

Applied Biological Data Analysis

Statistics and R for Biologists

Nick Green, Kennesaw State University

January 13, 2022

Contents

Preface	7
License and permissions	7
Course description	7
Course objectives	8
Course requirements	8
Recommended reading	8
Course organization	9
About the author	9
1 Statistics in modern biology	11
1.1 Overview	11
1.2 Statistics in modern biology	11
1.3 Misuses of statistics	16
1.4 <i>P</i> -values and null hypothesis significance testing (NHST)	35
1.5 Alternatives to NHST	43
2 Introduction to R	63
2.1 Getting started with R	63
2.2 Download and install R (and RStudio)	66
2.3 Using R	67
2.4 A first R session	70
2.5 Write and execute commands in the R console	86
2.6 Basic R data structures	108
2.7 R data types	125
2.8 Manage R code as scripts (.r files)	131
2.9 Manage and use R packages	132
2.10 R documentation	138
3 Data manipulation with R	145
3.1 Data import and export	145
3.2 Making values in R	163
3.3 Selecting data with []	171
3.4 Managing dates and characters	180

3.5	Data frame management	197
4	Exploratory data analysis	217
4.1	Descriptive and summary statistics	218
4.2	Visualizing data distributions	234
4.3	Statistical distributions	264
4.4	Fitting and testing distributions	315
4.5	Data transformations	332
4.6	Multivariate data exploration	355
4.7	Ordination (brief introduction)	390
4.8	Common statistical problems	401
5	Generalized linear models (GLM)	417
5.1	Prelude with linear models	417
5.2	GLM basics	447
5.3	Log-linear models	457
5.4	Poisson GLM for counts	476
5.5	Quasi-Poisson and negative binomial GLM	494
5.6	Logistic regression for binary outcomes	512
5.7	Binomial GLM for proportional data	541
5.8	Gamma models for overdispersed data	549
5.9	Beyond GLM: Overview of GAM and GEE	555
6	Nonlinear models	557
6.1	Background	557
6.2	Nonlinear least squares (NLS)	560
6.3	Michaelis-Menten curves	561
6.4	Biological growth curves	584
6.5	Dose response curves	594
6.6	Alternatives to NLS	600
7	Mixed models	621
7.1	Prelude (GLM)	621
7.2	Linear mixed models (LMM)	622
7.3	Generalized linear mixed models (GLMM)	647
7.4	Nonlinear mixed models (NLME)	664
7.5	(Generalized) additive mixed models (AMM/GAMM)	685
8	Multivariate data analysis	697
8.1	Multivariate data	697
8.2	Distance metrics: biological (dis)similarity	701
8.3	Clustering	710
8.4	Analyzing dissimilarity	720
8.5	Ordination	732
9	Planning your analysis (what test?)	795
9.1	How to use this guide	795

CONTENTS	5
9.2 What question are you trying to answer?	796
9.3 Testing for a difference in mean or location	797
9.4 Testing for a continuous relationship between two or more variables	798
9.5 Classifying observations	799
9.6 Conclusions	799
Literature Cited	801

Preface

This website accompanies a course I developed in Fall 2021: **Applied Biological Data Analysis**. The course is currently a “special topics” (i.e., seminar), but will hopefully someday evolve into a regular course. The target audience is biology graduate students and advanced undergraduates who need to analyze data for their research projects. If you find the material here helpful, please let me know! Likewise, if you have comments for improvement, I’d love to hear from you.

License and permissions

This work and its content is released under the Creative Commons Attribution-ShareAlike 4.0 license. This means that you are allowed to share the material and adapt it for your purposes, under the following conditions:

1. Give appropriate credit to the author. This includes citing the original sources of any third-party datasets used in some of the tutorials. I’ve done my best to provide appropriate citations alongside these datasets.
2. If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

This site is for educational purposes only. This work and its content is released under the Creative Commons Attribution-ShareAlike 4.0 license. Inclusion of third-party data falls under guidelines of fair use as defined in section 107 of the US Copyright Act of 1976.

Course description

This course is a survey of data analysis skills and statistical methods that are essential for modern biology. The course takes a holistic approach to the data analysis workflow in biology using the open-source environment R, including data management, exploratory data analysis, data modeling, and reproducible science practices. Statistical topics covered include generalized linear models, mixed effects models, non-linear models, and ordination. Students are required

to apply techniques learned in class to real or simulated biological datasets as a course project.

Course objectives

1. Explain the role of statistics in the biological sciences and the ways in which analytical results are communicated.
2. Manipulate, summarize, display, and analyze data using the open-source environment and language R.
3. Use probability distributions to model and think about biological phenomena. Students should come away from the course able to translate biological hypotheses and ideas into statistical statements and vice versa.
4. Conduct exploratory data analyses in support of scientific investigations, particularly to detect and diagnose common problems with biological datasets.
5. Employ modern statistical methods to answer biological questions.
6. Communicate data and analytical results to audiences who may or may not have statistical backgrounds.

Course requirements

- Access to a computer or laptop capable of running R version ≥ 4.0 .
- Having a recent version of RStudio is helpful but not essential.
- A 64-bit system is highly recommended for speed and stability.
- A separate program in which to write and edit code.
 - Most people prefer to use an “integrated development environment” (IDE) or dedicated code editor for coding.
 - IDEs and code editors are often subject to irrationally strong personal and organizational preferences, so just pick the one that works for you. Or, use the one your supervisor tells you to use.
 - My usual workflow includes two windows: the base R GUI, and Notepad++.
- It would helpful to have a dataset of your own to work with. Each course module uses numerous examples with simulated or published datasets. But, at the end of the day, you’re probably here because you need to apply these methods to your own work.

Recommended reading

The primary textbook for the course is Bolker (2008). This is a general statistics textbook for biologists. The applications and examples are focused on ecology, but the statistical exposition is relevant to any area of biology.

Some other general statistics and R books you may find helpful include Dalgaard

(2002) and Zuur et al. (2009b). A more advanced statistics book that also introduces machine learning is James et al. (2013), which is also available for free online. Parts of the course also lean heavily on Zuur et al. (2007), McCune et al. (2002), Legendre and Legendre (2012), and Kéry (2010).

Course organization

This course is organized into *modules* that each focus on one of the objectives of the course. The links below will take you straight to the start of each module.

Module 1 is an introduction to the use of statistics in biology, and explores some fundamental issues related to how biologists make inferences based on data.

Module 2 is an introduction to the R program and programming language.

Module 3 takes a deep dive into manipulating data in R. This includes data import and export, working with special data types such as dates and times, and data frame management.

Module 4 is a survey of common techniques for exploratory data analysis. This is a vital step in the data analysis workflow because it can reveal unexpected patterns and problems in the data.

Module 5 introduces generalized linear models (GLM), an extremely general and powerful framework for analyzing biological data. This module presents worked examples of some of the most common GLMs in biology.

Module 6 explores nonlinear models for relationships that don't follow a straight line. These models include biological growth curves and dose response curves.

Module 7 introduces mixed models, which extend GLM and other models to include random effects. Mixed models allow biologists to account for variability from unknown sources.

Module 8 explores methods for multivariate data analysis, including distance measures and ordination.

Module 9 is a guide to selecting statistics for your research. This module contains dichotomous keys to help you decide what statistical methods are most appropriate for the question you are trying to analyze.

About the author

I'm a quantitative ecologist at Kennesaw State University who studies human impacts on animal populations. Since earning my PhD at Baylor University in 2012, I've had the good fortune to have worked in a variety of government and industry roles. Those experiences have shaped my perspective on biology and education. In my courses and advising I do my best to impress upon students the variety of biological experiences and perspectives that different domains

bring to the table... and the value that they all bring. Most of my current research focuses on understanding human impacts on animal communities and how animal populations adapt to rapidly changing environments.



Why am I making this R guide? I have been using R since about 2008, when I was a graduate student trying to make a multi-panel plot for a manuscript. I had seen R mentioned in publications and thought that using R would be more effective than kludging something together with Excel and Powerpoint. Needless to say, jumping straight to making plots with the `lattice` package was not the easiest way to learn a new programming language. The final result was a little clunky (Kirchner et al. 2011) but I was really proud of it. Eventually I took some courses and did a lot of self-teaching, and am finally (14 years later) mostly decent at R. I have used R in my work pretty much continuously since 2008 for everything from Bayesian data analysis (e.g., Green et al. 2020) to simulation modeling (e.g., Green et al. 2019). This guide is meant to be a primer for self-starting with R. A lot of the explanations that are included are there to hopefully save someone some of the trouble I had getting started.

Chapter 1

Statistics in modern biology

1.1 Overview

Science works by testing ideas to see if they line up with observations. In other words, checking the predictions of hypotheses against *data*. This process is called **data analysis**, because it involves careful examination of numerical patterns in observational or experimental data. The branch of mathematics concerned with analyzing data is called **statistics**. Statistics is vital to modern science in general, including biology.

This website and the course it supports is designed to introduce the ways that statistics are used in biology. This includes strategies for data analysis and an introduction to the open source program and language R. Specific tasks in the data analysis workflow (e.g., data manipulation) and specific analysis methods (e.g., generalized linear models) are covered on individual pages. This first module is a more general exploration of the role of statistics in modern biology, common statistical frameworks, and common misuses of statistics.

1.2 Statistics in modern biology

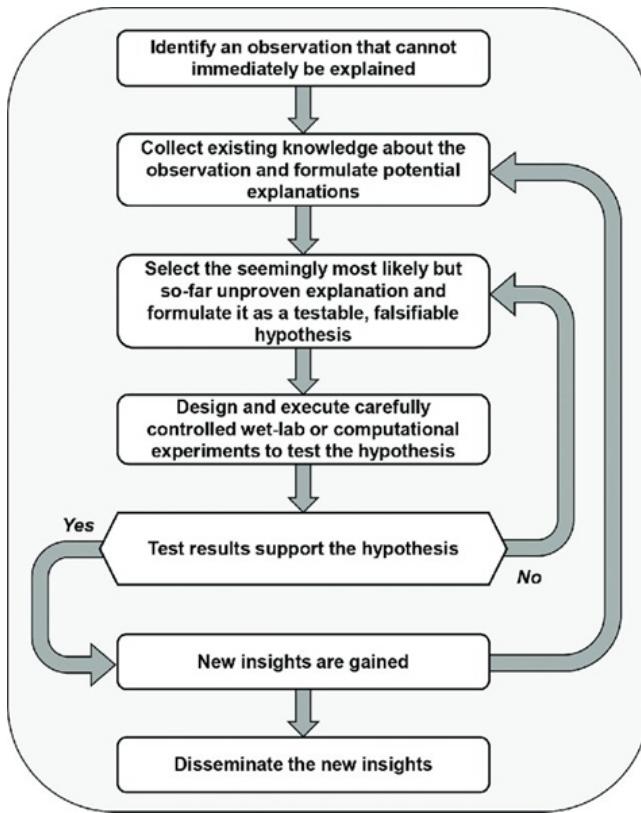
1.2.1 The scientific method

The famous astronomer and popularizer of science Carl Sagan used to say that, “Science is much more than a body of knowledge. It is a way of thinking.” This quote sums up the way that you, as an emerging scientist, should start to think about the way science works. As an undergraduate, you were (or are) primarily a consumer of scientific knowledge. As a graduate student you will start to become a producer of that knowledge. The key step in that transition is to start to view science as a method for finding answers rather than a set of subjects with answers to be memorized.

Biology, and science in general, depends on the **scientific method**. What is the scientific method? In school you probably learned about a series of discrete steps that goes something like this:

1. Observation or question
2. Hypothesis
3. Experiment
4. Conclusions

The reality is a little more complicated. A more realistic representation might be something like this (Voit 2019):



This model includes the key decision point *Test results support the hypothesis*. If this is the case, new insight has been gained which can be shared with the world and which should provoke new questions. If the test results do not support the hypothesis, then there is more work to do before any conclusions can be drawn. Namely, one hypothesis may have been eliminated, so an experiment must be devised to test another candidate explanation.

This course is about that decision process: determining whether biological data support a hypothesis or not. Every step before and after that decision point

depends on your subject matter expertise as a biologist. The logic of the decision process itself, however, is in the realm of statistics. Being a biologist requires being able to use the tools of statistics to determine whether data support or refute a hypothesis. Even if you do not end up working as a biologist, a career in any knowledge field (scientific or otherwise) will be greatly enhanced by this ability.

Many biology majors, including myself as an undergraduate, seem to have a phobia of statistics and mathematics. That's perfectly understandable. In some ways, biology and mathematics use very different mindsets. Biologists tend to revel in the gory details (literally) of anatomy, taxonomy, biochemistry, ecology, behavior, and genetics. These details are often taxon and even situation-specific. Mathematicians, on the other hand, spend a lot of effort trying to abstract those details away and discover general underlying patterns. But, as we will see this semester, as biologists we can benefit a lot from trying to express our ideas in terms of mathematics. Specifically, in the language of statistics, which is the discipline of applied mathematics that is concerned with measuring and interpreting data. The modern discipline of statistics was developed in large part to help answer the kinds of questions that scientists ask¹. In other words, to help make the critical decision about whether test results support a hypothesis or not.

1.2.2 Example data analysis

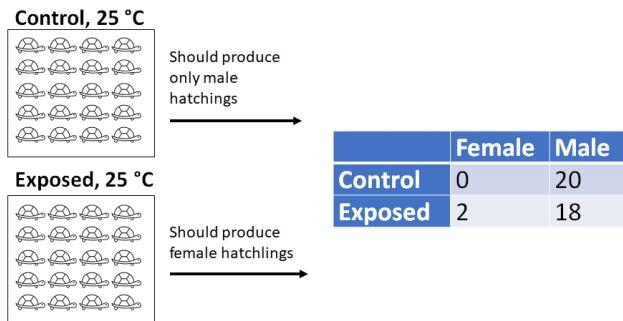
Perhaps the best way to see how statistics are used in biology is to work through the process, paying special attention to how different statistical methods are used. The data and conclusions for this example are mostly fictitious, but the thought process is the same as for any good research project.

Imagine a study where researchers are interested in the effects of agricultural chemicals on turtle reproduction. This study is motivated by anecdotal evidence that baby turtles are not observed in ponds downstream from farms, prompting concern from a state fish and wildlife agency. The researchers hypothesize that a common herbicide, atrazine, acts as an estrogen-antagonist and thus interferes with the sexual maturation of male turtles.

The researchers performed two experiments:

- **Experiment 1:** 40 turtles are raised in the lab: 20 exposed to low levels of atrazine, and 20 not exposed to atrazine. Turtle eggs were incubated at 25 °C, a temperature known to produce only male hatchlings. They then counted the number of male and female hatchlings.

¹Also, gambling.



- **Experiment 2:** 30 ponds are surveyed for turtle nests. The number of nests at each pond and eggs in each nest are counted. Surrounding land use and other environmental variables are also recorded.

Control: farms not using atrazine



Predictions:

1. More nests
2. More eggs/nest

Exposed: farms using atrazine



Predictions:

1. Fewer nests
2. Fewer eggs/nest

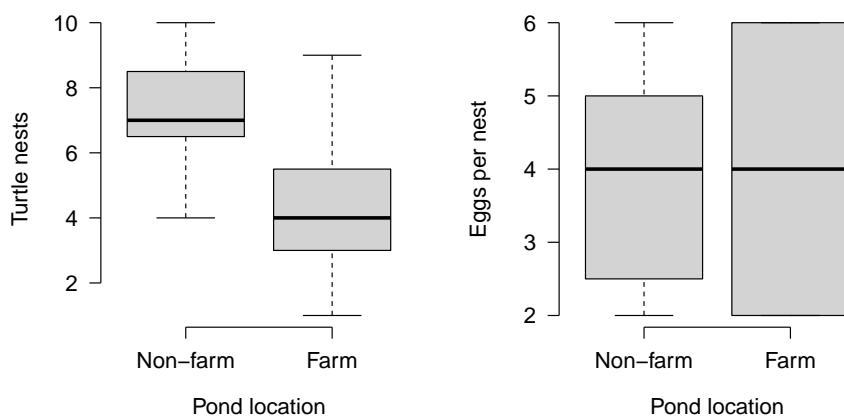
In experiment 1, at the incubation temperature of 25 °C, all embryos should have developed into male hatchlings. The actual results were:

	Female	Male
Control	0	20
Exposed	2	18

The researchers performed a χ^2 test and found that the proportion of hatchlings that were male did not differ significantly between the control and exposed treatments ($\chi^2=0.526$, 1 d.f., $P = 0.4682$). From this they concluded that atrazine does not interfere substantially with male turtle development².

²This imaginary experiment is *very* loosely inspired by a real study (De Solla et al. 2006), with some details changed to facilitate classroom discussion. The results and discussion in this

In the experiment 2, researchers found that ponds near farms tended to have fewer turtle nests. Their data are shown graphically below. The researchers compared the mean number of nests in the farm vs. non-farm ponds and found that, on average, ponds surrounded by farms tended to have about 2.86 fewer nests (95% confidence interval of difference = [1.46, 4.38]). A *t*-test found that this difference was statistically significant ($t = 4.1737$, 27.253 d.f., $P = 0.0002$). When the researchers compared the number of eggs per nest, they found no difference in mean eggs per nest between pond types ($t = 0$, 27.253 d.f., $P > 0.9999$).



So, what can the researchers conclude? From experiment 2, they can conclude that something about being surrounded by farms appears to reduce the number of turtle nests in the ponds, but does not affect the number of eggs per nest. From experiment 1, they can also conclude that atrazine by itself does not appear to affect development of male turtles.

Notice that in the investigation described above, the researchers did not obtain a definitive answer to their question from any single test. Instead, they used their working hypothesis (proposed explanation) to deduce several predictions of results they should have observed if their hypothesis was correct. Here is the thought process more explicitly:

- **Hypothesis:** Atrazine interferes with the sexual maturation of male turtles because it acts as an estrogen-antagonist.
 - **Deduction 1:** If atrazine interferes with an embryo's estrogen receptors, that embryo should not be able to develop as a male. If individual embryos cannot develop as male, then a group of embryos exposed to atrazine should all develop as female.
 - * **Specific prediction based on deduction 1:** A greater proportion of eggs will develop into females in the experimental group

exposed to atrazine than in the group not exposed to atrazine.

- **Deduction 2:** If atrazine reduces turtle reproductive output, then turtle populations exposed to atrazine should have lower reproductive success.
 - * **Specific prediction based on deduction 2:** Habitats surrounded by agricultural fields (and thus exposed to atrazine) should have fewer turtle nests than habitats surrounded by other land use types (and thus not exposed to atrazine).
 - * **Another specific prediction based on deduction 2:** Turtle nests in habitats surrounded by agricultural fields (and thus exposed to atrazine) should have fewer eggs than nests in habitats surrounded by other land use types (and thus not exposed to atrazine).

The χ^2 test was used to evaluate the specific prediction based on deduction 1, because the researchers were comparing proportions. The *t*-tests were used to test the specific predictions of deduction 2, because the researchers were comparing means. What were the results? The table below lays the results out:

Prediction	Correct?
Exposed eggs all develop as female	No
Fewer nests in agricultural landscapes	Yes
Fewer eggs/nest in agricultural landscapes	No

So, what should the researchers conclude? The results suggest that whatever is happening between agricultural fields and turtle nests, it does not appear that pesticides are hampering turtle reproduction. This is seen in the lack of effect of the chemical in experiment 1, and the lack of a relationship between eggs per nest and agricultural land. However, the researchers can't rule out that something about agriculture reduces the likelihood that turtles will build nests. If you were in their position, what would you investigate next?

1.3 Misuses of statistics

There are too many ways to misuse statistics, both intentionally and accidentally, to cover here. In this course we'll explore a few examples that are closely related to this lesson's main ideas. In my experience, there are 4 common ways in which biologists misuse statistics (intentionally or not):

1. Proving the trivial
2. Inappropriate methods
3. *P*-value abuse
4. Inadequate sample size

Let's examine each in turn.

1.3.1 “Proving” the trivial and meaningless hypotheses

As valuable as statistics are, it can be tempting to use them too much. Sometimes we do this from an abundance of caution—after all, we want to be sure about the conclusions we draw! However, there is such a thing as too cautious. For example, consider the two versions of a statement below from a fictional article about the effects of forestry practices on soil nitrogen fixation in a national forest:

Version 1:

All trees were removed by clearcutting between 2 and 4 June 2021.

Version 2:

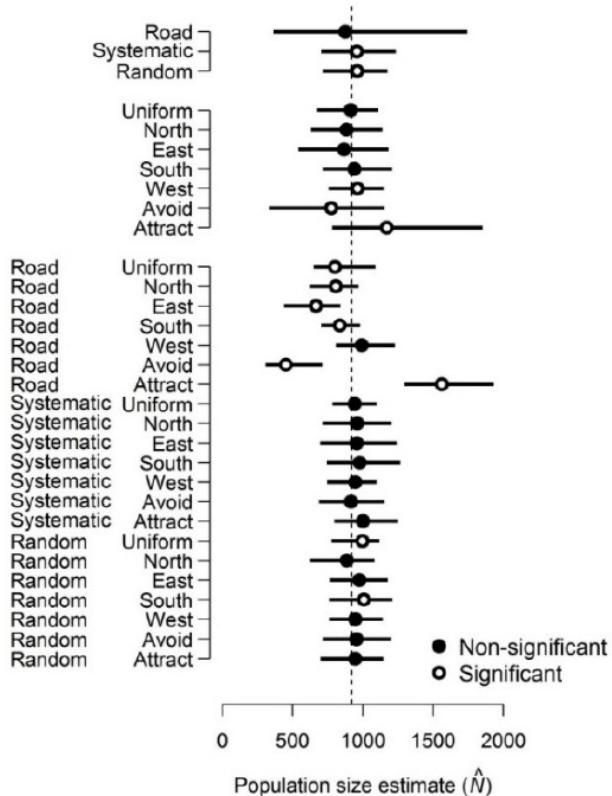
Aboveground woody plant stem count was reduced by 63.1 ± 3.8 trees ha^{-1} days^{-1} (95% CI = [61.5, 64.9] trees ha^{-1} days^{-1}); this effect was statistically significant ($t = 77.56$, 19 d.f., $P < 0.0001$).

Both statements convey essentially the same information. There were some trees in the forest, then trees were removed by clear-cutting, then there were no trees.

The first statement is just fine. The second statement is also fine, in the sense that it is true and because it includes a perfectly legitimate t -test. But, the test reported in the second statement is of a completely trivial question. Of course there were fewer trees after clear-cutting...that's what clear-cutting means! The additional verbiage adds no real information to the paper because the paper wasn't about aboveground woody biomass, but rather about soil N fixation. The reviewers (or editors) would be justified in requiring the removal or shortening of the second statement.

Including unnecessary statistical tests can be tempting in situations where there aren't many significant test results. For example, if your study produces no significant statistical differences related to your main hypothesis, or a marginal effect that is hard to interpret. Padding the manuscript with additional P -values <0.05 can make the work feel more legitimate and on less shaky ground. Resist this temptation.

The logic of null hypothesis significance testing (NHST; see below) sometimes requires us to consider, at least statistically, hypotheses that we know to be false. One such example involves simulation: when data are simulated, it is meaningless to test whether or not two simulations with different parameters were “significantly different” because they are *known* to be different. Consider the example below (Green et al. 2021):



In this study, researchers simulated deer populations and the results of surveying those deer using different survey methods. They then used analysis of variance (ANOVA) to compare the mean population estimates resulting from different methods. Their goal was not to determine whether the population estimates were different between different methods. The estimates were known to be different, because the data were simulated in the first place! What they were trying to do was partition the variance; i.e., measure how much of the difference in estimated population size was attributable to survey method (White et al. 2014). If you are conducting a simulation study, remember that the literal null hypothesis of your statistical test is likely meaningless, or at least, known *a priori* to be false.

1.3.2 Inappropriate methods

Another very common misuse of statistics is simply using statistics incorrectly. Many of these mistakes can be mitigated by better understanding of the methods. In other words, researchers should read the manual. I also like to call this a “problem between chair and keyboard” error³.

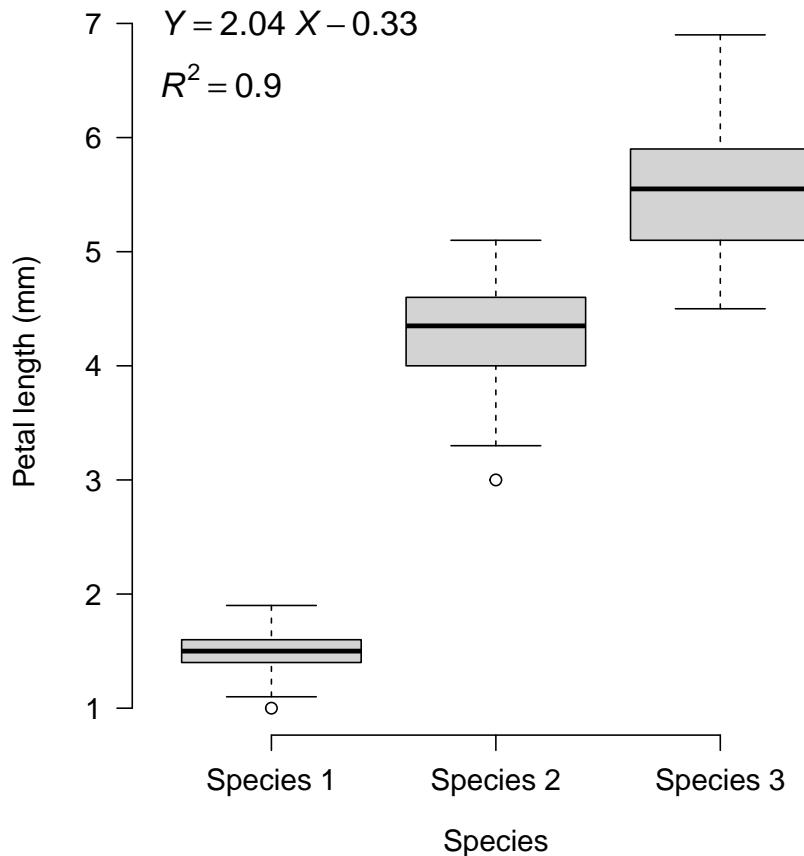
³PBCAK is a very common acronym in IT circles.

1.3.2.1 Common mistake 1: Mischaracterizing a variable

When you perform a statistical test, the variables must have a well-defined type. Some variables are continuous, others are discrete but numerical, and others are categorical. If a variable is treated as the wrong type, it can produce meaningless statistical results. Consider the example below in which the researchers investigated the length of flower petals across 3 species of flower:

```
iris$x <- as.numeric(iris$Species)
summary(lm(Petal.Length~x, data=iris))
```

```
##
## Call:
## lm(formula = Petal.Length ~ x, data = iris)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -1.303 -0.313 -0.113  0.342  1.342
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -0.33200   0.12060 -2.753  0.00664 **
## x            2.04500   0.05582 36.632 < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.5582 on 148 degrees of freedom
## Multiple R-squared:  0.9007, Adjusted R-squared:   0.9
## F-statistic: 1342 on 1 and 148 DF,  p-value: < 2.2e-16
```



See the problem? The dependent variable is modeled as a linear function of the species' identities: literally the numbers 1, 2, and 3. The category labels (species) were treated as numeric values. While the plot shows some clear differences between the species, what if the species had been labeled differently? There is a correct way to analyze these data, but it is not linear regression⁴.

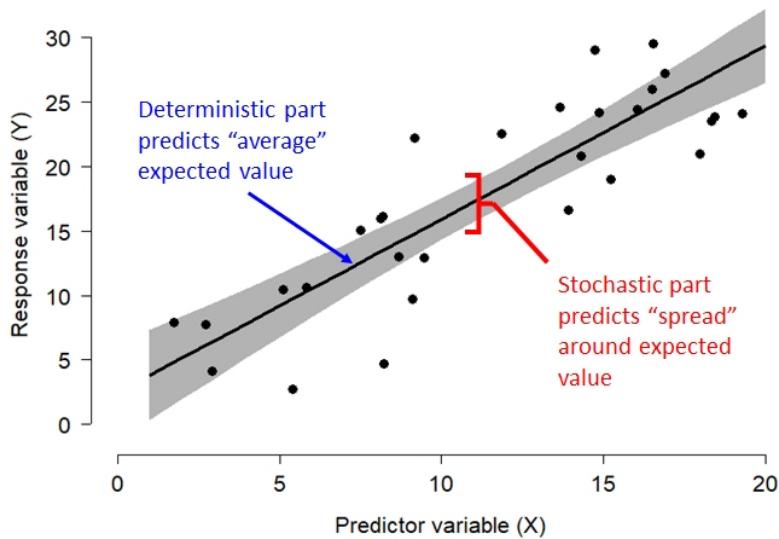
1.3.2.2 Common mistake 2: Overfitting

Overfitting is when a statistical model is fit with more variables than the data can justify. Generally, a model is overfit when *random noise is modeled as if it was part of the deterministic part of the model*. Most statistical models have a deterministic part and a stochastic part. The **deterministic part** predicts the “average” or expected value of the response variable as some function of the explanatory variables. It usually takes the form of an equation (or set of

⁴The best bet is probably analysis of variance (ANOVA). Both linear regression and ANOVA are special cases of **linear models**.

equations). The **stochastic part** of the model describes how observations vary from the expected value predicted by the deterministic part. The stochastic part usually includes one or more probability distributions. We'll discuss common probability distributions in a later module.

The figure below shows the relationship between the deterministic part and stochastic part of a common statistical model, the linear model.



$$\text{Deterministic part: } E(Y_i) = \beta_0 + \beta_1 X_i$$

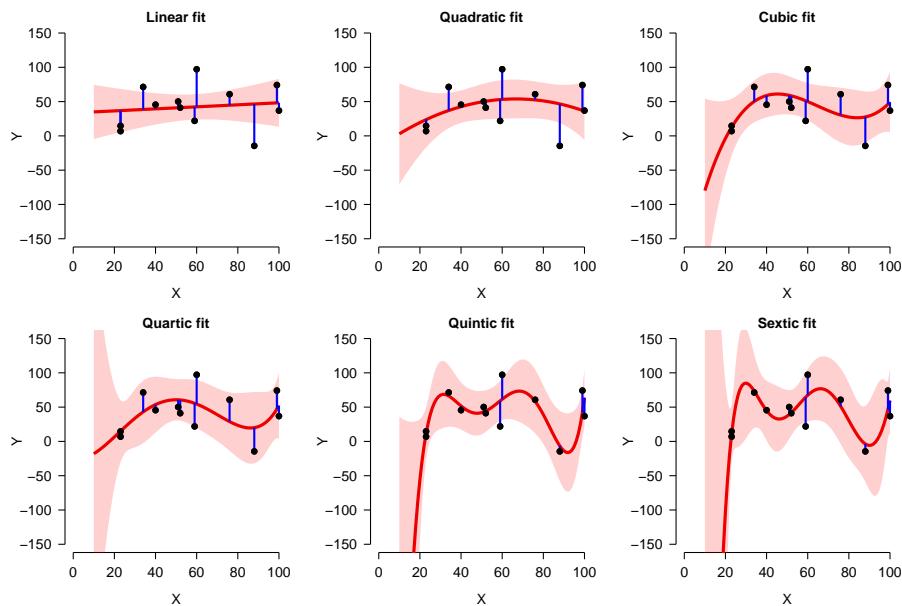
$$\text{Stochastic part: } Y_i \sim \text{Normal}(E(Y_i), \sigma^2)$$

Using computers, it is easy to fit equations to data. In most software it is trivial to add predictor variables to a model until all of the variance is explained (i.e., $R^2 = 1$). However, this is usually not a good idea. There is *always* random noise in any dataset, and the appropriate strategy is always to model that noise rather than try to explain it deterministically. This is because some of that noise is unique to the specific entities or samples that are being sampled. Modeling the noise that is specific to a particular study as if it was representative of all potential studies makes an analysis too parochial to be generally applicable.

Consider the example below. In this plot, 12 data points were generated using a linear model ($Y \sim X$) with normally distributed residuals. Then, various regression models were fit to the data. Each model contained one more term than the last:

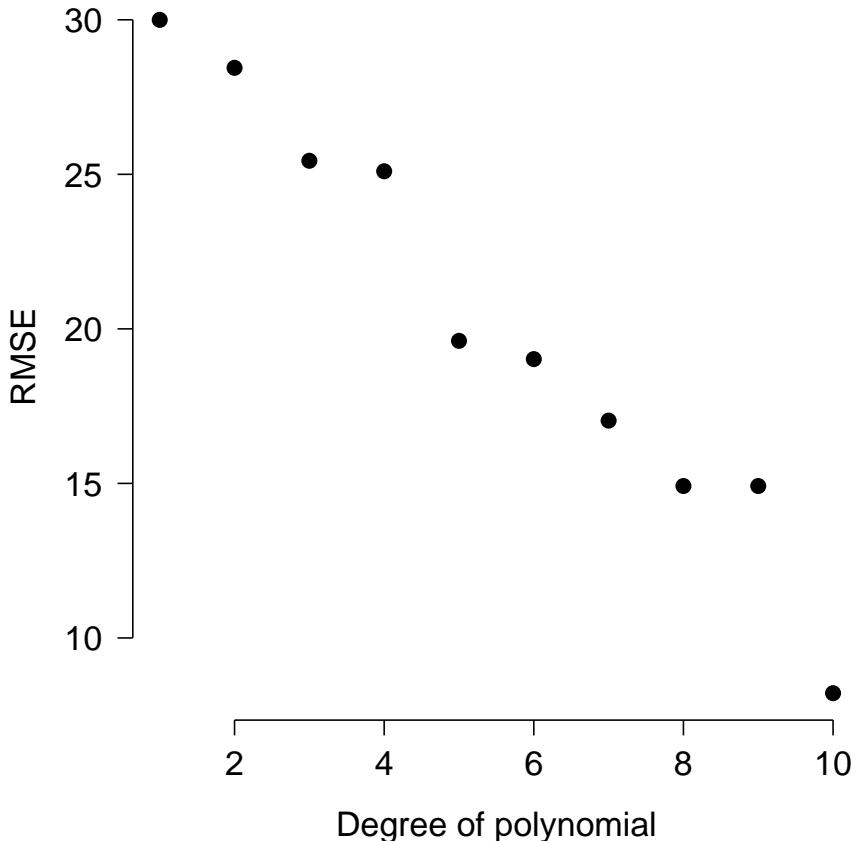
<i>Linear model</i>	$Y = \beta_0 + \beta_1 X$
<i>Quadratic model</i>	$Y = \beta_0 + \beta_1 X + \beta_2 X^2$
<i>Cubic model</i>	$Y = \beta_0 + \beta_1 X + \beta_2 X^2 + \beta_3 X^3$
<i>Quartic model</i>	$Y = \beta_0 + \beta_1 X + \beta_2 X^2 + \beta_3 X^3 + \beta_4 X^4$
<i>Quintic model</i>	$Y = \beta_0 + \beta_1 X + \beta_2 X^2 + \beta_3 X^3 + \beta_4 X^4 + \beta_5 X^5$
<i>Sextic model</i>	$Y = \beta_0 + \beta_1 X + \beta_2 X^2 + \beta_3 X^3 + \beta_4 X^4 + \beta_5 X^5 + \beta_6 X^6$

In this figure, the model prediction and 95% confidence interval (CI) are shown as a red line and red shaded area. The model residuals are shown as blue lines. The original data are shown as black points.



Notice that as the model includes more terms, the fit improves. This can be seen because the residuals are generally getting smaller and the predicted values are getting closer to the observed values.

The figure below shows how the fit improved as more terms were added. This figure shows how root mean squared error (RMSE), a measure of model predictive ability, decreases with increasing model complexity.



Notice what happens as the curves pass through more and more of the points: for X values other than the input points, the curves vary wildly. What's going on here is that the higher-order polynomials are forcing the curves through each point, at the expense of reasonableness between the points. In fact, for any dataset with n observations, you can obtain a perfect fit to the points with a polynomial curve of order n . But, such a curve is highly unlikely to be reasonable because it is not likely to be very applicable to other data. In other words, the model includes terms that are fitting random noise, but do not have general applicability.

A simpler model with fewer terms, and thus greater variance between the data and their predicted values, would likely have more predictive power for new data. Thus, the simpler model might be a better representation of the underlying process. In data science this phenomenon is often called the **bias-variance trade-off** because in general it's easy to minimize prediction error on the training data ("variance") but at the cost of decreasing prediction error on new observations ("bias"). This is because some of the variation in the training

data is actually random noise and thus fitting the model to explain it is really explaining nothing.

1.3.3 *P*-hacking and data dredging

The third misuse of statistics that we'll explore here is that of searching for significant *P*-values, and worrying later whether or not the tests make sense. ***P*-hacking** refers to massaging or adjusting an analysis to produce a *P*-value <0.05 . This can take many forms: post hoc rationalizations for removing certain observations, reframing a question, dropping categories or levels of a factor, and so on. **Data dredging** is a similar kind of practice where a researcher compares many variables or performs many tests, and builds a story out of the tests with $P < 0.05$. Both of these activities can produce interesting results, but the results might not be very meaningful.

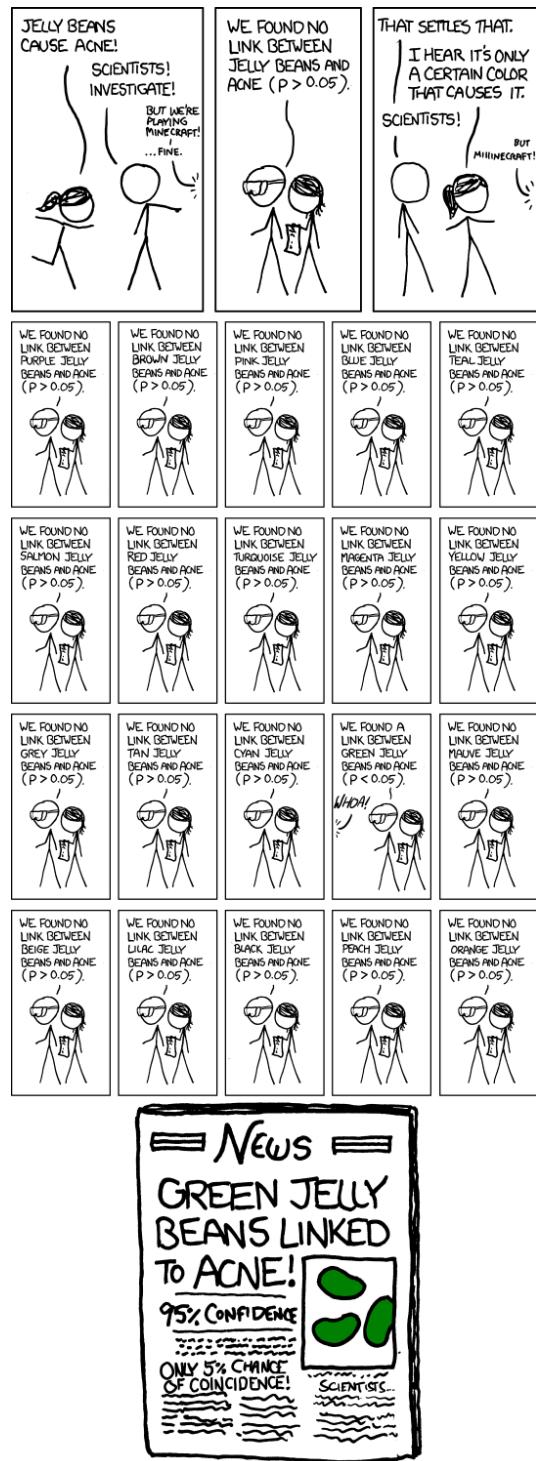
The problems with *P*-hacking are legion, but I'll just point out two.

1. *P*-hacking allows the researcher to commit scientific misconduct, whether intentionally or not. Your objective should be to find the correct answer, not to minimize a *P*-value.
2. *P*-hacking sets you up to commit what's called the **Texas Sharpshooter Fallacy**. This name evokes the story of a (Texan) gunman shooting a barn, then drawing targets around the bullet holes to make it look like he hit his targets. Another expression that describes this is, "Hypothesizing after results known," or HARK.



Data dredging has all of the same problems as P -hacking, but arrives at them a different way. Whereas a P -hacker adjusts the analysis until the P -value is significant, a data dredger just keeps trying new variables until one of them hits. This approach is perhaps best summarized in the following comic⁵, although Head et al. (2015) provide a more rigorous treatment.

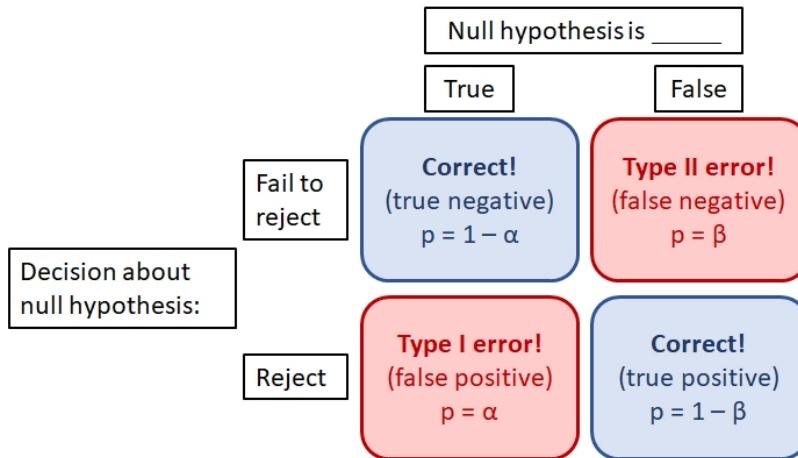
⁵Source: XKCD



1.3.4 Inadequate sample sizes and pseudoreplication

1.3.4.1 Sample sizes and statistical power

Size matters in biology (Smith and Lyons 2013). Size also matters in statistics (Makin and Xivry 2019). As we'll see in the next section, the number of observations in a study can have a large impact on whether or not a pattern is detected in the data. Generally, studies with greater sample sizes can detect smaller effects. Put another way, studies with greater sample sizes are more likely to detect an effect of a given magnitude. On the other hand, studies with smaller sample sizes are less likely to detect an effect. Both of these situations present their own kinds of problems, summarized in the table below.



Ideally we should reject the null hypothesis when it is false, and fail to reject it when it is true. However, data always have some element of randomness in them and thus we can never be perfectly sure that the conclusions of a test are correct. Statisticians have thought about this problem a lot and boiled it down to two probabilities: α and β .

- α is the **type I error rate**, the probability of rejecting the null hypothesis when it is true.
- β is the **type II error rate**, the probability of failing to reject the null hypothesis when it is false.

There is a trade-off between the two error probabilities: decreasing one usually increases the other. By convention, α is usually set to 5% (0.05). P -values are compared to this value α : if $P \geq \alpha$, then we "fail to reject" the null hypothesis. If $P < \alpha$, then we reject the null hypothesis with the understanding that P is the probability of a type I error.

What does all of this have to do with sample size? That probability that a test will correctly reject a false null hypothesis, $1 - \beta$, is known as the **power** of a test. The power of a test tends to increase with larger α , greater sample size,

and larger effect size. The R commands below demonstrate this:

```

set.seed(123)
# type II error:
x1 <- rnorm(10, 5, 2)
x2 <- rnorm(10, 5.2, 2)
t.test(x1, x2)

##
## Welch Two Sample t-test
##
## data: x1 and x2
## t = -0.5249, df = 17.872, p-value = 0.6061
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
## -2.342098 1.406112
## sample estimates:
## mean of x mean of y
## 5.149251 5.617244
# larger sample size:
x3 <- rnorm(1000, 5, 2)
x4 <- rnorm(1000, 5.2, 2)
t.test(x3,x4)

##
## Welch Two Sample t-test
##
## data: x3 and x4
## t = -2.6522, df = 1998, p-value = 0.008059
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
## -0.41274937 -0.06182918
## sample estimates:
## mean of x mean of y
## 5.036722 5.274011
# larger effect size:
x5 <- rnorm(10, 5, 2)
x6 <- rnorm(10, 10, 2)
t.test(x5, x6)

##
## Welch Two Sample t-test
##
## data: x5 and x6
## t = -5.4059, df = 16.404, p-value = 5.348e-05
## alternative hypothesis: true difference in means is not equal to 0

```

```
## 95 percent confidence interval:
## -6.524121 -2.853887
## sample estimates:
## mean of x mean of y
## 5.522159 10.211163
```

In practical terms, this means that conducting a test with too few observations can make it more likely that you will fail to detect a pattern that is really there: a type II error. This happens all the time in biology, and it can be devastating to a research program. To reduce the chances that it happens to you, consider these steps:

- **Increase your sample size.** When in doubt, increase your sample size.
- **Decrease sources of variation** that you can control. Any extraneous variation caused by experimental errors, sloppy technique, improperly trained lab assistants, etc., will reduce the power of your test.
- **Conduct a power analysis.** Power analysis is a set of techniques for estimating the probability of detecting an effect given sample size, effect size, and other factors. It works best when you have some pilot data or comparable data from literature. Power analysis is extremely useful for answering questions like, “How many samples should I collect?” or “What is the relative benefit of collecting 10 more samples or 20 more samples?”. Power analyses for simple methods like *t*-tests or ANOVA are very straightforward; more complex methods may require simulation.

1.3.4.2 Pseudoreplication

Pseudoreplication is a situation where samples are treated as independent when they are not (Hurlbert 1984). This leads to an analysis where the sample size is artificially inflated, and thus the analysis has greater apparent power than it actually does. Pseudoreplication is essentially the opposite problem as inadequate sample size: rather than having fewer observations than needed to do the analysis, the researcher does the analysis as if they have more observations than they really do. The seminal paper on this topic in ecology is Hurlbert (1984)..

To understand pseudoreplication, it’s important to keep in mind the difference between an **observation** and a **degree of freedom**. Degrees of freedom can be thought of as **independent pieces of information**. Ideally, each observation contributes information to the dataset and so a dataset contains as many pieces of information as it does observations. In statistics, however, we find that this is not possible. Every time we calculate a summary statistic or estimate a model parameter, we use information from the dataset. If we know the data and know a statistic such as the mean, then some of the information must be in the statistic and not the dataset. For example, if you have 10 values and calculate the mean, the mean is considered to be known with 9 degrees of freedom. This is because if you know the mean, 9 of the values are free to vary without taking away your

ability to calculate the 10th value given those 9 values and the mean. The R code below illustrates this:

```
set.seed(123)
a <- sample(1:10, 10, replace=TRUE)
a

## [1] 3 3 10 2 6 5 4 6 9 10
(mean(a)*length(a))-sum(a[1:9]) == a[10]

## [1] TRUE
```

Pseudoreplication often occurs when researchers consider multiple measurements from the same sampling unit as different observations. For example, if an ecologist measures tree diameters from multiple trees within 20 m of each other. Or, a microbiologist measures cell counts from multiple assays of the same culture tube. In both cases, there is strong reason to suspect that the multiple values are not independent of each other. In fact, it might be more appropriate to take the average of the multiple values and treat them as a single value (a “measure twice, cut once” approach).

Pseudoreplication can be a serious issue in investigations because most statistical models (e.g., ANOVA) assume that treatments are replicated. This means that each treatment is applied to multiple experimental units (i.e., samples). Applying a treatment or condition to multiple samples allows estimation of variability within a treatment. This variability must be separated from variability between treatments in order for a researcher to be able to infer how much variation is caused by the treatments.

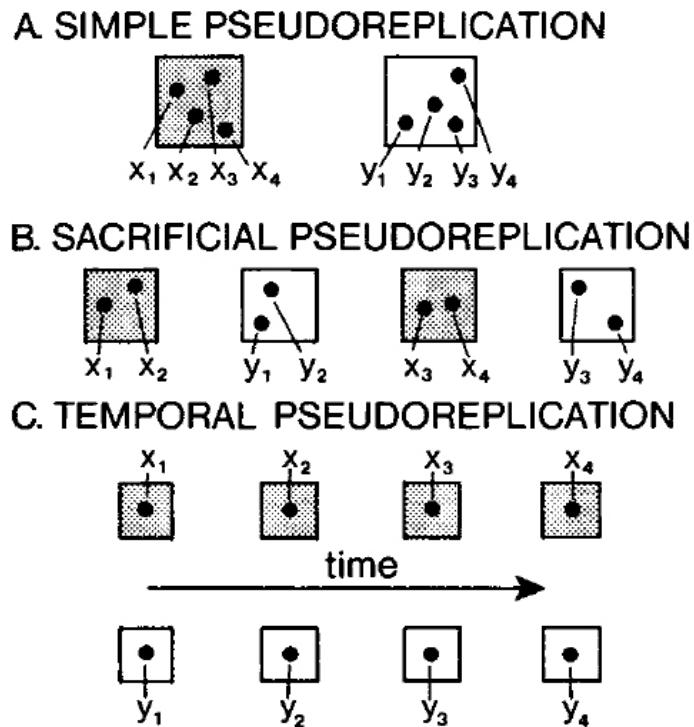
For example, imagine a drug trial where Drug A and Drug B are each given to one person. This trial cannot distinguish whether Drug A or Drug B is more effective, because there is no way to know whether any difference in outcome between the two subjects is due to the drugs or due to differences between people. But, if Drug A and Drug B were each tried on 100 people, the variability among the 100 people who received Drug A and among the 100 people who receive Drug B could be compared to the variability between people who received Drug A or Drug B. This would allow the researchers to make some inferences about how effective each drug was.

The best strategy for mitigating pseudoreplication is to **avoid it**. This requires thinking very carefully about your experimental design and about what the experimental units (i.e., samples) really are. Avoiding pseudoreplication often involves some degree of randomization, in order to break any connection between potentially confounding variables.

If avoiding pseudoreplication is not possible, then you need to **account for it**. One way is to take steps to mitigate confounding variables. For example, you could rearrange plants in a growth chamber each day so that any effects of location within the chamber would be minimized. Another way is to acknowledge

pseudoreplication in your analysis. Some methods such as mixed effects models or autocorrelation structures can account for some pseudoreplication (but you need to be very careful about doing this and very explicit about what you did). Finally, you can be open about the limitations of your study and present your findings as preliminary, or as not conclusive. Note that this strategy, while certainly ego-deflating, does not mean that your work is invalid. Openness and honesty are vital parts of the scientific process. What would be invalid would be ignoring any pseudoreplication in your study and overstating the validity of your results.

Hurlbert (1984) defined four kinds of pseudoreplication: simple, temporal, sacrificial, and implicit. The first 3 is shown in the figure below.

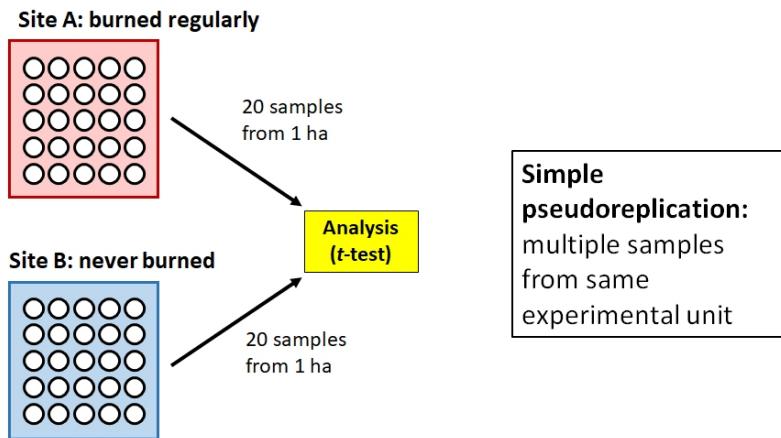


1.3.4.2.1 Simple pseudoreplication Simple pseudoreplication occurs when there is one experimental unit, or replicate, per treatment, regardless of how many samples are taken within each replicate. When this occurs, variability between replicates cannot be separated from variation between treatments.

Example

Barney is interested in the effects of forest management practices on soil microbes. He visits two forest sites: Site A is burned regularly,

and Site B is unmanaged. Barney uses a GIS to randomly identify a 1 ha plot within each site. He then visits each site and takes 20 soil samples from the randomly selected plot. Barney then measures the metabolic profile of each soil microbe sample ($n = 40$) and compares the 20 samples from the burned plot to the 20 samples from the unburned plot.



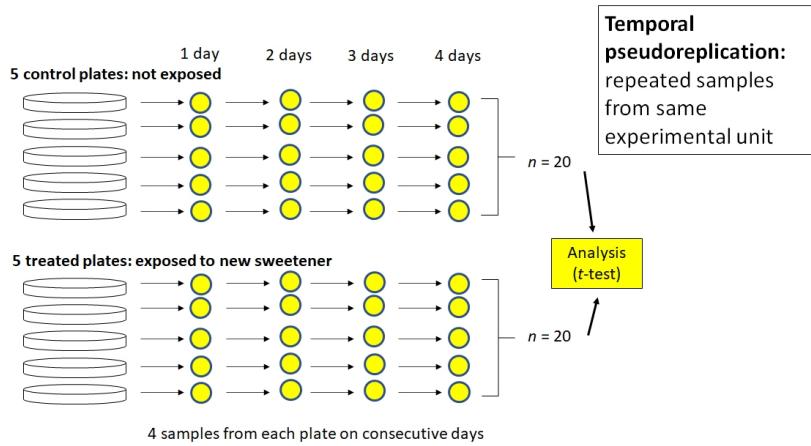
The problem here is that Barney might actually only have 2 samples: one from the burned plot, and one from the unburned plot. If the plots are relatively homogenous, then we should expect their microbiota to be similar simply because of their proximity and shared environment. So, the proper experimental unit is the plot, not the soil sample. Barney has collected a dataset with simple pseudoreplication because he confused taking multiple measurements within an experimental unit with taking measurements at multiple experimental units. In other words, his “replicates” are not true replicates. Barney’s thesis advisor tells him to sample more sites and to investigate mixed models and block designs for his analysis.

1.3.4.2.2 Temporal pseudoreplication Temporal pseudoreplication occurs when multiple samples from an experimental unit are taken over time and then treated as statistically independent. Because successive samples from the same experimental unit are likely to be correlated with each other, they cannot be treated as independent.

Example

Betty is interested in the effects of a new sugar substitute on the growth of gut bacteria. She inoculates 10 plates with *E. coli* bacteria and gives the new sweetener to 5 of them. She then counts the number of colony forming units (CFU) on each plate at 1, 2, 3, and 4 days after exposure. Betty analyzes her data using a t-test, with 20 exposed samples (5 plates \times 4 sampling occasions) and 20 control

samples.

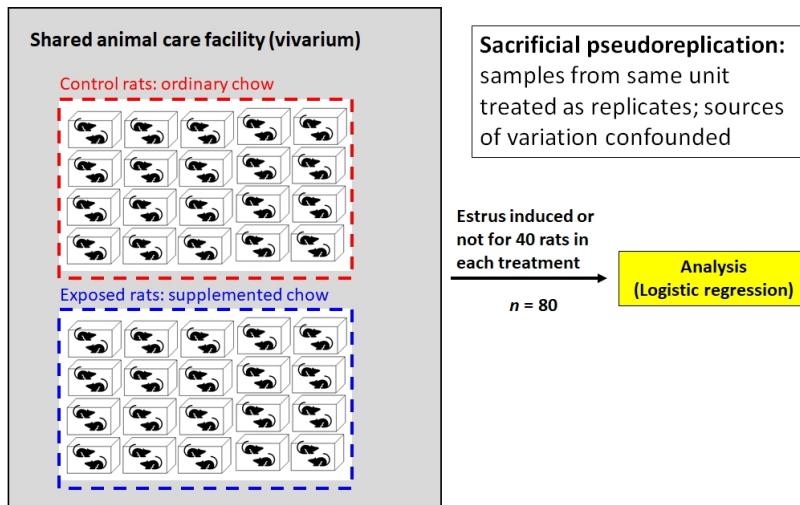


Betty's problem is like Barney's problem, but with time instead of space. The subsequent samples from each plate are not independent of each other. For example, the number of CFU in plate A at 2 days was at least partially dependent on the number of CFU in plate A at 1 day. Using consecutive samples from the same plates does not increase the number of samples in a test of the effect of the sweetener. Instead, Betty introduced a new variable, time, into the system that added variation instead of increasing sample size. She could account for the temporal pseudoreplication in her analysis by including time as a covariate or using some sort of autocorrelation structure. If she does not include time in her analysis, however, she's going to have a bad time.

1.3.4.2.3 Sacrificial pseudoreplication Sacrificial pseudoreplication occurs when data for replicates are pooled prior to analysis, or where two or more observations from the same experimental unit are treated as independent replicates. The term "sacrificial" comes from the fact that information about variation among replicates is confounded with variation among samples within replicates, and thus "lost".

Example

Fred is studying the onset of estrus in female cotton rats (*Sigmodon hispidus*) and suspects that it may be driven by exposure to a compound produced by certain grasses in the spring. He places 80 female rats in 40 cages (2 per cage). Rats in 40 cages are fed ordinary rodent chow, while the rats in the other cages are fed chow supplemented with the suspect compound. Rats are checked each day for signs of estrus. Clearly, each treatment (control vs. supplemented) is replicated. How many times? Fred tells his advisor that each treatment is replicated 40 times, because 40 rats received each treatment.



Not so fast. Mammalogists know that estrus can be induced in female rodents by exposure to airborne chemical cues (pheromones) from other females already in estrus. So, if there are two females in one cage and one goes into estrus (no matter the reason), the other animal might go into estrus in response to the first animal's pheromones and not at all in response to its diet. This means that the two animals in each cage cannot be considered statistically independent. So, Fred has at most 20 replicates of each treatment, not 40.

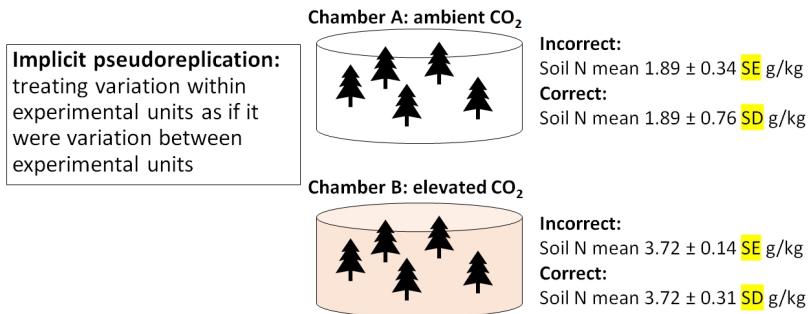
But it gets worse for Fred. The vivarium is old and cramped and poorly ventilated. All 40 cages are kept on a single shelving unit in a single room, and the air always smells like fouled rodent cages. Are the 40 cages really independent of each other, if chemical signals can waft between cages? So, depending on the ventilation in the room, Fred might have 0 replicates because none of the animals are statistically independent of each other. Fred's thesis defense does not go well.

1.3.4.2.4 Implicit pseudoreplication The last kind of pseudoreplication, implicit pseudoreplication, occurs when researchers present means and standard errors (SE) (or 95% CI) for observations *within* experimental units. Doing so implicitly tests for a difference between units, effectively getting a significance test for free without actually performing one! Meaningful means and SE should be calculated *between* experimental units, not within units. Variability within experimental units should be expressed using standard deviation, not standard error.

Example

Wilma is studying soil nutrient content in forests under different climate change scenarios. She grows 5 pine tree seedlings in growth chamber A under atmospheric CO₂ (400 ppm) and 5 seedlings in

growth chamber B under elevated CO₂ (600 ppm). She then calculates the mean and SE of soil N content from the 5 seedlings in each chamber. In chamber A, soil N concentration was 1.89 ± 0.34 SE g/kg; in chamber B, N concentration was 3.72 ± 0.14 g/kg. Because the mean soil N concentration was greater in chamber B, and the means \pm SE did not overlap, she concluded that growing pine seedlings in elevated CO₂ increased soil N concentration.



Not quite. Wilma summarized the uncertainty surrounding soil N measurements, which was a good idea. But, she presented the uncertainty as standard error, not standard deviation. Variation within each chamber should be calculated as SD. The SE would be appropriate for describing uncertainty about the difference in means between chambers. By presenting SE instead of SD, Wilma made it look like there was much less variability within treatments than there really was. As a rule, you should always calculate SD. SE usually comes up as part of the results of a statistical test.

1.3.4.3 Sample size and pseudoreplication summary

The problems related to statistical power are not new in biology. Button et al. (2013) reviewed some causes and consequences of low power in neuroscience. Lazic et al. (2018) describe some of the difficulties with defining sample sizes in cell culture and animal studies, and the resulting issue of pseudoreplication. In ecology and environmental biology, Hurlbert (1984) is the seminal reference on sample size and pseudoreplication.

1.4 P-values and null hypothesis significance testing (NHST)

1.4.1 Definition

When scientists design experiments, they are testing one or more hypotheses: proposed explanations for a phenomenon. Contrary to population belief, the statistical analysis of experimental results usually does not address the researchers'

hypotheses directly. Instead, statistical analysis of experimental results focuses on comparing experimental results to the results that might have occurred if there was no effect of the factor under investigation. Some terms that you need to understand:

- **Hypothesis:** proposed explanation of a phenomenon
- **Null hypothesis:** default assumption that some effect (or quantity, or relationship, etc.) is zero and does not affect the experimental results
- **Alternative hypothesis:** assumption that the null hypothesis is false

In (slightly) more concrete terms, if researchers want to test the hypothesis that some factor X has an effect on measurement Y , then they set up an experiment comparing observations of Y across different values of X . They would then have the following set up:

- **Null hypothesis:** Factor X does not affect Y .
- **Alternative hypothesis:** Factor X does affect Y

Notice that in NHST the alternative hypothesis is the **logical negation** of the null hypothesis, not a statement about another variable altogether (e.g., “Factor Z affects Y ”). This is an easy mistake to make and one of the weaknesses of NHST: the word “hypothesis” in the phrase “null hypothesis” is not being used in the sense that scientists usually use it. This unfortunate choice of vocabulary is responsible for generations of confusion among researchers⁶.

In order to test their “alternative hypothesis”, what researchers usually do is test the null hypothesis. Statistical evidence that the null hypothesis can be rejected is then taken as evidence in favor of the alternative hypothesis. The null hypothesis is usually translated to a prediction like one of the following:

- The mean of Y is the same for all levels of X .
- The mean of Y does not differ between observations exposed to X and observations not exposed to X .
- Outcome Y is equally likely to occur when X occurs as it is when X does not occur.
- Rank order of Y values does not differ between levels of X .

Notice that all these predictions have something in common: Y is independent of X . That’s what the null hypothesis means. To test the predictions of the null hypothesis, we need a way to calculate what the data might have looked like if the null hypothesis was correct. Once we calculate that, we can compare the data to those calculations to see how “unlikely” the actual data were. Data that are sufficiently unlikely under the assumption that the null hypothesis is correct—a “significant difference”—are a kind of evidence that the null hypothesis is false⁷.

This kind of analysis is referred to as **Null Hypothesis Significance Testing (NHST)** and is one of the most important paradigms in modern science.

⁶Not to mention among generations of students taking statistics courses.

⁷If this sounds convoluted, that’s because it is.

Whether or not this is a good thing is a subject of fierce debate among scientists, statisticians, and philosophers of science. For better or worse, NHST is all but the de facto standard of inference in scientific publishing, so regardless of its merits in your particular case you will need to be able to use it. The purpose of this course module is to encourage you to use it correctly.

1.4.2 History and status of *P*-values

The *P*-value is a statistical statement about the likelihood of observing a pattern in data when the null hypothesis is true. They were first developed in the 1700s, and eventually popularized by the statistician R.A. Fisher in the 1920s. In Fisher's formulation, the *P*-value was only meant as a "rule-of-thumb" or **heuristic** for determining whether the null hypothesis could be safely rejected. His original publications did not imply that rejecting the null hypothesis should automatically imply acceptance of the alternative hypothesis, or offer any justification for particular cut-offs or critical values of *P*. These ideas came later. Fisher suggested a critical *P*-value of 0.05 for convenience, but did not prescribe its use. In fact, he also suggested using other cut-offs, and even using the *P*-value itself as a measure of support for a hypothesis. The adoption of 0.05 as a threshold for significance proved popular, and has been taken as a kind of gospel for generations of researchers ever since.

In my opinion, the ubiquity of *P*-values on scientific research illustrates Goodhart's Law: "When a measure becomes a target, it ceases to be a good measure". Because the cut-off value of $P < 0.05$ is so widely used and accepted as a measure of the "truth" of a hypothesis, many researchers will take steps to ensure that their experiments produce P -values < 0.05 . This can lead to the practices of *P*-hacking or data dredging. Or worse, setting up experiments to test meaningless null hypotheses or analyzing data selectively. The attention and effort spent increasing the likelihood of a significant outcome is attention and effort that is not spent on formulating hypotheses, developing theory, or conducting experiments. Thus, intensive focus on *P*-values can be a real distraction from more important things.

These criticisms of *P*-values are neither new nor little known. Criticisms of the uncritical devotion to *P*-values can be found going back decades in the literature. In ecology, for example, debates about the place of NHST flare up occasionally, as new advances in statistical methods have offered opportunities for a new inferential paradigms.

Recently, the American Statistical Association (ASA) published a number of papers criticizing the current use of *P*-values and offering alternatives. Wasserstein and Lazar (2016) provide a good starting point to the series and an overview of the major themes. Some of the proposed alternative or complementary methods have started to make their way into various scientific disciplines. Others have not. By and large, NHST remains the standard method of testing hypotheses with data and will likely remain so for a long time.

1.4.3 Where P -values come from

So where do P -values come from, anyway? Remember, P -values are an expression of how likely the observed data would be if the null hypothesis was true. The actual calculation of a P -value involves assumptions about the size of the effect (i.e., difference between the results if the null is true vs. not true), the variability of the outcome (random or otherwise), and the distribution of different summary statistics of the results assuming that the null hypothesis is true. Let's consider the example of a t -test. A t -test is used to compare means between two groups. We will use R to simulate some data and explore a basic t -test.

```
set.seed(123)
n <- 10
mu1 <- 6
mu2 <- 8
sigma <- 3
x1 <- rnorm(n, mu1, sigma)
x2 <- rnorm(n, mu2, sigma)
t.test(x1, x2)

##
## Welch Two Sample t-test
##
## data: x1 and x2
## t = -1.796, df = 17.872, p-value = 0.08941
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
## -5.2131465 0.4091686
## sample estimates:
## mean of x mean of y
## 6.223877 8.625866
```

The results show us that the means of groups `x1` and `x2` are not significantly different ($t = -1.796$, 17.87 d.f., $P = 0.0894$). Because $P \geq 0.05$, we cannot reject the null hypothesis of no difference; i.e., the hypothesis that the means of `x1` and `x2` are the same.

But why? Where do the numbers in the output come from?

The t value is what's called a **test statistic**. This is a summary of the quantity of interest—difference in means—that depends on the quantity of interest *and* additional terms that describe how that summary might vary randomly. In the case of the t -test, the test statistic t is calculated as:

$$t = \frac{\bar{X}_1 - \bar{X}_2}{\sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}}$$

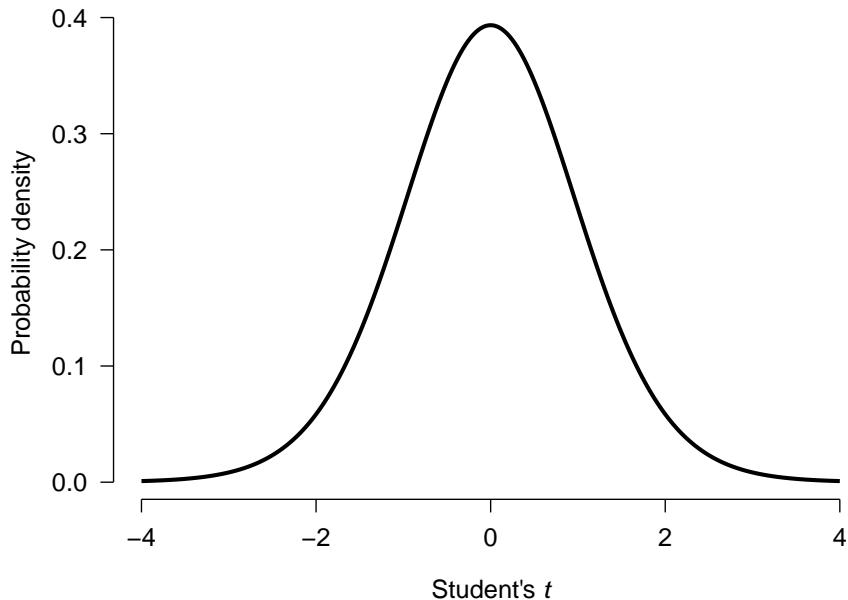
In this equation:

1.4. P-VALUES AND NULL HYPOTHESIS SIGNIFICANCE TESTING (NHST)39

- \bar{X}_1 and \bar{X}_2 are the means of x_1 and x_2
- s_1^2 and s_2^2 are the variances in x_1 and x_2 . The **variance** is a measure of how spread out values are away from the mean.
- n_1 and n_2 are the number of observations in x_1 and x_2

The numerator is the quantity we are interested in: the difference between the means in group 1 and group 2. The denominator scales the difference according to the sample size and the amount of variation in both groups. Thus, t accounts for the quantity of interest (the pattern), the variability in the pattern, and the number of observations of the pattern.

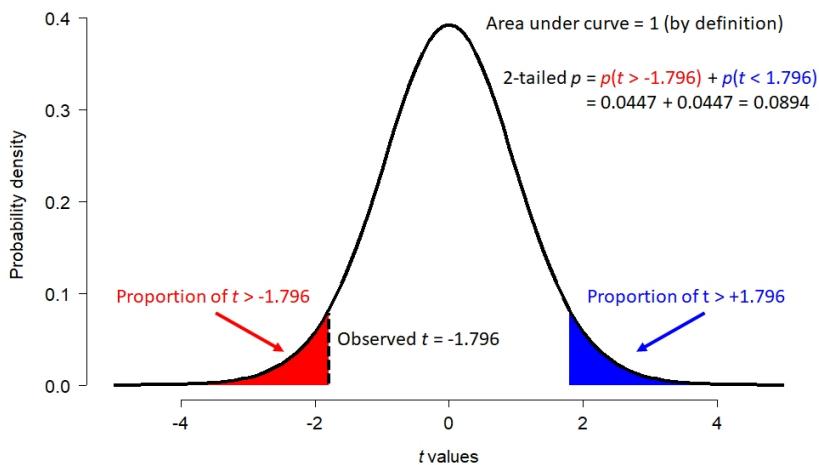
In the example above, the t statistic was -1.796. This was the difference in means (-2.402) scaled by the variability in means given the variability in observations. The “significance” of the test comes from the fact that t -statistics themselves follow a pattern called the **Student’s t distribution**. The Student’s t distribution describes how likely different values of t are given a certain sample size. This sample size used is not the total number of observations, but instead a scaled version of sample size called **degrees of freedom (d.f.)**. For the example above with 17.872 d.f., the t distribution looks like this:



If there was no difference between the means of x_1 and x_2 , then t statistics should cluster around 0 with more extreme values being much less common. Test statistics ≥ 4 or ≤ -4 almost never happen.

What about our t statistic, -1.796? The figure below shows how the total possible

space of t statistics (given 17.872 d.f.) is partitioned. The total area under this curve, called a “probability density plot”, is equals 1. If you think about it, t must take on a value... so the sum of probabilities for all t is 1. The red area shows the probability of a t occurring randomly that is < -1.796 . That is, more extreme than the observed t . But “more extreme” can work the other way too: t could randomly be > 1.796 as well. This probability of this possibility is shown by the blue area. The total of the red and blue areas is the total probability of a t value more extreme than the observed t .



But as we shall see, increasing the sample size can make the same difference appear significant:

```
set.seed(123)
n <- 1000
mu1 <- 6
mu2 <- 8
sigma <- 3
x1 <- rnorm(n, mu1, sigma)
x2 <- rnorm(n, mu2, sigma)
t.test(x1, x2)

##
## Welch Two Sample t-test
##
## data: x1 and x2
## t = -15.485, df = 1997.4, p-value < 2.2e-16
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
## -2.342319 -1.815705
## sample estimates:
## mean of x mean of y
```

```
## 6.048384 8.127396
```

We can also achieve significance by reducing the amount of variation (`sigma`, or σ). Here variance within each group is expressed as the residual standard deviation (SD). **Residual** variation is variation not explained by the model (i.e., not related to the difference in means between groups). The SD is the square root of the variance, and is how R expresses the variation in a normal distribution (`rnorm()`).

```
set.seed(123)
n <- 10
mu1 <- 6
mu2 <- 8
sigma <- 0.001
x1 <- rnorm(n, mu1, sigma)
x2 <- rnorm(n, mu2, sigma)
t.test(x1, x2)

##
## Welch Two Sample t-test
##
## data: x1 and x2
## t = -4486.7, df = 17.872, p-value < 2.2e-16
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
## -2.001071 -1.999197
## sample estimates:
## mean of x mean of y
## 6.000075 8.000209
```

So, what the P -value in our t -test tells us depends on not just the actual difference, but how variable that difference is and how many times we measure it (metaphorically speaking).

The table below summarizes the results of the three simulations we just ran:

True difference $(\bar{x}_1 - \bar{x}_2)$	Residual SD	Sample size	P	Significant?
-2	3	20	0.0894	No
-2	3	2000	<0.0001	Yes
-2	0.001	20	<0.0001	Yes

1.4.4 What P -values mean and do not mean

In the simulations above, we know that the true difference was -2 because that is how we programmed the simulations. But the conclusion of the experiments

depended on experimental parameters other than the one of interest! For each of the three scenarios above, consider these questions:

- Should the researchers reject the null hypothesis, and conclude that there is a difference between x_1 and x_2 ?
- Should the researchers accept the null hypothesis, and conclude that there is no difference between x_1 and x_2 ?
- Should the researchers accept the alternative hypothesis, and conclude that there is a difference between x_1 and x_2 ?

The researchers should reject the null hypothesis only in scenarios 2 and 3. In scenario 1, they should not accept the null hypothesis. The correct action is **fail to reject** the null. Accepting the null hypothesis would imply some certainty that the difference was 0. But that is not what the t -test actually says. What the t -test really says is that a difference (measured as t) between -1.796 and 1.796 is likely to occur about 91% (i.e., $1-P$) of the time even if the true difference is 0. In other words, if we repeated the experiment, we would get a $|t| > 1.796$ over 9 times out of 10!

One of the most common misconceptions about P -values is that they represent the probability of the null hypothesis being true, or of the alternative hypothesis being false. Neither of these interpretations is correct. Similarly, many people think that 1 minus the P -value represents the probability that the alternative hypothesis is true. This is also incorrect. Remember:

The P -value is only a statistical summary, and says nothing about whether or not a hypothesis is true.

The P -value is only a statement about how unlikely the result would be if there was no real effect.

A P -value is a purely statistical calculation and says nothing about biological importance or the truth of your hypothesis. Interpreting P -values and other statistical outputs is the job of the scientist, not the computer.

P -values and NHST in general is that each test can only ever make binary distinctions:

- Reject or fail to reject the null.
- True difference is 0 or not 0.
- Difference is ≥ 3 or not ≥ 3 .
- And so on.

In each example, each of the two alternatives is the logical negation of the other. But, *rejecting* one side of each dichotomy is *not the same as accepting* the other. By analogy, juries in criminal trials in the US make one of two determinations: guilty or not guilty. A verdict of not guilty includes innocent as a possibility, but does not mean we can conclude the defendant is innocent. Likewise, the P -value might help distinguish “difference in means is different from 0 ($P < 0.05$)” vs. “difference in means is not different from 0 ($P \geq 0.05$)”. The latter

choice includes a difference in means of 0 as a possibility, but does not actually demonstrate that the value is 0.

1.4.5 Do you need a *P*-value?

Now that we have a better handle on what a *P*-value really is, you might be tempted to ask, “Do I need a *P*-value?” Yes, you probably do. For better or for worse, *P*-values and NHST are firmly entrenched in most areas of science, including biology. Despite its shortcomings, the NHST paradigm does offer a powerful way to test and reject proposed explanations. This utility means that NHST and *P*-values are not going away any time soon. The rest of this module will explore some alternatives to NHST.

1.5 Alternatives to NHST

1.5.1 Bayesian inference

Bayesian inference is a framework for evaluating evidence and updating beliefs about hypotheses based on evidence. When conducting Bayesian inference, researchers start with some initial idea about their model system: a **prior**. They then collect evidence, and update their idea based on the evidence. The updated idea is called the **posterior**.

- When the prior is strong, or the evidence weak, their ideas will not be updated much and the posterior will be strongly influenced by the prior.
- When the evidence is strong, or the prior weak, the posterior will be updated more and thus less influenced by the prior.

Some researchers prefer to use naïve or uninformative priors, so that their conclusions are influenced mostly by the evidence. Bayesian statistical methods will return essentially the same parameter estimates (e.g., differences in means or slopes) as frequentist methods when uninformative priors are used. This means that almost any traditional statistical analysis can be replaced with an equivalent Bayesian one. The statistical models that are fit, such as linear regression or ANOVA, are the same. All that really changes is how the researcher interprets the relationship between the model and the data.

Bayesian inference depends on **Bayes’ theorem**:

$$p(H|E) = \frac{P(E|H) P(H)}{P(H) P(E|H) + P(\neg H) P(E|\neg H)} = \frac{P(E|H) P(H)}{P(E)}$$

where

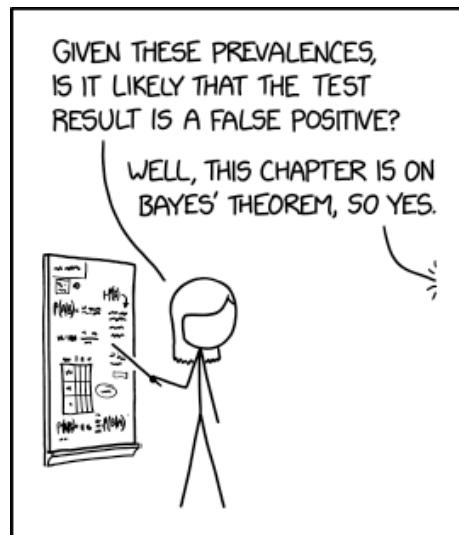
- H is a hypothesis that can be true or false
- $p(H)$ is the prior, an estimate of the probability of the hypothesis being true before any new data (E) are observed

- E is evidence, specifically new data used to update the prior
- $p(H|E)$ is the probability of H given E . I.e., the probability of the hypothesis being true after new evidence is observed. This is also called the **posterior probability**.
- $p(E)$ is the probability of E regardless of H . I.e., the probability of observing the evidence whether or not the hypothesis is true. $p(E)$ is called the **marginal likelihood**.
- $p(E|H)$ is the probability of observing the evidence E given H . I.e., the probability of observing the evidence if the hypothesis is true. $p(E|H)$ is also called the **likelihood**.

Bayes' theorem is powerful because it is very general. The precise nature of E and H do not matter. All that matters is that one could sensibly define an experiment or set of observations in which E might depend on H and vice versa.

1.5.1.1 Example Bayesian analysis—simple

Imagine you go to the doctor and get tested for a rare condition, Niemann-Pick disease. Your doctor tells you that this disease affects about 1 in 250000 people. Unfortunately, you test positive. But, the doctor tells you not to worry because the test only 99% reliable. Why is the doctor so sanguine, and what is the probability that you have Niemann-Pick disease⁸?



The probability of having this disease can be estimated using Bayes' theorem.

⁸Source: XKCD

First, use what you know to assemble the terms:

- The overall incidence is 1 in 250000 people, or 0.0004%. So, the prior probability $P(H)$ is 0.000004.
- The probability of a positive test given that you have the disease, $P(E|H)$, is 0.99 because the test is 99% reliable.
- The denominator, $P(E)$, is a little trickier to calculate. What we need is the unconditional probability of a positive test. This probability includes two situations: either a positive test when someone has the disease, or a positive test when someone doesn't have the disease. Because of this, the denominator of Bayes rule is often rewritten as:

$$P(E) = P(H)P(E|H) + P(\neg H)P(E|\neg H)$$

Where $P(\neg H)$ is the probability of H being false (\neg means “negate” or “not”). If you think about it, $P(\neg H)$ is just $1 - P(H)$, and $P(E|\neg H)$ is just $1 - P(E|H)$.

Next, plug the terms into Bayes' rule:

$$\begin{aligned} p(H|E) &= \frac{P(E|H)P(H)}{P(H)P(E|H) + P(\neg H)P(E|\neg H)} \\ p(H|E) &= \frac{(0.99)(0.000004)}{(0.000004)(0.99) + (0.999996)(0.01)} \\ p(H|E) &= \frac{0.00000396}{0.01000392} = 0.000396 \end{aligned}$$

That's not very worrying! In this example, the test evidence can't overcome the fact that the disease is so rare. In other words, even among people who get a positive test, it's far more likely that you don't have the disease but got a false positive than that you have the disease and got a true positive. This is an example of a very strong prior being more influential than the evidence.

There's an issue with this calculation, though. The prior probability of 0.000004 was based on the assumption that the test was conducted on a random person from the population. Do physicians conduct tests for vanishingly rare diseases on random people? Of course not. Maybe this physician only prescribes this test if they think that there is a 10% chance that you really have the disease. This might be because, in their experience, 10% of people with your symptoms turn out to have the disease. In that case, the calculation becomes:

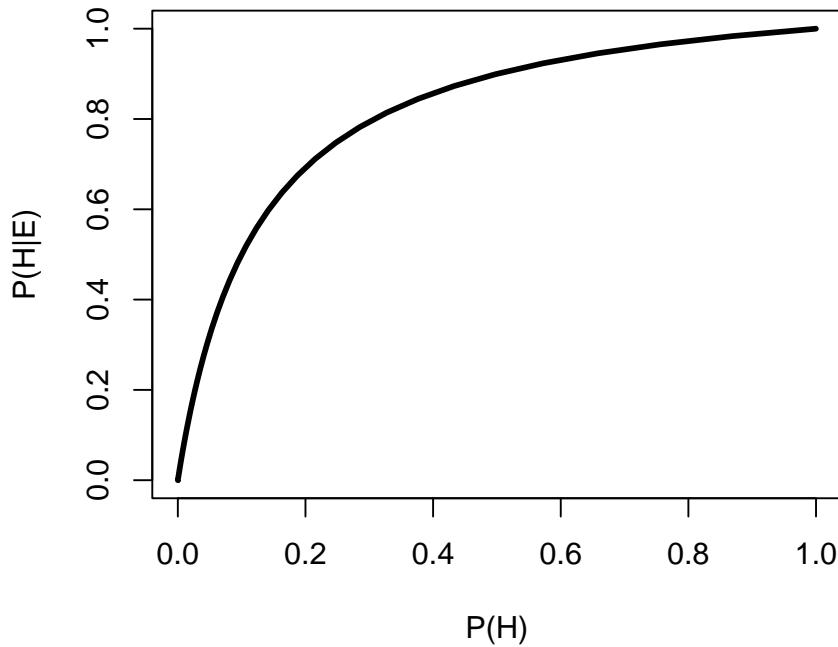
$$\begin{aligned} (H|E) &= \frac{(0.99)(0.1)}{(0.1)(0.99) + (0.9)(0.01)} \\ p(H|E) &= \frac{0.099}{0.108} = 0.916 \end{aligned}$$

91.6% is a lot more worrying than 0.0004%⁹. This example shows that the prior probability can have a huge effect on the posterior probability. Below is an R function that will calculate and print out posterior probabilities conditional on a fixed $P(H)$ and varying $P(E|H)$, or on a fixed $P(E|H)$ and varying $P(H)$. One and only one of the parameters must have more than 1 value.

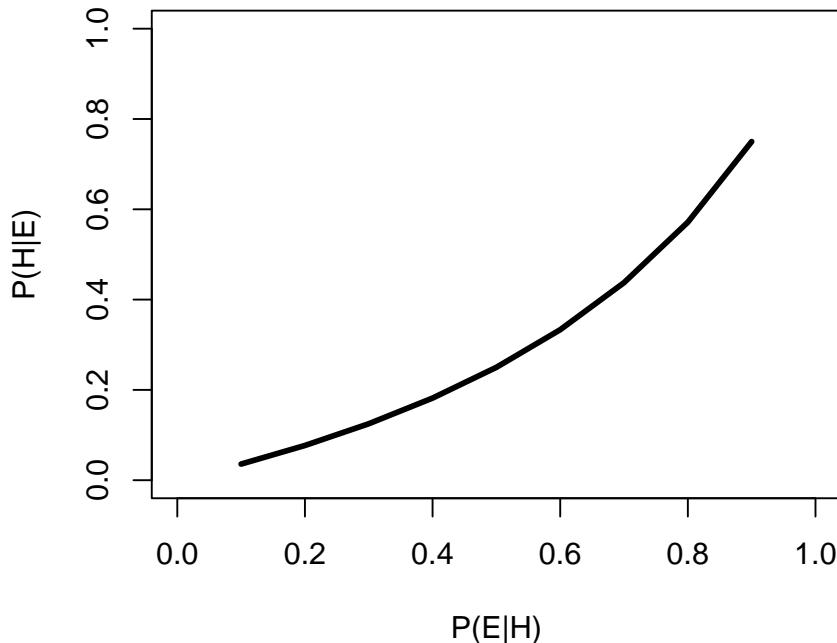
```
test.bt <- function(peh, ph){
  if(length(peh)==1){
    xl <- "P(H)"
    use.x <- ph
  } else {
    xl <- "P(E|H)"
    use.x <- peh
  }
  y <- (peh*ph)/((peh*ph)+((1-ph)*(1-peh)))

  plot(use.x, y, type="l", lwd=3, xlab=xl,
       ylab="P(H|E)", xlim=c(0,1), ylim=c(0,1))
}
# example usage:
test.bt(peh=0.9, ph=10^seq(-6, -0.0001, length=100))
```

⁹Citation needed



```
test.bt(peh=1:9/10, ph=0.25)
```



Let's change the numbers a bit to get a more visual understanding of how this rule is working. Imagine another scenario:

About 24% of US adults are nearsighted¹⁰. As of 2019, about 6% of US adults worked in education¹¹. If someone is nearsighted, what is the probability that they work in education?

First, assemble the terms:

- $P(H)$, the prior unconditional probability that someone works in education, 6% or 0.06
- $P(E)$, the unconditional probability of being nearsighted, or marginal likelihood, is 24% or 0.24.
- $P(E|H)$, the probability of a nearsighted educational worker (assuming independence) is $0.24 \times 0.06 = 0.0144$.

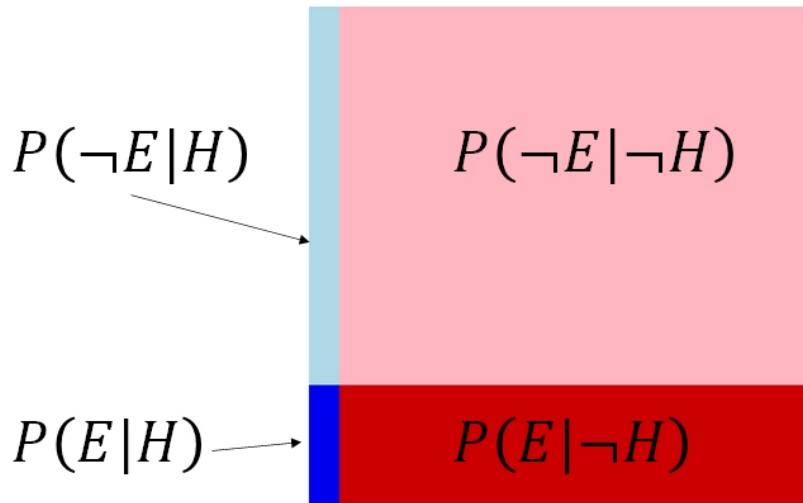
We can then calculate $P(H|E)$ as:

¹⁰Source

¹¹Source

$$P(\text{educator}|\text{nearsighted}) = \frac{P(H)P(E|H)}{P(H)P(E|H) + (1 - P(H))(P(E)(1 - P(H)))} = 0.064$$

This might make sense if we draw the probabilities in rectangle with area = 1. The probability of the hypothesis being true given the evidence collected is the ratio of the probability that the evidence is observed and the hypothesis is true— $P(E|H)$ —and the total probability that the evidence is observed at all¹²

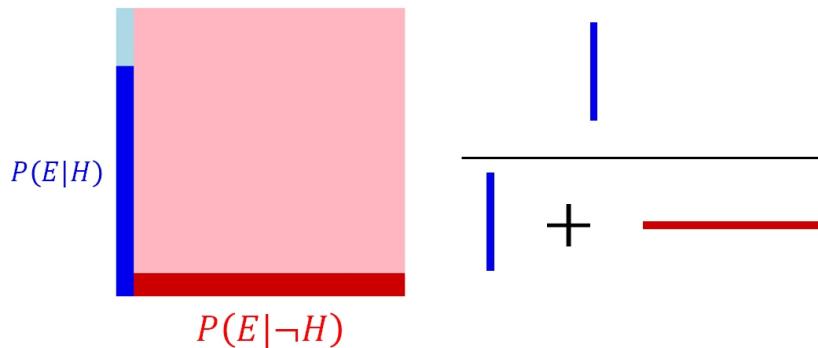


And the critical calculation:

$$\frac{P(E|H)}{P(E|H) + P(E|\neg H)}$$

This example was kind of trivial, but what about if the probability of observing the evidence is not the same in each group? What if educators were 10× as likely to be nearsighted as other workers? Then the rectangle and calculation would look like this:

¹²Here's a nice video illustrating this concept. I can't recommend this channel highly enough!



In the second example, the probability that someone is an educational worker given that they are nearsighted is about 38.9%... quite a step up from the first number! This latter example shows the power of the evidence when the evidence is strong. Even though the probability of the posterior seems quite low relative to the space of all possibilities, a Bayesian would evaluate that probability in light of the evidence. The evidence restricts the range of possibilities to the cases where the evidence was actually observed (the darker sections of the rectangles). In the Bayesian framework, the probability associated with the lighter sectors, $P(\neg E|H)$ and $P(\neg E|\neg H)$, *don't matter because they weren't observed*.

This latter point is one of the key points of difference between a traditional frequentist analysis (i.e., NHST) and a Bayesian analysis. In a frequentist analysis, the evidence is viewed as coming from a random distribution conditional on some true set of model parameters (i.e., on H). In Bayesian inference, the evidence is taken as given, and the model parameters conditional on the the data.

1.5.1.2 Example Bayesian analysis—not so simple

Let's try another Bayesian analysis, this one not so simple. Usually, a biologist will use Bayesian inference to fit models to biological data, not make trivial calculations about the probability of observing near-sighted college professors (although you could). The example below illustrates how to fit a simple linear regression model using Bayesian inference. We will use the program JAGS, short for Just Another Gibbs Sampler (Plummer 2003), called from R using package `rjags` (Plummer 2021).

Let's simulate a dataset for our example analysis:

```
set.seed(123)
x <- runif(100, 1, 16)
beta0 <- 40    # intercept
beta1 <- -1.5 # slope
sigma <- 5      # residual SD
y <- beta0 + beta1*x + rnorm(length(x), 0, sigma)
```

Next, we need to write the model in the language used by JAGS. JAGS was

designed to mostly work with a language developed for an older program for Bayesian modeling called BUGS (Bayesian inference Using Gibbs Sampling) (Lunn et al. 2009). BUGS itself has very similar syntax to R, and for that among other reasons is one of the most popular Bayesian statistics platforms. Both use a technique called **Markov Chain Monte Carlo (MCMC)** to fit models and sample from the posterior distributions of the parameters. In a nutshell, MCMC works by trying lots of random values in a sequence. At each step, the algorithm randomly changes each parameter. Parameters can change a lot when the model fit is poor, or a little bit when the model fit is good. Eventually the chain of values converges on a set of parameters with good model fit¹³.

JAGS is a separate program from R that must be installed on your machine. When called from R, JAGS will read a plain text file that contains a model. We will use the `sink()` command to make that file within R and save it to the home directory.

```
mod.name <- "mod01.txt"
sink(mod.name)
cat("
model{
  # priors
  beta0 ~ dnorm(0, 0.001)
  beta1 ~ dnorm(0, 0.001)
  sigma ~ dunif(0, 10)
  tau.y <- 1 / (sigma * sigma)

  # likelihood
  for(i in 1:N){
    y[i] ~ dnorm(y.hat[i], tau.y)
    y.hat[i] <- beta0 + beta1 * x[i]
  }# i for N
}#model
", fill=TRUE)
sink()
```

This model defines the important parts of a Bayesian model: the priors $P(H)$ and the likelihood $P(E|H)$. JAGS will automatically calculate the other terms for you (it is also calculating the actual probabilities of the priors and likelihood). Notice that the priors are very uninformative, so that the posteriors are driven by the data. For example, for the intercept and slope we assume that they fall somewhere in a normal distribution with mean 0 and variance 1000. That's a pretty big range. We could also specify something like a uniform distribution with limits ± 1000 , or ± 10000 , something even wider. The wider and thus less informative the priors, the more influence the data will have. However, that comes at the cost of longer model run times required to zero in on the right

¹³As you might suspect, the method is MUCH more complicated than this. McCarthy (2007) has a good introduction to MCMC. A more compact version is in Link et al. (2002)

solution.

Next, we must define initial values for the model, package up the data for JAGS, and set the MCMC parameters.

```
# define initial values for MCMC chains
init.fun <- function(nc){
  res <- vector("list", length=nc)
  for(i in 1:nc){
    res[[i]] <- list(beta0=rnorm(1, 0, 10),
                      beta1=rnorm(1, 0, 10),
                      sigma=runif(1, 0.1, 10))
  }
  return(res)
}
nchains <- 3
inits <- init.fun(nchains)

# parameters to monitor
params <- c("beta0", "beta1", "sigma")

# MCMC parameters
n.iter <- 5e4
n.burnin <- 1e4
n.thin <- 100

# package data for JAGS
in.data <- list(y=y, x=x, N=length(x))
```

Finally we can run the model:

```
library(rjags)
library(R2jags)

model01 <- jags(data=in.data, inits=inits,
                  parameters.to.save=params,
                  model.file=mod.name,
                  n.chains=nchains, n.iter=n.iter,
                  n.burnin=n.burnin, n.thin=n.thin) #jags

## Compiling model graph
## Resolving undeclared variables
## Allocating nodes
## Graph information:
##   Observed stochastic nodes: 100
##   Unobserved stochastic nodes: 3
##   Total graph size: 414
##
```

```
## Initializing model
##
## | |
## | |
```

The most important part of the output is here:

```
model01

## Inference for Bugs model at "mod01.txt", fit using jags,
## 3 chains, each with 50000 iterations (first 10000 discarded), n.thin = 100
## n.sims = 1200 iterations saved
##      mu.vect sd.vect   2.5%    25%    50%    75%   97.5% Rhat n.eff
## beta0     39.937   1.084  37.698  39.247  39.941  40.650  41.920 1.001 1200
## beta1     -1.524   0.114  -1.742  -1.598  -1.525  -1.449  -1.293 1.001 1200
## sigma      4.908   0.366   4.236   4.642   4.876   5.140   5.684 1.000 1200
## deviance  600.500   2.511 597.596 598.648 599.891 601.643 606.925 1.001 1200
##
## For each parameter, n.eff is a crude measure of effective sample size,
## and Rhat is the potential scale reduction factor (at convergence, Rhat=1).
##
## DIC info (using the rule, pD = var(deviance)/2)
## pD = 3.2 and DIC = 603.7
## DIC is an estimate of expected predictive error (lower deviance is better).
```

This table shows the posterior distribution of each model parameter: the intercept **beta0**, the slope **beta1**, and the residual SD **sigma**. There is also a measure of model predictive power called **deviance**. Better-fitting models have smaller deviance. The posterior distributions are given by their mean (**mu**), SD, and various quantiles. The **Rhat** statistic (\hat{R}) is also called the **Gelman-Rubin statistic**, and helps determine whether or not the model converged. Values of \hat{R} close to 1 are better (usually <1.001 or <1.01 are desirable, but there is no universally agreed-upon rule).

These parameter estimates are not far off from the true values of 40, -1.5, and 5. Notice that the means of the posterior distributions are close to the values estimated by ordinary linear regression:

```
summary(lm(y~x))

##
## Call:
## lm(formula = y ~ x)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -11.1899  -3.0661  -0.0987   2.9817  11.0861
##
## Coefficients:
```

```

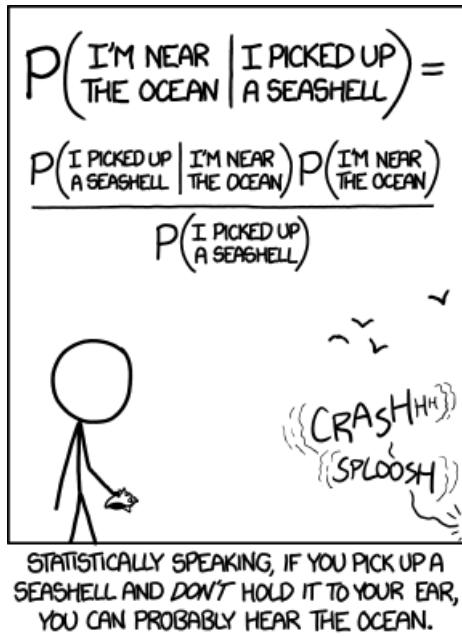
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 39.9851     1.0808   37.00  <2e-16 ***
## x           -1.5299     0.1139  -13.43  <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 4.846 on 98 degrees of freedom
## Multiple R-squared:  0.6479, Adjusted R-squared:  0.6443
## F-statistic: 180.3 on 1 and 98 DF,  p-value: < 2.2e-16

```

So why go through all of that Bayesian trouble? There are a few reasons where Bayesian inference might be preferable:

1. The model to be fit is very complex. Because the user can specify any model they want, they can fit any model they want.
2. Maximum-likelihood methods often require explicit derivations of model likelihoods, whereas an MCMC approach does not. This means that models whose likelihoods have no closed form solution, or computationally difficult solutions, can be estimated more easily with Bayesian methods than maximum likelihood (sometimes).
3. The researcher has prior information that they would like to incorporate into the analysis. This can reduce the amount of data needed to reach certain conclusions (see below)¹⁴.
4. Maximum likelihood methods are not effective in some situations.
5. Error propagation is very easy under Bayesian inference (especially MCMC).
6. The researcher prefers the philosophy of Bayesian inference.

¹⁴Source: XKCD



Bayesian methods are largely beyond the scope of this course, but we will use MCMC in JAGS from time to time to fit complicated models where maximum likelihood doesn't work so well.

1.5.2 Information-theoretic methods

Another alternative to NHST is **information-theoretic (IT)** modeling. IT methods borrow insights from information theory to draw conclusions about the extent to which statistical models describe data (Burnham and Anderson 2002). Rather than estimate parameters that minimize a loss function (e.g., residual sums of squared errors) and eliminate variables individually based on P -values, IT methods are used to compare different data models in terms of how much information is lost by using one of the models to represent the underlying process (Hobbs and Hilborn 2006). In other words, NHST arrives at a model by testing individual variables, while IT tests entire models.

There are several types of information criteria in use, but they all boil down to the sum of two terms: one that expresses the likelihood function of the data given the model, and another that penalizes the model for excess complexity (i.e., number of parameters). Information criteria such are used to compare models to each other, not to determine whether any particular model is “significant” or whether any of its terms are significant. For this reason, researchers using IT methods need to be very cautious to avoid overfitting.

The most common information criterion is Akaike's Information Criterion (AIC) (Burnham and Anderson 2002):

$$AIC = -2L + 2K = 2K - 2 \log(\hat{L})$$

where L is the natural logarithm of the likelihood function (\hat{L}) and K is the number of parameters in the model. The $2\times$ penalty on K makes it so that simpler models are preferred. Without this penalty, the likelihood could be increased simply by adding additional parameters (i.e., by overfitting). The likelihood function is expressed as -2 times its logarithm so that better fits (greater likelihoods) reduce AIC. The logarithm makes this effect nonlinear.

Other information criteria exist, such as the AIC corrected for small sample sizes (AIC_C):

$$AIC_C = AIC + \frac{2K(K+1)}{n-K-1}$$

Where n is the number of observations used to fit the model. Exactly what constitutes a “small” sample size is subjective, but Burnham and Anderson (2002) suggest that AIC_C should be used when $n/K < 40$. Many other criteria exist, such as QAIC and $QAIC_C$ for use when overdispersion is suspected (Lebreton et al. 1992), Bayesian information criterion (BIC) for a more conservative alternative to AIC; and DIC (deviance information criterion) for use in Bayesian contexts.

1.5.2.1 Likelihood?

Before we jump into an example IT data analysis, we also need to understand what a “likelihood” function is. We’ve encountered the term several times in both Bayesian and non-Bayesian contexts. Formally, a **likelihood** is a function of the probability of observing the data under a particular data model. A data model is an expression of the mathematical relationships between variables; e.g., the linear regression model. When statisticians work with likelihoods, they are usually trying to find the model parameters that maximize the likelihood, or minimize the negative log-likelihood. Likelihoods are related to but not the same thing as probabilities.

1.5.2.2 Example IT analysis

As before, we will simulate a dataset so we know exactly what the results should be. Let’s create a dataset where some outcome y depends on 1 of 3 predictor variables: x_1 , x_2 , and x_3 . To illustrate how AIC can distinguish between models with better or worse predictive power, we’ll make the dependent variable dependent on only one predictor, x_1 .

```
set.seed(456)
# sample size
```

```

n <- 100

# draw potential predictor variables
x1 <- sample(20:100, n, replace=TRUE)
x2 <- rnorm(n, 45, 10)
x3 <- sample(41:42, n, replace=TRUE)

# model parameters
beta0 <- 3
beta1 <- 3.2
beta2 <- -8
beta3 <- 0.03
sigma <- 5

# "true" model
y <- beta0 + beta1*x1 + rnorm(n, 0, sigma)

```

Next, fit linear regression models (“candidate models”) using different combinations of predictors. You can fit as many models as you like in this part, but should try to restrict the candidate set to those models you really think are viable, realistic hypotheses. Datasets with many predictors can easily generate hundreds or thousands of candidate models by assembling all of the combinations and permutations of different numbers of predictors. Don’t do that.

```

m00 <- lm(y~1)
m01 <- lm(y~x1)
m02 <- lm(y~x2)
m03 <- lm(y~x3)
m04 <- lm(y~x1+x2)
m05 <- lm(y~x1+x3)
m06 <- lm(y~x2+x3)
m07 <- lm(y~x1+x2+x3)

```

You can inspect these models (e.g., `summary(m01)`) and see that `x1` is always a significant predictor, while `x2` and `x3` are not. In frequentist terms, we would say that there was no significant effect of `x2` or `x3` on `y` because $P \geq 0.05$. In IT terms we would say that adding `x2` or `x3` to the model did not add explanatory power. Those statements are similar, but not quite the same because they use different criteria to infer how the variables are related.

We can compare the models using AIC. The command below will make a data frame holding the name of each model and its AIC.

```

aic.df <- AIC(m00, m01, m02, m03, m04, m05, m06, m07)
aic.df

```

```

##      df      AIC

```

```
## m00  2 1145.3398
## m01  3 623.8889
## m02  3 1144.8685
## m03  3 1147.1488
## m04  4 625.6944
## m05  4 624.2252
## m06  4 1146.5993
## m07  5 626.0870
```

What do we do with this? One common approach is to calculate what's called an **AIC weight**. This quantity is an estimate of the probability that a model is the best model out of present candidates. This is NOT the same as the probability that a model is correct, or the best of all possible models—only the best out of the current set of candidates. We can calculate AIC weights in R with a few commands:

```
aic.df$delta <- aic.df$AIC - min(aic.df$AIC)
aic.df$wt <- exp(-0.5*aic.df$delta)
aic.df$wt <- aic.df$wt/sum(aic.df$wt)
aic.df <- aic.df[order(-aic.df$wt),]
aic.df
```

	df	AIC	delta	wt
## m01	3	623.8889	0.0000000	3.870182e-01
## m05	4	624.2252	0.3362974	3.271187e-01
## m04	4	625.6944	1.8055142	1.569166e-01
## m07	5	626.0870	2.1981493	1.289464e-01
## m02	3	1144.8685	520.9796540	2.873670e-114
## m00	2	1145.3398	521.4509418	2.270378e-114
## m06	4	1146.5993	522.7103962	1.209514e-114
## m03	3	1147.1488	523.2599265	9.189293e-115

In this example, the model with the smallest AIC—and thus greatest AIC weight—was model 1 (the correct model). Interestingly, models 4 and 5 had $\Delta\text{AIC} < 2$, which is usually interpreted as not being distinguishable from the best-supported model. This shouldn't be surprising, because models 4 and 5 also included the true predictor `x1`. Model 7, which also included `x3`, was nearly as good but with a $\Delta\text{AIC} \geq 2$ we can exclude it. Notice that the worst-supported models (2, 0, 6, and 3) were the models that did not include the true predictor `x1`.

One key advantage of the IT framework is that it focuses less on *P*-values and permits the consideration of multiple hypotheses at once. If competing hypotheses can be expressed as a different statistical model (like in the example above), then an information criterion can be used to estimate which hypothesis is best supported by the data (although not necessarily whether any of them is correct). Another key advantage is that it allows averaging parameter estimates across multiple models. This allows researchers to say something about the effect

of a factor without concluding exactly which statistical model is the correct one. Essentially, the weighted mean of parameter estimates across the models is calculated. The AIC weights are used to weight the mean:

```
res.list <- list(m01, m05, m04, m07)
x1.est <- sapply(res.list, function(x){x$coefficients["x1"]})
x1.est

##      x1      x1      x1      x1
## 3.232883 3.234502 3.231263 3.233119
weighted.mean(x1.est, aic.df$wt[1:4])

## [1] 3.233189
```

Not too far off from the true value of 3.2!

We'll use an IT model selection approach occasionally in this class. While the example above was easy and relatively straightforward, there are some important caveats to using AIC and similar methods:

1. Information criteria are only a measure of relative model performance, not model correctness or overall goodness of fit.
2. Information criteria are only interpretable for sets of nested models: models that can be transformed to each other by setting one or more coefficients to 0.
3. Rules-of-thumb about information criteria are just as arbitrary as the $P < 0.05$ criterion.
4. Not every information criterion can be used in every situation. Read the friendly manual.
5. IT methods are vulnerable to overfitting because they evaluate entire models, not individual variables. Researchers must think carefully about what variables to include.

1.5.3 Machine learning

The last alternative to NHST that we'll explore in this class is machine learning (ML). ML is a huge and rapidly growing field of methods for extracting patterns from complex datasets. Many of these methods bear little resemblance to traditional statistics¹⁵.

¹⁵Source: XKCD



There are too many ML algorithms out there to even begin to cover here, but they all have the same underlying philosophy: ML builds a model on a sample of the data, called the “training data”, and refines or improves the model based on its ability to predict additional “test data” that were not used in the model building process.

This strategy has some considerable advantages over traditional frequentist and Bayesian statistics. In those paradigms, researchers must specify a data model ahead of time and then test how well the data support that model. Specifying a data model makes many assumptions about the distribution of the response variable, the shape of the relationship between the variables (e.g., linear vs. curved vs. stepped), the relationships between predictors, and so on. ML usually makes no such assumptions. Instead, the model is treated as a “black box” that takes in input and spits out predictions. Researchers using ML methods typically don’t worry about or even try to interpret the inner workings of the black box. Those inner workings are often nonsensical to humans anyway. Instead, users of ML report and interpret the predictions of their models and different measures of predictive accuracy.

The disadvantages of ML compared to traditional statistics are also considerable. By not specifying a data model, researchers must assume that their training data are representative of the process they are trying to model. There’s an old

saying in statistics and modeling: “Garbage in, garbage out.” If training data are not appropriate, or if there really is no relationship between the predictor variables and the response variable, there is no guarantee that a ML algorithm won’t find one anyway (this is similar to how traditional statistics sometimes get false positives). It’s still up to the researcher to interpret the patterns that the ML method detects and think long and hard about whether those patterns are reasonable.

Other disadvantages of ML are more practical. Most ML techniques require a lot of computing power, that may not be feasible for very large datasets for some researchers. ML techniques are also not standard practice in most fields of science, so researchers wanting to use ML will need to do extra work to show to editors and reviewers that their methods are legitimate and appropriate to their question. ML techniques are not as widely implemented in software packages as traditional statistical methods, and are not as well documented. This means that you will have fewer choices for what software to use and fewer places to get help. Finally, ML techniques can be easy to use but the details can be very hard to understand. As with any data analysis method, it is very easy to get yourself stuck or get into trouble using a technique you only partially understand.

I won’t include an example ML analysis here but can point you to some resources. For ecologists, De’ath (2007) and Elith et al. (2008) are the standard references. Elith et al. (2008) also points to an online source for some R functions that simplify the process (Elith and Leathwick 2017). Tarca et al. (2007) provided an early review for biology in general. Jones (2019) reviewed some applications of ML in cell biology. ML methods are being applied to problems that are not amenable to traditional statistics, particularly image analysis (Kan 2017, Whytock et al. 2021).

Chapter 2

Introduction to R

This module is an introduction to the R program and language. We will begin with a brief discussion of R itself, then run through a typical R work session, and then take a deeper dive into some R basics—including how to get help. Most of the topics in this module are covered in greater detail in the module on data manipulation. The current module, *Module 2*, is meant to get you started before moving on to more advanced material.

By the end of this module you should be able to:

- Explain what R is and what it is used for
- Explain the object-oriented programming paradigm used by R
- Download and install R and RStudio
- Write and execute commands in the R console
- Describe basic R data types and structures
- Manage R code as scripts (.r files)
- Manage and use R packages
- Identify sources of help for R programming

2.1 Getting started with R

2.1.1 What is R?

R is an open-source language and environment for data analysis, statistics, and data visualization. It is descended from an older language called S, which is the basis for the commercial statistics software S-Plus. R is part of part of the GNU free software project. The GNU project is a free and open-source software ecosystem including an operating system and many programs that can replace paid proprietary software. GNU is a recursive acronym that stands for “GNU’s Not Unix”.

R is designed for statistics and data analysis, but it is also a general-purpose programming language. One of the most important features of the R language is that it is **object-oriented**. R commands and programs focus on manipulating **objects** in the computer memory. An object is a construct that has a **class**, and can be acted upon by **methods**.

- A class is like a blueprint for an object. The class defines the data format in an object, and how other objects can interact with that object.
- An object is an instance of a class.
- A method is a function that works with objects according to their class.

For example, a graphics program might have classes such as “circle”, “square”, and “triangle”. Every circle, square, or triangle that the user draws is an instance of one of those classes. This way the user can create different examples of common shape types without having to define the shapes anew every time one is created.

Everything in R is an object of one kind or another! Understanding this idea is the key to R success. We will learn about some of the most important R classes later.

2.1.2 Advantages of R

If you are going to choose a tool for statistics in your research, you need to have a good reason. R offers several very good reasons to choose it for your statistical computing needs. The key advantages of R are:

- R is free!
- R is widely used for statistics and data analysis.
- R has a large and active community of users and developers. This means that new statistical techniques are often implemented in R before they are implemented in other software.
- R is a highly marketable professional skill because it is widely used in academia, government, and industry.
- R Is open source: you can modify it to suit your needs, and see exactly how it does what it does. The R source code is hosted on github (accessed 2021-12-20).

2.1.3 Disadvantages of R

For all of its advantages, R has some drawbacks. Chief among them:

- R is free. Unlike some paid and proprietary software, there is no official help line to call if you have issues.
- Along the same lines, R is distributed with absolutely no warranty (it says so every time you open R). The accuracy of your analysis depends on the competence of the (unpaid) contributors and the user.

- R often requires more coding than SAS or other tools to get a similar amount of output or perform a similar amount of work. The **tidyverse** ecosystem of packages addresses some of these issues.
- R is slower than SAS and other tools at handling large datasets because it holds all data and objects in memory (RAM) at once. This limits the performance of your machine if you are doing other tasks.
- R is single-threaded by default. This means that R does not handle parallel computation very easily, which could potentially speed up many tasks.
- R has a difficult learning curve because it is so programming-focused.
- Error messages that R gives are often vague and infuriating.

2.1.4 Base R and (vs.?) **tidyverse**

A colleague of mine who moved from using mostly SAS to using mostly R says that the **tidyverse** makes R feel more like SAS. The difference he was referring to is the way that SAS functions (“PROCs”, or procedures) tend to automate or abstract away a lot of the low-level functionality and options associated with their methods. Those functions and options are there, but they can be inconvenient to access. The equivalent base R functions, on the other hand, require the user to specify options and manipulate output to a far greater extent. Or, depending on your perspective, they *allow* the user to specify options and manipulate output. Some people like that, and some people don’t. The amount of interaction and coding in R require to get outputs that other programs like SAS produce automatically is, in my experience, one of the biggest stumbling blocks for new R users.

Because of the amount of work required to get base R to do anything, the degree of automation and abstraction available in the **tidyverse** packages (Wickham et al. 2019) is very attractive to many R users: they can focus on high-level decisions without having to get into every picayune detail of routine workflows and standard analyses. Tidyverse packages allow for very streamlined and (once you learn it) easy to understand code. These advantages come at a cost: because the functions abstract the finer details away, those details are harder to see. This can be problematic if you need to work more directly with function inputs and outputs, or manipulate options or data in non-standard ways¹. The abstraction of the low-level workings of an analysis or program can save a lot of time and headache, but can also make it harder to track down or even to notice errors (although to be fair, systematic approaches to data manipulation such as in package **dplyr** can also **prevent** many errors as well).

In my own workflow, and in this course, I tend to use base R methods instead of **tidyverse** for a few reasons. The first three are entirely personal and subjective.

1. I learned R before **tidyverse** became a thing. This isn’t an advantage or disadvantage to either paradigm; it just is. For me, the time and effort

¹For example, if a journal editor is adamant about some minute detail of formatting on a figure.

- costs of switching to `tidyverse` would likely outweigh the benefits.
2. I prefer the syntax of base R to the “grammar” of `tidyverse`. In particular, `tidyverse` seems to make extensive use of the pipe operator `%>%`, which tends to make for less clear (to me) code.
 3. I prefer to write code with as few dependencies as possible. This makes it less likely that some random update will break my code.

The other two reasons are more functional and less subjective:

4. Debugging and error correction is easier in base R because of the *lack of abstraction*. If every step is coded manually and out in the open, errors are easier to find, isolate, and fix.
5. Debugging and fixing `tidy` code often requires knowing base R, but the reverse is never true.

The takehome message is that while both base R and `tidyverse` offer powerful tools for working with and analyzing data, they represent two overlapping but different programming paradigms. Neither is objectively “better” than the other. Which framework you use depends on what level of abstraction vs. explicitness you are comfortable with in your own code. Base R makes it easier to work flexibly with the low-level inputs and outputs of different methods, but usually at the cost of a steeper learning curve. `tidyverse` makes routine tasks more streamlined with a unified syntax and grammar for data manipulation, but at the cost of flexibility. My advice is to not restrict yourself to one or the other.

2.2 Download and install R (and RStudio)

If you are on a university campus, many on-campus computers may already have R installed on them. This is the bare minimum you need to work with R. Many people also prefer to use another program for writing R code. The most popular is probably RStudio, which is a more user-friendly interface for R. If you are on a PC or Mac, you will need to download and install both R and RStudio. If you are using a Linux machine, you probably already have R because it is included in most distros. You can still install RStudio to make your life easier.

First, install the latest version of R from the R-project website (accessed 2021-12-20). Select a mirror (download server) from the list. Once you select a mirror, you will be taken to a page where you will select the version of R that corresponds to your operating system. Click on the appropriate link for your operating system and follow the instructions from there to install R. Like many programs you download from the internet, R will come with an installer that will largely take care of everything automatically. Just let the installer run and accept the default options.

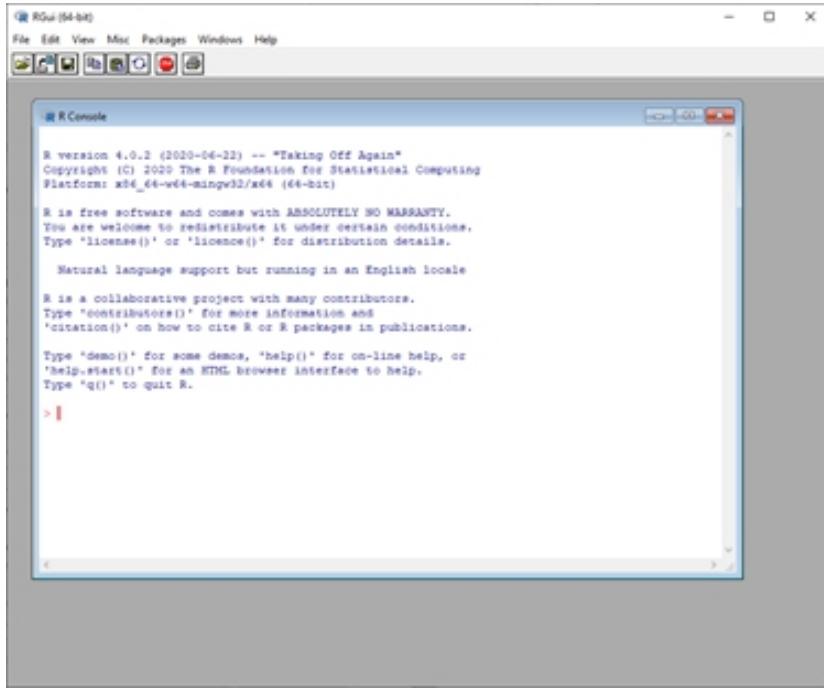
Next, you need to install RStudio. It can be installed from here (accessed 2021-12-20). As with R, select the version that corresponds to your operating system. As you did when you installed R above, just let the installer run and

accept the default options.

2.3 Using R

2.3.1 Using the base R GUI

R has a very basic front end, or **graphical user interface (GUI)**. Most users find it easier to write their R code in another program. Some of these programs, such as RStudio, are **integrated development environments (IDE)**. Others are text editors with programming capabilities. We'll explore the default R GUI first and then meet some editor options. The basic R GUI is shown below.



The main window with R is the **R console**. This is the command line interface to R. Some important features of the console are:

1. Command prompt: The > symbol signifies the beginning of a command. You can enter commands here.
2. Continue prompt: + signifies the continuation of a previous command. If you have a + at the beginning of your line, you can get back to a new line using ESCAPE. Note that this will delete the currently incomplete line or command.
3. The ENTER key executes the current command. This could mean the current line, or multiple lines going back to the most recent >.

4. [1] at the beginning of results: because your result is an object (with $\text{length} \geq 1$). The very first value (“element”) in the result is element 1.
5. Up arrow key: cycles through previously entered commands. You can see many previous commands at once by using the command `history()`.

Other important features of the R GUI are found in the menu bar:

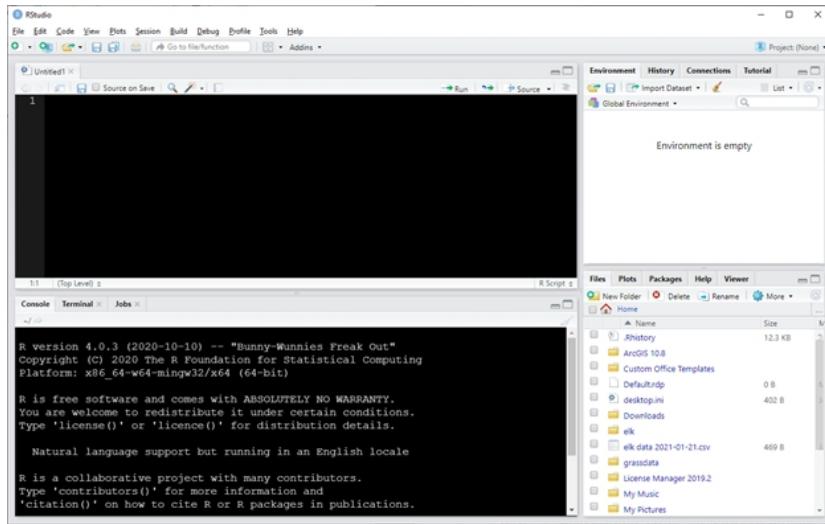
- File
 - New script: opens a new script window within the GUI.
 - Ability to open saved scripts or load previous workspaces
- Edit
 - Copy, paste, etc.: very useful!
 - Data editor: opens a rudimentary spreadsheet-like viewer for datasets that can be used to make edits. Not recommended for routine use.
 - GUI preferences: options to change colors, fonts, font sizes, etc., in the GUI. To make changes permanent you may need to save a new `.RConsole` file in your home directory.
- View
- Misc
 - List objects: lists all objects in workspace. Equivalent to command `ls()`.
 - Remove all objects: removes all objects from workspace. Equivalent to command `rm(list=ls(all=TRUE))`.
 - List search path: lists currently attached packages. Equivalent to command `search()`.
- Packages: functions to download and install packages.
- Windows: options to change how windows are displayed within GUI.
- Help: options and resources for getting help, including several free R manuals in PDF format.

2.3.2 Using R in RStudio

Although you can type R code directly into the console, or type in script windows within the base R GUI, it is usually easiest to use a separate program for writing code. The two best, in my experience, are RStudio and Notepad++ (see below). **RStudio** is an IDE designed to manage R projects, and has many useful features. Unlike some other options, RStudio has R built in so you can write code and execute it within the same program. RStudio can be used on Windows, Mac, and Linux operating systems. RStudio also has built-in support for utilities like RMarkdown.

Once you have RStudio installed, open it and you will see a screen like this²

²Your RStudio might look different at first. You can change the color scheme by going to Tools - Global Options - Appearance and selecting a theme. Here I'm using the “Classic” theme with size 12 Courier New font and 110% zoom.



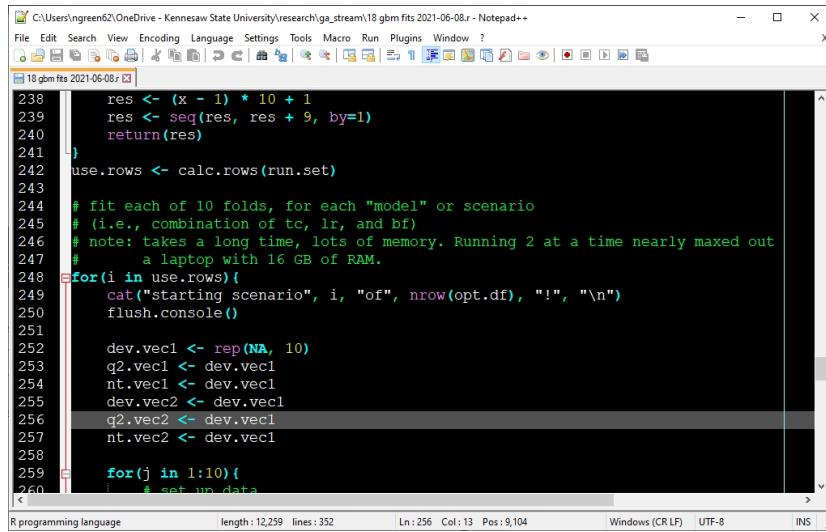
RStudio has an entire environment for working with R in one window. Each of the panels performs different functions. Starting at the bottom left and going clockwise, the panels are:

- Console: command line interface for R. You can type commands here and execute them by pressing ENTER.
- Script: a space where you can type code and later execute it by either clicking “Run” or pressing CRTL-ENTER (CMD-ENTER on a Mac). Usually people write code in the script window rather than typing it directly into the console.
- Environment: this window lists all of the data objects that you have loaded.
- Files: this window lists all of the files available in your R home directory. You can also navigate to other folders on your machine and select files to open.

The bottom right panel also has a tab called “Plots”. This is where plots that you make can be seen. Sometimes you will need to resize this panel to see the plots clearly. All of the panels can be resized by clicking and dragging the borders between them.

2.3.3 Using R with other programs

Notepad++ is an open-source code editor that is available on Windows (accessed 2021-12-20). Notepad++ is a more generic source code editor with built-in R support and many programming features. It includes programming support for R as well as other languages used by biologists, such as Python, C++, Visual Basic, and LaTeX. The key disadvantage of Notepad++ is that it does not connect to R, so you must copy and paste your code to an R console. Another disadvantage is that it is only available for Windows. The image below shows a script in Notepad++,



```

238     res <- (x - 1) * 10 + 1
239     res <- seq(res, res + 9, by=1)
240     return(res)
241 }
242 use.rows <- calc.rows(run.set)
243
244 # fit each of 10 folds, for each "model" or scenario
245 # (i.e., combination of tc, lr, and bf)
246 # note: takes a long time, lots of memory. Running 2 at a time nearly maxed out
247 #      a laptop with 16 GB of RAM.
248 for(i in use.rows){
249   cat("starting scenario", i, "of", nrow(opt.df), "!", "\n")
250   flush.console()
251
252   dev.vec1 <- rep(NA, 10)
253   q2.vec1 <- dev.vec1
254   nt.vec1 <- dev.vec1
255   dev.vec2 <- dev.vec1
256   q2.vec2 <- dev.vec1
257   nt.vec2 <- dev.vec1
258
259   for(j in 1:10){
260     # set up data
<

```

R programming language length : 12,299 lines : 352 Ln : 256 Col : 13 Pos : 9,104 Windows (CR LF) UTF-8 INS

emacs is a GNU program that is an extremely powerful text editor, code editor, and pretty much everything else. With its extension ess or “emacs speaks statistics”, it can also be an R IDE. Be forewarned, getting emacs and ess to work on Windows can be a major pain.

Other good options include Tinn-R, Jupyter notebooks, and many more. All of these options are subject to irrationally strong personal and organizational preferences. For example, I use Notepad++ almost exclusively in my research for writing and editing R code. I would prefer to use emacs, but it doesn’t play nice with my university PCs. Many people swear by RStudio. Try a few R interfaces and use what works best for you. Or, use the interface that your manager tells you to use.

Whatever you do, you should not use a word processor (e.g., Microsoft Word) to write and edit R code. The reason that you should not use Word for code editing is because Word documents include lots of invisible formatting that can cause issues in R. Worse, Word autocorrects some characters in counter-productive ways. The most problematic are automatic capitalization (because R is case sensitive) and quotation marks. Word autocorrects "" to “” as you type, which makes for nice looking text. However, the curved quotation marks (“”) are not read as valid quotation marks by R. Only “straight quotes” (") are recognized by R.

2.4 A first R session

One of the best ways to get a feel for R is to step through a typical R workflow. We are going to analyze a classic dataset from evolutionary biology that contains data on the body size and antler size of many species of deer (family Cervidae). The data were collected by Stephen Jay Gould (Gould 1974) to investigate the

reason for the massive antler size in an extinct mammal, the Irish elk (*Megaloceros giganteus*), seen below³. Gould was interested in whether the massive antlers might have resulted from runaway sexual selection or were simply the result of allometric growth. The Irish elk lived across northern Eurasia during the Pleistocene epoch, until about 7700 years ago. Adults stood around 2.1 m high at the shoulder and males bore antlers that could reach up to 3.65 m (\approx 12 ft) across. Body mass of adult males is estimated to have ranged from 450 to 700 kg (990 to 1540 lb). Irish elk were on average larger than the largest extant cervid, the moose (*Alces alces*).



2.4.1 Import data

First, load the data into R. The data are contained in the text file `elk_data_2021-01-21.csv`. Download this file and save it on your computer. If you want to save some time later, save the data file in your **R home directory**. This is the default folder where R and RStudio will look for files. The R home directory depends on your operating system:

- **Windows users:** the home directory is your “Documents” folder. The address of this folder is “**C:/Users/**username**/Documents****”, where “**username**” is your Windows logon name. You can get to your Documents folder by opening Windows Explorer (**Win-E**) and double-clicking “**Documents**”.**

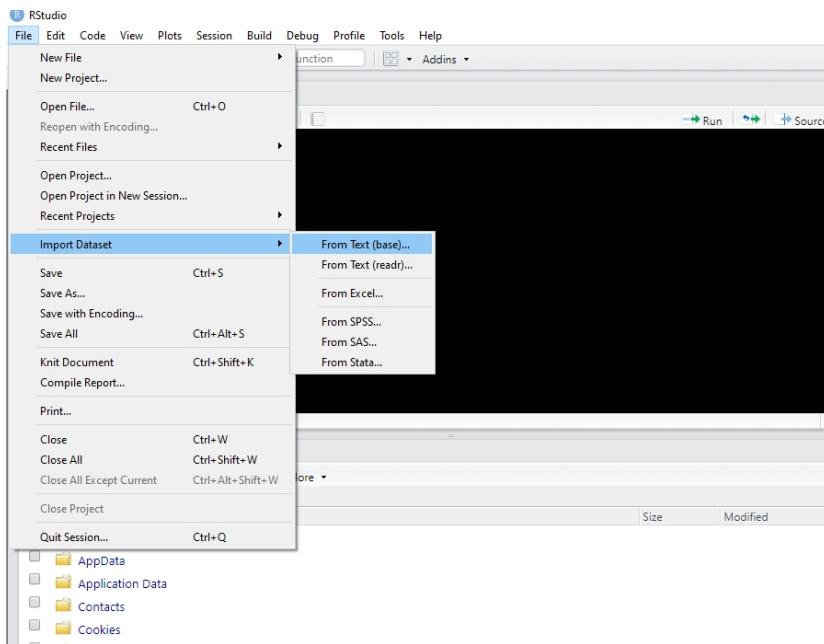
³Image: F. Atirador. URL: https://upload.wikimedia.org/wikipedia/commons/5/5a/Irish_Elk_Side_%28white_background%29.jpg (accessed 2021-12-20).

- **Mac users:** the home directory is your **home folder**. The address of this folder is “/Users/username” (where username is your username) or “~/”. You can open your home folder by pressing **CMD-SHIFT-H**, or by using the **Go** pulldown menu from the menu bar.
 - Note for Mac users: When you download a .csv file in MacOS, your computer might open it in a program called “Numbers”. This is a Mac-specific spreadsheet program. When you close the program it is possible to inadvertently save the data file as a new file in .numbers format, which is not readable by R. If you do this, your datafile will become “elk_data_2021-01-21.numbers”, and you will not be able to open it in R or RStudio. **.numbers is a different file format than .csv.** A .numbers file has the same file name, but it has a different file extension and so it is not the same as a .csv⁴.

Once you save the data file on your computer, you can import it to RStudio in several ways. Notice that in both methods shown here, nothing is printed to the console after a successful import. Instead, the function reads the CSV file and saves the data to an object that we are calling **dat**.

2.4.1.1 Method 1: Import in RStudio using File–Import Dataset

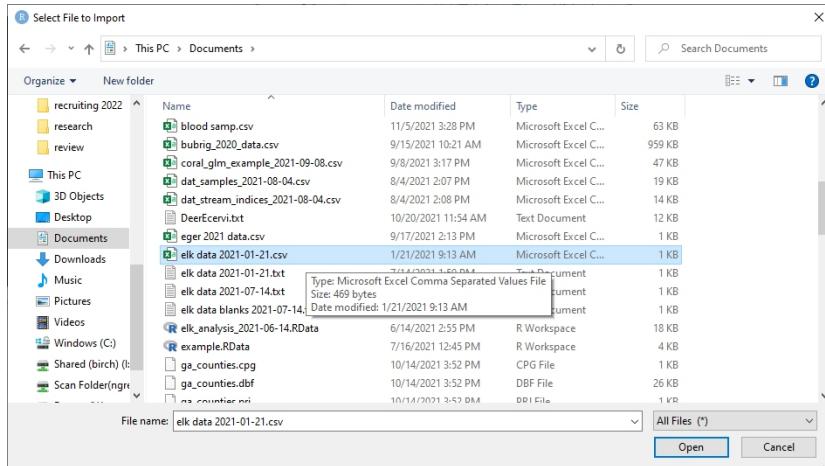
In RStudio, click on **File–Import Dataset–From Text (base)**.



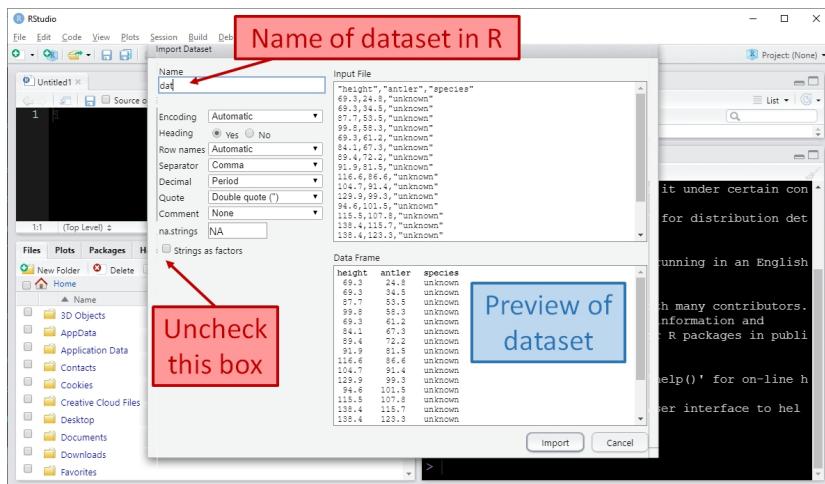
Navigate to the data file just as you would if opening a file in any other program.

⁴This issue seems to confuse a lot students who are Mac users, so I'm belaboring the point here.

Click Open.



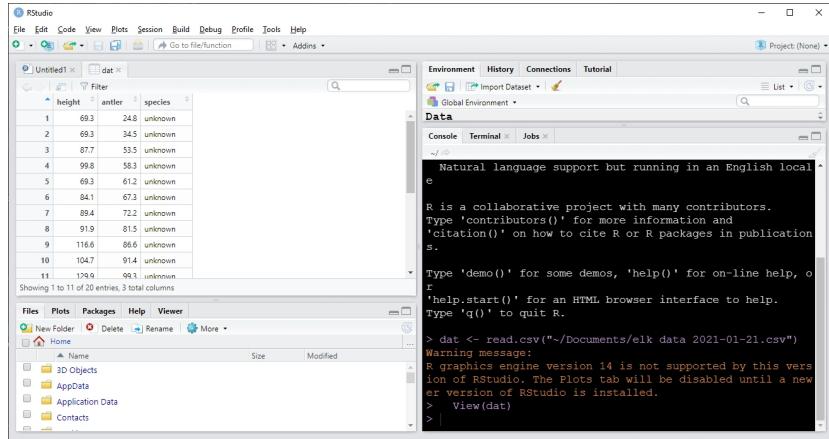
In the dialog box that pops up, change the name of the incoming dataset to “dat”. This is what we will call the dataset when working with it in R. Make sure that the “Strings as factors” box is NOT checked.



Click Import. RStudio will import the data for you and save it in the workspace as an object called “dat”. Notice that a command like `dat <- read.csv("~/elk_data_2021-01-21.csv")` was run in the console. This command is what RStudio used internally to import the data. If you want to know more, look at method 2 below.

After you click Import, RStudio will import the data and display it for you. The image below shows what that looks like. Notice that RStudio automatically generated the code to import the data and executed it (red arrow); that the dataset is listed in the “Environment” box in the top right (blue arrow), and the

dataset is displayed in the top left (green arrow).



Once the dataset is imported, you can view it in the console by typing its name and pressing ENTER. Each method above saved the dataset to an object named `dat`. Obviously, this is short for “data”⁵. However, we didn’t call our dataset “data” because `data()` is already the name of an R function and we don’t want to accidentally overwrite that function.

2.4.1.2 Method 2: Import from command line

You can also import files from the command line directly. This option is preferred by users who need to import files automatically, or who aren’t using RStudio. To use this method, you need to know the name of the folder where the data file is stored. If the data file is in your home directory (see previous section), the command is simple:

```
dat <- read.csv("elk_data_2021-01-21.csv")
```

The command above will look in your home directory for a file with the filename you specify. The filename must match *exactly*, including capitalization. For convenience I usually split the command up into two, with one specifying the name and the other importing the file:

```
dat.name <- "elk_data_2021-01-21.csv"
dat <- read.csv(dat.name)
```

If the data file is NOT in your home directory, you need to tell R which folder to look in. In the example below, I put the file on my Windows desktop⁶

⁵As we’ll discuss in the data manipulation module, naming R objects is a bit of an art. Names should be informative yet brief. I use a standard list of names like “`dat`”, “`met`”, and “`res`”, etc. in all of my projects because they are short but easy to identify (“`data`”, “`metadata`”, and “`results`”, respectively).

⁶I don’t recommend doing this, as the desktop can become easily cluttered.

(C:/Users/ngreen62/Desktop)⁷. This folder address is provided to R as a character string, and that string is combined with the file name using the `paste()` command.

```
use.fold <- "C:/Users/ngreen62/Desktop"
dat <- read.csv(paste(use.fold, dat.name, sep="/"))
```

Notice that the folder address above uses front-slashes (/). Windows uses back-slashes ("\\") in folder addresses, while R requires forward slashes ("/"). Mac and Linux already use front-slashes. If you are a Windows user, you will need to change the back-slashes to front-slashes in the folder address.

Once you have imported the data, you can type the name of the dataset into the console and press ENTER to view it:

```
dat
```

```
##   height antler   species
## 1    69.3    24.8  unknown
## 2    69.3    34.5  unknown
## 3    87.7    53.5  unknown
## 4    99.8    58.3  unknown
## 5    69.3    61.2  unknown
## 6    84.1    67.3  unknown
## 7    89.4    72.2  unknown
## 8    91.9    81.5  unknown
## 9   116.6    86.6  unknown
## 10   104.7   91.4  unknown
## 11   129.9   99.3  unknown
## 12   94.6   101.5  unknown
## 13   115.5   107.8 unknown
## 14   138.4   115.7 unknown
## 15   138.4   123.3 unknown
## 16   127.5   128.5 unknown
## 17   175.1   131.4    alces
## 18   164.3   162.6  unknown
## 19   204.5   197.5    alces
## 20   183.6   239.1 megaloceros
```

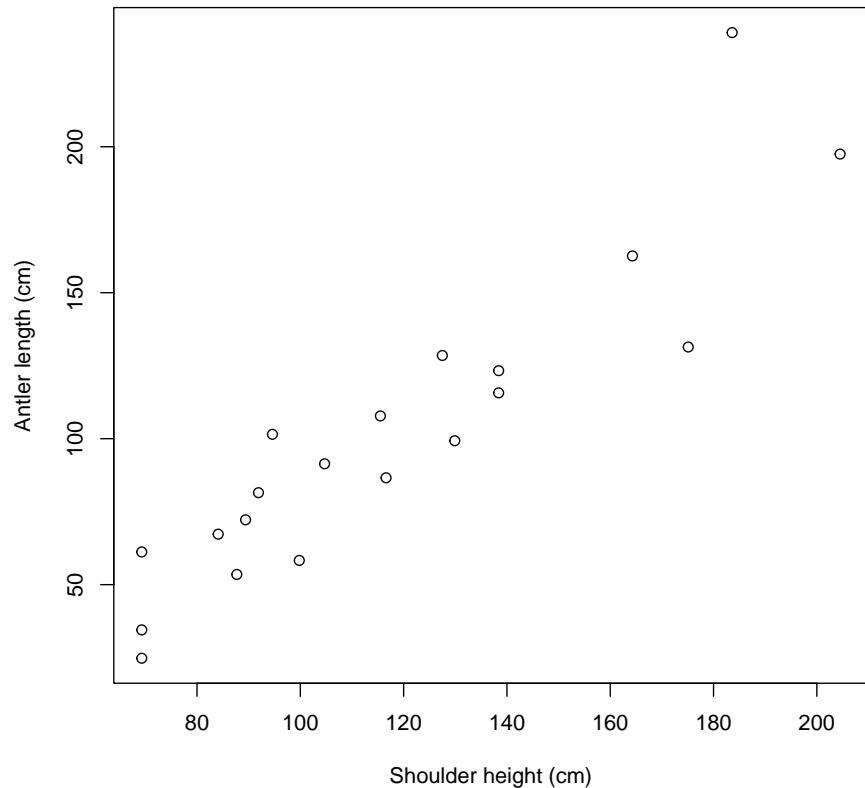
2.4.2 Explore and visualize data

Before we do anything, we should look at the dataset. We have two continuous variables, `height` and `antler`, so the natural way to examine their relationship is with a scatterplot. Scatterplots are made by the `plot()` function. By default, the first argument to `plot()` contains the *x* values, and the second contains the *y* values. The values are accessed from the data frame `dat` using the \$ operator.

⁷The Mac desktop is /Users/username/Desktop (where `username` is your username) or just ~/Desktop.

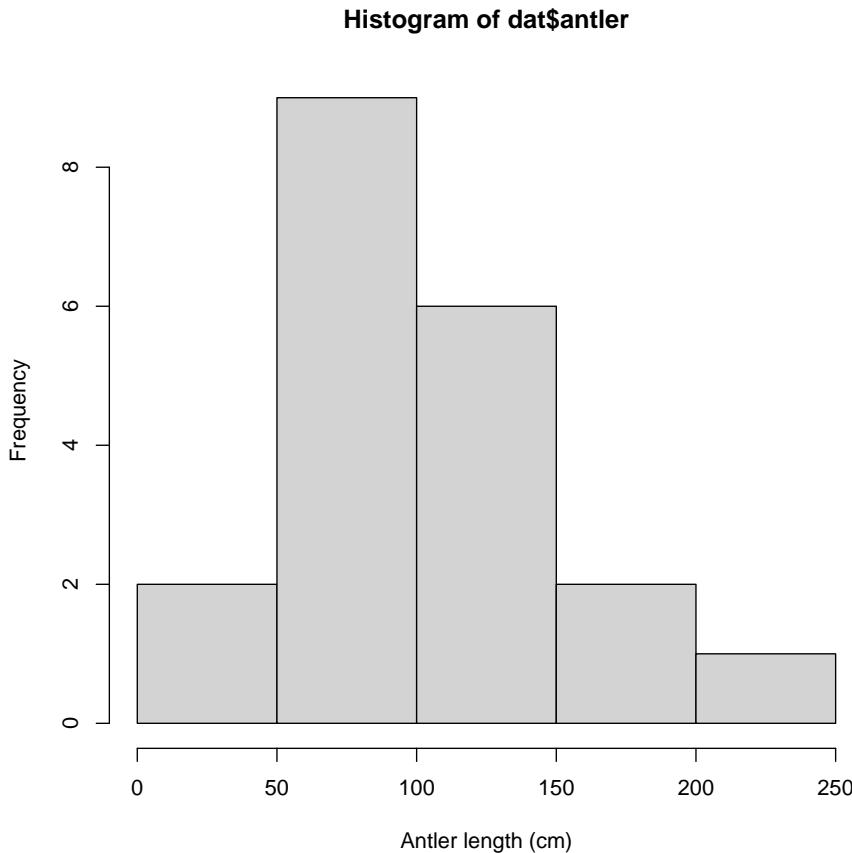
The command below also uses the arguments `xlab` and `ylab` to set *x*-axis and *y*-axis labels. Notice that R will set the limits of the graph according to the range of each variable; we'll go over how to set those limits later on.

```
plot(dat$height, dat$antler,
     xlab="Shoulder height (cm)",
     ylab="Antler length (cm)")
```



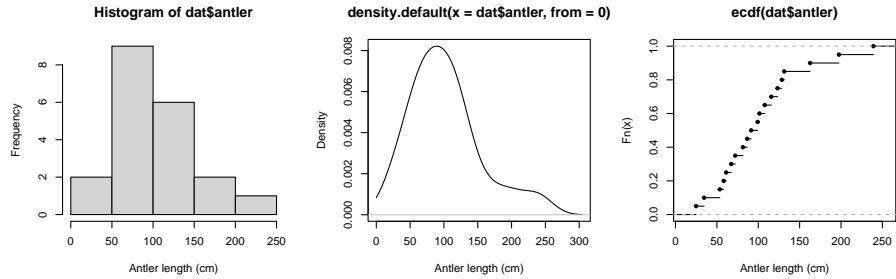
We might also be interested in how the data are distributed. This will be important later because the data need to meet particular assumptions for our statistical analysis. The simplest way to view a data distribution is with a histogram, `hist()`.

```
hist(dat$antler, xlab="Antler length (cm)")
```



We could also try a **probability density function (PDF)** plot, or plotting the **empirical cumulative distribution function (ECDF)**. Histograms, PDF plots, and ECDF plots are just different ways of displaying how the data are spread out. In the figure below, we use the function `par()` to set some graphical parameters (including a $1 \text{ row} \times 3 \text{ column}$ layout using argument `mfrow`). We also saved the text string “Antler length (cm)” as an object (`use.xlab`), and then used it in each of the plotting commands so we didn’t have to type the same thing over and over.

```
use.xlab <- "Antler length (cm)"
par(mfrow=c(1,3))
hist(dat$antler, xlab=use.xlab)
plot(density(dat$antler, from=0), xlab=use.xlab)
plot(ecdf(dat$antler), xlab=use.xlab)
```



All three plots above show that the antler lengths probably follow a **normal distribution**. Notice how the PDF plot (center) looks like a very smoothed version of the histogram. This is because it basically is. A histogram shows how many observations fall into discrete ranges, or bins. The PDF of a distribution shows how likely every possible value is relative to other values. That is, the PDF is basically the heights of a histogram with bin widths = 0. The rightmost plot, the ECDF, shows what percentage of observations in a distribution are equal to or less than each value (this is the “cumulative” part of ECDF). If we wanted, we could estimate the actual CDF of the distribution. The PDF and CDF are intimately related because the PDF is the derivative of the CDF (or, the CDF is the integral of the PDF). This is why the *y*-axis values on a PDF plot look so weird: they are the instantaneous rate of change in the CDF at each value.

Don’t worry if that doesn’t make a lot of sense right now. We will explore data distributions some more later in the course, both in practical terms and in the language of PDFs and CDFs.

2.4.3 Transform data

The plots above show that the antler lengths probably follow a normal distribution, which is convenient for statistical analysis. In fact, many statistical methods were developed to work only with normally distributed data! But, there is a problem. The normal distribution can take on any real value. In statistical jargon, we say that the normal distribution is supported on the interval $[-\infty, +\infty]$. A normally distributed variable can thus take on negative values. Does this make sense for antler lengths? Or for any kind of length? Of course not.

One way to avoid the awkwardness of a statistical model that predicts negative lengths is to log-transform the variables. This ensures that any value predicted by the model must be positive. We are actually going to log-transform both the response and predictor variable for reasons that will become clear in the next step. In R this can be done with the `log()` command:

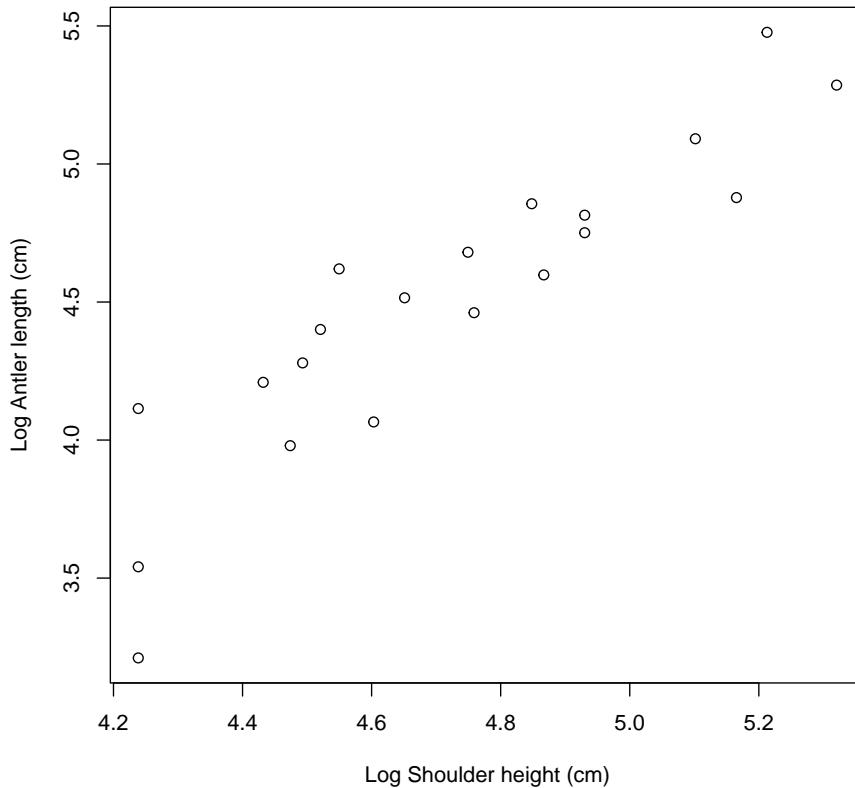
```
dat$logy <- log(dat$antler)
dat$logx <- log(dat$height)
```

Notice that we used the natural logarithm (base e) rather than \log_{10} . This will matter later when we plot the model and its predictions.

2.4.4 Analyze data

Now that we have log-transformed data, we should plot them to help us decide on an appropriate statistical model. We can make a new version of the scatterplot from earlier. Notice that the axes are now on a different scale, the logarithm of the original values. So, the x -coordinates around 4.9 on the log scale correspond to shoulder heights of $e^{4.9}$, or about 134.3 cm.

```
par(mfrow=c(1,1)) # reset plot layout
plot(dat$logx, dat$logy,
     xlab="Log Shoulder height (cm)",
     ylab="Log Antler length (cm)")
```



It looks like antler length and shoulder height have a linear relationship on the log scale. That is exactly what we are going to fit, but we need to understand what this represents. The linear model we can fit with the log-transformed data is:

$$\log(Y) = \beta_0 + \beta_1 \log(X) + \varepsilon$$

In this equation:

- Y is the response or dependent variable (antler length)
- X is the explanatory or independent variable (shoulder height)
- β_0 is the y -intercept (i.e., the value of $\log(Y)$ when $\log(X) = 0$). Called “beta zero” or “beta naught”.
- β_1 is the slope or regression coefficient (i.e., the change in $\log(Y)$ per unit change in $\log(X)$). If $\log(X)$ increases by 1, then $\log(Y)$ increases by β_1 . Called “beta one”.
- ε is a random error term that describes residual variation not explained by the model. Called “epsilon”. In a linear regression model, residuals are identically and independently distributed (*i.i.d.*) according to a normal distribution with mean 0 and variance σ^2 . Almost all statistical models assume *i.i.d.* residuals, which means that errors all come from the same distribution, and are completely independent of each other.

If we wanted to estimate actual antler lengths rather than the logarithm of antler lengths, we have to exponentiate both sides. Ignoring the residuals term for the moment, this gives us:

$$Y = e^{\beta_0} X^{\beta_1}$$

Which is more commonly written as a **power law**. Power laws are very common in anatomy and morphology. The equation below is the usual form of a power law, where the coefficient $a = e^{\beta_0}$.

$$Y = aX^b$$

The linear model is fit using the `lm()` function. Notice that the model terms are specified as a “formula”. The response variable is on the left, then a `~`, then the predictor variable (or variables) on the right. The model is saved to an object called `mod1`, short for “model 1”.

```
mod1 <- lm(logy~logx, data=dat)
```

We can view the results using the `summary()` command on the output:

```
summary(mod1)
```

```

## 
## Call:
## lm(formula = logy ~ logx, data = dat)
## 
## Residuals:
##      Min       1Q   Median       3Q      Max
## -0.55752 -0.12729  0.00769  0.15201  0.38060
## 
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)    
## (Intercept) -2.6476    0.7918  -3.344  0.00361 **  
## logx         1.5138    0.1675   9.037 4.14e-08 *** 
## ---        
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
## 
## Residual standard error: 0.24 on 18 degrees of freedom
## Multiple R-squared:  0.8194, Adjusted R-squared:  0.8094 
## F-statistic: 81.67 on 1 and 18 DF,  p-value: 4.141e-08

```

This tells us that the intercept and slope are statistically significant ($P < 0.05$ for both terms), and that the model explains about 81% of variation in antler length (adjusted $R^2 = 0.8094$). That's pretty good.

What if we had not transformed the data? Then we would have had to fit the power law directly. This is possible in R using the `nls()` function (nonlinear least squares), but this route is usually harder than using `lm()` on transformed data. Here is the equivalent model fit using the nonlinear model function `nls()`:

```

mod2 <- nls(antler~a*height^b, data=dat,
            start=list(a=exp(-2.6), b=1.51))
summary(mod2)

## 
## Formula: antler ~ a * height^b
## 
## Parameters:
##             Estimate Std. Error t value Pr(>|t|)    
## a          0.13219   0.09861   1.341   0.197    
## b          1.38863   0.14929   9.302 2.69e-08 *** 
## ---        
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
## 
## Residual standard error: 21.49 on 18 degrees of freedom
## 
## Number of iterations to convergence: 5
## Achieved convergence tolerance: 8.994e-07

```

Fitting nonlinear models in R can be tricky because you usually need to supply

starting values for the algorithm to try. In the example above I used the estimates from the linearized version of the model as starting values. If the starting values are too far away from the “correct” values then the model may not converge, leaving you without a model fit. Even if the model does converge, there is no guarantee that the fitted parameters are correct because there might be multiple stable solutions. Furthermore, some of the common postprocessing done on fitted models such as multiple inference, model comparison, prediction, and error propagation are harder to do in R with nonlinear models (`nls()` outputs) than with linear models (`lm()` outputs). We’ll explore how to do this later in the course, but for now we will stick with the linearized fit.

2.4.5 Write out results

Now that we have successfully fit a statistical model, we need to report our findings. Reporting model coefficients is easy enough: just put them into a table.

Table 1. Log-transformed antler length (cm) varied as a linear function of log-transformed shoulder height (cm). Model $R^2 = 0.8094$. Parameters shown are estimate \pm standard error (SE); t is t -statistic, and P is P -value.

