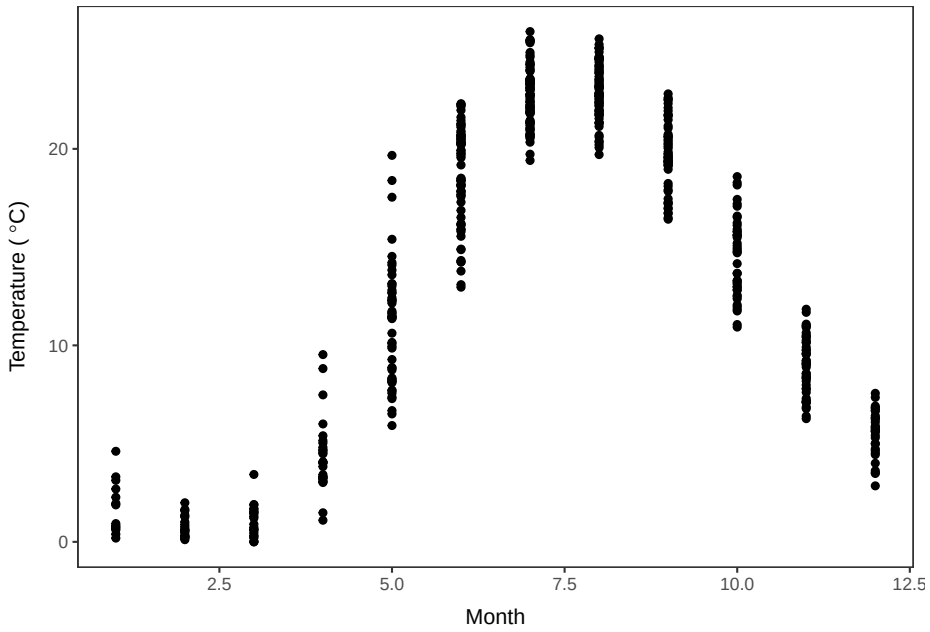
```
s <- ggplot(surface, aes(x = month, y = temp)) +
  geom_point()
print(s)
```



At a glance, this already looks a lot nicer than the default scatterplots from base graphics, but we still have a lot of work to do. Plus, we get ggplots own odd behaviors when it comes to the x-axis scale and titles. So, let's get to work!

First, we'll replace the default axis titles and add the `theme_bw()` that we used above, with the same modifications to axis positions and grid lines.

```
s <- ggplot(surface, aes(x = month, y = temp)) +
  geom_point() +
  xlab("Month") +
  ylab(expression(paste("Temperature ( ", degree, "C)"))) +
  theme_bw() +
  theme(axis.title.x = element_text(vjust = -1),
        axis.title.y = element_text(vjust = 3),
        panel.grid = element_blank()
  )
print(s)
```



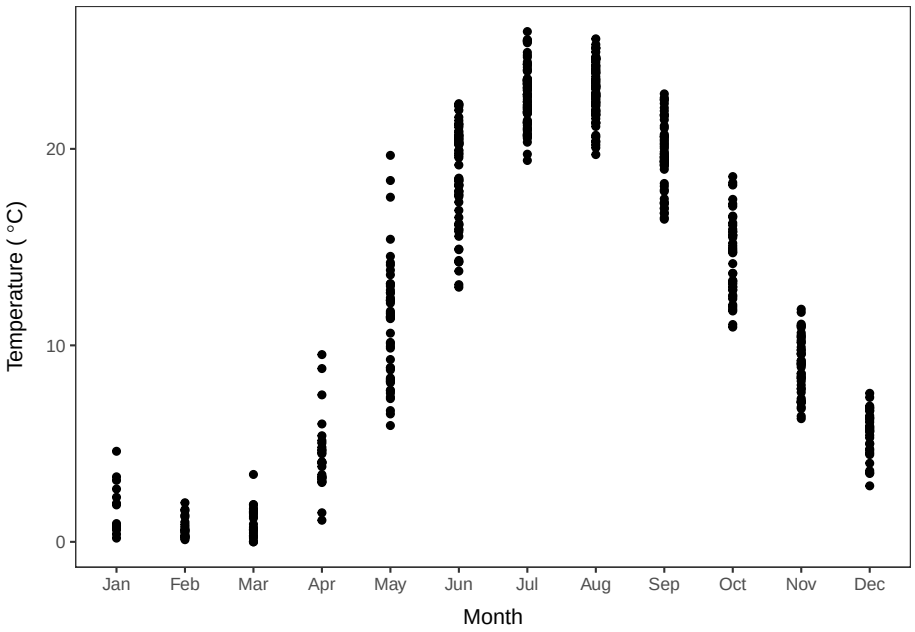
Okay, now we need to fix that pesky x-axis scale to use whole months or text labels.

To fix the axis scales, we've actually got to do a little bit of work this time. In this case the easiest thing to do is probably to make a categorical variable out of the column `month`, which is an integer. We can do this using some fancy indexing with the built-in object that contains month abbreviations, `month.abb` in base R.

```
surface$c_month <- factor(month.abb[surface$month], levels=month.abb)
```

Whoa, that was a heavy lift (sarcasm). Let's see how that changes the appearance of our plot:

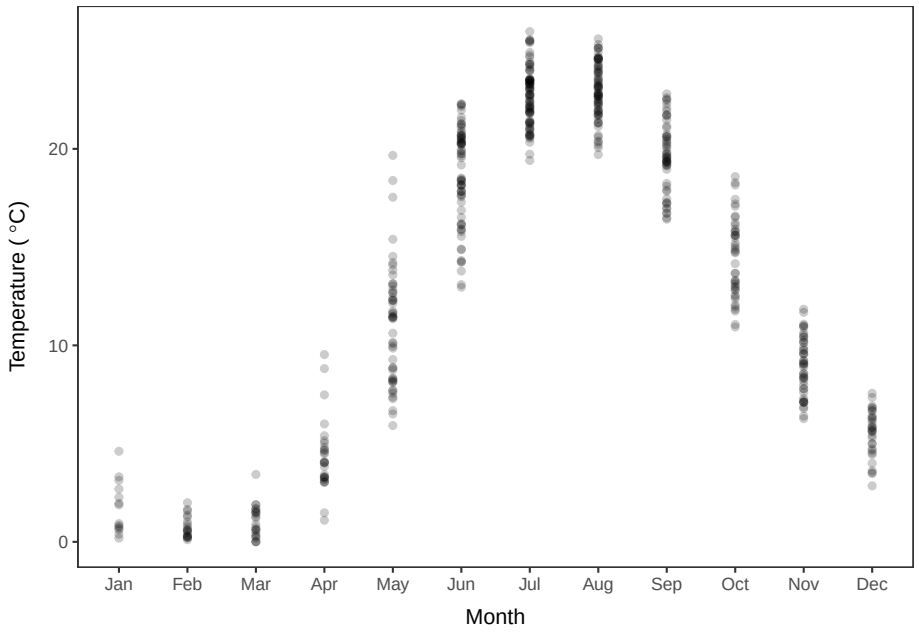
```
s <- ggplot(surface, aes(x = c_month, y = temp)) +
  geom_point() +
  xlab("Month") +
  ylab(expression(paste("Temperature ( ", degree, "C)"))) +
  theme_bw() +
  theme(axis.title.x = element_text(vjust = -1),
        axis.title.y = element_text(vjust = 3),
        panel.grid = element_blank()
  )
print(s)
```



This is starting to look really nice.

Finally, we just add a little transparency to the points by specifying `alpha = 0.2` inside of `geom_point()` and we are good to go!

```
s <- ggplot(surface, aes(x = c_month, y = temp)) +
  geom_point(alpha = 0.2) +
  xlab("Month") +
  ylab(expression(paste("Temperature ( ", degree, "C)"))) +
  theme_bw() +
  theme(axis.title.x = element_text(vjust = -1),
        axis.title.y = element_text(vjust = 3),
        panel.grid = element_blank()
  )
print(s)
```



Looks just like the one we made with base graphics!

4.3.3 Lines

Most of the time we plot line graphs, whether in base graphics or using `ggplot2`, we are going to be adding them to existing plots. This was really straightforward in base graphics. It is only slightly more complicated in `ggplot2`.

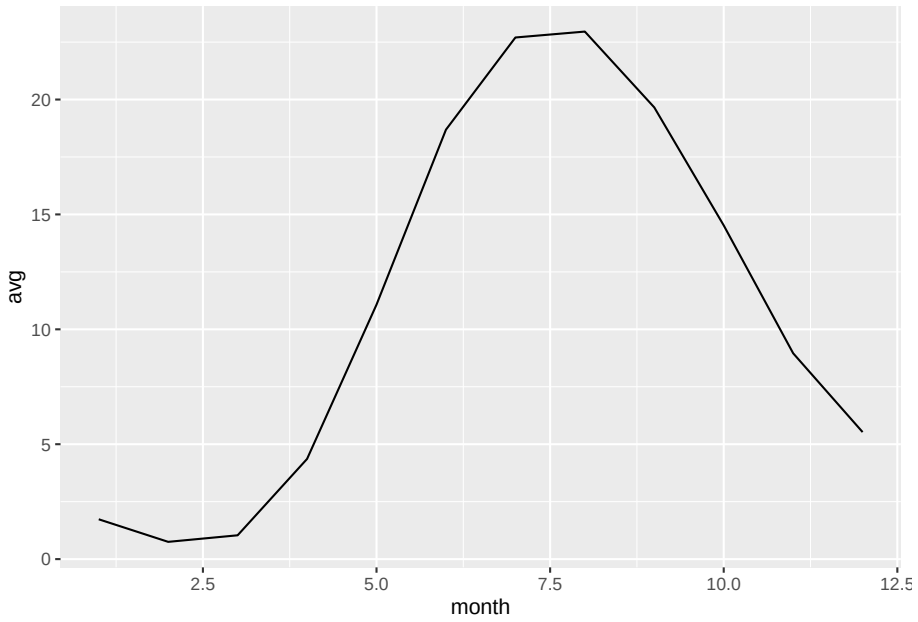
We'll start with the default line graph, and then add it to the scatter plot from the previous section.

Let's calculate monthly means of surface temperature in Otsego Lake again:

```
mids <- surface %>%
  group_by(month) %>%
  summarize(avg = mean(temp))
```

Now plot it with `ggplot()`:

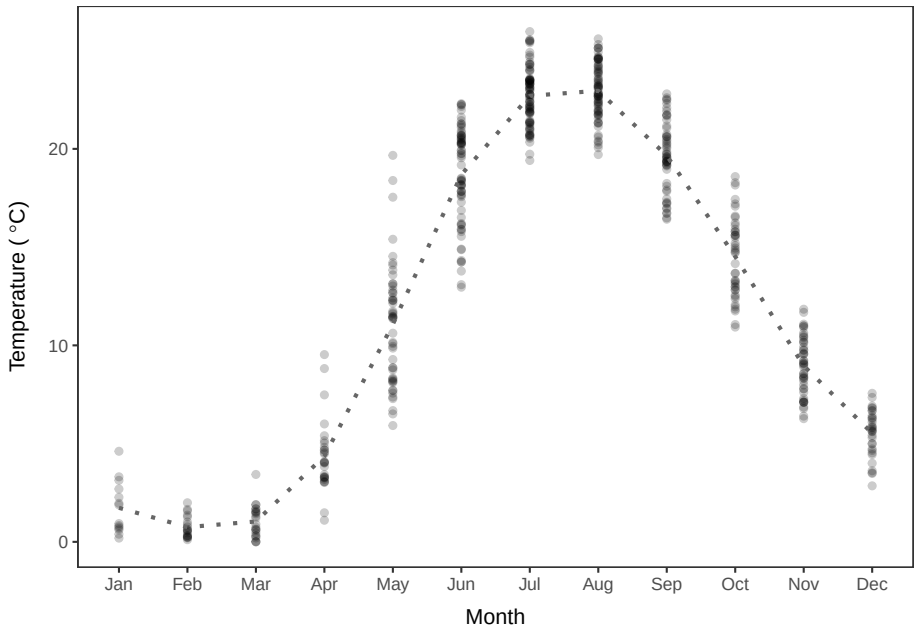
```
lp <- ggplot(mids, aes(x = month, y = avg)) +
  geom_line()

print(lp)
```



There you have it!

Now, we just need to add this to our scatterplot that we made previously. To do this, we have to insert `geom_line()` in the code, but we must specify

```
s <- ggplot(data = surface, mapping = aes(x = c_month, y = temp)) +
  geom_point(alpha = 0.20) +
  geom_line(mapping = aes(x = month, y = avg),
            data = mids,
            color = 'gray40',
            lty = 3,
            lwd = 1) +
  xlab("Month") +
  ylab(expression(paste("Temperature ( ", degree, "C)"))) +
  theme_bw() +
  theme(axis.title.x = element_text(vjust = -1),
        axis.title.y = element_text(vjust = 3),
        panel.grid = element_blank()
  )

print(s)
```



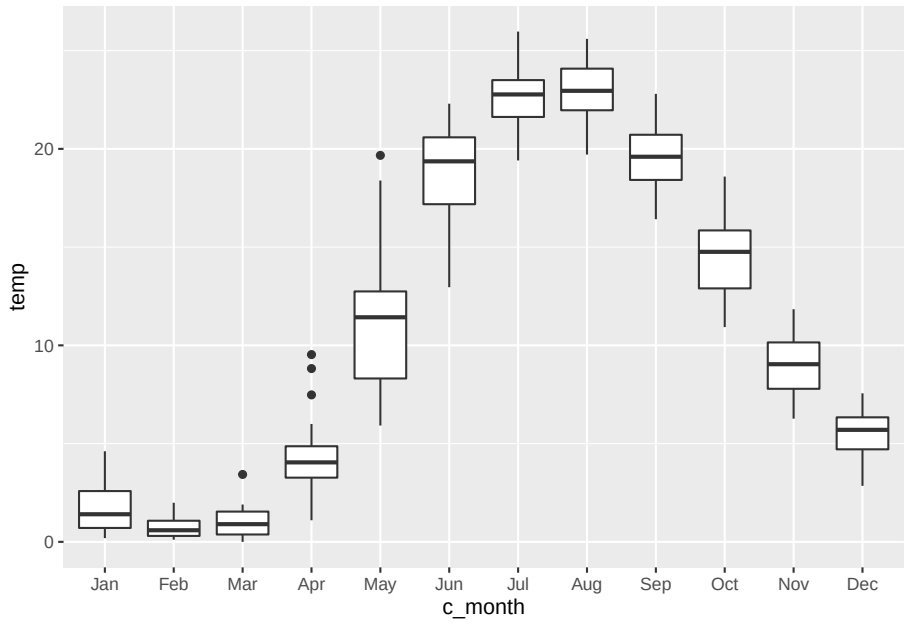
We will continue to use this approach throughout the book to plot raw data and model predictions. So, if it is giving you trouble now, spend some extra time with it.

4.3.4 Boxplots and

To wrap up our tour of plotting examples in `ggplot2`, we will reproduce (more or less) the box plots we made in base graphics.

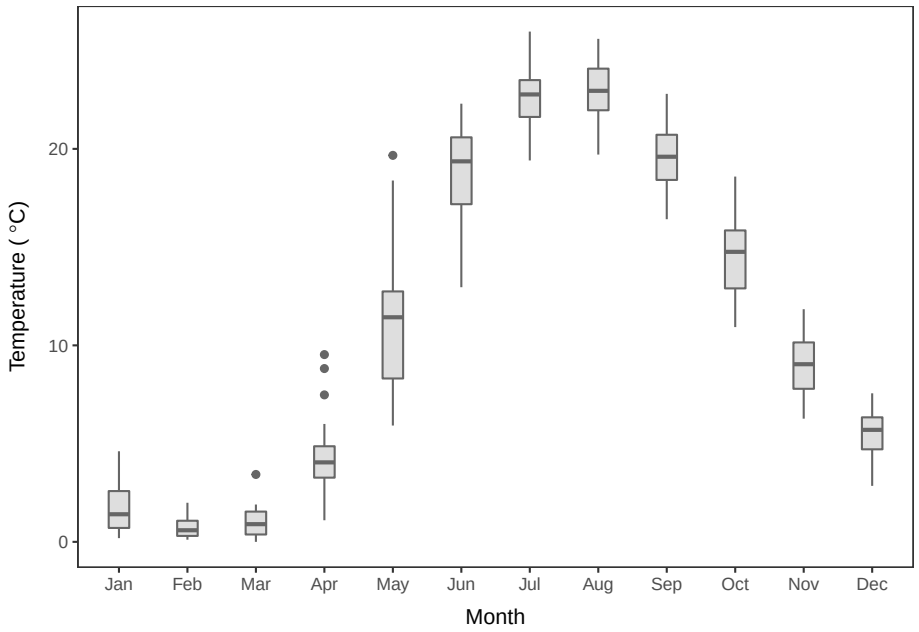
Make the default box plot of surface water temperatures in Otsego Lake, NY. Notice that we use the `c_month` variable that we made previously in the `surface` data so R knows these are groups.

```
bp <- ggplot(surface, aes(x = c_month, y = temp)) + geom_boxplot()

print(bp)
```



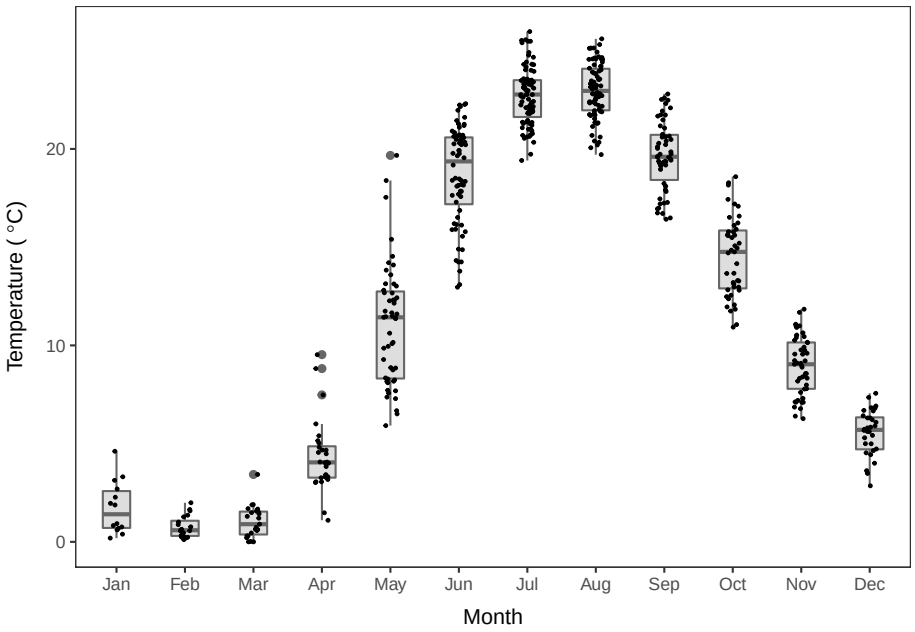
If we add changes we made to previous plots here, then we can get a cleaner look:

```
bp <- ggplot(surface, aes(x = c_month, y = temp)) +
  geom_boxplot(color = 'gray40', fill = 'gray87', width = 0.3) +
  xlab("Month") +
  ylab(expression(paste("Temperature ( ", degree, "C)"))) +
  theme_bw() +
  theme(axis.title.x = element_text(vjust = -1),
        axis.title.y = element_text(vjust = 3),
        panel.grid = element_blank()
  )

print(bp)
```



And, of course, we can add our "jittered", raw data points over the top to show the spread.

```
bp <- ggplot(surface, aes(x = c_month, y = temp)) +
  geom_boxplot(color = 'gray40', fill = 'gray87', width = 0.4) +
  geom_jitter(size = .5, width = 0.1) +
  xlab("Month") +
  ylab(expression(paste("Temperature ( ", degree, "C)"))) +
  theme_bw() +
  theme(axis.title.x = element_text(vjust = -1),
        axis.title.y = element_text(vjust = 3),
        panel.grid = element_blank()
  )

print(bp)
```



4.4 Next steps

Hopefully this chapter provided you with a basic overview of plotting in R. If you struggled with these exercises, practice them again, or check out some additional online resources. In the coming chapters, we will continue to add functionality and complexity to how we use these kinds of plots. We'll look at how to compare two groups within a single plot in Chapter 6 when we dive into sampling distributions. And, eventually we will learn how to plot predictions from our statistical model against observed data from Chapter 8 onward.

Chapter 5

Sampling distributions in R

If we can describe the shape of a probability distribution for a random variable like temperature we can make predictions about the world. Sinister? Maybe. These are temperatures that I simulated from the Hudson River using historical data to estimate parameters of a multivariate normal distribution (muwahahaha).

In this Chapter, we'll talk about probability and probability distributions as a backdrop for the models that we will be working with during the next several chapters. When we describe these from data we have collected, we call them **sampling distributions**. Probability theory is central to statistical techniques, so it will be important for you to have a pretty firm understanding of this to grab hold of big ideas later on. For now, play along and try to understand how they work. We'll swing back later for a refresher.

In order to complete this Chapter, you will need to have the `ggplot2`, `MASS`, and `Rlab` packages loaded. The only one you should need to install is the `Rlab` package because `MASS` is installed when you install R, and we already installed `ggplot2` with the `tidyverse` in Chapter3.

I'm going to load these now. In general, it is good practice to put these at the top of the script so we know they are needed.

```
library(ggplot2)
library(MASS)
library(Rlab)
```

None of the class data are required to complete this chapter.

5.1 What are sampling distributions?

When we talk about sampling distributions, we are talking about the probability that a variable we can measure (e.g. temperature) takes on some value. In most cases, there is a higher probability that the variable will take on certain values than others. That probability may be governed by any number of processes and thus may assume a number of different shapes with respect to the likelihood of any given value of our variable. The differences in the shapes that we assume, and the mathematical parameters that we use to describe those shapes are called "probability distributions". And, when they are estimated from data, they are sampling distributions.

There was a time when biologists were largely restricted to using models that relied heavily on the assumption that the things we measured, and their errors, followed "normal" distributions, which you have probably heard of or seen in a scientific paper. This was because of how computationally intensive other methods were. This often led to the use of strictly parametric tools like ANOVA and t-tests, or the use of strictly non-parametric tools like frequency analyses and rank-order methods. While these are still useful techniques in our toolboxes, that time has passed, and now we have access to a wide range of tools that allow us to extend simple parametric and non-parametric tools to relax or change distributional assumptions. We will discuss these throughout the book, but we need to look at the underlying distributions that govern our decisions about which of these tools to use. So, this week we'll look at a few probability distributions that correspond to sampling distributions we frequently encounter in biology. To wrap-up, we will use this new information to talk about how we calculate descriptive statistics such as means and standard deviations from samples.

5.2 Probability distributions in R

R has a number of built-in distribution types, and there are random-number generators associated with most or all of these that will allow us to take random samples from a distribution (like picking numbers out of a hat!). This is useful for data simulation, but is also helpful for us to learn about probability distributions and how their parameters affect the shape, spread, scale, location, etc. of those distributions. We will briefly discuss concepts like skew because of how they can help us think about the assumptions that we are making (or breaking!) in the models that we use.

For this class, we will focus on one major family of distributions and then zero in on a few distributions within this family that you are guaranteed to encounter throughout your career.

5.3 Exponential family

Most or all of the distributions we will use for this class come from the **exponential family** of distributions.

The exponential family is very flexible, and whether you know it or not, it includes most of the distributions one might use to describe every day phenomena. It includes most of the probability distributions with which you are familiar, and many more. Just ask this *very* reliable Wikipedia entry. Oh, let's face it, you were going there anyway, I just cut out the Google step.

Take a look at the table at the bottom of this Wikipedia page just to get an idea of how many distributions are included within the exponential! Holy cow! We're not going to look at all of these in this class- I just want you to be aware that this is a **huge** family of specific distributions.

Distributions that we'll focus on in this chapter:

1. Continuous distributions Normal (Gaussian) Lognormal Beta Uniform
2. Discrete distributions Bernouli Binomial Multinomial Poisson Negative binomial

5.4 Continuous distributions

The normal distribution

This is one distribution with which most of you have at least some nodding acquaintance. It is the classic "bell curve" that college students once dreaded in upper-level courses. I don't know if it's a thing anymore. Go Google it.

The **normal distribution** is defined by two parameters:

1. The mean (μ)
2. The variance (σ^2)

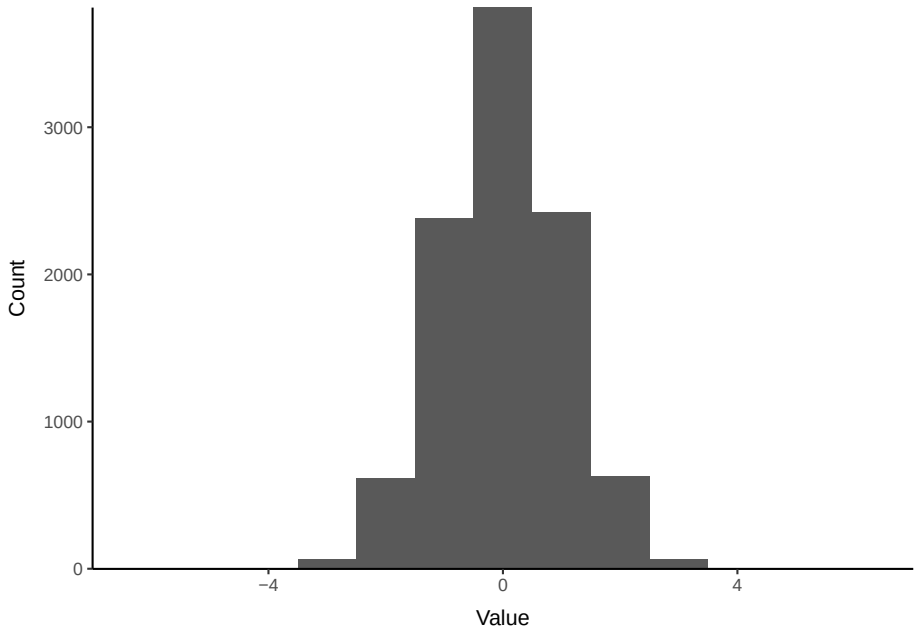
Let's take a look at what the normal distribution looks like. We'll start with a special one called the standard normal (or z) distribution. The standard normal is a normal distribution with a mean of zero and a variance of 1. This one is really cool because the standard deviation (σ) is the square-root of the variance, and in this special case $\sqrt{1} = 1$, so the variance and standard deviation are equal! And because of this property, and other normal distribution can be converted to a standard normal using z-standardization, which we'll talk about later. How exciting is that?

First, take a sample from a normal distribution:

```
samp <- rnorm(n = 10000, mean = 0, sd = 1)
```

Now, plot a histogram using the sick new skills you got in Chapter 4

```
p <- ggplot() +
  geom_histogram(aes(samp), binwidth = 1) +
  scale_x_continuous(limits=c(-7,7), expand = c(0, 0)) +
  scale_y_continuous(expand = c(0, 1)) +
  xlab("Value") +
  ylab("Count") +
  theme_classic() +
  theme(
    axis.title.x = element_text(vjust = -1),
    axis.title.y = element_text(vjust = 3),
    panel.grid = element_blank()
  )
print(p)
```



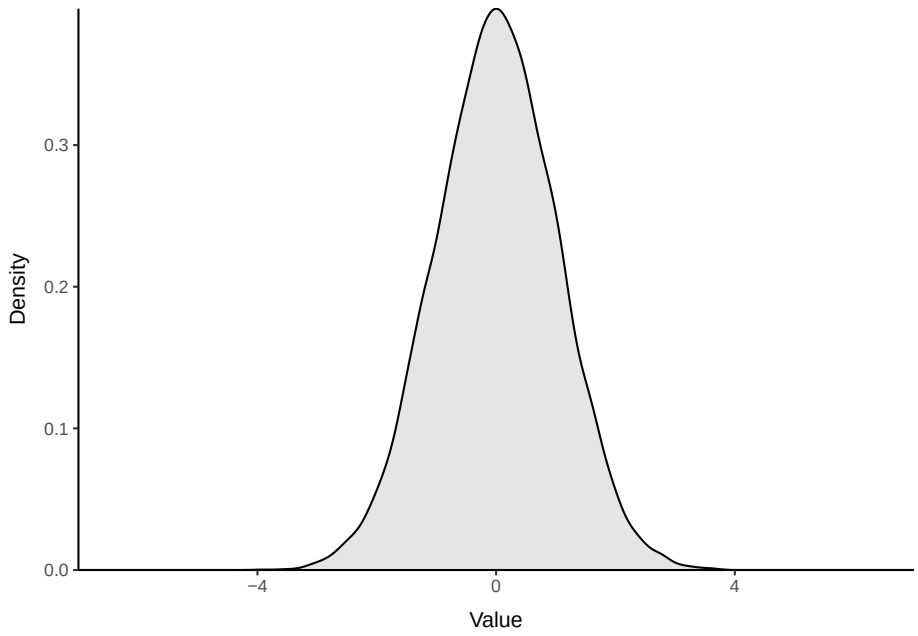
Pretty!

Because this sample is from a **continuous** distribution, we might actually wish to represent this distribution with a probability density function. You can think of this as R calculating the relative probability of an given value. It implies a continuous surface, rather than the discrete bins like the histogram. In reality it doesn't matter because at best we chop continuous distributions into tiny little

bins when we do things like integrals, and R bases `binwidth` in histograms off a density function anyway (aaaaah!).

Density plots are a new one for us, so let's try them out. If you scroll back and forth, you'll notice that the code below is basically identical to the histogram above except for labels and scales. We just replaced the histogram geometry (`geom_histogram`) with a density-based geometry (`geom_density`). Here, we use `fill = 1` to trick R into filling the area under the line because we have no grouping variables. By default, this is interpreted as `'black'`, so we add an alpha channel for transparency.

```
p <- ggplot() +
  geom_density(aes(samp), alpha = .1, fill = 1) +
  scale_x_continuous(limits = c(-7,7), expand = c(0, 0)) +
  scale_y_continuous(expand = c(0, 0)) +
  xlab("Value") +
  ylab("Density") +
  theme_classic() +
  theme(
    axis.title.x = element_text(vjust = -1),
    axis.title.y = element_text(vjust = 3),
    panel.grid = element_blank()
  )
print(p)
```



Excellent use of `ggplot()` to make a figure that looks like the clunky base graphics. Maybe you can improve on it in the homework assignment?

We can change the parameters of the standard normal to change both the location and the scale of our distribution. The influence of changing the mean, or average, on the location of a distribution is perhaps obvious. But, the influence of variance may be less intuitive, so let's have a look!

Create two more random samples, one with a larger `sd` and one with a smaller `sd`, to see how this changes the shape of the distribution:

```
samp2 <- rnorm(n = 1e4, mean = 0, sd = 2)
samp3 <- rnorm(n = 1e4, mean = 0, sd = .5)
```

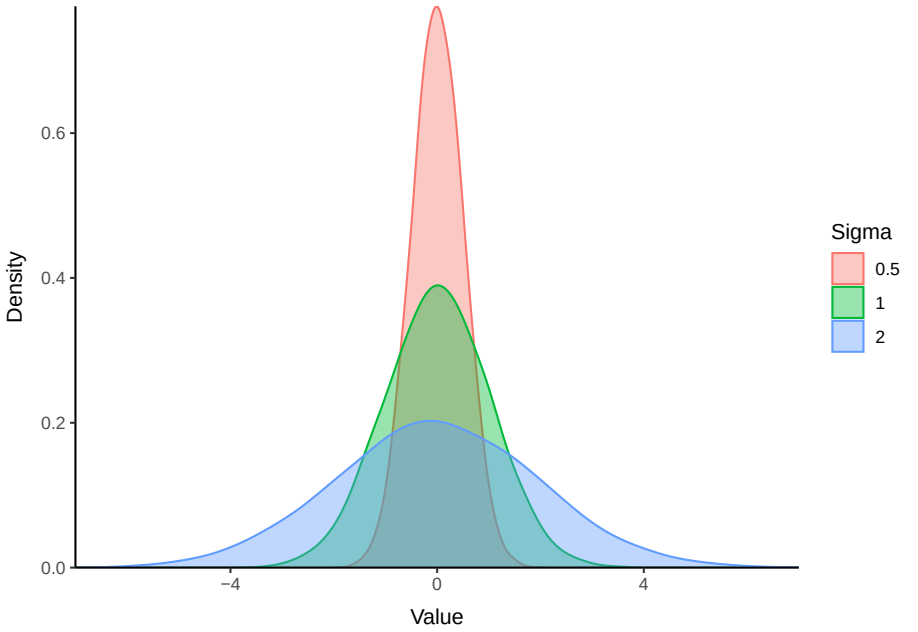
Let's put them in a data frame with `samp` so they're easy to plot. We combine all three random samples into one column called `Value`. Then, we create a column to hold the standard deviation used in each sample (`Sigma`). If we make that one into a factor, we can use the `Sigma` columns to plot the samples as separate lines by tweaking our plotting code.

```
normals <- data.frame(
  Value = c(samp, samp2, samp3),
  Sigma = factor(
    c(
      rep(1, length(samp)),
      rep(2, length(samp2)),
      rep(.5, length(samp3))
    )
  )
)
```

Next, we can just add these to the plot to compare the sampling distributions. This time, we tell R to fill the area under our lines based on sample ID with a default color scheme by saying `fill = Sigma` in our `ggplot()` call. We also added `color = Sigma` to make the lines the same default colors. Remember, you can specify your own.

```
p <- ggplot(data = normals,
            aes(x = Value, group = Sigma, fill = Sigma, color = Sigma)) +
  geom_density(adjust = 1.5, alpha = .4) +
  scale_x_continuous(limits = c(-7, 7), expand = c(0, 0)) +
  scale_y_continuous(expand = c(0, 0)) +
  xlab("Value") +
  ylab("Density") +
  theme_classic() +
  theme(
```

```
    axis.title.x = element_text(vjust = -1),
    axis.title.y = element_text(vjust = 3),
    panel.grid = element_blank()
  )
print(p)
```



The blue polygon in the plot above shows a distribution with greater variance than our z distribution (green). The red polygon shows a distribution with a smaller variance. Hopefully this helps demonstrate how variance influences the scale of the distribution.

5.4.1 The lognormal distribution

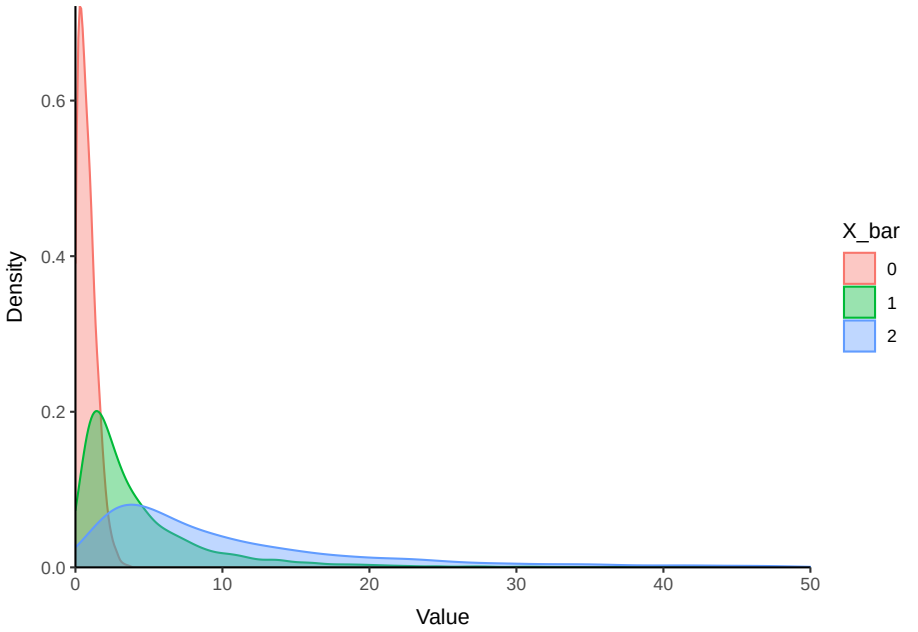
The **lognormal distribution** is a probability distribution that assumes our random variable is normally distributed on the **log scale**. This assumption allows us to incorporate **skew** into the normal distribution and change the location and scale of the normal distribution by transforming the parameters (μ and σ) onto the log scale. This is one of the more common data transformations that you will run into, e.g.: "We log-transformed the data to achieve normality…". One of the other reasons for that is that all values (positive or negative) transformed from the log to the real scale are positive, so it helps prevent us from making negative predictions about phenomena or variables that can't be less than zero.

Let's take a look at how changes to the mean change the location of this distribution:

```
# Create random samples from log-normal
# distributions with different means
samp1 <- rlnorm(n=1e4, mean=0, sd=1)
samp2 <- rlnorm(n=1e4, mean=1, sd=1)
samp3 <- rlnorm(n=1e4, mean=2, sd=1)

# Put them in a data frame with the values
# of the means used to create them
lognormals <- data.frame(
  Value = c(samp, samp2, samp3),
  X_bar = factor(
    c(
      rep(0, length(samp)),
      rep(1, length(samp2)),
      rep(2, length(samp3))
    )
  )
)
```

Now you can plot these using the code above with a couple of modifications to show how the mean of the log-normal distribution influences the location.

```
p <- ggplot(data = lognormals,
            aes(x = Value, group = X_bar, fill = X_bar, color = X_bar)) +
  geom_density(adjust = 1.5, alpha = .4) +
  scale_x_continuous(limits =c(0, 50), expand = c(0, 0)) +
  scale_y_continuous(expand = c(0, 0)) +
  xlab("Value") +
  ylab("Density") +
  theme_classic() +
  theme(
    axis.title.x = element_text(vjust = -1),
    axis.title.y = element_text(vjust = 3),
    panel.grid = element_blank()
  )
print(p)
```



You can see that the relative scale of these three distributions is similar, but the location shifts to the right on our x-axis as the value of `X_bar` (the mean) increases. Note also how this affects *kurtosis*.

5.4.2 The beta distribution

The **beta distribution** is a probability distribution that is constrained to the interval [0, 1]. But, it is incredibly flexible in its parameterization, and as a result is very useful for stochastic simulation of variables on the probability scale, such as survival.

The parameters of the beta distribution are α and β, or commonly `a` and `b` or `shape 1` and `shape 2` in R. Within this distribution, α pushes the distribution to the right (toward 1), and β pushes the distribution back toward the left (toward 0). The relative magnitude of α and β determine the location, shape, and scale of the probability distribution for our random variable. When α and β are equal, and greater than 1, the beta distribution looks like a normal distribution within the interval [0, 1].

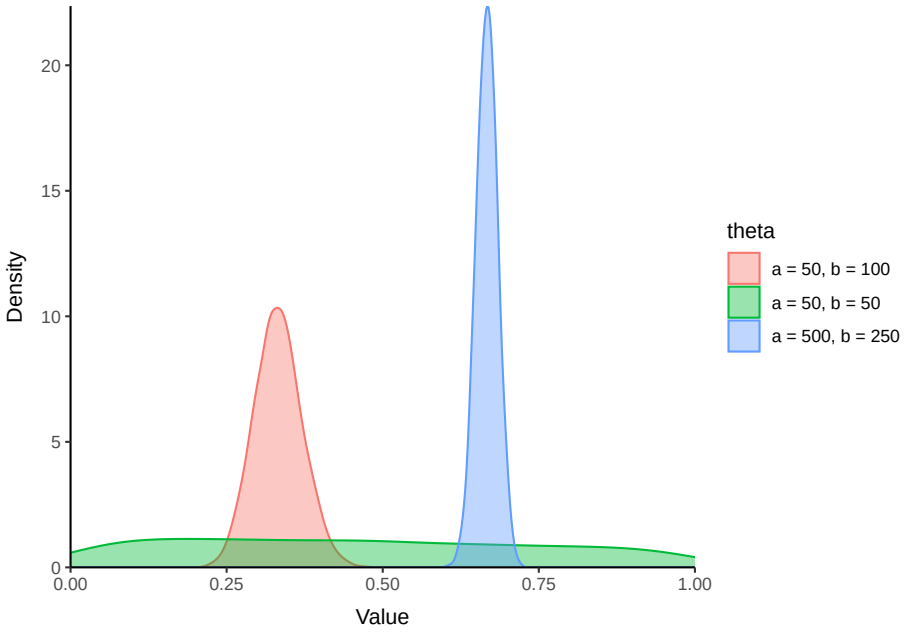
Let's take a look:

```
# Simulate random values from 3 different beta distributions
# so we can compare them
samp1 <- rbeta(n=1e4, shape1=50, shape2=50)
samp2 <- rbeta(n=1e4, shape1=50, shape2=100)
```

```
samp3 <- rbeta(n=1e4, shape1=500, shape2=250)

# Put them in a data frame with the values
# of the means used to create them. I am
# using "theta" because often that is how we
# refer collectively to a group of parameters
betas <- data.frame(
  Value = c(samp, samp2, samp3),
  theta = factor(
    c(
      rep('a = 50, b = 50', length(samp)),
      rep('a = 50, b = 100', length(samp2)),
      rep('a = 500, b = 250', length(samp3))
    )
  )
)
```

And then, we can plot them just like we did above. Copy and paste it - change what you need. Isn't code great?. Just don't forget to change the scale and the data in the plotting code!

```
p <- ggplot(data = betas,
            aes(x = Value, group = theta, fill = theta, color = theta)) +
  geom_density(adjust = 1.5, alpha = .4) +
  scale_x_continuous(limits =c(0, 1), expand = c(0, 0)) +
  scale_y_continuous(expand = c(0, 0)) +
  xlab("Value") +
  ylab("Density") +
  theme_classic() +
  theme(
    axis.title.x = element_text(vjust = -1),
    axis.title.y = element_text(vjust = 3),
    panel.grid = element_blank()
  )
print(p)
```



Play around with these to see what kind of cool shapes you can make and where they are located within the range between zero and one.

5.5 Discrete distributions

Discrete probability distributions are useful for situations in which our random variable of interest can only take specific values within the interval of interest. For example, this might include age, counts, pass/fail, or any number of conceivable categories. As a result, these require a slightly different treatment of probability as a discrete, rather than continuous phenomenon. (Think back to our histogram that we started with in this chapter.)

5.5.1 Bernoulli

The **Bernoulli distribution** is a special case of the binomial distribution with a single trial (see below for clarification). Bernoulli outcomes are those for which the variable we are measuring can take on one of two values: a one or a zero. This distribution is useful for visualizing processes such as coin flips, yes/no responses, live/dead endpoints in lab studies, and a number of other very interesting phenomena. The Bernoulli distribution has a single parameter: the probability of success, but the number of successful outcomes is also governed by sample size: n, which R calls `size` because `n` was already taken.

We can simulate data from a Bernoulli distribution in one of two ways in R.

The "old-school" way of doing this was to draw from a binomial with a single **trial**. Here we randomly draw a single sample from a binomial with a single trial, and a 50% chance of success. We'll use the example of hatching chicken eggs with some probability of success. If you are boring, you can think about flipping coins, too!