Parameter	Estimate \pm SE	t	P
$\log(\beta_0)$	-2.65 ± 0.79	-3.34	0.0036
β_1	1.51 ± 0.17	9.04	<0.0001

We might also want to present the fitted model in context with the original data. In R this is done using **predicted values** and their associated uncertainties. Instead of presenting predicted values for the original data, it is common practice to present predicted values for a smooth set of predictor values within the domain of the original data. We can see the minimum and maximum values of the predictor with `range()`. Then, we use `seq()` to generate a regular sequence between those limits. Finally, calculate the predictions using `predict()`, with argument `se.fit` to get the uncertainty associated with the predictions.

```
n <- 100
new.x <- seq(from=min(dat$logx), to=max(dat$logx), length=n)
new.x2 <- exp(new.x) # needed for plot later
pred <- predict(mod1,
                 newdata=data.frame(logx=new.x),
                 se.fit=TRUE)
```

The last piece we will need is the 95% **confidence limits** of the predictions. We can calculate these from the predicted means and SE in object `pred`. Because the predictions are on the logarithmic scale, we will back-transform the predictions. We can get the back-transformed predictions by exponentiating the predicted mean, and by requesting quantiles from the lognormal distribution with function

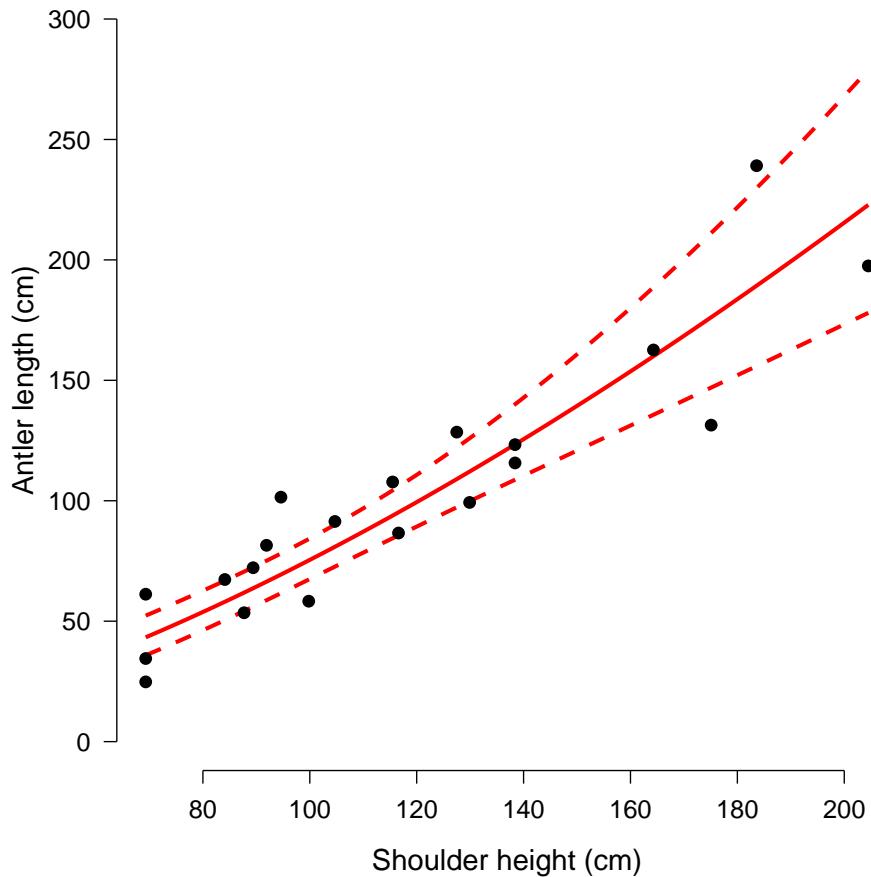
`qlnorm()`. To get the 95% CI, we request the 2.5% and 97.5% percentiles (i.e., 0.025 and 0.975 quantiles) because the interval between them covers 95% of the distribution.

```
mn <- exp(pred$fit)
lo <- qlnorm(0.025, pred$fit, pred$se.fit)
hi <- qlnorm(0.975, pred$fit, pred$se.fit)

# not run: alternative (sometimes more reliable)
#           way to get predicted mean/median
# mn <- qlnorm(0.5, pred$fit, pred$se.fit)
```

Now we have all the pieces we need to put together a polished, manuscript-quality figure with the original data and the model predictions. We will use the `par()` command again to set some additional graphics options.

```
par(mfrow=c(1,1),                      # 1 x 1 layout
    mar=c(4.1, 4.1, 1.1, 1.1), # margin sizes (clockwise from bottom)
    bty="n",                  # no box around plot
    las=1,                    # axis labels in reading direction
    lend=1,                   # flat line ends
    cex.lab=1.3,              # axis title size
    cex.axis=1.2)              # axis label size
plot(dat$height, dat$antler, type="n",
     xlab="Shoulder height (cm)",
     ylab="Antler length (cm)",
     ylim=c(0, 300))
points(new.x2, lo, type="l", lwd=3, col="red", lty=2)
points(new.x2, hi, type="l", lwd=3, col="red", lty=2)
points(new.x2, mn, type="l", lwd=3, col="red")
points(dat$height, dat$antler, cex=1.3, pch=16)
```



There are some other graphics options we could set, but we'll save those for later in the course. This figure is clean and spare, with a very high data to ink ratio... just the sort of figure that we want to represent our work (Tufte 2001).

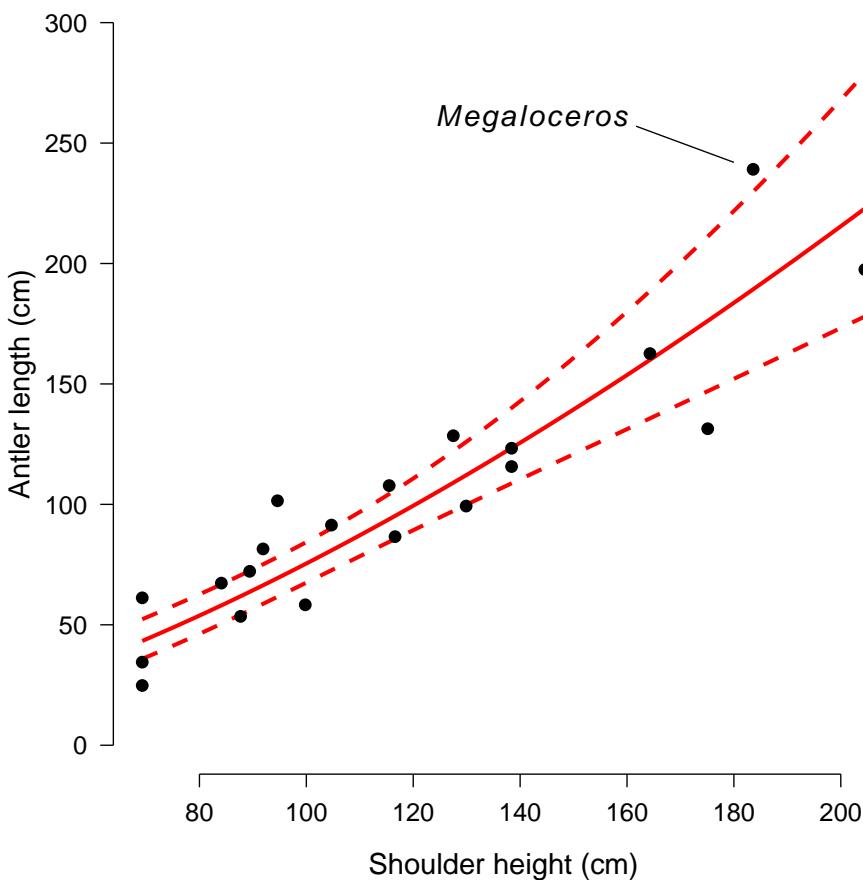
Just for fun, let's label the original object of our investigation, the Irish elk. We can add a text label and a pointer with the commands `text()` and `segments()`. This can require some fiddling to get the coordinates just right. First we'll use function `locator()` to find some appropriate coordinates. To use `locator()`, first run that command in the console. Then, click the points on the graph where you want coordinates. Then, right-click and select "Stop". The coordinates you clicked will be printed to the console.

```
# not run:
#locator()
# click on the figure where you want arrow ends to be
# $x
# [1] 161.7056 177.5281
```

```
# $y
# [1] 257.0432 244.5633
```

Then, use the coordinates you found and add the pointer and text. Of course you could do this in another program like Powerpoint, but keeping everything in R can streamline your workflow.

```
# add pointer and genus name:
segments(161.7, 257.0, 180, 242)
text(160, 260,
     expression(italic(Megaloceros)),
     adj=1,    # right-aligned at point
     cex=1.3) # text size
```



2.4.6 Save your work?

Now that we are done with the analysis, we can save our work. If you have the code in an R script, you can just save the script and rerun it whenever you want. The results and outputs will be exactly the same next time you open a new R console and run the script⁸. R scripts have the extension **.r** or **.R**.

If you want to save the actual R workspace, you can do this in the console with the `save.image()` command. By default, this command saves the workspace in your home directory. R workspaces have the extension **.RData**.

```
save.image("elk_analysis_2021-06-14.RData")
```

The resulting file is the R workspace created by running the script. If you open it, a new R instance will open that contains the same objects and command history as the original. This is more useful in situations involving calculations that take a long time to run, or when the outputs are more complicated than simple text files or images. Notice that the workspace was saved with an informative name and a date. This is important because without the user-supplied name, R will save the workspace with a generic name... potentially overwriting a previously saved workspace!

2.4.7 What's next?

This section was designed to demonstrate a typical R workflow with straightforward examples and good coding practices. In the next section, we'll take a closer look at how R works. Some the material in the upcoming sections was already demonstrated, but not explained, on this page.

2.5 Write and execute commands in the R console

Now that you've survived your first R adventure, it's time to take a closer look at how R works. In this section we'll focus on getting familiar with the basic syntax of the R language and how to enter R commands.

2.5.1 R commands–basics

Open up a new R instance or RStudio session and take a look at the **console**. Here commands can be entered at the command prompt, **>**. The ENTER or RETURN key will execute the current command and print the results (if any) to the console. Try a simple command like **2+2**. Notice how the output prints to the console, with a **[1]** at the start of the line. The **[1]** signifies that that line

⁸This is true if there are no random elements in your code. If you draw any random numbers, you will need to set the random number seed in your script to ensure exact reproducibility.

starts with the first element of the output. This is to help read outputs with lots of elements...try the command `rnorm(100)` to see why.

2+2

```
## [1] 4
rnorm(100)

## [1] 0.1868067548 1.2953417219 1.3496341764 -1.8199058851 -1.0129597934
## [6] 0.7758079984 -0.2446729873 -0.7628311610 1.1871287322 0.5687341349
## [11] 0.2515907763 -1.3367505458 -1.1523874739 2.4543256127 0.1869963139
## [16] -0.4393163702 -0.0591246386 2.3218882814 0.6452875006 0.3217042164
## [21] -1.0585980833 -1.8577768116 0.2816177047 -0.2696880316 0.6292340639
## [26] -2.0491752819 0.3865173066 0.9388559144 0.9614268935 -1.2941535699
## [31] 0.1987585317 1.3948937976 -0.4443539013 -0.3467818080 -1.3394891028
## [36] 0.3248352500 0.3453442934 1.0882126165 -1.0850969947 1.1509566751
## [41] -1.2319974844 1.3428871601 -0.9701891498 1.9068274828 0.3370097559
## [46] -1.3898744374 -0.4298016089 -0.0442773529 -0.2836812472 -0.0520081219
## [51] -1.0893354258 -0.5703620158 1.1049930829 1.2552194034 -1.4890854621
## [56] 1.3473576941 0.0748210623 -1.6124316377 0.4919105726 0.1515581094
## [61] 0.5180248518 -0.4768916424 0.1350865322 0.1408681220 0.6483650912
## [66] 0.2824644386 -0.0001100443 -0.8806680807 -2.3269645388 1.1982580677
## [71] -0.4832922566 0.5087733866 0.2542475936 -1.2960109532 0.5023409534
## [76] 1.0496202326 -0.2426459452 -1.4389497661 -0.9029431446 0.8612725843
## [81] -0.8210071585 -1.4689588806 0.1794126511 0.1425198141 -1.0560004007
## [86] -1.2274682135 -0.6413455301 0.2250168317 0.3148611203 -0.2958914087
## [91] -1.6643713237 1.8103941493 0.9552156749 -0.8964807728 -1.2136214600
## [96] 0.5029205644 0.8911613760 -0.4542943772 0.2878110215 -0.2596923043
```

Commands can be split over multiple lines. When that happens, a + will be seen at the left side of the window instead of a >. A split command will be executed once a complete expression is input. If you accidentally start a command and aren't sure how to finish it, you can exit and cancel the command with the ESCAPE key. R executes commands, not lines. You can have more than one command on a line separated by a semi-colon ;, but this generally frowned upon.

command split across lines:

2+

1

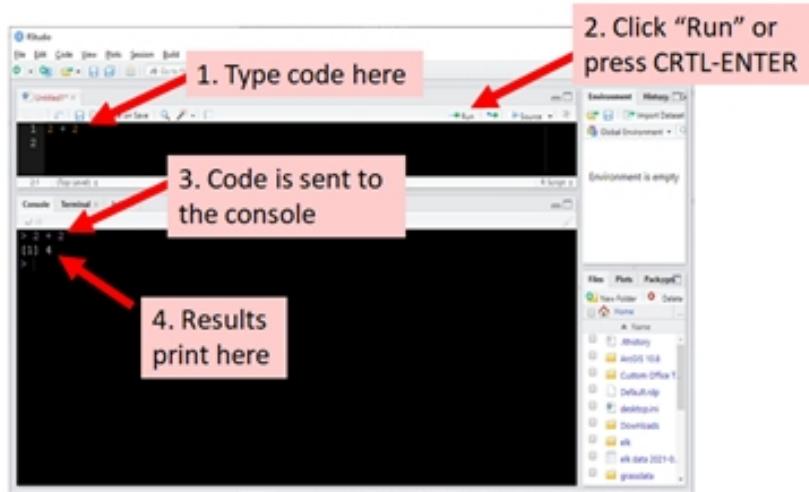
```
## [1] 3
# two commands on a line
2+2; 3+3
```

```
## [1] 4
```

```
## [1] 6
```

For this course you should be able to copy and paste most commands directly

from the page to your R console. You can also copy and paste commands to the script window and run them from there. This is useful for running multiple commands at once, or for writing more complicated programs. The image below shows this method in action:



2.5.2 Elements of R code

As you start to learn R, it is important that you save the code you write for later. This way you can refer back to how you solved problems in the past, and have a record of what you did. Sections of code, or whole files filled with code, that accomplish a specific task are referred to as scripts. Get in the habit of scripting everything. This not only documents your workflow, but makes your work reproducible, modifiable, and eventually more efficient. R scripts can even be called from R and executed, which has obvious implications for reproducibility and automation.

2.5.2.1 Comments

One of the most important elements of your scripts are **comments**. Comments are notes to yourself (or a future reader) that are not read by R. It is difficult to put too many comments in code. Comments are a mechanism for explaining what your code is doing and why. Some comments are just place markers (e.g., `# Analysis begins here`) to help you find important sections. Other comments might be more lengthy, holding explanations of why you chose to code a task the way that you did (e.g., `# condense with sapply() (maintains order of subunits)`). The latter kind of comment can be extremely valuable to your future self.

The comment symbol in R is `#`. Everything on a line after `#` is not interpreted by R.

```
# Here is a comment.
2+2      # Here is a comment on the end of a line.
```

Below is an example where the `a` command is broken across lines, with each line getting an explanatory comment. This is possible because R ignores **whitespace** (spaces, tabs, returns, etc.). Very complex functions or loops might benefit from this kind of commenting. Some programmers do not like this kind of comment... but you comment code the way that works for you.

```
rnorm(
  5,  # sample size
  10, # mean
  2   # sd
) #rnorm
```

Notice in the above example that the closing parenthesis for `rnorm()` is marked with a comment. This is a good practice if the open and close parentheses are separated by many lines of code. Most IDEs will highlight matching parentheses, but marking them yourself can potentially save some headache.

When writing comments you should keep in mind that most code editors don't do line breaks very well. So, you will need to break up your comment into multiple lines.

2.5.2.2 Objects

Everything in R is an **object** of some kind. When you import data, run an analysis, or even just when you do basic calculations, you are creating objects that contain those values. Most of R programming is just calling and manipulating objects in your workspace. That's sort of like saying that language is just calling and manipulating sounds.

The commands below show how to create an object by assignment (`<-`), and print the value assigned to an object to the console. The third command shows how the name of an object can be used in place of the value it holds. The fourth command demonstrates that even simple calculations return objects as results, even if the object is just printed to the console instead of saved.

```
a <- 3
a
## [1] 3
a+4
## [1] 7
class(a+4)
## [1] "numeric"
```

All objects have at least one **class**, and their values have a **type**. We'll explore some of the most important classes and types later. For now, the class of an

object is what kind of object it is; the type of a value is the kind of value. The class of an object or the type of a value determines how those things interact with R functions.

2.5.2.3 Functions

R code is built around using **functions** to manipulate objects (and functions are themselves a kind of object). Functions are called by their names, with the arguments to the function specified within parentheses. This is referred to as a function **call** or a **command**. The arguments of an R function can be named or not. Below are some examples using `rnorm()`, the function that draws random values from the normal distribution.

```
# commands with named arguments:
rnorm(n=10, mean=5, sd=2)
## [1] 5.951771 4.566881 5.216120 2.718580 5.581812 5.251499 2.496277 3.445045
## [9] 1.361433 2.033040
rnorm(mean=5, sd=2, n=10)
## [1] 3.177625 2.191203 3.969389 6.378298 4.807159 5.320653 4.335875 5.363562
## [9] 8.644716 3.552593
```

When the arguments of a function are not named, then R will assume which value to use for which argument. This decision is based on a default order of arguments. The default order can be determined from the help page for the function (accessed in this case by `?rnorm`). Below are two examples that call `rnorm()` with unnamed arguments. Notice how different the results are. This is because R expects unnamed arguments to be the number of values, the mean, and the standard deviation (SD), *in that order*.

```
rnorm(10,5,2)
## [1] 6.725276 8.163568 7.725576 4.101334 7.069661 3.312772 5.516937 6.931724
## [9] 1.116817 5.648863
rnorm(2,5,10)
## [1] 8.180806 14.923327
```

Most R functions require their arguments to be in the correct format (or type). For example, a function might require a number and not a character string. The example below illustrates this by providing the SD as a character string and not as a number. This line should return an error because it cannot run with an incorrectly formatted SD argument.

```
rnorm(10,5,"2")
## Error in rnorm(10, 5, "2"): invalid arguments
```

To recap, argument **order** is important when using functions. You can supply arguments to a function without naming them if you put them in the proper order. See the help files to see what the order is. *When in doubt, name your arguments.*

Functions expect arguments in a particular format. If a function is not working, you might have an argument formatted incorrectly. Sometimes it's helpful to look at each argument separately in the R console, outside of the function. To do this, copy/paste each argument to the R console separately and execute by hitting ENTER.

Some function arguments have default values, so you don't have to supply them. For example, `runif()`, which draws random numbers from the uniform distribution, defaults to the interval between 0 and 1. So, the two `runif()` commands below are equivalent because 0 and 1 are the defaults for the limits of the interval. The `set.seed()` command before each `runif()` command is to set the random number seed, ensuring that the results are identical. If you run the `runif()` commands without resetting the random number seed, the commands will return different results because of the random nature of the function.

```
set.seed(42)
runif(10,0,1)

## [1] 0.9148060 0.9370754 0.2861395 0.8304476 0.6417455 0.5190959 0.7365883
## [8] 0.1346666 0.6569923 0.7050648

# again, but with default arguments implied
set.seed(42)
runif(10)

## [1] 0.9148060 0.9370754 0.2861395 0.8304476 0.6417455 0.5190959 0.7365883
## [8] 0.1346666 0.6569923 0.7050648
```

2.5.3 The R workspace

When you use R, all objects and data that you use exist in the **R workspace**. Objects in the workspace can be called from the console. You can think of the workspace like a kind of sandbox or workbench.

Most of the time, when you start R this creates a new workspace with no objects. If you open a previously saved workspace, you will see the message **[Previously saved workspace restored]**. If you saved your R workspace without a unique name, it can be hard to figure out what workspace you are in. You can use the `ls()` command to see what is in the workspace, which might help you figure out what workspace has been loaded. If there are no objects in the workspace, the result of `ls()` will be `character(0)`. This output means that the `ls()` command returned a vector of text values (`character`) with 0 elements. In other words, the workspace contains no objects.

As you work, your workspace will fill up with objects. This isn't a big deal until you start to either have too many names to keep track of, or start to use too much memory. The first problem can be dealt with by commenting your code. The second problem can be solved by deleting unneeded objects to save

space. You can check to see how much memory your R objects are using with the following command:

```
sort(sapply(mget(ls()),object.size),decreasing=TRUE)
```

This will print the sizes of objects in memory in the current workspace, in descending order. The default units are bytes. If you want the results in Mb, divide the output by 1.024e6.

```
a <- sort(sapply(mget(ls()),object.size),decreasing=TRUE)
a/1.024e6
```

R objects can be removed using the `rm()` function. You can remove objects one at a time or by supplying a set of names. Note that the set of names is supplied to an argument called `list`, which is confusing because `list` is also the name of a special type of object in R.

```
# not run, but try on your machine
x <- 1
y <- 2
z <- 3
ls() # shows x, y, and z in workspace
rm(x)
ls() # x is gone
rm(list=c("y", "z"))
ls() # x, y, and z are gone
```

2.5.4 R code basics: assignment and operators

2.5.4.1 Assignment

The most important R function is **assignment**. When you **assign** a value to a variable, you can then use that variable in place of the value. Assigning a value to variable name will automatically create a new object in the workspace with that name. If an object with that name already exists, assigning a new value to that name will overwrite the old object.

The left arrow `<-` is the assignment operator. You can use `=` for assignment but really shouldn't because `=` is easily confused with the equality symbol `==`. Also, `=` is a mathematical symbol, but `<-` is unambiguously an R operation. You can also use `->`, but doing so usually makes your code harder to read rather than easier.

```
aa <- 4
bb <- 1:10
```

Don't do this:

```
cc = 5
```

Don't do this either.

```
5 -> cc
```

And definitely don't do this⁹:

```
a <- 5 -> b
```

You can chain multiple assignments together in a single command. This is useful if you need to quickly create several objects with the same structure:

```
list3 <- list2 <- list1 <- vector("list", 5)
```

Assignment can also be used to *change or overwrite parts of pre-existing objects*. Usually this involves the bracket notation. In the example below, the third element of object a (`a[3]`) is replaced with the number 42. We'll practice with the bracket notation in the data manipulation module.

```
a <- 1:5
```

```
a
```

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

```
a[3] <- 42
```

```
a
```

```
## [1] 1 2 42 4 5
```

One little-known feature of R assignment is that it can **resize** objects. For example, if you assign a value to the n -th element of a vector that has fewer than n elements, R will increase the length of the object to the size needed. It should be noted that this can be done accidentally as well as intentionally, so you need to be very careful if you resize objects using assignment. Personally, I don't do this because it is often clearer to change the size of objects in a separate command.

```
# resize by assignment (not recommended)
```

```
a <- 1:3
```

```
a[5] <- 5
```

```
a
```

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

```
# resize in separate command (preferred)
```

```
a <- 1:3
```

```
length(a) <- 5
```

```
a
```

```
## [1] 1 2 3 NA NA
```

```
a[5] <- 5
```

```
a
```

⁹It never even occurred to me to try this until I was prepping these course notes.

```
## [1] 1 2 3 NA 5
# resize by concatenation (preferred)
a <- 1:3
a <- c(a, 4:5)
a

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

2.5.4.2 R operators

Most math operators in R are similar to those in other languages/programs. For math expressions, the standard order of operations (PEMDAS) applies, but you can use parentheses if you want to be sure. The integer sequence operator `:` comes first, and comparison tests (`==`, `<`, `<=`, `>`, and `>=`) come last. The examples below show how many common R operators function, taking advantage of the fact that you can use an object's name in place of its value in most situations.

```
# make some values to work with
aa <- 5
bb <- 8

# multiply
aa*3
## [1] 15

# add
bb+3
## [1] 11

# square root
sqrt(aa)
## [1] 2.236068

# natural logarithm (base e)
log(aa)
## [1] 1.609438

# logarithm (arbitrary base)
log(aa, base=3)
## [1] 1.464974
log(aa, pi)
## [1] 1.405954

# common logarithm (base 10)
log10(aa)
## [1] 0.69897
```

```
# exponentiation with base e (antilog)
exp(aa)
## [1] 148.4132

# Vectorized (element-wise) multiplication
bb*2:5
## [1] 16 24 32 40

# Exponentiation
bb^2
## [1] 64

# Alternate symbol for exponentiation (rare)
bb**2
## [1] 64

# Remainder (modulus)
bb %% 3
## [1] 2

# Quotient without remainder
bb %/% 3
## [1] 2

# Order of operations
aa+2*3
## [1] 11

# Order of operations
aa+(2*3)
## [1] 11

# Order of operations
(aa+2)*3
## [1] 21

# Sequence of integers
1:10
## [1] 1 2 3 4 5 6 7 8 9 10

# Note that ":" comes first!
1:10*10
## [1] 10 20 30 40 50 60 70 80 90 100

# Comparisons are last
```

```
2*2 > 1:5*2
## [1] TRUE FALSE FALSE FALSE FALSE
```

Some of these examples illustrate another key concept in R programming: vectorization. **Vectorization** refers to how many R functions can operate on sets of numbers (vectors) just as well as on single values (scalars). The function will operate on every element of the vector in order. Most R functions are vectorized to some extent. This is very handy because many operations in statistics deal with vectors of values.

Vectorization can cause headaches if the vectors are not of the same length. If the length of one vector is a multiple of the length of the other, then the shorter vector gets recycled. This is sometimes called the “recycling rule”. If the length of the longer vector is not a multiple of the length of the shorter vector, the shorter one will be recycled a non-integer number of times and R will return a warning.

```
# lengths compatible:
x <- 1:2
y <- 3:6
x+y

## [1] 4 6 6 8

# lengths not compatible:
a <- 1:3
b <- 4:5
a + b

## Warning in a + b: longer object length is not a multiple of shorter object
## length

## [1] 5 7 7
```

2.5.4.3 Selecting with brackets []

The most common way to select parts of R objects is with the **bracket notation**. The square bracket symbols [] are used to define subsets of an object. Within the brackets, pieces of an object are selected by the indices of the elements to be selected. Unlike many programming languages, indices in R start at 1 (instead of 0). The example below shows how brackets work on a vector of numeric values.

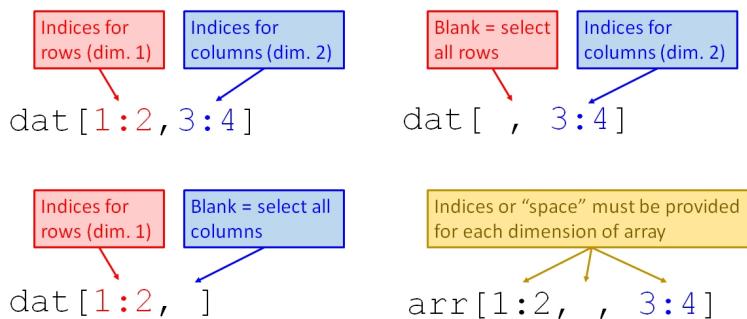
```
a <- 3:8
a[1]    # first element of a
## [1] 3
a[1:3]  # first through third elements
## [1] 3 4 5
a[c(2,4)] # second and fourth elements
## [1] 4 6
```

Remember, everything in R is an object. You can save the output from one of the commands above as an object:

```
b <- a[c(2,4)]
b
```

```
## [1] 4 6
```

Brackets work on more complicated objects as well. The figure below shows how brackets work on a matrix, data frame, or array, which have 2, 2, and ≥ 3 dimensions, respectively.



The examples below select parts of the `iris` dataset. The `iris` dataset is a data frame, one of the most important types of objects in R. Unlike the vector `a` above, `iris` has two dimensions: **rows** and **columns**. In the brackets, the dimensions are given separated by a comma. You *must* include the comma which separates the dimensions, even if you don't supply a value for one of the dimensions. Supplying the wrong number of dimensions will either cause an error (good) or something unexpected (bad). A missing value for rows or columns will select all rows or columns.

```
# first 4 rows, all columns of iris
iris[1:4, ]
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1       3.5      1.4       0.2  setosa
## 2          4.9       3.0      1.4       0.2  setosa
## 3          4.7       3.2      1.3       0.2  setosa
## 4          4.6       3.1      1.5       0.2  setosa
```

```
# first 3 columns, all rows of iris
iris[, 1:3]
```

```
##   Sepal.Length Sepal.Width Petal.Length
## 1          5.1       3.5      1.4
## 2          4.9       3.0      1.4
## 3          4.7       3.2      1.3
```

## 4	4.6	3.1	1.5
## 5	5.0	3.6	1.4
## 6	5.4	3.9	1.7
## 7	4.6	3.4	1.4
## 8	5.0	3.4	1.5
## 9	4.4	2.9	1.4
## 10	4.9	3.1	1.5
## 11	5.4	3.7	1.5
## 12	4.8	3.4	1.6
## 13	4.8	3.0	1.4
## 14	4.3	3.0	1.1
## 15	5.8	4.0	1.2
## 16	5.7	4.4	1.5
## 17	5.4	3.9	1.3
## 18	5.1	3.5	1.4
## 19	5.7	3.8	1.7
## 20	5.1	3.8	1.5
## 21	5.4	3.4	1.7
## 22	5.1	3.7	1.5
## 23	4.6	3.6	1.0
## 24	5.1	3.3	1.7
## 25	4.8	3.4	1.9
## 26	5.0	3.0	1.6
## 27	5.0	3.4	1.6
## 28	5.2	3.5	1.5
## 29	5.2	3.4	1.4
## 30	4.7	3.2	1.6
## 31	4.8	3.1	1.6
## 32	5.4	3.4	1.5
## 33	5.2	4.1	1.5
## 34	5.5	4.2	1.4
## 35	4.9	3.1	1.5
## 36	5.0	3.2	1.2
## 37	5.5	3.5	1.3
## 38	4.9	3.6	1.4
## 39	4.4	3.0	1.3
## 40	5.1	3.4	1.5
## 41	5.0	3.5	1.3
## 42	4.5	2.3	1.3
## 43	4.4	3.2	1.3
## 44	5.0	3.5	1.6
## 45	5.1	3.8	1.9
## 46	4.8	3.0	1.4
## 47	5.1	3.8	1.6
## 48	4.6	3.2	1.4
## 49	5.3	3.7	1.5

```
## 50      5.0      3.3      1.4
## 51      7.0      3.2      4.7
## 52      6.4      3.2      4.5
## 53      6.9      3.1      4.9
## 54      5.5      2.3      4.0
## 55      6.5      2.8      4.6
## 56      5.7      2.8      4.5
## 57      6.3      3.3      4.7
## 58      4.9      2.4      3.3
## 59      6.6      2.9      4.6
## 60      5.2      2.7      3.9
## 61      5.0      2.0      3.5
## 62      5.9      3.0      4.2
## 63      6.0      2.2      4.0
## 64      6.1      2.9      4.7
## 65      5.6      2.9      3.6
## 66      6.7      3.1      4.4
## 67      5.6      3.0      4.5
## 68      5.8      2.7      4.1
## 69      6.2      2.2      4.5
## 70      5.6      2.5      3.9
## 71      5.9      3.2      4.8
## 72      6.1      2.8      4.0
## 73      6.3      2.5      4.9
## 74      6.1      2.8      4.7
## 75      6.4      2.9      4.3
## 76      6.6      3.0      4.4
## 77      6.8      2.8      4.8
## 78      6.7      3.0      5.0
## 79      6.0      2.9      4.5
## 80      5.7      2.6      3.5
## 81      5.5      2.4      3.8
## 82      5.5      2.4      3.7
## 83      5.8      2.7      3.9
## 84      6.0      2.7      5.1
## 85      5.4      3.0      4.5
## 86      6.0      3.4      4.5
## 87      6.7      3.1      4.7
## 88      6.3      2.3      4.4
## 89      5.6      3.0      4.1
## 90      5.5      2.5      4.0
## 91      5.5      2.6      4.4
## 92      6.1      3.0      4.6
## 93      5.8      2.6      4.0
## 94      5.0      2.3      3.3
## 95      5.6      2.7      4.2
```

## 96	5.7	3.0	4.2
## 97	5.7	2.9	4.2
## 98	6.2	2.9	4.3
## 99	5.1	2.5	3.0
## 100	5.7	2.8	4.1
## 101	6.3	3.3	6.0
## 102	5.8	2.7	5.1
## 103	7.1	3.0	5.9
## 104	6.3	2.9	5.6
## 105	6.5	3.0	5.8
## 106	7.6	3.0	6.6
## 107	4.9	2.5	4.5
## 108	7.3	2.9	6.3
## 109	6.7	2.5	5.8
## 110	7.2	3.6	6.1
## 111	6.5	3.2	5.1
## 112	6.4	2.7	5.3
## 113	6.8	3.0	5.5
## 114	5.7	2.5	5.0
## 115	5.8	2.8	5.1
## 116	6.4	3.2	5.3
## 117	6.5	3.0	5.5
## 118	7.7	3.8	6.7
## 119	7.7	2.6	6.9
## 120	6.0	2.2	5.0
## 121	6.9	3.2	5.7
## 122	5.6	2.8	4.9
## 123	7.7	2.8	6.7
## 124	6.3	2.7	4.9
## 125	6.7	3.3	5.7
## 126	7.2	3.2	6.0
## 127	6.2	2.8	4.8
## 128	6.1	3.0	4.9
## 129	6.4	2.8	5.6
## 130	7.2	3.0	5.8
## 131	7.4	2.8	6.1
## 132	7.9	3.8	6.4
## 133	6.4	2.8	5.6
## 134	6.3	2.8	5.1
## 135	6.1	2.6	5.6
## 136	7.7	3.0	6.1
## 137	6.3	3.4	5.6
## 138	6.4	3.1	5.5
## 139	6.0	3.0	4.8
## 140	6.9	3.1	5.4
## 141	6.7	3.1	5.6

```

## 142      6.9      3.1      5.1
## 143      5.8      2.7      5.1
## 144      6.8      3.2      5.9
## 145      6.7      3.3      5.7
## 146      6.7      3.0      5.2
## 147      6.3      2.5      5.0
## 148      6.5      3.0      5.2
## 149      6.2      3.4      5.4
## 150      5.9      3.0      5.1

# rows 12-16 and columns 1-4
iris[12:16, 1:4]

##   Sepal.Length Sepal.Width Petal.Length Petal.Width
## 12      4.8      3.4      1.6      0.2
## 13      4.8      3.0      1.4      0.1
## 14      4.3      3.0      1.1      0.1
## 15      5.8      4.0      1.2      0.2
## 16      5.7      4.4      1.5      0.4

```

The bracket notation can be extremely powerful when logical tests are used instead of supplying numbers directly. Any R expression or command that evaluates to numeric values can be supplied inside the brackets. This is how I usually subset my datasets.

```

flag1 <- which(iris$Species == "setosa")
flag2 <- which(iris$Sepal.Length > 5.4)
iris[flag1,]                                # meets first condition

##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1      5.1      3.5      1.4      0.2  setosa
## 2      4.9      3.0      1.4      0.2  setosa
## 3      4.7      3.2      1.3      0.2  setosa
## 4      4.6      3.1      1.5      0.2  setosa
## 5      5.0      3.6      1.4      0.2  setosa
## 6      5.4      3.9      1.7      0.4  setosa
## 7      4.6      3.4      1.4      0.3  setosa
## 8      5.0      3.4      1.5      0.2  setosa
## 9      4.4      2.9      1.4      0.2  setosa
## 10     4.9      3.1      1.5      0.1  setosa
## 11     5.4      3.7      1.5      0.2  setosa
## 12     4.8      3.4      1.6      0.2  setosa
## 13     4.8      3.0      1.4      0.1  setosa
## 14     4.3      3.0      1.1      0.1  setosa
## 15     5.8      4.0      1.2      0.2  setosa
## 16     5.7      4.4      1.5      0.4  setosa
## 17     5.4      3.9      1.3      0.4  setosa
## 18     5.1      3.5      1.4      0.3  setosa

```

```

## 19      5.7      3.8      1.7      0.3  setosa
## 20      5.1      3.8      1.5      0.3  setosa
## 21      5.4      3.4      1.7      0.2  setosa
## 22      5.1      3.7      1.5      0.4  setosa
## 23      4.6      3.6      1.0      0.2  setosa
## 24      5.1      3.3      1.7      0.5  setosa
## 25      4.8      3.4      1.9      0.2  setosa
## 26      5.0      3.0      1.6      0.2  setosa
## 27      5.0      3.4      1.6      0.4  setosa
## 28      5.2      3.5      1.5      0.2  setosa
## 29      5.2      3.4      1.4      0.2  setosa
## 30      4.7      3.2      1.6      0.2  setosa
## 31      4.8      3.1      1.6      0.2  setosa
## 32      5.4      3.4      1.5      0.4  setosa
## 33      5.2      4.1      1.5      0.1  setosa
## 34      5.5      4.2      1.4      0.2  setosa
## 35      4.9      3.1      1.5      0.2  setosa
## 36      5.0      3.2      1.2      0.2  setosa
## 37      5.5      3.5      1.3      0.2  setosa
## 38      4.9      3.6      1.4      0.1  setosa
## 39      4.4      3.0      1.3      0.2  setosa
## 40      5.1      3.4      1.5      0.2  setosa
## 41      5.0      3.5      1.3      0.3  setosa
## 42      4.5      2.3      1.3      0.3  setosa
## 43      4.4      3.2      1.3      0.2  setosa
## 44      5.0      3.5      1.6      0.6  setosa
## 45      5.1      3.8      1.9      0.4  setosa
## 46      4.8      3.0      1.4      0.3  setosa
## 47      5.1      3.8      1.6      0.2  setosa
## 48      4.6      3.2      1.4      0.2  setosa
## 49      5.3      3.7      1.5      0.2  setosa
## 50      5.0      3.3      1.4      0.2  setosa

iris[intersect(flag1, flag2),]    # meets both conditions

##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 15      5.8      4.0      1.2      0.2  setosa
## 16      5.7      4.4      1.5      0.4  setosa
## 19      5.7      3.8      1.7      0.3  setosa
## 34      5.5      4.2      1.4      0.2  setosa
## 37      5.5      3.5      1.3      0.2  setosa

iris[union(flag1, flag2),]        # meets either condition

##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1       5.1      3.5      1.4      0.2  setosa
## 2       4.9      3.0      1.4      0.2  setosa

```

```
## 3      4.7      3.2      1.3      0.2    setosa
## 4      4.6      3.1      1.5      0.2    setosa
## 5      5.0      3.6      1.4      0.2    setosa
## 6      5.4      3.9      1.7      0.4    setosa
## 7      4.6      3.4      1.4      0.3    setosa
## 8      5.0      3.4      1.5      0.2    setosa
## 9      4.4      2.9      1.4      0.2    setosa
## 10     4.9      3.1      1.5      0.1    setosa
## 11     5.4      3.7      1.5      0.2    setosa
## 12     4.8      3.4      1.6      0.2    setosa
## 13     4.8      3.0      1.4      0.1    setosa
## 14     4.3      3.0      1.1      0.1    setosa
## 15     5.8      4.0      1.2      0.2    setosa
## 16     5.7      4.4      1.5      0.4    setosa
## 17     5.4      3.9      1.3      0.4    setosa
## 18     5.1      3.5      1.4      0.3    setosa
## 19     5.7      3.8      1.7      0.3    setosa
## 20     5.1      3.8      1.5      0.3    setosa
## 21     5.4      3.4      1.7      0.2    setosa
## 22     5.1      3.7      1.5      0.4    setosa
## 23     4.6      3.6      1.0      0.2    setosa
## 24     5.1      3.3      1.7      0.5    setosa
## 25     4.8      3.4      1.9      0.2    setosa
## 26     5.0      3.0      1.6      0.2    setosa
## 27     5.0      3.4      1.6      0.4    setosa
## 28     5.2      3.5      1.5      0.2    setosa
## 29     5.2      3.4      1.4      0.2    setosa
## 30     4.7      3.2      1.6      0.2    setosa
## 31     4.8      3.1      1.6      0.2    setosa
## 32     5.4      3.4      1.5      0.4    setosa
## 33     5.2      4.1      1.5      0.1    setosa
## 34     5.5      4.2      1.4      0.2    setosa
## 35     4.9      3.1      1.5      0.2    setosa
## 36     5.0      3.2      1.2      0.2    setosa
## 37     5.5      3.5      1.3      0.2    setosa
## 38     4.9      3.6      1.4      0.1    setosa
## 39     4.4      3.0      1.3      0.2    setosa
## 40     5.1      3.4      1.5      0.2    setosa
## 41     5.0      3.5      1.3      0.3    setosa
## 42     4.5      2.3      1.3      0.3    setosa
## 43     4.4      3.2      1.3      0.2    setosa
## 44     5.0      3.5      1.6      0.6    setosa
## 45     5.1      3.8      1.9      0.4    setosa
## 46     4.8      3.0      1.4      0.3    setosa
## 47     5.1      3.8      1.6      0.2    setosa
## 48     4.6      3.2      1.4      0.2    setosa
```

## 49	5.3	3.7	1.5	0.2	setosa
## 50	5.0	3.3	1.4	0.2	setosa
## 51	7.0	3.2	4.7	1.4	versicolor
## 52	6.4	3.2	4.5	1.5	versicolor
## 53	6.9	3.1	4.9	1.5	versicolor
## 54	5.5	2.3	4.0	1.3	versicolor
## 55	6.5	2.8	4.6	1.5	versicolor
## 56	5.7	2.8	4.5	1.3	versicolor
## 57	6.3	3.3	4.7	1.6	versicolor
## 59	6.6	2.9	4.6	1.3	versicolor
## 62	5.9	3.0	4.2	1.5	versicolor
## 63	6.0	2.2	4.0	1.0	versicolor
## 64	6.1	2.9	4.7	1.4	versicolor
## 65	5.6	2.9	3.6	1.3	versicolor
## 66	6.7	3.1	4.4	1.4	versicolor
## 67	5.6	3.0	4.5	1.5	versicolor
## 68	5.8	2.7	4.1	1.0	versicolor
## 69	6.2	2.2	4.5	1.5	versicolor
## 70	5.6	2.5	3.9	1.1	versicolor
## 71	5.9	3.2	4.8	1.8	versicolor
## 72	6.1	2.8	4.0	1.3	versicolor
## 73	6.3	2.5	4.9	1.5	versicolor
## 74	6.1	2.8	4.7	1.2	versicolor
## 75	6.4	2.9	4.3	1.3	versicolor
## 76	6.6	3.0	4.4	1.4	versicolor
## 77	6.8	2.8	4.8	1.4	versicolor
## 78	6.7	3.0	5.0	1.7	versicolor
## 79	6.0	2.9	4.5	1.5	versicolor
## 80	5.7	2.6	3.5	1.0	versicolor
## 81	5.5	2.4	3.8	1.1	versicolor
## 82	5.5	2.4	3.7	1.0	versicolor
## 83	5.8	2.7	3.9	1.2	versicolor
## 84	6.0	2.7	5.1	1.6	versicolor
## 86	6.0	3.4	4.5	1.6	versicolor
## 87	6.7	3.1	4.7	1.5	versicolor
## 88	6.3	2.3	4.4	1.3	versicolor
## 89	5.6	3.0	4.1	1.3	versicolor
## 90	5.5	2.5	4.0	1.3	versicolor
## 91	5.5	2.6	4.4	1.2	versicolor
## 92	6.1	3.0	4.6	1.4	versicolor
## 93	5.8	2.6	4.0	1.2	versicolor
## 95	5.6	2.7	4.2	1.3	versicolor
## 96	5.7	3.0	4.2	1.2	versicolor
## 97	5.7	2.9	4.2	1.3	versicolor
## 98	6.2	2.9	4.3	1.3	versicolor
## 100	5.7	2.8	4.1	1.3	versicolor

```

## 101      6.3      3.3      6.0      2.5 virginica
## 102      5.8      2.7      5.1      1.9 virginica
## 103      7.1      3.0      5.9      2.1 virginica
## 104      6.3      2.9      5.6      1.8 virginica
## 105      6.5      3.0      5.8      2.2 virginica
## 106      7.6      3.0      6.6      2.1 virginica
## 108      7.3      2.9      6.3      1.8 virginica
## 109      6.7      2.5      5.8      1.8 virginica
## 110      7.2      3.6      6.1      2.5 virginica
## 111      6.5      3.2      5.1      2.0 virginica
## 112      6.4      2.7      5.3      1.9 virginica
## 113      6.8      3.0      5.5      2.1 virginica
## 114      5.7      2.5      5.0      2.0 virginica
## 115      5.8      2.8      5.1      2.4 virginica
## 116      6.4      3.2      5.3      2.3 virginica
## 117      6.5      3.0      5.5      1.8 virginica
## 118      7.7      3.8      6.7      2.2 virginica
## 119      7.7      2.6      6.9      2.3 virginica
## 120      6.0      2.2      5.0      1.5 virginica
## 121      6.9      3.2      5.7      2.3 virginica
## 122      5.6      2.8      4.9      2.0 virginica
## 123      7.7      2.8      6.7      2.0 virginica
## 124      6.3      2.7      4.9      1.8 virginica
## 125      6.7      3.3      5.7      2.1 virginica
## 126      7.2      3.2      6.0      1.8 virginica
## 127      6.2      2.8      4.8      1.8 virginica
## 128      6.1      3.0      4.9      1.8 virginica
## 129      6.4      2.8      5.6      2.1 virginica
## 130      7.2      3.0      5.8      1.6 virginica
## 131      7.4      2.8      6.1      1.9 virginica
## 132      7.9      3.8      6.4      2.0 virginica
## 133      6.4      2.8      5.6      2.2 virginica
## 134      6.3      2.8      5.1      1.5 virginica
## 135      6.1      2.6      5.6      1.4 virginica
## 136      7.7      3.0      6.1      2.3 virginica
## 137      6.3      3.4      5.6      2.4 virginica
## 138      6.4      3.1      5.5      1.8 virginica
## 139      6.0      3.0      4.8      1.8 virginica
## 140      6.9      3.1      5.4      2.1 virginica
## 141      6.7      3.1      5.6      2.4 virginica
## 142      6.9      3.1      5.1      2.3 virginica
## 143      5.8      2.7      5.1      1.9 virginica
## 144      6.8      3.2      5.9      2.3 virginica
## 145      6.7      3.3      5.7      2.5 virginica
## 146      6.7      3.0      5.2      2.3 virginica
## 147      6.3      2.5      5.0      1.9 virginica

```

```

## 148      6.5      3.0      5.2      2.0  virginica
## 149      6.2      3.4      5.4      2.3  virginica
## 150      5.9      3.0      5.1      1.8  virginica
iris[setdiff(flag1, flag2),]      # meets neither condition

```

```

##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1      5.1      3.5      1.4      0.2  setosa
## 2      4.9      3.0      1.4      0.2  setosa
## 3      4.7      3.2      1.3      0.2  setosa
## 4      4.6      3.1      1.5      0.2  setosa
## 5      5.0      3.6      1.4      0.2  setosa
## 6      5.4      3.9      1.7      0.4  setosa
## 7      4.6      3.4      1.4      0.3  setosa
## 8      5.0      3.4      1.5      0.2  setosa
## 9      4.4      2.9      1.4      0.2  setosa
## 10     4.9      3.1      1.5      0.1  setosa
## 11     5.4      3.7      1.5      0.2  setosa
## 12     4.8      3.4      1.6      0.2  setosa
## 13     4.8      3.0      1.4      0.1  setosa
## 14     4.3      3.0      1.1      0.1  setosa
## 17     5.4      3.9      1.3      0.4  setosa
## 18     5.1      3.5      1.4      0.3  setosa
## 20     5.1      3.8      1.5      0.3  setosa
## 21     5.4      3.4      1.7      0.2  setosa
## 22     5.1      3.7      1.5      0.4  setosa
## 23     4.6      3.6      1.0      0.2  setosa
## 24     5.1      3.3      1.7      0.5  setosa
## 25     4.8      3.4      1.9      0.2  setosa
## 26     5.0      3.0      1.6      0.2  setosa
## 27     5.0      3.4      1.6      0.4  setosa
## 28     5.2      3.5      1.5      0.2  setosa
## 29     5.2      3.4      1.4      0.2  setosa
## 30     4.7      3.2      1.6      0.2  setosa
## 31     4.8      3.1      1.6      0.2  setosa
## 32     5.4      3.4      1.5      0.4  setosa
## 33     5.2      4.1      1.5      0.1  setosa
## 35     4.9      3.1      1.5      0.2  setosa
## 36     5.0      3.2      1.2      0.2  setosa
## 38     4.9      3.6      1.4      0.1  setosa
## 39     4.4      3.0      1.3      0.2  setosa
## 40     5.1      3.4      1.5      0.2  setosa
## 41     5.0      3.5      1.3      0.3  setosa
## 42     4.5      2.3      1.3      0.3  setosa
## 43     4.4      3.2      1.3      0.2  setosa
## 44     5.0      3.5      1.6      0.6  setosa

```

```

## 45      5.1      3.8      1.9      0.4  setosa
## 46      4.8      3.0      1.4      0.3  setosa
## 47      5.1      3.8      1.6      0.2  setosa
## 48      4.6      3.2      1.4      0.2  setosa
## 49      5.3      3.7      1.5      0.2  setosa
## 50      5.0      3.3      1.4      0.2  setosa

```

When using brackets you need to be careful to only supply indices that actually exist. R's behavior when invalid indices are requested depends on the type of object you are trying to extract from. Requesting invalid indices from an array (vectors, matrices, or arrays) will return NA or an error.

```

# NA:
x <- 1:5
x[6]

```

```

## [1] NA
# error:
x <- matrix(1:12, nrow=3)
x[7,]

```

```
## Error in x[7, ]: subscript out of bounds
```

Requesting a row from a data frame that does not exist will return a data frame with the same columns as the data frame, but with row full of NA. This feels like an error, but can be useful if you want to make a different version of a data frame with a similar structure.

```

x <- iris[1:5,]
x[6,]

```

```

##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## NA          NA          NA          NA          NA    <NA>
x[11:15,]

```

```

##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## NA          NA          NA          NA          NA    <NA>
## NA.1        NA          NA          NA          NA    <NA>
## NA.2        NA          NA          NA          NA    <NA>
## NA.3        NA          NA          NA          NA    <NA>
## NA.4        NA          NA          NA          NA    <NA>

```

Requesting a column that does not exist from a data frame will return an error:

```
x[,7]
```

```
## Error in `[, .data.frame` (x, , 7): undefined columns selected
```

We will go over brackets and their usage in much more detail in a later section.

2.6 Basic R data structures

In R, data are stored in **objects** with a well-defined structure. The structures of these objects can facilitate manipulating and retrieving your data. The structure of an R object is determined by its class. The format of data (i.e., the kinds of values contained in an object) is referred to as the data type.

For a solid introduction to R classes and data types, refer to An introduction to R, one of the official R help manuals. The book *Data manipulation with R* (Spector 2008) is also good. If you want to explore the odder and less benign aspects of R data structures, I recommend *The R Inferno* (accessed 2021-12-20).

2.6.1 Vectors

In R, as in mathematics, a **vector** is an ordered set of values. Vectors are a key concept in statistics and linear algebra. As a language designed primarily for statistics, R inherited a strong emphasis on vectors and their manipulation. Vectors are probably the simplest type of data object in R. Understanding them is the key to understanding other data objects in R, so we'll start with vectors.

A vector is also a special case of the idea of an **array**, which is a systematic arrangement of similar elements. A vector is just a one-dimensional array. A two-dimensional array is a **matrix**, and an array with ≥ 3 dimensions is just called an array. Vectors, matrices, and arrays in R can be a convenient way to store structured data that are all of the same type (numeric, logical, character, etc.).

The simplest kind of array is a vector. For example, the set [1, 2, 3, 4, 5] is a vector. It is different than [2, 3, 1, 5, 4] even though it contains the same values because the order is different. In R we could call either of those sets a “vector”. Note that we don't call them “lists” because a list is a special kind of object in R that is not synonymous with vector.

Vectors can be created by the function `vector()` as well as by functions specific to the type of value to be stored. The most common types are **numeric**, **logical**, and **character**. These hold numbers, logical values, and text, respectively. Notice that each type has a default value (0, FALSE, and "" (blank), respectively). Notice also that in each command below R prints the results to the console because the resulting vectors are not assigned to an object.

```
vector("numeric", 5)
## [1] 0 0 0 0 0
vector("logical", 5)
## [1] FALSE FALSE FALSE FALSE FALSE
vector("character", 5)
```

```
## [1] "" "" "" "" ""
# equivalent:
numeric(5)

## [1] 0 0 0 0 0
logical(5)

## [1] FALSE FALSE FALSE FALSE FALSE
character(5)

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

Creating vectors this way is cumbersome, but useful when programming complex tasks. It is often more efficient, from a memory usage standpoint, to create an object and fill it later than to create it and repeatedly change its size.

You can also create vectors by defining their contents. When you do this, R will guess at the type of vector based on the values you give it. These guesses are rarely problematic.

```
# numeric:
class(runif(10))

## [1] "numeric"
# integer, which is stored in numeric vectors
# despite being a different type:
class(1:5)

## [1] "integer"
is.numeric(1:5)

## [1] TRUE
```

Vectors can also be made by putting values together using the function “c” (short for “combine” or “concatenate”). Note that if you try to combine values of different types, R will convert all values to character strings.

```
my.vec <- c(1,2,4:8,10)
my.vec2 <- c("zebra","hippo","lion","rhino","rhino")
my.vec3 <- c(1,"hippo",1+1==3)
```

Each value within a vector is called an **element**. Elements of a vector are accessed by the bracket notation.

```
a <- runif(5)

# print to console
a
```

```
## [1] 0.90403139 0.13871017 0.98889173 0.94666823 0.08243756

# 3rd element
a[3]
## [1] 0.9888917

# elements 2-4
a[2:4]
## [1] 0.1387102 0.9888917 0.9466682

# elements other than 1
a[-1]
## [1] 0.13871017 0.98889173 0.94666823 0.08243756

# elements other than 1 and 2
a[-c(1:2)]
## [1] 0.98889173 0.94666823 0.08243756
```

You can inspect a vector by printing it to the console. Do this by typing the object's name into the console at the command prompt > and pressing ENTER.

my.vec

```
## [1] 1 2 4 5 6 7 8 10
```

The [1] at the start of the line with your results indicates that that line starts with the first element of the object. If an object runs over to multiple lines, each line will begin with the index of the first value printed on that line. Make your console window narrower and try this command:

1:50

```
## [1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
## [26] 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
```

Each element of a vector can have a name, set or accessed using the function `names()`. Element names can even be used to access elements of a vector.

```
a <- runif(3)
names(a) <- c("value1", "value2", "value3")
names(a)
## [1] "value1" "value2" "value3"
a
##   value1   value2   value3
## 0.5142118 0.3902035 0.9057381
a["value3"]
##   value3
## 0.9057381
```

Vectors can be extended using `c()` or by changing their length. Changing the length of a vector can also shorten it. If a vector is made longer than the number of values it has, the extra elements will be filled in with `NA`, which is the label R uses for blanks.

```
b <- 1:8
b <- c(b, 10)
b
## [1] 1 2 3 4 5 6 7 8 10
b <- 1:8
length(b) <- 10
b
## [1] 1 2 3 4 5 6 7 8 NA NA
b <- 1:8
length(b) <- 6
b
## [1] 1 2 3 4 5 6
```

As mentioned elsewhere, many R functions take vectors as input and use them one element at a time. This is very handy when working with data, because you will often need to perform the same operation on every observation. Like any R data object, you can use a vector in a function simply by using the name in place of the values:

```
my.vec <- 1:8
my.vec*2

## [1] 2 4 6 8 10 12 14 16
mean(my.vec)

## [1] 4.5
```

2.6.1.1 Atomic vectors

A final note about vectors: some R functions and their help pages refer to “atomic vectors”. This is a pedantic way to refer to vectors of length 1. A vector of length 1 cannot be split into subsets, so it is “atomic” in the literal sense.

2.6.2 Data frames

Most of the time when you work with data in R, you will work with objects of the **data frame** class. A data frame looks and acts a lot like a spreadsheet. One key aspect of data frames is that data frames can store data of more than one type, while vectors, matrices, and arrays cannot. The examples below use the built-in data frame `iris`.

Each **row** of a data frame usually corresponds to one **observation**. Each **column** contains the values of one **variable**. You can access data frame columns

by name with the \$ operator:

```
iris$Sepal.Length
```

```
## [1] 5.1 4.9 4.7 4.6 5.0 5.4 4.6 5.0 4.4 4.9 5.4 4.8 4.8 4.3 5.8 5.7 5.4 5.1
## [19] 5.7 5.1 5.4 5.1 4.6 5.1 4.8 5.0 5.0 5.2 5.2 4.7 4.8 5.4 5.2 5.5 4.9 5.0
## [37] 5.5 4.9 4.4 5.1 5.0 4.5 4.4 5.0 5.1 4.8 5.1 4.6 5.3 5.0 7.0 6.4 6.9 5.5
## [55] 6.5 5.7 6.3 4.9 6.6 5.2 5.0 5.9 6.0 6.1 5.6 6.7 5.6 5.8 6.2 5.6 5.9 6.1
## [73] 6.3 6.1 6.4 6.6 6.8 6.7 6.0 5.7 5.5 5.5 5.8 6.0 5.4 6.0 6.7 6.3 5.6 5.5
## [91] 5.5 6.1 5.8 5.0 5.6 5.7 5.7 6.2 5.1 5.7 6.3 5.8 7.1 6.3 6.5 7.6 4.9 7.3
## [109] 6.7 7.2 6.5 6.4 6.8 5.7 5.8 6.4 6.5 7.7 7.7 6.0 6.9 5.6 7.7 6.3 6.7 7.2
## [127] 6.2 6.1 6.4 7.2 7.4 7.9 6.4 6.3 6.1 7.7 6.3 6.4 6.0 6.9 6.7 6.9 5.8 6.8
## [145] 6.7 6.7 6.3 6.5 6.2 5.9
```

You can also access data frame columns by number or name with bracket notation:

```
iris[,1]
```

```
## [1] 5.1 4.9 4.7 4.6 5.0 5.4 4.6 5.0 4.4 4.9 5.4 4.8 4.8 4.3 5.8 5.7 5.4 5.1
## [19] 5.7 5.1 5.4 5.1 4.6 5.1 4.8 5.0 5.0 5.2 5.2 4.7 4.8 5.4 5.2 5.5 4.9 5.0
## [37] 5.5 4.9 4.4 5.1 5.0 4.5 4.4 5.0 5.1 4.8 5.1 4.6 5.3 5.0 7.0 6.4 6.9 5.5
## [55] 6.5 5.7 6.3 4.9 6.6 5.2 5.0 5.9 6.0 6.1 5.6 6.7 5.6 5.8 6.2 5.6 5.9 6.1
## [73] 6.3 6.1 6.4 6.6 6.8 6.7 6.0 5.7 5.5 5.5 5.8 6.0 5.4 6.0 6.7 6.3 5.6 5.5
## [91] 5.5 6.1 5.8 5.0 5.6 5.7 5.7 6.2 5.1 5.7 6.3 5.8 7.1 6.3 6.5 7.6 4.9 7.3
## [109] 6.7 7.2 6.5 6.4 6.8 5.7 5.8 6.4 6.5 7.7 7.7 6.0 6.9 5.6 7.7 6.3 6.7 7.2
## [127] 6.2 6.1 6.4 7.2 7.4 7.9 6.4 6.3 6.1 7.7 6.3 6.4 6.0 6.9 6.7 6.9 5.8 6.8
## [145] 6.7 6.7 6.3 6.5 6.2 5.9
```

```
iris[, "Petal.Width"]
```

```
## [1] 0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 0.2 0.2 0.1 0.1 0.2 0.4 0.4 0.3
## [19] 0.3 0.3 0.2 0.4 0.2 0.5 0.2 0.2 0.4 0.2 0.2 0.2 0.2 0.4 0.1 0.2 0.2 0.2
## [37] 0.2 0.1 0.2 0.2 0.3 0.3 0.2 0.6 0.4 0.3 0.2 0.2 0.2 0.2 1.4 1.5 1.5 1.3
## [55] 1.5 1.3 1.6 1.0 1.3 1.4 1.0 1.5 1.0 1.4 1.3 1.4 1.5 1.0 1.5 1.1 1.8 1.3
## [73] 1.5 1.2 1.3 1.4 1.4 1.7 1.5 1.0 1.1 1.0 1.2 1.6 1.5 1.6 1.5 1.3 1.3 1.3
## [91] 1.2 1.4 1.2 1.0 1.3 1.2 1.3 1.1 1.3 2.5 1.9 2.1 1.8 2.2 2.1 1.7 1.8
## [109] 1.8 2.5 2.0 1.9 2.1 2.0 2.4 2.3 1.8 2.2 2.3 1.5 2.3 2.0 2.0 1.8 2.1 1.8
## [127] 1.8 1.8 2.1 1.6 1.9 2.0 2.2 1.5 1.4 2.3 2.4 1.8 1.8 2.1 2.4 2.3 1.9 2.3
## [145] 2.5 2.3 1.9 2.0 2.3 1.8
```

Rows of a data frame are accessed by number with bracket notation. As seen above, you can also select using logical tests.

```
iris[1,]
## Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5         1.4         0.2  setosa
iris[which(iris$Species == "setosa"),]
## Sepal.Length Sepal.Width Petal.Length Petal.Width Species
```

## 1	5.1	3.5	1.4	0.2	setosa
## 2	4.9	3.0	1.4	0.2	setosa
## 3	4.7	3.2	1.3	0.2	setosa
## 4	4.6	3.1	1.5	0.2	setosa
## 5	5.0	3.6	1.4	0.2	setosa
## 6	5.4	3.9	1.7	0.4	setosa
## 7	4.6	3.4	1.4	0.3	setosa
## 8	5.0	3.4	1.5	0.2	setosa
## 9	4.4	2.9	1.4	0.2	setosa
## 10	4.9	3.1	1.5	0.1	setosa
## 11	5.4	3.7	1.5	0.2	setosa
## 12	4.8	3.4	1.6	0.2	setosa
## 13	4.8	3.0	1.4	0.1	setosa
## 14	4.3	3.0	1.1	0.1	setosa
## 15	5.8	4.0	1.2	0.2	setosa
## 16	5.7	4.4	1.5	0.4	setosa
## 17	5.4	3.9	1.3	0.4	setosa
## 18	5.1	3.5	1.4	0.3	setosa
## 19	5.7	3.8	1.7	0.3	setosa
## 20	5.1	3.8	1.5	0.3	setosa
## 21	5.4	3.4	1.7	0.2	setosa
## 22	5.1	3.7	1.5	0.4	setosa
## 23	4.6	3.6	1.0	0.2	setosa
## 24	5.1	3.3	1.7	0.5	setosa
## 25	4.8	3.4	1.9	0.2	setosa
## 26	5.0	3.0	1.6	0.2	setosa
## 27	5.0	3.4	1.6	0.4	setosa
## 28	5.2	3.5	1.5	0.2	setosa
## 29	5.2	3.4	1.4	0.2	setosa
## 30	4.7	3.2	1.6	0.2	setosa
## 31	4.8	3.1	1.6	0.2	setosa
## 32	5.4	3.4	1.5	0.4	setosa
## 33	5.2	4.1	1.5	0.1	setosa
## 34	5.5	4.2	1.4	0.2	setosa
## 35	4.9	3.1	1.5	0.2	setosa
## 36	5.0	3.2	1.2	0.2	setosa
## 37	5.5	3.5	1.3	0.2	setosa
## 38	4.9	3.6	1.4	0.1	setosa
## 39	4.4	3.0	1.3	0.2	setosa
## 40	5.1	3.4	1.5	0.2	setosa
## 41	5.0	3.5	1.3	0.3	setosa
## 42	4.5	2.3	1.3	0.3	setosa
## 43	4.4	3.2	1.3	0.2	setosa
## 44	5.0	3.5	1.6	0.6	setosa
## 45	5.1	3.8	1.9	0.4	setosa

## 46	4.8	3.0	1.4	0.3	setosa
## 47	5.1	3.8	1.6	0.2	setosa
## 48	4.6	3.2	1.4	0.2	setosa
## 49	5.3	3.7	1.5	0.2	setosa
## 50	5.0	3.3	1.4	0.2	setosa

Unlike matrices, rows and columns of data frames are something different. This is because under the surface, a data frame is really a list, and what looks like a column of a data frame is really an element of the list. So, a row of a data frame is really a new data frame with one row. When you work with data frames in R, the data frame class and its associated methods offer a convenient way to interact with the underlying list structure.

```
class(iris[1,])
```

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

```
class(iris[,1])
```

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

Practically, this means that if you select a column from a data frame you get a vector, and if you select a row (or rows) you get another data frame. So, if you want to operate on values within a row, you might need to use an `apply()` function.

```
mean(iris[1,1:4])          # apply function needed
```

```
## Warning in mean.default(iris[1, 1:4]): argument is not numeric or logical:  
## returning NA
```

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

```
apply(iris[1,1:4], 1, mean) # works!
```

```
##      1
```

```
## 2.55
```

```
sum(iris[1,1:4])          # works!
```

```
## [1] 10.2
```

Columns can be added to data frames by assigning them a value:

```
iris2 <- iris    # Make a spare copy
```

```
iris2$petal.area <- iris2$Petal.Length*iris2$Petal.Width
```

You can view the first few rows of a data frame with the command `head()`:

```
head(iris2)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species petal.area  
## 1           5.1         3.5         1.4        0.2  setosa       0.28
```

```
## 2      4.9      3.0      1.4      0.2  setosa   0.28
## 3      4.7      3.2      1.3      0.2  setosa   0.26
## 4      4.6      3.1      1.5      0.2  setosa   0.30
## 5      5.0      3.6      1.4      0.2  setosa   0.28
## 6      5.4      3.9      1.7      0.4  setosa   0.68
```

Columns can be removed from a data frame by assigning them a value of `NULL`.

```
iris2$petal.area <- NULL
head(iris2)

##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1      5.1      3.5      1.4      0.2  setosa
## 2      4.9      3.0      1.4      0.2  setosa
## 3      4.7      3.2      1.3      0.2  setosa
## 4      4.6      3.1      1.5      0.2  setosa
## 5      5.0      3.6      1.4      0.2  setosa
## 6      5.4      3.9      1.7      0.4  setosa
```

We'll explore more aspects of data frames in later section.

2.6.3 Matrices and arrays

Sometimes you have a structured set of values that does not need to be stored as a data frame, or should not be stored as a data frame for ease of access. These situations call for matrices and arrays.

An array with two dimensions is a matrix. The dimensions of a matrix are its **rows** and **columns**. Matrices look a little like data frames, but internally they are very different. Unlike a data frame and like all arrays, a matrix can contain values of *only one type*. Matrices are necessary for any matrix operations (e.g., Leslie matrices) and for some function inputs. For most routine data analyses, data frames are easier to work with.

Matrices are created with the `matrix()` function. The matrix function must be provided with the values, the dimensions of the matrix, and the method for filling (by rows or by columns). If you don't know the values yet, you can just use `NA` or `0`.

```
my.mat1 <- matrix(1:12, nrow=3, ncol=4)
my.mat1

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

my.mat2 <- matrix(1:12, nrow=3, ncol=4, byrow=TRUE)
my.mat2
```

```

##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    5    6    7    8
## [3,]    9   10   11   12
my.mat3 <- matrix(NA,nrow=3,ncol=4)
my.mat3

##      [,1] [,2] [,3] [,4]
## [1,]    NA    NA    NA    NA
## [2,]    NA    NA    NA    NA
## [3,]    NA    NA    NA    NA

```

Just as with other objects, the values in a matrix can be extracted using bracket notation. Because matrices have two dimensions, you must specify indices for both dimensions. This is done using a comma , to separate the row indices and column indices within the brackets. Row indices are before the comma; column indices are after the comma. Leaving the row index or the column index blank will select all rows or columns, respectively.

```
my.mat1[2,1] # row 2, column 1
```

```

## [1] 2
my.mat1[3,3] # row 3, column 3

```

```
## [1] 9
```

Entire rows or columns of a matrix can be extracted by leaving the other dimension blank. You must still include the comma within the brackets. Notice that a row or a column of a matrix is a vector, not a matrix. If you want to get a row or column as a 1 row or 1 column matrix, you must specifically request this.

```

# first row
my.mat1[1,]
## [1] 1 4 7 10
is.matrix(my.mat1[1,])
## [1] FALSE

# third column
my.mat1[,3]
## [1] 7 8 9

# convert extracted part to matrix
my.mat3 <- matrix(my.mat1[1,], nrow=1)
is.matrix(my.mat3)
## [1] TRUE

```

An array in R is usually only called an array if it has ≥ 3 dimensions. Data stored this way can make programming easier, if there is a clear relationship

between the dimensions. Arrays are not often encountered by the average user. The example below has 3 “layers”, each of which has 4 rows and 2 columns.

```
my.array <- array(1:24, dim=c(4,2,3))
my.array
```

```
## , , 1
##
##      [,1] [,2]
## [1,]    1    5
## [2,]    2    6
## [3,]    3    7
## [4,]    4    8
##
## , , 2
##
##      [,1] [,2]
## [1,]    9   13
## [2,]   10   14
## [3,]   11   15
## [4,]   12   16
##
## , , 3
##
##      [,1] [,2]
## [1,]   17   21
## [2,]   18   22
## [3,]   19   23
## [4,]   20   24
```

One very nifty (and potentially confusing) ability of arrays is the ability to be sliced into differently shaped pieces. Consider the examples below and see if you can work out why the results are what they are.

```
#  
my.array[, , 1]  
  
##      [,1] [,2]
## [1,]    1    5
## [2,]    2    6
## [3,]    3    7
## [4,]    4    8  
  
my.array[, 1, ]  
  
##      [,1] [,2] [,3]
## [1,]    1    9   17
## [2,]    2   10   18
```

```

## [3,]    3   11   19
## [4,]    4   12   20
my.array[1,,]

##      [,1] [,2] [,3]
## [1,]    1    9   17
## [2,]    5   13   21
my.array[1,2,]

## [1] 5 13 21
my.array[,2,1:2]

##      [,1] [,2]
## [1,]    5   13
## [2,]    6   14
## [3,]    7   15
## [4,]    8   16

```

2.6.4 Lists

One of the most important kinds of R object is the **list**. A list can be thought of like a *series of containers or buckets*. Each bucket can contain completely unrelated things, or even other series of buckets. Because of this flexibility, lists are extremely versatile and useful in R programming. Many function outputs are really lists (e.g., the outputs of `lm()` or `t.test()`).

Lists can be created using the function `vector()` and then filled later. The following example creates an empty list with 4 elements (i.e., buckets), then fills each element with something different.

```

my.list <- vector("list",length=4)
my.list[[1]] <- 1:10
my.list[[2]] <- c("a","b","c")
my.list[[3]] <- t.test(iris$Sepal.Length)
my.list[[4]] <- iris
my.list

## [[1]]
## [1] 1 2 3 4 5 6 7 8 9 10
##
## [[2]]
## [1] "a" "b" "c"
##
## [[3]]
##
## One Sample t-test

```

```
##  
## data: iris$Sepal.Length  
## t = 86.425, df = 149, p-value < 2.2e-16  
## alternative hypothesis: true mean is not equal to 0  
## 95 percent confidence interval:  
## 5.709732 5.976934  
## sample estimates:  
## mean of x  
## 5.843333  
##  
##  
## [[4]]  
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species  
## 1       5.1      3.5       1.4      0.2   setosa  
## 2       4.9      3.0       1.4      0.2   setosa  
## 3       4.7      3.2       1.3      0.2   setosa  
## 4       4.6      3.1       1.5      0.2   setosa  
## 5       5.0      3.6       1.4      0.2   setosa  
## 6       5.4      3.9       1.7      0.4   setosa  
## 7       4.6      3.4       1.4      0.3   setosa  
## 8       5.0      3.4       1.5      0.2   setosa  
## 9       4.4      2.9       1.4      0.2   setosa  
## 10      4.9      3.1       1.5      0.1   setosa  
## 11      5.4      3.7       1.5      0.2   setosa  
## 12      4.8      3.4       1.6      0.2   setosa  
## 13      4.8      3.0       1.4      0.1   setosa  
## 14      4.3      3.0       1.1      0.1   setosa  
## 15      5.8      4.0       1.2      0.2   setosa  
## 16      5.7      4.4       1.5      0.4   setosa  
## 17      5.4      3.9       1.3      0.4   setosa  
## 18      5.1      3.5       1.4      0.3   setosa  
## 19      5.7      3.8       1.7      0.3   setosa  
## 20      5.1      3.8       1.5      0.3   setosa  
## 21      5.4      3.4       1.7      0.2   setosa  
## 22      5.1      3.7       1.5      0.4   setosa  
## 23      4.6      3.6       1.0      0.2   setosa  
## 24      5.1      3.3       1.7      0.5   setosa  
## 25      4.8      3.4       1.9      0.2   setosa  
## 26      5.0      3.0       1.6      0.2   setosa  
## 27      5.0      3.4       1.6      0.4   setosa  
## 28      5.2      3.5       1.5      0.2   setosa  
## 29      5.2      3.4       1.4      0.2   setosa  
## 30      4.7      3.2       1.6      0.2   setosa  
## 31      4.8      3.1       1.6      0.2   setosa  
## 32      5.4      3.4       1.5      0.4   setosa  
## 33      5.2      4.1       1.5      0.1   setosa
```

```

## 34      5.5    4.2    1.4    0.2   setosa
## 35      4.9    3.1    1.5    0.2   setosa
## 36      5.0    3.2    1.2    0.2   setosa
## 37      5.5    3.5    1.3    0.2   setosa
## 38      4.9    3.6    1.4    0.1   setosa
## 39      4.4    3.0    1.3    0.2   setosa
## 40      5.1    3.4    1.5    0.2   setosa
## 41      5.0    3.5    1.3    0.3   setosa
## 42      4.5    2.3    1.3    0.3   setosa
## 43      4.4    3.2    1.3    0.2   setosa
## 44      5.0    3.5    1.6    0.6   setosa
## 45      5.1    3.8    1.9    0.4   setosa
## 46      4.8    3.0    1.4    0.3   setosa
## 47      5.1    3.8    1.6    0.2   setosa
## 48      4.6    3.2    1.4    0.2   setosa
## 49      5.3    3.7    1.5    0.2   setosa
## 50      5.0    3.3    1.4    0.2   setosa
## 51      7.0    3.2    4.7    1.4 versicolor
## 52      6.4    3.2    4.5    1.5 versicolor
## 53      6.9    3.1    4.9    1.5 versicolor
## 54      5.5    2.3    4.0    1.3 versicolor
## 55      6.5    2.8    4.6    1.5 versicolor
## 56      5.7    2.8    4.5    1.3 versicolor
## 57      6.3    3.3    4.7    1.6 versicolor
## 58      4.9    2.4    3.3    1.0 versicolor
## 59      6.6    2.9    4.6    1.3 versicolor
## 60      5.2    2.7    3.9    1.4 versicolor
## 61      5.0    2.0    3.5    1.0 versicolor
## 62      5.9    3.0    4.2    1.5 versicolor
## 63      6.0    2.2    4.0    1.0 versicolor
## 64      6.1    2.9    4.7    1.4 versicolor
## 65      5.6    2.9    3.6    1.3 versicolor
## 66      6.7    3.1    4.4    1.4 versicolor
## 67      5.6    3.0    4.5    1.5 versicolor
## 68      5.8    2.7    4.1    1.0 versicolor
## 69      6.2    2.2    4.5    1.5 versicolor
## 70      5.6    2.5    3.9    1.1 versicolor
## 71      5.9    3.2    4.8    1.8 versicolor
## 72      6.1    2.8    4.0    1.3 versicolor
## 73      6.3    2.5    4.9    1.5 versicolor
## 74      6.1    2.8    4.7    1.2 versicolor
## 75      6.4    2.9    4.3    1.3 versicolor
## 76      6.6    3.0    4.4    1.4 versicolor
## 77      6.8    2.8    4.8    1.4 versicolor
## 78      6.7    3.0    5.0    1.7 versicolor
## 79      6.0    2.9    4.5    1.5 versicolor

```

```

## 80      5.7      2.6      3.5      1.0 versicolor
## 81      5.5      2.4      3.8      1.1 versicolor
## 82      5.5      2.4      3.7      1.0 versicolor
## 83      5.8      2.7      3.9      1.2 versicolor
## 84      6.0      2.7      5.1      1.6 versicolor
## 85      5.4      3.0      4.5      1.5 versicolor
## 86      6.0      3.4      4.5      1.6 versicolor
## 87      6.7      3.1      4.7      1.5 versicolor
## 88      6.3      2.3      4.4      1.3 versicolor
## 89      5.6      3.0      4.1      1.3 versicolor
## 90      5.5      2.5      4.0      1.3 versicolor
## 91      5.5      2.6      4.4      1.2 versicolor
## 92      6.1      3.0      4.6      1.4 versicolor
## 93      5.8      2.6      4.0      1.2 versicolor
## 94      5.0      2.3      3.3      1.0 versicolor
## 95      5.6      2.7      4.2      1.3 versicolor
## 96      5.7      3.0      4.2      1.2 versicolor
## 97      5.7      2.9      4.2      1.3 versicolor
## 98      6.2      2.9      4.3      1.3 versicolor
## 99      5.1      2.5      3.0      1.1 versicolor
## 100     5.7      2.8      4.1      1.3 versicolor
## 101     6.3      3.3      6.0      2.5 virginica
## 102     5.8      2.7      5.1      1.9 virginica
## 103     7.1      3.0      5.9      2.1 virginica
## 104     6.3      2.9      5.6      1.8 virginica
## 105     6.5      3.0      5.8      2.2 virginica
## 106     7.6      3.0      6.6      2.1 virginica
## 107     4.9      2.5      4.5      1.7 virginica
## 108     7.3      2.9      6.3      1.8 virginica
## 109     6.7      2.5      5.8      1.8 virginica
## 110     7.2      3.6      6.1      2.5 virginica
## 111     6.5      3.2      5.1      2.0 virginica
## 112     6.4      2.7      5.3      1.9 virginica
## 113     6.8      3.0      5.5      2.1 virginica
## 114     5.7      2.5      5.0      2.0 virginica
## 115     5.8      2.8      5.1      2.4 virginica
## 116     6.4      3.2      5.3      2.3 virginica
## 117     6.5      3.0      5.5      1.8 virginica
## 118     7.7      3.8      6.7      2.2 virginica
## 119     7.7      2.6      6.9      2.3 virginica
## 120     6.0      2.2      5.0      1.5 virginica
## 121     6.9      3.2      5.7      2.3 virginica
## 122     5.6      2.8      4.9      2.0 virginica
## 123     7.7      2.8      6.7      2.0 virginica
## 124     6.3      2.7      4.9      1.8 virginica
## 125     6.7      3.3      5.7      2.1 virginica

```

```

## 126      7.2      3.2      6.0      1.8 virginica
## 127      6.2      2.8      4.8      1.8 virginica
## 128      6.1      3.0      4.9      1.8 virginica
## 129      6.4      2.8      5.6      2.1 virginica
## 130      7.2      3.0      5.8      1.6 virginica
## 131      7.4      2.8      6.1      1.9 virginica
## 132      7.9      3.8      6.4      2.0 virginica
## 133      6.4      2.8      5.6      2.2 virginica
## 134      6.3      2.8      5.1      1.5 virginica
## 135      6.1      2.6      5.6      1.4 virginica
## 136      7.7      3.0      6.1      2.3 virginica
## 137      6.3      3.4      5.6      2.4 virginica
## 138      6.4      3.1      5.5      1.8 virginica
## 139      6.0      3.0      4.8      1.8 virginica
## 140      6.9      3.1      5.4      2.1 virginica
## 141      6.7      3.1      5.6      2.4 virginica
## 142      6.9      3.1      5.1      2.3 virginica
## 143      5.8      2.7      5.1      1.9 virginica
## 144      6.8      3.2      5.9      2.3 virginica
## 145      6.7      3.3      5.7      2.5 virginica
## 146      6.7      3.0      5.2      2.3 virginica
## 147      6.3      2.5      5.0      1.9 virginica
## 148      6.5      3.0      5.2      2.0 virginica
## 149      6.2      3.4      5.4      2.3 virginica
## 150      5.9      3.0      5.1      1.8 virginica

```

Another way to make a list is to make its elements first and combine them into a list.

```

x <- runif(10)
y <- "zebra"
z <- matrix(1:12, nrow=3)
a <- list(x, y, z)
a

## [[1]]
## [1] 0.446969628 0.836004260 0.737595618 0.811055141 0.388108283 0.685169729
## [7] 0.003948339 0.832916080 0.007334147 0.207658973
##
## [[2]]
## [1] "zebra"
##
## [[3]]
## [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12

```

Optionally, you can name each element as you make the list.

```
b <- list(x=x, y=y, z=z)
b

## $x
## [1] 0.446969628 0.836004260 0.737595618 0.811055141 0.388108283 0.685169729
## [7] 0.003948339 0.832916080 0.007334147 0.207658973
##
## $y
## [1] "zebra"
##
## $z
##      [,1] [,2] [,3] [,4]
## [1,]     1     4     7    10
## [2,]     2     5     8    11
## [3,]     3     6     9    12
```

Much like vectors, lists can be extended by using concatenation:

```
a2 <- c(a,b)
a2

## [[1]]
## [1] 0.446969628 0.836004260 0.737595618 0.811055141 0.388108283 0.685169729
## [7] 0.003948339 0.832916080 0.007334147 0.207658973
##
## [[2]]
## [1] "zebra"
##
## [[3]]
##      [,1] [,2] [,3] [,4]
## [1,]     1     4     7    10
## [2,]     2     5     8    11
## [3,]     3     6     9    12
##
## $x
## [1] 0.446969628 0.836004260 0.737595618 0.811055141 0.388108283 0.685169729
## [7] 0.003948339 0.832916080 0.007334147 0.207658973
##
## $y
## [1] "zebra"
##
## $z
##      [,1] [,2] [,3] [,4]
## [1,]     1     4     7    10
## [2,]     2     5     8    11
## [3,]     3     6     9    12
```

Single elements of a list can be accessed with **double brackets** `[][]`. As with other objects, elements can be selected by name or by index. If selecting by name, the `$` symbol can be used instead of double brackets.

```
a[[1]]
## [1] 0.446969628 0.836004260 0.737595618 0.811055141 0.388108283 0.685169729
## [7] 0.003948339 0.832916080 0.007334147 0.207658973
b[[2]]
## [1] "zebra"
b[["z"]]
## [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12
b$bz
## [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12
```

Using **single brackets** to access a list will return a *list* of the requested elements, not the requested elements themselves. This can be a little confusing. Compare the following two commands and notice that the first returns a vector containing the numbers 1 through 10 (the first element of `my.list`), while the second returns a list with 1 element, and that element is a vector with the numbers 1 through 10.

```
my.list[[1]]
## [1] 1 2 3 4 5 6 7 8 9 10
my.list[1]
## [[1]]
## [1] 1 2 3 4 5 6 7 8 9 10
```

The reason for this confusion is that the single brackets notation allows you to extract multiple elements of the list at once to make a new list.

```
a2 <- a[1:2]
a2
```

```
## [[1]]
## [1] 0.446969628 0.836004260 0.737595618 0.811055141 0.388108283 0.685169729
## [7] 0.003948339 0.832916080 0.007334147 0.207658973
##
## [[2]]
## [1] "zebra"
```

2.6.5 S4 objects

We've seen so far that R is object-oriented. What is little appreciated is that R has at least three object-oriented paradigms: **S3**, **S4**, and **R5**. Most objects in R are S3 objects: data frames, matrices, vectors, etc. The R5 objects are more formally called "Reference Classes" and are still being developed. The S4 objects are only occasionally encountered by the average user, although some R developers believe that S4 objects should replace S3 in general use. Their reasons are esoteric and we won't get into them here. In a nutshell, S4 classes are structured more formally than S3 classes and are better for validation and code testing.

One key difference in practice between S4 and S3 objects is that S4 objects contain **slots** or **attributes**, which are accessed using `@` instead of `$`. Within each slot, an S4 object can contain all sorts of data, often in list form.

2.7 R data types

In R, data are stored in objects with a well-defined structure. The structures of these objects can facilitate manipulating and retrieving your data. The structure of an R object is determined by its class. The format of data (i.e., the kinds of values contained in an object) is referred to as the data **type**. Most R users only encounter types when trying to understand error messages. This is because many R functions will return errors if their inputs are not of the correct type.

2.7.1 Character type

Character data are text strings. They are input and printed using quotation marks (""). Note that straight quotes are used. No other kind of quotation mark is valid in R code. If you paste in text from another program that has "curly quotes" (e.g., from Word), you will get an error.

```
a <- c("aardvark", "bunny", "caterpillar")
a

## [1] "aardvark"    "bunny"       "caterpillar"
```

Notice that in the command above the values in `a` are defined in quotation marks. Without quotation marks, R would interpret those words as the names of objects. For example:

```
caterpillar <- "lion"
c("aardvark", "bunny", caterpillar)

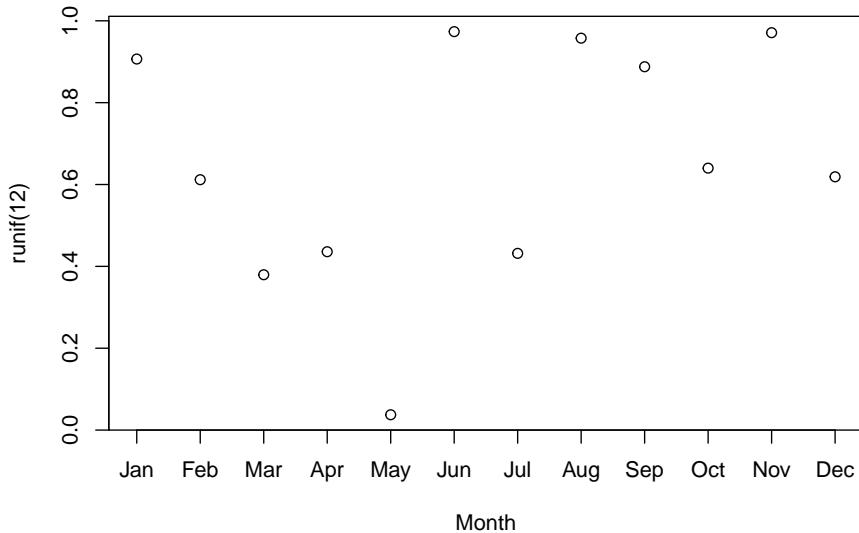
## [1] "aardvark" "bunny"     "lion"
```

There are some special character vectors pre-built into R that can be useful in programming: `letters` returns the lower case letters a-z, while `LETTERS` returns the upper-case letters A-Z.

```
letters
## [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"
## [20] "t" "u" "v" "w" "x" "y" "z"
LETTERS
## [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R" "S"
## [20] "T" "U" "V" "W" "X" "Y" "Z"
letters[11:20]
## [1] "k" "l" "m" "n" "o" "p" "q" "r" "s" "t"
```

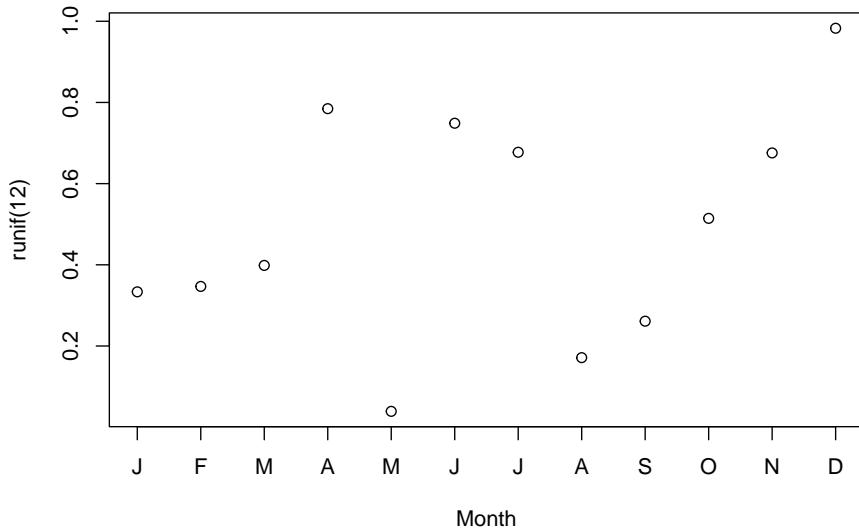
Similarly, `month.name` and `month.abb` contain the names of the 12 months and their 3 letter abbreviations, respectively. The latter is very useful for making figure axes.

```
plot(1:12, runif(12), xaxt="n", xlab= "Month")
axis(side=1, at=1:12, labels=month.abb)
```



As an aside, the function `substr()` extracts parts of character strings (“substring”). One application is getting one letter abbreviations of the months for a figure axis. The example below selects from the month names starting and ending at character 1.

```
use.months <- substr(month.name, 1, 1)
plot(1:12, runif(12), xaxt="n", xlab= "Month")
axis(side=1, at=1:12, labels=use.months)
```



2.7.2 Numeric type

Most data are stored as the **numeric** type. This includes **floating point numbers** (e.g., 3.14159 or 3.14e0). Floating point numbers are referred to as type “double” in R (short for “double-precision floating point”).

```
typeof(2)
```

```
## [1] "double"
typeof(pi)
## [1] "double"
```

2.7.3 Integer type

Integers are numbers with no fractional components. Some numbers are stored as integers, but integers are not always interchangeable with floating point numbers. For example, some probability distribution functions return integer values rather than floating point values (e.g., binomial or Poisson). This can be problematic if very large numbers are expected because the maximum integer value in R is much smaller than the maximum floating-point value. Note that having the type **integer** in R and being an integer from a mathematical standpoint are not synonymous.

```
typeof(7)
## [1] "double"
typeof(as.integer(7))
## [1] "integer"
is.integer(7)
## [1] FALSE
is.numeric(7)
## [1] TRUE
typeof(rbinom(1, 10, 0.7))
## [1] "integer"
7 %% 1
## [1] 0
```

Integers are used as indices for subsetting and accessing other objects. A sequence of integers can be made using the colon operator as follows:

```
1:10
## [1] 1 2 3 4 5 6 7 8 9 10
```

An important consequence of using integers as index values is that object dimensions are limited to the **R integer limit**. This is probably not a major concern for most users. Your machine will likely run out of memory before you could make such an object anyway. You can check the integer limit on your machine with this line:

```
.Machine$integer.max
## [1] 2147483647
```

On my machine, this is about 2.14×10^9 . If you need data objects with dimensions larger than this, you may need to either use a specialized R package or rethink what you are doing.

2.7.4 Logical type

The **logical** data type stores TRUE and FALSE values. Note that TRUE and FALSE are always written in capital letters. Logical values are produced by logical tests, such as the test for equality using ==:

```
1+1 == 2
## [1] TRUE
typeof(1+1 == 2)
## [1] "logical"
typeof(1+1 == 3)
## [1] "logical"
```

Another useful test you can do is to test whether elements of one set are in another:

```
5 %in% 1:10
## [1] TRUE
5 %in% 6:10
## [1] FALSE
1:10 %in% 5:15
## [1] FALSE FALSE FALSE FALSE TRUE TRUE TRUE TRUE TRUE TRUE
```

You can test with inequalities:

```
2 >= 1
## [1] TRUE
3 < 2
## [1] FALSE
```

Sometimes it can be useful to store logical values as 1 or 0. You can do this by multiplying a logical by 1.

```
1*(1+1 == 2)
## [1] 1
1*(1+1 == 3)
## [1] 0
typeof(1+1 == 2)
## [1] "logical"
typeof(1*(1+1 == 2))
## [1] "double"
```

The ! symbol **negates** or **reverses** logical values. It does not negate logical values stored as numbers.

```
2+2 == 4
## [1] TRUE
! 2+2 == 4
## [1] FALSE
1:10 %in% 5:15
## [1] FALSE FALSE FALSE FALSE TRUE TRUE TRUE TRUE TRUE
!1:10 %in% 5:15
## [1] TRUE TRUE TRUE TRUE FALSE FALSE FALSE FALSE FALSE
```

The **any()** and **all()** functions test to see if any values in a set of logical values is true (**any()**) or if all of them are true (**all()**).

```
1:5 < 3
## [1] TRUE TRUE FALSE FALSE FALSE
any(1:5 < 3)
## [1] TRUE
all(1:5 < 3)
## [1] FALSE
```

2.7.5 Special values

Missing data in R are labeled as `NA`, or “Not Available”. This is R-speak for a blank or missing value. Some functions can ignore `NA`, some cannot. Some functions produce `NA` when you supply them with improper inputs. Note that this is a feature, not a bug. R is set up to produce `NA` by default in many cases to help you track down where the missing values are coming from. Function `is.na()` tests to see if a value is missing.

```
aa <- c(1:8,NA)

aa + 2
## [1] 3 4 5 6 7 8 9 10 NA

# returns NA
mean(aa)
## [1] NA

# option to ignore NA values
mean(aa,na.rm=TRUE)
## [1] 4.5

# test for NA
is.na(aa)
## [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE

# which elements are NA?
which(is.na(aa))
## [1] 9

# remove NA elements
aa[!is.na(aa)]
## [1] 1 2 3 4 5 6 7 8

# another way to remove NA elements
aa[which(!is.na(aa))]
## [1] 1 2 3 4 5 6 7 8

# returns NA: Poisson distribution supported only on [0,+Inf]
rpois(10,-2)
## Warning in rpois(10, -2): NAs produced
## [1] NA NA NA NA NA NA NA NA NA NA
```

Positive and negative infinity ($+\infty$ and $-\infty$) are labeled in R as `Inf` and `-Inf`. Dividing a non-zero number by 0 produces `Inf`. Taking the log of 0 produces `-Inf`. Trying to generate a value greater than R’s maximal value can produce

`Inf.`

```
5/0
## [1] Inf
log(0)
## [1] -Inf
10^10^10
## [1] Inf
```

Some values that are undefined are labeled `NaN` (not a number). This means that a value is undefined, not that it is missing (which would be `NA`) or infinite. Dividing 0 by 0, or taking the log of a negative number, produces `NaN`. The function `is.nan()` tests to see if a value is `NaN`.

```
0/0
## [1] NaN
log(-1)
## Warning in log(-1): NaNs produced
## [1] NaN
```

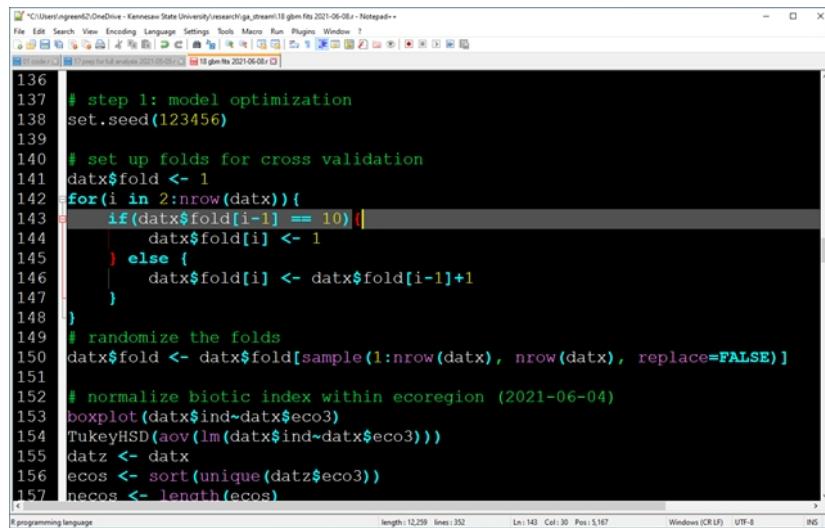
Why are there so many special values? To help track down where erroneous values are coming from. If every undefined value returned the same symbol (or word, etc.), then it would be hard to know why a program was returning undefined values. Instead, the different results help point the way:

- `Inf`: Positive non-zero number divided by 0
- `-Inf`: Negative non-zero number divided by 0 *or* `log(0)`.
- `NaN`: 0/0 or logarithm of a negative number

2.8 Manage R code as scripts (.r files)

Over your career you will probably write a lot of code. You will download and run even more code. This page describes how to manage code that you write (scripts) and code that you get from R repositories (packages).

Often it is easiest to write your R code in separate files, called **scripts**. An R script has the file extension **.r** or **.R**. Scripts can be used as files that store R code (basically plain text files) or as stand-alone R programs. One advantage of keeping your R code in scripts is that you will avoid the formatting issues that arise when writing R code in word processors like Microsoft Word. Another advantage comes from using specialized code editors like Notepad++, RStudio, or others: syntax highlighting. Consider the image below:



```

136
137 # step 1: model optimization
138 set.seed(123456)
139
140 # set up folds for cross validation
141 datx$fold <- 1
142 for(i in 2:nrow(datx)){
143   if(datx$fold[i-1] == 10){
144     datx$fold[i] <- 1
145   } else {
146     datx$fold[i] <- datx$fold[i-1]+1
147   }
148 }
149 # randomize the folds
150 datx$fold <- datx$fold[sample(1:nrow(datx), nrow(datx), replace=FALSE)]
151
152 # normalize biotic index within ecoregion (2021-06-04)
153 boxplot(datx$ind~datx$eco3)
154 TukeyHSD(aov(lm(datx$ind~datx$eco3)))
155 datz <- datx
156 ecos <- sort(unique(datz$eco3))
157 necos <- length(ecos)
158

```

The screenshot shows a Notepad++ window displaying R code. The code is for model optimization and cross-validation. It includes setting a seed, creating a fold column, randomizing the folds, and normalizing biotic indices within ecoregions. The code uses comments, loops, conditionals, and various R functions like `set.seed`, `for`, `if`, `sample`, and `boxplot`. The Notepad++ interface shows syntax highlighting for R code, with different colors for comments, keywords, and variables.

This shows some typical R code in Notepad++ with comments, a `for` loop, some functions being called, datasets being subsetted, etc. Notice how code elements are colored differently, so it's easier to see the different parts. Notice also how matching braces {} are highlighted. This is *extremely* helpful when programming complex operations.

The final advantage to working with R scripts that I'll mention is automation and running from source. R can read and run .r files with the `source()` command. This can be useful when managing a large set of simulations or analyses: code the individual simulations in a stand-alone script, and then repeatedly call that script from a different script or R instance. This can work as a simple (albeit labor-intensive) way to achieve parallelization.

2.9 Manage and use R packages

“Packages” are modules that can be added to R to extend its capabilities. Most packages are written by R users and researchers who need functions that are not provided with the base R installation. These add-ons are one of the greatest strengths of R compared to other software packages. Commercial and proprietary statistics programs (like SAS) are upgraded only by a single group of developers, if at all. In contrast, new functions and capabilities are added to R by the user community every day. Some packages are widely used and cited, others less so. Do your homework before depending on a package. Is the package included or recommended with the base R installation? Is the package associated with any peer-reviewed publications? Does the package author appear to have the credentials that would qualify them to produce a package? Do other packages depend on this package? All of these are good signs that a package is reliable. Most packages are stored on The Comprehensive R Archive Network, or CRAN.

Other repositories exist, such as the Geological Survey R Archive Network, or GRAN. R-Forge, BioConductor, and GitHub also host some R packages. R-Forge and BioConductor can usually be accessed from the R command line just like CRAN. Installing packages from GitHub may require a different procedure. Most R packages are written in the R language itself, so even if you can't find a way to install a package you can usually just copy and run its source code.

2.9.1 Your R library

When you install R packages, the files are downloaded and stored on your machine for R to access when needed. R will probably create a folder in your R home directory for this purpose. The first time you install a package in R you will be asked if R can create a personal library in which to store the package. Package installations are specific to the version of R that you used for the installation, and specific to your operating system login. This means that if someone else logs in to your machine and wants to use a package, they will have to install the package for themselves. This also means that if you update your R installation to a new version, you may need to install packages again because some packages are version-specific.

2.9.2 Installing packages using the R GUI

In the R GUI, go to Packages – Install package(s).... You will be prompted to select a CRAN mirror (i.e., server from which to download). Pick any mirror you like. Then, scroll to and select your package. If the install fails, try a different mirror by going to Packages – Set CRAN mirror..., and then retrying the installation.

2.9.3 Installing packages in RStudio

You can install a package in RStudio by going to Tools–Install Packages... and searching for the name of the package you want.

2.9.4 Installing packages using the R console

In the R console, you can install packages using function `install.packages()` as shown below. You will still be asked to select your CRAN mirror (most packages are hosted on CRAN).

```
install.packages("vegan")
```

2.9.5 Working with packages in R

From R, you can access the packages you have installed by **loading** them. In most situations you must load a package before you use it. The usual way is to use the `library()` function. Once a package is loaded, you can use its functions and datasets. The function `require()` does the same thing as `library()`, but

it is designed for use inside other functions. Most of the time you should use `library()`. The command below will load the package `vegan` so its functions and datasets become available:

```
library(vegan)
```

Notice that the package name does not have to be in quotation marks within the `library()` function.

You can see what packages are currently attached with `search()` or `sessionInfo()`. Packages can be unloaded detaching them from the workspace: `detach(package:x)`, where `x` is the name of the package you want to detach. In the example below, the first command loads the `MASS` package. Then, we can see that is loaded in the workspace with `search()`. The next command unloads `MASS`, then verifies that `MASS` was unloaded from the workspace with another `search()` command.

```
search()
## [1] ".GlobalEnv"      "package:bookdown"  "package:stats"
## [4] "package:graphics" "package:grDevices" "package:utils"
## [7] "package:datasets" "package:methods"   "Autoloads"
## [10] "package:base"

library(MASS)
search()
## [1] ".GlobalEnv"      "package:MASS"      "package:bookdown"
## [4] "package:stats"    "package:graphics" "package:grDevices"
## [7] "package:utils"    "package:datasets" "package:methods"
## [10] "Autoloads"        "package:base"

detach(package:MASS)
search()
## [1] ".GlobalEnv"      "package:bookdown"  "package:stats"
## [4] "package:graphics" "package:grDevices" "package:utils"
## [7] "package:datasets" "package:methods"   "Autoloads"
## [10] "package:base"
```

There is rarely any need to unload a package from the workspace unless there are naming conflicts. The most common situation is when two attached packages have functions with the same name. In this situation, R will use the most recently loaded package's function. This phenomenon is called **masking**. When there is a name conflict, you can use the double colon operator `::` to specify which package's function you want to use.

Using `::` can also access a function without loading the entire package into the workspace. If both packages are already loaded into your workspace, then `::` will simply select which package's function to use. Note that a package must be installed on your machine before you can use its functions.

The example below shows how `::` can be used to access the `rtriangle()` and

`fitdistr()` functions without loading their packages.

```
base:::log(10)
## [1] 2.302585
log(10)
## [1] 2.302585
triangle::rtriangle(10, 1, 10, 5)
## [1] 5.583211 7.399238 3.611716 4.125111 7.219196 6.284387 3.942722 2.244024
## [9] 3.248832 3.791035
MASS::fitdistr(rnorm(100, 50, 10), "normal")
##      mean         sd
## 49.1385032 10.3490806
## ( 1.0349081) ( 0.7317905)

# neither package (triangle or MASS) is attached:
search()
## [1] ".GlobalEnv"          "package:bookdown"   "package:stats"
## [4] "package:graphics"    "package:grDevices" "package:utils"
## [7] "package:datasets"    "package:methods"   "Autoloads"
## [10] "package:base"
sessionInfo()
## R version 4.1.1 (2021-08-10)
## Platform: x86_64-w64-mingw32/x64 (64-bit)
## Running under: Windows 10 x64 (build 19043)
##
## Matrix products: default
##
## locale:
## [1] LC_COLLATE=English_United States.1252
## [2] LC_CTYPE=English_United States.1252
## [3] LC_MONETARY=English_United States.1252
## [4] LC_NUMERIC=C
## [5] LC_TIME=English_United States.1252
##
## attached base packages:
## [1] stats      graphics   grDevices  utils      datasets  methods   base
##
## other attached packages:
## [1] bookdown_0.24
##
## loaded via a namespace (and not attached):
## [1] R2WinBUGS_2.1-21 lattice_0.20-44 digest_0.6.28   MASS_7.3-54
## [5] R2jags_0.7-1    grid_4.1.1     magrittr_2.0.1  evaluate_0.14
## [9] coda_0.19-4     rlang_0.4.12   stringi_1.7.5  rstudioapi_0.13
## [13] rjags_4-12     boot_1.3-28   rmarkdown_2.11 triangle_0.12
## [17] tools_4.1.1    stringr_1.4.0  parallel_4.1.1 abind_1.4-5
```

```
## [21] xfun_0.27          yaml_2.2.1        fastmap_1.1.0      compiler_4.1.1
## [25] htmltools_0.5.2    knitr_1.36
```

2.9.6 Package dependencies

Many R packages are built taking advantage of other R packages. The packages that a package depends on are called its **dependencies** or **depends**. Some packages have no dependencies (other than R itself), while others have many. Something to think about when deciding whether to use a package are the number and nature of its dependencies. If you use a package that depends on many other packages, then you are relying on the authors of those packages to maintain and update their code to work with new versions of R and new versions of packages. And, relying upon the authors of all of those packages' dependencies. And so on. You can see the dependencies of a package by checking out its page on CRAN. For example, the `triangle` package (Carnell 2019) is here. You can see the current version, what packages it depends on (“Depends”), and what packages depend on it (“Reverse depends”).

The point here is that you don't want your code to break because one of the packages you use hasn't been updated. This can get very frustrating if, for example, several packages you use all depend on each other, or even different versions of each other! Most R package authors and/or maintainers are pretty good about keeping their packages up to date, but some are not. If you really need a function from a package that is no longer maintained, or incompatible with your current version of R, you have three options:

1. Download and use the older version of R that is compatible with the package. This is not recommended because you will lose the benefit of recent bug fixes and modifications to R. This option is also not recommended because by rolling back to an older version of R, you will likely need to roll back to older versions of other packages.
2. Find another, current package that does what you need to do. This can be a quite a headache if you have a large codebase or a very specialized function.
3. Access the source code to the package you need, and “borrow” the functions or code you need.

Sometimes, if you only need 1 or 2 functions from a package, or you need to modify a function from a package to suit your needs. In these cases, you can “borrow” directly from the package source code. For most packages, the CRAN page will have a link to an online code repository (usually Github) and/or the **package source** as a “tarball” (“.tar.gz” file). The latter is a compressed version of the package source code.

Because most R packages are written in R itself, you can usually find and adapt the code you need. This is almost always permitted by the license of the package because most packages are released under some version of the GNU Public

License (GPL). If you do borrow code from a package, it is considered good form to attribute the package authors in some way (e.g., “we adapted code from the package “triangle” version 0.12 (Carnell 2019)”).

2.9.7 Citing packages

Just like any software, you need to cite a package if you use it. This is both to give appropriate credit to the package authors, and to allow others to understand what you did. The appropriate citation for a package can be accessed from the console with the `citation()` command.

Citation for R itself:

```
citation()

##
## To cite R in publications use:
##
##   R Core Team (2021). R: A language and environment for statistical
##   computing. R Foundation for Statistical Computing, Vienna, Austria.
##   URL https://www.R-project.org/.
##
## A BibTeX entry for LaTeX users is
##
##   @Manual{,
##     title = {R: A Language and Environment for Statistical Computing},
##     author = {{R Core Team}},
##     organization = {R Foundation for Statistical Computing},
##     address = {Vienna, Austria},
##     year = {2021},
##     url = {https://www.R-project.org/},
##   }
##
## We have invested a lot of time and effort in creating R, please cite it
## when using it for data analysis. See also 'citation("pkgname")' for
## citing R packages.
```

Citation for a package (note the quotation marks):

```
citation("MASS")

##
## To cite the MASS package in publications use:
##
##   Venables, W. N. & Ripley, B. D. (2002) Modern Applied Statistics with
##   S. Fourth Edition. Springer, New York. ISBN 0-387-95457-0
##
## A BibTeX entry for LaTeX users is
```

```

## @Book{,
##   title = {Modern Applied Statistics with S},
##   author = {W. N. Venables and B. D. Ripley},
##   publisher = {Springer},
##   edition = {Fourth},
##   address = {New York},
##   year = {2002},
##   note = {ISBN 0-387-95457-0},
##   url = {https://www.stats.ox.ac.uk/pub/MASS4/},
## }
citation("triangle")

##
## To cite package 'triangle' in publications use:
##
## Rob Carnell (2019). triangle: Provides the Standard Distribution
## Functions for the Triangle Distribution. R package version 0.12.
## https://CRAN.R-project.org/package=triangle
##
## A BibTeX entry for LaTeX users is
##
## @Manual{,
##   title = {triangle: Provides the Standard Distribution Functions for the Triangle
## Distribution},
##   author = {Rob Carnell},
##   year = {2019},
##   note = {R package version 0.12},
##   url = {https://CRAN.R-project.org/package=triangle},
## }

```

2.10 R documentation

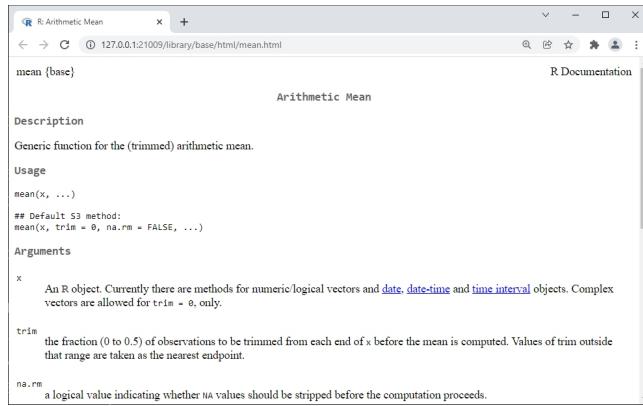
The most important lesson of this R and biological data analysis course is simple: ***Don't panic!*** Lots of people have trouble with R. The learning curve is steep, and the error messages are inscrutable. But, there are a lot of resources available to help you.

2.10.1 Documentation (help) files

Documentation for each function, or help files, come included with the base R installation. You can access them from within R by using the `? command`. `?functionname` will open the documentation file for function `functionname`. `? will search all currently attached packages`. The command below opens the page for the `mean` function. Notice that this file is stored on your local machine, so

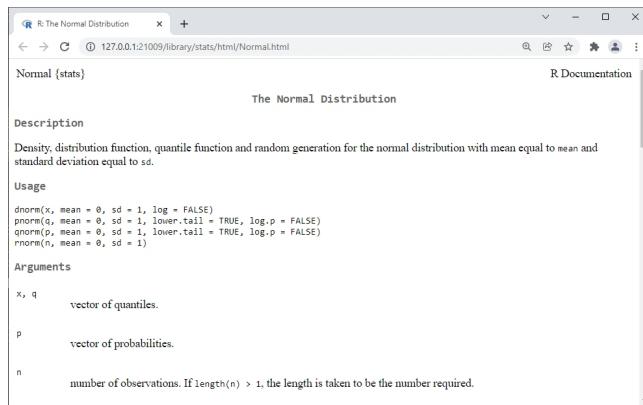
no internet access is required.

?mean



Some functions belong to a family of related functions that share a common documentation page. For example, all of the functions related to the normal distribution share a common page:

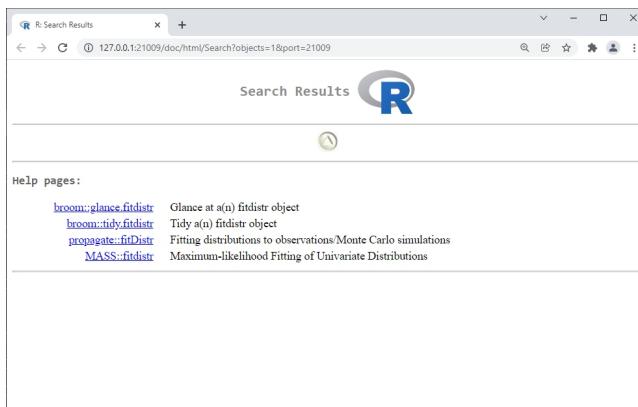
?dnorm
?pnorm
?qnorm
?rnorm



You can also search for help on functions in packages that are not attached. Or, you might want to know about a function but don't know what package it belongs to. Either way, `??functionname` is a more general search option that will search any package in your library. Compare what happens if search for function `fitdistr()` using `?` and `??` (this function is in package MASS, and so not loaded by default). With `??`, R searches all installed packages and returns a browser window with every result that might match your search term.

```
?fitdistr
## No documentation for 'fitdistr' in specified packages and libraries:
## you could try '??fitdistr'

??fitdistr
```



The R documentation pages all tend to follow the same format:

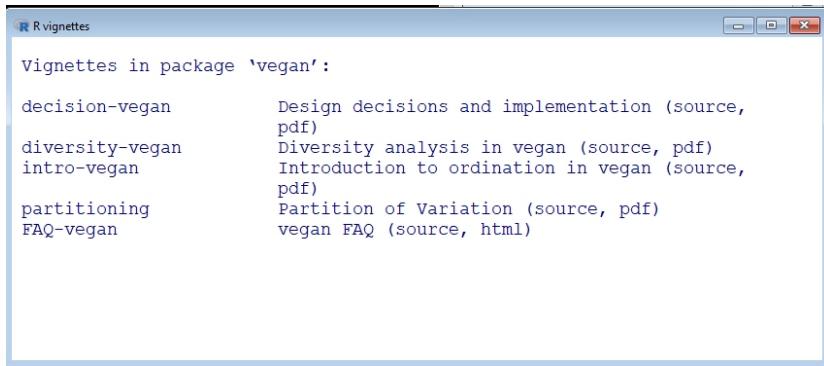
- **Description:** brief description of what the function does.
- **Usage:** presentation of the function with its arguments in their typical format *and expected order*.
- **Arguments:** list of the arguments to the function *in order*, their required type/format, and allowed values. This section is very important. When my code isn't working, the answer to why is usually in the arguments section.
- **Details:** additional information necessary to use the function.
- **Value:** description of the function's outputs.
- **Source:** description of where the original algorithm or method came from.
- **References:** citations for the methods used.
- **See also:** list of similar or related R functions.
- **Examples:** illustration of how to use the function. This section is *very* important. If your code is not working, compare how you are using the function to the examples. Are your inputs in the right format? Did you name your arguments or put unnamed arguments in the wrong order? Careful examination and running of the examples will usually reveal why your code is not working.

R documentation pages tend to be very terse and include very little in the way of background or explanation. They are generally meant to refresh the memory of people already familiar with R and with statistics, not to educate new R users or inexperienced statisticians. If you fall into one of the latter categories, or if you are trying to learn a new function, read on.

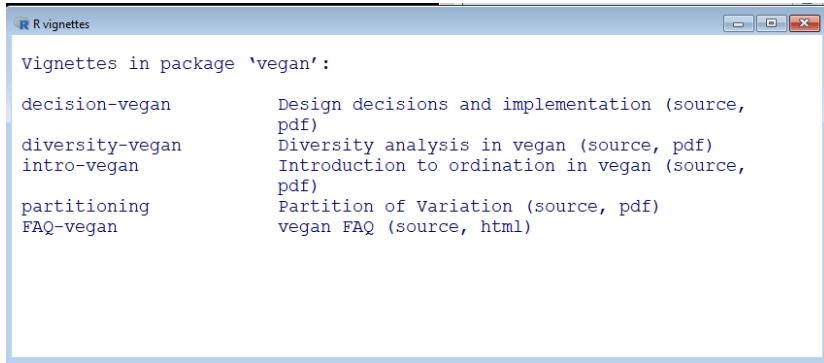
2.10.2 R vignettes

A **vignette** is a document demonstrating the use of an R package or its functions, with more details and explanation than in the help files. Not all packages offer vignettes. You can see what vignettes are available with the command `vignette()`. Specific vignettes are accessed by name and package. The examples below show how to get a list of available vignettes, available vignettes by package, and open specific vignettes.

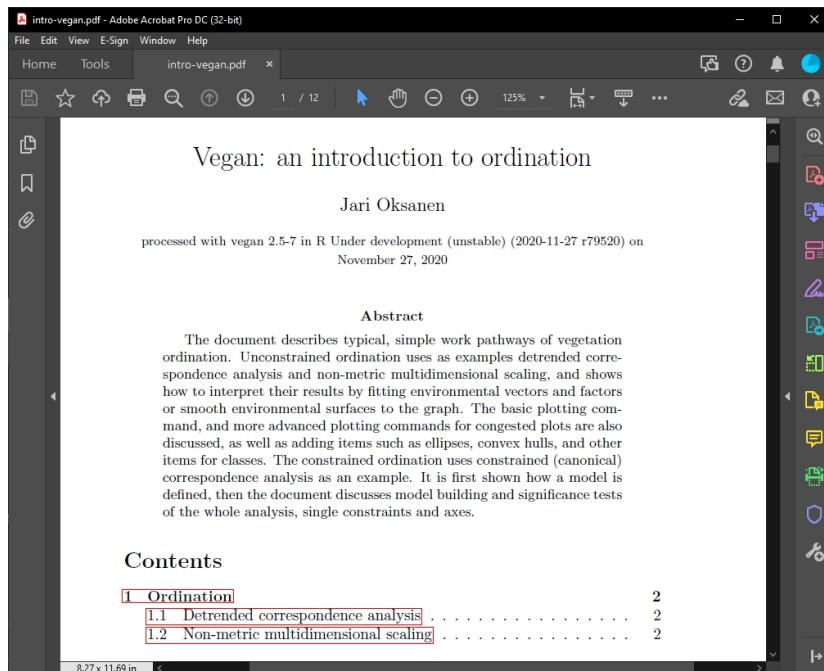
```
# list vignettes in all installed packages
vignette()
```



```
# list vignettes in package vegan
vignette(package="vegan")
```



```
# open vignette "intro-vegan" in local PDF reader
vignette("intro-vegan", package="vegan")
```



2.10.3 Official R Project resources

2.10.3.1 R Project help pages

The R project maintains an official help page with lists of places to get help. They also maintain a list of manuals and documentation. If you want a handy desk reference, you could do worse than the R reference card (accessed 2021-12-21).

The Journal of Statistical Software publishes papers about new statistical algorithms and software packages. Many of its papers are about new R packages and applications. The papers about new R packages and methods often include extensive code examples.

2.10.3.2 R task views

The Task Views on CRAN contain frequently updated lists of packages that are useful for specific domains. For example, ecologists might check the Environmetrics, Hydrology or Spatial task views. Geneticists might be more interested in Genetics or Phylogenetics. Physiologists and biochemists might be interested in Pharmacokinetics, Clinical Trials, or Survival. I've spent a lot of time in the Bayesian, Numerical Mathematics, and Machine Learning task views. There's something for everyone... even a Teaching Statistics task view with resources to help you learn R and stats at the same time.

(All links accessed 2021-12-21).

2.10.3.3 R mailing lists

If your question has been asked before—and this is very likely—there may be an answer waiting for you on the R Help mailing list. Results from the R help mailing list will usually show up in Google results. You can submit a question to the R mailing list, but be very cautious about doing so. You will need to provide a short, clear, reproducible example in your question so that other people can see the issue you are having¹⁰. You also need to be sure that your question hasn't been asked before. Some members of the mailing list will respond with great hostility and condescension if you fail to follow those rules.

There are several special interest groups (SIG), e-mail mailing lists for R users who work in specific fields. For example, R-sig-ecology is an e-mail list for ecologists who use R. Other SIG of interest to biologists include R-sig-genetics for genetics, R-sig-mixed-models for mixed models, R-sig-phylo for phylogenetics and related methods, and R-sig-Geo for GIS and spatial analysis.

2.10.4 Unofficial online resources

Unofficial resources are becoming more and more abundant and useful. One of the most important sources for R help is StackOverflow. StackOverflow is a hugely popular forum site where programmers ask and answer questions, and has an active R section. When (not if) you find a solution on StackOverflow, include a link to the answer in your code as a comment. This can save you some headache down the line when you need to remember how you solved a problem. Other useful resources include RSeek.org, a search engine for R resources; and R-Bloggers, which aggregates many blogs about R and statistics.

Many R users maintain blogs or websites that may or may not be aggregated on R-Bloggers. You can usually find these via a Google search. I have used the pages on Quick-R many times. The UCLA Statistical Consulting Group has an excellent collection of tutorials.

2.10.5 R books

The last 10 years have seen an explosion of books published specifically about the R language and its uses. Some of these books are general introductions to R and statistics; others are very domain specific. Inclusion of a book here does not constitute specific endorsement. I own some of these, but others are included because they are common R-related references that you might find helpful.

- The Use R! series contains many high-quality general and domain-specific books, although some of the older titles are a bit dated. For example:

¹⁰This means a *very* minimal code example that people can use to get the same problem that you are having. Don't post your entire dataset or include many lines of pre-processing steps or other code that led up to your problem. Use base R functions or built-in datasets to create a small dataset or set of values that will immediately provoke the same error code. Often the process of doing this will help you solve your own problem!

- A Beginner’s Guide to R by Alain Zuur et al.
- Data Manipulation with R by Phil Spector
- Introductory time series with R by P.S.P. Cowperthwaite and A.V. Metcalfe.
- Bayesian computation with R by Jim Albert
- Introductory Statistics with R by Peter Dalgaard
- Morphometrics with R by Julien Claude
- The R Book by Michael Crawley
- Advanced R by Hadley Wickham
- R for Dummies by Andrie DeVries

When I was first starting out with R I found several books invaluable. Having code examples printed where I could highlight, annotate, etc., was really helpful. Over time I’ve found myself relying less on textbooks and more on online sources (especially StackOverflow). Still, there are a few books that I use frequently, for their statistical exposition more than their code examples. At least one of Bolker (2008), Kéry (2010), Legendre and Legendre (2012), Zuur et al. (2007), and Zuur et al. (2009a) is usually on my desk at any given time.

2.10.6 Two reminders

I said it at the beginning of this section, and it bears repeating: ***Don’t panic!*** R has a steep learning curve, but the pay-off is worth it. There are plenty of resources and people out there who can help you.

Finally, be sure to ***read the manual*** when you are stuck. The R help pages and code examples usually have a solution. Make sure your inputs are in the right format. Check for syntax and spelling error—including capitalization! As a funny meme on I saw on Reddit said, *6 hours of debugging can save you 5 minutes of reading documentation*¹¹.

¹¹I think this was originally a tweet that went viral, but the joke spread through the internet and now you can buy it on a t-shirt. Regardless, this phenomenon is much older than Twitter.

Chapter 3

Data manipulation with R

3.1 Data import and export

Before you can use R to analyze your data, you need to get your data into R. Most of us store our data in an Excel spreadsheet or some kind of database. These are fine, but if you plan to have R in your workflow I strongly recommend you get in the habit of storing your data in plain text files. Plain text files store data separated (or “delimited”) by a particular character such as a comma or a tab. Unsurprisingly, these are called **comma-delimited (.csv)** or **tab-delimited (.txt)** text files. The reason for storing and importing your data this way is simple: plain text files are a stable, almost-universally supported file format that can be read and opened by many programs on any computer. Proprietary formats such as Excel spreadsheets (.xlsx or .xls) or the Mac “Numbers” program (.numbers) are not as widely supported and change more frequently than plain text formats. The fact that plain text formats are typically smaller than equivalent Excel files is a nice bonus.

Reading your data in from text files has another benefit: repeatability. Chances are you will need to revisit or repeat your data analysis several times. If you program the importation of data in a few lines of R code, you can just run the code. If, however, you use one of the “point-and-click” methods available in RStudio or base R you will have to repeat those steps manually. This can get very aggravating and time consuming if you have multiple datasets to import. This page is designed to demystify the process of getting data into R and to demonstrate some good practices for data storage.

Note that many of the code blocks below have `# not run:` in them, and no output. These indicate that either single commands or the entire block was not actually evaluated when rendering the R Markdown. This was either because the code was designed to return an error (which would prevent R Markdown rendering), produce large outputs, or would clutter up the folder where I store

the files for this website. These commands will work on your machine, and will demonstrate important R concepts, so try them out!

3.1.1 Importing data: preliminaries

Before you try to import your data into R, take a few minutes to set yourself up for success.

3.1.1.1 Naming data files

Good file names are essential for a working biologist. Chances are you will work on many projects over the course of your career, each with its own collection of data files. Even if you store each project's data in a separate folder, it is a very good idea to give data files within those folders names that are descriptive. I have worked with many people who insist on using file names like “data.xlsx” and relying on their own memory or paper notes to tell them which file is which. Consider this list of particularly egregious file names:

- data.txt
- Data.txt
- data.set.csv
- analysis data.xlsx
- thesis data.xlsx
- DATA FINAL - copy.txt
- data current.revised.2019.txt.xlsx
- final thesis data revised revised final revised analysis.csv

We are all guilty of this kind of bad naming from time to time. What do all of the above have in common? They are vague. The names contain no meaningful information about what the files actually contain. Some of them are ambiguous: different operating systems or programs might read “data.txt” and “Data.txt” as the same file, or as different files!

Data file names should be **short**, **simple**, and **descriptive**. There are no hard and fast rules for naming data files so I'm not going to prescribe any. Instead, I'll share some examples and explain why these names are helpful. As you go along and get better at working with data you will probably develop your own habits, and that's ok. Just try to follow the guiding principles of short, simple, and descriptive.

Here are some example file names with explanations of their desirable properties:

sturgeon_growth_data_2017-02-21.txt

This name declares what the file contains (growth data for sturgeon), and the date it was created. Notice that the date is given as “YYYY-MM-DD”. This format is useful because it automatically sorts chronologically. This file name separates its words with underscores rather than spaces or periods. This can

prevent some occasional issues with programs misreading file names (white space and periods can have specific meanings in some contexts).

dat_combine_2020-07-13.csv

This name signifies that the file contains a “combined” dataset, and includes the date of its creation. This would be kept in a folder with the files that were combined to make it. Within a project I usually use prefixes on file names to help identify them: “dat” for datasets with response and explanatory variables; “met” for “metadata”; “geo” or “loc” for geospatial datasets and GIS layers; and “key” for lookup tables that relate dat and met files. You might develop your own conventions for types of data that occur in your work.

dat_vegcover_nrda_releaseversion_2021-05-03.csv

This name is similar to the last name, but includes some additional signifiers for the project funder (NRDA) and the fact that this is the final released version.

3.1.1.2 Formatting data files

Recently Broman and Woo (2018) reviewed common good practices for storing data in spreadsheets. Their article is short and easy to read, and gave a few recommendations for data formatting:

- Be consistent
- Choose good names for things
- Write dates as YYYY-MM-DD
- No empty cells
- Put just one thing in a cell
- Make it a rectangle
- Create a data dictionary
- No calculations in the raw data files
- Do not use font color or highlighting as data
- Make backups
- Use data validation to avoid errors
- Save the data in plain text files

Most people who work with data, myself included, probably agree with their recommendations. There is room for individual preferences in the exact implementation of each principle, but the general ideas are solid. You will save yourself a lot of headache if you follow their advice. I strongly suggest you read their paper.

3.1.1.2.1 Variable names Variable names, like data file names, should be brief but informative. There are few restrictions on what variable names can be in R. For simplicity, I recommend you use only **lowercase letters**, **numbers**, **underscores** (_), and **periods** (.) in variable names. Other symbols are either inconvenient to type, or can cause issues being parsed by R. Some common pitfalls include:

Leading numerals will cause an error if not enclosed in quotation marks:

```
x <- iris[1:5,]
names(x)[1] <- "3a"
x$3a
x$"3a"
x[, "3a"]
```

Numerical-only names will cause an error. They can also be mis-interpreted as column numbers. Don't give your variables numeral-only names.

```
x <- iris[1:5,]
names(x)[1] <- "3"
x$3      # error
x$"3"    # works
x[,3]    # column number 3
x[, "3"] # column named "3"
```

Spaces in variable names will cause errors because R interprets a space as whitespace, not part of a name. If you insist on using spaces, enclose the variable name in quotation marks every time you use it. This is one of the few places where R cares about whitespace.

```
x <- iris[1:5,]
names(x)[1] <- "a b"
# not run (error):
# x$a b
x$"a b'

## [1] 5.1 4.9 4.7 4.6 5.0
```

Special characters in variable names can cause issues with parsing. Many symbols that you might think to use, such as hyphens (or dashes, -) or slashes / are also R operators. Again, try to avoid these issues by using letters, numbers, underscores, and periods.

```
x <- iris[1:5,]
names(x)[1] <- "a/b"
b <- 10
# not run (error):
# x$a/b
# works:
x$"a/b'

## [1] 5.1 4.9 4.7 4.6 5.0
names(x)[1] <- "surv%"
x

## surv% Sepal.Width Petal.Length Petal.Width Species
```

```

## 1 5.1 3.5 1.4 0.2 setosa
## 2 4.9 3.0 1.4 0.2 setosa
## 3 4.7 3.2 1.3 0.2 setosa
## 4 4.6 3.1 1.5 0.2 setosa
## 5 5.0 3.6 1.4 0.2 setosa

# not run (error):
#x$surv%
x$'surv%'

## [1] 5.1 4.9 4.7 4.6 5.0

```

Uppercase letters in variable names work just fine, but they can be inconvenient. Because R is case sensitive, you need to type the variable exactly the same way each time. This can be tedious if you have several variables with similar names.

```

x <- iris[1:5,]
names(x)[1] <- "sep.len" # convenient
names(x)[2] <- "SEP.LEN" # slightly inconvenient
names(x)[3] <- "Sep.Len" # more inconvenient
names(x)[4] <- "SeP.1En" # just don't
x

##   sep.len SEP.LEN Sep.Len SeP.1En Species
## 1 5.1 3.5 1.4 0.2 setosa
## 2 4.9 3.0 1.4 0.2 setosa
## 3 4.7 3.2 1.3 0.2 setosa
## 4 4.6 3.1 1.5 0.2 setosa
## 5 5.0 3.6 1.4 0.2 setosa

```

3.1.1.2.2 Data values All data in R have a type: numeric, logical, character, etc. Within a vector, matrix, or array, all values must have the same type or they will be converted to character strings. Each variable in a data frame (the most common R data structure) is essentially a vector. This means that in your input file you should try to have one and only one type of data in each column. Otherwise, R will coerce the values into character strings, which may or may not be a problem. Putting only one type of value into each column will also help get your data into “wide” or “tidy” format (Wickham (2014)).

Entering data should also include some measure of **quality assurance and quality control (QA/QC)**. This basically means making sure that your data are entered correctly, and are formatted correctly. Correct entry can be accomplished by direct comparison to original datasheets or lab notebooks. Cross-checking is a valuable tool here: have two people enter the data and compare their files.

Data formatting requires thinking about what kinds of data are being entered. For numbers, this means entering the correct values and an appropriate number of significant digits. For text values, this may mean spell-checking or settling on

a standard set of possible values. The figure below shows some problems that a QA/QC step might fix:

	A	B	C
1	species	tsn	common
2	Peromyscus leucopus	180278	white-footed mouse
3	Peromyscus leucopus	180278	White foot mouse
4	Peromyscus leucodus	180278	White_footed_mouse
5	Microtus ochrogaster	180312	Prairie_vole
6	Microtus ochrogaster	180312	Prairie vole
7

3.1.2 Importing data from text files with `read.csv()` and `read.table()`

Plain text files are imported, or read, using functions that are named for that purpose. The function `read.table()` works on many types of text files, including tab-delimited and comma-delimited. The function `read.csv()` works on comma-delimited files only. The syntax for both is very similar.

Put the files `elk_data_2021-01-21.txt` and `elk_data_2021-01-21.csv` in your R home directory. The R home directory depends on your operating system:

- *Windows users*: the home directory is your **Documents** folder. The address of this folder is `C:/Users/username/Documents`, where `username` is your Windows logon name. You can get to your Documents folder by opening Windows Explorer (Win-E) and double-clicking “Documents”.
- *Mac users*: the home directory is your **home** folder. The address of this folder is `/Users/username` (where `username` is your username) or `~`. You can open your home folder by pressing CMD-SHIFT-H, or by using the Go pulldown menu from the menu bar.

Import a tab-delimited file from your home directory:

```
dat <- read.table("elk_data_2021-01-21.txt", header=TRUE)
```

Import a comma-delimited version from your home directory. Note how similar the syntax is.

```
dat <- read.csv("elk_data_2021-01-21.csv", header=TRUE)

# alternative, using read.table():
dat <- read.table("elk_data_2021-01-21.csv",
                  header=TRUE, sep=",")
```

The examples above required the name of the data file—notice the file extensions!—and an argument `header` which specifies that the first row of the data file contains column names. Try one of those commands without that argument and

see what happens.

```
dat <- read.table("elk_data_2021-01-21.txt", header=TRUE)
head(dat)
```

```
##   height antler species
## 1   69.3    24.8 unknown
## 2   69.3    34.5 unknown
## 3   87.7    53.5 unknown
## 4   99.8    58.3 unknown
## 5   69.3    61.2 unknown
## 6   84.1    67.3 unknown

dat <- read.table("elk_data_2021-01-21.txt")
head(dat)

##      V1      V2      V3
## 1 height antler species
## 2   69.3    24.8 unknown
## 3   69.3    34.5 unknown
## 4   87.7    53.5 unknown
## 5   99.8    58.3 unknown
## 6   69.3    61.2 unknown
```

Without the `header` argument, the column names were read as the first row of the dataset and the columns of the data frame were given generic names `V1`, `V2`, and so on. As a side effect, the values in each of the columns are all of character type instead of numbers. This is because both `read.table()` and `read.csv()` import all values as character strings by default, and convert to other types only if possible. In this case, conversion is not possible because the strings `height`, `antler`, and `species` cannot be encoded as numbers, but the numbers form the original file can be encoded as character strings. Further, the values within a single column of a data frame are a vector, and all elements of a vector must have the same type. Thus, importing without the `header` argument will likely result in a data frame full of text representations of your data.

Both `read.table()` and `read.csv()` will bring in data stored as a table, and produce a type of R object called a **data frame**. This data frame must be saved to an object to be used by R. If not saved, the data frame will be printed to the console. Compare the results of the two lines below on your machine.

```
read.table("elk data 2021-01-21.txt", header=TRUE)
dat <- read.table("elk data 2021-01-21.txt", header=TRUE)
```

The first command read the file, and printed it to the console. The second command read the file and stored it in an object named `dat`. Nothing is returned to the console when this happens.

3.1.2.1 Important arguments to `read.table()` and `read.csv()`

3.1.2.1.1 `file` The first argument to `read.table()` and `read.csv()` is `file`, which is the filename, including directory path, of the file to be imported. By default, R will look in your home directory for the file. If the file is stored in the home directory, then you can just supply the filename as a character string. Don't forget the file extension! Most of the examples on this site will, for convenience, use data files stored in your home directory. In your own work you may want to use other folders.

If you want to import a file from another folder, you need to supply the folder name as part of the argument `file`. My preferred way to do this is by specifying the folder and file name separately, and combining them with the `paste()` command. The function `paste()` concatenates, or sticks together, character strings. Separating the folder from the file name also makes your code more portable. For example, if you needed to run your code on another machine you would only need to update the directory. An example of how this might work is shown below:

```
# not run:

# directory that contains all data and folders
top.dir <- "C:/_data"

# project folder
proj.dir <- paste(top.dir, "ne_stream/dat", sep="/")

# folder that holds dataset
data.dir <- proj.dir

# name of dataset
data.name <- "dat_combine_2020-07-13.csv"

# import dataset
in.name <- paste(data.dir, data.name, sep="/")
dat <- read.csv(in.name, header=TRUE)
```

In the example above, moving to a new machine would only necessitate changing one line, the definition of `data.dir` (or `top.dir`, if you keep the same folder structure on multiple machines).

3.1.2.1.2 `header` We've already seen the importance of `header`. You'll use `header=TRUE` in almost every import command you ever run. Occasionally you'll run into datasets that do not come with a header row. Of course, you won't store your data that way because it complicates data management.

3.1.2.1.3 `sep` As seen in one of the examples above, the `sep` argument specifies what character separates values on each line of the data file. This argument has some useful default values. If you are using `read.table()` with a tab-delimited file, or `read.csv()` with a comma-delimited file, you don't need to set `sep`.

The default in `read.table()` reads any white space as a delimiter. This is fine if you have a tab-delimited file, but if the file contains a mix of spaces and tabs you might have some problems. For example, if entries are separated by tabs, but some values contain spaces. In those cases, specify the appropriate delimiter:

```
# not run:
in.name <- "elk_data_2021-01-21.txt"

# tab-delimited (sep="\t"):
dat <- read.table(in.name, header=TRUE, sep="\t")

# space-delimited (sep=" "):
#dat <- read.table(in.name, header=TRUE, sep=" ")
```

The `sep` argument is usually necessary when using the “clipboard trick” (see below).

3.1.2.1.4 `row.names` When R imports a table of data, it will assign names to the rows. The default names are just the row numbers. Some data files will have their own row names. If you want to not import these names, you can set `row.names` argument to FALSE to get the default numbers. You can also specify names as a character vector.

```
# not run:
use.rownames <- paste("row", 1:20, sep="_")
dat <- read.table(in.name, header=TRUE, row.names=use.rownames)
dat
```

3.1.2.1.5 `col.names` The argument `col.names` works in a similar manner to `row.names`, but for column names. If column names are not provided with argument `header`, R will assign the columns V followed by a column number (V1, V2, etc.). If you want to provide your own names, use `col.names`. Personally, I've never used this argument and instead prefer to either include column names in my data files or just assign names after importing.

```
# not run:
use.colnames <- c("ht", "an", "sp")
dat <- read.table(in.name, header=TRUE, col.names=use.colnames)
dat
```

3.1.2.1.6 `skip` Some data files will contain rows of text or other information before the tabular data starts. The argument `skip` allows you to skip these lines.

Without `skip`, R will try to read those rows of text as if they were data and fail to import. Use the file `elk_data_2021-07-14.txt`, which has some extra rows of text that would otherwise cause `read.table()` to fail.

```
# not run:
# alternate file with some header rows
in.name <- "elk_data_2021-07-14.txt"
# fails:
dat <- read.table(in.name, header=TRUE)
# works:
dat <- read.table(in.name, header=TRUE, skip=2)
dat
```

The error message when you tried to import the file without `skip=2` comes up a lot when trying to import data into R. What is happening is that R uses the number of elements in the first row to calculate the dimensions of the data frame that it makes upon import. In this case, it interpreted the 7 words in the first row of text, `Here is some text about the dataset`, as 7 column headings `Here, is, some, text, about, the, and dataset`. Then, when next line `that came from Gould (1974)`. only had 5 elements (`that, came, from, Gould, and (1974)`), this caused an error.

Why does this issue vex so many new R users (and experienced R users)? Often we will have incomplete records in a data set, where a row is missing values in one or more columns. Sometimes blank values are not read correctly. Consider the file `elk_data_blanks_2021-07-14.txt`, which has a blank in line 13.

When we try to import that file, we get a similar error:

```
# not run:
in.name <- "elk_data_blanks_2021-07-14.txt"
dat <- read.table(in.name, header=TRUE)
```

There are a few ways to deal with this error. The first is to go back to the original datafile and put R's blank value, `NA`, in the blanks. You could also put in a value that is not likely to occur, such as `-9999` in variables that must be positive. This would make your file more readable, but without locking you in to using R. Either solution will ensure that missing values are not interpreted as extra white space. This method can be very time consuming for very large datasets, even in Excel (or especially in Excel). The best time to fix this issue is at data entry. Recall that having no empty cells is one of the recommendations of Broman and Woo (2018).

Another way is to save the file as a comma-delimited file, because commas are less ambiguous than white space for defining missing values. This method usually works, but might still create problems if there are multiple blank values per record or blanks at the ends of rows. The third option is to use the `fill` argument in the R import functions (see below).

3.1.2.1.7 fill As described above, missing values can cause problems when importing data into R. One way to deal with missing values that occur at the end of a line in the data file is to use the `fill` argument. Setting `fill=TRUE` will automatically pad the data frame with blanks so that every row has the same number of values.

```
in.name <- "elk_data_blanks_2021-07-14.txt"
dat <- read.table(in.name, header=TRUE, fill=TRUE)
```

As convenient as `fill=TRUE` can be, you need to be careful when using it. R may add the wrong number of values, or values of the wrong type. In the example above, the value of species in row 13 is a blank character string (""), not a missing value `NA`. This is an important difference because "" and `NA` are treated very differently by many R functions. The commands below show that R interprets the padded entry "" as a character string of length 0, not a missing value.

```
table(dat$species)

##
##          alces megaloceros     unknown
##      1           2           1           16
aggregate(dat$antler, by=list(dat$species), mean)

##
##      Group.1      x
## 1       107.8000
## 2       alces 164.4500
## 3    megaloceros 239.1000
## 4     unknown  85.1375
is.na(dat$species)

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

3.1.2.1.8 stringsAsFactors A **factor** is a variable in a dataset that defines group membership: control vs. treated, red vs. blue, gene 1 vs. gene 2 vs. gene 3, and so on. The different groups to which an observation can belong are called the **levels** of the factor. This is very useful for statistical modeling, but can be a pain in the neck when trying to work with data in R. Functions `read.table()` and `read.csv()` have an argument to automatically convert character strings to factors for convenience: `stringsAsFactors`. This is a logical switch that will interpret character strings as levels of a factor (`stringsAsFactors=TRUE`, the default) or as character strings (`stringsAsFactors=FALSE`).

The reason that factors can cause headaches in R is because although they look like character strings, they are really numeric codes and the text you see is just a label. If you try to use a factor as if it was a character string, you can get strange

errors. My advice is to always import data with `stringsAsFactors=FALSE` and only convert to factor later if needed. Most statistical functions will convert to factor for you automatically anyway. The only time when you may want to store data as a factor is when you have a very large dataset with many levels of a factor; in those situations, factors will use memory more efficiently.

If you get tired of typing `stringsAsFactors=TRUE` in your import commands, you can set an option at the beginning of your script that will change the default behavior of `read.table()` and `read.csv()`:

```
options(stringsAsFactors=FALSE)
```

I put this command at the start of every new R script as a matter of habit. Your mileage may vary.

3.1.3 Importing data from saved workspaces

3.1.3.1 Importing R objects with `readRDS()`

Some R objects are not easily represented by tables. For example, the outputs of statistical methods are usually **lists**. Such objects can be saved using function `saveRDS()` and read into R using `readRDS()`. The example below will save an R data object to file `rds_test.rds` in your R working directory.

```
mod1 <- lm(Petal.Width~Sepal.Width, data=iris)
mod1

##
## Call:
## lm(formula = Petal.Width ~ Sepal.Width, data = iris)
##
## Coefficients:
## (Intercept) Sepal.Width
##           3.1569      -0.6403
saveRDS(mod1, "rds_test.rds")
b <- readRDS("rds_test.rds")
b

##
## Call:
## lm(formula = Petal.Width ~ Sepal.Width, data = iris)
##
## Coefficients:
## (Intercept) Sepal.Width
##           3.1569      -0.6403
```

There is a lazier way to accomplish this task using function `dput()`. This function saves a plain text representation of an object that can later be used to recreate the object (i.e., R code that can reproduce the object). The help file for

`dput()` cautions that this is not a good method for transferring objects between R sessions and recommends using `saveRDS()` instead. That being said, I use this method all the time for simple objects and have never had a problem. The procedure is to copy the output of `dput()` into your R script, and later assign that text to another object. Use `dput()` at your own risk.

```
a <- iris[1:6,]
dput(a)

## structure(list(Sepal.Length = c(5.1, 4.9, 4.7, 4.6, 5, 5.4),
## Sepal.Width = c(3.5, 3, 3.2, 3.1, 3.6, 3.9), Petal.Length = c(1.4,
## 1.4, 1.3, 1.5, 1.4, 1.7), Petal.Width = c(0.2, 0.2, 0.2,
## 0.2, 0.2, 0.4), Species = structure(c(1L, 1L, 1L, 1L, 1L,
## 1L), .Label = c("setosa", "versicolor", "virginica"), class = "factor")), row.names = c(NA
## 6L), class = "data.frame")

b <- dput(a)

## structure(list(Sepal.Length = c(5.1, 4.9, 4.7, 4.6, 5, 5.4),
## Sepal.Width = c(3.5, 3, 3.2, 3.1, 3.6, 3.9), Petal.Length = c(1.4,
## 1.4, 1.3, 1.5, 1.4, 1.7), Petal.Width = c(0.2, 0.2, 0.2,
## 0.2, 0.2, 0.4), Species = structure(c(1L, 1L, 1L, 1L, 1L,
## 1L), .Label = c("setosa", "versicolor", "virginica"), class = "factor")), row.names = c(NA
## 6L), class = "data.frame")

b

##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1         5.1       3.5          1.4        0.2  setosa
## 2         4.9       3.0          1.4        0.2  setosa
## 3         4.7       3.2          1.3        0.2  setosa
## 4         4.6       3.1          1.5        0.2  setosa
## 5         5.0       3.6          1.4        0.2  setosa
## 6         5.4       3.9          1.7        0.4  setosa
```

3.1.3.2 Importing R workspaces with `load()`

Everything you do in R takes place within the **workspace**. This can be thought of as a sandbox where all of your data and functions reside. If you want to save this entire environment and continue working later you can do this by saving the workspace (see below). Workspaces can be loaded using function `load()`. The example below loads workspace `example.RData` from your home directory. Notice that this command will load every object from the saved workspace into your current workspace. This might cause problems if the saved workspace and current workspace have objects with the same names.

```
# not run:
load("example.RData")
```

When loading a workspace, it is sometimes more useful to import it into a new environment. This will let you import objects from that workspace without putting its objects into your current workspace. I use this method a lot for working with model outputs. The typical workflow is:

1. Run an analysis or set of simulations in multiple R instances and save the results of each in its own R workspace.
2. After simulations are complete, import the results of each simulation from its individual workspace into a new environment.
3. Copy the objects from the environment containing the loaded workspace to the main workspace.
4. Delete the environment containing the loaded workspace.
5. Repeat until all results are imported.

Here is an example of importing individual objects from a workspace into a new environment, then copying that object into the current workspace:

```
# not run:
rm(a) # will return error if a does not already exist
temp <- new.env()
load("example.RData", env=temp)
a <- temp$a
rm(temp)
a
```

3.1.4 Importing data: special cases:

3.1.4.1 Getting data from the clipboard

Sometimes it can be convenient to copy and paste data directly from a spreadsheet into R. This is fine for quick or one-off data imports, but not recommended for routine use because it is tedious to reproduce and error-prone. The procedure is as follows:

1. In the R console, type this command but do NOT run it: `a <- read.table("clipboard", header=TRUE, sep="\t")`
2. In Excel, highlight your data and copy it to the clipboard using Ctrl-C.
3. Click in the R console to move your focus there. In R, hit the ENTER key.

The command you typed in step 1 will read the table from the clipboard. Note that if you try to copy/paste the `read.table("clipboard", ...)` command into the R console, you will then have that command in your clipboard and not your data.

3.1.4.2 Entering data using `scan()`

The `scan()` function lets you enter values one-by-one, separated by pressing ENTER. You can use this just like typing values into Excel. Or, you can copy and paste a single column of values from Excel. The latter case works because

R interprets the values within a column as being separated by a line return (i.e., pressing ENTER).

Type the name of an object to which data will be saved, the assignment operator `<-`, `scan()`, then ENTER, then values (each followed by ENTER), then hit ENTER twice after the last value. This is obviously inefficient for large datasets, but it's a nice trick to be aware of.

```
a <- scan()
1
2
3
4
5

a
```

To enter values from a column in Excel, highlight and copy the column. Then go to R, enter the command below, and press ENTER. Then paste your values and press ENTER again.

```
# not run:
a <- scan()
# paste in values
# press ENTER again
a
```

3.1.4.3 Entering data directly using `c()`

Function `c()` combines its arguments as a vector. As with all functions, arguments must be separated by commas and can be on different lines.

```
c(1:3, 9:5)

## [1] 1 2 3 9 8 7 6 5

a <- c(1,
      3,
      5
      )
```

Like `scan()`, data can be entered directly into the console using `c()`. Also like `scan()`, this is horribly inefficient but can be useful for small numbers of values.

3.1.5 Export data from R

3.1.5.1 Writing out tables (best method)

Everything you do in R takes place within the “workspace”. You can save entire workspaces, but in most situations it is more useful to write out data or results

as delimited text files. “Delimited” means that individual data within the file are separated, or delimited, by a particular symbol. The most common formats are comma-delimited files (.csv) or tab-delimited files (.txt). Which format to use is largely a matter of personal preference. I’ve found that comma-delimited files are slightly less likely to have issues when opened in Excel.

Tab-delimited files are written out using `write.table()`. In the example below, the first argument `dat` is the object from the R workspace that you want to write out. The argument `sep` works the same way as in `read.table()`. The example below will save a file called `mydata.txt` in your home directory (R’s working directory). Option `sep="\t"` can be provided to ensure that the file is written as a tab-delimited file and not a space-delimited file.

```
# not run:
dat <- iris[1:6,]
write.table(dat, "mydata.txt", sep="\t")
```

Comma-delimited files are written out by `write.csv()`. The syntax is similar to `write.table()`. Note that the file extension must be “.csv” instead of “.txt”. If you save a file with the wrong extension, then it may be unusable.

```
# not run:
write.csv(dat, "mydata.csv")
```

Most of the time you will not be saving files to your home directory, but to another folder. The folder and the file name must both be specified to save somewhere else. It is usually a good idea to define the file name and file destination separately. This way if you re-use your code on another machine, or a different folder, you only need to update the folder address once.

```
# not run:

# do this:
out.dir <- "C:/_data"
out.name <- "mydata.csv"
write.csv(dat, paste(out.dir, out.name, sep="/"))

# or this:
out.dir <- "C:/_data"
out.name <- "mydata.csv"
out.file <- paste(out.dir, out.name, sep="/")
write.csv(dat, out.file)

# not this:
write.csv(dat, "C:/_data/mydata.csv")
```

The latter example is problematic because the output file name contains both the destination folder and the filename itself. If you try to run your code on

another machine, it will take longer to update the code with a new folder address. This method is also problematic because exporting multiple files will require typing the folder address multiple times. Consider this example:

```
# not run:
```

```
dat1 <- iris[1:6,]
dat2 <- iris[7:12,]

write.csv(dat1, "C:/_data/mydata1.csv")
write.csv(dat2, "C:/_data/mydata2.csv")
```

Running that code on a new machine, or modifying it for a new project in a different folder, requires changing the folder address twice. The better way is to define the folder name only once:

```
# not run:
```

```
out.dir <- "C:/_data"
write.csv(dat1, paste(out.dir, "mydata1.csv", sep="/"))
write.csv(dat2, paste(out.dir, "mydata2.csv", sep="/"))
```

See the difference? The revised code is much more portable and reusable. This illustrates a good practice for programming in general (not just R) called “Single Point of Definition”. Defining variables or objects once and only once makes your code more robust to changes and easier to maintain or reuse. The example below shows a more compact way to export the files `mydata1.csv` and `mydata2.csv`, with even fewer points of definition for file names:

```
# not run:
```

```
out.dir <- "C:/_data"
out.names <- paste0("mydata", 1:2, ".csv")
out.files <- paste(out.dir, out.names, sep="/")
write.csv(dat1, out.files[1])
write.csv(dat2, out.files[2])
```

One handy trick is to auto-generate an output filename that includes the current date. This can really help keep multiple versions of analyses or model runs organized. The `paste0()` command pastes its arguments together as they are, with no separator between them (equivalent to `paste(..., sep="")`). Function `Sys.Date()` prints the current date (according to your machine) in YYYY-MM-DD format.

```
# not run:
```

```
out.dir <- "C:/_data"
out.name <- paste0("mydata", "_", Sys.Date(), ".csv")
write.csv(dat, paste(out.dir, out.name, sep="/"))
```

3.1.5.2 Writing out individual R objects

Some data do not fit easily into tables. Individual R objects can be saved to an “.rds” file using the function `saveRDS()`. This function saves a single object that can later be loaded into another R session with function `readRDS()`.

```
# not run:

x <- t.test(rnorm(100))

# save to working directory:
saveRDS(x, "zebra.rds")

# load the file:
z <- readRDS("zebra.rds")
```

3.1.5.3 Saving entire R workspaces

Entire R workspaces or parts of workspaces can be saved using the function `save.image()`. The command by itself will save the entire current workspace with a generic name to the working directory. It is more useful to save the workspace under a particular name and/or only save specific objects.

```
# not run:

# save entire workspace with generic name (avoid):
save.image()

# save entire workspace with specific name (better):
save.image(file="test1.RData")

# save specific objects (best):
x <- rnorm(100)
z <- t.test(x)
save(list=c("x", "z"), file="testworkspace.RData")
```

3.1.5.4 Writing text to a file (special cases)

Sometimes you need to write out data that do not fit easily into tables. An example might be the results of a single *t*-test or ANOVA. This is also sometimes necessary when working with programs outside of R. For example, the programs OpenBUGS and JAGS require a text file that defines the model to be fit. This text file can be generated by R, which makes your workflow simpler because you don’t need to use a separate program to store the outputs.

The function to write such files is `sink()`. Functions `print()` and `cat()` are also useful in conjunction with `sink()`.

- `sink()` works in pairs. The first command defines a file to put text into. The second command finishes the file and writes it.
- `print()` prints its arguments as-is. It should produce the same text as running its argument in the R console.
- `cat()` stands for “concatenate and print”. Use this to combine expressions and then print them.

```
# not run:

my.result <- t.test(rnorm(50))
sink("C:/_data/myresult.txt")
cat("t-test results:", "\n") #\n = new line
print(my.result)
sink()
```

3.2 Making values in R

The typical analytical workflow in R involves importing data from text files or other sources. Sometimes, however, it is necessary to make values within R. This can be for a variety of reasons:

1. Producing test data sets with desired distributions
2. Making sets of indices or look-up values to use in other functions
3. Modifying existing datasets
4. Running simulations
5. And many more...

This section demonstrates some methods for producing values—i.e., data—within R.

3.2.1 Producing arbitrary values with `c()`

In a previous section we saw how function `c()` combines its arguments as a **vector**. As with all functions, arguments must be separated by commas and can be on different lines. This method can be used to make vectors with any set of values.

```
c(1:3, 9:5)

## [1] 1 2 3 9 8 7 6 5

a <- c(1,
      3,
      5
     )
```

Like `scan()`, this is an inefficient way to make vectors but can be useful for small numbers of values. Or, for sets of values that can't be defined algorithmically.

3.2.2 Generating regular values

3.2.2.1 Consecutive values with :

Consecutive values can be generated using the colon operator `:`. A command like `X:Y` will generate consecutive values from `X` to `Y` in increments of 1 or -1. If `X < Y`, the increment is 1; if `X > Y`, the increment is -1. The usual use case is to make sets of consecutive integers.