We'll start with one chicken egg that has a 50% chance of successfully hatching (probability of success = 0.50).

```
rbinom(n=1, size=1, prob=.5)
```

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

There is also a function called `rbern` in the `Rlab` package that simplifies this for the specific case of a Bernoulli.

Let's do it again with that function:

```
# Hatch one egg with 50% success rate
rbern(n = 1, prob = .5)
```

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

Or we could hatch a whole bunch of eggs:

```
# Hatch ten eggs, each with p = 0.5
rbern(n = 10, prob = .5)
```

```
##  [1] 1 1 1 0 0 0 0 0 0 1
```

Then, we could even count how many of those were successful. Do you remember how to do that? There are several different ways. You'll have to come up with one for the homework assignment (hint: see Chapter 2).

5.5.2 Binomial

The **binomial distribution** is pretty similar to the Bernoulli distribution. In fact, the Bernoulli is just a special kind of binomial. The binomial includes a parameter called N (`size` in R) which corresponds to a number of trials per sample. We assume that this is 1 in the case of Bernoulli. In most cases in biology, it will suffice to use the Bernoulli, but for modeling we will want to understand the binomial for things like random stratified designs and nested

models that rely on the use of binomial distribution. Later in your career, you might even get into cool models that estimate N as a latent state to estimate population size (for example). Plus, using the binomial is way faster and can be more precise for certain regression applications [okay, that one should probably have a citation, but this is The Worst Stats Text eveR, so go Google it].

To sample data from a binomial distribution, we can use `rbinom` from base R. In this example we tell R that we want 10 samples (`n`) from a binomial distribution that has 10 trials (`size`) and a probability of success (`prob`) of 0.5. This is like hatching ten eggs from each of ten chickens instead of just one chicken laying ten eggs.

```
# Take a random draw of 10 samples
# from a binomial distribution with 10 trials
# and probability of success equal to 0.50
rbinom(n=10, size=10, prob=0.5)
```

```
##  [1] 5 5 5 4 7 6 4 5 3 8
```

Remember as you look through these that your numbers should look different than mine (at least most of the time) because these are being generated randomly.

5.5.3 Multinomial

The **multinomial distribution** is a further generalization of the Binomial and Bernoulli distributions. Here, there are one or more possible categorical outcomes (states), and the probability of each one occurring is specified individually **but all of them must sum to one**. The categories are, in this case, assumed to be a **mutually exclusive** and **exhaustive** set of possible outcomes.

We can use the multinomial distribution to randomly sample from categories (imagine our response variable is a categorical variable, like the names of the students in this class).

To do this, we need to read in the `s_names.csv` file from our `data` folder that is definitely in your working directory (**remember to set your working directory first**).

Read in the data file with `stringsAsFactors = FALSE` for purposes of demonstrating with categorical variables (not factors).

```
s_names <- read.csv('data/s_names.csv', stringsAsFactors = FALSE)
```

Next, let's assign the variable `name` in `s_names` to a vector for simplicity.

```
name <- s_names$name
```

Then, we can assign a uniform probability of drawing any given name if we divide one by the number of names.

```
# Calculate probability of getting a given
# name based on the length of the vector
prob_each <- 1 / length(name)

# Repeat this probability for each
# name in our vector of names
probs <- rep(prob_each, times = length(name))
probs
```

```
##  [1] 0.03846154 0.03846154 0.03846154 0.03846154 0.03846154 0.03846154
##  [7] 0.03846154 0.03846154 0.03846154 0.03846154 0.03846154 0.03846154
## [13] 0.03846154 0.03846154 0.03846154 0.03846154 0.03846154 0.03846154
## [19] 0.03846154 0.03846154 0.03846154 0.03846154 0.03846154 0.03846154
## [25] 0.03846154 0.03846154
```

This shows us that the probability of drawing any of the individual names is {r prob_each}.

Now, we can sample from a multinomial distribution using our objects. Here we are taking 5 samples from the distribution, each time we sample there is only one trial, and we are sampling with the 26 probabilities above.

Have a look:

```
rmultinom(n=5, size=1, prob=probs)
```

```
##       [,1] [,2] [,3] [,4] [,5]
##  [1,]    0    0    0    0    0
##  [2,]    1    0    0    0    0
##  [3,]    0    0    0    0    1
##  [4,]    0    0    0    0    0
##  [5,]    0    0    0    0    0
##  [6,]    0    0    0    0    0
##  [7,]    0    0    0    0    0
##  [8,]    0    0    0    0    0
##  [9,]    0    0    0    0    0
## [10,]    0    0    0    0    0
## [11,]    0    0    0    0    0
## [12,]    0    0    0    0    0
```

```
## [13,]    0    0    0    0    0
## [14,]    0    0    0    0    0
## [15,]    0    0    1    0    0
## [16,]    0    0    0    0    0
## [17,]    0    1    0    0    0
## [18,]    0    0    0    1    0
## [19,]    0    0    0    0    0
## [20,]    0    0    0    0    0
## [21,]    0    0    0    0    0
## [22,]    0    0    0    0    0
## [23,]    0    0    0    0    0
## [24,]    0    0    0    0    0
## [25,]    0    0    0    0    0
## [26,]    0    0    0    0    0
```

WHOA a matrix??!!! **What does it all mean**?

Take a step back, breathe, and think about this. The rows in this matrix are you and your classmates. If we took one random sample from the multinomial distribution, it would look like this:

```
# Take a single sample from
# the list of student names
rmultinom(n=1, size=1, prob=probs)
```

```
##       [,1]
##  [1,]    0
##  [2,]    0
##  [3,]    0
##  [4,]    0
##  [5,]    0
##  [6,]    0
##  [7,]    0
##  [8,]    0
##  [9,]    0
## [10,]    0
## [11,]    0
## [12,]    0
## [13,]    0
## [14,]    0
## [15,]    0
## [16,]    0
## [17,]    0
## [18,]    0
## [19,]    0
## [20,]    0
```

```
## [21,]    0
## [22,]    0
## [23,]    0
## [24,]    1
## [25,]    0
## [26,]    0
```

Here, we pulled a single sample from the distribution, and probability of sampling a given individual was 0.04 (1/26). If it makes it easier, we can put your names next to it:

```
cbind(name, rmultinom(n=1, size=1, prob=probs))
```

```
##       name
##  [1,] "Ava"         "0"
##  [2,] "Dillon"      "0"
##  [3,] "Delaney"     "0"
##  [4,] "Manolo"      "0"
##  [5,] "Sarah"       "0"
##  [6,] "Shannon"     "0"
##  [7,] "Olivia"      "0"
##  [8,] "Ebony"       "0"
##  [9,] "Julia"       "0"
## [10,] "Davi"        "0"
## [11,] "Gabrielle"   "0"
## [12,] "Jordan"      "0"
## [13,] "Tayler"      "0"
## [14,] "Summer"      "0"
## [15,] "Leah"        "0"
## [16,] "Christine"   "0"
## [17,] "Ashley"      "0"
## [18,] "Katherine"   "0"
## [19,] "James"       "0"
## [20,] "Emily"       "0"
## [21,] "Cassidy"     "0"
## [22,] "Maximillion" "0"
## [23,] "Sierra"      "0"
## [24,] "Kyle"        "0"
## [25,] "Diana"       "1"
## [26,] "Amanda"      "0"
```

Now, if I was calling on you randomly in class, after 10 questions, the spread of people who would have participated in class might look like this (or whatever you got - remember, it is random):

```
cbind(name, rmultinom(n=10, size=1, prob=probs))
```

```
##       name
##  [1,] "Ava"         "0" "0" "0" "0" "0" "0" "0" "0" "0" "0"
##  [2,] "Dillon"      "0" "0" "0" "0" "0" "0" "0" "0" "1" "0"
##  [3,] "Delaney"     "0" "0" "0" "0" "0" "0" "0" "0" "0" "0"
##  [4,] "Manolo"      "0" "0" "0" "0" "0" "0" "0" "0" "0" "0"
##  [5,] "Sarah"       "0" "0" "0" "0" "1" "0" "0" "0" "0" "1"
##  [6,] "Shannon"     "0" "0" "0" "1" "0" "0" "0" "0" "0" "0"
##  [7,] "Olivia"      "0" "0" "0" "0" "0" "0" "0" "0" "0" "0"
##  [8,] "Ebony"       "0" "0" "0" "0" "0" "0" "0" "0" "0" "0"
##  [9,] "Julia"       "0" "0" "1" "0" "0" "0" "0" "0" "0" "0"
## [10,] "Davi"        "0" "0" "0" "0" "0" "0" "0" "0" "0" "0"
## [11,] "Gabrielle"   "0" "1" "0" "0" "0" "0" "0" "0" "0" "0"
## [12,] "Jordan"      "0" "0" "0" "0" "0" "0" "0" "0" "0" "0"
## [13,] "Tayler"      "1" "0" "0" "0" "0" "0" "0" "0" "0" "0"
## [14,] "Summer"      "0" "0" "0" "0" "0" "0" "0" "0" "0" "0"
## [15,] "Leah"        "0" "0" "0" "0" "0" "0" "0" "0" "0" "0"
## [16,] "Christine"   "0" "0" "0" "0" "0" "0" "0" "0" "0" "0"
## [17,] "Ashley"      "0" "0" "0" "0" "0" "0" "0" "0" "0" "0"
## [18,] "Katherine"   "0" "0" "0" "0" "0" "0" "0" "1" "0" "0"
## [19,] "James"       "0" "0" "0" "0" "0" "0" "0" "0" "0" "0"
## [20,] "Emily"       "0" "0" "0" "0" "0" "0" "0" "0" "0" "0"
## [21,] "Cassidy"     "0" "0" "0" "0" "0" "0" "1" "0" "0" "0"
## [22,] "Maximillion" "0" "0" "0" "0" "0" "0" "0" "0" "0" "0"
## [23,] "Sierra"      "0" "0" "0" "0" "0" "0" "0" "0" "0" "0"
## [24,] "Kyle"        "0" "0" "0" "0" "0" "1" "0" "0" "0" "0"
## [25,] "Diana"       "0" "0" "0" "0" "0" "0" "0" "0" "0" "0"
## [26,] "Amanda"      "0" "0" "0" "0" "0" "0" "0" "0" "0" "0"
```

Taking this one step further, we could just draw a name and stop looking at these ugly (no but really they are **awesome**!) matrices:

```
name[which(rmultinom(n=1, size=1, prob=probs)>0)]
```

```
## [1] "Christine"
```

And now we have a way to randomly select an individual based on a multinomial distribution. What fun!

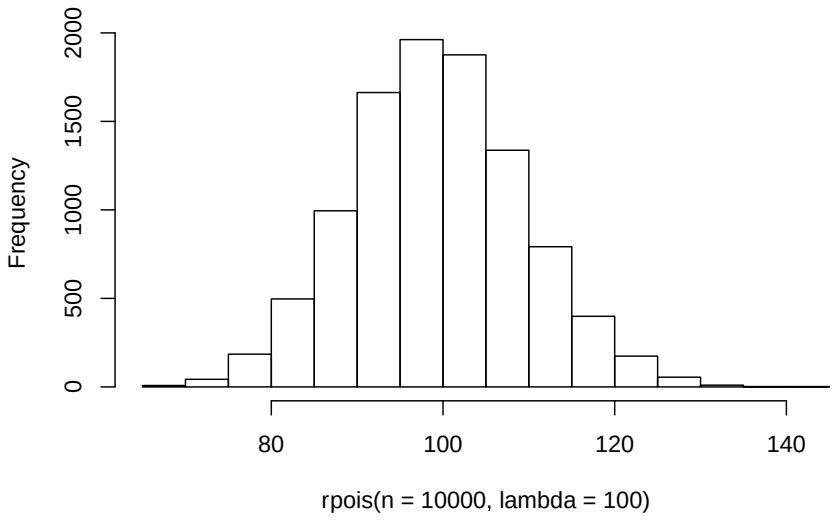
5.5.4 Poisson

The **Poisson distribution** is used for counts or other integer data. This distribution is widely used (and just as widely misused!) for its ability to account

for a large number of biological and ecological processes in the models that we will discuss this semester. The Poisson distribution has a single parameter, λ, which is both the mean and the variance of the distribution. So, despite its utility, the distribution is relatively inflexible with respect to shape and spread. **Fun fact**: this distribution was made widely known by a Russian economist to predict the number of soldiers who were accidentally killed from being kicked by horses in the Prussian army each year. It is named, however, after French mathematician Siméon Denis Poisson. [fails to provide citations for any of this]

Take a look at how the distribution changes when you change λ, and you will get an idea of how this one works. It is probably the most straightforward of any we've considered.

```
hist(rpois(n=1e4, lambda=100), main='')
```



We'll set it aside for now because it often fails us (or our data fail it, I suppose).

5.5.5 The negative binomial distribution

Okay, this one can be a little difficult to wrap your head around but it's an important one for us to know about. So, we will spend a little extra time setting this one up to try and be clear. Often, folks start out thinking that they're going to use a Poisson distribution and they end up collecting with data that do not conform to the relative inflexibility of that single-parameter

distribution. Where they end up usually tends to be a negative binomial in a best case (we'll talk about challenges associated with lots of zeros later in the book).

For the purpose of this class, we are not going to dive into the mechanics of the **negative binomial distribution**, but we do need to know what it looks like and why we might need it.

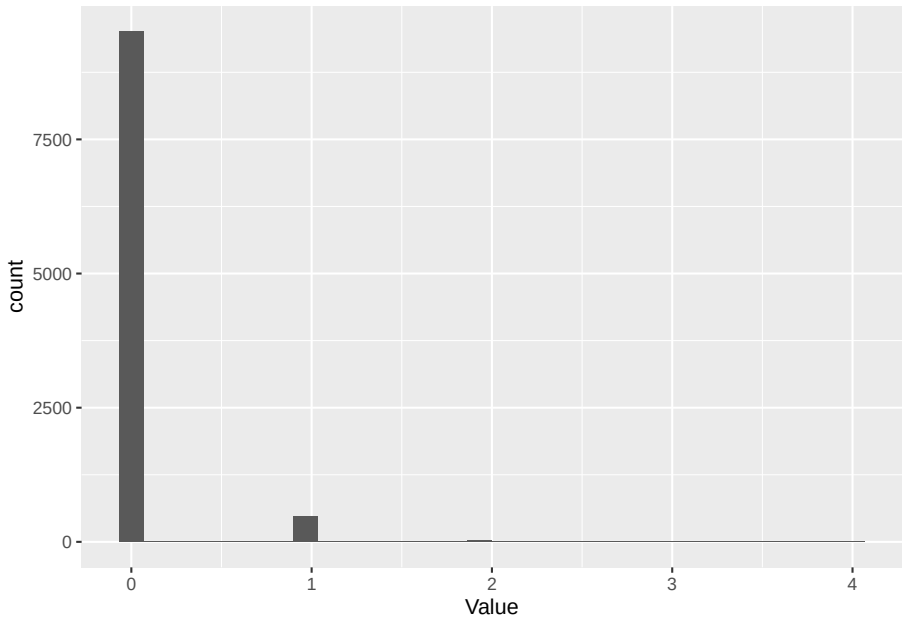
One useful way to conceptualize the negative binomial is "how long does it take for some event to occur?" For example, we might ask how long it takes a fish to start migrating, how long it takes a sea turtle to recover in a rehabilitation center, how long it will take for a terminal patient to expire (ooh, that's dark), or how frequently we see the expression of a gene of interest. These kinds of questions are asked in aptly named "time-to-event" models that rely on the variance structure of the negative binomial. In the context of these kinds of questions, the negative binomial is a discrete probability distribution (and not a continuous distribution) because the "time" component of the distribution is actually a series of independent Bernoulli trials (holy crap!). For example: if we want to know how many days it will take for a turtle to recover from an injury, what we are really doing is asking on each day until recovery, "Is today the day?". Then, we flip a coin and find out. So, each day in this example is a Bernoulli trial. Another way to think about this is the number of failures occurring in a sequence before a target number of sucesses is achieved.

For the classical parameterization:

We will start by looking at how many failures are observed before one success in a sequence of Bernoulli trials.

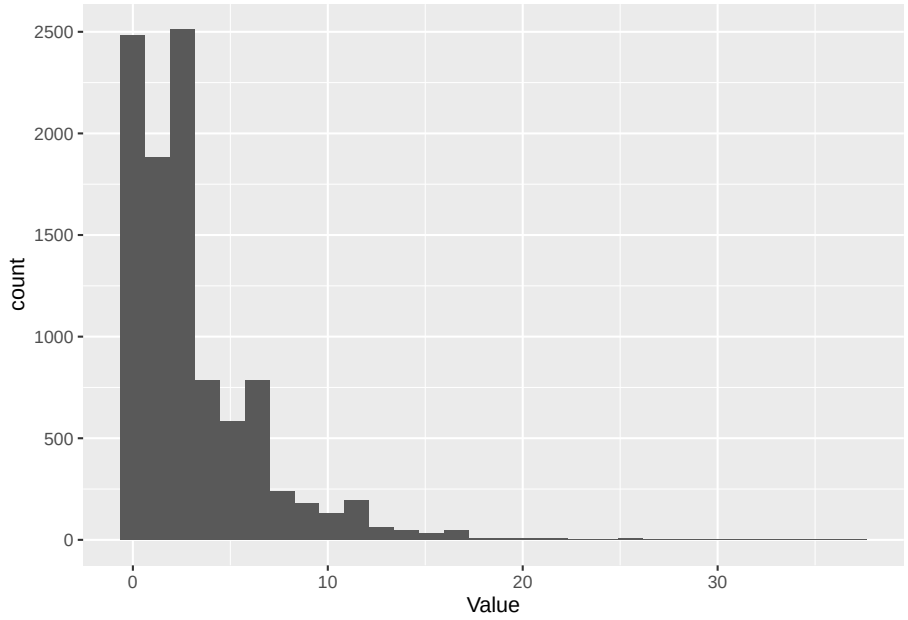
With probability of succes equal to 0.95, it doesn't take long and most of the probability mass is near zero, with a couple of stragglers further out.

```
# Take a random sample from the negative binomial
Value <- rnbinom(1e4, size=1, prob=.95)

# Make a histogram of it with ggplot
ggplot() + geom_histogram( aes(x = Value) )
```



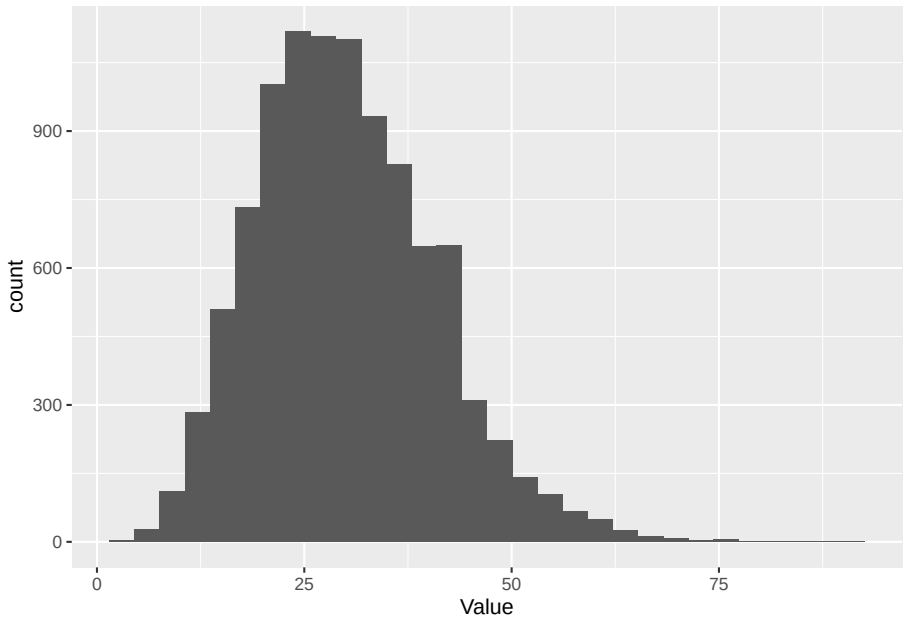
If we decrease probability of success in each trial to 0.25, we see more failures on average before we reach success. Most of the time, it still takes less than 5 trials to reach a success, but some times it takes much longer.

```
# Take a random sample from the negative binomial
Value <- rnbinom(1e4, size=1, prob=.25)

# Make a histogram of it with ggplot
ggplot() + geom_histogram( aes(x = Value) )
```



And, if we increase the number of successes that we use for our criterion, or target, then it spreads the distribution out even further.

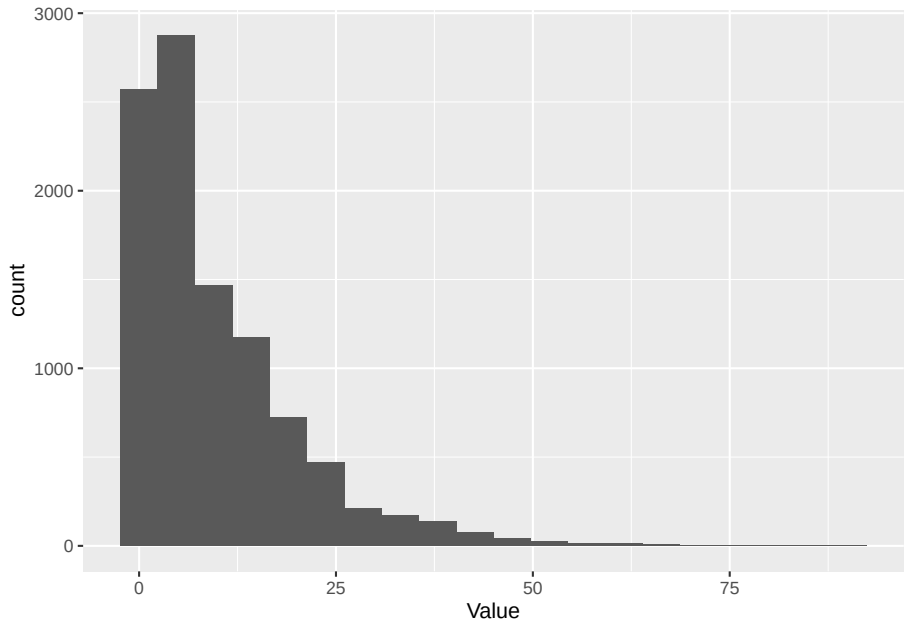
```
# Take a random sample from the negative binomial
Value <- rnbinom(1e4, size=10, prob=.25)

# Make a histogram of it with ggplot
ggplot() + geom_histogram( aes(x = Value) )
```



Now, because of it's properties, the negative binomial is also useful for number of other applications that have nothing to do with interpretting the results of repeated binomial trials. Specifically, it has been widely used to represent Poisson-like processes in which the mean and variance are not equal (e.g., **overdispersion**). This has seen a lot of application in the field of ecology, especially for overdispersed count data.

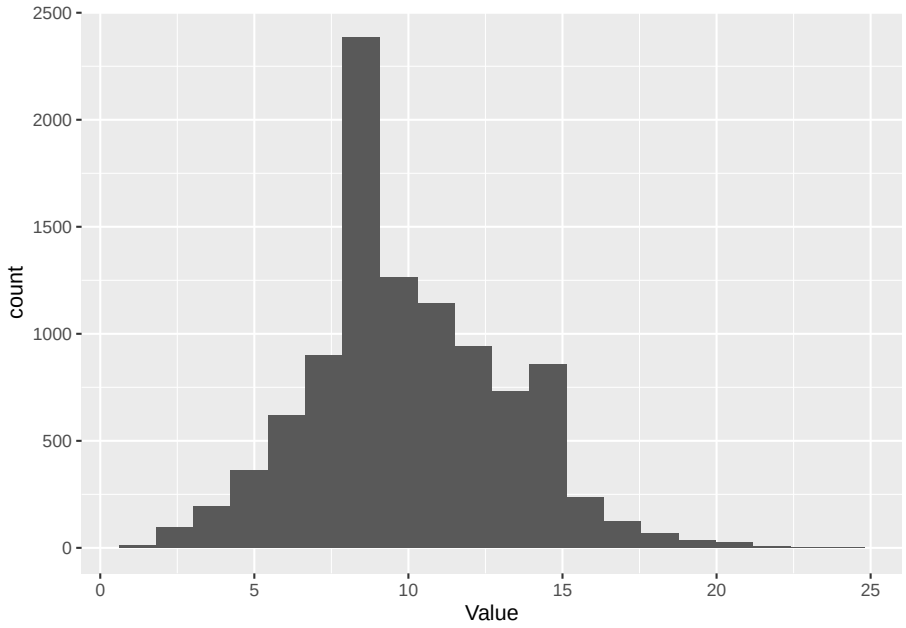
Here, we draw 10,000 random samples from a negative binomial distribution with a mean of 10 and an overdispersion parameter of 1. The overdispersion parameter is called 'size' because this is an alternative parameterization that is just making use of the relationships between existing parameters of the negative binomial. It's easy to grasp how the mean changes the location of the distribution.

```
# Take a random sample from the negative binomial
Value <- rnbinom(1e4, mu = 10, size = 1)

# Make a histogram of it with ggplot
ggplot() + geom_histogram( aes(x = Value), bins = 20 )
```



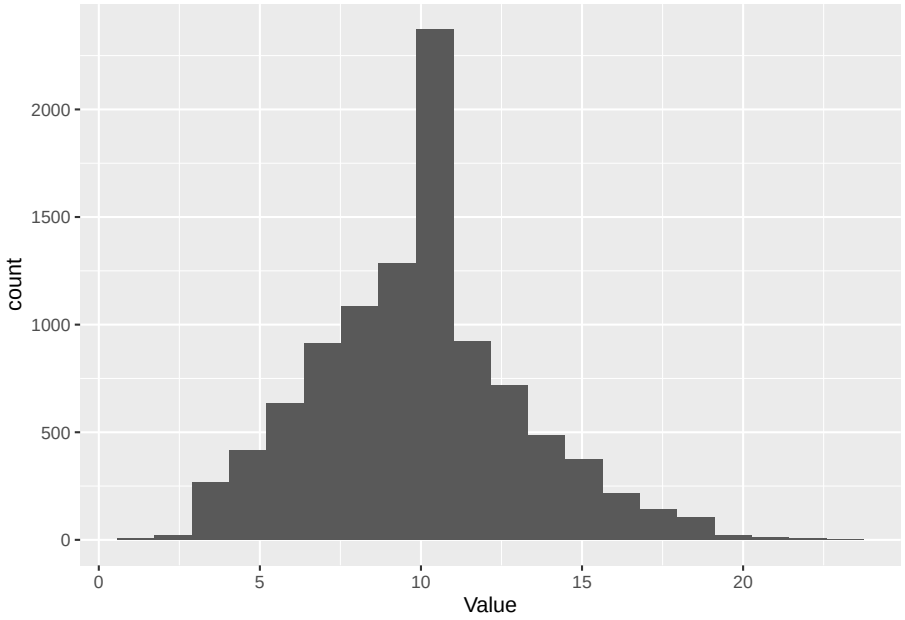
But, note how the overdispersion parameter changes things if you run the following code:

```
# Take a random sample from the negative binomial
Value <- rnbinom(1e4, mu = 10, size = 1000)

# Make a histogram of it with ggplot
ggplot() + geom_histogram( aes(x = Value), bins = 20 )
```



A more intuitive way (I think) to work with the negative binomial in R is by using the `MASS` package. In this parameterization, we use the mean and the dispersion parameter explicitly so it makes more sense:

```
# Take a random sample from the negative binomial
Value <- rnegbin(1e4, mu = 10, theta = 1000)

# Make a histogram of it with ggplot
ggplot() + geom_histogram( aes(x = Value), bins = 20 )
```



The results are pretty much identical. Just two different naming systems for the parameters.

5.6 Sample statistics

In this section, we will learn how to derive the parameters of the **normal distribution** using a few different methods in R. We will use this opportunity to re-introduce the parameters as **moments** of the distribution so we can talk about what we mean by **confidence intervals**. We also will introduce a couple of different methods for calculating moments of a distribution. Specifically, we will look at how to derive…

5.6.1 Moments about the mean

Sounds fancy, huh? Here they are, like a bandaid:

1. Zeroth moment
    - This is the sum of the total probability of the distribution 1.00, always
2. First moment
    - The mean
    - We will look at a few ways to calculate this

3. Second moment
    - The variance
    - As with the mean, we will examine a couple of options for calculating
4. Third moment
    - Skew
    - We won't calculate for this class, but we have discussed, and this parameter contributes to the location/spread of the distribution (how far left or right the peak is)
5. Fourth moment
    - Kurtosis
    - Similarly, we won't cover the calculation, but this is another moment that we may have discussed with respect to departure from a z distribution in the normal

5.6.2 Estimating parameters of the normal distribution from a sample

The tools demonstrated below can be used for most of the probability distributions that have been implemented in R, and we could go on and on forever about them. But, for the sake of our collective sanity we will walk through the tools available using the normal distribution alone. Most of the time this will suffice because our objective in understanding other distributions is really just so that we can use them to assume asymptotic normality in response variables (with transformations) or parameter distributions (with link functions) later on anyway.

5.6.2.1 Method of moments estimator

The moments of the normal distribution are well defined, and you are probably familiar with how to calculate a mean (average) already. See if you can rearrange this in a way that makes sense with how you know to calculate a **mean** and a **variance**!

Start by simulating a variable with a known mean and standard deviation. We'll pretend that we are simulating cold temperatures here:

```
# Take a random sample from a normal
# with a mean of 20 and a standard
# deviation of 2
test_norm <- rnorm(1e4, 20, 2)
```

First, we'll estimate it by making our own function:

```
# Write the function
# First, define a function by name
norm.mom = function(x){

  # Calculate mean
  x_bar = (1/length(x)) * sum(x)

  # Calculate variance
  sigma2 = (1/length(x)) * sum((x-x_bar)^2)

  # Return the calculations
  return(c(x_bar, sigma2))

}
# Test the function
norm.mom(test_norm)
```

```
## [1] 19.975688  4.017443
```

Because this one is so common, R has built-in estimators that rely on the exact solution provided by the formulas for the first two moments of the normal distribution:

```
mean(test_norm)
```

```
## [1] 19.97569
```

```
var(test_norm)
```

```
## [1] 4.017845
```

Wow, that was a lot less code. That is the beauty of having these functions available. How do these compare to the answers returned by our function if you scroll back up?

5.6.2.2 Maximum likelihood estimator

R also has built-in **maximum likelihood** estimators that provide an exact solution to the first two moments of the normal distribution. These are available through the `MASS` package.

```
fitdistr(test_norm, 'normal')
```

```
##       mean           sd
##   19.97568847    2.00435612
##  ( 0.02004356) ( 0.01417294)
```

Only one problem here: R doesn't report the second moment! It reports the square root of the second moment: the **standard deviation**!

Finally, let's write our own function and maximize the likelihood with the `optim()` function in R.

```
# Define the function
normal.lik = function(theta, y){

  # The starting value for
  # mu that we provide
  mu = theta[1]

  # The starting value for
  # sigma2 that we provide
  sigma2 = theta[2]

  # Number of observations in the data
  n = nrow(y)

  # Compute the log likelihood of the
  # data (y) using the likelihood
  # function for the normal distribution
  # given the starting values for our
  # parameters (contained in the vector 'theta')
  logl = -.5*n*log(2*pi) -.5*n*log(sigma2)-(1/(2*sigma2))*sum((y-mu)**2)
  return(-logl)
}
```

Now, we use the `optim` function to maximize the likelihood of the data (technically by minimizing the -2*log[likehood]) given different values of our parameters (`mu` and `sigma2`).

To get started, we need to take a guess at what those parameters could be. (Yes, we know they are mu = 20 and sd = 2)

```
optim(c(20, 4), normal.lik, y=data.frame(test_norm))
```

```
## $par
```

```
## [1] 19.975735  4.016827
##
## $value
## [1] 21142.61
##
## $counts
## function gradient
##       49       NA
##
## $convergence
## [1] 0
##
## $message
## NULL
```

The pieces are in `pars` here (right where we told R to put them!). We can also make the output into an object and call the parts by name:

```
# Make it into an object
est = optim(c(0, 1),
            normal.lik,
            y=data.frame(test_norm)
            )
```

```
## Warning in log(sigma2): NaNs produced

## Warning in log(sigma2): NaNs produced

## Warning in log(sigma2): NaNs produced

## Warning in log(sigma2): NaNs produced

## Warning in log(sigma2): NaNs produced

## Warning in log(sigma2): NaNs produced
```

Look at the structure I'll be damned, it's a list! Hey, we learned about those!

```
str(est)
```

```
## List of 5
##  $ par         : num [1:2] 19.98 4.02
##  $ value       : num 21143
##  $ counts      : Named int [1:2] 97 NA
```

```
##   ..- attr(*, "names")= chr [1:2] "function" "gradient"
##  $ convergence: int 0
##  $ message    : NULL
```

And, here are the estimates:

```
# Both
est$par
```

```
## [1] 19.977955  4.019973
```

```
# The mean
est$par[1]
```

```
## [1] 19.97796
```

```
# The variance
est$par[2]
```

```
## [1] 4.019973
```

There you have it, a couple of different ways to calculate parameters of the normal distribution using a couple of different approaches each.

5.6.3 Quantiles and other descriptive statistics

There are a number of other ways we might like to describe this this (or any) sampling distribution. Here are a few examples that we will work with this semester.

```
# Here is the median, or 50th percentile
median(test_norm)
```

```
## [1] 19.95321
```

```
# The 95% confidence limits
quantile(test_norm, probs = c(0.025, 0.975))
```

```
##     2.5%    97.5%
## 16.05628 23.93365
```

```r
# Interquartile range (Q1 and Q3)
quantile(test_norm, probs = c(0.25, 0.75))
```

```
##       25%       75%
## 18.62750 21.32331
```

```r
# Range of sample
range(test_norm)
```

```
## [1] 12.86480 27.12729
```

5.7 Next steps

Here, we have explored some of the probability distributions that we use to describe samples (data) that we collect from the real world. In Chapter 6 we will explore how these sampling distributions can be used for statistical inference before diving into applied statistical analyses for the remainder of the book. Hopefully the order of things is starting to make some sense! If not, well…this *is* The Worst Stats Text eveR.

Chapter 6

Inferential statistics

These are hot peppers. Like hot peppers, statistics can cause pain and heartburn if you are not accustomed to them. Ready or not, let's get munching!!

This week we will begin conducting our first statistical tests! We are going to start small and simple, and we will build complexity during the remainder of the semester. We will also start to make more use of some of the programming techniques that you have been developing, and we will build a foundation for moving into regression models in coming weeks.

We'll start with some simple methods for testing hypotheses about sampling distributions this week. Although relatively limited in scope within the fields of biology and ecology, these tend to be fairly robust tests, and can be powerful tools if studies are designed thoughtfully. For this week, we will focus on implementation of one-sample t-tests, two-sample t-tests, Wilcox tests, and frequency analysis using a χ^2 test. Within the context of the assumptions of these tests we will also discuss the F-test and the Shapiro-Wilk test of normality. In short, you probably will learn more statistical tests in this preliminary chapter about statistical inference than you have in your college career to this point. Take your time and soak in all the mathy goodness. We'll need it!

For this Chapter, we will continue working with packages from the `tidyverse`. You can go ahead and put this in the top of your code for the chapter if you want to load it all at once:

```
library(tidyverse)
```

We will also need the grass carp data for this exercise, which we will load from `grasscarp.csv`. Remember that you can download all of the class data here or you can get the individual `grasscarp.csv` file by clicking here and saving with `Ctrl + S` (Windows) or `Command + S` (Mac OS-X).

These data come from a long-term study of fish population responses to changes in their primary food source, the invasive hydrilla (*Hydrilla verticallata*). There are a whole bunch of columns in here! The important variables for this chapter are `Year` (year of fish collection), `Age` (the age of each fish), `Length` (total length of fish in mm), and `hydrilla` (hectares of hydrilla measured each `Year`).

6.1 One-sample tests

Sometimes, we are interested in simply knowing whether or not the measurements we've obtained from an individual or a group are representative of a larger population. For example, we may have a 'control' group in an experiment and we want to know if the group is truly representative of the population average or some measurement we have collected from a different biological population. For these situations, we will rely on one-sample tests this week and we'll look at other (totally related) options moving forward.

6.1.1 One sample t-test

We will examine parametric and non-parametric examples of one-sample tests here to demonstrate why and how we use them.

Let's start with a simple example of how we might do this, and what the results actually mean. We'll use some data from grass carp (*Ctenopharyngodon idella*) from Lake Gaston, Virginia and North Carolina, USA for this example. We will compare the size of grass carp at specific ages with their population density using a few different tools

Read in the data set:

```
grasscarp <- read.csv('data/grasscarp.csv')
```

Just for funsies, you could also read this in directly from the link to the raw data in the GitHub repository for this book if you have an internet connection:

```
grasscarp <- read.csv('https://raw.githubusercontent.com/danStich/worst-r/master/data/g
```

Remember to check out the data set in your Environment tab so you understand how many observations there are and how many variables (as well as their types).

Let's start by asking a simple biological question: is the size of age-3 grass carp different from the average size of fish in this population?

First, let's create a sample that includes only age-3 fish. We will store this to a new vector called `age3_lengths`.

```
age3_lengths <- grasscarp$Length[grasscarp$Age == 3]
```

Now, let's compare the `Length` of age-3 fish to the rest of the population using a one-sample t-test. To do this, we need to pass `age3_lengths`

```
# Run the test and save the output to an object
our_test = t.test(age3_lengths,
                  mu = mean(grasscarp$Length),
                  conf.level = 0.95
                  )

# Print the results of the object to the console
print(our_test)
```

```
##
##  One Sample t-test
##
## data:  age3_lengths
## t = -18.829, df = 47, p-value < 2.2e-16
## alternative hypothesis: true mean is not equal to 973.0118
## 95 percent confidence interval:
##  749.2547 792.4536
## sample estimates:
## mean of x
##  770.8542
```

Okay, so what does that mean???

> First, let's look at what we've done here.

We've conducted a one-sample t-test.

The null hypothesis was that:

> (H_0): the sample (age-3 fish) did not differ in `Length` from the mean of the population.

This is because we stated no specific alternative hypothesis when we executed the t-test above. If we had used a different alternative hypothesis (i.e. `greater` or `less` in the argument `alternative`) then our null would be formalized as: "The length of age-3 fish is not significantly greater (or less) than the population mean".

Finally, we specified the confidence level. Here, we are told R that we want to know the result with a confidence level of 95% (0.95). This corresponds to a

Type-I error rate (α) of 0.05. This means we are looking for $p < 0.05$ to conclude that the sample is statistically different from the population mean. Since p < 0.001, we reject the H_0 and conclude that age-3 fish are significantly shorter than the population mean.

6.1.1.1 Output

R returns the output of statistical tests as objects, and you can reference any part of those objects by name or index. The type of object that R returns, and how you access the parts depends on the type of test you ran and with what options.

Like so many other models objects, our one sample t-test is stored as a list:

```
str(our_test)
```

```
## List of 10
##  $ statistic  : Named num -18.8
##   ..- attr(*, "names")= chr "t"
##  $ parameter  : Named num 47
##   ..- attr(*, "names")= chr "df"
##  $ p.value    : num 1.57e-23
##  $ conf.int   : num [1:2] 749 792
##   ..- attr(*, "conf.level")= num 0.95
##  $ estimate   : Named num 771
##   ..- attr(*, "names")= chr "mean of x"
##  $ null.value : Named num 973
##   ..- attr(*, "names")= chr "mean"
##  $ stderr     : num 10.7
##  $ alternative: chr "two.sided"
##  $ method     : chr "One Sample t-test"
##  $ data.name  : chr "age3_lengths"
##  - attr(*, "class")= chr "htest"
```

We can just look at the names if there are specific pieces in which we are interested. For example, we might want to save the p-value (`p.value`):

```
# Shows us the names of the things inside the model list
  names(our_test)
```

```
##  [1] "statistic"   "parameter"   "p.value"     "conf.int"    "estimate"
##  [6] "null.value"  "stderr"      "alternative" "method"      "data.name"
```

```
# This stores our p-value to an object for use
  p_out = our_test$p.value

# And of course we can look at it
  print(p_out)
```

```
## [1] 1.572767e-23
```

Now we can go through the output as it is displayed by:

```
# Print a summary of the test
  print(our_test)
```

```
##
##  One Sample t-test
##
## data:  age3_lengths
## t = -18.829, df = 47, p-value < 2.2e-16
## alternative hypothesis: true mean is not equal to 973.0118
## 95 percent confidence interval:
##  749.2547 792.4536
## sample estimates:
## mean of x
##  770.8542
```

The first line of the output gives us the actual data that with which we are working- nothing of interest here other than a quick sanity check until later on in the course.

The second line shows the 'statistics' that we are interested in: `t` is the calculated value of the test statistic for the t-test in this case. The `df`, or degrees of freedom, is the number of observations in the sample, minus the number of parameters that we are estimating (in this case, just one: the mean). Our `p-value` is the probability of observing data that are more extreme than what we observed if the null hypothesis is in fact true (i.e. the probability that rejection of the null is inappropriate). Again, because it is smaller than α we reject the null and accept the alternative hypothesis.

Our `alteranive hypothesis` (H_A) was that the sample mean is not equal to population mean. We can specify other alternatives (and therefore nulls) in this and other models in R.

Finally, R reports the mean and the 95% confidence interval of `age3_lengths`.

6.1.1.2 Assumptions

It's always important for us to think about the assumptions that we are making when (read *before*) conducting a statistical test. First, there are implicit assumptions that we make. For example, we assume that the data are representative and were collected in a random manner in this case. Then, there are explicit assumptions that we make in specific tests.

For the one-sample t-test, the assumption that we really care about is:

1. The data are normally distributed

The t-test is generally robust to violations of this assumption provided that sample sizes are large enough (Google Central Limit Theorem, this is The Worst Stats Text eveR). But, it is always good to check. In particular, when we are working with small sample sizes like this example (n = 48), we should really make sure that things look okay.