```
# Simple examples:
1:10

## [1] 1 2 3 4 5 6 7 8 9 10

5:2

## [1] 5 4 3 2
-1:10

## [1] -1 0 1 2 3 4 5 6 7 8 9 10
10:-1

## [1] 10 9 8 7 6 5 4 3 2 1 0 -1
```

Sometimes you may need to use parentheses to get the intended values because of the order in which R runs operators (aka: “operator precedence”).

```
# effect of parentheses (operator precedence):
a <- 2:6
length(a)

## [1] 5

1:2*length(a)

## [1] 5 10
1:(2*length(a))

## [1] 1 2 3 4 5 6 7 8 9 10
```

`X` and `Y` don't have to literally be numbers; they can be variables or R expressions that evaluate to a number. Below are some examples:

```
a <- 2:6
# length(a) evaluates to a number (5)
length(a):10

## [1] 5 6 7 8 9 10
```

Most people use `:` to produce sets of integers, but it can also make non-integers. The values will be separated by 1 or -1. The first value will be the value before the `:`. The last value might not be the value after the `:`. It will be the value up to or before the second number along the sequence from the first number. If the value before the `:` is an integer, then the sequence will consist of integers.

2.3:5

```
## [1] 2.3 3.3 4.3
5:2.3
## [1] 5 4 3
2.7:6.1
## [1] 2.7 3.7 4.7 5.7
```

3.2.2.2 Regular sequences with `seq()`

Function `seq()` generates regular sequences of a given length (`length=`), or by an interval (`by=`). The command must be supplied with a length or an interval, but not both. If you specify a length, R will calculate the intervals for you. If you specify an interval, R will calculate the length for you. If you specify a length and an interval, R will return an error.

```
seq(0,20,by=0.5)
## [1]  0.0  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
## [16]  7.5  8.0  8.5  9.0  9.5 10.0 10.5 11.0 11.5 12.0 12.5 13.0 13.5 14.0 14.5
## [31] 15.0 15.5 16.0 16.5 17.0 17.5 18.0 18.5 19.0 19.5 20.0
seq(-pi,pi,by=0.1)
## [1] -3.14159265 -3.04159265 -2.94159265 -2.84159265 -2.74159265 -2.64159265
## [7] -2.54159265 -2.44159265 -2.34159265 -2.24159265 -2.14159265 -2.04159265
## [13] -1.94159265 -1.84159265 -1.74159265 -1.64159265 -1.54159265 -1.44159265
## [19] -1.34159265 -1.24159265 -1.14159265 -1.04159265 -0.94159265 -0.84159265
## [25] -0.74159265 -0.64159265 -0.54159265 -0.44159265 -0.34159265 -0.24159265
## [31] -0.14159265 -0.04159265  0.05840735  0.15840735  0.25840735  0.35840735
## [37]  0.45840735  0.55840735  0.65840735  0.75840735  0.85840735  0.95840735
## [43]  1.05840735  1.15840735  1.25840735  1.35840735  1.45840735  1.55840735
## [49]  1.65840735  1.75840735  1.85840735  1.95840735  2.05840735  2.15840735
## [55]  2.25840735  2.35840735  2.45840735  2.55840735  2.65840735  2.75840735
## [61]  2.85840735  2.95840735  3.05840735
seq(-1,1,length=20)
## [1] -1.00000000 -0.89473684 -0.78947368 -0.68421053 -0.57894737 -0.47368421
## [7] -0.36842105 -0.26315789 -0.15789474 -0.05263158  0.05263158  0.15789474
## [13]  0.26315789  0.36842105  0.47368421  0.57894737  0.68421053  0.78947368
```

```
## [19] 0.89473684 1.00000000
# returns an error:
## not run:
seq(1, 20, by=3, length=3)
```

If you want the values to go from greater to smaller, the by argument must be negative.

```
# returns an error:
## not run:
seq(-1, -10, by=1)

# works:
seq(-1, -10, by=-1)

## [1] -1 -2 -3 -4 -5 -6 -7 -8 -9 -10
```

3.2.2.3 Repeated values with `rep()`

The function `rep()` creates a sequence of repeated elements. It can be used in several ways.

One way is to specify the number of times its first argument gets repeated. Notice that in the second command `rep(1:3, 5)`, the entire first argument 1 2 3 is repeated 5 times.

```
rep(2, times=5)

## [1] 2 2 2 2 2
rep(1:3, times=5)

## [1] 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3
```

Another way is to repeat each element of its input a certain number of times each:

```
rep(1:3, each=5)

## [1] 1 1 1 1 1 2 2 2 2 2 3 3 3 3 3 3
```

The arguments `times` and `each` can be combined. Notice that `each` takes precedence over `times`.

```
rep(1:3, each=4, times=2)

## [1] 1 1 1 1 2 2 2 2 3 3 3 3 1 1 1 1 2 2 2 2 3 3 3 3
rep(1:3, each=2, times=4)

## [1] 1 1 2 2 3 3 1 1 2 2 3 3 1 1 2 2 3 3 1 1 2 2 3 3
```

3.2.2.4 The “Recycling Rule”

One quirk of R sequences and vectors is the **recycling rule**. This describes a phenomenon in which values in one vector can be “recycled” to match up with values in another vector. This rule is *only* invoked if the length of the longer vector is a multiple of the length of the shorter vector. Otherwise you will get an error. Consider the following example:

```
my.df <- data.frame(x=1:12)

# add a value with 3 unique values
my.df$x3 <- 1:3

# add a value with 4 unique values
my.df$x4

## NULL

# add a value with 2 random values:
my.df$x2 <- runif(2)
```

Trying to add a variable with a number of values that is not a factor of 12–5, 7, 8, 9, 10, or 11—will usually return an error.

```
# not run:
my.df$x7 <- 1:7
```

3.2.3 Generating random values

R can generate many types of **random values**. This makes sense because R was designed for statistical analysis, and randomness is a key feature of statistics. When pulling random values in R (or any other program, for that matter) it is advisable to set the **random number seed** so that your results are reproducible. The random number seed is needed because “random” numbers from a computer are not actually random. They are **pseudo-random**, which means that while the values appear random, and can pass many statistical tests for randomness, they are calculated deterministically from the state of your computer. If you are running simulations, or an analysis that involves random sampling, you need to set the random number seed. Otherwise, you might get different results every time you run your code. This can make reproducing an analysis or set of simulations all but impossible.

The random number seed can be set with function `set.seed()`. Setting the seed defines the initial state of R’s pseudo-random number generator.

```
# different results each time:
runif(3)
```

```
## [1] 0.3367135 0.3192741 0.4037828
```

```
runif(3)

## [1] 0.4790773 0.3679018 0.4656906

runif(3)

## [1] 0.04989215 0.18735671 0.98265941
# same results each time:
set.seed(42)
runif(3)

## [1] 0.9148060 0.9370754 0.2861395

set.seed(42)
runif(3)

## [1] 0.9148060 0.9370754 0.2861395
set.seed(42)
runif(3)

## [1] 0.9148060 0.9370754 0.2861395
```

3.2.3.1 Random values from a set with `sample()`

The function `sample()` pulls random values from a set of values. This can be done with or without replacement. If you try to pull too many values from a set without replacement R will return an error. The default argument for `replace=` is `FALSE`.

```
sample(1:10, 5, replace=FALSE)

## [1] 9 4 2 8 1

sample(1:10, 5, replace=TRUE)

## [1] 8 7 4 9 5
# returns an error
## (not run):
sample(1:10, 20)

# works:
sample(1:10, 20, replace=TRUE)

## [1] 4 10 2 3 9 9 4 5 5 4 2 8 3 10 1 10 8 6 10 8
```

You can use `sample()` with values other than numbers. The function will pull values from whatever its first argument is.

```
x1 <- c("heads", "tails")
sample(x1, 10, replace=TRUE)
```

```

## [1] "tails" "tails" "tails" "tails" "heads" "tails" "heads" "tails" "tails"
## [10] "tails"

x2 <- c(2.3, pi, -3)
sample(x2, 10, replace=TRUE)

## [1] 3.141593 -3.000000 -3.000000 2.300000 2.300000 3.141593 3.141593
## [8] 3.141593 3.141593 3.141593

```

By default, every element in the set has an equal probability of being chosen. You can alter this with the argument `prob`. This argument must have a probability for each element of the set from which value are chosen, and those probabilities must add up to 1.

```

x <- c("heads", "tails")
probs <- c(0.75, 0.25)
a <- sample(x, 100, replace=TRUE)
b <- sample(x, 100, replace=TRUE, prob=probs)

# compare:
table(a)

## a
## heads tails
##    50    50
table(b)

## b
## heads tails
##    79    21

```

3.2.3.2 Random values from probability distributions

One of the strengths of R is its ease of working with probability distributions. We will explore probability distributions more in another section, but for now just understand that a probability distribution is a function that describes how likely different values of a random variable are. In R, random values from a probability distribution can be drawn using a function with a name like `r_()`, where `_` is the (abbreviated) name of the distribution.

The syntax of the random distribution functions is very consistent across distributions. The first argument is always the sample size (`n`), and the next few arguments are the distributional parameters of the distribution. These arguments can be provided with or without their names. The default order of the arguments is shown on the help page for each function (e.g., `?rnorm`). Below are some examples of random draws from different distributions, displayed as histograms.

```

# sample size
N <- 1000

```

```
# draw random distributions
## normal with mean 0 and SD = 1
a1 <- rnorm(N)

## normal with mean 25 and SD = 100
a2 <- rnorm(N, 25, 100)

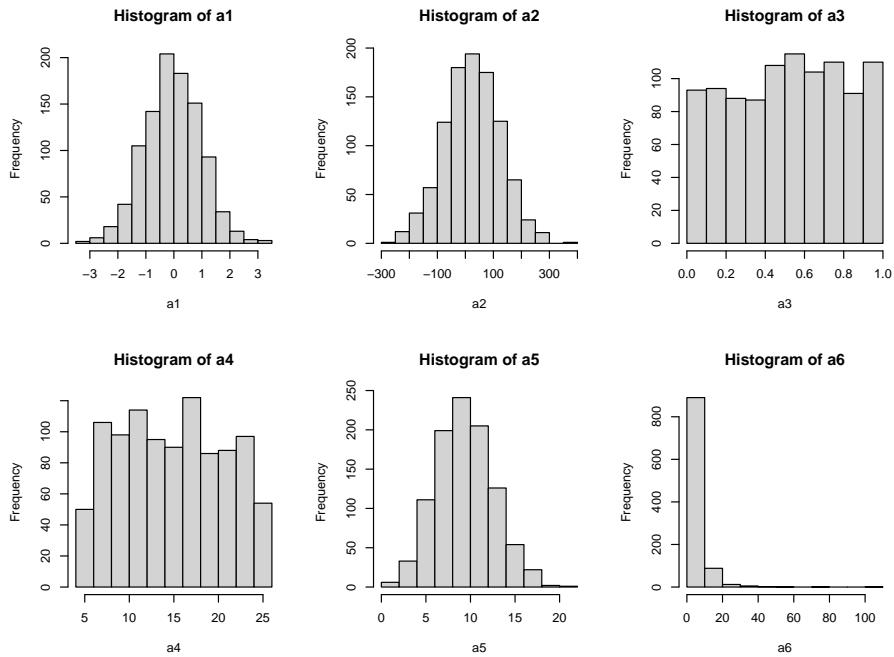
## uniform in [0, 1]
a3 <- runif(N)

## uniform in [5, 25]
a4 <- runif(N, 5, 25)

## Poisson with lambda = 10
a5 <- rpois(N, 10)

## lognormal with logmu = 1.2 and logsd = 2
a6 <- rlnorm(N, 1.2, 0.9)

# produce histogram of each distribution above
par(mfrow=c(2,3))
hist(a1)
hist(a2)
hist(a3)
hist(a4)
hist(a5)
hist(a6)
```



```
# reset graphical parameters
par(mfrow=c(1,1))
```

3.3 Selecting data with []

One of the most common data manipulation tasks is selecting, or **extracting**, from your data. The most common way of selecting subsets of an object (aka: extracting) is the R **bracket notation**. This refers to the symbols used, `[]`. These are also sometimes called **square brackets**, to distinguish them from **curly brackets**, `{}`. The latter are more properly called **braces** and are used for an entirely different purpose in R.

Brackets tell R that you want a part of an object. Which part is determined either by a name, or an **index** inside the brackets. Indices should usually be supplied as numbers, but other types can be used. For objects with more than 1 dimension, you need to supply as many indices, or vectors of indices, as the object has dimensions. Within brackets, vectors of indices are separated by commas.

Brackets can also be used to exclude certain elements by using the negative sign, but this can be tricky. We'll explore how negative indices work below.

3.3.1 Basics of brackets

3.3.1.1 Selecting from a vector

Vectors have 1 dimension, so you must specify a single vector of indices. Notice that the brackets can be empty, in which case all elements can be returned. If the vector supplied is of length 0, then no elements will be returned. When a vector of indices with > 0 elements is specified, R will return the elements of the vector in the order requested. For example, `a[c(1,3,5)]` and `a[c(3,5,1)]` will return the same values but in different orders.

```
a <- 11:20
# all elements returned
a[]

## [1] 11 12 13 14 15 16 17 18 19 20
# second element
a[2]

## [1] 12
# elements 3, 4, 5, 6, and 7
a[3:7]

## [1] 13 14 15 16 17
# error (not run)
# a[3:7,2]

# correct version of previous command
a[c(3:7,2)]

## [1] 13 14 15 16 17 12
# notice different ordering from previous command
a[c(2:7)]

## [1] 12 13 14 15 16 17
```

3.3.1.2 Selecting from a matrix

Matrices have 2 dimensions, so you must specify two dimension's worth of indices. If you want all of one dimension (e.g., rows 1 to 3 and all columns), you can leave the spot for the index blank. Separate the dimensions using a comma. Positive indices select elements; negative elements remove elements.

I'll reiterate this because it is a common mistake: you must specify all dimensions using a comma. Failing to do so will result in an error (bad) or unexpected results (worse).

Remember: in R, indices start at 1. In some languages such as Python indices start at 0.

```
a <- matrix(1:12, nrow=3, byrow=TRUE)

# value in row 1, column 2
a[1,2]

## [1] 2

# values in rows 1-3 and columns 1-2
a[1:3, 1:2]

##      [,1] [,2]
## [1,]    1    2
## [2,]    5    6
## [3,]    9   10

# rows 1 and 2, all columns
a[1:2,]

##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    5    6    7    8

# columns 1 and 2, all rows
a[,1:2]

##      [,1] [,2]
## [1,]    1    2
## [2,]    5    6
## [3,]    9   10
```

Negative indices remove elements, or return all positive indices.

```
# rows except for row 1
a[-1,]

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

Removing multiple elements with - can be a bit tricky. All elements of the vector of things to remove must be negative. In the first example below, R returns an error because you request elements -1, 0, 1, and 2. Element -1 does not exist, and you don't want elements 1 and 2!

```
# rows except for row 1 and 2
## returns error (not run)
a[-1:2,]
```

There are several correct ways to get rows other than 1 and 2. Notice that all of

them result in indices with a 1-, a -2, and no positive values. The first option is probably the best for most situations.

```
a[-c(1,2),]

## [1] 9 10 11 12
a[-1*1:2,]

## [1] 9 10 11 12
a[-2:0,]

## [1] 9 10 11 12
```

Notice that all three of these lines return a vector, not a matrix. This is because a single row or column of a matrix simplifies to a vector. If you want to keep your result in matrix format, you can specify this using the argument `drop=FALSE` after the last dimension and a comma:

```
a[-c(1,2),,drop=FALSE]

##      [,1] [,2] [,3] [,4]
## [1,]    9    10    11    12
```

3.3.1.3 Selecting from a data frame

Selecting from a data frame using brackets is largely the same as selecting from a matrix. Just keep in mind that the result might come back as a vector (if you select rows in a single column) or a data frame (any other situation).

```
a <- iris[1:5,]

# returns a vector
a[,1]

## [1] 5.1 4.9 4.7 4.6 5.0
# returns a data frame
a[1,]

##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5         1.4         0.2  setosa
# returns a data frame
a[1:3,]

##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5         1.4         0.2  setosa
## 2          4.9         3.0         1.4         0.2  setosa
## 3          4.7         3.2         1.3         0.2  setosa
```

```
# returns a data frame
a[1:3, 1:3]

##   Sepal.Length Sepal.Width Petal.Length
## 1          5.1         3.5          1.4
## 2          4.9         3.0          1.4
## 3          4.7         3.2          1.3
```

3.3.1.4 Selecting from arrays

Coming soon.

3.3.1.5 Selecting from lists

One of the most important kinds of R object is the **list**. A list can be thought of like a series of containers or buckets. Each bucket can contain completely unrelated things, or even other series of buckets. Each bucket is called an **element** of the list. Because of this flexibility, lists are extremely versatile and useful in R programming. Many function outputs are really lists (e.g., the outputs of `lm()` or `t.test()`).

Single elements of a list can be accessed with **double brackets** `[][]`. Elements can be selected by name or by index. Elements that have a name can be selected by name using the `$` symbol instead of by double brackets.

```
# make a list "a"
x <- runif(10)
y <- "zebra"
z <- matrix(1:12, nrow=3)
a <- list(e1=x, e2=y, e3=z)

a[[1]]
## [1] 0.48455613 0.46315243 0.42924154 0.26881092 0.13951478 0.03108833
## [7] 0.37119592 0.28906588 0.14279648 0.64557472

a[[2]]
## [1] "zebra"
a[["e1"]]

## [1] 0.48455613 0.46315243 0.42924154 0.26881092 0.13951478 0.03108833
## [7] 0.37119592 0.28906588 0.14279648 0.64557472

a$e1
## [1] 0.48455613 0.46315243 0.42924154 0.26881092 0.13951478 0.03108833
## [7] 0.37119592 0.28906588 0.14279648 0.64557472
```

Multiple elements of a list can be selected using **single brackets**. Doing so will return a new list containing the requested elements, **not** the requested elements themselves. This can be a little confusing. Compare the following two commands and notice that the first returns the 10 random numbers in the first element of **a**, while the second returns a list with 1 element, and that element is the numbers 1 through 10.

```
a[[1]]  
  
## [1] 0.48455613 0.46315243 0.42924154 0.26881092 0.13951478 0.03108833  
## [7] 0.37119592 0.28906588 0.14279648 0.64557472  
  
a[1]  
  
## $e1  
## [1] 0.48455613 0.46315243 0.42924154 0.26881092 0.13951478 0.03108833  
## [7] 0.37119592 0.28906588 0.14279648 0.64557472
```

Next, observe what happens when we select more than one element of the list **a**. The output is a list.

```
a[1:2]  
  
## $e1  
## [1] 0.48455613 0.46315243 0.42924154 0.26881092 0.13951478 0.03108833  
## [7] 0.37119592 0.28906588 0.14279648 0.64557472  
##  
## $e2  
## [1] "zebra"  
class(a[1:2])  
  
## [1] "list"
```

The reason for this syntax is that the single brackets notation allows you to extract multiple elements of the list at once to quickly make a new list.

```
a2 <- a[1:2]  
a2  
  
## $e1  
## [1] 0.48455613 0.46315243 0.42924154 0.26881092 0.13951478 0.03108833  
## [7] 0.37119592 0.28906588 0.14279648 0.64557472  
##  
## $e2  
## [1] "zebra"
```

3.3.2 Extracting and selecting data with logical tests

Selecting data from a data frame is one of the most common data manipulations. R offers several ways to accomplish this. One of the most fundamental is the

function `which()`, returns the indices in a vector where a logical condition is TRUE. This means that you can select any piece of a dataset that can be identified logically. Consider the example below, which extracts the rows of the `iris` dataset where the variable `Species` matches the character string "`setosa`".

```
iris[which(iris$Species == "setosa"),]  
  
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species  
## 1       5.1      3.5       1.4      0.2    setosa  
## 2       4.9      3.0       1.4      0.2    setosa  
## 3       4.7      3.2       1.3      0.2    setosa  
## 4       4.6      3.1       1.5      0.2    setosa  
## 5       5.0      3.6       1.4      0.2    setosa  
## 6       5.4      3.9       1.7      0.4    setosa  
## 7       4.6      3.4       1.4      0.3    setosa  
## 8       5.0      3.4       1.5      0.2    setosa  
## 9       4.4      2.9       1.4      0.2    setosa  
## 10      4.9      3.1       1.5      0.1    setosa  
## 11      5.4      3.7       1.5      0.2    setosa  
## 12      4.8      3.4       1.6      0.2    setosa  
## 13      4.8      3.0       1.4      0.1    setosa  
## 14      4.3      3.0       1.1      0.1    setosa  
## 15      5.8      4.0       1.2      0.2    setosa  
## 16      5.7      4.4       1.5      0.4    setosa  
## 17      5.4      3.9       1.3      0.4    setosa  
## 18      5.1      3.5       1.4      0.3    setosa  
## 19      5.7      3.8       1.7      0.3    setosa  
## 20      5.1      3.8       1.5      0.3    setosa  
## 21      5.4      3.4       1.7      0.2    setosa  
## 22      5.1      3.7       1.5      0.4    setosa  
## 23      4.6      3.6       1.0      0.2    setosa  
## 24      5.1      3.3       1.7      0.5    setosa  
## 25      4.8      3.4       1.9      0.2    setosa  
## 26      5.0      3.0       1.6      0.2    setosa  
## 27      5.0      3.4       1.6      0.4    setosa  
## 28      5.2      3.5       1.5      0.2    setosa  
## 29      5.2      3.4       1.4      0.2    setosa  
## 30      4.7      3.2       1.6      0.2    setosa  
## 31      4.8      3.1       1.6      0.2    setosa  
## 32      5.4      3.4       1.5      0.4    setosa  
## 33      5.2      4.1       1.5      0.1    setosa  
## 34      5.5      4.2       1.4      0.2    setosa  
## 35      4.9      3.1       1.5      0.2    setosa  
## 36      5.0      3.2       1.2      0.2    setosa  
## 37      5.5      3.5       1.3      0.2    setosa  
## 38      4.9      3.6       1.4      0.1    setosa
```

```

## 39      4.4      3.0      1.3      0.2  setosa
## 40      5.1      3.4      1.5      0.2  setosa
## 41      5.0      3.5      1.3      0.3  setosa
## 42      4.5      2.3      1.3      0.3  setosa
## 43      4.4      3.2      1.3      0.2  setosa
## 44      5.0      3.5      1.6      0.6  setosa
## 45      5.1      3.8      1.9      0.4  setosa
## 46      4.8      3.0      1.4      0.3  setosa
## 47      5.1      3.8      1.6      0.2  setosa
## 48      4.6      3.2      1.4      0.2  setosa
## 49      5.3      3.7      1.5      0.2  setosa
## 50      5.0      3.3      1.4      0.2  setosa

```

In most situations, `which()` can be omitted because it is implicit in the command. However, not using it can sometimes produce strange results. Because of this, I always use `which()` in my code. Including `which()` can also make your code a little clearer to someone who is not familiar with R. For example, the two commands below are exactly equivalent:

```

# not run:
iris[which(iris$Species == "setosa"),]
iris[iris$Species == "setosa",]

```

What `which()` is really doing in the commands above is returning a vector of numbers. That vector is then used by the containing brackets `[]` to determine which pieces of the data frame to pull. The numbers returned by `which()` are the indices at which a logical vector is TRUE. Compare the results of the two commands below, which were used in the commands above to select from `iris`:

```

# logical vector:
iris$Species == "setosa"

```

```

## [1] TRUE TRUE
## [13] TRUE TRUE
## [25] TRUE TRUE
## [37] TRUE TRUE
## [49] TRUE TRUE FALSE FALSE
## [61] FALSE FALSE
## [73] FALSE FALSE
## [85] FALSE FALSE
## [97] FALSE FALSE
## [109] FALSE FALSE
## [121] FALSE FALSE
## [133] FALSE FALSE
## [145] FALSE FALSE FALSE FALSE FALSE FALSE

```

```

# which identifies which elements of vector are TRUE:
which(iris$Species == "setosa")

```

```
## [1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
## [26] 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
```

Multiple conditions can be used to select data using the `&` (and) and `|` (or) operators.

```
# not run (long output):
iris[which(iris$Species == "versicolor" & iris$Petal.Length <= 4),]
iris[which(iris$Species == "versicolor" | iris$Petal.Length <= 1.6),]
```

The examples above could also be accomplished in several lines; in fact, splitting the selection criteria up can make for cleaner code. The commands below also show how splitting the selection commands up can help you abide by the “single point of definition” principle.

```
# not run (long output):
flag1 <- iris$Species == "versicolor"
flag2 <- iris$Petal.Length <= 4
iris[flag1 & flag2,]
iris[flag1 | flag2,]
```

More than 2 conditions can be combined just as easily:

```
flag1 <- iris$Species == "versicolor"
flag2 <- iris$Petal.Length <= 4
flag3 <- iris$Petal.Length > 0.5
iris[flag1 & flag2 & flag3,]
```

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
## 54	5.5	2.3	4.0	1.3	versicolor
## 58	4.9	2.4	3.3	1.0	versicolor
## 60	5.2	2.7	3.9	1.4	versicolor
## 61	5.0	2.0	3.5	1.0	versicolor
## 63	6.0	2.2	4.0	1.0	versicolor
## 65	5.6	2.9	3.6	1.3	versicolor
## 70	5.6	2.5	3.9	1.1	versicolor
## 72	6.1	2.8	4.0	1.3	versicolor
## 80	5.7	2.6	3.5	1.0	versicolor
## 81	5.5	2.4	3.8	1.1	versicolor
## 82	5.5	2.4	3.7	1.0	versicolor
## 83	5.8	2.7	3.9	1.2	versicolor
## 90	5.5	2.5	4.0	1.3	versicolor
## 93	5.8	2.6	4.0	1.2	versicolor
## 94	5.0	2.3	3.3	1.0	versicolor
## 99	5.1	2.5	3.0	1.1	versicolor

Selections can return 0 matches. This often manifests as a data frame or matrix with 0 rows.

```

flag1 <- iris$Species == "versicolor"
flag2 <- iris$Petal.Length <= 1
iris[flag1 & flag2,]

## [1] Sepal.Length Sepal.Width  Petal.Length Petal.Width  Species
## <0 rows> (or 0-length row.names)

```

A related function is the binary operator¹ `%in%`. For a command like `A %in% B`, R returns a vector of logical values for each element in A, indicating whether those values occur in B.

```

A <- c("a", "b", "c", "d", "e")
B <- c("c", "d", "e", "f")
A %in% B

## [1] FALSE FALSE  TRUE  TRUE  TRUE
B %in% A

## [1]  TRUE  TRUE  TRUE FALSE

```

One example of how this might be useful is if you want to select observations from a dataset containing one of several values. The example below illustrates this with `iris`:

```

# not run (long output):
iris[iris$Species %in% c("setosa", "versicolor"),]

```

We'll go over some more complicated selection methods in a future section, but what you've seen so far can be very powerful.

3.4 Managing dates and characters

Most of the data we deal with in R are numbers, but textual and temporal data are important two. Working with dates, times, and character strings can be significantly more complicated than working with numbers...so, I've put together an entire page on it. I am the first to admit that the approaches demonstrated here are a little “old-school” compared to some of the newer packages. In particular, many people prefer the tidyverse package `lubridate` (Gromelund and Wickham 2011) for temporal data. Much of this guide is based on Spector (2008).

3.4.1 Temporal data and dates

Temporal data can be read and interpreted by R in many formats. The general rule is to use the simplest way that you can, because the date and time classes

¹A binary operator uses two arguments, one before the symbol and one after. For example, `+` is a binary operator that adds the argument before the `+` to the argument after the `+`.

can be very complicated. For example, don't store date and time if you only need date. This doesn't mean to discard potentially useful data; just to be judicious in what you import to and try to manipulate in R.

It should be noted that the easiest way to deal with dates and times is "don't". In many situations it's actually not necessary to use temporal data classes. For example, if a field study was conducted in two years, and there is no temporal component to the question, it might be sufficient to just call the years `year1` and `year2` in the dataset. Or, if an elapsed time is used as a variable, it might make more sense to store the elapsed time (e.g., 10 minutes, 11.2 minutes, etc.) rather than use full date/time codes for each observation time.

3.4.1.1 Dates without times

The easiest way to handle dates is to use the base function `as.Date()`. This function converts character strings to the R date format. This implies that you will have a much easier time if you send dates to R in YYYY-MM-DD format. You can do this in Excel by setting a custom number format. Once in R, use `as.Date()` to convert your properly-formatted text strings into date objects. Once converted, R can do simple arithmetic such as finding intervals between dates.

```
a <- c("2021-07-23", "2021-08-01", "2021-09-03")
b <- as.Date(a)
class(a)

## [1] "character"

class(b)

## [1] "Date"
b[2] - b[1]

## Time difference of 9 days
```

The examples above use the ISO 8601 date format, Year-Month-Day (or YYYY-MM-DD). This format has the advantage that sorting it alphanumerically is the same as sorting it chronologically. Each element has a fixed number of digits to avoid issues like January and October (months 1 and 10) being sorted next to each other.

I suggest that you get in the habit of storing dates this way. This format is the default date format in R (and many other programs), such that R can read dates in this format without additional arguments. If you insist on storing dates in other formats, you can still read those dates into R. You just need to tell `as.Date()` the format that the date is in. The syntax for formatting dates and times is handled by the function `strptime()` (see `?strptime` for more examples). Something common to all of the formats below is that each element has a *fixed number of digits*, 4 for year and 2 for month or day. This is to

avoid ambiguity: does “7/2/2019” “July second, 2019”, or “February 7, 2019”? This also avoids issues arising from treating numbers as character strings. For example, January and October (months 1 and 10) might be sorted next to each other unless represented by “01” and “10” instead of “1” and “10”.

```
# YYYYMMDD
a <- "20210721"
as.Date(a, "%Y%m%d")

## [1] "2021-07-21"

# MMDDYYYY
a <- "07212021"
as.Date(a, "%m%d%Y")

## [1] "2021-07-21"

# MM/DD/YYYY
a <- "07/21/2021"
as.Date(a, "%m/%d/%Y")

## [1] "2021-07-21"

# DD-MM-YYYY
a <- "21-07-2021"
as.Date(a, "%d-%m-%Y")

## [1] "2021-07-21"
```

An even simpler option for dates, if all dates are within the same year, is to use Julian dates (i.e., day-of-year). If you want to read Julian dates into R as dates, you can also use `as.Date()`. Notice in the example below that the Julian dates must be supplied as *character strings with 3 digits*, not as numerals. Notice also that R assigns the current year (2021) to the dates unless you specifically tell it to do something else.

```
a <- as.character(c(204, 213, 246))
as.Date(a, "%j")

## [1] "2022-07-23" "2022-08-01" "2022-09-03"

## not run:
## causes error because nchar("15") < 3:
#a <- c(15, 204, 213, 246)
#as.Date(a, "%j")

# convert all numbers to 3 digit
# character strings first with formatC():
a <- c(15, 204, 213, 246)
x <- formatC(a, width=3, flag="0")
as.Date(x, "%j")
```

```
## [1] "2022-01-15" "2022-07-23" "2022-08-01" "2022-09-03"
```

Dates can be reformatted using the function `format()`. Notice that the output is a character string, not a date. The syntax for the conversion comes from `strptime()`, just as with `as.Date()` above. If you need to use a formatted date (character string) as a date (R class) you need to convert it with `as.Date()`.

```
a <- c("2021-07-23", "2021-08-01", "2021-09-03")
x <- as.Date(a)
format(x, "%j")
```

```
## [1] "204" "213" "246"
format(x, "%Y-%m-%d")

## [1] "2021-07-23" "2021-08-01" "2021-09-03"
format(x, "%b/%d/%y")

## [1] "Jul/23/21" "Aug/01/21" "Sep/03/21"
```

3.4.1.2 Dates with times (and time zones)

Here be dragons.

Things get more complicated when time is included with a date, but the basic procedures are the same as date-only data. By default, R assumes that times are in the time zone of the current user. So, if you are collaborating with someone in a different time zone, you need to account for time zone in your code. Another default behavior is that R will assume that times without dates are on the current day.

The main function for working with dates *and* times is `strptime()`. This function takes in a character vector and a format argument, which tells R how the character vector is formatted. It returns objects of class `POSIXlt` and `POSIXt`, which contain date and time information in a standardized way.

```
a <- c("01:15:03", "14:28:21")
strptime(a, format="%H:%M:%S") # includes seconds

## [1] "2022-01-13 01:15:03 EST" "2022-01-13 14:28:21 EST"
strptime(a, format="%H:%M")      # seconds dropped

## [1] "2022-01-13 01:15:00 EST" "2022-01-13 14:28:00 EST"
```

Notice that the current date was included in the result of `strptime()` (at least, the date when I rendered this page with R Markdown).

Time zones should be specified when working with data that contain dates and times. Remember that R will assume the time is in the current user's time zone. This almost always means the time zone used by the operating system in which

R is running. If you did something silly like record data in one time zone and analyze it in another, you have to be careful. For example, I once ran a field study in Indiana where data were recorded in Eastern time. But, the office where I analyzed the data was in Missouri (Central time). R kept converting the time-stamped data to Central time, which interfered with the analysis until I learned how to work with time zones in R.

```
a <- c("2021-07-23 13:21", "2021-08-01 09:15")
x <- strptime(a, "%Y-%m-%d %H:%M")
x

## [1] "2021-07-23 13:21:00 EDT" "2021-08-01 09:15:00 EDT"

# compare to :
a2 <- c("13:21", "09:15")
x2 <- strptime(a2, "%H:%M")
x2

## [1] "2022-01-13 13:21:00 EST" "2022-01-13 09:15:00 EST"
```

Time zone names are not always portable across systems. R has a list of built-in time zones that it can handle, and these should work, but be careful and *always* validate any code that depends on time zones. The list of time zones R can use can be printed to the console with the command `OlsonNames()`.

```
# US Central Daylight Time (5 hours behind UTC)
a <- c("2021-07-23 13:21", "2021-08-01 09:15")
b <- strptime(a, "%Y-%m-%d %H:%M", tz="US/Central")
```

Internally, dates and times are stored as the number of seconds elapsed since 00:00 on 1 January 1970, in UTC. You can see these values with `as.numeric()`. The time zone of a value affects what is printed to the console. That time zone is stored by R as an **attribute** of the value. Attributes are a kind of metadata associated with some R data types that can be accessed with `attributes()`.

```
attributes(b)$tzone

## [1] "US/Central" "CST"          "CDT"

To change the time zone associated with value, simply set the attribute to the desired time zone. This is basically telling R the time zone that a date and time refer to. Note that will change the actual value of a date/time. It does NOT translate times in one time zone to another.

as.numeric(b)

## [1] 1627064460 1627827300
attr(b, "tzone") <- "America/Los_Angeles"

# different by 7200 seconds (2 hours)
```

```

as.numeric(b)

## [1] 1627071660 1627834500

To convert one time zone to another, the values must first be converted to the
POSIXct class.

b <- strptime(a, "%Y-%m-%d %H:%M", tz="US/Central")
b2 <- as.POSIXct(b)

# check time zone
attributes(b2)$tzone

## [1] "US/Central"

# check actual time (seconds since epoch)
as.numeric(b2)

## [1] 1627064460 1627827300

# change time zone of POSIXct object
attr(b2, "tzone") <- "Europe/London"

# actual time same as before:
as.numeric(b2)

## [1] 1627064460 1627827300

```

3.4.1.3 Working with dates and times

Why should we go through all of this trouble to format dates and times as dates and times? The answer is that R can perform many useful calculations with dates and times. One of these is time intervals by using subtraction or the function `difftime()`. Subtracting two date/times will return an answer but perhaps not in the units you want; use `difftime()` if you want to specify the units.

```

# how many days, hours, and minutes until Prof. Green can retire?

## current time:
a <- Sys.time()

## retirement date per USG:
b <- "2048-07-01 00:00"

## convert to POSIXlt
x <- strptime(a, "%Y-%m-%d %H:%M")
y <- strptime(b, "%Y-%m-%d %H:%M")

```

```
## calculate time in different units:
difftime(y,x, units="days")

## Time difference of 9665.276 days
difftime(y,x, units="hours")

## Time difference of 231966.6 hours
difftime(y,x, units="mins")

## Time difference of 13917997 mins
y-x

## Time difference of 9665.276 days
```

Another application: what is the time difference between Chicago and Honolulu?

```
a <- c("2021-07-23 13:21")
x <- strptime(a, "%Y-%m-%d %H:%M", tz="US/Central")
y <- strptime(a, "%Y-%m-%d %H:%M", tz="US/Hawaii")
difftime(x,y)
```

```
## Time difference of -5 hours
```

Oddly, `difftime()` cannot return years. This is probably because there are different types of years (sidereal, tropical, anomalistic, lunar, etc.). You can convert a time difference in days to one in years by dividing by the number of days in the kind of year you want. The example below converts to does this and changes the “units” attribute of the `difftime` output to match.

```
d <- difftime(y,x, units="days")

# julian year
d <- d/365.25
attr(d, "units") <- "years"
d
```

```
## Time difference of 0.0005703856 years
```

The Julian year is useful because it is defined in terms of an SI units as 31557600 seconds, or 365.25 days². Most of the alternative years at the link above vary by less than a few minutes.

R can also generate regular sequences of dates and times just like it can numeric values. See `?seq.date` for details and syntax.

```
a <- as.Date("2021-07-23")
seq(a, by="day", length=10)
```

²This is slightly longer than a *Rent* year, which is 525600 minutes (31536000 seconds).

```

## [1] "2021-07-23" "2021-07-24" "2021-07-25" "2021-07-26" "2021-07-27"
## [6] "2021-07-28" "2021-07-29" "2021-07-30" "2021-07-31" "2021-08-01"
seq(a, by="days", length=6)

## [1] "2021-07-23" "2021-07-24" "2021-07-25" "2021-07-26" "2021-07-27"
## [6] "2021-07-28"
seq(a, by="week", length=10)

## [1] "2021-07-23" "2021-07-30" "2021-08-06" "2021-08-13" "2021-08-20"
## [6] "2021-08-27" "2021-09-03" "2021-09-10" "2021-09-17" "2021-09-24"
seq(a, by="6 weeks", length=10)

## [1] "2021-07-23" "2021-09-03" "2021-10-15" "2021-11-26" "2022-01-07"
## [6] "2022-02-18" "2022-04-01" "2022-05-13" "2022-06-24" "2022-08-05"
seq(a, by="-4 weeks", length=5)

## [1] "2021-07-23" "2021-06-25" "2021-05-28" "2021-04-30" "2021-04-02"

```

3.4.2 Character data (text)

Recall from earlier that “character” is just another **type** of data in R—the type that contains text. This means that text values can be stored and manipulated in many of the same ways as numeric data. The vectorized functions in R can make managing your text-based data very fast and efficient.

A value containing text, or stored as text, is called a **character string**. Character strings can be stored in many R objects. A set of character strings is called a **character vector**.

3.4.2.1 Combining character strings

Character strings can be combined together into larger character strings using the function `paste()`. Notice that this is not the same as making a vector containing several character strings. That operation is done with `c()`.

```

x <- paste("a", "b", "c")
y <- c("a", "b", "c")
x

## [1] "a b c"
y

## [1] "a" "b" "c"
length(x)

## [1] 1

```

```
length(y)
```

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

The function `paste()`, and its cousin `paste0()`, combine their arguments together and output a character string. The inputs do not have to be character strings, but the outputs will be. The difference between `paste()` and `paste0()` is that `paste0()` does not put any characters between its inputs, while `paste()` can place any valid character between its inputs. The default separator is a space.

```
paste0("a", "zebra", 3)
```

```
## [1] "azebra3"
```

```
paste("a", "zebra", 3)
```

```
## [1] "a zebra 3"
```

```
paste("a", "zebra", 3, sep="_")
```

```
## [1] "a_zebra_3"
```

```
paste("a", "zebra", 3, sep=".")
```

```
## [1] "a.zebra.3"
```

```
paste("a", "zebra", 3, sep="/")
```

```
## [1] "a/zebra/3"
```

```
paste("a", "zebra", 3, sep="giraffe")
```

```
## [1] "agiraffezebrafragiraffe3"
```

One of my favorite uses for the `paste` functions is to automatically generate folder names and output file names. The example below adds the current date (`Sys.Date()`) into a file name.

```
# not run
top.dir <- "C:/_data"
out.dir <- paste(top.dir, "test", sep="/")
out.name <- paste0("output_", Sys.Date(), ".csv")
write.csv(iris, paste(geo.dir, out.name, sep="/"))
```

When the inputs to a `paste()` function are contained in a single vector, then the desired separator must be supplied using the argument `collapse` instead of `sep`.

```
a <- c("a", "b", "c")
paste(a, collapse="-")
```

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

3.4.2.2 Printing character strings

The `cat()` function can not only put strings together, but also print them to the R console. This is useful if you want to generate messages from inside functions or loops. When `cat()` receives inputs that are not character strings, it evaluates them and converts them to character strings. The examples below show how inputs can be evaluated and printed. It also illustrates the use of `\n` and `\t` to insert new lines and tabs, respectively. Finally, the command `flush.console()` ensures that the console is updated with the most recent outputs.

```
# using cat() in a function:
my.fun <- function(x){
  cat("You entered", x, "!", "\n")
  cat("\t", "1 more is", x+1, "\n")
}
my.fun(7)

## You entered 7 !
##   1 more is 8
my.fun(42)

## You entered 42 !
##   1 more is 43

# using cat() in a loop:
for(i in 1:5){
  cat("starting iteration", i, "\n")
  for(j in 1:3){
    cat("\t", "iteration", i, "part", j, "complete!", "\n")
    flush.console()
  }
  cat("iteration", i, "complete!", "\n")
}

## starting iteration 1
##   iteration 1 part 1 complete!
##   iteration 1 part 2 complete!
##   iteration 1 part 3 complete!
## iteration 1 complete!
## starting iteration 2
##   iteration 2 part 1 complete!
##   iteration 2 part 2 complete!
##   iteration 2 part 3 complete!
## iteration 2 complete!
## starting iteration 3
```

```

##   iteration 3 part 1 complete!
##   iteration 3 part 2 complete!
##   iteration 3 part 3 complete!
## iteration 3 complete!
## starting iteration 4
##   iteration 4 part 1 complete!
##   iteration 4 part 2 complete!
##   iteration 4 part 3 complete!
## iteration 4 complete!
## starting iteration 5
##   iteration 5 part 1 complete!
##   iteration 5 part 2 complete!
##   iteration 5 part 3 complete!
## iteration 5 complete!

```

The other printing function, `sink()`, is specifically for printing outputs into text files. Refer back to the section on data import/export for an example.

3.4.2.3 Matching character strings

Like numeric data, character strings can be matched exactly with the comparison operator `==`.

```
x <- c("zebra", "hippo")
```

```
x == "zebra"
```

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

```
which(x == "zebra")
```

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

```
which(x != "zebra")
```

```
## [1] 2
```

```
any(x == "zebra")
```

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

```
all(x == "zebra")
```

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

Notice that this kind of matching is case-sensitive. If you need to match regardless of case, you can either force all of the comparands to be the same case, or use regular expressions (see next section).

```
x <- c("zebra", "hippo")
```

```
any(x == "Zebra")
```

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

```

any(x == tolower("zebra"))

## [1] TRUE

R is smart enough to sort text alphabetically, and to make comparisons based
on alphabetical sorting.

x <- c("zebra", "hippo", "lion")
sort(x)

## [1] "hippo" "lion"  "zebra"
"zebra" > "hippo"

## [1] TRUE
"lion" > "zebra"

## [1] FALSE
which(x > "jaguar")

## [1] 1 3

```

3.4.2.4 Searching with grep() and grepl()

Regular expressions, or **regex**, are a syntax for searching text strings. Learning how to use regular expressions is easily a semester-long course on its own. Here we'll just explore a few common use cases. The R functions that deal with regular expressions are `grep()` and its cousins. The most commonly used versions are `grep()` and `grepl()`.

- `grep()` returns the indices in the input vector that match the pattern
- `grepl()` returns a logical value (TRUE or FALSE) telling whether each element in the input vector matches the pattern.

This means that `grepl()` is analogous to using `==` to test for equality, while `grep()` is analogous to using `which()`. Compare the results of the two commands on the simple vector `x` below. It's not hard to imagine how powerful these functions can be.

```

x <- c("zebra", "hippo", "lion", "tiger")
grep("e", x)

## [1] 1 4

grepl("e", x)

## [1] TRUE FALSE FALSE TRUE

```

Some of the arguments to `grep()` can save you some time. Setting `value=TRUE` will return the values where the input matches the pattern.

```
grep("e", x, value=TRUE)
```

```
## [1] "zebra" "tiger"
# equivalent to:
x[grep("e", x)]
```

```
## [1] "zebra" "tiger"
```

Setting `invert=TRUE` will find elements of the input that do NOT match the pattern.

```
grep("e", x)
```

```
## [1] 1 4
grep("e", x, invert=TRUE)
```

```
## [1] 2 3
```

By default, `grep()` is case-sensitive. This can be overridden with the option `ignore.case`.

```
x <- c("zebra", "hippo", "lion", "tiger", "Tiger")
grep("tiger", x)
```

```
## [1] 4
grep("Tiger", x)
```

```
## [1] 5
grep("tiger", x, ignore.case=TRUE)
```

```
## [1] 4 5
grep("TIGER", x, ignore.case=TRUE)
```

```
## [1] 4 5
```

We will see some uses of `grep()` and its cousins in the examples later.

3.4.2.5 Replacing parts of character strings

Text within character strings can be replaced or modified in several ways. If you are comfortable with regular expressions (see previous section) then you can use function `gsub()`. The syntax for `gsub()` is similar to that of `grep()`, with an additional input for the replacement string. Note that `gsub()` will replace every occurrence of the pattern, whether or not an occurrence is a whole word or not. The related function `sub()` works like `gsub()`, but only replaces the first occurrence of the pattern.

```
x <- c("zebra", "hippo", "lion", "tiger", "Tiger")
gsub("tiger", "giraffe", x)

## [1] "zebra"    "hippo"     "lion"      "giraffe"   "Tiger"
gsub("tiger", "giraffe", x, ignore.case=TRUE)

## [1] "zebra"    "hippo"     "lion"      "giraffe"   "giraffe"
gsub("z", "Z", x)

## [1] "Zebra"    "hippo"     "lion"      "tiger"     "Tiger"
sub("tiger", "giraffe", x, ignore.case=TRUE)

## [1] "zebra"    "hippo"     "lion"      "giraffe"   "giraffe"
```

3.4.2.6 Extracting from and splitting character strings

Character strings can be split up using the function `strsplit()` (short for “string split”). This function can use regular expressions like `grep()` or `gsub()` can. It works by breaking a character string apart at a given text string. Unfortunately, its output can be a little confusing if you’re not used to working with R objects:

```
x <- c("Peromyscus_leucopus",
       "Sigmodon_hispidus",
       "Microtus_pennsylvanicus")
strsplit(x, "_")

## [[1]]
## [1] "Peromyscus" "leucopus"
##
## [[2]]
## [1] "Sigmodon"   "hispidus"
##
## [[3]]
## [1] "Microtus"    "pennsylvanicus"
```

Function `strsplit()` returns a list with one element for each element in the input vector. Each element of that list is a character vector, consisting of the pieces of that element of the input vector split by the splitting string. So, if you want only one part of each element of the input vector, you need to extract them from the output list. In this example, we can get the generic epithets by pulling the first element of each list element. The function `sapply()` applies a function to each element of a list and returns a result with the simplest possible structure (`sapply` is short for “apply and simplify”). The two commands below are equivalent, but the first is much simpler and should be the method you use.

```
sapply(strsplit(x, "_"), "[", 1)

## [1] "Peromyscus" "Sigmodon"    "Microtus"
# equivalent, not as easy:
do.call(c, lapply(strsplit(x, "_"), "[", 1))

## [1] "Peromyscus" "Sigmodon"    "Microtus"
```

The “`sapply-strsplit`” method is useful for breaking apart strings whose parts carry information. One common use case is breaking up dates into year, month, and day. Note that there are more elegant ways to do this using the R date classes, but treating dates as text strings can be convenient because the date and time classes are such a pain in the neck. When using this method, you will need to keep track of when date components are stored as text and when they are stored as numbers.

The code block below shows how to break apart dates using `strsplit()` and `sapply()`.

```
x <- c("2019-08-03", "2020-08-15", "2021-08-07")
ssx <- strsplit(x, "-")
a <- data.frame(year=sapply(ssx, "[", 1),
                 mon=sapply(ssx, "[", 2),
                 day=sapply(ssx, "[", 3))

class(a$year) # character, not a number!

## [1] "character"
class(a$mon) # character, not a number!

## [1] "character"
class(a$day) # character, not a number!

## [1] "character"
a$year <- as.numeric(a$year)
a$mon <- as.numeric(a$mon)
a$day <- as.numeric(a$day)

class(a$year) # numbers!

## [1] "numeric"
class(a$mon) # numbers!

## [1] "numeric"
```

```
class(a$day) # numbers!
```

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

3.4.2.7 Matching text strings

As biologists we often keep our data in several separate spreadsheets or tables. These tables are interrelated by a set of “key” variables. If that sounds like a sketchy, haphazard way to build a database, it is. Databases per se are outside the scope of this course, but we will learn how to relate tables using R. If you’ve ever used the VLOOKUP function in Excel, then you know exactly what problem `match()` is designed to solve.

The function `match()` finds the indices of the matches of its first argument in its second. In other words, it searches for the values in one vector in a second vector, then returns the positions of the matches from the second vector. A simple example is below:

```
vec1 <- c("gene1", "gene2", "gene3", "gene4")
vec2 <- c("gene1", "gene2", "gene1", "gene2",
         "gene5", "gene3", "gene3", "gene4")
vec3 <- c("gene1", "gene2", "gene1", "gene2",
         "gene4", "gene3", "gene3", "gene4")

match(vec1, vec2)
## [1] 1 2 6 8
match(vec2, vec1)
## [1] 1 2 1 2 NA 3 3 4
match(vec3, vec1)
## [1] 1 2 1 2 4 3 3 4
```

The first example shows what `match()` is doing. For element in the first vector, `match()` returns the first position where that value occurs in the second vector. The second example shows what happens when trying to match a value that is not in the second vector: R returns `NA`. The third example shows how `match()` can be used to relate tables. The longer vector `vec3` is matched to the shorter vector `vec1`, which contains each unique value only once. If `vec1` was a variable in a data frame that contained other variables associated with genes, then the result of `match(vec3, vec1)` would return the rows in `vec1`’s data frame associated with each row of `vec3`’s data frame.

The code below demonstrates how `match()` might be used in practice. This example uses three tables from a real dataset. The three tables contain data and metadata for samples taken from stream across Georgia. The commands use

`match()` to assign site numbers, dates, and geographic coordinates to observations using different key variables. We'll revisit these data again in the next section where we see how to merge datasets.

```
# not run (see next section)

options(stringsAsFactors=FALSE)
name1 <- "dat_samples_2021-08-04.csv"
name2 <- "dat_stream_indices_2021-08-04.csv"
name3 <- "loc_meta_vaa_2020-11-18.csv"

x1 <- read.csv(name1, header=TRUE)
x2 <- read.csv(name2, header=TRUE)
x3 <- read.csv(name3, header=TRUE)

# x2 contains the response variable
dat <- x2
# add site name, year, and date from x1
matchx <- match(dat$uocc, x1$uocc)
dat$site <- x1$loc[matchx]
dat$year <- x1$year[matchx]
dat$doy <- x1$doy[matchx]

# equivalent to:
#dat$site <- x1$loc[match(dat$uocc, x1$uocc)]
#dat$year <- x1$year[match(dat$uocc, x1$uocc)]
#dat$doy <- x1$doy[match(dat$uocc, x1$uocc)]

# grab latitude and longitude from x3
matchx <- match(dat$site, x3$loc)
dat$lat <- x3$lat[matchx]
dat$long <- x3$long[matchx]
```

3.4.2.8 Standardizing numbers as text

The function `formatC()` formats numbers in a manner similar to the language C. This is useful for naming objects or parts of objects with numeric names that have a fixed width or format. The most common use case is converting numbers like “1, 12, 112” to strings like “001, 012, 112”. The example below uses `formatC()` in combination with `paste()` to generate some file names.

```
file.nums <- formatC(1:4, width=3, flag="0")
file.names <- paste("file", file.nums, sep="_")
file.names

## [1] "file_001" "file_002" "file_003" "file_004"
```

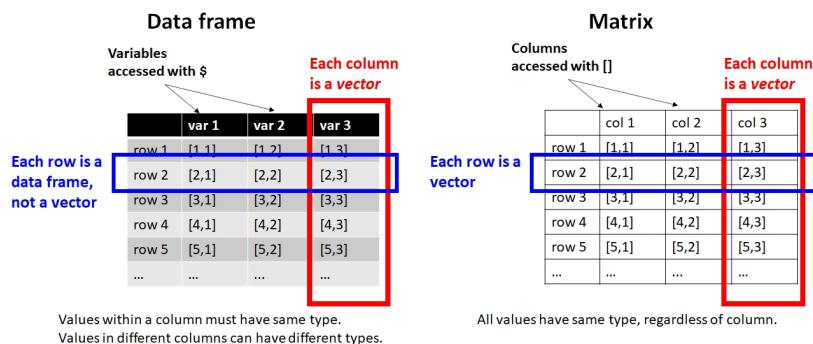
3.5 Data frame management

Most of the time when you work with data in R, those data will be stored in **data frames**. A data frame looks like a matrix, but functions in many ways spreadsheet. Internally, a data frame is really a list, but for most day-to-day use you can think of a data frame as a table.

Being able to manage data frames is an important part of working with R. This page demonstrates some of the most common data frame operations.

3.5.1 Data frame structure

Data frames and matrices look very similar, but their internal structures are very different. This has huge implications for how you work with and manipulate data frames and matrices. The figure below shows some of the most important differences.



All pieces of a matrix are accessed using bracket notation `([])`. Pieces of a data frame can also be accessed using brackets, but the dollar sign `$` can be used to access columns.

```
# single variables (columns): $  
iris$Petal.Length  
  
## [1] 1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 1.5 1.6 1.4 1.1 1.2 1.5 1.3 1.4  
## [19] 1.7 1.5 1.7 1.5 1.0 1.7 1.9 1.6 1.6 1.5 1.4 1.6 1.6 1.5 1.5 1.4 1.5 1.2  
## [37] 1.3 1.4 1.3 1.5 1.3 1.3 1.6 1.9 1.4 1.6 1.4 1.5 1.4 4.7 4.5 4.9 4.0  
## [55] 4.6 4.5 4.7 3.3 4.6 3.9 3.5 4.2 4.0 4.7 3.6 4.4 4.5 4.1 4.5 3.9 4.8 4.0  
## [73] 4.9 4.7 4.3 4.4 4.8 5.0 4.5 3.5 3.8 3.7 3.9 5.1 4.5 4.5 4.7 4.4 4.1 4.0  
## [91] 4.4 4.6 4.0 3.3 4.2 4.2 4.2 4.3 3.0 4.1 6.0 5.1 5.9 5.6 5.8 6.6 4.5 6.3  
## [109] 5.8 6.1 5.1 5.3 5.5 5.0 5.1 5.3 5.5 6.7 6.9 5.0 5.7 4.9 6.7 4.9 5.7 6.0  
## [127] 4.8 4.9 5.6 5.8 6.1 6.4 5.6 5.1 5.6 6.1 5.6 5.5 4.8 5.4 5.6 5.1 5.1 5.9  
## [145] 5.7 5.2 5.0 5.2 5.4 5.1  
  
# multiple variables: brackets []  
iris[,c("Petal.Length", "Sepal.Length")]
```

```
##      Petal.Length Sepal.Length
## 1          1.4         5.1
## 2          1.4         4.9
## 3          1.3         4.7
## 4          1.5         4.6
## 5          1.4         5.0
## 6          1.7         5.4
## 7          1.4         4.6
## 8          1.5         5.0
## 9          1.4         4.4
## 10         1.5         4.9
## 11         1.5         5.4
## 12         1.6         4.8
## 13         1.4         4.8
## 14         1.1         4.3
## 15         1.2         5.8
## 16         1.5         5.7
## 17         1.3         5.4
## 18         1.4         5.1
## 19         1.7         5.7
## 20         1.5         5.1
## 21         1.7         5.4
## 22         1.5         5.1
## 23         1.0         4.6
## 24         1.7         5.1
## 25         1.9         4.8
## 26         1.6         5.0
## 27         1.6         5.0
## 28         1.5         5.2
## 29         1.4         5.2
## 30         1.6         4.7
## 31         1.6         4.8
## 32         1.5         5.4
## 33         1.5         5.2
## 34         1.4         5.5
## 35         1.5         4.9
## 36         1.2         5.0
## 37         1.3         5.5
## 38         1.4         4.9
## 39         1.3         4.4
## 40         1.5         5.1
## 41         1.3         5.0
## 42         1.3         4.5
## 43         1.3         4.4
## 44         1.6         5.0
## 45         1.9         5.1
```

```
## 46      1.4      4.8
## 47      1.6      5.1
## 48      1.4      4.6
## 49      1.5      5.3
## 50      1.4      5.0
## 51      4.7      7.0
## 52      4.5      6.4
## 53      4.9      6.9
## 54      4.0      5.5
## 55      4.6      6.5
## 56      4.5      5.7
## 57      4.7      6.3
## 58      3.3      4.9
## 59      4.6      6.6
## 60      3.9      5.2
## 61      3.5      5.0
## 62      4.2      5.9
## 63      4.0      6.0
## 64      4.7      6.1
## 65      3.6      5.6
## 66      4.4      6.7
## 67      4.5      5.6
## 68      4.1      5.8
## 69      4.5      6.2
## 70      3.9      5.6
## 71      4.8      5.9
## 72      4.0      6.1
## 73      4.9      6.3
## 74      4.7      6.1
## 75      4.3      6.4
## 76      4.4      6.6
## 77      4.8      6.8
## 78      5.0      6.7
## 79      4.5      6.0
## 80      3.5      5.7
## 81      3.8      5.5
## 82      3.7      5.5
## 83      3.9      5.8
## 84      5.1      6.0
## 85      4.5      5.4
## 86      4.5      6.0
## 87      4.7      6.7
## 88      4.4      6.3
## 89      4.1      5.6
## 90      4.0      5.5
## 91      4.4      5.5
```

```
## 92      4.6      6.1
## 93      4.0      5.8
## 94      3.3      5.0
## 95      4.2      5.6
## 96      4.2      5.7
## 97      4.2      5.7
## 98      4.3      6.2
## 99      3.0      5.1
## 100     4.1      5.7
## 101     6.0      6.3
## 102     5.1      5.8
## 103     5.9      7.1
## 104     5.6      6.3
## 105     5.8      6.5
## 106     6.6      7.6
## 107     4.5      4.9
## 108     6.3      7.3
## 109     5.8      6.7
## 110     6.1      7.2
## 111     5.1      6.5
## 112     5.3      6.4
## 113     5.5      6.8
## 114     5.0      5.7
## 115     5.1      5.8
## 116     5.3      6.4
## 117     5.5      6.5
## 118     6.7      7.7
## 119     6.9      7.7
## 120     5.0      6.0
## 121     5.7      6.9
## 122     4.9      5.6
## 123     6.7      7.7
## 124     4.9      6.3
## 125     5.7      6.7
## 126     6.0      7.2
## 127     4.8      6.2
## 128     4.9      6.1
## 129     5.6      6.4
## 130     5.8      7.2
## 131     6.1      7.4
## 132     6.4      7.9
## 133     5.6      6.4
## 134     5.1      6.3
## 135     5.6      6.1
## 136     6.1      7.7
## 137     5.6      6.3
```

```

## 138      5.5      6.4
## 139      4.8      6.0
## 140      5.4      6.9
## 141      5.6      6.7
## 142      5.1      6.9
## 143      5.1      5.8
## 144      5.9      6.8
## 145      5.7      6.7
## 146      5.2      6.7
## 147      5.0      6.3
## 148      5.2      6.5
## 149      5.4      6.2
## 150      5.1      5.9

# rows: brackets []
iris[1:4,]

##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1      5.1        3.5       1.4        0.2  setosa
## 2      4.9        3.0       1.4        0.2  setosa
## 3      4.7        3.2       1.3        0.2  setosa
## 4      4.6        3.1       1.5        0.2  setosa

iris[1,]

##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1      5.1        3.5       1.4        0.2  setosa

```

Notice that the syntax for accessing a data frame with [] is the same as for accessing a matrix.

When a data frame is printed to the console, this is only a representation of the data frame's true form: a **list**. This is why single columns are accessible by a \$: this is the symbol for accessing an element of a list. The list nature of a data frame is also why **rows** of a dataset are actually data frames themselves and not vectors.

3.5.2 Common data frame operations

3.5.2.1 Adding and deleting variables

New variables can be added to a data frame by **assignment** (<-). Assigning values to a variable in a data frame will create that variable if it doesn't already exist—and overwrite values if the variable is already in the data frame.

```

x <- iris[1:5,] # spare copy for demonstration
x$Petal.Area <- apply(x[,3:4], 1, prod)

# shows new variable:

```

```
x

##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species Petal.Area
## 1      5.1       3.5      1.4       0.2  setosa    0.28
## 2      4.9       3.0      1.4       0.2  setosa    0.28
## 3      4.7       3.2      1.3       0.2  setosa    0.26
## 4      4.6       3.1      1.5       0.2  setosa    0.30
## 5      5.0       3.6      1.4       0.2  setosa    0.28

# shows variable overwritten:
x$Petal.Area <- 3
```

```
x

##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species Petal.Area
## 1      5.1       3.5      1.4       0.2  setosa     3
## 2      4.9       3.0      1.4       0.2  setosa     3
## 3      4.7       3.2      1.3       0.2  setosa     3
## 4      4.6       3.1      1.5       0.2  setosa     3
## 5      5.0       3.6      1.4       0.2  setosa     3
```

Variables can also be added using function `cbind()`. This name is short for “column bind”³. To use `cbind()`, the input data frames must have the same number of rows. Note that `cbind()` sticks the data frames together as they are. If the records in the inputs need to be matched up and are not already in the same order, then either the data frames need to be aligned, merged with function `merge()`, or the variables need to be added using `match()` and a key variable.

```
x <- iris[1:5, 1:2]
y <- iris[1:5, 3:5]
cbind(x,y)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1      5.1       3.5      1.4       0.2  setosa
## 2      4.9       3.0      1.4       0.2  setosa
## 3      4.7       3.2      1.3       0.2  setosa
## 4      4.6       3.1      1.5       0.2  setosa
## 5      5.0       3.6      1.4       0.2  setosa

# shows that cbind() uses inputs as-is
cbind(x, y[5:1,])
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1      5.1       3.5      1.4       0.2  setosa
## 2      4.9       3.0      1.5       0.2  setosa
## 3      4.7       3.2      1.3       0.2  setosa
## 4      4.6       3.1      1.4       0.2  setosa
## 5      5.0       3.6      1.4       0.2  setosa
```

³`cbind()` also works on matrices.

Variables can be deleted by setting them to NULL (notice all caps) or by making a new version of the data frame without the variable. If I am only removing a few variables, I prefer setting them individually to NULL. This makes it less likely that I will accidentally remove the wrong columns.

```
x$Petal.Area <- NULL
x

## Sepal.Length Sepal.Width
## 1      5.1      3.5
## 2      4.9      3.0
## 3      4.7      3.2
## 4      4.6      3.1
## 5      5.0      3.6

# delete 1 variable:
x <- x[,which(names(x) != "Petal.Area")]
x

## Sepal.Length Sepal.Width
## 1      5.1      3.5
## 2      4.9      3.0
## 3      4.7      3.2
## 4      4.6      3.1
## 5      5.0      3.6

# delete >1 variable
x <- x[,which(!names(x) %in% c("Petal.Area", "Petal.Length"))]
x

## Sepal.Length Sepal.Width
## 1      5.1      3.5
## 2      4.9      3.0
## 3      4.7      3.2
## 4      4.6      3.1
## 5      5.0      3.6
```

3.5.2.2 Modifying variables

Variables can be modified by assigning them their new values. Note that this overwrites the previous values. If you think you might want the old values for some reason, save the modified values in a new variable instead.

```
x <- iris[1:5,] # spare copy for demonstration
x$Petal.Length <- log(x$Petal.Length)

# better: keep modified values in new variable
x <- iris[1:10,] # spare copy for demonstration
x$log.petlen <- log(x$Petal.Length)
```

3.5.2.3 Changing values in a variable using a rule

By combining the methods for data selection and assignment, values in a variable can be replaced according to their values.

```
# not run (long output):
x <- iris

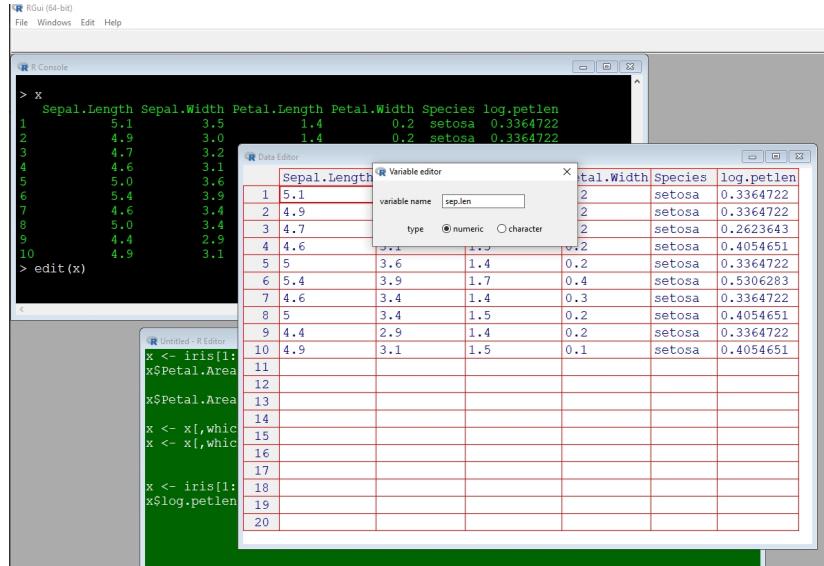
# select and replace by one condition:
flag <- which(x$Sepal.Length >= 6)
x$Sepal.Length[flag] <- 5.99

# select and replace by 2 conditions:
flag1 <- which(x$Petal.Length >= 3)
flag2 <- which(x$Petal.Length <= 5)
x$Petal.Length[intersect(flag1, flag2)] <- 4
```

3.5.2.4 Renaming variables

Renaming variables in Excel is easy: just enter the new name into the cell at the top of a column. Things are a little more complicated in R but not much.

One way is to use the **data frame editor**. Use the `edit()` function to open the data editor, a spreadsheet-like interface for working with data frames. Then click on the column name and enter the new name (and then press ENTER). You can also edit individual values this way, but I don't recommend this method for routine use because there is no record of what was done.



The better way is to rename the variable using R code. There are a few ways to

do this, but they all involve accessing the variable names using function `names()`.

```
x <- iris[1:5,] # spare copy for demonstration

# by column number:
names(x)[1] <- "sep.len"
x

##   sep.len Sepal.Width Petal.Length Petal.Width Species
## 1      5.1        3.5       1.4        0.2  setosa
## 2      4.9        3.0       1.4        0.2  setosa
## 3      4.7        3.2       1.3        0.2  setosa
## 4      4.6        3.1       1.5        0.2  setosa
## 5      5.0        3.6       1.4        0.2  setosa

# match name exactly:
names(x)[which(names(x) == "Petal.Length")] <- "pet.len"
x

##   sep.len Sepal.Width pet.len Petal.Width Species
## 1      5.1        3.5       1.4        0.2  setosa
## 2      4.9        3.0       1.4        0.2  setosa
## 3      4.7        3.2       1.3        0.2  setosa
## 4      4.6        3.1       1.5        0.2  setosa
## 5      5.0        3.6       1.4        0.2  setosa

# by position:
names(x)[ncol(x)] <- "species"
x

##   sep.len Sepal.Width pet.len Petal.Width species
## 1      5.1        3.5       1.4        0.2  setosa
## 2      4.9        3.0       1.4        0.2  setosa
## 3      4.7        3.2       1.3        0.2  setosa
## 4      4.6        3.1       1.5        0.2  setosa
## 5      5.0        3.6       1.4        0.2  setosa

names(x)[1] <- "petal.len"
x

##   petal.len Sepal.Width pet.len Petal.Width species
## 1      5.1        3.5       1.4        0.2  setosa
## 2      4.9        3.0       1.4        0.2  setosa
## 3      4.7        3.2       1.3        0.2  setosa
## 4      4.6        3.1       1.5        0.2  setosa
## 5      5.0        3.6       1.4        0.2  setosa

# change several names at once:
names(x)[1:4] <- paste0("x", 1:4)
x
```

```
##   x1  x2  x3  x4 species
## 1 5.1 3.5 1.4 0.2 setosa
## 2 4.9 3.0 1.4 0.2 setosa
## 3 4.7 3.2 1.3 0.2 setosa
## 4 4.6 3.1 1.5 0.2 setosa
## 5 5.0 3.6 1.4 0.2 setosa
```

Finally, you can make a new copy of the variable with the name you want, then delete the original version. This has the disadvantage of changing the order of columns.

```
x <- iris[1:5,] # spare copy for demonstration
x$pet.len <- x$Petal.Length
x$Petal.Length <- NULL
x
```

```
##   Sepal.Length Sepal.Width Petal.Width Species pet.len
## 1          5.1         3.5        0.2  setosa     1.4
## 2          4.9         3.0        0.2  setosa     1.4
## 3          4.7         3.2        0.2  setosa     1.3
## 4          4.6         3.1        0.2  setosa     1.5
## 5          5.0         3.6        0.2  setosa     1.4
```

3.5.2.5 Rearranging columns

Reordering records or columns within a data frame is accomplished by creating a new version of the dataset with the desired arrangement. This rearrangement is accomplished using methods we have already seen.

```
x <- iris[1:5,]

# rearrange by number
x <- x[,c(5, 3, 4, 1, 2)]
x

##   Species Petal.Length Petal.Width Sepal.Length Sepal.Width
## 1 setosa      1.4       0.2       5.1       3.5
## 2 setosa      1.4       0.2       4.9       3.0
## 3 setosa      1.3       0.2       4.7       3.2
## 4 setosa      1.5       0.2       4.6       3.1
## 5 setosa      1.4       0.2       5.0       3.6

# rearrange by name
col.order <- c("Species", "Sepal.Length", "Sepal.Width",
              "Petal.Length", "Petal.Width")
x <- x[,col.order]
x

##   Species Sepal.Length Sepal.Width Petal.Length Petal.Width
```

```
## 1 setosa      5.1      3.5      1.4      0.2
## 2 setosa      4.9      3.0      1.4      0.2
## 3 setosa      4.7      3.2      1.3      0.2
## 4 setosa      4.6      3.1      1.5      0.2
## 5 setosa      5.0      3.6      1.4      0.2
```

3.5.2.6 Sorting and ordering data

Sorting data frames is mostly done using the function `order()`. This function is usually put inside brackets in the rows slot. The arguments to `order()` define the variables by which the data frame will be sorted. Rows are sorted by the first variable, then the second, and so on. By default, rows are placed in *ascending* order.

```
# not run (long output):
DNase
DNase[order(DNase$conc),]
iris
iris[order(iris$Species, iris$Petal.Length),]
```

You can multiple variables to `order()`, by name, and R will order the rows in each variable in the order supplied. By default, records are sorted in ascending order. If you want a mix of ascending and descending order, you have to specify the `radix` method and specify whether you want the order to be decreasing (TRUE) or not (FALSE) for each variable:

```
# not run (long output):
iris[order(iris$Species, iris$Petal.Length,
           decreasing=c(TRUE, FALSE),
           method="radix"),]

iris[order(iris$Species, iris$Petal.Length,
           decreasing=c(TRUE, TRUE),
           method="radix"),]
```

3.5.2.7 Adding or removing rows

Removing rows works a lot like removing columns. You can use the bracket notation to either remove the rows with negative indices, or make a new version of the data frame with only some of the rows.

```
x <- iris[1:10,]

# remove by row number
x[-1,]

##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
```

```

## 2      4.9      3.0      1.4      0.2  setosa
## 3      4.7      3.2      1.3      0.2  setosa
## 4      4.6      3.1      1.5      0.2  setosa
## 5      5.0      3.6      1.4      0.2  setosa
## 6      5.4      3.9      1.7      0.4  setosa
## 7      4.6      3.4      1.4      0.3  setosa
## 8      5.0      3.4      1.5      0.2  setosa
## 9      4.4      2.9      1.4      0.2  setosa
## 10     4.9      3.1      1.5      0.1  setosa
x[-c(1:3),]

##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 4      4.6      3.1      1.5      0.2  setosa
## 5      5.0      3.6      1.4      0.2  setosa
## 6      5.4      3.9      1.7      0.4  setosa
## 7      4.6      3.4      1.4      0.3  setosa
## 8      5.0      3.4      1.5      0.2  setosa
## 9      4.4      2.9      1.4      0.2  setosa
## 10     4.9      3.1      1.5      0.1  setosa
x[-c(1, 3, 5, 7, 9),]

##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 2      4.9      3.0      1.4      0.2  setosa
## 4      4.6      3.1      1.5      0.2  setosa
## 6      5.4      3.9      1.7      0.4  setosa
## 8      5.0      3.4      1.5      0.2  setosa
## 10     4.9      3.1      1.5      0.1  setosa
# or, make a new copy with the
# rows to keep
x[2:10,]

##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 2      4.9      3.0      1.4      0.2  setosa
## 3      4.7      3.2      1.3      0.2  setosa
## 4      4.6      3.1      1.5      0.2  setosa
## 5      5.0      3.6      1.4      0.2  setosa
## 6      5.4      3.9      1.7      0.4  setosa
## 7      4.6      3.4      1.4      0.3  setosa
## 8      5.0      3.4      1.5      0.2  setosa
## 9      4.4      2.9      1.4      0.2  setosa
## 10     4.9      3.1      1.5      0.1  setosa
x[c(2, 4, 6, 8, 10),]

##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 2      4.9      3.0      1.4      0.2  setosa

```

```

## 4      4.6      3.1      1.5      0.2  setosa
## 6      5.4      3.9      1.7      0.4  setosa
## 8      5.0      3.4      1.5      0.2  setosa
## 10     4.9      3.1      1.5      0.1  setosa

```

Adding rows can be done with `rbind()` (short for “row bind”). This function is analogous to `cbind()`, which joins objects by their columns. Adding rows to a data frame is tantamount to combining it with another data frame. The data frames being combined must have the same columns, with the same names, but the order of columns can be different.

```

x <- iris[1:10,]
y <- iris[15:20,]

# columns match exactly:
rbind(x, y)

##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1      5.1      3.5      1.4      0.2  setosa
## 2      4.9      3.0      1.4      0.2  setosa
## 3      4.7      3.2      1.3      0.2  setosa
## 4      4.6      3.1      1.5      0.2  setosa
## 5      5.0      3.6      1.4      0.2  setosa
## 6      5.4      3.9      1.7      0.4  setosa
## 7      4.6      3.4      1.4      0.3  setosa
## 8      5.0      3.4      1.5      0.2  setosa
## 9      4.4      2.9      1.4      0.2  setosa
## 10     4.9      3.1      1.5      0.1  setosa
## 15     5.8      4.0      1.2      0.2  setosa
## 16     5.7      4.4      1.5      0.4  setosa
## 17     5.4      3.9      1.3      0.4  setosa
## 18     5.1      3.5      1.4      0.3  setosa
## 19     5.7      3.8      1.7      0.3  setosa
## 20     5.1      3.8      1.5      0.3  setosa

# columns in different order:
y2 <- y[,c(3,5,2,1,4)]
rbind(x,y2)

##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1      5.1      3.5      1.4      0.2  setosa
## 2      4.9      3.0      1.4      0.2  setosa
## 3      4.7      3.2      1.3      0.2  setosa
## 4      4.6      3.1      1.5      0.2  setosa
## 5      5.0      3.6      1.4      0.2  setosa
## 6      5.4      3.9      1.7      0.4  setosa
## 7      4.6      3.4      1.4      0.3  setosa
## 8      5.0      3.4      1.5      0.2  setosa

```

```

## 9      4.4      2.9      1.4      0.2  setosa
## 10     4.9      3.1      1.5      0.1  setosa
## 15     5.8      4.0      1.2      0.2  setosa
## 16     5.7      4.4      1.5      0.4  setosa
## 17     5.4      3.9      1.3      0.4  setosa
## 18     5.1      3.5      1.4      0.3  setosa
## 19     5.7      3.8      1.7      0.3  setosa
## 20     5.1      3.8      1.5      0.3  setosa
# result inherits order of columns from first argument:
rbind(y2,x)

##   Petal.Length Species Sepal.Width Sepal.Length Petal.Width
## 15      1.2  setosa      4.0       5.8      0.2
## 16      1.5  setosa      4.4       5.7      0.4
## 17      1.3  setosa      3.9       5.4      0.4
## 18      1.4  setosa      3.5       5.1      0.3
## 19      1.7  setosa      3.8       5.7      0.3
## 20      1.5  setosa      3.8       5.1      0.3
## 1       1.4  setosa      3.5       5.1      0.2
## 2       1.4  setosa      3.0       4.9      0.2
## 3       1.3  setosa      3.2       4.7      0.2
## 4       1.5  setosa      3.1       4.6      0.2
## 5       1.4  setosa      3.6       5.0      0.2
## 6       1.7  setosa      3.9       5.4      0.4
## 7       1.4  setosa      3.4       4.6      0.3
## 8       1.5  setosa      3.4       5.0      0.2
## 9       1.4  setosa      2.9       4.4      0.2
## 10      1.5  setosa      3.1       4.9      0.1

```

3.5.3 Other data frame operations

3.5.3.1 Merging datasets

Data sets for a complex project might reside in several tables, related by **key variables**. These are variables that are shared by more than one table, and define which records in each table are related to records in another table. This is exactly what relational databases do, but most biologists have small enough datasets that databases aren't necessary.

Data frames can be combined by manually matching records between tables, using the function `match()`. The example below uses three interrelated tables from a stream dataset I work with. You need the datasets `dat_samples_2021-08-04.csv`, `dat_stream_indices_2021-08-04.csv`, and `loc_meta_vaa_2020-11-18.csv`. Download them and put them in your R home directory.

```
options(stringsAsFactors=FALSE)
```

```

# data file names
name1 <- "dat_samples_2021-08-04.csv"
name2 <- "dat_stream_indices_2021-08-04.csv"
name3 <- "loc_meta_vaa_2020-11-18.csv"

# import
x1 <- read.csv(name1, header=TRUE)
x2 <- read.csv(name2, header=TRUE)
x3 <- read.csv(name3, header=TRUE)

# notice that the tables have only "key" variables
# in common, and tables don't necessarily share any
# variables with every other table
intersect(names(x1), names(x2))

## [1] "uocc"
intersect(names(x1), names(x3))

## [1] "loc"
intersect(names(x2), names(x3))

## character(0)

# x2 contains the response variable, so
# use it to build main dataset "dat"
dat <- x2

# add site name, year, and date from x1
# by matching unique occasion (uocc) from
# dat to metadata table x1
matchx <- match(dat$uocc, x1$uocc)
dat$site <- x1$loc[matchx]
dat$year <- x1$year[matchx]
dat$doy <- x1$doy[matchx]

# dat now has site, which allows it to be
# related to the geographic metadata x3.
# grab latitude and longitude from x3
matchx <- match(dat$site, x3$loc)
dat$lat <- x3$lat[matchx]
dat$long <- x3$long[matchx]

```

Matching many variables by hand can become very tedious and error-prone. An alternative, and usually faster way is the function `merge()`. This is a shortcut for the matching that we did manually above. The downside of using `merge()` is that it can be a little obtuse—`merge()` will attempt to match records exactly,

which may or may not return the results you expect. Using `merge()` can also sometimes result in many duplicate columns or extra rows in the result. Always inspect the results of `merge()` to make sure it is doing what you expect (i.e., that you wrote the code correctly to get what you wanted!).

The commands below duplicate the merging done piecemeal above.

```
# x2 contains the response variable
dat <- x2

# merge by the variables in common:
dat2 <- merge(dat, x1)
dat3 <- merge(dat2, x3)
head(dat3)

##      loc    uocc    ind year doy     lat    long   basin
## 1 RV_01_16345 occ_001 48.40576 2017 54 34.05400 -83.24100 Savannah (01)
## 2 RV_01_240 occ_002 56.78960 2006 25 32.67461 -81.46906 Savannah (01)
## 3 RV_01_241 occ_003 60.06336 2003 33 33.58468 -82.65235 Savannah (01)
## 4 RV_01_242 occ_004 21.30232 2002 292 33.38387 -82.00422 Savannah (01)
## 5 RV_01_243 occ_005 51.50136 2002 12 33.83757 -82.91251 Savannah (01)
## 6 RV_01_244 occ_006 73.91343 2007 16 34.95895 -83.57158 Savannah (01)
##          huc12 state county      ecoregion3 ecoregion4
## 1 30601040402 Georgia Madison      Piedmont      45b
## 2 30601090104 Georgia Screven Southern Coastal Plain 75f
## 3 30601050203 Georgia Warren      Piedmont      45c
## 4 30601060602 Georgia Richmond Southeastern Plains 65c
## 5 30601040504 Georgia Wilkes      Piedmont      45c
## 6 30601020101 Georgia Towns      Blue Ridge     66d
##          eco4 gridcode catchkm2 fid bfi
## 1 Southern Outer Piedmont 2331987 2.3616 6278849 59.00000
## 2 Sea Island Flatwoods 2339456 12.5667 20103925 57.01117
## 3 Carolina Slate Belt 2334311 11.7000 6289897 44.00000
## 4 Sand Hills 2335367 1.8666 22725381 63.94214
## 5 Carolina Slate Belt 2331832 13.6170 6280051 50.75585
## 6 Southern Crystaline Ridges and Mountains 2328238 4.6449 6267588 64.17283
##          elev roaddens storder flow slope huc08 fish.rich
## 1 206.97063 2.4394053 3 -0.24495427 0.00001000 3060104 35
## 2 28.11548 2.1069683 3 -0.23087494 0.00176311 3060109 61
## 3 142.94652 1.6804648 1 -0.81467829 0.00877661 3060105 21
## 4 55.97662 3.4675126 3 -0.03230713 0.00055628 3060106 83
## 5 166.52806 1.5691058 1 -0.72841189 0.00605539 3060104 35
## 6 925.20189 0.3386832 1 -0.52734597 0.03813850 3060102 51
# what if you only want some variables to be merged?
x3.cols <- c("loc", "lat", "long")
dat4 <- merge(x2, x1)
```

```
dat4 <- merge(dat4, x3[,x3.cols])
head(dat4)

##           loc    uocc     ind year doy      lat      long
## 1 RV_01_16345 occ_001 48.40576 2017 54 34.05400 -83.24100
## 2 RV_01_240   occ_002 56.78960 2006 25 32.67461 -81.46906
## 3 RV_01_241   occ_003 60.06336 2003 33 33.58468 -82.65235
## 4 RV_01_242   occ_004 21.30232 2002 292 33.38387 -82.00422
## 5 RV_01_243   occ_005 51.50136 2002 12 33.83757 -82.91251
## 6 RV_01_244   occ_006 73.91343 2007 16 34.95895 -83.57158
```

Some of the tidyverse packages offer alternative solutions for merging datasets that can be more or less elegant than `merge()`. However, the methods shown above use only base R functions and will get the job done.

3.5.4 Reshaping data frames

In your career you are likely to encounter datasets organized in a variety of formats. Most of these formats fall into one of two categories: **wide** and **long**.

Long format			Wide format		
site	attribute	value	site	lat	long
site001	lat	33.7579	site001	33.7579	-82.3836
site002	lat	35.0121	site002	35.0121	-83.2067
site003	lat	34.952	site003	34.952	-83.5166
site004	lat	34.6649	site004	34.6649	-83.3698
site001	long	-82.3836			
site002	long	-83.2067			
site003	long	-83.5166			
site004	long	-83.3698			

In a **long-format dataset**, every value is stored in a separate *row*. The variable (AKA: feature or attribute), that each value represents is kept in a separate *column*. Many public or online data sources serve datasets in long format. This is usually because the data are pulled from a database, and it left to the user to reshape the data to fit their needs. Storing data in this manner can sometimes be more efficient from a computational standpoint, but it is usually not convenient for analysis. The reason that long format is inconvenient for analysis is because all of the information about individual samples (i.e, observations) are on different rows of the dataset.

In a **wide-format dataset**, each row contains **all values** associated with a particular **sample**. This means that each variable gets its own column, and each cell contains a single value. This format is sometimes called **tidy** format by the

authors of the `tidyverse` set of R packages (Wickham 2014), but researchers were formatting data this way long before the `tidyverse` packages or R were developed (Broman and Woo 2018). As a rule, you should store and manage your data in wide/tidy format. This format makes it easy to take advantage of R’s vectorized nature.

Base R has some functions for reshaping data, but most sources recommend using the `reshape2` package or the `tidyverse`. The examples below demonstrate reshaping a dataset from a typical long format to a more useful wide format. You will need the file `reshape_example.csv` in your R home directory and package `reshape2`.

```
options(stringsAsFactors=FALSE)
library(reshape2)
in.name <- "reshape_example.csv"
x <- read.csv(in.name)
head(x)

##          site attribute      value
## 1 RV_01_245      lat  33.757913
## 2 RV_01_245     long -82.383578
## 3 RV_01_245   catchkm2    11.079
## 4 RV_01_245 ecoregion3 Piedmont
## 5 RV_01_248      lat  34.952033
## 6 RV_01_248     long -83.516598
class(x$value)

## [1] "character"
```

Notice that the `value` column contains a mix of numeric and text values. This means that the numbers are stored as text. If we wanted to do something with those values, they need to be pulled out of the vector `value` and converted to numeric values. Because a vector can have only one type of value, the attributes will need to end up in their own columns. The fastest way to do this is to use the function `dcast()` from package `reshape2`.

```
dx <- dcast(x, site~attribute)
dx$catchkm2 <- as.numeric(dx$catchkm2)
dx$lat <- as.numeric(dx$lat)
dx$long <- as.numeric(dx$long)
head(dx)

##          site catchkm2 ecoregion3      lat      long
## 1 RV_01_245  11.0790  Piedmont 33.75791 -82.38358
## 2 RV_01_248   7.5924 Blue Ridge 34.95203 -83.51660
## 3 RV_01_250   0.6165 Blue Ridge 34.66491 -83.36978
## 4 RV_01_253   2.5614 Piedmont 33.84107 -82.95028
## 5 RV_01_255   7.8687 Piedmont 33.75356 -82.54828
```

```
## 6 RV_01_257 1.9629 Piedmont 33.66435 -82.55340
```

If we needed to go the other way, from wide to long format, we use function `melt()`. This is essentially the reverse of `dcast()`.

```
mx <- melt(dx,
            id.vars="site",
            measure.vars=c("lat", "long", "catchkm2", "ecoregion3"))
head(mx)

##      site variable     value
## 1 RV_01_245     lat 33.757913
## 2 RV_01_248     lat 34.952033
## 3 RV_01_250     lat 34.664911
## 4 RV_01_253     lat 33.841068
## 5 RV_01_255     lat 33.753558
## 6 RV_01_257     lat 33.664353

head(mx[order(mx$site, mx$variable),])

##      site   variable     value
## 1  RV_01_245       lat 33.757913
## 11 RV_01_245       long -82.383578
## 21 RV_01_245   catchkm2    11.079
## 31 RV_01_245  ecoregion3  Piedmont
## 2  RV_01_248       lat 34.952033
## 12 RV_01_248       long -83.516598
```

The `tidyverse` equivalents for `melt()` and `dcast()` are `pivot_longer()` and `pivot_wider()`, respectively. The syntax of these `tidyverse` functions are similar to the `reshape2` functions. Note that the `tidyverse` functions will output a `tibble`, which is a form of data frame peculiar to the `tidyverse`. Some non-`tidyverse` functions can use tibbles, and some cannot. Tibbles can be converted to base R data frames with function `data.frame()`.

```
library(tidyr)

##
## Attaching package: 'tidyr'

## The following object is masked from 'package:reshape2':
##
##     smiths

tdx <- pivot_wider(x,
                     names_from="attribute",
                     values_from="value")
head(tdx)

## # A tibble: 6 x 5
```

```
##   site      lat      long      catchkm2 ecoregion3
##   <chr>    <chr>    <chr>    <chr>    <chr>
## 1 RV_01_245 33.757913 -82.383578 11.079   Piedmont
## 2 RV_01_248 34.952033 -83.516598 7.5924   Blue Ridge
## 3 RV_01_250 34.664911 -83.369779 0.6165   Blue Ridge
## 4 RV_01_253 33.841068 -82.95028  2.5614   Piedmont
## 5 RV_01_255 33.753558 -82.548275 7.8687   Piedmont
## 6 RV_01_257 33.664353 -82.553397 1.9629   Piedmont

tmx <- pivot_longer(tdx,
                     cols=c("lat", "long", "catchkm2", "ecoregion3"),
                     names_to="variable")
head(tmx)

## # A tibble: 6 x 3
##   site     variable value
##   <chr>    <chr>    <chr>
## 1 RV_01_245 lat      33.757913
## 2 RV_01_245 long     -82.383578
## 3 RV_01_245 catchkm2 11.079
## 4 RV_01_245 ecoregion3 Piedmont
## 5 RV_01_248 lat      34.952033
## 6 RV_01_248 long     -83.516598

# if needed:
tdx.df <- data.frame(tdx)
```

Chapter 4

Exploratory data analysis

Sometimes you have to “pre-analyze” your data before you can really analyze it. This process of **exploratory data analysis** can be very informative and can reveal unexpected patterns (or problems) in your dataset. This module will demonstrate some common exploratory processes in R. Along the way we will explore how to summarize data, common data distributions and how to fit them to your data, and how to manage your data to avoid common statistical pitfalls.

One of the main questions that biologists have after collecting data is, “*What statistical test do I use?*” Ignoring the fact that they should have thought about that before collecting data, the answer to that question depends on the answers to two other questions:

1. What question are you trying to answer?
2. What kind of data do you have?

Exploratory data analysis is the process of answering the second question. Common objectives in exploratory data analysis include:

- Describing the central tendency and variability of variables
- Identifying potential outliers
- Identifying how variables are distributed
- Testing whether data meet the assumptions of statistical tests
- Checking for common issues such as autocorrelation, collinearity, and missingness.
- Searching for patterns in your data that may or may not be related to your main analysis.

How much exploratory data analysis is needed varies from project to project. At a minimum, you should be checking to see that your data meet the assumptions of your analysis. Failure to verify that your main analysis is justified can lead to huge problems down the line. Zuur et al. (2010) reviewed a host of common

data exploration methods and motivations; although aimed at ecologists, most biologists would benefit from reading their paper.

4.1 Descriptive and summary statistics

4.1.1 Basic summary statistics

One of the first steps in understanding your dataset is to summarize the values in each variable. You can get a quick numerical summary of a vector or a dataset with `summary()`. The examples below use the built-in dataset `iris`.

```
summary(iris)

##   Sepal.Length     Sepal.Width     Petal.Length     Petal.Width
## Min. :4.300      Min. :2.000      Min. :1.000      Min. :0.100
## 1st Qu.:5.100    1st Qu.:2.800    1st Qu.:1.600    1st Qu.:0.300
## Median :5.800    Median :3.000    Median :4.350    Median :1.300
## Mean   :5.843    Mean   :3.057    Mean   :3.758    Mean   :1.199
## 3rd Qu.:6.400    3rd Qu.:3.300    3rd Qu.:5.100    3rd Qu.:1.800
## Max.  :7.900    Max.  :4.400    Max.  :6.900    Max.  :2.500
##
##          Species
## setosa      :50
## versicolor:50
## virginica :50
##
## 
## 
## 
summary(iris$Petal.Length)

##   Min. 1st Qu. Median  Mean 3rd Qu.  Max.
## 1.000 1.600 4.350 3.758 5.100 6.900
```

Function `range()` reports the minimum and maximum values in a vector. Like many of the functions for descriptive statistics, this function does not handle missing (`NA`) values by default. The argument `na.rm=TRUE` must be set if there are missing values.

```
range(iris$Petal.Length)

## [1] 1.0 6.9
# illustration of working with missing values
x <- c(rnorm(20), NA)
range(x)

## [1] NA NA
```

```
range(x, na.rm=TRUE)

## [1] -2.714130 1.077231
```

Central tendency can be estimated as the **mean** or **median** using eponymous functions. The mean is what most people refer to as the “average” in everyday conversation: the sum of values divided by the number of values. The mean is the value that values would be if they were all the same.

The median is the central value. If the values are sorted from smallest to greatest, the median is the value in the middle. I.e., half of the values are less than the median, and half are greater. This is also referred to as the “50th percentile” because 50% of values are \leq the median.

For many datasets, the mean and median will be approximately the same. Data that are skewed will have a median different from their mean. Biologists typically summarize data using the mean, but median can be appropriate in some situations—especially when data are not normally distributed.

Like range, these functions for mean and median do not handle NA values unless you specify `na.rm=TRUE`.

```
a <- rnorm(50)
mean(a)

## [1] 0.07336362

median(a)

## [1] 0.02716257
```

Variance and **standard deviation (SD)** are calculated by `var()` and `sd()`. There is no built-in function for **standard error (SE)** so it must be calculated using the sample size (function `length()`). Note that you should very rarely compute and report SE because SD is more appropriate in almost every situation that is likely to come up.

```
a <- rnorm(50)
var(a)

## [1] 1.148712

sd(a)

## [1] 1.07178

sqrt(var(a)) # equivalent to sd(a)

## [1] 1.07178

# standard error of the mean
sd(a)/sqrt(length(a))
```

```
## [1] 0.1515725
```

Quantiles of data are calculated using the function `quantile()`. A quantile is a value in a distribution or sample such that some specified proportion of observations are less than or equal to that value. For example, the 0.5 quantile is the value such that half of values are less than or equal to that value. Quantiles are usually expressed either as proportions in [0, 1] or as percentiles in [0, 100]. R uses proportions, but you can always multiply by 100 to get percentiles.

Special quantiles are also available as their own functions: minimum `min()`, median `median()`, maximum `max()`, and interquartile range `IQR()`. Remember that argument `na.rm` is needed for many of these functions if there are missing values!

```
a <- rnorm(50)
# 20th percentile (aka: 1st quintile)
quantile(a, probs=0.2)

##      20%
## -0.4469422

# 75th percentile (aka: 3rd quartile)
quantile(a, probs=0.75)

##      75%
## 0.6407535

# 2.5 and 97.5 percentiles
# (i.e. 95% CI of a normal distribution)
quantile(a, probs=c(0.025, 0.975))

##      2.5%    97.5%
## -1.780417  1.493172

# minimum, or 0th percentile
min(a)

## [1] -2.178898

# maximum, or 100th percentile
max(a)

## [1] 1.7061

# interquartile range (75th - 25th percentile)
IQR(a)

## [1] 1.008555

# demonstration of na.rm
b <- c(a, NA)
min(b)
```

```
## [1] NA
min(b,na.rm=TRUE)
## [1] -2.178898
```

4.1.2 Summarizing data with the `apply()` family

The `apply()` family is a group of functions that operate over the **dimensions** of an object. You can use the `apply()` family to summarize across rows or columns, or across elements of a list, and so on. The function used most often is `apply()`, which works on data frames, matrices and arrays. There are also `lapply()`, which works on lists and returns a list; `sapply()`, which operates on lists and returns the simplest object possible; and many others.

4.1.2.1 `apply()` for arrays and data frames

Function `apply()` works on arrays and data frames with ≥ 2 dimensions. Its arguments are the object to be operated on, the margin (or dimension) on which to operate, and the operation (or function). The first margin is rows, the second is columns, and so on. The result of `apply()` will always have one fewer dimension than the input.

Some typical `apply()` commands are demonstrated below:

```
# test dataset with only numeric values
x <- iris[,1:4]

# margin 1 = rows --> row sums
apply(x, 1, sum)

## [1] 10.2 9.5 9.4 9.4 10.2 11.4 9.7 10.1 8.9 9.6 10.8 10.0 9.3 8.5 11.2
## [16] 12.0 11.0 10.3 11.5 10.7 10.7 10.7 9.4 10.6 10.3 9.8 10.4 10.4 10.2 9.7
## [31] 9.7 10.7 10.9 11.3 9.7 9.6 10.5 10.0 8.9 10.2 10.1 8.4 9.1 10.7 11.2
## [46] 9.5 10.7 9.4 10.7 9.9 16.3 15.6 16.4 13.1 15.4 14.3 15.9 11.6 15.4 13.2
## [61] 11.5 14.6 13.2 15.1 13.4 15.6 14.6 13.6 14.4 13.1 15.7 14.2 15.2 14.8 14.9
## [76] 15.4 15.8 16.4 14.9 12.8 12.8 12.6 13.6 15.4 14.4 15.5 16.0 14.3 14.0 13.3
## [91] 13.7 15.1 13.6 11.6 13.8 14.1 14.1 14.7 11.7 13.9 18.1 15.5 18.1 16.6 17.5
## [106] 19.3 13.6 18.3 16.8 19.4 16.8 16.3 17.4 15.2 16.1 17.2 16.8 20.4 19.5 14.7
## [121] 18.1 15.3 19.2 15.7 17.8 18.2 15.6 15.8 16.9 17.6 18.2 20.1 17.0 15.7 15.7
## [136] 19.1 17.7 16.8 15.6 17.5 17.8 17.4 15.5 18.2 18.2 17.2 15.7 16.7 17.3 15.8

# margin 2 = columns --> column sums
apply(x, 2, sum) # 2 = columns

## Sepal.Length Sepal.Width Petal.Length Petal.Width
##          876.5        458.6       563.7       179.9
```

The first `apply()` command above calculates the sum of values in each row (margin 1) of its input `x`. The second `apply()` command calculates the sum of

values in each column (margin 2). The results are vectors whose lengths are the same as the corresponding dimension of the input (number of rows or columns).

The first command above is equivalent to:

```
y <- numeric(nrow(x))
for(i in 1:nrow(x)){y[i] <- sum(x[i,])}

# verify that results are the same:
all(apply(x, 1, sum) == y)

## [1] TRUE
```

Below are some examples of calculations you can do with the `apply()` function. Notice that when the function to be applied takes its own arguments, those arguments are supplied following the function name, in the same order as they would be supplied to the function if used outside of `apply()`.

```
# column minima
apply(x, 2, min)

## Sepal.Length Sepal.Width Petal.Length Petal.Width
##       4.3        2.0       1.0        0.1

# 75th percentile of each column
apply(x, 2, quantile, 0.75)

## Sepal.Length Sepal.Width Petal.Length Petal.Width
##       6.4        3.3       5.1        1.8

# interquartile range of each row
apply(x, 1, IQR)

## [1] 2.800 2.375 2.550 2.300 2.850 2.900 2.575 2.625 2.175 2.400 2.950 2.500
## [13] 2.375 2.475 3.500 3.500 3.200 2.775 2.925 2.925 2.575 2.825 3.050 2.350
## [25] 2.275 2.250 2.500 2.750 2.750 2.325 2.275 2.675 3.225 3.425 2.375 2.700
## [37] 2.975 2.850 2.325 2.650 2.825 1.800 2.475 2.525 2.600 2.325 2.875 2.450
## [49] 2.925 2.625 2.525 2.200 2.700 2.325 2.600 2.375 2.225 1.650 2.600 1.850
## [61] 2.125 2.000 2.600 2.525 1.600 2.300 2.150 2.250 2.900 2.175 2.225 2.100
## [73] 3.000 2.650 2.325 2.350 2.850 2.750 2.325 1.850 2.150 2.100 2.050 2.900
## [85] 2.100 1.925 2.500 2.825 1.900 2.175 2.425 2.375 2.200 1.750 2.200 2.025
## [97] 2.075 2.275 1.375 2.075 2.975 2.775 3.425 3.150 3.175 4.075 2.300 3.925
## [109] 3.700 3.050 2.550 3.075 3.050 2.800 2.575 2.600 3.050 3.550 4.575 3.225
## [121] 3.025 2.475 4.350 2.775 2.950 3.450 2.600 2.500 3.175 3.500 3.850 3.425
## [133] 3.150 2.925 3.425 3.675 2.625 2.950 2.400 2.925 2.950 2.650 2.775 3.150
## [145] 2.850 2.750 2.975 2.775 2.475 2.600
```

Arrays with >2 dimensions can have functions applied across multiple dimensions:

```
# object with 3 rows, 2 columns, and 3 "layers":
x <- array(1:18, dim=c(3,2,3))
```

```
# how many dimensions does the result have?
apply(x, c(1,2), sum)

##      [,1] [,2]
## [1,]    21   30
## [2,]    24   33
## [3,]    27   36
```

You can also write custom functions and apply them to rows or columns. Below are some useful functions to apply. These functions can be defined inside `apply()` or outside `apply()`. Each of the functions below takes an input called `x`, but this is not as the `x` that is the input to `apply()`. The `x` in `function(x)` is used inside the braces `{}` as the name for the input passed from `apply()` to the inner function.

```
x <- iris[,1:4]

# count values > 2
apply(x, 2, function(x){length(which(x > 2))})

## Sepal.Length Sepal.Width Petal.Length Petal.Width
##          150           149           100            23

# count missing (NA) values
apply(x, 2, function(x){length(which(is.na(x)))})

## Sepal.Length Sepal.Width Petal.Length Petal.Width
##          0           0           0            0

# count non-missing values
count.nonmissing <- function(x){length(which(!is.na(x)))}
apply(x, 2, count.nonmissing)

## Sepal.Length Sepal.Width Petal.Length Petal.Width
##          150           150           150            150
```

4.1.2.2 `lapply()` and `sapply()` for lists and other objects

The other members of the `apply()` family that are most often used are `lapply()` and `sapply()`.

- `lapply()`, operates on lists and returns a *list*. Short for “list apply”, and usually pronounced “el apply” or “lap-lie”.
- `sapply()` operates on many object types (including lists) and returns the simplest object possible. Short for “simplify apply”, and usually pronounced “ess apply” or “sap-lie”. Usually the goal with `sapply()` is to get a vector or matrix out of a list.

The code block below makes a list called `mylist` and illustrates the use of `lapply()` and `sapply()`.

```
mylist <- vector("list", 5)
for(i in 1:length(mylist)){
  x <- rnorm(20)
  mylist[[i]] <- t.test(x)
}#i

# get estimated mean of each element of mylist, in a list
lapply(mylist, function(x){x$estimate})

## [[1]]
##   mean of x
## -0.2686405
##
## [[2]]
##   mean of x
## -0.203565
##
## [[3]]
##   mean of x
## 0.09948298
##
## [[4]]
##   mean of x
## -0.04637868
##
## [[5]]
##   mean of x
## 0.1247556

# get estimated mean of each element, simplified to vector
sapply(mylist, function(x){x$estimate})

##   mean of x   mean of x   mean of x   mean of x   mean of x
## -0.26864053 -0.20356498  0.09948298 -0.04637868  0.12475562
```

The function `lapply()` can sometimes be combined with `do.call()` to produce similar outputs as `sapply()`:

```
do.call(c, lapply(mylist, function(x){x$estimate}))
```

```
##   mean of x   mean of x   mean of x   mean of x   mean of x
## -0.26864053 -0.20356498  0.09948298 -0.04637868  0.12475562
```

4.1.3 Tabulation and aggregation

4.1.3.1 Tabulation of a single variable

The `table()` function tallies the occurrences of each unique value in a vector. Notice that `table()` only returns a result for the values that actually occur in the input.

```
aa <- rpois(50,10) # Random Poisson distribution
table(aa)

## aa
## 4 6 7 8 9 10 11 12 13 14 15 16 17 18 19
## 1 1 5 4 5 5 8 7 6 1 3 1 1 1 1 1
```

If you need a count that includes 0s for missing values, there are a few ways to do this. My preferred solution is to use the custom function below. This function counts how many times each value in `values` occurs in `input`.

```
table.all <- function(input, values){
  res <- sapply(values,
                function(x){length(which(input == x))})
  names(res) <- as.character(values)
  return(res)
}

# use with random poisson values:
x1 <- rpois(20, 10)
table.all(x1, 0:30)

## 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
## 0 0 0 0 0 1 4 1 1 3 0 2 1 2 2 2 1 0 0 0 0 0 0 0 0 0 0 0 0
## 26 27 28 29 30
## 0 0 0 0 0

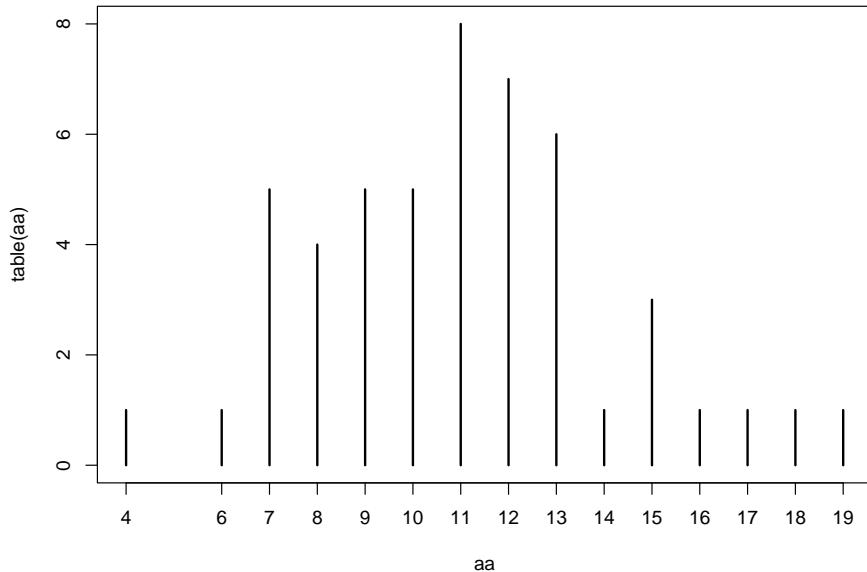
# compare to:
table(x1)

## x1
## 5 6 7 8 9 11 12 13 14 15 16
## 1 4 1 1 3 2 1 2 2 2 1
```

If you plot a `table()` result, you get a sort of bar graph. This is similar to what you get with a command like `plot(..., type = "h")`¹.

```
plot(table(aa))
```

¹R users often use an ellipsis `...` as a shorthand for unshown arguments to the function, not literally three periods. In this text, the ellipsis is a stand in for other arguments to the function that are irrelevant to the point being made. There are some circumstances where an ellipsis is necessary to control how variables are passed from one function to another



4.1.3.2 Cross-tabulation (aka: frequency tables)

Contingency tables, or **frequency tables** are calculated with `ftable()`. This function counts the number of values in each combination of different factors. The example below uses the built-in dataset `iris` modified to have some additional factors.

```
set.seed(123)
x <- iris
x$color <- c("purple", "white")
x$bloom <- sample(c("early", "late"), nrow(x), replace=TRUE)
handed.table <- ftable(x$color~x$bloom)
handed.table

##          x$color purple white
## x$bloom
## early           39     37
## late            36     38
```

Contingency tables produced by `ftable()` can be used directly for chi-squared (χ^2) tests:

```
chisq.test(handed.table)
```

```
##
```

```
## Pearson's Chi-squared test with Yates' continuity correction
##
## data: handed.table
## X-squared = 0.026671, df = 1, p-value = 0.8703
```

The function `ftable()` also has a formula interface, much like many statistical functions (e.g., `lm()`). This is useful if you want more complicated tables or more combinations of categories. Notice that the order of variables affects the structure of the resulting table. The 6 commands below produce equivalent tables, but the results are organized differently.

```
ftable(x$Species~x$color+x$bloom)

##          x$Species setosa versicolor virginica
## x$color x$bloom
## purple   early      17      13      9
##        late       8      12     16
## white   early      13      14     10
##        late      12      11     15

ftable(x$Species~x$bloom+x$color)

##          x$Species setosa versicolor virginica
## x$bloom x$color
## early    purple      17      13      9
##        white      13      14     10
## late    purple       8      12     16
##        white      12      11     15

ftable(x$color~x$Species+x$bloom)

##          x$color purple white
## x$Species x$bloom
## setosa    early      17      13
##        late       8      12
## versicolor early      13      14
##        late      12      11
## virginica early      9      10
##        late      16      15

ftable(x$color~x$bloom+x$Species)

##          x$color purple white
## x$bloom x$Species
## early    setosa      17      13
##        versicolor    13      14
##        virginica      9      10
## late    setosa       8      12
##        versicolor    12      11
```

```

##          virginica      16      15
ftable(x$bloom~x$color+x$Species)

##          x$bloom early late
## x$color x$Species
## purple setosa      17      8
##        versicolor    13     12
##        virginica     9     16
## white  setosa      13     12
##        versicolor    14     11
##        virginica     10     15
ftable(x$bloom~x$Species+x$color)

##          x$bloom early late
## x$Species x$color
## setosa   purple      17      8
##        white       13     12
## versicolor purple     13     12
##        white       14     11
## virginica purple     9     16
##        white       10     15

```

Variables on the left side of the ~ will be on the top of the table, and variables on the right side of the ~ will be on the left side of the table². On each side, variables are ordered in reverse order from the table matrix outwards. Compare the results of these commands:

```

ftable(x$Species~x$color+x$bloom)

##          x$Species setosa versicolor virginica
## x$color x$bloom
## purple  early      17      13      9
##        late       8      12     16
## white   early      13      14     10
##        late       12      11     15
ftable(x$Species~x$bloom+x$color)

##          x$Species setosa versicolor virginica
## x$bloom x$color
## early   purple      17      13      9
##        white      13      14     10
## late    purple      8      12     16
##        white      12      11     15

```

²I can never remember how this works, so usually have to tinker with the code until I get the table I want :)

The output of `ftable()` is of class `ftable`, which *looks* like a matrix or data frame but does not function like one. Sometimes it is necessary to convert an `ftable()` output to different class in order to pull values from it. This can be done with base R conversion functions. Notice that the data frame form of `handed.table` resembles the “long” data format we discussed in module 3.

```

handed.table <- ftable(x$Species~x$color)
as.matrix(handed.table)

##           x$Species
## x$color   setosa versicolor virginica
##  purple     25      25      25
##  white      25      25      25

as.data.frame(handed.table)

##   x.color x.Species Freq
## 1 purple   setosa    25
## 2 white    setosa    25
## 3 purple   versicolor 25
## 4 white    versicolor 25
## 5 purple   virginica  25
## 6 white    virginica  25

```

4.1.4 Aggregation (aka: pivot tables)

One of the most common data operations is **aggregation**, or **summarization by groups**. In Excel, this is done using **Pivot Tables**. In R, the function `aggregate()` is used for pivot table-like functionality. But, as we shall see, the R `aggregate()` function is much more powerful than an Excel pivot table. The `tidyverse` equivalent is `dplyr::summarise()`.

	A	B	C	D	E	F	G	H	I	J
1	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species					
2	5.1	3.5	1.4	0.2	setosa					
3	4.9	3	1.4	0.2	setosa					
4	4.7	3.2	1.3	0.2	setosa					
5	4.6	3.1	1.5	0.2	setosa					
6	5	3.6	1.4	0.2	setosa					
7	5.4	3.9	1.7	0.4	setosa					
8	4.6	3.4	1.4	0.3	setosa					
9	5	3.4	1.5	0.2	setosa					
10	4.4	2.9	1.4	0.2	setosa					
11	4.9	3.1	1.5	0.1	setosa					
12	5.4	3.7	1.5	0.2	setosa					
13	4.8	3.4	1.6	0.2	setosa					
14	4.8	3	1.4	0.1	setosa					
15	4.3	3	1.1	0.1	setosa					
16	5.8	4	1.2	0.2	setosa					
17	5.7	4.4	1.5	0.4	setosa					
						Row Labels	Average of Sepal.Length	StdDev of Sepal.Length		
						setosa	5.006	0.352489687		
						versicolor	5.936	0.516171147		
						virginica	6.588	0.635879593		
						Grand Total	5.843333333	0.828046128		

The most important inputs to `aggregate()` are the data to be summarized, the variable or variables that define the groups, and the function to summarize by. The examples below shows how to calculate the mean petal length for each species in the `iris` dataset. If the variable to be summarized and the grouping variables are in the same data frame (which is usually the case), you can use the formula interface to `aggregate()`; otherwise, use the `by=` interface.

```
# example using by=
aggregate(iris$Petal.Length,
         by=list(iris$Species),
         mean)

##      Group.1      x
## 1     setosa 1.462
## 2 versicolor 4.260
## 3 virginica 5.552

# better method using formula
aggregate(Petal.Length~Species,
          data=iris,
          mean)

##      Species Petal.Length
## 1     setosa      1.462
## 2 versicolor      4.260
## 3 virginica      5.552
```

The formula interface is cleaner and easier than the `by=` interface (first example), and has the key advantage that it automatically passes variable names to the result. If you use `by=` you may want to rename the columns in the result.

In my own code I prefer to name my aggregation tables `agg`, with additional parts like `agg1`, `agg.fish`, and so on if needed. Function `aggregate()` produces a data frame with a column for each grouping variable (`Group.1`, `Group.2` and so on) and a single column `x` containing the summarized values. We can use this property to construct more complicated tables:

```
x <- iris
# add another grouping variable
x$color <- c("red", "white")

# by= example: notice how columns in result need to be renamed
agg <- aggregate(x$Petal.Length,
                  by=list(x$Species, x$color),
                  mean)
agg

##      Group.1 Group.2      x
## 1     setosa    red 1.456
```

```

## 2 versicolor    red 4.308
## 3 virginica     red 5.564
## 4      setosa    white 1.468
## 5 versicolor    white 4.212
## 6 virginica     white 5.540

# change column names
names(agg) <- c("spp", "color", "mean")
agg

##           spp color  mean
## 1      setosa  red 1.456
## 2 versicolor  red 4.308
## 3 virginica  red 5.564
## 4      setosa white 1.468
## 5 versicolor white 4.212
## 6 virginica white 5.540

# formula example: much easier
agg <- aggregate(Petal.Length~Species+color, data=x, mean)
agg

##      Species color Petal.Length
## 1      setosa  red       1.456
## 2 versicolor  red       4.308
## 3 virginica  red       5.564
## 4      setosa white     1.468
## 5 versicolor white     4.212
## 6 virginica white     5.540

```

Notice that in `agg` the values are sorted by the grouping variables from right to left: in this example, by `color`, then `Species`. This is the reverse of the order in the original command. The result of `aggregate()` is a data frame, so data within it can be rearranged using `order()`.

Often we want to summarize a variable by multiple functions—for example, to get a table with the mean and SD in each group. The way to do this is to use several `aggregate()` commands, and then combine the results. The examples below shows two ways to construct a table with the mean and SD of a variable.

```

# define a formula and save some typing:
f1 <- formula(Petal.Length~Species)

# method 1: single table
agg <- aggregate(f1, data=x, mean)
agg$sd <- aggregate(f1, data=x, sd)$Petal.Length
names(agg)[which(names(agg) == "Petal.Length")] <- "mn"

agg

```

```

##      Species     mn      sd
## 1    setosa 1.462 0.1736640
## 2 versicolor 4.260 0.4699110
## 3 virginica 5.552 0.5518947

```

The `$Petal.Length` on the end of `aggregate()` is important because we want only column `Petal.Length` of the output. Without `$Petal.Length`, the result will be messy when we try to add it to `agg`. If you ever want to do this with the `by=` method, the column of the output containing the summarized values will be named `x`.

```

# method 1: single table, with by=
agg <- aggregate(iris$Petal.Length, by=list(iris$Species), mean)
## note use of $x:
agg$sd <- aggregate(iris$Petal.Length, by=list(iris$Species), sd)$x

# change names in result:
names(agg)[which(names(agg) == "x")] <- "mn"
names(agg)[which(names(agg) == "Group.1")] <- "species"
agg

##      species     mn      sd
## 1    setosa 1.462 0.1736640
## 2 versicolor 4.260 0.4699110
## 3 virginica 5.552 0.5518947

```

The second approach is to make several `aggregate()` tables, and combine them into a single result.

```

# method 2: multiple tables
# (note: there are lots of ways to do this)
agg.mn <- aggregate(f1, data=x, mean)
agg.sd <- aggregate(f1, data=x, sd)
agg <- data.frame(agg.mn, sd=agg.sd$Petal.Length)
names(agg)[2] <- "mn"
agg

##      Species     mn      sd
## 1    setosa 1.462 0.1736640
## 2 versicolor 4.260 0.4699110
## 3 virginica 5.552 0.5518947

```

If you've ever used Excel Pivot Tables, you will have noticed that data can only be summarized by a limited set of functions. On the other hand, `aggregate()` can be used with pretty much any function that takes in a vector and returns a scalar. You can even write your own functions! Below are some examples. Notice in the first example that arguments to the summarizing function follow

its name.

```
# first quartile (25th percentile)
aggregate(Petal.Length~Species, data=x, quantile, 0.25)

##      Species Petal.Length
## 1      setosa      1.4
## 2 versicolor      4.0
## 3 virginica      5.1

# number of values
aggregate(Petal.Length~Species, data=x, length)

##      Species Petal.Length
## 1      setosa      50
## 2 versicolor      50
## 3 virginica      50
```

The following examples illustrate **anonymous functions**: functions defined within another function and never defined in the R environment. The `x` in `function(x)` is taken to be the values within each group defined by the variable(s) on the right-hand side of `~` or in `by=`. Notice that the end of such a command will need a lot of `)`, `}`, and sometimes `]`. Using an editor with syntax highlighting is very helpful!

```
# number of missing values
aggregate(Petal.Length~Species,
         data=x,
         function(x){length(which(is.na(x)))})

##      Species Petal.Length
## 1      setosa      0
## 2 versicolor      0
## 3 virginica      0

# number of non-missing values
aggregate(Petal.Length~Species,
         data=x,
         function(x){length(which(!is.na(x)))})

##      Species Petal.Length
## 1      setosa      50
## 2 versicolor      50
## 3 virginica      50

# number of values >= 2
aggregate(Petal.Length~Species,
         data=x,
         function(x){length(which(x >= 2))})
```

```

##      Species Petal.Length
## 1      setosa         0
## 2 versicolor        50
## 3 virginica         50
# number of UNIQUE values >= 2
aggregate(Petal.Length~Species,
           data=x,
           function(x){length(unique(x[which(x >= 2)]))})

```



```

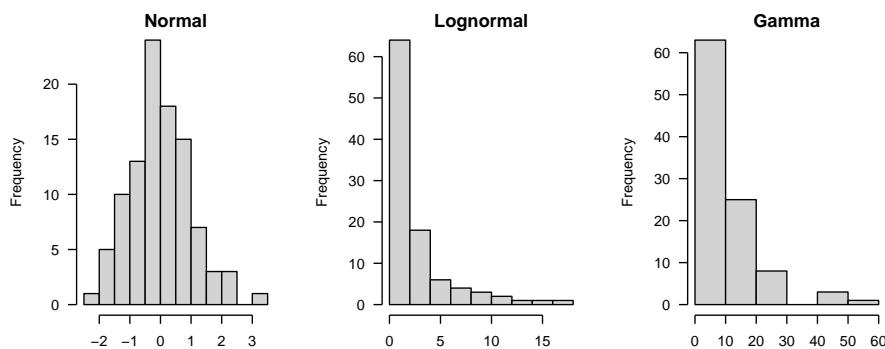
##      Species Petal.Length
## 1      setosa         0
## 2 versicolor        19
## 3 virginica         20

```

Being able to use almost any function, and to define your own functions, makes `aggregate()` an immensely powerful tool to have in your R toolbox.

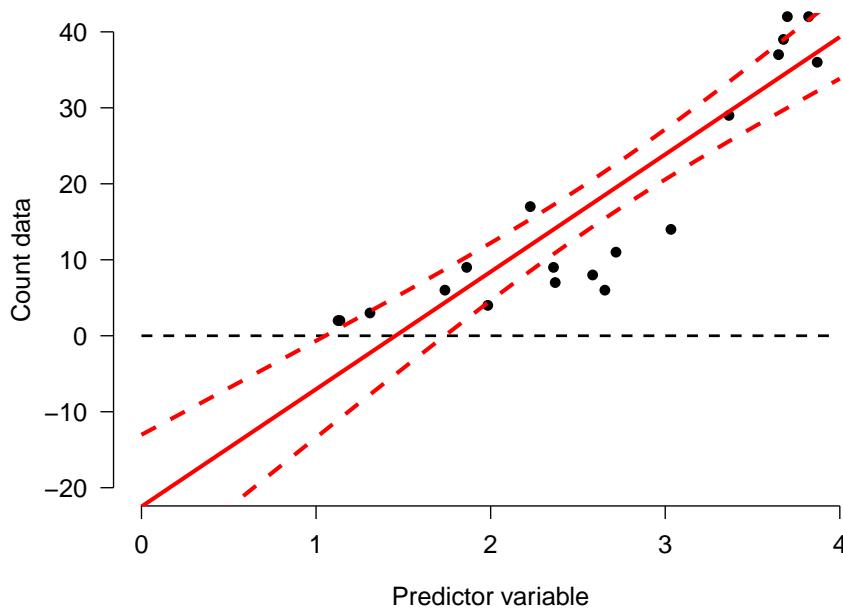
4.2 Visualizing data distributions

Probability distributions are mathematical functions that describe the way that values vary randomly. This is a key idea in statistics and data analysis. We usually consider data to be essentially random, but with the values forming predictable patterns over many observations. The nature of those patterns is what probability distributions attempt to model. A probability distribution does not tell us the value of any particular observation, but it does let us estimate the likelihood of observing any particular value. The figure below demonstrates some common probability distributions: the heights of the bars reflect the likelihood of the values on the x -axes occurring under those distributions.



Successful data analysis requires paying attention to and thinking deeply about the way that your data are distributed. Different kinds of biological processes, like counts, waiting times, measurements, etc., have randomness that is described

by different probability distributions. The assumption that values follow certain distributions is baked into most statistical methods. Trying to use a method that assumes an inappropriate distribution will probably lead to invalid results. For example, if you try to analyze count data as if they came from a continuous, unbounded distribution, your statistical model could predict nonsensical outcomes such as negative or non-integer counts. The figure below shows this:

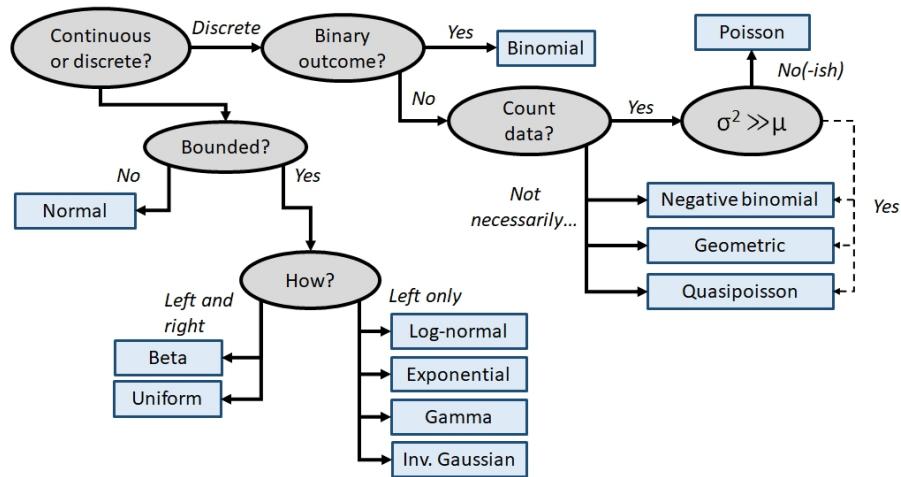


Sometimes it is not always obvious what distribution your data should follow. Some cases are relatively straight forward: counts should follow a Poisson or negative binomial distribution. Other cases are less straightforward: should expression data be considered normally, log-normally, or some other-ly distributed? Usually there is a clear *a priori* answer to this question that can be obtained by thinking about what kind of process gave rise to the numbers.

A related question to “what distribution does my data follow?” is “do my data follow this distribution?” Even if you have some expectation of how your data should be distributed, that is no guarantee that your study system cooperated. You should always check to see if your data follow the distribution you assumed, and the distribution assumed by your statistical test. Deciding what distribution your data follow is always somewhat subjective, and real datasets often contain some departure from expectations. In this section we will explore some graphical techniques and heuristics for determining how data are distributed.

Knowing about data distributions is a one thing, but figuring out which distribution best fits your data is quite another. Like much of the data analysis

workflow, picking a distribution can be as much an art as it is a science. This is because distributions of real data are often messy, or do not conform exactly to any one distribution, or may conform partly to several! Sometimes the answer to “what distribution should I use to analyze this data?” is not clear-cut. The figure below gives a (very) rough guide to starting to identify a response distribution. Note that this is only a guide for where to look first, and not a definitive guide to selecting a response distribution. Even if your choices lead you to, say, the normal distribution, you must still verify that your data conform (at least roughly) to that distribution. Note also that this diagram does not include every distribution out there, but rather a select set of commonly encountered distributions in biology.

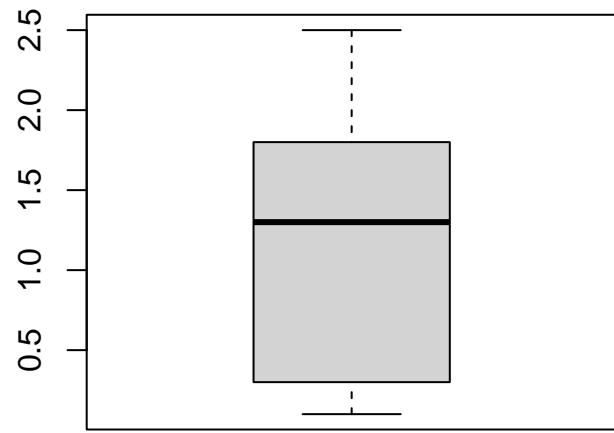


4.2.1 Boxplots (aka: box-and-whisker plots)

Boxplots, or **box-and-whisker plots**, summarize the distribution of continuous variables. They are often used to summarize data by a factor, making it easier to see how the distribution of a variable differs between groups. The boxplot shows extreme values (whiskers), the first and third quartiles, and the median.

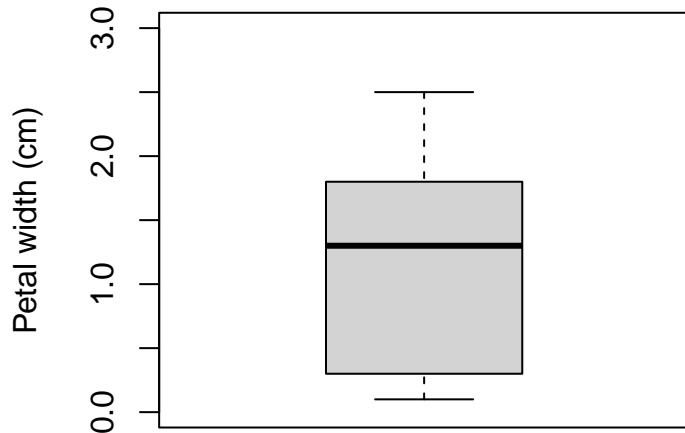
Boxplot of a single variable:

```
boxplot(iris$Petal.Width)
```



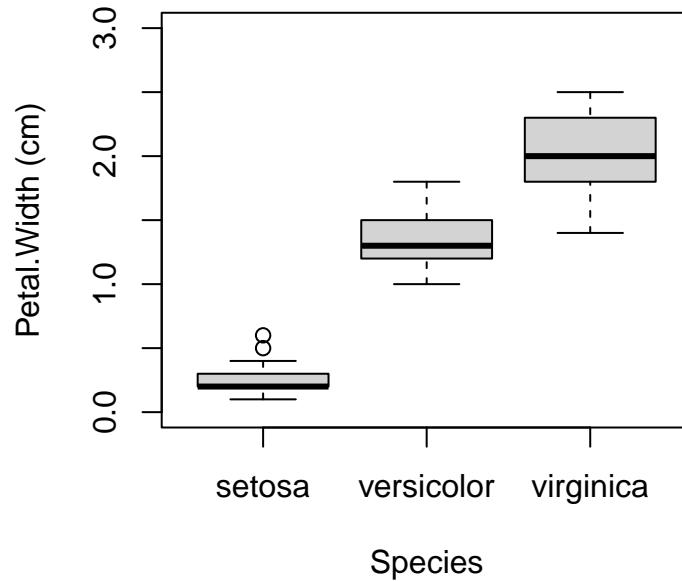
Boxplot with some options for a nicer plot:

```
boxplot(iris$Petal.Width,  
       ylab="Petal width (cm)",  
       ylim=c(0,3))
```



Boxplot across levels of a factor (note formula interface):

```
boxplot(iris$Petal.Width~iris$Species,  
       xlab="Species",  
       ylab="Petal.Width (cm)",  
       ylim=c(0,3))
```

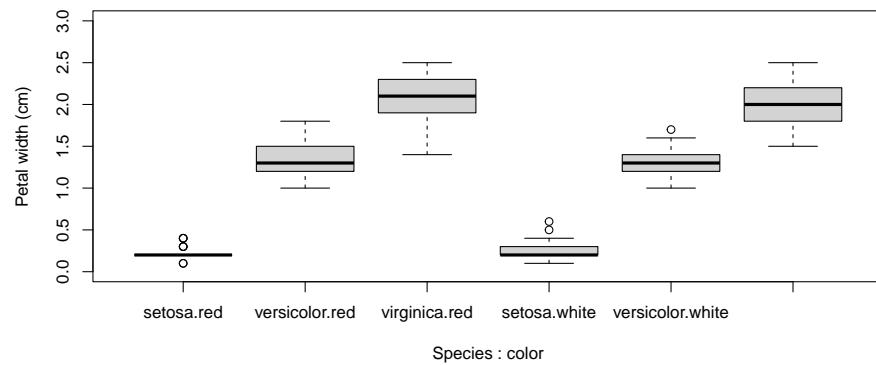


A highly asymmetric box-and-whisker plot can indicate that data are skewed or non-normal. The base `boxplot()` function can produce summaries across multiple variables. Essentially it treats each combination of grouping variables as a separate set of values. There are better ways to plot group differences (e.g., for publication), but the base `boxplot()` does just fine for data exploration.

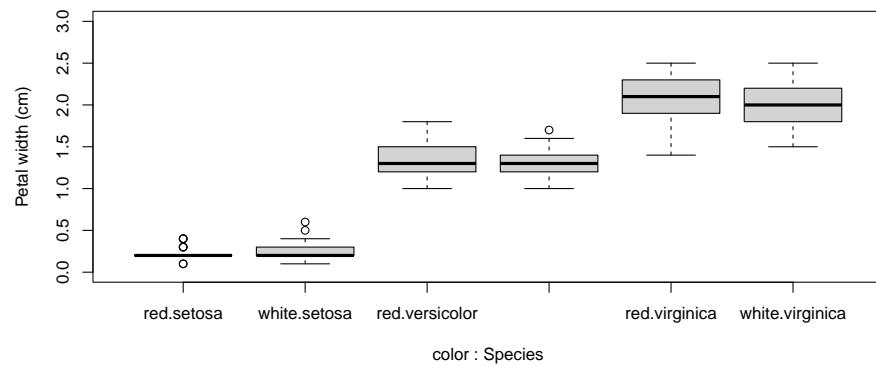
Boxplots can be constructed using several grouping variables. The order of grouping variables in the formula will determine the order of boxes in the plot.

```
x <- iris
x$color <- c("red", "white") #made up variable

# group by species and color:
boxplot(Petal.Width~Species+color, data=x,
        ylab="Petal width (cm)", ylim=c(0,3))
```



```
# notice difference in order of grouping:  
boxplot(Petal.Width~color+Species, data=x,  
        ylab="Petal width (cm)", ylim=c(0,3))
```

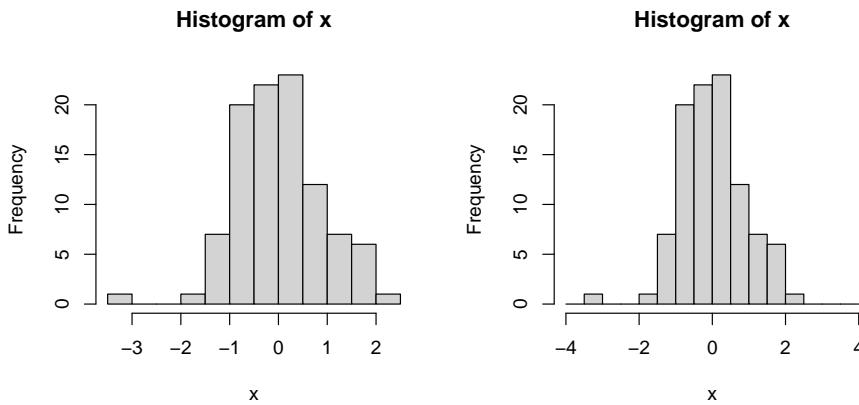


4.2.2 Histograms

Histograms show how values of a distribution are spread across different intervals. These intervals are sometimes called **cells** or **bins**. A good histogram will show, approximately, the shape of a **probability density function (PDF)** or the **probability mass function (PMF)**. The base function `hist()` will automatically select intervals that look nice, but you can specify the intervals with argument `breaks`.

```
x <- rnorm(100)
```

```
par(mfrow=c(1,2))
hist(x)
hist(x, breaks=seq(-4, 4, by=0.5))
```



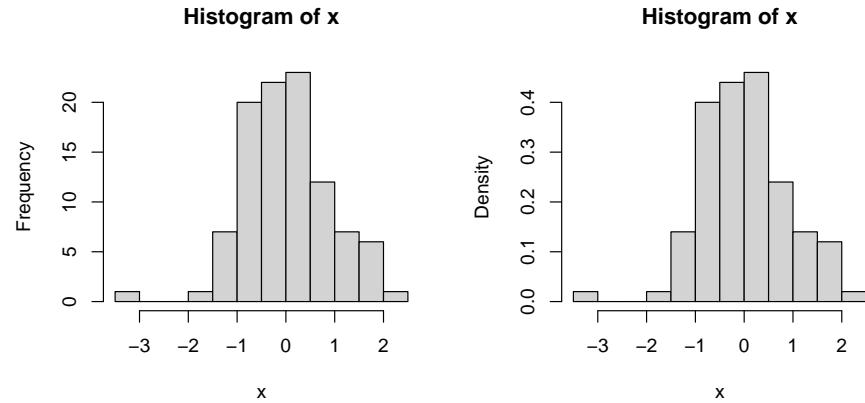
```
# reset
par(mfrow=c(1,1))
```

The *area* of each bar is also proportional to the number of values in each interval.

R histograms can be presented as counts (`freq=TRUE`, the default), or as probability density (`freq=FALSE`). We'll talk more about probability density later in the next section.

Compare these results:

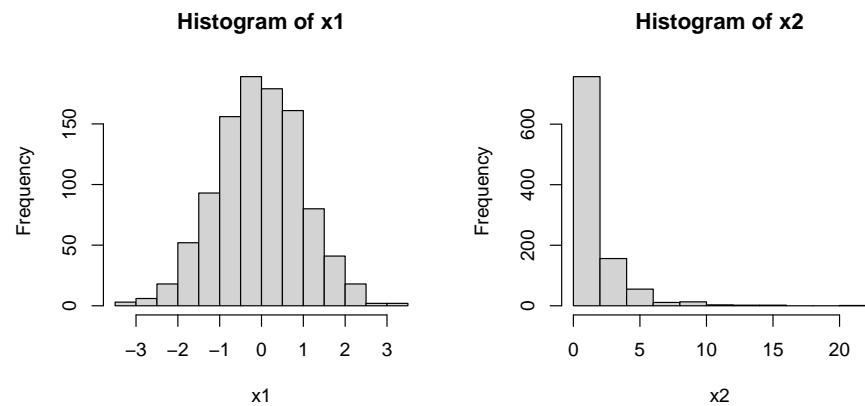
```
par(mfrow=c(1,2))
hist(x)
hist(x, freq=FALSE)
```



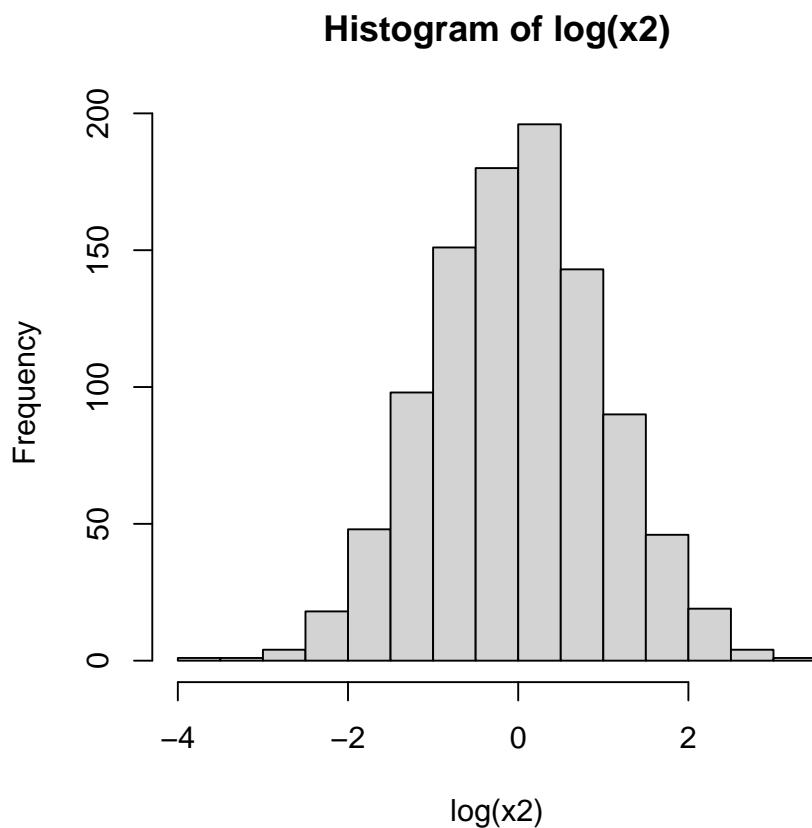
```
# reset
par(mfrow=c(1,1))
```

Because the second histogram shows probability densities, the areas under the bars sum to 1 (this is part of the definition of a PDF).

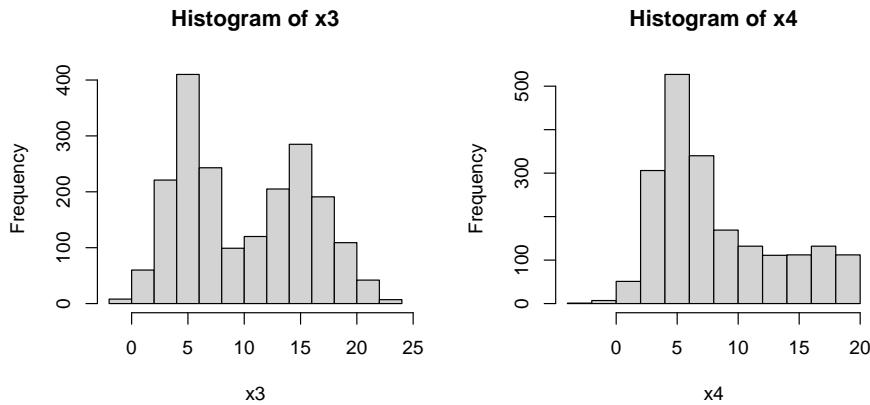
A histogram can provide a visual first clue as to the shape of a distribution's PDF or PMF. For example, the histograms of `x1` and `x2` below suggest very different distributions:



Notice that `x1` appears roughly normal. It is symmetric, has a bell-shape, and is concentrated near its center. Contrast this with `x2`. Distribution `x2` is strongly right-skewed (i.e., lots of small values with a long positive tail), with most values near 0. It also has no negative values (`range(x2)`). Because of these properties we might suspect that `x2` is really a log-normal distribution. We can check this by making a histogram of the log of `x2`:



Other distributions might be tricky. Consider the histograms below:



What a mess! Histogram x_3 shows what is commonly referred to as a **bimodal** distribution. This is a distribution with two **modes**, or most common values, with a gap between them. Bimodal distributions often arise from a **mixture** of two distributions (in this case, two normals). Or, they can indicate that something in the data generating process leads to diverging outcomes. For example, in many college courses the grade distribution can be bimodal, with most students making either a B or D.

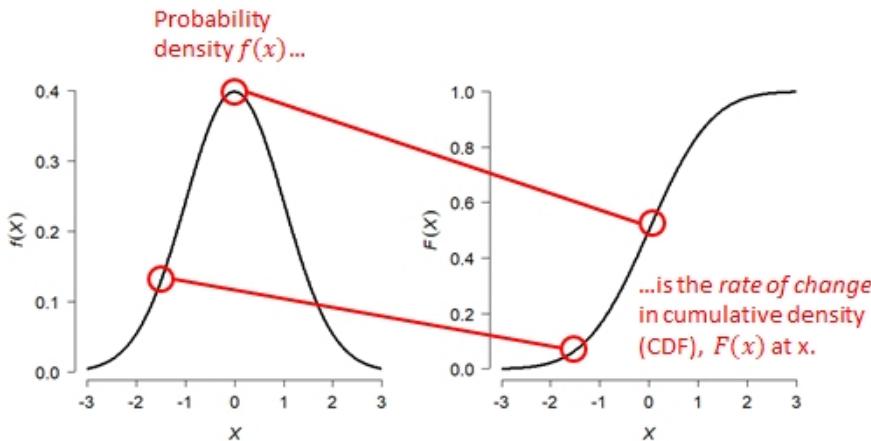
Histogram x_4 shows a distribution with some right skew, but not the long, tapering tail characteristic of the lognormal distribution (see x_2 , above). The fact that the mean is near 5 and not 0 is another point against a lognormal. But the clincher is that there are negative values, which a lognormal distribution cannot take (think about why this is). Think for a few minutes about what distribution x_4 might be, then click the footnote to find out³.

4.2.3 Kernel density plots

Kernel density plots are a way to empirically estimate the **probability distribution function (PDF)** of a distribution. A plot of the estimated PDF is essentially a smoothed histogram (technically, it is the heights of a histogram as the bin width approaches 0).

The PDF tells us two things. Practically speaking, the PDF of a distribution at a given value is related to how likely that value is to occur relative to other values. However, that is NOT a probability of that value occurring. What the PDF really expresses is the *rate of change in cumulative density at a value*. In other words, the PDF is the slope or first derivative of the CDF. Conversely, the CDF is the integral of the PDF. The PDF is the rate at which the probability of observing a value $\leq x$ increases at each x . Again, this is related to but not the same as the probability of that value occurring.

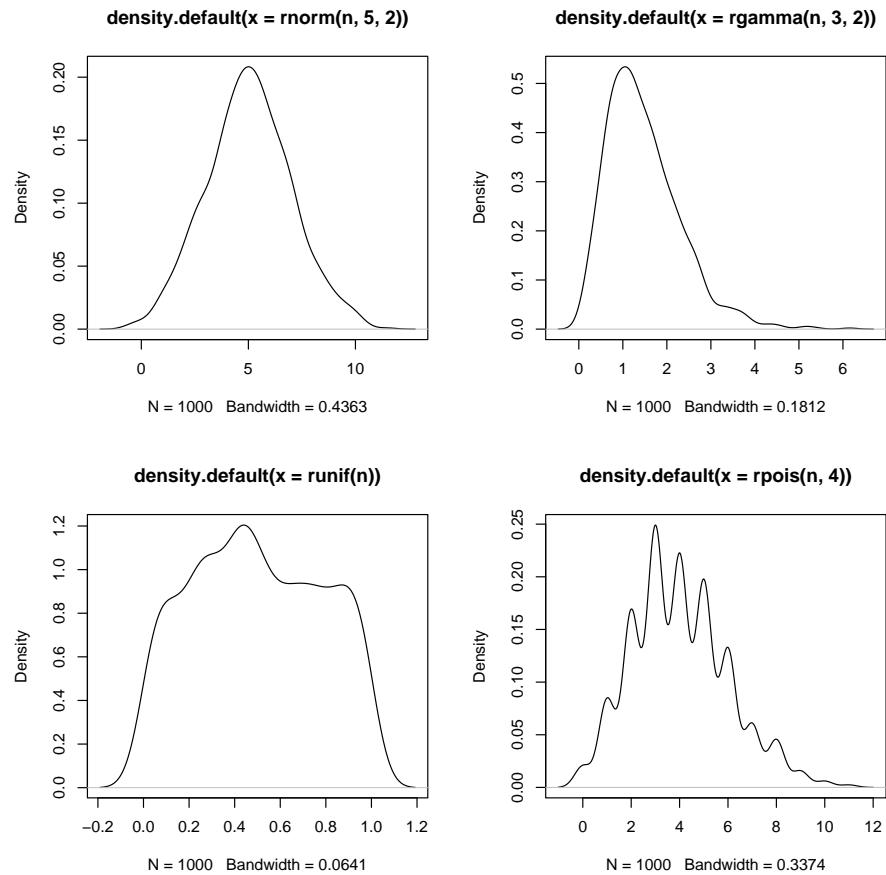
³Distribution x_4 is a mixture of a normal (mean = 5, SD = 2) with a uniform in [3, 20].



The kernel density plot is to the PDF what the ECDF is to the CDF. Just like an ECDF plot presents an estimate of the true CDF, the kernel density plot presents an estimate of the true PDF. This means that presenting a kernel density plot or an ECDF plot is largely a matter of preference because they convey the same information in different ways.

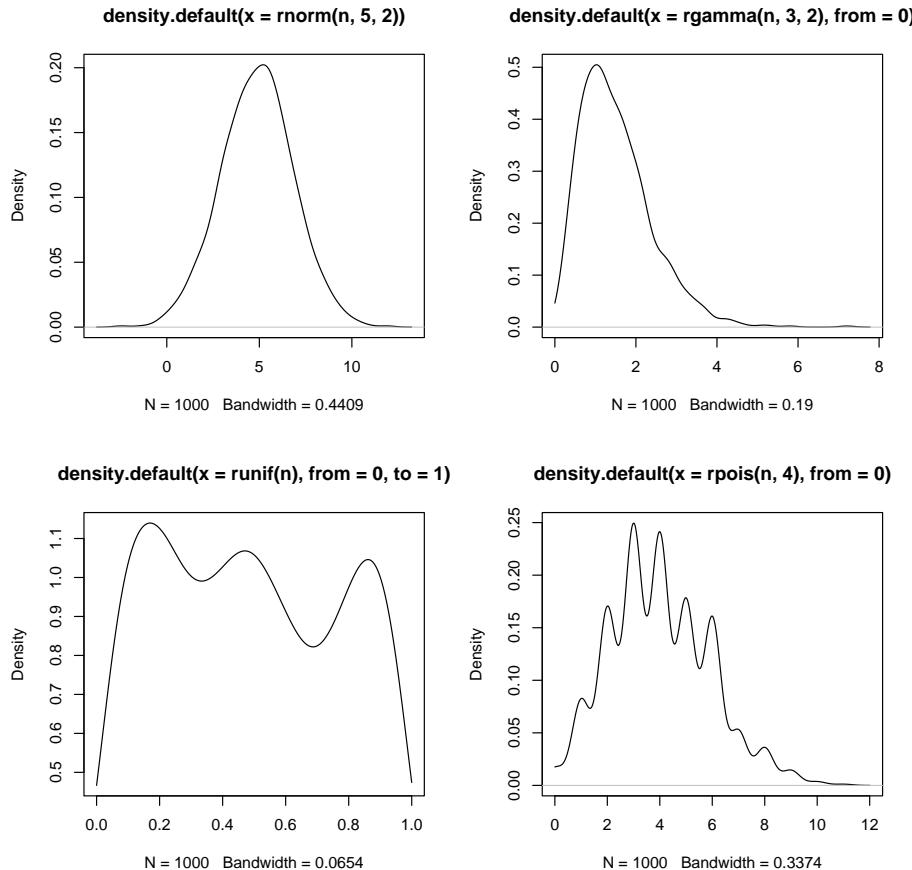
Note that a kernel density plot makes more sense for a continuous distribution than for a discrete distribution. The equivalent plot for a discrete distribution is the **probability mass function (PMF)** plot. The PMF of a discrete distribution is 0 for any non-integer value. The example below shows kernel density plots for four distributions: Normal(mean=5, SD=2), Gamma(k=3, $\theta=2$), Uniform(min=0, max=1), and Poisson($\lambda=4$). The first 3 are continuous distributions and the last is a discrete distribution.

```
set.seed(123)
n <- 1e3
par(mfrow=c(2,2))
plot(density(rnorm(n,5,2)))
plot(density(rgamma(n, 3, 2)))
plot(density(runif(n)))
plot(density(rpois(n,4)))
```



The estimated density functions aren't too far off from the real functions. But, notice that the plots for the gamma, uniform, and Poisson variables extend outside of the domains of the underlying distributions. For example, the density for the Poisson variable extends below 0. You can truncate the plot using arguments `from` or `to` for density. This is a good idea if you are exploring data that you suspect have a natural domain. For example, lengths and times must be non-negative, so you should include `from=0` in the `density()` command.

```
par(mfrow=c(2,2))
plot(density(rnorm(n,5,2)))
plot(density(rgamma(n, 3, 2), from=0))
plot(density(runif(n), from=0, to=1))
plot(density(rpois(n,4), from=0))
```

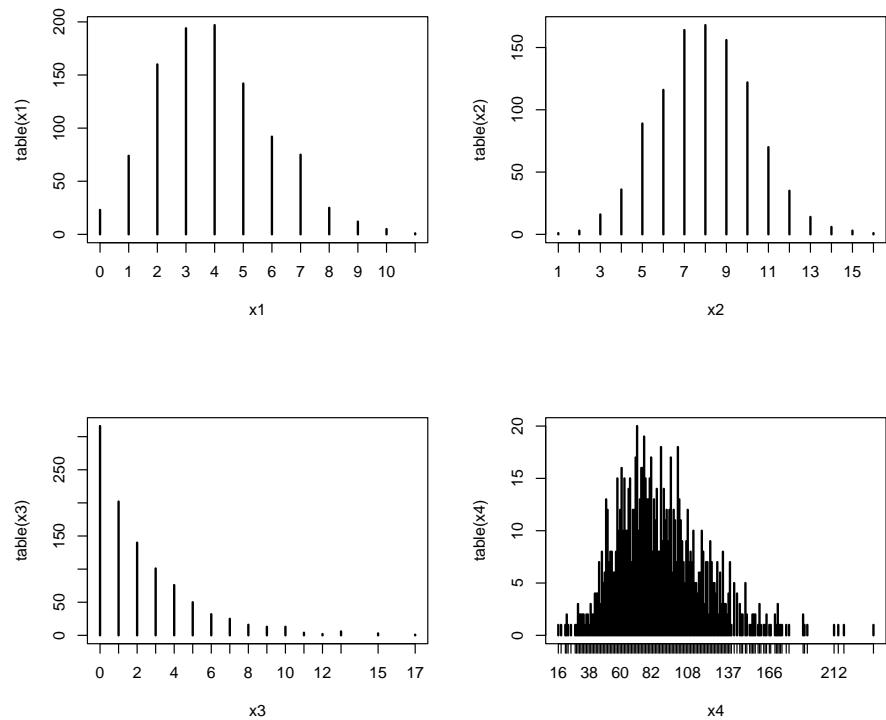


The estimated density functions are now in their supported intervals, which is nice. There's one more issue with the estimates for the Poisson distribution. Notice that the bottom right plot shows non-zero probability density for non-integer values. This doesn't make sense for a Poisson distribution, which can only take non-negative integer values. What's going on here is that the `density()` function does not know whether a distribution is supposed to be discrete or continuous. As far as the function is concerned, the distribution is continuous and just happens to have no non-integer values.

If you have data that you have good reason to suspect are discrete, there is a better way than `density()` to visualize the relative likelihood of different values: the probability mass function (PMF, not PDF). The PMF of a discrete distribution can be calculated rather than estimated. In the example below we use the `table()` function, which tallies up the number of times each value occurs in a vector. We could convert the cell counts to PMF estimates by dividing by the number of observations (e.g., `plot(table(x1)/n)`).

```
# make some data
set.seed(42)
n <- 1e3
x1 <- rpois(n, 4)
x2 <- rbinom(n, 20, 0.4)
x3 <- rgeom(n, 0.3)
x4 <- rnbinom(n, 10, 0.1)

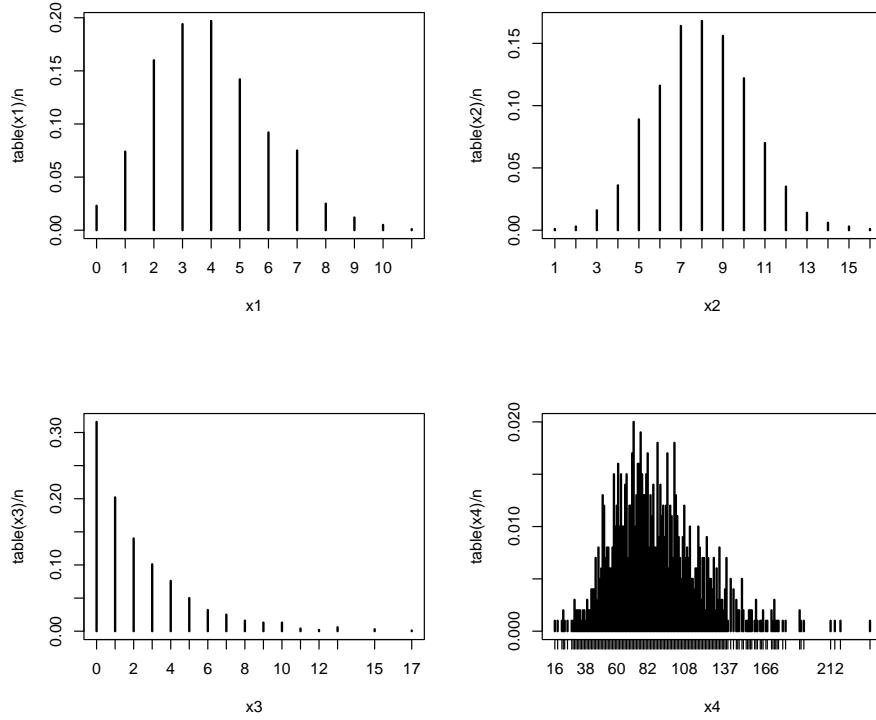
# plot cell counts:
par(mfrow=c(2,2))
plot(table(x1))
plot(table(x2))
plot(table(x3))
plot(table(x4))
```



Dividing the cell counts by the number of total observations (n) estimates the empirical PMF:

```
par(mfrow=c(2,2))
plot(table(x1)/n)
```

```
plot(table(x2)/n)
plot(table(x3)/n)
plot(table(x4)/n)
```



We can verify that the empirical PMFs that we calculated match the actual PMFs:

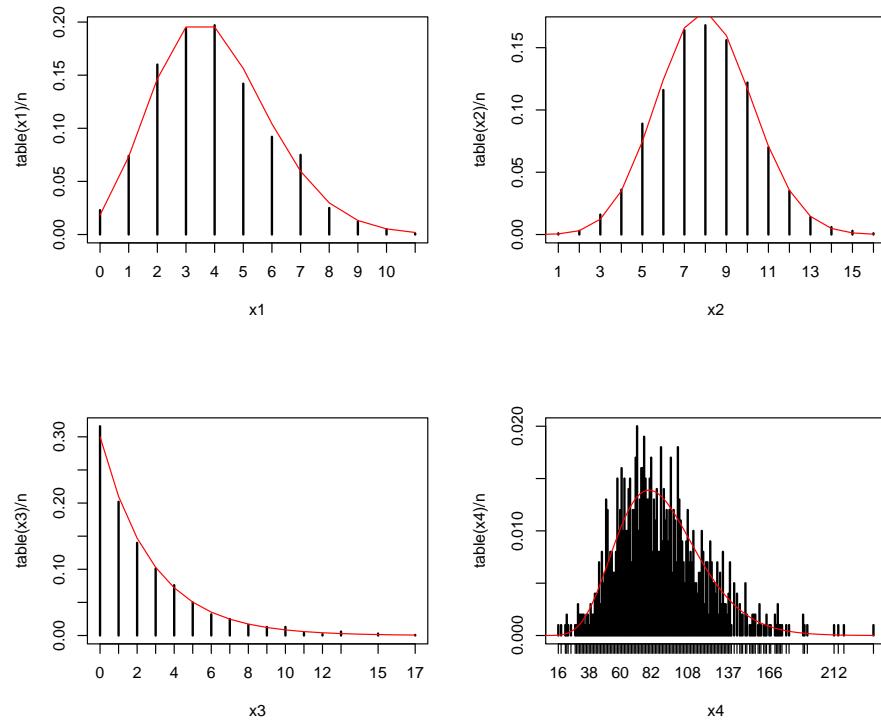
```
# compare to true PMF:
v1 <- 0:max(x1)
v2 <- 0:max(x2)
v3 <- 0:max(x3)
v4 <- 0:max(x4)
y1 <- dpois(v1, 4)
y2 <- dbinom(v2, 20, 0.4)
y3 <- dgeom(v3, 0.3)
y4 <- dnbinom(v4, 10, 0.1)

par(mfrow=c(2,2))
plot(table(x1)/n)
points(v1, y1, type="l", col="red")
```

```

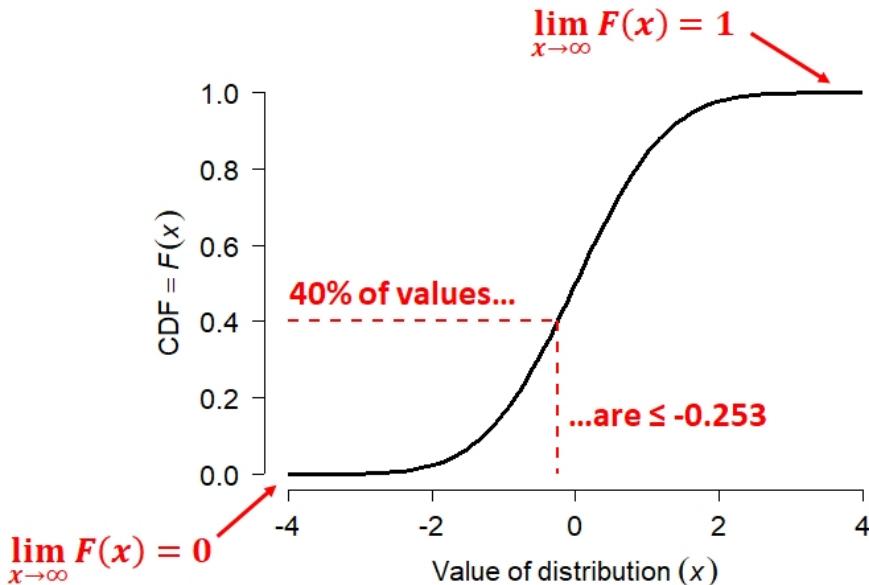
plot(table(x2)/n)
points(v2, y2, type="l", col="red")
plot(table(x3)/n)
points(v3, y3, type="l", col="red")
plot(table(x4)/n)
points(v4, y4, type="l", col="red")

```



4.2.4 Empirical cumulative distribution plots (ECDF)

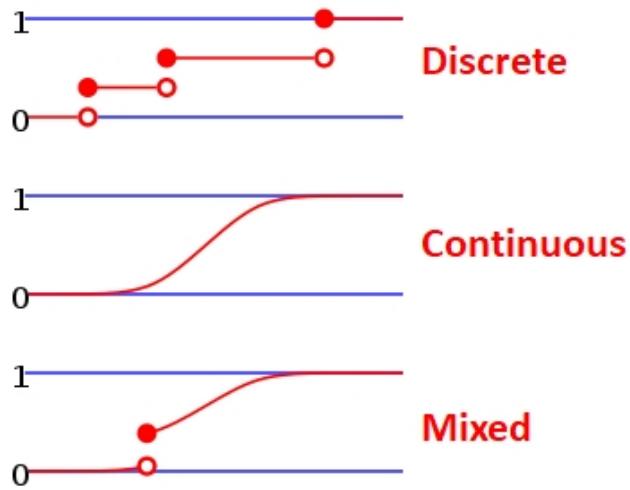
The **cumulative distribution function (CDF)** is the probability that a random variable will take on a value less than or equal to some value. Formally, we say that a continuous distribution X will take on value $\leq x$ with probability $F(x)$. The CDF is $F(x)$. The figure below shows what this means.



The figure shows the CDF of a normal distribution with mean = 0 and SD = 1 (i.e., the standard normal distribution). Like all CDFs, $F(x)$ increases monotonically (never decreasing) from 0 to 1 as x increases from the lower bound to the upper bound of the distribution. In the case of the normal distribution, the bounds of x are $[-\infty, +\infty]$. The CDF approaches 0 as x decreases, and approaches 1 as x increases. The red dashed lines show how to interpret the relationship between the axes. For any value on the x -axis, the y -axis shows what proportion of values are $\leq x$. For any value on the y -axis, the x -axis shows the value at the y -th quantile of the distribution.

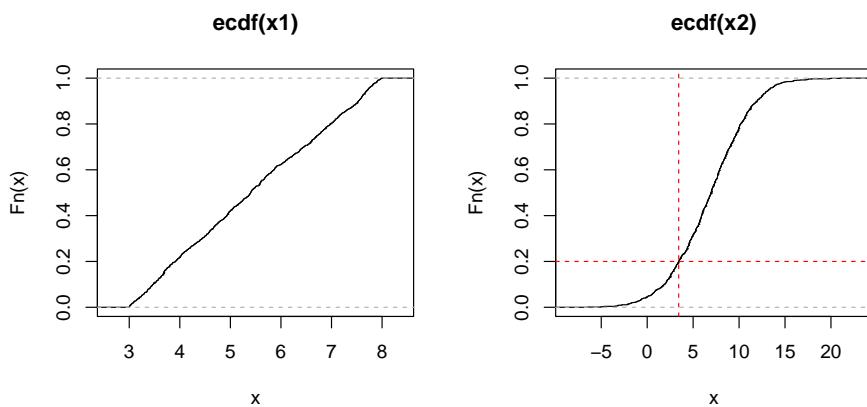
The CDF is in a very real sense the *definition* of a probability distribution. Every probability distribution can be identified by a unique CDF. It doesn't matter whether a distribution is continuous, discrete, or a mixture of both. The figure below shows what the CDFs of a discrete or mixed distribution look like compared to that of a continuous distribution⁴.

⁴adapted from https://commons.wikimedia.org/wiki/File:Discrete_probability_distribution.svg (public domain)



The examples below calculate and plot ECDF plots for a uniform (left) and a normal distribution (right).

```
set.seed(42)
n <- 1e3
x1 <- runif(n, 3, 8)
x2 <- rnorm(n, 7, 4)
par(mfrow=c(1,2))
plot(ecdf(x1))
plot(ecdf(x2))
# add lines at 20th percentile:
abline(h=0.2, lty=2, col="red")
abline(v=quantile(x2, 0.2), lty=2, col="red")
```

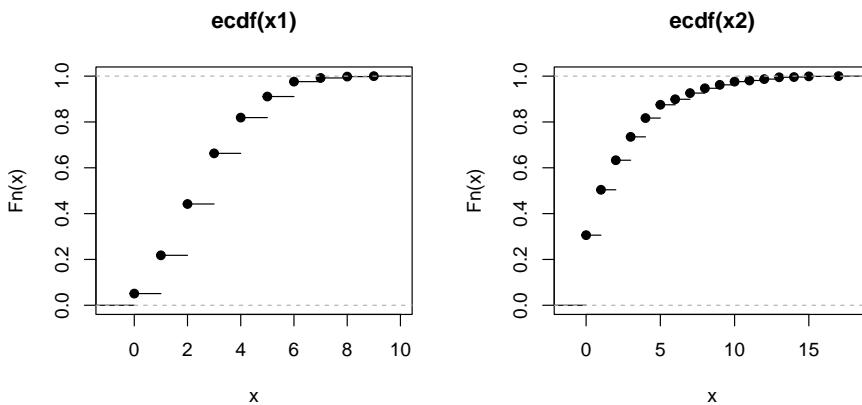


The y -axis of an ECDF plot shows the calculated quantile of the distribution

(e.g., 0.4 quantile = 40th percentile \equiv 40% of x are $\leq F(x)$). The x -axis shows the values of the distribution. In the left plot, we see that the slope of the ECDF plot is roughly constant. This suggests a uniform distribution. The plot on the right shows the S-shaped ECDF typical of a bell-shaped curve, so we might suspect a normal distribution. The red lines on the right plot illustrate that about 20% of values are ≤ 3.6 .

ECDF plots can be calculated for discrete distributions as well—they just look like step functions. Step-like ECDF plots can also appear when a continuous distribution has many more observations than unique values.

```
set.seed(42)
n <- 1e3
x1 <- rpois(n, 3) # Poisson
x2 <- rgeom(n, 0.3) # geometric
par(mfrow=c(1,2))
plot(ecdf(x1))
plot(ecdf(x2))
```

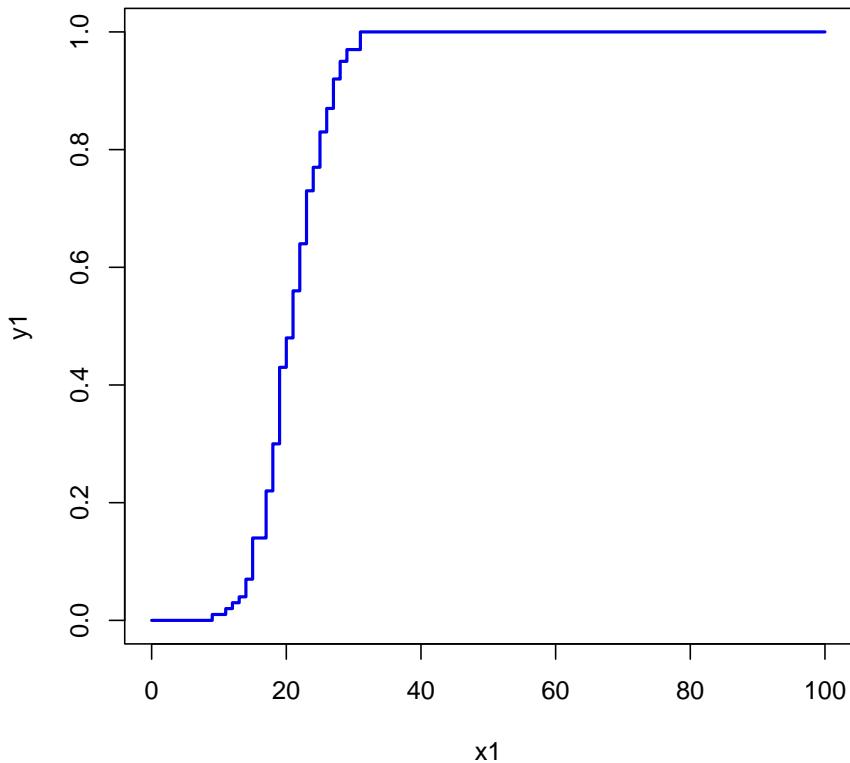


Recall that every result in R is an object. The output of the function `ecdf()` is actually another function that calculates the estimated CDF for a new value. This can be very handy if you want to interpolate the ECDF to make a smoother curve.

```
class(ecdf(rnorm(100)))
## [1] "ecdf"      "stepfun"    "function"
```

Here is how to use it. The example below can be useful if you need to calculate the ECDF for values that are not in the original dataset, but within its domain. In the example below, the ECDF is estimated for $x = 30$, which is not in the original data x . Notice what happens for values in $x1$ that are greater than `max(x)` and less than `min(x)`.

```
set.seed(123)
x <- rnorm(1e2, 20, 5)
e1 <- ecdf(x) # define function for CDF(x)
x1 <- 0:100 # define new x values at which to calculate CDF
y1 <- e1(x1) # calculate CDF at each new x
par(mfrow=c(1,1))
plot(x1, y1, type="s", lwd=2, col="blue2")
```



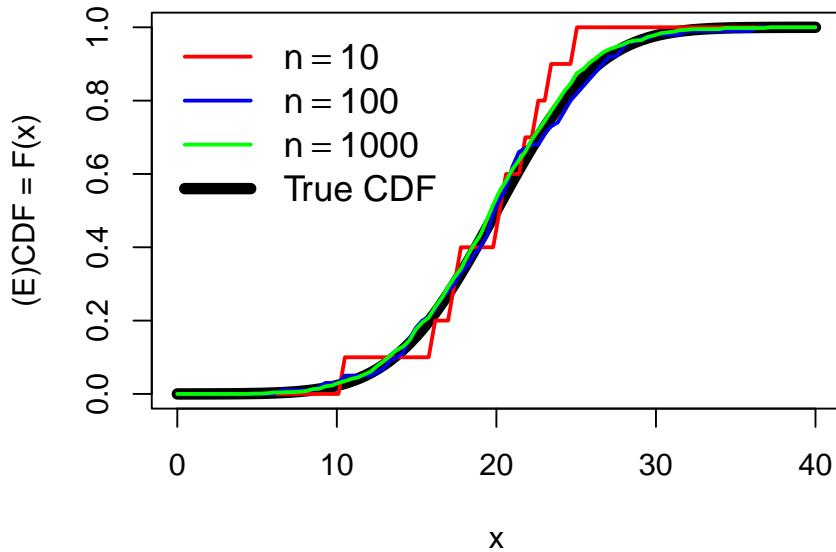
Obviously, the more data points you have, the better and smoother the estimate of the ECDF will be. Consider the simulated example below that estimates the ECDF using different numbers of points.

```
set.seed(123)
n <- 1e4
mu <- 20
```

```
sig <- 5
x4 <- rnorm(n, mu, sig)
x1 <- x4[sample(1:n, 10, replace=TRUE)]
x2 <- x4[sample(1:n, 100, replace=TRUE)]
x3 <- x4[sample(1:n, 1000, replace=TRUE)]

xseq <- seq(0, 40, length=100)
y1 <- ecdf(x1)(xseq)
y2 <- ecdf(x2)(xseq)
y3 <- ecdf(x3)(xseq)
yt <- pnorm(xseq, mu, sig)

plot(xseq, yt, type="l", lwd=6, xlab="x",
      ylab="(E)CDF = F(x)")
points(xseq, y1, type="l", lwd=2, col="red")
points(xseq, y2, type="l", lwd=2, col="blue2")
points(xseq, y3, type="l", lwd=2, col="green")
legend("topleft",
       legend=c(expression(n==10), expression(n==100),
              expression(n==1000), "True CDF"),
       lwd=c(2,2,2,6),
       col=c("red", "blue2", "green", "black"),
       bty="n", cex=1.2)
```



The estimate with 10 points (red) is a very rough fit, but serviceable. The estimates using 100 or 1000 points are much closer to the truth (black line).

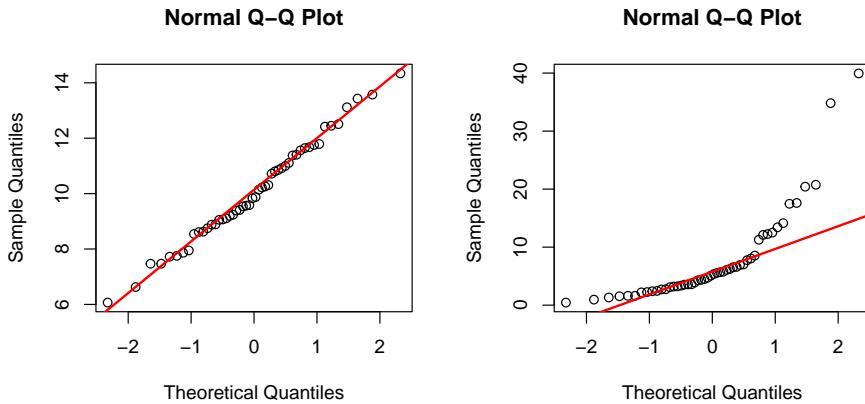
4.2.5 Quantile-quantile (QQ) plots

A **quantile-quantile plot**, or **QQ plot**, is a method for visualizing whether two sets of values come from the same distribution. This is done by plotting the quantiles of one data set against the quantiles of another data set. If the two sets come from the same distribution, then the points should fall along a straight line. QQ plots are usually used to test whether data come from a normal distribution, but could be used with any distribution. The key advantage of QQ plots is that the reference line is straight, and deviations from it are easy to see. Contrast this with the reference line in an ECDF plot, whose shape varies by distribution.

QQ plots are generated by function `qqplot()` or `qnorm()`. The latter function is a shortcut for a QQ plot comparing the data to a normal distribution. The function `qqline()` is used to add a reference line. Typical use of `qnorm()` and `qqline()` is shown below.

```
set.seed(123)
a <- rnorm(50, 10, 2)
b <- rlnorm(50, 1.5, 1)
```

```
par(mfrow=c(1,2))
qqnorm(a)
qqline(a, col="red", lwd=2)
qqnorm(b)
qqline(b, col="red", lwd=2)
```

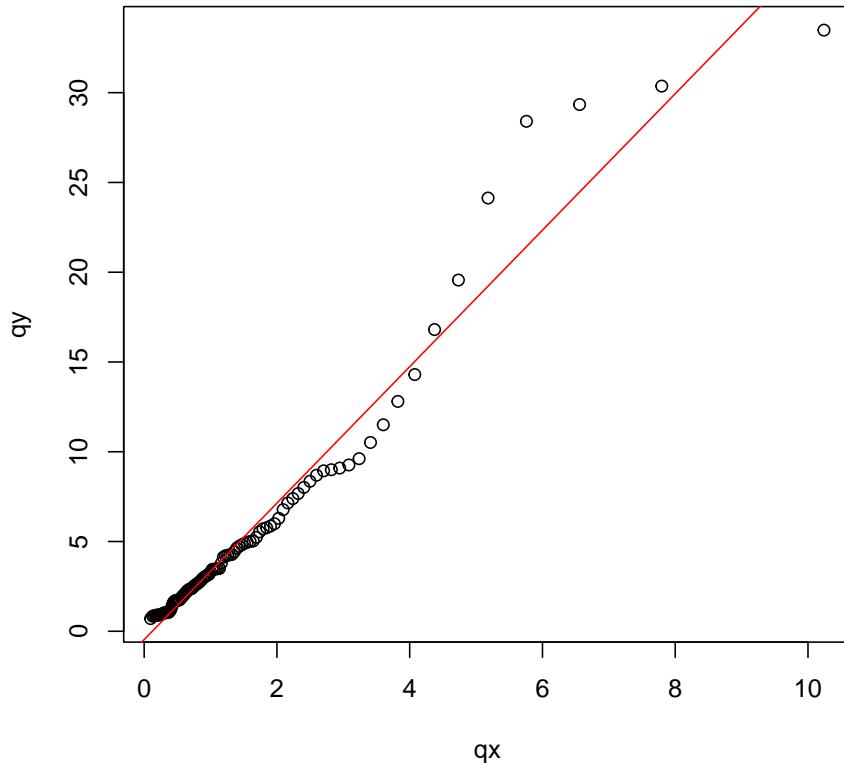


The left panel above shows that distribution a likely follows a normal distribution because the points fall mostly on the reference line. The right panel, however, shows that distribution b likely does not follow a normal distribution. This can be seen because many of the points do not fall on the line. The arch-shaped pattern indicates that the distribution differs from the normal mainly on the tails.

QQ plots can be made to compare your data to any arbitrary distribution. If your target distribution has default parameters in R (e.g., mean = 0 and SD = 1 for the normal distribution), then the method is simple:

```
x <- rlnorm(50, 1.5, 1)
qqs <- 1:99/100
qx <- qlnorm(qqs)
qy <- quantile(x, qqs)

par(mfrow=c(1,1))
plot(qx, qy)
abline(lm(qy~qx), col="red")
```



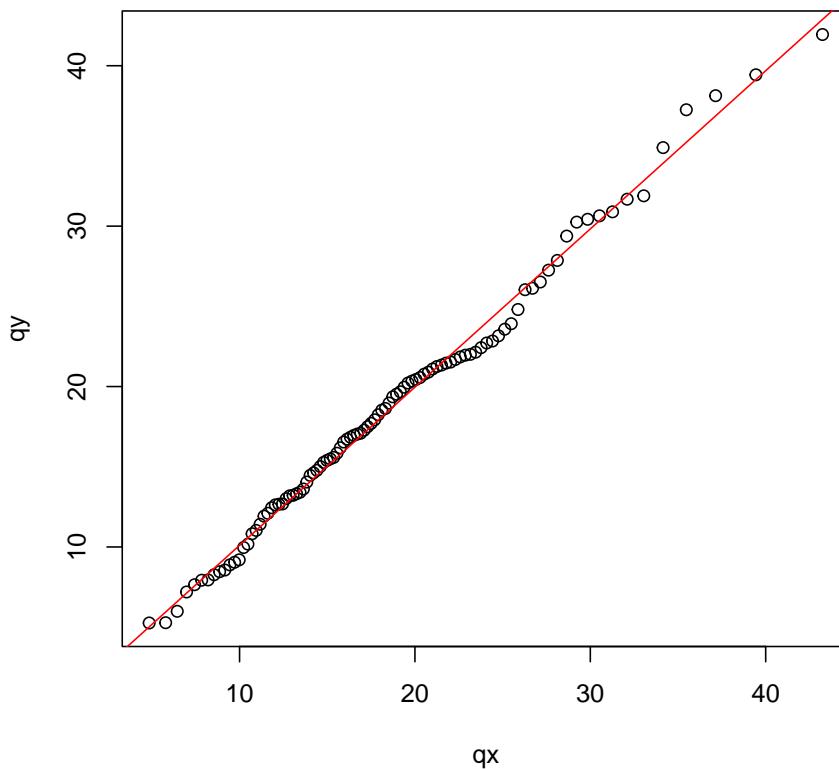
If the target distribution does not have default parameters, or if you don't want to use the default parameters, you can still make a QQ plot. The example below shows two methods for comparing a suspected gamma distribution to a reference gamma distribution using QQ plots. The parameters of the reference gamma distribution are estimated using the function `fitdistr()` from package MASS.

```
library(MASS)
set.seed(123)
n <- 100
x <- rgamma(100, 4, 0.2)

fd <- fitdistr(x, "gamma")
qq <- 1:99/100

# method 1: calculate QQ plot manually
qx <- qgamma(qq, shape=fd$estimate[1], rate=fd$estimate[2])
qy <- quantile(x, qq)
```

```
plot(qx, qy)
abline(lm(qy~qx), col="red")
```

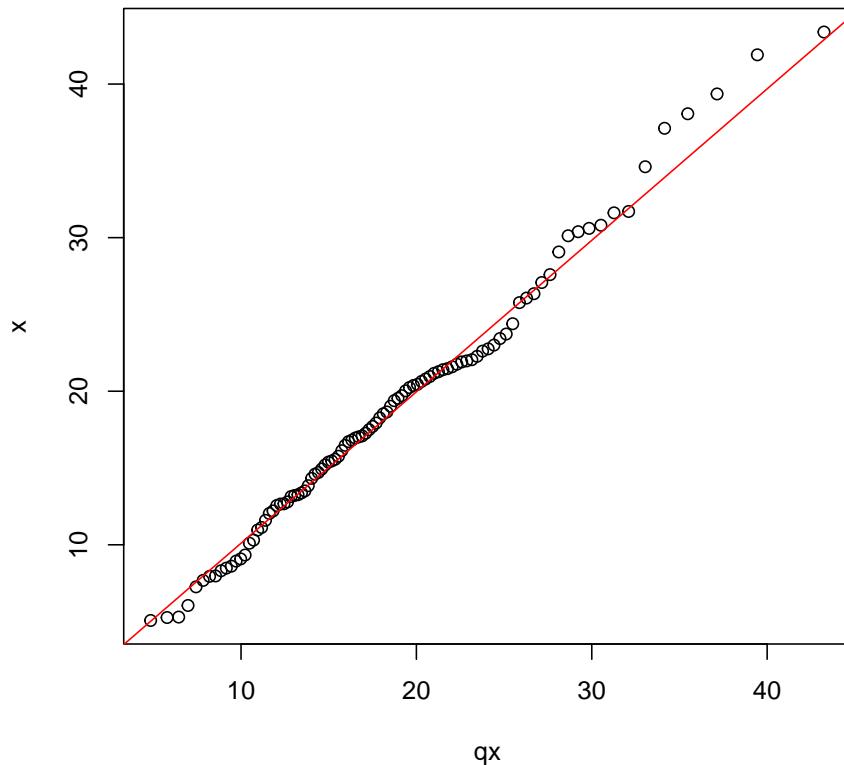


Here is an alternative strategy using `qqplot()`:

```
# method 2: use qqplot()
# quantiles of reference distribution

# function to draw from reference distribution
dfun <- function(p){
  qgamma(qqs, shape=fd$estimate[1], rate=fd$estimate[2])
}
# get values at each reference quantile
qx <- dfun(qqs)
```

```
# make plot
qqplot(qx, x)
abline(lm(qy~qx), col="red")
```



Here is another example with the negative binomial distribution:

```
library(MASS)
set.seed(123)
n <- 50
a <- rnbnom(n, mu=5, size=10)

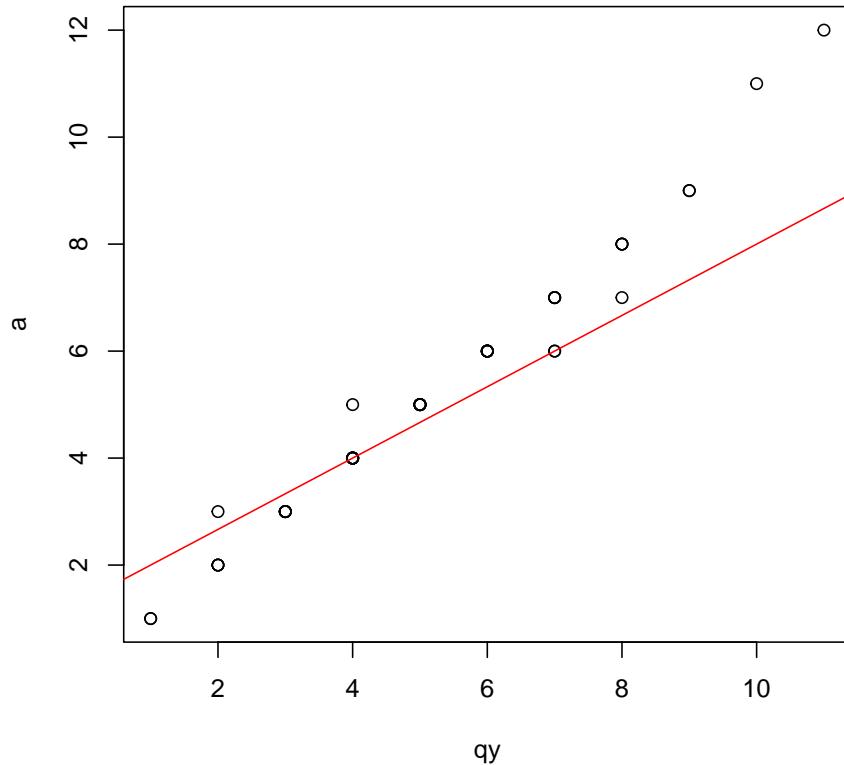
# get parameters of distribution
fd <- fitdistr(a, "negative binomial")
fd

##          size           mu
## 10.0000000 5.0000000
```

```
##   118.3001950      5.2000006
## (394.2399860) ( 0.3295018)

# quantiles of reference distribution
qx <- ppoints(n)
# function to draw from reference distribution
dfun <- function(p){
  qnbinom(p,
           size=fd$estimate[1],
           mu=fd$estimate[2])
  }#function
# get values at each reference quantile
qy <- dfun(qx)

# make plot
qqplot(qy, a)
qqline(a, distribution=dfun, col="red")
```



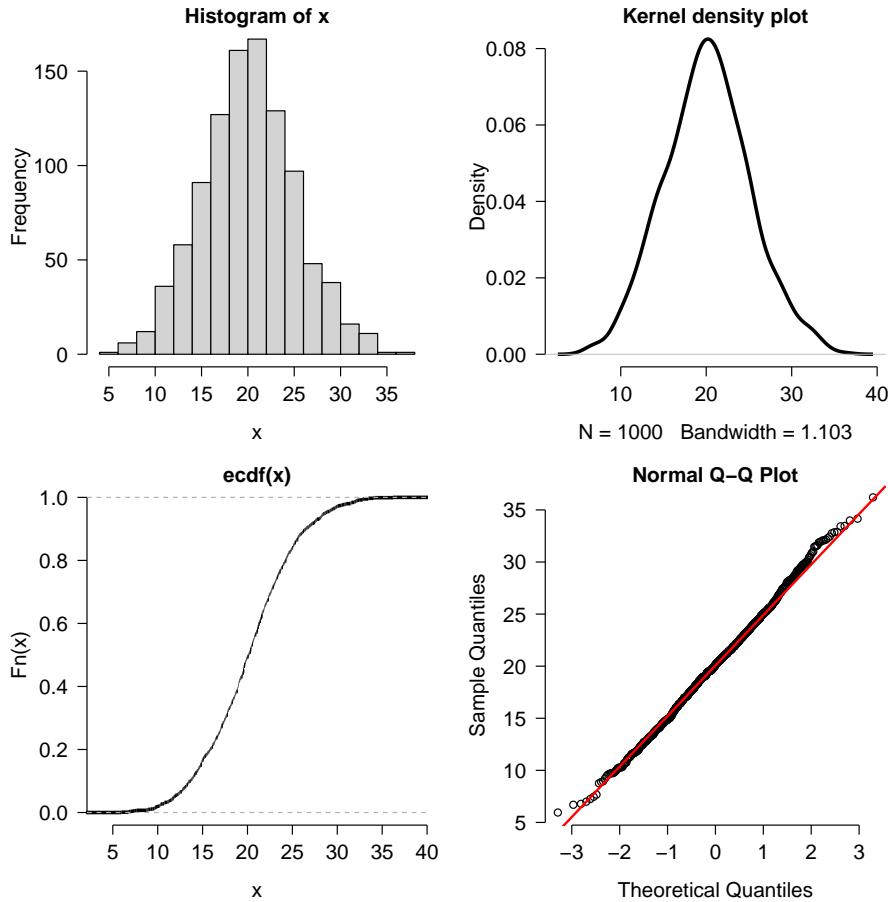
4.2.6 How should I plot my data?

Plotting and examining the distribution of your data is always a good idea. How you plot the distributions is largely up to you. As mentioned above, histograms, ECDF plots, and kernel density plots convey the same information. The information is just presented in different ways. Some people prefer one kind of distribution plot over the others. You should use whichever plot best helps you understand the distribution. Or, use whichever plot your supervisor tells you to use. Consider the figure below, which plots a normal distribution with mean = 20 and SD = 5 in four different ways.

```
x <- rnorm(1e3, 20, 5)

par(mfrow=c(2,2), mar=c(5.1, 5.1, 1.1, 1.1),
    bty="n", lend=1, las=1,
    cex.axis=1.3, cex.lab=1.3, cex.main=1.3)
```

```
hist(x)
plot(density(x), main="Kernel density plot", lwd=3)
plot(ecdf(x), lwd=3)
qqnorm(x)
qqline(x, col="red", lwd=2)
```



Which way is the correct way to plot the data? It depends:

- If your goal is to show how the data are spread out, use a histogram or kernel density plot.
- If your goal is to explore quantiles of the distribution, use an ECDF plot.
- If your goal is to compare to a specific distribution, use a QQ plot or an ECDF (with the reference distribution's CDF superimposed).

Methods for visualizing data distributions can be highly field-specific. Usually a histogram, ECDF, or both, will be enough to get a sense of how a variable is distributed. The goal of this visualization is usually to determine what probability

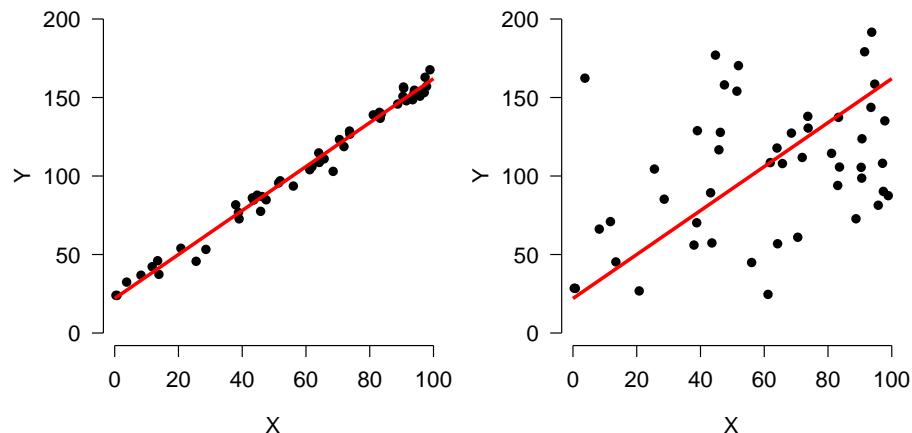
distribution is most appropriate for a variable. As we will see in the next section, that can be as much a biological decision as a statistical one.

4.3 Statistical distributions

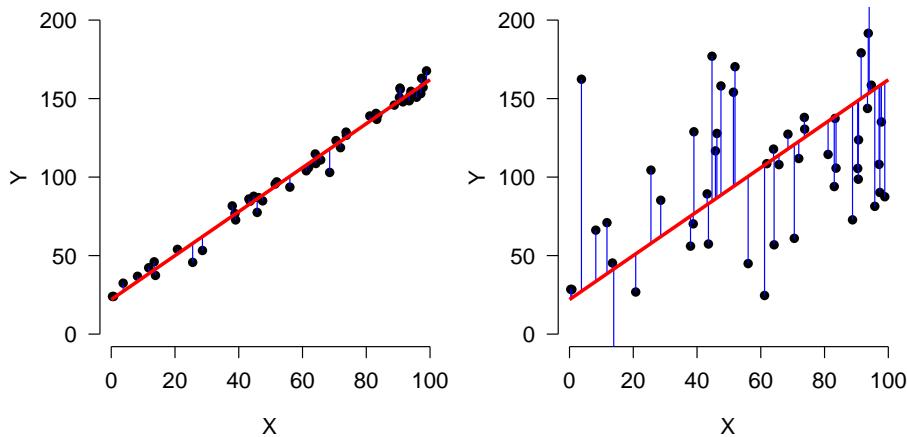
All data contain some element of randomness. Understanding the nature and consequences of that randomness is what motivates much of modern statistics. **Probability distributions** are the mathematical constructs that describe randomness in statistics. This page describes some probability distributions commonly encountered in biology (and a few that aren't common). The emphasis here is on practical understanding of what the distributions imply about data—not on the theoretical underpinnings or mathematical details. If you want or need a rigorous introduction to probability theory, this is probably not the right place.

4.3.1 Probability distributions

All data contain some element of randomness. Understanding the nature and consequences of that randomness is what motivates much of modern statistics. Consider the figures below:



Both scatterplots show a linear relationship between X and Y . But what is different about the plot on the right? Variation. Both plots show the same relationship ($Y = 22 + 1.4X$), but they differ in that the variation about the expected value (the red line) is much greater in the right plot than the left plot. Consequently, X appears to explain much more variation in Y in the left plot than in the right plot. The next figure shows that the **residual variation**, or difference between the expected and observed values, is much greater in the right panel. Each of these differences between the observed value of Y (Y_i) and the expected value of Y ($E(Y_i)$), or $Y_i - E(Y_i)$, is called a **residual**.

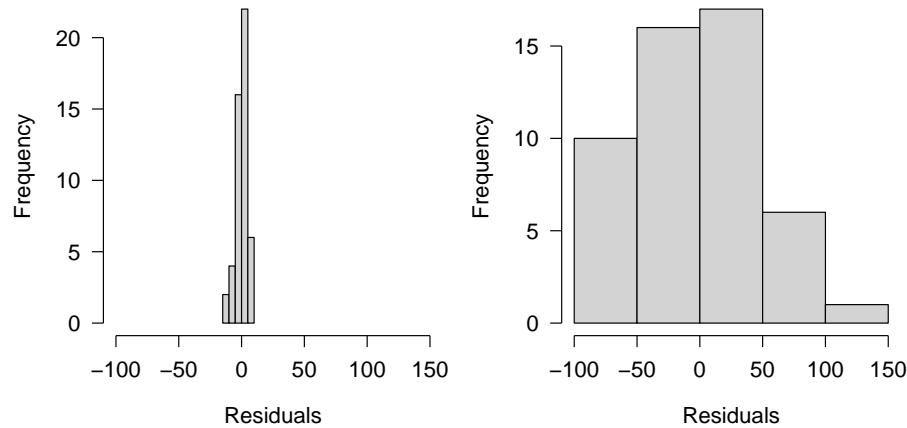


Residuals are an important part of statistical analysis for two reasons. First, most statistical models make assumptions about the distribution of residuals. Second, the total magnitude of the residuals (usually expressed as sum of squared residuals) is useful for calculating many measures of how well a model fits the data. The residuals of a statistical model are the differences between the observed values and the values predicted by the model. The residual for observation i in variable Y , R_i , is calculated as:

$$R_i = Y_i - E[Y_i]$$

where $E(Y_i)$ is the **expected value** of Y_i . This means that when an observation has a greater value than predicted, the residual is positive; similarly, when an observation is smaller than expected, the residual is negative. In many statistical methods, residuals are squared so that (1) positive and negative residuals do not cancel out; and (2) larger residuals carry more weight.

The figures below show the distributions of residuals for the example linear regressions above. Notice that the left dataset has a much smaller distribution of residuals than the right dataset. This is because of the much tighter fit of the left dataset's Y values to the predicted curve.



Notice also that both distributions of residuals have a similar shape, despite the difference in width. This shape is actually very important: the **normal distribution**. If the residuals were not distributed this way (i.e., did not follow a normal distribution), then we would be in trouble for two reasons. First, the data were generated using a normal distribution for residuals, so something would have had to have gone wrong with our statistical test; and second, the linear regression model used to analyze the data assumes that residuals are normally distributed. If they are not, then the test is not going to produce valid estimates of statistical significance or model parameters.

The example above is an example of the importance of thinking about statistical distributions when analyzing data. So just what is a statistical distribution? Distributions are mathematical functions that define the probabilities of random outcomes. Here **random** means that there is some element of chance in what we observe. This randomness is not completely unpredictable. While the outcome of specific observations might be unknowable, we can make predictions about long run frequencies or averages of lots of observations. In other words, single observations are not predictable, but the properties of sets of observations are. Such properties might include the number of times a specific outcome occurs (e.g., number of times a flipped coin comes up heads) or some summary of observations (e.g., the mean tail length of chipmunks). Another term for this kind of randomness that follows a pattern is **stochastic**.

Another example: coin flips

Consider flipping a coin. A fair coin will come up “heads” 50% of the time, and “tails” the other 50%. If you flip a coin once, the probability of getting heads is 50%. But what about if you flip the coin 10 times? How many heads should you get? 5 is a reasonable guess. The table below shows the possible outcomes to 10 coin flips:

Heads	Tails
0	10
1	9
2	8
3	7
4	6
5	5
6	4
7	3
8	2
9	1
10	0

This table shows that 5 heads is only one of 11 possibilities! So, is the probability of 5 heads 1 in 11 (≈ 0.091)? Of course not, because not all outcomes are equally likely.

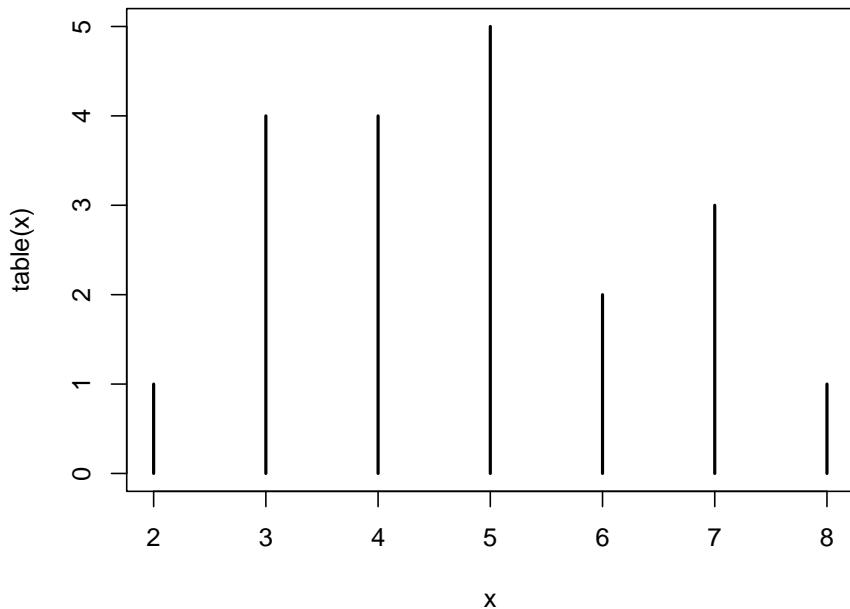
One of the reasons statistics is so fun is that we can often use simulation to discover patterns. The R code below will simulate the effects of flipping 1 coin 10 times. Run this code a few times and see what happens.

```
rbinom(1,10,0.5)
```

```
## [1] 4
```

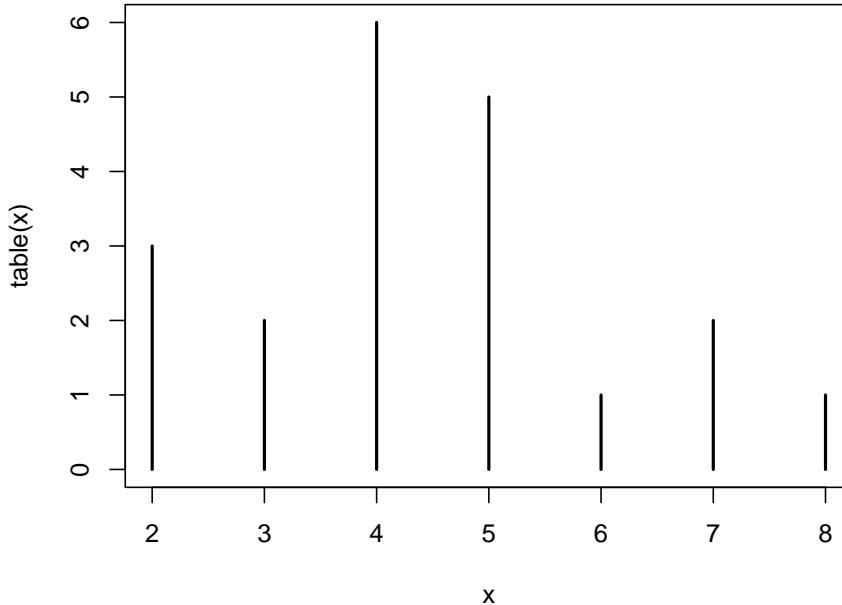
A typical run of results might be something like 4 5 5 4 3 5 6 6 5 5 4 6. We can repeat this line many times and graph the results:

```
x <- numeric(20)
for(i in 1:20){x[i] <- rbinom(1,10,0.5)}
plot(table(x))
```



There is actually a faster way, by requesting 20 draws of 10 flips each all at once:

```
x <- rbinom(20, 10, 0.5)
plot(table(x))
```



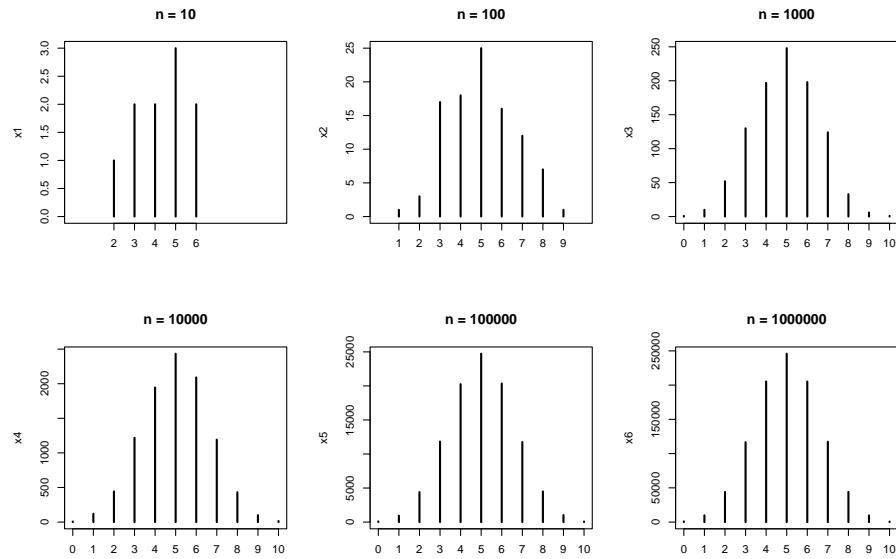
Is 5 the most likely value? What about if we get a larger sample?

```

x1 <- table(rbinom(10,10,0.5))
x2 <- table(rbinom(100,10,0.5))
x3 <- table(rbinom(1000,10,0.5))
x4 <- table(rbinom(10000,10,0.5))
x5 <- table(rbinom(100000,10,0.5))
x6 <- table(rbinom(1000000,10,0.5))

par(mfrow=c(2,3))
plot(x1, xlim=c(0, 10), main="n = 10")
plot(x2, xlim=c(0, 10), main="n = 100")
plot(x3, xlim=c(0, 10), main="n = 1000")
plot(x4, xlim=c(0, 10), main="n = 10000")
plot(x5, xlim=c(0, 10), main="n = 100000")
plot(x6, xlim=c(0, 10), main="n = 1000000")

```



```
# reset graphical parameters
par(mfrow=c(1,1))
```

You may have guessed by now that the properties of a set of coin flips—such as what number of heads to expect—are described by some kind of mathematical idea. This idea is called a statistical distribution. This particular distribution is the **binomial distribution**, which describes the outcome of any random process with a binary outcome. The name “binomial” is descriptive: “bi” for two, and “nomial” for names or states. Can you think of a biological situation where the binomial distribution might apply?

The sections below describe some distributions commonly encountered in biology, and what kinds of processes give rise to them. It is important to be able to relate biological phenomena to statistical distributions, because the underlying nature of the randomness in some process can affect how we analyze that process statistically. Note that in this class we will focus on the practical applications of these distributions, rather than their mathematical derivations.

4.3.2 Probability distributions in R

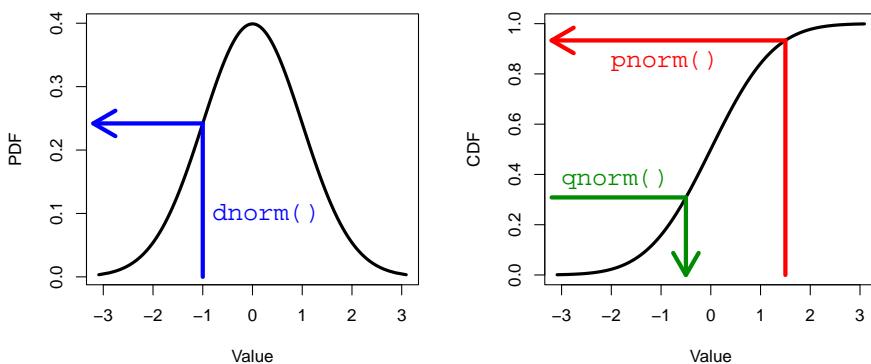
Every distribution supported in R has four functions associated with it: `r_()`, `q_()`, `d_()`, and `p_()`, where `_` is the name of the distribution (or an abbreviation thereof). These 4 functions calculate different values defined by the distribution:

- `r_()` draws **random numbers** from the distribution.
- `p_()` calculates the **cumulative distribution function (CDF)** at a

given value. The reverse of `q_()`.

- `d_()` calculates the **probability density function (PDF)**; i.e., the height of the density curve, or first derivative of the CDF.
- `q_()` calculates the **value at a given quantile**. The reverse of `p_()`.

The figure below shows these functions in relation to the PDF and CDF for a normal distribution.



4.3.3 Discrete distributions

Discrete distributions can take on integer values only. Because of this, many discrete distributions are related in some way to **count data**. Count data result from, well, counting things. Count data can also describe the number of times something happened.

4.3.3.1 Bernoulli distribution

The simplest discrete distribution is the **Bernoulli distribution**. It describes the outcome of a single random event with probability p . Thus, the Bernoulli distribution takes the value 1 with probability p or the value 0 with probability $q = (1 - p)$. Any opportunity for the event to happen is also called a **Bernoulli trial**, and the process that it describes a **Bernoulli process**. The probability p can take on any value in the interval $[0, 1]$.

For convenience we usually consider a Bernoulli distribution to take the value of 0 or 1, but really it could represent any binary outcome. For example, *yes vs. no*, *dead vs. alive*, < 3 vs. ≥ 3 , etc., are all outcomes that could be modeled as Bernoulli variables. By convention, the event that occurs with probability p is considered a **success** and has the numerical value 1; the even that occurs with probability $1-p$ is considered a **failure** and takes the numerical value 0. This is how Bernoulli variables are represented in most statistical packages, including R.

The Bernoulli distribution is rarely used on its own in an analysis⁵. Instead, it's useful to think of the Bernoulli distribution as a special case of, or a building block of, more complicated distributions such as the binomial distribution. For example, a single observation of a binomially-distributed variable could be thought of as a Bernoulli distribution.

The Bernoulli distribution is a special case of the binomial distribution, and so it is accessed in R using the functions associated with the binomial. With the `size` argument set to 1, the binomial distribution *is* the Bernoulli distribution.

```
# flip one fair coin
rbinom(1, 1, 0.5)

## [1] 0

# flip 10 fair coins
rbinom(10, 1, 0.5)

## [1] 1 1 0 1 1 0 1 0 1 1
# flip 10 coins with a 70% chance of heads
rbinom(10, 1, 0.7)

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

4.3.3.2 Binomial distribution

The binomial distribution describes the **number of successes** in a set of independent Bernoulli trials. Each trial has probability of success p , and there are n trials. The values n and p are the **parameters** of the binomial distribution—the values that describe its behavior. The coin flipping example above is an example of a binomial distribution. Biological examples of binomial processes might be the number of fish that die in an experiment, or the number of plants that flower in a season.

Because it is so simple, thinking about the binomial distribution is a good warm up for learning about the characteristics of probability distributions. One of the most important characteristics is the **expected value** of the distribution. This is the value that most likely to occur, or the **central tendency** of values. There are several kinds of expected value, but they are all related to the most common or central value. The expected value, or mean, of a binomial distribution X is

$$E(X) = \mu = np$$

This is the answer to the question earlier about how many heads to expect if a fair coin is flipped 10 times:

⁵After all, how useful is a dataset with a sample size of 1?

$$E(\text{heads}) = n(\text{flips})p(\text{heads})$$

If a variable comes from a binomial process, we can make other inferences about it. For example, we can estimate its variance as:

$$\text{Var}(X) = \sigma^2 = np(1 - p) = npq$$

We can also estimate the probability of any number of successes k as:

$$P(k) = \binom{n}{k} p^k (1 - p)^{n-k}$$

The first term (n over k) is known as the **binomial coefficient** and is calculated as:

$$\binom{n}{k} = \frac{n!}{k!(n - k)!}$$

This term represents the number of ways of seeing k successes in n trials. For example, a set of 4 trials could have 2 successes in 6 ways: HHTT, HTHT, THHT, HTTH, THTH, and TTTH.

When n is small, the binomial distribution can be quite skewed (i.e., asymmetric) because the distribution has a hard lower bound of 0. Note that the expression for $P(k)$ is what is calculated by function `dbinom()` below, and related to what is calculated by function `pbinom()`. For large n , the binomial distribution can be approximated by a normal distribution (see below) with mean = np and variance = npq .

4.3.3.2.1 Binomial distribution in R R uses a family of 4 functions to work with each probability distribution. The binomial and Bernoulli distributions are accessed using the `_binom` group of functions: `dbinom()`, `pbinom()`, `qbinom()`, and `rbinom()`. Each function calculates or returns something different about the binomial distribution:

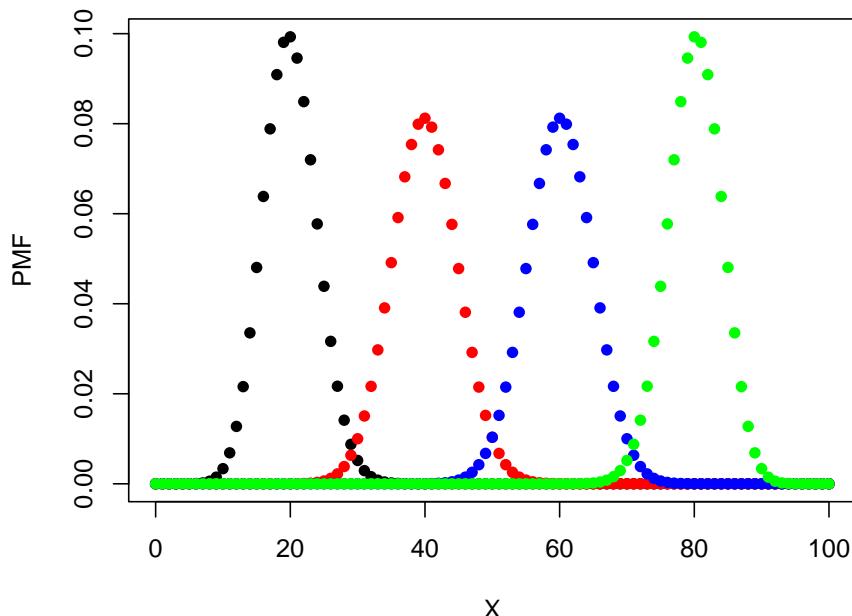
- `dbinom()`: Calculates probability mass function at x for binomial distribution given n and p . Answers the question “What is the probability of x successes in n trials with probability p ?”
- `pbinom()`: Calculates integral of the probability mass function for a binomial distribution given n and p , from 0 up to x . In other words, given some binomial distribution, at what quantile of that distribution should some value fall? The reverse of `qbinom()`. Answers the question “What is the probability of at least x successes in n trials with probability p ?”
- `qbinom()`: Calculates the value at specified quantile of a binomial distribution. Essentially the reverse of `pbinom()`.

- `rbinom()`: Draws random numbers from the binomial distribution defined by n and p (or from the Bernoulli distribution if $n = 1$).

Let's explore the binomial distribution using these functions. In the plots produced in the two examples, notice how the variance of each distribution (x_1 , x_2 , etc.) depends on both n and p . The variance in these plots is shown by the width of the distribution.

```
# N = 100, P various
N <- 100
X <- 0:100
x1 <- dbinom(X, N, 0.2)
x2 <- dbinom(X, N, 0.4)
x3 <- dbinom(X, N, 0.6)
x4 <- dbinom(X, N, 0.8)

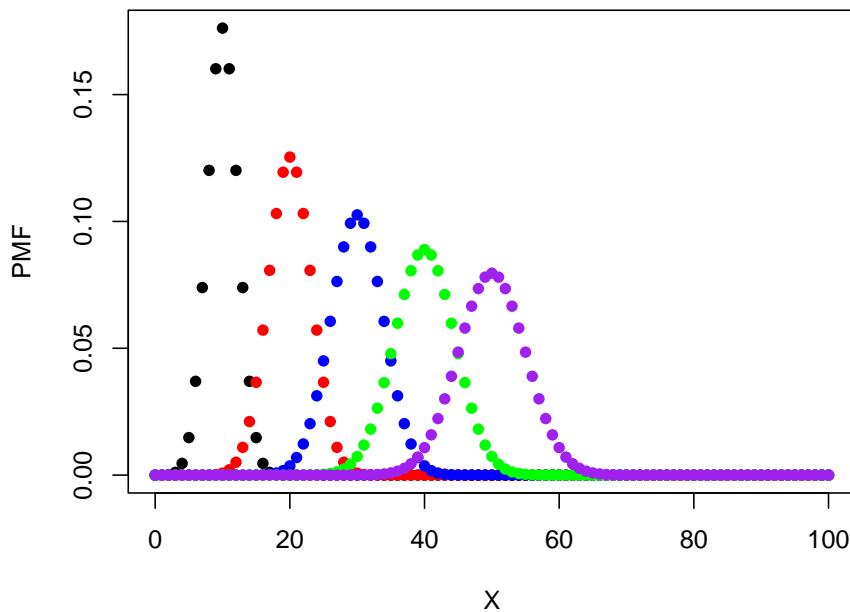
par(mfrow=c(1,1))
plot(X, x1, pch=16, xlab="X", ylab="PMF")
points(X, x2, pch=16, col="red")
points(X, x3, pch=16, col="blue")
points(X, x4, pch=16, col="green")
```



Here is a plot showing the effect of varying n .

```
# P = 0.5, N various
P <- 0.5
x1 <- dbinom(0:100, 20, P)
x2 <- dbinom(0:100, 40, P)
x3 <- dbinom(0:100, 60, P)
x4 <- dbinom(0:100, 80, P)
x5 <- dbinom(0:100, 100, P)

plot(0:100, x1, pch=16, xlab="X", ylab="PMF")
points(0:100, x2, pch=16, col="red")
points(0:100, x3, pch=16, col="blue")
points(0:100, x4, pch=16, col="green")
points(0:100, x5, pch=16, col="purple")
```

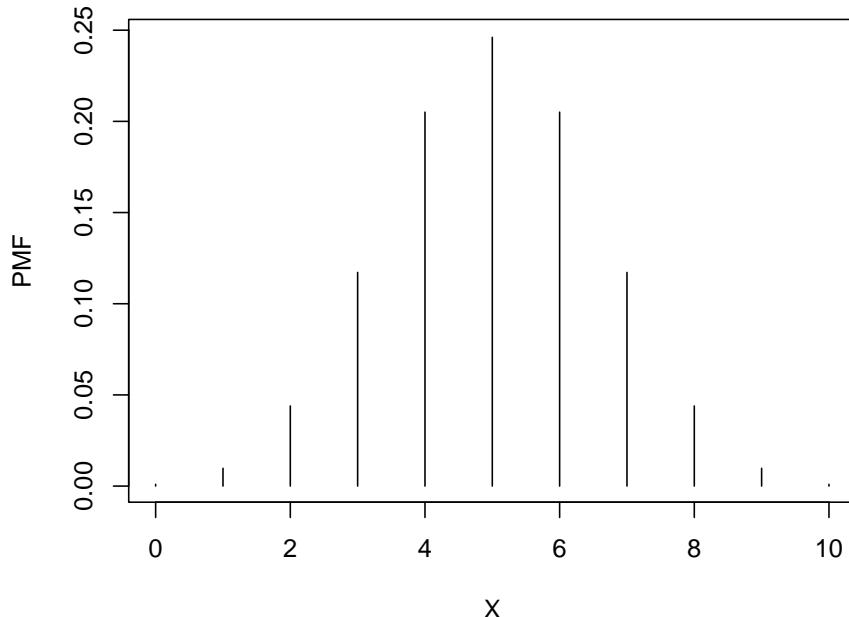


In these plots, the height of the points is called the **probability mass function (PMF)**. For discrete distributions like the binomial, the PMF of any value is the probability that the distribution takes on that value. The sum of the PMF for all integers from 0 to n (inclusive) must be equal to 1. You can verify this by calculating the sum of any of the vectors of densities above.

The function `pbinom()` sums the PMF from 0 up to and including some value. In other words, it calculates the **cumulative distribution function (CDF)**. This answers the question “where in the distribution is value x ?”; put another way, “at what quantile of the distribution does value x lie?”. Yet another way to ask this is, “What is the probability of at least X successes?”.

Consider a binomial distribution with $n = 10$ and $p = 0.5$. What is the probability that the distribution takes a value ≤ 7 ? We can calculate this as the sum of the PMF for 0 through 7.

```
# plot the distribution to see it
N <- 10
P <- 0.5
xd <- dbinom(0:10, N, P)
plot(0:10, xd, type="h", xlab="X", ylab="PMF")
```



```
# calculate p(x<=7)
sum(xd[1:8]) # indices 1:8 correspond to values 0:7
```

```
## [1] 0.9453125
```

```
# same value:  
pbinom(7, N, P)  
  
## [1] 0.9453125
```

Or, we could want to know the probability that the distribution takes on a value > 6 . This would be the complement of the sum up to and including 6.

```
1-sum(xd[1:7])  
  
## [1] 0.171875  
  
# same value:  
1-pbinom(6,N,P)  
  
## [1] 0.171875
```

If the last two calculations seem familiar, that's because this is exactly how P values for statistical tests are calculated (e.g., using `pf()` to get the probability from an F distribution for an ANOVA).

The function `qbinom()` is basically the reverse of function `pbinom()`. Rather than calculate the quantile at which a value falls in the distribution, `qbinom()` calculates the value at which a quantile falls. For example, what value do we expect to find at the 60th percentile of a distribution? Put another way, how many successes should 60% of experiments have, on average? The example below shows how `qbinom()` and `pbinom()` are reversible.

```
N <- 10  
P <- 0.5  
pbinom(6, N, P)  
  
## [1] 0.828125  
qbinom(0.828125, N, P)
```

```
## [1] 6
```

Finally, `rbinom()` draws random values from the binomial or Bernoulli distributions. The syntax of this function can be a little confusing. The first argument, `n`, is the *number of random draws* that you want. The second argument, `size`, is the *number of values in each draw*; that is, the parameter n of the binomial distribution. Compare these results, all with $p = 0.5$:

```
# 1 draw of 1 trial  
rbinom(1, 1, 0.5)  
  
## [1] 0  
# 1 draw of 10 trials  
rbinom(1, 10, 0.5)
```

```

## [1] 8
# 10 draws of 1 trial per draw
rbinom(10, 1, 0.5)

## [1] 0 0 1 0 1 1 1 0 1 0
# 10 draws of 10 trials per draw
rbinom(10, 10, 0.5)

## [1] 3 5 5 7 3 5 5 5 4 3

```

Result 1 shows a Bernoulli distribution with $n = 1$. Result 2 shows a single value from a binomial distribution with $n = 10$. Result 3 shows 10 results from Bernoulli distributions. Notice that if you add up the values in result 3, you get a result like Result 2. Finally, Result 4 shows 10 draws from a binomial distribution, which itself has $n = 10$. The take home message is that the first argument to `rbinom()` is not a parameter of the binomial distribution. It is instead the number of draws from the distribution that you want. The `size` parameter is the argument that helps define the distribution.⁶

4.3.3.3 Poisson distribution

The **Poisson distribution** is widely used in biology to model **count data**. If your data result from some sort of count or abundance per time interval, spatial extent, or unit of effort, then the Poisson distribution should be one of the first things to try in the analysis. The Poisson distribution has one parameter, λ (“lambda”), which represents the **expected number** of objects counted in a sample (objects being trees, fish, cells, mutations, kangaroos, etc.). This parameter is also the **variance** of the distribution.

Like the binomial distribution, the Poisson distribution is discrete, meaning that it can only take on integer values. Unlike the binomial distribution, which is bounded by 0 and n , the Poisson distribution is bounded by 0 and $+\infty$. However, values $\gg \lambda$ are highly improbable. If there is a well-defined upper bound for your count, then you might consider them to come from a binomial distribution instead of a Poisson. For example, if you are counting the number of fish in a toxicity trial that survive, the greatest possible count is the number of fish in the trial. On the other hand, if your counts have no *a priori* upper bound, then use the Poisson.

The expected value and the variance of the Poisson distribution are both λ :

$$E(X) = Var(X) = \lambda$$

4.3.3.3.1 Poisson distribution in R

The Poisson distribution is accessed using the `_pois` group of functions, where the space could be `d`, `p`, `q`, or `r`. These

⁶If your brain hurts, that’s completely normal. This function gives me headaches too

functions calculate or returns something different:

- **dpois()**: Calculates **probability mass function (PMF)** at x for Poisson distribution given λ . Answers the question, “what is the probability of observing a count of x given λ ?“
- **ppois()**: Calculates CDF, or integral of PMF, from 0 up to x given λ . In other words, given some Poisson distribution, at what quantile of that distribution should some value fall? The reverse of **qpois()**.
- **qpois()**: Calculates the value at specified quantile of a Poisson distribution. The reverse of **ppois()**.
- **rpois()**: Draws random numbers from the Poisson distribution defined by λ .

Something important to keep in mind about the Poisson distribution is that it only makes sense for discrete counts, not for continuous measurements that are rounded. For example, if you are measuring leaf lengths and round every length to the nearest mm, it might be tempting to use a Poisson distribution to analyze the data because all of the values are integers. But that would be incorrect, because leaf lengths could theoretically take on any positive value. Furthermore, treating rounded continuous values as discrete leads to the awkward issue that changing measurement units can change dimensionless statistics.

For example, the **coefficient of variation (CV)** is the ratio of a distribution’s SD to its mean. Thus, it is unitless and should be independent of units. The CV of a Poisson distribution X is:

$$CV(X) = \frac{\sqrt{\lambda}}{\lambda}$$

So, if you measured 20 leaves and found a mean length of 17 cm, the CV would thus be ≈ 0.242 or 24%. But if you convert the measurements to mm, the CV would be about 0.077, or 7.7%! (In fact, if you change λ by some factor a , then the CV will be scaled by \sqrt{a}/a).

The figure below shows the effect of varying λ on a Poisson distribution.

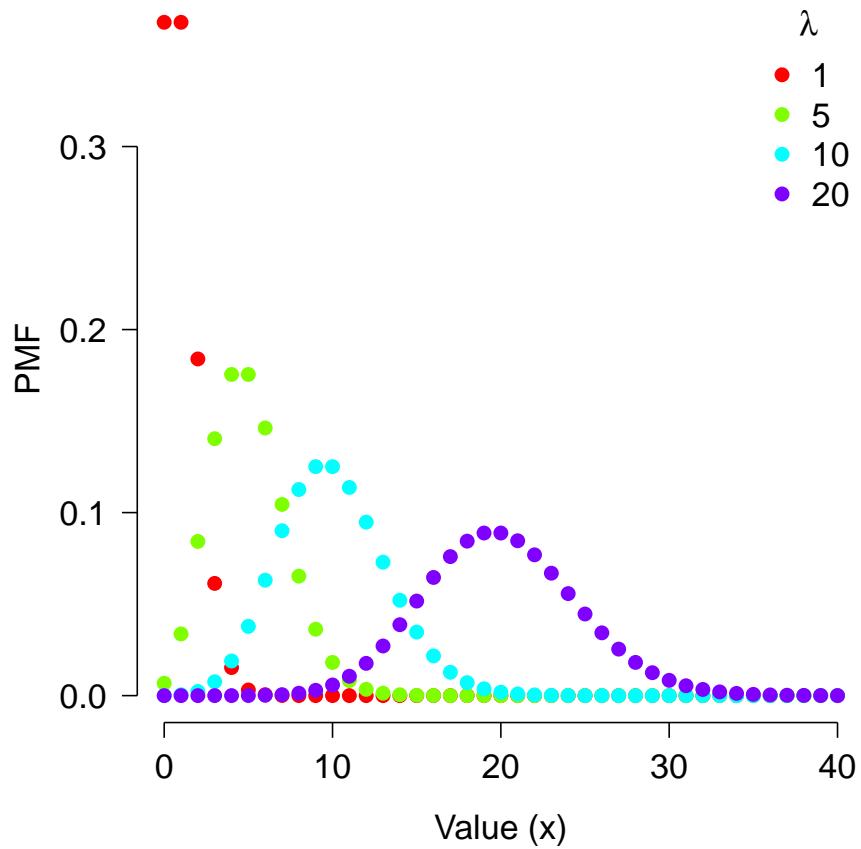
```
x <- 0:40
lams <- c(1,5, 10, 20)
nlams <- length(lams)
dlist <- vector("list", nlams)
for(i in 1:nlams){dlist[[i]] <- dpois(x, lams[i])}
cols <- rainbow(nlams)

par(mfrow=c(1,1), mar=c(5.1, 5.1, 1.1, 1.1),
    las=1, bty="n", lend=1,
    cex.lab=1.3, cex.axis=1.3)
plot(x, dlist[[1]], ylim=c(0, max(sapply(dlist, max))),
    type="n", ylab="PMF", xlab="Value (x)")
```

```

for(i in 1:nlams){
  points(x, dlist[[i]], pch=16, cex=1.3,
         col=cols[i])
}
legend("topright", legend=lams,
       pch=16, pt.cex=1.3, col=cols, bty="n", cex=1.3,
       title=expression(lambda))

```



4.3.3.4 Negative binomial distribution

The **negative binomial distribution** has two common definitions. The original definition is as the number of failures that occur in a series of Bernoulli trials until some predetermined number of successes is observed. For example, when flipping a fair coin, how many times should you expect to see tails before you observe 8 heads? The second definition is as an alternative to the Poisson

distribution when variance is not equal to the mean⁷. This makes the negative binomial a bit of an odd duck in biological data analysis because it is defined in terms of one distribution, but used as an alternate version of another. The traditional definition of the negative binomial distribution is what names the negative binomial.

Biologists often use the second definition as an **overdispersed** alternative to the Poisson distribution. Overdispersed means that a distribution has a variance greater than expected given other parameters. This is *very common* in biological count data. For example, a bird species might have low abundance (0 to 4) at most sites in a study, but very high abundance (30 to 40) at a handful of sites. In that case using the Poisson distribution would not be appropriate because doing so would imply that the variance λ was greater than the mean (also λ); in other words, that $\lambda > \lambda$.

The version of the negative binomial that biologists use is parameterized by its mean μ and its overdispersion k . This definition views the negative binomial as a Poisson distribution with parameter λ , where λ itself is a random variable that follows a Gamma distribution (see below). For this reason, some authors refer to the negative binomial as a **Gamma-Poisson mixture distribution**. A mixture distribution is exactly what it sounds like: a distribution that is formed by “mixing” or combining two distributions. Usually this manifests as having one or more parameters of one distribution vary as another distribution.

The overdispersion parameter k is called **size** in the R functions that work with the negative binomial. Counterintuitively, the overdispersion in a negative binomial distribution gets larger as k becomes smaller. This is seen in the expression for the variance of a negative binomial distribution:

$$\text{Var}(X) = \mu + \frac{\mu^2}{k}$$

As k becomes large, the ratio μ^2/k becomes small, and thus $\text{Var}(x)$ approaches μ . This means that a negative binomial distribution with large k approximates a Poisson distribution. As k approaches 0, the ratio μ^2/k becomes larger, and thus $\text{Var}(x)$ increases to be much larger than μ .

4.3.3.4.1 Negative binomial distribution in R The negative binomial distribution is accessed using the `_nbinom` group of functions, where the space could be `d`, `p`, `q`, or `r`. These functions calculate or returns something different:

- `dnbinom()`: Calculates PMF at x . Answers the question, “what is the probability of observing a count of x ? ”
- `pnbinom()`: Calculates CDF, or integral of PMF, from 0 up to x . In other words, given some negative binomial distribution, at what quantile of that distribution should some value fall? The reverse of `qnbinom()`.

⁷usually greater than the mean

- `qnbnom()`: Calculates the value at specified quantile of a Poisson distribution. The reverse of `pnbnom()`.
- `rnbnom()`: Draws random numbers from the negative binomial distribution.

The R functions for the negative binomial distribution can work with either parameterization (waiting time or mean with overdispersion). Some of the argument names are used for both methods. If you are working with the negative binomial distribution in R you *need to name your arguments to make sure you get the version of the negative binomial that you want*.

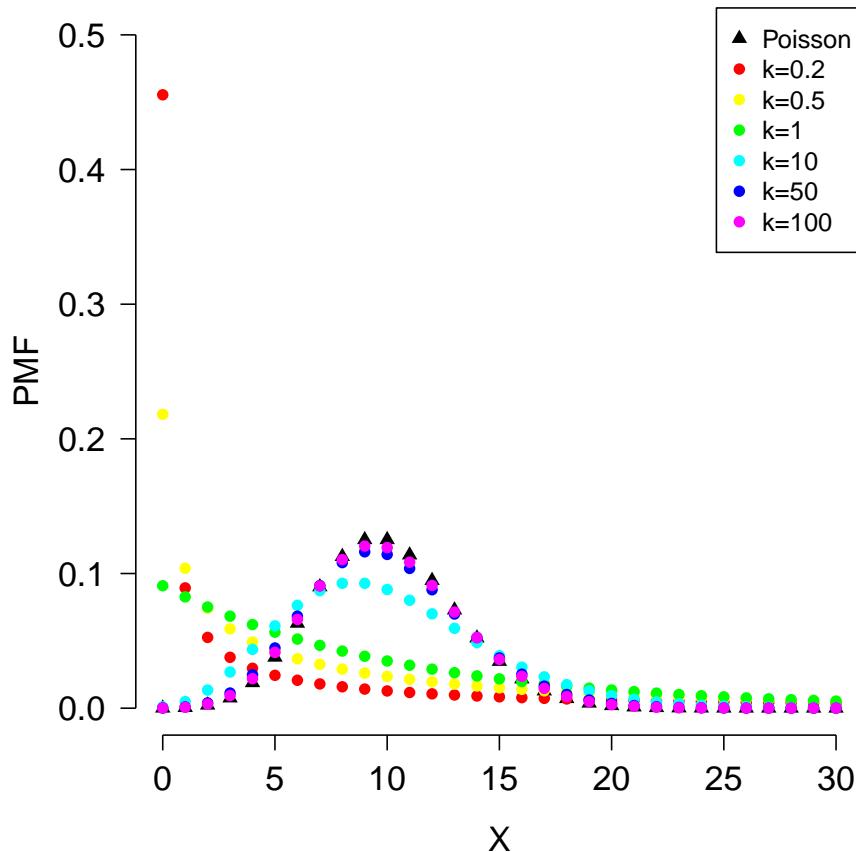
The figure below shows the effect of different overdispersion parameters. Notice that as k increases, the distribution looks more and more like a Poisson distribution with $\lambda = 10$. As k gets smaller, the distribution gets more and more concentrated near 0, and more and more right-skewed.

```

xp <- 0:30
y1 <- dpois(xp, 10)
y2 <- dnbinom(xp, size=0.2, mu=10)
y3 <- dnbinom(xp, size=0.5, mu=10)
y4 <- dnbinom(xp, size=1, mu=10)
y5 <- dnbinom(xp, size=10, mu=10)
y6 <- dnbinom(xp, size=50, mu=10)
y7 <- dnbinom(xp, size=100, mu=10)

cols <- rainbow(6)
par(mfrow=c(1,1), mar=c(5.1, 5.1, 1.1, 1.1),
    las=1, bty="n", lend=1,
    cex.lab=1.3, cex.axis=1.3)
plot(xp, y1, pch=17, ylim=c(0, 0.5), xlab="X", ylab="PMF")
points(xp, y2, pch=16, col=cols[1])
points(xp, y3, pch=16, col=cols[2])
points(xp, y4, pch=16, col=cols[3])
points(xp, y5, pch=16, col=cols[4])
points(xp, y6, pch=16, col=cols[5])
points(xp, y7, pch=16, col=cols[6])
legend("topright",
       legend=c("Poisson", "k=0.2", "k=0.5", "k=1", "k=10",
               "k=50", "k=100"),
       pch=c(17, rep(16,6)), col=c("black", cols))

```



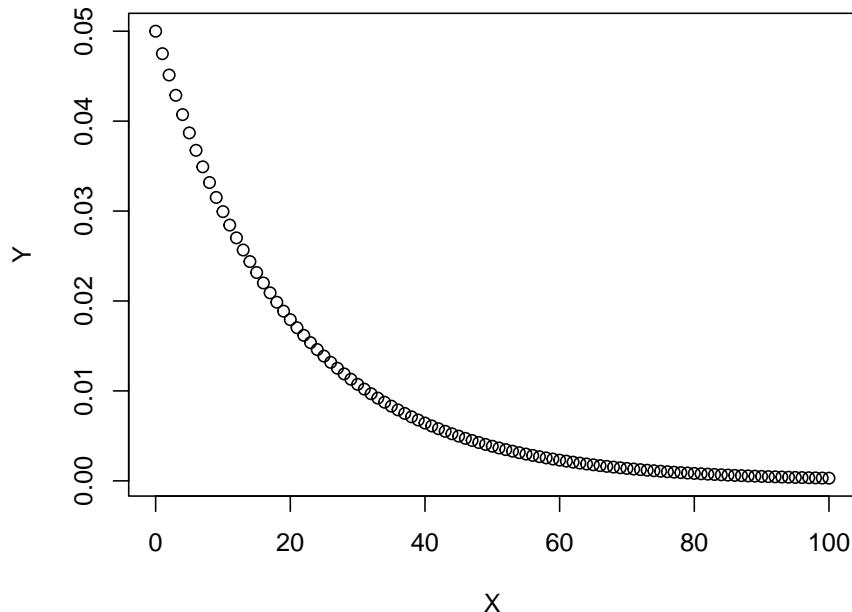
4.3.3.5 Geometric distribution

The **geometric distribution** describes the number of Bernoulli trials that are seen before observing a single failure. It is thus related to the binomial and negative binomial. In fact, the geometric distribution is a special case of the negative binomial, with the number of successes = 1.

The geometric distribution arises in biology when modeling waiting times or life spans. For example, the number of years that an individual organism lives, if it has a constant probability of surviving each year. Or, the number of generations until a mutation in a cell line occurs, if mutations occur at a constant rate per generation.

Imagine a fish that has a 5% chance of dying each year; i.e., a 95% annual survival rate. How long should we expect this fish to live, on average?

```
P <- 0.05
X <- 0:100
Y <- dgeom(X, P)
plot(X, Y)
```



The figure shows that the probability of living to any given age falls off quickly at first, then more gradually as time goes on. What is the mean life expectancy?

```
qgeom(0.5, P)
```

```
## [1] 13
```

What range of life expectancies can we expect for 95% of the population?

```
qgeom(c(0.025, 0.975), P)
```

```
## [1] 0 71
```

That's quite a wide range! What is the probability that a fish lives to be at least 30 years old? 40 years old?

```
1-pgeom(30, P)
```

```
## [1] 0.2039068
1-pgeom(40, P)
## [1] 0.1220865
```

4.3.3.6 Beta-binomial distribution

The **beta-binomial distribution** is a mixture that allows for modeling binomial processes with more variation than would be expected from a garden variety binomial distribution. The beta-binomial is a binomial distribution, but with the probability parameter p itself varying randomly according to a beta distribution (see below). This is similar to how a negative binomial distribution can be defined as a Poisson distribution mixed with a gamma distribution (with the Poisson λ varying as a gamma variable).

A beta-binomial distribution X is given by:

$$X \sim \text{Bin}(n, p)$$

$$p \sim \text{Beta}(\alpha, \beta)$$

The parameters of the beta distribution are positive shape parameters (see below). The parameters of the beta part of the beta-binomial distribution can make the binomial part overdispersed relative to an ordinary binomial distribution.

4.3.3.7 Multinomial distribution

The **multinomial distribution** is a generalization of the binomial distribution to cases with more than 2 possible outcomes. For example, rolling a 6-sided die many times would yield a distribution of outcomes (1, 2, 3, 4, 5, or 6) described by the multinomial distribution. The multinomial distribution is parameterized by the number of possible outcomes k , the number of trials n , and the probabilities of each outcome p_1, \dots, p_k . The probabilities must sum to 1.

There are some special cases of the multinomial distribution:

- When $k = 2$ and $n = 1$, the multinomial distribution reduces to the Bernoulli distribution
- When $k = 2$ and $n > 1$, the multinomial distribution reduces to the binomial distribution
- When $k > 2$ and $n = 1$, the multinomial distribution is the categorical distribution.

The example below shows how to work with the multinomial distribution in R.

```

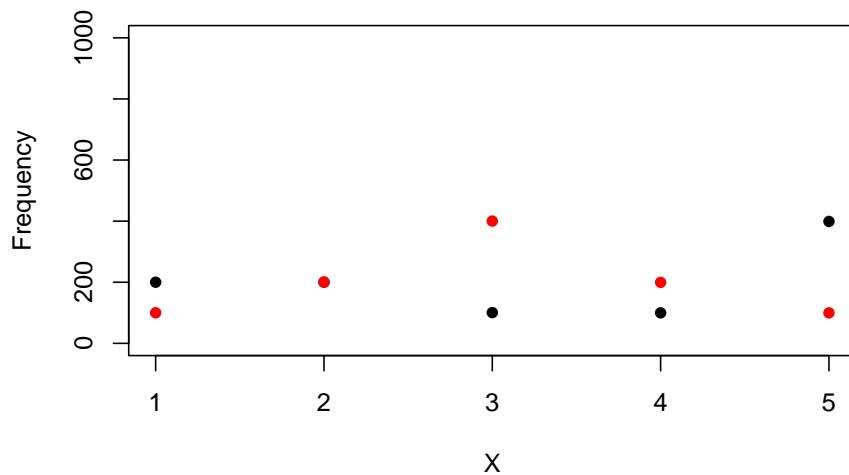
# define 2 different sets of probabilities for
# 5 different outcomes
p1 <- c(0.2, 0.2, 0.1, 0.1, 0.4)
p2 <- c(0.1, 0.2, 0.4, 0.2, 0.1)

# sample from the multinomial distribution
N <- 1e3
r1 <- rmultinom(N, N, prob=p1)
r2 <- rmultinom(N, N, prob=p2)

# summarize by outcome
y1 <- apply(r1, 1, mean)
y2 <- apply(r2, 1, mean)

# plot the results
plot(1:5, y1, pch=16, xlab="X",
      ylab="Frequency", ylim=c(0, 1000))
points(1:5, y2, pch=16, col="red")

```



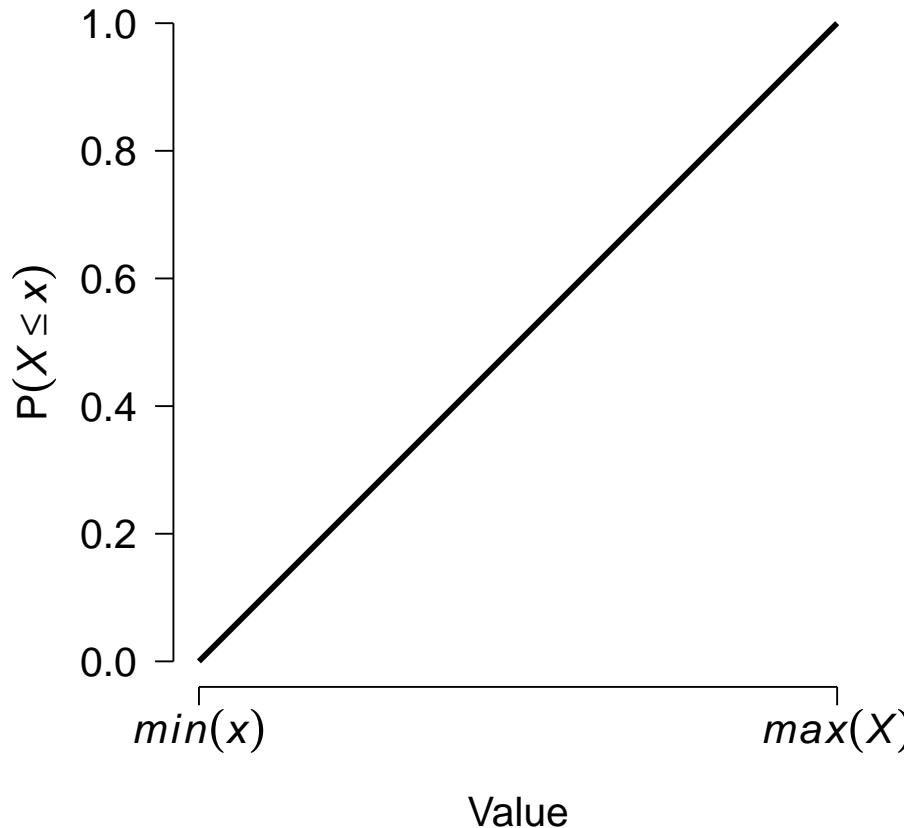
4.3.4 Continuous distributions

A **continuous distribution** can take an infinite number of values. These values can come from an interval on the real line, or the entire real line. Contrast this to a discrete distribution, which can only take on integer values. Computing

the probability of any possible outcome of a discrete distribution is relatively straightforward because there are a limited number of outcomes. For example, a binomial distribution with $n = 10$ and $p = 0.5$ has only 11 possible outcomes (0, 1, 2, ..., 9, or 10). The probabilities $p(0), p(1), \dots, p(10)$ must all sum to 1. The probability mass function (PMF) of the binomial distribution at any value is the probability of that value occurring.

This leads to an interesting question. How could we calculate the probability of any given outcome of a continuous distribution? If the probabilities of each individual value must all sum to 1, and there are infinitely many possible values, then doesn't the probability of each value equal 0? If the probability of every outcome is 0, how can the distribution take on any value?

To resolve this apparent contradiction, we have to take a step back and think about what distributions really mean. The first step is to realize that even if any single value has probability 0, an *interval* of values can have a non-zero probability. After all, the entire domain of a distribution has probability 1 (by definition). We can define some interval in the domain that covers half of the total probability, or one-quarter, or any arbitrary fraction. This means that for any value x in a distribution X (note lower case vs. upper case) we can calculate the probability of observing a value $\leq x$. The figure below shows this:



The y -axis in the figure, $P(X \leq x)$, is called the **cumulative distribution function (CDF)**. This name derives from the fact that the CDF of some value x gives the cumulative probability of all values up to and including x . The CDF is sometimes labeled “ $\mathbf{F}(x)$ ” (note the capitalized F). Another way to interpret this is that a value x lies at the $CDF(x)$ quantile of a distribution. For example, if $CDF(x) = 0.4$, then x lies at the 0.4 quantile or 40th percentile of the distribution. Critically, this means that 40% of the total probability of the distribution lies at or to the left of x .

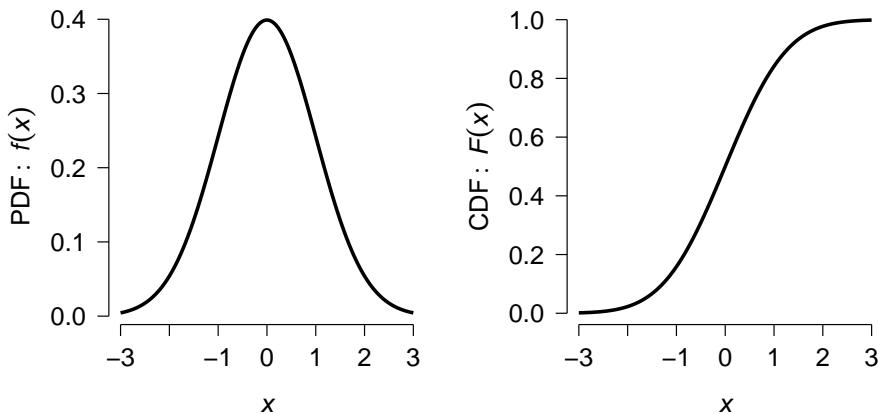
Just as with discrete distributions, the relative probability of any given value in a continuous distribution is given by the rate at which the CDF changes at that value. For discrete distributions this relative probability also happens to be the absolute probability (because $dx = 1$ in the expression below). For continuous distributions, this probability is relative. The relative probability of a value in a continuous distribution is given by the **probability density function (PDF)**. The PDF, $f(x)$, is the derivative of the CDF:

$$f(x) = \frac{dF(x)}{dx}$$

This also means that the CDF is the integral of the PDF, according to the fundamental theorem of calculus:

$$F(x) = \int_{-\infty}^x f(x) dx$$

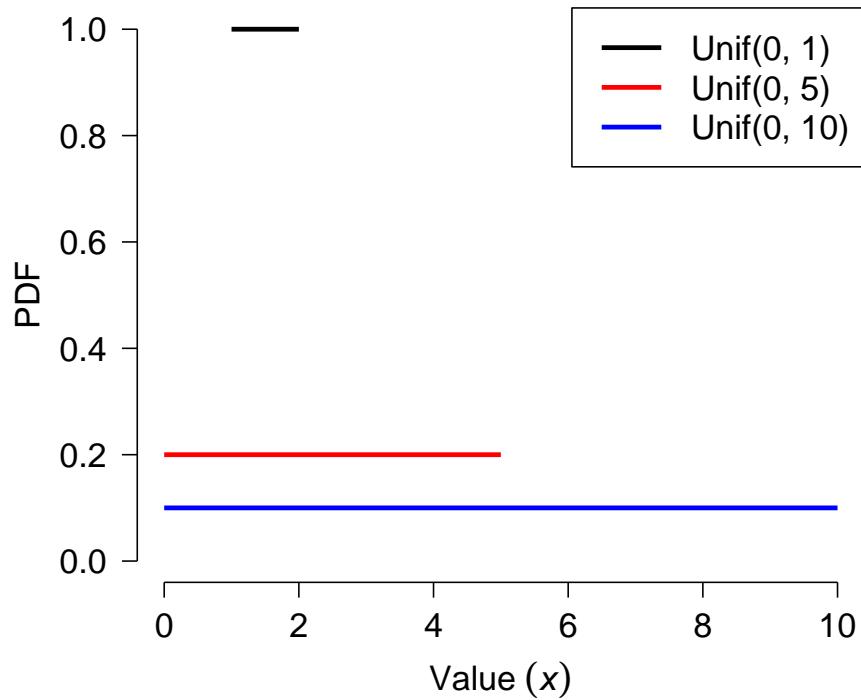
This might be easier to see with an example. The plots below show the PDF and CDF of a standard normal distribution (see below). Because the normal is unbounded, its PDF and CDF have the domain $[-\infty, +\infty]$, but we usually truncate the plot to a range that covers most of the CDF.



On the left, notice that the PDF peaks near $x = 0$ and tapers off quickly to either side. On the right, the slope of the CDF is greatest at $x = 0$. The value of the PDF at $x = 0$, about 0.4, is neither the probability of observing $x = 0$ nor the probability of observing $x \leq 0$, which is 0.5. The value $f(0) = 0.4$ is the rate of change in $F(X)$ at $x = 0$.

4.3.4.1 Uniform distribution

The simplest continuous distribution is the **uniform distribution**. The uniform distribution is parameterized by its upper and lower bounds, usually called a and b , respectively. All values of a uniform distribution in the interval $[a, b]$ are equally likely, and all values outside $[a, b]$ have probability 0. The figure below shows the PDF of 3 different uniform distributions.



Notice how each distribution has a flat PDF, but that the value of the PDF decreases as the width of the uniform distribution increases. Given what you learned above about the relationship between the CDF and the PDF, can you work out why this is?

Like the normal distribution, there is a standard uniform distribution that is frequently used. The standard uniform is defined as *Uniform*(0, 1): a uniform distribution in the interval [0, 1]. The uniform distribution can sometimes be abbreviated as *U(a,b)* or *Unif(a,b)* instead of *Uniform(a,b)*.

The mean of a uniform distribution is just the mean of its limits:

$$\mu(X) = \frac{a+b}{2}$$

The variance of a uniform distribution is:

$$\sigma^2(X) = \frac{(b-a)^2}{12}$$

4.3.4.1.1 Uniform distribution in R The uniform distribution is accessed using the `_unif` group of functions, where the space could be `d`, `p`, `q`, or `r`. These

functions calculate or returns something different:

- **dunif()**: Calculates probability density function (PDF) at x .
- **punif()**: Calculates CDF, from a up to x . Answers the question, “at what quantile of the distribution should some value fall?”. The reverse of **qunif()**.
- **qunif()**: Calculates the value at a specified quantile or quantiles. The reverse of **punif()**.
- **rufunif()**: Draws random numbers from the uniform distribution.

The help files for many R functions include a call to **runif()** to generate example values from the uniform distribution. This can be confusing to new users, who might interpret “runif” as “run if” (i.e., some sort of conditional statement).

One extremely useful application is the generation of values from the standard uniform distribution:

```
x <- runif(20)
```

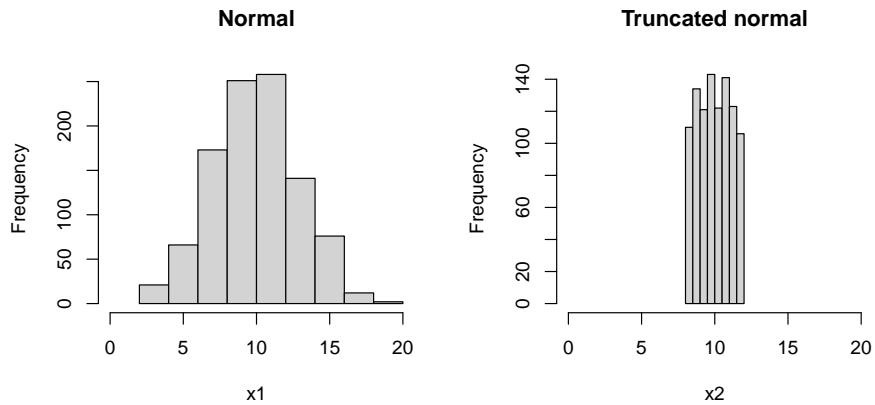
The line above generates 20 numbers from the interval $[0, 1]$. These values can be used as probabilities or values of a CDF.

If you want to implement a new probability distribution in R, you can use **runif()** followed by the **q_()** function that you define for your new distribution to implement a random number generator for it. The example below shows how to implement a “truncated normal” distribution:

```
rtnorm <- function(N, lower, upper, mu, sigma){
  a <- pnorm(lower, mu, sigma)
  b <- pnorm(upper, mu, sigma)
  x <- runif(N, a, b)
  y <- qnorm(x, mu, sigma)
  return(y)
}

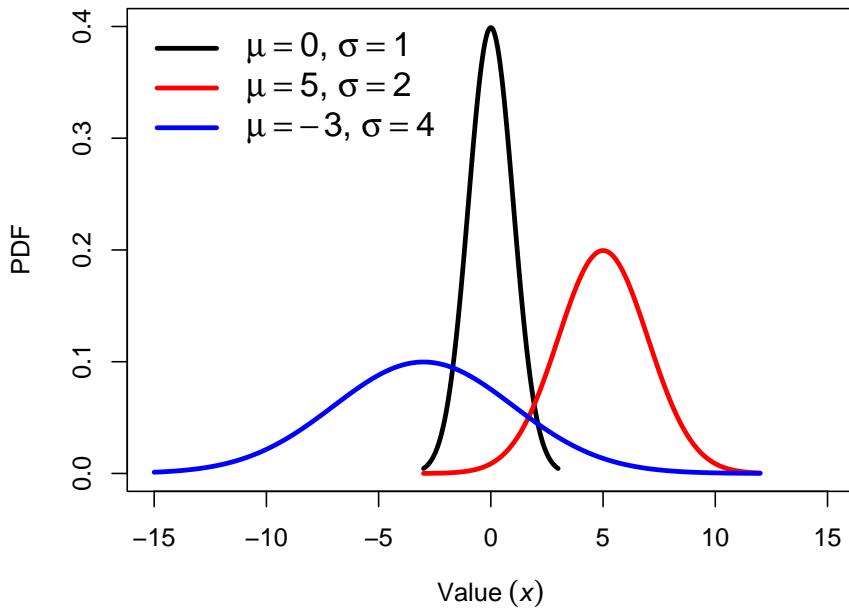
use.n <- 1e3
use.mu <- 10
use.sd <- 3
x1 <- rnorm(use.n, use.mu, use.sd)
x2 <- rtnorm(use.n, 8, 12, use.mu, use.sd)

par(mfrow=c(1,2))
hist(x1, main="Normal", xlim=c(0, 20))
hist(x2, main="Truncated normal", xlim=c(0, 20))
```



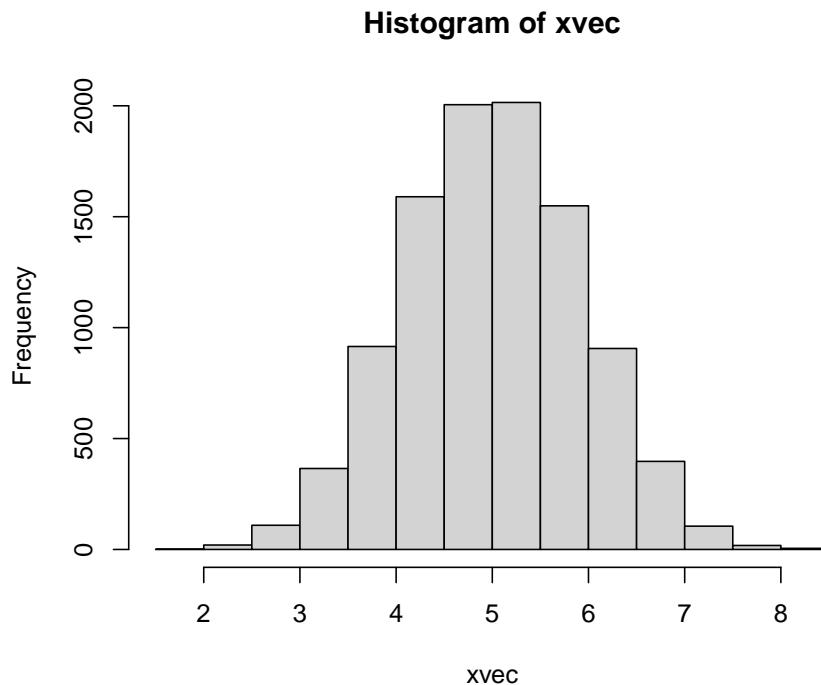
4.3.4.2 Normal distribution

The **normal distribution** is probably the most important distribution in statistics. Many natural processes result in normal distributions. Consequently, most classical statistical methods (e.g., *t*-tests, ANOVA, and linear models) assume that data come from a normal distribution. The normal distribution is also sometimes called the **Gaussian** distribution because of the work of mathematician Carl Friedrich Gauss, although he did not discover it. The normal distribution is also sometimes called the bell curve because of the shape of its PDF. This term should be avoided because other probability distributions also have a bell shape, and because normal distributions are not always bell-shaped. The figure below shows several normal distributions with different means (μ) and standard deviations (SD, or σ).



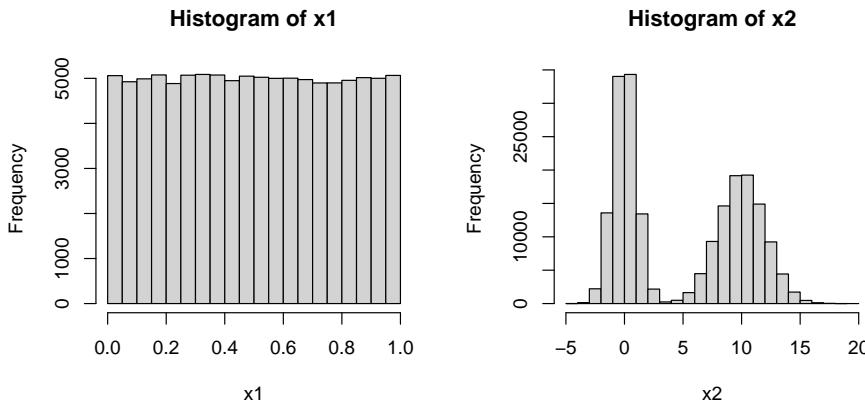
Many phenomena result in normal distributions because of the **Central Limit Theorem (CLT)**. The CLT states that under certain conditions the sum of many random variables will approximate a normal distribution. The main condition is that the random variables will be independent and identically distributed (often abbreviated “*i.i.d.*”). In other words, the values do not depend on each other, and they all come from the same kind of distribution. The exact distribution does not matter. The example below uses the uniform distribution. The CLT also works with the mean of the random variables instead of the sum, because mean is just the sum scaled by sample size.

```
N <- 1e4
xvec <- numeric(N)
for(i in 1:N){xvec[i] <- sum(runif(10))}
hist(xvec)
```



It should be noted that just because a sample size is large does not mean that it is normal. For example, the R commands below produce distributions `x1` and `x2` that are in no way normal.

```
par(mfrow=c(1,2))
x1 <- runif(1e5)
hist(x1)
x2 <- c(rnorm(1e5), rnorm(1e5, 10, 2))
hist(x2)
```



The normal distribution is parameterized by its mean (μ , or “mu”) and its variance (σ^2 , “sigma squared”). Some people prefer to use the mean and standard deviation (σ , “sigma”). This is fine because, as the symbols imply, the SD is simply the square root of the variance. Just pay attention to the notation being used. Using σ instead of σ^2 can be convenient because σ is in the same units as μ .

The **standard normal distribution** has $\mu = 0$ and $\sigma = 1$. This is often written as $N(0,1)$. These parameters are the defaults in the R functions that work with the normal distribution.

4.3.4.2.1 Normal distribution in R The normal distribution is accessed using the `_norm` group of functions, where the space could be `d`, `p`, `q`, or `r`. These functions calculate or returns something different:

- `dnorm()`: Calculates probability density function (PDF) at x .
- `pnorm()`: Calculates CDF, from $-\infty$ up to x . Answers the question, “at what quantile of the distribution should some value fall?” The reverse of `qnorm()`.
- `qnorm()`: Calculates the value at a specified quantile or quantiles. The reverse of `pnorm()`.
- `rnorm()`: Draws random numbers from the normal distribution.

The normal distribution in R is parameterized by the mean (argument `mean`) and the SD, not the variance (argument `sd`). The figure generated below shows the effect of increasing the SD on the shape of a distribution. Greater variance (i.e., greater SD) means that the distribution is more spread out.

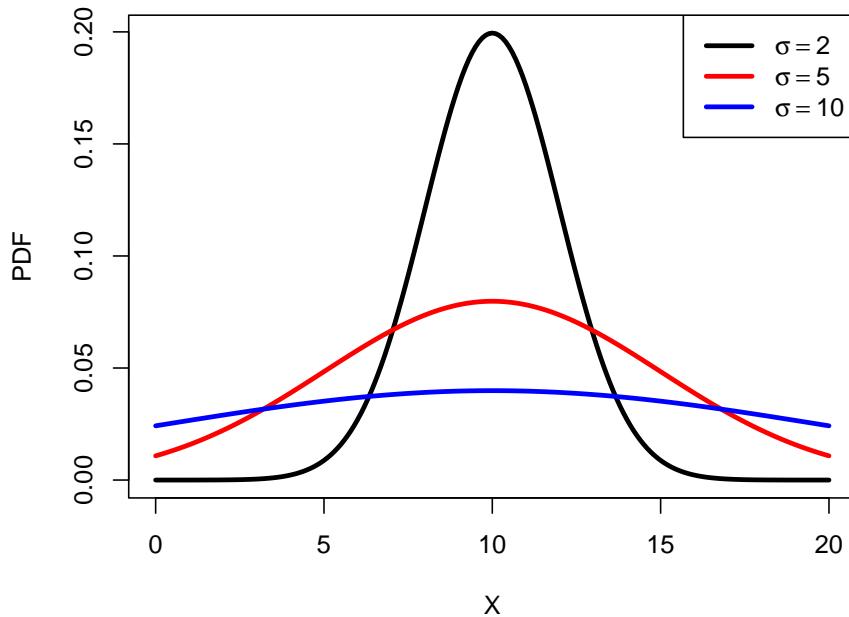
```
mu <- 10
x <- seq(0, 20, by=0.1)
y1 <- dnorm(x, mu, 2)
y2 <- dnorm(x, mu, 5)
```

```

y3 <- dnorm(x, mu, 10)

par(mfrow=c(1,1))
plot(x, y1, type="l", lwd=3, xlab="X", ylab="PDF")
points(x, y2, type="l", lwd=3, col="red")
points(x, y3, type="l", lwd=3, col="blue")
legend("topright", legend=c(expression(sigma==2),
    expression(sigma==5), expression(sigma==10)),
    lwd=3, col=c("black", "red", "blue"))

```



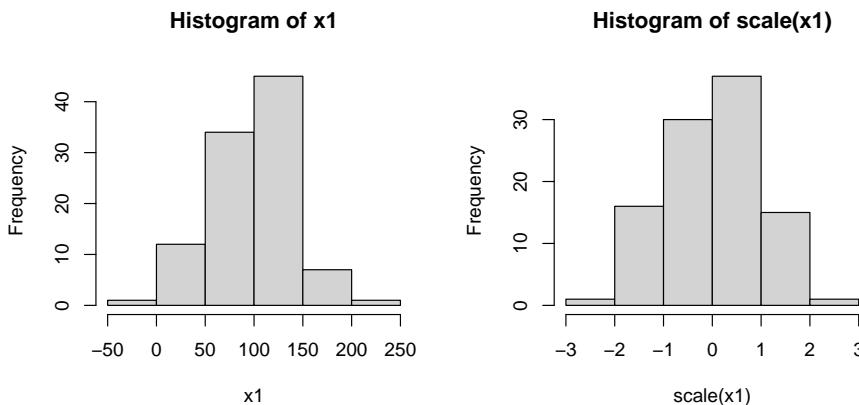
The normal distribution has several useful properties. Generally, the mean and median are the same (if the distribution is not skewed). Approximately 68% of values fall within 1 SD of the mean, 95% of values fall within about 2 SD of the mean (technically 1.9599 SD, or `qnorm(0.975)`), and 99% of values fall within about 3 SD of the mean. So, deviations of >3 SD are very rare and may signify outliers. The number of SD a value falls away from the mean is sometimes referred to as a ***z-score*** or a ***sigma***. Z-scores are calculated as:

$$z(x) = \frac{x - \mu}{\sigma}$$

Where x is the value, μ is the mean, and σ is the SD.

Z -scores are sometimes used when comparing variables that are normally distributed, but on very different scales. Converting the raw values to their z -scores is referred to as **standardizing** them. Standardized values are sometimes used in regression analyses in place of raw values. Many ordination methods such as principal components analysis (PCA) standardize variables automatically. Z -scores (or standardized values) are calculated in R using the `scale()` function:

```
x1 <- rnorm(100, 100, 40)
par(mfrow=c(1,2))
hist(x1)
hist(scale(x1))
```



4.3.4.3 Lognormal distribution

The **lognormal distribution** (a.k.a.: **log-normal**) is, as the name implies, a distribution that is *normal on a logarithmic scale*. By convention the **natural log** (\log_e , \ln , or just \log) is used although you can use \log_{10} if you pay attention to what you are doing and are explicit in your write-up⁸. The functions in R that deal with the lognormal use the natural log⁹.

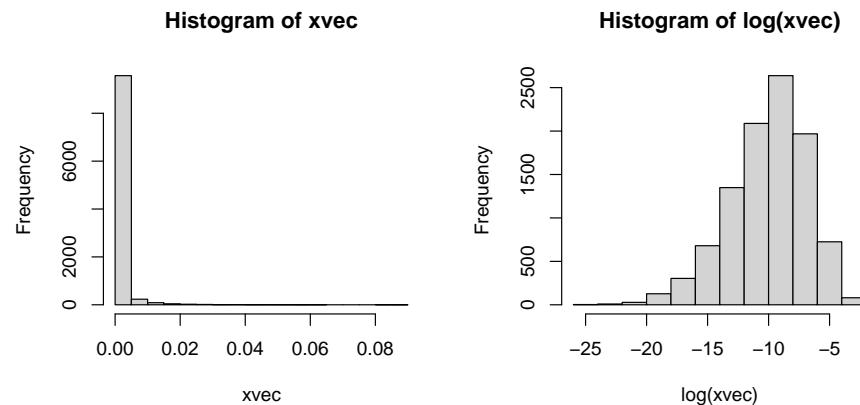
The lognormal distribution arises in two ways. First, it comes about from processes where quantities are *multiplied together rather than added*. For example, growth of organisms or population sizes. This is because of the way that exponentiation relates multiplication and addition. This relationship to multiplication also means that the lognormal distribution results from a version

⁸It is possible, and not uncommon, to analyze data using the natural log scale but plot results on the \log_{10} scale. This works because the choice of base is arbitrary for most analyses and because it is simple to convert values between scales with different bases (i.e., change of base).

⁹In R, statistics, and in mathematics in general, if you see “log” it usually means “natural log”.

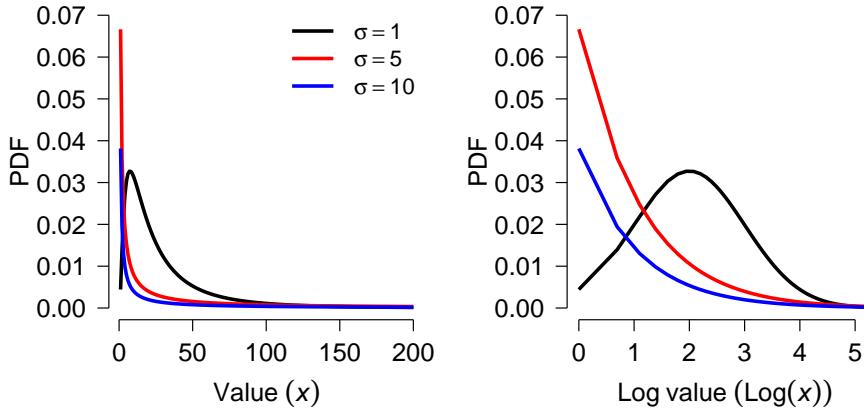
of the CLT where values are multiplied, not added. The example below shows this.

```
N <- 1e4
xvec <- numeric(N)
for(i in 1:N){xvec[i] <- prod(runif(10))}
par(mfrow=c(1,2))
hist(xvec)
hist(log(xvec))
```



The second way that the lognormal arises is more phenomenological (i.e., descriptive). Processes that result in positive values with a long tail (mostly small values with few very large values) or positive values with a variance much larger than the mean. The latter case can also be well described by the gamma distribution. The classic example of a lognormal distribution is personal income: most populations will have mostly small values, but with a small proportion of extreme outliers.

The lognormal distribution is parameterized by a mean (μ) and variance (σ^2), or by a mean and standard deviation (σ). These parameters are on the logarithmic scale. The figure below shows various lognormal distributions with $\mu = 3$ and different SD on a logarithmic scale (left) and a linear scale.



The figures show the effect that variance can have on the lognormal distribution. When the variance on the log scale is $<\mu$, the distribution is approximately normal on the log scale. As the variance gets larger, the distribution gets more and more right-skewed. Right-skewed means that the values are more and more concentrated near 0, and there are fewer very large values.

Interestingly, the mean of a log-normal distribution is not simply e^μ as one might expect. The mean of a lognormal distribution X on the linear scale is

$$\text{mean}(X) = e^{\left(\frac{\mu+\sigma^2}{2}\right)}$$

In this equation, μ and σ are the mean and SD on the logarithmic scale. The variance on the linear scale is:

$$\text{Var}(X) = e^{2\mu+\sigma^2} \left(e^{\sigma^2} - 1 \right)$$

What this means is that you cannot transform the parameters of a lognormal distribution from the logarithmic to the linear scale by simply exponentiating.

4.3.4.3.1 Lognormal distribution in R The lognormal distribution is accessed using the `lnorm` group of functions, where the space could be `d`, `p`, `q`, or `r`. These functions calculate or returns something different:

- `dlnorm()`: Calculates probability density function (PDF) at x .
- `plnorm()`: Calculates CDF, from 0 up to x . Answers the question, “at what quantile of the distribution should some value fall?”. The reverse of `qlnorm()`.
- `qlnorm()`: Calculates the value at a specified quantile or quantiles. The reverse of `plnorm()`.
- `rlnorm()`: Draws random numbers from the lognormal distribution.

When working with the lognormal distribution in R it is important to keep in mind that the parameters (`meanlog` and `sdlog`) of the `rlnorm` functions are on the *natural log scale*. The `meanlog` parameter is the *mean of the logarithm* of the values, not the mean of the values or the logarithm of the mean of the values. Similarly, `sdlog` is the SD of the *logarithm* of the values, not the SD of the values or the logarithm of the SD of the values. The example below shows this:

```
x1 <- rlnorm(1e4, 2, 1)

# mean of the values vs. mean of the log(values):
mean(x1)

## [1] 12.10312
mean(log(x1))

## [1] 2.01172
# sd of the values vs. sd of the log(values):
sd(x1)

## [1] 14.81899
sd(log(x1))

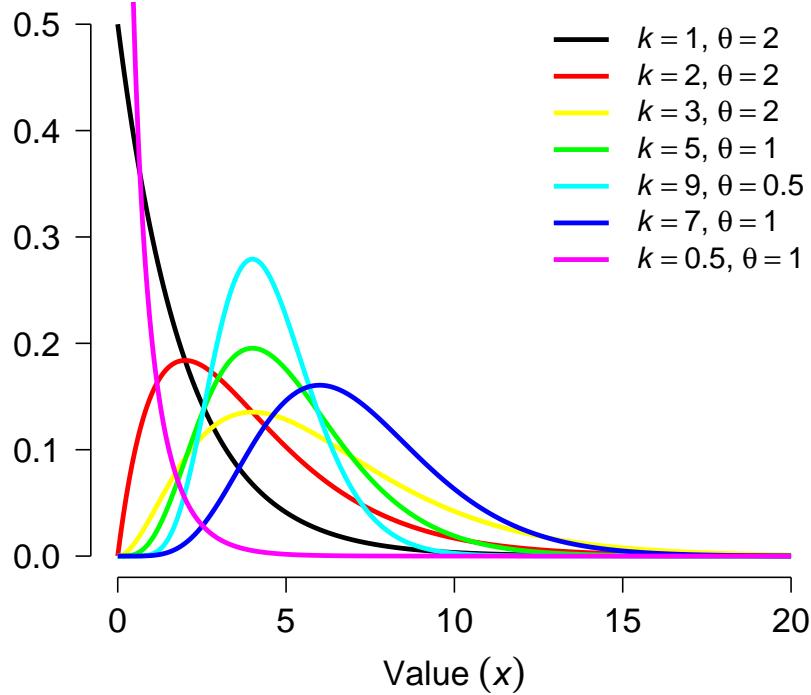
## [1] 0.9923424
# exp(meanlog) != mean(x1)
exp(2)

## [1] 7.389056
mean(x1)

## [1] 12.10312
```

4.3.4.4 Gamma distribution

The **gamma distribution** describes waiting times until a certain number of events takes place. This means it is the continuous analogue of the negative binomial distribution, which describes the number of trials until some certain number of successes. The gamma distribution can also be used to describe positive data whose SD is much larger than the mean. This makes it an alternative to the lognormal distribution for right-skewed data, much like the negative binomial is an alternative to the Poisson. Example gamma distributions are shown below.



The Gamma distribution is parameterized by its **shape** and **scale**. The shape parameter k describes the number of events; the scale parameter θ (“theta”) describes the mean time until each event. The scale is sometimes expressed as the **rate**, which is the reciprocal of scale. Both versions are implemented in R, so you need to specify your arguments when working with the gamma distribution. For example, $\text{Gamma}(4, 2)$ is the distribution of length of time in days it would take to observe 4 events if events occur on average once every two days. Equivalently, it is the time to observe 4 events if events occur at a rate of 1/2 event per day. Consequently, the mean of a gamma distribution given shape k and scale θ (or rate r) is:

$$\mu = k\theta = \frac{k}{r}$$

The variance is calculated as:

$$\sigma^2 = k\theta^2 = \frac{k}{r^2}$$

Like the negative binomial, the gamma distribution is often used phenomenologically; that is, used to describe a variable even if its mechanistic description

(waiting times) doesn't make sense biologically. Just as the negative binomial can be used to model count data with variance greater than the mean (i.e., an overdispersed Poisson), a gamma distribution can be used for positive data with $\sigma \gg \mu$. I.e., the use case for the gamma is similar to that of the lognormal.

4.3.4.4.1 Gamma distribution in R The gamma distribution is accessed using the `_gamma` group of functions, where the space could be `d`, `p`, `q`, or `r`. These functions calculate or returns something different:

- `dgamma()`: Calculates probability density function (PDF) at x .
- `pgamma()`: Calculates CDF, from 0 up to x . Answers the question, "at what quantile of the distribution should some value fall?". The reverse of `qgamma()`.
- `qgamma()`: Calculates the value at a specified quantile or quantiles. The reverse of `pgamma()`.
- `rgamma()`: Draws random numbers from the gamma distribution.

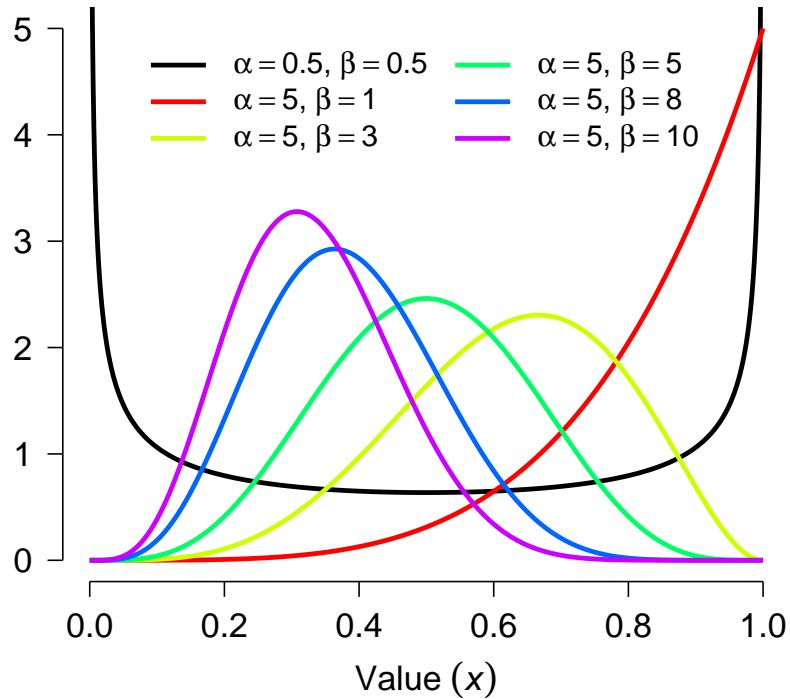
Note that when working with the gamma distribution in R, you must supply either the `shape` and the `rate`, or the `shape` and the `scale`. Supplying `scale` and `rate` will return an error.

```
# will work:  
rgamma(10, shape=3, scale=2)  
rgamma(10, shape=3, rate=0.5)  
  
# will return error:  
rgamma(10, scale=2, rate=0.5)
```

4.3.4.5 Beta distribution

The **beta distribution** is defined on the interval $[0, 1]$ and is parameterized by two positive shape parameters, α and β . For this reason, the beta distribution is often used to model the distribution of probabilities. Besides the uniform, the beta distribution is probably the most well-known continuous distribution that is bounded by an interval. The beta distribution can also be used to model any continuous variable that occurs on a bounded interval, by rescaling the variable to the interval $[0, 1]$. However, this may be better accomplished with a logit transformation (see the section on transformations).

The parameters of the beta distribution are a little tricky to understand, and relate to the binomial distribution. Each shape parameter is one plus the number of successes (α) and failures (β) in a binomial trial. This gets a little weird when α and β are <1 (how can you have <0 successes or failures?) but the distribution is still defined. The shape parameters can also be non-integers. The figure below shows various beta distributions. Notice how the distribution becomes U-shaped when α and β both get close to 0.



The beta distribution has mean

$$\mu = \frac{\alpha}{\alpha + \beta}$$

and variance

$$\sigma^2 = \frac{\alpha\beta}{(\alpha + \beta)^2 (\alpha + \beta + 1)}$$

These expressions can be re-arranged to solve for α and β given a mean and variance:

$$\alpha = \left(\frac{1 - \mu}{\sigma^2} - \frac{1}{\mu} \right) \mu^2$$

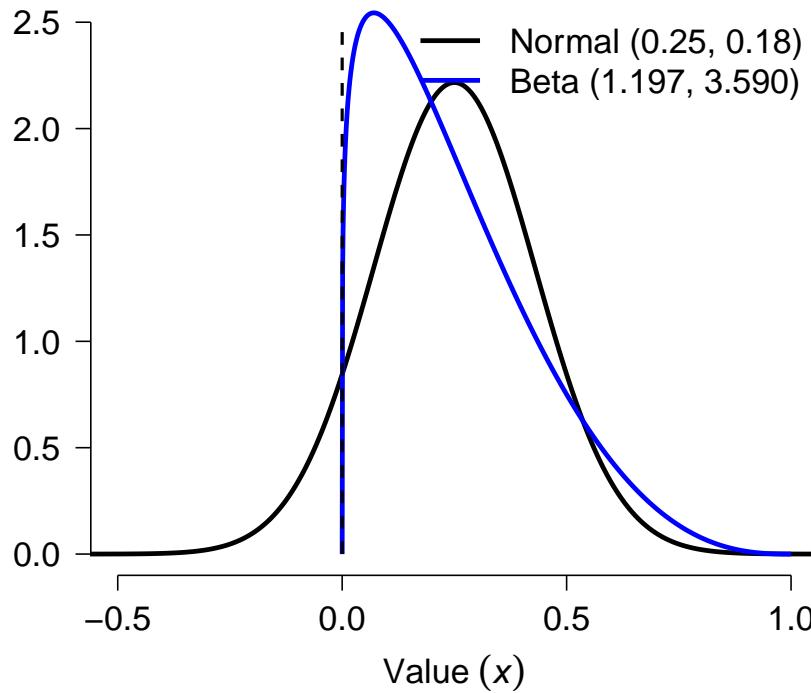
$$\beta = \alpha \left(\frac{1}{\mu} - 1 \right)$$

This is useful if you want to use data that were reported as a mean and variance (or SD, or SE) that describe a proportion. For example, if a paper reports a survival rate estimate as 0.25 ± 0.18 SE, this implicitly suggests that the survival rate is a normally-distributed variable because a normal distribution is defined by the mean and SD (SE of a parameter estimate is equivalent to the SD of a distribution). However, this would imply that about 8% of the time, the survival rate was negative! (Verify this using R: `pnorm(0, 0.25, 0.18)`). So, you should convert these estimates to a beta distribution if you want to understand how they might vary:

$$\alpha = \left(\frac{1 - 0.25}{(0.18)^2} - \frac{1}{0.25} \right) 0.25^2 \approx 1.1968$$

$$\beta = 1.1968 \left(\frac{1}{0.25} - 1 \right) \approx 3.5903$$

The figure below shows the two distributions, the normal in black and the beta in blue. The dashed vertical line at 0 shows that part of the normal distribution implied by the mean and SE of a survival “probability” is not possible. Converting to a beta distribution preserves the mean and variance, but probably more accurately reflects the true nature of the uncertainty about the probability. Note that this conversion cannot be done for some combinations of μ and σ : both α and β must both be positive, so if you calculate a non-positive value for either parameter then the conversion won’t work. This can sometimes happen for probabilities very close to 0 or 1 with large SE.



4.3.4.5.1 Beta distribution in R The beta distribution is accessed using the `_beta` group of functions, where the space could be `d`, `p`, `q`, or `r`. These functions calculate or returns something different:

- `dbeta()`: Calculates probability density function (PDF) at x .
- `pbeta()`: Calculates CDF, from 0 up to x . Answers the question, “at what quantile of the distribution should some value fall?”. The reverse of `qbeta()`.
- `qbeta()`: Calculates the value at a specified quantile or quantiles. The reverse of `pbeta()`.
- `rbeta()`: Draws random numbers from the beta distribution.

4.3.4.5.2 The beta and the binomial The relationship between the beta distribution and the binomial distribution should be obvious: uncertainty about the probability parameter of a binomial distribution (p) can be modeled using a beta distribution. In fact, this practice is so common that it has a name: the beta-binomial distribution.

We can take advantage of the relationship between the binomial and beta distributions to make inferences from count data. Imagine a study where biologists track the probability that fruit flies die before they are 3 weeks old.

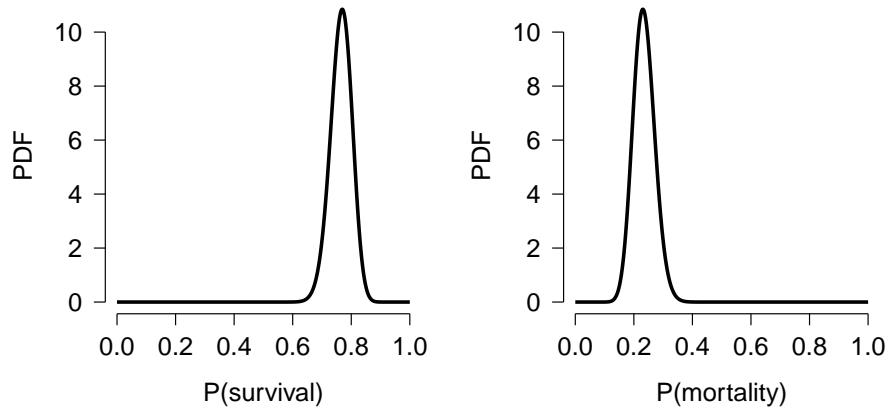
They report that 30 out of 130 flies die on or before day 21 of life. What can we estimate about the underlying distribution of 3-week survival rates?

First, express the values as *successes* and *failures*. Let's consider survival to be success ($n = 100$) and mortality to be failure ($n = 30$). This suggests that the underlying probability of survival to 21 days follows a beta distribution with $\alpha = 101$ and $\beta = 31$. Alternatively, we could define “success” as mortality. In this case, the mortality rate would follow a beta distribution with $\alpha = 31$ and $\beta = 101$. In R this can be calculated and plotted as shown below. Notice how reversing α and β , or reversing perspective from survival to mortality, changes the distribution to its complement:

```
# domain of probability
x <- seq(0, 1, length=1e3)

# survival rate distribution
y.surv <- dbeta(x, shape1=101, shape2=31)
y.mort <- dbeta(x, shape1=31, shape2=101)

par(mfrow=c(1,2), mar=c(5.1, 5.1, 1.1, 1.1),
    bty="n", lend=1, las=1,
    cex.axis=1.3, cex.lab=1.3)
plot(x, y.surv, type="l", lwd=3,
      xlab="P(survival)", ylab="PDF")
plot(x, y.mort, type="l", lwd=3,
      xlab="P(mortality)", ylab="PDF")
```



We could calculate this distribution in another way. Consider the number of flies that survive to come from a binomial distribution. The distribution would have $n = 130$, and $p = 100/130 \approx 0.7692$. Using the properties of the binomial distribution, we could infer that the mean of the binomial distribution was

$$\mu = np = 100$$

and its variance was

$$\sigma^2 = np(1-p) \approx 23.08$$

This would imply that the SD of the binomial distribution was about 4.8038, and thus the CV was about 0.048038. The CV is the ratio of the SD to the mean. We could then estimate the SD of p was

$$\sigma(p) = (0.048)(0.7692) \approx 0.0369$$

We could then use these values, $\mu(p) = 0.7692$ and $\sigma(p) = 0.0369$ into the expressions for α and β to obtain:

$$\alpha = \left(\frac{1 - 0.7692}{(0.0369)^2} - \frac{1}{0.7692} \right) (0.7692)^2 \approx 99.23$$

$$\beta = 99.52 \left(\frac{1}{0.7692} - 1 \right) \approx 29.77$$

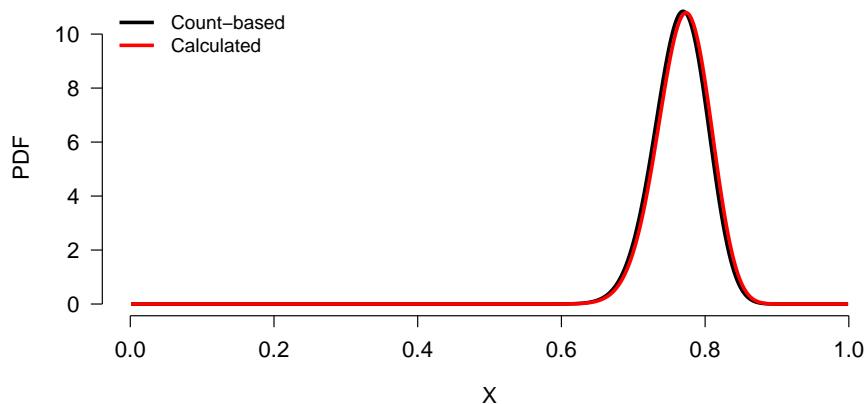
These expressions get us almost the exact answers. The R code to perform these calculations is below:

```
n <- 130
p <- 100/130
sigma2 <- n*p*(1-p)
sigma <- sqrt(sigma2)
cv <- sigma/100
sdp <- cv*p
Alpha <- (((1-p)/(sdp^2))-(1/p))*(p^2)
Beta <- Alpha*((1/p)-1)
```

We can compare the distributions using `dbeta`.

```
x1 <- 1:999/1e3
y1 <- dbeta(x1, 101, 31) # based on counts
y2 <- dbeta(x1, Alpha, Beta) # calculated from binomial

par(mfrow=c(1,1), bty="n", mar=c(5.1, 5.1, 1.1, 1.1),
    las=1, cex.axis=1.2, cex.lab=1.2, lend=1)
plot(x1, y1, type="l", lwd=3, xlab="X", ylab="PDF")
points(x1, y2, type="l", lwd=3, col="red")
legend("topleft", legend=c("Count-based", "Calculated"),
    lwd=3, col=c("black", "red"), bty="n")
```



The calculated values in `y2` result in a beta distribution of probabilities very close to the correct values. The difference results from a result known as the **rule of succession**, first published by French mathematician Pierre-Simon LaPlace in the 18th century. Briefly, we should assign the mean probability of success in a future Bernoulli trial \hat{p} as

$$\hat{p} = \frac{x + 1}{n + 2}$$

where x is the number of successes in n trials. In other words, for some vector x of successes and failures (1s and 0s) we need to add one additional 1 and one additional 0 in order to estimate underlying distribution of success rates. The R code below produces the correct result (notice the new values of n and p).

```

n <- 132
p <- 101/132
sigma2 <- n*p*(1-p)
sigma <- sqrt(sigma2)
cv <- sigma/100
sdp <- cv*p

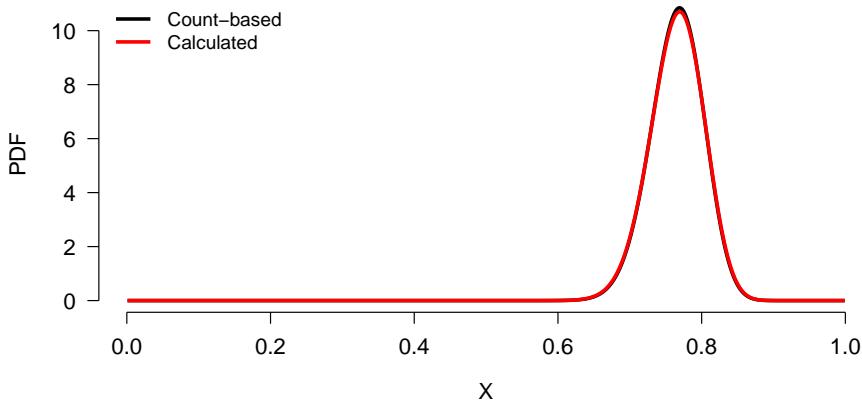
Alpha <- (((1-p)/(sdp^2))-(1/p))*(p^2)
Beta <- Alpha*((1/p)-1)

x1 <- 1:999/1e3
y1 <- dbeta(x1, 101, 31) # based on counts
y2 <- dbeta(x1, Alpha, Beta) # calculated from binomial
# and rule of succession

par(mfrow=c(1,1), bty="n", mar=c(5.1, 5.1, 1.1, 1.1),
  las=1, cex.axis=1.2, cex.lab=1.2, lend=1)

```

```
plot(x1, y1, type="l", lwd=3, xlab="X", ylab="PDF")
points(x1, y2, type="l", lwd=3, col="red")
legend("topleft", legend=c("Count-based", "Calculated"),
       lwd=3, col=c("black", "red"), bty="n")
```



4.3.4.6 Dirichlet distribution

The **Dirichlet distribution** (pronounced “Deer-eesh-lay”) is the distribution of probabilities for multiple outcomes of a random process. This makes it the generalization of beta distribution to multiple outcomes. The values of a Dirichlet distribution must add up to 1. For example, the beta distribution models the probability of throwing heads when you flip a coin. The Dirichlet distribution describes the probability of getting 1, 2, 3, 4, 5, or 6 when you throw a 6-sided die. Just as the probabilities of heads and not-heads must add to 1, the probabilities of 1, 2, 3, 4, 5, and 6 must add up to 1.

The Dirichlet distribution doesn’t come up on its own very often. Usually, biologists encounter the Dirichlet distribution when performing Bayesian analysis of multinomial outcomes. The Dirichlet distribution is the “conjugate prior” of the categorical and multinomial distributions.

4.3.4.7 Exponential distribution

The **exponential distribution** describes the distribution of waiting times until a single event occurs, given a constant probability per unit time of that event occurring. This makes it the continuous analog of the geometric distribution. The exponential distribution is also a special case of the gamma distribution where $k = 1$.

The exponential distribution should not be confused with the **exponential family of distributions**, although the exponential distribution is part of that

family. The exponential family of distributions includes many whose PDF can be expressed as an exponential function (e.g., $f(x) = e^x$). The exponential family includes the normal, exponential, gamma, beta, Poisson, and many others.

The exponential distribution is parameterized by a single rate parameter, λ , which describes the number of events occurring per unit time. This means that λ must be >0 . This is exactly the same as the rate parameter r sometimes used to describe the gamma distribution. The mean of an exponential distribution X is:

$$\mu(X) = \frac{1}{\lambda}$$

and the variance is:

$$\sigma^2(X) = \frac{1}{\lambda^2}$$

Interestingly, this implies that the standard deviation σ is the same as the mean μ . Contrast this with the Poisson distribution, where the *variance* σ^2 is the same as the mean. Thus, the CV of the exponential distribution is always 1.

The exponential distribution is supported for all non-negative real numbers; i.e., the half-open interval $[0, +\infty)$. Like the gamma distribution, the exponential distribution can be used to model biological processes for which its mechanistic description makes sense (e.g., waiting times or lifespans), or for any situation resulting in primarily 0 or small values and few large values.

4.3.4.7.1 Exponential distribution in R The exponential distribution is accessed using the `_exp` group of functions, where the space could be `d`, `p`, `q`, or `r`. These functions calculate or returns something different:

- `dexp()`: Calculates probability density function (PDF) at x .
- `pexp()`: Calculates CDF up to x . Answers the question, “at what quantile of the distribution should some value fall?”. The reverse of `qexp()`.
- `qexp()`: Calculates the value at a specified quantile or quantiles. The reverse of `pexp()`.
- `rexp()`: Draws random numbers from the exponential distribution.

4.3.4.8 Triangular distribution

The **triangular distribution** is sometimes called the “distribution of ignorance” or “lack of knowledge distribution”. Sometimes we need to model a process about which we have very little information. For example, we may want to simulate population dynamics without knowing a key survival rate, or organismal growth without knowing a key growth constant. In these situations we might have only a vague idea of how a parameter or an outcome are distributed. At a minimum,

we can usually infer or estimate the range and central tendency of a variable. Those are enough to estimate a triangular distribution.

The triangular distribution is parameterized by three parameters: the lower limit a , the upper limit b , and the mode (most common value) c . Any triangular distribution must satisfy $a < b$ and $a \leq c \leq b$.

The mean of a triangular distribution X is the mean of its parameters:

$$\mu(X) = \frac{a + b + c}{3}$$

and the variance is

$$\sigma^2(X) = \frac{a^2 + b^2 + c^2 - ab - ac - bc}{18}$$

In addition to serving as a stand-in distribution when data are scarce, the triangular distribution can arise in several natural situations. For example, the mean of two standard uniform variables follows a triangular distribution (see below).

4.3.4.8.1 Triangular distribution in R The triangular distribution is available in the add-on package “triangle” (Carnell 2019). The triangular distribution is accessed using the `_triangle` group of functions, where the space could be `d`, `p`, `q`, or `r`. These functions calculate or returns something different:

- `dtriangle()`: Calculates probability density function (PDF) at x .
- `ptriangle()`: Calculates CDF up to a to x . Answers the question, “at what quantile of the distribution should some value fall?”. The reverse of `qtriangle()`.
- `qtriangle()`: Calculates the value at a specified quantile or quantiles. The reverse of `ptriangle()`.
- `rtriangle()`: Draws random numbers from the triangular distribution.

The code below shows the PDFs of various triangular distributions. Notice that `c` is not used as a variable name, to avoid potentially masking the very critical R function `c`.

```
library(triangle)

## Warning: package 'triangle' was built under R version 4.1.2
ax <- c(1, 1, 3)
bx <- c(5, 10, 7)
cx <- c(3, 3, 6)

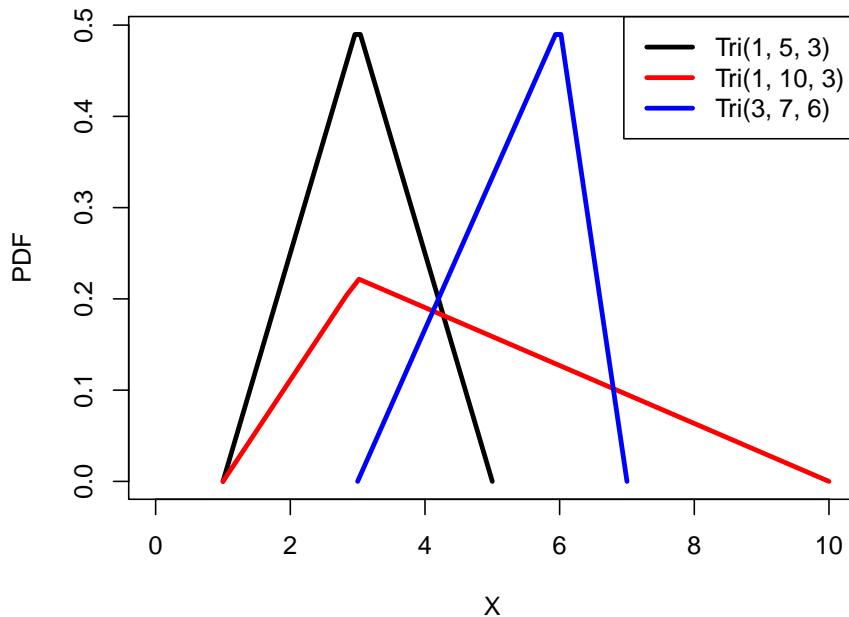
x1 <- seq(ax[1], bx[1], length=50)
x2 <- seq(ax[2], bx[2], length=50)
```

```

x3 <- seq(ax[3], bx[3], length=50)
y1 <- dtriangle(x1, ax[1], bx[1], cx[1])
y2 <- dtriangle(x2, ax[2], bx[2], cx[2])
y3 <- dtriangle(x3, ax[3], bx[3], cx[3])

plot(x1, y1, type="l", lwd=3, xlab="X", ylab="PDF",
      xlim=c(0, 10))
points(x2, y2, type="l", lwd=3, col="red")
points(x3, y3, type="l", lwd=3, col="blue")
legend("topright", legend=c("Tri(1, 5, 3)", "Tri(1, 10, 3)", "Tri(3, 7, 6)",
                           lwd=3, col=c("black", "red", "blue")))

```



The code below demonstrates how a triangular distribution can arise as the distribution of means of two standard uniform variables x_1 and x_2 . The command `density` estimates the kernel density estimate of a vector. The kernel density estimate is an empirical estimate of the PDF of a variable.

```

set.seed(42)
n <- 10^(2:4)
x1 <- y <- x <- vector("list", length(n))

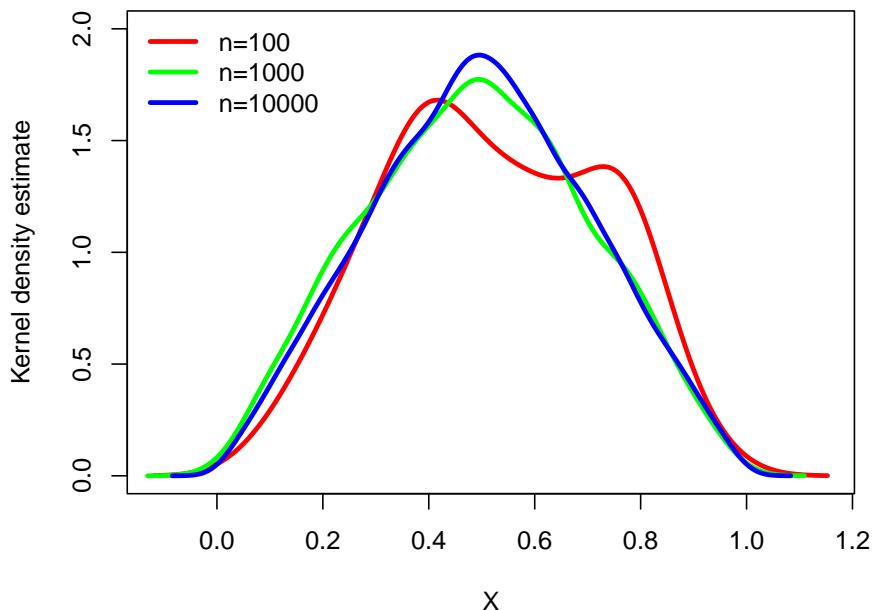
```

```

for(i in 1:length(n)){
  x[[i]] <- (runif(n[i]) + runif(n[i]))/2
}
for(i in 1:length(n)){
  z <- density(x[[i]])
  x1[[i]] <- z$x
  y[[i]] <- z$y
}

cols <- rainbow(3)
plot(x1[[1]], y[[1]], type="l", lwd=3, col=cols[1],
      xlab="X", ylab="Kernel density estimate", ylim=c(0, 2))
for(i in 2:3){
  points(x1[[i]], y[[i]], type="l", lwd=3, col=cols[i])
}
legend("topleft", legend=c("n=100", "n=1000", "n=10000"),
      bty="n", lwd=3, col=cols[1:3])

```



4.3.5 Distributions summary

The table below summarizes some the key features of the **discrete distributions** we explored in this module.

Distribution	Support	What it models
Bernoulli	0 or 1	Outcome of a single binary trial
Binomial	$[0, n]$	Number of successes in n trials with constant probability p .
Poisson	$[0, +\infty]$	Number of events occurring independently at constant rate (i.e., count data)
Negative binomial	$[0, +\infty]$	Number of failures until a success; or, counts with overdispersion
Geometric	$[0, +\infty]$	Number of trials until a single failure with constant probability
Beta-binomial	$[0, n]$	Number of successes in n trials, with varying p .
Multinomial	$[0, n]$ for each x_i , with $\sum(x_i) = n$	Number of counts x_i in k categories out of n trials.

The table below summarizes some the key features of the **continuous distributions** we explored in this module.

Distribution	Support	What it models
Uniform	$[\min, \max]$	Continuous variables where all values are equally likely.
Normal	$[-\infty, +\infty]$	Many natural processes
Lognormal	$(0, +\infty)$	Many processes that are normally distributed on log scale; values arising from multiplicative processes
Gamma	$(0, +\infty]$	Waiting times until an event occurs
Beta	$[0, 1]$	Probabilities of single events (i.e., probabilities for binomial processes)

Distribution	Support	What it models
Dirichlet	$[0, 1]$ for each x_i , and $\sum(x_i) = 1$	Probabilities of multiple events (i.e., probabilities for multinomial processes)
Exponential	$[0, +\infty]$	Waiting times between events
Triangular	$[\min, \max]$	Continuous variables about which little is known; requires minimum, maximum, and mode.

4.4 Fitting and testing distributions

Deciding what probability distribution to use to analyze your data is an important step in biological data analysis. The next step is to see how well your data actually match a distribution. This page demonstrates ways to compare data to probability distributions. We will also explore some methods to explicitly test whether or not a set of values came from a particular distribution.

4.4.1 Estimating distributional parameters

The function `fitdistr()` in package `MASS` estimates the **parameters** of a distribution most likely to produce random variables supplied as input. Simply put, if you have a variable x and want to know what parameters of some distribution are most likely to produce x , then `fitdistr()` will estimate this for you. The method of estimation is called **maximum-likelihood**, which we will explore in more detail later. Examples of distribution fitting are shown below. Note that for the beta distribution, starting values for the fitting function must be supplied. This is because for this distribution the parameter estimation process is stochastic and iterative and needs somewhere to start.

The function `fitdistr()` is in the package `MASS`, which is short for “Modern Applied Statistics with S”. This widely used package implements many methods described in the textbook of the same name (Venables and Ripley 2002). The titular language S was a precursor language to R.

```
library(MASS)

set.seed(123)
n <- 50
```

```

x1 <- rnorm(n, 12, 2)
fitdistr(x1, "normal")

##      mean          sd
##  12.0688071   1.8331290
##  ( 0.2592436) ( 0.1833129)

x2 <- rpois(n, 8)
fitdistr(x2, "Poisson")

##      lambda
##  8.2200000
##  (0.4054627)

x3 <- rgeom(n, 0.7)
fitdistr(x3, "geometric")

##      prob
##  0.90909091
##  (0.03876377)

x4 <- rbeta(n, 2, 0.5)
fitdistr(x4, "beta", start=list(shape1=1, shape2=1))

## Warning in densfun(x, parm[1], parm[2], ...): NaNs produced
## Warning in densfun(x, parm[1], parm[2], ...): NaNs produced
## Warning in densfun(x, parm[1], parm[2], ...): NaNs produced

##      shape1          shape2
##  2.1520168   0.5070293
##  (0.4828815) (0.0844015)

x5 <- rnbinom(n, 4, 0.2)
fitdistr(x5, "negative binomial")

##      size          mu
##  3.674590   15.660000
##  ( 0.907684) ( 1.283731)

```

How did `fitdistr()` do? Most of the parameter estimates were pretty close to the true values. The estimates generally get closer as the sample size increases. Try increasing `n` to 500 and rerunning the code above.

4.4.2 Graphical methods for examining distributions

4.4.2.1 Comparing distributions with PDF or PMF plots

We can compare the distributions implied by the estimated parameters to the actual data using the quantile or distribution functions. The basic idea is to plot the data distribution, and superimpose the equivalent plots of the hypothetical distributions.

In the example below, we compare the probability mass function (PMF) of a discrete distribution (e.g., counts) to the PMFs of two possible distributions that might match the data. Note that the PMF of a discrete distribution can be estimated as the proportion of observations that take on each value.

```
set.seed(123)

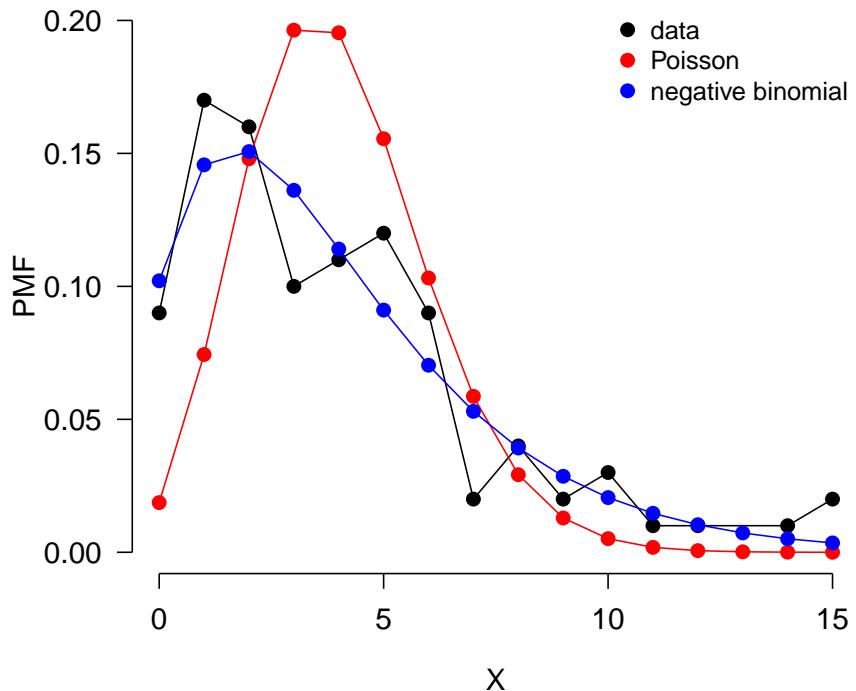
# the counts:
x <- rnbinom(100, mu=4, size=2)

# fit parameters for poisson and negative binomial
f1 <- fitdistr(x, "Poisson")
f2 <- fitdistr(x, "negative binomial")

# calculate PMFs of candidate distributions across
# range of data
xx <- 0:max(x)
y1 <- dpois(xx, f1$estimate)
y2 <- dnbinom(xx, size=f2$estimate["size"],
              mu=f2$estimate["mu"])

# empirical PMF of counts
y3 <- table(x)/length(x)
plot.x <- as.numeric(names(y3))
plot.y <- as.numeric(y3)

# make plot
par(mfrow=c(1,1), mar=c(5.1, 5.1, 1.1, 1.1),
     las=1, lend=1,
     cex.axis=1.2, cex.lab=1.2, bty="n")
plot(plot.x, plot.y, type="o",
      pch=16, cex=1.3, ylim=c(0, 0.2),
      ylab="PMF", xlab="X")
points(xx, y1, type="o", pch=16, cex=1.3, col="red")
points(xx, y2, type="o", pch=16, cex=1.3, col="blue")
legend("topright", legend=c("data", "Poisson", "negative binomial"),
       pch=16, pt.cex=1.3,
       col=c("black", "red", "blue"), bty="n")
```

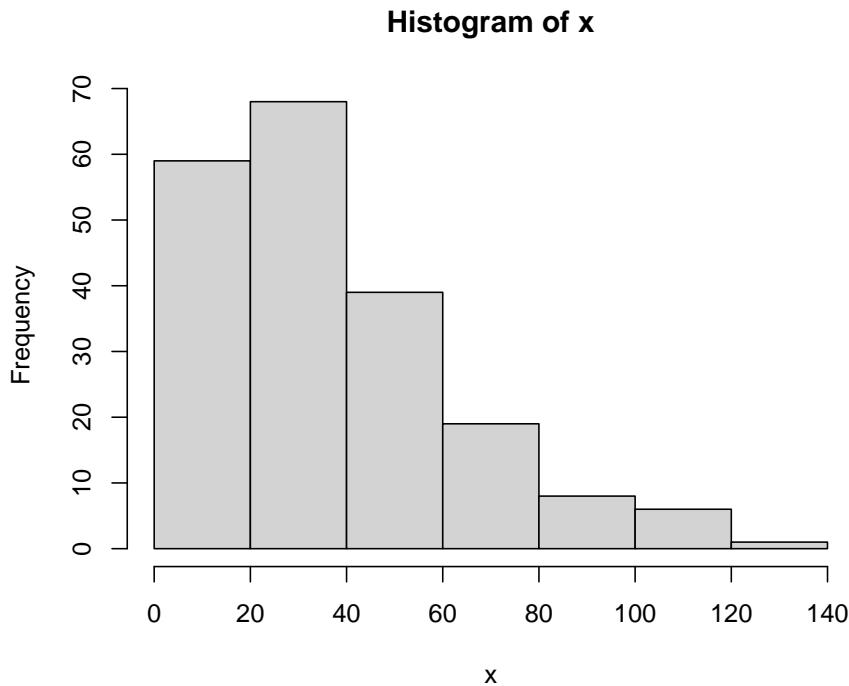


In this example, the data (black) seem to be better described by the negative binomial (blue) than by the Poisson (red).

The example below shows the same procedure applied to a continuous distribution. The only difference is in how the probability density function (PDF) of the data is estimated (note PDF, not PMF). The PDF of a continuous distribution is estimated with function `density()`. It is a good idea to limit the estimation of density functions to the domain of the data. Keep in mind, however, that different distributions have different limits. If a suspected distribution is supported outside the possible range of the data, then it might not be the right one. The example below illustrates this: the data (`x`) are all positive, but one of the distributions tested (normal) will produce negative values using the estimated parameters (`f1`). That alone should lead the researcher in this situation to eliminate the normal as a possibility; the plot only confirms the unsuitability of the normal for modeling `x`.