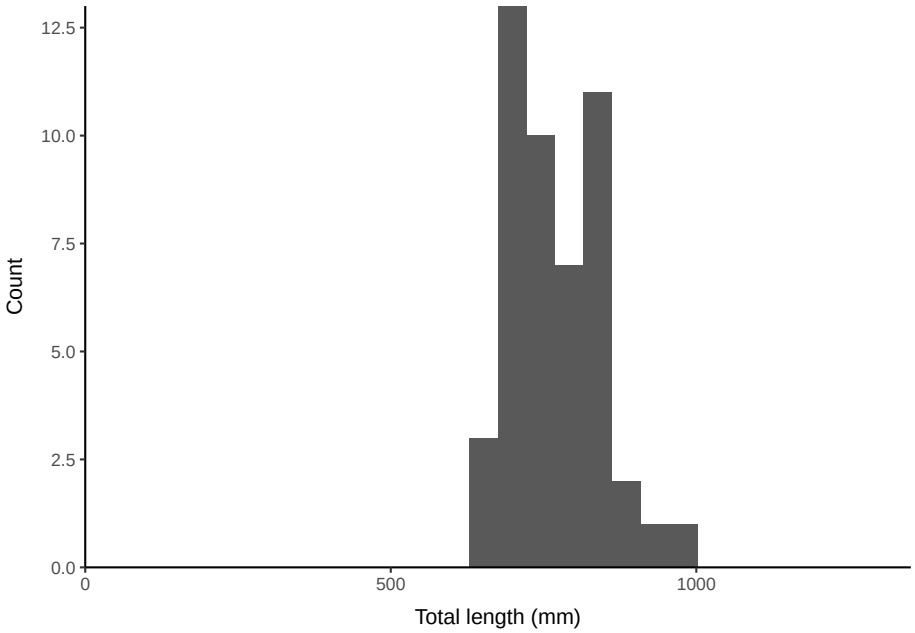
6.1.1.3 Checking assumptions

Visual check for normality

One simple way to gauge assumptions of normality is to look at a plot of the data. As you will see later, we are usually concerned with the *residuals*, but we can look at the actual data here because we have only one group and if it's normal so are its errors.

Have a quick look at these to see what we are working with using the histogram code from Chapter 4. I set the x-axis limits below using the maximum `Length` from the `grasscarp` data so we can see what part of the length range we've sampled here.

```
p <- ggplot() +
  geom_histogram(aes(age3_lengths), bins = 30) +
  scale_x_continuous(limits=c(0, max(grasscarp$Length)), expand = c(0, 0)) +
  scale_y_continuous(expand = c(0, 0)) +
  xlab("Total length (mm)") +
  ylab("Count") +
  theme_classic() +
  theme(
    axis.title.x = element_text(vjust = -1),
    axis.title.y = element_text(vjust = 3),
    panel.grid = element_blank()
  )
print(p)
```



Sure, looks totally normal to me? Wouldn't it be great if there were a statistical test for determining whether this sample is different from the normal? Great that you should ask.

6.1.1.3.1 Tests of normality (Shapiro-Wilk) The Shapiro-Wilk test is commonly used to test normality of a distribution as a check of assumptions. We can use this to test whether our data deviate from normal in the following manner:

```
shapiro.test(age3_lengths)
```

```
##
##  Shapiro-Wilk normality test
##
## data:  age3_lengths
## W = 0.95715, p-value = 0.07746
```

First, note that the test statistic is the `W` statistic for this test.

Second, we have a p-value of `0.0774637`. **Oh, no**! Wait, what does that mean?

For this test, we actually don't want $p < 0.05$ if we are relying on assumptions of normality, so this is a "good" thing. But, it doesn't necessarily mean `age3_lengths` is normally distributed. It just means that we can't tell if the sample we have collected is different from normal (we fail to reject the null but

can't "accept" it). I guess that is good enough, but that p-value is awfully close to 0.05 for my taste.

So, are we up that proverbial tributary without a paddle, or can we salvage the mess and move on with life? Don't worry, there's a statistical test for that, too.

6.1.2 Wilcox test

You can think of the Wilcox test as a non-parametric analog of the t-test. In general, non-parametric tests tend to be slightly more "conservative" than the parametric alternatives because they require fewer assumptions. However, non-parametric tests can be useful where our data are not normal, or we don't feel we have sufficient data to know (hmm…maybe don't conduct any tests, in that case!).

Here, we are checking for shifts in the median (not the mean) of one or more samples.

Why is this? The mean of a non-normal distribution is not always a useful descriptor of the probability mass under a distribution (it still describes 'central tendency' but does not necessarily describe the place where 'most of the data are'). But, the median always (as in always, always, always) describes central tendency of the data, so we can pretty much use it for describing any sample. This is because the median is defined as the "middle" value. That is, half of the data should fall on either side of the median if you lined up all of your data on the equator (wrong metaphor?).

…back to the Wilcox test.

```
# First, do this and have a quick read:
  ?wilcox.test
```

Now we can actually run a test to see if the median length of age-3 fish is statistically different from the median value of length in the sample.

```
    wilcox.test(
      age3_lengths,
      mu = median(grasscarp$Length),
      alternative = 'less',     # Note different alternative here!
      exact = FALSE             # Not much data, so not exact
    )
```

```
##
##  Wilcoxon signed rank test with continuity correction
##
## data:  age3_lengths
```

```
## V = 0, p-value = 8.414e-10
## alternative hypothesis: true location is less than 1007
```

Interpretting the results is essentially the same as for the t-test, but without the degrees of freedom, so we won't belabor this. Importantly, the test, being robust to any distributional assumptions, should also (and does) tell us that the length of age-3 fish is significantly shorter than the population mean (or median- whichever you used).

6.2 Two-sample tests

Okay, with that out of the way, now we can get on to some tests that might be a little more meaningful to most people. We can use **two-sample** tests to whether two groups differ in some metric of interest. This lends itself naturally to use in controlled experiments that we conduct in laboratories, for example.

6.2.1 The two-sample t-test

If you have been exposed to only one statistical test already it is probably the two-sample t-test. This is a test that is used to test for differences in some continuous variable between two groups. The test statistic itself is pretty trivial to calculate. You can find a video of that here. **Seriously, if you have never done a t-test, watch the 6-minute video now**. Otherwise, you may not understand what follows. I am not going to go into the math here because this is The Worst Stats Text eveR. The video will also help you understand how ANOVA and other tests work later. Understanding how these tests work will give you phenomenal cosmic powers when it comes to analyzing biological data. If you email asking me how a t-test works, I am going to send you this video.

Let's keep working with the `grasscarp` data for now for the sake of consistency. But, now we want to know if there is a difference in mean length of fish depending on whether their population density is high or low. To look at this, we'll need to make some groups in our `grasscarp` data that correspond to years of high and low density.

You can compare fish density between years quickly using the summary pipeline demonstrated in Chapter 3

```
grasscarp %>%
  group_by(Year) %>%
  summarize(dens = unique(nha))
```

```
## # A tibble: 5 x 2
```

```
##    Year  dens
##   <int> <int>
## 1  2006    21
## 2  2007    58
## 3  2009    44
## 4  2010    43
## 5  2017   127
```

You can see that density was much higher in `2017` than in any of the preceding years. This is because hydrilla area was reduced by several hundred hectares (`ha`) between 2010 and 2014 (which was actually the reason we went out to collect more data in 2017). But, these are just means and we need to be able to account for the variability in these measurements to call it science.

So, let's build some groups based on high and low density years. First, we'll add a new categorical variable to `grasscarp` called "`density`", and we'll fill it all in with the word `"low"` because there is only one year when density was high.

```
grasscarp$density <- "low"
```

Next, we'll change all of the observations for 2017 to `"high"` so we have low density and high density groupings in our `density` column. This way, we only have to change the variable for one year.

```
grasscarp$density[grasscarp$Year == 2017] <- "high"
```

Then, we'll subset the data to look at a single age so our comparisons are fair between years. I picked `Age == 10` because 10 years is in the middle of the range of ages in the data set. You can try it with another age as long as there are enough data.

```
mid_carps <- grasscarp %>% subset(Age == 10)
```

Now, we can conduct our two-sample t-test!

The syntax is pretty straightforward, and is similar to what we used above, except that now we have two groups so we will omit `mu` and specify the t-test as a formula with independent (x, `density`) and dependent (y, `Length`) variables. There is no pairing of our observations, so we specify `paired = FALSE`, and we tell R we don't want to assume that the variance of `Length` is equal between `density` groups.

```
  t.test(Length ~ density,
         data = mid_carps,
         paired = FALSE,      # 2-sample test, not "paired"
```

```
        var.equal = FALSE,    # We make no variance assumption
        conf.level = 0.95     # Alpha = 0.05
    )
```

```
##
##  Welch Two Sample t-test
##
## data:  Length by density
## t = -3.2263, df = 11.133, p-value = 0.007952
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -190.03710  -36.03433
## sample estimates:
## mean in group high  mean in group low
##           987.7143          1100.7500
```

The interpretation of the results is much the same as with the one-sample t-test, except that we are now testing the null hypothesis that there is no difference between groups.

We reject the null hypothesis, and we conclude that age-10 fish were significantly larger during periods of low population density than they were during years of high population density ($t =$ `-3.2262903`, df = `11.1326183`, $p < 0.05$). Makes perfect sense!

6.2.1.1 Assumptions

Equal variance

Now that we are using two samples, we should be cognizant that this test assumes equal variances in the independent variable between our two groups. If our variances are not equal, then we need to account for that (R actually assumes that the variances are unequal by default).

Let's test to see if the variances were equal between age-10 fish in the `high` and `low` density groups. To do this, we will conduct an F-test on the ratio of the two variances. If the ratio of the variances is different than one, we reject the null that the variances are the same.

```
var.test(Length~density, mid_carps)
```

```
##
##  F test to compare two variances
##
## data:  Length by density
```

```
## F = 0.72988, num df = 20, denom df = 7, p-value = 0.5423
## alternative hypothesis: true ratio of variances is not equal to 1
## 95 percent confidence interval:
##  0.1634026 2.1950440
## sample estimates:
## ratio of variances
##           0.729877
```

Wow, this is way to easy. I hope that you are beginning to understand the **GLORY OF R**. This test could be a real pain in other software programs, and may not even be an option in many.

Back on topic...we fail to reject the null hypothesis that the variances were equal. In this case, we now feel validated in the use of a two-sample t-test regardless of what R uses as the default (yes, sarcasm intended).

6.2.1.1.1 Normality Yes, we are still worried about this one because of the reasons given in the previous section. We can check this the same way as before. End of section.

6.2.2 Two-sample Wilcox test

If we were in violation of normality, we would use the Wilcox test to test for differences in ranks. I will not go through the whole thing again here. As with the t-test, if you have not been exposed to doing a rank-sum test by hand you really should **watch a video of how to do it**. It really is easy once you've seen it and the video demystify the test for you.

I will note that the syntax is very much the same to that of the t-test now. This will pretty much stay the same for the next 6 chapters. Thank R, not me.

```
wilcox.test(Length~density, mid_carps)
```

```
##
##  Wilcoxon rank sum test with continuity correction
##
## data:  Length by density
## W = 26, p-value = 0.005015
## alternative hypothesis: true location shift is not equal to 0
```

As expected, this test also shows that the two samples differ significantly.

Note: this is equivelent to the Mann-Whitney U-test you may have learned about elsewhere. Had these samples been paired, R would have defaulted to a signed-rank test, with which you may also be familiar.

6.2.3 Presenting your results

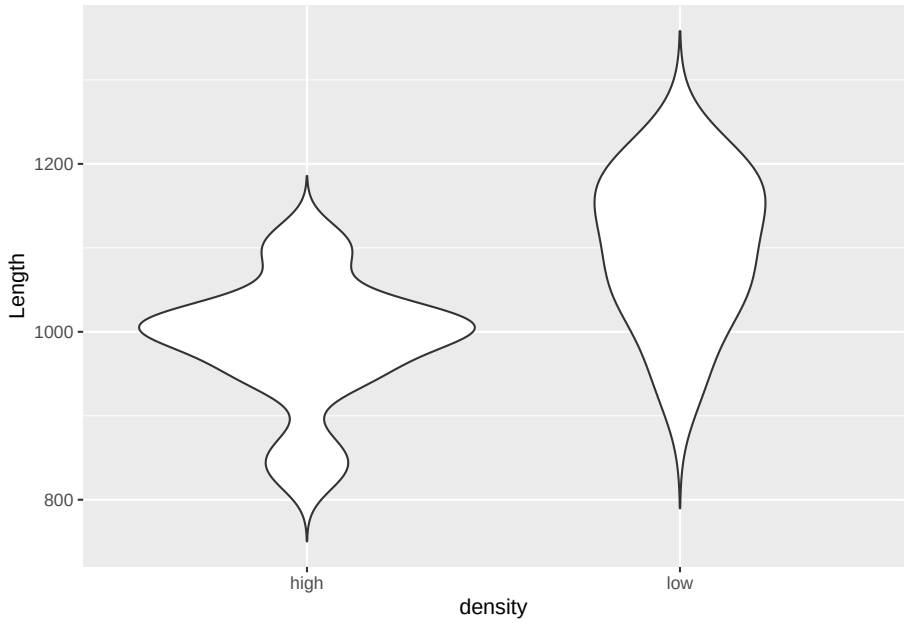
While it is important to report the test statistics, df, etc., it can be just as meaningful to give the sample means (reported in the `t.test`) and show a graph. **Remember**: don't make shitty graphs. Be proud of your results and show your readers what they mean.

In this case, a boxplot or a violin plot would work great. We haven't looked at violin plots yet, so let's give them a whirl!

Violins are a lot like box plots except they give us a little better visual description of the shape of sampling distributions within groups. I added some `ggplot2` functions to control fill and color of the violins in the example below. You can check out this blog post for some other cool examples with other `ggplot` geometries. Play around with the plotting code above to change what you like. Remember, all of the customization achieved using the `theme()` function is the same across plot types.

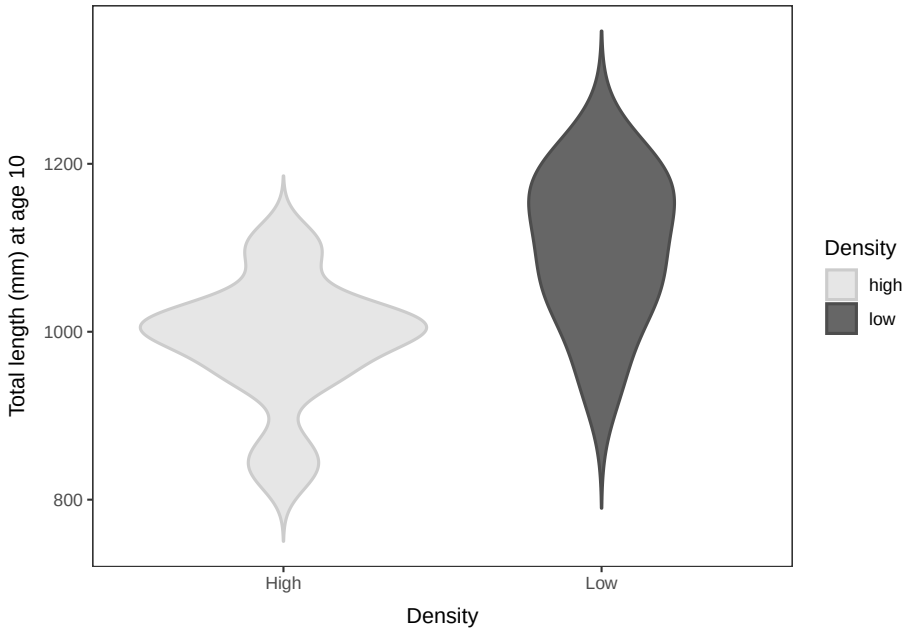
Here is a quick, ugly violin plot with some basic options. Pretty easy to make, but also kind of makes you want to puke.

```
ggplot(mid_carps, aes(x = density, y = Length)) +
  geom_violin(aes(group = density), trim = FALSE)
```



Here is a much better plot. Not that much more difficult to make, and doesn't make you want to puke even if the code does a little bit.

```
mid_carps %>%
ggplot(aes(x = density, y = Length, fill = density, color = density)) +
  geom_violin(aes(group = density), trim = FALSE, size = .75) +
  scale_x_discrete(breaks=c("high", "low"), labels = c("High", "Low")) +
  scale_fill_grey(start = 0.9, end = 0.4) +
  scale_color_grey(start = 0.8, end = 0.3) +
  xlab("Density") +
  ylab("Total length (mm) at age 10") +
  labs(fill = "Density", color = "Density") +
  theme_bw() +
  theme(
    axis.title.x = element_text(vjust = -1),
    axis.title.y = element_text(vjust = 3),
    panel.grid = element_blank()
  )
```



There, isn't that fancy? I think that gives you a much more detailed understanding of how `Length` varies between `high` and `low` population density than a simple p-value. But, maybe its just me...

6.3 Frequency analysis

Now, what if we didn't collect very good data, or we binned our data into low-resolution categories for the sake of ease in our study design? Often, and for a variety of reasons other than crappy data collection, we want to compare frequencies of events between two (or more) groups. We may even design studies specifically to test these kinds of hypotheses when we think about rates, for example. This is very common in studies of population genetics [definitely citations available for that one - go Google them]

The simplest way to test for differences in the frequency of a categorical response between two groups is (some would argue) the χ^2 test. The χ^2 is another one of those that you should really work out by hand because it is used in a variety of settings "under the hood" of more complex routines. Here is your token video link showing an example. Watch it. Please.

6.3.1 Worked example

Let's say we want to know if the number of grass carp in a given age group (say age 10) varies between years. These fish are sterile hybrids, so we would expect that the number of fish in each age would change drastically with increasing time since the year of initial stocking (1995).

First, make a table showing the number of fish in each `Age` by `Year` with the `grasscarp` data.

```
agexyear <- with(grasscarp, table(Age, Year))

print(agexyear)
```

```
##     Year
## Age  2006 2007 2009 2010 2017
##   1     0    0    0    2    0
##   2     6    7    7    8    0
##   3    11   10   10   14    3
##   4     4    3    6    3    3
##   5     0    1    3    2   10
##   6     0    1    2    1   12
##   7     0    0    1    3   11
##   8     0    1    1    2   12
##   9     1    0    1    3   21
##   10    4    1    1    2   21
##   11    4    9    0    6   15
##   12    3   15    2    3   18
##   13    0    1    4    3    7
```

```
##   14    0    0   11    8    6
##   15    0    0    2   12   12
##   16    0    0    0    2   10
##   17    0    0    0    0    8
##   18    0    0    0    0    9
##   19    0    0    0    0   12
##   20    0    0    0    0   12
##   21    0    0    0    0    5
##   22    0    0    0    0    3
##   23    0    0    0    0    2
```

Basically what we are going to do is analyze the proportion of total fish in each column by age.

You should see some pretty obvious patterns here. We have a couple of things to think about now. First, this is the kind of question you don't need statistics for. Second, we have a whole bunch of empty groups, and these are not random with respect to year. Some of these come from ages that were not yet available in years 2006 - 2010 and some from patterns in fish stocking. The large number of empty pairings and the fact that most age classes had fewer than five fish in any year prior to 2017 means we should probably break the data down a little further. This stinks because we lose resolution, but that is the cost.

For the sake of demonstration, let's summarize the data by `high` and `low` density again and we'll look at the number of fish collected in each age class during high and low density years.

```
freqs <- grasscarp %>% # Pass grass carp data frame to group_by()
    filter(Age %in% c(10:15)) # Select only shared age range

head(freqs)
```

```
##   Year Length Weight_kg Age cohort n_stocked     n acre   ha nha     kg kg_ha
## 1 2006   1262    23.154  12   1994      7000 25128 2957 1203  21 129929   108
## 2 2006   1138    23.923  10   1996      7000 25128 2957 1203  21 129929   108
## 3 2006   1187    18.614  10   1996      7000 25128 2957 1203  21 129929   108
## 4 2006   1207    19.522  12   1994      7000 25128 2957 1203  21 129929   108
## 5 2006   1086    21.285  10   1996      7000 25128 2957 1203  21 129929   108
## 6 2006   1095    18.160  12   1994      7000 25128 2957 1203  21 129929   108
##   density
## 1     low
## 2     low
## 3     low
## 4     low
## 5     low
## 6     low
```

We will test the null hypothesis that there is no difference in the number of age-10 fish between high and low densities.

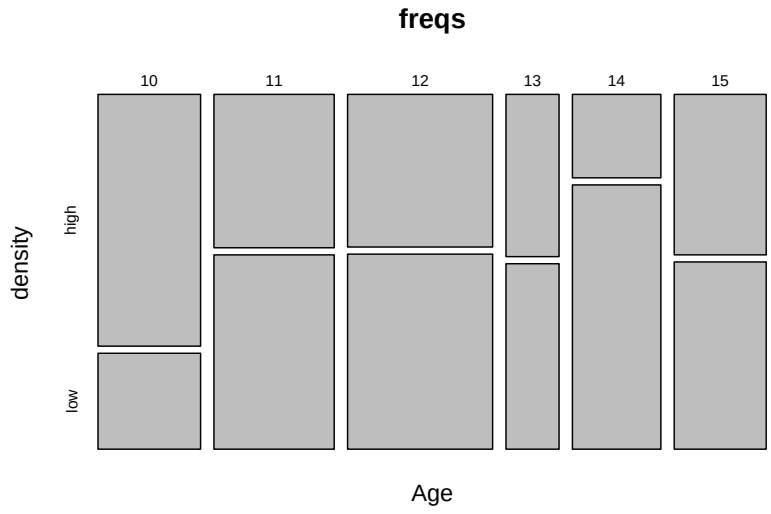
```
# Run the test
chi_test <- with(freqs, chisq.test(x = density, y = Age))

# Have a look
print(chi_test)
```

```
##
##  Pearson's Chi-squared test
##
## data:  density and Age
## X-squared = 13.107, df = 5, p-value = 0.0224
```

And, bam! We see that there is a difference in the frequency of fish collected in each age class in high and low density years. Shocker.

Data visualization techniques for contingency table analyses like this seem to have generally lagged behind theory in terms of wide-spread implementation. There is a base R `mosaicplot` that plots relative freqencies. You can interpret the width of the bars as the proportion of total observations in each age class. Likewise, the height of the vertical segnmens corresponds to proportion of `high` or `low` `density` observations in each `Age`.

```
mosaicplot(Age ~ density, data = freqs)
```



The need for improved graphical representation for these data types is recognized. There have been recent efforts to extend the philosophies used `ggplot()` to contingency analysis by r developers (see Wickham and Hofmann 2011). It was even the topic of a recent master's thesis (see Grant 2017). But as far as I know the ideas from these works have not been integrated into `ggplot2` or the `tidyverse` yet. Sorry about the citations. I don't know what I was thinking. This is supposed to be the Worst Stats Text eveR.

6.4 Next steps

In this chapter, we introduced inferential statistics and walked through examples of a few simple statistical tests for comparing samples to one another using two-sample tests, or to a single value using a one-sample test. In Chapter 7 we will continue to build on these tools as we press on to linear models and the rest of statistics.

Chapter 7

Linear models

Yeah, I know this is the picture from Chapter 4. I only have like five pictures. This is the Worst Stats Text eveR! But, both of the graphs in this picture are just applications of linear regression, which is one kind of linear model, which is also called the general linear model.

Introduction

In this chapter, we will introduce a class of statistical tools known collectively as **linear models**. This class of tools includes such examples as analysis of variance (ANOVA), linear regression and correlation, and by extension includes n-way ANOVA, multiple linear regression, and analysis of covariance (ANCOVA). Later this semester, we will see that these models be extended even further to include generalized linear models, generalized linear mixed models, multivariate techniques and even machine learning algorithms.

Linear models are, therefore, the gateway into the rest of the world of statistics. We will focus primarily on parametric applications this week and next. The over-arching theme for this week is that any of these methods can be expressed as the formula for a line, which is how they got their names (oh, snap!). We will start with ANOVA because it is analogous to many of the methods that we've already discussed. However, it is important to recognize that this is just a special case of the linear model. This will help you think about how we test statistical assumptions, test hypotheses, and communicate results of models.

Because we are now entering into the realm of 'the rest of statistics' we also need to start 'talking the talk' in addition to 'walking the walk', so we will practice how to write methods sections for these tests and how to report the results. In reality, once you are comfortable using a couple of functions in R, writing up the methods and results is more challenging than fitting models.

```
library(tidyverse)
```

7.1 Analysis of variance (ANOVA)

We will use some of the built-in datasets in R this week to demonstrate our analyses and show how to communicate the methods and the results of our statistical inference.

Analysis of variance is typically used when we have a continuous dependent variable (y, response) and a categorical independent variable (x, explanatory) containing three or more groups. As with the t-test, we assume that the error (variance) within groups is normal (we'll discuss in detail in Chapter 8). Most of the math behind ANOVA is basically the same as a t-test, but we add a couple steps for the third group, and now it essentially becomes the same thing as an F test (`var.test()` from Chapter 6[#Chapter6]) on three groups. In fact, the t-test, the F test, and one-way anova are all pretty much the same thing!

```
mind == blown
```

7.1.1 One-way analysis of variance

The number of grouping variables used in ANOVA confers different fancy naming conventions. The simplest of these contains a single grouping variable (e.g. treatment **or** year) and is referred to as one-way ANOVA. In theory, these models can be extended to include any number n of grouping variables (e.g. treatment **and** year) and are commonly referred to as n-way ANOVA.

Let's start by loading the `PlantGrowth` dataset in R:

```
data(PlantGrowth)
```

`PlantGrowth` is a dataframe with 30 observations of two variables. The first variable `weight` describes plant growth (in units of mass), and the second variable `group` contains control (`ctrl`) and two treatment groups (`trt1` and `trt2`) for individual plants. Have a look, as always:

```
str(PlantGrowth)
```

```
## 'data.frame': 30 obs. of  2 variables:
##  $ weight: num  4.17 5.58 5.18 6.11 4.5 4.61 5.17 4.53 5.33 5.14 ...
##  $ group : Factor w/ 3 levels "ctrl","trt1",..: 1 1 1 1 1 1 1 1 1 1 ...
```

Let's begin by using a one-way ANOVA to determine if the mass of plants differs between groups in `PlantGrowth`. In practice, this is *very* easy. First of all, though, we would report our **methods** something like this:

> We used a one-way analysis of variance (ANOVA) to estimate the effects of treatment group on the mass (g) of plants assuming a Type-I error rate of $\alpha = 0.05$. Our null hypothesis was that all group means were equal (H_0: $\mu_{ctrl} = \mu_{trt1} = \mu_{trt2}$).

Therefore, if any one of the means is not equal to the others, then we reject the null hypothesis.

You can fit an ANOVA using either the `aov()` function or the `lm()` function in base R. I prefer to use `lm()` for two reasons. First, there is output from `lm()` that I don't get with `aov()`. Second, the `lm()` function is the one we'll use for linear regression, multiple regression, and analysis of covariance. This reminds us that these models are all special cases of the glorious, over-arching group of general linear models Chapter 8 and will help us develop a standard workflow moving forward.

```
# Fit the model
model <- lm(weight~group, data=PlantGrowth)

# Print the model object to the console
model
```

```
##
## Call:
## lm(formula = weight ~ group, data = PlantGrowth)
##
## Coefficients:
## (Intercept)    grouptrt1    grouptrt2
##       5.032       -0.371        0.494
```

Wow, that is dangerously easy to do. But, this output is not very useful for getting the information we need if you don't already know what you are looking at. What we get here is essentially just one part of the information that we would like to (should) report.

We'll proceed with a more standard ANOVA table for now using the `anova()` function:

```
# Save anova table to an object
plant_nova <- anova(model)

# Have a look at the goodness
print(plant_nova)
```

```
## Analysis of Variance Table
##
## Response: weight
##           Df  Sum Sq Mean Sq F value  Pr(>F)
## group      2  3.7663  1.8832  4.8461 0.01591 *
## Residuals 27 10.4921  0.3886
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Okay, this is really what we needed in order to evaluate our null hypothesis: an ANOVA table with a break down of the residuals, mean squared errors, and test statistic(s). We interpret the test statistic and the p-value the same way we did in Chapter 6 when we did t-tests and Wilcox tests. And, we can now say:

> We found that the treatment had a significant effect on plant weight (ANOVA, $F = 4.846$, $df_1 = 2$, $df_2 = 27$, $p = 0.0159$).

We can also calculate the R^2 value for the ANOVA, which is a statistic used to describe the amount of variation in the data explained by the model relative to the total variation in the data set. More correctly, we are actually comparing the sum of squared errors for the model we fit (SSB) to the total sum of squares (SST = SSB + SSE).

For what it's worth, this is a super useful statistic for getting the big-picture perspective on whether your model is useful or crap. You calculate it pretty easily from the `anova()` output like as:

$$R^2 = \frac{SSB}{SST}$$

...or...

Why wouldn't you do this in R?

```
# Look at the names of the anova() output
# We want the "Sum Sq" bit
names(plant_nova)
```

```
# Here is the sum of squares for the model
# Have to use back-ticks for spaces, sigh
ssb <- plant_nova$`Sum Sq`[1]

# And the sum of squares total is the sum
# of the column in this case
sst <- sum(plant_nova$`Sum Sq`)
```

```
# Some quick division, and we get...
R2 <- ssb/sst
```

Have a look:

```
print(R2)
```

```
## [1] 0.2641483
```

Now, we can say that our treatment effect explained about 26% of the variation in the data. The rest is a combination error and unexplained variation in the data that might require further investigation.

The only problem here is that this is an awful lot of work to get something that should be really easy to do in R. And, we still don't know how `weight` varied between `groups`. We just know that at least one group is different from the other.

Thankfully, the default output of `summary()` for linear models fit with `lm()` does a lot of this for us.

```
# Print the summary of the model
summary(model)
```

```
##
## Call:
## lm(formula = weight ~ group, data = PlantGrowth)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -1.0710 -0.4180 -0.0060  0.2627  1.3690
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)   5.0320     0.1971  25.527   <2e-16 ***
## grouptrt1    -0.3710     0.2788  -1.331   0.1944
## grouptrt2     0.4940     0.2788   1.772   0.0877 .
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.6234 on 27 degrees of freedom
## Multiple R-squared:  0.2641,Adjusted R-squared:  0.2096
## F-statistic: 4.846 on 2 and 27 DF,  p-value: 0.01591
```

That's better, we get some useful information here. First of all, we get the value of the test statistic, the df, and the p-value for the model. We also get the R^2 for the model, 0.26, as part of the default output.

But, what if we want to know more about how treatment affected weight? Like, which groups are different? Can we use the p-values reported in the column `Pr(>|t|)` to infer group-wise differences? The quick answer is "sometimes".

The `Coefficients` chunk of this output can help us with inference in simple situations, and it really is the key to making predictions from our models (see Chapter 10).

Remember, most model output in R is stored as lists, so we can extract the coefficients table like this if we look at 'names(summary(model)) to find what we want:

```
coeffs <- data.frame( summary(model)$coefficients )
```

Okay, what is going on here? This looks nothing like the output we got from `anova()`.

The `coeffs` object is a dataframe with columns for mean coefficient estimates, the standard error of those estimates, t-statistic, and p-value. We are actually not going to worry about the p-values here for a hot minute.

Let's focus on the estimate column first. There are three values here. Each of these represents one of the factor levels in the `group` variable in `PlantGrowth`. They are assigned alpha-numerically, and the first (`ctrl`) is assigned as the `(Intercept)` or base level against which all others are compared. In this sense, the `(Intercept)` coefficient is an estimate of the mean value of `weight` for the `group` called `ctrl`.

Do a quick check:

```
# Calculate and print mean weight
# for the group ctrl in PlantGrowth
PlantGrowth %>%
  filter(group == 'ctrl') %>%
  summarize(avg = mean(weight))
```

```
##     avg
## 1 5.032
```

As you can see, the prediction from the ANOVA is identical to the group mean estimated directly from the data.

The coefficients for `grouptrt1` and `grouptrt2` can be thought of adjustments to the `(Intercept)` coefficient, or the difference between the mean of `ctrl` and

`trt1` or `trt2`. If the `Estimate` for `grouptrt1` or `grouptrt2` is negative, then the mean for that group is less than `ctrl` and if it is positive, the mean for the group is greater than `ctrl`.

If we wanted to calculate the mean `weight` of the `trt1` and `trt2` groups, we would add them to the `(Intercept)` coefficient like this:

```
# Assign model coefficients to objects
ctrl <- coeffs$Estimate[1]
trt1 <- coeffs$Estimate[2]
trt2 <- coeffs$Estimate[3]

# Calculate group means for trt1 and trt2
# from the model
trt1_prediction <- ctrl + trt1
trt2_prediction <- ctrl + trt2

print(c(trt1_prediction, trt2_prediction))
```

```
## [1] 4.661 5.526
```

If you calculate the means for these groups directly from the data you'll see that these values are identical to the mean `weight` of the `trt1` and `trt2` groups.

In Chapter 10 we will examine how to estimate confidence intervals around these estimates and make predictions from the model that include our uncertainty. But for that, we'll need to talk about a little bit of math and we're dealing with enough right now already!

Finally, the p-values associated with `trt1` and `trt2` indicates whether each group is significantly different from `ctrl`. In the case of the intercept, the p-value simply tells us whether the mean `weight` of `ctrl` is significantly different from zero. A fundamentally dull question - of course it is. This is the first time we really need to think about the differences between our statistical null hypotheses and our biological null hypotheses.

If we want to do further comparisons between groups (other than just comparing `trt1` and `trt2` to `ctrl` by themselves), then we need to add on a little "post-hoc" testing to find out which groups differ. We can use a 'pair-wise' comparison to test for differences between factor levels. Because this essentially means conducting a whole bunch of t-tests, we need a way to account for our repeated Type-I error rate, because at $\alpha = 0.05$ we stand a 1 in 20 chance of falsely rejecting the null even if it is true.

One tool that allows us to make multiple comparisons between groups while adjusting for elevated Type-I error is the Tukey HSD (honest significant difference) test. This test makes comparisons between each pairing of groups while controlling for Type-I error. Essentially, this makes it harder to detect differences

between groups but when we do we are more sure that they are not spurious ("Honest significant difference", say it with me).

Sound confusing? At least it's easy to do in R.

We need to recast our model this as an `aov` object in R...this is essentially the same thing as the `lm` function, but in a different wrapper (literally) that allows us to access the info in a different way. It would be a fine default function for doing ANOVA if we weren't interested in going any further with linear models.

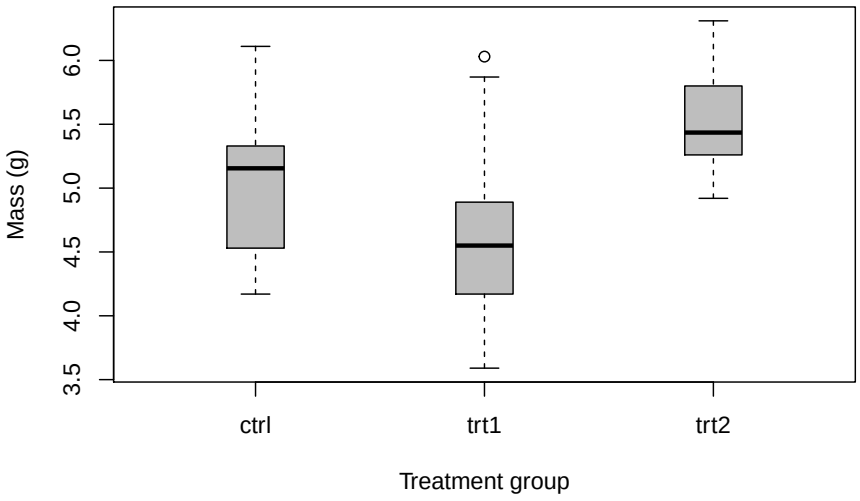
```
TukeyHSD(    # The function that does the Tukey test
  aov(       # A wrapper for lm objects
    model    # The model that we ran above
    )
  )
```

```
##   Tukey multiple comparisons of means
##     95% family-wise confidence level
##
## Fit: aov(formula = model)
##
## $group
##              diff        lwr       upr     p adj
## trt1-ctrl -0.371 -1.0622161 0.3202161 0.3908711
## trt2-ctrl  0.494 -0.1972161 1.1852161 0.1979960
## trt2-trt1  0.865  0.1737839 1.5562161 0.0120064
```

This report shows us exactly how `weight` differs between each pair of treatment groups. Here, we see that the only significant difference (`p adj < 0.05`) occurs between `trt2` and `trt1`.

For the readers, and for us, it may be easier to see this information displayed graphically:

```
# Make a boxplot of weight by group
# using the PlantGrowth data and
# base graphics
boxplot(
  weight~group,           # Relationship of interest as formula
  data=PlantGrowth,       # Name of the data set
  col='gray',             # Fill color for boxes
  ylab='Mass (g)',        # Y-axis label
  xlab='Treatment group', # X-axis label
  boxwex=.25
  )
```



In addition to what we wrote before, we can now say something along the lines of:

> "We found that the mass of plants in the trt2 group (5.5 ± 0.4 g) was significantly greater than plants in the trt1 group (4.7 ± 0.8 g; Tukey HSD, p = 0.012). However, we failed to detect differences in mass between plants in the control group (5.0 ± 0.6 g) and trt1 (p = 0.39) or trt2 (p = 0.20)."

7.1.2 Two(n)-way ANOVA

Next, we'll step up the complexity and talk about cases for which we have more than one grouping variable and some kind of numeric response. In these cases, we can use a two-way ANOVA (or 'n-way' depending on number of factors) to examine effects of more than one grouping variable on the response.

Here, we will use a data set describing differences in the mass of belly- button lint collected from males and females of three species of apes.

```
# Read in the data:
lint <- read.csv('data/lint.txt')
```

7.1.2.1 Main effects model

Now we can fit a model to the data. This will work the same way as for the one-way ANOVA above, but this time we will add more terms on the right hand side of the equation. We will start by looking at the *main effects* model for this data set.

What is a main-effects model? This model assumes that the response of interest, in this case the mass of belly button lint, `lintmass`, is affected by both `species` and `gender`, and that within species the effect of gender is the same. For example, the mass of belly button lint could be greater in one species compared to others, and if there is a difference between sexes we would expect that trend to be the same across species (e.g., boys always have more lint than girls - sorry guys, it's probably true!).

```
# Fit the model and save it to an object
lint.model<- lm(lintmass~species + gender, data = lint)

# Look at the summary of the model fit
summary(lint.model)
```

```
##
## Call:
## lm(formula = lintmass ~ species + gender, data = lint)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -2.5792 -0.9021  0.0875  0.8448  2.3917
##
## Coefficients:
##                  Estimate Std. Error t value Pr(>|t|)
## (Intercept)       15.5458     0.6133  25.346  < 2e-16 ***
## speciesSpecies 2  -3.4375     0.7512  -4.576 0.000183 ***
## speciesSpecies 3  -1.8750     0.7512  -2.496 0.021414 *
## genderMale         4.9083     0.6133   8.003 1.16e-07 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.502 on 20 degrees of freedom
## Multiple R-squared:  0.8096,Adjusted R-squared:  0.781
## F-statistic: 28.35 on 3 and 20 DF,  p-value: 2.107e-07
```

```
# Print the anova table
anova(lint.model)
```

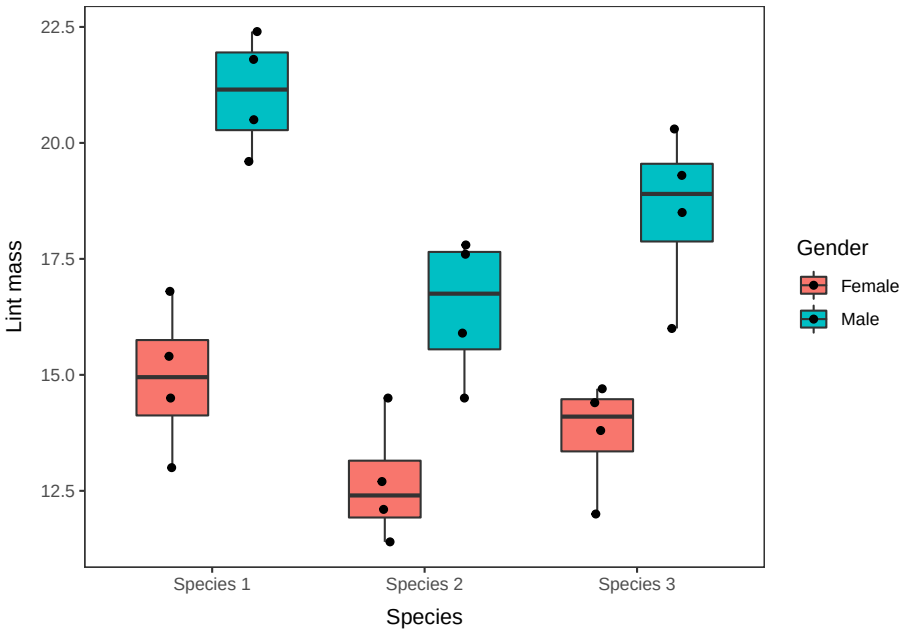
```
## Analysis of Variance Table
```

```
##
## Response: lintmass
##           Df  Sum Sq Mean Sq F value    Pr(>F)
## species    2  47.396  23.698  10.499 0.0007633 ***
## gender     1 144.550 144.550  64.041  1.16e-07 ***
## Residuals 20  45.143   2.257
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

As you can see, the output for the model is much the same as for the one-way ANOVA. The only real difference is that we have more than one grouping variable here. We conclude that there is a significant difference in `lintmass` between `species` ($F = 10.50$, $df = 2$, $p < 0.05$) and between `genders` ($F = 64.01$, $df = 1$, $p < 0.05$).

As before, we can make a quick boxplot and overlay our raw data to visualize these differences:

```
lint %>%
  ggplot(aes(x = species, y = lintmass, fill=gender)) +
    geom_boxplot() +
    geom_point(position=position_jitterdodge(.1)) +
    xlab('Species') +
    ylab('Lint mass') +
    labs(fill = "Gender") +
    theme_bw() +
    theme(axis.title.x = element_text(vjust=-1),
          axis.title.y = element_text(vjust=3),
          panel.grid = element_blank()
          )
```



Hopefully after seeing these results you are now starting to realize how important a few well-placed figures and tables can be for clearly communicating the results of your research (even if it is about belly-button lint).

The math for making predictions becomes a little more complicated once we add a second grouping variable. Even the numbers of pair-wise comparisons can become overwhelming in a simple situation like this. Therefore, we'll hold off on digging too much deeper into the math until next week.

7.1.2.2 Interaction terms

The n-way ANOVA is the first kind of model we have used in which it is possible to consider *interactions* between two or more factors. An interaction occurs when the effects of two or more factors are not additive. This means that the effect of `gender` might change for different `species`. For example, let us consider the following scenario in the `lint` data.

Perhaps we hypothesize that lint accumulation in the belly buttons of females differs in pattern from males due to social grooming patterns and sex-specific behavioral patterns favoring females in only certain species. As a result, we might expect that `gender` and `species` could have some kind of non-additive effect on `lintmass` in these apes such that there are significant, sex-specific differences only in some species. To test this, we would use the following:

```
# Fit a new model that includes an interaction, signified by '*'
lint.modeli <- lm(lintmass~species * gender, data=lint)

# Summarize the model
summary(lint.modeli)
```

```
##
## Call:
## lm(formula = lintmass ~ species * gender, data = lint)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -2.525 -0.750  0.050  1.019  1.875
##
## Coefficients:
##                             Estimate Std. Error t value Pr(>|t|)
## (Intercept)                  14.9250     0.7404  20.159 8.40e-14 ***
## speciesSpecies 2             -2.2500     1.0471  -2.149   0.0455 *
## speciesSpecies 3             -1.2000     1.0471  -1.146   0.2668
## genderMale                    6.1500     1.0471   5.874 1.46e-05 ***
## speciesSpecies 2:genderMale  -2.3750     1.4808  -1.604   0.1261
## speciesSpecies 3:genderMale  -1.3500     1.4808  -0.912   0.3740
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.481 on 18 degrees of freedom
## Multiple R-squared:  0.8335,Adjusted R-squared:  0.7873
## F-statistic: 18.03 on 5 and 18 DF,  p-value: 1.898e-06
```

```
# Print an ANOVA table for the model
anova(lint.modeli)
```

```
## Analysis of Variance Table
##
## Response: lintmass
##                Df  Sum Sq Mean Sq F value    Pr(>F)
## species         2  47.396  23.698 10.8079 0.0008253 ***
## gender          1 144.550 144.550 65.9253 1.983e-07 ***
## species:gender  2   5.676   2.838  1.2943 0.2984104
## Residuals      18  39.468   2.193
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Alas, in the case of the lint model, this interaction is not significant, so we lack

the evidence we would need to say that lint accumulation changes differently between genders within species.

7.2 Simple linear regression

We have now considered the case of what to do when we have a numerical response and categorical explanatory variable(s) with any number of groups or grouping variables. But, what if we have both a numerical response and numerical explanatory variables? Fear not, there is a stat for that! Now we are entering the realm of correlation and regression. Next week, we'll show that ANOVA is, in fact, just a special kind of regression.

When we fit a linear regression model, we are trying to explain relationships between some response of interest (dependent variable y) and one or more explanatory (independent) variables, x.

As with all linear models the goal of regression analysis is, in it's simplest sense, to fit a line through all of the points in bivariate space that minimizes the distance between the points and a line of the form:

$$y = mx + b$$

That ought to look familiar!

In the case of statistics, we usually represent the formula for a line like this:

$$Y_i = \beta_0 + \beta_i X_i$$

We are ignoring an important part of these statistical models for now. In most cases, though, we will be estimating a parameter for the intercept and one parameter for each explanatory variable of interest.

7.2.1 Simple linear regression

Since most folks are probably more familiar with linear regression than with ANOVA whether they know it or not, we'll jump right into this one with an example using the `swiss` data.

These data are for fertility and infant mortality rates as related to a number of socio-economic indicators. Take a moment to look at them:

```
data(swiss)
```

You can see the description of these data by looking at the help file for the data set as always. Have look on your own:

```
?swiss
```

Now, let's get cracking.

We'll start by fitting a simple model and then build complexity.

Fit a model that relates fertility to education level. Notice that this looks exactly the same as the call to `lm` for the ANOVAs above? That's because they are the same thing and people have been lying to you your whole life. Perhaps it took reading The Worst Stats Text eveR to learn it? If so, I aplogize for your misfortune.

```
# Fit the model and assign it to a named object
fert_mod <- lm(Fertility ~ Education, data = swiss)

# Summarize the model
summary(fert_mod)
```

```
##
## Call:
## lm(formula = Fertility ~ Education, data = swiss)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -17.036  -6.711  -1.011   9.526  19.689
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  79.6101     2.1041  37.836  < 2e-16 ***
## Education    -0.8624     0.1448  -5.954 3.66e-07 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 9.446 on 45 degrees of freedom
## Multiple R-squared:  0.4406,Adjusted R-squared:  0.4282
## F-statistic: 35.45 on 1 and 45 DF,  p-value: 3.659e-07
```

The `(Intercept)` term in this summary is the y-intercept from our formula for a line and the `Education` coefficient is the slope of the line. Our intercept tells us that mean `Fertility` (y) is about 79.6 when `Education` (x) is zero. Note that this interpretation does not change even if we did not observe an education of zero in the data - something to think about in the weeks to come. The p-value for the intercept just tells us that this value is significantly different from zero (snores).

The p-value for the `Education` coefficient tells us that the slope of the line is also significantly different from zero. Because this number is negative, we know that

there is an inverse relationship between `Education` and `Fertility`. In other words, more highly educated individuals have fewer children. You can tell this is an inverse relationship because of the minus sign in front of the coefficient for `Education`. We know that the relationship is significant because of the small p-value and corresponding significance codes.

We explained a little more than 40% of the variability in the response with this one explanatory variable if we look at the R^2 value that is returned (we'll work with the `Multiple R-squared` by default).

This is as far as the summary goes for linear regression for now. That is, we don't need the ANOVA table to assess significance any more because we have no factors - just continuous variables. What we end up with in this summary are the coefficients that can be used to describe the line that passes through the data and minimizes the residual errors (that's the part we ignored above).

> WHAT??

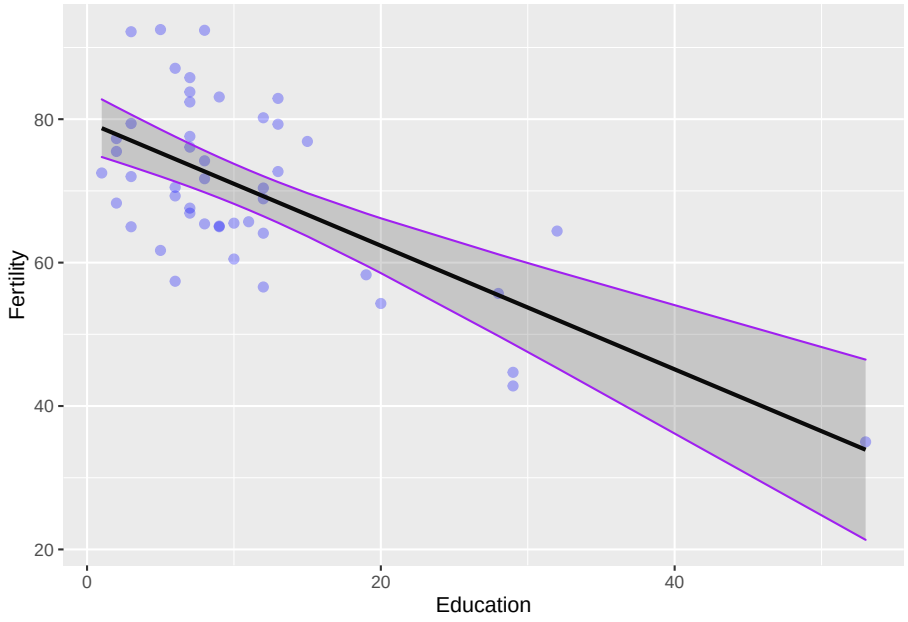
Let's explain this by actually looking at the data and plotting our model over the top of it.

First, we'll use the built-in `predict()` function to create a trend line and a prediction interval. We'll dig deeper into how to do this in Chapter 10.

```
# Make predictions from the fitted model object using observed data
predicted_fertility = predict(fert_mod, interval = 'confidence')

# Add these to the swiss data
swiss_pred <- cbind(swiss, predicted_fertility)
```

Now, we can plot the raw data as a scatterplot and add our model estimates over the top. You should notice that the confidence interval is much wider at high values of `Education` because there are few data points and thus more uncertainty in that part of the data.

```
# Sets up data and aesthetics
ggplot(swiss_pred,
       aes(x = Education, y = Fertility)) +
  # Adds raw data as points
  geom_point(colour = 'blue', fill = 'blue', alpha = 0.3, size = 2) +
  # Adds regression line
  geom_line( aes(y = fit), size = 1) +
  # Adds 95% confidence interval
  geom_ribbon(aes(ymin = lwr, ymax = upr), color = 'purple', alpha = .2) +
  # Adds sweet style tweaks of your choosing
  theme(legend.position = "none")
```



Again, dangerously easy.

7.3 Multiple linear regression

We can, of course, extend this to include multiple continuous explanatory variables of interest just as we did with ANOVA for multiple categorical explanatory variables!

Here is an example to whet your appetite. Let's say we want a multiple regression model that includes both `Education` and `Catholic`?

```
multiple_mod <- lm(Fertility ~ Education + Catholic, data = swiss)

summary(multiple_mod)
```

```
##
## Call:
## lm(formula = Fertility ~ Education + Catholic, data = swiss)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -15.042  -6.578  -1.431   6.122  14.322
##
## Coefficients:
```

```
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 74.23369    2.35197  31.562  < 2e-16 ***
## Education   -0.78833    0.12929  -6.097 2.43e-07 ***
## Catholic     0.11092    0.02981   3.721  0.00056 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 8.331 on 44 degrees of freedom
## Multiple R-squared:  0.5745,Adjusted R-squared:  0.5552
## F-statistic:  29.7 on 2 and 44 DF,  p-value: 6.849e-09
```

Or if we really want to get crazy with the hot sauce:

```
full_mod <- lm(
  Fertility ~ Agriculture + Examination + Education + Catholic,
  data = swiss
  )

summary(full_mod)
```

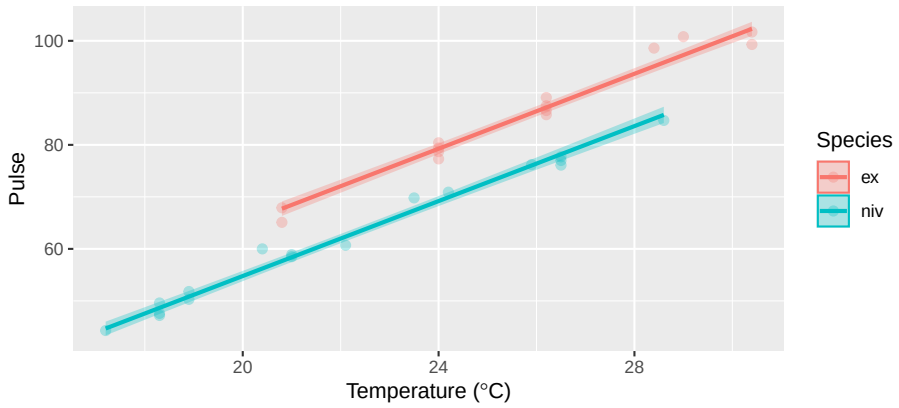
```
##
## Call:
## lm(formula = Fertility ~ Agriculture + Examination + Education +
##     Catholic, data = swiss)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -15.7813  -6.3308   0.8113   5.7205  15.5569
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 91.05542    6.94881  13.104  < 2e-16 ***
## Agriculture -0.22065    0.07360  -2.998  0.00455 **
## Examination -0.26058    0.27411  -0.951  0.34722
## Education   -0.96161    0.19455  -4.943 1.28e-05 ***
## Catholic     0.12442    0.03727   3.339  0.00177 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 7.736 on 42 degrees of freedom
## Multiple R-squared:  0.6498,Adjusted R-squared:  0.6164
## F-statistic: 19.48 on 4 and 42 DF,  p-value: 3.95e-09
```

7.4 Next steps

During the next couple of weeks, we'll try to figure out a way to deal with this rat's nest of different explanatory variables and how they are interpretted. But first, we'll talk about combining ANOVA and linear regression into a general linear model (analysis of covariance) in Chapter 8 and how to assess assumptions (Chapter 9) and communicate our results effectively (Chapter 10).

Chapter 8

General linear models



This is not the picture from Chapter 4. It is a new picture but it looks just like that one. That's because they are both linear models. This one just has two intercepts!

8.1 Analysis of covariance (ANCOVA)

Alright, to wrap up our crazy, eye-opening introduction to linear models we are going to unleash the power of ANCOVA, or the *general linear model*. Hopefully the power and limitations of this approach will be readily apparent to you. If not, we will talk about them a lot more so don't worry.

ANCOVA is the way into the world of real, complex data analyses. It will serve as the foundation for the next several weeks in this course. Get to know it well, it is your friend. That said, ANCOVA is just another type of linear model (see

Chapter title!), so it really doesn't need it's own book chapter except that it sounds scary to people. Plus, this *is* The Worst Stats Text eveR.

We won't spend a tone of time on the development of these models as we covered most of the important ideas in Chapter 4. Instead, we are going to jump right in with an example. We'll need the `tidyverse` for this chapter, as well as the data contained in `crickets.txt`. Go ahead and load the `tidyverse` now so you don't forget.

```
library(tidyverse)
```

8.2 Motivation

So here we are:

We have multiple explanatory variables that we would like to test; some are factors and some are continuous. Each of those factors has some set of statistical and biological hypotheses associated with them as related to our continuous (and still normal) response of interest. We want a nice elegant way of wrapping these all in to one analysis.

How in the world are we supposed to do that? It's easier than you might think.

8.3 Data

Read in a new data set.

This data set contains pulses of two species of crickets collected under varying temperatures.

```
# Read in the cricket data and assign it to a named object
crickets <- read.csv('data/crickets.txt')

# Have a look
head(crickets)
```

```
##   Species Temp Pulse
## 1      ex 20.8  67.9
## 2      ex 20.8  65.1
## 3      ex 24.0  77.3
## 4      ex 24.0  78.7
## 5      ex 24.0  79.4
## 6      ex 24.0  80.4
```

8.4 Analysis

Here we want to investigate the effects of species and temperature on pulses of individual crickets. Our null hypotheses are that there is no difference in `Pulse` between `Species` and no change in `Pulse` with increasing temperature. We conduct the test at the default $\alpha = 0.05$.

We use the `lm()` function to fit the model, and the formula looks identical to the main-effects ANOVA and linear regression models from Chapter 4. Isn't that handy?

```
# Fit the model
cricket_mod <- lm(Pulse ~ Species + Temp, data=crickets)
```

Install the `car` package. We need a function from this package for model summary because now we have a mix of categorical and continuous explanatory variables. This means we want to calculate the sums of squared errors a little differently than we did before.

```
# Load the package after it's installed
library(car)
```

Now we create the ANVOA table for our ANCOVA model

```
car::Anova(cricket_mod, type='III')
```

```
## Anova Table (Type III tests)
##
## Response: Pulse
##             Sum Sq Df   F value    Pr(>F)
## (Intercept)   25.5  1    7.9906  0.008582 **
## Species      598.0  1  187.3994 6.272e-14 ***
## Temp        4376.1  1 1371.3541 < 2.2e-16 ***
## Residuals     89.3 28
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

And we can look at the summary:

```
summary(cricket_mod)
```

```
##
## Call:
```

```
## lm(formula = Pulse ~ Species + Temp, data = crickets)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -3.0128 -1.1296 -0.3912  0.9650  3.7800
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)  -7.21091    2.55094  -2.827  0.00858 **
## Speciesniv  -10.06529    0.73526 -13.689 6.27e-14 ***
## Temp          3.60275    0.09729  37.032  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.786 on 28 degrees of freedom
## Multiple R-squared:  0.9896,Adjusted R-squared:  0.9888
## F-statistic:  1331 on 2 and 28 DF,  p-value: < 2.2e-16
```

We see that there are significant effects of species and temperature on the pulse of individual crickets. Everything else proceeds as in the analyses Chapter 4! We can build in complexity as needed, and we can make predictions as we did before.

8.5 Predictions

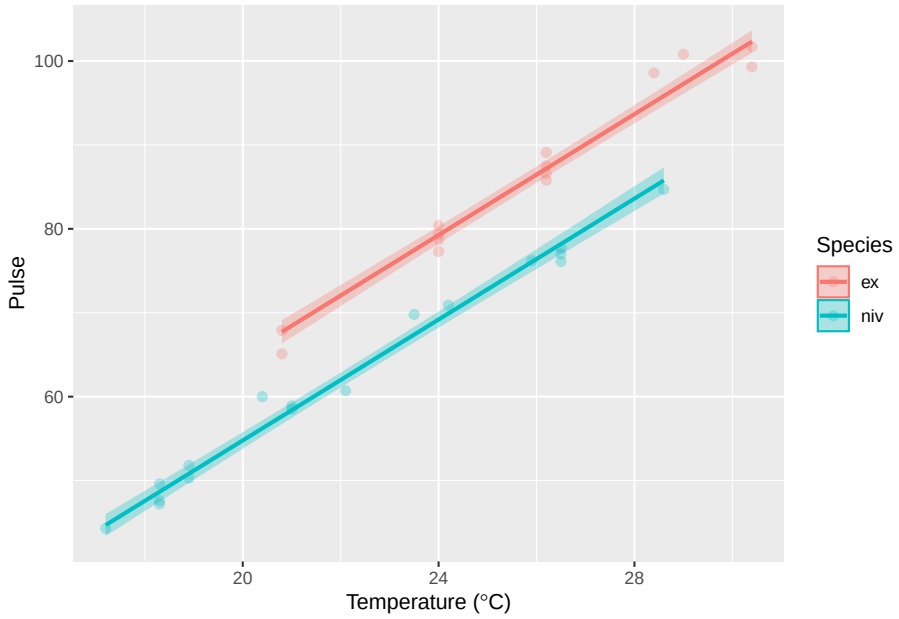
Here we will take a quick look at how to plot model predictions over our raw data to demonstrate the relationships we have discovered and to show how they compare to our observations. We should have a separate line for each group based on differences in `Pulse` between species, but the lines should be parallel based on how our model was formulated. Again, we will dig deep into why this is the case in Chapter 10.

Note that this procedure is identical to the one we used for linear regression. That is because linear regression is just one special case of the general linear model!

```
# Make predictions from the fitted model object using observed data
predicted_pulse = predict(cricket_mod, interval = 'confidence')

# Add these to the cricket data
cricket_pred <- cbind(crickets, predicted_pulse)
```

Now, we can plot the raw data as a scatterplot and add our model estimates over the top just like we did for the `swiss` data in Chapter 4.

```
# Sets up data and aesthetics
ggplot(cricket_pred,
       aes(x = Temp, y = Pulse, color = Species, fill = Species)) +
  geom_point(alpha = 0.3, size = 2) +
  geom_line(aes(y = fit), size = 1) +
  geom_ribbon(aes(ymin = lwr, ymax = upr, color = NULL), alpha = .3)+
  xlab(expression(paste('Temperature (', degree, 'C)')))
```



8.6 Next steps

Now that you hold real power in your hands to do data analysis, we need to to have our first talk about due diligence and assumptions of the statistical models that we use.

There are three fundamental assumptions that we either need to validate or address through experimental design in this class of models.

1. Independence of observations.
2. Normality of residuals (with mean = 0)
3. Homogeneity of variances

We will discuss what each of these means and how to assess them in Chapter 9. During remaining Chapters, we will continue to discuss methods for verifying or relaxing these assumptions to meet our needs through specific techniques.

Chapter 9

Assumptions of linear models

Statistics is like a fine-tuned machine that relies on many moving parts to work reliably, unlike the broken watch in the image above. What, you expected a working watch? Maybe you need to **check your assumptions**! This is The Worst Stats Text eveR. Just goes to show that even the fanciest model is useless if you don't validate that it works.

9.1 Introduction

In this chapter, we will start by taking a step back for an in-depth look at the assumptions we make when we fit parametric models to data in an effort to explain the effects of explanatory variables on some response of interest, using linear models as the backdrop for our discussions. In previous chapters we learned how to fit linear models. The purpose of this chapter is to provide you with the tools you need on the front end and the back end of that process so we are surrounding linear models with the goodness they deserve.

We will also continue to talk about linear models that include multiple explanatory variables. Specifically, we will discuss how relationships between these variables might influence which ones we include in a given model and how we make defensible decisions when it comes to these choices. We will further probe the concept of the R^2 statistic as a measure of model fit, and how this is influenced by the inclusion of multiple explanatory variables.

Finally, we will conclude our discussions this week with tools for communicating the results of our analyses once we have verified that we are not in major violation of assumptions in Chapter 10. To do this, we will need to look a little more

closely at the math behind linear models (not too closely!) and what exactly we are doing when we fit a linear model. These discussions will include the essential concepts of main effects, interaction effects, and response 'surfaces' for the case in which we include more than one explanatory variable. Please keep in mind that although we are using strictly linear models to introduce these concepts their application in the suite of models that we will discuss for the next several weeks is virtually identical, and we will discuss exactly why this is.

We'll be working with the functions from various packages in the `tidyverse` and with the `turtles.txt` data file for this chapter. You'll also need to install the `GGally` package if you don't have it. Go ahead and load those in your code whenever you are ready to get started. I'll keep track of how long it takes on my broken watch.

9.2 Assumptions of linear models

From last week:

> Now that you hold real power in your hands to do data analysis, we need to to have our first talk about due diligence and assumptions of the statistical models that we use. There are three fundamental assumptions that we either need to validate or address through experimental design in this class of models.
>
> **1.** Independence of observations. **2.** Normality of residuals (with mean=0) **3.** Homogeneity of variances (i.e. homoscedasticity)

We will discuss what each of these means in class this week, and during the next several weeks we will discuss methods for verifying these assumptions or relaxing the assumptions to meet our needs through specific techniques.

9.3 WTF is a residuals?

Up until now, we've been talking about the formula of a line in geometric terms as $y = mx + b$ or $y = \beta_0 + \beta X$. In Chapter 7 we extended this simple linear form to be:

$$y = \beta_0 + \beta_1 X_1 ... + \beta_k X_k$$

or

$$\sum_{k=1}^{K} \beta_0 + \beta_k X_k$$

for however many K explanatory variables we may wish to include in a linear model. That's gross, but it's about to get grosser. (More gross? Who cares, this is The Worst Stats Text eveR - go Google it)

In this chapter we are going to acknowledge for the first time that it has all been a lie even though those summation symbols really make this book look more official.

From now on, we are going to think about linear models, and all their generalizations or specializations, like this:

$$y = \beta_0 + \beta X + \epsilon$$

or

$$\sum_{k=1}^{K} \beta_0 + \beta_k X_k + \epsilon$$

if you like that one better.

> Don't freak out. The only thing that has changed is that we added an error term.

The error term, ϵ, is called the **residual error**. For grouping variables, it is **the difference between each i$^{\text{th}}$ observation x and the mean** ($\bar{x}$):

$\epsilon_i = x_i - \bar{x}$

This should look really familiar if you've seen the formula for the variance of a normal distribution (which you have because you definitely read and understood Chapter 5):

$$\sigma^2 = \frac{\sum_{i=1}^{n}(x - \bar{x})^2}{n - 1}$$

9.3.1 Residuals in ANOVA

The error for each observation is calculated relative to both the grand mean and group-specific means for each observation (data point) in ANOVA. And, these errors are directly related to the calculation of the sum of squares calculations we talked about for t-tests in Chapter 6 and ANOVA in Chapter 7. As an example of what this looks like, we can calculate the residual error (ϵ) of `Petal.Length` for each `Species` in the `iris` data like this:

```
# Load the iris data
data(iris)

# Calculate mean of each group
means <- iris %>%
  group_by(Species) %>%
  summarise(x_bar = mean(Petal.Length))

# Have a look
means
```

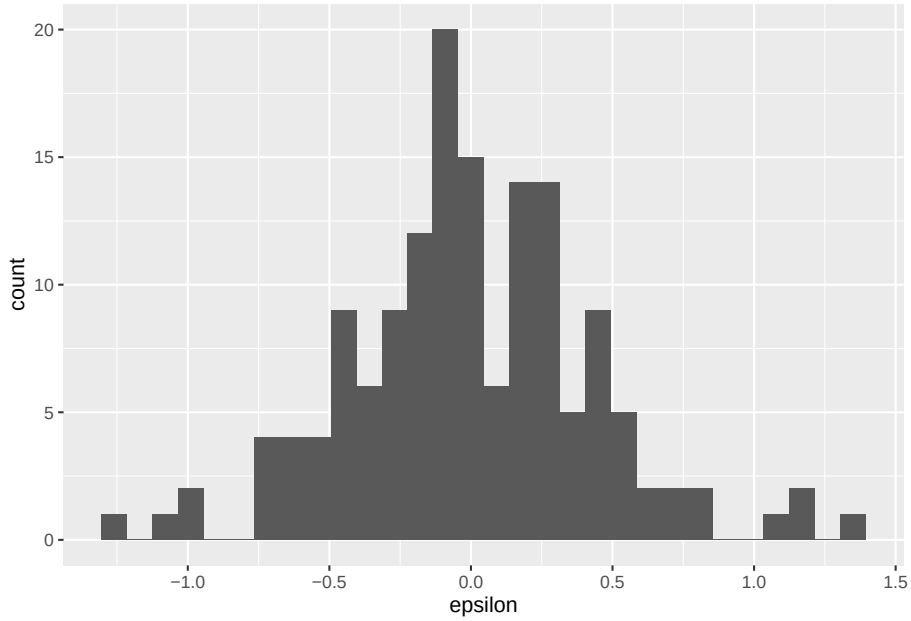
```
## # A tibble: 3 x 2
##   Species    x_bar
##   <fct>      <dbl>
## 1 setosa      1.46
## 2 versicolor  4.26
## 3 virginica   5.55
```

If we merge these group `means` with the `iris` data, it is really easy to calculate the error for each observation in each `Species`, or group:

```
# Merge them. R will use "Species"  in both by default
resid_df <- merge(iris, means)

# Calculate residual error:
resid_df$epsilon <- resid_df$Petal.Length - resid_df$x_bar
```

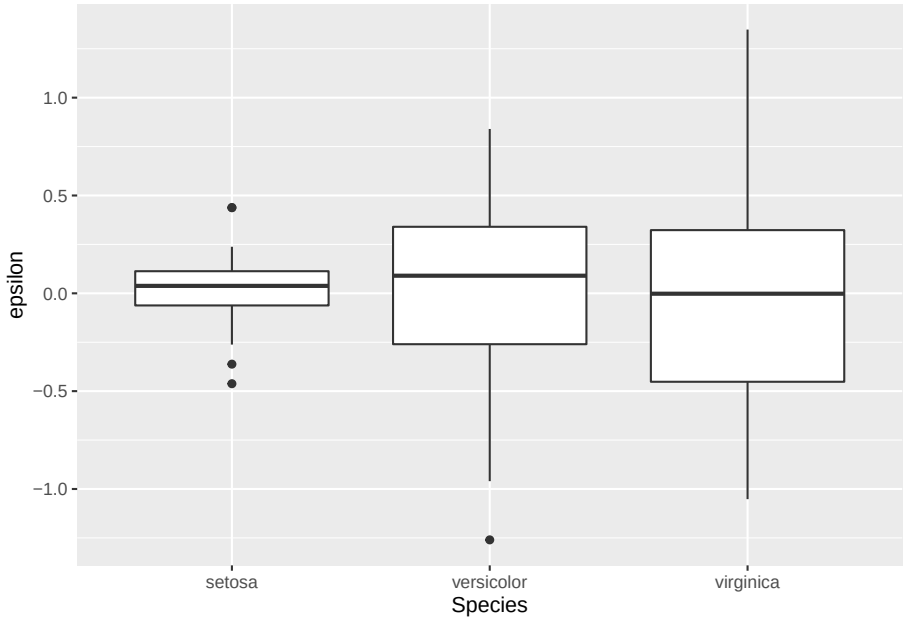
We can make a histogram of the residuals to confirm the assumption that the residuals are normally distributed with a mean of zero. This assumption is important because it allows us to drop ϵ from the equations above and fall back to our old friend $y = mx + b$. As you can see below, the mean of our residuals is about zero, and the distribution of residuals also appears to be symmetrical (normal).

```
ggplot(resid_df, aes(x = epsilon)) +
  geom_histogram()
```



We could also examine residuals within `Species` using a box plot. Again, we should see that our residuals are normally distributed with a mean of zero within groups. However, you may notice that the variance of ϵ is clearly not equal between groups.

```
ggplot(resid_df, aes(x = Species, y = epsilon)) +
  geom_boxplot()
```



9.3.2 Residuals in linear regression

For linear regression (continuous X), the residuals are calculated as the difference between each data point (x) and the corresponding prediction of $\hat{y}$ at that value of x from the line of best fit ($\epsilon_i = x_i - \hat{y}$). These are referred to as `fitted` (x) and `predicted` ($\hat{y}$) values in R.

Here's some code in case the math isn't doing it for you. Don't worry, we'll make some graphs, too.

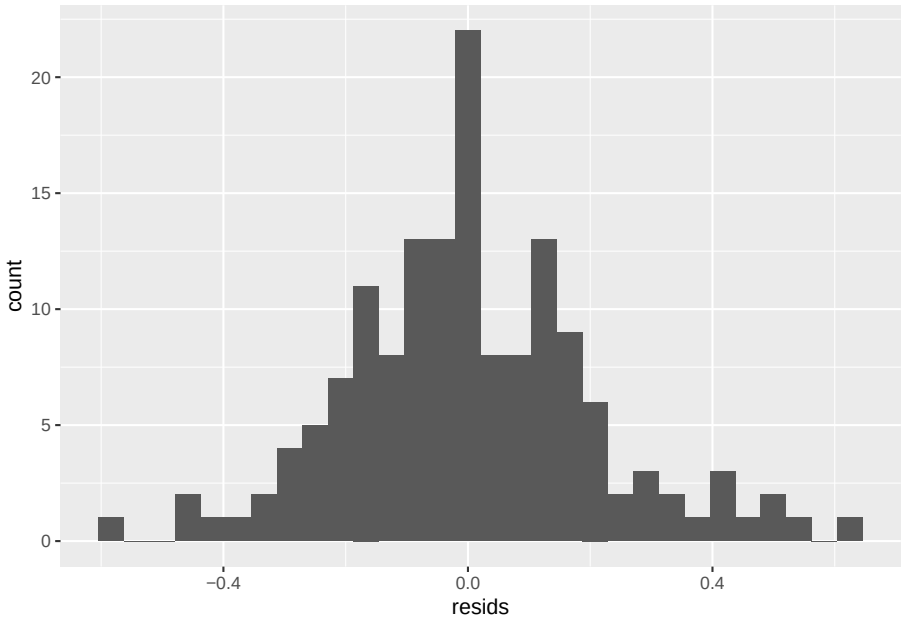
```
# Fit a linear regression to estimate change
# in Petal.Width with Petal.Length
fit_lm <- lm(Petal.Width ~ Petal.Length, data = iris)

# Now extract the residuals from
# the fitted model object
resids <- fit_lm$residuals
```

The order of values in the vector `resids` in the code above matches the order of the data in `iris`, so we can combine these as we did above:

```
iris$resids <- resids
```

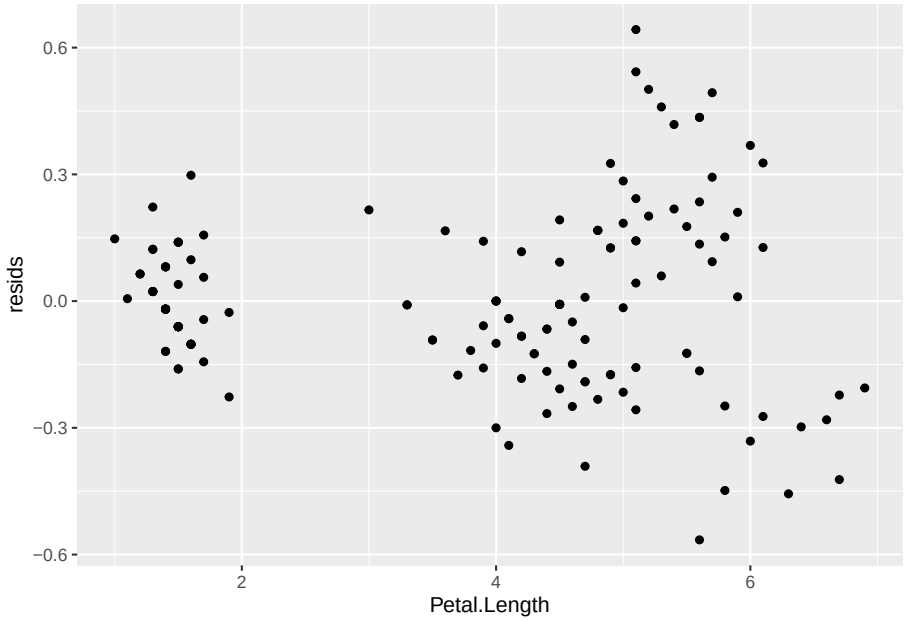
And now we can make a histogram to see if they are normal with a mean of zero.

```
ggplot(iris, aes(x = resids)) +
  geom_histogram()
```



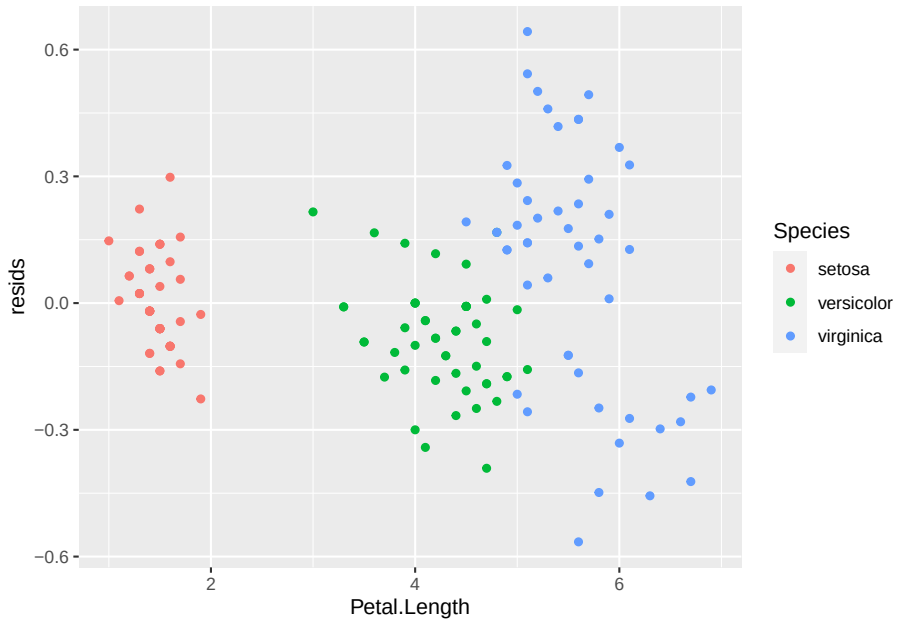
This also allows us to determine whether there are any changes in the residuals along the range of x values to assess whether we have satisfied the assumption of independence of observations. To do this, we just need to plot the residuals against the fitted values (the data in the `iris$Petal.Length` column).

```
ggplot(iris, aes(x = Petal.Length, y = resids)) +
  geom_point()
```



If we've met assumptions of independence of observations, the plot above should look like random scatter from left to right and top to bottom. Looks like that is not the case here because the group of data on the left have a much lower spread of residuals than the rest of the data. In fact, if you color by `Species` it becomes obvious that these are samples for `setosa`.

```
ggplot(iris, aes(x = Petal.Length, y = resids, color = Species)) +
  geom_point()
```

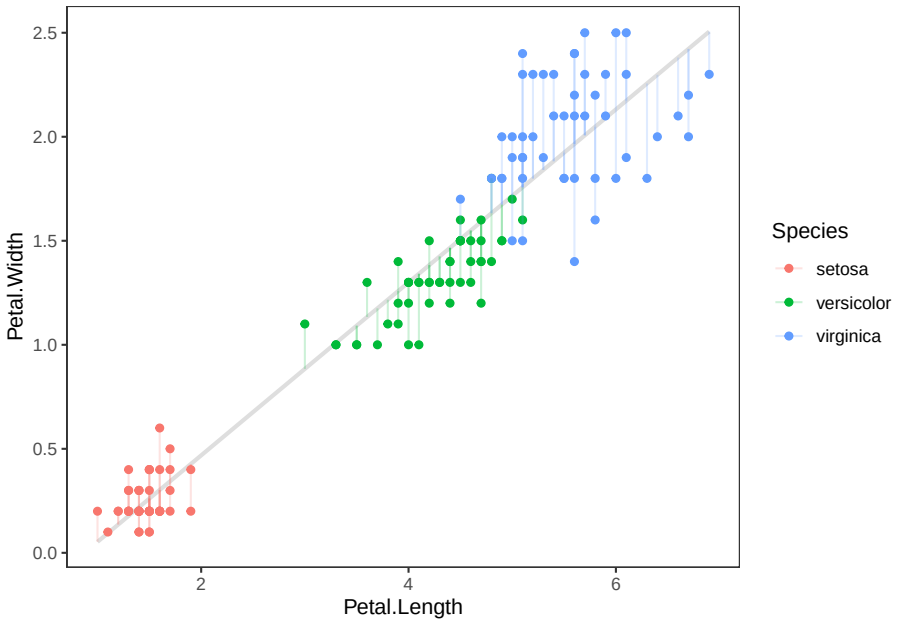


Boo `setosa`!

Finally, if the code doesn't do it for you, we can graph the regression to see what residuals actually look like for our model. It is the squared sum of these errors, specifically, which R is trying to minimize when it estimates the coefficients for the formula of our line. That is why we talk about "sums of squares" in ANOVA tables.

Here is a visual representation of residuals. The points are our raw data, the diagonal line is our model prediction, and the vertical lines represent the residual error for each observation.

```
# Make predictions from the model
y_hat <- predict(fit_lm)

ggplot(iris, aes(x = Petal.Length, y = Petal.Width, color = Species) ) +
  geom_smooth(method = "lm", se = FALSE, color = "gray87") +
  geom_point() +
  geom_segment(aes(xend = Petal.Length, yend = y_hat), alpha = .2) +
  theme_bw() +
  theme(panel.grid = element_blank())
```



So, now that we know what residuals *are* or at least what they *look like* we can talk about how they are used.

We will keep making tweaks to our equation in Chapter 10 when we start to think of the linear model more accurately as $y = \beta X + \epsilon$ or $\sum_{k=1}^{K} \beta_k X_k + \epsilon$ to unify the t-test, ANOVA, linear regression, and ANCOVA into a single general framework (the general linear model you def read about in Chapter 8).

9.4 The turtle problem

Let's get some data to demonstrate these assumptions.

These are data that were collected 2013-2015 for Kemp's Ridley sea turtles incidentally caught by anglers in the Gulf of Mexico. After being caught, the turtles were taken to a wildlife rehabilitation center so they could have fishing hooks removed and recover.

```
# Read in the turtles data,
# It's a bit messy, so we will read it
# in with an extra option to strip white spaces.
turtles = read.csv('data/turtles.txt', header = TRUE, strip.white = TRUE)
```

Here is a quick explanation of the variables (columns) in the dataframe:

`ID`: turtle ID `Year`: year of capture `Gear`: the gear type with which the turtle was hooked `Width`: the gape width of the hook `Removed`: the location from which the hook was removed `Status`: survived (1) or did not (0) `Stay`: length of stay in the rehab facility `nHooks`: Number of hooks in the turtle

We will use `Stay` as the response variable here. This is a great data set because `Stay` has all kinds of problems related to assumptions of linear models that require analyzing it in a different framework than those we have discussed so far (or will for a few weeks!).

9.5 Data exploration

9.5.1 Independence of observations

This assumption basically means that each row of your data was collected independently of all others. In other words, no two rows of your data are related to one another.

We can relax this assumption by explicitly including variables and constructs within the models that actually account for these kinds of relationships. For example, in one-way ANOVA we include grouping (factor) variables to to account for non-independence of some observations. In fact, this lack of independence is often the very thing we are interested in testing! In ANOVA, we are interested in whether individuals from the same group respond in the same way. Note that this in turn places the assumption of independence on individual measurements within our groups. It's turtles all the way down. (I say this a lot. If you don't know what it means go Google it.)

This assumption really should be be addressed within the context of experimental design. Violations generally require alternatives to the simple cases of one-way ANOVA, ANCOVA or simple linear regression that we will discuss in later chapters. We will discuss specific extensions of our basic linear models (ANOVA and regression) to relax more difficult violations such as repeated observations, and temporal or spatial autocorrelation among observations. Although we can't cover all of these extensions in one book or a single bottom-up biometry class, we can point you in the right direction for most of them.

You: *Get to the point, what are we looking for here*?

Me: *Sorry.* [writes rest of chapter]

For linear models, we want data that were sampled randomly and independently from all other data points. For this information, we have to examine the actual experimental design. In a best case, an experiment is designed intentionally so that all groups get opposite treatments and those there is no correlation (relationship) between treatments ("orthogonal design"). This is easy to achieve with some thought in the design of controlled experiments, but can be difficult

or impossible to do in semi-controlled experiments or observational studies. This is one reason why controlled experimentation has long been thought of as the gold standard in science.

There is one obvious thing that is going to tell us that the observations in `turtles` are not collected independently, but there are a few others. What is it? You can probably infer the answer just from the variable names.

```
head(turtles, 10)
```

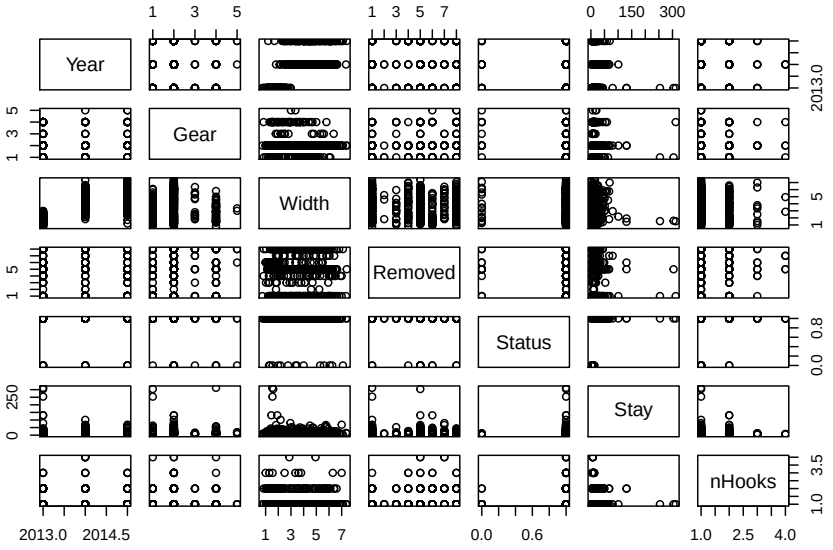
```
##              ID Year   Gear Width Removed Status Stay nHooks
## 1       AL-002 2013      J  1.45      UE      1    9      1
## 2  AL-LT14-001 2014      J  6.25      ME      0    4      1
## 3  AL-LT14-003 2014      J  5.45     NaN      1   16      1
## 4  AL-LT14-004 2014  Kahle  2.77       B      0    6      1
## 5  AL-LT14-006 2014      J  4.75      ME      1   49      2
## 6  AL-LT14-006 2014 Circle  2.92     NaN      1   49      2
## 7     LT13-001 2013  Kahle  2.22      ME      1   29      1
## 8     LT13-003 2013      J  1.34      ME      0    4      1
## 9     LT13-004 2013      J  1.63      ME      1   33      1
## 10    LT13-008 2013      J    NA      LE      1   26      2
```

If you look closely, you'll notice that we have repeated observations of individuals here. So, we already know that our data do not conform to the assumptions of ANOVA, linear regression, or ANCOVA - but let's keep using these data for the sake of demonstration. There is, of course another major violation of our assumptions that has to do with experimental design (that is commonly violated): these are **discrete** data! We'll pretend for now that we can think of `Stay` as a continuous variable, though.

You can see how this could get tedious to do for every level of every grouping factor and then for each continuous variable. So, we can also take a "shotgun" approach and look at how variables are related to one another.