```
set.seed(123)

# the data:
x <- rgamma(200, shape=2, scale=20)
hist(x)
```



```

# fit parameters for poisson and negative binomial
f1 <- fitdistr(x, "normal")
f2 <- fitdistr(x, "lognormal")
f3 <- fitdistr(x, "gamma")

## Warning in densfun(x, parm[1], parm[2], ...): NaNs produced
# calculate PMFs of candidate distributions across
# range of data
xx <- seq(min(x), max(x), length=1000)
y1 <- dnorm(xx, f1$estimate[1], f1$estimate[2])
y2 <- dlnorm(xx, f2$estimate[1], f2$estimate[2])
y3 <- dgamma(xx, shape=f3$estimate[1], rate=f3$estimate[2])

# empirical PDF of data
dx <- density(x, from=min(x), to=max(x))

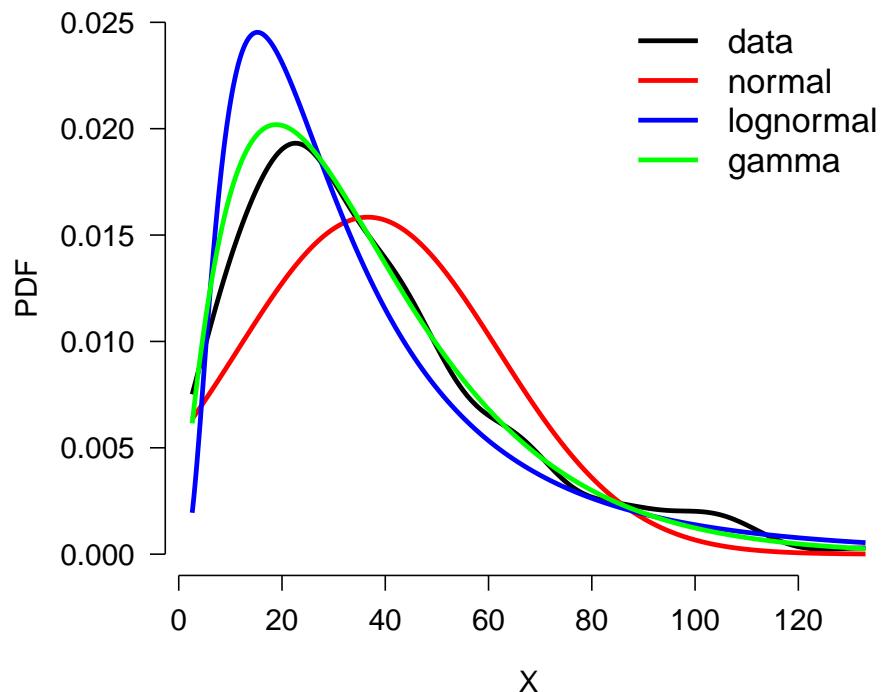
# make plot
par(mfrow=c(1,1), mar=c(5.1, 5.1, 1.1, 1.1),
  las=1, lend=1,
  cex.axis=1.2, cex.lab=1.2, bty="n")

```

```

plot(dx$x, dx$y,
     type="l", lwd=3, ylim=c(0, 0.025),
     ylab="", xlab="X")
title(ylab="PDF", line=4)
points(xx, y1, type="l", lwd=3, col="red")
points(xx, y2, type="l", lwd=3, col="blue")
points(xx, y3, type="l", lwd=3, col="green")
legend("topright",
       legend=c("data", "normal", "lognormal", "gamma"),
       lwd=3, col=c("black", "red", "blue", "green"),
       cex=1.3, bty="n")

```



The plot suggests that the data are better described by a gamma distribution than by a normal or lognormal. This is because the curve for the data is most closely matched by the curve for the gamma. Try adjusting the sample size in the example above (e.g., `n <- 50`) and see if the correct distribution always matches the data.

4.4.2.2 Comparing distributions using CDF and ECDF

Another way to compare data to distributions is with the cumulative distribution function (CDF). The basic idea is the same as using density functions: compare the empirical CDF (ECDF) of the data to the CDFs potential distributions and judge which one best matches the data.

The example below shows how to use ECDF and CDF to compare a continuous distribution to various target distributions.

```
set.seed(123)

# make the data:

x <- rlnorm(100, meanlog=log(4), sdlog=log(3))

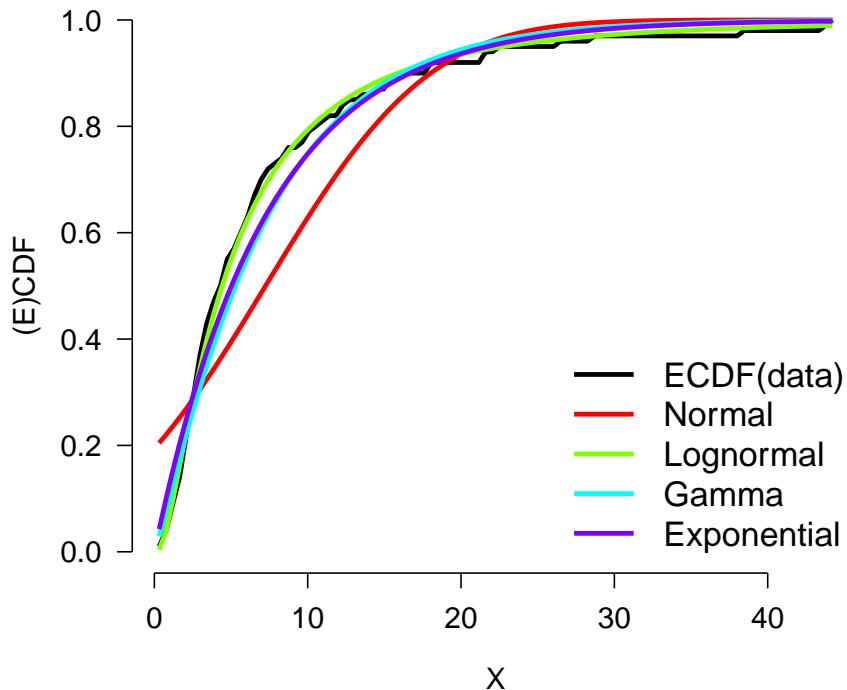
# fit some distributions
f1 <- fitdistr(x, "normal")
f2 <- fitdistr(x, "lognormal")
f3 <- fitdistr(x, "gamma")
f4 <- fitdistr(x, "exponential")

# calculate density functions over domain of x
xx <- seq(min(x), max(x), length=100)
y1 <- pnorm(xx, f1$estimate["mean"], f1$estimate["sd"])
y2 <- plnorm(xx, f2$estimate["meanlog"], f2$estimate["sdlog"])
y3 <- pgamma(xx,
              shape=f3$estimate["shape"],
              rate=f3$estimate["rate"])
y4 <- pexp(xx, rate=f4$estimate["rate"])

# ecdf of data
y5 <- ecdf(x)(xx)

# fancy plot
cols <- c("black", rainbow(4))
par(mfrow=c(1,1), mar=c(5.1, 5.1, 1.1, 1.1),
    las=1, lend=1,
    cex.axis=1.2, cex.lab=1.2, bty="n")
plot(xx, y5, type="l", lwd=3, xlab="X",
      ylab="(E)CDF", ylim=c(0,1), col=cols[1])
points(xx, y1, type="l", lwd=3, col=cols[2])
points(xx, y2, type="l", lwd=3, col=cols[3])
points(xx, y3, type="l", lwd=3, col=cols[4])
points(xx, y4, type="l", lwd=3, col=cols[5])
legend("bottomright",
       legend=c("ECDF(data)", "Normal", "Lognormal", "Gamma", "Exponential"),
       bty="n")
```

```
lwd=3, col=cols, bty="n", cex=1.3)
```



In this example, the ECDF of the original data (black) appears to best match the lognormal (green)—as well it should, because the data came from a lognormal distribution. The normal (red) is clearly inappropriate because it diverges from the actual data over most of the range of the data. The gamma and exponential are each better than the normal, but not as close to the data as the lognormal.

4.4.2.3 Comparing distributions using QQ plots

Finally, you can use **quantile-quantile (QQ) plots** to compare data to different distributions that you suspect they may follow. The example below uses a QQ plot instead of ECDF as seen above.

```
set.seed(123)

# true distribution: lognormal
n <- 100
x <- rlnorm(n, meanlog=log(4), sdlog=log(3))
```

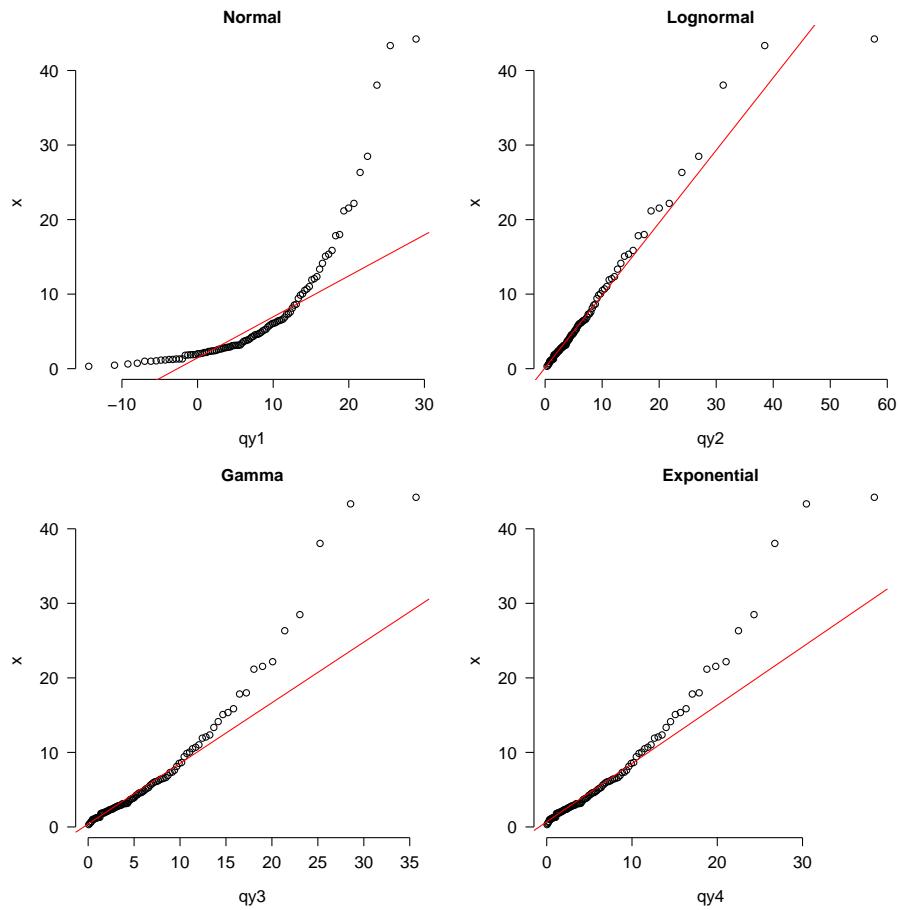
```
# estimate distributional parameters
f1 <- fitdistr(x, "normal")
f2 <- fitdistr(x, "lognormal")
f3 <- fitdistr(x, "gamma")
f4 <- fitdistr(x, "exponential")

# quantiles of reference distribution
qx <- ppoints(n)

# define functions to draw from each reference distribution
dfun1 <- function(p){qnorm(p, f1$estimate[1], f1$estimate[2])}
dfun2 <- function(p){qlnorm(p, f2$estimate[1], f2$estimate[2])}
dfun3 <- function(p){qgamma(p, shape=f3$estimate[1],
                           rate=f3$estimate[2])}
dfun4 <- function(p){qexp(p, f4$estimate)}

# get values at each reference quantile
qy1 <- dfun1(qx)
qy2 <- dfun2(qx)
qy3 <- dfun3(qx)
qy4 <- dfun4(qx)

# make plot
par(mfrow=c(2,2), mar=c(5.1, 5.1, 1.1, 1.1),
     las=1, lend=1,
     cex.axis=1.2, cex.lab=1.2, bty="n")
qqplot(qy1, x, main="Normal")
qqline(x, distribution=dfun1, col="red")
qqplot(qy2, x, main="Lognormal")
qqline(x, distribution=dfun2, col="red")
qqplot(qy3, x, main="Gamma")
qqline(x, distribution=dfun3, col="red")
qqplot(qy4, x, main="Exponential")
qqline(x, distribution=dfun4, col="red")
```



The figure shows that the normal is clearly inappropriate because the points do not fall on the line. The gamma and exponential are okay over most of the domain of x , but depart for large values. The lognormal plot (top right) shows the best match between the data and the reference distribution.

It is important to keep in mind that the comparisons we've made here are of idealized cases. Real data are often much messier and noisier than the distributions used in these examples. Sometimes real data are best characterized by mixture distributions for which there are no straightforward ways to estimate parameters. Still, the steps outlined above for estimating distributional parameters and comparing to idealized distributions are important first steps in data analysis.

4.4.3 Formal tests for distributions

4.4.3.1 Tests for normality

The normal distribution is so important for so many statistical methods that people have developed tests to formally determine whether data come from a normal distribution. The most common test for normality is the **Shapiro-Wilk test**. The Shapiro-Wilk test tests the null hypothesis that a sample x_1, \dots, x_n comes from a normally distributed population. The test statistic W is calculated as:

$$W = \frac{\left(\sum_{i=1}^n a_i x_{(i)}\right)^2}{\sum_{i=1}^n (x_i - \bar{x})^2}$$

where \bar{x} is the sample mean, $x_{(i)}$ is the i^{th} order statistic, and a_i is a coefficient related to expected values of *i.i.d.* normal variables. When the null hypothesis of a Shapiro-Wilk test is rejected, then there is evidence that the data are not normally distributed. The code below shows some examples of the Shapiro-Wilk test in R.

```
set.seed(123)
x1 <- rnorm(20)
shapiro.test(x1)
##
##  Shapiro-Wilk normality test
##
## data: x1
## W = 0.96858, p-value = 0.7247

x2 <- rpois(20, 10)
shapiro.test(x2)
##
##  Shapiro-Wilk normality test
##
## data: x2
## W = 0.90741, p-value = 0.0569

x3 <- runif(20)
shapiro.test(x3)
##
##  Shapiro-Wilk normality test
##
## data: x3
## W = 0.9316, p-value = 0.1658

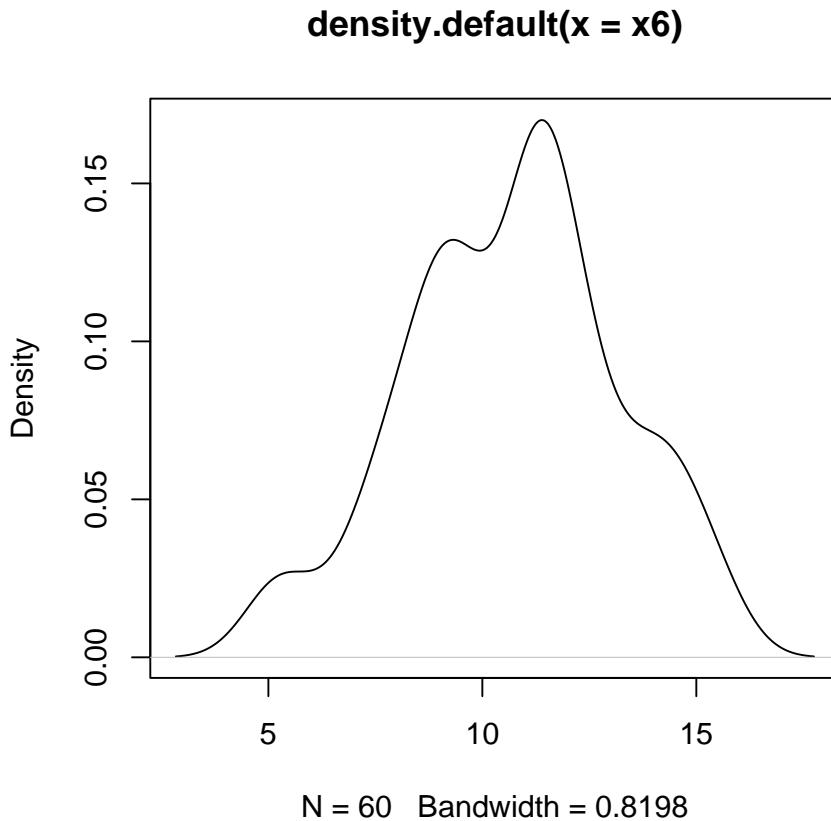
x4 <- runif(100)
```

```
shapiro.test(x4)
##
##  Shapiro-Wilk normality test
##
## data: x4
## W = 0.96392, p-value = 0.007723
```

Interestingly, the Shapiro-Wilk test turns out to be very powerful. That is, it can detect very small differences from the null hypothesis (i.e., normality) if the sample size is large enough. So, if you have data that you suspect are normal, but the Shapiro-Wilk test says otherwise, you need to visually inspect the data to determine how serious the departure from normality is. Another situation where the Shapiro-Wilk test does not perform well is when data with many repeated values. Consider the following example:

```
# repeat each value 3 times
x6 <- rep(rnorm(20, 10, 3), each=3)
# P < 0.05, so test says nonnormal
shapiro.test(x6)
```

```
##
##  Shapiro-Wilk normality test
##
## data: x6
## W = 0.97026, p-value = 0.1502
# density plot look normal!
par(mfrow=c(1,1))
plot(density(x6))
```



4.4.3.2 Kolmogorov-Smirnov tests

The **Kolmogorov-Smirnov (KS) test** is a non-parametric test that compares a dataset to a distribution. Unlike the Shapiro-Wilk test, which only tests for normality, the KS test can compare data to any distribution. The KS test is useful because it is sensitive to differences in both location (i.e., mean) and shape. The example below illustrates the use of the KS test and plots the distributions.

```
n <- 1e2
x1 <- rnorm(n, 10, 2)
x2 <- rnorm(n, 10, 2)
x3 <- rnorm(n, 14, 2)
x4 <- rnorm(n, 14, 20)
x5 <- rpois(n, 14)

# run KS tests:
```

```
ks.test(x1, x2)

##
##  Two-sample Kolmogorov-Smirnov test
##
## data: x1 and x2
## D = 0.12, p-value = 0.4676
## alternative hypothesis: two-sided
ks.test(x1, x3)

##
##  Two-sample Kolmogorov-Smirnov test
##
## data: x1 and x3
## D = 0.64, p-value < 2.2e-16
## alternative hypothesis: two-sided
ks.test(x3, x4)

##
##  Two-sample Kolmogorov-Smirnov test
##
## data: x3 and x4
## D = 0.41, p-value = 1.001e-07
## alternative hypothesis: two-sided
ks.test(x3, x5)

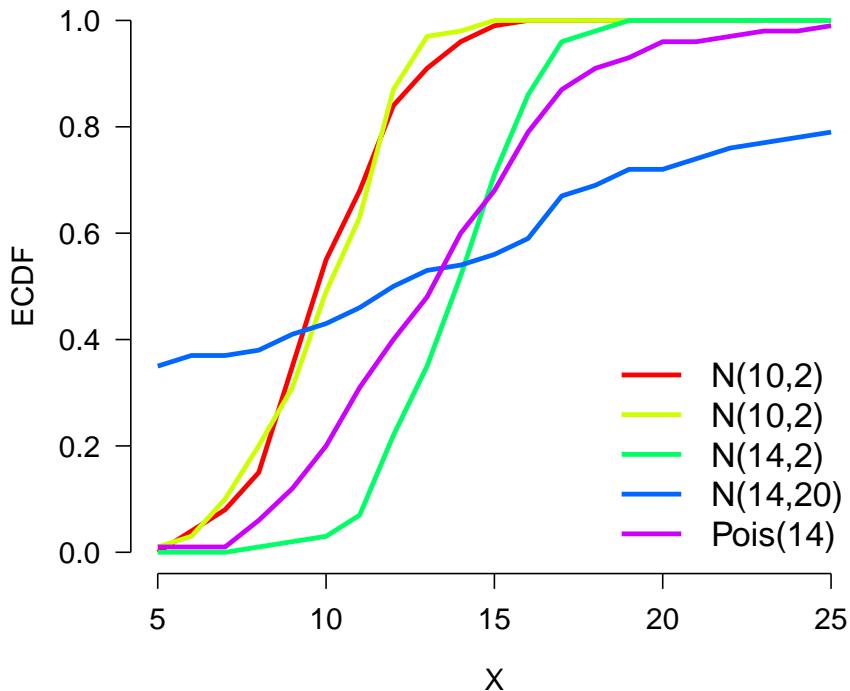
## Warning in ks.test(x3, x5): p-value will be approximate in the presence of ties
##
##  Two-sample Kolmogorov-Smirnov test
##
## data: x3 and x5
## D = 0.24, p-value = 0.006302
## alternative hypothesis: two-sided
# make a plot
xx <- 5:25
y1 <- ecdf(x1)(xx)
y2 <- ecdf(x2)(xx)
y3 <- ecdf(x3)(xx)
y4 <- ecdf(x4)(xx)
y5 <- ecdf(x5)(xx)

cols <- rainbow(5)
par(mfrow=c(1,1), mar=c(5.1, 5.1, 1.1, 1.1),
    bty="n", las=1, lend=1,
```

```

cex.axis=1.2, cex.lab=1.2)
plot(xx, y1, type="l", lwd=3,
      xlab="X", ylab="ECDF", col=cols[1])
points(xx, y2, type="l", lwd=3, col=cols[2])
points(xx, y3, type="l", lwd=3, col=cols[3])
points(xx, y4, type="l", lwd=3, col=cols[4])
points(xx, y5, type="l", lwd=3, col=cols[5])
legend("bottomright",
       legend=c("N(10,2)", "N(10,2)",
               "N(14,2)", "N(14,20)", "Pois(14)"),
       lwd=3, col=cols, bty="n", cex=1.3)

```



The KS test results show that distributions x_1 and x_2 are not different ($P > 0.05$). This makes sense because they are both normal distributions with the same mean and SD. Distributions x_1 and x_3 are different ($P < 0.05$) because they have different means (10 vs. 14). Distributions x_3 and x_4 were different ($P < 0.05$) because they had different SD (2 vs. 20). Finally, distributions x_3 and x_5 were different ($P < 0.05$) because they were different distributions with different shapes (normal vs. Poisson).

The KS test can be useful when you want to see how well your data matches up with their theoretical distribution. Consider the example of a lognormally distributed response below. If we suspect that the data come from a lognormal distribution or a gamma distribution, we can use the function `fitdistr()` to estimate the parameters of a lognormal and a gamma distribution that are most likely to produce those data. Then, we use a KS test to compare the data to the idealized candidates.

```
set.seed(123)
x1 <- rlnorm(100, 2, 2)
# estimate moments
fit1 <- fitdistr(x1, "lognormal")
fit2 <- fitdistr(x1, "gamma")
```

Notice that the parameters from `fitdistr()` are supplied as arguments to `ks.test()`, following the name of the distribution we want to compare to. If parameters are not supplied, then R will perform the KS test using the default parameter values (`meanlog=0` and `sdlog=1` for `plnorm()`; none for `pgamma()`), which is not what we want.

```
ks.test(x1, "plnorm", fit1$estimate[1], fit1$estimate[2])

##
##  One-sample Kolmogorov-Smirnov test
##
## data: x1
## D = 0.058719, p-value = 0.8808
## alternative hypothesis: two-sided

ks.test(x1, "pgamma", shape=fit2$estimate[1],
        rate=fit2$estimate[2])

##
##  One-sample Kolmogorov-Smirnov test
##
## data: x1
## D = 0.16211, p-value = 0.01043
## alternative hypothesis: two-sided

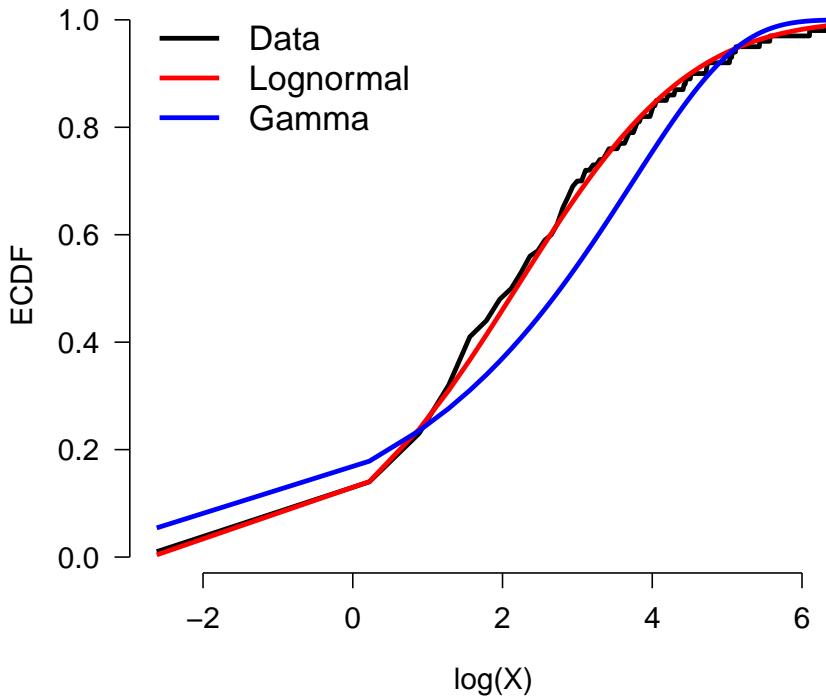
# plot x1 and its theoretical distribution
xx <- seq(min(x1), max(x1), length=500)
y1 <- ecdf(x1)(xx)
y2 <- plnorm(xx, fit1$estimate[1], fit1$estimate[2])
y3 <- pgamma(xx, shape=fit2$estimate[1], rate=fit2$estimate[2])

par(mfrow=c(1,1), mar=c(5.1, 5.1, 1.1, 1.1), bty="n",
    las=1, lend=1, cex.axis=1.2, cex.lab=1.2)
plot(log(xx), y1, type="l", lwd=3, xlab="log(X)", ylab="ECDF")
```

```

points(log(xx), y2, type="l", lwd=3, col="red")
points(log(xx), y3, type="l", lwd=3, col="blue")
legend("topleft",
       legend=c("Data", "Lognormal", "Gamma"),
       lwd=3,
       col=c("black", "red", "blue"),
       bty="n",
       cex=1.3)

```



While neither the lognormal distribution nor the gamma distribution are awful at representing the data, the KS test suggests that the simulated data `x1` come from a lognormal distribution, not a gamma distribution. This is because the KS test was non-significant when comparing the data to the lognormal, and significant ($P < 0.05$) when comparing the data to the gamma distribution. The test comparing the data to the gamma distribution was, however, close to non-significant ($P \approx 0.01$). See what happens if you change the random number seed and re-run the random generation and the test.

4.5 Data transformations

Data transformation is the process of changing the values in your data according to a function. This is usually done to make the data better conform to the assumptions of a statistical test, or to make graphs easier to interpret. Transformation is done to make patterns easier to detect. In many fields transformations may be standard practice, and with good reason. Just make sure that you understand what the transform function is doing, and most importantly, why you are transforming your data.

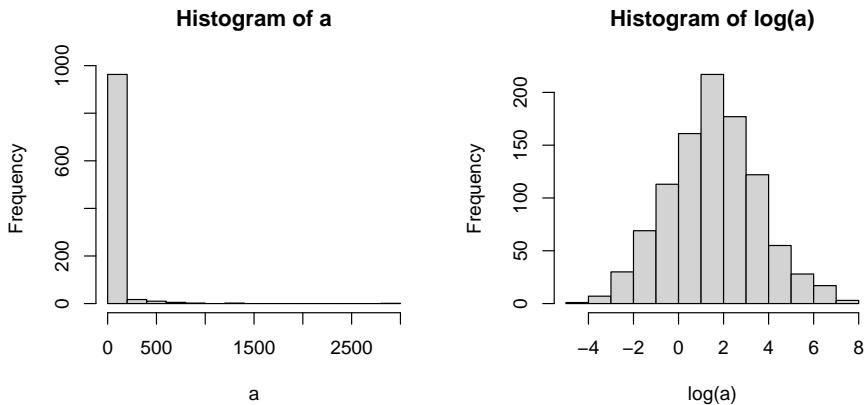
4.5.1 Why transform?

Data transformation is the process changing the values of your data according to a function. This is usually done so that the data better conform to the assumptions of a statistical test, or to make graphs of your data more interpretable. The functions that transform data are usually called “transforms”. There are many transforms out there, but only a few are very common in biology. All convert values on the **original scale** to values on the **transformed scale**. Variables often have different properties after a transformation, such as normality, or a different domain, or a different distribution shape. In general, a function for transforming data should be **deterministic** and **invertible**:

- **Deterministic:** means that the function always returns the same transformed value for a given input.
- **Invertible:** means that values on the transformed scale can be converted unambiguously to values on the original scale. In mathematics such functions are also called “bijections”.

Data transformations are commonly used to make data fit certain distributions. For example, consider the figure below:

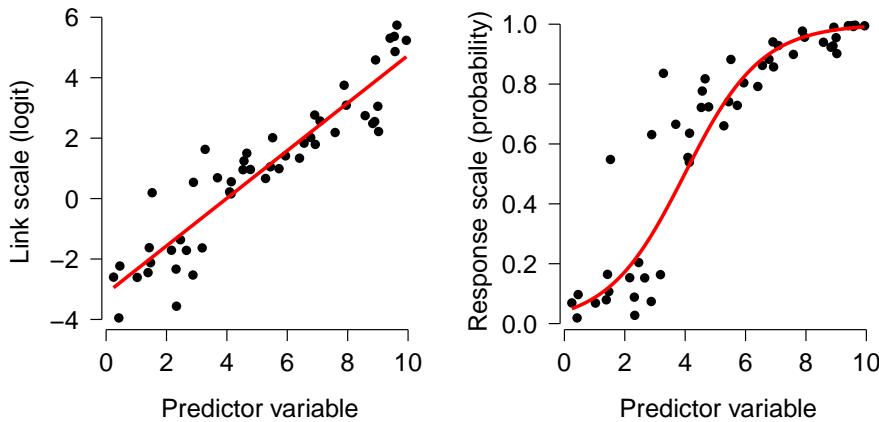
```
a <- rlnorm(1000, 1.5, 2)
par(mfrow=c(1,2))
hist(a)
hist(log(a))
```



The left panel shows data with mostly small values and a strong positive skew. The right panel shows the same data on the log scale. Notice how the log-transformed values look much more like a normal distribution. This is very important for many statistical methods, which require normality either in the values or in the residuals. The log-transformed plot also lets us see more details about the distribution that are obscured on the original scale.

4.5.2 Transforms vs. link functions

In generalized linear models (GLM) the response variable is modeled on a scale defined by a **link function**. That link function works a little like a transform, and is often confused with a transform, but it is not a transform. The figure below shows an example of this: the response variable on its original scale (right panel), a proportion, is analyzed on the logit scale (left panel). The ranks of observations do not change, but their values do. Additionally, putting the values on another scale can affect the variance of those values.



GLM link functions are not transforms because the variance is still modeled on the original scale. However, when data are transformed and then analyzed, the variance is modeled on the transformed scale. The equations below show the differences between a Gaussian GLM with a link function and an ordinary linear model on transformed data. Notice the difference in how each model includes stochasticity.

Linear model (LM) with transformation

$$\begin{aligned}y_i &= e^{z_i} \\z_i &\sim \text{Normal}(mu_i, \sigma^2) \\mu_i &= f(\theta)\end{aligned}$$

Generalized linear model (GLM) with link function

$$\begin{aligned}y_i &\sim \text{Normal}(z_i, \sigma^2) \\z_i &= e^{mu_i} \\mu_i &= f(\theta)\end{aligned}$$

In the LM, the expected value mu_i and its variance are both on the log scale. In the GLM, mu_i is on the log scale but the variance in observed values is not. This difference can have important consequences on the estimates of a statistical model. See the example here for a worked example.

4.5.3 Log transformation

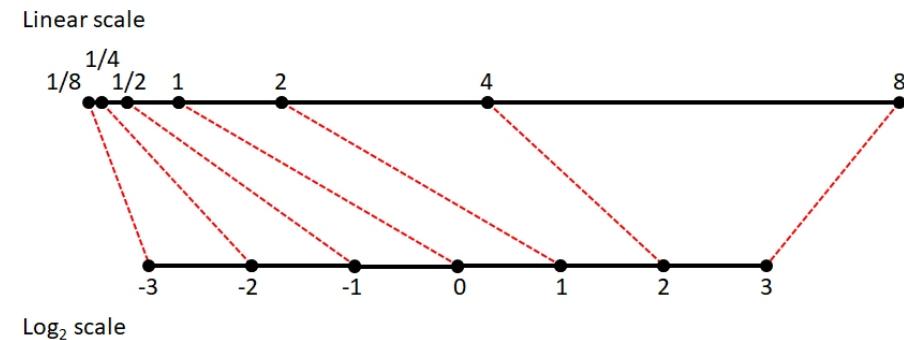
The **log-transform** is one of the most common transforms in biology. It is simply the **logarithm of the values**¹⁰. Most biologists use either the natural logarithm (\log_e , \ln or just \log) or the base-10 logarithm (\log_{10}). The choice of base is not that important, because any log-transform will accomplish the same effect. The default in math and statistics (and in R) is the natural logarithm. \log_{10} is also very common in biology and engineering. It is sometimes called the “common log”, but most people just say “log 10” when referring to this function. Whatever base you use, just be consistent and make it clear to your audience what you did¹¹. It is depressingly common to be unable to use values from old papers because they didn’t specify which base logarithm they used.

There are usually two reasons to use a log-transform: normalization (i.e., variance stabilization) and positivity. As seen above, many variables that appear non-normal turn out to be normally distributed on a logarithmic scale. This often occurs when something about the values is multiplicative in nature. For example,

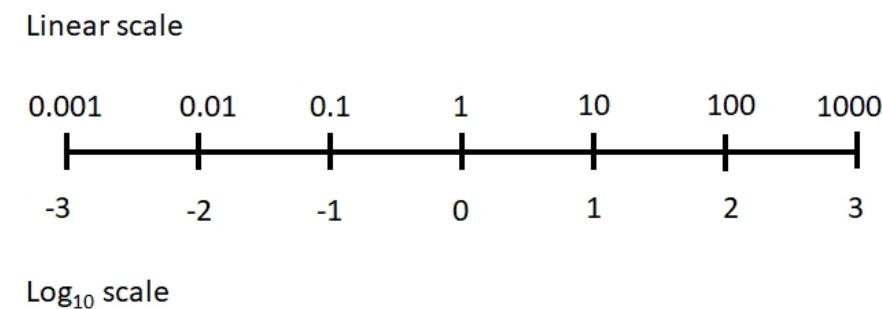
¹⁰Recall that the logarithm is the inverse of exponentiation: if $a = b^c$, then $\log_b(a) = c$. For example, $\log_2(8) = 3$ because $2^3 = 8$.

¹¹Whatever you do, don’t call the natural logarithm “lawn” unless you want to look silly.

body sizes, population sizes, and bank account balances are often log-normally distributed. This is because these quantities tend to grow at a rate that is proportional to their current size. Transforming to the logarithmic scale changes multiplicative changes to additive changes. The figure below shows how values separated by a factor of 2 on the linear scale map to values separated by 1 (i.e., 20) on the \log_2 scale. Values separated by a common *scaling* factor on the original scale are mapped to values separated by a common *additive* constant on the log scale¹².



As a side effect of this normalization, log-transformation allows researchers to work with variables that vary over several orders of magnitude. This is because scalar multiplication on the untransformed scale is the same as addition on the log-transformed scale¹³. The figure below shows how values ranging from 0.001 to 1000 (spanning 6 orders of magnitude) can be managed on the log-transformed scale.



The second reason that the log-transform is commonly used is to ensure that values are always positive. One of the properties of the exponential function is that for any positive base, raising that base to a real power will always return a positive result. For this reason, it is common practice to analyze variables with a

¹²This video illustrates the usefulness of logarithms in statistics *and* has a nice theme song.

¹³Remember that exponential functions relate addition and multiplication. This is how exponentiation is extended to non-integer powers: $X^a X^b = X^{a+b}$.

positive domain on the logarithmic scale. Such variables include lengths, masses, and counts, which must be positive or non-negative. Log transformation can ensure that your analysis does not predict impossibilities like negative masses.

4.5.3.1 Log transformation in R

Values in a data frame can be log-transformed with the `log()` (natural log) or `log10()` (\log_{10}) function. Other bases can be used via arguments to the `log()` function (e.g., `log(8,base=2)`). It is usually a good idea to keep the transformed values in their own variable, rather than overwriting the original values. This way you can quickly access the original values without having to back-transform. It is often useful to have the original values around for making figures. Keeping the original values also means you don't have to remember whether a variable or object contains the original or transformed values.

```
# spare copy:
x <- iris

# natural log transform:
x$log.petal.len <- log(x$Petal.Length)

# log base 10 transform:
x$log.petal.wid <- log10(x$Petal.Width)
```

You can also log-transform many variables at once using the function `apply()`. In a data frame this means applying the `log()` function to multiple columns (margin 2).

```
# load the dataset from package MASS
data(crabs, package="MASS")

# make a spare copy
x <- crabs
head(x)

##   sp sex index   FL   RW   CL   CW   BD
## 1  B    M      1  8.1  6.7 16.1 19.0 7.0
## 2  B    M      2  8.8  7.7 18.1 20.8 7.4
## 3  B    M      3  9.2  7.8 19.0 22.4 7.7
## 4  B    M      4  9.6  7.9 20.1 23.1 8.2
## 5  B    M      5  9.8  8.0 20.3 23.0 8.2
## 6  B    M      6 10.8  9.0 23.0 26.5 9.8

# overwrite (transform in place)
# (ok but can cause headaches later)
x[,4:8] <- apply(x[,4:8], 2, log)
head(x)
```

```

##   sp sex index      FL      RW      CL      CW      BD
## 1  B   M     1 2.091864 1.902108 2.778819 2.944439 1.945910
## 2  B   M     2 2.174752 2.041220 2.895912 3.034953 2.001480
## 3  B   M     3 2.219203 2.054124 2.944439 3.109061 2.041220
## 4  B   M     4 2.261763 2.066863 3.000720 3.139833 2.104134
## 5  B   M     5 2.282382 2.079442 3.010621 3.135494 2.104134
## 6  B   M     6 2.379546 2.197225 3.135494 3.277145 2.282382

# better way:
# make new variables and
# add to original data frame (better)
x <- crabs
z <- apply(x[,4:8], 2, log)
z <- as.data.frame(z)
names(z) <- paste(names(z), "log", sep=".")
x <- cbind(x,z)
head(x)

##   sp sex index      FL      RW      CL      CW      BD    FL.log    RW.log    CL.log    CW.log
## 1  B   M     1 8.1 6.7 16.1 19.0 7.0 2.091864 1.902108 2.778819 2.944439
## 2  B   M     2 8.8 7.7 18.1 20.8 7.4 2.174752 2.041220 2.895912 3.034953
## 3  B   M     3 9.2 7.8 19.0 22.4 7.7 2.219203 2.054124 2.944439 3.109061
## 4  B   M     4 9.6 7.9 20.1 23.1 8.2 2.261763 2.066863 3.000720 3.139833
## 5  B   M     5 9.8 8.0 20.3 23.0 8.2 2.282382 2.079442 3.010621 3.135494
## 6  B   M     6 10.8 9.0 23.0 26.5 9.8 2.379546 2.197225 3.135494 3.277145
##       BD.log
## 1 1.945910
## 2 2.001480
## 3 2.041220
## 4 2.104134
## 5 2.104134
## 6 2.282382

```

4.5.3.2 Log transforming with 0 values

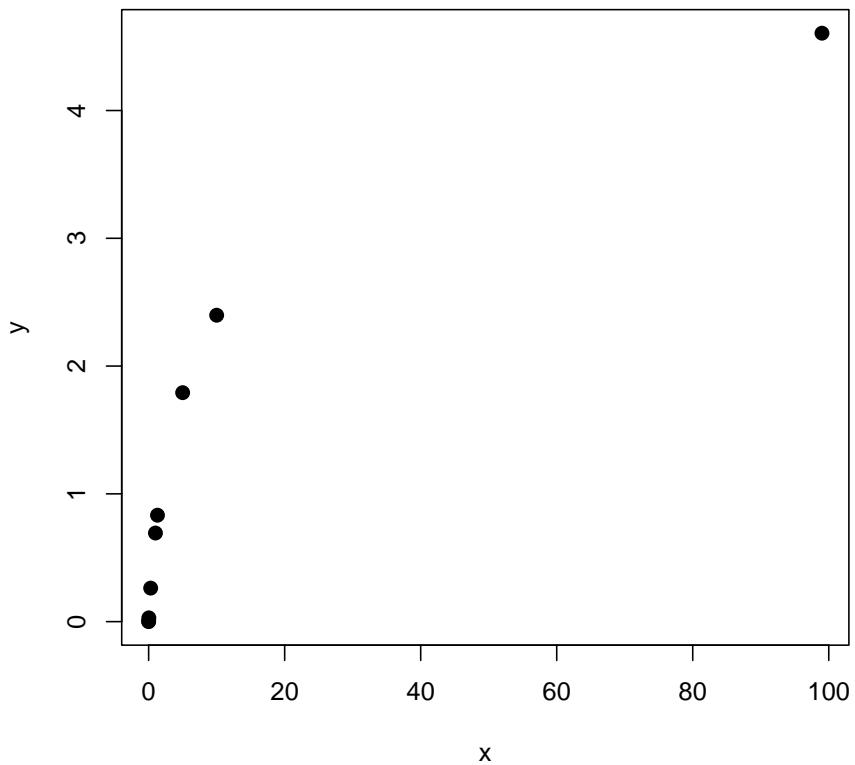
One problem that comes up with log-transformation is what to do with values of 0. The logarithm of 0 is undefined, and returns $-\infty$ in R. This is usually a problem with counts, rather than with measurements. The simplest solution is to add a small constant to the values before transforming. If the smallest nonzero value is 1, then it is easiest to just add 1 to the values:

$$y_i = \log(x_i + 1)$$

Where x_i is the original value and y_i is the transformed value (i.e., $f(x_i)$). Again, this works just fine if the smallest nonzero is about 1. Values between 0 and 1 can cause problems. Adding 1 to these values before log-transforming will tend

to compress the lower end of the distribution and expand the upper end of the distribution. The R commands below show this:

```
x <- c(0, 0.003, 0.03, 0.3, 1, 1.3, 5, 10, 99)
y <- log(x+1)
plot(x,y, cex=1.3, pch=16)
```



If the smallest nonzero value differs from 1 by more than an order of magnitude (i.e., ≥ 10 or ≤ 0.1), then a slightly different procedure is called for. The transformation below generalizes the “log of $x+1$ ” method in such a way as to result in 0 for 0 values and somewhat preserve the original orders of magnitude (McCune et al. 2002).

$$y_i = \log(x_i + d) - c$$

In this equation:

$$\begin{aligned}
 c &= \text{Trunc}(\log(\min(x))) \\
 d &= e^c \\
 \min(x) &= \text{smallest nonzero } x \\
 \text{Trunc}(x) &= \text{function that truncates } x \text{ to an integer}
 \end{aligned}$$

This can be implemented in R using the following custom function:

```

mgtrans <- function(x){
  minx <- min(x[x > 0])
  cons <- trunc(log(minx))
  y <- log(x + exp(cons)) - cons
  return(y)
}

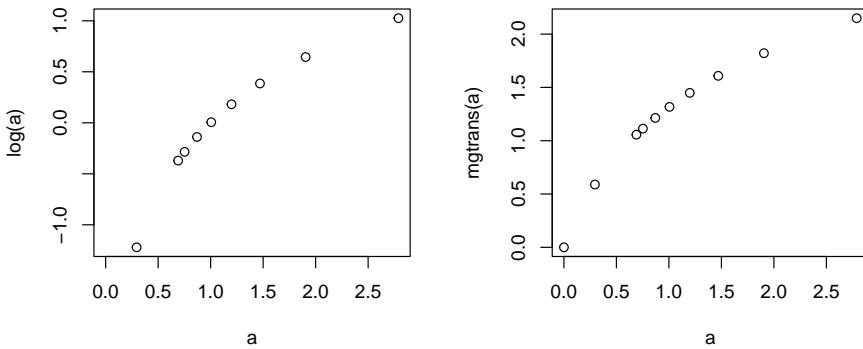
# example usage:
a <- c(0, rlnorm(9))
log(a)

## [1]      -Inf  1.025571370 -0.284773007 -1.220717712  0.181303480
## [6] -0.138891362  0.005764186  0.385280401 -0.370660032  0.644376549
mgtrans(a)

## [1]  0.0000000 2.1494853 1.1133862 0.5888655 1.4489449 1.2136603 1.3174789
## [8] 1.6086268 1.0565297 1.8209591

# count the points:
par(mfrow=c(1,2))
plot(a, log(a))    # missing one!
plot(a, mgtrans(a))

```



Note that if you use the McCune and Grace transform, you should keep the

original values around for plotting because back-transforming can be tricky (if not impossible, if you don't know d and c). If you are going to back-transform other values, such as model predictions, you will need to know d and c .

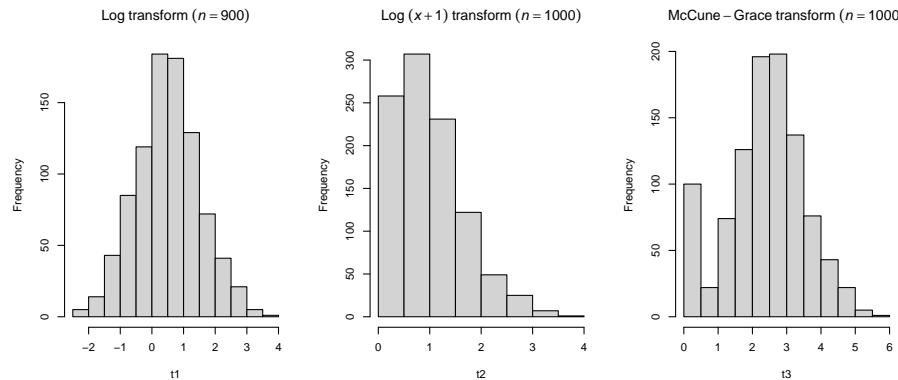
The example below shows how the McCune and Grace transform can be advantageous. First, we create 1000 random values from a lognormal distribution. Then, randomly set 100 of the values to 0. Log-transforming the data to achieve normality comes at the cost of losing 10% of the values because $\log(0)$ is undefined (left panel). The $\log(x + 1)$ transform retains all of the values, but distorts the shape of the distribution (middle panel) because many values are between 0 and 1. The McCune and Grace method (right panel) achieves normality (mostly) and retains all values.

```
set.seed(123)
n <- 1e3
x <- rlnorm(n, 0.5, 1)
x[sample(1:n, 100, replace=FALSE)] <- 0

t1 <- log(x)
t2 <- log(x+1)
t3 <- mgtrans(x)

length(which(is.finite(t1)))

## [1] 900
par(mfrow=c(1,3), cex.main=1.2)
hist(t1, main=expression(Log~transform~italic(n)==900))
hist(t2, main=expression(Log~(italic(x)+1)~transform~italic(n)==1000))
hist(t3, main=expression(McCune-Grace~transform~italic(n)==1000))
```



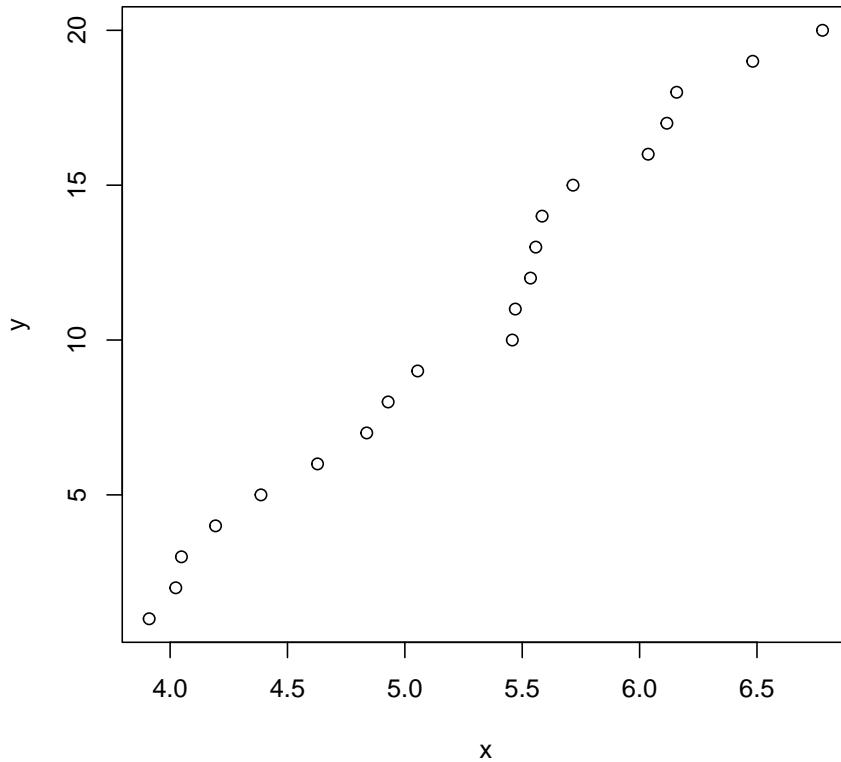
The spike in the right histogram comes from the fact that the McCune and

Grace method maps values of 0 to 0.

4.5.4 Rank transformation

The **rank transform** is just what it sounds like: replacing original values with their ranks. In R the function `rank()` does just that. Smaller values having smaller ranks and greater values having greater ranks. The smallest value maps to value 1 and the largest value maps to the number of values. That is, for any vector `x` `rank(x)[x==min(x)] = 1` and `rank(x)[x==max(x)] = length(x)`.

```
x <- rnorm(20, 5, 1)
y <- rank(x)
plot(x,y)
```



Notice how the transformed variables `y` are evenly spaced while the original values are not. This can be useful if values are very “clumped” or “spread out”.

Rank transforms are commonly used in nonparametric procedures. Note that the `rank()` function may return fractional or repeated ranks if there are ties:

```
# make some random values
x <- runif(10)

# force a tie
x[1] <- x[2]

# fractional ranks for ties:
rank(x)

## [1] 1.5 1.5 4.0 6.0 7.0 9.0 5.0 10.0 8.0 3.0

# force a three-way tie:
x[3] <- x[2]

# repeated ranks:
rank(x)

## [1] 2 2 2 6 7 9 5 10 8 4
```

4.5.5 Other transforms (less common)

4.5.5.1 Square-root transformation

Square-root transforms map values to their square roots. This is sometimes used as a variance-stabilizing transformation, similar to the log transform. Square-root transforms are useful when a response variable is proportional to the square of the explanatory variable, or when the data show heteroscedasticity (non-constant variance). Counts are sometimes analyzed using a square-root transform. The square-root transform is a special case of the **power transform** (where values are raised to a power) and a special case of the Box-Cox transform (see below). The square-root function in R is `sqrt()`. There are several ways to accomplish a square-root transform in R:

```
x <- rnorm(20, 5, 1)
y <- sqrt(x)
z <- x^0.5
```

Higher roots (cube root, fourth root, etc.) are sometimes used as transforms, but this is not very common.

```
w <- x^(1/3) # cube root
```

4.5.5.2 Logit transformation

The **logit transformation** is used to map proportions to the real number line. Usually this changes values into something resembling a normal distribution. The logit is defined as the *logarithm of the odds ratio*. Its inverse (back-transformation

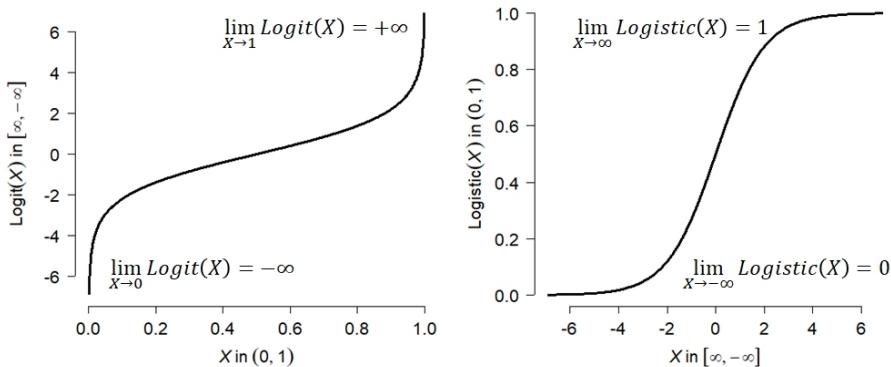
function) is the **logistic function**. For any value x in the open interval¹⁴ $(0, 1)$, the logit is calculated as:

$$\text{logit}(x) = \log\left(\frac{x}{1-x}\right)$$

If x is a probability or proportion, then $\text{logit}(x)$ is also the logarithm of the odds ratio. For this reason, the logit is also called the “log-odds”. The logit function is also the inverse of the logistic function. The logistic function of a variable x is:

$$\text{logistic}(x) = \frac{e^x}{e^x + 1} = \frac{1}{1 + e^{-x}}$$

The figure below shows the relationship between the logit and logistic functions:

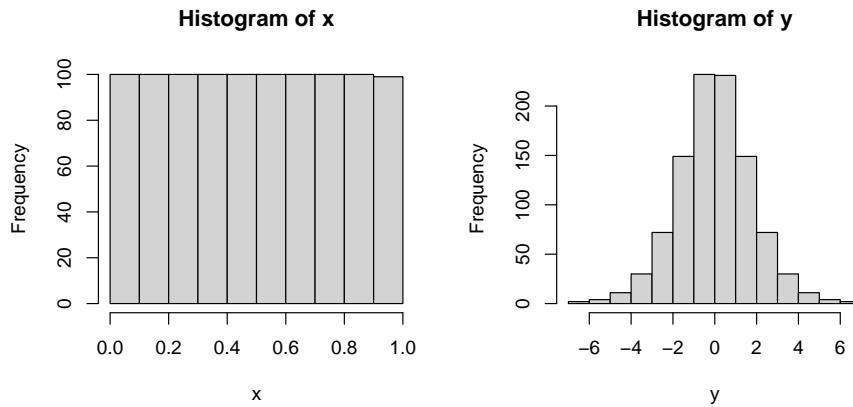


The logit transformation has the effect of stretching out the ends of a distribution, while keeping the relationship between the original and transformed values roughly linear near the middle of the distribution. The logit transformation can be accomplished in R using the function `qlogis()`; its inverse, the logistic function, is `plogis()`.

```
x <- 1:999/1000
y <- qlogis(x)

# compare:
par(mfrow=c(1,2))
hist(x)
hist(y)
```

¹⁴An **open interval** is an interval (a, b) that contains values $> a$ and $< b$, but never a or b . Open intervals are denoted by parentheses $()$. Intervals that contain their limits are called **closed intervals**, denoted $[a, b]$.



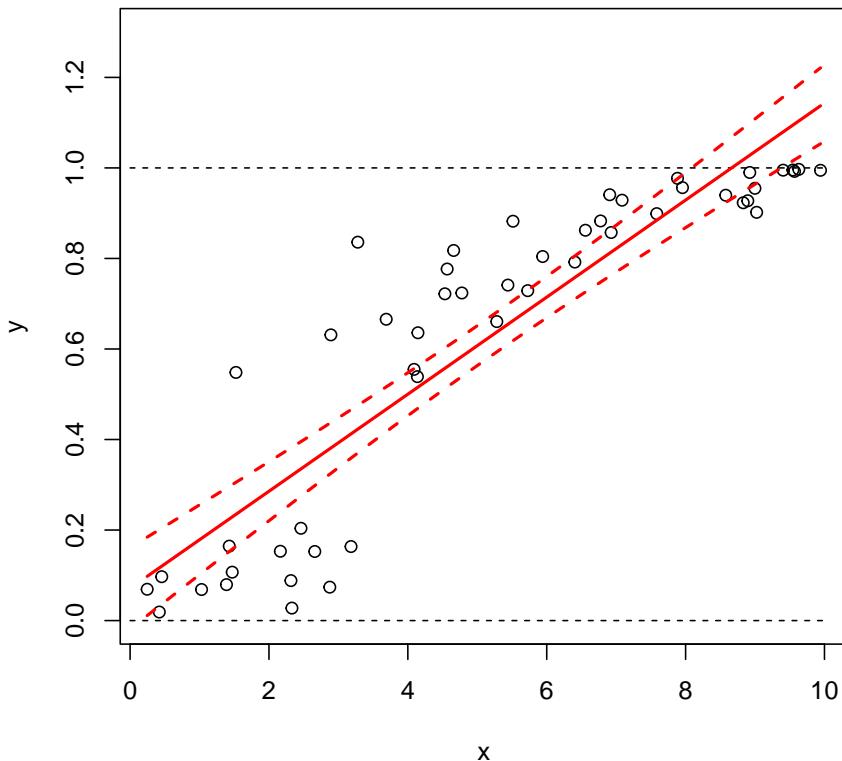
Why is this useful? The logistic function takes any real input and outputs a value between 0 and 1. This means that the logit function takes values between 0 and 1 and outputs a real number. The example below shows how this can be applied to analyze proportions:

The first figure below shows how analyzing proportions with linear models leads to clearly inappropriate results. The predicted values of the model extend well outside the possible range of a probability.

```
set.seed(123)
n <- 50
x <- runif(n, 0, 10)
z <- -3 + 0.75*x + rnorm(n, 0, 1)
y <- plogis(z)

mod1 <- lm(y~x)
px <- seq(min(x), max(x), length=100)
pred <- predict(mod1, newdata=data.frame(x=px), se.fit=TRUE)
mn <- pred$fit
lo <- qnorm(0.025, mn, pred$se.fit)
up <- qnorm(0.975, mn, pred$se.fit)

par(mfrow=c(1,1))
plot(x, y, ylim=c(0, 1.3))
segments(0, 1, 10, 1, lty=2)
segments(0, 0, 10, 0, lty=2)
points(px, mn, type="l", lwd=2, col="red")
points(px, lo, type="l", lwd=2, lty=2, col="red")
points(px, up, type="l", lwd=2, lty=2, col="red")
```



The linear model predicts probabilities (y) less than 0 and greater than 1, so clearly it is inappropriate for analyzing probabilities. However, applying a logit transform to the data for analysis (and later back-transforming using the logistic function) can allow you to analyze proportions. Another (and sometimes better way) to accomplish this is to use a GLM with binomial family and logit link.

```

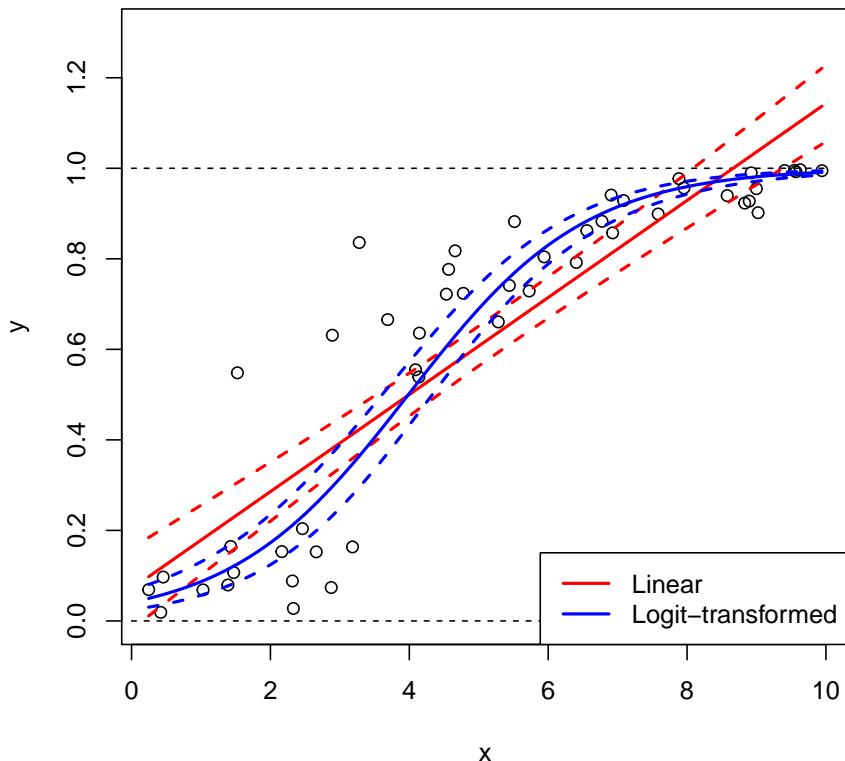
y2 <- qlogis(y)
mod2 <- lm(y2~x)
pred2 <- predict(mod2, newdata=data.frame(x=px), se.fit=TRUE)
mn2 <- pred2$fit
lo2 <- qnorm(0.025, mn2, pred2$se.fit)
up2 <- qnorm(0.975, mn2, pred2$se.fit)
# backtransform
mn2 <- plogis(mn2)
lo2 <- plogis(lo2)
up2 <- plogis(up2)

```

```

par(mfrow=c(1,1))
plot(x, y, ylim=c(0, 1.3))
segments(0, 1, 10, 1, lty=2)
segments(0, 0, 10, 0, lty=2)
points(px, mn, type="l", lwd=2, col="red")
points(px, lo, type="l", lwd=2, lty=2, col="red")
points(px, up, type="l", lwd=2, lty=2, col="red")
points(px, mn2, type="l", lwd=2, col="blue")
points(px, lo2, type="l", lwd=2, lty=2, col="blue")
points(px, up2, type="l", lwd=2, lty=2, col="blue")
legend("bottomright",
       legend=c("Linear", "Logit-transformed"),
       lwd=2, col=c("red", "blue"), bg="white")

```



The downside of the logit transform is that it is undefined at 0 and 1 (it

approaches $-\infty$ and $+\infty$ as the inputs approach 0 and 1). One workaround is to add a small constant to avoid taking $\log(0)$ or dividing by 0 (McCune et al. 2002, Warton and Hui 2011). The latter reference suggests adding the minimum non-zero value. You could also censor the values to be arbitrarily close to 0 or 1: for example, set all values < 0.001 to 0.001 and all values > 0.999 to 0.999. This practice is sometimes called “stabilizing the logit” (Kéry 2010).

```
# change some values in y to 1 and 0
y[which.max(y)] <- 1
y[which.min(y)] <- 0

# logit transformed values include infinities!
qlogis(y)

## [1] -2.5298619  3.7500756  0.2207000  2.4844936  5.3073196 -2.2318620
## [7]  0.6657197  4.5882685  2.0138961  1.2461916  4.8648903  0.9539238
## [13] 2.0198681  0.9887879 -2.6085359  3.0539803 -1.3622593          -Inf
## [19] 1.6283614  5.3667394  2.5484363  1.7931407  1.3371457  5.2369884
## [25] 1.8344244  2.5672970  1.0519484  1.4131947  0.5373003 -2.1224186
## [31]           Inf  2.2184900  2.7649033  3.0898599 -2.5994558  0.9631093
## [37] 2.1861231 -1.7101479 -1.6322178 -2.3345978 -1.6254712  0.5573073
## [43] 0.1559367  0.6886083  0.1934203 -2.4499857 -3.5614131  1.5004569
## [49] -1.7144060  2.7456992

# stabilize logit by setting max to 0.999 and min to 0.001
y <- pmin(0.999, y)
y <- pmax(0.001, y)
qlogis(y)

## [1] -2.5298619  3.7500756  0.2207000  2.4844936  5.3073196 -2.2318620
## [7]  0.6657197  4.5882685  2.0138961  1.2461916  4.8648903  0.9539238
## [13] 2.0198681  0.9887879 -2.6085359  3.0539803 -1.3622593 -6.9067548
## [19] 1.6283614  5.3667394  2.5484363  1.7931407  1.3371457  5.2369884
## [25] 1.8344244  2.5672970  1.0519484  1.4131947  0.5373003 -2.1224186
## [31] 6.9067548  2.2184900  2.7649033  3.0898599 -2.5994558  0.9631093
## [37] 2.1861231 -1.7101479 -1.6322178 -2.3345978 -1.6254712  0.5573073
## [43] 0.1559367  0.6886083  0.1934203 -2.4499857 -3.5614131  1.5004569
## [49] -1.7144060  2.7456992
```

The probit function is similar to the logit function, is used in similar situations, and produces similar results, but uses the CDF of the standard normal distribution instead of the logit function. The probit function is given by:

$$\text{probit}(x) = \left(\sqrt{2}\right) \text{erf}^{-1}(2x - 1) = \left(\sqrt{2}\right) \left(\frac{2}{\sqrt{\pi}}\right) \int_0^{2x-1} e^{-t^2} dt$$

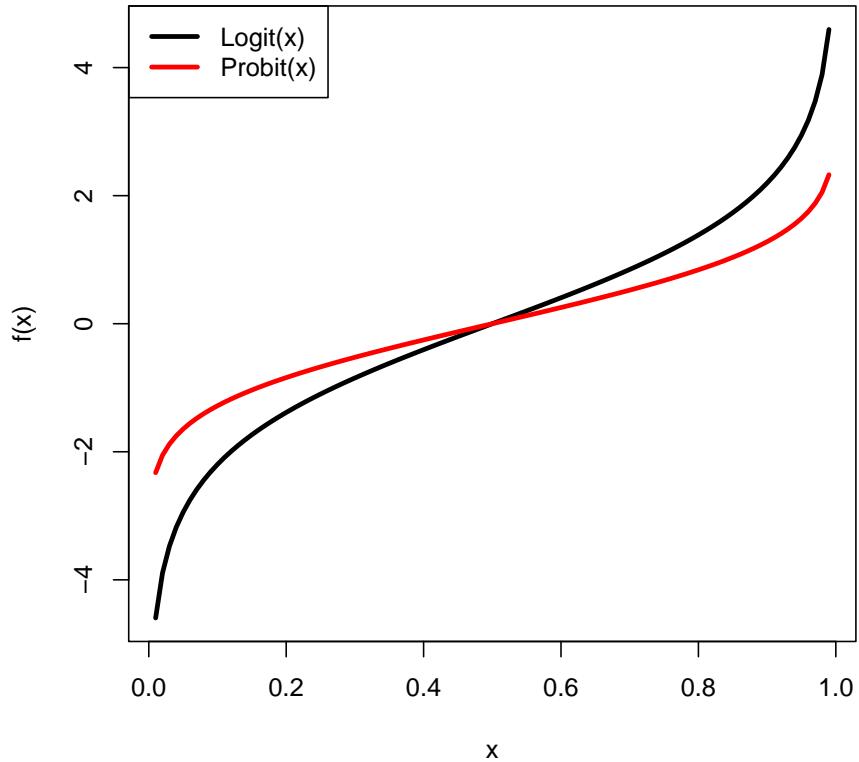
The R command for the probit function is just the function `qnorm()`. As you might suspect, the inverse function is `pnorm()`.

```

x <- 1:99/100
x.log <- qlogis(1:99/100)
x.prb <- qnorm(1:99/100)

plot(x, x.log, type="l",
      lwd=3, xlab="x", ylab="f(x)")
points(x, x.prb, type="l", lwd=3,
       col="red")
legend("topleft",
       legend=c("Logit(x)", "Probit(x)"),
       lwd=3, col=c("black", "red"))

```



4.5.5.3 Arcsine transformation

Another option for proportional data is to use the arcsine transform:

$$y_i = \frac{2}{\pi} \arcsin(\sqrt{x_i})$$

Some sources omit the coefficient $2/\pi$; others do not. Including the $2/\pi$ ensures that the transformed values range from 0 to 1 (McCune et al. 2002). It should be noted that the arcsine transformation should probably *not* be used when the data come from a binomial process. In those cases, logistic or binomial GLMs are likely to be more appropriate than an arcsine transformation. Simulation results have found that the arcsine transform should probably *not* be used for ecological data (Warton and Hui 2011). I couldn't find a reference on its acceptability in other areas of biology but I'd be surprised if it was appropriate given the arguments presented by Warton and Hui (2011).

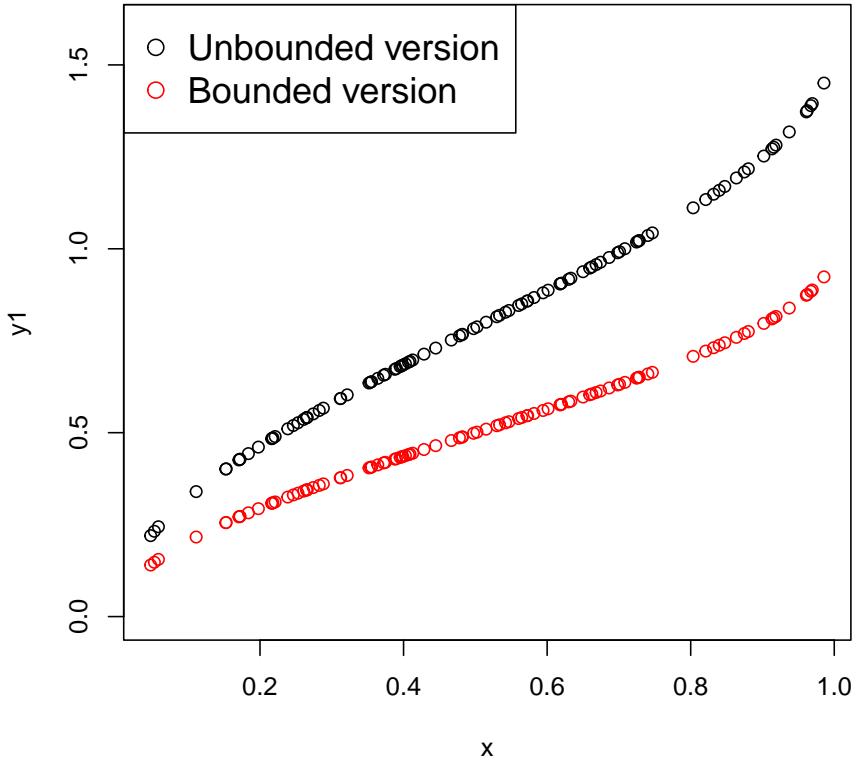
If you insist on using the arcsine transform, you can do it in R using `asin()` function. In the examples below, `asin()` is embedded inside custom functions that perform the other parts of the arcsine transform.

```
# unbounded version:
asin.trans1 <- function(p) {asin(sqrt(p))}

# bounded version:
asin.trans2 <- function(p) {2/pi*asin(sqrt(p))}

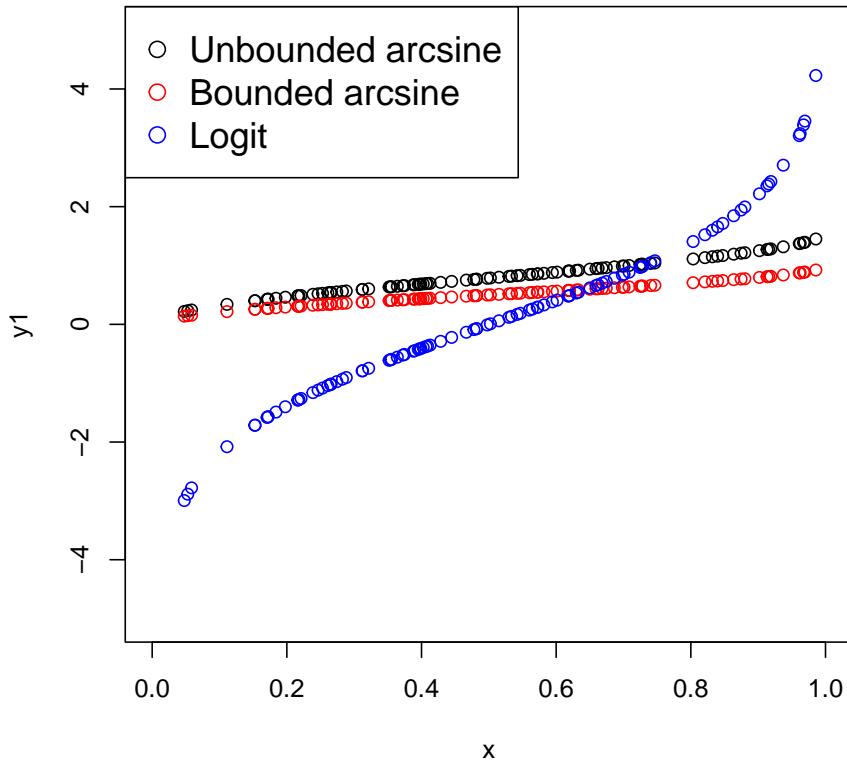
# example of use
x <- runif(100)
y1 <- asin.trans1(x)
y2 <- asin.trans2(x)

# compare with and without 2/pi coefficient:
plot(x,y1, ylim=c(0, 1.6))
points(x, y2, col="red")
legend("topleft",
       legend=c("Unbounded version", "Bounded version"),
       pch=1, col=c("black", "red"),
       cex=1.4)
```



The plot below compares the arcsine to the logit transformation for proportional data.

```
# compare to logit transform:
y3 <- qlogis(x)
plot(x,y1, ylim=c(-5, 5), xlim=c(0,1))
points(x, y2, col="red")
points(x, y3, col="blue")
legend("topleft",
       legend=c("Unbounded arcsine",
               "Bounded arcsine",
               "Logit"),
       pch=1, col=c("black", "red", "blue"), cex=1.4)
```



4.5.5.4 Box-Cox transformation

The **Box-Cox transformation** is a kind of power transform that can achieve a normal distribution in a non-normally distributed response variable (Box and Cox 1964).

$$y_i = \begin{cases} \log(x_i) & \text{if } \lambda = 0 \\ (x_i^\lambda - 1) / \lambda & \text{otherwise} \end{cases}$$

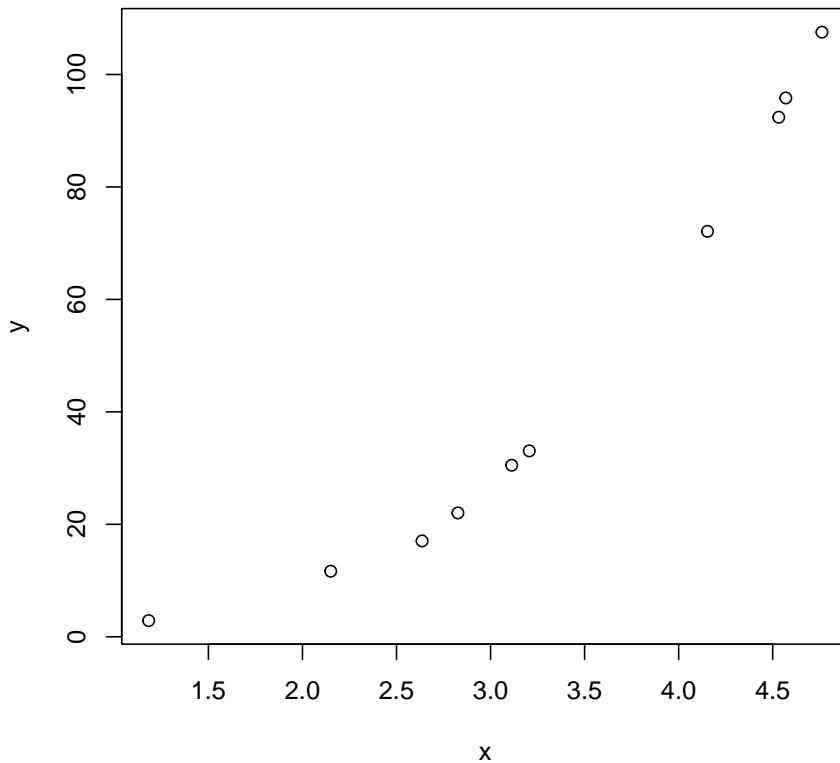
Here the parameter λ is some constant that must be optimized for a given variable. When $\lambda = 0$, the transformation is defined as $y_i = \log(x_i)$. Unlike the other transformations presented here, the Box-Cox transformation only makes sense in the context of a data model such as linear regression. The basic procedure is to first estimate the optimal λ for a given model, then use that λ to transform the data.

The Box-Cox transformation is available in several packages, but not base R. Here is an example using the functions in package MASS:

```
library(MASS)

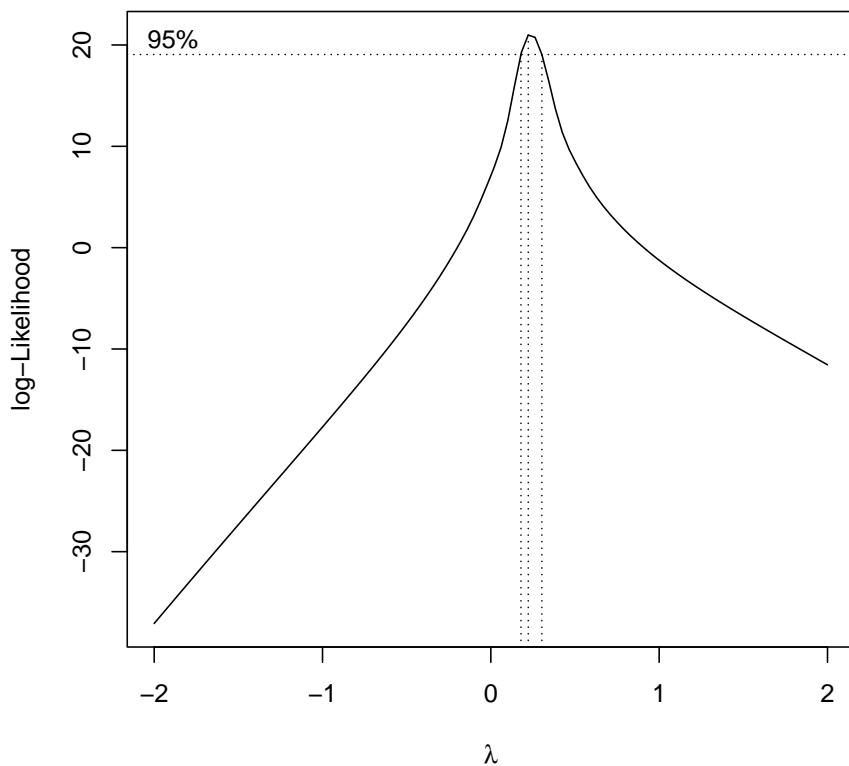
# generate some data
set.seed(123)
n <- 10
x <- runif(n, 1, 5)
y <- x^3 + rnorm(n)

plot(x,y)
```

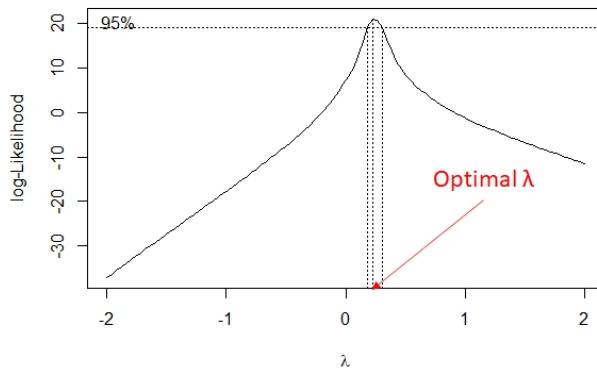


```
# fit a linear model
m1 <- lm(y ~ x)
```

```
# run the box-cox transformation  
bc <- boxcox(m1)
```



The figure below illustrates how the optimal λ is determined:



```

# get optimal lambda:
lambda <- bc$x[which.max(bc$y)]

# boxcox transformation
bc.trans <- function(x, lam){
  if(lam == 0){
    log(x)
  } else {
    ((x^lam)-1)/lam
  }
}

yt <- bc.trans(y, lambda)
m2 <- lm(yt~x)
# note difference in R-squared:
summary(m1)

##
## Call:
## lm(formula = y ~ x)
##
## Residuals:
##      Min       1Q   Median       3Q      Max 
## -12.037 -10.734  -1.496   6.798  22.138 
## 
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)    
## (Intercept) -56.86      12.34  -4.606  0.00174 ***
## x            31.80      3.53   9.009 1.84e-05 ***
## ---        
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

```

```

## 
## Residual standard error: 12.49 on 8 degrees of freedom
## Multiple R-squared:  0.9103, Adjusted R-squared:  0.8991
## F-statistic: 81.15 on 1 and 8 DF,  p-value: 1.84e-05
summary(m2)

## 
## Call:
## lm(formula = yt ~ x)
## 
## Residuals:
##      Min       1Q   Median       3Q      Max
## -0.15135 -0.04794 -0.01449  0.06522  0.11884
## 
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -1.08013   0.08491 -12.72 1.37e-06 ***
## x            1.96623   0.02428  80.97 6.04e-13 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
## 
## Residual standard error: 0.08589 on 8 degrees of freedom
## Multiple R-squared:  0.9988, Adjusted R-squared:  0.9986
## F-statistic: 6556 on 1 and 8 DF,  p-value: 6.035e-13

```

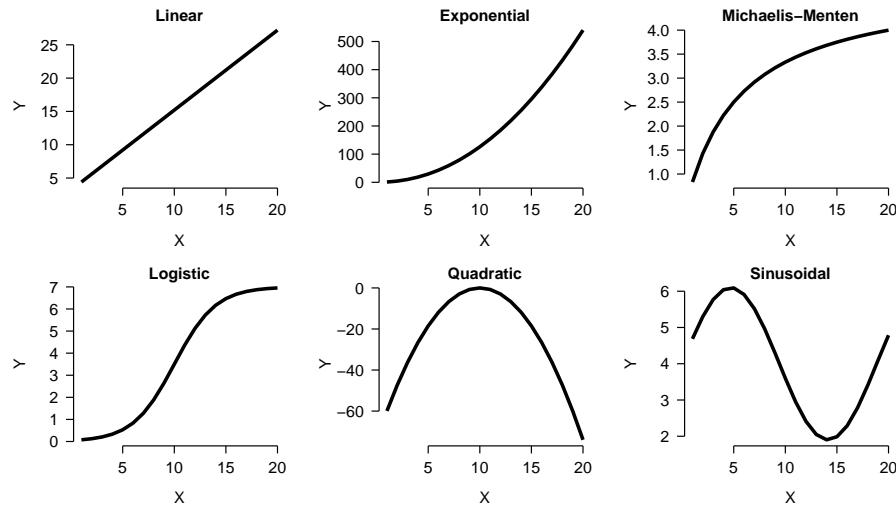
4.6 Multivariate data exploration

So far in this course we have explored ways to explore distributions of single variables: histograms, ECDF plots, probability density plots, and so on. On this page we will explore ways to explore relationships between variables. This is a vital preliminary step in studying how two or more variables might be correlated with each other, or how one might cause the other. The focus on this page is on exploratory, correlative methods. Actual inference about relationships between two variables is better handled by linear models (LM), generalized linear models (GLM), or other kinds of models. This page also contain a brief introduction to ordination, which can help identify patterns in datasets with many variables. A more complete presentation of ordination can be found on another page.

4.6.1 Scatterplots for two variables

The **scatterplot** is one of the most important tools in the biologist's data toolbox. It is probably the simplest type of figure in routine use: simply plot the values in one variable on one axis, and the values of another variable on another (perpendicular) axis. The role of the scatterplot in exploratory data analysis is to help visualize the form of the relationship between two variables (Bolker

2008). The figure below shows just 6 of the possible curve types that you might discover.

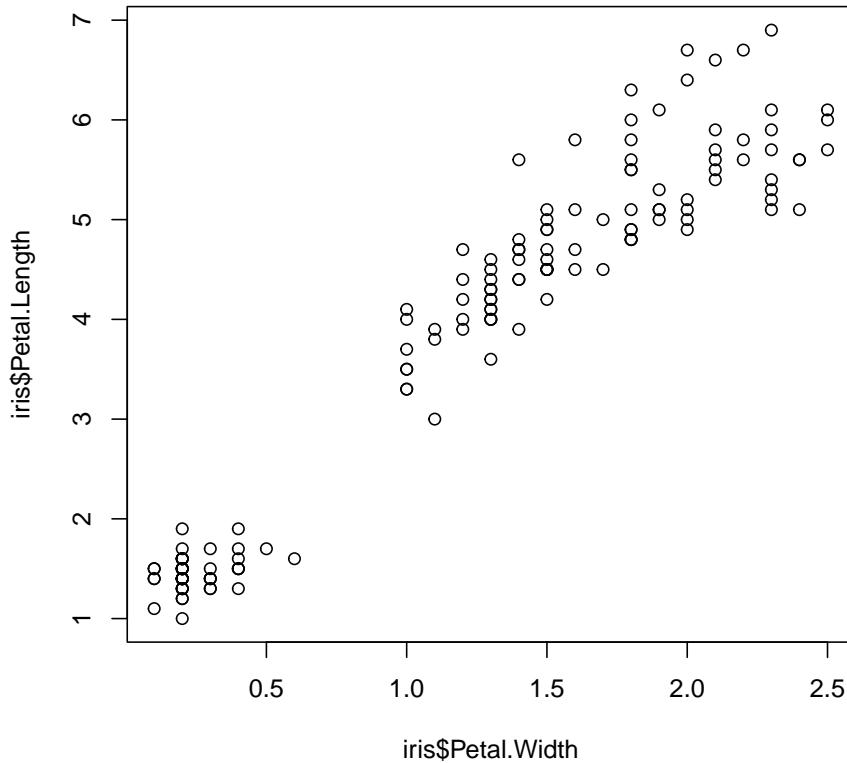


We have already made some scatterplots, but it's worth taking a closer look at the R methods for scatterplots. The basic function for scatterplots is `plot()`. The first argument is taken to be the x coordinates, and the second argument to be the y coordinates. You can also use the formula interface (`y ~ x`), but the coordinates method (`x, y`) usually makes for cleaner code.

Calling `plot()` does two things: first, it *creates a new plot*; second, it *plots the data* on the new plot area. Other components can be added to the plot with subsequent commands: `points()` for points, `text()` for text, `lines()` for lines, `polygon()` for polygons, `segments()` for line segments, and so on. These functions will add to an existing plot, but will not create a new plot area. So, call `plot()` first, then `points()` or one of the others if needed. Calling `points()` or one of the “adding” functions if there is not already an active plot area will return an error.

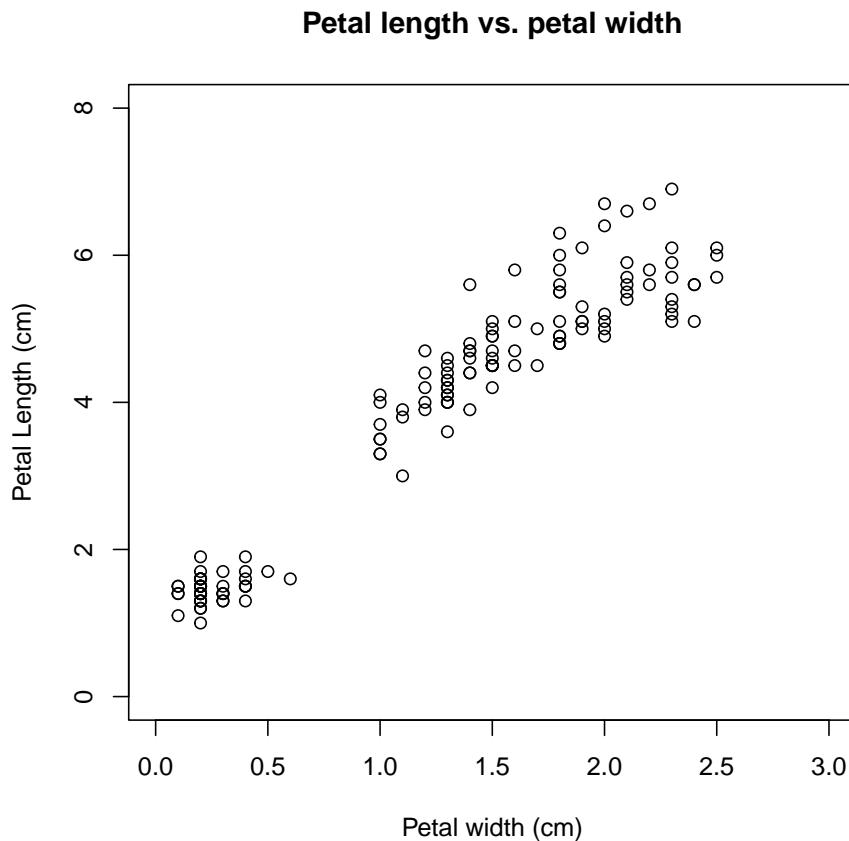
Here is a basic scatterplot:

```
plot(iris$Petal.Width,iris$Petal.Length)
```



The default scatterplot is rather unattractive, but works just fine for exploring data. Fortunately, almost every aspect of a plot can be customized using different plot arguments and graphical parameters. Here are some commonly used plot options:

```
plot(iris$Petal.Width,iris$Petal.Length,
  xlab="Petal width (cm)",           # x label
  ylab="Petal Length (cm)",          # y label
  xlim=c(0,3),                      # x axis limits
  ylim=c(0,8),                      # y axis limits
  main="Petal length vs. petal width" # plot title
) #plot
```



Sometimes it helps to change the color or style of points to show categories in the data. This can be done using some arguments to `plot()`:

- `pch` changes the symbol used for points. Think “**point character**”. See `?points` for a list of available symbols.
- `cex` changes the size of symbols. Think “**character expansion**”. The default `cex` is 1; other values scale the points relative to this. E.g., `cex=2` makes points twice as large.
- `col` changes the color. Think “**color**”. R can produce many colors; run the command `colors()` to see a named list¹⁵.

Each of these arguments can take vector of values, so you can assign colors or shape to each point. The values will be used in the same order as the observations used to draw the points (e.g., rows of a data frame). I find it convenient to use

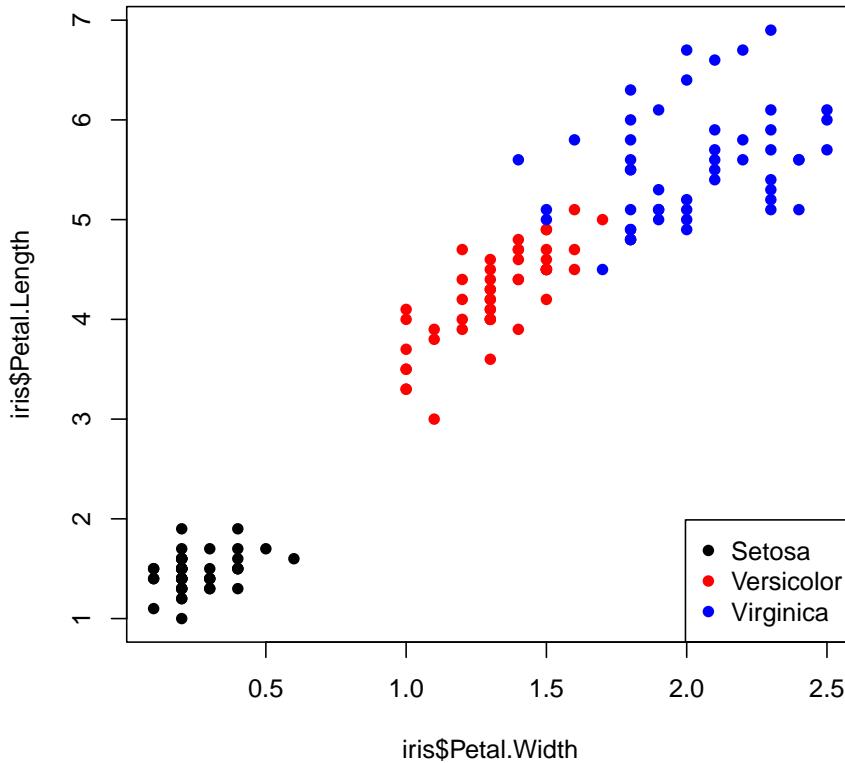
¹⁵You can also specify colors by their RGB or hex codes. A graphical reference is here (accessed 2021-12-17). The ColorBrewer website (accessed 2021-12-17) and R package can help picking appropriate color schemes.

rules like `ifelse()` commands to define symbology for plots.

The examples below illustrate several ways of assigning symbology to observations within a data frame. In the first, `ifelse()` is used to make vectors of colors (`use.col`) defined by the variable `species`. Notice that a legend is provided to tell the reader which symbols mean what.

```
# method 1: ifelse()
use.col <- ifelse(iris$Species == "setosa", "black",
                  ifelse(iris$Species == "versicolor", "red", "blue"))

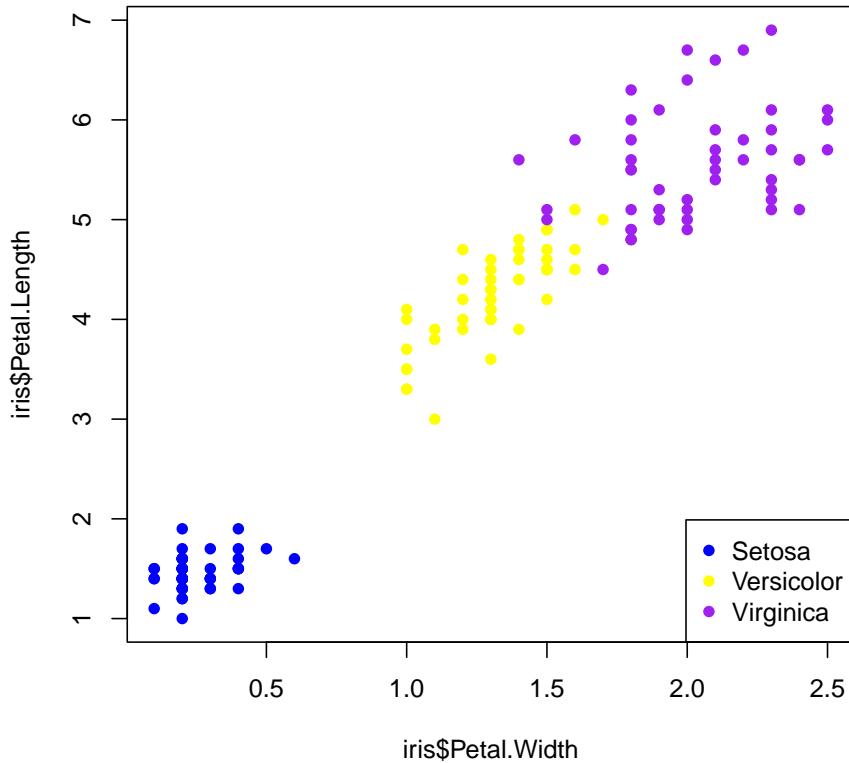
plot(iris$Petal.Width,iris$Petal.Length,
      col=use.col, # color by species
      pch=16       # solid dots
)
legend("bottomright",
       legend=c("Setosa", "Versicolor", "Virginica"),
       pch=16, col=c("black", "red", "blue"))
```



The next example uses `match()` instead of `ifelse()` to assign colors by species.

```
# method 2: match()
spps <- sort(unique(iris$Species))
cols <- c("blue", "yellow", "purple")
use.col <- cols[match(iris$Species, spps)]

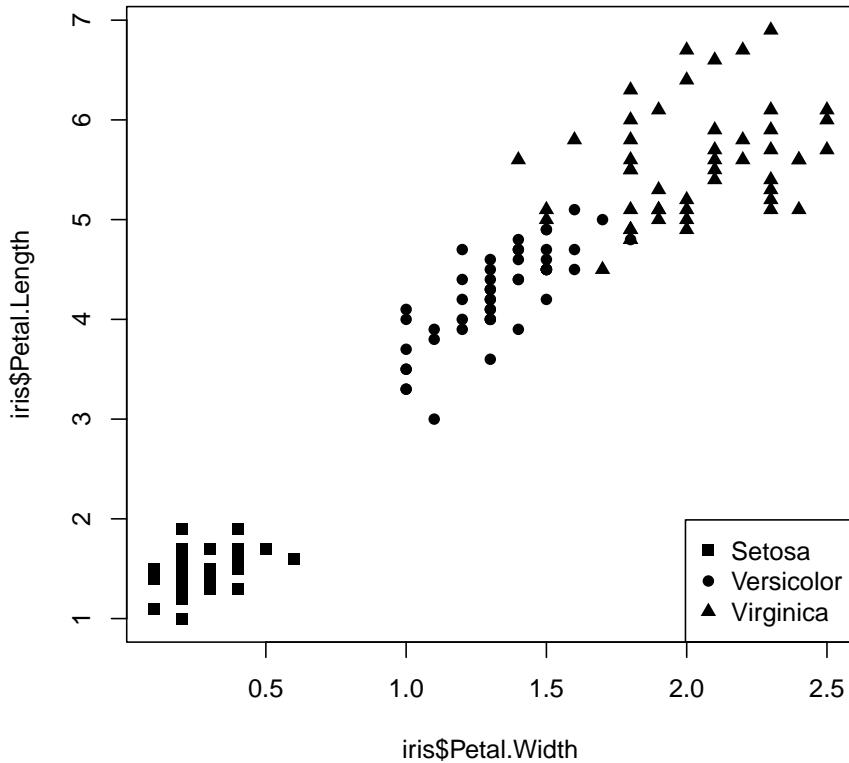
plot(iris$Petal.Width,iris$Petal.Length,
      col=use.col, # color by species
      pch=16        # solid dots
)
legend("bottomright",
       legend=c("Setosa", "Versicolor", "Virginica"),
       pch=16, col=cols)
```



Here is an example of using shapes (`use.pch`) to define groups:

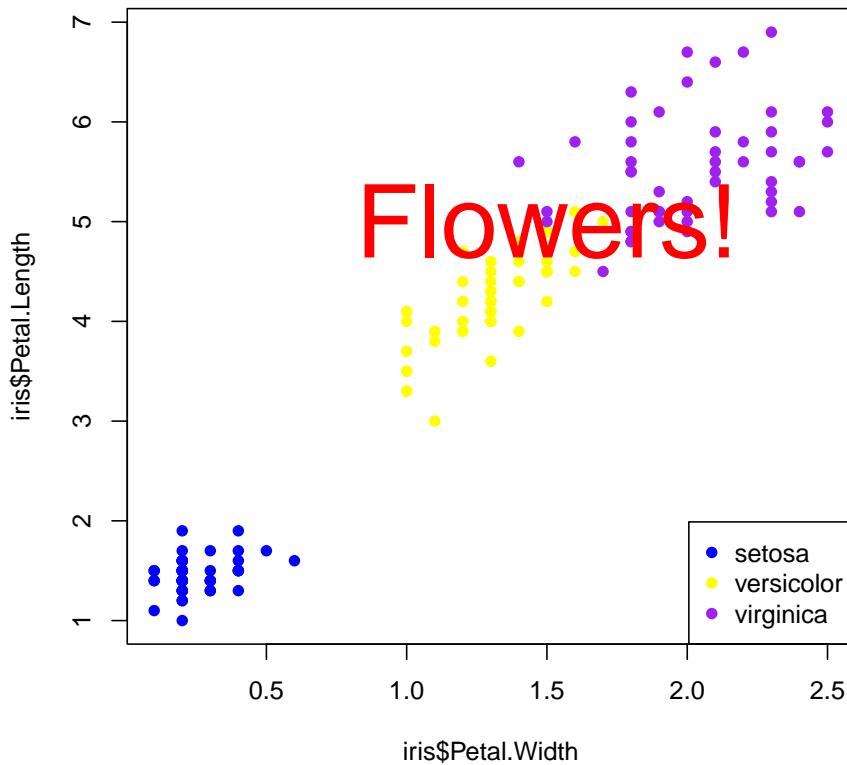
```
# method 3: match(), but with shape
spps <- sort(unique(iris$Species))
shps <- c(15, 16, 17)
use.shp <- shps[match(iris$Species, spps)]

plot(iris$Petal.Width,iris$Petal.Length,
      pch=use.shp # shape by species
)
legend("bottomright",
      legend=c("Setosa", "Versicolor", "Virginica"),
      pch=shps)
```



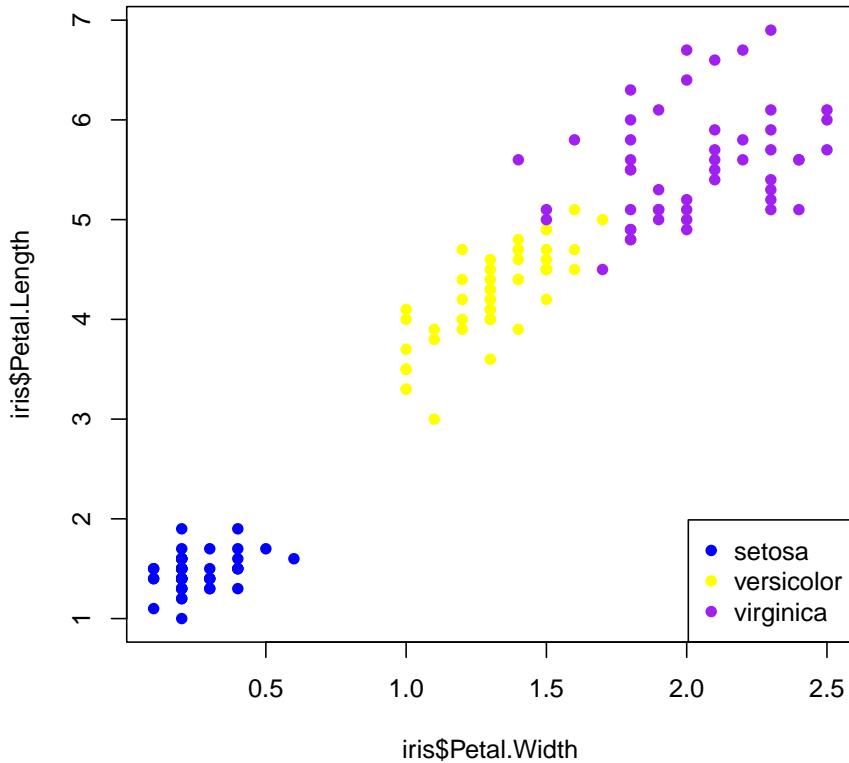
You can think of the plot generated by `plot()` like a canvas. Once the canvas is in place, things can be “painted” on, but like paint cannot be removed. If you want to remove something, you’ll need to remake the plot without it. You can also just comment out the command you don’t want.

```
plot(iris$Petal.Width,iris$Petal.Length,
     col=use.col, pch=16
)
text(1.5, 5, "Flowers!", cex=4, col="red")
legend("bottomright", legend=spps, col=cols, pch=16)
```



There's no way to remove the offending text: R doesn't have an "Undo" button! Our only option is just to remake the plot without the text.

```
# remake without the offending text:
plot(iris$Petal.Width,iris$Petal.Length,
      col=use.col, pch=16
)
legend("bottomright", legend=spps, col=cols, pch=16)
```



Sometimes I'll just comment out the plot component that needs deleting; this way it is easier to add back in later.

```
# not run:  
  
# remake without the offending text (commented out):  
plot(iris$Petal.Width,iris$Petal.Length,  
      col=use.col, pch=16  
)  
#text(1.5, 5, "Flowers!", cex=4, col="red")  
legend("bottomright", legend=spps, col=cols, pch=16)
```

4.6.1.1 Graphical parameters and the `par()` command

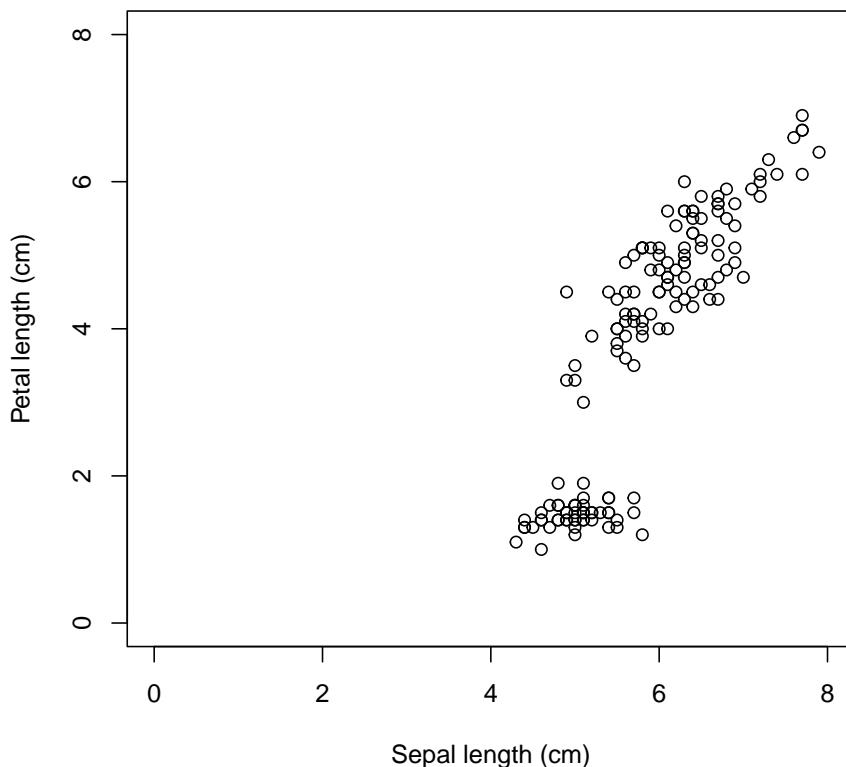
Many options that affect plots can only be set using the `par()` function. Take a look at the `par()` help page to get a sense of the variety of options available (`?par`). It is important to keep in mind is that once `par()` options are set,

they will stay that way until you change them using `par()` again. So, if you are making multiple figures within the same R workspace you will need to pay attention to what you have done. You can always see all current settings by running the command `par()`.

Below is an illustration of using `par()` to change graphics options:

Without `par()`:

```
plot(iris$Petal.Length~iris$Sepal.Length,
     xlab="Sepal length (cm)",
     ylab="Petal length (cm)",
     xlim=c(0,8), ylim=c(0,8))
```

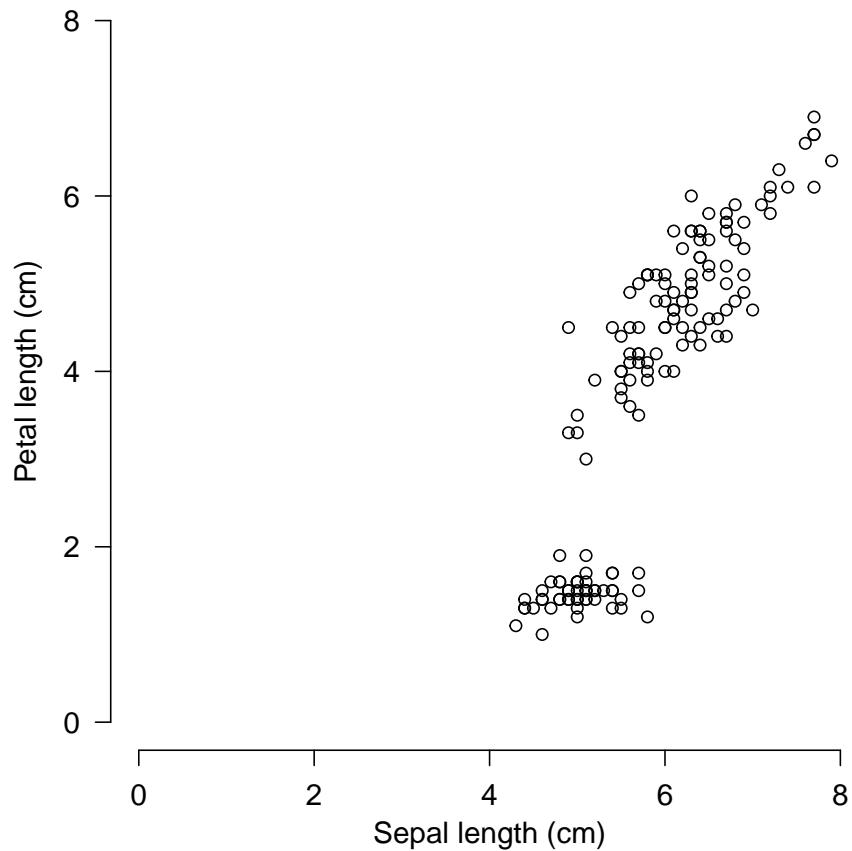


With `par()`:

- Narrow the margins (`mar`)

- Remove the box (`bty`)
- Turn y -axis numbers to horizontal (`las`)
- Bring axis labels in closer (`mgp`)
- Set size of axis text (`cex.axis`) and labels (`cex.lab`)

```
par(mar=c(4.1, 4.1, 1.1, 1.1), # margin sizes
    bty="n",                      # no box around plot
    las=1,                        # axis labels in reading direction
    mgp=c(2.25, 1,0),            # position of axis components
    cex.axis=1.2,                 # size of axis numbers
    cex.lab=1.2)                  # size of axis titles
plot(iris$Petal.Length~iris$Sepal.Length,
      xlab="Sepal length (cm)",
      ylab="Petal length (cm)",
      xlim=c(0,8), ylim=c(0,8))
```



We will use `par()` often in this course to clean up figures and make them more attractive. Most of the time you won't need to mess with `par()` (except for making multi-panel figures), but you should definitely use `par()` when preparing figures for publication. Proper use of `par()` is the key to making clean, informative, and professional-looking figures.

4.6.1.2 Multi-panel figures with `par()`

One of the most common ways to use `par()` is to make multi-panel figures. The panels are specified in terms of the number of rows and columns in the figure. The arguments `mfrow` and `mfcol` define the panel layout.

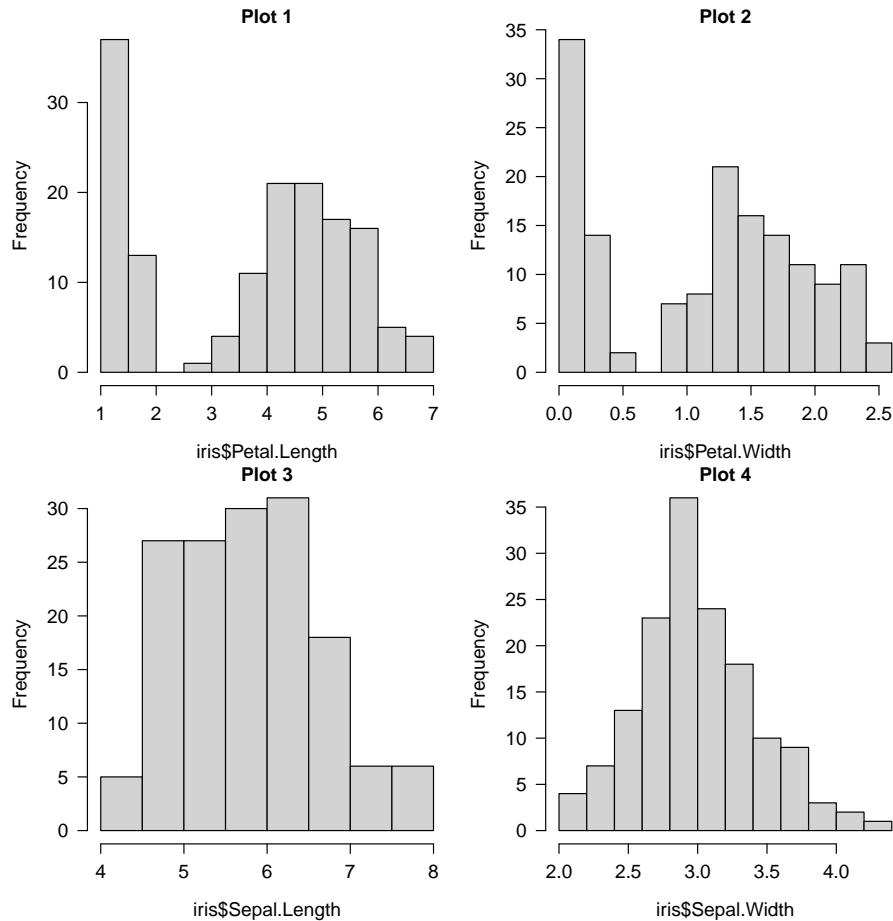
- `mfrow`: you supply the number of rows and number of columns, in that order.
- `mfcol`: works the same way, but in reverse: supply the number of columns and number of rows, in that order.

Once you set `mfrow` or `mfcol` (but never both), a new plot panel will be produced each time you call `plot()`. The graphics window will be divided evenly according to the number of panels you requested (e.g., `mfrow=c(2,3)` will yield 6 panels). Panels are drawn by row and then by column (with `mfrow`) or by column and then by row (with `mfcol`).

If you produce more plots than you have “slots” specified by `mfrow` or `mfcol`, the graphics device will be cleared and the plot panels will be filled again, in the same order as before. So, if you set `mfrow=c(2,2)`, and then make 5 plots, you will end up with a graphics window that has 1 plot in the upper left corner.

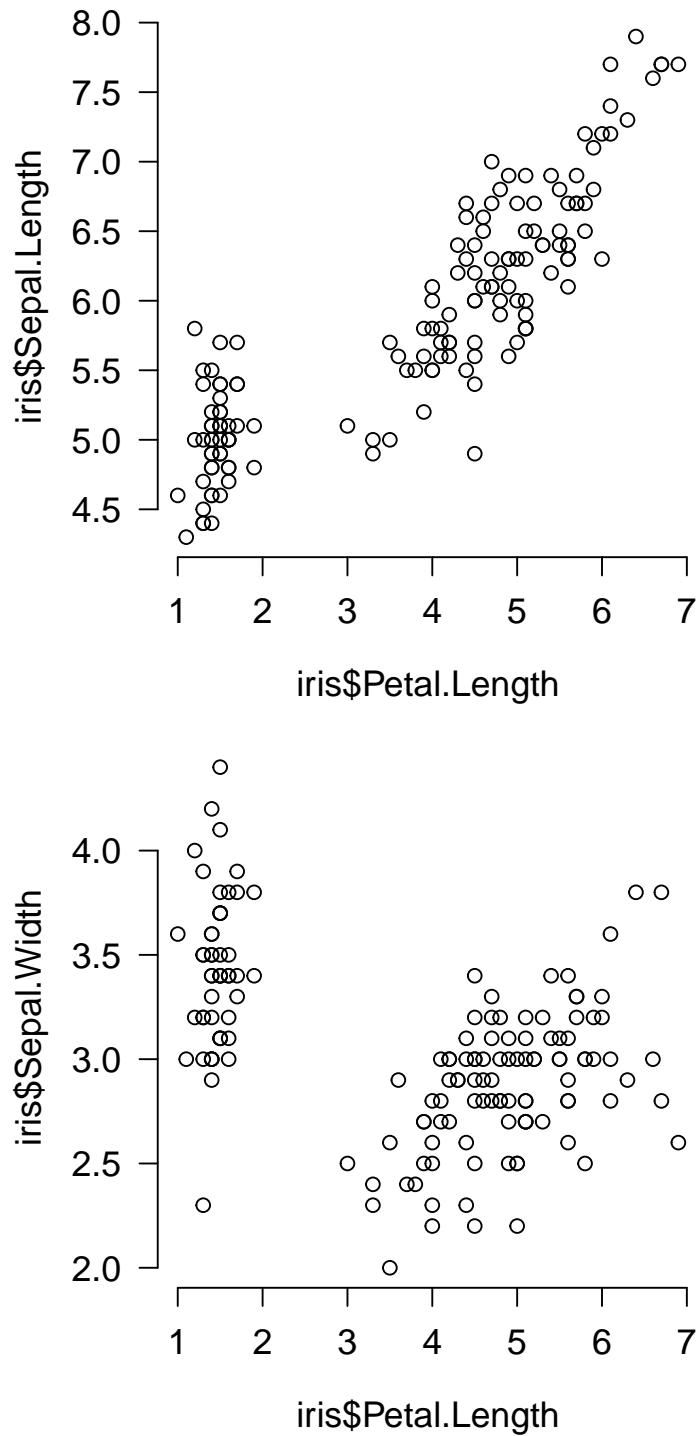
The example below shows the use of `par()$mfrow` to make a 2×2 figure.

```
par(mfrow=c(2,2),           # layout
    mar=c(4.1,4.1,1.1,1.1), # margin sizes
    bty="n",                 # no box around plot
    las=1,                   # rotate axis text
    cex.axis=1.2,             # axis text size
    cex.lab=1.2)              # label text size
hist(iris$Petal.Length, main="Plot 1")
hist(iris$Petal.Width, main="Plot 2")
hist(iris$Sepal.Length, main="Plot 3")
hist(iris$Sepal.Width, main="Plot 4")
```



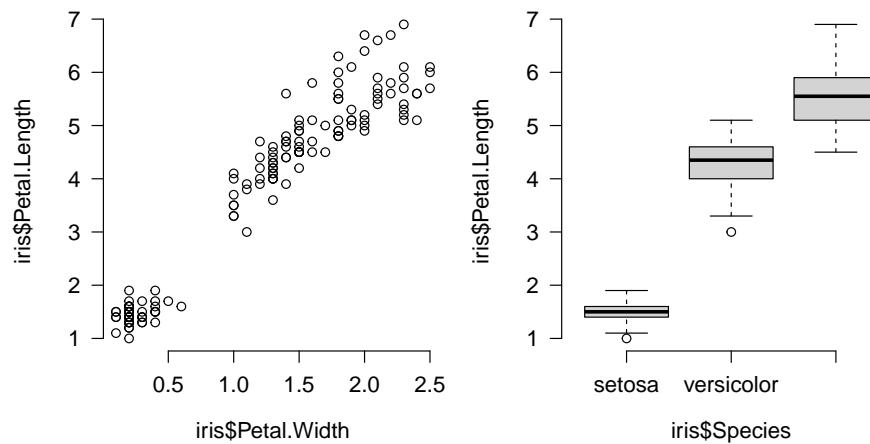
When making multi-panel plots, it helps to line up axes that correspond to each other. The commands below show two separate figures with aligned axes. This can be helpful when showing different responses to the same explanatory variable (the 2×1 figure):

```
par(mfrow=c(2,1),          # layout
  mar=c(4.1,4.1,1.1,1.1), # margin sizes
  bty="n",                 # no box around plot
  las=1,                   # rotate axis text
  cex.axis=1.2,            # axis text size
  cex.lab=1.2)             # label text size
plot(iris$Petal.Length, iris$Sepal.Length)
plot(iris$Petal.Length, iris$Sepal.Width)
```



The figure below is a 1×2 figure that shows how a single response variable relates to two different predictors.

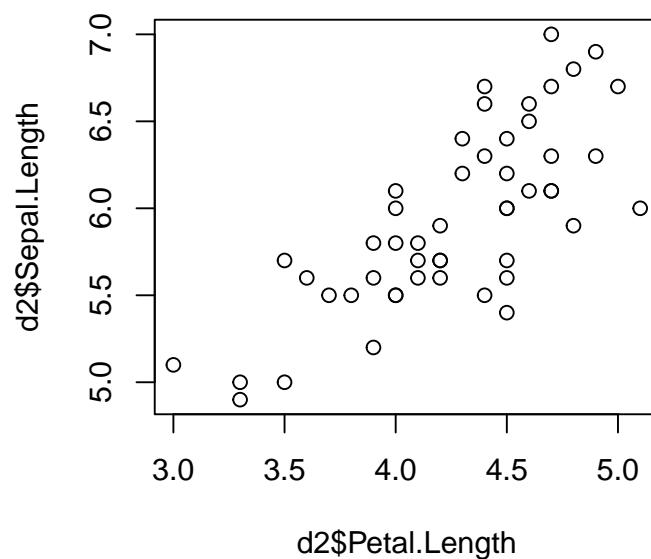
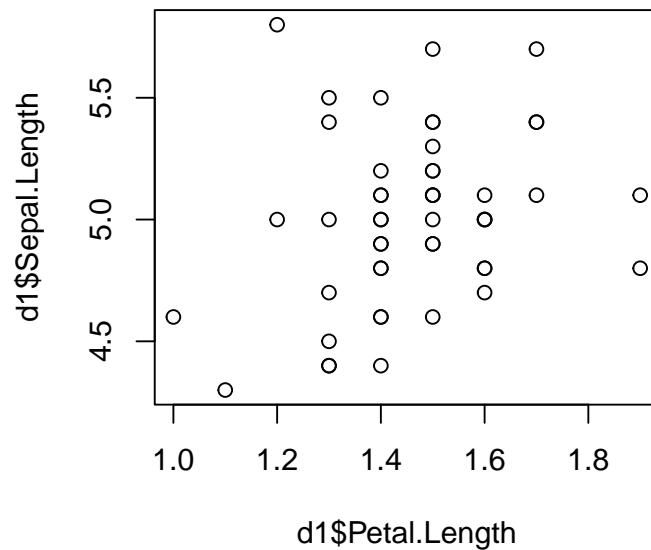
```
par(mfrow=c(1,2),          # layout
    mar=c(4.1,4.1,1.1,1.1), # margin sizes
    bty="n",                 # no box around plot
    las=1,                   # rotate axis text
    cex.axis=1.2,             # axis text size
    cex.lab=1.2)              # label text size
plot(iris$Petal.Width, iris$Petal.Length)
boxplot(iris$Petal.Length~iris$Species)
```



Sometimes you may need to manually set axis limits to make sure that the panels line up exactly.

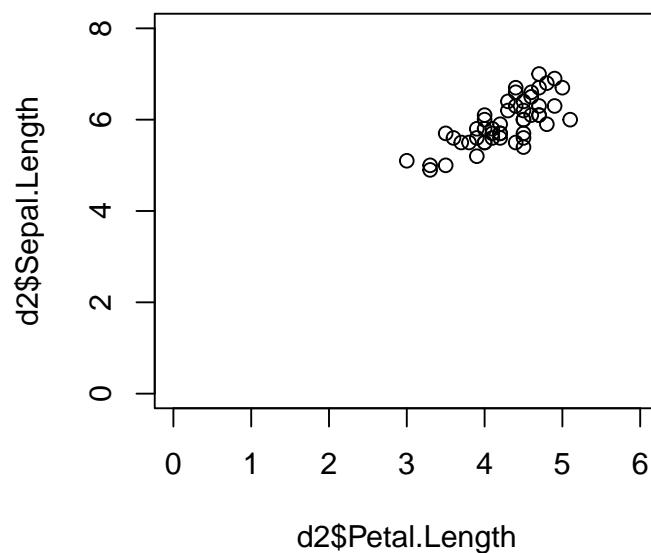
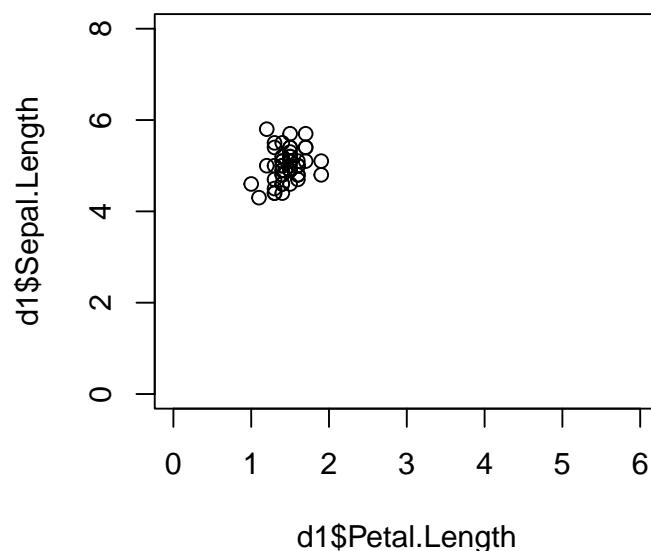
```
# example with X and Y axes not aligned:
d1 <- iris[which(iris$Species == "setosa"),]
d2 <- iris[which(iris$Species == "versicolor"),]

par(mfrow=c(2,1))
plot(d1$Petal.Length, d1$Sepal.Length)
plot(d2$Petal.Length, d2$Sepal.Length)
```

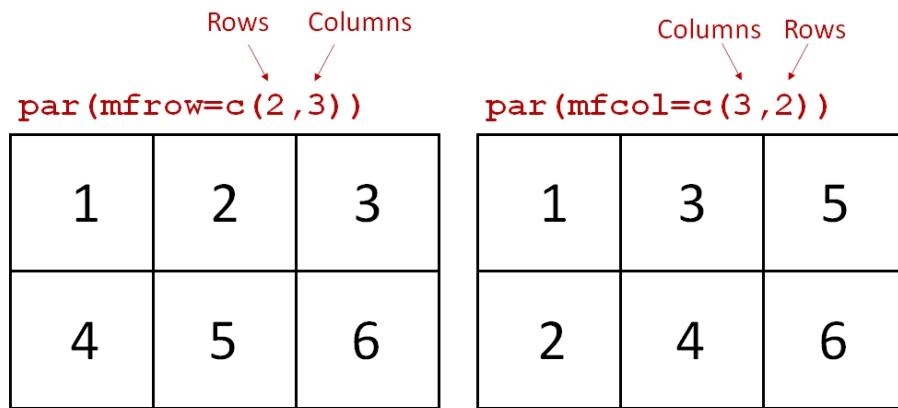


Here's the same plot, but with the axes lined up by setting the axis limits with `xlim` and `ylim`.

```
# same plot but with axes lined up:  
par(mfrow=c(2,1))  
plot(d1$Petal.Length, d1$Sepal.Length, xlim=c(0,6), ylim=c(0,8))  
plot(d2$Petal.Length, d2$Sepal.Length, xlim=c(0,6), ylim=c(0,8))
```



The argument `mfcoll` to `par()` works similarly to `mfrow`, but on columns instead of rows. For `mfcoll` you supply the number of columns, then the number of rows. Likewise, panels are filled by column first instead of row first. The panel layout `mfrow=c(2,3)` is the same as `mfcoll=c(3,2)`, but the panels will be filled in a different order:

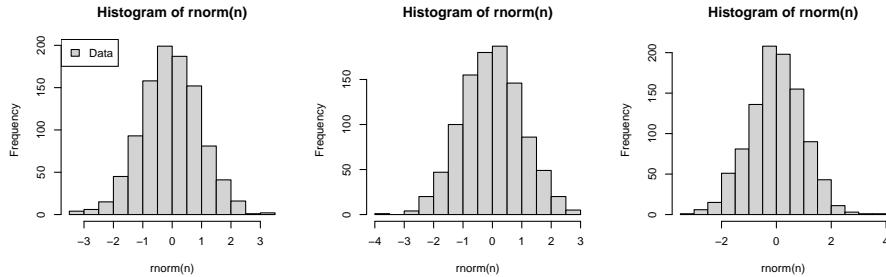


Whether to use `mfrow` or `mfcoll` is usually a matter of preference. Clever use of one or the other can allow you to automatically make multi-panel plots in a preferred layout from data stored in a list or indexed by a vector.

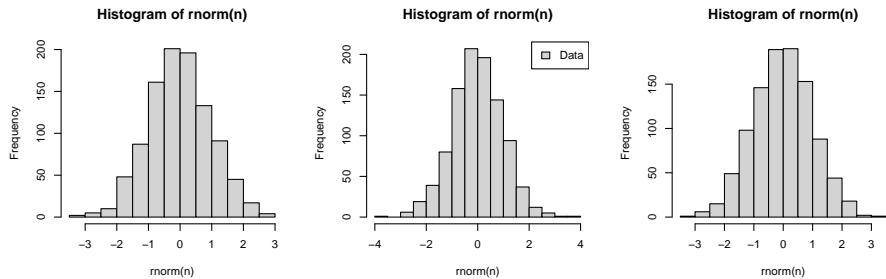
One last thing about multi-panel plots: the `plot()` function both creates new panels and the plot within each panel. Part of this process is defining the coordinate system for a panel. When you make a plot, subsequent commands that add elements to plots such as `points()`, `abline()`, `legend()`, etc., will use the coordinate system of the most recent panel. Think about this when designing complicated figures. The example below demonstrates how a legend is placed according to the coordinate system of the most recently-created plot.

```
n <- 1e3

# legend in first panel:
par(mfrow=c(1,3))
hist(rnorm(n))
legend("topleft", legend="Data", fill="lightgrey")
hist(rnorm(n))
hist(rnorm(n))
```



```
# legend in second panel (note where legend() is):
par(mfrow=c(1,3))
hist(rnorm(n))
hist(rnorm(n))
legend("topright", legend="Data", fill="lightgrey")
hist(rnorm(n))
```



4.6.2 Scatterplot matrices for many variables

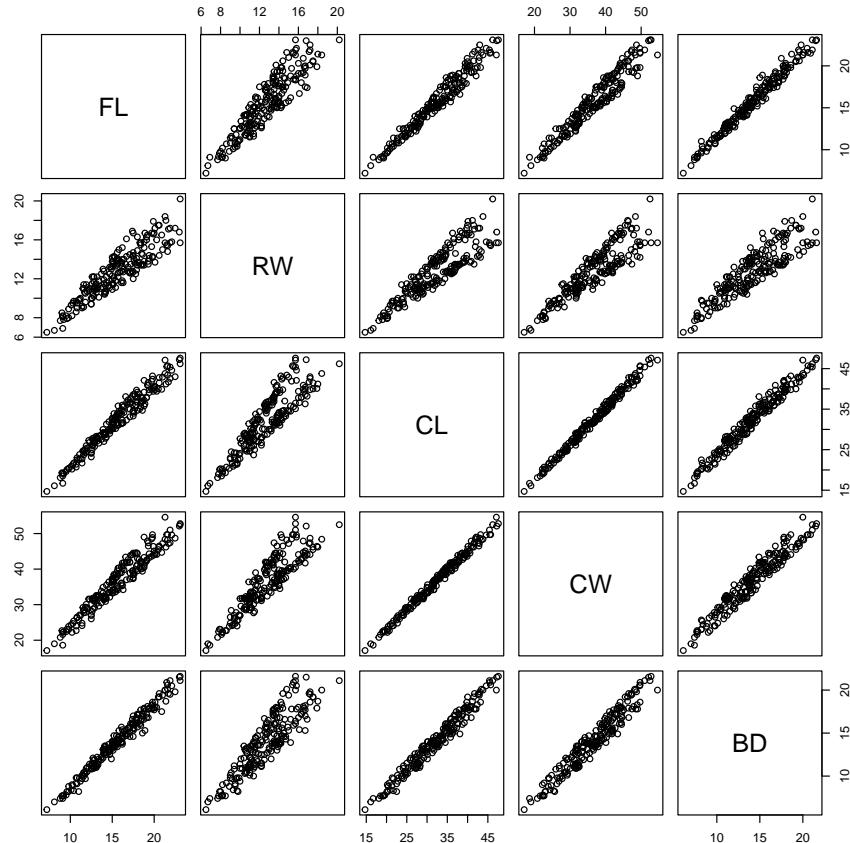
A scatterplot plots one variable against another, and is probably the best way to see the relationship (if any) between two variables. But what if you have many variables? Making dozens of scatterplots can be tedious and time consuming. A scatterplot matrix makes many scatterplots at once, allowing relationships between many variables to be visualized at once. The base function to do this is `pairs()`.

```
# load some data
data(crabs, package="MASS")

# make a spare copy
x <- crabs

# define columns for scatterplot matrix
dat.cols <- 4:8
```

```
# make scatterplot matrix
pairs(x[,dat.cols])
```



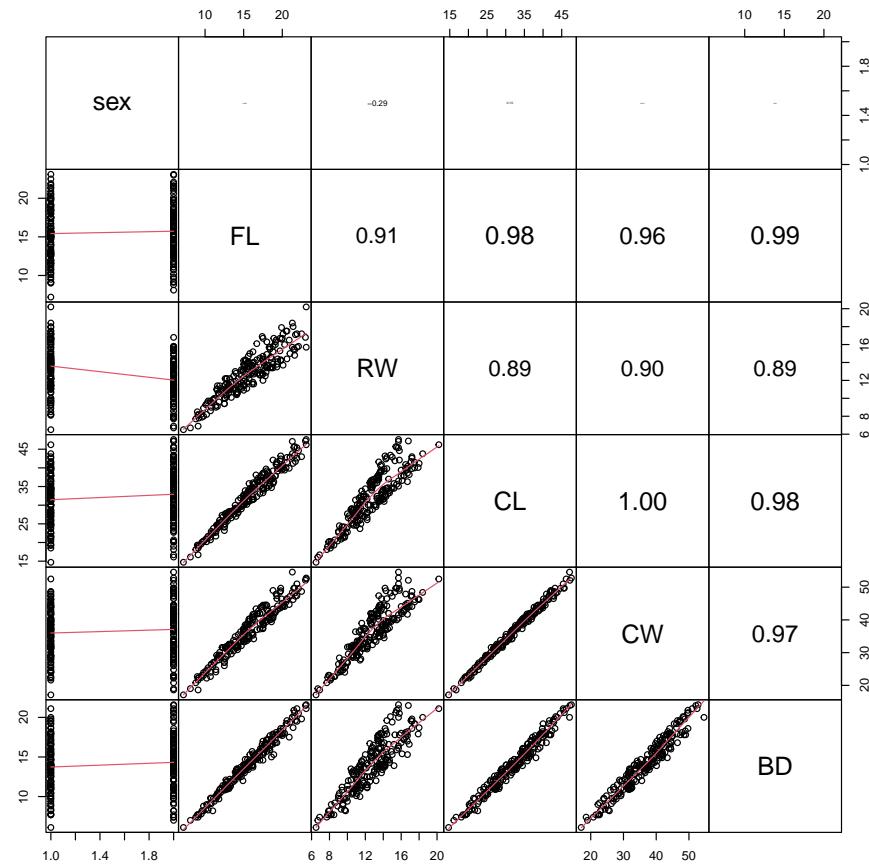
In the scatterplot matrix, every variable is plotted against every other variable. Variables are labeled on the diagonal. E.g., the plots in the first row have “FL” as their y -axis, and “RW”, “CL”, “CW”, and “BD” as their x -axes. Likewise, plots in the first column have “FL” as their x -axis, and “RW”, “CL”, “CW”, and “BD” as their y -axes.

One common modification to the default `pairs()` plot is to replace scatterplots above the diagonal with correlation coefficients. Another modification is to add linear regression or LOESS¹⁶ lines to the lower plots to help highlight the relationships. The function below is adapted from the help page for `pairs()`. Notice that the size of the text for each coefficient is scaled to the magnitude of the coefficient. Can you figure out what piece of the code is doing that?

¹⁶Locally estimated sums of squares, a common smoothing curve algorithm.

```
# define a function to add correlation coefficients
panel.cor <- function(x, y, digits = 2, prefix = "", cex.cor, ...)
{
  usr <- par("usr"); on.exit(par(usr))
  par(usr = c(0, 1, 0, 1))
  r <- cor(x, y)
  txt <- format(c(r, 0.123456789), digits = digits)[1]
  txt <- paste0(prefix, txt)
  if(missing(cex.cor)) cex.cor <- 0.8/strwidth(txt)
  text(0.5, 0.5, txt, cex = 2*abs(r))
}

# example of use:
pairs(x[,c(2, dat.cols)],
      lower.panel=panel.smooth,
      upper.panel=panel.cor, gap=0)
```

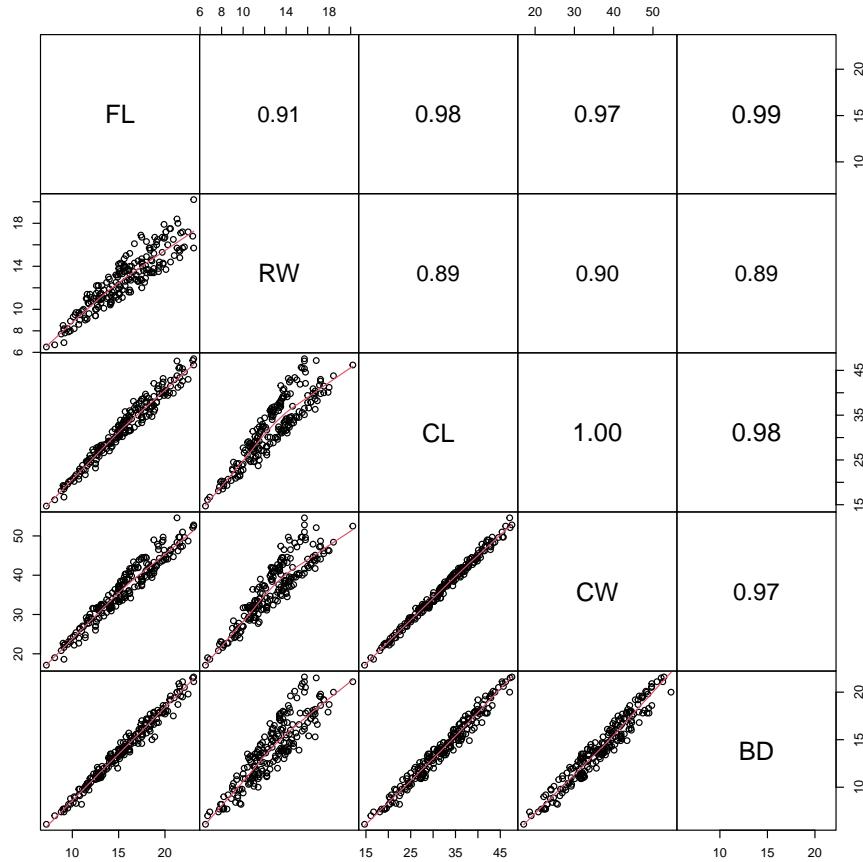


Sometimes the Spearman rank correlation coefficient ρ is more informative than the Pearson linear correlation coefficient r . The function `panel.cor()` can be modified to use ρ instead of r :

```
panel.cor2 <- function(x, y, digits = 2, prefix = "", cex.cor, ...)
{
  usr <- par("usr"); on.exit(par(usr))
  par(usr = c(0, 1, 0, 1))
  r <- cor.test(x, y, method="spearman")$estimate
  txt <- format(c(r, 0.123456789), digits = digits)[1]
  txt <- paste0(prefix, txt)
  if(missing(cex.cor)) cex.cor <- 0.8/strwidth(txt)
  text(0.5, 0.5, txt, cex = 2*abs(r))
}

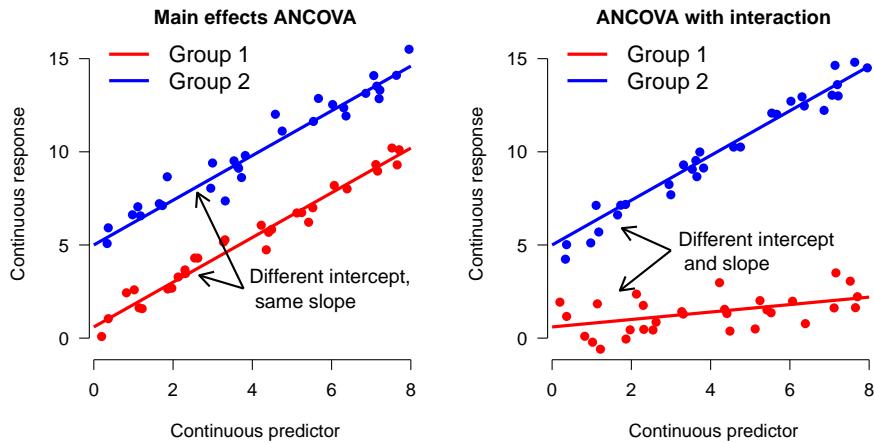
# example of use:
# (might return warnings but these are ok)
```

```
pairs(x[,dat.cols],
      lower.panel=panel.smooth,
      upper.panel=panel.cor2, gap=0)
```



4.6.3 Lattice plots for hierarchical data

In another module we will explore mixed models, also known as hierarchical models, which can account for relationships that differ between groups of data. These are similar in some ways to analysis of covariance (ANCOVA) models with interactions. An example of what ANCOVA can look like is shown below:

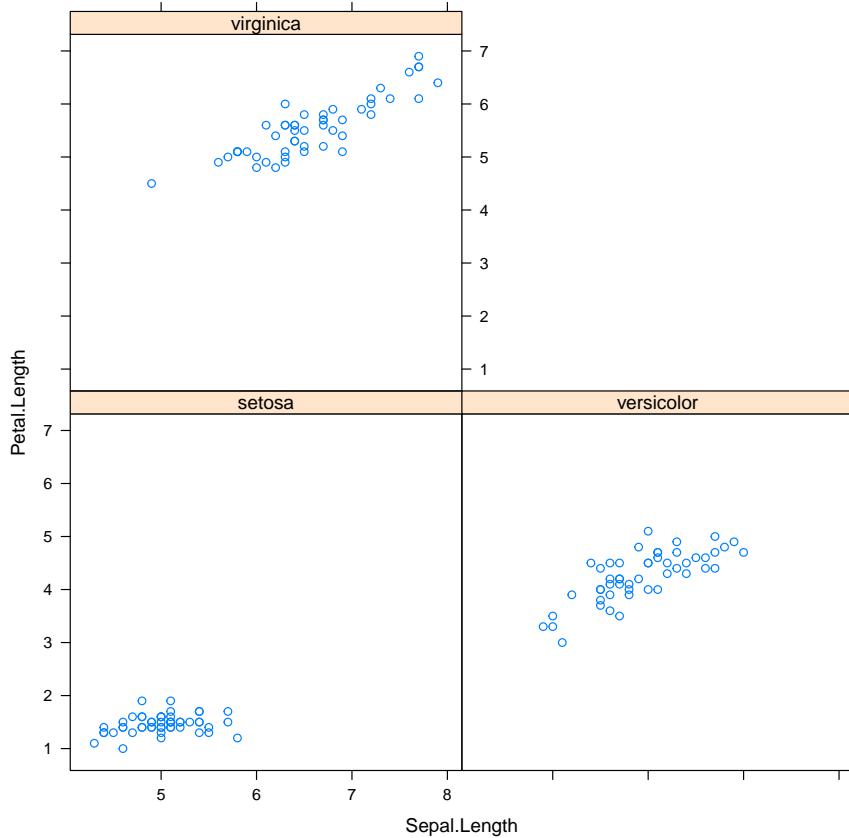


In the left panel, the Y variable increases as a function of the X variable. There is also an effect of the group variable (i.e., a factor): Y values in group 2 are on average greater than values in group 1. This is called “main effects” ANCOVA. In the right panel, Y increases with X , and the slope of Y with respect to X varies between groups. In other words, X has a greater effect on Y in group 2 than in group 1. This is called an “interaction” between X and the grouping variable. Analyses that include such interactions are sometimes called “interaction effects” ANCOVA because the effects of one variable “interact with” the effects of another variable. Both types of ANCOVA are extremely common in biology.

One key difference between an ANCOVA model and a mixed model is that while in ANCOVA the slope and intercept are fitted specifically to each level of the grouping variable, consuming degrees of freedom, in a mixed model these slopes and intercepts are considered to be drawn from a random distribution. For now, all you need to understand is that the relationship between response and predictor variables can differ between groups. A **lattice** or **trellis** plot is a way to visualize such differences. In R these are named for package **lattice** which is an older package for producing hierarchical plots. The newer **tidyverse** packages can also do the job (and are becoming much more popular than **lattice**, from what I can tell).

```
library(lattice)

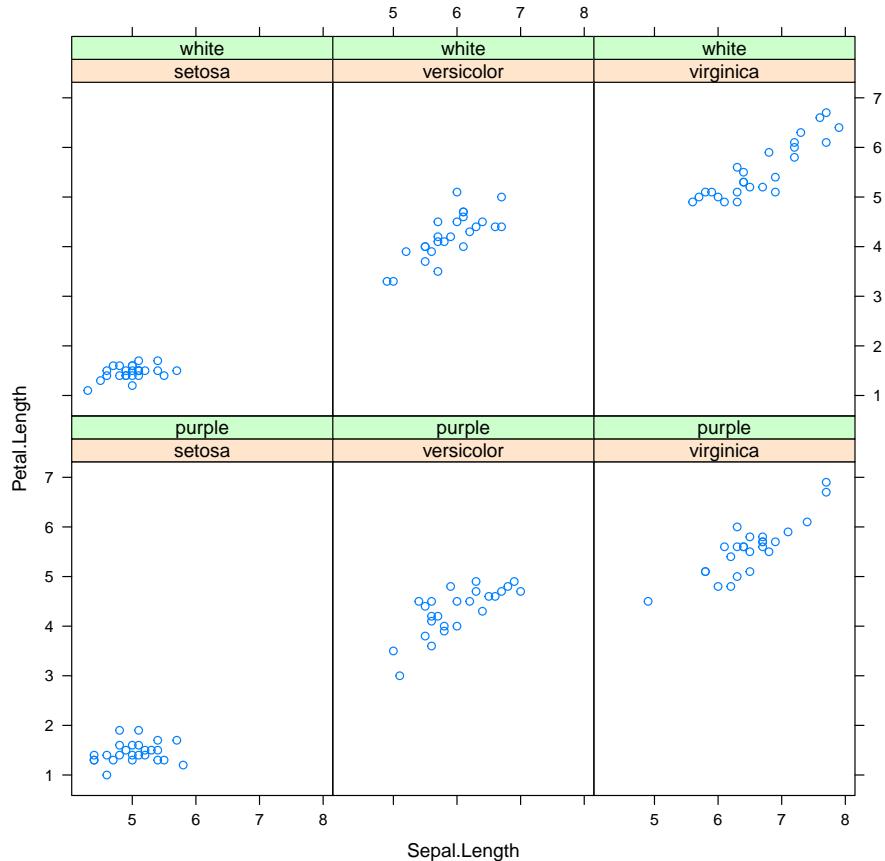
# scatterplot of 2 continuous variables
# by species
xyplot(Petal.Length ~ Sepal.Length | Species, data=iris)
```



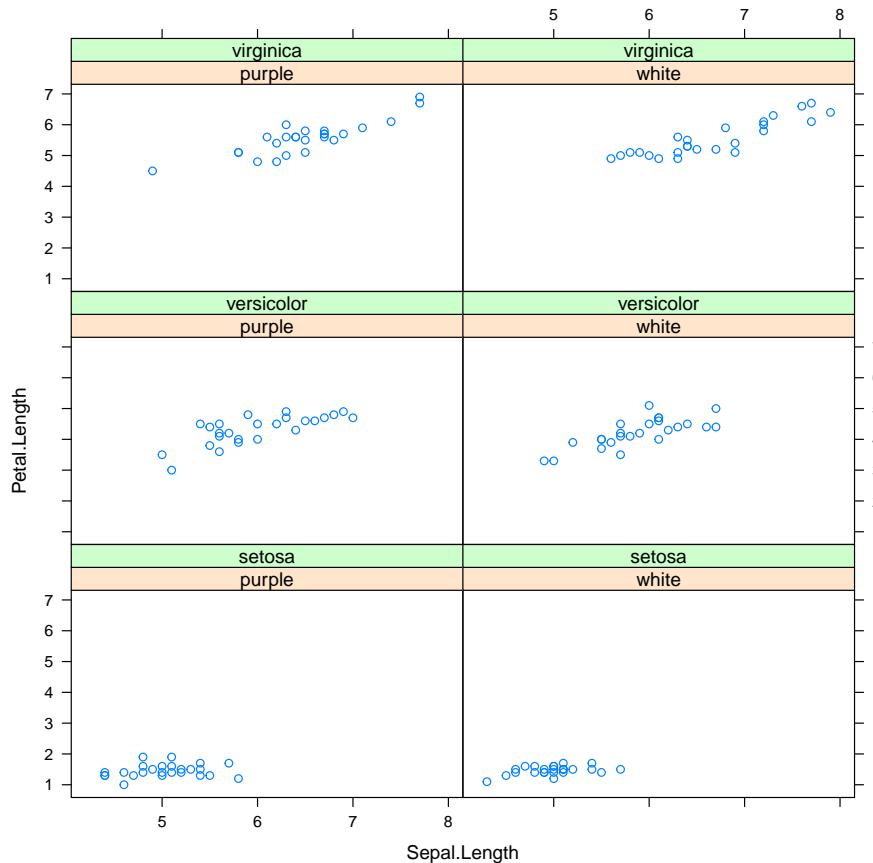
Lattice plots can group observations by more than one variable, as shown below.

```
# add another (pretend) variable
x <- iris
x$color <- c("purple", "white")

xyplot(Petal.Length ~ Sepal.Length | Species+color, data=x)
```

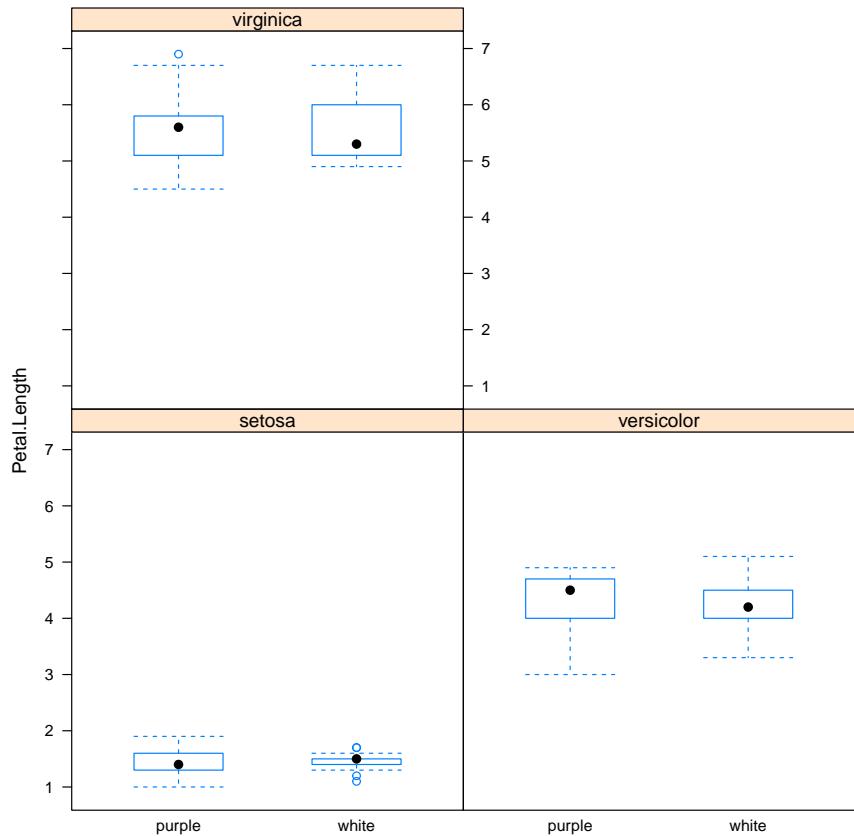


```
# same plot, different layout:  
xyplot(Petal.Length ~ Sepal.Length | color+Species, data=x)
```



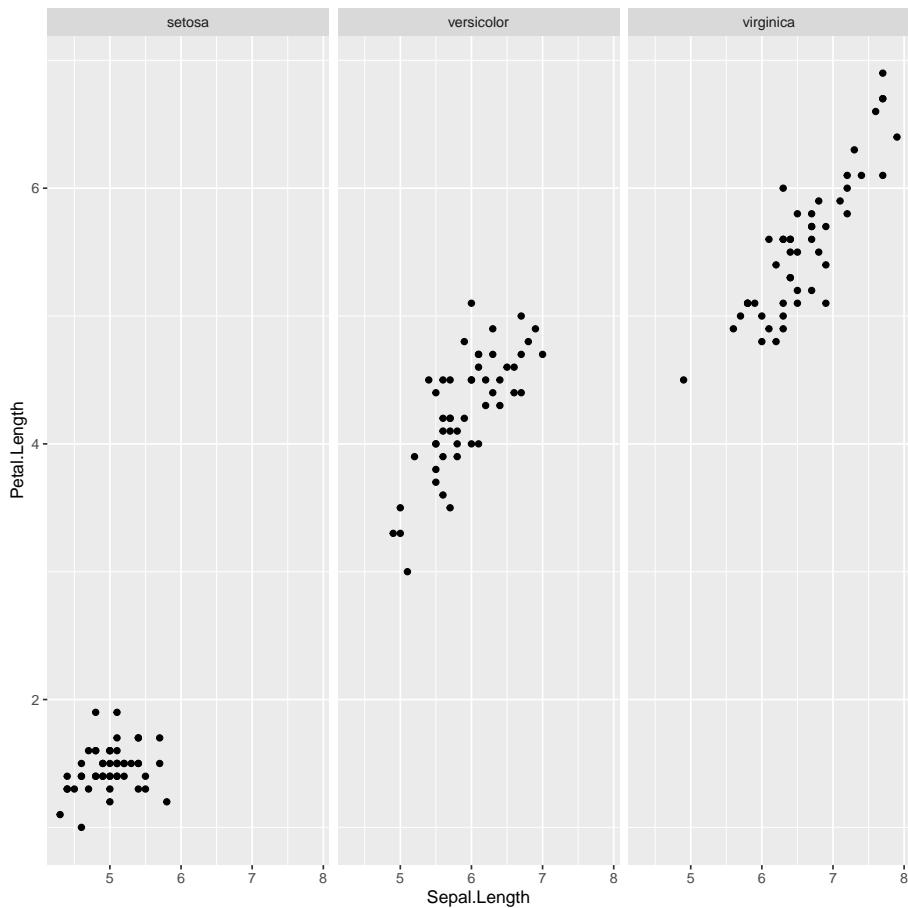
Functions in `lattice` can also produce hierarchical boxplots. This is the graphical equivalent of aggregating by more than one variable.