This approach allows us to look at relationships between all variables at once. Note below that I am using all columns except `ID` because that graph would be a mess.

```
# Look at correlations between variables
pairs(turtles[ , 2:ncol(turtles)])
```



But, for this data set this is really gross and difficult to interpret due to the nature of the variables. This should tell you that thinking about this stuff before you collect data is really important!

It would make a lot more sense for something like the built-in `swiss` data because those variables are all continuous. You can check that out by running `pairs(swiss)` in your code. Look at all of the related variables in the `swiss` data! This relatedness is called **collinearity** and can cause some issues if we are not careful about it.

We'll talk about problems related to correlation between variables (e.g. temperature and photoperiod) in detail when we discuss model selection and collinearity. In the sections that follow, we'll just focus our efforts on diagnosing `Year` as a categorical explanatory variable and `Width` as a continuous explanatory variable.

9.5.2 Normality

In all linear models we make the assumption that the residual error of our model is normally distributed with a mean of zero. This allows us to drop the error term, ϵ from computation in the model fitting and allows us to calculate an exact solution in the case of ANOVA and linear regression. (Technological advances have really made this unnecessary because we can solve everything through optimization now).

There are a multitude of tools at our disposal for examining normality of the

residuals for linear models. One option is to examine group-specific error structures as a surrogate for residual error prior to analysis. The other option is to examine diagnostic plots of residuals directly from a fitted model object in R or other software programs (this is actually the more appropriate tool).

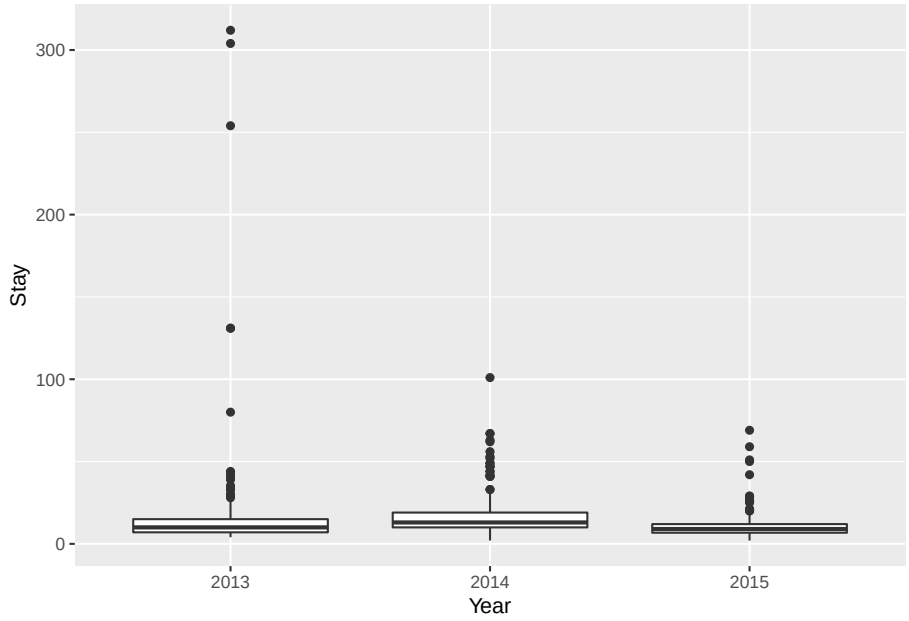
We are looking to see if the response variable within each group is normally distributed. To assess this, we need to think in terms of the moments of a normal distribution that we learned about earlier in the course, specifically skew and kurtosis. Here we are looking for outliers in the data, or sample distributions that are highly skewed.

First, we could go level by level for all of our grouping variables and conduct Shapiro tests (not shown here).

We can look at a few different plots of our response to start teasing apart some of the potential violations of our assumptions.

We know we will need to look at a year effect here because that is yet another form of non-independence (and potentially homogeneity) in our data. Let's start with a boxplot:

```
ggplot(turtles,
       aes(x = factor(Year), y = Stay, group = Year), fill = 'gray87') +
  geom_boxplot() +
  xlab("Year")
```



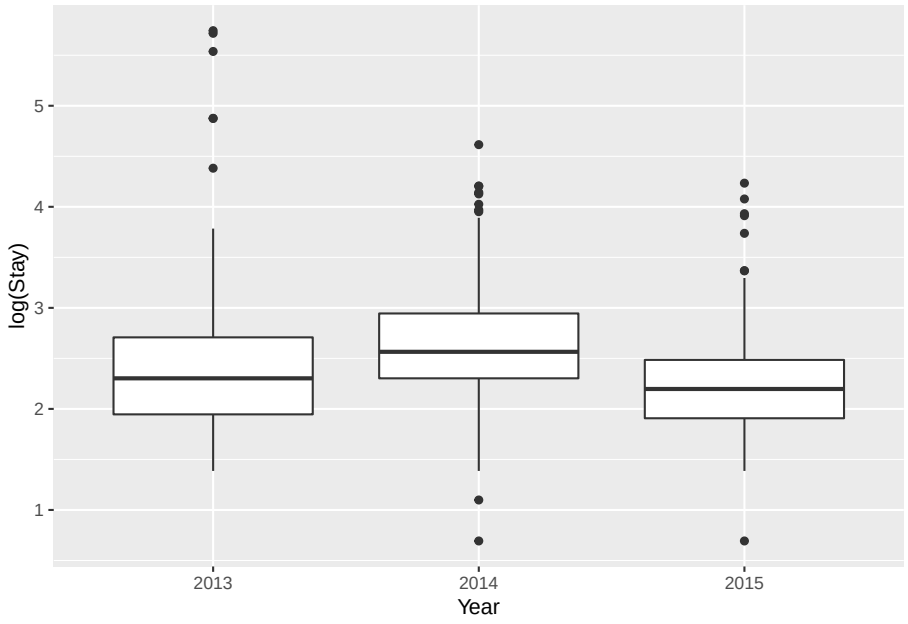
Whoa! We have a couple of issues here.

First of all: we have clearly identified a number of 'outliers' in our data. These are the circles that are outside the whiskers of our box plots.

One way to address these outliers is by dropping them from the data. We only want to do this if we have a pretty good justification for this ahead of time ("*a priori*"). And, sometimes these can be some of the most interesting observations.

Another way to deal with this is through data transformation. For example, we could use a log transformation in an attempt to normalize extreme values in our data. This certainly looks a little better, but may not get us all the way there...

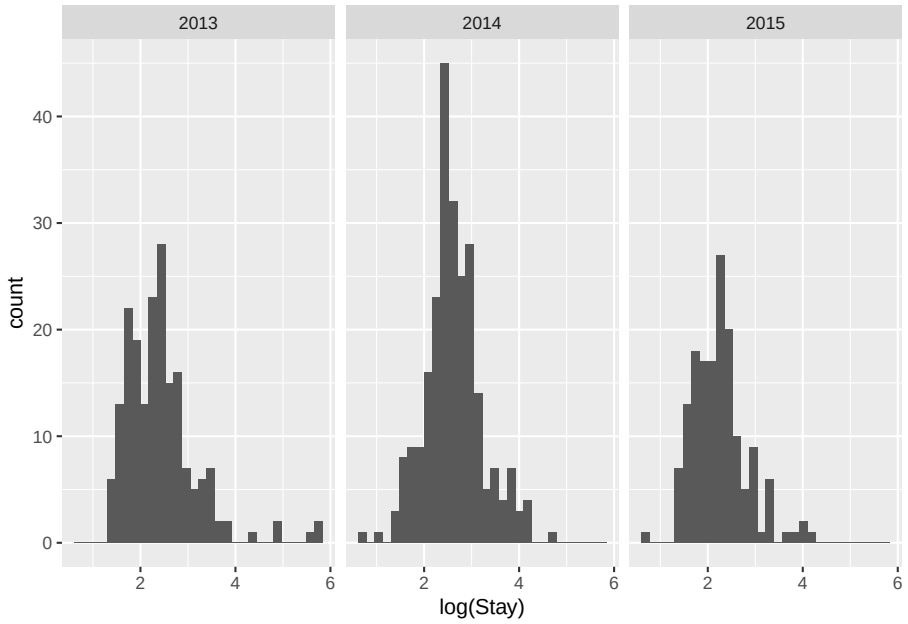
```
ggplot(turtles,
       aes(x = factor(Year), y = log(Stay), group = Year),
       fill = 'gray87') +
  geom_boxplot() +
  xlab("Year")
```



NOTE: I will not cover variable transformation extensively in this class or text. The justification is: 1) you can Google it to learn more about what transformations are useful for what, and 2) I will argue that most of the time there are better methods for dealing with non-normal data and then I will show you how to use those methods as we go.

We can also look at histograms to investigate normality within groups. We'll continue using log(Stay) for now.

```
ggplot(turtles, aes(x = log(Stay))) +
  geom_histogram() +
  facet_wrap(~ Year)
```



Again, a little better, but perhaps not as good as we'd like.

9.5.3 Homogeneity of variances

Finally, we assume in all of our linear models that variability in our residuals (which are really just part of our variances) are constant among groups or across the range of continuous variables x. This is the same assumption that we made for t-tests and tested with the F-test in Chapter 6. We'll now look at a few options for linear models in this chapter depending on how the data are structured.

A quick check of variances in the `Stay` variable by `Year` will make it clear that we are also in violation of this assumption if we do not log-transform the data.

```
turtles %>%
  group_by(Year) %>%
  summarize(var(Stay))
```

```
## # A tibble: 3 x 2
##    Year `var(Stay)`
##   <int>       <dbl>
## 1  2013      1435.
## 2  2014       164.
## 3  2015        91.7
```

You can go ahead and conduct a few F-tests if you don't believe me that these are different, but I'm pretty sure you won't convince me that the ratios of any two of these variances are equal to 1.00!

This, too, is made magically (A little) better when we log-transform `Stay`:

```
turtles %>%
  group_by(Year) %>%
  summarize(var(log(Stay)))
```

```
## # A tibble: 3 x 2
##    Year `var(log(Stay))`
##   <int>            <dbl>
## 1  2013            0.556
## 2  2014            0.360
## 3  2015            0.321
```

9.6 ANOVA Diagnostics

The preferred method for examining the normality of residuals for us is going to be actually looking at the diagnostics from a fitted model object regardless of the models we choose. This same approach can be applied to t-tests, ANOVA, linear regression, and ANCOVA. We'll start with ANOVA and wrap up with linear regression.

Here, we will conduct an ANOVA to test the null hypothesis that there is no difference in `Stay` between years (`Year`) assuming a Type-I error rate (α) of 0.05.

We are going to need to change `Year` to a factor for this analysis.

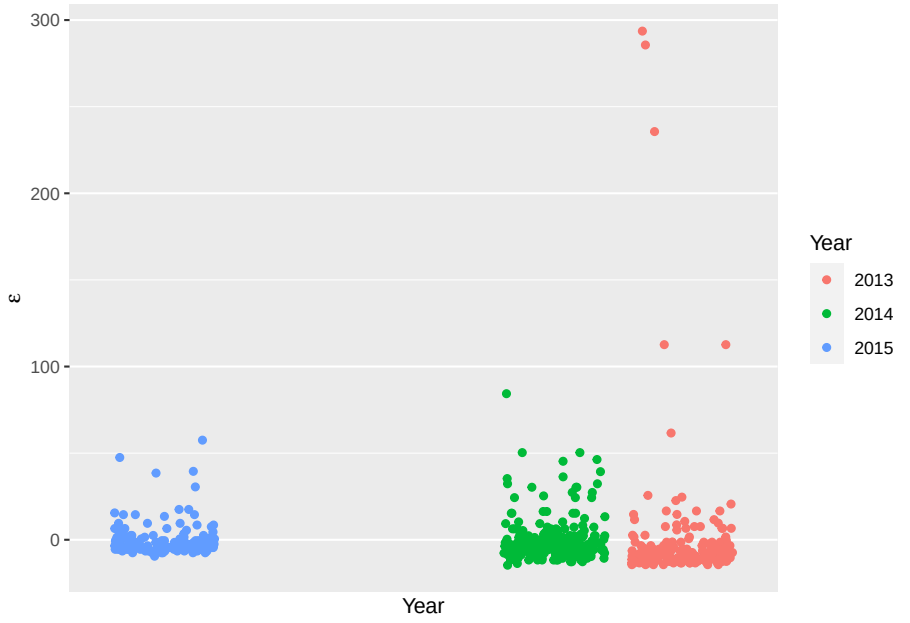
```
turtles$Year <- factor(turtles$Year, levels = c(2013, 2014, 2015))
```

Fit the model.

```
# First fit a model - the easy part
turdel <- lm( Stay ~ Year, data = turtles)
```

The `ggplot()` function knows just what to do with `lm()` objects!

```
ggplot(turdel, aes(x = .fitted, y = .resid, color=Year)) +
  geom_jitter() +
  scale_x_discrete() +
  xlab("Year") +
  ylab(expression(paste(epsilon)))
```



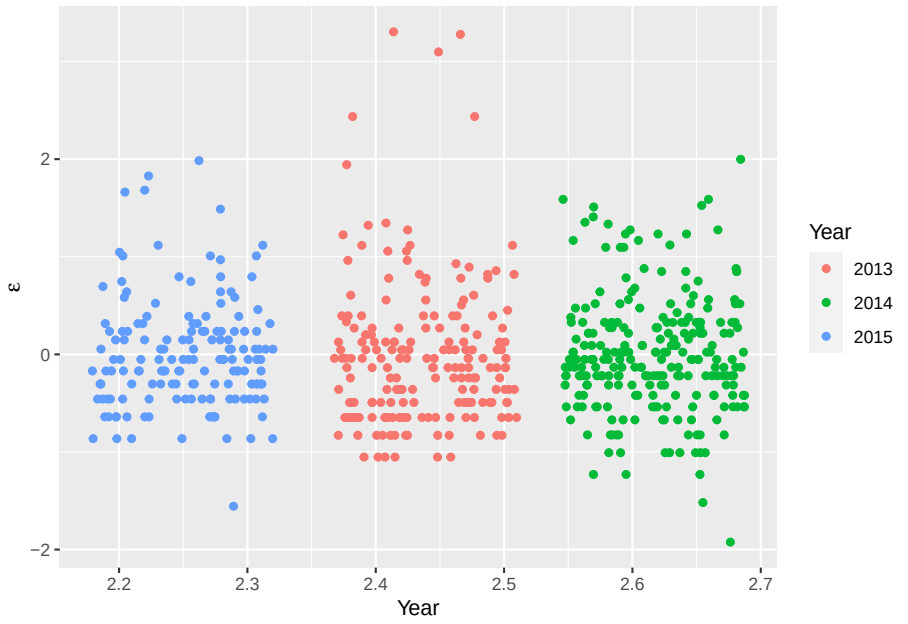
Cool! But…what the heck are we looking at here??

We have pretty much everything we need here to understand whether we have violated the assumptions of linear models from this graph (other design issues notwithstanding).

Remember, the mean of the residuals in each group is supposed to be **normally distributed with a mean of zero** and the **variance is equal between groups**. Now, I don't think we need a K-S test or an F-test to say that 1) these resids is def not normal and 2) no way the variances are equal. You can also see that `ggplot()` has nicely organized our groups in order of increasing magnitude of the residuals from left to right, and that `2013` and `2014` were more variable than `2015`.

We can hit the data with a log transformation to see if it fixes any of our problems:

```
# First fit a model
log_turdel <- lm( log(Stay) ~ Year, data = turtles)

# Now plot the residuals
ggplot(log_turdel, aes(x = .fitted, y = .resid, color=factor(Year))) +
  geom_jitter() +
  scale_color_discrete(name = "Year") +
  xlab("Year") +
  ylab(expression(paste(epsilon)))
```

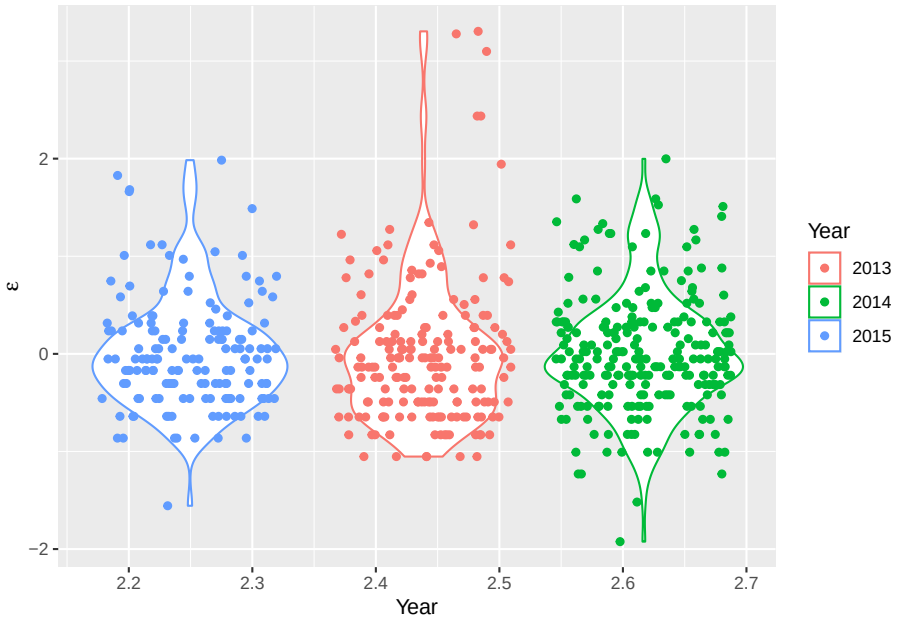


In fact, we see that the model fit has improved substantially, although the outliers in our data are still outliers and there is still some skew in the residuals. But, at least now all three years have residuals with a mean near zero and they're more symmetrical than they were before. You can also see that the groups are now placed more uniformly along the x axis.

If we wanted to investigate further the extent of remaining issues, we could visualize this a little better using a violin plot of the residuals:

```
# Now plot the residuals
ggplot(log_turdel, aes(x = .fitted, y = .resid, color=factor(Year))) +
  geom_violin() +
  geom_jitter() +
  scale_color_discrete(name = "Year") +
  xlab("Year") +
```

```
  ylab(expression(paste(epsilon)))
```



This plot now shows pretty clearly that the variance in the residuals increases from 2013 to 2015, and that there are a lot more outliers in 2013. But, there aren't a ton of outliers, so maybe this is something we could accomodate with the right assumptions about our sampling distribution down the road.

Notice that we still haven't looked at the model results yet? That's not just because this is The Worst Stats Text eveR.

9.7 Linear regression diagnostics

Finally, what if we have a continuous explanatory variable and a continuous response that necessitates use of linear regression or ANCOVA?

We use the same approach (whoa, that's sweet, huh?). Here, let's fit and assess a model that predicts the length of `Stay` in turtle rehab based on the hook `Width` that caught them in the first place.

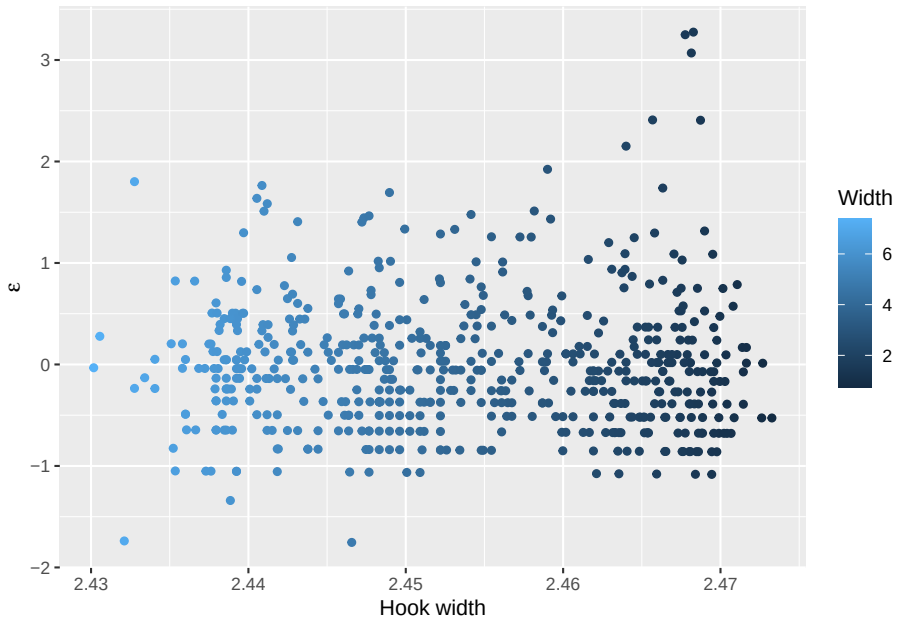
```
# Fit the model
fit_width <- lm( Stay ~ Width, data = turtles)

# Now plot the residuals
ggplot(fit_width, aes(x = .fitted, y = .resid, color = Width)) +
```

```
  geom_point() +
  xlab("Hook width") +
  ylab(expression(paste(epsilon)))
```

Bleh. As with our example above, we can see that we clearly fail the assumption that the residuals are normally distributed with a mean of zero! It also looks like the residual error increases with increasing hook width, which means our observations are also not independent. This specific type of non-independence is called *heteroscedasticity*. That's a real word.

Let's see if our new toy, the log-transformation can help us here:

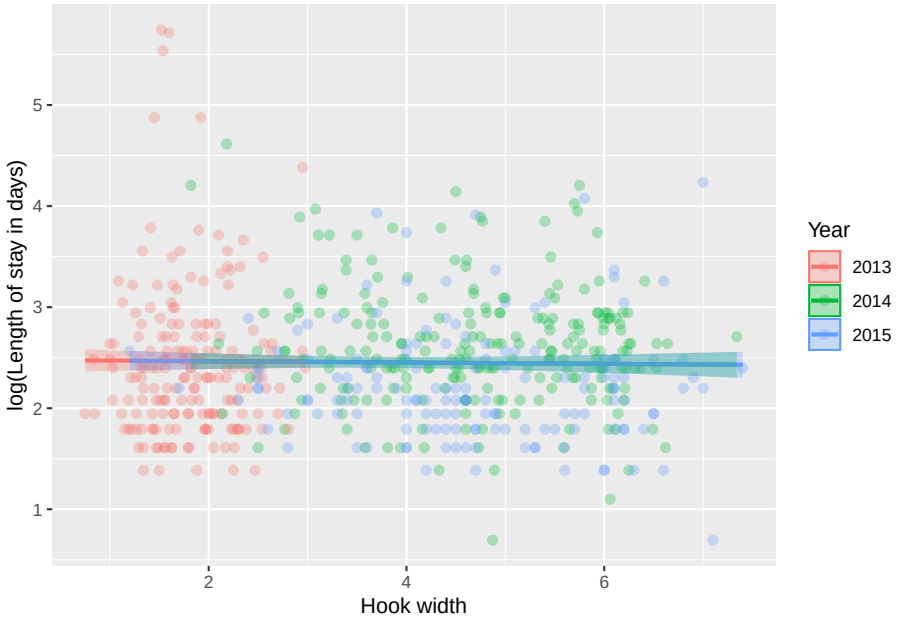
```
# Fit the model
log_fit_width <- lm( log(Stay) ~ Width, data = turtles)

# Now plot the residuals
ggplot(log_fit_width, aes(x = .fitted, y = .resid, color = Width)) +
  geom_point() +
  xlab("Hook width") +
  ylab(expression(paste(epsilon)))
```



Wow, that actually looks a whole lot better! There are still a couple of data points flying high that we would want to investigate further in this data set, but the pukey feeling in my stomach is slowly subsiding here.

And remember, if you *really* want to see how your model fits the data, you could always plot the predictions over the raw data:

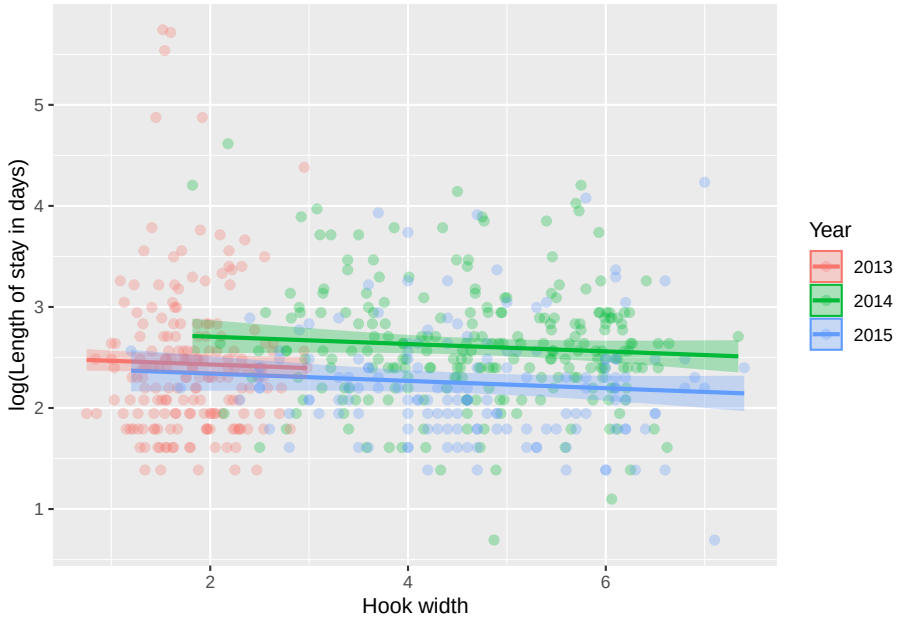
```
# Need to get rid of a couple NA values
turts <- subset(turtles, !is.na(Width))

log_fit_width <- lm( log(Stay) ~ Width, data = turts)
turt_pred <- cbind(turts, predict(log_fit_width, interval = 'confidence'))

ggplot(turt_pred, aes(x = Width, y = log(Stay), color = Year, fill = Year)) +
  geom_point(alpha = 0.3, size = 2) +
  geom_line(aes(y = fit), size = 1) +
  geom_ribbon(aes(ymin = lwr, ymax = upr, color = NULL), alpha = .3) +
  xlab("Hook width") +
  ylab("log(Length of stay in days)")
```



Surely this plot alone is evidence enough that we need to investigate confounding between hook `Width` and `Year` in any further investigation into this data set! Maybe with an ANCOVA?

```
# Need to get rid of a couple NA values
turts <- subset(turtles, !is.na(Width))

log_fit_width <- lm( log(Stay) ~ Year + Width, data = turts)
turt_pred <- cbind(turts, predict(log_fit_width, interval = 'confidence'))
```

```
ggplot(turt_pred,
       aes(x = Width, y = log(Stay), color = Year, fill = Year)) +
  geom_point(alpha = 0.3, size = 2) +
  geom_line(aes(y = fit), size = 1) +
  geom_ribbon(aes(ymin = lwr, ymax = upr, color = NULL), alpha = .3) +
  xlab("Hook width") +
  ylab("log(Length of stay in days)")
```



Holy crap…**three lines**. Where did the other two come from? Keep reading to find out in Chapter 10.

Notice that we still have not looked at the results of any of these models.

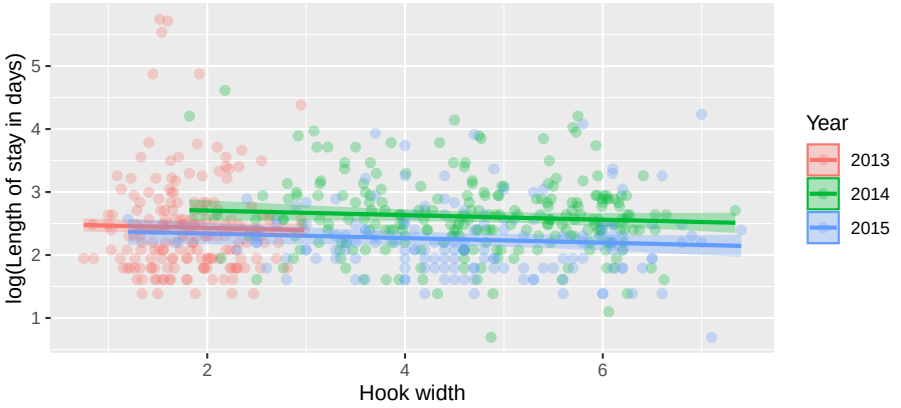
9.8 Next steps

This is a lot to take in. But this stuff is important for everything we do in statistics. Do this stuff **before** you start fitting all kinds of models because it is important to think about ahead of time! Examination of residual plots should become second nature to you in these analyses because it is the most powerful tool you have for testing assumptions. Don't freak out if things don't look perfect (they almost never will), and realize that there may be ways of dealing with violations within the context of linear models. If not, there certainly are other models designed specifically for this purpose.

In Chapter 10 we will continue to unpack the linear model as we talk about how to communicate the results of the models after we fit them and validate assumptions.

Chapter 10

Communicating effect sizes



This is the graph showing how the amount of time spent in rehab by sea turtles changed with fishing hook size during three years. We made this in Chapter 3, but I never told you if the effect of hook width was significant because this is The Worst Stats Text eveR. It was not. But then again, you probably could have figured that out from the graph. That's why it's important to show your predictions.

Here's the bad news: add/drop is over and we're about to do some math. Here's the good news: the math will stay the same from now until the end of this book because it is the beautiful, unifying math behind most of the tools we've discussed so far and all the tools to come. Well, I don't know if that's actually good news, but it does sound nice when I say it like that.

Now that we have a handle on interpreting the statistical results of linear models we need to think about how to communicate biological differences (effects) and the uncertainty associated with our predictions. This is a major short coming

of many scientific studies, and has led to wide-spread reporting of statistically significant results that confer minimal biological meaning. On the other hand, if we do have really cool biological results, we want to be able to show those to people! A well designed graphic will tell most of your readers more than a parentheses-packed, numerically dense Results section - I don't care how well you write.

How we approach communication of our results can range from summarizing and graphing raw data to plotting futuristic curves over raw data depending on the type of effect we are trying to communicate. That depends, of course, on the model that we fit, the data that we collected, and how they were collected. To do this well, we have to at least understand how R is using our data, and that requires at least a superficial understanding of the actual math we are doing. Sorry. We'll take it one step at a time and work through ANOVA (so hard), linear regressions (super easy), and ANCOVA (not that hard once you "get it").

For this chapter, we will revisit some of the built-in data sets we've been using for hypothesis testing and linear models and introduce the `dental` data set. We'll also be working with a few packages in the `tidyverse` so you can go ahead and load that now if you want to.

```
library(tidyverse)
```

10.1 One-way ANOVA

When we are working in the world of one-way ANOVA, or even more complex models that contain only "main effects" of categorical, explanatory variables, the interpretation of these effects is relatively straightforward. Let's use the `PlantGrowth` data as an example.

```
data("PlantGrowth")
```

We'll start here by fitting a one-way anova to test effects of treatment `group` on on plant `weight`.

```
# Get the names of the df
names(PlantGrowth)
```

```
## [1] "weight" "group"
```

```
# Fit the model
mod <- lm(weight~group, data = PlantGrowth)
```

We've seen these data and this model before. We know there was a significant effect of treatment on plant weight but only `trt1` and `trt2` were different when we checked with the Tukey test. So, for now we will ignore the ANOVA table and just look at the summary.

```
summary(mod)
```

```
##
## Call:
## lm(formula = weight ~ group, data = PlantGrowth)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -1.0710 -0.4180 -0.0060  0.2627  1.3690
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)   5.0320     0.1971  25.527   <2e-16 ***
## grouptrt1    -0.3710     0.2788  -1.331   0.1944
## grouptrt2     0.4940     0.2788   1.772   0.0877 .
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.6234 on 27 degrees of freedom
## Multiple R-squared:  0.2641,Adjusted R-squared:  0.2096
## F-statistic: 4.846 on 2 and 27 DF,  p-value: 0.01591
```

This summary gives us three coefficients corresponding to the coefficients of a linear model. Up until now, we've mostly ignored these for ANOVA and focused on hypothesis testing. But we need to use the coefficients to make predictions from our model and communicate biological results - which is why there is a history of people not doing this effectively.

If we wanted to write out that linear model, we could write it like this:

$y = \beta_0 + \beta_1 X_1 + \beta_2 X_2$

But this is confusing because we only gave R one variable for `X`! How did it get three? Plus, we've been thinking of ANOVA like t-tests. Much puzzlement.

10.1.1 Unifying the linear model

To really get full control over making predictions from linear models and the models to come we need to understand a little bit more about what R is doing here. I mentioned in Chapter 9.3 that we would need to start thinking about

the linear model as $y = \beta X + \epsilon$ or $\sum_{k=1}^{K} \beta_k X_k + \epsilon$ to unify the t-test, ANOVA, linear regression, and ANCOVA into a single general framework. The reason for this is that R (and the math) actually don't see `X` in our linear models the way we've been writing it in our code. The models we've talked about so far are solved through Least Squares estimation. This involves solving for however many β we might have using linear algebra and a little calculus to minimize the sum of ϵ^2, or our squared residuals. To do the math, `X` must be a matrix of values that can be multiplied by a vector coefficients (β) because as we now know, $y = \beta X + \epsilon$.

So, how does this relate to β_0 and the fact that we supposedly have three `X` variables in the `PlantGrowth` ANOVA even though it is just one column?

I've already told students in my class by this point in the semester, but I'll repeat here that β_0 has a special place in my heart. It is the thing that allows us to relate all of this crap back to $y = mx + b$ and makes me feel like I understand statistics a little bit. But, it is also the hard part behind understanding the predictions you make from linear models if you don't know or like (love) the algebra. Especially for ANVOA and ANCOVA-like models. And let's face it, most of us as biologists don't understand let alone love the algebra. We'll try to keep avoiding that here as long as we can.

Up until now, we have thought of β_0 as our `(Intercept)` term in linear models, and that is both truthful and useful. But, it is just another β in the matrix multiplication used to solve least-squares regression.

How, then, is the intercept represented mathematically in ANOVA?

10.1.2 The model matrix

In order to understand how the intercept works in ANOVA, we must look at the model matrix.

The **model matrix** or **design matrix** is `X` from the really gross equations that I started showing all of a sudden now that the Add/Drop period has ended. (Muahaha). It really isn't as sinister as it sounds though.

For our plant model, we wrote `weight ~ group` in our call to `lm()` and didn't have to think twice about what was happening after that. In the meantime, R had re-written the equation as $y = \beta_i X_i$ or `y = (Intercept)*model.matrix$`(Intercept)`+ grouptrt1*model.matix$grouptrt1 + grouptrt2*model.matix$grouptrt2`. To begin understanding that difference, we obviously need to see this `model.matrix` object.

First, look at the actual data used to fit the model:

```
head(mod$model)
```

```
##   weight group
## 1   4.17  ctrl
## 2   5.58  ctrl
## 3   5.18  ctrl
## 4   6.11  ctrl
## 5   4.50  ctrl
## 6   4.61  ctrl
```

You can see that we have one column for our response, `weight`, and one column for our explanatory variable, `group`, just as you thought.

Now here is the design matrix:

```
# Extract design matrix from fitted
# PlantGrowth model
X <- model.matrix(mod)

# Have a look
head(X)
```

```
##   (Intercept) grouptrt1 grouptrt2
## 1           1         0         0
## 2           1         0         0
## 3           1         0         0
## 4           1         0         0
## 5           1         0         0
## 6           1         0         0
```

Okay, that's actually not so bad. So, this is how R sees our data now. What R has done is to **dummy code** our `group` variable from the `PlantGrowth` data for each row of the data. The first column, `(Intercept)` contains only `1`. You can think of this as `X_0` in our linear model. It is multiplied by β_0 in $y = \beta_0 + \beta_k X_k$. But, since it is always `1` we just don't write it when we write the formula for a line and β_0 is always in the model! OMG that is sooooo annoying. The second column is an indicator variable for whether `group` is equal to `trt1` for a given observation (row in `PlantGrowth`). If `group == trt1` for that row, then the column `grouptrt1` gets a `1`. If not, it gets a `0`. Same for `grouptrt2`. The columns `grouptrt1` and `grouptrt2` are each multiplied by their own β in our formula:

$$y = \beta_{(Intercept)} X_{(Intercept)} + \beta_{grouptrt1} X_{grouptrt1} + \beta_{grouptrt1} X_{grouptrt1}$$

If the columns `grouptrt1` or `grouptrt2` have `0`, then $\beta_i X_i = 0$ and the term for that group falls out of the equation, leaving only `ctrl` or the `(Intercept)`. We can use this to make predictions directly from our model coefficients.

Before moving on to prediction, it is helpful if you think of the coefficients for ANOVA as being an intercept (mean of the alphabetically first group) and

offsets, or adjustments, to that intercept for each subsequent group. That is, ANOVA is kind of like a linear model with multiple intercepts and no slopes. We are just estimating a bunch of points on the y-axis.

10.1.3 Prediction

Now that we've seen what R is actually doing, it becomes pretty trivial to make predictions from one-way ANOVA by hand.

We can get the model coefficients (β) like this:

```
# We can get the model coefficients like this:
names(mod)
```

```
##  [1] "coefficients"  "residuals"     "effects"       "rank"
##  [5] "fitted.values" "assign"        "qr"            "df.residual"
##  [9] "contrasts"     "xlevels"       "call"          "terms"
## [13] "model"
```

```
coeffs <- data.frame(mod$coefficients)

# And now we have a vector of beta
betas <- coeffs$mod.coefficients
```

We can use `betas` to make predictions from the formula of our linear model for each group by taking advantage of the dummy coding that R uses.

```
# From the model, we can estimate:

# Mean of control
y_control <- betas[1] + betas[2]*0 + betas[3]*0

# Mean of trt1
y_trt1 <- betas[1] + betas[2]*1 + betas[3]*0

# Mean of trt2
y_trt2 <- betas[1] + betas[2]*0 + betas[3]*1
```

Or if you wanted to get really fancy, you could do this with matrix math:

```
# Get unique groups in dummy coded matrix
X_pred <- as.matrix(unique(model.matrix(mod)))

# Multiply betas by dummy coded
```

```
# matrix using transpose of both
# These are your predictions
# for ctrl, trt1, and trt2
y_pred <- t(betas) %*% t(X_pred)
```

Of course, either of these approaches is super useful but R also has default `predict()` methods for most or all of the models we will work with in this book. We will use these for the most part, as will `ggplot()`, which is more convenient than you will ever be able to appreciate.

To make predictions of `y` from the original data that you used to fit the model (`mod`), you can just do this:

```
# Get unique values of groups and put it in
# a data frame. The predict function expects
# original x variable as a vector or a named
# column in a data.frame
groups <- data.frame(group = unique(PlantGrowth$group) )

# Make the prediction
y <- predict(mod, newdata = groups, interval = "confidence")

# Add it to the data frame
pred_plant <- data.frame(groups, y)
```

If we want `confidence` intervals for the predictions, we can add that, too:

```
# Make the prediction with confidence
yCI <- predict(mod, newdata = groups, interval = "confidence")

# Add it to the data frame
pred_plantCI <- data.frame(groups, yCI)
```

You could print this and get a nice clean table of estimated means and 95% confidence intervals for each group.

```
print(pred_plantCI)
```

```
##   group   fit      lwr      upr
## 1  ctrl 5.032 4.627526 5.436474
## 2  trt1 4.661 4.256526 5.065474
## 3  trt2 5.526 5.121526 5.930474
```

Now, let's compare our model predictions to the actual means.

```
# Calculate group means
means <- PlantGrowth %>%
  group_by(group) %>%
  summarize(mean(weight))

print(means)
```

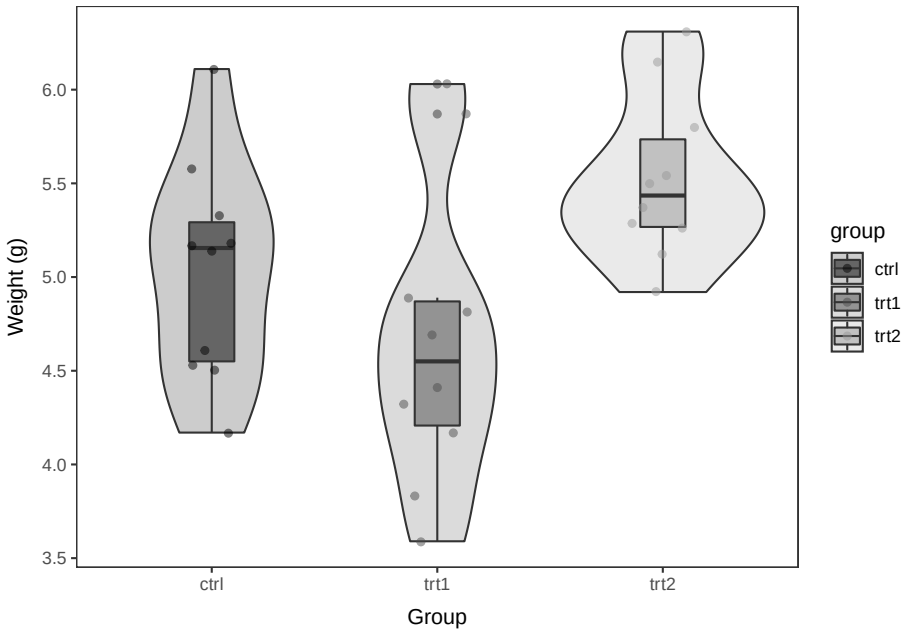
```
## # A tibble: 3 x 2
##   group `mean(weight)`
##   <fct>          <dbl>
## 1 ctrl            5.03
## 2 trt1            4.66
## 3 trt2            5.53
```

Pretty much spot on!

10.1.4 Plotting

We could use any number of graphical tools to represent these results. Given that we've met the assumptions of normality, and we've determined that statistical differences exist, the simplest (and most common) method for visualizing these results is to just show a box plot or a violin plot, or both, with the raw data. Hmmm…I never realized how small this data set was.

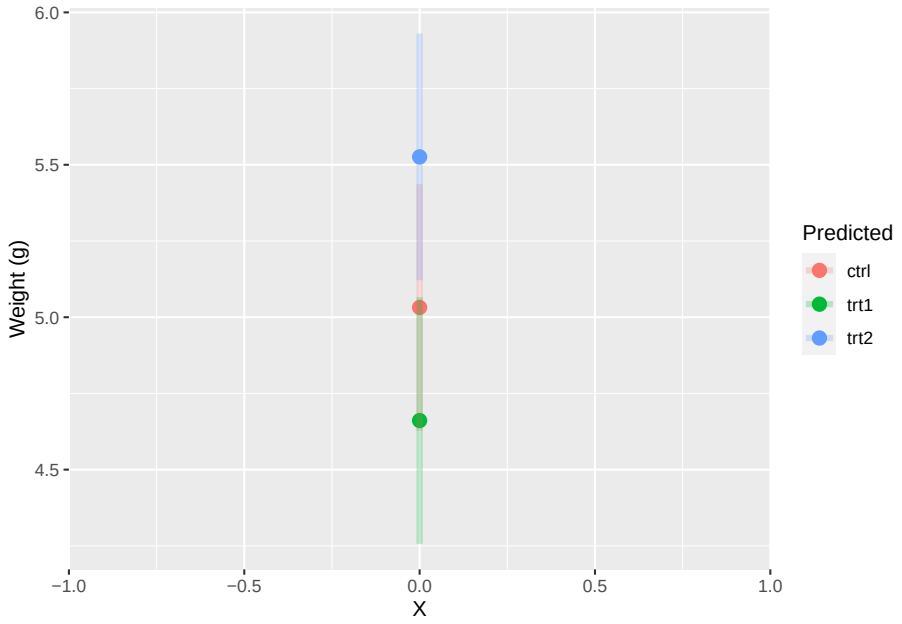
```
ggplot(PlantGrowth, aes(x = group, y = weight)) +
  geom_violin(aes(fill=group), alpha=0.2) +
  geom_boxplot(aes(fill=group), width = 0.2, alpha = 0.5) +
  geom_jitter(aes(color=group), width = 0.15, alpha=0.5) +
  scale_fill_manual(values=c('black', 'gray30', 'gray60')) +
  scale_color_manual(values=c('black', 'gray30', 'gray60')) +
  xlab('Group') +
  ylab('Weight (g)') +
  theme_bw() +
  theme(axis.title.x = element_text(vjust = -1),
        axis.title.y = element_text(vjust = 3),
        panel.grid = element_blank()
        )
```



This plot is really cool, but it doesn't actually show us how our model predictions compare to the raw data!

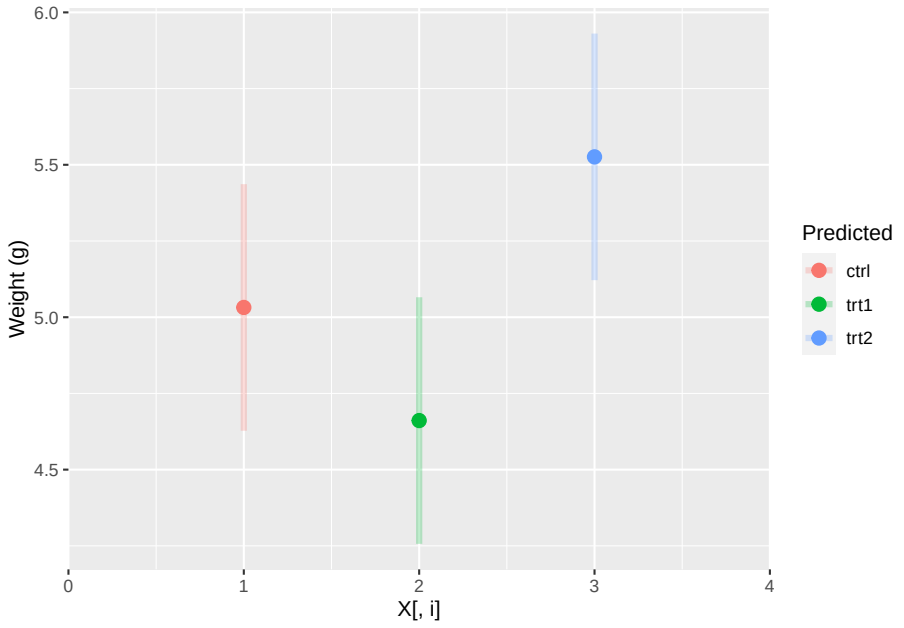
However, we could also think of our model predictions as just being different y-intercepts, which will be helpful when we start to work with ANCOVA. If we plotted them that way, they would look like this:

```
ggplot(pred_plantCI, aes(x = 0, y = fit, color = group)) +
  geom_point(size = 3) +
  scale_x_continuous(limits = c(-1,  1), expand=c(0,0)) +
  geom_segment(aes(x = 0, xend = 0, y = lwr, yend = upr),
               lwd = 1.5, alpha = 0.25) +
  xlab("X") +
  ylab("Weight (g)") +
  labs(color = "Predicted")
```



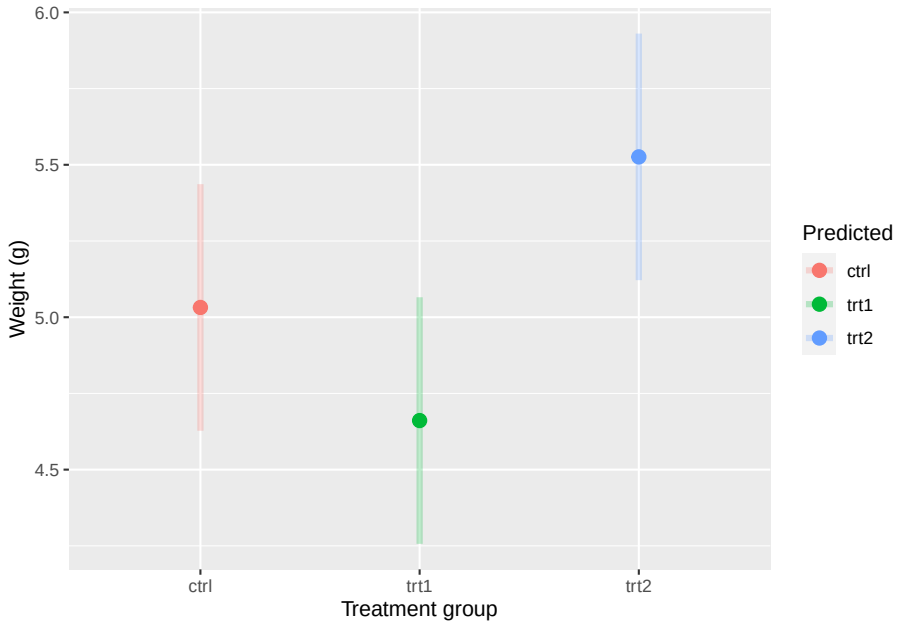
But this is really hard to see and understand. So, we usually look at it like this in keeping with the dummy coding that is used in the model matrix:

```
ggplot(pred_plantCI, aes(x = 1:3, y = fit, color = group)) +
  geom_point(size = 3) +
  scale_x_continuous(limits = c(0,  4), expand=c(0, 0)) +
  geom_segment(aes(x = 1:3, xend = 1:3, y = lwr, yend = upr),
               lwd = 1.5, alpha = 0.25) +
  xlab("X[, i]") +
  ylab("Weight (g)") +
  labs(color = "Predicted")
```



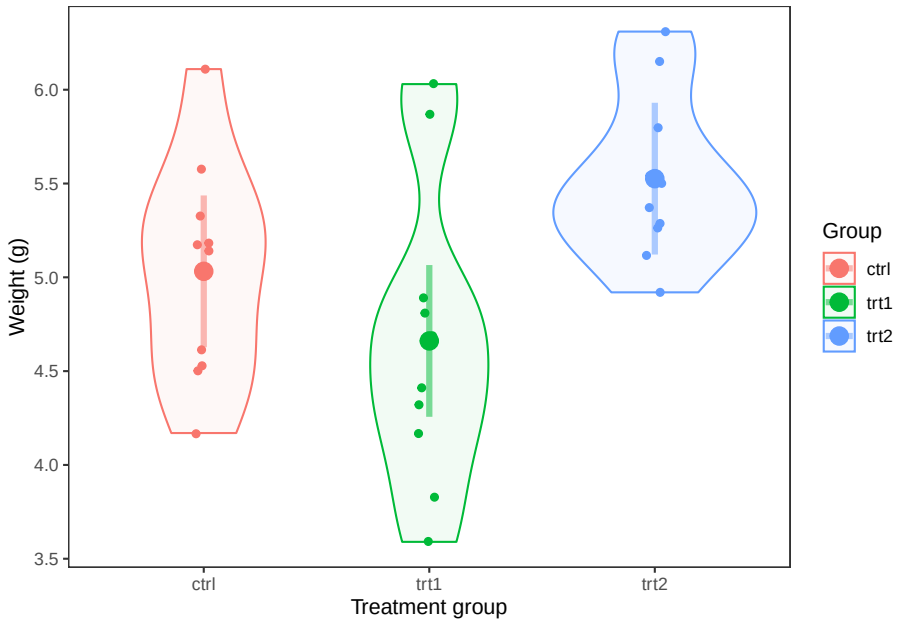
Or, perhaps more mercifully:

```
ggplot(pred_plantCI, aes(x = group, y = fit, color = group)) +
  geom_point(size = 3) +
  geom_segment(aes(x = group, xend = group, y = lwr, yend = upr),
               lwd = 1.5, alpha = 0.25) +
  xlab("Treatment group") +
  ylab("Weight (g)") +
  labs(color = "Predicted")
```



Finally, we could put this right over the top of our raw data and/or violin to see how well the model predictions match up with the data:

```
ggplot(PlantGrowth, aes(x = group, y = weight, color = group)) +
  geom_violin(aes(fill = group), alpha = 0.05) +
  geom_jitter(size = 1.5, width = 0.05) +
  geom_point(mapping = aes(x = group, y = fit),
             data = pred_plantCI,
             size = 4) +
  geom_segment(aes(x = group, xend = group, y = lwr, yend = upr),
               data = pred_plantCI,
               lwd = 1.5,
               alpha = 0.5) +
  theme_bw() +
  theme(panel.grid = element_blank()) +
  xlab("Treatment group") +
  ylab("Weight (g)") +
  labs(fill = "Group", color = "Group")
```



10.2 Two-way ANOVA

Two-way ANOVA works the same way as one-way ANOVA, except that now we have multiple dummy-coded variables tied up in the intercept. For this example, we will consider a new data set. These data are from an experiment in Restorative Dentistry and Endodontics that was published in 2014. The study examines effects of drying light and resin type on the strength of a bonding resin for teeth.

The full citation for the paper is:

Kim, H-Y. 2014. Statistical notes for clinical researchers: Two-way analysis of variance (ANOVA)-exploring possible interaction between factors. Restorative Dentistry and Endodontics 39(2):143-147.

Here are the data:

```
dental <- read.csv('data/dental.csv', stringsAsFactors = FALSE)
```

10.2.1 Main effects model

We will start by fitting a linear model to the data that tests effects of `lights` and `resin` on adhesive strength `mpa`. Since both `lights` and `resin` are categorical, this is a two-way ANOVA. We use the `+` to imply additive, or main-effects only.

```
# We are looking only at main effects for now
dental.mod <- lm(mpa ~ lights + resin, data = dental)
```

If we make an ANOVA table for this two-way ANOVA, we see that there are significant main effects of resin type but not lights used for drying.

```
anova(dental.mod)
```

```
## Analysis of Variance Table
##
## Response: mpa
##           Df Sum Sq Mean Sq F value    Pr(>F)
## lights     1   34.7   34.72  0.6797    0.4123
## resin      3 1999.7  666.57 13.0514 6.036e-07 ***
## Residuals 75 3830.5   51.07
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

We can also examine the model coefficients for a closer look at what this means.

```
summary(dental.mod)
```

```
##
## Call:
## lm(formula = mpa ~ lights + resin, data = dental)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -14.1162  -4.9531   0.1188   4.4613  14.4663
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)   19.074      1.787  10.676  < 2e-16 ***
## lightsLED     -1.318      1.598  -0.824  0.41229
## resinB         3.815      2.260   1.688  0.09555 .
## resinC         6.740      2.260   2.982  0.00386 **
## resinD        13.660      2.260   6.044 5.39e-08 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 7.147 on 75 degrees of freedom
## Multiple R-squared:  0.3469,Adjusted R-squared:  0.312
## F-statistic: 9.958 on 4 and 75 DF,  p-value: 1.616e-06
```

Remember, in our data we had 2 kinds of lights, and 4 kinds of resin. But, here we have one less of each! Why is this? It is because of the way categorical variables are dummy coded for linear models. But, now we have two separate sets of adjustments, so one level of each variable is wrapped up in the estimate for our intercept (`lightsHalogen` and `resinA`).

When in doubt, have a look at the model matrix:

```
X <- model.matrix(dental.mod)

head(X)
```

```
##   (Intercept) lightsLED resinB resinC resinD
## 1           1         0      0      0      0
## 2           1         0      0      0      0
## 3           1         0      0      0      0
## 4           1         0      0      0      0
## 5           1         0      0      0      0
## 6           1         0      0      0      0
```

Right now, you might be a little confused about how to calculate and show the effect size for these variables. If you are not, you should probably take a more advanced stats class and get a better book.

One reasonable option might be to summarize the data by the means and plot the means, but we already decided that we are going to plot our model predictions against the raw data following the form of the statistical model. Rather than go throught the math again, let's just use the built-in `predict()` function that I know you now appreciate!

First, we need to do a little magic to get the group combinations for `lights` and `resin` into a data.frame that we can use for prediction.

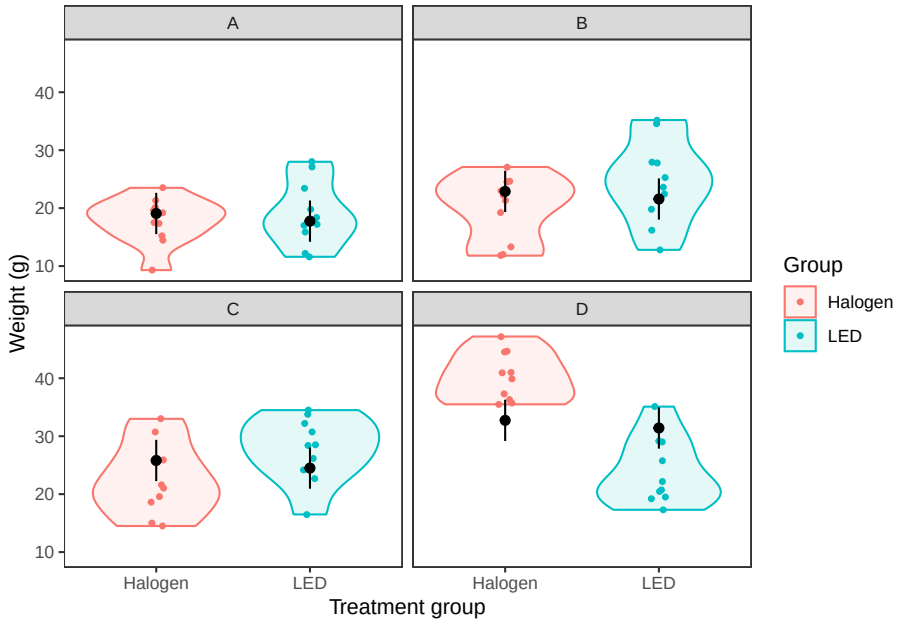
```
groups <- data.frame(
  with(dental, unique(data.frame(lights, resin)))
)
```

Now we can make our predictions:

```
dental_y_pred <- predict(
  dental.mod, newdata = groups, interval = "confidence"
  )

pred_dental <- data.frame(groups, dental_y_pred)
```

And now we can plot it just like we did for ANOVA:

```
ggplot(dental, aes(x = lights, y = mpa, color = lights)) +
  geom_violin(aes(fill=lights), alpha = 0.1) +
  geom_jitter(size = 1, width = 0.05) +
  geom_point(mapping = aes(x = lights, y = fit),
             data = pred_dental,
             color = 'black',
             size = 2) +
  geom_segment(
    aes(x = lights, xend = lights, y = lwr, yend = upr),
    data = pred_dental,
    color = 'black') +
  facet_wrap(~resin) +
  theme_bw() +
  theme(panel.grid = element_blank()) +
  xlab("Treatment group") +
  ylab("Weight (g)") +
  labs(fill = "Group", color = "Group")
```



Now, this looks really nice, but there is definitely something funky going on with `Halogen` in panel D in the figure above! We have clearly done a poor job of predicting this group. The reason for this, in this case, is because we need to include an **interaction** term in our model, which makes things even grosser in terms of the math, but is easy to do in R. Of course, if we had been doing a good job of data exploration and residual analysis, we would have noticed this before making predictive plots…

10.2.2 Interactions

To make a model that includes an interaction between `lights` and `resin` in the `dental` data, we will need to go all the way back to our model fitting process.

```
# The "*" operator is shorthand for what we want to do
# here - more of an "advanced" stats topic
dental_int <- lm(mpa ~ lights * resin, data = dental)
```

We just have three more columns in our model matrix to distinguish between coefficients for `resin` that correspond to `LED` and coefficients for `resin` that correspond to `Halogen`. It is at this point that not even I want to do the math by hand!

```
# Have a look on your own:
head(model.matrix(dental_int))
```

```
##   (Intercept) lightsLED resinB resinC resinD lightsLED:resinB lightsLED:resinC
## 1           1         0      0      0      0                0                0
## 2           1         0      0      0      0                0                0
## 3           1         0      0      0      0                0                0
## 4           1         0      0      0      0                0                0
## 5           1         0      0      0      0                0                0
## 6           1         0      0      0      0                0                0
##   lightsLED:resinD
## 1                0
## 2                0
## 3                0
## 4                0
## 5                0
## 6                0
```

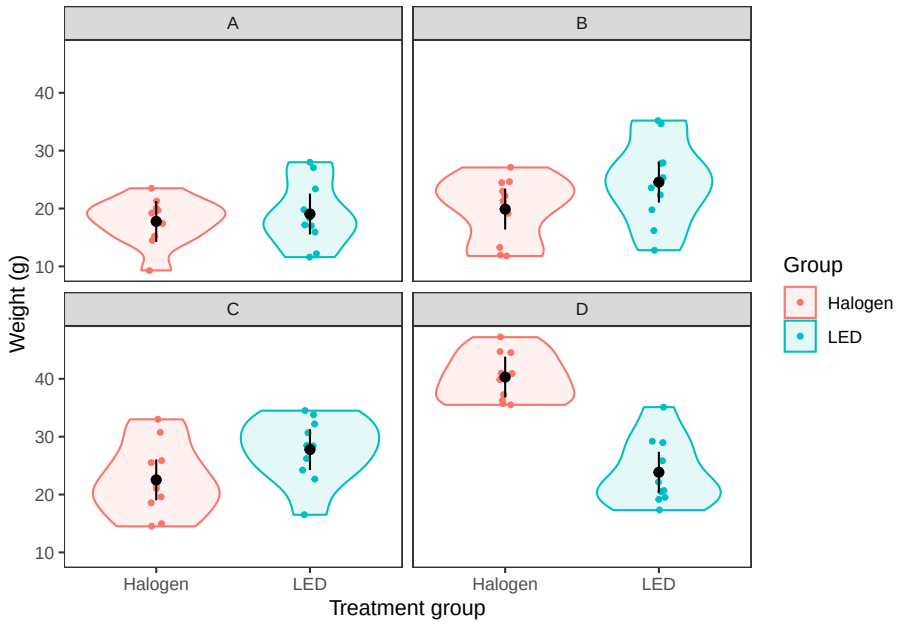
The process for making predictions, thankfully, is identical to two-way ANOVA in R.

Using the groups we made for the main-effects model:

```
int_y_pred <- predict(
  dental_int, newdata = groups, interval = "confidence"
  )

int_pred <- data.frame(groups, int_y_pred)
```

And now we plot the predictions against the raw data changing only the name of the data containing our predictions, `int_pred`.

```
ggplot(dental, aes(x = lights, y = mpa, color = lights)) +
  geom_violin(aes(fill=lights), alpha = 0.1) +
  geom_jitter(size = 1, width = 0.05) +
  geom_point(mapping = aes(x = lights, y = fit),
             data = int_pred,
             color = 'black',
             size = 2) +
  geom_segment(
    aes(x = lights, xend = lights, y = lwr, yend = upr),
    data = int_pred,
    color = 'black') +
  facet_wrap(~resin) +
  theme_bw() +
  theme(panel.grid = element_blank()) +
  xlab("Treatment group") +
  ylab("Weight (g)") +
  labs(fill = "Group", color = "Group")
```



You should see below that **all** of our means match up much better with the observed data in the violins. And, if you go back to the ANOVA output for this model you will see that the interaction term is significant even though `lights` still is not on it's own. In coming chapters, we'll talk about how to design and compare multiple models like these to compare meaningful biological hypotheses against one another.

```
anova(dental_int)
```

```
## Analysis of Variance Table
##
## Response: mpa
##              Df  Sum Sq Mean Sq F value    Pr(>F)
## lights        1   34.72   34.72  1.1067    0.2963
## resin         3 1999.72  666.57 21.2499 5.792e-10 ***
## lights:resin  3 1571.96  523.99 16.7043 2.457e-08 ***
## Residuals    72 2258.52   31.37
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Interactions between categorical variables generally are more complicated to deal with than interactions between categorical and continuous variables, because then we are only dealing with straight lines that differ by level. This does not, however, make it any less important for us to communicate how our models fit the observations we have collected. If you can get these tools under your belt, they will be extremely powerful for preparing journal articles, and perhaps more importantly, for communicating your results and the uncertainty surrounding them to stakeholders and public audiences.

10.3 Linear regression

Compared to interpreting group effects from ANOVA, the interpretation of a single, continuous predictor in linear regression is pretty straightforward. Here, all we are doing is looking to use the equation for a line $y = mx + b$ to predict the effects of one continuous variable on another. Most of us probably did this for the first time in middle school. But, if we look at the math in the same way that we did for ANOVA, it will make understanding ANCOVA a lot easier.

Let's use the `swiss` data again for this like we did in Chapter 8. Remember that this data set compares fertility rates to a number of socio-economic indicators:

```
data("swiss")
```

We'll make a model to predict the effect of education level on fertility:

```
# Make the model and save it to a named object called 'swiss.mod'
swiss.mod <- lm(Fertility ~ Education, data = swiss)
```

Have a look at the design matrix and you can see that R still includes a column for the intercept that is all `1`, so this is the same as ANOVA. But, instead

of having dummy variables in columns representing groups, we just have our observed values of `Education`

```
head( model.matrix(swiss.mod) )
```

```
##              (Intercept) Education
## Courtelary             1        12
## Delemont               1         9
## Franches-Mnt           1         5
## Moutier                1         7
## Neuveville             1        15
## Porrentruy             1         7
```

Next, we can look at the coefficient estimates for `swiss.mod`. Remember that each of these coefficients corresponds to one and only one column in our design matrix `X`.

```
# Summarize the model
summary(swiss.mod)
```

```
##
## Call:
## lm(formula = Fertility ~ Education, data = swiss)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -17.036  -6.711  -1.011   9.526  19.689
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  79.6101     2.1041  37.836  < 2e-16 ***
## Education    -0.8624     0.1448  -5.954 3.66e-07 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 9.446 on 45 degrees of freedom
## Multiple R-squared:  0.4406,Adjusted R-squared:  0.4282
## F-statistic: 35.45 on 1 and 45 DF,  p-value: 3.659e-07
```

10.3.1 Prediction

As with the case of categorical explanatory variables, we are now interested in predicting the mean expected `Fertility` for any given value of `Education` based on our model coefficients. Recall from ANOVA that we can do this "by hand":

```
betas <- swiss.mod$coefficients
X_pred <- as.matrix(model.matrix(swiss.mod))

# Multiply betas by dummy coded
# matrix using transpose of both
# These are your predictions
# for ctrl, trt1, and trt2
y_pred <- as.vector( t(betas) %*% t(X_pred) )

swiss_pred <- data.frame(swiss, y_pred)
```

Or, we can use the built-in `predict()` function to get confidence intervals on our predictions, too! That's a pain to do by hand every time, so we will use the `predict()` function from here out for linear regression!

Here I'll ask R for `"prediction"` intervals. To avoid warnings about predicting from the same data to which the model was fit, we need to either pass the `model` part of `swiss.mod` to the function as new data or we need to simulate new data. As models become increasingly complex, it becomes increasingly complicated to simulate data appropriately. Therefore, if I am just interested in communicating my results, I do so with the model data.

```
# Make predictions using the model data
y_pred2 <- predict(swiss.mod,
                   newdata = swiss.mod$model,
                   interval = "prediction")

# Combine with original data
swiss_pred2 <- data.frame(swiss, y_pred2)
```

Whichever way you do this, you'll notice that we have a unique value of fit for every value of `Education` in the original data because `fit` is predicted as a continuous function of `Education` in this case:

```
head( swiss_pred2)
```

```
##              Fertility Agriculture Examination Education Catholic
## Courtelary        80.2        17.0          15        12     9.96
## Delemont          83.1        45.1           6         9    84.84
## Franches-Mnt      92.5        39.7           5         5    93.40
## Moutier           85.8        36.5          12         7    33.77
## Neuveville        76.9        43.5          17        15     5.16
## Porrentruy        76.1        35.3           9         7    90.57
##              Infant.Mortality      fit      lwr      upr
```

```
## Courtelary              22.2 69.26186 50.03294 88.49077
## Delemont                22.2 71.84891 52.61363 91.08418
## Franches-Mnt            20.2 75.29831 55.99275 94.60387
## Moutier                 20.3 73.57361 54.31199 92.83522
## Neuveville              20.6 66.67480 47.41244 85.93717
## Porrentruy              26.6 73.57361 54.31199 92.83522
```

We could also make predictions for specific values of `Education` by creating (simulating) new values of `Education`. Below, we make a sequence of new values for `Education` from the minimum to the maximum in 30 equal increments and then make predictions with that object instead of the model data.

```
new_ed <- data.frame(
  Education = seq(from = min(swiss$Education),
                  to = max(swiss$Education),
                  length.out = 30
                  )
  )
```

Now make predictions across the range of observed data.

```
new_y_preds <- predict(swiss.mod, newdata = new_ed, interval = "prediction")

new_preds <- data.frame(new_ed, new_y_preds)
```

Or, you could make a prediction for a single value. Let's say I ask you to find the mean and 95% confidence interval for a specific value of `Education`. Usually we are interested in the maximum and minimum for communicating change in y across the range of x. To do this, you can just make some new data and print the predictions!

```
# Make a data frame containing only the max and min values for Education
point_ed = data.frame(Education = c(min(swiss$Education), max(swiss$Education)))

# Predict new values of y from the model
point_y_pred <- predict(swiss.mod, point_ed, interval = 'confidence')

# Put the predictions in a data frame with
# the min and max values of Education
point_preds <- data.frame(point_ed, point_y_pred)

# Now you can see your predictions
print(point_preds)
```

```
##   Education      fit      lwr      upr
## 1         1 78.74771 74.72578 82.76963
## 2        53 33.90549 21.33635 46.47464
```

Now, it is really easy for us to say:

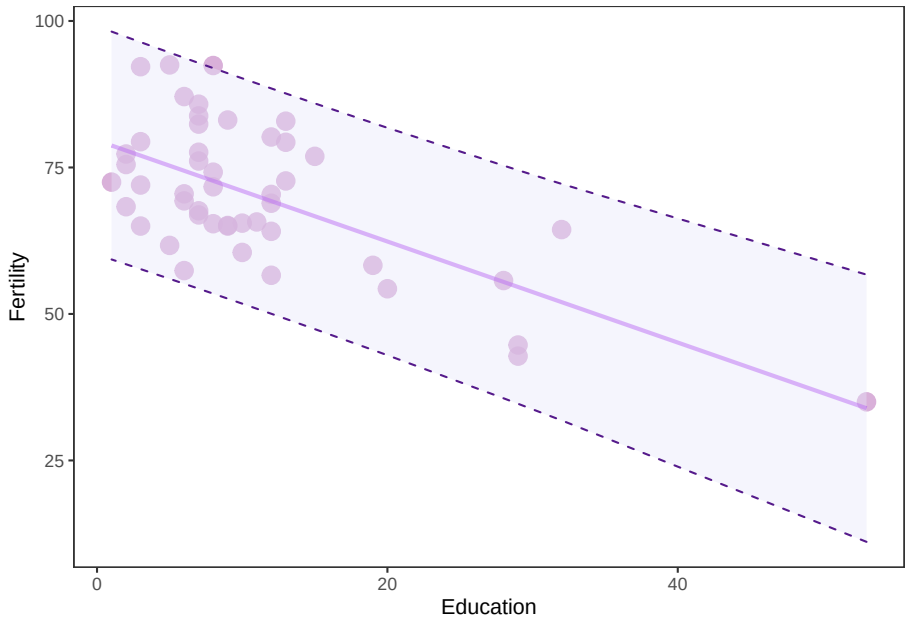
> Fertility rate was inversely related to Education (t = 5.95m, p < 0.05), and Education explained about 44% of the variation in Fertility rate. Across the range of observed education values Fertility decreased from a maximum of 78 (95% CI 75 - 83) at Education of 1 to a minimum of 34 (95% CI 21 - 46) at Education of 53 (Figure 1).

Where is Figure 1?

10.3.2 Plotting

Once we are happy with our predictions, we can go ahead and plot them against the raw data to show how our model fit the data. Here is the code that we used to do this in Chapter 7 but a little more purpley.

```
# Make a pretty plot showing raw data and model predictions
ggplot(swiss_pred2, aes(x = Education, y = Fertility)) +
  geom_point(colour = 'plum3', fill = 'plum3', alpha = 0.75, size = 4) +
  geom_line( aes(y = fit), size = 1, color='purple', alpha = .5) +
  geom_ribbon(aes(ymin = lwr, ymax = upr),
              color = 'purple4',
              fill = 'lavender',
              alpha = .4,
              lty = 2,
              lwd = .5) +
  theme_bw() +
  theme(legend.position = "none", panel.grid = element_blank())
```



Now that is a **money** figure that shows your raw data, the model predictions, and the uncertainty associated with both of these. That's what we want to go for every time - with or without the purpleyness.

Multiple regression proceeds in much the same way. In most cases, it is easiest to make model predictions directly from the observed data because when we have multiple continuous `X` variables they are often correlated with one another. We will examine this in detail in Chapter 11 when we discuss model selection.

10.4 ANCOVA

Now, we are going to step up the complexity a little bit and start to look at how to interpret linear models with more than one variable, and more than one variable type. Exciting, I know!

Last week we worked with the `crickets` data to demonstrate ANCOVA. Let's keep working with that one.

```
# Read cricket data
# This data set contains pulses of
# 2 species of crickets collected under
# varying temperatures
  crickets <- read.csv('data/crickets.txt')
```

We investigated the additive effects of `Species` and temperature (`Temp`) on chirpy pulses of individual crickets and found significant evidence of both.

```
# Fit the model
cricket.mod <- lm(Pulse~Species + Temp, data=crickets)
```

Here is the summary of the linear model:

```
summary(cricket.mod)
```

```
##
## Call:
## lm(formula = Pulse ~ Species + Temp, data = crickets)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -3.0128 -1.1296 -0.3912  0.9650  3.7800
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)  -7.21091    2.55094  -2.827  0.00858 **
## Speciesniv  -10.06529    0.73526 -13.689 6.27e-14 ***
## Temp          3.60275    0.09729  37.032  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.786 on 28 degrees of freedom
## Multiple R-squared:  0.9896,Adjusted R-squared:  0.9888
## F-statistic:  1331 on 2 and 28 DF,  p-value: < 2.2e-16
```

And the model matrix:

```
X <- model.matrix(cricket.mod)
```

You can see that the model matrix still has a column for the `(Intercept)` that represents `Species ex` and a dummy variable called `Speciesniv` to indicate rows in the `cricket` data where `Species == niv`. But, now we also have a column in the model matrix for a continuous variable. Not to fear, the math works exactly the same way as it did for ANOVA and for linear regression.

10.4.1 Prediction

We could do this using linear algebra (matrix math). Note that the math has stayed the same for ANOVA, regression and ANCOVA. That is because they are all just different special cases of the same general model.

```
X_pred <- as.matrix(model.matrix(cricket.mod))
betas <- cricket.mod$coefficients

# Multiply betas by dummy coded
# matrix using transpose of both
# These are your predictions
# for ctrl, trt1, and trt2
y_pred <- as.vector( t(betas) %*% t(X_pred) )

cricket_pred <- data.frame(crickets, y_pred)
```

But since it is a pain to get prediction intervals like this, we'll use the default `predict()` function here as well. I am not going to lie, I am literally just copying and pasting code from ANOVA and regression here and changing the names. This is the power of understanding what actually goes on under the hood for us!

```
# Make predictions
y_pred <- predict(cricket.mod, interval = "prediction")

# Combine with original data
cricket_pred <- data.frame(crickets, y_pred)
```

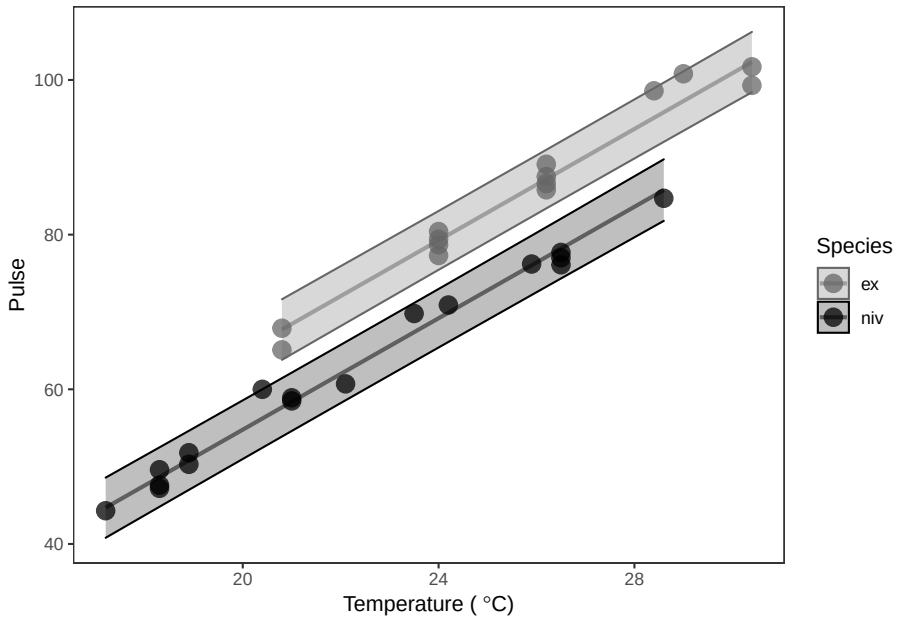
10.4.2 Plotting

Plot the predictions by species. Again, I am pretty much changing the names of the data and the colors at this point. Who'd have thought that fitting and and making predictions from scary ANCOVA models could be so easy!? Dangerously easy…

In this case, though, we should expect to see two lines on our graph if I have not completely lied to you. This is because we have both categorical and continuous explanatory variables in `X`. **Remember that the βs for categorical variables are just adjustments or offsets to the intercept in linear models.** That means that we should have two parallel lines given that we had two groups (so one intercept + 1 offset) and a single slope.

```
# Make a pretty plot showing raw data and model predictions
ggplot(cricket_pred, aes(x = Temp, y = Pulse, group = Species)) +
  geom_point(aes(colour = Species, fill = Species), alpha = 0.75, size = 4) +
  geom_line( aes(y = fit, color = Species), size = 1, alpha = .5) +
  geom_ribbon(aes(ymin = lwr, ymax = upr, color = Species, fill = Species),
              alpha = 0.25) +
  scale_fill_manual(values = c('gray40', 'black')) +
```

```
  scale_color_manual(values = c('gray40', 'black')) +
  xlab(expression(paste("Temperature ( ", degree, "C)"))) +
  theme_bw() +
  theme(panel.grid = element_blank())
```



Ta-da!

10.5 Next steps

Here, we have demonstrated how to communicate the biological predictions of statistical models that we use to test hypotheses. These included most common frameworks for data analysis within the context of linear models. We will continue to apply these tools to as we extend our modeling framework to include non-normal response variables of interest in later chapters. First, in Chapter 11, we'll explore model selection as a way of choosing between hypotheses that are represented by all of these different models.

Chapter 11

Model selection

Sometimes it can be difficult to see the forest for the trees (or anything at all if you spend all your time looking through the bottom of a wine glass). When faced with multiple competing explanations for observed phenomena we may represent these by multiple competing models. Model selection helps us see through the fog to tell us which is best supported. But, it doesn't tell you whether any of your models is good. The wine was good.

11.1 Introduction

As we have learned in the past couple weeks, we often encounter situations for which there are multiple, competing hypotheses about what factors, or combinations of factors, best explain the observed patterns in our response of interest. This uncertainty arises for two primary reasons:

1. Complexity of the study system

Biological systems are complex, and often we are interested in which factor, or set of factors, best predict the patterns we observe in the natural world. In carefully designed experiments, we might be interested in evaluating competing hypotheses about mechanistic drivers of biological phenomena. In complex observational studies, we might simply wish to know what factor or subset of possible factors best predicts the patterns we observe, with the understanding that these findings cannot be used to infer causality (or 'mechanism') although they can help us better design studies that do.

2. Collinearity

Oh, snap! What did he just say? **Collinearity** is the idea that certain explanatory variables are related to one another. I know, I know; last week I told you that *independence of observations* was one of the fundamental assumptions

that we make about linear models. That is, all observations (rows of data) are sampled independently from one another. This is a nice ideal, and in certain experimental designs that are "orthogonal", we can ensure that variables are not collinear. But, in the real world, this is almost never the case.

Model selection offers a means for us to weigh effects of collinearity against the information that is gained as a result of including explanatory variables that are related to one another. In real-world cases, our best model will almost always fall somewhere between a model that contains all of the variables we want to include, and a model that contains only one of those variables.

However, model selection is just as useful for testing hypotheses of rigorously designed, controlled experiments. And as we'll see it can often help to provide more meaningful interpretation of those hypotheses than do p-values alone.

We will be working out of the `tidyverse` as usual for this chapter. We will also be working with functions from the `AICcmodavg` package. You will need to install `AICcmodavg` if you do not already have it installed.

11.2 Model selection tools

Here, we discuss a few different approaches to model selection. As always, I know it is hard to believe, but there is some controversy as to which method of model selection is best for a given situation. We'll cover three types of model selection: stepwise selection, all possible subsets, and *a priori* subsets. While *a priori* model selection is generally preferred for most applications in biology and ecology, step-wise selection is widely used for phylogenetic analyses and all subsets regressions may be useful for some exploratory studies sometimes if that is really the only choice you have you can tell I don't like this one. Therefore, we will briefly discuss all subsets before demonstrating stepwise selection and moving with commonly used *a priori* tools for the rest of the semester.

11.3 All subsets

Just as the name states - compare all possible combinations of variables and pick the one that gives the most information with the least collinearity between predictors. This is largely an exploratory approach or is reserved for cases in which we care solely about prediction. There are a number of R packages that implement all subsets with varying utility and efficiency. These approaches historically relied on Mallow's Cp, but most are now updated to use Akaike's information criterion (see Chapter 11.4)

We will not discuss these techniques in this class because 1) they are usually not needed 2) they can lead to laziness in formulation of hypotheses and in a

worst case data dredging, and 3) plain and simple: there are just better tools available for these purposes now (e.g. GAMM, CART, and network analysis).

Now can you tell I am not a fan?

11.4 Stepwise selection

The basic idea behind stepwise model selection is that we wish to create and test models in a variable-by-variable manner until only "important" (say "well supported") variables are left in the model. The support for each variable is evaluated in turn relative to some pre-determined criterion and an arbitrary (or not) starting point.

While convenient, this approach has some well-known pitfalls. First, this generally is not a useful way to construct biological hypotheses for experiments or for observational studies. Second, it is easy to miss out on important relationships that are not considered because of the automated inclusion or exclusion of 'significant' explanatory variables and the order in which they are entered or dropped. For example, in most readily accessible applications, this tool also does not include interaction terms that might be of biological interest by default. Therefore, regardless of the method used, careful thought is warranted regarding the variables included and their potential mathematical combinations. For most purely predictive situations better tools now exist since the advent of machine learning algorithms.

11.4.1 Forward selection

We start by making a "null" model that includes no explanatory variables. This model is simply a least-squares estimate of the mean. If we think about it in terms of a linear model, the only parameter is the intercept $Y = \beta_0$, so the estimate of `(Intercept)` that R reports from the model is simply the mean of `y` in the data. Let's demonstrate with the `swiss` data.

We write the **null model** like this.

```
data(swiss)
null <- lm(Fertility ~ 1, data = swiss)
```

Mathemetically, the `1` just tells R to make a model matrix with a single column of `1`s called `(Intercept)`. Have a look:

```
head(model.matrix(null))
```

```
##              (Intercept)
## Courtelary             1
## Delemont               1
## Franches-Mnt           1
## Moutier                1
## Neuveville             1
## Porrentruy             1
```

Now that we have a `null` model, we need to make a full model. The **full model** is the model that includes all the variables we want to consider in different combinations. In phylogenetics, these would be different trees that consider varying numbers of splits and different groupings. We can write out the formula for the full model by hand in the `lm()` function, or we can use `.` to tell R that we want it to consider *additive* combinations of all columns other than `Fertility`.

```
full <- lm(Fertility ~ ., data = swiss)
```

Now we perform the forward selection using the `step()` function. Watch them fly by in real time! Here we are telling R to start with the `null` model we created above using `object = null`, but we could actually specify any other model between the `null` and `full` if we wanted to. Next, we tell R that the `scope` of models to consider should include all combinations of explanatory variables (`Education`, `Catholic`, `Infant.Mortality`, and `Agriculture`), including none of them (`null`) and all of them (`full`). Then, we tell R what direction to build models in, either `forward`, `backward`, or `both`.

```
step(object = null, scope = list(lower = null, upper = full), direction = "forward")
```

```
## Start:  AIC=238.35
## Fertility ~ 1
##
##                    Df Sum of Sq    RSS    AIC
## + Education         1    3162.7 4015.2 213.04
## + Examination       1    2994.4 4183.6 214.97
## + Catholic          1    1543.3 5634.7 228.97
## + Infant.Mortality  1    1245.5 5932.4 231.39
## + Agriculture       1     894.8 6283.1 234.09
## <none>                          7178.0 238.34
##
## Step:  AIC=213.04
## Fertility ~ Education
##
##                    Df Sum of Sq    RSS    AIC
## + Catholic          1    961.07 3054.2 202.18
```

```
## + Infant.Mortality  1    891.25 3124.0 203.25
## + Examination       1    465.63 3549.6 209.25
## <none>                          4015.2 213.04
## + Agriculture       1     61.97 3953.3 214.31
##
## Step:  AIC=202.18
## Fertility ~ Education + Catholic
##
##                     Df Sum of Sq    RSS    AIC
## + Infant.Mortality  1    631.92 2422.2 193.29
## + Agriculture       1    486.28 2567.9 196.03
## <none>                          3054.2 202.18
## + Examination       1      2.46 3051.7 204.15
##
## Step:  AIC=193.29
## Fertility ~ Education + Catholic + Infant.Mortality
##
##               Df Sum of Sq    RSS    AIC
## + Agriculture  1   264.176 2158.1 189.86
## <none>                     2422.2 193.29
## + Examination  1     9.486 2412.8 195.10
##
## Step:  AIC=189.86
## Fertility ~ Education + Catholic + Infant.Mortality + Agriculture
##
##               Df Sum of Sq    RSS    AIC
## <none>                     2158.1 189.86
## + Examination  1    53.027 2105.0 190.69


##
## Call:
## lm(formula = Fertility ~ Education + Catholic + Infant.Mortality +
##     Agriculture, data = swiss)
##
## Coefficients:
##      (Intercept)         Education          Catholic  Infant.Mortality
##          62.1013           -0.9803            0.1247            1.0784
##      Agriculture
##          -0.1546
```

Here, we see that the best model is that which includes the additive effects of `Education`, `Catholic`, `Infant.Mortality`, and `Agriculture`, or our `full` model. Go ahead and try it with a different starting `object` or `direction` to see if this changes the result.