```
# box-and-whisker plots by color within species
bwplot(Petal.Length~color|Species, data=x)
```

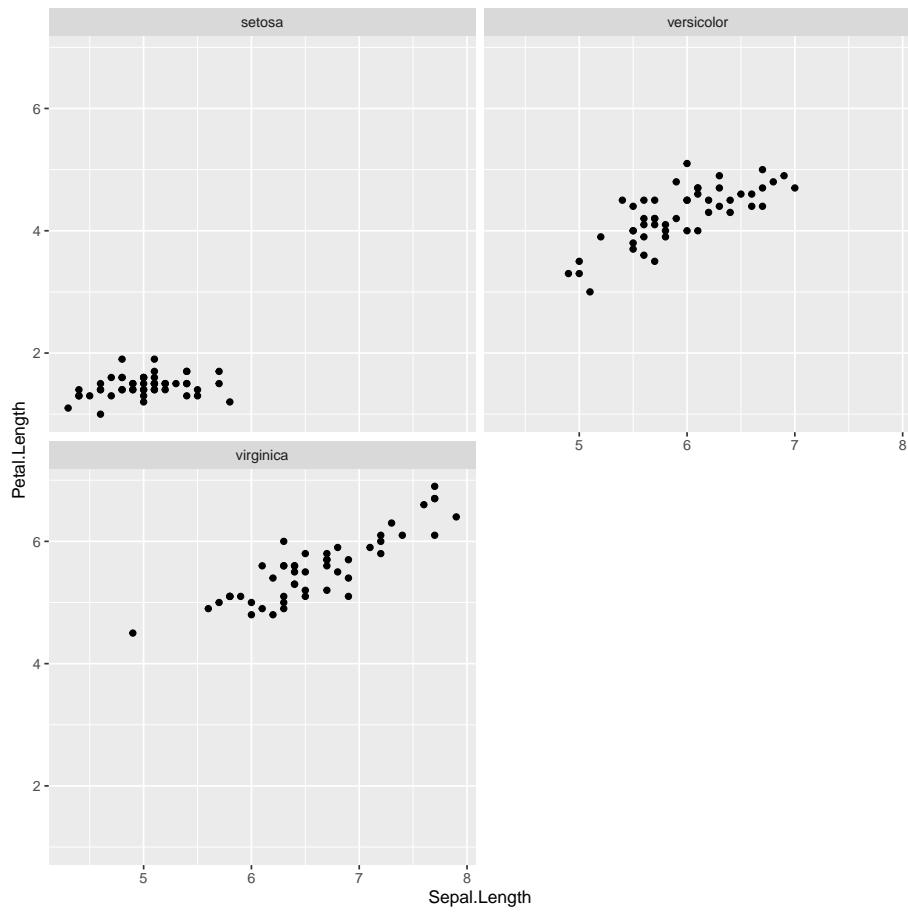


The `tidyverse` equivalent to the first `lattice` plot above is:

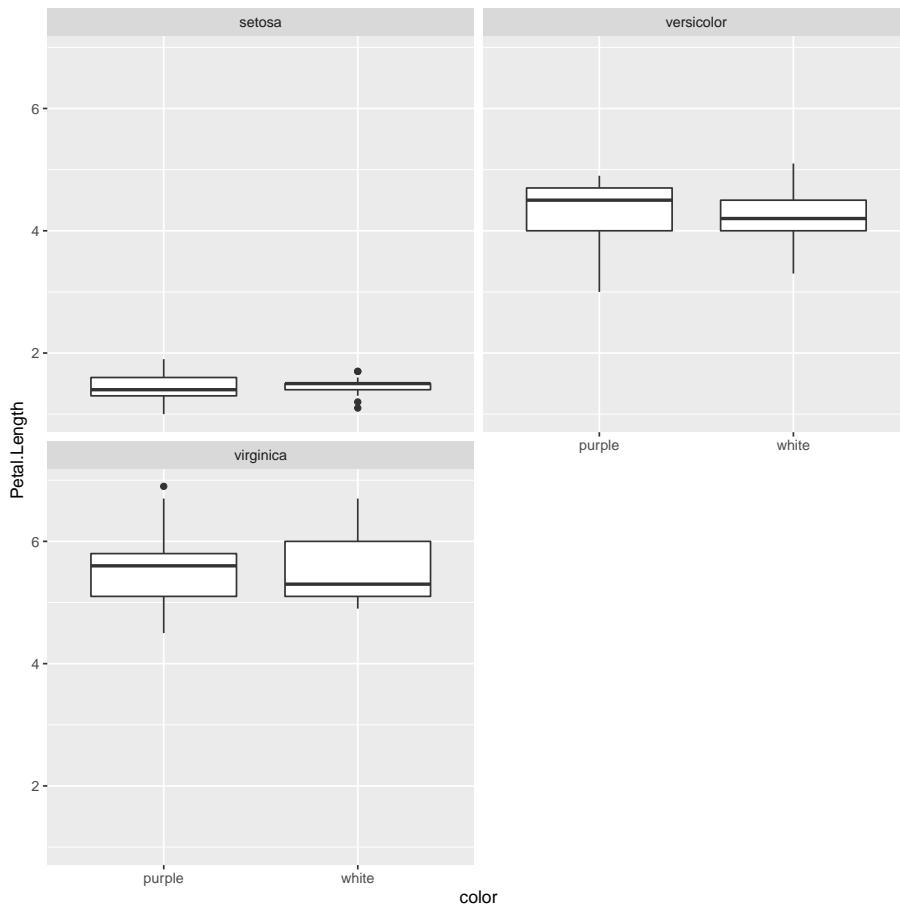
```
library(ggplot2)
ggplot(data=x) +
  geom_point(mapping=aes(x=Sepal.Length, y=Petal.Length)) +
  facet_wrap(~Species, nrow=1)
```



```
# same plot, different layout:  
ggplot(data=x) +  
  geom_point(mapping=aes(x=Sepal.Length, y=Petal.Length)) +  
  facet_wrap(~Species, nrow=2)
```



```
# same plot, another layout:  
ggplot(data=x) +  
  geom_boxplot(aes(y=Petal.Length, x=color)) +  
  facet_wrap(~Species, nrow=2)
```



Whether you use `lattice` or `ggplot2` to make your hierarchical graphs is largely a matter of personal preference. Both packages work just fine for exploratory data analysis. Both share the advantage of being very fast, with the ability to represent a lot of data with very little “ink” and code. Both share the disadvantage of being tricky to customize because many of the graphical options are “hidden” inside layers of other functions.

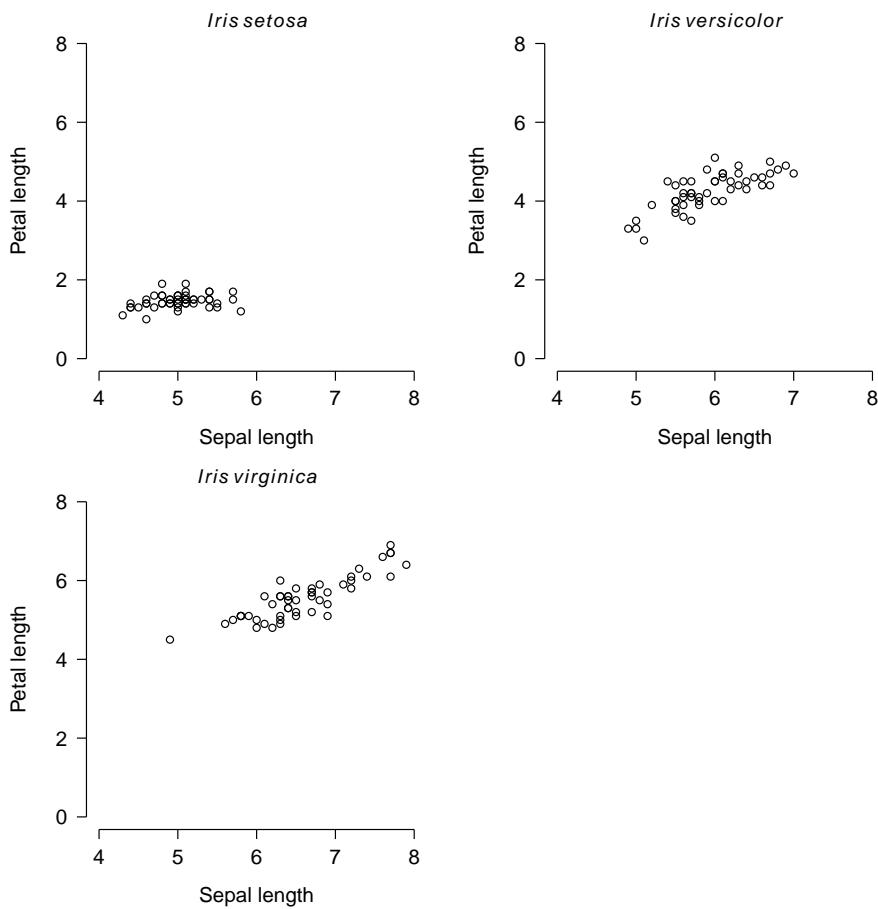
If you need to present a hierarchical plot in a presentation or manuscript, and need to meet very specific formatting requirements, you may need to make your figure using `base` graphics. The commands below reproduce the figures above in `base` graphics. Note that while using `base` graphics requires more coding to get the same result, it is much easier to customize the plots because the plotting options are not buried inside other functions.

```
flag1 <- which(x$Species == "setosa")
flag2 <- which(x$Species == "versicolor")
flag3 <- which(x$Species == "virginica")
```

```

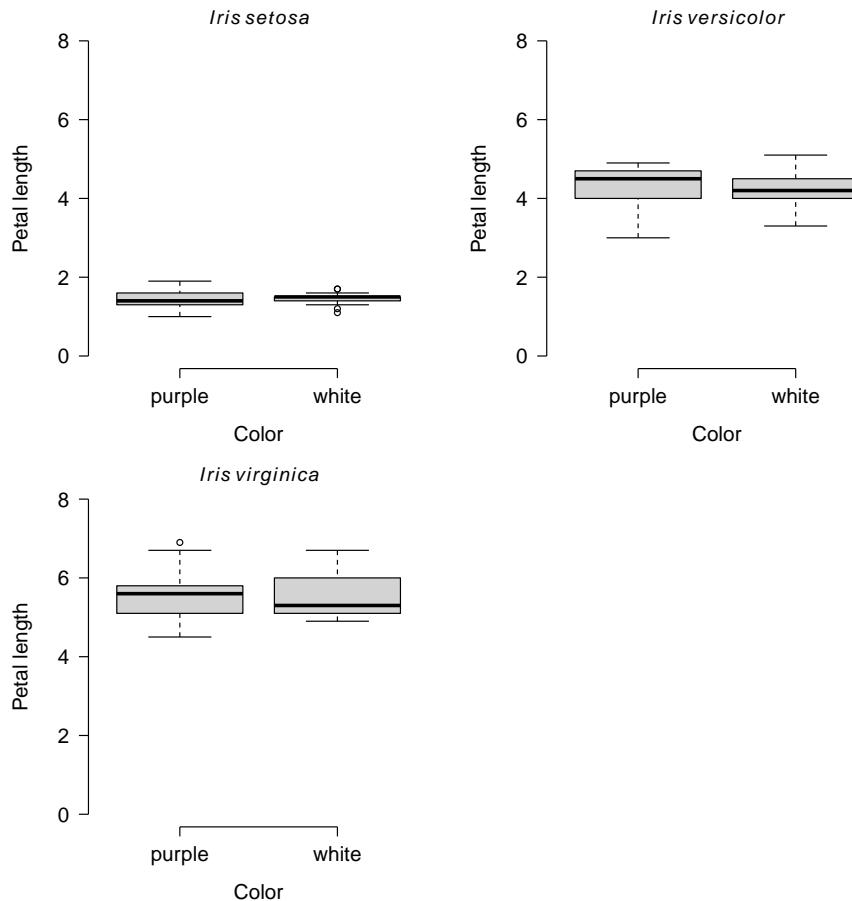
par(mfrow=c(2,2), mar=c(5.1, 5.1, 1.1, 1.1),
  las=1, lend=1, bty="n",
  cex.lab=1.3, cex.axis=1.3)
plot(x$Sepal.Length[flag1], x$Petal.Length[flag1],
  xlim=c(4, 8), ylim=c(0, 8),
  xlab="Sepal length", ylab="Petal length")
title(main=expression(italic(Iris)~italic(setosa)))
plot(x$Sepal.Length[flag2], x$Petal.Length[flag2],
  xlim=c(4, 8), ylim=c(0, 8),
  xlab="Sepal length", ylab="Petal length")
title(main=expression(italic(Iris)~italic(versicolor)))
plot(x$Sepal.Length[flag3], x$Petal.Length[flag3],
  xlim=c(4, 8), ylim=c(0, 8),
  xlab="Sepal length", ylab="Petal length")
title(main=expression(italic(Iris)~italic(virginica)))

```



Here is the hierarchical boxplot:

```
par(mfrow=c(2,2), mar=c(5.1, 5.1, 1.1, 1.1),
  las=1, lend=1, bty="n",
  cex.lab=1.3, cex.axis=1.3)
boxplot(Petal.Length~color, data=x[flag1,],
  xlab="Color", ylab="Petal length",
  ylim=c(0, 8))
title(main=expression(italic(Iris)~italic(setosa)))
boxplot(Petal.Length~color, data=x[flag2,],
  xlab="Color", ylab="Petal length",
  ylim=c(0, 8))
title(main=expression(italic(Iris)~italic(versicolor)))
boxplot(Petal.Length~color, data=x[flag3,],
  xlab="Color", ylab="Petal length",
  ylim=c(0, 8))
title(main=expression(italic(Iris)~italic(virginica)))
```



4.7 Ordination (brief introduction)

The methods above are a way to examine single variables, or single pairs of variables. But what about many variables at once? In other words, how can we examine variation between samples by taking into account multiple variables at once? This is the realm of **multivariate statistics**. In biology many of the most common multivariate methods revolve around ordination.

Ordination is literally plotting or ordering observations along two or more axes. How that ordering is done varies wildly among techniques. We will get deeper into ordination later in the course, but for now let's look at just two ordination techniques, **principal components analysis (PCA)** and **nonmetric multidimensional scaling (NMDS or NMS)**, with a focus on interpretation rather than calculation. Some of the goals of all ordination techniques are:

- **Cluster identification:** observations that are closer to each other in the ordination space are more similar to each other
- **Dimension reduction:** the axes of an ordination are estimates of synthetic variables that combine information about many variables at once

4.7.1 Principal components analysis (PCA)

Principal components analysis (PCA) is a method for extracting gradients from a multivariate dataset that capture most of the variation in that dataset. These gradients are calculated by finding linear combinations of the variables that minimize sums of squared deviations from the gradient. This means that PCA has a lot in common with linear regression, and many of the same assumptions apply.

If you've never heard of PCA or ordination, it might be worth watching a video that explains and shows the basic ideas. Here is [one that only takes 5 minutes and has a nice theme song](#) (accessed 2021-08-10). For a more in-depth introduction to PCA, see this page. Then come back here for some notes on how to use PCA for data exploration.

PCA is available in base R using the functions `prcomp()` and `princomp()`. Both of these functions produce similar outputs, but there are some subtle differences between them. For this course we are going to use the functions in package `vegan`. Although designed for community ecology, `vegan` is widely used for ordination and multivariate analysis in many fields. Importantly, `vegan` is actively maintained and updated with new techniques as they are developed. Also importantly, `vegan` offers a common interface for many kinds of ordination.

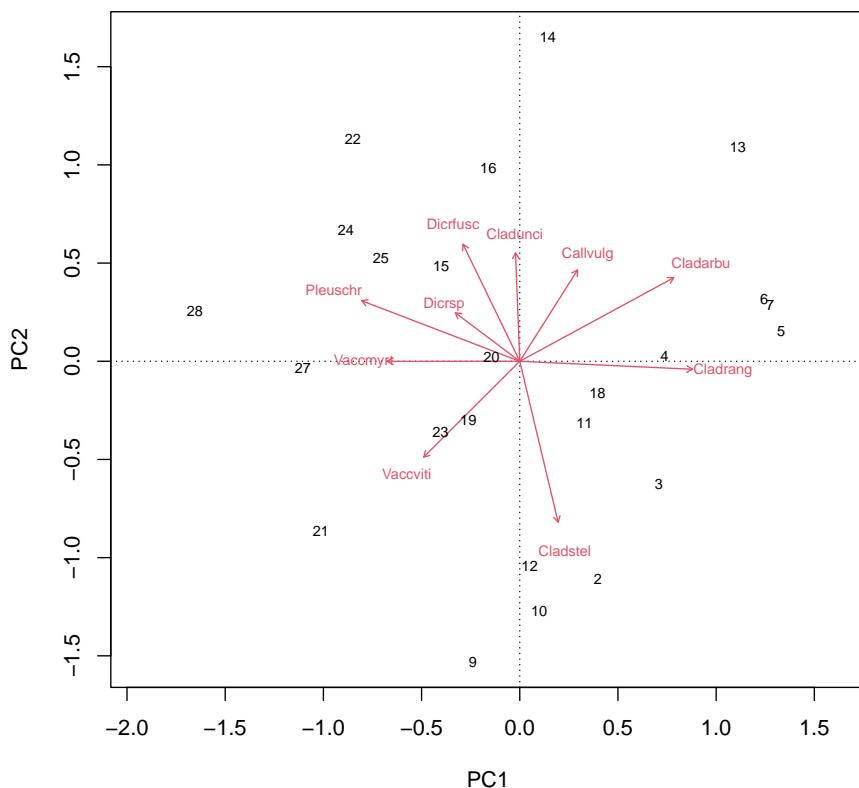
The example below uses the `varespec` dataset from `vegan`. Each column contains % cover by different species of plants in lichen pastures in northern Finland and Scandinavia. For the sake of making a simple example, we will ordinate only the 15 species with the most variance (many species in the dataset are rare or sparse, so including them makes the plot harder to interpret... in "real life", don't remove variables from your analysis without good reason!).

```
library(vegan)

## Loading required package: permute
## This is vegan 2.5-7
# get data and make a spare copy
data(varespec)
x <- varespec

# identify the variables with most variance
vx <- rev(sort(apply(x, 2, var)))
x <- x[,names(vx)[1:10]]
```

```
# calculate PCA and plot it
p1 <- rda(x, scale=TRUE)
biplot(p1)
```



The figure above is called a **PCA biplot**. Each point is a sample (site) where vegetation was sampled. By default, the samples are identified by their row number. The red arrows show sites where different species were more abundant (or, more generally, where the variables are increasing). For example, sites towards the top right had greater cover by species “Cladstel” (short for *Cladonia stellaris*). Sites towards the lower left had less cover by Cladstel. Sites towards the top left had more cover by “Pleuschr” (*Pleurozium schreberi*).

The `summary()` command will print a lot of information about the ordination. It prints the eigenvalues of each PC, then scales them as “Proportion explained”. I.e., the proportion of variation explained by each PC is the same as its eigenvalue

divided by the sum of all eigenvalues.

```
summary(p1)
```

```
##
## Call:
## rda(X = x, scale = TRUE)
##
## Partitioning of correlations:
##           Inertia Proportion
## Total          10      1
## Unconstrained 10      1
##
## Eigenvalues, and their contribution to the correlations
##
## Importance of components:
##           PC1     PC2     PC3     PC4     PC5     PC6     PC7
## Eigenvalue   2.7808 1.9382 1.3148 1.0987 0.83622 0.7150 0.69134
## Proportion Explained 0.2781 0.1938 0.1315 0.1099 0.08362 0.0715 0.06913
## Cumulative Proportion 0.2781 0.4719 0.6034 0.7132 0.79687 0.8684 0.93750
##           PC8     PC9     PC10
## Eigenvalue  0.33428 0.17151 0.11919
## Proportion Explained 0.03343 0.01715 0.01192
## Cumulative Proportion 0.97093 0.98808 1.00000
##
## Scaling 2 for species and site scores
## * Species are scaled proportional to eigenvalues
## * Sites are unscaled: weighted dispersion equal on all dimensions
## * General scaling constant of scores: 3.894323
##
##
## Species scores
##
##           PC1     PC2     PC3     PC4     PC5     PC6
## Cladstel  0.22980 -0.9638221 0.2063 -0.5766 0.22839 -0.13546
## Pleuschr -0.94726  0.3626752 -0.2495 0.2859 0.31322 -0.08035
## Cladrang  1.03450 -0.0478938 -0.2618 0.4311 0.14433 -0.14610
## Cladarbu  0.92222  0.5003246 -0.1221 0.3915 -0.31092 0.13843
## Dicrfusc -0.33989  0.7013881 -0.1721 -0.5513 -0.19752 -0.32352
## Vaccviti -0.57531 -0.5738560  0.1310 0.1739 -0.66855 0.52296
## Dicrsp    -0.38606  0.2903390  0.8144 0.3716 0.52606 0.26321
## Callvulg  0.34621  0.5471556 -0.3068 -0.6120 0.25486 0.73026
## Cladunci -0.02526  0.6491055  0.7160 -0.1866 -0.41338 -0.19511
## Vaccmyrt -0.79400 -0.0005864 -0.6992 0.1767 -0.04322 -0.01670
##
##
```

```

## Site scores (weighted sums of species scores)
##
##          PC1       PC2       PC3       PC4       PC5       PC6
## 18  0.39657 -0.16342 -0.079895  0.68235 -0.98085  0.6198150
## 15 -0.40025  0.48524 -0.119881  0.10251 -0.25276 -0.2907006
## 24 -0.88573  0.67138  2.334307  1.08698  1.26348  0.8403431
## 27 -1.10452 -0.03240 -0.660890  0.55886  0.14165 -0.0003119
## 23 -0.40297 -0.35825  0.084662  0.42892 -1.19313  0.8452582
## 19 -0.26161 -0.30040 -0.123290  0.01983  0.17886 -0.1684937
## 22 -0.85012  1.13056 -0.797382 -1.33729 -0.65017 -0.5866657
## 16 -0.15994  0.98635 -0.571647 -0.91477  0.37822 -0.8158903
## 28 -1.65484  0.25754 -1.629606  0.80391  1.00293 -0.5951382
## 13  1.11019  1.09130 -0.918028 -1.32805  1.01767  2.7409133
## 14  0.14308  1.65235  1.451371 -1.01710 -1.53004 -0.9476702
## 20 -0.14436  0.02087  0.385155  0.19971 -0.55881  0.0248203
## 25 -0.70668  0.52484  0.951767  0.41019  0.99674  0.1995443
## 7   1.27601  0.28464 -0.392360  1.25885 -0.39985 -0.1596378
## 5   1.33049  0.15587 -0.500076  1.08816  0.58014 -0.9228755
## 6   1.24463  0.31814 -0.096950  1.06221 -0.85622 -0.0198950
## 3   0.70748 -0.62339  0.020031 -0.26749  0.77095 -0.8805476
## 4   0.73772  0.02904 -0.134862 -0.22867  0.70704 -0.3411419
## 2   0.39724 -1.10654  0.219538 -0.67452  0.72211 -0.6843507
## 9   -0.23911 -1.53329  0.521191 -0.91391 -0.36176  0.3560478
## 12  0.05053 -1.04256  0.379673 -0.50633 -0.09530  0.0281206
## 10  0.09930 -1.27173  0.418480 -1.00789  0.35197 -0.2898518
## 11  0.32926 -0.31288 -0.003708  0.12732  0.07082  0.0582269
## 21 -1.01237 -0.86325 -0.737599  0.36622 -1.30368  0.9900814

```

The “Importance of components” part tells us that the first principal component (PC1) explains about 55% of the variation in the dataset. PC2 explains an additional 26%. The rule of thumb is to present enough PCs to account for $\geq 80\%$ of the variation. In this case, PC1 and PC2 are sufficient. If it takes many PCs to reach 80% of variation, that may be a sign that the PCs are not capturing much meaningful structure in your dataset (or, there simply isn’t much structure to capture).

Finally, we can see the variable “loadings”, which tell us how much each variable is correlated with each principal component¹⁷. These values are interpreted the same way as linear correlation coefficients. In the example below the loadings for PCs 1-4 are requested using argument choices = 1:4.

```

scores(p1, choices = 1:4, display = "species", scaling = 0)

##          PC1       PC2       PC3       PC4
## Cladstel  0.1118986 -0.5621722123  0.14606363 -0.4467186

```

¹⁷In the base R PCA functions, these values are obtained from the `rotation` part of the output in `prcomp()` or from the `loadings` component in `princomp()`.

```

## Pleuschr -0.4612677  0.2115389675 -0.17670529  0.2214946
## Cladrang  0.5037497 -0.0279351716 -0.18537465  0.3339431
## Cladarbu  0.4490762  0.2918262462 -0.08645467  0.3032787
## Dicrfusc -0.1655069  0.4091013432 -0.12191016 -0.4270946
## Vaccviti -0.2801468 -0.3347151730  0.09280300  0.1347303
## Dicrsp   -0.1879928  0.1693471498  0.57673161  0.2878989
## Callvulg 0.1685866  0.3191415425 -0.21724150 -0.4741263
## Cladunci -0.0123003  0.3786062372  0.50705444 -0.1445745
## Vaccmyrt -0.3866379 -0.0003420561 -0.49518742  0.1368990
## attr(,"const")
## [1] 3.894323

```

The loadings tell us that PC1 is strongly influenced by species Cladstel and less so by Pleuschr. Compare this to the orientation and lengths of the arrows for these species in the biplot and see if you can find a connection.

4.7.2 Nonmetric multidimensional scaling (NMDS)

PCA and many related techniques are based on linear algebra and eigenvalues. This is fine for datasets where variables are mostly linearly related to each other, or can be transformed to be linearly related. Most of the eigenvalue-based techniques also require data to be mostly normally distributed.

If relationships between variables are nonlinear, or if the many other assumptions of eigenvalue-based ordination cannot be met, then the next best option is a non-parametric ordination technique called nonmetric multidimensional scaling (NMDS or NMS). Unlike PCA, which solves linear algebra problems to extract synthetic gradients (“principal components”), NMDS works by trying iteratively to arrange the samples into a reduced dimensional space that preserves the rank order of the distance matrix.

What? Let’s break that down:

Distance matrix: a matrix containing a measure of dissimilarity (aka: “distance”) between each pair of samples. This is exactly analogous to a road mileage chart in an old paper road atlas.

Distance (dissimilarity) measure: a quantity that measures how different two samples are in terms of several variables. Greater values indicate that two samples are more different from each other; smaller values indicate that two samples are more similar to each other. See this page for a more in-depth explanation of distance measures.

The most famous distance metric is the Euclidean distance metric, shown here as the distance between two points on the XY plane:

$$D_{Euc.} = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

The metric can be generalized to any number of dimensions k .

$$D_{Euc.} = \sqrt{\sum_{k=1}^k (x_1 - x_2)^2}$$

The Euclidean distance is rarely used for NMDS. Instead, the default distance metric is the **Bray-Curtis distance**. This metric is called **Sørensen distance** when used for binary values (e.g., presence-absence), and Bray-Curtis when used for quantitative values.

The distance matrix contains the dissimilarities between samples, in terms of all of the variables in the dataset. If there are k variables, then these are distances through k -dimensional space. NMDS tries to represent the relative dissimilarities between samples in fewer than k dimensions, while preserving the **rank order** of those dissimilarities.

The R function `metaMDS()` performs NMDS. Getting a biplot of the ordination is a little more involved than for PCA. Note that NMDS involves some random number sampling in its algorithms, so you should set the random number seed to make sure your analysis is reproducible.

```
set.seed(123)

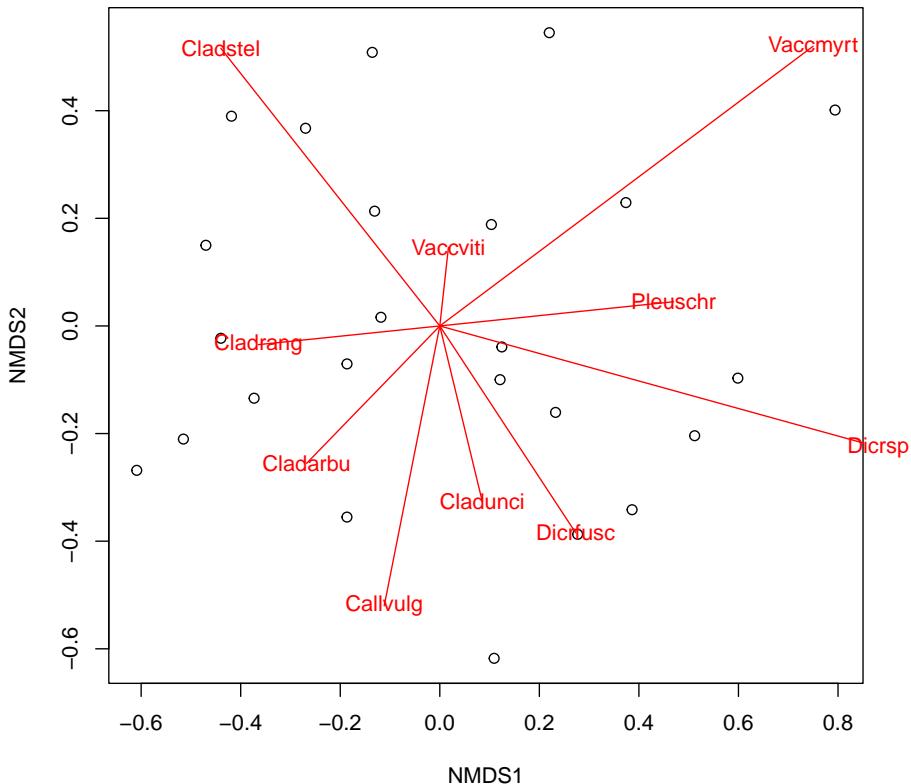
# fit NMDS
n1 <- metaMDS(x)

## Square root transformation
## Wisconsin double standardization
## Run 0 stress 0.1258291
## Run 1 stress 0.128478
## Run 2 stress 0.128478
## Run 3 stress 0.1273261
## Run 4 stress 0.2014993
## Run 5 stress 0.1991787
## Run 6 stress 0.128478
## Run 7 stress 0.128478
## Run 8 stress 0.128478
## Run 9 stress 0.1258291
## ... Procrustes: rmse 2.910913e-06 max resid 9.961616e-06
## ... Similar to previous best
## Run 10 stress 0.1258291
## ... Procrustes: rmse 1.699748e-06 max resid 4.320923e-06
## ... Similar to previous best
## Run 11 stress 0.128478
## Run 12 stress 0.128478
## Run 13 stress 0.128478
```

```
## Run 14 stress 0.1258291
## ... Procrustes: rmse 1.791907e-06 max resid 5.731764e-06
## ... Similar to previous best
## Run 15 stress 0.128478
## Run 16 stress 0.1976983
## Run 17 stress 0.1258291
## ... Procrustes: rmse 3.90069e-06 max resid 1.369466e-05
## ... Similar to previous best
## Run 18 stress 0.1273261
## Run 19 stress 0.1258291
## ... Procrustes: rmse 2.251913e-06 max resid 7.461718e-06
## ... Similar to previous best
## Run 20 stress 0.235875
## *** Solution reached

# extract "scores" (coordinates in NMDS space)
# these are matrices: column 1 = x, column 2 = y
n1s <- scores(n1, "sites")
n1p <- scores(n1, "species")

# basic NMDS plot
plot(n1s)
# add biplot arrows and labels
segments(0, 0, n1p[,1], n1p[,2], col="red")
text(n1p[,1], n1p[,2], rownames(n1p), col="red", xpd=NA)
```



Because NMDS solutions are reached through random sampling, they are not guaranteed to reach a stable condition every time. You will need to set the random number seed to get reproducible results. There is also no guarantee that the solution reached by NMDS will be the optimal one.

Another consequence of the way that NMDS works is that the axes themselves are arbitrary, unlike the axes in PCA. This means that we can't calculate a percentage of variation explained by each axis the way we did for PCA. By default, R will rotate NMDS solutions so that the greatest amount of variation is associated with axis 1, but there is no guarantee that this axis captures a meaningful biological gradient.

The most important diagnostic parameter in an NMDS is the **stress**, which is a measure of how much the distance matrix is distorted by compression into low dimensional space. Lower stress value indicate better fit. Most people consider a stress ≥ 0.2 to indicate an unreliable fit. Some authors recommend lower thresholds like 0.15 or 0.1. Package **vegan** and most sources use stress values < 1 .

(e.g., Legendre and Legendre 2012); some authors scale stress by a factor of 100 relative to the values in `vegan`. For example, the program PC-ORD (McCune et al. 2002) uses stress values on this greater scale.

4.7.3 Plotting ordinations

Extracting the scores from an ordination object can let you plot points with all of the `base` graphics formatting tricks we learned above, such as using color or shape to indicate group membership. In my experience this is more flexible than using the built-in graphics capabilities in `vegan`¹⁸, but your mileage may vary.

```
x <- crabs
p1 <- rda(x[,4:8])
n1 <- metaMDS(x[,4:8])

## Square root transformation
## Wisconsin double standardization
## Run 0 stress 0.08013593
## Run 1 stress 0.08013594
## ... Procrustes: rmse 2.47761e-06 max resid 3.004629e-05
## ... Similar to previous best
## Run 2 stress 0.08013594
## ... Procrustes: rmse 5.318632e-06 max resid 5.42044e-05
## ... Similar to previous best
## Run 3 stress 0.08013593
## ... Procrustes: rmse 1.789737e-05 max resid 0.0001945269
## ... Similar to previous best
## Run 4 stress 0.08013596
## ... Procrustes: rmse 1.021893e-05 max resid 0.0001280391
## ... Similar to previous best
## Run 5 stress 0.08013593
## ... New best solution
## ... Procrustes: rmse 3.631654e-06 max resid 4.608495e-05
## ... Similar to previous best
## Run 6 stress 0.08013593
## ... Procrustes: rmse 4.019493e-06 max resid 5.419834e-05
## ... Similar to previous best
## Run 7 stress 0.08013592
## ... New best solution
## ... Procrustes: rmse 6.01397e-06 max resid 8.261179e-05
## ... Similar to previous best
## Run 8 stress 0.08013594
## ... Procrustes: rmse 1.059936e-05 max resid 0.0001315101
## ... Similar to previous best
```

¹⁸An introduction by the package's lead author can be found here: <https://cran.r-project.org/web/packages/vegan/vignettes/intro-vegan.pdf> (accessed 2021-09-17)

```

## Run 9 stress 0.08013592
## ... New best solution
## ... Procrustes: rmse 2.632514e-06 max resid 3.234954e-05
## ... Similar to previous best
## Run 10 stress 0.08013593
## ... Procrustes: rmse 1.374691e-05 max resid 0.0001352853
## ... Similar to previous best
## Run 11 stress 0.08013593
## ... Procrustes: rmse 9.848983e-06 max resid 9.509947e-05
## ... Similar to previous best
## Run 12 stress 0.08013593
## ... Procrustes: rmse 9.871024e-06 max resid 0.0001008813
## ... Similar to previous best
## Run 13 stress 0.08013593
## ... Procrustes: rmse 6.457749e-06 max resid 6.335005e-05
## ... Similar to previous best
## Run 14 stress 0.08013595
## ... Procrustes: rmse 1.626131e-05 max resid 0.0001518457
## ... Similar to previous best
## Run 15 stress 0.08013595
## ... Procrustes: rmse 1.446168e-05 max resid 0.0001679656
## ... Similar to previous best
## Run 16 stress 0.08013593
## ... Procrustes: rmse 1.148406e-05 max resid 0.0001448735
## ... Similar to previous best
## Run 17 stress 0.08013593
## ... Procrustes: rmse 9.122826e-06 max resid 0.000112683
## ... Similar to previous best
## Run 18 stress 0.08013594
## ... Procrustes: rmse 1.22868e-05 max resid 0.0001430928
## ... Similar to previous best
## Run 19 stress 0.08013595
## ... Procrustes: rmse 1.588575e-05 max resid 0.000217428
## ... Similar to previous best
## Run 20 stress 0.08013592
## ... Procrustes: rmse 2.291841e-06 max resid 1.545649e-05
## ... Similar to previous best
## *** Solution reached

p1s <- scores(p1)$sites
p1p <- scores(p1)$species

n1s <- scores(n1, "sites")
n1p <- scores(n1, "species")

cols1 <- ifelse(x$sex == "F", "red", "blue")

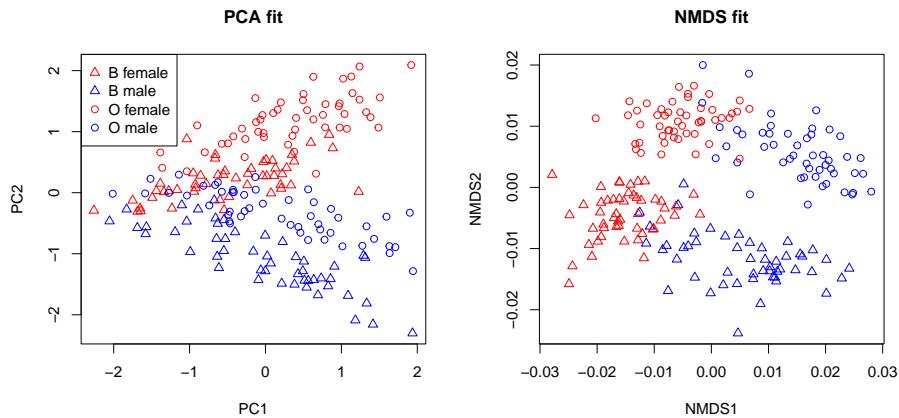
```

```

pch1 <- ifelse(x$sp == "B", 2, 1)

par(mfrow=c(1,2))
plot(p1s, col=cols1, pch=pch1, main="PCA fit")
legend("topleft",
       legend=c("B female", "B male", "O female", "O male"),
       pch=c(2, 2, 1, 1),
       col=c("red", "blue", "red", "blue"))
plot(n1s, col=cols1, pch=pch1, main="NMDS fit")

```



4.7.4 Ordination wrap-up (for now)

Ordination is a collection of methods for reducing the dimensionality of data. Biological datasets often have many variables, but most variation in those datasets can often be captured using only a few of them (or few combinations of the variables). The positions of the samples in the ordination space can thus reveal patterns that are too complicated to see in the full data space. Interpreting ordination axes in biological terms can be very fruitful. When an axis captures important information about the samples, it can be used as a variable in another method (e.g., as a predictor variable in a linear model). Analysis of clustering or positional differences between groups in ordination space (especially NMDS space) can allow for researchers to test for differences in many variables at once.

4.8 Common statistical problems

Many statistical methods assume that data follow certain distributions, and we have already explored methods for making sure your data follow those distributions. Other problems are not so easy to diagnose or fix. This section will introduce you to some methods for dealing with more subtle data problems.

Note that none of these issues have one-size-fits-all solutions. Dealing with these issues will require understanding your study system and thinking through the consequences of potential solutions.

4.8.1 Outliers and erroneous values

Outliers are values that much larger or smaller than other values in the data set. While these values are theoretically possible in any distribution, they are by their nature extremely unlikely. Outliers can cause problems with many statistical tests because they are often given undue weight. This is because deviations from the mean are often squared.

There are two problems with outliers: how to detect them, and what to do with them.

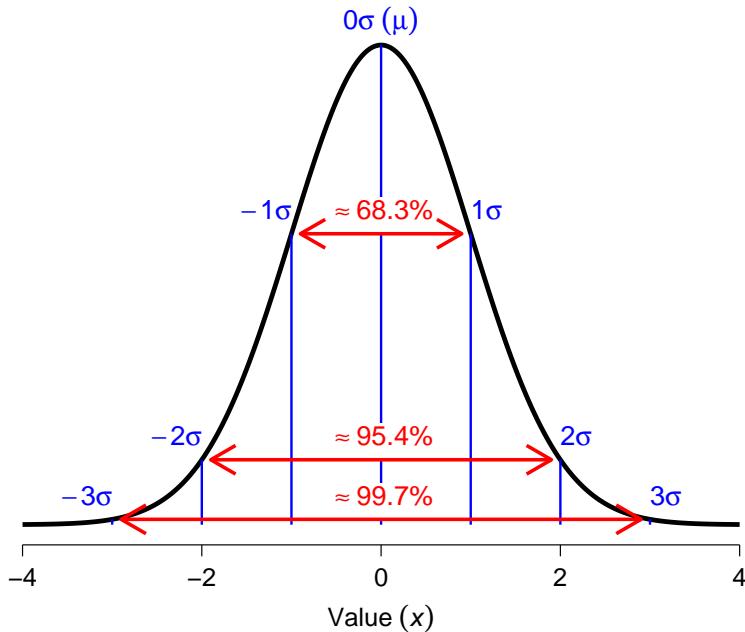
4.8.1.1 Detecting outliers

There are several definitions of outlier. Some researchers have suggested simple measures such as Z -scores to detect outliers. Z -scores measure how many standard deviations (SD) an observation is away from the mean. The Z -score of an observation x_i (observation i of variable x) is calculated as:

$$Z(x_i) = \frac{x_i - \bar{x}}{\sigma(x)}$$

where \bar{x} is the mean of x and $\sigma(x)$ is the SD of x . For example, Z -scores of -2.3 and +4.1 indicate observations 2.3 SD below and 4.1 SD above the mean, respectively. Z -scores are also referred to as **sigmas**, referring to the symbol for SD. E.g., a 4σ observation is 4 SD away from the mean. Various σ levels have been used to detect outliers, in which researchers discard any observation 2σ , 3σ , etc., away from the mean. This threshold varies by discipline so follow the conventions in your field.

The figure below shows Z -scores relative to the standard normal distribution ($\mu = 0$, $SD = 1$). About 68% of observations are expected to have a Z -score between -1 and 1—i.e., deviate by $\leq 1\sigma$. About 95% of observations should be $\leq 2\sigma$, and over 99% of observations should be $\leq 3\sigma$. The greater the magnitude of a value's Z -score, the less likely it is. Values with > 3 or 4σ should almost never occur. Thus, observations with large-magnitude Z -scores may turn out to be outliers.



It must be noted that a large Z -score does *not* prove that an observation is an outlier. Like p -values, Z -scores are just a heuristic for making decisions. Don't just delete any observation $|Z| \geq 3$; examine those observations to see if they are legitimate and don't take any action without a good biological reason.

4.8.1.2 Dealing with outliers

The easiest way to deal with outliers is simply to delete them. This can be fine if you have a lot of data—many 1000s or even millions of records. I've worked with several statisticians who routinely delete observations with response variables in the top and bottom 5 percentiles. This may not always be an appropriate strategy in biology. For one thing, outliers can occur for completely legitimate reasons and deleting them would be discarding important information about the study system. You should never delete an observation based solely on its magnitude. There should be some other reason: e.g., experimental errors, inappropriate sampling, extraordinary field conditions, etc. Furthermore, you should always document when observations were deleted and why, and disclose that information to reviewers. The possibility of including supplemental information or documents with manuscripts leaves little excuse not to be transparent about what observations you deleted and why.

Another way to deal with outliers is to transform the data. Variance-stabilizing transformations, such as the log-transform, often have the side-effect of reducing outliers in the data.

The third method for dealing with outliers is **censoring**. Censoring data in statistics means analyzing data as either the measured value or the interval in which the value falls. For example, if a mouse is weighed with a scale that goes up to 60 g, and maxes out the scale, we record the mouse as weighing ≥ 60 g. Statistical methods for censored data have been developed but are outside the scope of this course. One popular form of censoring is called **Winsorizing**, where values below the 5th percentile are set to the 5th percentile and values above the 95th percentile are set to the 95th percentile. Censoring is common practice in fields involving certain types of instrumentation. For example, many spectrophotometers (e.g., microplate readers) record **optical density (OD)** as the amount of light passing through a specimen as a measurement of concentration. Most machines have a lower limit to the OD that they can detect. Observations below this minimum OD might be censored to 0, or to some pre-determined fraction of the minimum detectable OD.

Similar to censoring, outlying data can be removed by **truncation**. Truncation in statistics means ignoring values outside of a certain range. As with deletion, you must be very explicit about what criteria were used to truncate the data.

Weighting is a strategy where observations can be given greater or lesser weight to adjust how much influence they have. This strategy can become very complicated and involve a lot of arbitrary decisions. So, a simpler strategy such as truncation or transformation might be more appropriate.

Finally, researchers can use analysis strategies that are less sensitive to outliers. For example, **robust statistics** use absolute deviations rather than squared deviations. This gives outliers much less influence than under conventional methods based on sums of squared errors. **Nonparametric methods**, particularly those based on rank order, can also be less sensitive to outliers.

Whatever strategy you choose to deal with your outliers, you need to be aware of how much your decision affected the outcome. It is a good idea to analyze your data both with and without suspected outliers to measure how much influence they really have. If the analysis returns the same results with and without the outliers, leave them in.

4.8.2 Autocorrelation

Autocorrelation describes a phenomenon where set of values is correlated with itself, with the strength of that correlation determined by the distance between observations. In biology, the “distance” that defines autocorrelation is often spatial, temporal, or phylogenetic.

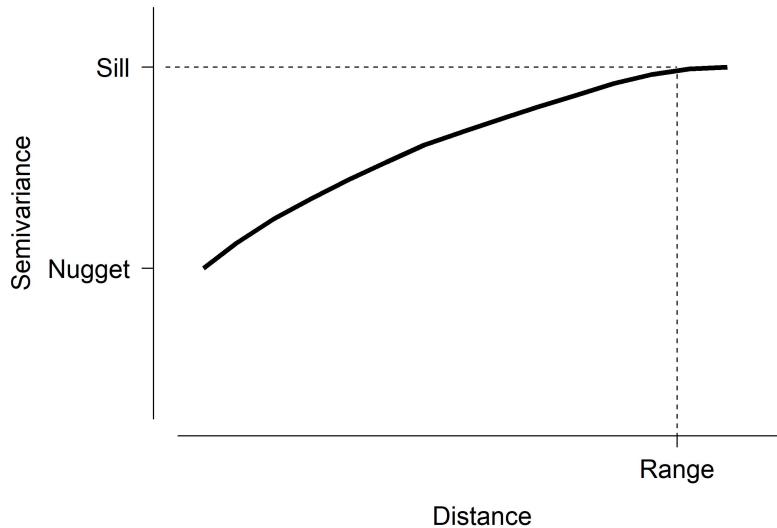
Autocorrelation type	Distance measure	Explanation
Spatial	Space (e.g., km)	Samples from sites that are close together in space are more similar to each other than they are to sites that are farther away, just because they are closer to each other.
Temporal	Time (e.g., days)	Observations taken closer together in time are more similar to each other than they are to observations separated by more time.
Phylogenetic	Relatedness	Taxa that share a more recent common ancestor (i.e., are more closely related) are more similar to each other than they are to taxa that are less closely related to each other.

The main consequence of autocorrelation is similar to the problems associated with pseudoreplication: because observations are not independent, the statistically relevant number of observations (degrees of freedom) is smaller than the nominal number of observations. This means that the power of the test is inflated. A related idea is that the residuals of a statistical model can be autocorrelated. In that case, the model violates the assumption of independent residuals shared by most statistical methods. Autocorrelation can be controlled for if you have some idea of its source and nature.

4.8.2.1 Graphical methods to identify autocorrelation

One way to explore potential autocorrelation is to construct a variogram. A variogram plots the squared differences between samples against their distance (temporal, spatial, etc.). A semivariogram uses half of that variance. An example semivariogram is below.

This figure shows some important quantities on the semivariogram:



- **Nugget:** The variance or semivariance at 0 distance. I.e., the y-intercept. Variation at zero distance represents other sources of error, such as sampling error, demographic stochasticity, etc.
- **Sill:** The variance or semivariance at which the curve gets close to its maximum or asymptotic value.
- **Range:** The distance at which the variance or semivariance reaches the sill.

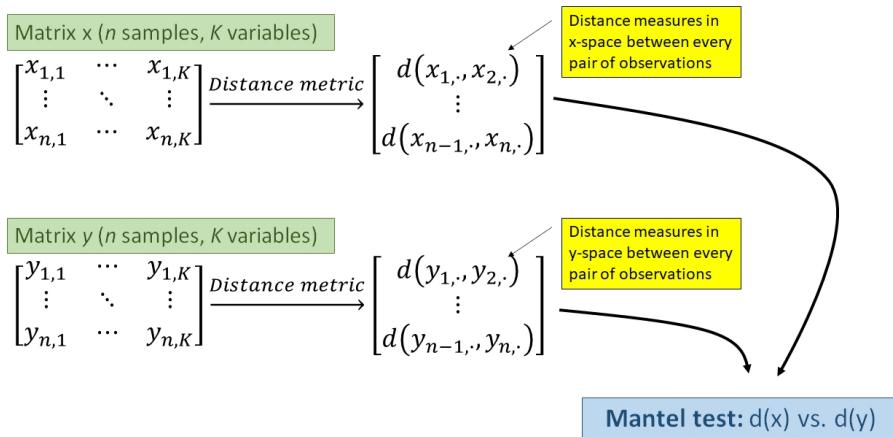
Semivariograms should always increase, at least initially, because closer observations should be similar to each other simply as a matter of proximity. Samples separated by distances less than the range are autocorrelated. Semivariograms sometimes reach a peak and then start decreasing, or varying up and down. The initial peak is usually interpreted as the sill, and the distance when the curve reaches the sill is considered the range. The importance and interpretation of the semivariogram is something that you, as a biologist, have to figure out.

4.8.2.2 Mantel tests for autocorrelation

Spatial autocorrelation is a particular problem for ecologists and environmental biologists. The **Mantel test** can be used to measure spatial autocorrelation as the correlation between distance in space and distance in terms of the measured variables. Generally, the Mantel test can be thought of as a multivariate generalization of linear correlation that measures correlation between two matrices. Whereas linear correlation measures how variation in one variable is related to variation in another variable, the Mantel test measures how variation in one matrix (i.e., set of variables) is related to variation in another matrix.

The application to spatial autocorrelation is obvious. The spatial distance matrix

can be calculated from coordinates such as latitude or longitude¹⁹, and compared to the biological distance matrix. When other types of autocorrelation are of interest, you can calculate a distance matrix based on time or on phylogenetic relatedness, or whatever else might be driving the autocorrelation.



The Mantel test works by first calculating two **distance matrices**, which define the “distances” or “dissimilarities” between each sample. One distance matrix relates to spatial, temporal, or phylogenetic distance. The other relates to dissimilarity in terms of the measured variables. The dissimilarity metric incorporates information about how much any pair of observations differ in terms of every variable represented in the matrix. The Mantel correlation coefficient r is the linear correlation coefficient (like Pearson’s r) between the two distance matrices. It answers the question, “How is variation in this set of variables related to distance in space (or time, or phylogeny)?”

The example below illustrates another use for the Mantel test: measuring association between two sets of variables taken from the same sample units. In this example, dissimilarity in terms of species composition is related to dissimilarity in soil chemical characteristics.

```
library(vegan)

# biological data
data(varespec)

# soil chemistry data
data(varechem)

x <- varechem
```

¹⁹When calculating a distance matrix using geographic coordinates, be sure to account for the curvature of the Earth. Often using unprojected latitude and longitude will produce a misleading distance matrix, because latitude and longitude are not equivalent.

```

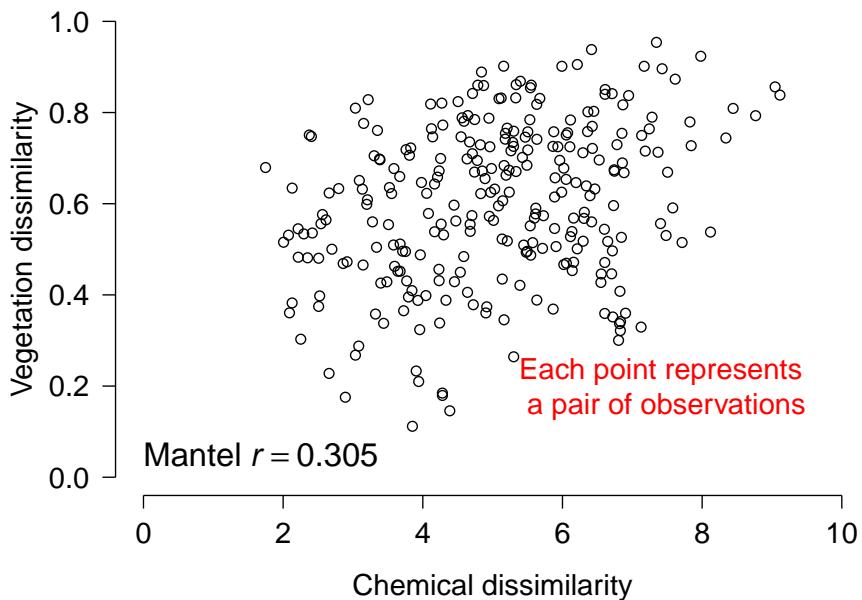
y <- varespec

dy <- vegdist(varespec) # Bray-Curtis metric
dx <- vegdist(scale(varechem), "euclid") # Euclidean metric
mantel(dx, dy)

## 
## Mantel statistic based on Pearson's product-moment correlation
##
## Call:
## mantel(xdis = dx, ydis = dy)
##
## Mantel statistic r: 0.3047
##      Significance: 0.002
##
## Upper quantiles of permutations (null model):
##    90%   95% 97.5%   99%
## 0.130 0.152 0.170 0.195
## Permutation: free
## Number of permutations: 999

par(mfrow=c(1,1), mar=c(5.1, 5.1, 1.1, 1.1),
    bty="n", lend=1, las=1, cex.axis=1.3, cex.lab=1.3)
plot(dx, dy, xlab="Chemical dissimilarity",
     ylab="Vegetation dissimilarity",
     ylim=c(0,1), xlim=c(0, 10))
text(0, 0.05,
     expression(Mantel~italic(r)==0.305),
     adj=0, cex=1.5)
text(5.38, 0.2,
     "Each point represents \n a pair of observations",
     cex=1.3, adj=0, col="red")

```



The figure above shows the result visually. The Mantel r statistic describes the sign and magnitude of the correlation between dissimilarity in terms of chemistry (x -axis) and dissimilarity in terms of the plant community (y -axis). It is interpreted just like Pearson's r . In the plot it is important to realize that each point represents a *pair* of observations, plotted by the dissimilarities between them. The correct interpretation of this plot is that samples with similar soil chemistry tend to also have similar plant communities. Alternatively, you could say that the larger the difference in soil chemistry, the larger the difference in plant communities (or something like that).

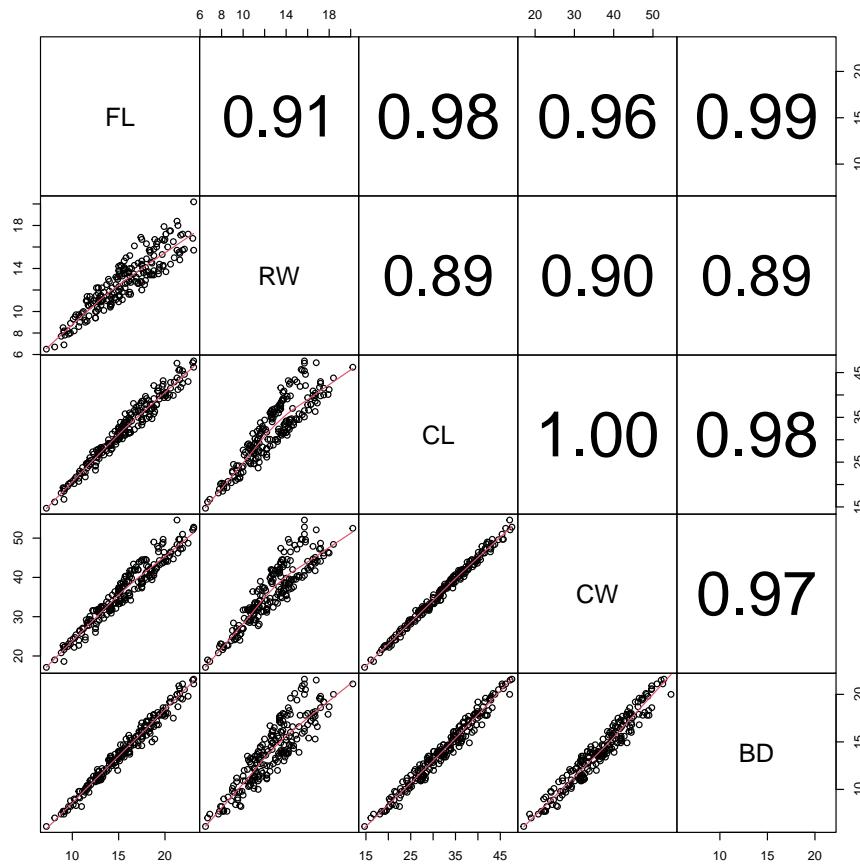
4.8.2.3 Dealing with autocorrelation

Autocorrelation may or may not be a problem in your dataset. As pointed out by Kuehn and Dormann (2012), there is a difference between autocorrelation in a response variable and autocorrelation in the residuals of a statistical model. Autocorrelation in a response variable can be managed by inclusion of appropriate predictor variables (which may themselves be autocorrelated or correlated; see next section). This works because if autocorrelated values can be predicted by a predictor variable, then the autocorrelation is controlled for, by definition. Autocorrelation in residuals, however, represents a pattern in the data that is not accounted for. There are methods for accounting for autocorrelated residuals such as conditional autoregressive models (CAR) (Gelfand and Vounatsou 2003), generalized or ordinary least squares models (GLS or OLS), classification and regression trees or CART (McCune et al. 2002), and mixed models (Bolker et al. 2009). None of these solutions is appropriate for every situation.

4.8.3 Collinearity

Sometimes we face the problem of having predictor variables that are not only related to the response variable, but also related to each other. This phenomenon is called **collinearity**. In large datasets the problem becomes **multicollinearity**, when many variables are related to each other. Including collinear predictor variables in an analysis can severely bias the parameter estimates of that analysis (e.g., regression coefficients) and as a result make the results misleading or not generalizable. The effects of collinear predictor variables are by definition confounded with each other, meaning that the effects cannot be separated (Dormann et al. 2013).

The most straightforward way of detecting collinearity is with scatterplot matrices and correlation coefficients (see here). This is a technique to visualize how many variables in a dataset relate to each other simultaneously. As shown in the linked section, replacing the upper triangle of boxes with the correlation coefficients can help detect potentially problematic pairs of variables. The figure below shows an example from the `crabs` dataset in package `MASS`. In this dataset, all of the variables are highly correlated with each other.



If you have many variables, then a figure like the one above can be hard to read or interpret. One solution might be to create a table of correlation coefficients between every pair of variables. The base function `cor()` can do this, but a better-formatted table takes a little more work.

```
data(crabs, package="MASS")
x <- crabs
dat.cols <- 4:8
cor(x[,dat.cols])
```

	FL	RW	CL	CW	BD
## FL	1.0000000	0.9069876	0.9788418	0.9649558	0.9876272
## RW	0.9069876	1.0000000	0.8927430	0.9004021	0.8892054
## CL	0.9788418	0.8927430	1.0000000	0.9950225	0.9832038
## CW	0.9649558	0.9004021	0.9950225	1.0000000	0.9678117
## BD	0.9876272	0.8892054	0.9832038	0.9678117	1.0000000

```

# better table:
vars <- names(x)[dat.cols]

# generate every unique pair of two variables by name:
cx <- data.frame(t(combn(vars, 2)))

# calculate correlation coefficients for each pair:
cx$r <- NA
for(i in 1:nrow(cx)){
  cx$r[i] <- cor(x[,cx$X1[i]], x[,cx$X2[i]])
}
cx

##      X1   X2       r
## 1  FL  RW 0.9069876
## 2  FL  CL 0.9788418
## 3  FL  CW 0.9649558
## 4  FL  BD 0.9876272
## 5  RW  CL 0.8927430
## 6  RW  CW 0.9004021
## 7  RW  BD 0.8892054
## 8  CL  CW 0.9950225
## 9  CL  BD 0.9832038
## 10 CW  BD 0.9678117

```

The resulting table can then be sorted or filtered by `r` to identify pairs of variables that are correlated with each other.

```

# not run:
# filter to |r| > 0.7
cx[which(abs(cx$r) > 0.7),]

```

Dormann et al. (2013) suggest to use a cutoff of $|r|$ in between 0.5 and 0.7 to identify collinear pairs of variables. When a pair of variables are identified as collinear, the easiest solution is to simply run the analysis without one of them. This is because the variables are redundant, and any conclusions made using one would also be obtained using the other.

4.8.4 Missing data

The last target of exploratory data analysis that we will explore is **missing data**. Data can be missing for many reasons. Equipment failures, human error, natural disasters, and other causes can cause values in your dataset to go missing or otherwise be invalid. How to handle the missing data is up to you as a biologist: *there is no one-size-fits-all statistical solution*. The two basic ways to deal with missing data are to (1) not deal with it and (2) estimate the missing values.

4.8.4.1 Option 1: Ignore missing data

The easiest solution is simply to ignore missing data. Consider this dataset, which contains morphological data on 63 species of bats (Hutcheon et al. 2002). This dataset is modified from an example in the unit on nonlinear models. Notice that in this version the brain masses (`brw`) are missing for 3 species of bats.

```
in.name <- "missing_data_example.csv"
dat <- read.csv(in.name, header=TRUE)
```

Because there are a lot of data, one option for analyzing this dataset would be to simply ignore the missing observations. For example, if someone was interested in the relationship between body size (variable `bow`) and brain mass, they could just fit a model and R would (for most methods) automatically ignore missing values. Note that in this example both variables are log-transformed to ensure normality and positivity.

```
dat$logbw <- log(dat$bow)
dat$logbr <- log(dat$brw)

mod1 <- lm(logbr~logbw, data=dat)
summary(mod1)
```

```
##
## Call:
## lm(formula = logbr ~ logbw, data = dat)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -0.82879 -0.16408  0.05009  0.19548  0.35881
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 3.96599   0.09476  41.85  <2e-16 ***
## logbw       0.75322   0.02881  26.14  <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.2616 on 58 degrees of freedom
##   (3 observations deleted due to missingness)
## Multiple R-squared:  0.9218, Adjusted R-squared:  0.9204
## F-statistic: 683.4 on 1 and 58 DF,  p-value: < 2.2e-16
```

The parameter estimates of intercept = 3.9659 and slope = 0.7532 are very close to the values obtained using the full dataset of 3.9685 and 0.7516. In this case, ignoring the missing data worked just fine.

4.8.4.2 Option 2: Interpolate or impute missing data

If you need to replace the missing data, then the best option is to **interpolate** or **impute** them. How exactly to do this depends on the question being asked. If there is a clear relationship between the variable with missing values and another variable, then you can use that relationship to estimate the missing values. You can also use **multiple imputation** to estimate the missing values using information from several related variables at once. The example below illustrates simple interpolation (aka: single imputation) to replace the missing brain weights using known body masses.

```
# flag missing observations
flag <- which(is.na(dat$brw))

# log transform (if not done already)
dat$logbw <- log(dat$bow)
dat$logbr <- log(dat$brw)

# fit model
mod1 <- lm(logbr~logbw, data=dat)

# predict missing logbr at logbw
px <- data.frame(logbw=dat$logbw[flag])
pr <- predict(mod1, newdata=data.frame(px))

# insert predicted values
dat$logbr[flag] <- pr

# back-transform to original scale
dat$brw[flag] <- exp(dat$logbr[flag])
```

The interpolated values are similar to the true values, but with quite a bit of error. The term for this is “correct to an order of magnitude”.

```
dat$brw[flag]

## [1] 729.7784 507.7160 2138.6809
true.vals <- c(597, 543, 2070)

# percentage bias in interpolated values:
100 * (dat$brw[flag]-true.vals) / true.vals

## [1] 22.240932 -6.497972 3.317919
```

Interpolation can be improved by predicting the missing values using additional variables. Multiple imputation, like any statistical model, can suffer from the effects of multicollinearity in the “predictors”, so be mindful that you don’t obtain—and use—biased estimates of your missing values. Penone et al. (2014)

reviewed some strategies for imputing missing data in the context of life history traits, but the approaches could be applied to other areas of biology.

Chapter 5

Generalized linear models (GLM)

5.1 Prelude with linear models

If you've had an introductory statistics course, then you are probably familiar with the **linear regression** model. Ordinary linear regression, or simple linear regression, is probably the simplest of the inferential statistical techniques that relates two continuous variables. In addition, linear models can be used with categorical predictors called **factors**. Several models that you've probably heard of are actually all special cases of the linear model:

- **Linear regression:** Continuous response and continuous one predictor.
- **Multiple linear regression:** Continuous response and more than 1 continuous predictor.
- **Two-sample *t*-test:** Continuous response and 1 factor with 2 levels.
- **Analysis of variance (ANOVA):** Continuous response and ≥ 1 factor, where each factor can have ≥ 2 levels.
- **Analysis of covariance:** Continuous response variable with ≥ 1 continuous and ≥ 1 factor predictor.
- ... and many more.

What makes all of these methods linear models is that there exists a way to rewrite them so that the response variable is a linear function of the predictors (Bolker 2008).

Many situations are, at least at first glance, amenable to linear regression. But, as we'll see later, many situations are better treated by more elaborate methods such as GLM or mixed models (Zuur et al. 2007, Bolker 2008, Bolker et al. 2009). When learning those more advanced methods it is easiest to think of them as extensions of the linear model, but it's actually the other way around:

linear regression is a special case of the more advanced methods.

In linear regression with one predictor variable, a response variable Y is modeled as a linear function of some explanatory variable X . The model can be written in several ways. The most common way is:

$$\begin{aligned} Y &= \beta_0 + \beta_1 X + \varepsilon \\ \varepsilon &\sim \text{Normal}(0, \sigma^2) \end{aligned}$$

In this equation:

- Y is the response variable (AKA: dependent variable)
- X is the explanatory variable (AKA: independent variable or predictor variable)
- β_0 is the Y -intercept (i.e., the value of Y when $X = 0$). Called “beta zero” or “beta naught”.
- β_1 is the slope or regression coefficient (i.e., the change in Y per unit change in X). Called “beta one”.
- ε is a random error term that describes residual variation not explained by the model. Residuals are identically and independently distributed (*i.i.d.*) according to a normal distribution with mean 0 and variance σ^2 . Called “epsilon” or “the error term”.
- The residual variance σ^2 is estimated from the data. As you might expect, larger residual variance indicates a poorer fit to the data. Called “sigma squared” or the “residual variance”. Some authors might use the standard deviation (SD) σ instead of σ^2 to describe the residual variation¹.

You might recognize most of this model as the equation for the slope of a line that you learned in high school algebra: $Y = mx + b$, but b is now called β_0 and m is now called β_1 . The model intercept β_0 is the **Y -intercept**, or the value that Y takes when $X = 0$. The model **slope** β_1 is the amount that Y changes per unit increase in X . That is, if X increases by 1, then Y increases by β_1 .

The linear regression model is sometimes written in other ways. Here are some other notations you might see:

Observation-wise notation

$$\begin{aligned} Y_i &= \beta_0 + \beta_1 X_i + \varepsilon_i \\ \varepsilon_i &\sim \text{Normal}(0, \sigma^2) \end{aligned}$$

This is very similar to the notation above, but it emphasizes that the residual is different for each observation i . More complicated models with additional

¹This is mirrored in how different statistical software packages express the variability of a normal distribution. R uses the SD; others use variance or precision (reciprocal of SD or variance). Read the manual!

predictors might use this form to emphasize that each observed Y value (Y_i) depends on a combination of predictor values ($X_{1,i}$, $X_{2,i}$, and so on) specific to observation i .

State-space notation

$$Y \sim \text{Normal}(E(Y), \sigma^2)$$

$$E(Y) = \beta_0 + \beta_1 X$$

The state-space notation explicitly separates the deterministic part of the model, which describes the “state” of Y , from the stochastic part, which has more to do with the “space” of potential observations. Thinking about statistical models in a state-space way is key to understanding many advanced statistical methods, including GLM and mixed models. The state-space form can also be written to include subscripts for each observation.

Matrix notation

$$\mathbf{Y} = \mathbf{X}\beta + \varepsilon$$

The bold letters in this notation signify that \mathbf{Y} , β , \mathbf{X} , and ε are either vectors or matrices². The matrix \mathbf{X} is sometimes called the **design matrix**. This term comes up a lot in the R documentation and in discussion of different statistical methods (this is how most statisticians think of many statistical models).

The matrix notation is a compact way of writing the linear model and shows how linear models can be expressed (and calculated) using linear algebra. Note that β is written after \mathbf{X} here, to signify the order in which terms must occur for matrix multiplication to be defined. The matrix notation above can be expanded as:

$$\begin{bmatrix} Y_1 \\ Y_2 \\ \vdots \\ Y_n \end{bmatrix} = \begin{bmatrix} 1 & X_{1,1} & \cdots & X_{1,p} \\ 1 & X_{2,1} & \cdots & X_{2,p} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & X_{n,1} & \cdots & X_{n,p} \end{bmatrix} \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \vdots \\ \beta_p \end{bmatrix} + \begin{bmatrix} \varepsilon_1 \\ \varepsilon_2 \\ \vdots \\ \varepsilon_n \end{bmatrix}$$

where n is the number of observations; Y_1 , Y_2 , ..., Y_n is a vector of observed values of the dependent variable (aka: response variable); p is the number of linear predictors; $X_{i,j}$ is the i -th value of predictor j ; $\beta_0, \beta_1, \dots, \beta_p$ is the vector of regression coefficients; and $\varepsilon_1, \varepsilon_2, \dots, \varepsilon_n$ is a vector of *i.i.d.* normal residuals. This matrix notation illustrates how linear regression can be easily extended to models with multiple predictor variables: multiple linear regression (see below).

²The β and ε are supposed to be bold, but I can't figure out the right combination of markdown and Latex to get bold Greek letters.

Linear regression models are fit by finding a slope and coefficient (or coefficients) that minimize the sums of squared errors, or sum of squared deviations from the expected value, or simply, residuals. Sum of squared residuals is calculated as:

$$SS_{res} = \sum_{i=1}^n (Y_i - \beta_0 + \beta_1 X_i)^2$$

This expression shows why some authors prefer the observation-wise notation for linear regression. For ordinary linear regression, the optimal values of β_0 and β_1 can be calculated as:

$$\hat{\beta}_1 = \frac{\sum_{i=1}^n (X_i - \bar{X})(Y_i - \bar{Y})}{\sum_{i=1}^n (X_i - \bar{X})^2}$$

$$\hat{\beta}_0 = \bar{Y} - \hat{\beta}_1 \bar{X}$$

The results $\hat{\beta}_0$ and $\hat{\beta}_1$ are referred to as “estimates” because we do not assume that our analysis gives us the “true” values of $\hat{\beta}_0$ and $\hat{\beta}_1$. The “hat” symbols signify that these parameters are estimates. These symbols are usually pronounced “beta zero hat” and “beta one hat”.

The residual variance $\hat{\sigma}^2$ is then estimated as:

$$\hat{\sigma}^2 = \frac{SS_{res}}{n - 2}$$

Expressions for SS_{res} and the estimators in cases with >1 linear predictor are just more elaborate versions of the expressions for simple linear regression.

5.1.1 Assumptions of linear models

Like all statistical models, linear regression carries with it many assumptions about the nature of the data and the relationships contained within the data. These assumptions are rarely met perfectly in real datasets, but minor violations are usually fine if they are identified and understood.

Assumption 1: Linearity

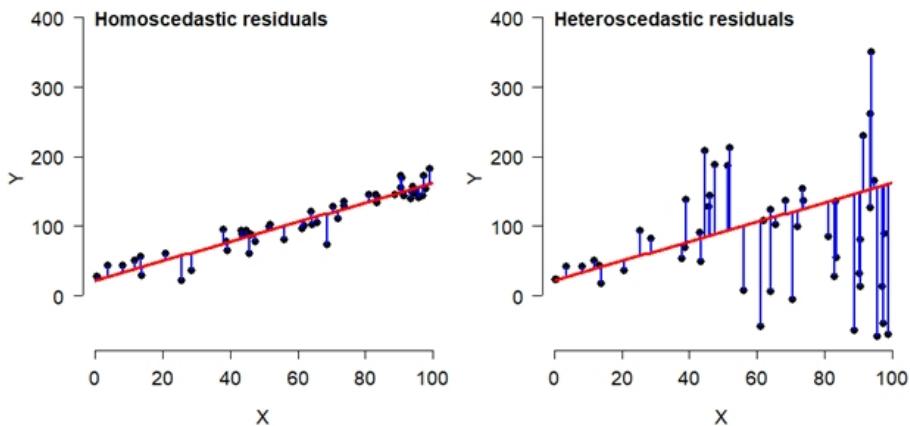
Linear regression assumes that there is a linear relationship between the predictor variable (or variables) and the response variable. This does not mean that the relationship is always a straight line: variables can be transformed to achieve linearity. For example, polynomial models can be considered linear because the dependent value Y varies as a linear function of X values raised to a power, not as powers of the X values:

$$Y = \beta_0 + \beta_1 X + \beta_2 X^2$$

In this quadratic model, Y is a linear function of X and X^2 , according to the slopes β_1 and β_2 . This is because a plot of Y vs. X would be a straight line with slope β_1 . Likewise, a plot of Y vs. X^2 would be a straight line with slope β_2 (even though a plot of Y vs. X would be a parabola).

Assumption 2: Constant variance (aka: homoscedasticity)

Another key assumption of linear models is that the variance of the errors is the same for all values of the predictor variables. This condition is called **homoscedasticity**. Violations of this assumption, where error variance is not the same for all X , lead to a situation called **heteroscedasticity**. To relate this to the model equations above, the residual variance σ^2 is the same for all X . Consider the two scatterplots below, both showing some data with a fitted linear model. The left plot shows a homoscedastic relationship. The right plot does not. Heteroscedasticity is a serious problem for linear models because it leads to biased parameter estimates and standard errors (SE) of those estimates. The latter issue means that significance tests on parameters will be incorrect.



Assumption 3: Fixed predictor values

A third assumption is that the X values are precisely known. If there is uncertainty in the X variables, this adds uncertainty to the Y values and the relationship between Y and X that linear regression cannot account for. Simply put, linear regression has a term for uncertainty in Y (σ^2), but no term for uncertainty in X .

Assumption 4: Independent and identically distributed errors (*i.i.d.*)

The assumption of independently and identically distributed (*i.i.d.*) errors is very important. It means that the residual, or predictive error, for each observation depends only on that observation and not on predictor variables in other observations. When the assumption of independence is violated, the degrees of freedom in the analysis is artificially inflated, increasing the chance of

a type I error (false positive). There are methods to deal with errors that are not independent, but linear regression is not one of them.

Assumption 5: Independent predictors

The final assumption of the linear model is that predictor variables are (mostly) independent of each other. When predictor variables are correlated, or collinear, the precision of parameter estimates suffers. When predictor variables are perfectly collinear, the linear model cannot be fit at all because the parameters (effects of different predictors) are not uniquely identifiable.

5.1.2 Linear regression in R

The function `lm()` performs linear regression. It is a good idea to save your model to an **object**. We'll use names like `mod1`, short for "model 1". Note that this name uses the numeral 1, not the lower-case letter L 1. These two symbols look very similar in the Courier font used by R.

```
mod1 <- lm(iris$Petal.Width~iris$Petal.Length)
```

Using the `data` argument can make code neater:

```
mod1 <- lm(Petal.Width~Petal.Length, data=iris)
```

The object `mod1` is really a type of object called a **list** with a lot of information about your model. When you write up your analysis you can extract information from `mod1`. The function `str()` prints a lot of information about `mod1` and what it contains:

```
str(mod1)
```

```
## List of 12
## $ coefficients : Named num [1:2] -0.363 0.416
##   ..- attr(*, "names")= chr [1:2] "(Intercept)" "Petal.Length"
## $ residuals    : Named num [1:150] -0.019 -0.019 0.0226 -0.0606 -0.019 ...
##   ..- attr(*, "names")= chr [1:150] "1" "2" "3" "4" ...
## $ effects      : Named num [1:150] -14.6888 8.9588 0.0257 -0.0576 -0.0159 ...
##   ..- attr(*, "names")= chr [1:150] "(Intercept)" "Petal.Length" "" "" ...
## $ rank         : int 2
## $ fitted.values: Named num [1:150] 0.219 0.219 0.177 0.261 0.219 ...
##   ..- attr(*, "names")= chr [1:150] "1" "2" "3" "4" ...
## $ assign        : int [1:2] 0 1
## $ qr           :List of 5
##   ..$ qr    : num [1:150, 1:2] -12.2474 0.0816 0.0816 0.0816 0.0816 ...
##   ... ..- attr(*, "dimnames")=List of 2
##     ... ...$ : chr [1:150] "1" "2" "3" "4" ...
##     ... ...$ : chr [1:2] "(Intercept)" "Petal.Length"
##   ... ..- attr(*, "assign")= int [1:2] 0 1
##   ..$ qraux: num [1:2] 1.08 1.1
```

```

## ..$ pivot: int [1:2] 1 2
## ..$ tol : num 1e-07
## ..$ rank : int 2
## ...- attr(*, "class")= chr "qr"
## $ df.residual : int 148
## $ xlevels      : Named list()
## $ call         : language lm(formula = Petal.Width ~ Petal.Length, data = iris)
## $ terms        :Classes 'terms', 'formula' language Petal.Width ~ Petal.Length
## ... ...- attr(*, "variables")= language list(Petal.Width, Petal.Length)
## ... ...- attr(*, "factors")= int [1:2, 1] 0 1
## ... ... .- attr(*, "dimnames")=List of 2
## ... ... .$. : chr [1:2] "Petal.Width" "Petal.Length"
## ... ... .$. : chr "Petal.Length"
## ... ...- attr(*, "term.labels")= chr "Petal.Length"
## ... ...- attr(*, "order")= int 1
## ... ...- attr(*, "intercept")= int 1
## ... ...- attr(*, "response")= int 1
## ... ...- attr(*, ".Environment")=<environment: R_GlobalEnv>
## ... ...- attr(*, "predvars")= language list(Petal.Width, Petal.Length)
## ... ...- attr(*, "dataClasses")= Named chr [1:2] "numeric" "numeric"
## ... ... .- attr(*, "names")= chr [1:2] "Petal.Width" "Petal.Length"
## $ model       :'data.frame': 150 obs. of 2 variables:
##   ..$ Petal.Width : num [1:150] 0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
##   ..$ Petal.Length: num [1:150] 1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
## ...- attr(*, "terms")=Classes 'terms', 'formula' language Petal.Width ~ Petal.Length
## ... ... .- attr(*, "variables")= language list(Petal.Width, Petal.Length)
## ... ... .- attr(*, "factors")= int [1:2, 1] 0 1
## ... ... .- attr(*, "dimnames")=List of 2
## ... ... .$. : chr [1:2] "Petal.Width" "Petal.Length"
## ... ... .$. : chr "Petal.Length"
## ... ...- attr(*, "term.labels")= chr "Petal.Length"
## ... ...- attr(*, "order")= int 1
## ... ...- attr(*, "intercept")= int 1
## ... ...- attr(*, "response")= int 1
## ... ...- attr(*, ".Environment")=<environment: R_GlobalEnv>
## ... ...- attr(*, "predvars")= language list(Petal.Width, Petal.Length)
## ... ...- attr(*, "dataClasses")= Named chr [1:2] "numeric" "numeric"
## ... ... .- attr(*, "names")= chr [1:2] "Petal.Width" "Petal.Length"
## - attr(*, "class")= chr "lm"

```

Use `summary()` to see the terms and coefficients in the model. Note that the result of `summary(lm())` is an object in its own right, which you can assign to a name and extract information from.

```
summary(mod1)
##
```

```

## Call:
## lm(formula = Petal.Width ~ Petal.Length, data = iris)
##
## Residuals:
##       Min     1Q Median     3Q    Max 
## -0.56515 -0.12358 -0.01898  0.13288  0.64272 
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)    
## (Intercept) -0.363076   0.039762 -9.131  4.7e-16 ***
## Petal.Length  0.415755   0.009582 43.387 < 2e-16 ***
## ---    
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1 
##
## Residual standard error: 0.2065 on 148 degrees of freedom
## Multiple R-squared:  0.9271, Adjusted R-squared:  0.9266 
## F-statistic: 1882 on 1 and 148 DF,  p-value: < 2.2e-16 
str(summary(mod1))
## List of 11
## $ call      : language lm(formula = Petal.Width ~ Petal.Length, data = iris)
## $ terms     : Classes 'terms', 'formula' language Petal.Width ~ Petal.Length
## ... .- attr(*, "variables")= language list(Petal.Width, Petal.Length)
## ... .- attr(*, "factors")= int [1:2, 1] 0 1
## ... .- attr(*, "dimnames")=List of 2
## ... .- .$. : chr [1:2] "Petal.Width" "Petal.Length"
## ... .- .$. : chr "Petal.Length"
## ... .- attr(*, "term.labels")= chr "Petal.Length"
## ... .- attr(*, "order")= int 1
## ... .- attr(*, "intercept")= int 1
## ... .- attr(*, "response")= int 1
## ... .- attr(*, ".Environment")=<environment: R_GlobalEnv>
## ... .- attr(*, "predvars")= language list(Petal.Width, Petal.Length)
## ... .- attr(*, "dataClasses")= Named chr [1:2] "numeric" "numeric"
## ... .- attr(*, "names")= chr [1:2] "Petal.Width" "Petal.Length"
## $ residuals  : Named num [1:150] -0.019 -0.019 0.0226 -0.0606 -0.019 ...
## ..- attr(*, "names")= chr [1:150] "1" "2" "3" "4" ...
## $ coefficients: num [1:2, 1:4] -0.36308 0.41576 0.03976 0.00958 -9.13122 ...
## ..- attr(*, "dimnames")=List of 2
## ... .$. : chr [1:2] "(Intercept)" "Petal.Length"
## ... .$. : chr [1:4] "Estimate" "Std. Error" "t value" "Pr(>|t|)" 
## $ aliased    : Named logi [1:2] FALSE FALSE
## ..- attr(*, "names")= chr [1:2] "(Intercept)" "Petal.Length"
## $ sigma      : num 0.206
## $ df         : int [1:3] 2 148 2
## $ r.squared   : num 0.927

```

```

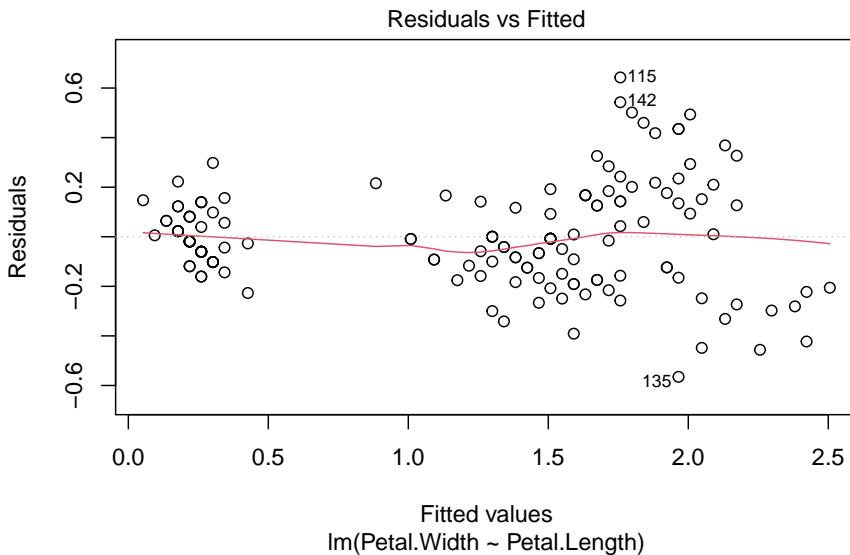
## $ adj.r.squared: num 0.927
## $ fstatistic : Named num [1:3] 1882 1 148
## ..- attr(*, "names")= chr [1:3] "value" "numdf" "dendf"
## $ cov.unscaled : num [1:2, 1:2] 0.03708 -0.00809 -0.00809 0.00215
## ..- attr(*, "dimnames")=List of 2
## ...$ : chr [1:2] "(Intercept)" "Petal.Length"
## ...$ : chr [1:2] "(Intercept)" "Petal.Length"
## - attr(*, "class")= chr "summary.lm"
summary(mod1)$coefficients
##             Estimate Std. Error t value    Pr(>|t|)
## (Intercept) -0.3630755 0.039761990 -9.131221 4.699798e-16
## Petal.Length  0.4157554 0.009582436 43.387237 4.675004e-86

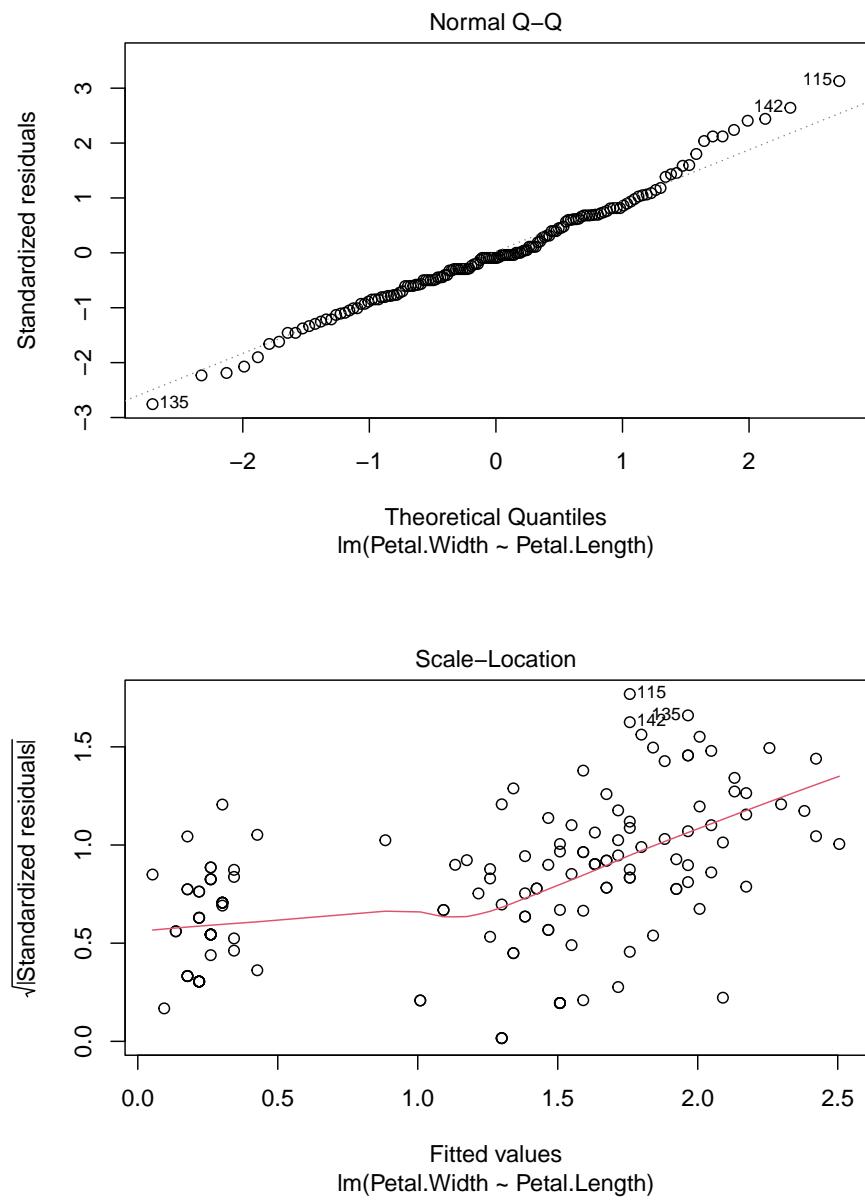
# alternatively:
mod1.sum <- summary(mod1)
mod1.sum$coefficients
##             Estimate Std. Error t value    Pr(>|t|)
## (Intercept) -0.3630755 0.039761990 -9.131221 4.699798e-16
## Petal.Length  0.4157554 0.009582436 43.387237 4.675004e-86

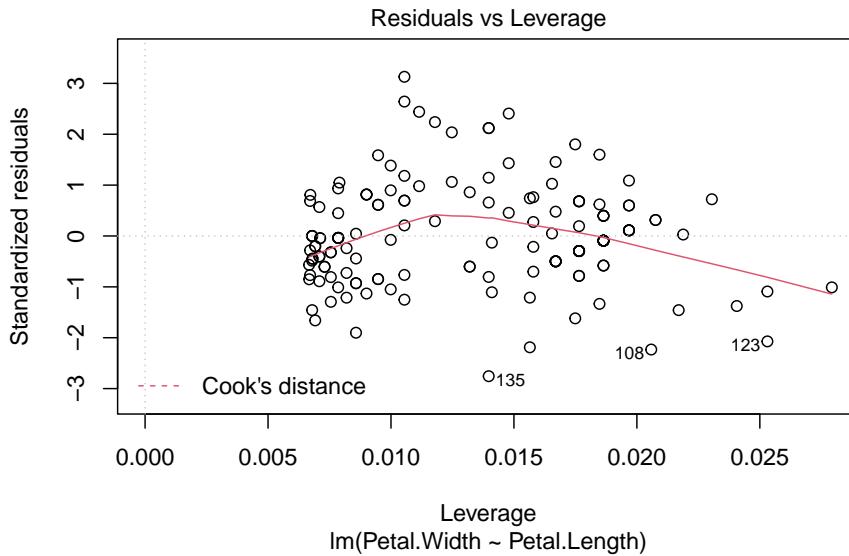
```

Finally, you can get diagnostic plots for the model using `plot()`. These plots are useful for checking whether your data contains heteroscedasticity.

```
plot(mod1)
```







Next, let's get the predicted values and 95% confidence interval (CI) from the model. First, we use `predict()` to calculate the expected value (i.e., mean value of the response variable) for a range of X values. Then, we use the predicted standard error of the prediction to calculate the CI. Finally, we assemble everything in a plot.

```
# number of points for prediction
new.n <- 100

# x values for prediction
newx <- seq(min(iris$Petal.Length),
             max(iris$Petal.Length),
             length=new.n)

# put new X values in data frame
newdat <- data.frame(Petal.Length=newx)

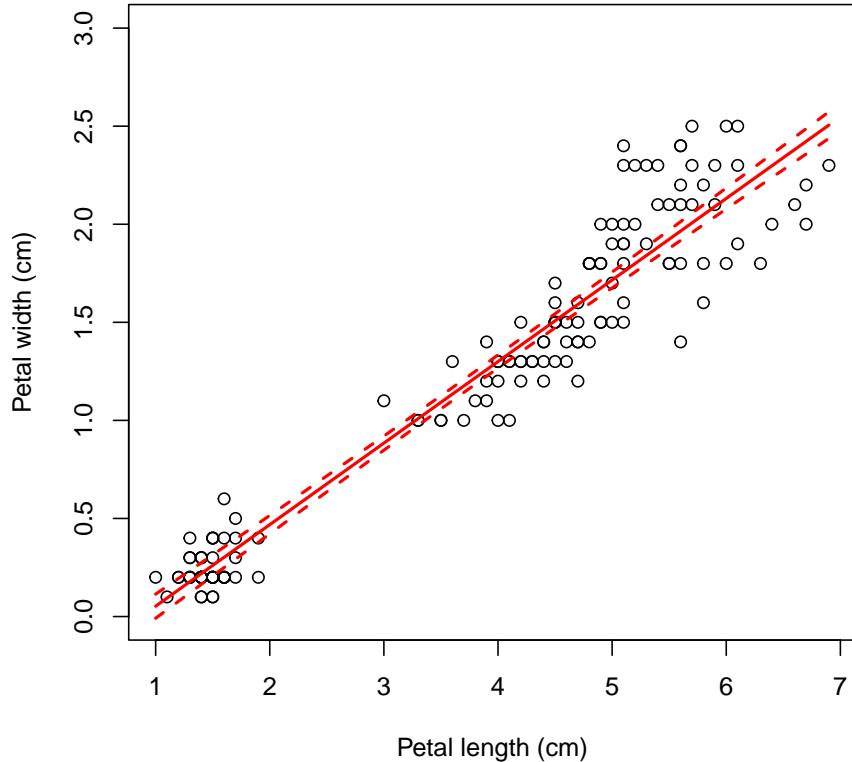
# calculate prediction and 95% CI
pred <- predict(mod1, newdata=data.frame(newdat),
                 se.fit=TRUE)
mm <- qnorm(0.5, pred$fit, pred$se.fit)
lo <- qnorm(0.025, pred$fit, pred$se.fit)
up <- qnorm(0.975, pred$fit, pred$se.fit)

# make plot with lines for prediction and CI
```

```

plot(iris$Petal.Length, iris$Petal.Width,
      ylim=c(0, 3),
      xlab="Petal length (cm)",
      ylab="Petal width (cm)")
points(newx, lo, type="l", col="red", lty=2, lwd=2)
points(newx, up, type="l", col="red", lty=2, lwd=2)
points(newx, mm, type="l", lwd=2, col="red")

```



Below is an alternative version of the plot that uses a shaded area to show the 95% CI. The shaded area is drawn with the `polygon()` function.

```

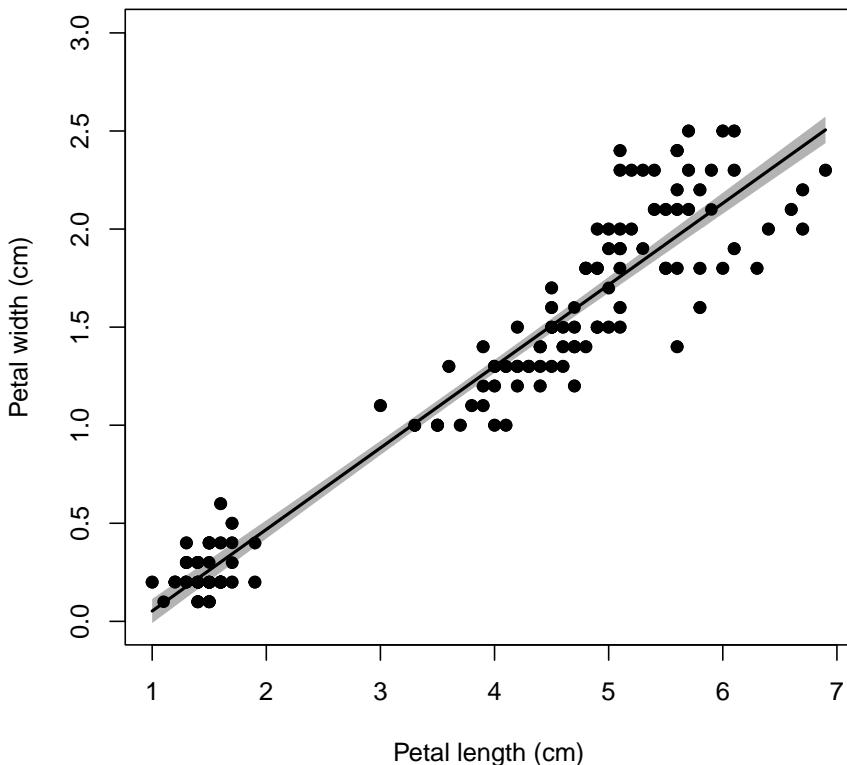
# alternative version: use shaded area for 95% CI
# note that polygon() goes first
plot(iris$Petal.Length, iris$Petal.Width,
      ylim=c(0, 3),
      xlab="Petal length (cm)",

```

```

ylab="Petal width (cm)")
polygon(x=c(newx, rev(newx)),
        y=c(lo, rev(up)),
        border=NA, col="grey70")
points(newx, mm, type="l", lwd=2)
points(iris$Petal.Length, iris$Petal.Width, pch=16, cex=1.1)

```



5.1.3 Multiple linear regression

The linear model can easily be extended to include multiple predictors. This can be seen in the matrix notation for linear regression. With a single predictor variable X , the matrix form of the model is:

$$\mathbf{Y} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\epsilon}$$

$$\begin{bmatrix} Y_1 \\ Y_2 \\ \vdots \\ Y_n \end{bmatrix} = \begin{bmatrix} 1 & X_1 \\ 1 & X_2 \\ \vdots & \vdots \\ 1 & X_n \end{bmatrix} \begin{bmatrix} \beta_0 \\ \beta_1 \end{bmatrix} + \begin{bmatrix} \epsilon_1 \\ \epsilon_2 \\ \vdots \\ \epsilon_n \end{bmatrix}$$

In this equation, the subscripts on Y , X , and ϵ ³ refer to observations; the subscripts on β identify the intercept (β_0) or slope (β_1). This matrix notation is shorthand for a system of equations:

$$\begin{aligned} Y_1 &= 1 \times \beta_0 + \beta_1 \times X_1 + \epsilon_1 \\ Y_2 &= 1 \times \beta_0 + \beta_1 \times X_2 + \epsilon_2 \\ &\vdots \\ Y_n &= 1 \times \beta_0 + \beta_1 \times X_n + \epsilon_n \end{aligned}$$

The matrix shorthand just takes advantage of the rules of matrix multiplication and addition. Multiplying the matrices \mathbf{X} and $\boldsymbol{\beta}$ will result in a matrix with the same number of columns as $\boldsymbol{\beta}$ and the same number of rows as \mathbf{X} . In other words, an $n \times 1$ matrix like \mathbf{Y} , also called a column vector.

With additional predictors, the design matrix is expanded with additional columns to contain the additional predictors. At the same time, the coefficient matrix $\boldsymbol{\beta}$ is expanded with the same number of additional rows. The example below shows the matrix form for a regression model with 2 predictors X_1 and X_2 .

$$\begin{bmatrix} Y_1 \\ Y_2 \\ \vdots \\ Y_n \end{bmatrix} = \begin{bmatrix} 1 & X_{1,1} & X_{2,1} \\ 1 & X_{1,2} & X_{2,2} \\ \vdots & \vdots & \vdots \\ 1 & X_{1,n} & X_{2,n} \end{bmatrix} \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \end{bmatrix} + \begin{bmatrix} \epsilon_1 \\ \epsilon_2 \\ \vdots \\ \epsilon_n \end{bmatrix}$$

Fortunately, R takes care of all this matrix algebra for you. The linear model function `lm()` will automatically define the design and coefficient matrices and estimate the latter. All you need to do is define the model equation using the

³This is another version of ϵ .

formula interface as before. Predictor variables go on the right side of the `~`, separated by `+`.

```
mod2 <- lm(Petal.Length~Sepal.Length+Sepal.Width, data=iris)
summary(mod2)

##
## Call:
## lm(formula = Petal.Length ~ Sepal.Length + Sepal.Width, data = iris)
##
## Residuals:
##     Min      1Q  Median      3Q     Max 
## -1.25582 -0.46922 -0.05741  0.45530  1.75599 
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)    
## (Intercept) -2.52476   0.56344 -4.481 1.48e-05 ***
## Sepal.Length  1.77559   0.06441 27.569 < 2e-16 ***
## Sepal.Width   -1.33862   0.12236 -10.940 < 2e-16 *** 
## ---      
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.6465 on 147 degrees of freedom
## Multiple R-squared:  0.8677, Adjusted R-squared:  0.8659 
## F-statistic:  482 on 2 and 147 DF,  p-value: < 2.2e-16
```

The results show that petal length increases by 1.77 cm for every additional cm of sepal length, and decreases by 1.33 cm for each additional cm of sepal width. The effects of the two predictors are assumed to be **orthogonal**, or independent of each other.

5.1.4 ANOVA and ANCOVA with `lm()`

The formula interface is very flexible and can be used to define many kinds of models. If all predictor variables are factors (AKA: grouping or categorical variables), then the result is an analysis of variance model (ANOVA). The ANOVA table can be obtained with function `anova()`, and post-hoc tests performed on the result of `aov()`. R uses function `lm()` to fit ANOVA models because ANOVA is a special case of the linear model.

```
mod3 <- lm(Petal.Length~Species, data=iris)
summary(mod3)

##
## Call:
## lm(formula = Petal.Length ~ Species, data = iris)
##
## Residuals:
```

```

##      Min     1Q Median     3Q    Max
## -1.260 -0.258  0.038  0.240  1.348
##
## Coefficients:
##                               Estimate Std. Error t value Pr(>|t|)
## (Intercept)           1.46200   0.06086  24.02 <2e-16 ***
## Speciesversicolor  2.79800   0.08607  32.51 <2e-16 ***
## Speciesvirginica   4.09000   0.08607  47.52 <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.4303 on 147 degrees of freedom
## Multiple R-squared:  0.9414, Adjusted R-squared:  0.9406
## F-statistic: 1180 on 2 and 147 DF,  p-value: < 2.2e-16

```

The model summary shows us that at least some of the species have different means. Notice that effects are shown only for two of the three species (*versicolor* and *virginica*). The level that is first alphabetically is used as a baseline. So, the effects shown for the other two species are the differences in means between those species and the baseline species. For example, species *versicolor* has a mean that is 2.798 cm greater than the mean in species *setosa*.

We can get the omnibus test with `anova()`.

```
# anova table:
anova(mod3)
```

```

## Analysis of Variance Table
##
## Response: Petal.Length
##                  Df Sum Sq Mean Sq F value    Pr(>F)
## Species          2  437.10 218.551 1180.2 < 2.2e-16 ***
## Residuals       147   27.22   0.185
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

```

The ANOVA table shows how the variance is partitioned between that explained by species (expressed as sum of squared errors, `Sum Sq`) and the residual variation. The F value for each explanatory variable is the ratio of the mean squared error (`Mean Sq`) for that factor divided by the mean squared error of the residuals. The F value is then used to calculate P -value (`Pr(>F)`).

```
# post-hoc test (Tukey):
TukeyHSD(aov(mod3))
```

```

## Tukey multiple comparisons of means
## 95% family-wise confidence level
##
## Fit: aov(formula = mod3)

```

```
##
## $Species
##           diff      lwr      upr p adj
## versicolor-setosa   2.798  2.59422 3.00178  0
## virginica-setosa    4.090  3.88622 4.29378  0
## virginica-versicolor 1.292  1.08822 1.49578  0
```

The Tukey honest significant difference (HSD) test shows which levels of the factors in the ANOVA have means that differ from each other. The `diff` is literally the difference between the means of the levels in each row. So, the mean of *I. versicolor* minus the mean of *I. setosa* is 2.798 cm. Negative numbers would indicate that the level listed first had a smaller mean. The 95% CI of the difference is given, followed by a *P*-value that is adjusted for multiple comparisons. The *P*-values shown are not literally 0 (which is not possible). Instead, these should be interpreted as very small. Reporting them as something like “<0.0001” would be appropriate.

Including a continuous predictor with a factor predictor will produce main-effects analysis of covariance (ANCOVA).

```
mod4 <- lm(Petal.Length~Sepal.Length+Species, data=iris)
summary(mod4)

##
## Call:
## lm(formula = Petal.Length ~ Sepal.Length + Species, data = iris)
##
## Residuals:
##     Min      1Q  Median      3Q      Max
## -0.76390 -0.17875  0.00716  0.17461  0.79954
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -1.70234   0.23013 -7.397 1.01e-11 ***
## Sepal.Length  0.63211   0.04527 13.962 < 2e-16 ***
## Speciesversicolor  2.21014   0.07047 31.362 < 2e-16 ***
## Speciesvirginica  3.09000   0.09123 33.870 < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.2826 on 146 degrees of freedom
## Multiple R-squared:  0.9749, Adjusted R-squared:  0.9744
## F-statistic: 1890 on 3 and 146 DF,  p-value: < 2.2e-16
```

The summary table now shows the effects of both the continuous predictor and the factor predictor. As before, the estimated coefficient for a continuous predictor is the change in the response variable for every unit increase in the predictor. The estimates shown for the levels of the factor are the differences

between the means of those levels and the baseline level. To fully make sense of this output, we also need to look at the omnibus test and the Tukey test outputs.

```
anova(mod4)
```

```
## Analysis of Variance Table
##
## Response: Petal.Length
##           Df Sum Sq Mean Sq F value    Pr(>F)
## Sepal.Length   1 352.87 352.87 4419.48 < 2.2e-16 ***
## Species        2  99.80  49.90  624.99 < 2.2e-16 ***
## Residuals     146  11.66   0.08
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
TukeyHSD(aov(mod4))

## Warning in replications(paste("~", xx), data = mf): non-factors ignored:
## Sepal.Length

## Warning in TukeyHSD.aov(aov(mod4)): 'which' specified some non-factors which
## will be dropped

## Tukey multiple comparisons of means
## 95% family-wise confidence level
##
## Fit: aov(formula = mod4)
##
## $Species
##               diff      lwr      upr     p adj
## versicolor-setosa 1.0696573 0.93584210 1.2034726 0.0000000
## virginica-setosa 1.1499590 1.01614380 1.2837743 0.0000000
## virginica-versicolor 0.0803017 -0.05351353 0.2141169 0.3327997
```

The omnibus test and Tukey test above demonstrate the effects of the continuous predictor and the factor are orthogonal: no sums of squared errors are shared between the variables. Another way of putting that is that the response to one predictor does not depend on any other predictor.

When the effect of one predictor alters the effect of another predictor, the two predictors are said to “interact”. Interactions between predictors can be specified in R in two ways: * or :. Most of the time you should use *, which automatically fits both predictors by themselves (“main effects”) and the interaction between them (“interaction term”). The model produced this way is also called an “interaction effects ANCOVA” or “ANCOVA with interaction”.

```
f1 <- formula("Petal.Length~Sepal.Length*Species")
mod5 <- lm(f1, data=iris)
summary(mod5)
```

```

## 
## Call:
## lm(formula = f1, data = iris)
##
## Residuals:
##   Min     1Q Median     3Q    Max
## -0.68611 -0.13442 -0.00856  0.15966  0.79607
##
## Coefficients:
##                               Estimate Std. Error t value Pr(>|t|)
## (Intercept)                 0.8031    0.5310   1.512   0.133
## Sepal.Length                  0.1316    0.1058   1.244   0.216
## Speciesversicolor          -0.6179    0.6837  -0.904   0.368
## Speciesvirginica           -0.1926    0.6578  -0.293   0.770
## Sepal.Length:Speciesversicolor  0.5548    0.1281   4.330 2.78e-05 ***
## Sepal.Length:Speciesvirginica  0.6184    0.1210   5.111 1.00e-06 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.2611 on 144 degrees of freedom
## Multiple R-squared:  0.9789, Adjusted R-squared:  0.9781
## F-statistic: 1333 on 5 and 144 DF, p-value: < 2.2e-16

```

The coefficients for the interaction term are `Sepal.Length:Speciesversicolor` and `Sepal.Length:Speciesvirginica`. These should be interpreted as “the change in the effect of sepal length when species is versicolor instead of setosa” and “the change in the effect of sepal length when species is virginica instead of setosa”.

The ANOVA table shows us that the effects of sepal length and species are NOT orthogonal, because some of the sums of squares (1.84) are associated with both variables.

```
anova(mod5)
```

```

## Analysis of Variance Table
##
## Response: Petal.Length
##                               Df Sum Sq Mean Sq  F value    Pr(>F)
## Sepal.Length                  1 352.87  352.87 5175.537 < 2.2e-16 ***
## Species                      2  99.80   49.90  731.905 < 2.2e-16 ***
## Sepal.Length:Species         2   1.84    0.92   13.489 4.272e-06 ***
## Residuals                     144   9.82    0.07
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

```

5.1.5 Variations on linear models

Models without Y -intercepts can be fit by including `0+` in the formula. Sometimes fitting a model without an intercept can be appropriate if there is a natural reason why the response variable must be 0 when the predictor variable is 0. In such a case, any β_0 fit by the model would likely be non-significant or meaningless anyway.

```
# simulate some random data
set.seed(123)
n <- 20
x <- runif(n, 0, 20)
y <- 2.3 * x + rnorm(n, 0, 3)

# full model with intercept and slope:
mod7a <- lm(y~x)
summary(mod7a)

##
## Call:
## lm(formula = y ~ x)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -4.2284 -1.8960 -0.2605  2.0599  5.6778
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 1.8488     1.3149   1.406   0.177
## x           2.1028     0.1044  20.146  8.5e-14 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 2.852 on 18 degrees of freedom
## Multiple R-squared:  0.9575, Adjusted R-squared:  0.9552
## F-statistic: 405.8 on 1 and 18 DF,  p-value: 8.498e-14

# model with slope only (i.e., no intercept)
mod7b <- lm(y~0+x)
summary(mod7b)

##
## Call:
## lm(formula = y ~ 0 + x)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -4.6702 -1.7713  0.2444  2.3790  5.4235
```

```
## 
## Coefficients:
##   Estimate Std. Error t value Pr(>|t|)    
## x  2.23111   0.05192  42.98   <2e-16 ***
## --- 
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
## 
## Residual standard error: 2.925 on 19 degrees of freedom
## Multiple R-squared:  0.9898, Adjusted R-squared:  0.9893 
## F-statistic: 1847 on 1 and 19 DF,  p-value: < 2.2e-16
```

You can also fit a model with only a constant as the predictor using `1` as the right-hand side of the model formula (numeral 1, not lower-case letter L). This is also called an **intercept-only model**, a **model of the mean**, or sometimes a **null model**. A null model with no predictors can be a useful baseline against which to measure other models.

```
mod8 <- lm(Petal.Width~1, data=iris)
summary(mod8)
```

```
## 
## Call:
## lm(formula = Petal.Width ~ 1, data = iris)
## 
## Residuals:
##    Min     1Q Median     3Q    Max 
## -1.0993 -0.8993  0.1007  0.6007  1.3007 
## 
## Coefficients:
##   Estimate Std. Error t value Pr(>|t|)    
## (Intercept)  1.19933   0.06224  19.27   <2e-16 ***
## --- 
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
## 
## Residual standard error: 0.7622 on 149 degrees of freedom
```

The last regression flavor that we'll cover is regression on **ranks**. This uses ranks of the variables rather than their actual values. A variable can be converted to ranks with function `rank()`. As with any other transformation, it might be better to transform the data in place and call the transformed data inside `lm()`.

```
mod10 <- lm(rank(Petal.Width)~rank(Petal.Length), data=iris)
summary(mod10)
```

```
## 
## Call:
## lm(formula = rank(Petal.Width) ~ rank(Petal.Length), data = iris)
## 
```

```

## Residuals:
##      Min     1Q Median     3Q    Max
## -42.506 -10.385   0.338 10.184 35.939
##
## Coefficients:
##                               Estimate Std. Error t value Pr(>|t|)
## (Intercept)           4.97766   2.47618   2.01   0.0462 *
## Petal.Length          0.93407   0.02846  32.82 <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 15.08 on 148 degrees of freedom
## Multiple R-squared:  0.8792, Adjusted R-squared:  0.8784
## F-statistic: 1077 on 1 and 148 DF, p-value: < 2.2e-16

# better: transform in place:
iris2 <- iris
iris2$pw.rank <- rank(iris2$Petal.Width)
iris2$pl.rank <- rank(iris2$Petal.Length)
mod10 <- lm(pw.rank ~ pl.rank, data=iris2)
summary(mod10)

##
## Call:
## lm(formula = pw.rank ~ pl.rank, data = iris2)
##
## Residuals:
##      Min     1Q Median     3Q    Max
## -42.506 -10.385   0.338 10.184 35.939
##
## Coefficients:
##                               Estimate Std. Error t value Pr(>|t|)
## (Intercept)           4.97766   2.47618   2.01   0.0462 *
## pl.rank              0.93407   0.02846  32.82 <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 15.08 on 148 degrees of freedom
## Multiple R-squared:  0.8792, Adjusted R-squared:  0.8784
## F-statistic: 1077 on 1 and 148 DF, p-value: < 2.2e-16

```

5.1.6 Example linear regression workflow

The following example shows a typical linear regression workflow. This example uses simulated data. So, the first few commands that simulate the dataset are not part of a usual regression analysis. After the data are generated, we will fit a linear model, examine the outputs, and then plot the results.

```

# simulate the data
set.seed(42)
n <- 30
x <- sample(1:50, n, replace=TRUE)
beta0 <- 30
beta1 <- 1.3
sigma <- 10
y <- rnorm(n, beta0 + beta1 * x, sigma)
dat <- data.frame(x=x, y=y)

# fit the model (analysis workflow starts here!)
mod <- lm(y~x, data=dat)

# examine the results
summary(mod)

##
## Call:
## lm(formula = y ~ x, data = dat)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -25.033  -7.130   2.620   6.855  18.035
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 18.2851    4.6818   3.906 0.000541 ***
## x            1.5752    0.1416  11.128 8.64e-12 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 11.12 on 28 degrees of freedom
## Multiple R-squared:  0.8156, Adjusted R-squared:  0.809
## F-statistic: 123.8 on 1 and 28 DF,  p-value: 8.638e-12

```

The output of `summary()` tells us a lot:

- **Call:** the model that was fit
- **Residuals:** range (min, max) and 1st, 2nd, and 3rd quartiles of the distribution of residuals
- **Coefficients:** the estimated intercept and slope. Note that the slope associated with each predictor is named for that predictor, exactly as it was named in the function call.
- **R-squared:** the coefficient of determination, or proportion of variance in Y explained by X . Use the adjusted R^2 , which penalizes models for the number of predictors and gives a more conservative estimate of the model goodness-of-fit.

- **Omnibus ANOVA test:** the last line of the output tells us a bit about the ANOVA test for overall model significance. You can get the full test with `anova(mod)`.

The “coefficients” section of the output is very important. You can access it directly as a matrix:

```
summary(mod)$coefficients
```

	Estimate	Std. Error	t value	Pr(> t)
## (Intercept)	18.28512	4.6817629	3.905606	5.409231e-04
## x	1.57522	0.1415513	11.128259	8.637921e-12

This table is really a matrix that contains the estimated values of each regression parameter, the standard error (SE) of each estimate, the test statistic t, and the P-value for each parameter. In standard linear regression, the test statistic t is calculated as:

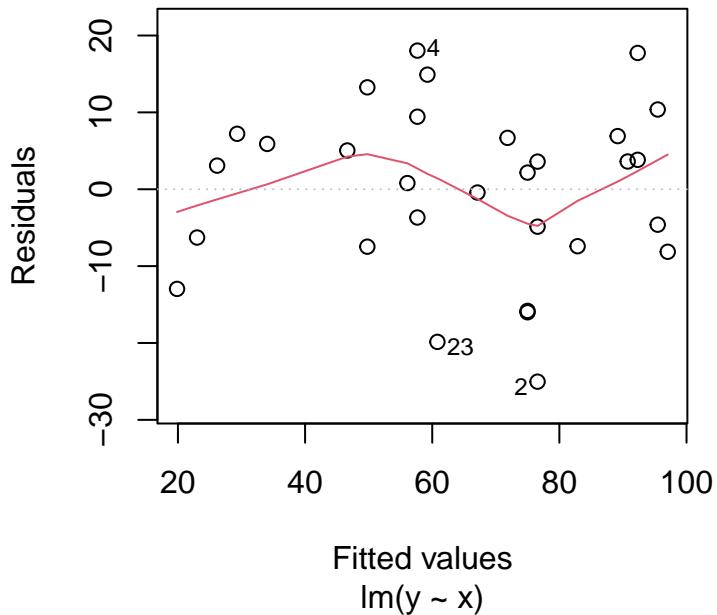
$$t = \frac{\hat{\beta}}{se(\hat{\beta})}$$

and the *P*-value is calculated by assuming that *t* follows a *t* distribution with $n - 2$ degrees of freedom. Note that a *P*-value is calculated for both the model slope and intercept. Often a model intercept will have $P \geq 0.05$. This does not mean that the regression model is invalid or non-significant. This simply means that the data do not support a model intercept significantly different from 0. This is not necessarily a problem: can you think of a biological scenario where an intercept = 0 makes sense?

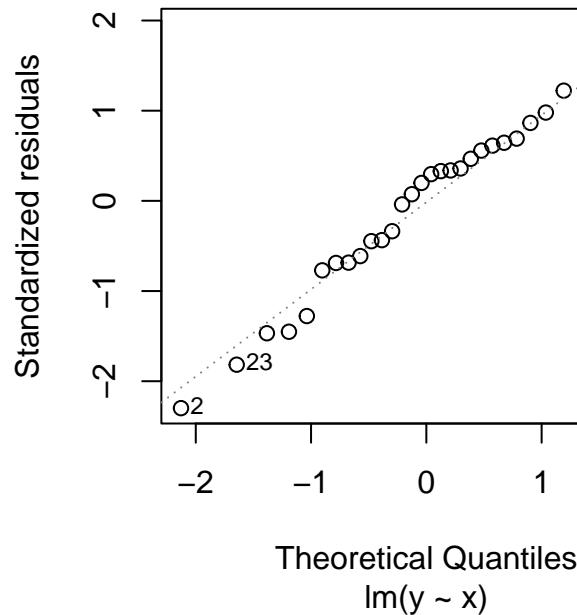
Before moving on, we should examine some of the diagnostic plots that describe our model. These are produced by plotting our model object. This will produce several plots that you can cycle through by clicking on the plot window. The first two are probably the most important.

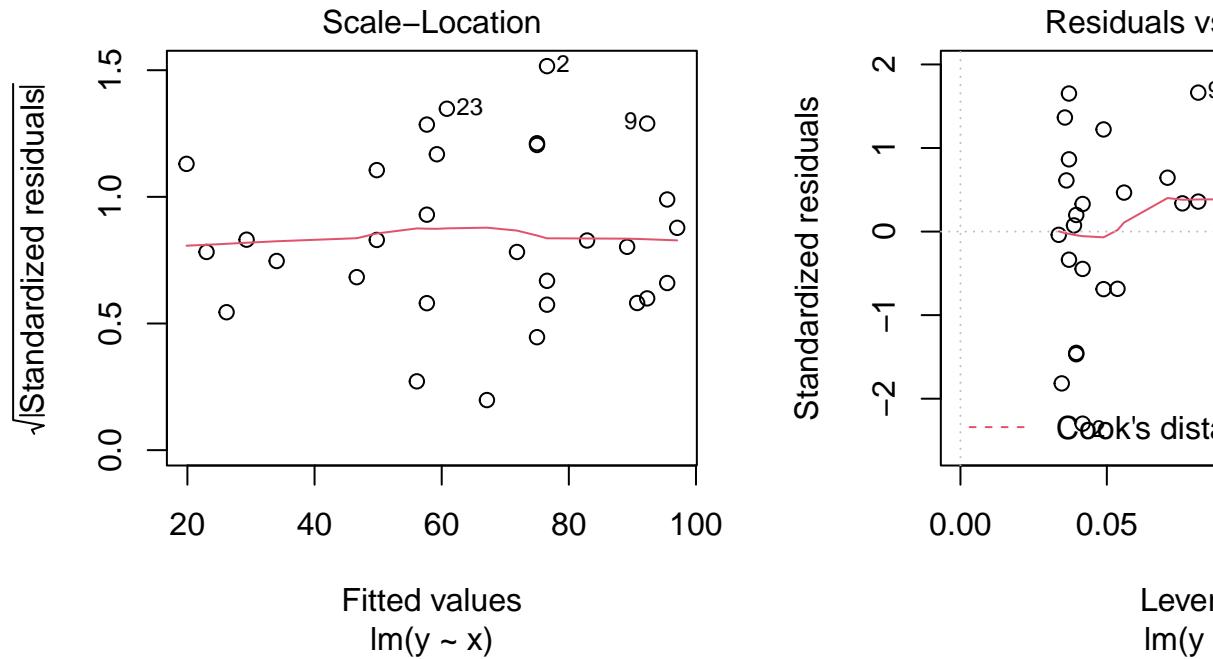
```
plot(mod)
```

Residuals vs Fitted



Normal Q-Q





The first plot shows model residuals vs the fitted values (i.e., the values predicted by the deterministic part of the model). This plot should show no obvious pattern. A pattern, such as residuals increasing or decreasing with predicted value, suggests that the residuals are not normal or i.i.d. Such a pattern is called heteroscedasticity. The plot below shows homoscedasticity, a much better situation. If a plot of residuals vs. fitted has a pattern, you need to investigate where that pattern might be coming from and how it can be eliminated. Potential solutions include adding additional predictors, transforming one or more predictor variables, transforming the response variable, or using a different statistical method (e.g., GLM) that can account for the various sources of heteroscedasticity.

The second plot is called a “quantile-quantile” plot, or “QQ plot” for short. This figure shows standardized residuals (see Module 4) on the y-axis vs. the standard scores of those quantiles in a reference distribution (in this case, a normal distribution with mean 0 and variance equal to the variance of the residuals). The points should fall on or near the diagonal line. Significant departures, particularly near the center of the distribution, suggest that the residuals are not normally distributed. If that is the case, then either the data need to be transformed to achieve normality in the residuals, or another statistical method needs to be explored.

Now that we have a fitted regression model, we might want to present it in a

publication. The usual way is to have (1) a table showing the model coefficients; (2) text describing the model in the Results section, including the R^2 ; and (3) a figure showing the data and fitted model.

The table should resemble the output of `summary(mod)$coefficients`. Note that estimates are rounded to a reasonable number of decimal places. Test statistics and P values can have a couple more digits. If you can, align the numbers on the decimal point, so it is easier to compare numbers up and down a column.

	Estimate \pm SE	t	P
Intercept (β_0)	18.3 \pm 4.7	3.906	0.0005
Slope (β_1)	1.6 \pm 0.1	11.128	<0.0001

In order to plot the data and the relationship, we need to put together a few pieces. Namely, the predicted values (so we can add a “trendline”, as it’s called in Excel) and the 95% CI of the predicted values (so we can show how uncertain the predictions are). R makes both of these tasks straightforward, although not as easy as simply clicking “Add Trendline...” in Excel!

In this example, `px` is a sequence of new values within the domain of the observed X values—do not generate predictions outside your dataset! We are going to use a sequence of new values so that we can smoothly cover the interval between the minimum and maximum of X . This will make for a nicer looking plot.

These values are then supplied to the `predict()` function (which internally calls another function, `predict.lm()`) in a data frame. Note that the names in the new data frame must match the names in original data exactly, and any variable in the model must be in the data frame used for prediction, whether or not a variable was significant. The final argument, `se.fit=TRUE`, specifies that we want the SE of the expected value in addition to the expected value.

```
# new data for prediction
px <- seq(min(x), max(x), length=50)

# prediction with expected mean and SE of Y
pred <- predict(mod, newdata=data.frame(x=px), se.fit=TRUE)
```

Next, we use the `fit` and `se.fit` from the result to define normal distributions at each X value. This way we can calculate what the expected mean and upper and lower confidence limits of the prediction are. You could subtract and add 1.96 SE from/to the mean, but the `qnorm()` method is much cleaner (and, much more adaptable to situations where other distributions apply).

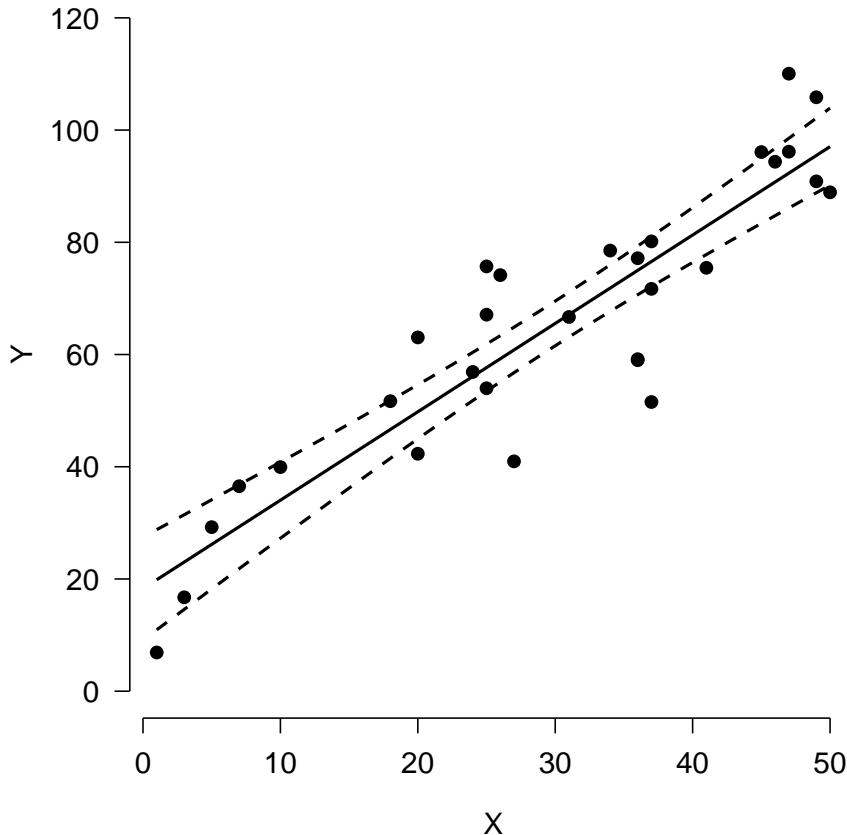
```
# predicted mean and 95% CI
y.mn <- pred$fit
y.lo <- qnorm(0.025, pred$fit, pred$se.fit)
```

```
y.up <- qnorm(0.975, pred$fit, pred$se.fit)
```

Now we are ready to make the plot. The code below sets some graphical options with `par()`, then makes the plot one piece at a time. The `par()` statement is not necessary if you are making graphs for yourself. However, if you are making graphs for your thesis or for a publication, it is essential to producing a clean, professional-quality figure.

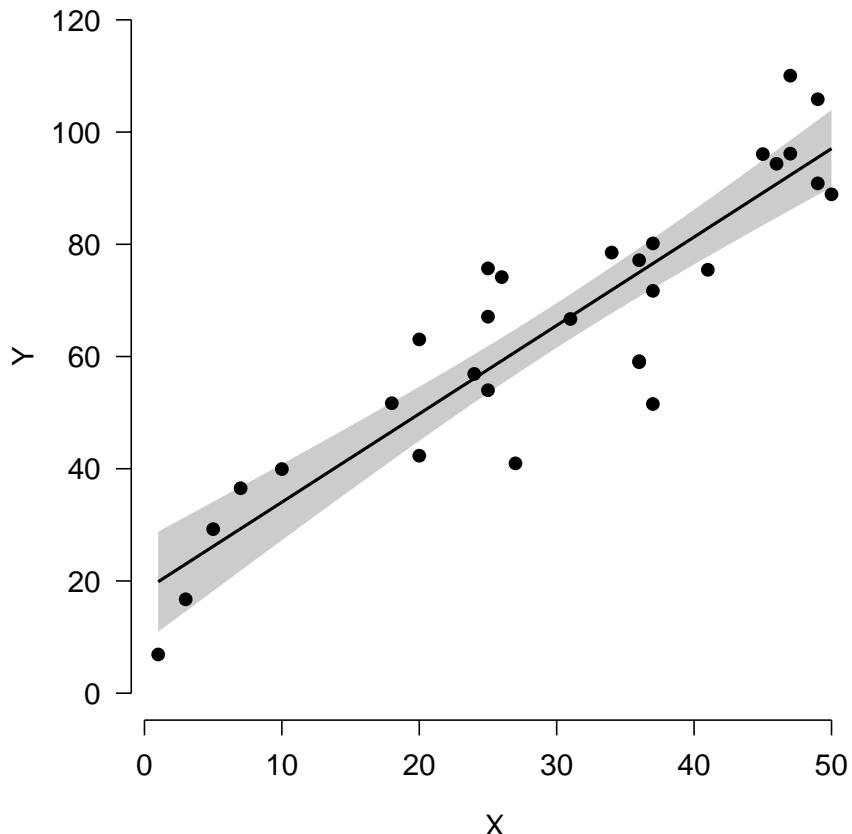
In my workflow I prefer to first make a blank plot, then add the CI and prediction, then the original data. This ensures that the data are not plotted over by one of the other pieces. Note that the *y*-axis limits are set so that 0 is included.

```
par(mfrow=c(1,1), mar=c(5.1, 5.1, 1.1, 1.1),
    bty="n", lend=1,
    las=1, cex.axis=1.2, cex.lab=1.2)
plot(dat$x, dat$y, type="n", ylim=c(0, 120),
     xlab="X", ylab="Y")
points(px, y.lo, type="l", lwd=2, lty=2)
points(px, y.up, type="l", lwd=2, lty=2)
points(px, y.mn, type="l", lwd=2)
points(dat$x, dat$y, pch=16, cex=1.2)
```



Here is an alternative version of the figure, using `polygon()` to plot the confidence interval. Notice how the vertices for the polygon are specified. Half of the values are reversed with `rev()`, because the `polygon()` function basically draws the polygon vertex by vertex in a “circle”.

```
par(mfrow=c(1,1), mar=c(5.1, 5.1, 1.1, 1.1),
  bty="n", lend=1, las=1, cex.axis=1.2, cex.lab=1.2)
plot(dat$x, dat$y, type="n",
  ylim=c(0, 120),
  xlab="X", ylab="Y")
polygon(x=c(px, rev(px)),
  y=c(y.lo, rev(y.up)),
  border=NA, col="grey80")
points(px, y.mn, type="l", lwd=2)
points(dat$x, dat$y, pch=16, cex=1.2)
```



There is one other consideration here: do we want to show the model's predictions and 95% CI of the expected values, or 95% CI of all values? The default is to show the 95% confidence interval, which is a statement about where we expect the **mean or expected value to fall**. The figures above show 95% CI. The alternative is the 95% **prediction interval (PI)**, sometimes called the tolerance interval. The PI describes the region where 95% of **all values might fall**, not just the mean. The PI is always wider than the CI. Most of the time showing the CI is appropriate...just be clear in your write-up what you are showing!

The figure below shows the difference between the CI and PI. Note that the output of `predict()` is different when using the argument `interval="prediction"`, so the code is a little different (we don't have to use `qnorm()` to get the limits).

```
pred2 <- predict(mod, newdata=data.frame(x=px),
  se.fit=TRUE,
  interval="prediction")
y.lo2 <- pred2$fit[, "lwr"]
```

```

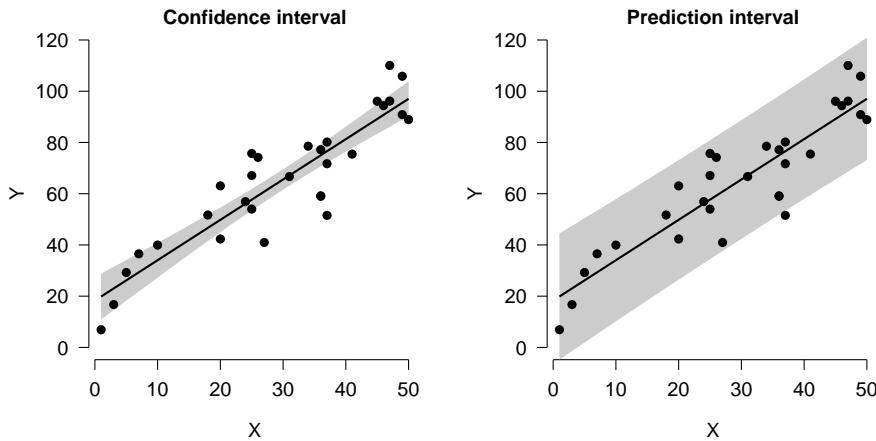
y.up2 <- pred2$fit[, "upr"]

par(mfrow=c(1,2), mar=c(5.1, 5.1, 1.1, 1.1), bty="n",
    lnd=1, las=1, cex.axis=1.2, cex.lab=1.2)

# plot 1: CI
plot(dat$x, dat$y, type="n", ylim=c(0, 120),
      xlab="X", ylab="Y",
      main="Confidence interval")
polygon(x=c(px, rev(px)),
        y=c(y.lo, rev(y.up)), 
        border=NA, col="grey80")
points(px, y.mn, type="l", lwd=2)
points(dat$x, dat$y, pch=16, cex=1.2)

# plot 2: PI
plot(dat$x, dat$y, type="n", ylim=c(0, 120),
      xlab="X", ylab="Y",
      main="Prediction interval")
polygon(x=c(px, rev(px)),
        y=c(y.lo2, rev(y.up2)), 
        border=NA, col="grey80")
points(px, y.mn, type="l", lwd=2)
points(dat$x, dat$y, pch=16, cex=1.2)

```



5.2 GLM basics

In the previous section we explored linear regression and other cases of the the linear model. Linear models are appropriate when there is a linear relationship

between a response variable and its predictors, when the predictor variables model the response variable on its original scale, and when the residuals of the model follow a normal distribution. However, the latter two conditions do not always apply in biology.

The **generalized linear model (GLM)** is a framework that generalizes the linear model by relaxing two of the conditions in which linear models are used. In a GLM:

- The response variable can be modeled on a different scale than its original scale. The original scale of the data is related to the scale of the linear prediction by a link function.
- The values of the response variable can come from many different distributions, not just the normal distribution.

The GLM was introduced by Nelder and Wedderburn (1972) and has since become a vital tool in many fields of study. Methods for fitting GLMs have been improved and expanded since the original publication, such that GLMs can be fit under many inference paradigms including least-squares, Bayesian, and machine learning frameworks.

The basic form of the GLM is shown below:

$$\begin{array}{ccc}
 \text{Expected value} & \text{Inverse link} & \text{Linear} \\
 \text{of } Y \text{ given } X & \text{function} & \text{predictor of } Y \\
 \downarrow & \downarrow & \nearrow \\
 E(Y|X) = \mu = g^{-1}(X\beta) & & \\
 \uparrow & & \\
 \text{Variance of } Y & \text{Design matrix} & \text{Coefficient matrix} \\
 \text{given } X & & \\
 \downarrow & \nearrow & \nearrow \\
 \text{Var}(Y|X) = \text{Var}(g^{-1}(X\beta)) & &
 \end{array}$$

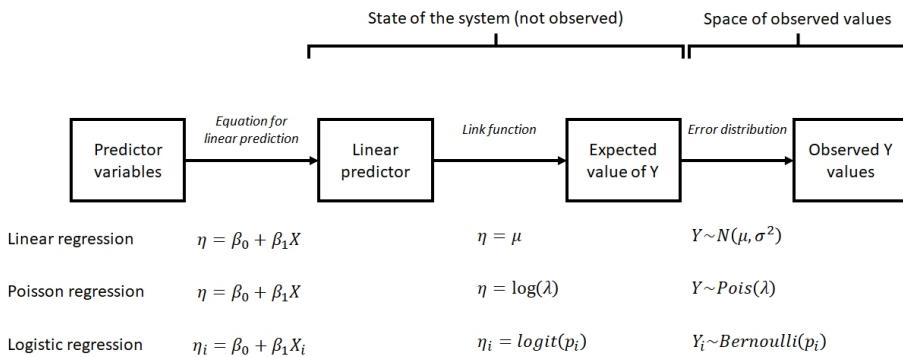
This form is a little abstract, but it shows the three basic parts of a GLM:

- Linear predictor $\mathbf{X}\beta$, which relates the expected value of \mathbf{Y} to a linear function of the predictors \mathbf{X} with parameters β .
- Link function g , which relates the original values of \mathbf{Y} to the values of the linear predictor.
- Probability distribution $\text{Var}(Y|X)$, which describes variability about the expected value.

Below are some more concrete examples of GLMs that show the relationships between the parts. Notice how each of these examples describes a system in terms

of its true, unobserved “state” and relates that state to the space of observed observations. The former part is sometimes also called the “deterministic” part of the model, while the latter is called the “stochastic” part. This framework is referred to as the “state-space” representation of a model. Thinking about your study system in state-space terms is the key to understanding GLM. This framework is useful because it allows us to think about and treat separately the deterministic part (state) and the stochastic part (space) of a statistical model.

Being able to think about and deconstruct a dataset in the language of GLMs is one the main points of this course. The GLM is a very useful tool to have in your toolkit as a biologist.



5.2.1 Example GLMS

5.2.1.1 Linear regression as a GLM

$$Y \sim Normal(\mu, \sigma^2)$$

$$\mu = \eta$$

$$\eta = \beta_0 + \beta_1 X$$

These equations show that linear regression can be thought of as a special case of GLM. The linear predictor η (“eta”) is a linear function of the predictor variable (or variables). The expected value of Y , μ , is equal to η . The variability in the response, σ^2 , is constant and does not depend on the μ (this is not the case for every GLM). Linear regression is a special case of GLM: the case with an identity link function ($\mu = \eta$) and a normal distribution for response variables.

5.2.1.2 GLMs for count data

$$Y \sim Poisson(\lambda)$$

$$\lambda = e^\eta$$

$$\eta = \beta_0 + \beta_1 X$$

The model above is a GLM used to model count data. Counts are often modeled as following a Poisson distribution. The Poisson distribution has one parameter, λ (“lambda”), which is both the expected count and the variance of counts. This means that the residual variation does depend somewhat on the expected value (unlike linear regression). Because counts must be non-negative, the **log link function** is used. In other words, $\log \lambda = \eta$. GLMs with well-known link functions are usually written without the intermediate η variable:

$$\begin{aligned} Y &\sim \text{Poisson}(\lambda) \\ \log(\lambda) &= \beta_0 + \beta_1 X \end{aligned}$$

5.2.1.3 Logistic regression: GLM for binary data

$$\begin{aligned} Y_i &\sim \text{Bernoulli}(p_i) \\ p_i &= \frac{e^{\eta_i}}{1 + e^{\eta_i}} \\ \eta_i &= \beta_0 + \beta_1 X_i \end{aligned}$$

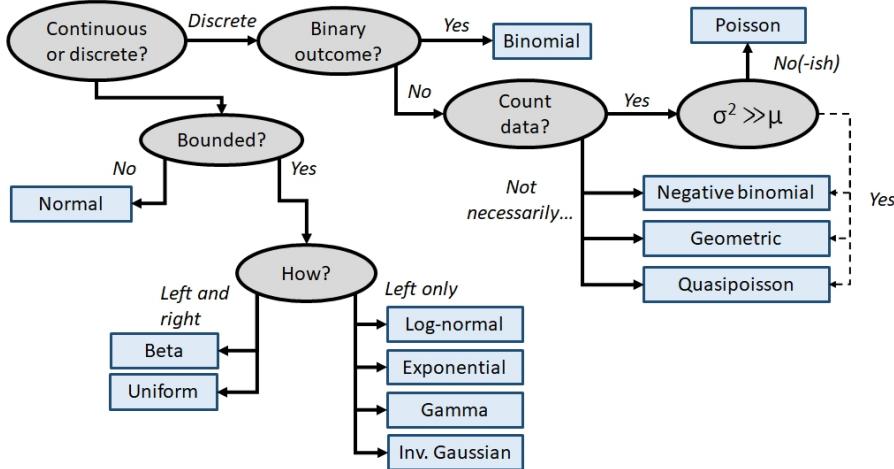
The **logistic regression** model is really a GLM with a logit link function and a binomial error distribution. The Y value for each observation i is either 0 or 1, with $P(1) = p_i$. The link function is the logit function, which means that the linear part of the model predicts the logit of p_i . The model can also be written as:

$$\begin{aligned} Y_i &\sim \text{Bernoulli}(p_i) \\ \text{logit}(p_i) &= \beta_0 + \beta_1 X_i \end{aligned}$$

5.2.2 GLM families

The **family** of a GLM refers to the distribution that observations follow. GLMs use distributions from the **exponential family** of distributions. The exponential family is a class of distributions whose PDF or PMF can be written in a particular form that includes the exponential function e^x or $\exp(x)$. The exponential family includes many well-known distributions such as the normal, exponential⁴, gamma, beta, Poisson, and many others. Two other common distributions, the binomial and negative binomial, can be included in the exponential family under certain conditions. Which family to use in your GLM analysis depends on what kind of data you are trying to model. It turns out that certain kinds of data tend to follow certain distributions. For example, counts often follow a Poisson distribution. The figure below shows the typical use cases of some common distributions:

⁴The *exponential distribution* is included in the *exponential family* of distributions, which includes many other distributions. Unfortunately the word “family” in a GLM context refers to individual distributions. So, fitting a GLM with an exponential family means using the exponential distribution for residuals.



In addition to *a priori* ideas about the nature of a response variable, we also need to consider how the data are distributed in reality. The most common reason that data do not conform to their expected distribution is probably **overdispersion**. This means that the data have a greater variance than expected. How to deal with overdispersion depends on the expected response distribution.

For normally distributed data, overdispersion is usually not a problem. Note that the term for residual variation σ^2 is independent of the expected value μ . This means that a dataset with a large variance isn't necessarily overdispersed. However, large skewness (another kind of apparent overdispersion) in the response variable may indicate that a log-normal or gamma distribution is more appropriate than a normal distribution.

For Poisson-distributed data (counts), overdispersion can be an issue. This is because part of the definition of the Poisson distribution is that the mean and variance are the same parameter (i.e., $\lambda = \mu = \sigma^2$). In real datasets, the variance is often larger than the mean... sometimes much larger! If this is the case, then it might make sense to fit a GLM with a quasi-Poisson or a negative binomial family instead of a Poisson family. Both of these options include an extra parameter to account for overdispersion.

Another form of overdispersion in count data that makes them not conform to the Poisson distribution is **zero-inflation**. This is a situation where a response variable has many more 0 values than would be expected based on the Poisson distribution. Mild to moderate zero inflation can be modeled by the negative binomial. Severe zero-inflation might require using zero-inflated models, an extension of GLM⁵.

⁵One of these days I might add a section on zero-inflated models.

5.2.3 GLM link functions

Unlike the family component, choosing a link function for your GLM is relatively straightforward. Most families have **canonical** link functions that should be used unless you have a very good reason not to. What makes these link functions “canonical” is that they are derived from the PDF or PMF of the response distributions in such a way as to relate an expected value or central tendency of the distribution to a linear function of the predictor variables. In addition to relating different parts of a GLM, link functions are kind of like transformations⁶. This means that using a link function can help put your data on a scale that is more amenable to analysis. However, there is a key difference between using a link function and using a transformation: with a link function, variance is estimated on the response scale, while with a transformation, variance is estimated on the transformed scale. This can have huge consequences depending on the data and the link/transformation function.

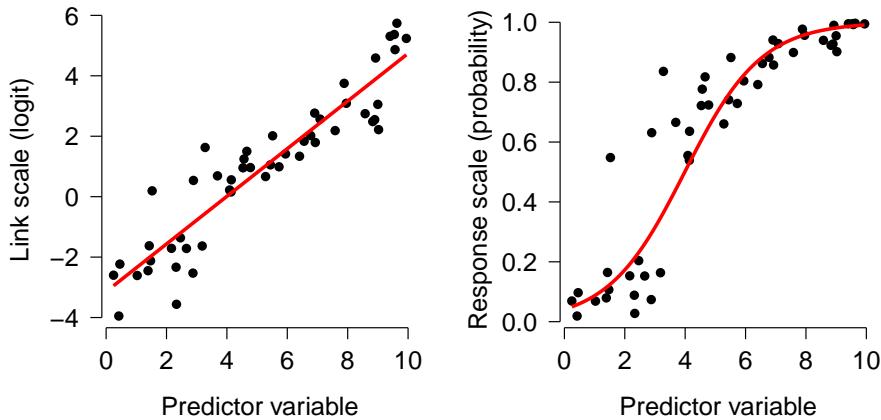
The table below shows the link functions and inverse link functions of some common distributions. The link functions are presented as the relationship between the linear predictor $\mathbf{X}\beta$, the expected value on the link scale η , and the expected value on the response scale μ .

Distribution	Link name	Link function	Inverse link
Normal	Identity	$X\beta = \mu$	$\mu = X\beta$
Poisson	Log	$X\beta = \log(\mu)$	$\mu = e^{X\beta}$
Binomial	Logit	$X\beta = \log\left(\frac{\mu}{1-\mu}\right)$	$\mu = \frac{e^{X\beta}}{1+e^{X\beta}} = \frac{1}{1+e^{-X\beta}}$
Exponential and gamma	Inverse ⁷	$X\beta = \mu^{-1} = \frac{1}{\mu}$	$\mu = (X\beta)^{-1} = \frac{1}{X\beta}$

It is important to remember that GLMs are linear on the link scale, not necessarily on the scale in which the data are recorded. When raw response variables are plotted against predictor variables, the plot may not be linear. The plot below shows how a relationship can appear linear on the link (i.e., transformed) scale (left), but nonlinear on the original scale (right).

⁶Emphasis on the “kind of”. There are some important ways that link functions are *not* like transformations, as we shall see later.

⁷Some sources say negative inverse, but R and most textbooks say to use (positive) inverse.

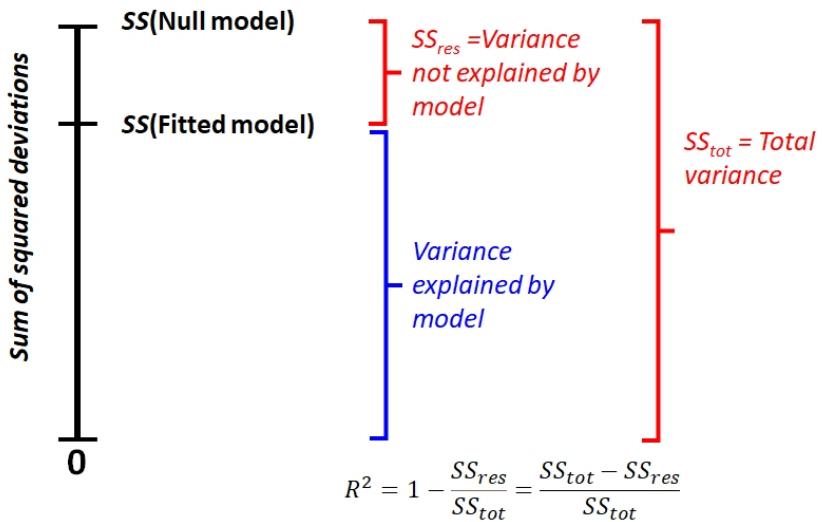


5.2.4 Deviance and other GLM diagnostics

One aspect of GLMs that confuses some people is how to measure how well the model fits the data. That is, how to calculate an R^2 value. In linear regression (and other linear models such as ANOVA), the coefficient of determination R^2 expresses the proportion of variation in the response variable that is attributable to variation in the predictor variable(s). R^2 is a widely-known and useful metric of model fit. However, R^2 is not defined for GLMs. This is because of the way R^2 is defined.

$$R^2 = 1 - \frac{SS_{res}}{SS_{tot}} = 1 - \frac{\sum_{i=1}^n (y_i - \mu_i)^2}{\sum_{i=1}^n (y_i - \bar{y})^2}$$

This can be visualized in the figure below. Notice how the minimum possible sum of squared deviations is 0, and R^2 is the proportion of how much variation (in terms of sums of squares) is reduced by the model. In other words, the performance of the model falls somewhere between the worst possible value and the best possible value. A key property of this calculation is that both the worst possible value, SS_{null} , and the best possible value, 0, are easily defined.



This definition assumes a normal distribution of residuals, because the variance of a normal distribution X is

$$Var(x) = \frac{\sum (x_i - \bar{x})^2}{n - 1}$$

However, the variance for other distributions is something else. This means that the denominator in R^2 (total sum of squared deviations) is often not an appropriate measure of variation. Instead of using R^2 , we have to use a more generalized way to measure model fit.

Deviance is a quantity that generalizes the concept of sum of squared residuals (SS_{res}) to models that were fit using maximum likelihood estimation (MLE) instead of classical least squares estimation. For linear regression and other models fit using least squares estimation (which is basically just solving some linear algebra problems), calculating SS_{res} is straightforward. This is because of the overlap between the definitions of SS_{res} and the variance of the normal distribution. However, models fit by MLE work by searching for combinations of values of model terms (the “parameter space”) for combinations that maximize a “likelihood function”. Because residuals are not used in the parameter estimation process, evaluating model fit based on SS_{res} doesn’t make sense.

Just like least squares methods partition total sums of squared errors into “total” and “residual” sums of squared errors (SS_{tot} and SS_{res} in the figure above), other methods partition total deviance into “residual deviance” (D_{res}) and the deviance explained by the model. However, calculating total deviance is not as straightforward as calculating SS_{tot} because there is no natural minimum against which to compare. Recall that in linear models, the minimum sum of

squared residuals was 0. In GLM terms, this creates a natural upper bound to the likelihood function. However, for most GLMs the theoretical upper bound cannot be calculated *a priori* the way it can with linear regression.

Deviance is defined by comparing the likelihood of the data under the fitted model to the likelihood of the data under the **saturated model**⁸. A saturated model is one that incorporates all of the information in the dataset by having as many parameters as it does observations⁹. This model is guaranteed to have the closest possible fit to the data. This means that the saturated model will have the greatest possible likelihood function. Any other model will have a smaller likelihood function. The difference between the likelihood of the saturated model and the likelihood of another model tells us something about how much explanatory power that the fitted model has, relative to a hypothetical model with perfect explanatory power.

Residual deviance (D_{res}) is calculated as twice the difference between the likelihood of the saturated model (L_{sat}) and the likelihood of the fitted model (L_{fitted}). For convenience, we usually work with the log of the likelihood (LL).

$$D_{resid} = 2(LL_{sat} - LL_{fitted})$$

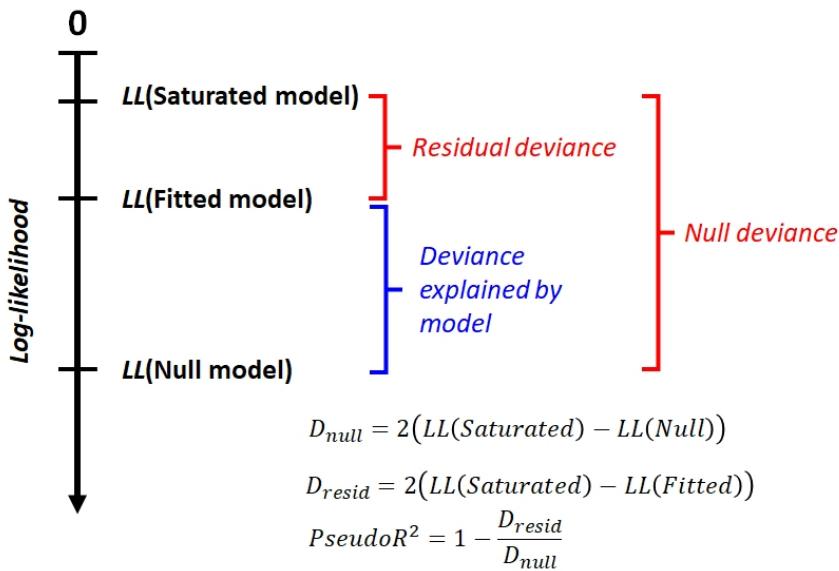
Null deviance (D_{null}) is similar, but compares the saturated model to the null model. The null model is a model with no predictor variables, which predicts the same value (usually the mean) for all observations.

$$D_{null} = 2(LL_{sat} - LL_{null})$$

The relationship between these deviances can be seen in the figure below. The null deviance is the difference in log-likelihood between the model that contains none of the information in the data (the null model) and the model that contains all of it (the saturated model). This makes the null deviance the baseline for the best that a model could possibly be. A well-fitting model should have very little residual deviance, because it performs similarly to the saturated model while having fewer parameters. Even though the log-likelihood scale includes 0, this is only to show that log-likelihood values are almost always negative.

⁸I highly recommend this video for an introduction to deviance, which also has a nice theme song

⁹Obviously, this is an overfit model, but that's ok because we are using it as a hypothetical "baseline".



Compare this figure to the one above that illustrates R^2 . Unlike the log-likelihood, there is a natural minimum of 0 for sums of squares in the best possible model. This is analogous to the log-likelihood of the saturated model. Without considering the saturated model, we can't calculate null deviance, and thus can't calculate how much of that deviance is explained by the fitted model.

Why does the formula for deviance includes a factor of 2? That factor is there to make it so that the deviance follows a chi-squared distribution with a number of degrees of freedom equal to the difference in the number of parameters between the models. We'll explore what that means later when we work through some GLM examples.

5.2.5 To pseudo- R^2 or not to pseudo- R^2 ?

One final note about goodness of fit in GLMs: no single measure of goodness of fit is universally agreed upon or appropriate for all situations. The pseudo- R^2 presented in subsequent sections is widely used, but some authors disagree about what it represents or measures. Other methods for evaluating GLM goodness of fit include:

- **Cross-validation:** measures the ability of the model to accurately predict data that were not used fit the model. This is widely used in by machine learning methods, and can be useful when predictive ability is the desired endpoint.
- **Receiver operating characteristic (ROC) and area under curve (AUC):** measures the predictive accuracy of models used for classification,

especially logistic regression. For logistic regression, AUC can be more informative than pseudo- R^2 .

5.2.6 Common GLMs

This site has worked examples of several GLMs commonly encountered by biologists. The links below will take you to the GLM flavor that you're interested in. The options are:

Model	Typical use case	Link
Log-linear GLM	Models for continuous values with non-normal errors	Click here!
Poisson GLM	Models for count data	Click here!
Quasi-Poisson and negative binomial GLM	Models for overdispersed count data	Click here!
Logistic regression	Models for binary outcomes	Click here!
Binomial GLM	Models for proportional data	Click here!
Gamma GLM	Models for heteroscedastic and right-skewed data	Click here!

5.3 Log-linear models

This is one of several sections exploring some common GLM applications. For most of these applications we will work through two examples. First, an analysis of simulated data, and second, an analysis of a real dataset. Simulating a dataset suitable for an analysis is an excellent way to learn about the analysis method, because it helps you see the relationship between the data-generating process and the analysis.

This module explores GLMs with a log link function and Gaussian (normal) family: a log-linear model. Log-linear models are models with a linear relationship between the logarithm of the response variable and the predictor variables. While “log-linear” often describes GLMs with Poisson families, it can mean any GLM with a log link function.

5.3.1 Example with simulated data

This example was designed to illustrate the difference between a log-linear GLM and a linear model on log-transformed data.

```
# random number seed for reproducibility
set.seed(42)

# sample size
n <- 50

# coefficients and residual SD
beta0 <- 2.5
beta1 <- 0.25
sigma <- 2

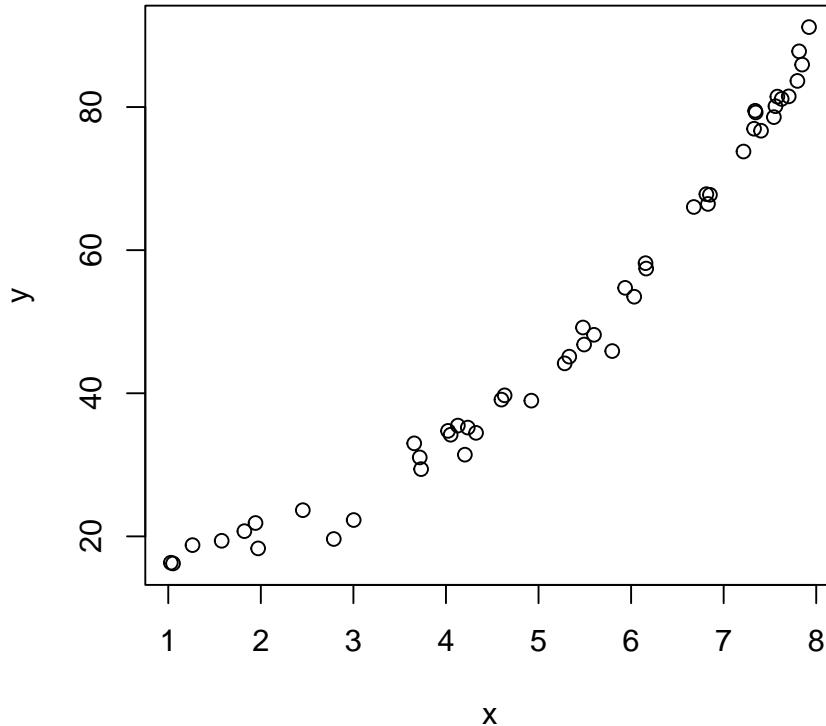
# x values
x <- runif(n, 1, 8)

# linear predictor
eta <- beta0 + beta1 * x

# inverse link function
mu <- exp(eta)

# response variable
y <- rnorm(n, mu, sigma)
dat <- data.frame(x=x, y=y)

# plot the data:
plot(x,y)
```



The relationship looks somewhat linear, but we might have some reason to suspect that it is not either because we have a biological reason or because we can see the slight curve characteristic of exponential functions. Let's try two models: a log-linear GLM, and a LM on log-transformed data.

```
# model 1: GLM with log link
mod1 <- glm(y~x, data=dat, family=gaussian(link = "log"))
# model 2: LM on log-transformed data
dat$logy <- log(dat$y)
mod2 <- lm(logy~x, data=dat)
```

You may get an error message that illustrates a problem with fitting GLMs that doesn't come up with linear models: starting values. Recall that linear models are fit by essentially solving linear algebra problems. GLMs are fit using an iterative process. This iterative process needs reasonable starting values in order to have a good chance of finding a good solution. R can figure out good starting values for many GLMs, particularly GLMs that use the canonical link function

for the family (see table in previous section). Because the canonical link for the Gaussian family is the identity function, not log, R may need you to provide the starting values. One way to find good starting values is to find a way to linearize the model. In this case, we can simply log-transform the response and use the coefficients as starting values for our GLM.

```
# not run:
starts <- coef(lm(log(y)~x))
mod1 <- glm(y~x, data=dat, family=gaussian(link="log"),
            start=starts)
```

Interestingly, both models estimated model parameters that were pretty close to the true values.