11.5 *A priori* selection

The most widely perpetuated approach to model selection is probably *a prior* model selection. This means consideration of only those models for which we have *a priori* reasons for inclusion. These are usually models that are designed to represent competing [biological] hypotheses or to balance those hypotheses within a framework for testing. In simple situations, we compare all the models in which we are interested through a single phase of model selection. Will stick to this approach for class. But, in more complex situations we might apply a hierarchical (multi-phase) approach to reduce complexity and test hypotheses about specific groups of parameters one at a time to avoid inflated type-I error rates (yup, that's still a thing!) when we have lots and lots of models.

11.5.1 Multi-phase (heirarchical) selection

Hierarchical model selection is widely used for complex models that have different underlying processes (and often "likelihoods") within. A great example of these kinds of models are occupancy and mark-recapture models that incorporate "sub-models" for estimating detection probabilities and other "sub-models" for estimating things like presence-absence, survival, or abundance. These methods are widely used in studies of fish and wildlife populations, and are also a cornerstone in modern epidemiology when we want to account for false positives or false negatives.

Essentially, multi-phase model selection means that we impose some kind of hierarchy on the steps we take to test competing hypotheses. For instance, we might first wish to compare hypotheses about factors influencing detection probabilities in our examples above. Then, we could use the best model(s) from that set of hypotheses as the basis for testing hypotheses about factors that influence processes such as survival or breeding probability.

11.5.2 Single-phase selection

This is where we'll spend the majority of our time for the rest of the chapter and the rest of the book. Single-phase selection means that we want to set up an compare a single group of models that each represent a distinct hypothesis (or set of hypotheses in the case of n-way ANOVA, ANCOVA, and multiple regression).

11.5.3 Tools for *a priori* model selection

Here, we will focus on a few common approaches to model selection that can be useful in different situations. We will also discuss the importance of thinking about the hypotheses that are represented by our models and how model

selection results are interpreted as we go. In this realm of model selection, it is important that we limit the number of models considered to avoid introducing spurious hypotheses and drawing junk conclusions. Remember, now matter what model we choose as best, it can't represent a good hypothesis if we don't know what it means. And, no matter what, we will always have a best model even if the best model is a shitty one.

In the words of the great Scroobius Pip in his *Death of the journalist*: > Throw enough shit at the wall and some of it will stick But make no mistake, you're wall's still covered in shit

To ensure that our walls don't get covered in excrement at all, we will examine the historical application of and difficulties of the Adjusted R^2 statistic, and then we will dig into information-theoretic approaches using the Akaike information criterion (AIC) as this, along with other information criteria, is now the primary method used for model selection.

Let's check some of these tools out!

Start by fitting some models, we will use the `swiss` data again this week for the purpose of demonstrating selection tools because it is a noisy data set with lots of complexity and colinearity between variables.

```
data("swiss")

# Fit the model that tests the
# effects of education on the fertility index
mod_Ed <- lm(Fertility ~ Education, data = swiss)

# Fit another model that tests
# effects of % Catholic on Fertility
mod_Cath <- lm(Fertility ~ Catholic, data = swiss)

# Fit a model with additive effects
# of both explanatory variables
mod_EdCath <- lm(Fertility ~ Education + Catholic, data = swiss)

# Fit a model with multiplicative
# effects of both explanatory variables
mod_EdxCath <- lm(Fertility ~ Education * Catholic, data = swiss)
```

We have four models that represent competing hypotheses:

1.`Education` alone is the best explanation among those considered for variability in `fertility`.

2.Percent `Catholic` alone is the best explanation among those considered for variability in `fertility`.

3.The additive effects of `Education` and percent `Catholic` are the best explanation among those considered for variability in `fertility`.

4.The interactive effects of `Education` and percent `Catholic` are the best explanation among those considered for variability in `fertility`.

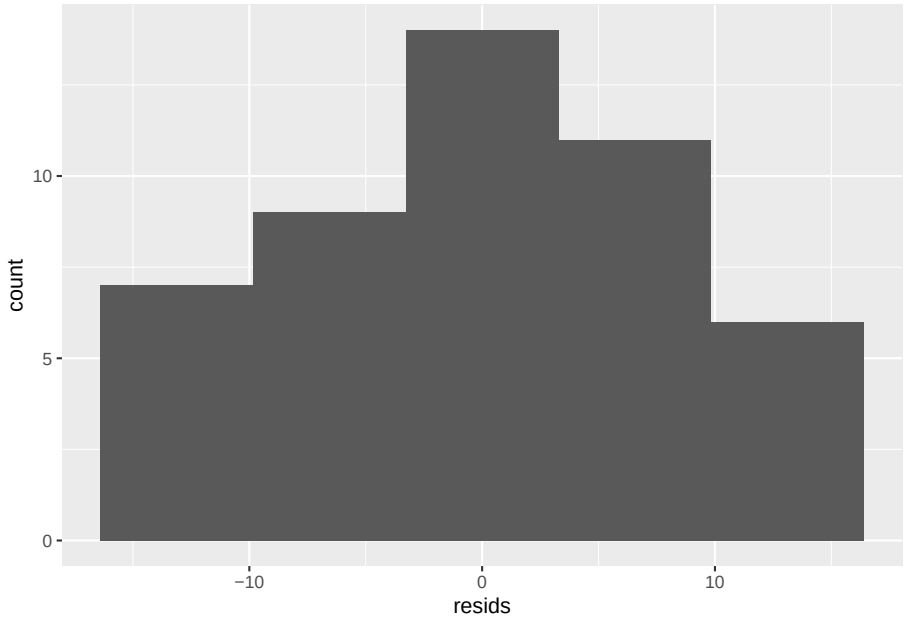
Great, but how can we evaluate which of these hypotheses is best supported by our data?

Have a look at the residuals of the most complex of these models to make sure we haven't shattered the assumptions of linear models. In this case, "most complex" means the model with the most parameters, or `mod.EdxCath`. If you are still unsure as to why this model has more parameters than `mod.EdCath`, then have a look at the output of `model.matrix()` for each of them.

```
# Extract the residuals from
# the fitted model object
resids <- mod_EdxCath$residuals

# Add the residuals to the swiss data
swiss_resids <- data.frame(swiss, resids)
```

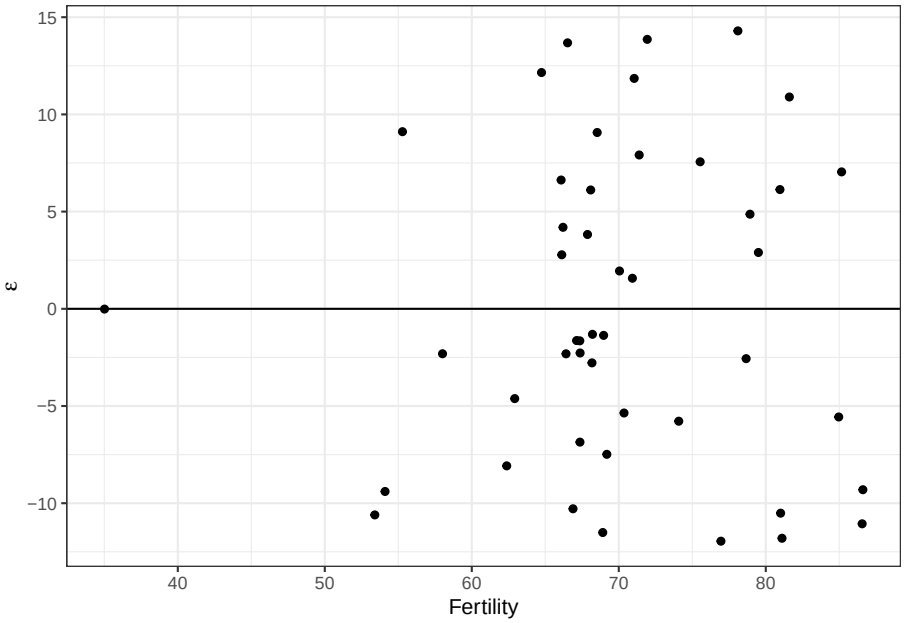
They definitely look like they are normal with a mean of zero:

```
ggplot(swiss_resids, aes(x = resids)) +
  geom_histogram(bins = 5)
```



A glance at the residuals vs fitted seems to indicate that we don't have any concerning patterns in the residuals with respect to the observed value of fertility, although we do see that one point all by itself over to the left that might make us want to puke a little.

```
# Make a pretty plot to make sure we haven't
# completely forgotten about those pesky
# assumptions from Chapter 9
ggplot(mod_EdxCath, aes(x = .fitted, y = .resid)) +
  geom_jitter() +
  geom_abline(intercept = 0, slope = 0) +
  xlab("Fertility") +
  ylab(expression(paste(epsilon))) +
  theme_bw()
```



Now that we have verified we are not in violation of assumptions we can apply model selection to find out if one is clearly better than the others and if so which. Then, we'll use our best model to make predictions just like last week and next week and the week after that…and so on.

Let's start by making a list of our models and giving each element (model) in the list a name:

```
# Making a list that holds are models inside it
mods <- list(mod_Ed, mod_Cath, mod_EdCath, mod_EdxCath)
```

```
# Give names to each element of the list (each model)
names(mods) <- c("Ed", "Cath", "EdCath", "EdxCath")
```

11.5.3.1 Adjusted R^2

The adjusted R^2 offers a relatively simple tool for model selection. It is superior to the multiple R^2 with which we have been working only because it balances the number of parameters in the model with the number of observations in our data.

Just as before, we can look at the summary of our model objects that we have stored in this list.

```
# Education only model, we can take a look, like this
summary(mods$Ed)
```

```
##
## Call:
## lm(formula = Fertility ~ Education, data = swiss)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -17.036  -6.711  -1.011   9.526  19.689
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  79.6101     2.1041  37.836  < 2e-16 ***
## Education    -0.8624     0.1448  -5.954 3.66e-07 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 9.446 on 45 degrees of freedom
## Multiple R-squared:  0.4406,Adjusted R-squared:  0.4282
## F-statistic: 35.45 on 1 and 45 DF,  p-value: 3.659e-07
```

```
# REMEMBER: this model is an object stored in R,
# so we can also look at the names of this summary,
# like this
names(summary(mods$Ed))
```

```
##  [1] "call"          "terms"         "residuals"     "coefficients"
##  [5] "aliased"       "sigma"         "df"            "r.squared"
##  [9] "adj.r.squared" "fstatistic"    "cov.unscaled"
```

Whoa, this is some heavy stuff. To recap, we have made a list of models, each of which are actually lists themselves. Each model has lots of elements. The output of `summary()` for each model is also a list with and the elements have names of their own. Within that final list we can find the `Adusted R-squared` or what `summary()` calls the `adj.r.squared`. It's turtles all the way down all over again.

We can, of course, extract the adjusted R^2 value from the output of `summary()` by name:

```
ed_r <- summary(mods$Ed)$adj.r.squared
cath_r <- summary(mods$Cath)$adj.r.squared
EdCath_r <- summary(mods$EdCath)$adj.r.squared
EdxCath_r <- summary(mods$EdxCath)$adj.r.squared
```

And, we could even put them in a data frame with the original model names to compare the R^2 values.

```
data.frame(
  model = names(mods),
  adj_r_squared = c(ed_r, cath_r, EdCath_r, EdxCath_r)
)
```

```
##     model adj_r_squared
## 1      Ed     0.4281849
## 2    Cath     0.1975591
## 3  EdCath     0.5551665
## 4 EdxCath     0.5700628
```

When we compare adjusted R^2, **the model with the highest R^2 is the "best model"**. So, in this case, we would conclude that `EdxCath` is the best model. But, we have two problems. First, how do we tell if an R^2 value of 0.57 is meaningfully better than an R^2 value of 0.57 statistically? Second, we know we have more parameters in `EdxCath` than in `EdCath`. Are these extra parameters worth the small increase in R^2?. Although we won't dive into statistics like the PRESS statistic, this and other traditional model-selection statistics suffer the same two deficiencies. Finally, the R^2 is a statistic derived from the sum of squared errors in least-squares estimation so we won't be able to use it starting in Chapter 12 when we start to estimate regression coefficients using maximum likelihood estimation from now on.

So, how to life?

11.5.3.2 Information theoretic approaches

Akaike's information criterion (AIC)

This tool (or the popular alternatives BIC, DIC, and WAIC) will be more useful for us during the next several weeks than any of the methods we've discussed so far because it allows us to draw inference based on the likelihood of the model rather than the sum of squared errors, which we will learn that GLMs and other generalizations do not have!

Information-theoretic approaches to model selection are based on the trade off in information gained through addition of parameters (explanatory variables and how they combine) and the added complexity of the models, with respect to sample size. I will hold off on a detailed explanation because you will learn more about this tool in your readings. So, let's cut straight to the chase.

Remember that we made a list of *a priori* models above that we would like to consider.

Have a look at the names of those models just in case you've forgotten them.

```
names(mods)
```

```
## [1] "Ed"      "Cath"    "EdCath"  "EdxCath"
```

We can extract the AIC value for each model in our list by using the `lapply()` or `mapply` functions. These functions will split up the list and "apply" a function to each of the elements of that list. The primary difference is that `lapply()` returns a named list and `mapply()` returns an atomic vector with named elements (easier to work with if we want to combine results with model names).

```
mods_AIC <- mapply(FUN = "AIC", mods)

data.frame(
  Model = names(mods),
  AIC = mods_AIC
)
```

```
##           Model      AIC
## Ed           Ed 348.4223
## Cath       Cath 364.3479
## EdCath   EdCath 337.5636
## EdxCath EdxCath 336.8823
```

Now we have printed a dataframe holding with the names of our models in one column and their AIC values in another. Unlike the R^2 statistic, smaller is better when it comes to AIC (and other I-T approaches), even if the values are negative. **The actual value of AIC for any given model has no interpretation other than relative to the remaining three**. To clarify: if I have a single AIC value for one model, it is meaningless. I must have another

model with the same response (and data!!) to compare with. There is no such think as an inherently "good" or "bad" AIC. They are only interpreted relative to other models in the same candidate set. This is fundamentally different than the R^2 statistic.

At a glance, we can see that our model with the interaction is the 'best' model in the set as indicated by our other statistics, but this time it is only better by less than 1 AIC (**lower AIC is better**). Can we say anything about that?

Funny you should ask. Yes, we can. We have a few general rules of thumb for interpreting the AIC statistic, and we can actually derive a whole set of statistics based on these rankings.

Open can of worms…

```
# First, we need another library
library(AICcmodavg)

# Let's start digging into this stuff
# by making a table that can help us along.
aictab(cand.set = mods, modnames = names(mods))
```

```
##
## Model selection based on AICc:
##
##         K   AICc Delta_AICc AICcWt Cum.Wt      LL
## EdxCath 5 338.35       0.00   0.52   0.52 -163.44
## EdCath  4 338.52       0.17   0.48   1.00 -164.78
## Ed      3 348.98      10.63   0.00   1.00 -171.21
## Cath    3 364.91      26.56   0.00   1.00 -179.17
```

Lots going on here…**What does it all mean**?

We'll walk through the table From left to right. First, notice that the `row.names()` are actually our model names, which is nice. Next, you should take note that the models are ranked in order of increasing AIC. With some context in-hand we can look at each column as follows:

`K` is the number of parameters in each of the models

`AICc` is the AIC score, but it is corrected for sample size. Generally speaking, this means models with many parameters and small number of observations are penalized for potential instability in the likelihood. In general, using the AIC_c is almost always a practical approach because it is conservative when we don't have much data and the effect of the penalty goes away with sufficiently large sample sizes (so it becomes equivelant to AIC).

`Delta_AICc` is the difference in AIC_c between the best model and each of the other models.

`AICcWt` is the probability that a given model is the best model in the candidate set.

`Cum.Wt` is the cumulative weights represented by each of the models from best to last. This can be used to create a 95% confidence set of models.

`LL` is the log likelihood of each model, the very same discussed at the beginning of our discussions about probability distributions!

11.5.3.3 Interpreting AIC statistics

In general:

A lower AIC is better.

Models with ΔAIC_c of less than 2.0 are considered to have similar support as the best model. Models with ΔAIC_c from 2 to 4 have some support in the data, but not as much. Models with $\Delta AIC_c > 4$ have virtually no support.

The ratio of AIC weights (w_i)can be used to interpret the improvement between the best model and each subsequent model. In this example, the best model is only $\frac{0.52}{0.48} = 1.08\times$ better supported than the next best model, but the best two models have all of the support.

Our results suggest that `Education` and `Catholic` are the both important in explaining the variation in `Fertility`, because both are included in any model receiving any support in the candidate set.

Unlike our previous results, we have no clear winner in this case, and we are left wondering whether it is the additive effects or the multiplicative effects of `Education` and `Catholic` that are important. But, we still may want to get estimates for our main effects, at least, so we can make some good solid inference on the effect sizes. If only we had a method for dealing with this uncertainty now…Oh wait, we do!

11.5.3.4 Model averaging

Using model averaging to account for the model uncertainty, we can see that the unconditional confidence interval for `Education` is negative and does not overlap zero, and the opposite trend is evident in the trend for `Catholic`. We also find out that the interaction between `Education` and `Catholic` is actually not significant, which is probably why the main effects model had equivelant support in the candidate set.

```
modavg(mods,
  parm = "Education", modnames = names(mods),
  conf.level = .95, exclude = TRUE
)
```

```
##
## Multimodel inference on "Education" based on AICc
##
## AICc table used to obtain model-averaged estimate:
##
##         K   AICc Delta_AICc AICcWt Estimate   SE
## Ed      3 348.98      10.63   0.00    -0.86 0.14
## EdCath  4 338.52       0.17   0.48    -0.79 0.13
## EdxCath 5 338.35       0.00   0.52    -0.43 0.26
##
## Model-averaged estimate: -0.6
## Unconditional SE: 0.28
## 95% Unconditional confidence interval: -1.14, -0.06
```

```
modavg(mods,
  parm = "Catholic", modnames = names(mods),
  conf.level = .95, exclude = TRUE
)
```

```
##
## Multimodel inference on "Catholic" based on AICc
##
## AICc table used to obtain model-averaged estimate:
##
##         K   AICc Delta_AICc AICcWt Estimate   SE
## Cath    3 364.91      26.56   0.00     0.14 0.04
## EdCath  4 338.52       0.17   0.48     0.11 0.03
## EdxCath 5 338.35       0.00   0.52     0.18 0.05
##
## Model-averaged estimate: 0.15
## Unconditional SE: 0.06
## 95% Unconditional confidence interval: 0.04, 0.26
```

```
modavg(mods,
  parm = "Education:Catholic", modnames = names(mods),
  conf.level = .95, exclude = TRUE
)
```

```
##
```

```
## Multimodel inference on "Education:Catholic" based on AICc
##
## AICc table used to obtain model-averaged estimate:
##
##         K   AICc Delta_AICc AICcWt Estimate   SE
## EdxCath 5 338.35          0      1    -0.01 0.01
##
## Model-averaged estimate: -0.01
## Unconditional SE: 0.01
## 95% Unconditional confidence interval: -0.02, 0
```

Isn't that fantastic? From here we could move on to make predictions based on the model-averaged parameter estimates using what you learned last week. But…what if we weren't convinced so easily and wanted a reliable means of seeing how well our model actually performs now that we've selected one (or more)?

The simple fact of the matter is that we have selected a "best" model, but that doesn't mean our model is necessarily a "good" model.

11.6 Model validation

Once we have selected a best model, or a set of explanatory variables that we want to consider in our analysis, it is important that we validate that model when possible. In truth, comparison of the validity of multiple models can even be a gold-standard method for model selection in itself (e.g. LOO-IC), but we are not going to go there this semester because it would require a much richer understanding of programming than we can achieve in a week.

Model validation is the use of external data, or subsets of data that we have set aside to assess the predictive ability of our models with data that were not used to estimate the parameters. That is, we can use new data to test how well our model works for making predictions about the phenomenon of interest. Pretty cool, I know!

There are lots of different methods for model validation, most of which use some of your data for fitting (or **training**) the model and then saves some of the data for predicting new observations based on your model parameters (**testing**). The most common form of model validation is called cross-validation.

Very generally speaking, there are a large (near-infinite) number of ways to do model validation based on how you split up your data set and how you choose to evaluate predictions. This blog gives a nice overview of these methods with the `iris` data set in R using the `caret` package. We'll write a little code so you can see how it works, though.

11.6.1 Leave-one-out cross validation

We will do some manual cross validation here to demonstrate the general procedure. In order to do that, we will use a "loop" to repeat the process over and over again. For each iteration (`i`), we will choose a subset of the `swiss` data to use for training and set the rest aside for comparing to our predictions. Then, we will fit the education model and store the result. Finally, in each iteration, we will store the training data and the predictions in separate lists that we can then use to visualize the results of our model validation. For this example, we will use leave-one-out (LOO) cross validation, but other methods such as "k-fold" that use specified chunks of data are also common. I just prefer LOO, especially because you are likely to run into this one in the near future as it becomes increasingly easy to use and increasingly useful for model comparison via LOO-IC in Bayesian statistics.

First, we need to make a couple of empty vectors to hold our training data and our predictions for each iteration of our cross-validation loop. We define `n` as the number of rows in the data and will use this as the total number of iterations so that each data point gets left out of the data set exactly once in the process.

```
# Number of rows in the data set
# Also the number of times the for loop will run
n <- nrow(swiss)

# Will hold observation of Fertility withheld for each iteration
fert_obs <- vector(mode = "numeric", length = n)

# Will hold observation of Education withheld for each iteration
ed_obs <- vector(mode = "numeric", length = n)

# Will hold our prediction for each iteration
fert_pred <- vector(mode = "numeric", length = n)
```

Now, drop one data point, fit the model, and predict the missing data point one row at a time until you have done them all.

```
# Repeat this for each row of the data set
# from 1 to n until we have left each row out
for (i in 1:n) {

  # Sample the data, leaving out the ith row
  # These will be our 'training data'
  data_train <- swiss[-i, ]

  # These will be the data we use for prediction
  # We are just dropping the rows that were used for training
```

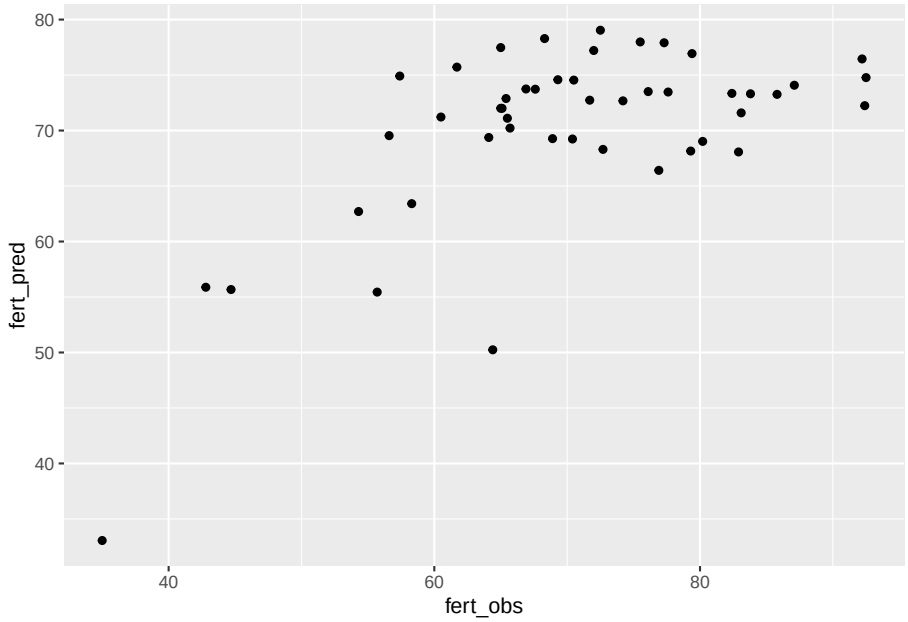
```
  data_pred <- swiss[i, ]

  # Fit the model that tests the effects of Education
  # on the Fertility
  mod_ed <- lm(Fertility ~ Education, data = data_train)

  # Predict Fertility from the fitted model and store it
  # Along with values of Fertility and Education
  fert_obs[i] <- swiss$Fertility[i]
  ed_obs[i] <- swiss$Education[i]
  fert_pred[i] <- predict(mod_ed, data_pred)
}
```

Let's put our observed (left-out) data and our predictions for each iteration in a dataframe that we can use for plotting the results of model validation.

```
loo_df <- data.frame(fert_obs, ed_obs, fert_pred)
```

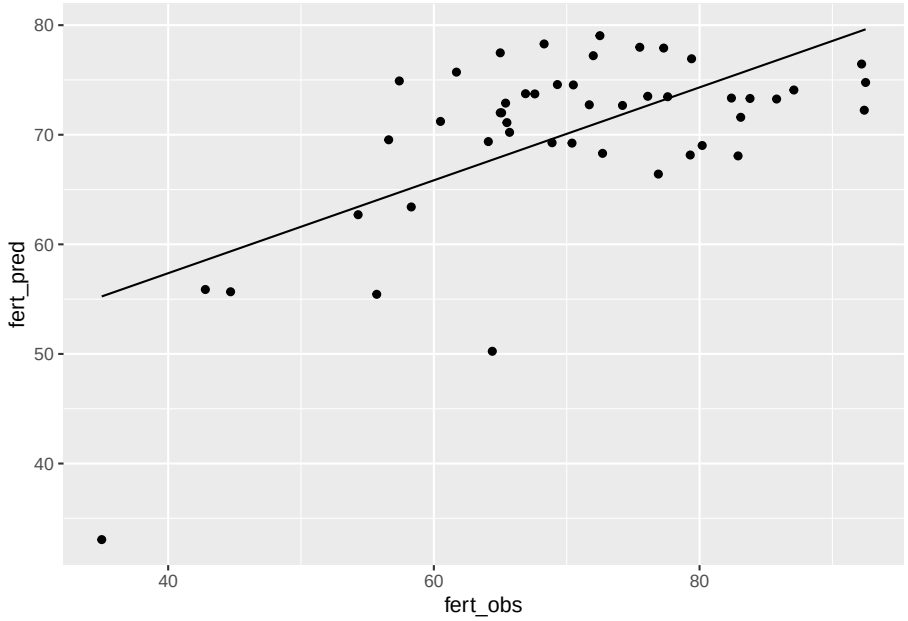
Now, We can look at a plot of our predicted values for each iteration against the data point that was withheld for making the prediction.

```
ggplot(loo_df, aes(x = fert_obs, y = fert_pred)) +
  geom_point()
```



We can add a regression line to the plot to see whether we are over or under predicting `Fertility` from our model in a systematic way:

```
# Fit the model
pred_line <- lm(fert_pred ~ fert_obs, data = loo_df)

# Make predictions
loo_pred <- predict(pred_line, loo_df)

# Add them to the loo_df
loo_df_pred <- data.frame(loo_df, loo_pred)

# Plot the line against the observed and predicted Fertility
ggplot(loo_df_pred, aes(x = fert_obs, y = fert_pred)) +
  geom_point() +
  geom_line(aes(y = loo_pred))
```



You can see that we are generally okay, but tend to under-predict at both low and high values of `Fertility` because the points on either end of the line fall mostly below the line. This is due either to a lack of data at extremes or some overlooked non-linearity in the relationship between X (`Education`) and Y (`Fertility`). If we intentionally collected more data at the extremes of `Education` that could resolve which is the case (because we would either improve prediction at extremes or see the non-linearity in the relationship more clearly). We will use log-transformations to deal help linearize these relationships in some

future examples.

We could also look at the summary of our observed vs predicted regression to see how good we are (how much variance in prediction is explained by the model). In this case, it is not great. If we were going to use this for model prediction, we might want there to be a stronger correlation between the observed and predicted values. In that case, it might just mean collecting more or better data or it could mean re-thinking what is being measured and how.

```
# Extract the R-squared for observed vs predicted
summary(pred_line)$r.squared
```

```
## [1] 0.3991715
```

There are lots of other takes on cross-validation, including popular approaches such as k-fold cross-validation, and a number of simulation-based tools: many of which can be implemented in wrappers available through various R packages. I will leave you to explore these in your leisure time. In general, the more data that are set aside, the more robust the validation is, but we usually don't want to set aside so much of our data that the training model isn't representative of the data we've collected.

If we are happy enough with our cross-validation results at this point we can go ahead and make predictions from our model.

11.7 Next steps

OMG there's more?? I know, right, you're so lucky. In the first 11 chapters of this book we focused on learning how to use R, how to fit statistical models that represent biological hypotheses, how to assess whether those methods are valid for the data collected, how to make predictions about phenomena of interest from our models, and now how to choose between multiple competing models. All of this has relied heavily on assumptions of linear models. Fundamental among those assumptions is that our response variables of interest (and their residuals) conform to the normal distribution. In Chapter 12, we will free ourselves of those shackles by stepping into the world of generalized linear models.

Chapter 12

Logistic regression

"Life or death" is a phrase we reserve for situations that are not normal. Coincidentally, life or death is also a binary variable, and therefore it's residuals are also not normal. Will these zebra mussels live or die? That will be our next adventure, but for that we need the generalized linear model (GLM).

12.1 Introduction

This week we will start to dive into the world of generalized linear models and their implementation and interpretation in R. Before we can do that, we will talk about why we might like to use these methods, and the fact that the GLM actually represents a broad class of models that are highly flexible and incredibly useful. By the end of this week, we want you to be thinking of this as a kind of "go-to" tool for modeling complex, real-world data. Then, we will continue to layer complexity on this framework to extend it further over the next couple of chapters.

We'll work with some packages from the `tidyverse` and we'll use the `StillwaterChoice` data from the class data folder. You can go ahead and load those whenever you are ready to get started.

```
library(tidyverse)
```

12.2 Assumptions of linear models

Wait, what? I thought we were talking about GLMs in this chapter? We are. The first thing you need to know is that linear models are just a special case

of the GLM. That is, the linear model assumes a certain error distribution (the normal) that helps things work smoothly and correctly. The next few weeks of class are all about relaxing the assumptions of linear models so we can actually use them in the real world.

Let's take another look at the assumptions of linear models:

Here are the basic assumptions that we explicitly make when we use linear models, just in case you've forgotten them:

1. Residuals are normally distributed with a mean of zero
2. Independence of observations (residuals)
3. Homogeneity of variances
4. Linear(izeable) relationship between X and Y

12.2.1 Assumption 1: normality of residuals

We've seen these before, but let's recap. For assumption 1, we are assuming a couple of implicit things: 1. The variable is *continuous* (it must be if it's error structure is normal), and 2. The error in our model is normally distributed. In reality, this is probably the least important assumption of linear models, and really only matters if we are trying to make predictions from the models that we make. Of course, we are often concerned with making predictions from the models that we make, so we can see why this might be important. However, more often we are in extreme violation of this assumption in some combination with assumption 4 above to such a degree that it actually does matter. For example, a response variable that is binomial (1 or zero) or multinomial in nature cannot possibly have normally distributed errors with respect to x unless there is absolutely no relationship between X and Y, right? So, if we wanted to predict the probability of patients dying from some medical treatment, or the presence/absence of species across a landscape then we *can't* use the linear models we've been using up until now.

12.2.2 Assumption 2: independence of observations

This time we'll separate assumption 2 into two components: collinearity and autocorrelation of errors. Remember that the manifestation of these problems is in the precision of our coefficient estimates, and this has the potential to change the Type-I/II error rates in our models, causing us to draw false conclusions about which variables are important. As we discussed earlier in the course we expect to see some collinearity between observations, and we can deal with balancing this in our modeling through the use of model selection techniques to reduce Type-I and Type-II error. In the next couple of weeks, we will examine

tools that will help us determine whether or not collinearity is actually causing problems in our models that go beyond minor nuisances. As for the second part, autocorrelation, we can actually use formulations of the GLM that use 'generalized least squares' to include auto-regressive correlation matrices in our analysis that will allow us to relax this assumption of linear models and improve the precision of our parameter estimates. Well, we *could*, we won't do that here.

12.2.3 Assumption 3: homogeneity of variances

Previously, we looked at ways to reduce this issue by introducing categorical explanatory variables to our models. During the coming weeks, we will look at models that allow us to relax this assumption further through the use of weighted least squares and random effects, which can be applied to a wide range of regression methods from linear models to GLMs and GLMMs in Chapter 14 and Chapter 15.

12.2.4 Assumption 4: linearity and additivity

We've already looked at a couple of ways to deal with violations of these two assumptions such as data transformation and/or polynomial formulations of the linear model. We will continue to apply these concepts during the next several weeks.

12.3 Introducing the GLM

There are a number of situations that should just scream "**GLM!!!**" at you. The majority of these are easy to identify because you will know right away that the response variable in which you are interested is clearly not a continuous or normally distributed variable. This is the number one reason for moving into the GLM framework for most people. These include response variables such as counts (integers) and binary (1 or 0) or categorical variables ("Jane", "Bill", "Tracy"), and even probabilities or proportions.

The standard GLM consists of three major components:

1. A random variable (Y) that is our response of interest,
2. Linear predictor(s) of Y, called X, and
3. A invertible "link function" that projects the expectation of Y onto some space based on assumptions about the distributional family of Y.

The first two components are familiar to us. They are the **exact same** basic components of **any** regression formula that takes the following form:

$Y_{i,j} = \beta_0 + \beta_j \cdot X_{i,j}$,

or

$Y = mX + b$,

if you prefer.

So, this much should be familiar. The major change from the linear models with which we have been working is the addition of this invertible link function, and it is the component from which the GLM inherits its name. The link function is just a way for us to put the expectation of the response within the context of an asymptotically normal distribution so that we can relax the assumptions of the linear model to accommodate new data types. In essence, it is very similar to the kinds of transformations that we talked about earlier in the semester, but is used during estimation rather than before hand.

To solve for the coefficients (betas) of a GLM, we move fully into the realm of maximum likelihood, with which you are all undoubtedly still familiar thanks to your close reading of Chapter 5. A given link function is used for the corresponding distribution that we assume for our data set, and a likelihood for that distribution can be defined such that we can calculate the likelihood of the data given our parameter estimates in a manner similar to the method we used for the standard normal distribution earlier this semester. Within this framework, we input different values (or guesses) about the parameter values that maximize the likelihood of our data one step at a time. Once the change in likelihood becomes sufficiently small derivative of y with respect to x = 0), we accept that the algorithm has 'converged' on the optimal estimates for our model parameters (our $\beta_{i,j}$), and the algorithm stops. This all assumes that the parameters follow defined sampling distributions - you guessed it, the normal! You do not need to be able to do this by hand (thank goodness for R!), but you do need to understand what is going on so you can troubleshoot when R says that the model failed to converge…

Let's take a look at a few variable types that we might consider to be common applications for GLM in biology and ecology. We will cover each of these below in detail, here is a list so you know what is coming:

1. Binary response (Chapter 12.4)

2. Count data (Poisson) (Chapter 13)

3. Overdispersed count data (negative binomial, also Chapter 13)

12.4 Binary (logistic) regression

Logistic regression generally is reserved for the case in which we have a binary response that, by definition, can take on values of either 1 or 0. These values can be expressed as outcomes of individual trials (Bernoulli) or as outcomes of some number of trials (Binomial). These data types are common in biological and ecological data analyses, and thus it is important that you understand how to analyze these data when you encounter them because **linear models will not accommodate this data type**. The easiest way to look at what is going on is to use a worked example.

Let's read in another smolt data set that we have not yet played with (it's the last new fish data set for the course, so soak it all up!).

```
choice <- read.csv("data/StillwaterChoiceData.csv")
```

Look at the first few rows of data:

```
head(choice)
```

```
##   path year hatchery length mass date flow
## 1    0 2010        1    176   57  118  345
## 2    0 2005        1    205  101  128 1093
## 3    0 2010        1    180   56  118  345
## 4    0 2010        1    193   74  118  345
## 5    0 2005        1    189   76  128 1093
## 6    0 2010        1    180   65  118  345
```

12.4.1 Data Explanation

These data are from a study that examined factors affecting path choice by wild and hatchery-reared endangered Atlantic salmon smolts during seaward migration in the Penobscot River, Maine. State, local, and federal fishery managers were interested in understanding what factors affected migratory routing through the lower river because there were different numbers of dams, with different estimated smolt mortality rates, on either side of a large island hydropower project in this system. If managers could understand factors influencing migratory route, they might be able to manipulate flows, stocking dates, and dam operation to improve survival of these endangered fish. Furthermore, the results of the study were used to predict the effects of dam removal, and hydropower re-allocation in the lower river on population-level consequences for these fish. These data were part of a larger analysis:

Stich, D. S., M. M. Bailey, and J. D. Zydlewski. 2014. Survival of Atlantic salmon (*Salmo salar*) smolts through a hydropower complex. Journal of Fish Biology 85:1074-1096.

The data consist of the following variables:

`path`: The migratory route used by individual fish. The choices were main-stem of the river (0) or the Stillwater Branch (1) around the island.

`year`: The year in which individual fish were tagged and relocated using acoustic telemetry.

`hatchery`: An indicator describing if fish were reared in the wild (0) or in the federal conservation hatchery (1)

`length`: Fish length (in mm)

`mass`: Fish mass (in grams)

`date`: Ordinal date on which the fish entered the hydrocomplex determined from time-stamps on acoustic receivers

`flow`: Discharge recorded at the USGS gauge in the headpond of the dam several kilometers upstream of the hydropower complex.

NOTE: the results of this analysis won't look like the results from the paper just yet. We will talk about why in a couple of weeks when we introduce generalized linear mixed models.

12.4.2 Data analysis

We are going to use the 1/0 binary data to estimate the effects of a number of covariates of interest on the probability that an individual fish used the Stillwater Branch for migration in each year of this study using logistic regression.

In order to do this, we will use the 'logit' link function, which can be defined as:

```
logit <- function(x) {
  log(x / (1 - x))
}
```

The inverse of the logit function is:

```
invlogit <- function(x) {
  exp(x) / (1 + exp(x))
}
```

We will use this function to transform the probability of using the Stillwater Branch (0 - 1) onto an asymptotically normal x-y space. So, logit(p(Stillwater)) is "normal" way down on the inside of our models.

If our response, `path` is a binary variable where 1 = Stillwater and 0 = mainstem for each fish 1 to n, we can think of p(Stillwater Branch) as:

$p(Stillwater) = \frac{\sum path}{n}$

and the logit of p(Stillwater Branch) can assumed to be normally distributed with a mean of .

$logit(p)\ Normal(\mu, \sigma^2)$

Now that we know we are doing *more or less* the same thing let's move on with fitting the model.

First, since we are interested in the fixed effects of year, and not the linear trend through time, we need to convert year to factor.

```
choice$year <- as.factor(choice$year)
```

Now, if we want to test hypotheses about the influences of explanatory variables on the probability of using the Stillwater Branch, we could make models to represent those hypotheses. For example, if we wanted to test whether `flow` had a significant influence on `path` across `years`, then we could build a model that looks like this:

```
flow_mod <- glm(path ~ year + flow, family = binomial, data = choice)
```

This is the GLM analogue to ANCOVA and it should look pretty much identical except that we now use `glm()` and we have to specify the `family` for the sampling distribution depending on what type of data we have. You can see what families are implemented by running `?glm` and scrolling down to the `family` argument in the help file. If you don't see the one you are looking for, don't worry - it has probably been implented in another package somewhere!

We could make another model that investigates effects of `length` instead of flow:

```
len_mod <- glm(path ~ year + length, family = binomial, data = choice)
```

Or a model that includes both with an annoying name:

```
flow_len_mod <- glm(path ~ year + flow + length, family = binomial, data = choice)
```

We could look at these individually to determine variable-level significance using p-values, or compare them as competing explanations using Akaike information criterion (AIC), which we discussed last week.

```
AIC(flow_mod, len_mod, flow_len_mod)
```

```
##               df      AIC
## flow_mod       7 565.0131
## len_mod        7 570.6512
## flow_len_mod   8 565.7209
```

But, we can also streamline this to get other information about the models. To do this:

First, let's define a slightly more complex set of models based on *a priori* combinations of explanatory variables.

```
# Make an empty list to hold the models
mods <- list()

# Now, add models to the list. Stop and think about what each one means.
mods[[1]] <- glm(path ~ year + hatchery + length + flow, family = binomial, data = choi
mods[[2]] <- glm(path ~ year + flow, family = binomial, data = choice)
mods[[3]] <- glm(path ~ year + hatchery, family = binomial, data = choice)
mods[[4]] <- glm(path ~ year + length, family = binomial, data = choice)
mods[[5]] <- glm(path ~ year + length + hatchery, family = binomial, data = choice)
mods[[6]] <- glm(path ~ year + length + flow, family = binomial, data = choice)
mods[[7]] <- glm(path ~ year + hatchery + flow, family = binomial, data = choice)
```

Next, give the models some names using the formulas for each of the models. *Remember*: models are stored as list objects in R, and each of those list objects (models) has names. We can reference those names using the `$` notation, and from there we can access the actual model `formula` from the `call`. The third element of this `formula` object contains the explanatory variables!! Whoa!

We can extract the `formula` for each model (which is an element in the `mods` list) using a `for` loop to assign them one at a time. Here, we are assigning the i^{th} formula to be the name of the i^{th} element in the list `mods`. Nifty.

```
# Assign the formula for each of the models as the name
for (i in 1:length(mods)) {
  names(mods)[i] <- as.character(mods[[i]]$call$formula)[3]
}
```

Now, we use the `AICcmodavg` package to make a model selection table like we did in Chapter 11.4:

```
# Load the library
library(AICcmodavg)

# Make the model selection table
mod_table <- aictab(cand.set = mods, modnames = names(mods))
```

12.4.3 Interpreting the results

This pretty much proceeds the same way for GLM as it does for linear models until we get to making predictions of the response based on our best model.