```
summary(mod1)

##
## Call:
## glm(formula = y ~ x, family = gaussian(link = "log"), data = dat,
##      start = starts)
##
## Deviance Residuals:
##      Min        1Q     Median        3Q       Max
## -5.7735   -1.1230    0.4361    1.4853   3.1162
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 2.478735  0.025559  96.98  <2e-16 ***
## x          0.252960  0.003716  68.07  <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for gaussian family taken to be 3.806117)
##
## Null deviance: 27488.14  on 49  degrees of freedom
## Residual deviance: 182.69  on 48  degrees of freedom
## AIC: 212.68
##
## Number of Fisher Scoring iterations: 3
summary(mod2)

##
## Call:
## lm(formula = logy ~ x, data = dat)
##
## Residuals:
##      Min        1Q     Median        3Q       Max
##
```

```

## -0.22312 -0.01971  0.01146  0.02941  0.11028
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 2.510009  0.022596 111.08 <2e-16 ***
## x           0.247561  0.004033  61.38 <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.06001 on 48 degrees of freedom
## Multiple R-squared:  0.9874, Adjusted R-squared:  0.9872
## F-statistic:  3767 on 1 and 48 DF,  p-value: < 2.2e-16

```

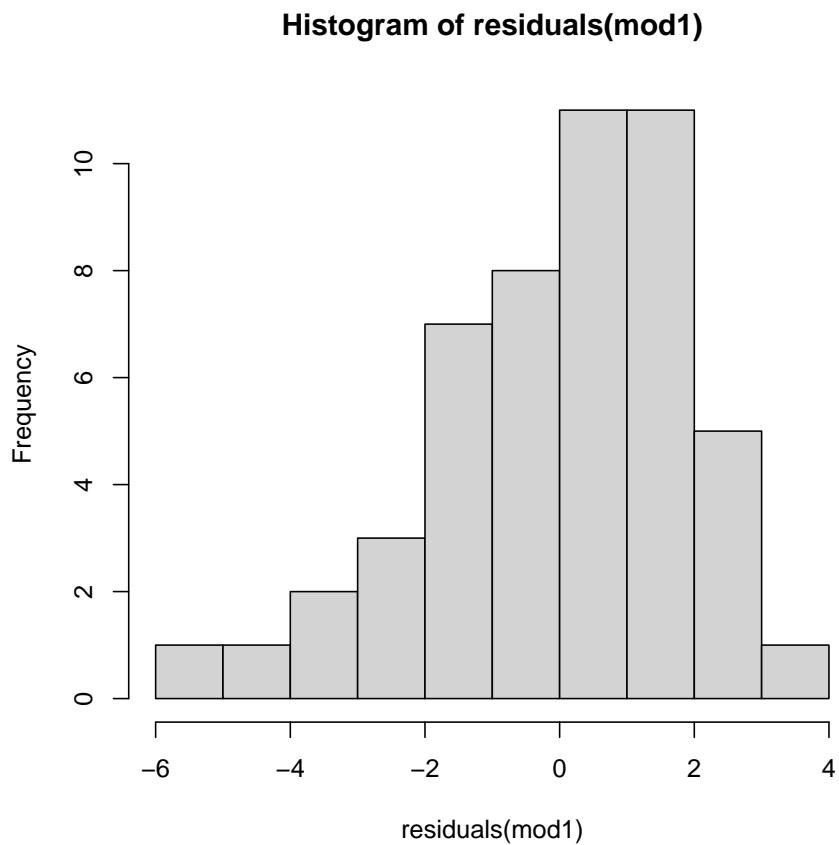
The R^2 for the LM fit is very high, 0.987. We can use the deviance of the fitted model to calculate a pseudo- R^2 for the GLM fit. The result, 0.933, is also pretty good!

```
1-(mod1$deviance/mod1>null.deviance)
```

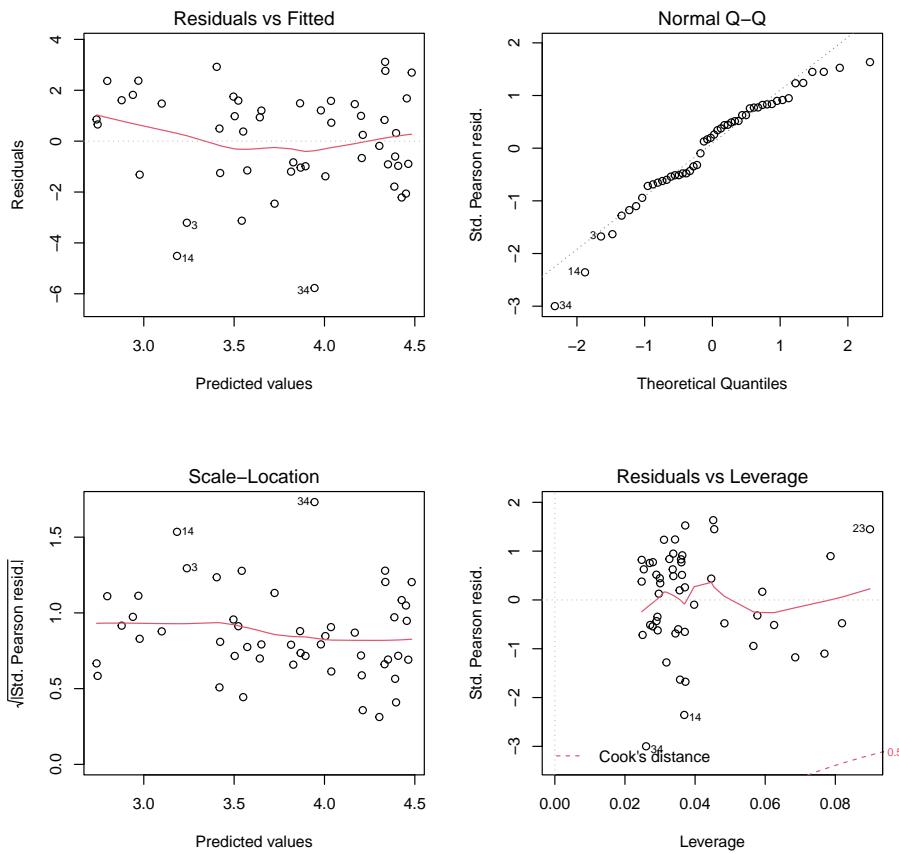
```
## [1] 0.9933537
```

As a final check, let's look at the residuals and other diagnostic plots for our models.

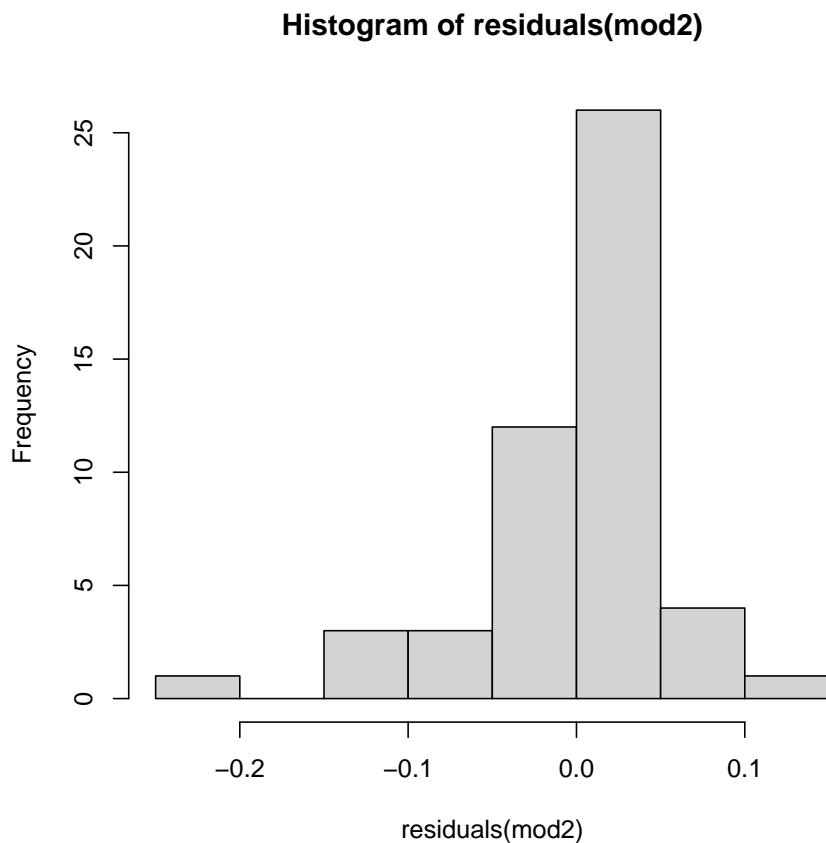
```
hist(residuals(mod1))
```



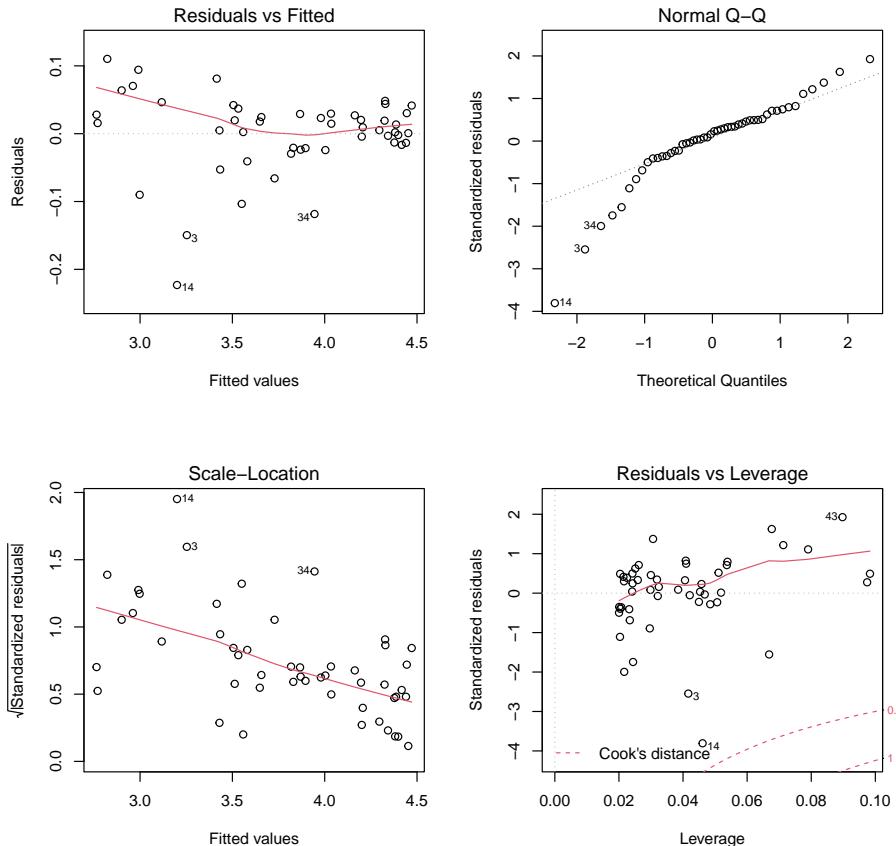
```
par(mfrow=c(2,2))
plot(mod1)
```



```
hist(residuals(mod2))
```



```
par(mfrow=c(2,2))
plot(mod2)
```



The plots for the GLM fit look pretty good. There is no evidence of heteroscedasticity, and no evidence of nonnormality in the residuals. The same plots for the LM fit, however, do show some potential problems with heteroscedasticity. In particular, the scale-location plot shows evidence that the size of the residuals varies with the fitted value.

The last step is to plot the data and the predicted values. When plotting predicted values (and their 95% confidence interval) from GLMs, the procedure is similar to that for linear models. However, we need to be careful not to treat variation on the response scale the same as variation on the link scale. R can generate predictions either scale, but the uncertainty it calculates for those predictions may or may not be reasonable for your situation. For this example, both methods below will produce the same predictions; with other families and link functions, that might not be the case.

Notice that the code below looks very similar to that used with the linear model output.

```

# new data for prediction
px <- seq(min(x), max(x), length=50)

# GLM prediction (mod1)
pred1 <- predict(mod1, newdata=data.frame(x=px),
                  type="link", se.fit=TRUE)
lo1 <- qnorm(0.025, pred1$fit, pred1$se.fit)
up1 <- qnorm(0.975, pred1$fit, pred1$se.fit)
mn1 <- pred1$fit

# inverse link function
lo1 <- mod1$family$linkinv(lo1)
up1 <- mod1$family$linkinv(up1)
mn1 <- mod1$family$linkinv(mn1)

# LM prediction (mod2)
pred2 <- predict(mod2, newdata=data.frame(x=px),
                  se.fit=TRUE)
lo2 <- qlnorm(0.025, pred2$fit, pred2$se.fit)
up2 <- qlnorm(0.975, pred2$fit, pred2$se.fit)

# backtransform
mn2 <- exp(pred2$fit)

```

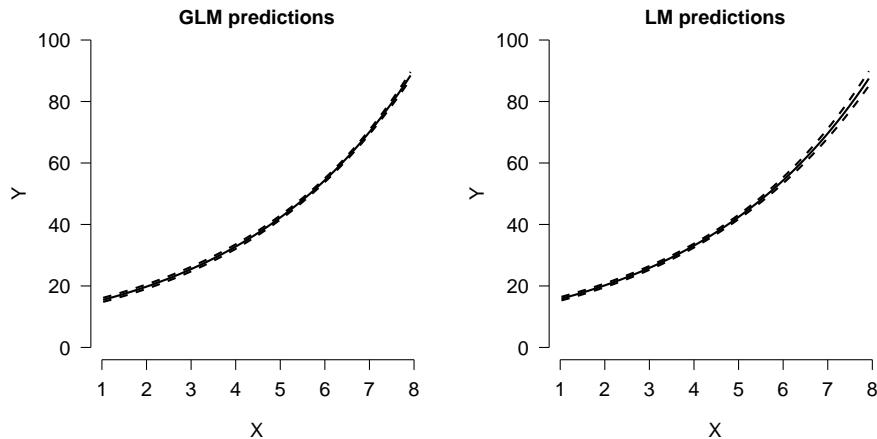
The code below plots the predicted values, 95% CI, and optionally the original data. The commands to plot the original data are commented out to make it easier to see the confidence intervals (which are very narrow).

```

par(mfrow=c(1,2), mar=c(5.1, 5.1, 1.1, 1.1),
    bty="n", lend=1,
    las=1, cex.axis=1.2, cex.lab=1.2)
plot(dat$x, dat$y, type="n", ylim=c(0, 100),
     xlab="X", ylab="Y",
     main="GLM predictions")
points(px, lo1, type="l", lwd=2, lty=2)
points(px, up1, type="l", lwd=2, lty=2)
points(px, mn1, type="l", lwd=2)
#points(dat$x, dat$y, pch=16, cex=0.8)

plot(dat$x, dat$y, type="n", ylim=c(0, 100),
     xlab="X", ylab="Y",
     main="LM predictions")
points(px, lo2, type="l", lwd=2, lty=2)
points(px, up2, type="l", lwd=2, lty=2)
points(px, mn2, type="l", lwd=2, pch=16, cex=0.8)

```



```
#points(dat$x, dat$y, pch=16, cex=0.8)
```

The figure shows that both models predicted the mean quite well. Notice what happens at the far right of each plot: the 95% CI for the LM predictions starts to get wider, while the interval for the GLM predictions does not. In other words, the LM fit is heteroscedastic, while the GLM fit was not. Were the original data heteroscedastic? No. The GLM accurately captured the homoscedastic nature of the data.

5.3.2 Example with real data

Chivers and Hladik (1980) investigated the morphology of the gut in primates and other mammals to see if there was a correlation with diet. They examined digestive tracts from 78 mammal species (50 of them primates) and measured the length, surface area, volume, and weight of different components of each species' gut. Each species was classified initially into one of three diet classes: - Faunivore: consumes primarily animals (i.e., a carnivore) - Frugivore: consumes primarily fruits, but could also eat a varied diet - Foliovore: consumes primarily leaves and grasses

The variables in this dataset are:

Variable	Meaning
spp	Taxon as genus (if only one member of genus present in dataset) or binomen (if >1 member of genus in dataset)
order	Taxonomic order
diet	Frugivore, foliovore, or faunivore
sex	Sex: f = female, m = male
len	Body length (cm)

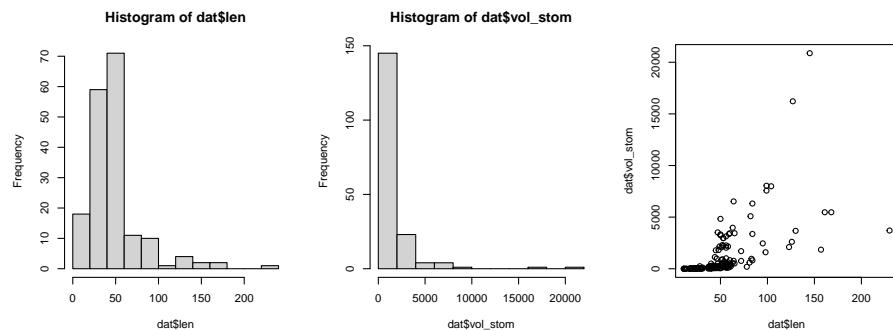
Variable	Meaning
sa_stom	Internal surface area of the stomach (cm^2)
sa_si	Internal surface area of the small intestine (cm^2)
sa_caec	Internal surface area of the cecum (cm^2)
sa_col	Internal surface area of the colon (cm^2)
vol_stom	Volume of the stomach (cm^3)
vol_si	Volume of the small intestine (cm^3)
vol_caec	Volume of the cecum (cm^3)
vol_col	Volume of the colon (cm^3)

Import the dataset chivers1980data.csv. The code below requires that you put the file in your R working directory.

```
in.name <- "chivers1980data.csv"
dat <- read.csv(in.name, header=TRUE)
```

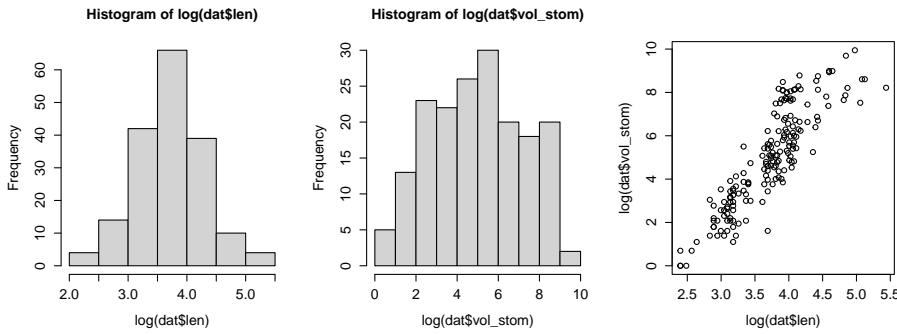
For this example, we will analyze the relationship between stomach size and body length. Examine both variables and their relationship to see what we're dealing with:

```
par(mfrow=c(1,3))
hist(dat$len)
hist(dat$vol_stom)
plot(dat$len, dat$vol_stom)
```



The histograms for both variables suggest that they are highly nonnormal and right-skewed. They are also continuous variables. This suggests that log-transforming them would be helpful.

```
par(mfrow=c(1,3), cex.lab=1.3, cex.axis=1.3)
hist(log(dat$len))
hist(log(dat$vol_stom))
plot(log(dat$len), log(dat$vol_stom))
```



Log-transforming the variables made it easier to see the relationship between body length and stomach volume. On a log-log scale, the relationship looks linear. At this point we could simply perform linear regression on the log-transformed variables. Instead, we will fit a “log-linear” model using GLM.

Remember that in a GLM the link function is what “transforms” the response variable. So, we only need to transform the predictor variable. We’ll use \log_{10} instead of \log so that plotting will be easier later.

```
dat$len.tr <- log10(dat$len)
```

Next, fit the model using function `glm()`. We are using the Gaussian (aka: normal) family with a log link function. Notice that the formula and data arguments are the same as for `lm()`.

```
mod1 <- glm(vol_stom~len.tr, data=dat,
             family=gaussian(link="log"))
summary(mod1)

##
## Call:
## glm(formula = vol_stom ~ len.tr, family = gaussian(link = "log"),
##      data = dat)
##
## Deviance Residuals:
##      Min        1Q    Median        3Q       Max
## -6813.5   -890.5   -528.0   -288.2   15418.2
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
```

```

## (Intercept) 1.5356    0.6813   2.254   0.0254 *
## len.tr      3.2709    0.3307   9.892   <2e-16 ***
## ---
## Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for gaussian family taken to be 3992734)
##
## Null deviance: 1092527506 on 178 degrees of freedom
## Residual deviance: 706707651 on 177 degrees of freedom
## AIC: 3232.8
##
## Number of Fisher Scoring iterations: 6

```

The coefficients part of the output looks similar to that of a linear model. The bottom part displays information about deviance rather than calculations related to R^2 . We can use those values to calculate a pseudo- R^2 .

```
1-(mod1$deviance/mod1>null.deviance)
```

```
## [1] 0.3531443
```

That's not great but not bad either. Let's plot the data and model predictions to see what might have gone wrong. We'll use `predict()` like we do with linear models, but with a few added wrinkles.

```

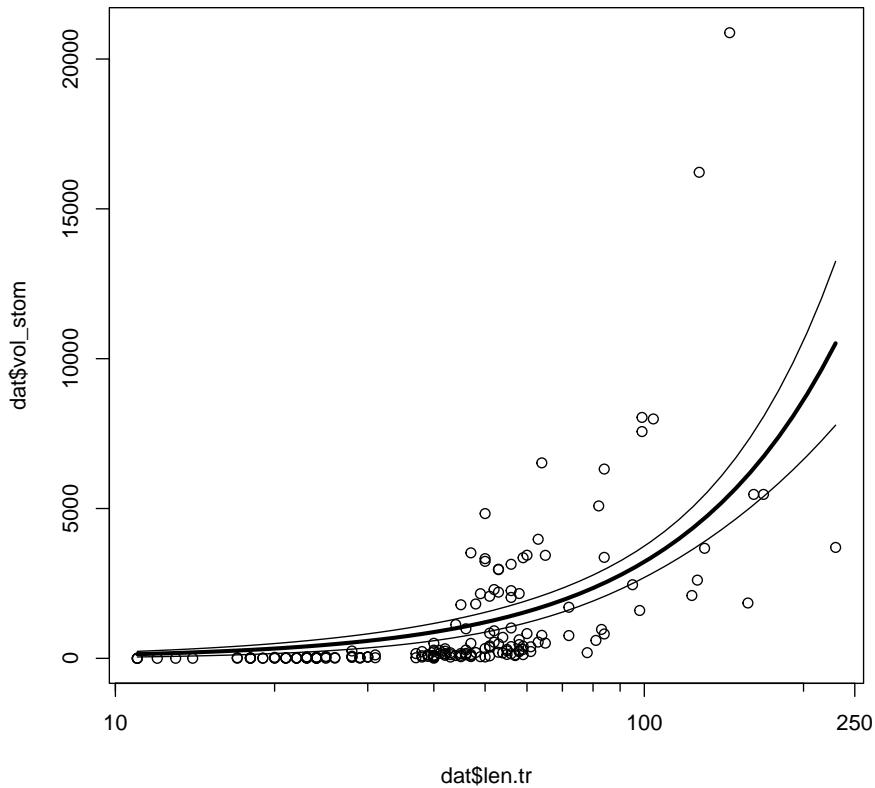
# new data for prediction
px <- seq(min(dat$len.tr), max(dat$len.tr), length=50)

# make predictions
pred <- predict(mod1, newdata=data.frame(len.tr=px),
                 se.fit=TRUE,
                 type="response")
mn <- pred$fit
lo <- qnorm(0.025, mn, pred$se.fit)
up <- qnorm(0.975, mn, pred$se.fit)

# reset plot layout
par(mfrow=c(1,1))

# make the plot
plot(dat$len.tr, dat$vol_stom, xaxt="n")
axis(side=1, at=log10(c(10, 100, 250)), labels=c(10, 100, 250))
axis(side=1, at=log10(c(1:9*10, 200)), labels=NA, tcl=-0.3)
points(px, lo, type="l")
points(px, up, type="l")
points(px, mn, type="l", lwd=3)

```



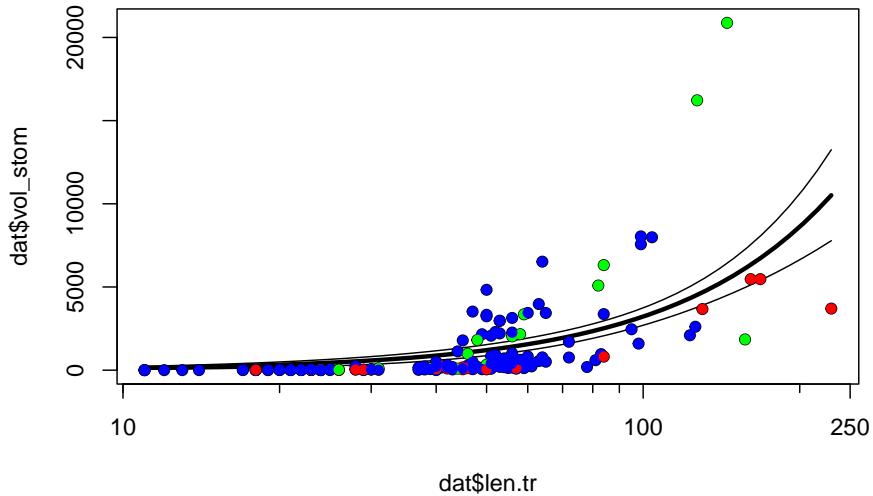
The model does reasonably well below 100 cm length, but there is quite a bit of spread as animals get larger. Let's superimpose the factor `diet` on the plot to see if that might help.

```

cols <- rainbow(3)
diets <- sort(unique(dat$diet))
use.cols <- cols[match(dat$diet, diets)]

plot(dat$len.tr, dat$vol_stom, xaxt="n")
axis(side=1, at=log10(c(10, 100, 250)), labels=c(10, 100, 250))
axis(side=1, at=log10(c(1:9*10, 200)), labels=NA, tcl=-0.3)
points(px, lo, type="l")
points(px, up, type="l")
points(px, mn, type="l", lwd=3)
points(dat$len.tr, dat$vol_stom, pch=16, col=use.cols)

```



It looks like some of the variation might be driven by diet. To investigate, we'll refit the model with the factor diet. We will fit two versions of the model with diet: one where the effects of diet and length are separate, and another where the effects of diet and length interact. We'll then use information theoretic inference to determine which model works best.

```
mod2 <- glm(vol_stom~len.tr+diet, data=dat,
             family=gaussian(link="log"))
mod3 <- glm(vol_stom~len.tr*diet, data=dat,
             family=gaussian(link="log"))
```

The pseudo- R^2 increased from model 1 to model 2, but not from model 2 to model 3. This suggests that adding the interaction didn't improve the fit much, but adding the factor diet did.

```
# check pseudo R squared (heuristic only)
1-(mod2$deviance/mod2$null.deviance)
```

```
## [1] 0.5342784
1-(mod3$deviance/mod3$null.deviance)

## [1] 0.5359137
```

We can use Akaike's information criterion (AIC) and AIC weight to compare the models.

```

aic.df <- AIC(mod1, mod2, mod3)
aic.df$delta <- aic.df$AIC - min(aic.df$AIC)
aic.df$wt <- exp(-0.5*aic.df$delta)
aic.df$wt <- aic.df$wt/sum(aic.df$wt)
aic.df <- aic.df[order(-aic.df$wt),]
aic.df

##      df      AIC      delta      wt
## mod2  5 3177.957  0.000000 8.435910e-01
## mod3  7 3181.327  3.370387 1.564090e-01
## mod1  3 3232.765 54.807812 1.058685e-12

```

The AIC weights suggest that model 2 is likely to be the best model *of these 3 models*. So, let's remake the figure with model predictions using the new model 2. This figure will be a little more complicated, because we need to show predictions within each of 3 levels of diet, while also being careful not to make predictions outside of the domain of length within any diet.

```

# number of points
n <- 50

# get range of X for each level of diet
agg <- aggregate(len.tr~diet, data=dat, range)

# define a sequence of X values for each diet
px1 <- seq(agg$len.tr[1,1], agg$len.tr[1,2], length=n)
px2 <- seq(agg$len.tr[2,1], agg$len.tr[2,2], length=n)
px3 <- seq(agg$len.tr[3,1], agg$len.tr[3,2], length=n)

# combine together to single data frame
dx <- data.frame(diet=rep(agg$diet, each=n),
                  len.tr=c(px1, px2, px3))

# calculate predictions and 95% CI
pred <- predict(mod2, newdata=dx,
                 se.fit=TRUE,
                 type="response")

mn <- pred$fit
lo <- qnorm(0.025, mn, pred$se.fit)
up <- qnorm(0.975, mn, pred$se.fit)

# put everything in a single data frame for convenience
dx$mn <- mn
dx$lo <- lo
dx$up <- up

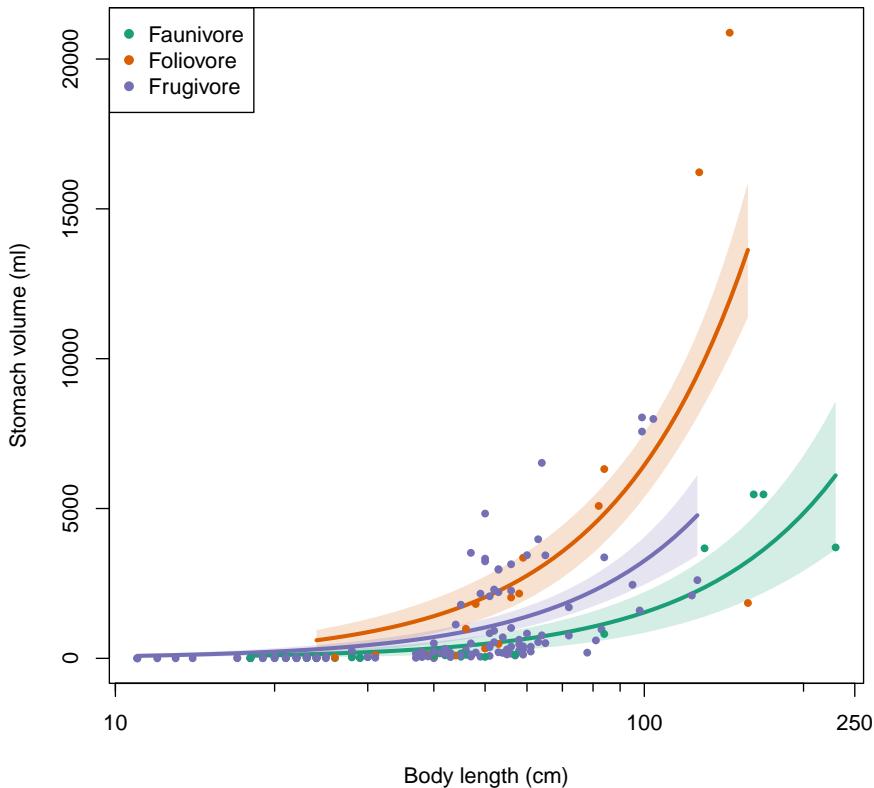
```

```
# unique diets
diets <- sort(unique(dx$diet))
ndiets <- length(diets)

# define a color palette and add to dat for plotting
# https://colorbrewer2.org/#type=qualitative&scheme=Dark2&n=3
cols <- c("#1b9e77", "#d95f02", "#7570b3")
cols2 <- paste0(cols, "30")
dat$col <- cols[match(dat$diet, diets)]
```

Now we're ready to make our plot. As before, we'll assemble the plot piece by piece. The preliminary code above seems like a lot of trouble, but it will allow us to make the plot very efficiently. First we'll make the plot and add the log X-axis. Then, we'll use a `for()` loop to plot the predicted mean and CI for each diet. Using a loop allows us to repeat commands many times, without having to type the commands over and over.

```
plot(dat$len.tr, dat$vol_stom, type="n",
      ylab="Stomach volume (ml)",
      xlab="Body length (cm)",
      xaxt="n")
axis(side=1, at=log10(c(10, 100, 250)), labels=c(10, 100, 250))
axis(side=1, at=log10(c(1:9*10, 200)), labels=NA, tcl=-0.3)
for(i in 1:ndiets){
  flag <- which(dx$diet == diets[i])
  polygon(x=c(dx$len.tr[flag], rev(dx$len.tr[flag])),
           y=c(dx$lo[flag], rev(dx$up[flag])), 
           border=NA, col=cols2[i])
  points(dx$len.tr[flag], dx$mn[flag],
         type="l", lwd=3, col=cols[i])
} #i
points(dat$len.tr, dat$vol_stom, pch=16, col=dat$col, cex=0.8)
legend("topleft", legend=diets, col=cols, pch=16)
```



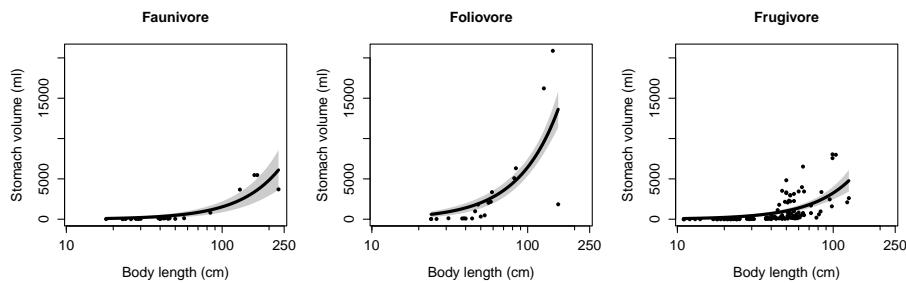
Here is an alternative version with each diet in its own panel:

```
# alternative version in 3 panels:
par(mfrow=c(1,3), cex.lab=1.3, cex.axis=1.3)
for(i in 1:ndiets){
  flag <- which(dx$diet == diets[i])
  plot(dat$len.tr, dat$vol_stom, type="n",
    ylab="Stomach volume (ml)",
    xlab="Body length (cm)",
    xaxt="n")
  axis(side=1, at=log10(c(10, 100, 250)),
    labels=c(10, 100, 250))
  axis(side=1, at=log10(c(1:9*10, 200)),
    labels=NA, tcl=-0.3)
  polygon(x=c(dx$len.tr[flag], rev(dx$len.tr[flag])),
    y=c(dx$lo[flag], rev(dx$up[flag])),
    border=NA, col="grey80")}
```

```

points(dx$len.tr[flag], dx$mn[flag],
      type="l", lwd=3)
title(main=diets[i])
flag2 <- which(dat$diet == diets[i])
points(dat$len.tr[flag2], dat$vol_stom[flag2],
      pch=16, cex=0.8)
}

```



5.4 Poisson GLM for counts

This is one of several sections exploring some common GLM applications. For most of these applications we will work through two examples. First, an analysis of simulated data, and second, an analysis of a real dataset. Simulating a dataset suitable for an analysis is an excellent way to learn about the analysis method, because it helps you see the relationship between the data-generating process and the analysis.

This module explores GLMs for **count** data. Count data are exactly what they sound like: values resulting from counting things. Count data take the form of non-negative integers. Non-negative integers, however, are not necessarily count data. For example, values can be rounded to the nearest integer, but they will not be counts.

Count data are usually well-modeled by a kind of GLM called **Poisson regression**. The “Poisson” part comes from the **Poisson distribution**, which describes counts and events that occur at a constant rate. The “regression” part of the name comes from the fact that the technique involves a linear relationship between the predictors and some function of the response.

In the next section we will explore another class of models for count data that do *not* follow the Poisson distribution. These models, the **quasi-Poisson** and **negative binomial** GLMs, have an extra term that accounts for overdispersion.

Poisson regression describes a GLM with a Poisson family and log link function. As with other GLMs, we do not fit an equation for the response values directly. Instead, we fit a linear model for the log of the expected value, and assume

that the actual observed counts follow a Poisson distribution. To see what this means in practice, let's simulate some data that might be modeled by Poisson regression.

5.4.1 Example with simulated data

In this example Poisson GLM workflow we will simulate some data, analyze it, and interpret the results.

```
# random number seed for reproducibility
set.seed(42)

# sample size
n <- 50

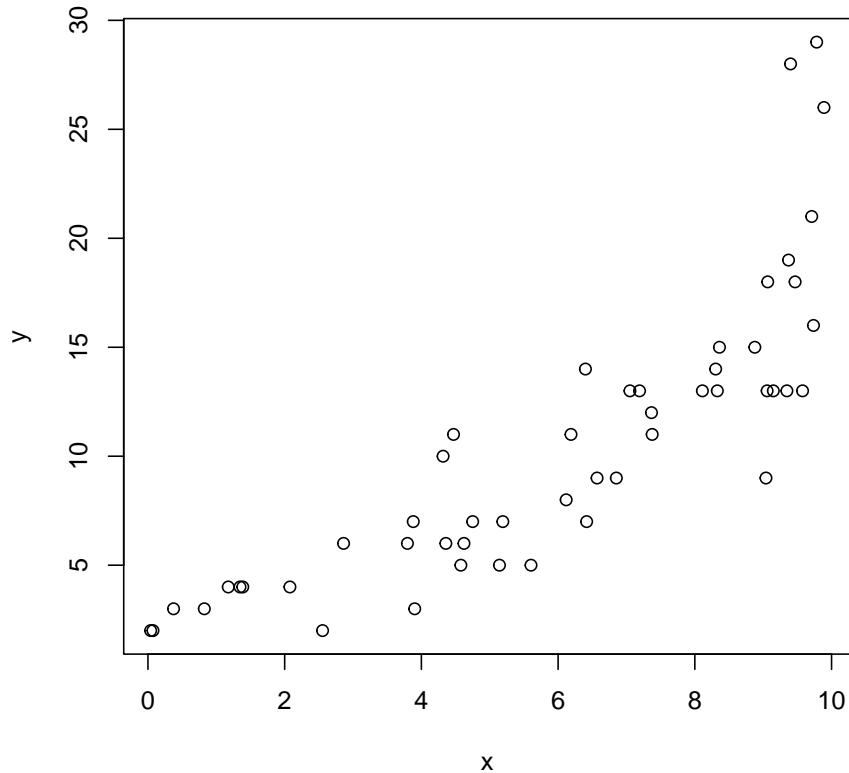
# coefficients
beta0 <- 0.9
beta1 <- 0.2

# x values
x <- runif(n, 0, 10)

# linear predictor
eta <- beta0 + beta1 * x

# inverse link function
lambda <- exp(eta)

# response variable
y <- rpois(n, lambda)
dat <- data.frame(x=x, y=y)
plot(x,y)
```



The relationship between Y and X looks linear, but because we are clever biologists, we know that linear models are not appropriate for two reasons. First, linear models predict continuous responses, and counts are not continuous (they are discrete). Second, linear models can predict negative responses, and counts cannot be negative. So, we will try Poisson regression because the Poisson distribution is so good at handling counts.

```
mod1 <- glm(y~x, data=dat, family=poisson)
summary(mod1)
```

```
##
## Call:
## glm(formula = y ~ x, family = poisson, data = dat)
##
## Deviance Residuals:
##      Min        1Q    Median        3Q       Max
## -2.06284 -0.46194 -0.06797  0.39314  2.18201
```

```

## 
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept)  0.9258     0.1447   6.397 1.59e-10 ***
## x           0.2089     0.0183  11.415 < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
## 
## (Dispersion parameter for poisson family taken to be 1)
## 
## Null deviance: 194.34 on 49 degrees of freedom
## Residual deviance: 37.16 on 48 degrees of freedom
## AIC: 241.37
## 
## Number of Fisher Scoring iterations: 4

```

The estimated parameters are very close to the correct values! The model also explains a lot of the variation in Y , as seen by the pseudo- R^2 of > 0.8 .

```
1-mod1$deviance/mod1>null.deviance
```

```
## [1] 0.8087827
```

The next step is to plot predicted values and the original data. If you have visited any of the other modules in this course, you should be familiar with the process by now. Even if you feel comfortable generating model predictions, there are some wrinkles to be aware of when working with non-normal models. We're going to present predictions in several ways, to illustrate the relationship between the GLM and the data.

First, let's present the 95% prediction interval (PI). The PI is an interval in which 95% of values should fall. It is necessarily larger than the 95% confidence interval (CI), which is the interval where the expected value should fall. Both intervals have their uses; just be clear about which one you are using.

Notice that we do not request `se.fit=TRUE` in the commands below, because it is not needed to get the PI of the Poisson distribution. This is because for the Poisson distribution the expected value, mean, and variance are all the same value (i.e., $\lambda = \mu = \sigma^2$).

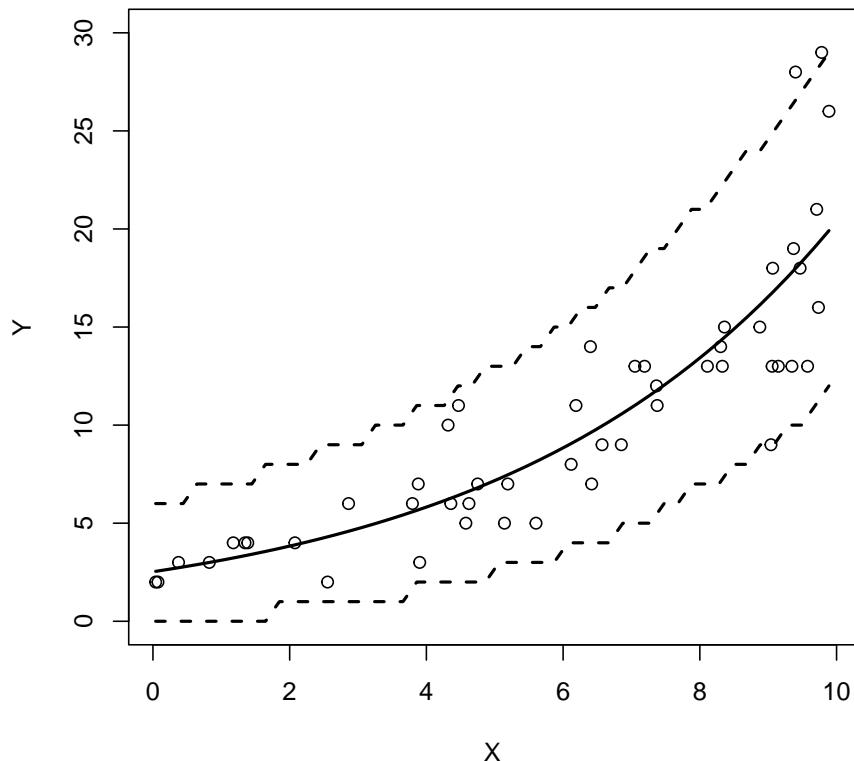
```

n <- 50
px <- seq(min(dat$x), max(dat$x), length=n)
pred1 <- predict(mod1,
                  newdata=data.frame(x=px),
                  type="response")

mn1 <- pred1
lo1 <- qpois(0.025, mn1)
up1 <- qpois(0.975, mn1)

```

```
# make the plot:
plot(x, y, xlab="X", ylab="Y", type="n",
      ylim=c(0, 30))
points(px, lo1, type="l", lty=2, lwd=2)
points(px, up1, type="l", lty=2, lwd=2)
points(px, mn1, type="l", lwd=2)
points(x,y)
```



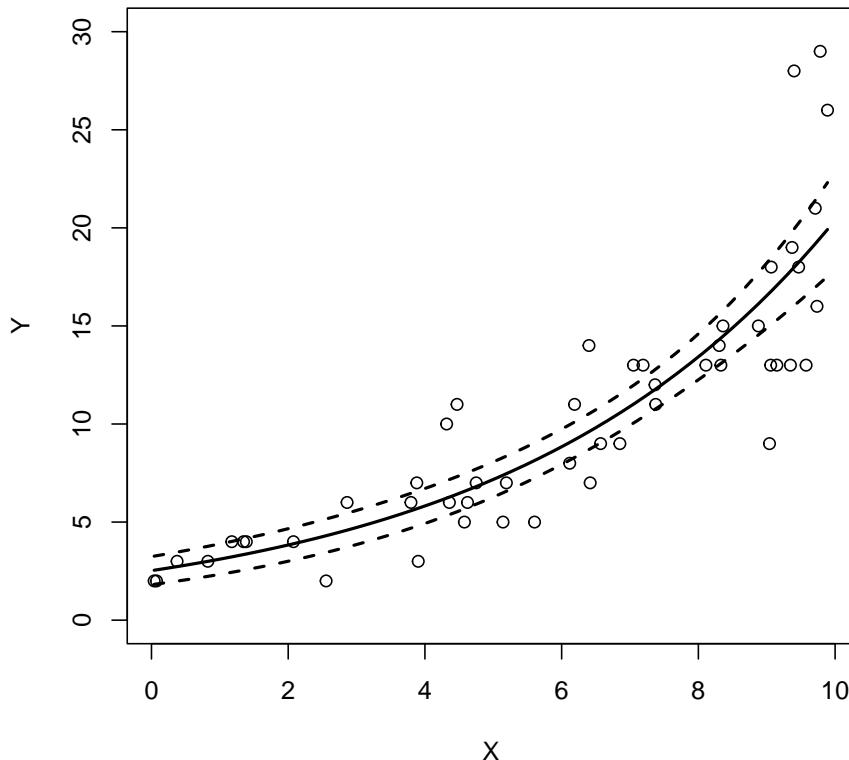
Notice anything odd? The 95% prediction limits increase in discrete steps rather than as smooth curves. This is because we calculated them as quantiles from the Poisson distribution, which can only take on integer values. This means that the PI can only be bounded by integer values. The plotted PI is correct, but looks odd because we are used to seeing smooth curves. It also looks odd because it includes most (if not all) values, which CI do not do.

The second method is a more traditional 95% CI. Unlike the PI, the CI shows the

uncertainty associated with the expected value. This means that the relationship between the PI and CI is similar to the relationship between the SD and SE.

```
pred2 <- predict(mod1,
                  newdata=data.frame(x=px),
                  type="response", se.fit=TRUE)
mn2 <- pred2$fit
lo2 <- mn2 - 1.96*pred2$se.fit
up2 <- mn2 + 1.96*pred2$se.fit

plot(x, y, xlab="X", ylab="Y", type="n",
      ylim=c(0, 30))
points(px, lo2, type="l", lty=2, lwd=2)
points(px, up2, type="l", lty=2, lwd=2)
points(px, mn2, type="l", lwd=2)
points(x,y)
```



Given that the Poisson distribution is defined by a single parameter that is both the mean and variance, it is worth wondering what the SE estimated by `se.fit=TRUE` in the command that created `pred2` really is (after all, it is separate from the mean!). In a nutshell, it is the SE of the expected value assuming that the expected value follows a normal distribution. This assumption turns out to be appropriate for many situations. Taking advantage of the properties of the normal distribution allowed us to use ± 1.96 SE to approximate the 95% CI (verify this with `qnorm(c(0.025, 0.975))`). Alternatively, we could have used:

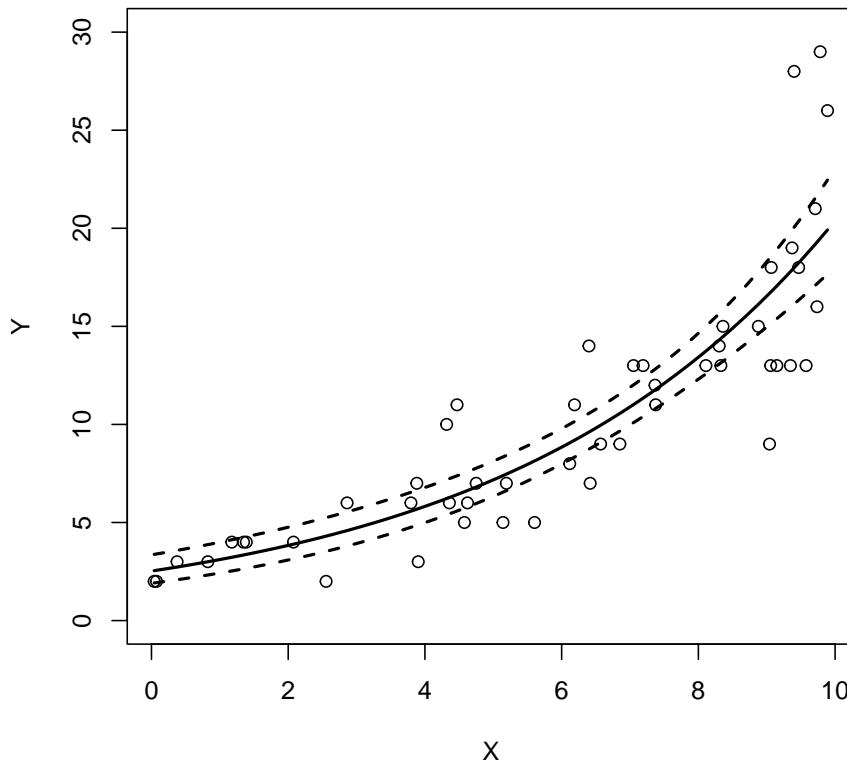
```
# not run:
lo2 <- qnorm(0.025, mn2, pred2$se.fit)
up2 <- qnorm(0.975, mn2, pred2$se.fit)
```

Finally, we could have also generated predictions on the link scale and transformed them to the response scale. This method is often safer than using `type="response"` because the resulting predictions are *guaranteed* to be inside the domain of the response variable.

```
# getting CI from link scale
pred3 <- predict(mod1,
                  newdata=data.frame(x=px),
                  type="link", se.fit=TRUE)
mn3 <- pred3$fit
lo3 <- mn3 - 1.96*pred3$se.fit
up3 <- mn3 + 1.96*pred3$se.fit

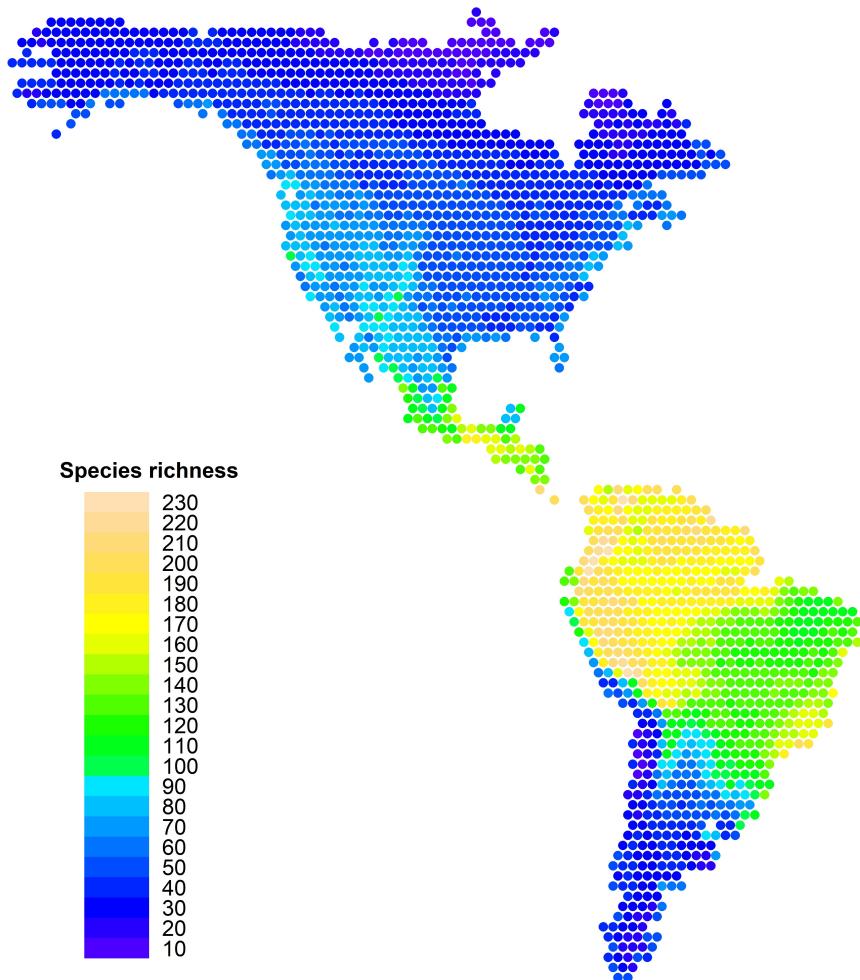
# use inverse link function from model object
mn3 <- mod1$family$linkinv(mn3)
lo3 <- mod1$family$linkinv(lo3)
up3 <- mod1$family$linkinv(up3)

# make plot
plot(x, y, xlab="X", ylab="Y", type="n",
      ylim=c(0, 30))
points(px, lo3, type="l", lty=2, lwd=2)
points(px, up3, type="l", lty=2, lwd=2)
points(px, mn3, type="l", lwd=2)
points(x,y)
```



5.4.2 Example with real data

The study of statistical patterns in biodiversity, abundance, and ecological function across large spatial scales is called **macroecology**. Rather than study individual species and sites, macroecologists integrate data across many studies and databases to search for broadly applicable patterns. Faurby et al. (2018) compiled basic biogeographic, conservation, and life history data for >5800 species of mammals and published them as the database PHYLACINE. We will use the data from PHYLACINE to explore the relationship between mammal species richness and geography in the western hemisphere. The figure below shows the species richness data at 2075 systematically placed points in mainland North America and South America.



The dataset `mammal_data_2021-09-08.csv` contains mammal species richness at about 2000 locations throughout North and South America. These points were defined by longitude and latitude in a regular grid that covered both continents. The PHYLACINE database was then queried to determine which species' ranges overlapped each point. The result is a data frame with the species richness of each point related to its geographic coordinates¹⁰.

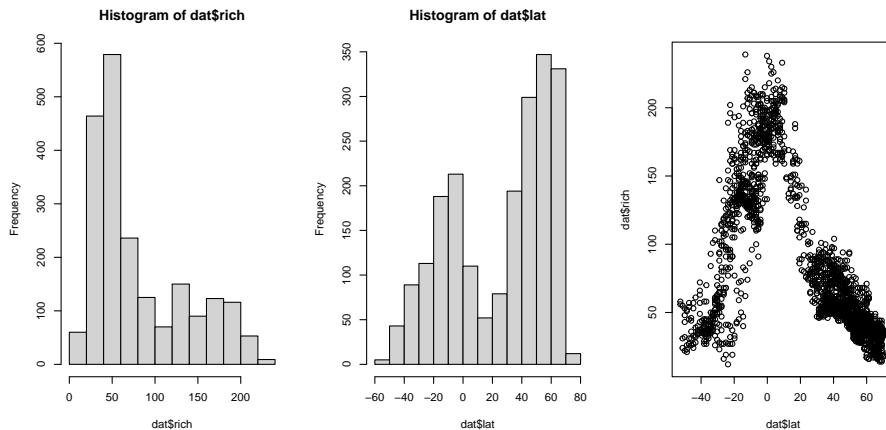
First, download the data and put it in your R home directory. We'll do some data exploration before jumping into the analysis.

¹⁰By the way, the map in this section and all geographic operations were done using the geographic information systems (GIS) capabilities in R.

```
dat <- read.csv("mammal_data_2021-09-08.csv", header=TRUE)
head(dat)

##   X      long     lat rich
## 1 1 -72.50909 -52.49019  58
## 2 2 -70.99915 -52.49019  56
## 3 3 -73.26407 -51.18254  55
## 4 4 -71.75412 -51.18254  31
## 5 5 -70.24417 -51.18254  24
## 6 6 -74.01904 -49.87489  43

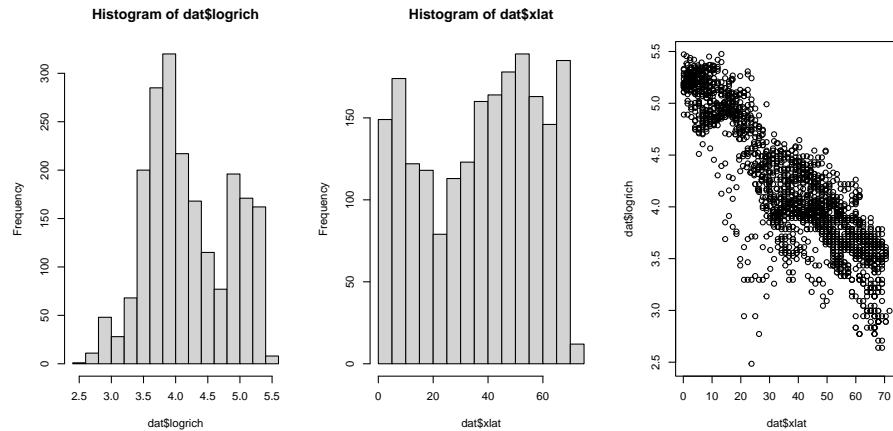
par(mfrow=c(1,3))
hist(dat$rich)
hist(dat$lat)
plot(dat$lat, dat$rich)
```



At first glance, it looks like mammal species richness peaks near the equator (0 = equator, + = north latitude, and - = south latitude). It also looks like the response to latitude is the same whether going north or south. So, let's use the absolute value of latitude instead. While we're at it, let's try out a log-transform on richness. This makes sense because richness (1) appears highly right-skewed; and (2) must be nonnegative.

```
# transform:
dat$xlat <- abs(dat$lat)
dat$logrich <- log(dat$rich)

# look again:
par(mfrow=c(1,3))
hist(dat$logrich)
hist(dat$xlat)
plot(dat$xlat, dat$logrich)
```



The rightmost plot shows that there is a linear relationship between the log-transformed response variable, which represents counts, and the explanatory variable. This means we should try modeling this relationship with a GLM, with a Poisson family and log link function. The Poisson distribution is because the data are counts. The log link is used because it is the canonical link function for the Poisson family.

```
mod1 <- glm(rich~xlat, data=dat, family=poisson)

# check model terms:
summary(mod1)

## 
## Call:
## glm(formula = rich ~ xlat, family = poisson, data = dat)
## 
## Deviance Residuals:
##      Min        1Q    Median        3Q       Max 
## -11.2645   -1.4664    0.0785    1.4371    8.4076 
## 
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)    
## (Intercept) 5.2859143  0.0038991 1355.7  <2e-16 ***
## xlat        -0.0282742  0.0001219 -231.9  <2e-16 ***
## ---        
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
## 
## (Dispersion parameter for poisson family taken to be 1)
## 
## Null deviance: 69333  on 2074  degrees of freedom
```

```

## Residual deviance: 11653  on 2073  degrees of freedom
## AIC: 24186
##
## Number of Fisher Scoring iterations: 4
# pseudo R squared:
1-mod1$deviance/mod1>null.deviance

## [1] 0.8319267

```

As before, we want to present the data with the predictions of the model, and the 95% CI of the predictions. As before, we'll explore different ways of calculating and presenting uncertainty about the predictions. First, we'll use the Poisson distribution to get the 95% prediction interval (the interval where 95% of values should fall). Then, we'll use the normal approximation to get the 95% confidence interval of λ (i.e., the interval where we are 95% certain that the true λ falls).

```

n <- 100
px <- seq(min(dat$xlat), max(dat$xlat), length=n)
dx <- data.frame(xlat=px)
pred1 <- predict(mod1,
                  newdata=data.frame(dx),
                  type="response", se.fit=TRUE)
mn1 <- pred1$fit

# Poisson prediction interval:
lo1 <- qpois(0.025, mn1)
up1 <- qpois(0.975, mn1)

# normal approximation confidence interval:
pred2 <- predict(mod1,
                  newdata=data.frame(dx),
                  type="link", se.fit=TRUE)
mn2 <- pred2$fit
lo2 <- qnorm(0.025, mn2, pred2$se.fit)
up2 <- qnorm(0.975, mn2, pred2$se.fit)

# inverse link function
mn2 <- mod1$family$linkinv(mn2)
lo2 <- mod1$family$linkinv(lo2)
up2 <- mod1$family$linkinv(up2)

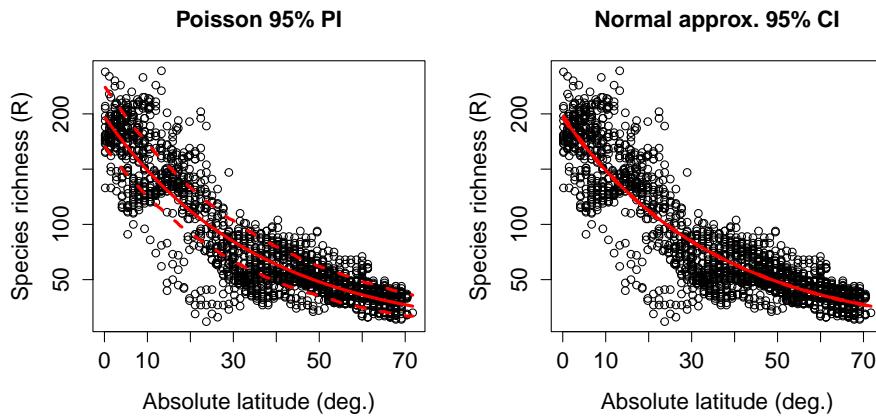
par(mfrow=c(1,2), cex.lab=1.3, cex.main=1.3, cex.axis=1.3)
plot(dat$xlat, dat$rich,
     main="Poisson 95% PI",
     xlab="Absolute latitude (deg.)",
     ylab="Species richness (R)")

```

```

points(px, lo1, type="l", col="red", lwd=3, lty=2)
points(px, up1, type="l", col="red", lwd=3, lty=2)
points(px, mn1, type="l", col="red", lwd=3)
plot(dat$xlat, dat$rich,
     main="Normal approx. 95% CI",
     xlab="Absolute latitude (deg.)",
     ylab="Species richness (R)")
points(px, lo2, type="l", col="red", lwd=3, lty=2)
points(px, up2, type="l", col="red", lwd=3, lty=2)
points(px, mn2, type="l", col="red", lwd=3)

```



The expected value λ is the same for both uncertainty estimations. As expected, the PI is wider than the CI. Why is the CI so narrow? Remember that the SE is proportional to the inverse square root of the sample size ($SE(x) = \sigma/\sqrt{n}$). The n of this model was very large, 2075 (try `nrow(dat)`). A model fit with smaller n would have a wider CI. We can see this for ourselves by refitting the model with fewer observations.

```

# number of x values for CI
n.pred <- 100

# number of observations in each simulation
use.n <- 1:6*10
n.sims <- length(use.n)

# lists to hold data, models, and predicted values/CI
mod.list <- vector("list", n.sims)
dat.list <- mod.list
pred.list <- mod.list

# fit models and calculate predictions in a loop

```

```

for(i in 1:n.sims){
  # randomly select rows from full dataset
  use.rows <- sample(1:nrow(dat), use.n[i])
  curr.d <- dat[use.rows,]

  # fit model to current subset of data
  curr.m <- glm(rich~xlat, data=curr.d, family=poisson)

  # calculate predictions on current model
  curr.px <- seq(min(curr.d$xlat),
                  max(curr.d$xlat),
                  length=n.pred)
  curr.pr <- predict(curr.m,
                      data.frame(xlat=px),
                      type="link",
                      se.fit=TRUE)

  # store mean, upper, and lower conf. limits in a matrix
  curr.ci <- matrix(NA, nrow=n.pred, ncol=3)
  curr.ci[,1] <- curr.pr$fit
  curr.ci[,2] <- qnorm(0.025, curr.ci[,1], curr.pr$se.fit)
  curr.ci[,3] <- qnorm(0.975, curr.ci[,1], curr.pr$se.fit)

  curr.ci[,1] <- curr.m$family$linkinv(curr.ci[,1])
  curr.ci[,2] <- curr.m$family$linkinv(curr.ci[,2])
  curr.ci[,3] <- curr.m$family$linkinv(curr.ci[,3])

  # save everything to the lists
  # (note double brackets for accessing elements of a list)
  mod.list[[i]] <- curr.m
  dat.list[[i]] <- curr.d
  pred.list[[i]] <- curr.ci
}

}

```

Inspect the outputs to see what the simulations did. Now let's plot the predicted values and CI for the models fitted to subsets of the data.

```

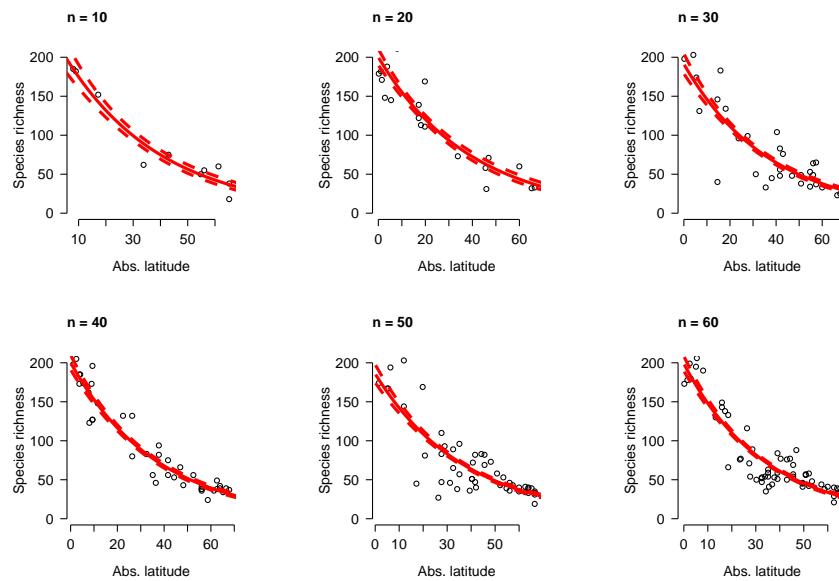
par(mfrow=c(2,3), mar=c(5.1, 5.1, 5.1, 5.1),
    bty="n", lend=1, las=1,
    cex.lab=1.3, cex.axis=1.3)
for(i in 1:n.sims){
  plot(dat.list[[i]]$xlat, dat.list[[i]]$rich,
       ylim=c(0, 200),
       xlab="Abs. latitude",
       ylab="Species richness")
  points(px, pred.list[[i]][,2],
         type="l", col="red", lwd=3, lty=2)
}

```

```

points(px, pred.list[[i]][,3],
      type="l", col="red", lwd=3, lty=2)
points(px, pred.list[[i]][,1],
      type="l", col="red", lwd=3)
title(main=paste("n =", use.n[i]), adj=0)
}

```



The simulations show that the 95% CI of λ narrows down very quickly as sample size increases from 10 to 60. No wonder we can't even see the interval when the model is fit with 2075 observations!

Just as linear regression can be extended to multiple linear regression, Poisson regression can be extended with additional predictor variables. Remember how earlier we lumped the northern and southern hemispheres together by taking the absolute value of latitude? Let's see if treating the hemispheres separately can improve the model fit.

First, make a factor that will identify the hemisphere of each observation.

```
dat$hemi <- ifelse(dat$lat < 0, "south", "north")
```

Next, refit the model with the new variable. We'll test whether latitude and hemisphere interact. In other words, we'll see whether the relationship between latitude and species richness varies by hemisphere.

```

mod2 <- glm(rich~xlat*hemi, data=dat, family=poisson)
# check model terms:
summary(mod2)

```

```

## 
## Call:
## glm(formula = rich ~ xlat * hemi, family = poisson, data = dat)
##
## Deviance Residuals:
##      Min       1Q   Median       3Q      Max
## -10.4317  -1.5948  -0.0442   1.3691   9.5437
##
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept) 5.3418617  0.0063816 837.073 <2e-16 ***
## xlat        -0.0287144  0.0001573 -182.532 <2e-16 ***
## hemisouth    -0.0163170  0.0085718  -1.904   0.057 .
## xlat:hemisouth -0.0057775  0.0003667  -15.757 <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for poisson family taken to be 1)
##
## Null deviance: 69333  on 2074  degrees of freedom
## Residual deviance: 10964  on 2071  degrees of freedom
## AIC: 23501
##
## Number of Fisher Scoring iterations: 4

```

Sure enough, the interaction appears significant. The coefficients tell us that in the baseline group (northern hemisphere), $\log(R)$ decreases by 0.0287 for every extra degree of latitude away from the equator. In the southern hemisphere, the effect is stronger: for every additional degree of latitude, $\log(R)$ decreases by $(-0.0287) + (-0.0058) = 0.0345$. The coefficient for the interaction `xlat:hemisouth` is the effect on the slope (`xlat`) of being in a group other than the baseline. The fact that the effect of `hemisouth` was nonsignificant just means that the intercept of the relationship for the southern hemisphere is not significantly different than the intercept of the relationship for the northern hemisphere (can you think of a biological reason why this should be the case?).

Did the additional term improve the model? Not much. Recall that the model without the interaction had a pseudo- R^2 of about 0.832. Still, the new model appears to be an improvement because of its much smaller AIC.

```

# pseudo R squared:
1-mod2$deviance/mod2>null.deviance

## [1] 0.8418692
AIC(mod1, mod2)

##      df      AIC

```

```
## mod1  2 24186.10
## mod2  4 23500.75
```

We can confirm that model 2 is a better fit using a **likelihood ratio test (LR test)**. The easiest way is with function `lrtest()` in package `lmtest`. A LR test tests the null hypothesis that the ratio of the likelihood functions of two models is equal to 1 (or, that the natural log of this ratio is different than 0). LR tests can be used to compare GLMs that are nested: the more complicated model can be converted into the simpler model by constraining some of its parameters. In our example, model 2 can be converted into model 1 by setting the coefficients for hemisphere and latitude \times hemisphere to 0. Thus, model 1 is nested within model 2. If models are not nested, then the LR test will not be valid.

```
library(lmtest)

## Loading required package: zoo
##
## Attaching package: 'zoo'
## The following objects are masked from 'package:base':
##       as.Date, as.Date.numeric
##
## Attaching package: 'lmtest'
## The following object is masked _by_ '.GlobalEnv':
##       ip
lrtest(mod1, mod2)

## Likelihood ratio test
##
## Model 1: rich ~ xlat
## Model 2: rich ~ xlat * hemi
##      #Df LogLik Df Chisq Pr(>Chisq)
## 1    2   -12091
## 2    4   -11746  2  689.35 < 2.2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

The fact that the LR test was significant shows that the fit of model 2 is significantly better than that of model 1, after penalizing for the number of parameters.

As always, let's calculate the model predictions and present them with the original data. We'll skip presenting the 95% CI, because it was so small in the previous example and won't tell us much. In the code below we need to define predictor variables in the range actually seen for each factor.

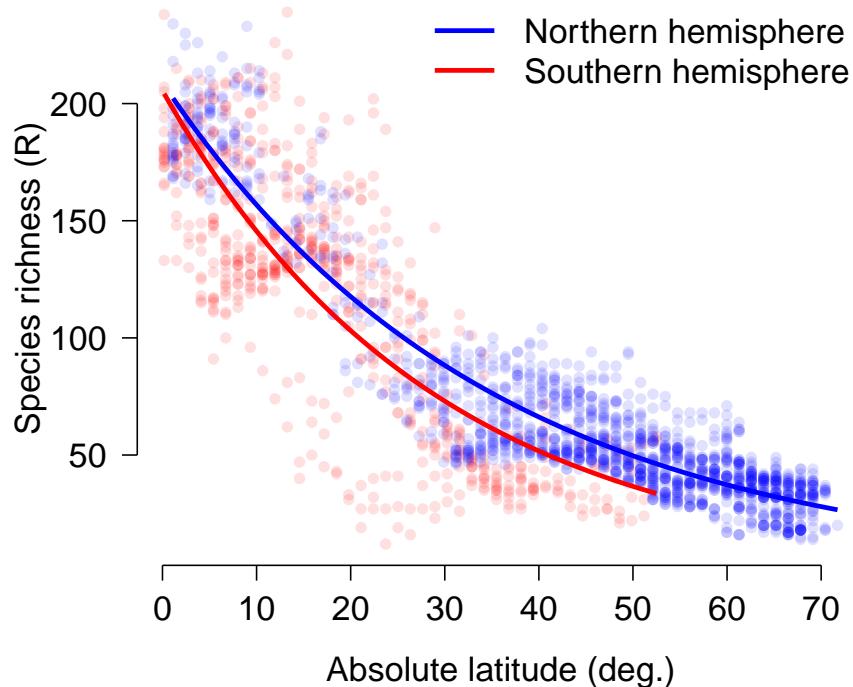
```

n <- 100
flag <- which(dat$hemi=="south")
px1 <- data.frame(
  hemi="south",
  xlat=seq(min(dat$xlat[flag]),
            max(dat$xlat[flag]), length=n))
px2 <- data.frame(
  hemi="north",
  xlat=seq(min(dat$xlat[-flag]),
            max(dat$xlat[-flag]), length=n))
dx <- rbind(px1, px2)

pred2 <- predict(mod2,
                  newdata=data.frame(dx),
                  type="link")
mn2 <- mod2$family$linkinv(pred2)

par(mfrow=c(1,1), cex.lab=1.3, cex.axis=1.3,
    mar=c(5.1, 5.1, 1.1, 1.1),
    bty="n", lend=1, las=1)
plot(dat$xlat, dat$rich, pch=16,
      col=ifelse(dat$hemi == "south", "#FF000020", "#0000FF20"),
      xlab="Absolute latitude (deg.)",
      ylab="Species richness (R)")
flag1 <- which(dx$hemi == "south")
flag2 <- which(dx$hemi != "south")
points(px1$xlat, mn2[flag1], type="l", col="red", lwd=3)
points(px2$xlat, mn2[flag2], type="l", col="blue", lwd=3)
legend("topright",
       legend=c("Northern hemisphere", "Southern hemisphere"),
       lwd=3, col=c("blue", "red"), bty="n", cex=1.3)

```



The figure above shows that species richness falls off with distance from the equator slightly faster in the southern hemisphere than in the northern hemisphere.

5.5 Quasi-Poisson and negative binomial GLM

This section explores GLMs for **overdispersed count data**. Count data are exactly what they sound like: values resulting from counting things. Count data are usually well-modeled by the Poisson distribution. In the Poisson distribution, the mean and variance are the same value (i.e., $\mu = \sigma^2$). Sometimes count data are **overdispersed**, which means that the variance is greater than would be expected for a Poisson distribution (i.e., $\sigma^2 > \mu$)¹¹. These data can still be modeled with GLM, but a different family is necessary to account for the overdispersion. The most common options for modeling overdispersed counts are the **quasi-Poisson distribution** and the **negative binomial distribution**. These are different distributions, and GLMs using them are technically different models, but we will see below that any situation where one is appropriate, the other is as well.

¹¹As you might suspect, the opposite situation, when $\mu > \sigma^2$, is called “underdispersion”. Underdispersion does not come up as often as overdispersion and poses less of a problem for analysis.

As useful as the Poisson distribution is for modeling counts, it has a serious limitation: the assumption that its single parameter λ is both the mean and variance (i.e., $\lambda = \mu = \sigma^2$). However, this is not always the case in real datasets. Many biological processes that produce counts have a variance much greater than their mean. This phenomenon is called “overdispersion”. The “over” in “overdispersion” is defined relative to the Poisson distribution. Using a different distribution in our GLM will account for the “over”. The two most common options for modeling overdispersed counts are the quasi-Poisson family and the negative binomial family.

The quasi-Poisson family is part of a class of **quasi-likelihood** models that inflate the variance to account for overdispersion (Bolker 2008). Quasi-likelihoods add a parameter that increases the variance in a distribution. This parameter is usually called ϕ (Greek letter “phi”) or \hat{c} (“c-hat”). This parameter is typically added as a scalar to the expression for variance, so it tends to be >1 . For example, the variance of a Poisson distribution is the same as its mean, λ . The variance of a quasi-Poisson distribution is $\phi\lambda$. The overdispersion parameter ϕ will be estimated by R as it fits the GLM. The disadvantage of using quasi-likelihood models is that the likelihood functions are never explicitly defined, which means that many goodness of fit measures like deviance and AIC cannot be easily calculated. Burnham and Anderson (2004) defined a **quasi-AIC** (**qAIC**) that can be used in place of ordinary AIC for model selection procedures.

The **negative binomial distribution** can also be used as a response distribution for overdispersed count data. The negative binomial distribution has two definitions. The original definition describes the number of failures that occur in a series of Bernoulli trials until some predetermined number of successes is observed. The version of the negative binomial that biologists use is parameterized by its mean μ and its overdispersion k . This definition reframes the negative binomial as a Poisson distribution with parameter λ , where λ is a random variable that follows a Gamma distribution. In practice this is sort of like a quasi-Poisson distribution, but turns out to be easier to work with in R. Compare the variance for a quasi-Poisson distribution, $\phi\lambda$, to the variance of a negative binomial distribution:

$$Var(X) = \mu + \frac{\mu^2}{k}$$

If the mean of a quasi-Poisson distribution is λ , then we can do some rearranging to find that the k parameter of a negative binomial is equivalent to

$$k = \frac{\lambda}{\phi - 1}$$

In practical terms, this means that the quasi-Poisson ϕ and the negative binomial k have opposite effects on the variance. This also means that any situation where one of these models is appropriate, the other is also appropriate. Personally, I

tend to use the negative binomial because it has a well-defined likelihood (unlike quasi-Poisson) and is thus easier to work with in R.

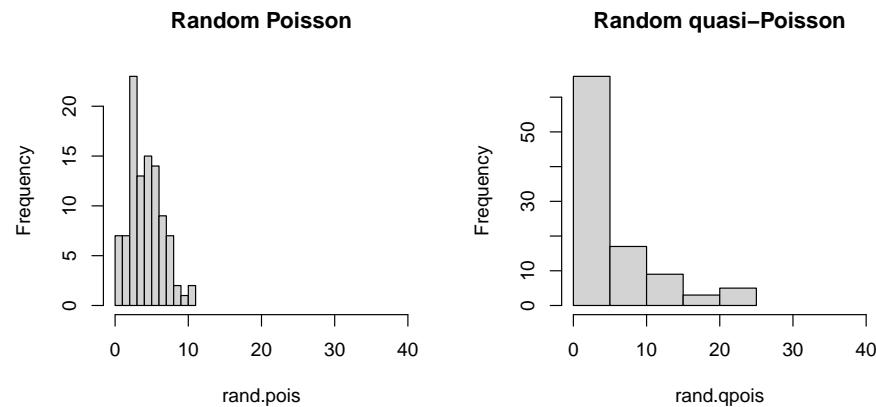
5.5.1 Example with simulated data

R does not have a function for drawing random numbers from a quasi-Poisson distribution, but it does have a function for drawing random numbers from a negative binomial distribution. We can take advantage of the relationship between the quasi-Poisson ϕ and the negative binomial k to define our own quasi-Poisson function:

```
rqpois <- function(n, lambda, phi){
  rnbinom(n=n, mu=lambda, size=lambda/(phi-1))
}

# compare the variances of these distributions:
rand.pois <- rpois(100, 5)
rand.qpois <- rqpois(100, 5, 6)
mean(rand.pois)
## [1] 4.68
var(rand.pois)
## [1] 5.351111
mean(rand.qpois)
## [1] 5.66
var(rand.qpois)
## [1] 35.11556

par(mfrow=c(1,2))
hist(rand.pois, xlim=c(0,40), main="Random Poisson")
hist(rand.qpois, xlim=c(0,40), main="Random quasi-Poisson")
```



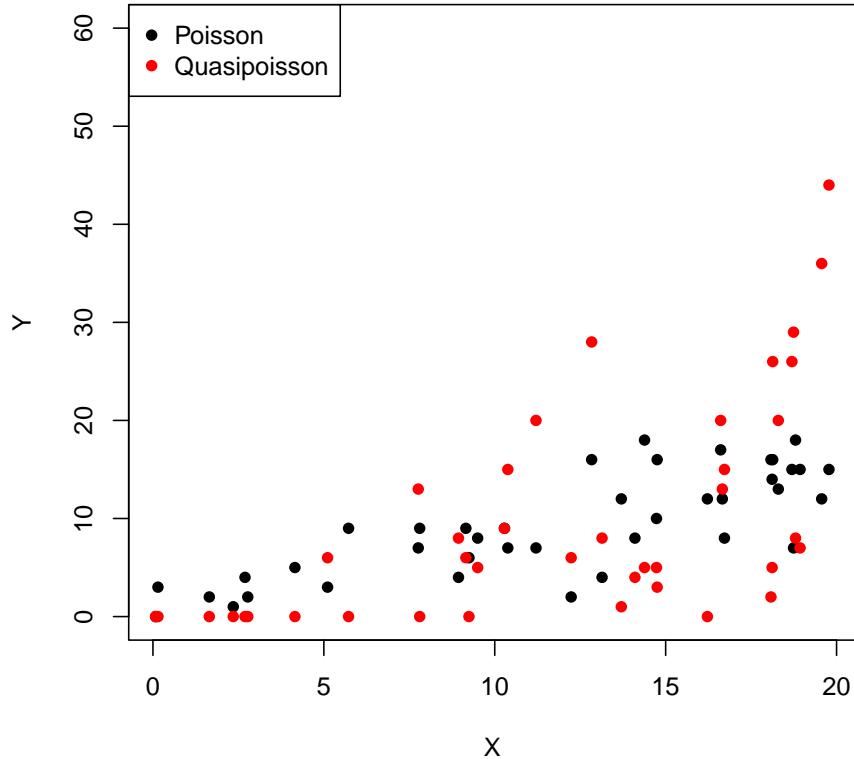
Now let's simulate some overdispersed count data.

```
set.seed(42)
n <- 40
x <- runif(n, 0, 20)
beta0 <- 1.1
beta1 <- 0.72
phi <- 16
eta <- beta0 + beta1 * x

# Poisson data
yp <- rpois(n, eta)

# quasi-Poisson data
yq <- rqpois(n, eta, phi)

# plot the data
plot(x, yp, pch=16, ylim=c(0, 60),
      ylab="Y", xlab="X")
points(x, yq, pch=16, col="red")
legend("topleft",
       legend=c("Poisson", "Quasipoisson"),
       pch=16, col=c("black", "red"))
```



See what the overdispersion term did? The data simulated using a quasi-Poisson distribution are much more spread out than the data simulated using the Poisson distribution. Analyzing the quasi-Poisson data as if they were Poisson distributed might lead to underestimating the variance, especially at greater X values.

```
dat <- data.frame(x=x, y=yq)
mod1 <- glm(y~x, data=dat, family=poisson)
mod2 <- glm(y~x, data=dat, family=quasipoisson)

summary(mod1)

##
## Call:
## glm(formula = y ~ x, family = poisson, data = dat)
##
## Deviance Residuals:
##      Min        1Q    Median        3Q       Max
## -10.0000  -4.8828   0.0000  10.0000  10.0000
```

```

## -5.223 -2.174 -1.024  1.405  5.439
##
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept) 0.12441   0.20218   0.615   0.538
## x          0.15343   0.01236  12.416  <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for poisson family taken to be 1)
##
## Null deviance: 487.11 on 39 degrees of freedom
## Residual deviance: 274.66 on 38 degrees of freedom
## AIC: 398.22
##
## Number of Fisher Scoring iterations: 6
summary(mod2)

##
## Call:
## glm(formula = y ~ x, family = quasipoisson, data = dat)
##
## Deviance Residuals:
##    Min      1Q  Median      3Q     Max
## -5.223 -2.174 -1.024  1.405  5.439
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 0.12441   0.53258   0.234   0.817
## x          0.15343   0.03255   4.713 3.24e-05 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for quasipoisson family taken to be 6.939018)
##
## Null deviance: 487.11 on 39 degrees of freedom
## Residual deviance: 274.66 on 38 degrees of freedom
## AIC: NA
##
## Number of Fisher Scoring iterations: 6

```

The model fits, parameter estimates, and diagnostics like deviance are essentially identical. But, as the figures below show, the Poisson GLM predicted much less variation about the mean than did the quasi-Poisson model.

```

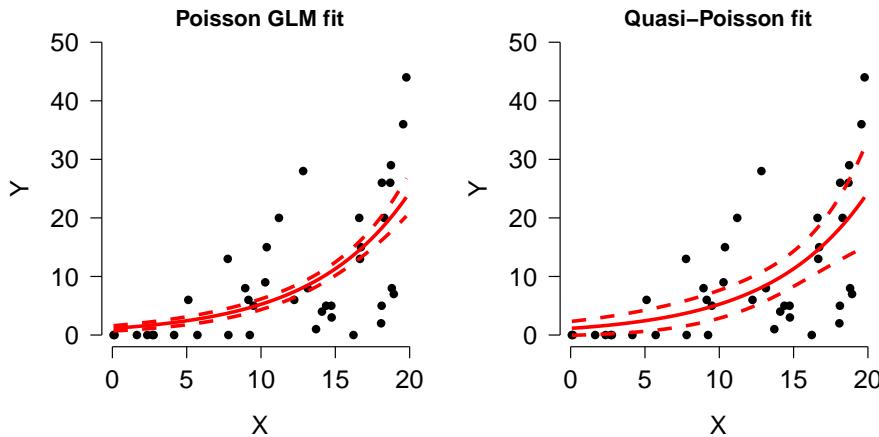
# values for prediction
n.pr <- 100
px <- seq(min(x), max(x), length=n.pr)
pred1 <- predict(mod1, newdata=data.frame(x=px),
                  type="response", se.fit=TRUE)
pred2 <- predict(mod2, newdata=data.frame(x=px),
                  type="response", se.fit=TRUE)

# pull out predictions and calculate CI
mn1 <- pred1$fit
mn2 <- pred2$fit
lo1 <- qnorm(0.025, mn1, pred1$se.fit)
lo2 <- qnorm(0.025, mn2, pred2$se.fit)
up1 <- qnorm(0.975, mn1, pred1$se.fit)
up2 <- qnorm(0.975, mn2, pred2$se.fit)

# make the plots
par(mfrow=c(1,2), bty="n",
    mar=c(5.1, 5.1, 1.1, 1.1),
    lend=1, las=1, cex.lab=1.3, cex.axis=1.3)
plot(x, yq, pch=16, ylim=c(0, 50),
      ylab="Y", xlab="X",
      main="Poisson GLM fit")
points(px, lo1, type="l", col="red", lwd=3, lty=2)
points(px, up1, type="l", col="red", lwd=3, lty=2)
points(px, mn1, type="l", col="red", lwd=3)

plot(x, yq, pch=16, ylim=c(0, 50),
      ylab="Y", xlab="X",
      main="Quasi-Poisson fit")
points(px, lo2, type="l", col="red", lwd=3, lty=2)
points(px, up2, type="l", col="red", lwd=3, lty=2)
points(px, mn2, type="l", col="red", lwd=3)

```



The right plot looks reasonable, but there is one small problem: the 95% CI calculated on the response scale extends below 0, which is not defined for a Poisson distribution. So, we should have calculated the CI on the link scale and back-transformed to get values on the response scale.

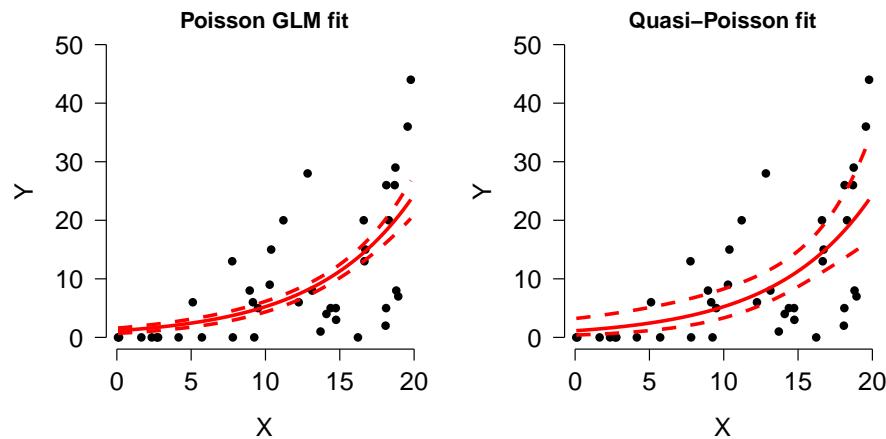
```
# predict on link scale instead:
pred2 <- predict(mod2, newdata=data.frame(x=px),
                  type="link", se.fit=TRUE)
mn2 <- pred2$fit
lo2 <- qnorm(0.025, mn2, pred2$se.fit)
up2 <- qnorm(0.975, mn2, pred2$se.fit)

# back-transform:
mn2 <- mod2$family$linkinv(mn2)
lo2 <- mod2$family$linkinv(lo2)
up2 <- mod2$family$linkinv(up2)

# make the plots
par(mfrow=c(1,2), bty="n",
     mar=c(5.1, 5.1, 1.1, 1.1),
     lend=1, las=1, cex.lab=1.3, cex.axis=1.3)
plot(x, yq, pch=16, ylim=c(0, 50),
      ylab="Y", xlab="X",
      main="Poisson GLM fit")
points(px, lo1, type="l", col="red", lwd=3, lty=2)
points(px, up1, type="l", col="red", lwd=3, lty=2)
points(px, mn1, type="l", col="red", lwd=3)

plot(x, yq, pch=16, ylim=c(0, 50),
      ylab="Y", xlab="X",
      main="Quasi-Poisson fit")
```

```
points(px, lo2, type="l", col="red", lwd=3, lty=2)
points(px, up2, type="l", col="red", lwd=3, lty=2)
points(px, mn2, type="l", col="red", lwd=3)
```



Notice that the lower confidence limit does not dip below 0 in the revised quasi-Poisson plot. The CI is somewhat asymmetrical, but that's okay and not at all uncommon with log-transformed data or in models with log-link functions.

5.5.2 Example with real data

The US National Oceanic and Atmospheric Administration (NOAA) conducts periodic coral reef surveys in the Atlantic and Caribbean to monitor coral status (NOAA 2018). For this example, we will use data from 2016-2018 to explore the relationship between coral species richness and environmental factors. The survey sites are shown on the map below in relation to mainland Florida, the Keys, and the Tortugas¹².

¹²As you may have guessed as you clicked the footnote, this map was made in R.



Download the dataset `coral_glm_example.csv` and put it in your R home directory.

```
in.name <- "coral_glm_example.csv"
dat <- read.csv(in.name, header=TRUE)
```

The dataset contains many variables, but we are interested in the following:

Variable	Units	Meaning
<code>date</code>	Days	Date sample was taken
<code>lat</code>	°N	North latitude
<code>long</code>	°E	Degrees east longitude
<code>rugo</code>	Unitless	Index of bottom roughness (rugosity)
<code>dep</code>	m	Mean depth of coral transect
<code>region</code>	Unitless	Geographic region where sample was taken
<code>rich</code>	Species	Number of coral species recorded on transect

Let's start by using histograms and scatterplot matrices to get a sense of what patterns might exist in the data.

```
head(dat)
```

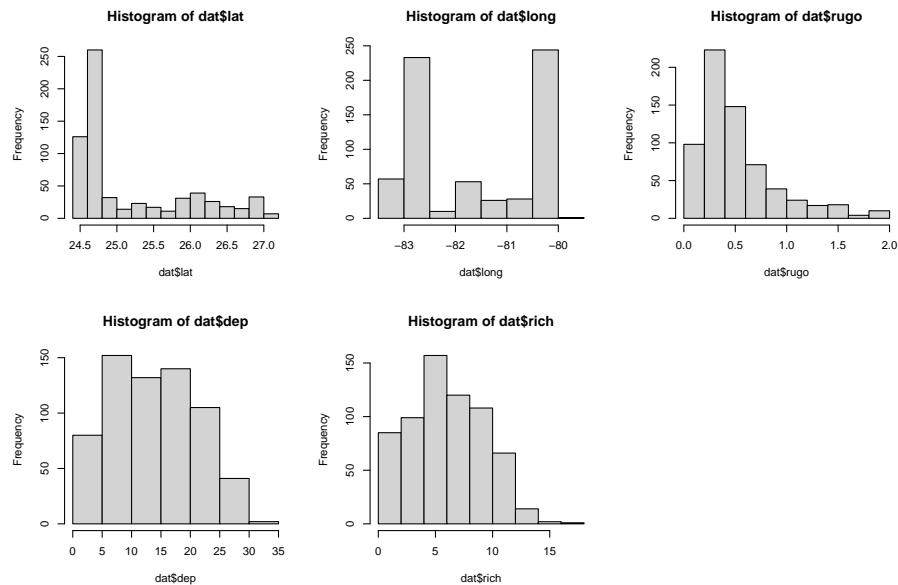
```
##      date      lat      long      rugo      dep region rich
## 1 5/12/2016 24.72435 -82.81070 0.8433333 16.3068 Tortugas     8
## 2 5/12/2016 24.72367 -82.78785 0.4900000 16.3068 Tortugas     8
```

```

## 3 5/12/2016 24.72623 -82.80449 0.1766667 17.0688 Tortugas 1
## 4 5/12/2016 24.71759 -82.80542 0.2966667 13.4112 Tortugas 7
## 5 5/12/2016 24.71971 -82.81462 0.6300000 11.1252 Tortugas 5
## 6 5/12/2016 24.72118 -82.80761 0.8833333 12.4968 Tortugas 9

par(mfrow=c(2,3))
hist(dat$lat)
hist(dat$long)
hist(dat$rugo)
hist(dat$dep)
hist(dat$rich)

```

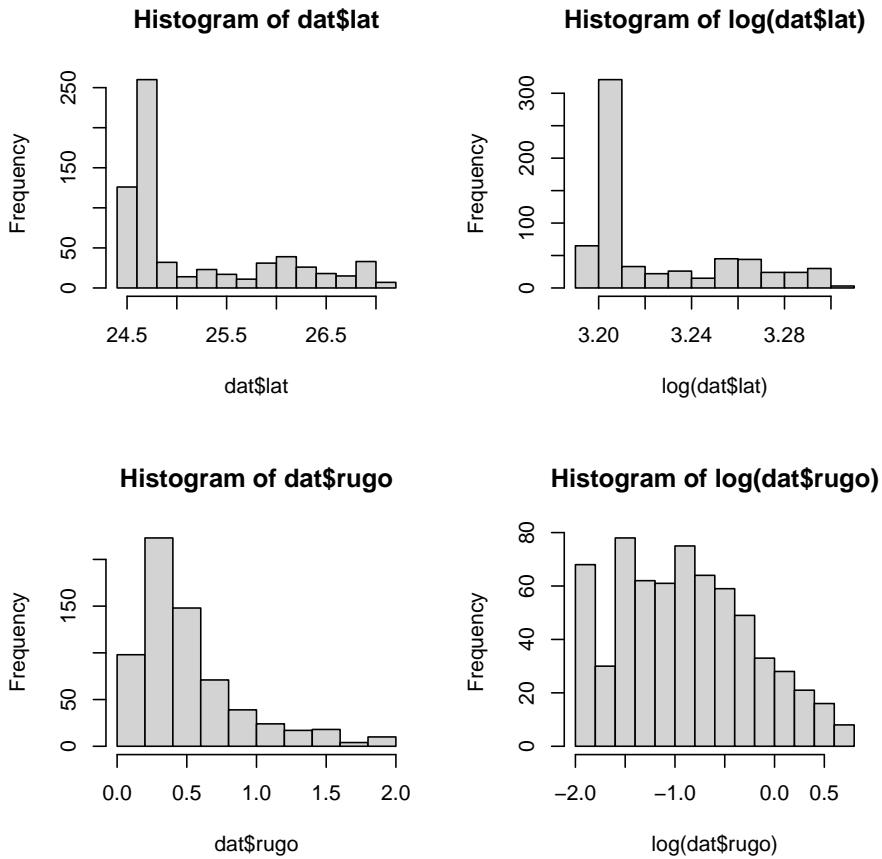


The histograms suggest that latitude and rugosity might cause problems because of how many observations are clustered around a few values. Perhaps a log-transform would help? The plots from the commands below suggest not.

```

par(mfrow=c(2,2))
hist(dat$lat)
hist(log(dat$lat))
hist(dat$rugo)
hist(log(dat$rugo))

```



Next, search for patterns in the dataset using some scatterplot matrices. We will make use of the custom function `panel.cor()`, which can be found in the examples in the help page for function `pairs()`.

```
# make day of year a number
dat$doy <- as.numeric(format(as.Date(dat$date), "%j"))

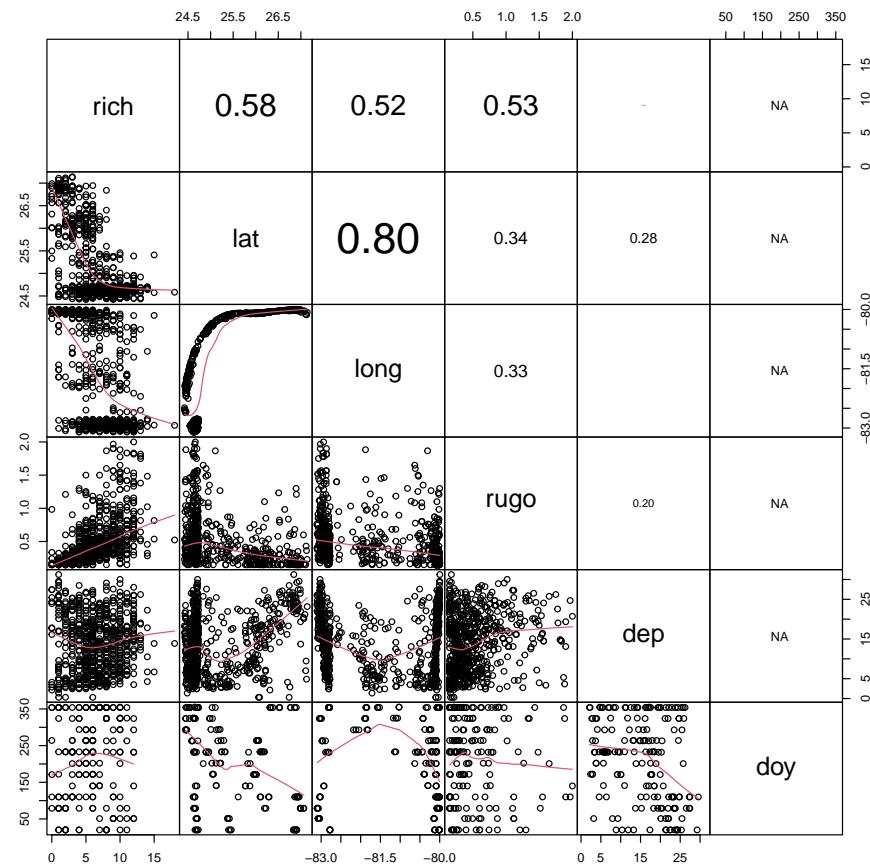
# scatterplot matrices to explore
panel.cor <- function(x, y, digits = 2, prefix = "", cex.cor, ...)
{
  usr <- par("usr"); on.exit(par(usr))
  par(usd = c(0, 1, 0, 1))
  r <- abs(cor(x, y))
  txt <- format(c(r, 0.123456789), digits = digits)[1]
  txt <- paste0(prefix, txt)
  if(missing(cex.cor)) cex.cor <- 0.8/strwidth(txt)
  text(0.5, 0.5, txt, cex = cex.cor * r)
```

```

}

num.cols <- c("rich", "lat", "long", "rugo", "dep", "doy")
pairs(dat[,num.cols], lower.panel = panel.smooth,
      upper.panel = panel.cor, gap=0)

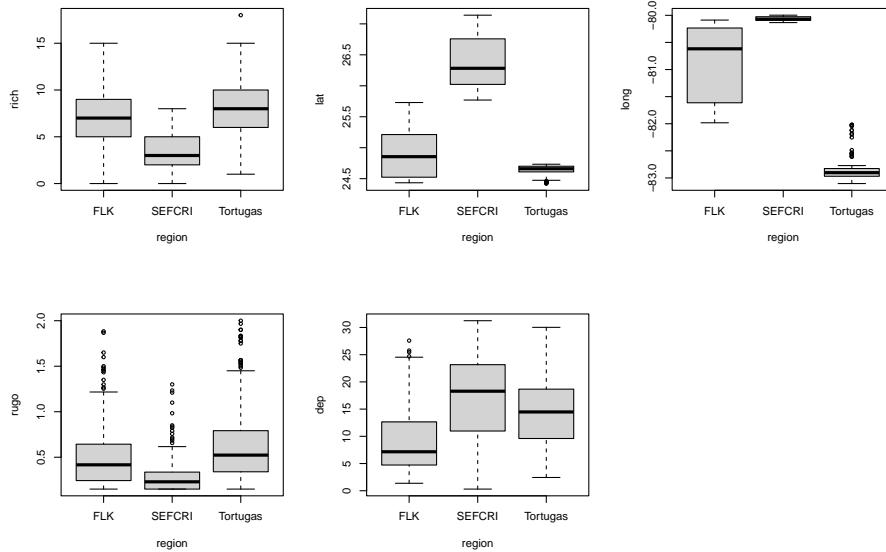
```



It appears that richness is somewhat correlated with latitude, longitude, and rugosity. Interestingly, latitude and longitude appear correlated. Can you think of why? (Hint: plot the coordinates and compare to a map of the Florida Keys). Because latitude and longitude are correlated, we should probably not fit a model with both of these variables included.

Based on the scatterplot matrix, it appears that latitude, longitude, and rugosity are most likely to be the variables driving coral species richness. There is a factor in our dataset, region, that also appears to play a role. However, we won't model region with latitude or longitude because it is confounded with both.

```
# possible effect of region, confounded with lat/long
par(mfrow=c(2,3))
boxplot(rich~region, data=dat)
boxplot(lat~region, data=dat)
boxplot(long~region, data=dat)
boxplot(rugo~region, data=dat)
boxplot(dep~region, data=dat)
```



Finally, we suspect that there might be overdispersion in our response variable richness. This can be checked comparing the mean and variance of its distribution. Recall that a Poisson distribution has a single parameter λ that is both its mean and variance. If the variance of a count variable is larger than its mean, overdispersion should be suspected.

```
# possible overdispersion because var() > mean()
var(dat$rich)
## [1] 11.15052
mean(dat$rich)
## [1] 6.504601
```

How much larger than the mean can variance be before we start to worry about overdispersion? Opinions differ. Some authors use the ratio $\rho = \sigma^2/\mu$ to define a rule of thumb (note that ρ is the Greek letter “rho”, not the Latin letter “p”). The threshold ρ varies by author: values as small as 2 and as large as 50 can be found in the literature! Our value, 1.71, is not very concerning but we should be

aware of possible overdispersion anyway.

The commands below define some models that we will compare using information theoretic methods (i.e., AIC). We will not fit and compare every possible combination of predictors. Instead, we will fit only a narrow subset suggested by the exploratory data analysis. Julian date (doy, for “day of year”) is not considered. The only model with two continuous predictors has longitude and depth. Any other pair has some risk of collinearity. Region will only be tested by itself because the other variables appear to vary by region. Finally, we will test two versions of each model: one with a Poisson family, and another with a quasi-Poisson family that accounts for overdispersion.

```
mod01 <- glm(rich~lat, data=dat, family=poisson)
mod02 <- glm(rich~long, data=dat, family=poisson)
mod03 <- glm(rich~rugo, data=dat, family=poisson)
mod04 <- glm(rich~dep, data=dat, family=poisson)
mod05 <- glm(rich~long+dep, data=dat, family=poisson)
mod06 <- glm(rich~region, data=dat, family=poisson)
mod07 <- glm(rich~lat, data=dat, family=quasipoisson)
mod08 <- glm(rich~long, data=dat, family=quasipoisson)
mod09 <- glm(rich~rugo, data=dat, family=quasipoisson)
mod10 <- glm(rich~dep, data=dat, family=quasipoisson)
mod11 <- glm(rich~long+dep, data=dat, family=quasipoisson)
mod12 <- glm(rich~region, data=dat, family=quasipoisson)
```

Next, we will use Akaike’s information criterion (AIC) to compare the goodness-of-fit of the models. Recall that AIC rewards predictive ability and penalizes model complexity (i.e., number of parameters), with smaller AIC indicating better fit.

```
aicdf <- AIC(mod01, mod02, mod03,
               mod04, mod05, mod06,
               mod07, mod08, mod09,
               mod10, mod11, mod12)

aicdf
```

	df	AIC
## mod01	2	3131.119
## mod02	2	3254.461
## mod03	2	3287.994
## mod04	2	3567.371
## mod05	3	3256.438
## mod06	3	3128.355
## mod07	2	NA
## mod08	2	NA
## mod09	2	NA
## mod10	2	NA
## mod11	3	NA

```
## mod12  3      NA
```

The table of AIC values illustrates a downside of using quasi-likelihoods: because the likelihood is not actually defined, it is hard to calculate an AIC value (which depends on the likelihood function). There is such a thing as a “quasi-AIC” that you can use. But, we can take advantage of the fact that the quasi-Poisson and negative binomial are more or less interchangeable and refit models 7-12 using the negative binomial family. The base `glm()` function does not fit negative binomial GLMs, but there are a few options for negative binomial GLMs in add-on packages. We will use the function `glm.nb()` from package MASS.

```
library(MASS)
mod13 <- glm.nb(rich~lat, data=dat)
mod14 <- glm.nb (rich~long, data=dat)
mod15 <- glm.nb (rich~rugo, data=dat)
mod16 <- glm.nb (rich~dep, data=dat)
mod17 <- glm.nb (rich~long+dep, data=dat)
mod18 <- glm.nb (rich~region, data=dat)

# calculate AIC
aicdf <- AIC(mod01, mod02, mod03,
               mod04, mod05, mod06,
               mod13, mod14, mod15,
               mod16, mod17, mod18)
aicdf

##      df      AIC
## mod01  2 3131.119
## mod02  2 3254.461
## mod03  2 3287.994
## mod04  2 3567.371
## mod05  3 3256.438
## mod06  3 3128.355
## mod13  3 3126.964
## mod14  3 3235.538
## mod15  3 3260.363
## mod16  3 3439.161
## mod17  4 3237.537
## mod18  4 3126.858
```

Next we calculate and use the **AIC weight** to determine which of these models is most likely to be best-fitting. *This is not the same as calculating the probability that any particular model is correct.* Rather, the AIC weight represents the probability that each model is the best-fitting of the models included in the AIC weight calculation.

```
aicdf <- aicdf[order(aicdf$AIC),]
aicdf$delta <- aicdf$AIC - min(aicdf$AIC)
```

```

aicdf$wt <- exp(-0.5*aicdf$delta)
aicdf$wt <- aicdf$wt/sum(aicdf$wt)
aicdf <- aicdf[order(-aicdf$wt),]
aicdf$cumsum <- cumsum(aicdf$wt)
aicdf

##      df      AIC      delta      wt      cumsum
## mod18  4 3126.858  0.0000000 3.937070e-01 0.3937070
## mod13  3 3126.964  0.1062192 3.733429e-01 0.7670499
## mod06  3 3128.355  1.4976903 1.861889e-01 0.9532388
## mod01  2 3131.119  4.2611044 4.676124e-02 1.0000000
## mod14  3 3235.538 108.6804455 9.897242e-25 1.0000000
## mod17  4 3237.537 110.6791284 3.643391e-25 1.0000000
## mod02  2 3254.461 127.6037518 7.697868e-29 1.0000000
## mod05  3 3256.438 129.5806317 2.864814e-29 1.0000000
## mod15  3 3260.363 133.5057437 4.025031e-30 1.0000000
## mod03  2 3287.994 161.1360641 4.026441e-36 1.0000000
## mod16  3 3439.161 312.3037967 6.015411e-69 1.0000000
## mod04  2 3567.371 440.5134831 8.687317e-97 1.0000000

```

The AIC weights suggest that models 18 and 13 are most likely to be the best models, because they have the greatest AIC weights and their AIC weights are nearly equal. Model 18 is the negative binomial GLM with only region as a predictor; model 13 is the negative binomial GLM with only latitude as a predictor. Recall from above that latitude and region are strongly related to each other: latitude increases from the Tortugas to the Keys (FLK) and to southeast Florida (SEFCRI). So, it shouldn't be surprising that these two models were nearly equivalent as measured by AIC. Because both models 18 and 13 have nearly equivalent support, we should present the predictions of both. In lieu of "predictions" of model 18, which has only factors as predictors, you can simply present the mean and SD or 95% CI within each group.

```

# plotting values for groups (model 18)
agg1 <- aggregate(rich~region, data=dat, mean)
agg1$lat <- aggregate(lat~region, data=dat, median)$lat
agg1$sd <- aggregate(rich~region, data=dat, sd)$rich
# reorder by latitude
agg1<- agg1[order(agg1$lat),]

lo1 <- agg1$rich - agg1$sd
up1 <- agg1$rich + agg1$sd
mn1 <- agg1$rich

# predicted values for model 13 (continuous predictor)
n <- 50
px <- seq(min(dat$lat), max(dat$lat), length=n)

```

```

pred2 <- predict(mod13, newdata=data.frame(lat=px),
                  type="link", se.fit=TRUE)
mn2 <- pred2$fit
lo2 <- qnorm(0.025, mn2, pred2$se.fit)
up2 <- qnorm(0.975, mn2, pred2$se.fit)

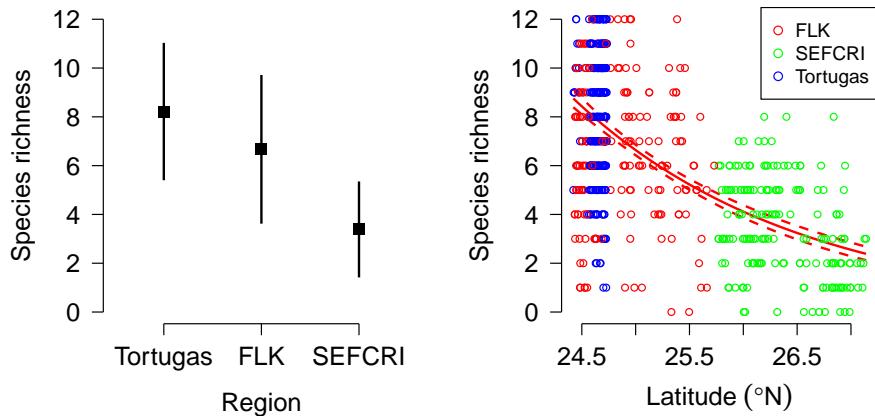
# back-transform to response scale
mn2 <- mod13$family$linkinv(mn2)
lo2 <- mod13$family$linkinv(lo2)
up2 <- mod13$family$linkinv(up2)

# define some colors for the model 13 plot
sites <- sort(unique(dat$region))
cols <- rainbow(length(sites))
dat$col <- cols[match(dat$region, sites)]

# make the plot
par(mfrow=c(1,2), bty="n", las=1, lend=1,
     mar=c(5.1, 5.1, 1.1, 1.1), cex.axis=1.3,
     cex.lab=1.3)
plot(1:3, mn1, pch=15, cex=1.3, xaxt="n",
      xlab="Region", ylim=c(0, 12), xlim=c(0.5, 3.5),
      ylab="Species richness")
segments(1:3, lo1, 1:3, up1, lwd=2)
axis(side=1, at=1:3, labels=agg1$region)

plot(px, mn2, type="n", ylab="Species richness",
      xlab="", ylim=c(0, 12))
title(xlab=expression(Latitude~(degree*N)))
points(px, lo2, type="l", lwd=2, col="red", lty=2)
points(px, up2, type="l", lwd=2, col="red", lty=2)
points(px, mn2, type="l", lwd=2, col="red")
points(dat$lat, dat$rich, col=dat$col, cex=0.8)
legend("topright", legend=sites, pch=1, col=cols)

```



The negative binomial model accounts for the fact that the data show more variability than would be expected under a Poisson distribution, and can be worked with in R much more easily than can the quasi-Poisson. The two fitted models should be presented together and compared. The authors should point out the relationship between latitude and region evident in the data. Another strategy for handling these data might be mixed models.

5.6 Logistic regression for binary outcomes

This section explores GLMs for **binary outcomes**. Binary outcomes are results that take one of two values: survival vs. non-survival, presence vs. absence, reproduction vs. not-reproducing, and so on. These data are usually coded as 1 and 0: 1 when the event occurs, and 0 when the event does not. No other values are allowed. The standard technique for modeling binary outcomes is **logistic regression**. This name refers to a GLM with a binomial family and logit link function.

Logistic regression is a technique for modeling the probability of a binary outcome: survival vs. death, reproductive vs. not reproductive, present vs. absent, etc. The response variable Y takes value 1 when the event occurs, and value 0 when the event does not occur. For each observation i , success occurs as the outcome of a Bernoulli trial with probability p_i . The logit of p_i (i.e., the logarithm of the odds that $Y = 1$) is modeled as a linear function of the predictor variables.

$$Y_i \sim \text{Bernoulli}(p_i)$$

$$\text{logit}(p_i) = \log\left(\frac{p_i}{1 - p_i}\right) = \beta_0 + \beta_1 X_i$$

The inverse link function is the **logistic function**, which is also where the name “logistic regression” comes from:

$$\text{logistic}(x) = \frac{e^x}{1 + e^x}$$

The linear predictor in logistic regression predicts the “log-odds” of an event occurring. The unit of the log-odds scale is the **logistic unit**, or **logit**. The logit link function is useful because it allows the outcome of a linear function—which can take on any real value—to be mapped smoothly to the natural domain of probabilities, the open interval $(0, 1)$. A key consequence of this link function is that increasing the value of one of the predictor variables by 1 will scale the log-odds by a constant. That constant is the coefficient estimated for the predictor.

The logistic regression model was developed in the 1940s, decades before the GLM was first proposed. Part of the development of the idea of the GLM was recognizing that many previously known models were actually special cases of a more general model. In GLM terms, we say that logistic regression is a GLM with a binomial family and logit link function.

5.6.1 Example with simulated data

As with the other GLM applications, we’ll start by simulating a suitable dataset and analyzing it. The code below will execute the simulation. Remember to set the random number seed for reproducibility.

```
set.seed(123)
n <- 30
x <- round(runif(n, 0, 12), 1)

# coefficients
beta0 <- -4
beta1 <- 0.8

# linear predictor
eta <- beta0 + beta1 * x

# logistic function (inverse link)
prob <- plogis(eta)

# draw y values (stochastic part)
y <- rbinom(n, 1, prob)

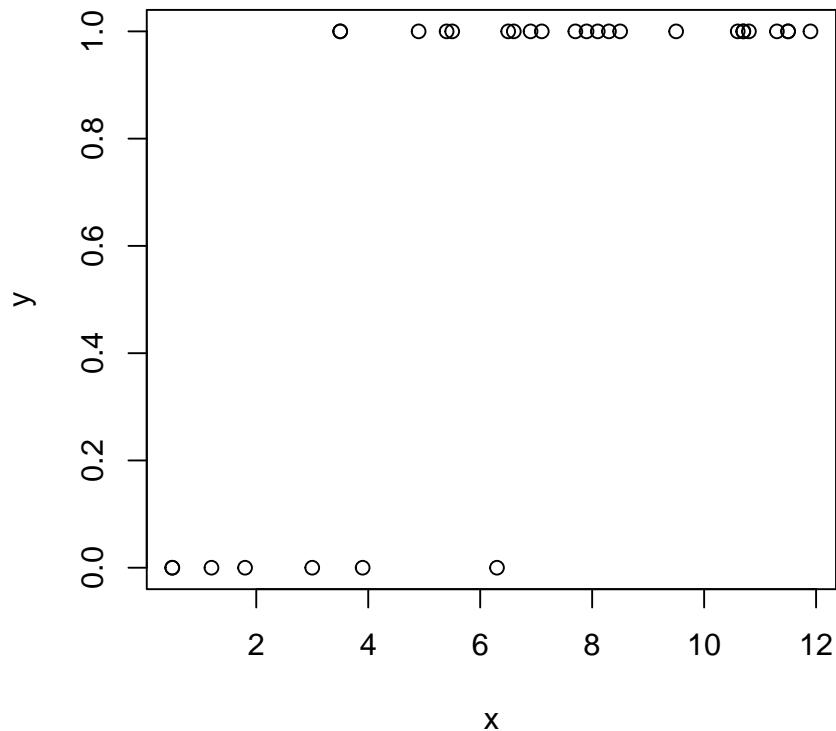
# assemble dataset
dat <- data.frame(x=x, y=y)
```

```
# take a look at the data:  
head(dat)
```

```
##      x y  
## 1 3.5 1  
## 2 9.5 1  
## 3 4.9 1  
## 4 10.6 1  
## 5 11.3 1  
## 6 0.5 0
```

Unlike other sorts of regression problems, plotting Y vs. X isn't very illuminating.

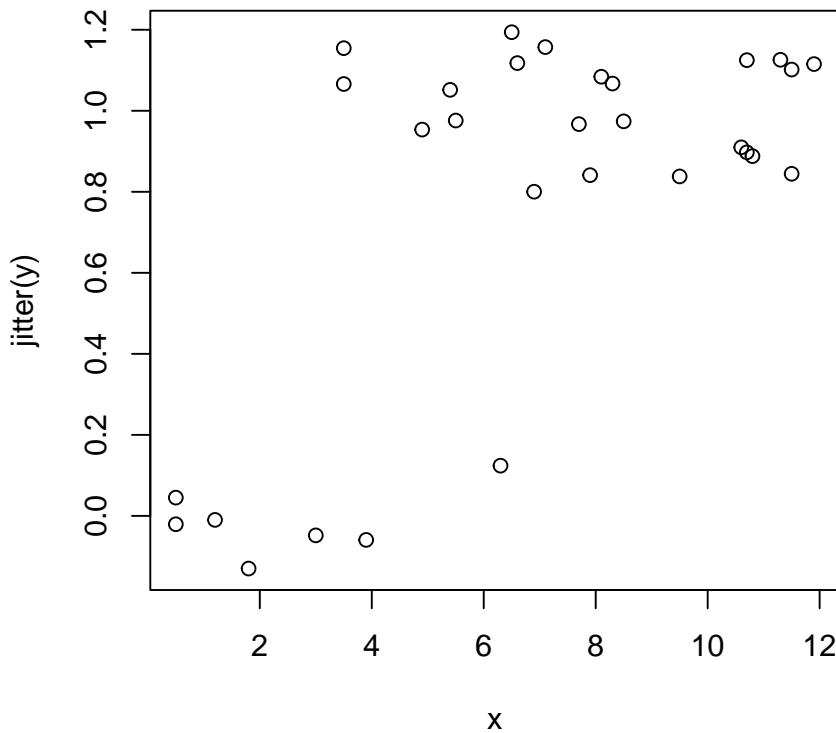
```
plot(x, y)
```



If you have a lot of duplicate coordinates, it can be hard to get a sense of how many 0s and 1s are in the dataset. The scatterplot above is also a little silly

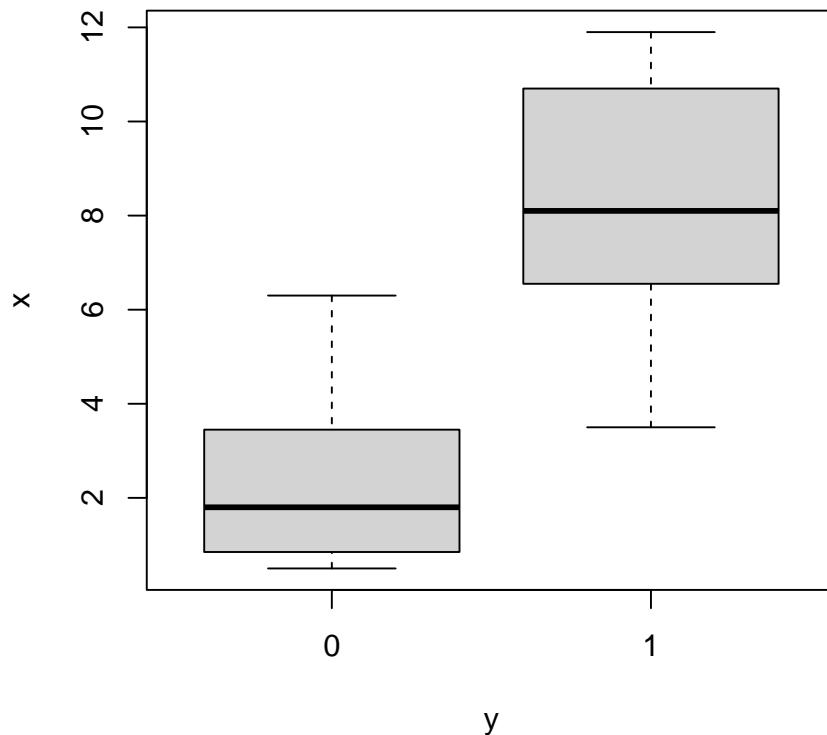
because most of the Y -axis range is not occupied. We can use function `jitter()` to add some random noise to the Y -values; this will help us see what is going on.

```
plot(x, jitter(y))
```



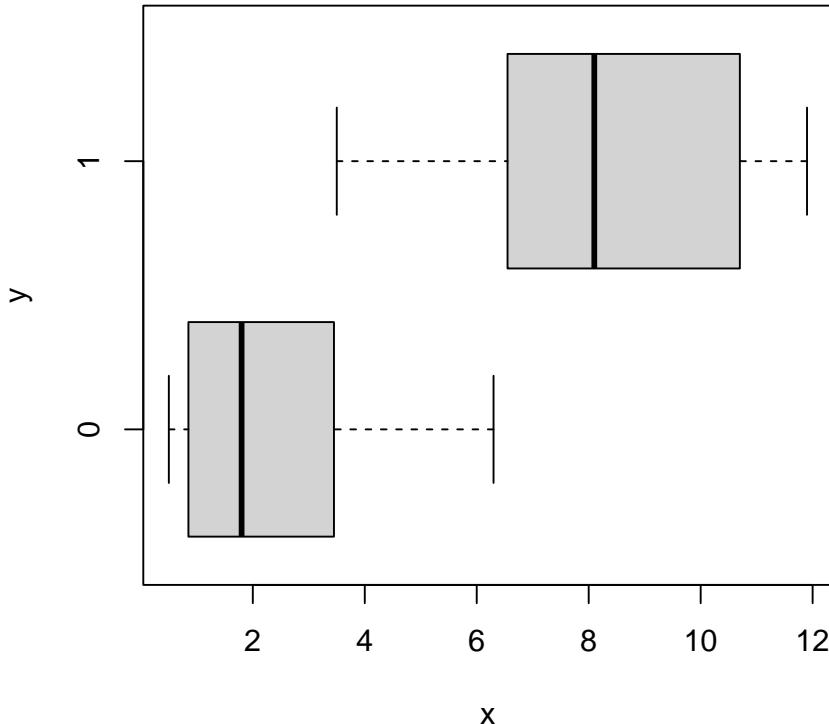
Another way to explore binary data is with a boxplot. A boxplot can show whether X tends to be smaller or greater when $Y = 1$; this is sort of the reverse of the question we are really asking, but it helps us make sense of the data as we explore.

```
boxplot(x~y)
```



The boxplot agrees with the scatterplot: Y seems more likely to occur at greater values of X . Making the boxplot horizontal makes the agreement more clear:

```
boxplot(x~y, horizontal=TRUE)
```



Logistic regression is fit using the `glm()` function. By definition, logistic regression is a GLM with a binomial family and logit link function. You don't need to specify the link function in R, as the binomial family uses the logit link function by default.

```
mod1 <- glm(y~x, data=dat, family=binomial)
summary(mod1)

##
## Call:
## glm(formula = y ~ x, family = binomial, data = dat)
##
## Deviance Residuals:
##      Min        1Q    Median        3Q       Max
## -2.25927   0.02705   0.06648   0.29492   1.30865
##
## Coefficients:
```

```

##             Estimate Std. Error z value Pr(>|z|)
## (Intercept) -3.7713     1.8349  -2.055  0.0399 *
## x           0.9909     0.4001   2.477  0.0133 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
## Null deviance: 32.596  on 29  degrees of freedom
## Residual deviance: 13.193  on 28  degrees of freedom
## AIC: 17.193
##
## Number of Fisher Scoring iterations: 7

```

The parameter estimates are pretty close to the correct values. We can calculate the pseudo- R^2 as usual:

```
1-mod1$deviance/mod1>null.deviance
```

```
## [1] 0.5952722
```

The pseudo- R^2 is pretty good for only having 30 observations. But, you should probably consider using AUC instead of pseudo- R^2 (see below). The next step is to generate and plot the model predictions. For logistic regression models, predictions should be calculated on the link scale and back-transformed to get probabilities.

```

# values for prediction
use.n <- 50
px <- seq(min(dat$x), max(dat$x), length=use.n)

# calculate predictions and 95% CI (link scale)
pred <- predict(mod1, newdata=data.frame(x=px),
                 type="link", se.fit=TRUE)
mn <- pred$fit
lo <- qnorm(0.025, mn, pred$se.fit)
up <- qnorm(0.975, mn, pred$se.fit)

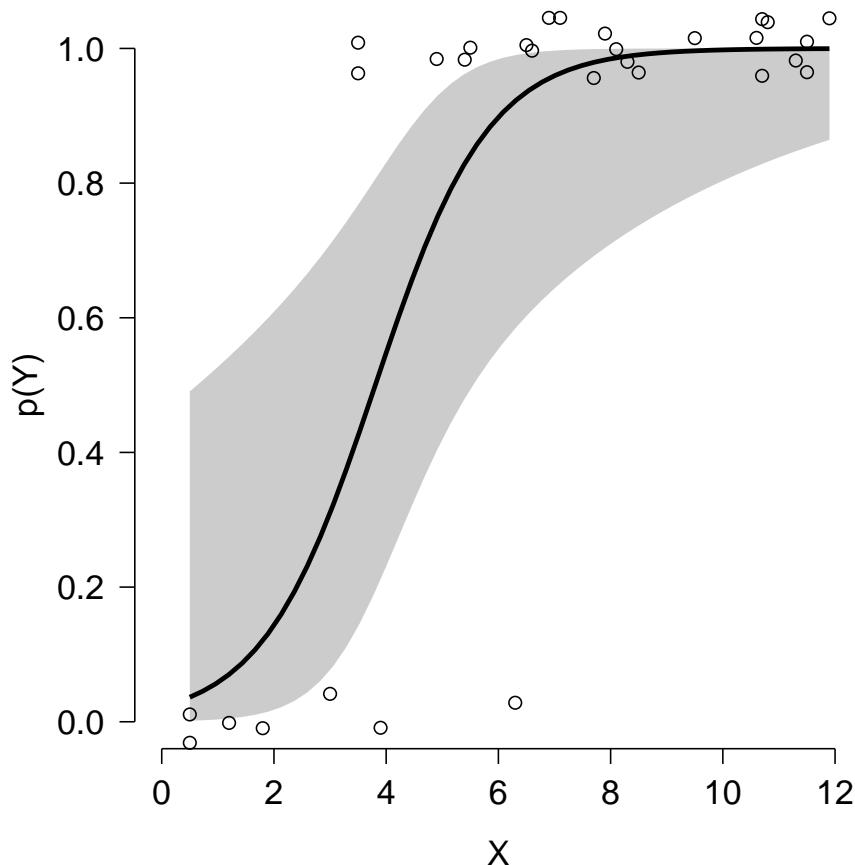
# inverse link function to get probabilities
mn <- mod1$family$linkinv(mn)
lo <- mod1$family$linkinv(lo)
up <- mod1$family$linkinv(up)

```

Now make the plot. The argument `xpd=NA` to `points()` makes sure that all points are plotted, even those that fall outside of the plot area. This is needed because `plot()` defines the plot area using the maximum Y value (1), but some of the jittered Y values will be <0 or > 1 and thus won't get plotted. We'll use a polygon instead of the usual lines to show the 95% CI of the predicted

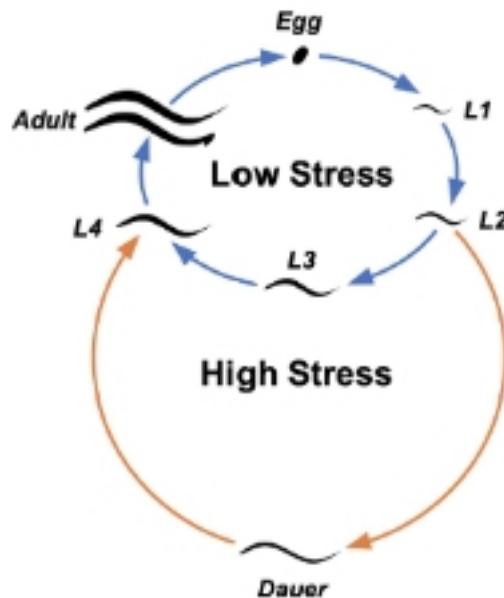
probability.

```
par(mfrow=c(1,1), mar=c(5.1, 5.1, 1.1, 1.1),
    bty="n", las=1, lend=1, cex.axis=1.3,
    cex.lab=1.3)
plot(dat$x, dat$y, type="n", xlab="X",
     ylab="p(Y)", ylim=c(0, 1), xlim=c(0, 12))
polygon(x=c(px, rev(px)),
        y=c(lo, rev(up)), # REVerse either upper or lower
        border=NA, col="grey80")
points(px, mn, type="l", lwd=3)
points(dat$x, jitter(dat$y, amount=0.05),
       cex=1.1, xpd=NA)
```



5.6.2 Example with real data

Bubrig et al. (2020) studied the influence of bacteria on the emergence of nematodes (*Caenorhabditis elegans*) from their dormant state. *C. elegans* sometimes enter a dormant state called a **dauer** that permits larvae to survive long periods of adverse conditions and to disperse long distances by passive transportation. The figure below shows the place of the dauer state in the normal *C. elegans* life cycle (Bubrig et al. 2020). Previous work had shown that successful colonization of new habitats by *C. elegans* is heavily influenced by the nearby bacterial community. The authors investigated whether the bacterial community could affect the emergence of individual worms from their dormant state.



In their experiment, they inoculated plates with one of three strains of *C. elegans* and one of five species of bacteria (or no bacteria as a control). After some time, the plates were treated with a chemical known to kill worms not in the dauer stage. The researchers then surveyed the plates and recorded each dead worm (which had emerged) as 1 and each worm still in dauer state (i.e., not emerged) as 0.

Download the dataset `bubrig_2020_data.csv` and put it in your R working directory.

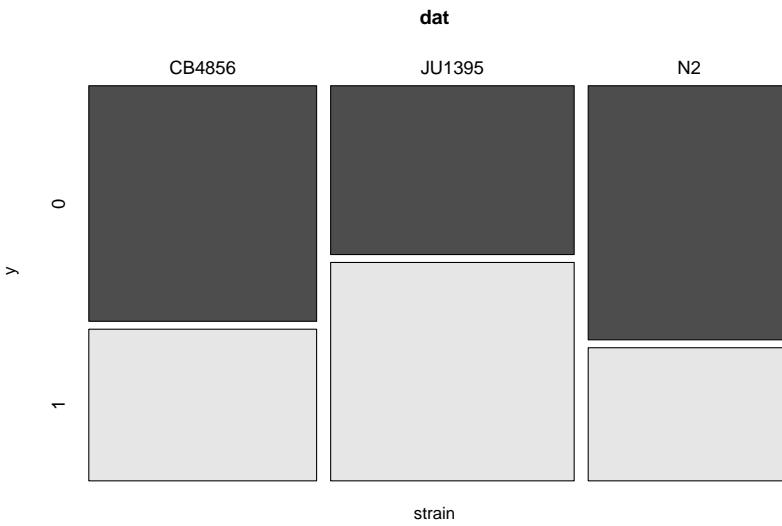
```
in.name <- "bubrig_2020_data.csv"
dat <- read.csv(in.name, header=TRUE)
head(dat)

##   trial repl strain   treat      dose y  categ
## 1     1     1      N2 Control 418.3225 1 Control
```

```
## 2     1     1     N2 Control 418.3225 1 Control
## 3     1     2     N2 Control 418.3225 1 Control
## 4     1     3     N2 Control 418.3225 1 Control
## 5     1     3     N2 Control 418.3225 1 Control
## 6     1     3     N2 Control 418.3225 1 Control
```

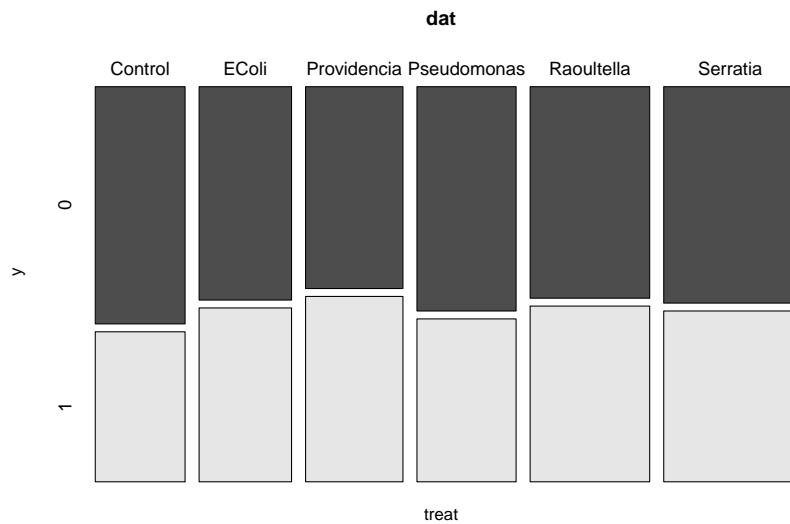
We suspect that the response variable `y` is related to the predictors `strain` (of worm) and `treat` (“treatment” of bacteria). Both of these predictor variables are factors, so scatterplots won’t be very helpful. Instead, we can try some methods for visualizing categorical data. The function `mosaicplot()` produces a plot that partitions the observations by different combinations of factors¹³:

```
mosaicplot(strain~y, data=dat, color=TRUE, cex.axis=1.1)
```



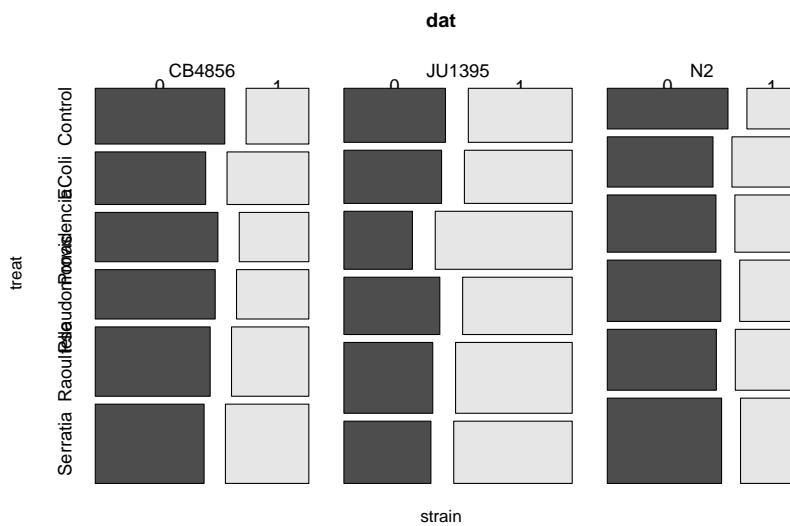
```
mosaicplot(treat~y, data=dat, color=TRUE, cex.axis=1.1)
```

¹³I have never really figured out the syntax for how `mosaicplot()` splits the data (i.e, order of terms in the formula). Try different permutations of the formula until you get the plot you want :)

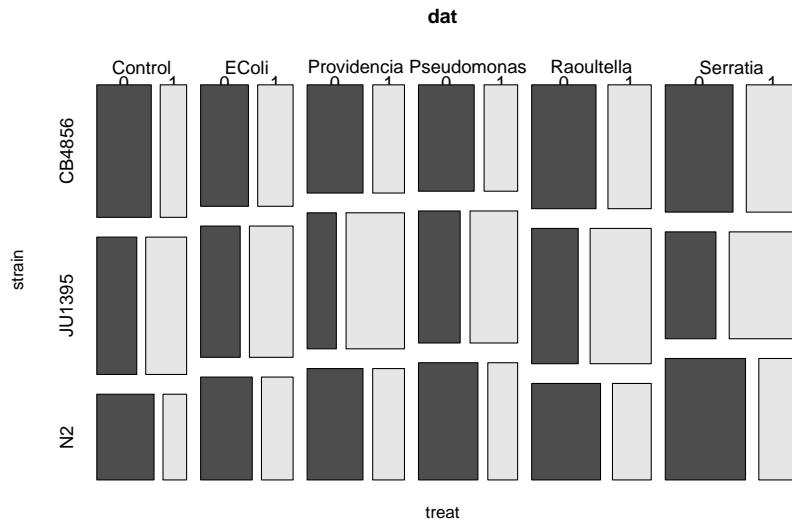


Observations can be split by an arbitrary number of factors:

```
mosaicplot(strain~treat+y, data=dat, color=TRUE, cex.axis=1.1)
```



```
mosaicplot(treat~strain+y, data=dat, color=TRUE, cex.axis=1.1)
```



As you might have guessed, **mosaic plots** are a graphical way of representing a **contingency table**. Mosaic plots are great tools for data exploration, but don't often make it into publications because they use a great deal of space and ink relative to the amount of information they convey. The equivalent contingency tables can be seen with the commands below:

```
ftable(y~strain, data=dat)
```

```
##          y    0    1
## strain
## CB4856   3925 2527
## JU1395   3007 3890
## N2       3755 1967
```

```
ftable(y~treat, data=dat)
```

```
##          y    0    1
## treat
## Control     1662 1051
## EColi       1539 1254
## Providencia 1533 1408
## Pseudomonas 1735 1260
## Raoultella   1973 1639
## Serratia     2245 1772
```

```
ftable(y~strain+treat, data=dat)

##          y   0   1
## strain treat
## CB4856 Control    681 330
##        EColi      548 406
## Providencia 571 324
## Pseudomonas 559 337
## Raoultella   752 506
## Serratia     814 624
## JU1395 Control    518 530
##        EColi      490 541
## Providencia 375 750
## Pseudomonas 519 593
## Raoultella   594 780
## Serratia     511 696
## N2 Control    463 191
##        EColi      501 307
## Providencia 587 334
## Pseudomonas 657 330
## Raoultella   627 353
## Serratia     920 452
```

Because we have a binary response variable, our first option should be logistic regression. Notice that we still call this model logistic “regression” even though there are no continuous predictors. Another name might be a “Binomial ANOVA”, in the same way that a regular ANOVA is just a special case of linear models. Just as we did in the Poisson regression examples, we will fit several models with different predictors and compare their efficacy using information theoretic methods.

For this example, we’re going to define model formulas ahead of time and fit the models in a loop. This is because the code for each model differs only in one part, the formula. Having the formulas in a separate vector and the models in a list will also make it easier to construct the AIC table for model selection. To match the original analysis done by Bubrig et al. (2020), we should treat strain N2 as the baseline. To do this, we need to re-specify the factor strain.

```
# convert to factor if not already
class(dat$strain)

## [1] "character"
# if "character", run next line:
dat$strain <- as.factor(dat$strain)
dat$strain <- relevel(dat$strain, ref="N2")
```

Now we define the model formulas and fit the models.

```
# define some model formulas
forms <- c("y~strain", "y~treat",
          "y~strain+treat", "y~strain*treat")

# make a list to hold models
mlist <- vector("list", length(forms))

# fit models in a loop
for(i in 1:length(forms)){
  mlist[[i]] <- glm(forms[i], data=dat, family=binomial)
}
```

Constructing the AIC table is made easy; all we have to do is pull AIC values from the list. Compare this to the method in the previous section, where we had to type out each model's name.

```
# construct AIC table
aicdf <- data.frame(mod=1:length(forms),
                      form=forms,
                      aic=sapply(mlist, AIC))

# calculate AIC weights
aicdf <- aicdf[order(aicdf$aic),]
aicdf$delta <- aicdf$aic - min(aicdf$aic)
aicdf$wt <- exp(-0.5*aicdf$delta)
aicdf$wt <- aicdf$wt/sum(aicdf$wt)
aicdf <- aicdf[order(-aicdf$wt),]
aicdf$cumsum <- cumsum(aicdf$wt)
aicdf

##   mod      form     aic    delta        wt cumsum
## 4   4 y~strain*treat 25356.38  0.00000 1.000000e+00      1
## 3   3 y~strain+treat 25400.32 43.93681 2.879002e-10      1
## 1   1      y~strain 25457.15 100.76847 1.313424e-22      1
## 2   2      y~treat 26114.37 757.98983 2.538633e-165      1
```

The AIC table suggests that the model with an interaction is by far the most likely to be the best-fitting model of these four. In a publication, the AIC weight for model 4 should be reported as > 0.9999 and the other models as <0.0001 , not the values shown here (the printed AIC = 1 for model 4 cannot be correct if any other model has a weight > 0 ; there is an R precision issue here). Let's check the summary table and see if that is reasonable. We should also compare the summary table to the mosaic plots. The command below shows another advantage of keeping models in a list: we can call models by their position in the ordered AIC table.

```

summary(mlist[[aicdf$mod[1]]])

##
## Call:
## glm(formula = forms[i], family = binomial, data = dat)
##
## Deviance Residuals:
##    Min      1Q  Median      3Q     Max
## -1.482  -1.014  -0.894   1.168   1.569
##
## Coefficients:
##                               Estimate Std. Error z value Pr(>|z|)
## (Intercept)                -0.88545  0.08600 -10.296 < 2e-16 ***
## strainCB4856                 0.16098  0.10906   1.476 0.139917
## strainJU1395                 0.90836  0.10589   8.578 < 2e-16 ***
## treatEColi                  0.39570  0.11247   3.518 0.000434 ***
## treatProvidencia              0.32157  0.10997   2.924 0.003453 **
## treatPseudomonas              0.19686  0.10931   1.801 0.071700 .
## treatRaoultella               0.31098  0.10873   2.860 0.004237 **
## treatSerratia                 0.17476  0.10342   1.690 0.091047 .
## strainCB4856:treatEColi      0.02885  0.14641   0.197 0.843775
## strainJU1395:treatEColi      -0.31958  0.14267  -2.240 0.025092 *
## strainCB4856:treatProvidencia -0.16375  0.14639  -1.119 0.263322
## strainJU1395:treatProvidencia  0.34868  0.14110   2.471 0.013471 *
## strainCB4856:treatPseudomonas  0.02154  0.14561   0.148 0.882395
## strainJU1395:treatPseudomonas -0.08647  0.13921  -0.621 0.534473
## strainCB4856:treatRaoultella   0.01729  0.14010   0.123 0.901753
## strainJU1395:treatRaoultella   -0.06146  0.13640  -0.451 0.652285
## strainCB4856:treatSerratia      0.28390  0.13426   2.115 0.034464 *
## strainJU1395:treatSerratia      0.11132  0.13381   0.832 0.405476
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
## Null deviance: 26159  on 19070  degrees of freedom
## Residual deviance: 25320  on 19053  degrees of freedom
## AIC: 25356
##
## Number of Fisher Scoring iterations: 4

```

In a typical ANOVA, the coefficients for each factor represent the difference in means from the baseline. In this logistic regression model, the coefficients represent differences in the log of the odds ratio. Recall that the link scale for logistic regression is the log of the odds ratio. The odds ratio of an event with probability p is $p/(1-p)$. In practical terms, this describes the relative likelihood

of an event occurring. For example, if the odds ratio for treatment *A* vs. *control* is 1.8, then the event is 1.8 times more likely to occur in treatment group *A*. Alternatively, being in treatment *A* increases the probability of the event by a factor of 1.8. The SE of the coefficient can be used to estimate a 95% CI of the odds ratio.

```
coefs <- summary(mlist[[4]])$coefficients
coefs

##                                     Estimate Std. Error      z value     Pr(>|z|)
## (Intercept)                 -0.88545363 0.08599663 -10.2963755 7.316629e-25
## strainCB4856                  0.16098397 0.10906031   1.4761005 1.399169e-01
## strainJU1395                  0.90835539 0.10589011   8.5782836 9.629926e-18
## treatEColi                     0.39569527 0.11246670   3.5183329 4.342673e-04
## treatProvidencia               0.32156980 0.10996822   2.9242066 3.453355e-03
## treatPseudomonas                0.19686226 0.10930596   1.8010204 7.169966e-02
## treatRaoultella                 0.31097514 0.10873444   2.8599508 4.237067e-03
## treatSerratia                   0.17476214 0.10341550   1.6899028 9.104656e-02
## strainCB4856:treatEColi       0.02885225 0.14640819   0.1970672 8.437750e-01
## strainJU1395:treatEColi       -0.31958315 0.14267199  -2.2399852 2.509188e-02
## strainCB4856:treatProvidencia -0.16374584 0.14638809  -1.1185735 2.633221e-01
## strainJU1395:treatProvidencia  0.34867562 0.14110387   2.4710563 1.347146e-02
## strainCB4856:treatPseudomonas  0.02154085 0.14561178   0.1479334 8.823953e-01
## strainJU1395:treatPseudomonas -0.08647351 0.13920564  -0.6211926 5.344729e-01
## strainCB4856:treatRaoultella    0.01729485 0.14009989   0.1234466 9.017535e-01
## strainJU1395:treatRaoultella   -0.06146231 0.13640390  -0.4505905 6.522847e-01
## strainCB4856:treatSerratia      0.28389752 0.13425558   2.1146050 3.446363e-02
## strainJU1395:treatSerratia      0.11131617 0.13381251   0.8318816 4.054758e-01
```

In our model, we have an interaction, so interpreting the coefficients is tricky. The interaction term between strain and treatment means that the effects of strain and the effects of treatment should not be interpreted by themselves. Let's start with the coefficients in a simpler version of the model, the one without the interaction:

Let's interpret the terms from top to bottom. The `Intercept` term describes the expectation for an observation with all factors at their baseline values (`strain = N2`, `treatment = control`). We can translate this to a probability using the inverse logit function:

```
plogis(coefs[1,1])
```

```
## [1] 0.2920489
```

So, an N2 worm in the control treatment has a 0.29 probability of emerging from dauer.

The next two coefficients, `strainCB4856` and `strainJU1395`, describe the odds ratios of animals from the other worm strains, in the control group. The odds

ratios for emerging from dauer while being in one of those strains instead of the baseline strain are obtained by exponentiating the estimates.

```
exp(coefs[2:3,1])

## strainCB4856 strainJU1395
##      1.174666    2.480240
```

This means that a CB4856 worm in the control treatment would be 1.17 times more likely to emerge than an N2 worm in the control group, and a JU1395 worm in the control treatment would be 2.48 times as likely to emerge as an N2 worm in the control group. The 95% CI for those odds ratios are:

```
exp(coefs[2:3,1]-1.96*coefs[2:3,2])

## strainCB4856 strainJU1395
##      0.9485942    2.0153861
exp(coefs[2:3,1]+1.96*coefs[2:3,2])

## strainCB4856 strainJU1395
##      1.454616    3.052314
```

So, within the control treatment, CB4856 worms are 1.17 times more likely to emerge, and we are 95% confident that the odds ratio is in the interval [0.948, 1.455].

The next set of coefficients, `treatEColi`, `treatProvidencia`, `treatPseudomonas`, `treatRaoultella`, and `treatSerratia`, are the effects of the bacterial treatments for worms in the control strain N2. So, N2 worms exposed to *E. coli* are 1.485 times as likely to emerge as control worms, with a 95% CI of [1.19, 1.85].

```
exp(coefs[4,1])

## [1] 1.485417
exp(coefs[4,1]-1.96*coefs[4,2])

## [1] 1.191557
exp(coefs[4,1]+1.96*coefs[4,2])

## [1] 1.851748
```

Now that we understand what the main effects represent, we need to understand what the main effects are not. The coefficients for the main effects (i.e., effects that are not part of the interaction) are the effects of changing one factor with the other factor held at its baseline. The main effects coefficients are ***NOT*** the overall effect of any factor. Talking about the “overall effect of a factor” or “independent effect of a factor” doesn’t even make sense in the presence of an interaction. This is because the effect of each factor affects the effects of the other. *This is the definition of an interaction*. In our example, the coefficient

0.16098397 estimated for strainCB4856 applies only for worms in the baseline strain, not for worms in other strains.

The interaction coefficients describe the changes in odds ratio for observations when both the strain and treatment factors are different from their baselines. For example, `strainCB4856:treatEColi` is the effect of worm strain CB4856 and treatment *E. coli* relative to worm strain N2 and the control treatment: 1.02 times, 95% CI [0.77, 1.37].

```
exp(coefs[9,1])

## [1] 1.029273
exp(coefs[9,1]-1.96*coefs[9,2])

## [1] 0.7725119
exp(coefs[9,1]+1.96*coefs[9,2])

## [1] 1.371373
```

We can use R to construct a table of the odds ratios for each treatment; this table can be presented as a result or used to build a figure.

```
coefs <- summary(mlist[[4]])$coefficients
coefs <- data.frame(coefs)
coefs <- coefs[,c(1,2,4)]
names(coefs) <- c("est", "se", "p")

coefs$strain <- "N2"
coefs$strain[grep("CB4856", rownames(coefs))] <- "CB4856"
coefs$strain[grep("JU1395", rownames(coefs))] <- "JU1395"

treats <- mlist[[4]]$xlevels$treat
coefs$trt <- "Control"
for(i in 1:length(treats)){
  flag <- grep(treats[i], rownames(coefs))
  coefs$trt[flag] <- treats[i]
}

# reorder using order from original factors
coefs$strain <- factor(coefs$strain, levels=mlist[[4]]$xlevels$strain)
coefs$trt <- factor(coefs$trt, levels=treats)
coefs <- coefs[order(coefs$strain, coefs$trt),]

# calculate odds ratios
coefs$or <- exp(coefs$est)
coefs$lo <- coefs$or - 1.96*coefs$se
coefs$up <- coefs$or + 1.96*coefs$se
```

```

# significant?
coefs$sig <- ifelse(coefs$p < 0.05, 1, 0)

# clean up
res <- coefs
rownames(res) <- NULL
res <- res[,-c(1:3)]

# by definition:
res$or[1] <- 1
res$lo[1] <- 1
res$up[1] <- 1
res$sig[1] <- NA

# admire your handiwork:
res

##   strain      trt      or      lo      up sig
## 1     N2 Control 1.0000000 1.0000000 1.000000  NA
## 2     N2    EColi 1.4854166 1.2649819 1.705851  1
## 3     N2 Providencia 1.3792913 1.1637536 1.594829  1
## 4     N2 Pseudomonas 1.2175763 1.0033366 1.431816  0
## 5     N2 Raoultella 1.3647553 1.1516358 1.577875  1
## 6     N2 Serratia 1.1909629 0.9882685 1.393657  0
## 7 CB4856 Control 1.1746661 0.9609079 1.388424  0
## 8 CB4856    EColi 1.0292725 0.7423125 1.316233  0
## 9 CB4856 Providencia 0.8489578 0.5620371 1.135878  0
## 10 CB4856 Pseudomonas 1.0217745 0.7363754 1.307174  0
## 11 CB4856 Raoultella 1.0174453 0.7428495 1.292041  0
## 12 CB4856 Serratia 1.3282968 1.0651559 1.591438  1
## 13 JU1395 Control 2.4802402 2.2726955 2.687785  1
## 14 JU1395    EColi 0.7264518 0.4468147 1.006089  1
## 15 JU1395 Providencia 1.4171894 1.1406258 1.693753  1
## 16 JU1395 Pseudomonas 0.9171598 0.6443168 1.190003  0
## 17 JU1395 Raoultella 0.9403884 0.6730367 1.207740  0
## 18 JU1395 Serratia 1.1177482 0.8554757 1.380021  0

```

We can use the table `res` to make a nice figure that distills the entire model down to the effect sizes (i.e., odds ratios):

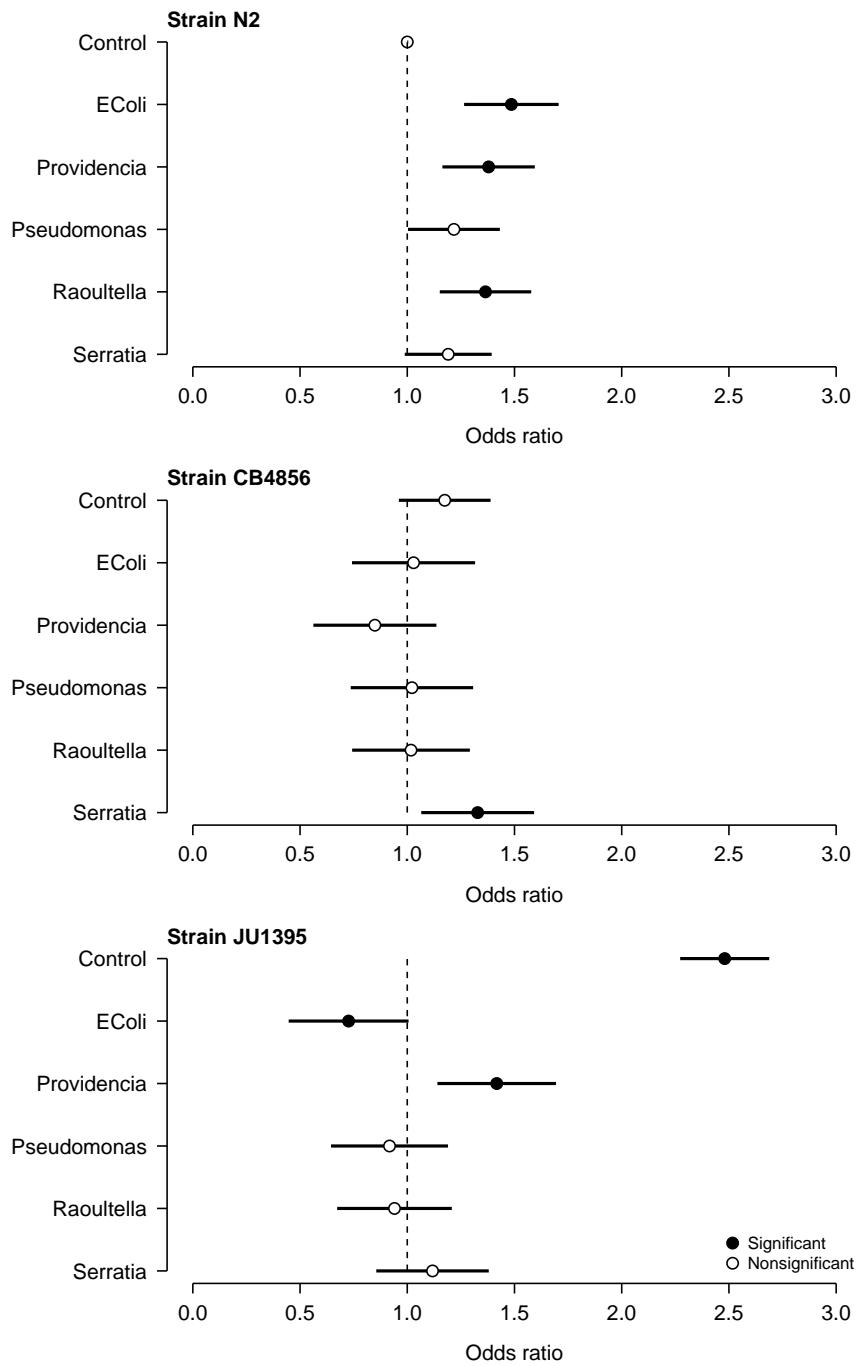
```

use.y <- length(treats):1
strains <- as.character(unique(res$strain))
res$col <- ifelse(res$sig == 1, "black", "white")

par(mfrow=c(3, 1), mar=c(5, 10, 1, 1), bty="n",
  las=1, lend=1, cex.lab=1.2, cex.axis=1.3)

```

```
for(i in 1:length(strains)){
  flag <- which(res$strain == strains[i])
  plot(res$or[flag], use.y, type="n",
       xlab="Odds ratio", yaxt="n",
       xlim=c(0, 3), ylab="")
  axis(side=2, at=use.y, labels=treats)
  segments(res$lo[flag], use.y,
           res$up[flag], use.y, lwd=2)
  title(main=paste("Strain", strains[i]), adj=0)
  segments(1, 1, 1, length(treats), lty=2)
  points(res$or[flag], use.y, cex=1.5,
         pch=21, bg=res$col[flag])
}
legend("bottomright",
       legend=c("Significant", "Nonsignificant"),
       pch=21, pt.bg=c("black", "white"),
       pt.cex=1.5, bty="n")
```

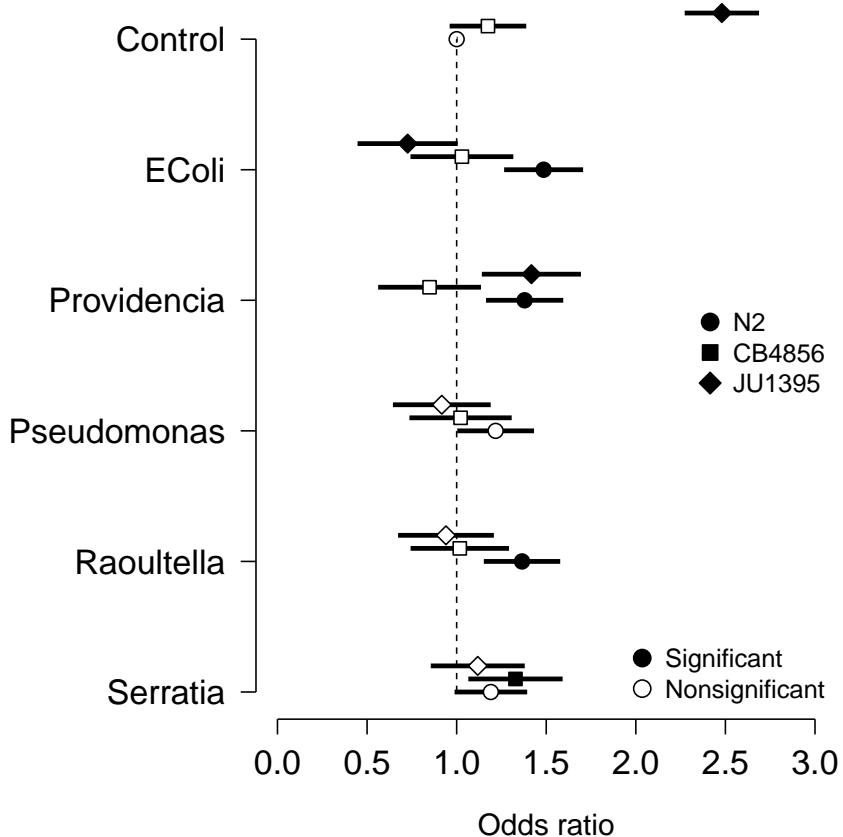


Or even better, combine the three strains onto a single panel. You can adjust the Y coordinate offsets (`res$yoff`) and other inputs to fine-tune the plot.

```
shps <- 21:23
matchx <- match(res$strain, unique(res$strain))
res$shape <- shps[matchx]
res$yoff <- matchx-1
res$y <- use.y[match(res$trt, unique(res$trt))]
res$y <- res$y + res$yoff/10

par(mfrow=c(1, 1), mar=c(5, 10, 1, 1), bty="n",
    las=1, lend=1, cex.lab=1.2, cex.axis=1.3)
plot(res$or, res$y, type="n",
     pch=res$shape, cex=1.3,
     bg=ifelse(res$sig == 1, "black", "white"),
     xlab="Odds ratio", yaxt="n",
     xlim=c(0, 3), ylab="")
segments(1, 1, 1, 6, lty=2)
segments(res$lo, res$y, res$up, res$y, lwd=3)
points(res$or, res$y,
       pch=res$shape, cex=1.3,
       bg=ifelse(res$sig == 1, "black", "white"))
axis(side=2, at=use.y, labels=unique(res$trt))

legend("bottomright",
       legend=c("Significant", "Nonsignificant"),
       pch=21, pt.bg=c("black", "white"),
       pt.cex=1.5, bty="n")
legend("right", legend=unique(res$strain),
       pch=shps, bty="n", pt.cex=1.5,
       pt.bg="black")
```



5.6.3 Logistic GLM diagnostics: AUC and ROC

Logistic regression is one way of solving what statisticians sometimes call a **classification problem**, where we want to use some set of predictor variables to classify observations into one category or another. The binary case is the simplest classification problem. The predictions of a logistic regression model will predict each observation to either be 0 or 1. We can use those predictions, and how often they are correct, to evaluate the efficacy of a logistic regression model.