Our model selection table is an object in R (*right?*), and we can reference that object using `$` notation, matrix notation `[ , ]`, or by calling rownames to get the index for each of the models. Let's use this approach to get the best model from our candidate set. Here is a worked example in the code that follows:

```
# Print the table
mod_table
```

```
##
## Model selection based on AICc:
##
##                                K   AICc Delta_AICc AICcWt Cum.Wt      LL
## year + flow                    7 565.16       0.00   0.31   0.31 -275.51
## year + hatchery + flow         8 565.23       0.07   0.30   0.61 -274.52
## year + length + flow           8 565.91       0.75   0.21   0.82 -274.86
## year + hatchery + length + flow 9 567.12       1.96   0.12   0.93 -274.44
## year + hatchery                7 569.50       4.34   0.04   0.97 -277.68
## year + length                  7 570.80       5.64   0.02   0.99 -278.33
## year + length + hatchery       8 571.37       6.21   0.01   1.00 -277.59
```

```
# Look at the structure just to show that it is, indeed, an object:
str(mod_table)
```

```
## Classes 'aictab' and 'data.frame': 7 obs. of  8 variables:
##  $ Modnames  : Factor w/ 7 levels "year + flow",..: 1 3 6 4 2 5 7
##  $ K         : num  7 8 8 9 7 7 8
##  $ AICc      : num  565 565 566 567 570 ...
##  $ Delta_AICc: num  0 0.0674 0.7506 1.9557 4.3408 ...
##  $ ModelLik  : num  1 0.967 0.687 0.376 0.114 ...
##  $ AICcWt    : num  0.3078 0.2976 0.2115 0.1158 0.0351 ...
##  $ LL        : num  -276 -275 -275 -274 -278 ...
##  $ Cum.Wt    : num  0.308 0.605 0.817 0.933 0.968 ...
```

Look at the `rownames` of the table. These `rownames` are the index for each of our models as they appear in the `mods` object, and we can use the index to reference objects inside of the `mods` list…

```
rownames(mod_table)
```

```
## [1] "2" "7" "6" "1" "3" "4" "5"
```

This tells us that the rowname for the best model (the one at the top of the table) is . That means that our best model is stored in position 2 of our model list that we named 'mods'. Let's double check it to make sure:

```
mods[[2]]
```

```
##
## Call:  glm(formula = path ~ year + flow, family = binomial, data = choice)
##
## Coefficients:
## (Intercept)     year2006     year2009     year2010     year2011     year2012
##   -2.911624    -0.518632     0.243194    -0.043979    -0.814334    -0.764289
##        flow
##    0.001642
##
## Degrees of Freedom: 758 Total (i.e. Null);  752 Residual
## Null Deviance:     580.2
## Residual Deviance: 551  AIC: 565
```

This looks pretty darn good! We could also do a summary of the model to get the coefficient estimates and the significance codes for the estimated coefficients:

```
summary(mods[[2]])
```

```
##
## Call:
## glm(formula = path ~ year + flow, family = binomial, data = choice)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -1.2183  -0.5757  -0.4401  -0.3564   2.4577
##
## Coefficients:
##              Estimate Std. Error z value Pr(>|z|)
## (Intercept) -2.9116243  0.7981931  -3.648 0.000265 ***
## year2006    -0.5186319  0.6237029  -0.832 0.405671
## year2009     0.2431939  0.4615202   0.527 0.598235
## year2010    -0.0439789  0.6525993  -0.067 0.946271
## year2011    -0.8143343  0.4029438  -2.021 0.043284 *
## year2012    -0.7642890  0.5137641  -1.488 0.136849
## flow         0.0016416  0.0006195   2.650 0.008052 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
```

```
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 580.15  on 758  degrees of freedom
## Residual deviance: 551.01  on 752  degrees of freedom
## AIC: 565.01
##
## Number of Fisher Scoring iterations: 5
```

Cool!! But, what if we wanted the script to always grab the summary of the top model in our model selection table no matter what the `rowname` was? Well, in that case, we could do this:

```
summary(mods[[as.numeric(rownames(mod_table[1, ]))]])
```

```
##
## Call:
## glm(formula = path ~ year + flow, family = binomial, data = choice)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -1.2183  -0.5757  -0.4401  -0.3564   2.4577
##
## Coefficients:
##               Estimate Std. Error z value Pr(>|z|)
## (Intercept) -2.9116243  0.7981931  -3.648 0.000265 ***
## year2006    -0.5186319  0.6237029  -0.832 0.405671
## year2009     0.2431939  0.4615202   0.527 0.598235
## year2010    -0.0439789  0.6525993  -0.067 0.946271
## year2011    -0.8143343  0.4029438  -2.021 0.043284 *
## year2012    -0.7642890  0.5137641  -1.488 0.136849
## flow         0.0016416  0.0006195   2.650 0.008052 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 580.15  on 758  degrees of freedom
## Residual deviance: 551.01  on 752  degrees of freedom
## AIC: 565.01
##
## Number of Fisher Scoring iterations: 5
```

Here we are asking for the `rowname` of the first row in our model selection table. We have to convert that to a number from a character string to reference the

index in the `mods` list, and then we can summarize the best model. Another way to do this is:

```
# First, get the number corresponding to the list index for the best
# model in the candidate set
best <- as.numeric(rownames(mod_table[1, ]))

# Now, get the summary for the model in mods that was the best
summary(mods[[best]])
```

```
##
## Call:
## glm(formula = path ~ year + flow, family = binomial, data = choice)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -1.2183  -0.5757  -0.4401  -0.3564   2.4577
##
## Coefficients:
##              Estimate Std. Error z value Pr(>|z|)
## (Intercept) -2.9116243  0.7981931  -3.648 0.000265 ***
## year2006    -0.5186319  0.6237029  -0.832 0.405671
## year2009     0.2431939  0.4615202   0.527 0.598235
## year2010    -0.0439789  0.6525993  -0.067 0.946271
## year2011    -0.8143343  0.4029438  -2.021 0.043284 *
## year2012    -0.7642890  0.5137641  -1.488 0.136849
## flow         0.0016416  0.0006195   2.650 0.008052 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 580.15  on 758  degrees of freedom
## Residual deviance: 551.01  on 752  degrees of freedom
## AIC: 565.01
##
## Number of Fisher Scoring iterations: 5
```

Since this is really the same thing as ANCOVA we can use the `Anova()` function from the `car` package to get an ANCOVA-like summary for the model to look at significance of our main effects in an Analysis of Deviance table:

```
library(car)
Anova(mods[[best]])
```

```
## Analysis of Deviance Table (Type II tests)
##
## Response: path
##      LR Chisq Df Pr(>Chisq)
## year  12.8043  5   0.025283 *
## flow   7.4471  1   0.006354 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Here, we see that there are significant effects of both `year` and `flow` on our response, `path`. But, how on Earth do we communicate these effects?

12.4.4 Making predictions

The first thing to remember here is that we have used a link function to estimate this model, so we cannot use the same method as before to make predictions about our response from the model coefficients.

The second thing to remember here is that by definition we have used an *invertible* link function to estimate this model so the previous statement is a lie and we actually can use the same method as before to make predictions about our response from the model coefficients. We just need to add an extra step so that we can invert our predictions about the expected value of Y with respect to X.

Confused? Yeah, it's a little confusing. As always an example always goes a long way...

Let's start by grabbing the summary for our `best` model.

```
c.res <- data.frame(summary(mods[[best]])$coefficients)
```

Now we can look at the coefficient estimates. These estimates may not make a lot of intuitive sense at first. That is because they are on the **logit** scale.

```
c.res
```

```
##                Estimate   Std..Error     z.value     Pr...z..
## (Intercept) -2.911624325 0.7981931477 -3.64776913 0.0002645272
## year2006    -0.518631856 0.6237029032 -0.83153670 0.4056705026
## year2009     0.243193850 0.4615202282  0.52694083 0.5982346820
## year2010    -0.043978930 0.6525992875 -0.06739041 0.9462709073
## year2011    -0.814334317 0.4029437518 -2.02096276 0.0432836201
## year2012    -0.764288967 0.5137640800 -1.48762632 0.1368494687
## flow         0.001641583 0.0006194926  2.64988401 0.0080519409
```

If it helps, we can make some predictions. Let's say we want to ask **what was the mean probability of using the Stillwater Branch in 2006 under average flow?**. To answer that question, we would do:

```
# Remember:  y = mx + b
logit_pred2006 <- -2.91162 + 0.00164 * mean(choice$flow)
```

This is the prediction on the logit scale that we used to fit the model:

```
print(logit_pred2006)
```

```
## [1] -1.721732
```

And here it is on the real (probability) scale:

```
invlogit(logit_pred2006)
```

```
## [1] 0.1516482
```

So, we would predict that about 15% of the fish used the Stillwater Branch during average flow periods in 2006. But what if we wanted to see the range of responses to flows across all years so we could compare years?

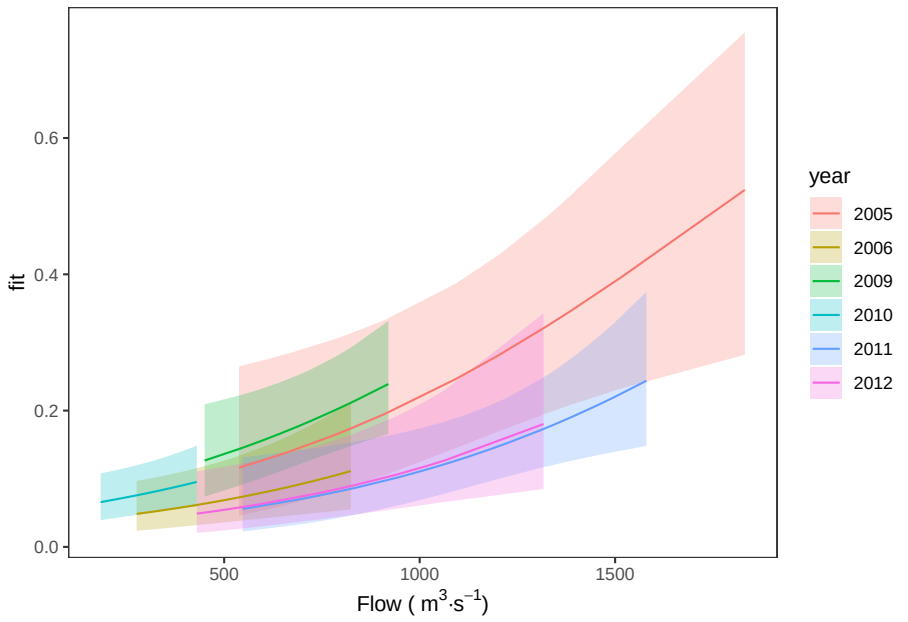
We can do this the same way we did in Chapter 10 with linear models! Now, instead of the `interval`, we need to tell R whether we want the predictions on the `link` scale or the `real` scale, and if it is on the `link` scale, we'll want to tell R that we need the estimated standard errors (`se.fit = TRUE`) so we can derive 95% confidence intervals on the logit scale before we convert them back into probabilities. Finally, we will convert the predictions to the real scale using the `invlogit()` function we wrote inside a call to `apply()`.

```
# Make predictions from the best model
logit_preds <- data.frame(predict(mods[[best]], type = "link", se.fit = TRUE))

# Calculate confidence intervals as 1.96 * standard error
logit_preds$lwr <- logit_preds$fit + 1.96 * logit_preds$se.fit
logit_preds$upr <- logit_preds$fit - 1.96 * logit_preds$se.fit

# Invert the link function
real_preds <- apply(logit_preds, 2, invlogit)

# Combine the predictions with the original data
choice_preds <- data.frame(choice, real_preds)
```

Go ahead and have a look at the `logit_preds` and `real_preds` objects to make sure you understand what we just did.

Now, we can finish by plotting our predictions:

```
ggplot(choice_preds, aes(x = flow, y = fit, fill = year)) +
  geom_ribbon(aes(ymin = lwr, ymax = upr, fill = year), alpha = 0.25) +
  geom_line(aes(color = year)) +
  xlab(expression(paste("Flow ( ", m^3, "\u00b7", s^-1, ")"))) +
  theme_bw() +
  theme(panel.grid = element_blank())
```



You can see that, in general, there is a relatively low probability of an individual fish using the Stillwater Branch, but we see increasing probability of using that route with increasing `flow`.

12.5 Next steps

Here we have demonstrated similarities between the GLM and the models with which we have worked previously. You should realize now that the linear models we've been using are really just a special kind of GLM that uses a "normal" or "Gaussian" error distribution. If we think about what kind of data we actually have, this can open up lots of other "non-normal" options without scaring the life out of us! Hopefully, logistic regression is now a useful tool in your statistical

hardware box. Next, we'll look at how to keep extending this GLM framework for analysis of **count** data in Chapter 13.

Chapter 13

GLM: Count models

13.1 Introduction

This week we continue our exploration of generalized linear models and their implementation and interpretation in R. We will continue to investigate the flexible formulation of GLM for use with count models. By the end of this week, you should have a reasonable understanding of why we might use different kinds of count models, how to use them, and what the results mean. We will dive back into the world of residual diagnostics again this week to look at a few different tools that we have at our disposal for GLM.

We'll keep working with the `tidyverse` this week. But, we'll also check out some utilities from the `MASS` and `boot`, packages to demonstrate new techniques. You'll probably need to install `boot` before you can load it, but we've worked with the others already. Go ahead and load them all when you're ready.

```
library(tidyverse) # For all the good stuff in life
library(boot) # For inv.logit() function
library(MASS) # For negative.binomial family in glm()
```

13.2 Poisson regression

Poisson regression is useful for situations in which we have a response (independent variable) that is a count. These are discrete data that cannot be considered continuous because it is impossible for them to take on non-integer or non-negative values. Common examples of these types of responses include species count data in ecology, cell or colony counts in biology, and the number of respondents or patients reporting a side-effect or symptom of interest in health care studies.

For the Poisson family, the link function that we will use is the "log" link function. This one is exactly what it sounds like. This link function allows us to work with data that are constrained to be non-negative, a desirable property when we are working with count data.

Let's use the `crab` data set to demonstrate the GLM with Poisson data. We will walk through this data set for both the Poisson and negative binomial examples, addressing some distributional assumptions and model fit along the way.

```
##   color spine width mass satellites
## 1     2     3  28.3 3.05          8
## 2     3     3  26.0 2.60          4
## 3     3     3  25.6 2.15          0
## 4     4     2  21.0 1.85          0
## 5     2     3  29.0 3.00          1
## 6     1     2  25.0 2.30          3
```

```
## 'data.frame': 173 obs. of  5 variables:
##  $ color     : int  2 3 3 4 2 1 4 2 2 2 ...
##  $ spine     : int  3 3 3 2 3 2 3 3 1 3 ...
##  $ width     : num  28.3 26 25.6 21 29 25 26.2 24.9 25.7 27.5 ...
##  $ mass      : num  3.05 2.6 2.15 1.85 3 2.3 1.3 2.1 2 3.15 ...
##  $ satellites: int  8 4 0 0 1 3 0 0 8 6 ...
```

13.2.1 Data explanation

These data represent the number of satellite male horseshoe crabs per female (rows) in relation to a number of characteristics of the females. Here, our response of interest is `satellites` and the female characteristics are our explanatory (independent) variables. These include female color, spine condition, carapace width, and mass (g).

The full citation for the paper on which this data set is based is here:

H. J. Brockmann. 1996. Satellite male groups in horseshoe crabs, *Limulus polyphemus*. Ethology 102:1-21. doi:10.1111/j.1439-0310.1996.tb01099.x

We are going to convert color to a `factor` to start because it is currently stored as a numeric variable.

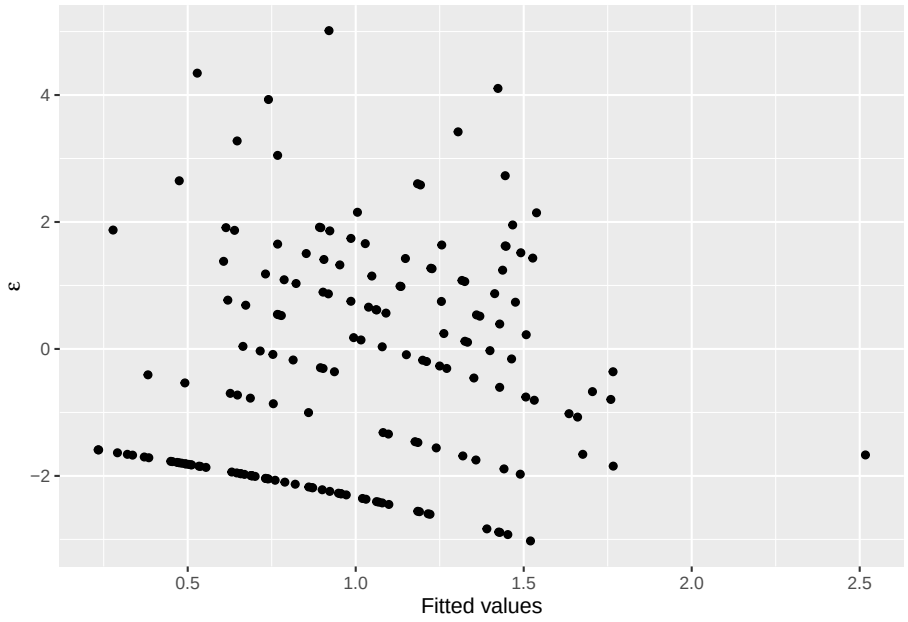
```
# We want to convert color to
# a factor right off the bat
crabs$color <- as.factor(crabs$color)
```

Next, we'll fit a "full" model that assumes the number of `satellites` is a function of width, mass, spine condition, and color.

```
# Fit a model
count_mod <- glm(
  satellites ~ width + mass + spine + color,
  data = crabs,
  family = "poisson"(link = log)
)
```

Before we go any further, let's have a quick look at the model diagnostics using the methods we applied to linear models in Chapter 9. Right away, we can see that this model is not a very good fit to the data.

```
ggplot(count_mod, aes(x = .fitted, y = .resid)) +
  geom_jitter() +
  xlab("Fitted values") +
  ylab(expression(paste(epsilon)))
```



This brings us to the next important point we need to make about GLMS…

> Even though we are relaxing the assumptions of linear models, we still need to check to make sure the models we use are valid with respect to our assumptions.

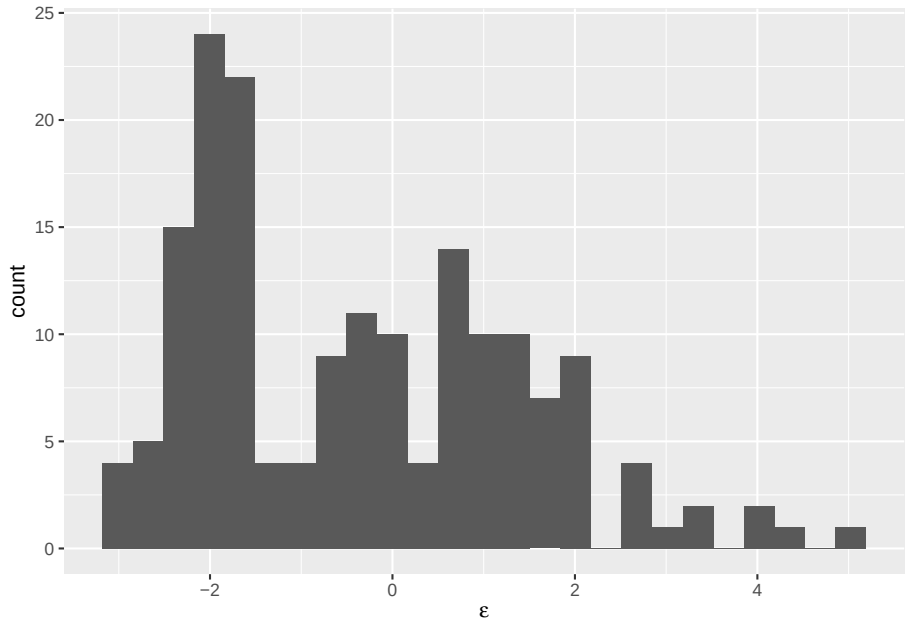
This will become considerably more complicated as we begin to move into distributions other than the binomial and the Poisson, as our standard methods

become less and less applicable and in-depth model validation becomes more obscure and more involved.

So, what is going on here? At this point a few things should jump out at you from this plot. First, there is some kind of trend happening at the bottom of our plot where the residuals slant downward from left to right. That's something that should scream "not good!" at you, even if you don't know why. Second, this plot doesn't really make it look like our residuals are symmetrically distributed with a mean of zero. But, it is really hard to tell from this graph, especially because of the weird patterns at the bottom of the scatter plot.

We can check this second concern using a histogram if we want to see if it "looks normal" like this:

```
ggplot(count_mod, aes(x = .resid)) +
  geom_histogram(bins = 25) +
  xlab(expression(paste(epsilon)))
```



Now that we see it presented this way, it is pretty clear that our residuals are not symmetrically distributed around zero even if the mean is about zero. We could also calculate some descriptive statistics for the residuals to really nail this down.

```
mean(count_mod$residuals)
```

```
## [1] -0.01809651
```

Hmmm...the mean is fairly close to zero here, but we've already talked about the fact that the mean isn't a very good descriptive statistic for distributions that we suspect or know are not normal. What about the median as a measure of central tendency?

```
median(count_mod$residuals)
```

```
## [1] -0.281012
```

Wow, okay, that is a bit more negative than the mean, and at this point we may start to question whether we can say that the residuals are normally distributed with a mean of zero. If we are uncomfortable coming right out and saying that the residuals are not normal, we could always use a Shapiro-Wilk normality test to check (see Chapter 6 if you've forgotten what this one is).

```
shapiro.test(count_mod$residuals)
```

```
##
##  Shapiro-Wilk normality test
##
## data:  count_mod$residuals
## W = 0.82351, p-value = 3.419e-13
```

Remember that the null hypothesis here is that the sampling distribution of our residuals is normal, so a `p-value` of < 0.05 tells us to reject the null and now our residuals are officially not normal! Crap, wasn't the whole point of using GLM supposed to be so we could keep meeting this assumption?

So, why has this happened to us? Recall that the Poisson distribution is controlled by a single parameter, λ, that is both the mean and the variance of the distribution. If we had started by doing data exploration we would have, of course, noticed that even though the data represent counts, they are pretty clearly over-dispersed (variance is much larger than mean) and are indicative of a negative binomial sampling distribution.

For now, we won't bother to look at these results from this model because the link function is the same, so we can get the results from the negative binomial regression in the same way. Plus, if our model is in violation of assumptions then the results will be unreliable anyway.

13.3 Negative binomial regression

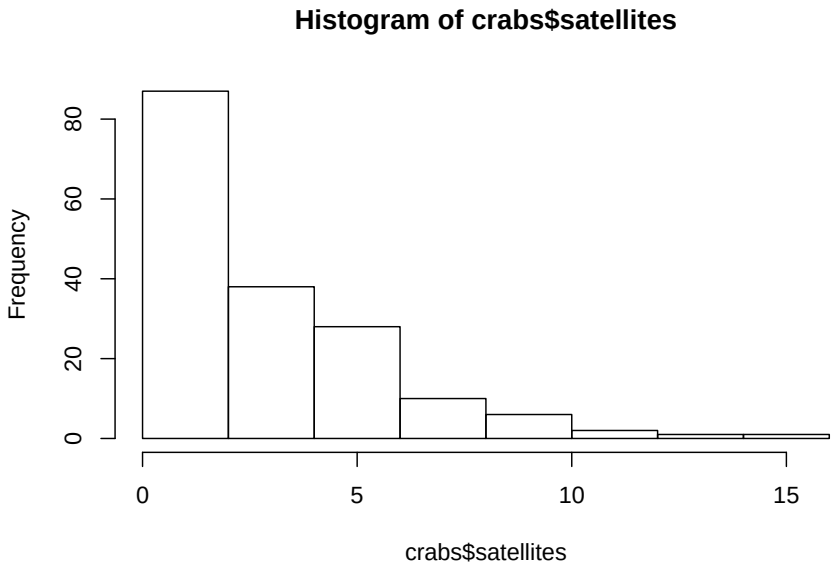
Okay, moving on with life, let's take a look at the negative binomial regression model as an alternative to Poisson regression. Truthfully, this is usually where

I start these days, and then I might consider backing down to use of Poisson if all assumptions are actually verified (but, **this has literally never happened for me**).

We will start this time by actually doing some data exploration before our analysis. This is really how we should have started above, but that would have ruined all the fun.

First, look at the distribution of the data. Here, it should be pretty obvious to you by now that these are count data for which the mean is not equal to the variance…right?

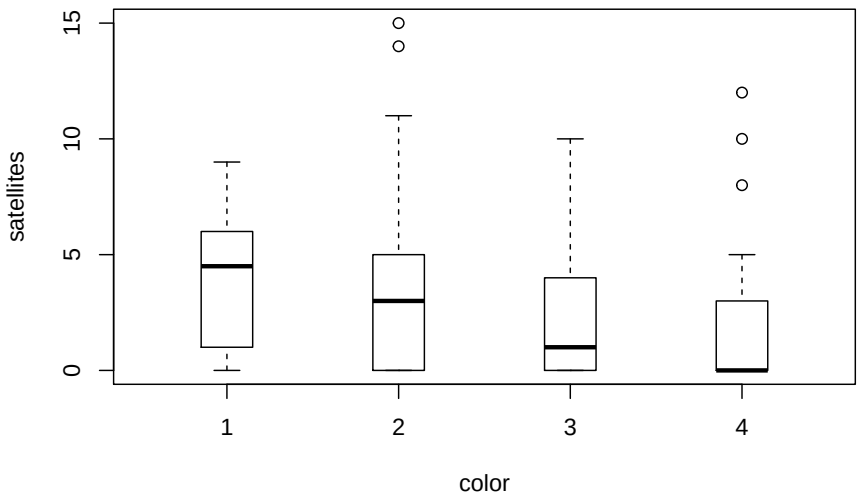
```
hist(crabs$satellites)
```



If you think back to a couple of weeks ago, you'll remember this is a pretty typical example of the negative binomial distribution.

We can take a look at how this shakes out between our groups (`color`) as well.

```
boxplot(satellites ~ color, data = crabs, boxwex = 0.3)
```



And, you can see here that even within groups the distributions do not look like they are normal or like they have equal variances, so we will fit a GLM that assumes the response is drawn from a negative binomial distribution.

For this example, we will use a function called `glm.nb`. This function allows us to estimate a GLM for lets us estimate parameters for a GLM that uses the negative binomial error distribution and estimates the "overdispersion parameter" for the negative binomial distribution. You, of course, remember this parameter and know it as `theta` or `size` from our discussions about sampling distributions.

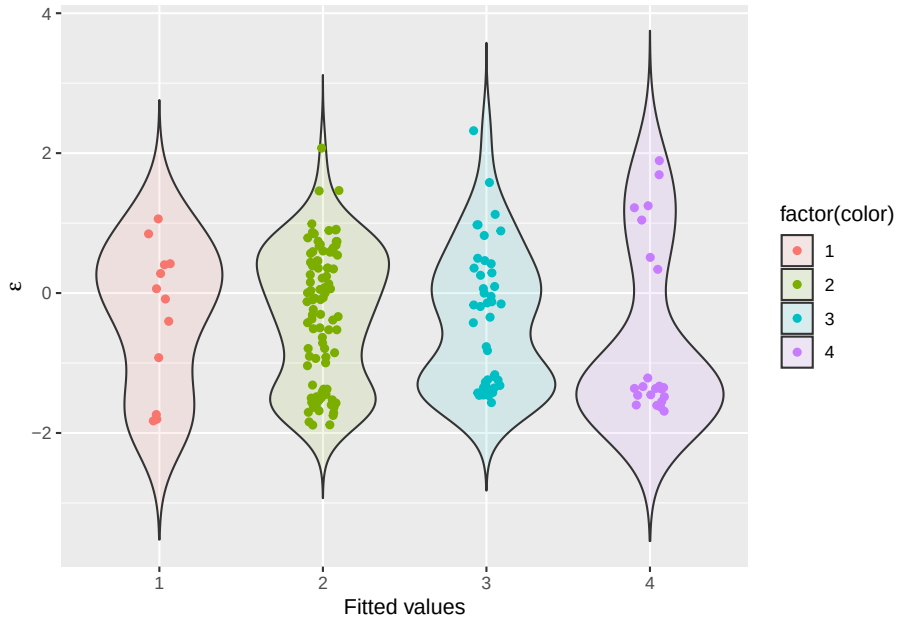
We can fit it with the GLM function like this as long as we have the `Mass` package loaded:

```
neg_mod <- glm(satellites ~ width + mass + color,
  data = crabs,
  family = "negative.binomial"(theta = 1)
)
```

Play around with the two formulations above and see if there's a difference. *Clue*: there's not, really. Just two different ways to do the same thing. The functionality in the `glm` function only came around recently, that's all.

Now, let's take a look at the distribution of the residuals. I am going to work with the object I fit using the `glm()` function. This time, we'll split our residuals out by `color` group so we can see where the problems are

```
ggplot(neg_mod, aes(x = color, y = .resid, fill = factor(color))) +
  geom_violin(alpha = 0.1, trim = FALSE) +
  geom_jitter(aes(color = factor(color)), width = .1) +
  xlab("Fitted values") +
  ylab(expression(paste(epsilon)))
```



Now we are starting to look a lot more "normal" within groups, and we are getting more symmetrical in our residual sampling distributions. If we had to move forward with this model we could probably do so with a straight face at this point. However, it looks like most of the area of each violin above is below zero (so the mean of our residuals is also negative and non-zero), so we may have some issues going on here that could make this model bad for prediction.

But, how does this compare to the Poisson model for count data? We can use model selection to compare the Poisson model to the negative binomial model, since the response is the same in both cases.

```
AIC(count_mod, neg_mod)
```

```
##           df      AIC
## count_mod  7 919.8796
## neg_mod    6 757.9566
```

Clearly the negative binomial model is far superior to the Poisson model here. Now, with a reasonable model in hand we could proceed with data visualization,

but we might also rather have a "good" model to work with instead of just one that is "good enough". It turns out that the really problem behind the weirdness in our residuals is actually due to an excess number of zero counts of `satellite` males on a lot of our females. This is actually really common in count data, where we might observe a lot of zeros before actually counting a success. And, once we have one success (e.g. a parasite infection or illness), we often have extreme events with lots of successes. For example, it might take a while to find a fish with a parasite, or a human with a specific illness, but then that individual could have dozens or hundreds of parasites, or the illness might pop up in a cluster of hundreds of individuals in one location depending on the intensity of the infection. For these cases, we'll need to deal with the issue these excessive zeros ("zero inflation") directly.

13.4 Zero inflation

The fits of these two models, in reality, suggest the need to for what is becoming an increasingly common statistical tool: the zero inflated count model. Zero inflation (excess zeroes in count data) can arise by one of two mechanisms: true ("process") zeros and observational zeros that result from imperfect detection.

One approach to dealing with this is to use a **hurdle model**. The idea is to make two separate models: 1) a logistic regression model to help us determine which factors influence whether the phenomenon of interest even occurred (0 or 1), and 2) a count model to help us determine what factors influence with the frequency of occurrence given that it occurred in the first place.

When these models are linked mathematically, we call it a "mixture model" - an approach that has become very popular for accounting for imperfect detection when estimating abundance of organisms. For now, let's just look at the hurdle model for our crab data as the n-mixture approach falls solidly in the realm of "advanced" methods we'll not discuss in this book.

First, we need to make a binary indicator variable to represent whether or not any satellites were present:

```
# Make a new column for count
# and absence (of satellite males)
# and initialize to zero
crabs$present <- 0

# Assign a '1' if any satellites were
# observed
crabs$present[crabs$satellites > 0] <- 1
```

Now, the first step in the hurdle model is to fit a logistic regression model

to predict how our response is affected by some combination of explanatory variables.

```
step1 <- glm(present ~ mass, data = crabs, family = "binomial")
summary(step1)
```

```
##
## Call:
## glm(formula = present ~ mass, family = "binomial", data = crabs)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -2.1106  -1.0750   0.5427   0.9122   1.6323
##
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept)  -3.6933     0.8799  -4.197 2.70e-05 ***
## mass          1.8145     0.3766   4.818 1.45e-06 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 225.76  on 172  degrees of freedom
## Residual deviance: 195.74  on 171  degrees of freedom
## AIC: 199.74
##
## Number of Fisher Scoring iterations: 4
```

Here we see that mass has a significant effect on whether or not *any* satellite males are present. You could imagine fitting any number of plausible biological models for comparison using AIC at this point.

Step 2 is to fit a count model to explain the effects of some combination of explanatory variables on the frequency with which the phenomenon occurs given that it ever occurred in the first place. **Note**: This does not have to be the same combination of explanatory variables. In fact, it is always conceivable that different processes influence these two distinct phenomena. As with the count-absence model, you could even fit a candidate set of models and proceed with model comparisons using AIC.

```
# Make a model relating the number
# of satellite males to the mass
# of female crabs
step2 <- glm(
```

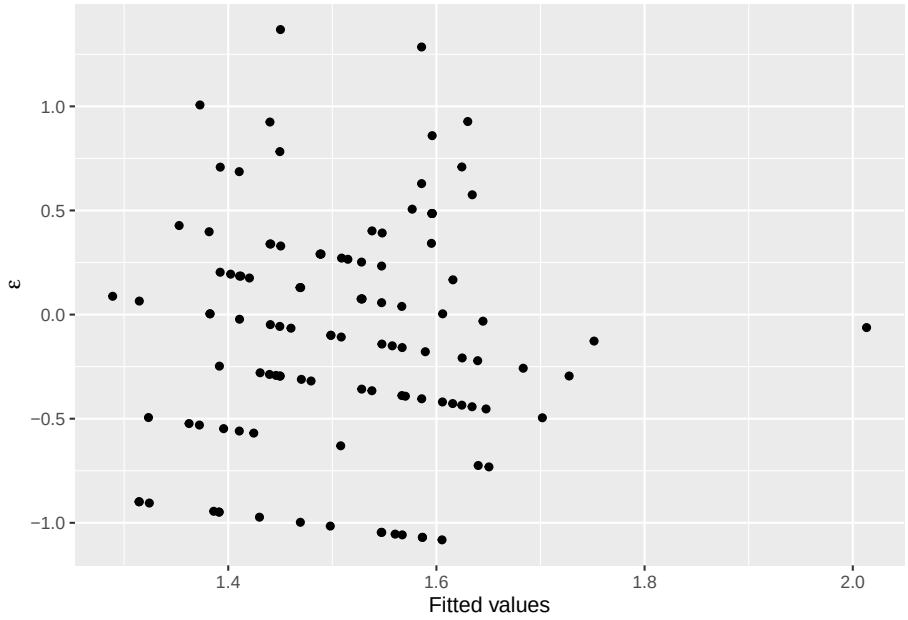
```
  satellites ~ mass,
  data = crabs[crabs$satellites != 0, ],
  family = "negative.binomial"(theta = 1)
)

# Print a summary of the model
summary(step2)
```

```
##
## Call:
## glm(formula = satellites ~ mass, family = negative.binomial(theta = 1),
##     data = crabs[crabs$satellites != 0, ])
##
## Deviance Residuals:
##      Min        1Q    Median        3Q       Max
## -1.08178  -0.43875  -0.09937   0.26840   1.36868
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)   1.0036     0.2726   3.681 0.000363 ***
## mass          0.1941     0.1018   1.907 0.059190 .
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for Negative Binomial(1) family taken to be 0.3128844)
##
##     Null deviance: 35.931  on 110  degrees of freedom
## Residual deviance: 34.849  on 109  degrees of freedom
## AIC: 584.04
##
## Number of Fisher Scoring iterations: 4
```

From these results, we can see that our count models in the previous sections were really just picking up on the large number of zeroes in our data set. We know this because of the differences in the results between the models `step1` and `step2`.

Likewise, we can take another look at our model diagnostics for `step2` to see if our diagnostic plots look more reasonable now

```
ggplot(step2, aes(x = .fitted, y = .resid)) +
  geom_jitter() +
  xlab("Fitted values") +
  ylab(expression(paste(epsilon)))
```



Here, we can see that our residual plots indicate a pretty drastic improvement.

13.4.1 Predictions

Now that we are finally happy with our residual plots (wow, that took a lot longer than fitting any of the models!) we can make a plot of our predictions against the raw data to see how we did.

Let's start with the count/absence component that we fit in `step1`:

Make some predictions from the model using a sequence of new values of `mass` based on what we observed in our data set:

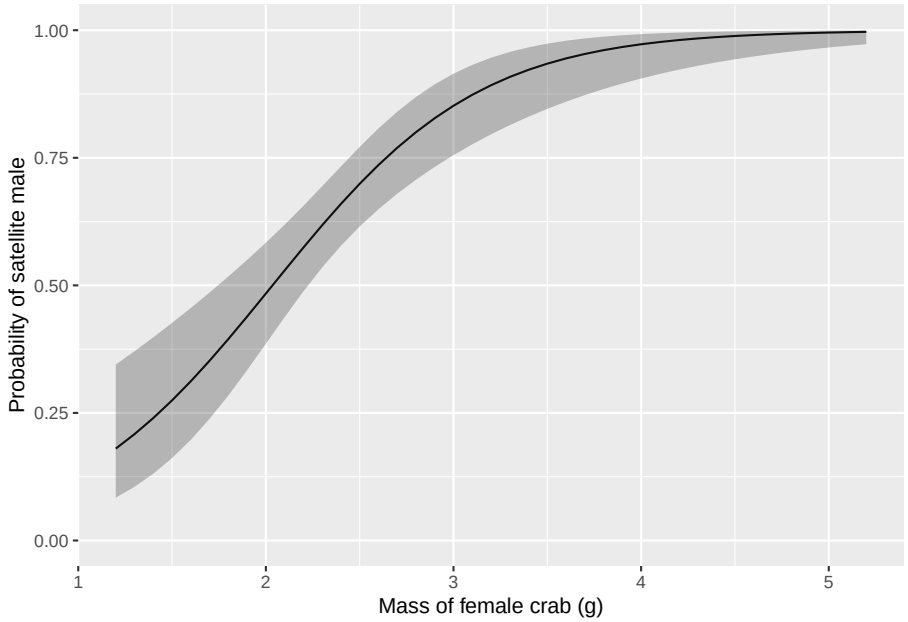
```
# Sequences of new masses based on the min and max observed
# in our data
new_mass <- data.frame(mass = seq(min(crabs$mass), max(crabs$mass), .1))

# Make predictions using step1 model and the new_mass df
count_preds <- data.frame(
  predict(step1, newdata = new_mass, type = "link", se.fit = TRUE)
)

# Get 95% confidence intervals
count_preds$lwr <- count_preds$fit + count_preds$se.fit * qnorm(0.025)
count_preds$upr <- count_preds$fit + count_preds$se.fit * qnorm(0.975)
```

```
# Invert the logit-link function
count_preds <- apply(count_preds, 2, boot::inv.logit)

# Combine the new masses and the predictions with which
# they are associated
pres_preds <- data.frame(new_mass, count_preds)
```

Important: Our hurdle model actually contains two models (`step1` and `step2`). The `step1` component is actually **logistic regression** and therefore uses the **logit** link function that we introduced in Chapter 12, so we need to invert the logit to get the probability of a female having any `satellite` males as a function of `mass`. This is done in the code above. Make sure you understand how and why we do this!

Once you've got that down, it's all over but the plotting. Here is how predicted probability of `satellite` male crab count changes across the range of observed female `mass`:

```
ggplot(pres_preds, aes(x = mass, y = fit)) +
  geom_line() +
  geom_ribbon(aes(ymin = lwr, ymax = upr, color = NULL), alpha = .3) +
  scale_y_continuous(limits = c(0, 1)) +
  xlab("Mass of female crab (g)") +
  ylab("Probability of satellite male")
```



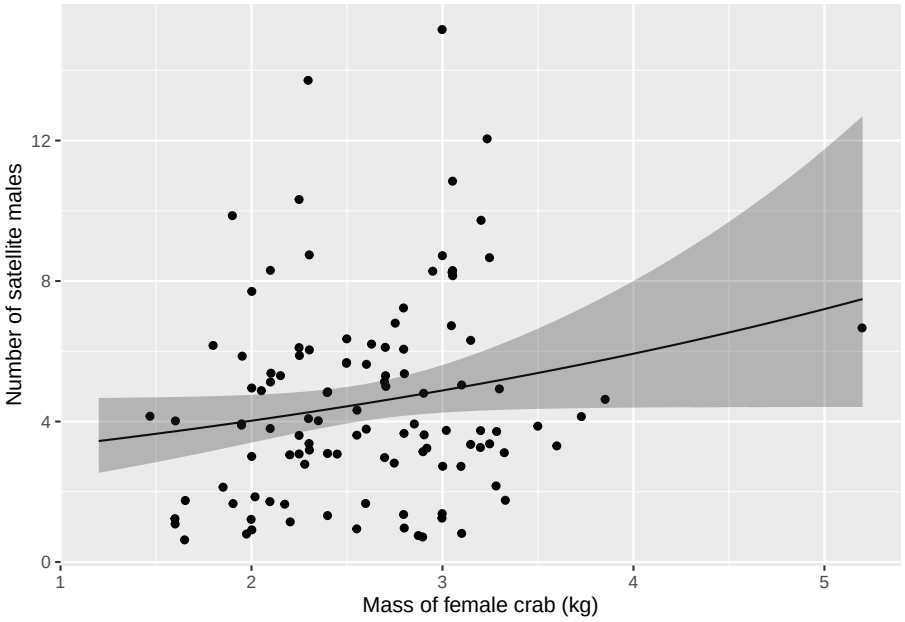
Finally, we can make a plot of the number of `satellite` males we would expect to see on a female crab given that she had attracted any males in the first place.

Also important: We need to remember here that we have two different models. The first model `step1` was a binary logistic regression, so it used the logit link. The second model `step2` was a count model and used the log link. That means we need to invert the log link to get our predicted counts back on the real scale.

```
# Sequences of new masses based on the min and max observed
# in our data
new_mass <- data.frame(mass = seq(min(crabs$mass), max(crabs$mass), .1))

# Make predictions using step2 model and the new_mass df
count_preds <- data.frame(
  predict(step2, newdata = new_mass, type = "link", se.fit = TRUE)
)

# Get 95% confidence intervals
count_preds$lwr <- count_preds$fit + count_preds$se.fit * qnorm(0.025)
count_preds$upr <- count_preds$fit + count_preds$se.fit * qnorm(0.975)

# Invert the log link function
count_preds <- apply(count_preds, 2, exp)

# Combine the new masses and the predictions with which
# they are associated - overwriting on the fly - yuck!!
count_preds <- data.frame(new_mass, count_preds)
```

Here is a plot showing how the number of `satellite` males observed changes with the `mass` of female horseshoe crabs:

```
ggplot(count_preds, aes(x = mass, y = fit)) +
  geom_line() +
  geom_jitter(aes(x = mass, y = satellites), data = crabs[crabs$satellites > 0, ]) +
  geom_ribbon(aes(ymin = lwr, ymax = upr, color = NULL), alpha = .3) +
  xlab("Mass of female crab (kg)") +
  ylab("Number of satellite males")
```



Well, it doesn't exactly inspire great confidence in the biological relationship between female horseshoe crab mass and the number of satellite males she attracts, but that is exactly why it is so important to communicate these effects

13.5 Next steps

This chapter has provided an overview of GLMs that we can use for count data, and demonstrates one way to handle cases of skewed counts or (more extreme case) zero-inflated counts. These are common "problems" in biological and ecological data that are easily resolved within the flexible framework of GLM, which includes all of the other models we've looked at since Chapter 6. In Chapter 14 and Chapter 15 we'll look at how to extend this framework even further (another umbrella) to include repeated observations and relatedness between groups when we introduce mixed effects models.