First let's define a function to get the predictions for a logistic regression model for its input data, with user-specified "cut-off" value that determines how great the predicted probability needs to be before declaring the prediction a 1 instead of a 0. The default, 0.5, means that any observation with a predicted probability >0.5 will be counted as 1.

```

# function to get prediction:
pred.fun <- function(mod, data, res = "y",
                      pos = 1, neg = 0, cut = 0.5) {
  probs <- predict(mod, newdata = data, type = "response")
  ifelse(probs > cut, pos, neg)
}

# test it out:
p1 <- pred.fun(mlist[[4]])
ftable(p1~dat$y)

##      p1    0    1
## dat$y
## 0      7680 3007
## 1      4494 3890

```

This contingency table is called a **confusion matrix** and it summarizes how often the model is correct for a given cutoff value. Here, when the model predicts a success ($y = 1$), it is correct for 3890 out of $(3890+3007) = 6897$ observations. Confusion matrices are usually summarized by two measures, sensitivity and specificity.

Sensitivity measures what proportion of successes were correctly identified. This is also called the **true positive rate**. Sensitivity answer the question, “What proportion of observations predicted to have $y = 1$ were actually 1? Sensitivity is calculated as:

$$\text{Sensitivity} = \frac{n(\text{true positives})}{n(\text{true positives}) + n(\text{false negatives})}$$

In our example, the sensitivity is $3890 / (3890 + 4494) = 0.463$.

Specificity measures what proportion of failures ($y = 0$) were correctly identified. It is also called the **true negative rate**. It is calculated as:

$$\text{Specificity} = \frac{n(\text{true negatives})}{n(\text{true negatives}) + n(\text{false positives})}$$

In our example, the specificity was $7680 / (7680 + 3007) = 0.718$.

The characteristics of the study system, the dataset, and the analysis can affect the sensitivity and the specificity of a model. If correctly identifying true positives is important (e.g., in disease screenings), then a model with greater sensitivity is preferred. If correctly identifying true negatives is important, then a model with greater specificity is preferred. The sensitivity and specificity can be varied by changing the threshold for 1 vs. 0.

A **receiver operating characteristic (ROC) curve** shows how sensitivity and specificity covary at different cut-off thresholds. We can construct an ROC curve by summarizing the confusion matrix at different cut-offs. The code below shows how to do this manually, so you can see for yourself how AUC is calculated. Afterwards, we'll use functions from an add-on package to save some work.

```
# vector of cut offs to test
co <- seq(0, 1, by=0.01)

# vectors to hold specificity and sensitivity
sen <- spe <- numeric(length(co))

# calculate sensitivity and specificity in a loop
for(i in 1:length(co)){
  # predictions for current cutoff value
  pred <- pred.fun(mlist[[4]], cut=co[i])

  # rearrange to data frame
  con.mat <- data.frame(ftable(dat$y~pred))
  con.mat$pred <- as.numeric(as.character(con.mat$pred))
  con.mat$dat.y <- as.numeric(as.character(con.mat$dat.y))

  # special case: no predicted 1s
  if(! 1 %in% con.mat$pred){
    add.df <- data.frame(pred=1, dat.y=0:1, Freq=0)
    con.mat <- rbind(con.mat, add.df)
  }

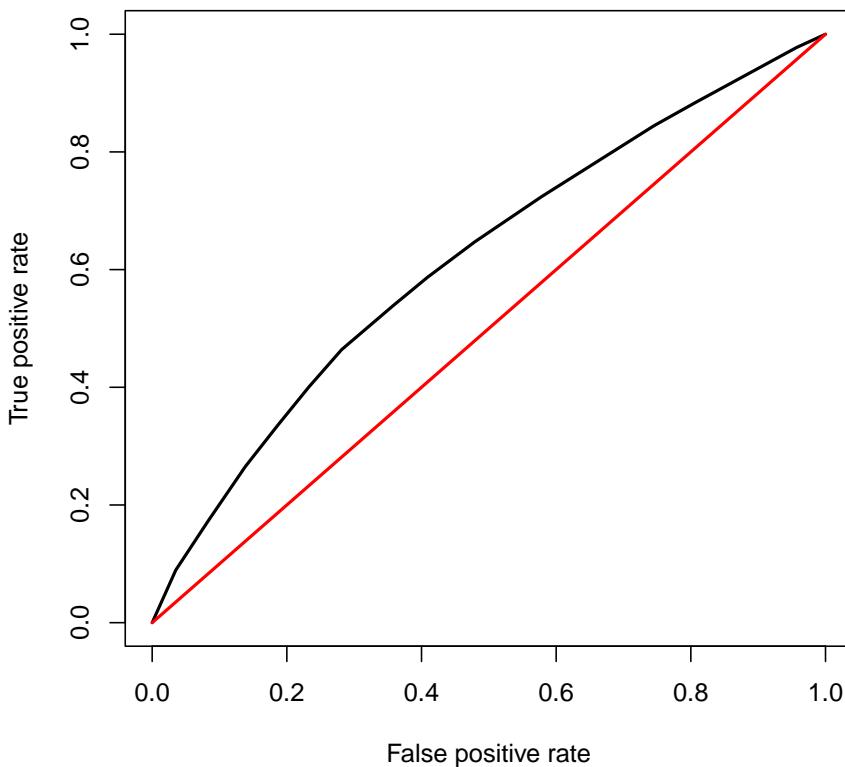
  # special case: no predicted 0s
  if(! 0 %in% con.mat$pred){
    add.df <- data.frame(pred=0, dat.y=0:1, Freq=0)
    con.mat <- rbind(con.mat, add.df)
  }

  # sensitivity:
  tp <- which(con.mat$pred == 1 & con.mat$dat.y == 1)
  fn <- which(con.mat$pred == 0 & con.mat$dat.y == 1)
  sen[i] <- con.mat$Freq[tp] / (
    con.mat$Freq[tp] + con.mat$Freq[fn])

  # specificity:
  tn <- which(con.mat$pred == 0 & con.mat$dat.y == 0)
  fp <- which(con.mat$pred == 1 & con.mat$dat.y == 0)
  spe[i] <- 1-con.mat$Freq[tn] /
    (con.mat$Freq[tn] + con.mat$Freq[fp])
}

}
```

```
plot(spe, sen, type="l", lwd=2,
      xlim=c(0,1), ylim=c(0,1),
      xlab="False positive rate",
      yla="True positive rate")
segments(0, 0, 1, 1, col="red", lwd=2)
```

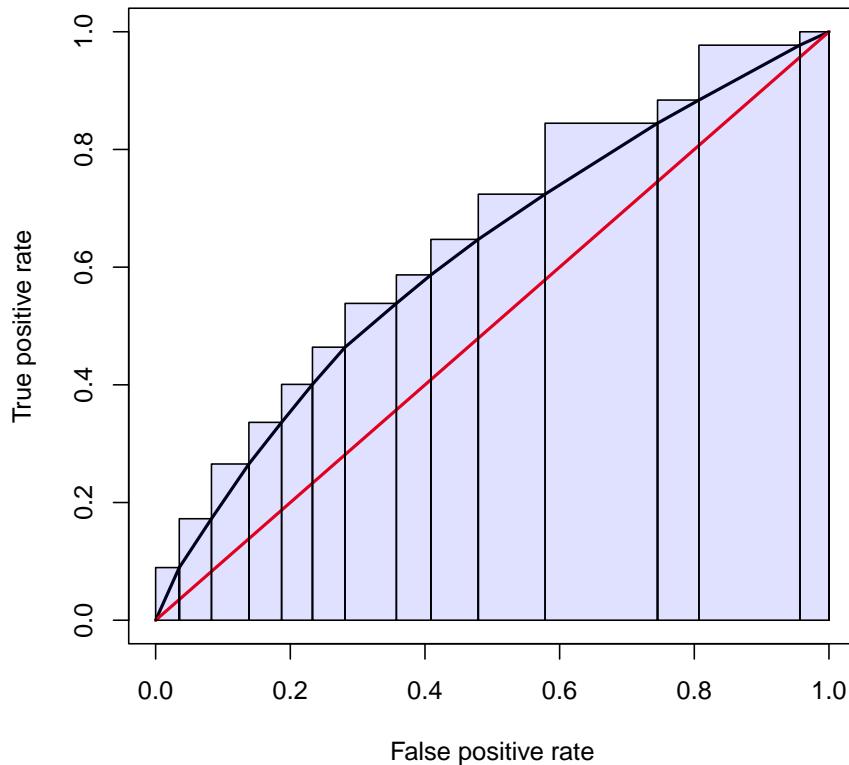


The diagonal red line shows the case of a model where predictions of 1 are just as likely to be true positives as false positives. Obviously, the curve for your model should fall above this line! The curve for a well-fitting model should increase quickly as false positive rate increases from 0, and reach a y value of 1 very quickly. A model that perfectly classifies observations should have an area under the curve (AUC) of 1 (can you see why?). A model that predicts an equal number of true positives and false positives has an AUC of 0.5. This represents a model that is no better than random chance at being correct, and is the “worst case scenario” for a classification model like logistic regression. We

can approximate the AUC using rectangles:

```
plot(spe, sen, type="l", lwd=2,
      xlim=c(0,1), ylim=c(0,1),
      xlab="False positive rate",
      yla="True positive rate")
segments(0, 0, 1, 1, col="red", lwd=2)

# add to previous plot:
rect(spe[-length(spe)], 0,
      spe[-1], sen, border="black",
      col="#0000FF20")
```



And the numerical integration (lazy way):

```
auc1 <- sum((diff(spe)*-1) * sen[-1])
auc2 <- sum((diff(spe)*-1) * sen[-length(sen)])
```

```
(auc1+auc2)/2
```

```
## [1] 0.6172765
```

The AUC of 0.617 is not too bad, but not great either. Remember that 0.5 is model no better than a coin flip, and 1 is perfect. As much fun as it is to manually calculate ROC curves and definite integrals, we can get the ROC and AUC much more quickly using the functions in package `pROC`.

```
library(pROC)
```

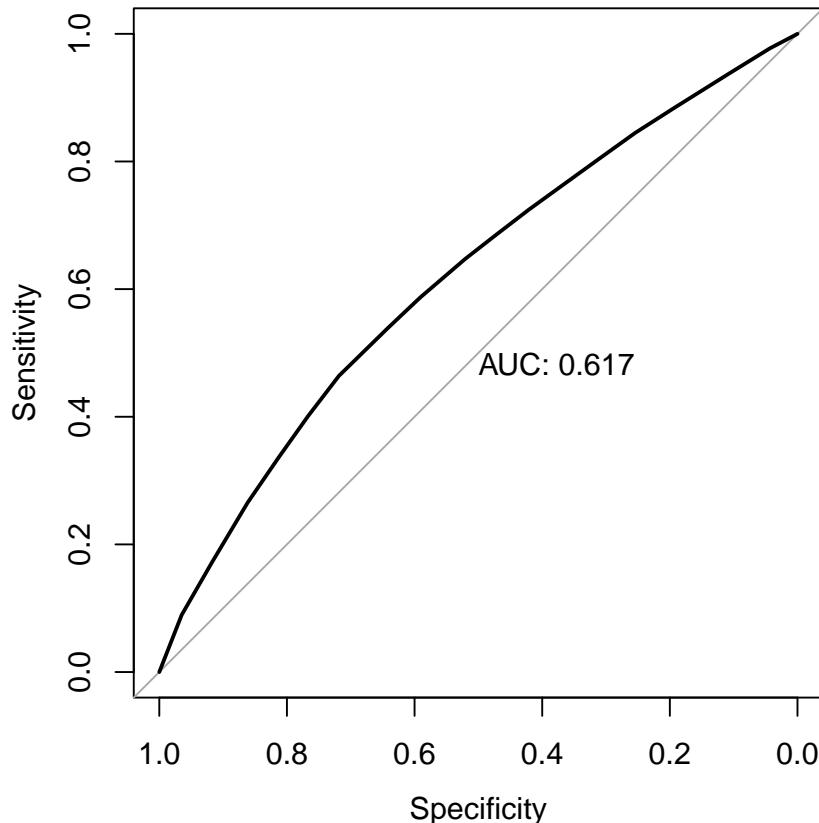
```
## Type 'citation("pROC")' for a citation.
```

```
##  
## Attaching package: 'pROC'
```

```
## The following objects are masked from 'package:stats':  
##  
## cov, smooth, var  
# generate predictions (probabilities)  
p1 <- predict(mlist[[4]], type="response")  
roc1 <- roc(dat$y ~ p1, plot = TRUE, print.auc = TRUE)
```

```
## Setting levels: control = 0, case = 1
```

```
## Setting direction: controls < cases
```

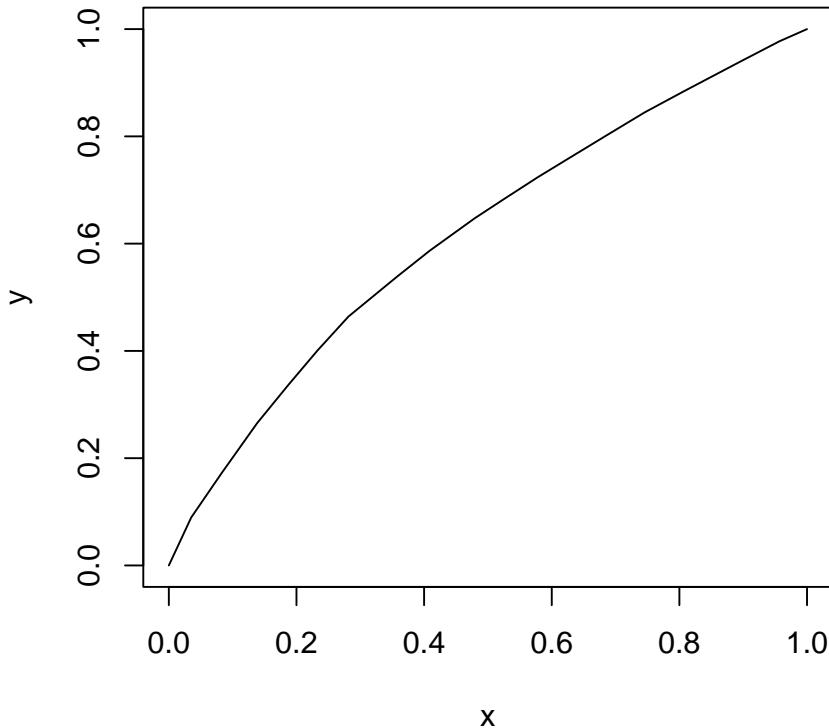


```
roc1

##
## Call:
## roc.formula(formula = dat$y ~ p1, plot = TRUE, print.auc = TRUE)
##
## Data: p1 in 10687 controls (dat$y 0) < 8384 cases (dat$y 1).
## Area under the curve: 0.6173
```

The AUC calculated by package `pROC` (0.6173) is close to the value we calculated by hand. If you don't like the default figure produced by `roc()`, you can extract the sensitivity and specificity scores and make your own plot.

```
# 1- to orient axis correct way:
x <- 1-roc1$specificities
y <- roc1$sensitivities
plot(x,y, type="l")
```



5.7 Binomial GLM for proportional data

This module explores GLMs for **proportional data**. Proportional data describe the number of times that some event occurs, relative to the total number of times it could occur. Many phenomena that biologists study are expressed as proportions: proportion of individuals surviving, proportion of individuals with trait A , proportion of area covered by species X , and so on. Analyzing proportions is a lot like analyzing binary outcomes; i.e., logistic regression. In fact, they are both GLMs with a binomial family. All that differs is the n term. In logistic regression the interest is in predicting the outcomes of single trials, while in binomial regression the interest is in predicting the **proportion** of observations in a certain state. Any dataset suitable for logistic regression can be converted to a dataset suitable for binomial regression. How you should analyze your data depends on your question:

- If the question is about outcomes at the individual observation level, then

- use logistic regression.
- If the question is about rates or probabilities of an event occurring, then use binomial regression.

One good clue is to consider the level at which predictor variables are measured. If many observations share the same value of a predictor variable (or the same values of many predictor variables), then binomial regression might be the right approach because the sample unit is really the group. For example, if you want to model nestling survival in birds, all of the chicks within a nest would have the same values of any predictor variable that applied to the nest. Thus, binomial regression would be appropriate. If, however, there were chick-level predictors, then logistic regression would be appropriate. Think about what level of organization your predictor and response variables occur at, and what question you are really asking.

5.7.1 Binomial GLM

Binomial regression or **binomial GLM** is related to logistic regression. Both are models for binary outcomes. Binary outcomes are those resulting from **trials** in which some event can either occur or not. When the event occurs, the trial is counted as a **success**. For example, if you flip a fair coin one time, that is one trial. The coin will come up heads with probability 0.5. If you flip the coin 10 times and get heads 4 times, then your observation (10 coin flips) had 10 trials (each flip) and 4 successes (times that heads came up). In logistic regression, each observation represents the outcome of one and only one trial (i.e., $n = 1$). In binomial regression, each observation can represent the outcome of many trials. This means that logistic regression is a special case of binomial regression.

Binary data can be represented in more than one way, so long as the number of trials and successes are known. Consider a study where researchers want to model the probability of a fish spawning in relation to environmental conditions. Their data will consist of the number of female fish in a population that spawn and the total number of female fish in the population.

- The number of **trials (n)** is the number of fish that could have spawned.
- The number of **successes** is the number of fish that spawned.
- The **probability of success (p)** is ratio of successes to trials; in this case, the number of spawning fish divided by the number of potential spawners.

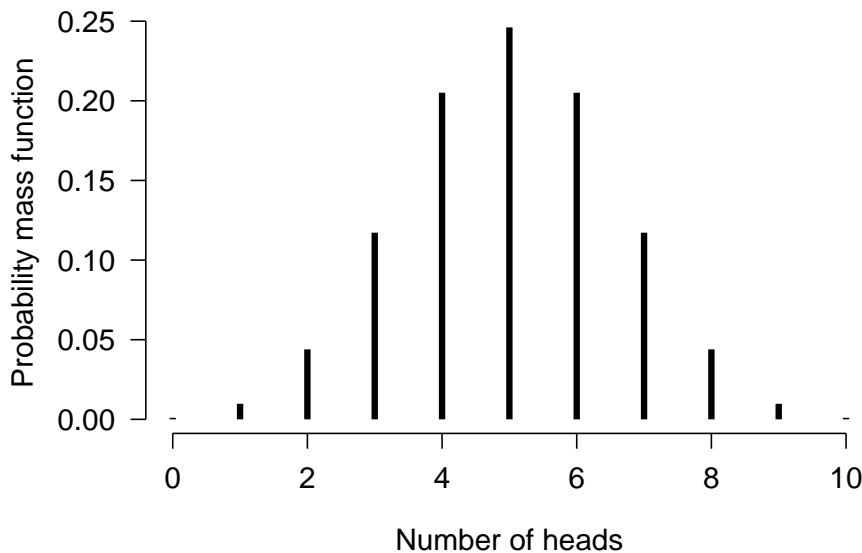
If the same number of fish were sampled in each stream reach, then the researchers could use the number of spawning fish as their response variable (i.e., number of successes). If the number of fish varied between observations, then they should use the proportion of spawners (p) as their response variable. Both ways of expressing the data are correct, but one may be more useful.

Binary data are usually modeled as coming from a **binomial distribution**. The binomial distribution describes the number of successes, or times an event occurs, in n trials with probability p in each trial. For example, if you flip a fair

coin 10 times, the number of heads n_{heads} will follow the distribution:

$$n_{heads} \sim Binomial(10, 0.5)$$

The distribution of outcomes is shown below:



Proportional data can be expressed in two ways: as the number of successes observed out of n trials, or as the proportion of successes (i.e., p). This proportion can be thought of as the **probability** that a success will occur. Both ways emphasize different components of the binomial distribution that generated the data.

When analyzing proportional data, we have the option of using one of two equivalent representations of the model.

Form 1: Response variable as number of occurrences

The first form frames the model in terms of the number of successes:

$$y_i \sim Binomial(n_i, p_i)$$

$$\text{logit}(p_i) = \beta_0 + \beta_1$$

In these equations,

- y_i is the number of trials in observation i in which some event occurs.

- n_i is the number of trials included in observation i .
- p_i is the probability of some event occurring in observation i .
- β_0 and β_1 are the regression intercept and slope.

The response variable y_i in this form is a non-negative integer. For example, if a toxicity trial exposes midges to 6 different concentrations of a toxin, and 20 organisms are exposed per concentration, then the number of organisms in each group that dies would be the response variable.

Form 2: Response variable as proportion of successes

The second way a binomial GLM can be expressed is in terms of the proportion of trials in which an event occurs:

$$y_i = \frac{z_i}{n_i}$$

$$z_i \sim \text{Binomial}(n_i, p_i)$$

$$\text{logit}(p_i) = \beta_0 + \beta_1$$

In this form:

- y_i is the proportion of trials in observation i in which some event occurs.
- z_i is the number of trials in observation i in which some event occurs.
- n_i is the number of trials included in observation i .
- p_i is the probability of some event occurring in observation i .
- β_0 and β_1 are the regression intercept and slope.

The response variable y_i in this form is a proportion in the interval $[0, 1]$, and represents the probability p_i . For example, if a toxicity trial exposes midges to 6 different concentrations of a toxin, and 20 organisms are exposed per concentration, then the proportion of organisms in each group that dies would be the response variable.

You can think of your binomial regression model in either format. Notice that the deterministic part of the model is exactly the same in both forms of the binomial GLM, and the same as in logistic regression. The stochastic part is also the same in both forms of the binomial GLM. The stochastic part of logistic regression is usually written with the Bernoulli distribution, but could be rewritten as $y_i \sim \text{Binomial}(1, p_i)$ to emphasize its status as a special case of binomial regression.

If you have proportional data, the proportion version (form 2) is more straightforward to fit in R. The regression coefficients will be the same no matter which format you use. The examples below use the proportion format.

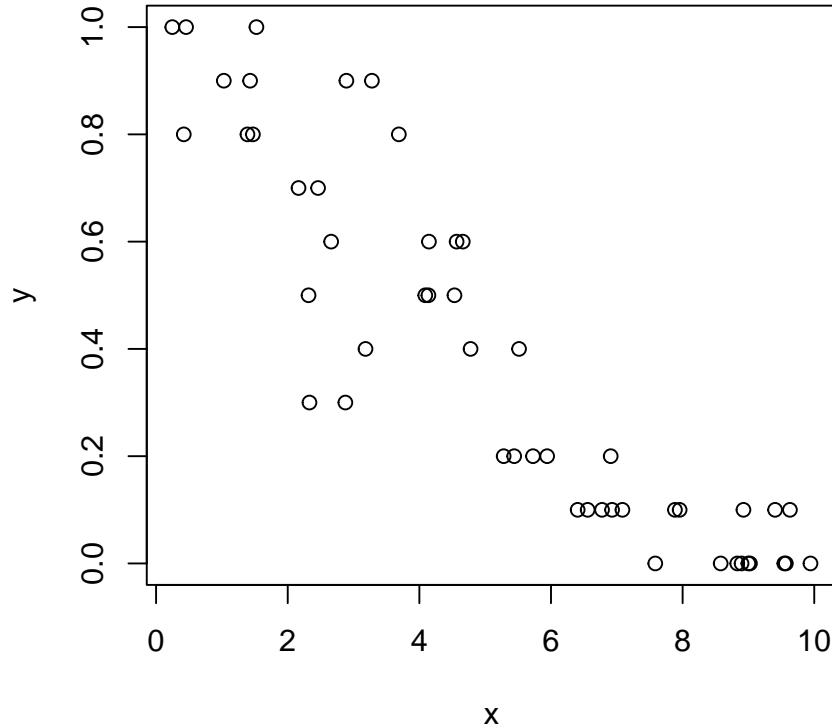
5.7.2 Example with simulated data

The example below shows how to simulate and fit a binomial GLM on proportions. First, let's simulate the data.

```
set.seed(123)
n <- 50
x <- runif(n, 0, 10)
z <- 3 + -0.75*x + rnorm(n, 0, 1)
# inverse logit
y <- round(plogis(z), 1)

# weights = sample sizes
# use same size for each observation for simplicity
wts <- rep(10, n)

# plot of a proportion (y) vs. a predictor (x)
plot(x, y)
```



We use GLM to fit the binomial model. The command below will specify a binomial GLM with the default logit link. In another context this is how we fit a logistic regression. Adding the argument `weights`, which defines the sample sizes represented by each probability (Y value) converts the model to a binomial regression. For example, an observation with $Y = 0.3$ and `weights = 10` represents a trial or group of observations with 3 successes in 10 trials. There are other ways of specifying these weights (see `?family`) but this is the simplest. Note that if you try to do this analysis without supplying weights, R will return a warning (because it expects 1s and 0s for logistic regression) and will not calculate the deviance correctly (because it doesn't know how many trials each probability represents).

```
mod1 <- glm(y~x, family=binomial, weights=wts)
summary(mod1)
```

```
##  
## Call:
```

```

## glm(formula = y ~ x, family = binomial, weights = wts)
##
## Deviance Residuals:
##      Min       1Q   Median       3Q      Max
## -2.8868  -0.7281  -0.2132   0.7710   2.0987
##
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept) 2.52369   0.26503  9.522 <2e-16 ***
## x          -0.63974   0.05588 -11.449 <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
## Null deviance: 288.207 on 49 degrees of freedom
## Residual deviance: 57.294 on 48 degrees of freedom
## AIC: 150.55
##
## Number of Fisher Scoring iterations: 4

```

The parameter estimates are close to the true values and the pseudo- R^2 is pretty good:

```

1-mod1$deviance/mod1>null.deviance
## [1] 0.8012043

```

Let's generate predicted values and plot them in the usual way:

```

# values for prediction
n <- 50
px <- seq(min(x), max(x), length=n)

# calculate predictions on link scale
pred1 <- predict(mod1,
                  newdata=data.frame(x=px),
                  type="link", se.fit=TRUE)
mn1 <- pred1$fit
lo1 <- qnorm(0.025, mn1, pred1$se.fit)
up1 <- qnorm(0.975, mn1, pred1$se.fit)

# backtransform with inverse link function
mn1 <- mod1$family$linkinv(mn1)
lo1 <- mod1$family$linkinv(lo1)
up1 <- mod1$family$linkinv(up1)

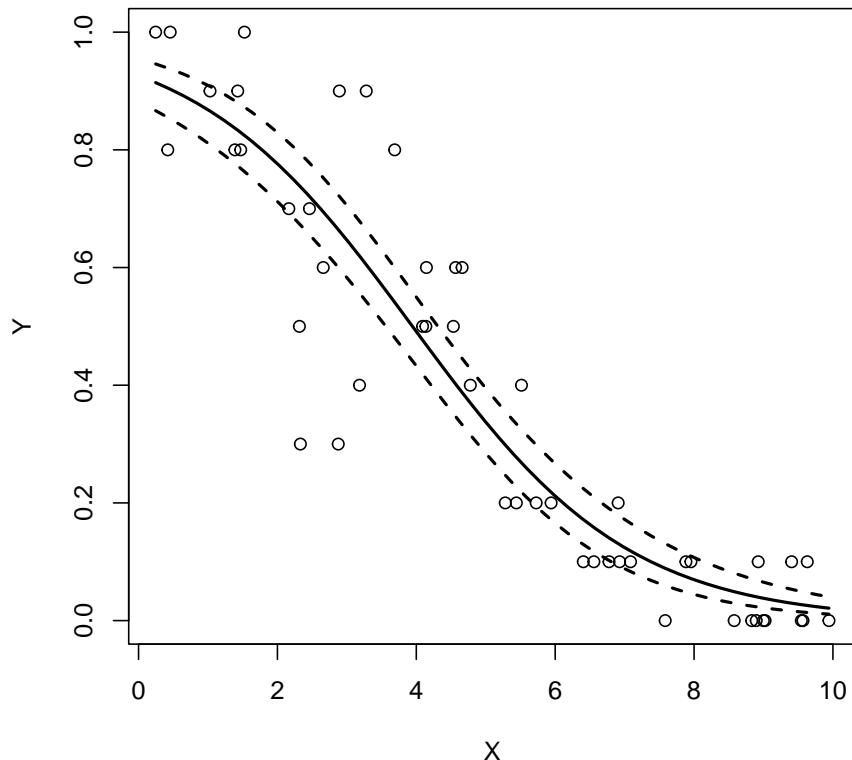
# make the plot:

```

```

plot(x, y, xlab="X", ylab="Y", type="n",
      ylim=c(0, 1))
points(px, lo1, type="l", lty=2, lwd=2)
points(px, up1, type="l", lty=2, lwd=2)
points(px, mn1, type="l", lwd=2)
points(x,y)

```

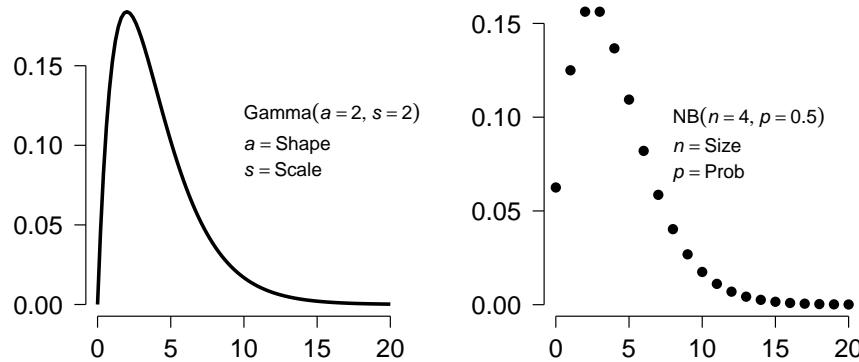


Remember that logistic regression and binomial regression are *the same model*. Both address a binary outcome, but at different levels. Which of the two you should use depends on the nature of your question, the way in which data were collected, and the kind of prediction you are trying to make.

5.8 Gamma models for overdispersed data

This section explores GLMs for data following a **gamma distribution**. Gamma-distributed data can show up in two ways in biology: waiting times, or overdispersed continuous data (i.e., data where $\sigma > \mu$). The latter case can also be well-modeled by a log-linear GLM, but the gamma GLM offers an elegant (and under-utilized) alternative.

Gamma regression is a GLM where response values follow a gamma distribution. The gamma distribution usually describes one of two kinds of data: waiting times until a certain number of events occur, or right-skewed continuous data. The gamma distribution is the continuous analogue of the negative binomial distribution. The figure below shows this:



A gamma distribution with shape a and scale s is basically a continuous version of a negative binomial distribution with size n and probability p according to the following relationships:

$$a = \frac{\mu^2}{\sigma^2}$$

$$s = \frac{\mu}{a}$$

Where μ and σ^2 can be calculated from the parameters of the negative binomial distribution as:

$$\mu = \frac{n(1-p)}{p}, \quad \sigma^2 = \frac{n(1-p)}{p^2}$$

The relationships above can be used to convert between gamma distributions and negative binomial distributions with other parameterizations. For example,

ecologists often use an alternate version of the negative binomial with mean μ and $\sigma^2 = \mu + (\mu^2/k)$.

Like the negative binomial, biologists can take advantage of the numerical properties of the gamma to model phenomena for which the variance is much greater than the mean. The **log-normal distribution** is another option for right-skewed data. Neither the log-normal nor the gamma distribution can take the value 0, so users of the gamma distribution may need to adjust their data so that 0s do not occur. Of the GLMs described in this guide, gamma models are probably the most under-utilized by biologists.

5.8.1 Example with simulated data

As always, we will start with analysis of simulated data. Notice that we are using **log link function**. The canonical link for the gamma distribution is the **reciprocal function** ($f(x) = 1/x$). However, the reciprocal function is more applicable for the “waiting time” interpretation of the distribution. For the “right-skewed non-negative distribution” interpretation, the log or identity links can be appropriate. However, the log-link cannot be used if your data contain 0s.

```
set.seed(123)
n <- 30
x <- round(runif(n, 0, 4), 1)

# coefficients
beta0 <- 0.9
beta1 <- 1.2

# linear predictor
eta <- beta0 + beta1 * x

# inverse link
y <- exp(eta)

# gamma parameter
shape <- 4

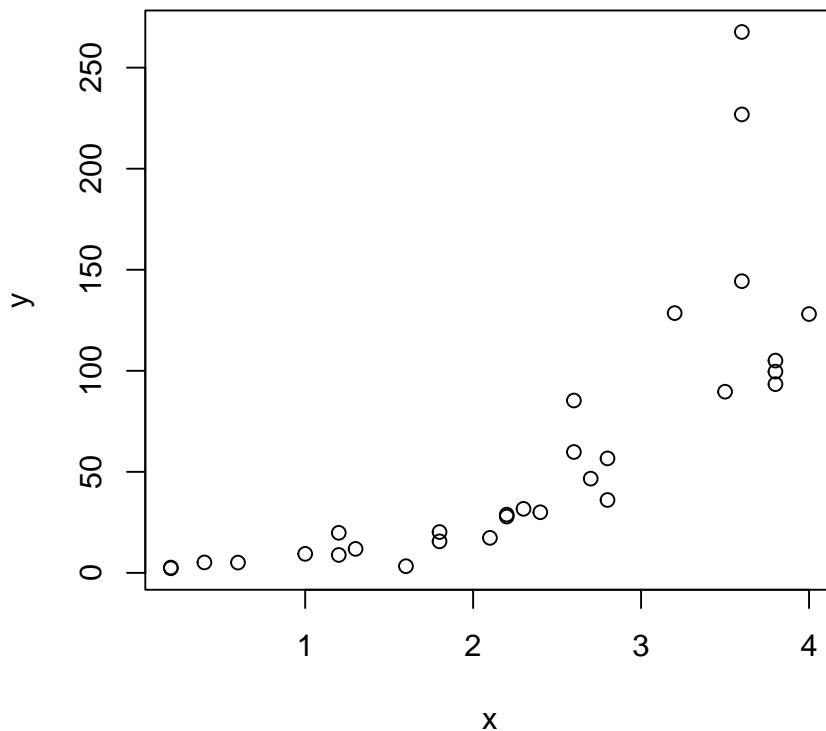
# draw y values (stochastic part)
y <- rgamma(n, rate=shape/y, shape=shape)

# assemble dataset
dat <- data.frame(x=x, y=y)
```

Now, take a look at the data.

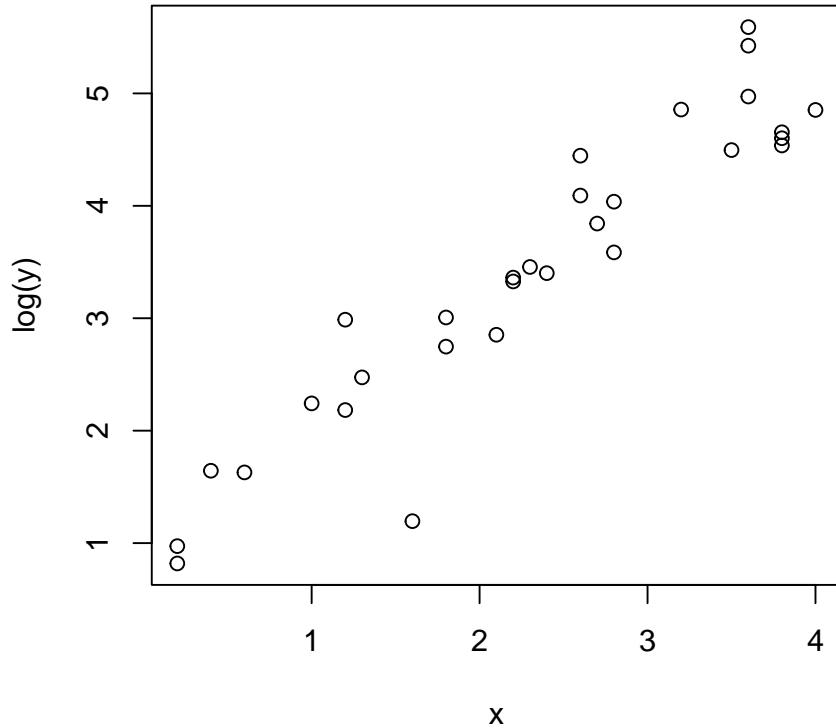
```
head(dat)
```

```
##      x          y
## 1 1.2 19.831466
## 2 3.2 128.548423
## 3 1.6   3.303748
## 4 3.5   89.653108
## 5 3.8 105.050148
## 6 0.2   2.268062
plot(x,y)
```



The relationship looks like it might be log-linear—i.e., linear on a logarithmic scale. We can check that with another scatterplot:

```
plot(x, log(y))
```



When y is log-transformed, things look linear. We will try a Gaussian GLM with log link function. Because the variance looks like it increases at larger X values, we will also try a gamma GLM. One characteristic of the gamma distribution is that the variance is proportional to the mean squared, which might be the case here.

```
mod1 <- glm(y~x, data=dat, family=gaussian(link="log"))
mod2 <- glm(y~x, data=dat, family=Gamma(link="log"))

## Call:
## glm(formula = y ~ x, family = gaussian(link = "log"), data = dat)
## 
## Deviance Residuals:
##      Min        1Q    Median        3Q       Max
## -52.584   -18.722   -8.663   -4.337  143.577
```

```

## 
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)    
## (Intercept) 1.8969     0.6141   3.089   0.0045 **  
## x           0.8122     0.1725   4.708 6.16e-05 *** 
## ---        
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
## 
## (Dispersion parameter for gaussian family taken to be 1650.88)
## 
## Null deviance: 128445  on 29  degrees of freedom
## Residual deviance: 46225  on 28  degrees of freedom
## AIC: 311.34
## 
## Number of Fisher Scoring iterations: 6
summary(mod2)

## 
## Call:
## glm(formula = y ~ x, family = Gamma(link = "log"), data = dat)
## 
## Deviance Residuals:
##      Min        1Q     Median        3Q       Max      
## -1.23251 -0.35454 -0.05758  0.13402  0.76848
## 
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)    
## (Intercept) 0.99844    0.17787   5.613 5.21e-06 ***  
## x           1.08949    0.06932  15.717 2.02e-15 *** 
## ---        
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
## 
## (Dispersion parameter for Gamma family taken to be 0.1887889)
## 
## Null deviance: 41.3449  on 29  degrees of freedom
## Residual deviance: 5.4667  on 28  degrees of freedom
## AIC: 245.54
## 
## Number of Fisher Scoring iterations: 4

```

Notice that the parameter estimates from the gamma GLM were much closer to “reality” than those of the Gaussian GLM. Which model performed better? The pseudo- R^2 suggests the gamma:

```
1-mod1$deviance/mod1>null.deviance
```

```
## [1] 0.6401217
```

```
1-mod2$deviance/mod2>null.deviance
```

```
## [1] 0.8677773
```

The best test is to plot the predicted values against the actual values and compare the models.

```
# values for prediction
use.n <- 50
px <- seq(min(x), max(x), length=use.n)

# calculate predictions and CI
pred1 <- predict(mod1, newdata=data.frame(x=px),
                  type="link", se.fit=TRUE)
mn1 <- pred1$fit
lo1 <- qnorm(0.025, mn1, pred1$se.fit)
up1 <- qnorm(0.975, mn1, pred1$se.fit)

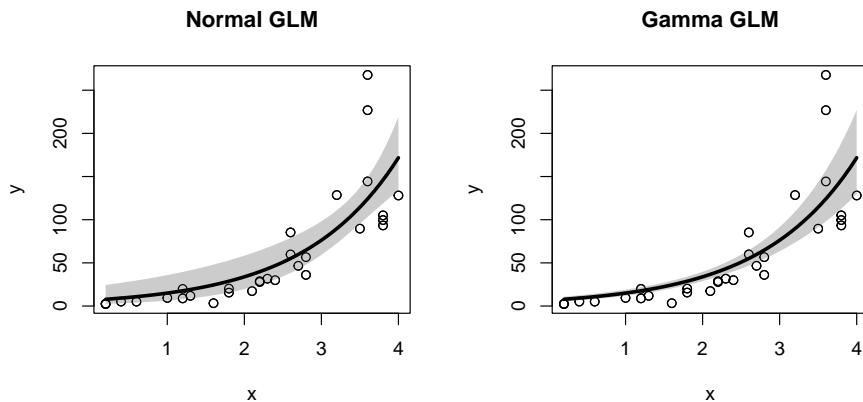
# inverse link function
mn1 <- mod1$family$linkinv(mn1)
lo1 <- mod1$family$linkinv(lo1)
up1 <- mod1$family$linkinv(up1)

# calculate predictions and CI for gamma model
pred2 <- predict(mod2, newdata=data.frame(x=px),
                  type="link", se.fit=TRUE)
mn2 <- pred2$fit
lo2 <- qnorm(0.025, mn2, pred2$se.fit)
up2 <- qnorm(0.975, mn2, pred2$se.fit)

# inverse link function
mn2 <- mod1$family$linkinv(mn2)
lo2 <- mod1$family$linkinv(lo2)
up2 <- mod1$family$linkinv(up2)

# make plot
par(mfrow=c(1,2))
plot(x,y, main="Normal GLM")
polygon(x=c(px, rev(px)), y=c(lo1, rev(up1)),
        border=NA, col="grey80")
points(px, mn1, type="l", lwd=3)
points(x, y)
plot(x,y, main="Gamma GLM")
polygon(x=c(px, rev(px)), y=c(lo2, rev(up2)),
        border=NA, col="grey80")
points(px, mn2, type="l", lwd=3)
```

```
points(x, y)
```



The gamma GLM captured the heteroscedasticity in the data (non-constant variance) better than the Gaussian (normal) GLM. We should conclude that the gamma GLM better represents the data. If you have a heteroscedastic relationship that isn't well-modeled by other GLMs, the gamma GLM might be worth a try.

5.9 Beyond GLM: Overview of GAM and GEE

In other sections we have explored the power and utility of GLMs for biological data. This page describes two methods that can be thought of as extensions of GLM: generalized additive models (GAM) and generalized estimating equations (GEE). GAM will get a more thorough treatment in their own section in the future.

5.9.1 Generalized additive models (GAM)

Generalized additive models (GAM) go one step further than GLMs by relaxing an additional assumption: the assumption of a linear relationship between the response variable on the link scale and the predictors. GAMs fit a curve to data that can vary in complexity. These curves are commonly called **smoothers**. However, the parameters of the curve are not tested individually in the way that the parameters in something like a quadratic model or non-linear model would be. Instead, the curve itself is treated as a regression coefficient. Compare the equation for the linear predictor of a GLM with k predictors

$$g(E(Y)) = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \dots + \beta_k X_k$$

to the equation of the predictor of a GAM with k predictors:

$$g(E(Y)) = \beta_0 + f_1(X_1) + f_2(X_2) + \dots + f_k(X_k)$$

Instead of each predictor being multiplied by a coefficient, a smoothing function of each predictor is estimated.

When should you use GAM? The most common application is situations where the response is not linearly related to the predictors, or if the response curve takes on a complicated shape. As you might suspect, the smoothers fitted in a GAM can be vulnerable to overfitting just as can be polynomial models or multiple regression models. Researchers usually control for this overfitting of the smoothers by limiting the complexity of the curves; this parameter is usually called the number of “knots”.

GAMs will get a more thorough treatment later in the context of nonlinear models. Some good references to get you started are Wood (2017) and Zuur et al. (2007). The best-known R package for fitting GAMs is mgcv (CRAN page, accessed 2021-09-17).

5.9.2 Generalized estimating equations (GEE)

All GLMs assume that observations are independent of each other. However, this cannot always be assumed in biological data. Observations may be correlated with each other across time and space, or within treatment or blocking groups. Such violations of the independence assumption can often be handled well by mixed models, which are described in their own pages elsewhere on this site. However, the technique of **generalized estimating equations (GEE)** is another way to analyze data with inherent correlations between observations. Some references to start with if you are interested are Liang and Zeger (1986) and Højsgaard et al. (2006). Two important packages used with GEE are **gee** (Carey 2019) and **geepack** (Højsgaard et al. 2006).

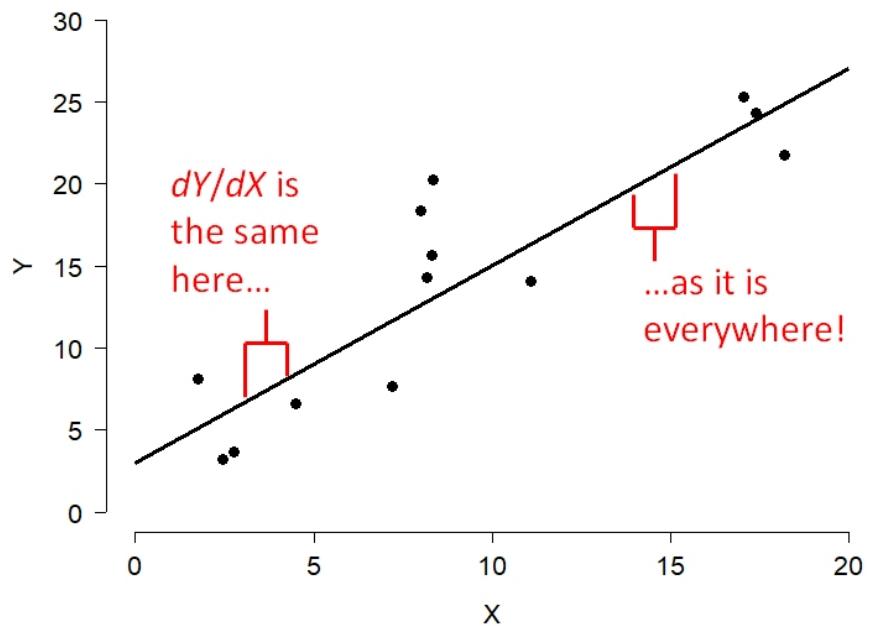
Chapter 6

Nonlinear models

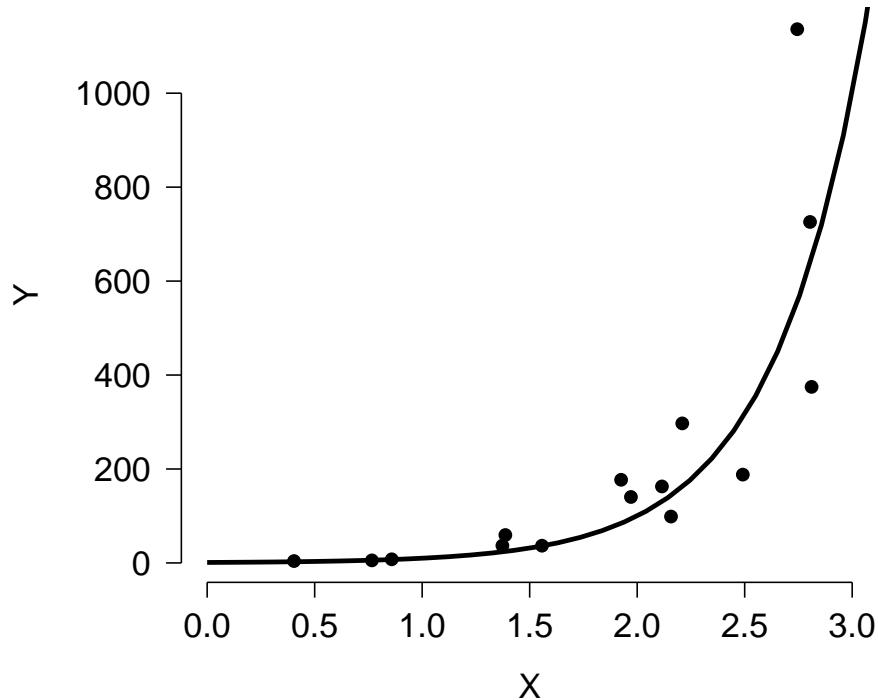
Many biological processes and phenomena are related in ways that do not form straight lines. While linear models and the assumption of linearity are powerful ways to think about and analyze biological phenomena, they are not always appropriate. This module will introduce you to some ways of exploring situations where one quantity does not scale as a constant proportion of another.

6.1 Background

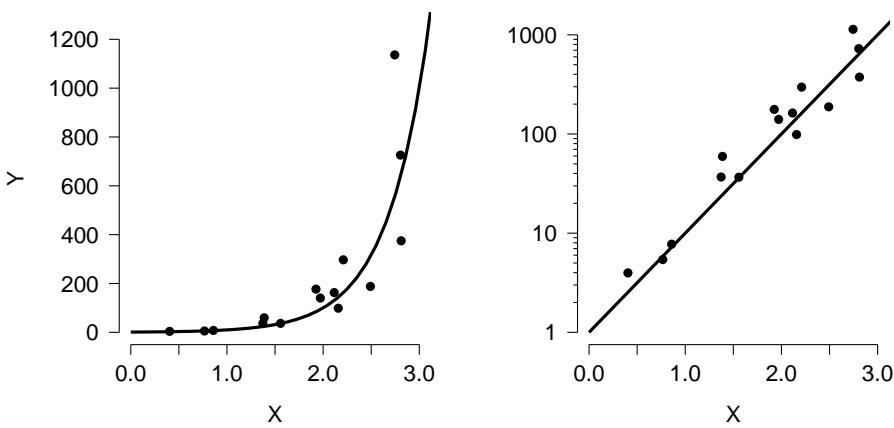
So far we have learned about a wide variety of linear models, which predict some response variable Y as a linear function of one or more predictor variables X . A linear function is one that takes the form $Y = mX + b$, where m and b are constants. The scalar m defines how much Y changes for every unit increase in X (i.e., if X increases by 1, then Y increases by m). In order for the relationship to be a straight line, the change in Y resulting from a change in X does not vary. In other words, the slope of the line (dY/dX) is a constant. This is what “linear” means.



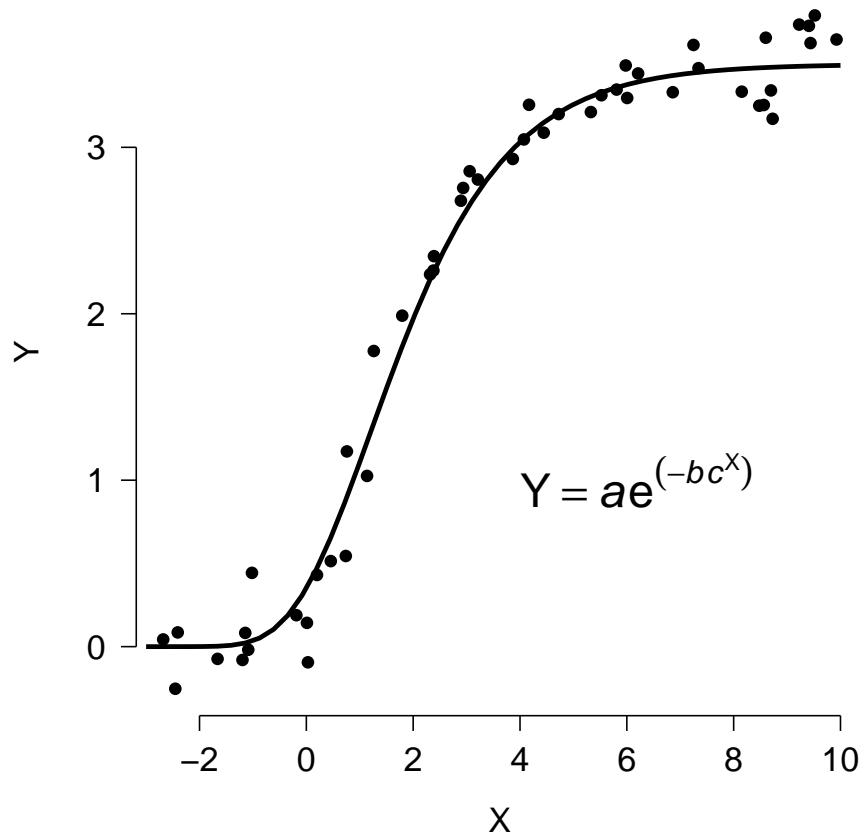
Some relationships do not follow such a rule. Consider the relationship below.
Is the slope constant?



As shown, the relationship between Y and X is nonlinear because the slope of the line is different at different X . However, we can linearize the relationship by taking the logarithm of Y . Compare the left panel below to the right panel. All that has changed is the y -axis scaling.



Other relationships are not amenable to this kind of transformation. For example, the figure below shows Y varying as a **Gompertz function** of X . Unlike the log-linear model shown above, there is no transform that will linearize the Gompertz curve. So, if we wanted to analyze data that followed such a pattern, we would need a different set of tools.



6.2 Nonlinear least squares (NLS)

The basic **nonlinear least squares (NLS)** model has some parts in common with the basic linear regression model (LM). In the state-space GLM notation, we can see that the relationship between the observed Y variables and the expected value μ is the same as in the LM. However, the relationship between the expected value and the predictor variables is not the same.

Component	Linear model (LM)	Nonlinear model (NLS)
Stochastic part	$Y \sim Normal(\mu, \sigma^2)$	$Y \sim Normal(\mu, \sigma^2)$
Link function	$\mu = \eta$	$\mu = \eta$
Expected value	$\eta = \beta_0 + \beta_1 X$	$\mu = f(X, \theta)$

In the NLS model, as with LM, the response variable Y is drawn from a normal distribution with mean μ and variance σ^2 . Also like LM, the mean μ is identical to the predictor η . Unlike LM, in NLS the expected value η is not a linear function but instead a function f of the explanatory variable X and some vector of parameters θ . The function $f(X, \theta)$ can be any number of functions. If $f(X, \theta)$ is a linear function, then the NLS model becomes a linear model.

The equations above show an important assumption of NLS: the errors are normally distributed with constant variance. In some cases, the function $f(X, \theta)$ can be transformed to make it linear. However, doing so will likely make the errors non-normal or heteroscedastic, which means that the LM on transformed data wouldn't be appropriate anyway.

There are several methods of fitting NLS models. Unlike LM, where it is relatively straightforward to solve some linear algebra problems and get stable parameter estimates, in NLS the parameters must be approximated by searching the parameter space (i.e., set of possible values) for θ that minimize the sum of squared residuals. The methods for computing these estimates are extremely complicated and sensitive to starting inputs, and we won't go into them here. Bolker (2008) gives an overview of some of these methods.

6.3 Michaelis-Menten curves

The **Michaelis-Menten model** is a nonlinear model that is usually used to describe processes that grow and approach an asymptote. One of the classic use cases is modeling the relationship between biochemical reaction rates and substrate concentrations. The Michaelis-Menten equation takes the general form

$$Y = \frac{aX}{b + X}$$

where a and b are constants.

- **a** defines the **asymptote**: the maximum Y value that is approached but never reached.
- **b** is the **Michaelis constant**. It represents the X value at which $Y = a/2$. Interestingly, for any proportion of the form $c/(c+1)$, where c is a positive integer, the X value where $Y = (c/(c+1))a$ is cb . For example, the X value when $Y = (3/4)a$ is $3b$.

The Michaelis-Menten curve goes by many names. In enzyme kinetics and biochemistry, it is called the Michaelis-Menten curve. Biologists also sometimes call it the **Monod function**, the **Holling type II** functional response, or the **Beverton-Holt model** (the latter name is probably the most common). This diversity of application speaks to the generality of this function. The original definition predicted a reaction rate. However, Michaelis-Menten curves can also be used to model other phenomena. For example, Kunz et al. (2019) and Green et al. (2020) used Michaelis-Menten curves to model the effectiveness of biodiversity monitoring programs.

6.3.1 Example with simulated data

For our simulated example, we will use parameters from the R help page for `SSmicmen()`, one of the utility functions for working with Michaelis-Menten models.

```
set.seed(123)

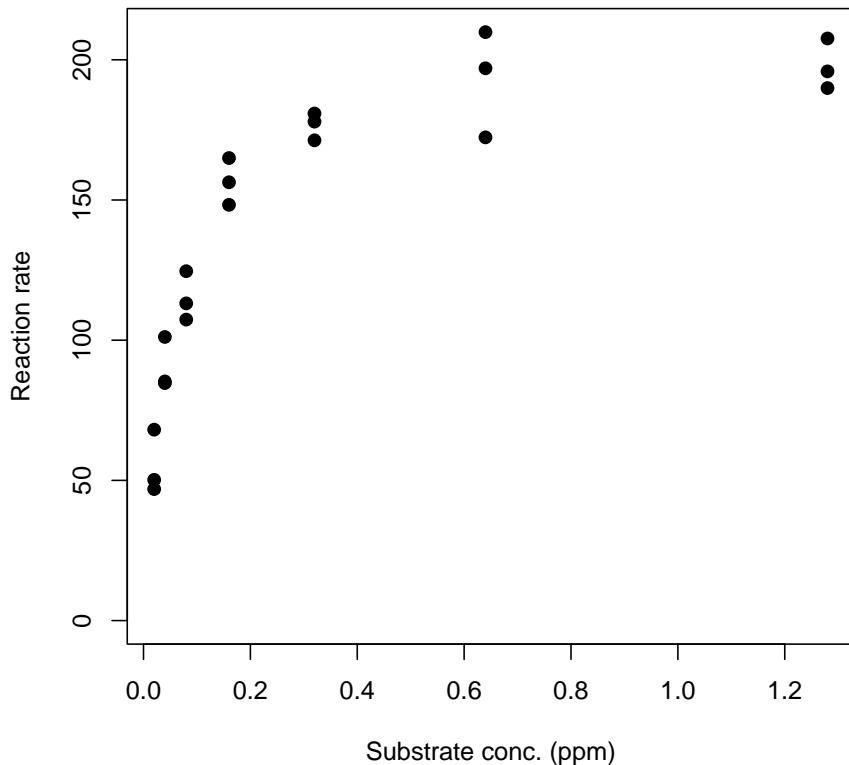
# x values: concentrations in [0.02, 1.28] ppm
x <- rep(2^(1:7)/100, each=3)

# define model parameters
## Vm = a = asymptote
## K = b = Michaelis constant
Vm <- 210
K <- 0.06

# expected values
y <- (Vm*x)/(K+x)

# add residual variation
y <- y + rnorm(length(x), 0, 10)

# plot data
plot(x,y, ylim=c(0, max(y)),
      xlab="Substrate conc. (ppm)",
      ylab="Reaction rate",
      pch=16, cex=1.2)
```



The R function that fits nonlinear least squares model is `nls()`. Like `lm()` and `glm()`, `nls()` has a formula interface to specifying models and produces an object that contains many of the diagnostics you need to evaluate your model. Unlike `lm()` and `glm()`, `nls()` usually needs to be supplied starting values for model parameters. These values are used as starting points for the numerical optimization algorithm. There are two basic ways to supply starting values. For some models you can use a “self-starting” model that will guess at good starting values. The other way is for you to guess.

The command below shows how to use the “self-start” Michaelis-Menten model. The righthand side of the formula is the function `SSmicmen()` (“self-starting Michaelis-Menten”) instead of the model equation. The arguments to a self-start function are usually the predictor variable followed by the names you want to use for the terms in the model. The order of those names matters, because R will use them in the order you provide them. Below we use the default names `Vm` and `K`, which R interprets as the asymptote (“`Vmax`” or “`V-max`”) and Michaelis constant, respectively. When using the Michaelis-Menten model for something

other than enzyme kinetics, I prefer to use a and b instead of V_m and K .

```
# method 1: fit with self-start models
mod1 <- nls(y~SSmicmen(x, Vm, K))

# same but with different parameter names:
mod1.v2 <- nls(y~SSmicmen(x, a, b))
summary(mod1.v2)

##
## Formula: y ~ SSmicmen(x, a, b)
##
## Parameters:
##   Estimate Std. Error t value Pr(>|t|)
## a 2.080e+02 4.499e+00 46.23 < 2e-16 ***
## b 5.640e-02 4.995e-03 11.29 7.21e-10 ***
## ---
## Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 10 on 19 degrees of freedom
##
## Number of iterations to convergence: 0
## Achieved convergence tolerance: 2.333e-07
```

If you are fitting a curve for which there is no self-start method, or the self-start method doesn't work, or if you enjoy reading error messages, you will need to supply starting values. Coming up with good starting values is part art, part science, and part random guessing. For many models you can often estimate some values by thinking about what they represent. For our Michaelis-Menten example, we can see that the curve stops increasing once it gets up to about $Y = 200$ or so. Thus, 200 might be a good guess for the asymptote (V_m in our code). Likewise, we can see that Y reaches half of its asymptotic value at around $X = 0.05$. So, 0.05 is probably a decent guess for the Michaelis constant K .

For models with more parameters, you can often guess at 1 or 2 parameters based on the scatterplot, then pick a few representative points and solve for other parameters. For example, if we had guessed that V_m might be 200, we could pick some points from our data and solve for K , conditional on $V_m = 200$. Below we calculate `test.k`, which contains estimates of K conditional on $V_m = 200$. The mean or median of that vector is a good guess for a starting value of K .

```
test.vm <- 200
test.k <- (test.vm*x)/y-x
mean(test.k)

## [1] 0.042005
```

Use these estimates as starting values to `nls()`. Notice that because we are not using a self-starting model, we must define the model equation manually. The

variable names that you use in the formula must match the variable names in the list of starting values *exactly*.

```
# method 2: eyeball some starting values
start.list <- list(Vm=200, K=0.042)
mod2 <- nls(y~(Vm*x)/(K+x),
            start=start.list)
summary(mod2)

##
## Formula: y ~ (Vm * x)/(K + x)
##
## Parameters:
##   Estimate Std. Error t value Pr(>|t|)
## Vm 2.080e+02 4.499e+00 46.23 < 2e-16 ***
## K  5.640e-02 4.995e-03 11.29 7.21e-10 ***
## ---
## Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 10 on 19 degrees of freedom
##
## Number of iterations to convergence: 4
## Achieved convergence tolerance: 2.899e-07
```

Both methods estimated parameters that were close to the true values. The coefficients table can be presented in much the same way that the coefficients table for a linear regression or GLM is presented. Unlike linear models, there is no straightforward way to calculate an R^2 value for nonlinear models. This is because of two key differences between nonlinear models and linear models:

- Nonlinear models do not have an intercept term, which is implicit in the R^2 calculation in linear models.
- In linear models, the equation $SS_{total} = SS_{model} + SS_{residuals}$ is always true, which implies that $SS_{model} \leq SS_{total}$. For nonlinear models, it is not. This means that the usual R^2 equation, SS_{model}/SS_{total} , is not guaranteed to be between 0 and 1.

Nevertheless, there are measures of model performance that we can use to approximate an R^2 value. One way is to calculate a pseudo- R^2 similar to that we calculated for GLM:

$$pseudo-R^2 = 1 - \frac{SS_{residual}}{SS_{total}}$$

This method is recommended by Schabenberger and Pierce (2001) and popular on Stack Overflow, but it can be misleading when model fit is poor (Spiess and Neumeyer 2010). The pseudo- R^2 should only ever be considered a rough guide to model fit. Unlike a true R^2 , which expresses the total variance explained by a

model because of the way it is defined to be in the interval [0, 1], the pseudo- R^2 is not restricted to [0, 1] and thus is not the same as variance explained. Better fitting models will usually have greater pseudo- R^2 than worse fitting models, but this statistic does not represent “variance explained” or anything similar.

There is no base function for pseudo- R^2 , but it is easy to calculate. The function below takes in the model object `mod` and the original response variable `Y`, and returns the pseudo- R^2 .

```
pseudoR2 <- function(mod, Y){
  ss.tot <- sum((fitted(mod)-mean(Y))^2)
  ss.res <- sum(residuals(mod)^2)
  1-(ss.res/ss.tot)
}

# test the function:
pseudoR2(mod2, y)
```

[1] 0.9651684

The package `rcompanion` (Mangiafico 2021) has a function called `nagelkerke()` that will calculate and present several alternative pseudo- R^2 values, including the version above. To use it, you need to specify a **null model** that includes no predictor variables. Null models usually just use the mean value of Y as the predictor of Y .

```
library(rcompanion)

# function that uses mean (m) to predict y
# x is included because it is the predictor in mod2
null.fun <- function(x, m){m}

# fit null model using nls().
# notice that m is initialized
# as the mean of y
null.mod <- nls(y~null.fun(x, m),
                  start=list(m=mean(y)))

# get the pseudo R-squared values (and other stuff)
nagelkerke(mod2, null=null.mod)

## $Models
##
## Model: "nls, y ~ (Vm * x)/(K + x), start.list, default, list(50, 1e-05, 0.000976562
## Null:  "nls, y ~ null.fun(x, m), list(m = mean(y)), default, list(50, 1e-05, 0.000976562
##
## ## $Pseudo.R.squared.for.model.vs.null
## Pseudo.R.squared
```

```

## McFadden          0.315231
## Cox and Snell (ML)    0.965975
## Nagelkerke (Cragg and Uhler) 0.965996
##
## $Likelihood.ratio.test
##   Df.diff LogLik.diff  Chisq   p.value
##      -1      -35.497 70.994 3.5836e-17
##
## $Number.of.observations
##
## Model: 21
## Null: 21
##
## $Messages
## [1] "Note: For models fit with REML, these statistics are based on refitting with ML"
##
## $Warnings
## [1] "None"

```

Once you have your fitted model, it's time to present the model predictions alongside the original data. This process is a little more complicated for `nls()` outputs than for `lm()` or `glm()`. Part of the problem is that calculating confidence intervals for NLS predictions is not straightforward: just like the parameters cannot be calculated directly, the uncertainty also cannot be calculated directly. Instead, we have to **approximate** the CI. The base R `predict()` method for `nls()` outputs does not do this¹. The package `propagate` (Spiess 2018) has function `predictNLS()` to approximate CI for NLS models. This function uses Monte Carlo sampling (i.e., random simulations) to estimate CI, so it can take a while to run.

```

library(propagate)
L <- 20
px <- seq(min(x), max(x), length=L)
pred <- predictNLS(mod2,
                    newdata=data.frame(x=px),
                    interval="confidence")
# we just need this part:
pred <- pred$summary

# look at result
# (rounded to fit better on the screen)
head(round(pred, 2))

##   Prop.Mean.1 Prop.Mean.2 Prop.sd.1 Prop.sd.2 Prop.2.5% Prop.97.5% Sim.Mean
## 1      54.46     54.63     2.81     2.82     48.73     60.53     54.66

```

¹Infuriatingly, the base function `predict.nls()` takes `se.fit` as an argument, but ignores it. This has been the case since I was grad student learning about nonlinear models.

```

## 2      125.81      125.90      3.02      3.02      119.57      132.22      125.91
## 3      151.89      151.92      2.52      2.53      146.63      157.21      151.93
## 4      165.41      165.41      2.44      2.44      160.30      170.53      165.41
## 5      173.68      173.67      2.55      2.55      168.34      179.00      173.67
## 6      179.26      179.25      2.70      2.70      173.60      184.90      179.25
##   Sim.sd Sim.Median Sim.MAD Sim.2.5% Sim.97.5%
## 1    3.03     54.46     2.86    49.27     61.21
## 2    3.21    125.82     3.08   119.83    132.54
## 3    2.68    151.89     2.57   146.71    157.32
## 4    2.59    165.41     2.49   160.32    170.56
## 5    2.69    173.68     2.60   168.33    179.00
## 6    2.86    179.26     2.75   173.55    184.86

```

The result contains several predicted means, SD, and quantiles for the response variable at every value of the input X values. The different versions are calculated using first- or second-order Taylor series (.1 and .2), or Monte Carlo simulation (Sim.). The first-order Taylor series approximation is also called the **Delta method**. For each method you can use the mean and SD, or the mean and 95% confidence limits, or use the mean and SD to calculate your own confidence limits. For most models there isn't much difference between the Taylor series approximations and the Monte Carlo estimates, so use whichever you prefer. In the example below we'll plot the Taylor series and Monte Carlo CI to compare; in your work you only need to present one (and identify which one you used!).

```

# 1st order Taylor series
mn1 <- pred$Prop.Mean.1
lo1 <- pred$'Prop.2.5%'
up1 <- pred$'Prop.97.5'

# Monte Carlo
mn2 <- pred$Sim.Mean
lo2 <- pred$'Sim.2.5%'
up2 <- pred$'Sim.97.5'

par(mfrow=c(1,2), bty="n", lend=1, las=1,
  cex.axis=1.3, cex.lab=1.3)
plot(x,y, ylim=c(0, max(y)),
  xlab="Substrate conc. (ppm)",
  ylab="Reaction rate",
  pch=16, cex=1.2,
  main="Taylor series CI")
points(px, lo1, type="l", col="red", lwd=3, lty=2)
points(px, up1, type="l", col="red", lwd=3, lty=2)
points(px, mn1, type="l", col="red", lwd=3)

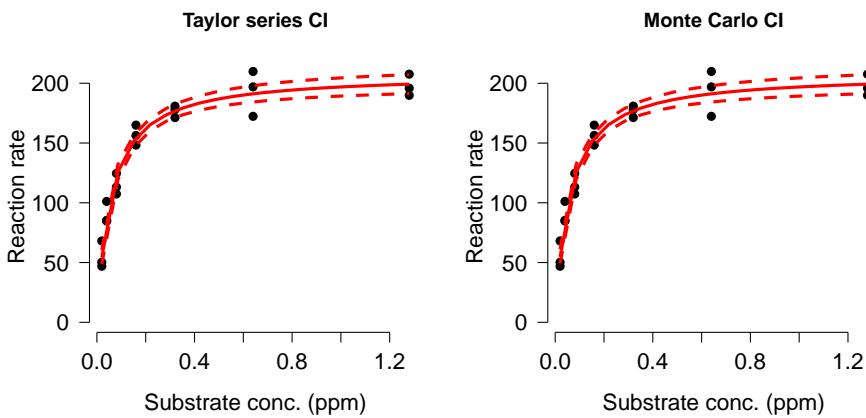
plot(x,y, ylim=c(0, max(y)),
  xlab="Substrate conc. (ppm)",

```

```

ylab="Reaction rate",
pch=16, cex=1.2,
main="Monte Carlo CI")
points(px, lo2, type="l", col="red", lwd=3, lty=2)
points(px, up2, type="l", col="red", lwd=3, lty=2)
points(px, mn2, type="l", col="red", lwd=3)

```



In our simulated example, the Taylor series approximation and Monte Carlo simulations produced nearly identical predictions and CI.

6.3.2 Example with real data

Leonard and Wood (2013) studied the bioaccumulation of nickel (Ni) by four species of invertebrates in different environmental conditions. Invertebrates were held for 14 days in aquaria at varying aqueous Ni concentrations in either hard water (140 mg/L CaCO₃) or soft water (40 mg/L CaCO₃), after which whole-body Ni residues were measured.



The authors used Michaelis-Menten models to measure the uptake of Ni from the water as a function of Ni concentration. We will focus on part of their dataset, accumulation of Ni by midge *Chironomus riparius* (above²). Data were extracted from the paper based on parameter estimates in their Table 4 and data in Figure 6. The variables in this dataset are:

Variable	Units	Meaning
ni	μ mol/L	Concentration of Ni in water
res	μ mol/kg	Concentration of Ni in body tissues
water	N/A	Water treatment: hard or soft (CaCO ₃ conc.)

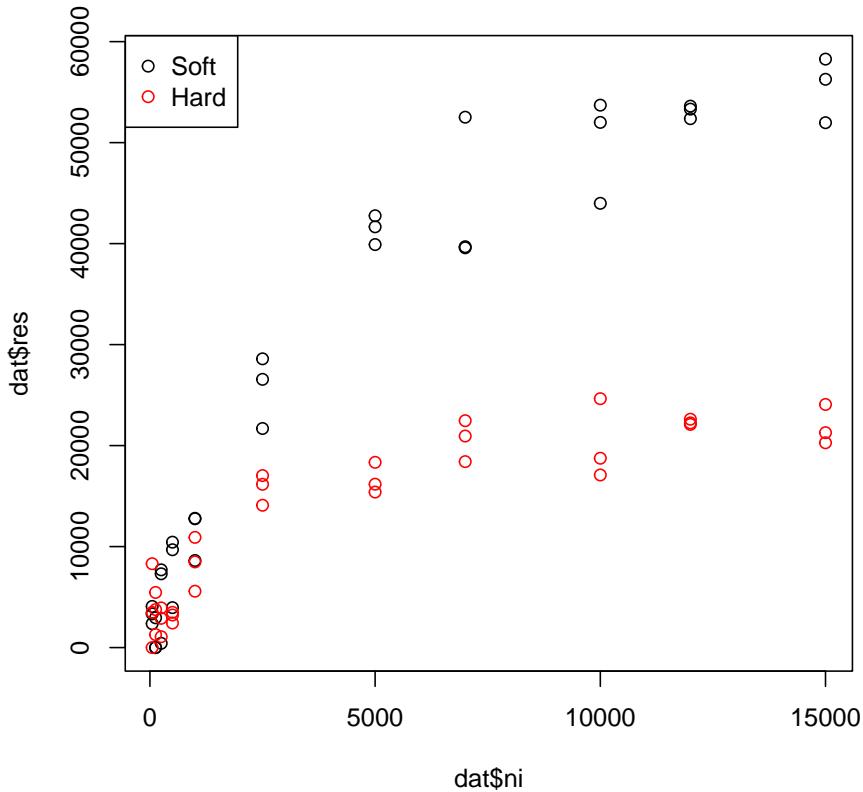
Download the dataset leonard_2013_data.csv and put it in your R home directory.

```
in.name <- "leonard_2013_data.csv"
dat <- read.csv(in.name, header=TRUE)
```

Part of the original experiment was to see whether water hardness affected uptake of Ni. This is recorded in the factor water in the dataset. Plot the data and color the points by water treatment to see if there are any patterns.

```
plot(dat$ni, dat$res,
      col=ifelse(dat$water=="soft", "black", "red"))
legend("topleft",
       legend=c("Soft", "Hard"),
       col=c("black", "red"),
       pch=1)
```

²Image: Foucault et al. (2018)



The figure suggests that Ni accumulates more slowly in hard water than in soft water, and that less Ni is taken up in hard water. The data seem to follow a curve that approaches an asymptote, around $60000 \mu\text{ mol/kg}$ for soft water and $20000 \mu\text{ mol/kg}$ for hard water. The biological interpretation is that organisms can accumulate Ni from their environment, but eventually the rate of uptake will slow as the organism's cells and tissues become saturated (or, because they reach an equilibrium with their environment). High levels of Ni accumulation can directly cause mortality, further limiting uptake. We might suspect that a Michaelis-Menten model is appropriate because it (1) has an asymptote term to capture the maximum accumulation; and (2) describes processes where a rate or first derivative slows as the system saturates. Because the dynamics appear so different for the hard water and soft water treatments, we will fit a separate model for each group.

```
# fit models using nls() and SSmicmen()
flag1 <- which(dat$water == "soft")
flag2 <- which(dat$water == "hard")
```

```
mod1 <- nls(res~SSmicmen(ni, a, b),
            data=dat[flag1,])
mod2 <- nls(res~SSmicmen(ni, a, b),
            data=dat[flag2,])
```

How good are the models? We can calculate the pseudo- R^2 values using the function `pseudoR2()` defined above. Both models look pretty good:

```
pseudoR2(mod1, dat$res[flag1])
```

```
## [1] 0.9754908
```

```
pseudoR2(mod2, dat$res[flag2])
```

```
## [1] 0.9125796
```

Finally, let's produce the usual predicted values and 95% CI for each group.

```
library(propagate)
```

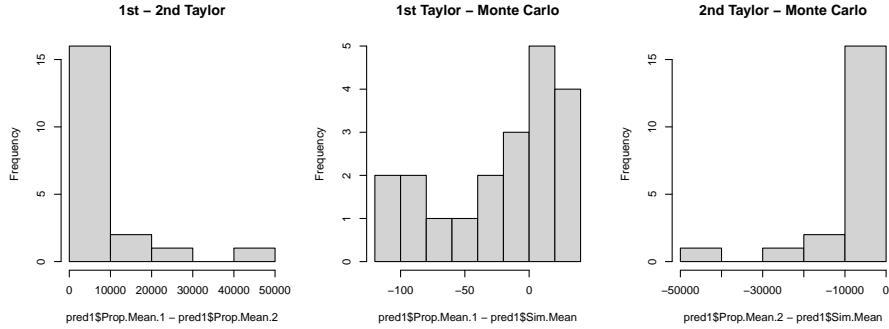
```
# values for prediction
L <- 20
px <- dat$ni[which(dat$ni > 0)]
px <- seq(min(px), max(px), length=L)

# calculate predictions and CI
pred1 <- predictNLS(mod1,
                      newdata=data.frame(ni=px),
                      interval="confidence")
pred2 <- predictNLS(mod2,
                      newdata=data.frame(ni=px),
                      interval="confidence")

# just need this part
pred1 <- pred1$summary
pred2 <- pred2$summary
```

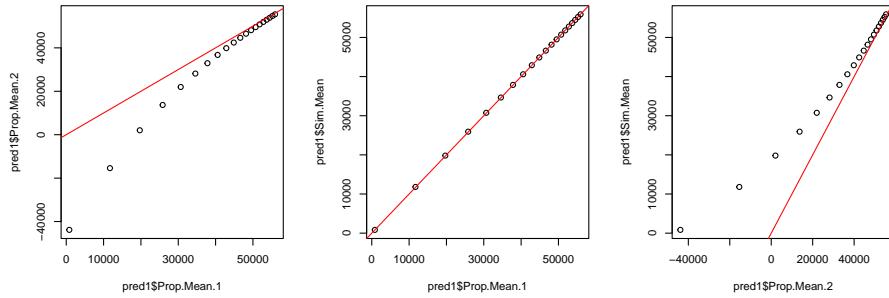
Which predictions should we use? It usually doesn't matter whether you use the 1st order Taylor series, 2nd order Taylor series, or Monte Carlo values, but sometimes it does. We can examine histograms of the differences between the predictions to get a sense of which methods agree with each other.

```
par(mfrow=c(1,3))
hist(pred1$Prop.Mean.1 - pred1$Prop.Mean.2, main="1st - 2nd Taylor")
hist(pred1$Prop.Mean.1 - pred1$Sim.Mean, main="1st Taylor - Monte Carlo")
hist(pred1$Prop.Mean.2 - pred1$Sim.Mean, main="2nd Taylor - Monte Carlo")
```



Take a look at the x -axis scales: the differences between the 1st order Taylor series and Monte Carlo predictions are quite small relative to some of the differences between either of those and the 2nd order Taylor series estimates. This suggests that we should use one of the two methods that closely agree with each other. Another way to compare the predictions is to plot them against each other, with a line that shows where predictions are equal.

```
par(mfrow=c(1,3))
plot(pred1$Prop.Mean.1, pred1$Prop.Mean.2)
abline(a=0, b=1, col="red")
plot(pred1$Prop.Mean.1, pred1$Sim.Mean)
abline(a=0, b=1, col="red")
plot(pred1$Prop.Mean.2, pred1$Sim.Mean)
abline(a=0, b=1, col="red")
```



The second set of figures shows a serious problem with the 2nd order approximation: it predicts negative concentrations! Because of this, let's ignore the 2nd order approximation and stick with the Monte Carlo predictions.

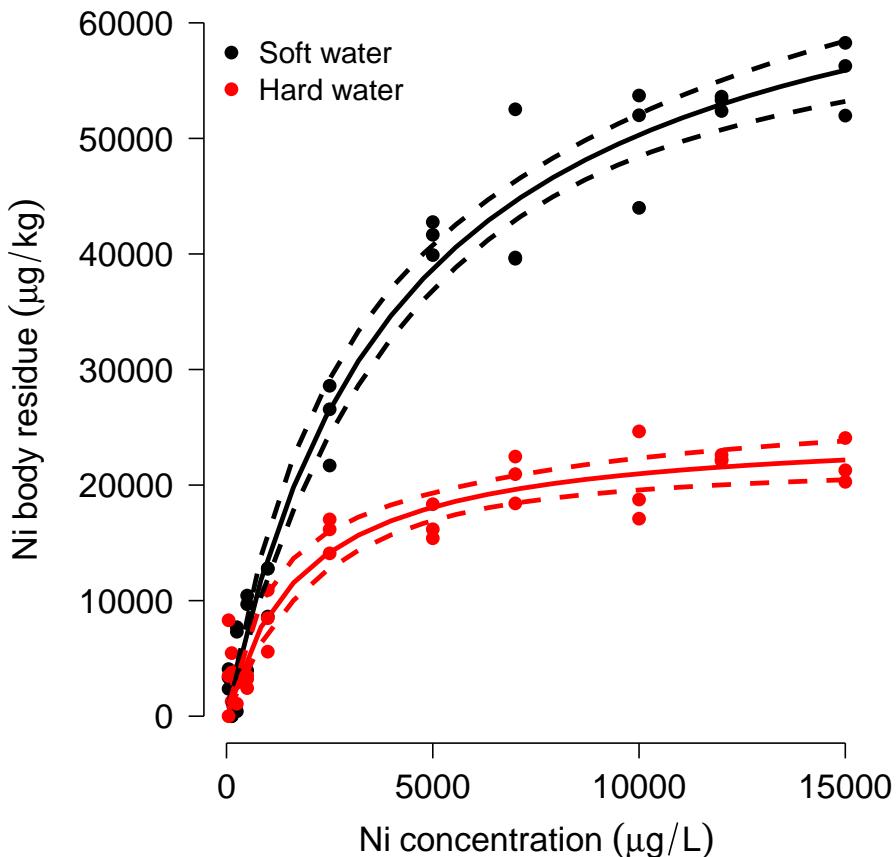
```
# extract predictions and CI
mn1 <- pred1$Sim.Mean
lo1 <- pred1$`Sim.2.5%'
```

```

up1 <- pred1$"Sim.97.5%"
mn2 <- pred2$Sim.Mean
lo2 <- pred2$"Sim.2.5%"
up2 <- pred2$"Sim.97.5%"

# fancy plot options
par(mfrow=c(1,1), mar=c(5.1, 7.1, 1.1, 1.1),
    las=1, lend=1, bty="n",
    cex.axis=1.3, cex.lab=1.3)
# make the plot
plot(dat$ni, dat$res,
      col=ifelse(dat$water=="soft", "black", "red"),
      pch=16, cex=1.2,
      xlab=expression(Ni~concentration~(mu*g/L)),
      ylab="")
# custom y axis (need to move outwards)
title(ylab=expression(Ni~body~residue~(mu*g/kg)), line=5)
points(px, lo1, type="l", lwd=3, lty=2)
points(px, up1, type="l", lwd=3, lty=2)
points(px, mn1, type="l", lwd=3)
points(px, lo2, type="l", col="red", lwd=3, lty=2)
points(px, up2, type="l", col="red", lwd=3, lty=2)
points(px, mn2, type="l", col="red", lwd=3)
legend("topleft",
       legend=c("Soft water", "Hard water"),
       col=c("black", "red"),
       pch=16, bty="n", cex=1.2)

```



6.3.3 Alternative strategies for the analysis

In the example above, we fit two separate models to understand the effect of a continuous predictor in two groups: hard water and soft water. It is also possible to estimate the effects in a single model in `nls()`. This produces a NLS model analogous to ANCOVA: the response variable can vary between groups, and/or the response to the continuous predictor can vary between groups. The models are little more complicated than the analogous LM. We can define models where either the asymptote (a), the Michaelis constant (b), or both a and b vary by group. The deterministic parts of these models can be written compactly as:

Model	Deterministic part
Asymptote by group	$\eta = \frac{a_{water}X}{b+X}$
Michaelis constant by group	$\eta = \frac{aX}{b_{water}+X}$

Model	Deterministic part
Asymptote and Michaelis constant by group	$\eta = \frac{a_{water}X}{b_{water}+X}$

These models can't be coded directly in `nls()`. Instead, we need to rewrite the model first to include some vectors of 1/0 to isolate the observations in each treatment. The equations below show what that looks like.

Model	Deterministic part
Asymptote by group	$\eta = \frac{\psi_{hard}a_{hard}X}{b+X} + \frac{\psi_{soft}a_{soft}X}{b+X}$
Michaelis constant by group	$\eta = \frac{\psi_{hard}a_X}{b_{hard}+X} + \frac{\psi_{soft}a_X}{b_{soft}+X}$
Asymptote and Michaelis constant by group	$\eta = \frac{\psi_{hard}a_{hard}X}{b_{hard}+X} + \frac{\psi_{soft}a_{soft}X}{b_{soft}+X}$

In these equations the coefficient ψ_{hard} takes value 1 for observations from the hard water group and value 0 for observations in the soft water group (ψ is the Greek letter “psi” and is often used for probabilities or binary variables). The coefficient ψ_{soft} does exactly the opposite: takes value 1 for observations from the soft water group and value 0 for observations in the hard water group. This form means that the parameters specific to hard water cannot affect observations from soft water and vice versa. These sorts of parameters are called **dummy variables**³.

We can code each of these models in `nls()`. For starting values we can use the parameter estimates from the models fitted to each group separately (`mod1` and `mod2`).

```
flag3 <- which(dat$ni > 0)
dat2 <- dat[flag3,]

dat2$psi.hard <- as.numeric(dat2$water == "hard")
dat2$psi.soft <- as.numeric(dat2$water == "soft")

# asymptote by group:
```

³Dummy variables are really a way to manipulate the matrix algebra underlying the model-fitting process. Usually, a factor with n levels will be encoded by $n - 1$ dummy variables where each observation has a 1 in at most one of the dummies—the dummy corresponding to the group they belong to. An observation with 0s for all dummies is interpreted as being in the control or baseline group. This example uses a slightly different approach to dummy coding where each level corresponds to one dummy variable and each observation has value 1 for exactly 1 dummy. The first method ($n - 1$ dummies) leads to an **effects parameterization** of the model, where effects are presented as changes from the baseline (e.g., as seen in R's `lm()` outputs). The second method (n dummies) produces a **means parameterization** of the model, where parameter estimates are presented as group- or level-specific point estimates.

```

mod3 <- nls(res~
  (psi.hard)*((a.hard*ni)/(b+ni))+  

  (psi.soft)*((a.soft*ni)/(b+ni)),  

  data=dat2,  

  start=list(a.hard=25038, a.soft=71997, b=2000))  

# Michaelis constant by group:  

mod4 <- nls(res~
  (psi.hard)*((a*ni)/(b.hard+ni))+  

  (psi.soft)*((a*ni)/(b.soft+ni)),  

  data=dat2,  

  start=list(a=48500, b.hard=1928, b.soft=4310))  

# both parameters by group  

mod5 <- nls(res~
  (psi.hard)*((a.hard*ni)/(b.hard+ni))+  

  (psi.soft)*((a.soft*ni)/(b.soft+ni)),  

  data=dat2,  

  start=list(a.hard=20000, a.soft=50000,  

            b.hard=1000, b.soft=4300))

```

For complex models like model 5, you may get the dreaded `singular gradient` error in `nls()`, which means that R can't find a solution from the starting values you provided. You might be able to get the model to work by changing the starting values, but probably not. We'll come back to model 5 later and explore an alternative way to fit it.

We can use AIC to compare the fits of the models that did work:

```
AIC(mod3, mod4, mod5)
```

```

##      df      AIC
## mod3  4 1256.312
## mod4  4 1293.576
## mod5  5 1250.118

```

We can also use package `propagate` to calculate fitted values and 95% CI.

```

px <- seq(min(dat2$ni), max(dat2$ni), length=20)
dx <- expand.grid(ni=px, psi.hard=c(1,0))
dx$psi.soft <- ifelse(dx$psi.hard==1, 0, 1)

pred <- predictNLS(mod5,
  newdata=data.frame(dx),
  interval="confidence")
pred <- pred$summary

```

Extract the values you need for a plot:

```

flag.soft <- which(dx$psi.soft == 1)
flag.hard <- which(dx$psi.hard == 1)

mn.soft <- pred$Sim.Mean[flag.soft]
lo.soft <- pred$"Sim.2.5%" [flag.soft]
up.soft <- pred$"Sim.97.5%" [flag.soft]

mn.hard <- pred$Sim.Mean[flag.hard]
lo.hard <- pred$"Sim.2.5%" [flag.hard]
up.hard <- pred$"Sim.97.5%" [flag.hard]

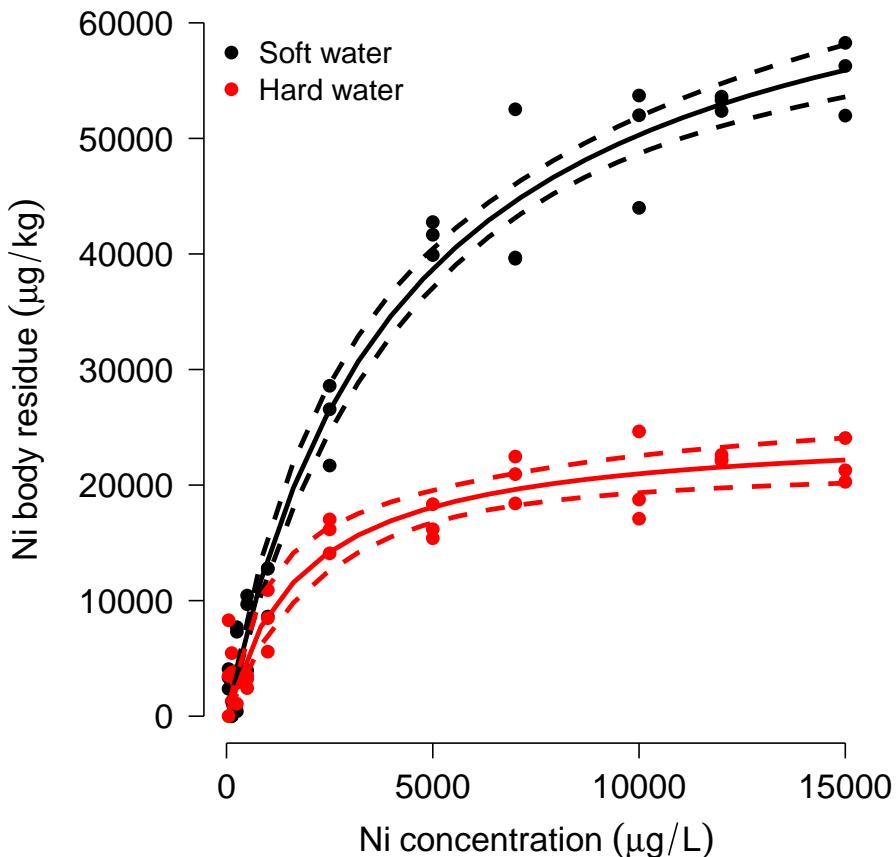
```

Now make a fancy plot:

```

# fancy plot options
par(mfrow=c(1,1), mar=c(5.1, 7.1, 1.1, 1.1),
    las=1, lend=1, bty="n",
    cex.axis=1.3, cex.lab=1.3)
# make the plot
plot(dat2$ni, dat2$res,
      col=ifelse(dat2$water=="soft", "black", "red"),
      pch=16, cex=1.2,
      xlab=expression(Ni~concentration~(mu*g/L)),
      ylab="")
# custom y axis (need to move outwards)
title(ylab=expression(Ni~body~residue~(mu*g/kg)), line=5)
points(px, lo.soft, type="l", lwd=3, lty=2)
points(px, up.soft, type="l", lwd=3, lty=2)
points(px, mn.soft, type="l", lwd=3)
points(px, lo.hard, type="l", col="red", lwd=3, lty=2)
points(px, up.hard, type="l", col="red", lwd=3, lty=2)
points(px, mn.hard, type="l", col="red", lwd=3)
legend("topleft",
       legend=c("Soft water", "Hard water"),
       col=c("black", "red"),
       pch=16, bty="n", cex=1.2)

```



Complicated nonlinear models like model 5 are difficult to fit using the optimization algorithms in R. There is an alternative set of techniques called Markov chain Monte Carlo (MCMC) that can be used for fitting complex models. MCMC uses long chains of random samples to try lots of different possible values for each parameter, and adjusts parameters at each iteration (step in the chain) depending on how well or poorly the model fits with the current parameters. A more in-depth explanation of MCMC was provided in Module 1.

In the next example we will use MCMC to refit model 5. This also turns your analysis into a Bayesian analysis instead of a frequentist analysis. For MCMC we will need the program JAGS as well as the R packages `rjags` and `R2jags`.

```
mod.name <- "mod05.txt"
sink(mod.name)
cat("
  model{
    # priors
```

```

## element 1 = soft
## element 2 = hard
#### asymptote
for(i in 1:2){a.water[i]~dunif(1e3, 1e5)}
#### Michaelis constant
for(i in 1:2){b.water[i]~dunif(1e2, 1e4)}
## residual SD -> precision (tau)
sigma ~ dunif(0, 1e4)
tau.y <- 1 / (sigma * sigma)
# likelihood
for(i in 1:N){
    y[i] ~ dnorm(eta[i], tau.y)
    eta[i] <- (a.water[water[i]] *
                x[i])/(b.water[water[i]]+x[i])
}# i for N
}#model
", fill=TRUE)
sink()

library(R2jags)
library(rjags)
# define initial values for MCMC chains
init.fun <- function(nc){
    res <- vector("list", length=nc)
    for(i in 1:nc){
        res[[i]] <- list(
            a.water=runif(2, 1e3, 1e5),
            b.water=runif(2, 1e2, 1e4),
            sigma=runif(1, 0.1, 1e4))
    }#i
    return(res)
}

nchains <- 3
inits <- init.fun(nchains)

# parameters to monitor
params <- c("a.water", "b.water", "sigma")

# MCMC parameters
n.iter <- 5e4
n.burnin <- 1e4
n.thin <- 100

# package data for JAGS

```

```

in.data <- list(y=dat2$res,
                 x=dat2$ni,
                 N=nrow(dat2),
                 water=dat2$psi.hard+1)

model05 <- jags(data=in.data, inits=inits,
                  parameters.to.save=params,
                  model.file=mod.name,
                  n.chains=nchains, n.iter=n.iter,
                  n.burnin=n.burnin, n.thin=n.thin) #jags

## Compiling model graph
## Resolving undeclared variables
## Allocating nodes
## Graph information:
##   Observed stochastic nodes: 66
##   Unobserved stochastic nodes: 5
##   Total graph size: 284
##
## Initializing model
##
## | | |
## | | |

```

The most important part of the output is here:

```

out <- model05$BUGSoutput$summary

# rounded to fit better on screen:
print(round(out, 1))

##          mean      sd    2.5%     25%     50%     75%    97.5% Rhat n.eff
## a.water[1] 72696.1 3204.7 66797.5 70527.3 72560.4 74638.8 79153.5    1 1200
## a.water[2] 25692.2 1840.1 22460.4 24394.4 25641.5 26795.7 29748.8    1 1200
## b.water[1]  4450.3  536.4  3527.9  4067.7  4424.8  4771.7  5598.3    1 1200
## b.water[2]  2190.3  593.7  1282.1  1770.2  2124.9  2507.9  3634.0    1 1200
## deviance    1245.8    3.4  1241.1  1243.2  1245.1  1247.6  1253.7    1 1200
## sigma       3067.9  289.0  2550.6  2867.5  3046.8  3243.5  3666.6    1 1200

```

The parameter estimates (mean) are close to their true values. The next few columns show SD and various quantiles of the posterior distribution of each parameter. The Gelman-Rubin statistic, \hat{R} (“R-hat”) are <1.01 for each parameter, suggesting that the MCMC chains converged. Using JAGS can be overkill sometimes, but might be the only way to get a model to fit (and there is no guarantee that JAGS can fit a model!). This method has the advantage of fitting the model directly, without adding in dummy variables for each treatment. We can use the MCMC samples saved in the model object to generate model

predictions. The intervals produced for this plot are not true confidence intervals (CI), but their Bayesian analogue, credible intervals (CRI).

```
# MCMC samples
z <- model05$BUGSoutput$sims.list

# make data frame for prediction:
px <- seq(min(dat2$ni), max(dat2$ni), length=20)

# water: 1 = soft, 2 = hard
dx <- expand.grid(ni=px, water=1:2)
dx$lo <- NA
dx$up <- NA
dx$mn <- NA

mx.mn <- matrix(NA, nrow=nrow(dx), ncol=nrow(z[[1]]))
mx.lo <- mx.mn
mx.up <- mx.mn

for(i in 1:ncol(mx.mn)){
  eta <- (z$a.water[i,dx$water]*dx$ni)/(
    z$b.water[i,dx$water]+dx$ni)
  mx.mn[,i] <- eta
  mx.lo[,i] <- pmax(0, eta-mean(z$sigma[,1]))
  mx.up[,i] <- eta+mean(z$sigma[,1])
}
dx$lo <- apply(mx.lo, 1, median)
dx$up <- apply(mx.up, 1, median)
dx$mn <- apply(mx.mn, 1, median)

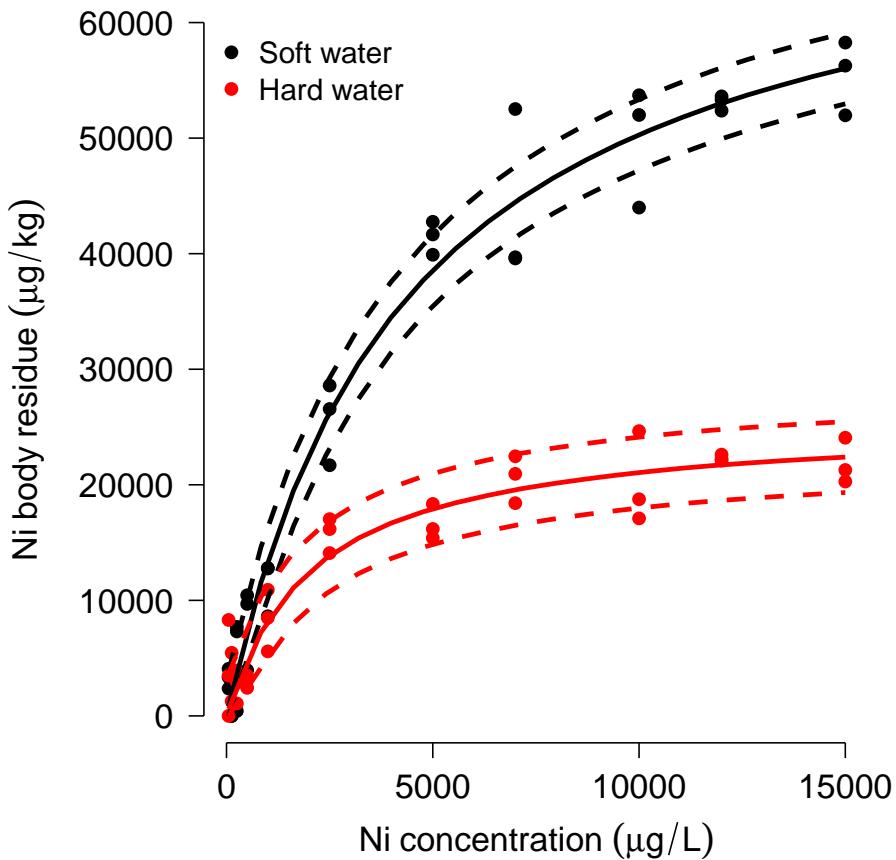
# make the plot
flag.soft <- which(dx$water == 1)
flag.hard <- which(dx$water == 2)

# fancy plot options
par(mfrow=c(1,1), mar=c(5.1, 7.1, 1.1, 1.1),
  las=1, lend=1, bty="n",
  cex.axis=1.3, cex.lab=1.3)
# make the plot
plot(dat2$ni, dat2$res,
  col=ifelse(dat2$water=="soft", "black", "red"),
  pch=16, cex=1.2,
  xlab=expression(Ni~concentration~(mu*g/L)),
  ylab="")
# custom y axis (need to move outwards)
title(ylab=expression(Ni~body~residue~(mu*g/kg)), line=5)
```

```

points(px, dx$lo[flag.soft], type="l", lwd=3, lty=2)
points(px, dx$up[flag.soft], type="l", lwd=3, lty=2)
points(px, dx$mn[flag.soft], type="l", lwd=3)
points(px, dx$lo[flag.hard], type="l", col="red", lwd=3, lty=2)
points(px, dx$up[flag.hard], type="l", col="red", lwd=3, lty=2)
points(px, dx$mn[flag.hard], type="l", col="red", lwd=3)
legend("topleft",
       legend=c("Soft water", "Hard water"),
       col=c("black", "red"),
       pch=16, bty="n", cex=1.2)

```



In this case, we got a model fit similar to what `nls()` produced, and similar predicted values. For this case, JAGS and MCMC are probably more trouble than they are worth. If you are having trouble getting `nls()` to fit a model, MCMC may be a more effective alternative... if the model can be fit and if the

pattern is really there.

6.4 Biological growth curves

This section explores several common biological growth curves. Living organisms often grow in nonlinear ways. For example, humans grow much faster during their first two years of life than during the rest of their lives. Biologists have developed many mathematical models to describe these patterns of growth. What is common to many of these models is that the rate of growth at any given time is proportional to the organism's size at that time. Consequently, biological growth models often include at least one **exponential** term. For mathematical convenience, that exponential term is usually of the form e^x , where e is Euler's constant (≈ 2.718282), also known as the base of the natural logarithm.

6.4.1 Gompertz and von Bertalanffy curves

We will practice fitting two common growth curves, the **Gompertz curve** and the **von Bertalanffy curve**. Both are very commonly used in biology to model the growth of individual organisms. The Gompertz curve is also used to model population growth. Both models are special cases of the generalized logistic function.

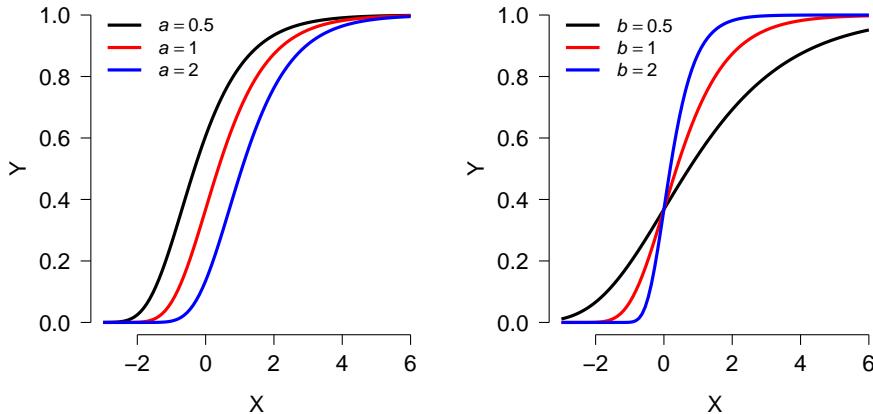
The Gompertz curve is written in several ways. You can use whichever form you want, but you need to be explicit about what you did. Most people use the second of the two forms presented below.

Gompertz model form 1 (Bolker (2008); page 96)

$$f(x) = e^{-ae^{-bx}}$$

- **a:** Displacement along x -axis (equivalent to b in form 2)
- **b:** Growth rate (similar to c in form 2)

The figure below shows the effects of varying the parameters in form 1 of the Gompertz model. Notice that the asymptote is fixed at 1. Because of this, form 2 is more commonly used.

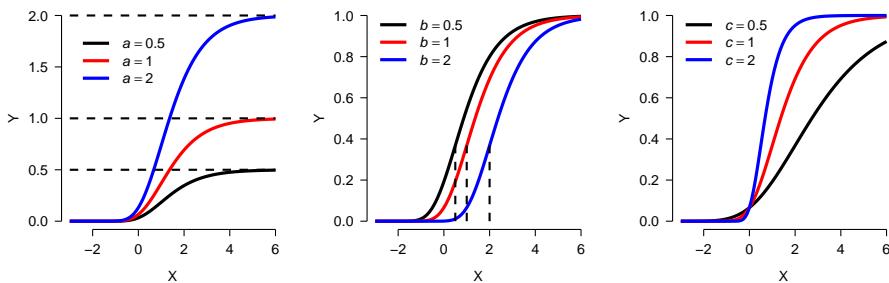


Gompertz model form 2 (more common)

$$f(x) = ae^{-e^{b-cx}}$$

- **a:** Asymptote (maximum value)
- **b:** Displacement along x-axis (shifts left or right)
- **c:** Growth rate (similar to b in form 1)

The figure below shows the effects of varying the parameters in the second form of the Gompertz model:



von Bertalanffy model

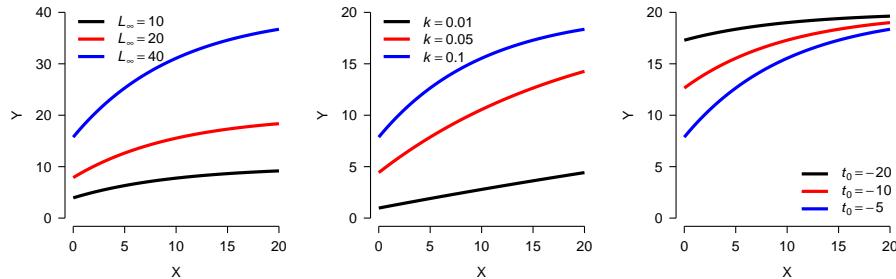
Like the Gompertz curve, the von Bertalanffy growth curve describes size increasing and approaching an asymptote. The von Bertalanffy model is commonly written as:

$$f(x) = L_\infty \left(1 - e^{-k(x-t_0)} \right)$$

- L_∞ : Asymptotic length as x approaches ∞
- k : Growth rate

- t_0 : Theoretical time at which $f(x) = 0$. I.e., displacement along x -axis. Should be negative.

The von Bertalanffy model is commonly used in fisheries to describe fish growth, but in principle it could be used for any situation where its parameters make sense. The same function is also sometimes called the **monomolecular curve**. Like the Gompertz curve, the von Bertalanffy model has several forms. The form given above is the most common. The figure below shows the effects of varying the parameters in the von Bertalanffy model:



Both the Gompertz and von Bertalanffy curves can also be used to model decreasing phenomena, but this is not commonly done.

6.4.2 Example with real data

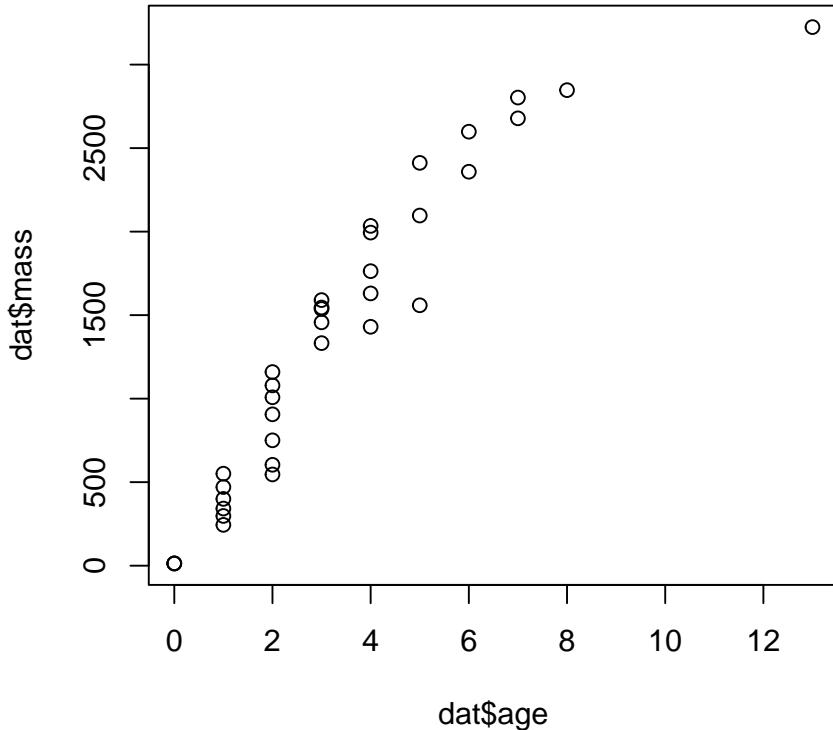
Woodward et al. (2015) studied the life history dynamics of the Cretaceous hadrosaurid dinosaur *Maiasaura peeblesorum* (below⁴). They used microscopic growth rings in the tibias of museum specimens to estimate age at death and the sizes of several bones to estimate total body length and mass. We will use their age and mass data to fit growth curves for this species.

⁴Photo: V. Socha. Wikimedia (accessed 2021-09-24)



The data are stored in the file `woodward2015data.csv`. Download the dataset and put it in your R home directory.

```
in.name <- "woodward2015data.csv"  
dat <- read.csv(in.name, header=TRUE)  
plot(dat$age, dat$mass)
```



The mean of the estimated k (\hat{k} , or “k-hat”) is about 0.09. So, we’ll use that for our starting guess. Now we can fit both the Gompertz and von Bertalanffy models:

```
mod1 <- nls(mass~SSgompertz(age, A, B, C), data=dat)
mod2 <- nls(mass~Linf*(1-exp(-k*(age-t0))),
            data=dat,
            start=list(Linf=4e3, k=0.09, t0=-1))
summary(mod1)

##
## Formula: mass ~ SSgompertz(age, A, B, C)
##
## Parameters:
##   Estimate Std. Error t value Pr(>|t|)
## A 3.145e+03 1.718e+02 18.31 < 2e-16 ***
## B 3.076e+00 2.858e-01 10.76 3.64e-12 ***
## C 6.514e-01 3.081e-02 21.14 < 2e-16 ***
## ---
## Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 208.4 on 32 degrees of freedom
##
## Number of iterations to convergence: 5
## Achieved convergence tolerance: 1.862e-06
summary(mod2)

##
## Formula: mass ~ Linf * (1 - exp(-k * (age - t0)))
##
## Parameters:
##   Estimate Std. Error t value Pr(>|t|)
## Linf 3.966e+03 3.902e+02 10.165 1.52e-11 ***
## k    1.572e-01 2.656e-02  5.917 1.38e-06 ***
## t0   2.315e-01 1.288e-01  1.797  0.0817 .
## ---
## Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 212.7 on 32 degrees of freedom
##
## Number of iterations to convergence: 7
## Achieved convergence tolerance: 1.781e-06
```

Both models were able to fit all of their parameters, although the t_0 term in the von Bertalanffy model was nonsignificant (i.e., not different from 0). This might indicate that we should go with the Gompertz model. The pseudo- R^2 values are

similar and don't strongly suggest choosing one model over another:

```
pseudoR2 <- function(mod, Y){
  ss.tot <- sum((fitted(mod)-mean(Y))^2)
  ss.res <- sum(residuals(mod)^2)
  1-(ss.res/ss.tot)
}
pseudoR2(mod1, dat$mass)

## [1] 0.946444
# [1] 0.946444
pseudoR2(mod2, dat$mass)

## [1] 0.9459103
# [1] 0.9459103
```

The AICs of the models suggest that the Gompertz model is slightly better, but not significantly so. The difference in AIC is <2, suggesting that the models are equivalent in their support. This means that we would be justified choosing either one, or not choosing and presenting both.

```
AIC(mod1, mod2)

##      df      AIC
## mod1  4 477.9660
## mod2  4 479.3922

#      df      AIC
# mod1  4 477.9660
# mod2  4 479.3922
AIC(mod2) - AIC(mod1)

## [1] 1.426267
```

Another way to determine which model fits better is to compare their ability to predict out-of-sample data. That is, data that were not used to fit the model. The most common strategy for this is called “cross-validation”. There are many kinds of cross-validation. What they all have in common is that the model is fit to some of the data, and then its predictive accuracy is checked using the held-out data. One popular strategy is k -fold cross-validation, the data are divided into k subsets, or “folds”. Then, the model is fit k times: each time to all of the data except those in one fold. The predictive accuracy of these models is then calculated for the data in the fold that was left out. The better the model is at predicting out-of-sample data, the smaller the average error should be.

Let's try another method called “leave-one-out” cross-validation. In this strategy, the model is fit to all observations except 1 at a time. Then, the root mean squared error (RMSE) is calculated for both models.

```

N <- nrow(dat)
err1 <- numeric(N)
err2 <- err1

for(i in 1:N){
  mod1i <- nls(mass~SSgompertz(age, A, B, C), data=dat[-i,])
  mod2i <- nls(mass~Linf*(1-exp(-k*(age-t0))),
    data=dat[-i,],
    start=list(Linf=4e3, k=0.09, t0=-1))
  px <- data.frame(age=dat$age[i])
  err1[i] <- predict(mod1i,
    newdata=data.frame(px)) - dat$age[i]
  err2[i] <- predict(mod2i,
    newdata=data.frame(px)) - dat$age[i]
}
sqrt(mean(err1^2))

## [1] 1599.034
sqrt(mean(err2^2))

## [1] 1634.066

```

Again, the Gompertz curve looks slightly better than the von Bertalanffy, but not by much. For our example let's go ahead and plot both models.

```

library(propagate)
px <- seq(min(dat$age), max(dat$age), length=25)
pred <- predictNLS(mod1,
  newdata=data.frame(age=px),
  interval="confidence")
pred <- pred$summary

mn1 <- pred$Sim.Mean
lo1 <- pred$'Sim.2.5%'
up1 <- pred$'Sim.97.5'

pred2 <- predictNLS(mod2,
  newdata=data.frame(age=px),
  interval="confidence")
pred2 <- pred2$summary

mn2 <- pred2$Sim.Mean
lo2 <- pred2$'Sim.2.5%'
up2 <- pred2$'Sim.97.5'

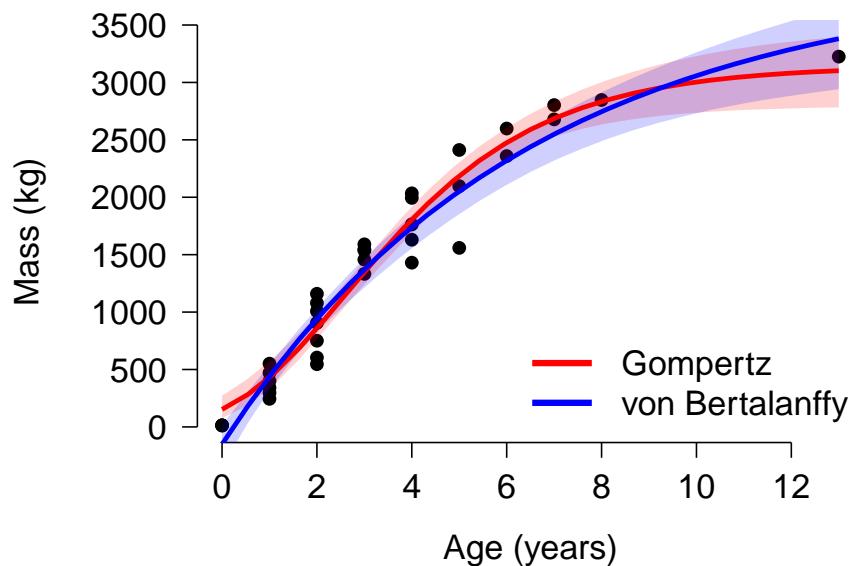
```

Now make the plot:

```

par(mfrow=c(1,1), mar=c(5.1, 7.1, 1.1, 1.1),
  las=1, lend=1, bty="n",
  cex.axis=1.3, cex.lab=1.3)
# make the plot
plot(dat$age, dat$mass,
  pch=16, cex=1.2,
  xlab="Age (years)",
  ylab="", ylim=c(0,3400))
title(ylab="Mass (kg)", line=5)
polygon(x=c(px, rev(px)), y=c(lo1, rev(up1)),
  border=NA, col="#FF000030")
polygon(x=c(px, rev(px)), y=c(lo2, rev(up2)),
  border=NA, col="#0000FF30")
points(px, mn1, type="l", lwd=3, col="red")
points(px, mn2, type="l", lwd=3, col="blue")
legend("bottomright",
  legend=c("Gompertz", "von Bertalanffy"),
  lwd=4, col=c("red", "blue"),
  bty="n", cex=1.3)

```



The plot reveals a problem with the von Bertalanffy: it predicts negative masses! This is probably because the model had a non-significant t_0 with a point estimate that was non-negative. The t_0 term is the age when the response variable is 0, and must be positive to ensure that only positive sizes are predicted.

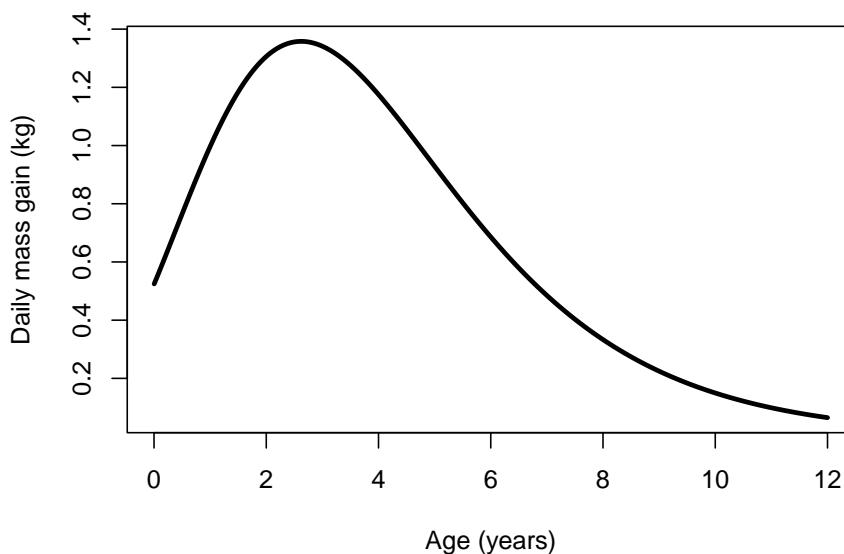
It might be worth asking how fast the animals grew, and how that growth rate changed over the course of their lives. Technically, the growth rate would be the first derivative of the fitted function. However, getting a first derivative might be difficult or impossible for many NLS models. Instead, we can calculate the values empirically. Over a period of 12 years (the domain of our fitted function), daily time steps offer a reasonable approximation of the instantaneous rate of change (daily time steps are more biologically interpretable than a literally instantaneous rate, as well).

```
# daily x values as fractional years
xday <- 0:(12*365.25) # Julian year length
xday <- xday / 365.25

# predicted daily mass
yday <- predict(mod1, newdata=data.frame(age=xday))

ydiff <- diff(yday)
xdiff <- 1:length(ydiff)/365.25

plot(xdiff, ydiff, type="l", lwd=3,
      xlab="Age (years)",
      ylab="Daily mass gain (kg)")
```



The figure shows that daily growth peaked during the 3rd year of life (age 2 to age 3), when animals gained an average of 1.5 kg per day. Interestingly, this is the age at which the authors of the original study hypothesized the animals reached sexual maturity (Woodward et al. 2015).

6.5 Dose response curves

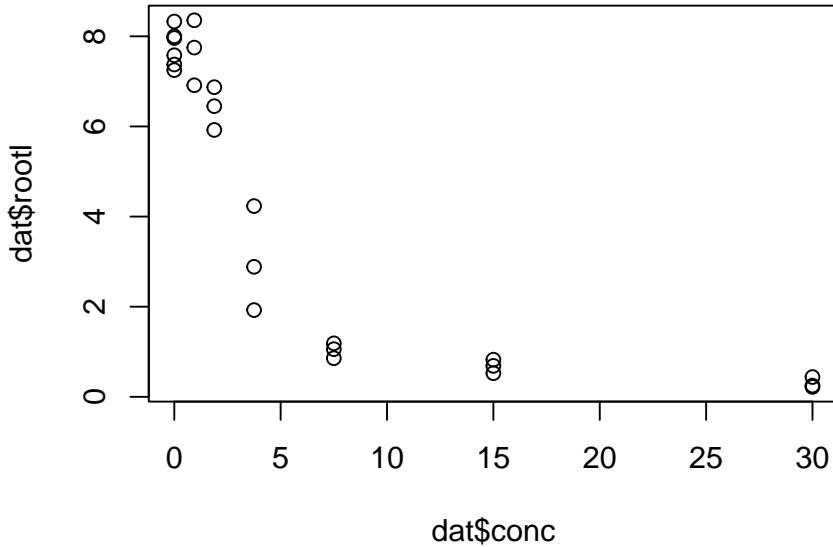
A **dose response curve** describes the change in a biological system following a quantitative perturbation. Most dose response models are nonlinear because of the dynamics of the systems they describe. Mechanistically, there is often a rate that scales with the current concentration of a compound, or some other aspect of the system. In other words, the dynamics of the system are described by a kind of **differential equation**. The state of the system is thus described by solutions to those equations.

There are many dose-response models out there. We will fit two common ones using the R package `drc` (Ritz et al. 2015). Many dose response models could also be fit by `nls()`, but we will use the `drc` package because of its convenience and so that we can learn an alternative approach.

We will use an example from the Supplemental Information provided by the authors of `drc` (Ritz et al. 2015). Inderjit et al. (2002) studied the effect of an herbicide mixture on perennial ryegrass (*Lolium perenne*). Rye grass seeds were placed in Petri dishes and treated with a solution of ferulic acid in concentrations ranging from 0 to 30 mM. The seeds were allowed to germinate and develop for 8 days. After 8 days the root length of each individual seed was measured.



```
library(drc)
dat <- ryegrass
plot(dat$conc, dat$rootl)
```



The plot suggests that increasing ferulic acid concentration decreases mean root length. The relationship also appears nonlinear: root length decreases very quickly at low ferulic acid concentration, and then levels out as concentration increases beyond about 5 mM.

We will try two nonlinear models found in the `drc` package. The first is the **log-logistic model**. The log-logistic model comes in several flavors, defined by the number of parameters used to specify the curve. Two common varieties are shown below. By convention, one of the parameters is labeled “ e ”, but it is not the same as Euler’s constant e (≈ 2.71828). The parameter labeled e shows up only in the exponent as $\log(e)$, but this does not mean $\log(\exp(1))$ (which would be 1, by definition). The bases of the exponents in the models below are Euler’s e . For clarity, the parameter e is sometimes labeled with another letter, or the exponential function e^x is written as $\exp(x)$.

Three-parameter log-logistic:

$$f(x, (b, d, e)) = \frac{d}{1 + e^{b(\log(x) - \log(e))}} = \frac{d}{1 + \exp(\log(x) - \log(e))}$$

Four-parameter log-logistic:

$$f(x, (b, c, d, e)) = \frac{d - c}{1 + e^{b(\log(x) - \log(e))}} = \frac{d - c}{1 + \exp(b(\log(x) - \log(e)))}$$

Just for fun we can also try a **Weibull type I** function:

$$\begin{aligned} f(x, (b, c, d, e)) &= c + (d - c) e^{-e^{b(\log(x) - \log(e))}} \\ f(x, (b, c, d, e)) &= c + (d - c) \exp(-\exp(b(\log(x) - \log(e)))) \end{aligned}$$

Fitting the models in `drc` is straightforward. First define the response and predictor variables in a formula (as in `lm()` and other functions), specify the dataset to be used, and finally the curve (`fct`) to be fitted. The full list of functions that `drm()` can fit can be found with the command `getMeanFunctions()`.

```
mod1 <- drm(rootl~conc, data=ryegrass, fct=LL.3())
mod2 <- drm(rootl~conc, data=ryegrass, fct=LL.4())
mod3 <- drm(rootl~conc, data=ryegrass, fct=weibull1())

summary(mod1)

##
## Model fitted: Log-logistic (ED50 as parameter) with lower limit at 0 (3 parms)
##
## Parameter estimates:
##
##           Estimate Std. Error t-value p-value
## b:(Intercept) 2.47033   0.34168  7.2299 4.011e-07 ***
## d:(Intercept) 7.85543   0.20438 38.4352 < 2.2e-16 ***
## e:(Intercept) 3.26336   0.19641 16.6154 1.474e-13 ***
## ---
## Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error:
##
## 0.5615802 (21 degrees of freedom)
summary(mod2)

##
## Model fitted: Log-logistic (ED50 as parameter) (4 parms)
##
## Parameter estimates:
##
##           Estimate Std. Error t-value p-value
## b:(Intercept) 2.98222   0.46506  6.4125 2.960e-06 ***
## c:(Intercept) 0.48141   0.21219  2.2688  0.03451 *
## d:(Intercept) 7.79296   0.18857 41.3272 < 2.2e-16 ***
```

```

## e:(Intercept) 3.05795    0.18573 16.4644 4.268e-13 ***
## ---
## Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error:
##
## 0.5196256 (20 degrees of freedom)
summary(mod3)

##
## Model fitted: Weibull (type 1) (4 parms)
##
## Parameter estimates:
##
##             Estimate Std. Error t-value p-value
## b:(Intercept) 2.39341   0.47832  5.0038 6.813e-05 ***
## c:(Intercept) 0.66045   0.18857  3.5023 0.002243 **
## d:(Intercept) 7.80586   0.20852 37.4348 < 2.2e-16 ***
## e:(Intercept) 3.60013   0.20311 17.7250 1.068e-13 ***
## ---
## Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error:
##
## 0.5488238 (20 degrees of freedom)

```

All three models fit, had statistically-significant parameters, and had similar residual standard errors (RSE; a measure of model fit similar to RMSE). Model 2 (4 parameter log-logistic) had the smallest RSE and the smallest AIC:

```
AIC(mod1, mod2, mod3)
```

```

##      df      AIC
## mod1  4 45.20827
## mod2  5 42.31029
## mod3  5 44.93439

```

This is moderately convincing evidence that model 2 is the best-fitting of the three models that we fit. Let's calculate predictions and 95% CI for a range of concentrations and present them in a graph. Unlike `nls()` objects, models fitted using `drm()` have a `predict` method that can generate confidence intervals. The code below may return warnings, but these messages can probably be ignored.

```

L <- 50
px <- seq(min(dat$conc), max(dat$conc), length=L)
pred <- predict(mod2,
                 newdata=data.frame(conc=px),
                 interval="confidence")

```

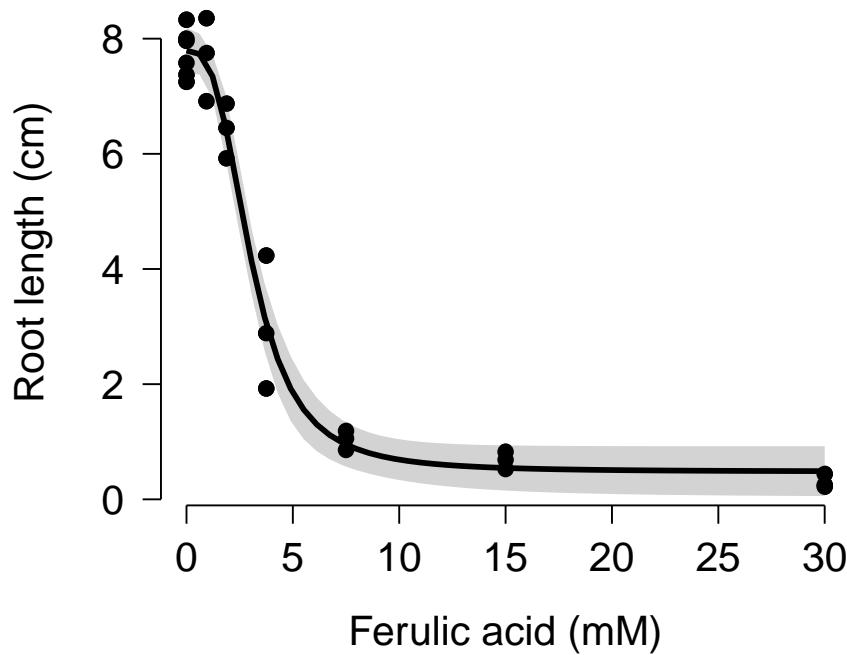
```

mn <- pred[,1]
lo <- pred[,2]
up <- pred[,3]

# fancy plot options
par(mfrow=c(1,1), mar=c(5.1, 5.1, 1.1, 1.1),
  las=1, lend=1, bty="n",
  cex.axis=1.3, cex.lab=1.3)

# make the plot
plot(dat$conc, dat$rootl,
  pch=16, cex=1.2,
  xlab="Ferulic acid (mM)",
  ylab="Root length (cm)")
polygon(c(px, rev(px)), c(lo, rev(up)),
  border=NA, col="lightgrey")
points(px, mn, type="l", lwd=3)
points(dat$conc, dat$rootl, pch=16, cex=1.2)

```



Regardless of whether or not you get warning messages, you should carefully inspect the data, the residuals, and the model predictions to be sure that the

estimates are biologically reasonable. The figure above shows no obvious red flags, but your data might not be so well-behaved.

6.6 Alternatives to NLS

If your goal is to model some response variable that varies nonlinearly with predictor variables, there are several good alternatives to NLS. The examples below introduce generalized nonlinear models (GNM), quantile regression, generalized additive models (GAM), and classification and regression trees (CART). This page briefly describes and demonstrates each method. More in-depth tutorials might be developed in the future.

6.6.1 Generalized nonlinear models

Just as generalized linear models (GLM) extend the linear model by allowing for non-normal response distributions, the generalized nonlinear model (GNM) extends nonlinear models with non-normal response distributions. GNMs are available in the R package `gnm` (Turner and Firth 2020). This package is fairly new and tricky to fit models with. If you have an idea for a GNM model to fit, you might consider using MCMC in JAGS instead.

6.6.2 Quantile regression

Quantile regression is a technique to model specific quantiles of a response variable as a function of predictor variables. In linear regression, GLM, NLS, and GNM, the fitted model predicts the mean value of the response variable. With quantile regression, you can model other quantiles of the distribution. Quantile regression for the 0.5 quantile (50th percentile) is very similar to regression on the mean. Regression for other quantiles can allow you to describe relationships where the mean and predictor variables are not linearly related. A good reference for quantile regression is Cade and Noon (2003). Quantile regression models are available in the R package `quantreg` (Koenker 2021). Like least-squares models, quantile regression can be used to model both linear and nonlinear relationships.

6.6.2.1 Linear quantile regression example

For this example and the nonlinear quantile regression example we will use the “Mammals” dataset found in package `quantreg`. This dataset contains the body weights and maximum running speeds of 107 species of mammals.

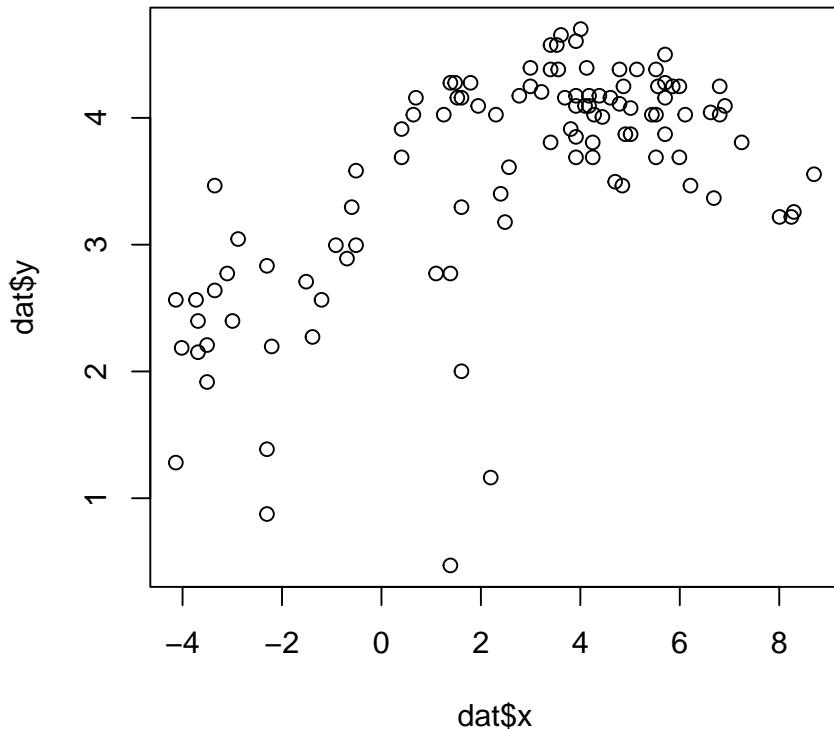
```
library(quantreg)

## Loading required package: SparseM
##
## Attaching package: 'SparseM'
```

```

## The following object is masked from 'package:base':
##
##      backsolve
data(Mammals)
dat <- Mammals
dat$x <- log(dat$weight)
dat$y <- log(dat$speed)
plot(dat$x, dat$y)

```



As you might expect, larger animals tend to run faster than smaller animals. The relationship might be linear or nonlinear, so we will try both linear and nonlinear quantile regression models. First we will define a series of quantiles in $[0, 1]$ at which to fit models. In my experience, quantiles ≤ 0.05 and ≥ 0.95 tend to have numerical problems, especially for small datasets. By convention, the quantiles in `quantreg` are identified by the Greek letter τ (“tau”). The fastest

way to fit quantile regressions is in a `for()` loop.

```
taus <- 5:95/100
ntau <- length(taus)
rq.list <- vector("list", ntau)
for(i in 1:ntau){
  rq.list[[i]] <- rq(y~x, data=dat, tau=taus[i])
}
```

The list `rq.list` contains a fitted model for each quantile τ in [0.05, 0.95]. We will extract some results from these model objects to present as our model outputs.

```
# data frame of parameter estimates and CI
res <- data.frame(tau=taus)
res$b0.mn <- sapply(rq.list, function(x){summary(x)$coefficients[1,1]})
res$b0.lo <- sapply(rq.list, function(x){summary(x)$coefficients[1,2]})
res$b0.up <- sapply(rq.list, function(x){summary(x)$coefficients[1,3]})
res$b1.mn <- sapply(rq.list, function(x){summary(x)$coefficients[2,1]})
res$b1.lo <- sapply(rq.list, function(x){summary(x)$coefficients[2,2]})
res$b1.up <- sapply(rq.list, function(x){summary(x)$coefficients[2,3]})
```

Rather than present model predictions for all 90 quantiles, let's just present a subset. Here we present the predicted 0.1, 0.3, 0.5, 0.7, and 0.9 quantiles of running speed (i.e., 10th, 30th, 50th, 70th, and 90th percentiles).

```
# predictions for selected quantiles
ptau <- c(1, 3, 5, 7, 9)/10
px <- seq(min(dat$x), max(dat$x), length=50)
pmat <- matrix(NA, nrow=length(px), ncol=length(ptau))
for(i in 1:length(ptau)){
  curr.mod <- rq.list[[which(taus == ptau[i])]]
  pmat[,i] <- predict(curr.mod, newdata=data.frame(x=px))
}
```

Finally, we can produce a figure that shows (1) the data and predicted quantiles; (2) the estimate and 95% CI of the model intercept for each τ ; and (3) the estimate and 95% CI of the model slope for each τ . The code for the figure has a lot of components, but each of them should be something familiar by now.

```
# plot data and parameter estimates vs. tau
par(mfrow=c(1,3), bty="n", mar=c(5.1, 5.1, 1.1, 1.1),
lend=1, las=1, cex.axis=1.3, cex.lab=1.3)
plot(dat$x, dat$y, ylim=c(0, 5.5),
xlab="Body mass (kg)",
ylab="Running speed (km/h)",
xaxt="n", yaxt="n")
for(i in 1:length(ptau)){
  points(px, pmat[,i], type="l")
```

```

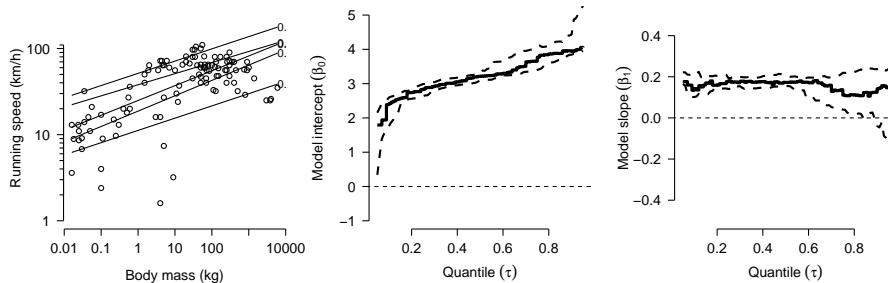
    text(px[length(px)], pmat[nrow(pmat),i], ptau[i], adj=0)
}

xax <- 10^seq(-2, 4, by=1)
axis(side=1, at=log(xax), labels=xax)
yax1 <- c(1, 10, 100)
yax2 <- c(2:9, 2:9*10, 110)
axis(side=2, at=log(yax1), labels=yax1)
axis(side=2, at=log(yax2), labels=NA, tcl=-0.3)

plot(taus, res$b0.mn, type="s",
      lwd=3, ylim=c(-1, 5),
      xlab=expression(Quantile~(tau)),
      ylab=expression(Model~intercept~(beta[0])))
points(taus, res$b0.lo, type="l", lwd=2, lty=2)
points(taus, res$b0.up, type="l", lwd=2, lty=2)
segments(0, 0, 1, 0, lty=2, lwd=1)

plot(taus, res$b1.mn, type="s", lwd=3,
      ylim=c(-0.5, 0.5),
      xlab=expression(Quantile~(tau)),
      ylab=expression(Model~slope~(beta[1])))
points(taus, res$b1.lo, type="l", lwd=2, lty=2)
points(taus, res$b1.up, type="l", lwd=2, lty=2)
segments(0, 0, 1, 0, lty=2, lwd=1)

```



How do we interpret this result? One result is that the model intercept increases steadily with quantile. In other words, the quantile regression line has a greater intercept for greater response quantiles. This is very common. The rightmost plot shows that the quantile regression slope doesn't vary much between quantiles. This suggests that the data would have worked well with ordinary linear regression. If there is a trend in slope (β_1) vs. quantile (τ), this often indicates that the predictor variables affect not only the mean response, but may limit the response (for significant β_1 at greater τ) or facilitate the response (for significant β_1 at smaller τ).

6.6.3 Generalized additive models (GAM)

Models such as the LM and GLM, when fit with >1 predictor variable, have another property that we have not discussed: the effects of the different predictors are additive. In other words, the effect of one predictor is added to the effect of the other predictors to define the expected value of the response. This means that the effects of the predictors are independent of each other.

Generalized additive models (GAM) go one step further than GLMs by relaxing an additional assumption: the assumption of a linear relationship between the response variable on the link scale and the predictors. GAMs fit a curve to data that can vary in complexity. These curves are commonly called “smoothers”. However, the parameters of the curve are not tested individually in the way that the parameters in something like a quadratic model or non-linear model would be. Instead, the curve itself is treated as a regression coefficient. Compare the equation for the linear predictor of a GLM with k predictors

$$g(E(Y)) = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \dots + \beta_k X_k$$

to the equation of the predictor of a GAM with k predictors:

$$g(E(Y)) = \beta_0 + f_1(X_1) + f_2(X_2) + \dots + f_k(X_k)$$

Instead of each predictor being multiplied by a coefficient, a smoothing function $f_k(x)$ of each predictor is estimated.

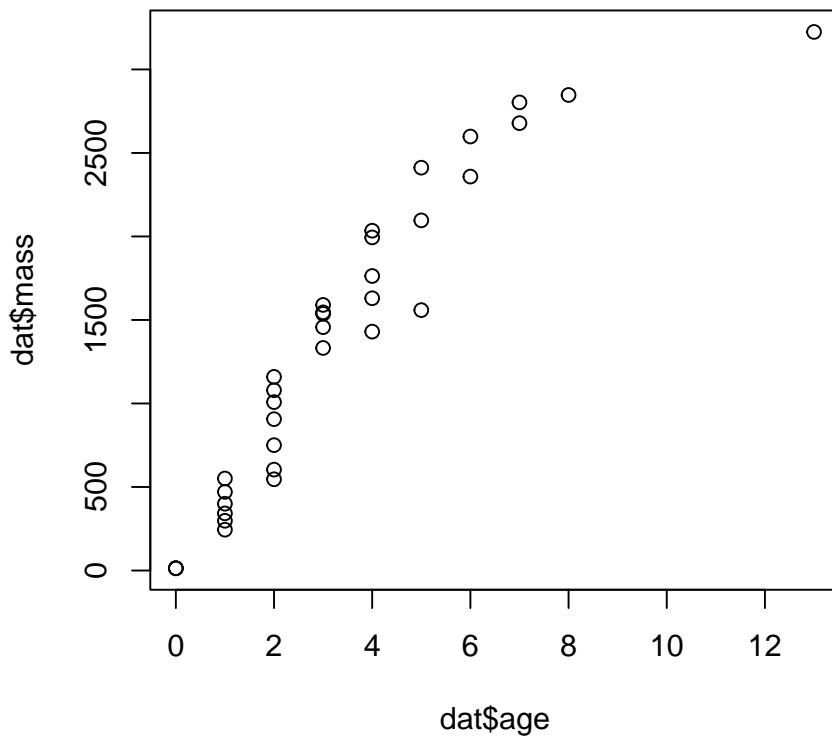
When should you use GAM? The most common application is situations where the response is not linearly related to the predictors, or if the response curve takes on a complicated shape. As you might suspect, the smoothers fitted in a GAM can be vulnerable to overfitting just as can be polynomial models or multiple regression models. Researchers usually control for this overfitting of the smoothers by limiting the complexity of the curves; this parameter is usually called the number of “knots”. Compared to NLS, GAMs have the significant advantage that the curve does not have to be defined ahead of time, and is thus much more flexible. The disadvantage is related to the advantage: unlike a LM, GLM, or NLS fit, a GAM fit will not result in model parameters (e.g., slopes) that can be interpreted.

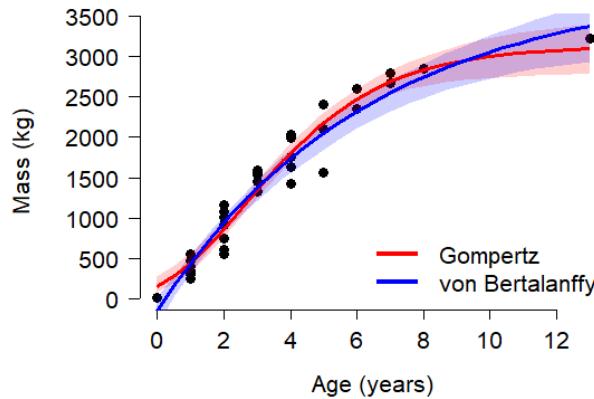
Two good references to get you started are Wood (2017) and Zuur et al. (2007). The best-known R package for fitting GAMs is mgcv (CRAN page, accessed 2021-09-17).

6.6.3.1 GAM example with real data

Let’s explore GAMs by re-analyzing one of the NLS models we saw earlier. Download the dataset here and put it in your R working directory.

```
in.name <- "woodward2015data.csv"
dat <- read.csv(in.name, header=TRUE)
plot(dat$age, dat$mass)
```





If we didn't know about the Gompertz growth model, or had no idea what curve to start with, we could start by fitting GAMs to get a sense of the kind of curve we need. This is marginally better than just plotting the data and eyeballing a curve. Fit the GAMs with the function `gam()`.

```
library(mgcv)

## Loading required package: nlme

## This is mgcv 1.8-36. For overview type 'help("mgcv-package")'.
mod1 <- gam(mass~s(age), data=dat)
summary(mod1)

##
## Family: gaussian
## Link function: identity
##
## Formula:
## mass ~ s(age)
##
## Parametric coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 1351.14     34.37   39.31  <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Approximate significance of smooth terms:
##             edf Ref.df    F p-value
## s(age) 2.887  3.535 182.7  <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
```

```
## R-sq.(adj) = 0.95 Deviance explained = 95.4%
## GCV = 46522 Scale est. = 41356 n = 35
```

Notice that the usual coefficients table is split into two parts: one for the intercept, and another for the smoother terms. The `s(age)` term in the model formula is how we requested a smoother on age. Without the `s()` function, we would get a linear term. The model summary also returns a pseudo- R^2 based on deviance.

Next, we get the model predictions in the usual way:

```
px <- seq(min(dat$age), max(dat$age), length=100)
pred <- predict(mod1, newdata=data.frame(age=px),
                 type="response", se.fit=TRUE)
```

The values in `pred` reveal a problem: the model predicts negative values! We probably should have used a log-link function to ensure positivity in the response.

```
mod2 <- gam(mass~s(age), data=dat, family=gaussian(link="log"))
summary(mod2)
```

```
##
## Family: gaussian
## Link function: log
##
## Formula:
## mass ~ s(age)
##
## Parametric coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 6.94435   0.05751 120.7 <2e-16 ***
## ---
## Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Approximate significance of smooth terms:
##          edf Ref.df F p-value
## s(age) 5.492 6.532 48.21 <2e-16 ***
## ---
## Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## R-sq.(adj) = 0.949 Deviance explained = 95.7%
## GCV = 52451 Scale est. = 42723 n = 35
```

Now make the predictions but on the link scale, which we can back-transform.

```
px <- seq(min(dat$age), max(dat$age), length=100)
pred <- predict(mod2, newdata=data.frame(age=px),
                 type="link", se.fit=TRUE)
mn <- pred$fit
```

```

lo <- exp(qnorm(0.025, mn, pred$se.fit))
up <- exp(qnorm(0.975, mn, pred$se.fit))
mn <- exp(mn)

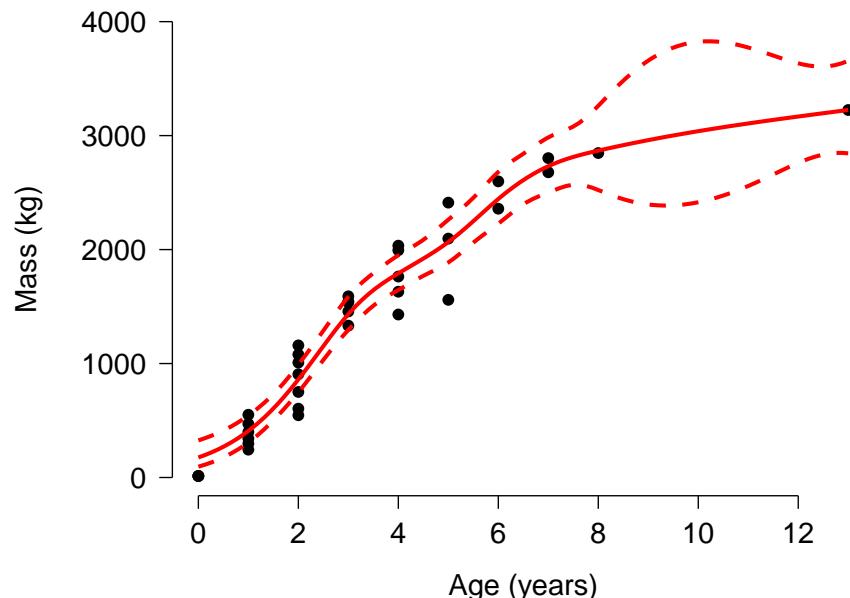
```

Now make a plot. Compare this to the equivalent plot from the Gompertz model fit above.

```

par(mfrow=c(1,1), mar=c(5.1, 7.1, 1.1, 1.1),
    las=1, lend=1, bty="n",
    cex.axis=1.3, cex.lab=1.3)
# make the plot
plot(dat$age, dat$mass,
      pch=16, cex=1.2,
      xlab="Age (years)",
      ylab="", ylim=c(0,4000))
title(ylab="Mass (kg)", line=5)
points(px, lo, type="l", col="red", lwd=3, lty=2)
points(px, up, type="l", col="red", lwd=3, lty=2)
points(px, mn, type="l", col="red", lwd=3)

```



Up to about 8 years of age, the GAM fit and the Gompertz fit look similar. However, above 8 years the GAM CI starts behaving strangely. This is because there are few data in that region with which to train the model. We could attempt to restrain the predictions by tweaking the model. One way is to reduce

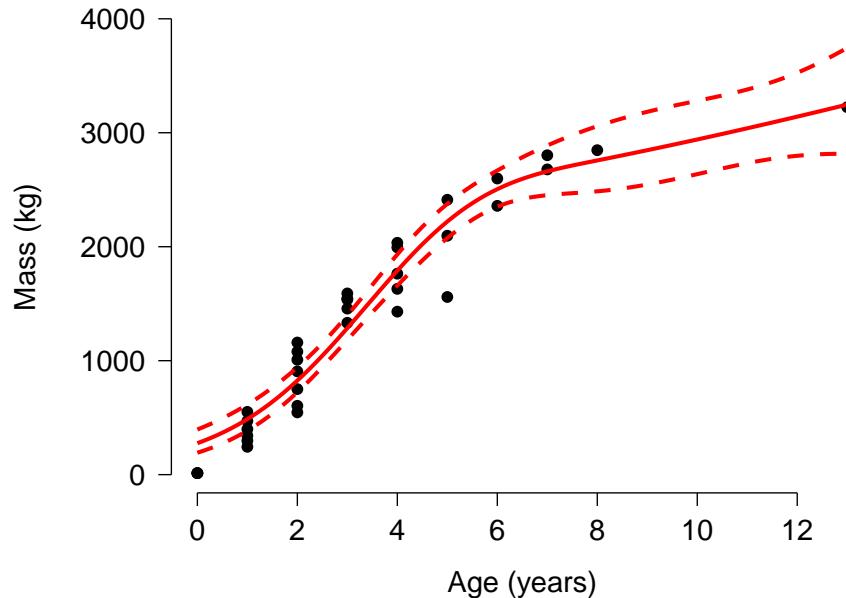
the number of “knots”, or bends in the smoother. This is done by adding an argument k to the `s()` function. Usually k should be ≥ 2 and $< n$, where n is the number of observations. Try changing k in the commands below and see how the predictions change.

```
mod3 <- gam(mass~s(age, k=4), data=dat, family=gaussian(link="log"))
summary(mod3)

##
## Family: gaussian
## Link function: log
##
## Formula:
## mass ~ s(age, k = 4)
##
## Parametric coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 6.99435   0.05263   132.9   <2e-16 ***
## ---
## Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Approximate significance of smooth terms:
##          edf Ref.df   F p-value
## s(age) 2.937 2.997 86.4   <2e-16 ***
## ---
## Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## R-sq.(adj) = 0.933 Deviance explained = 93.8%
## GCV = 63276 Scale est. = 56159 n = 35
px <- seq(min(dat$age), max(dat$age), length=100)
pred <- predict(mod3, newdata=data.frame(age=px),
                 type="link", se.fit=TRUE)
mn <- pred$fit
lo <- exp(qnorm(0.025, mn, pred$se.fit))
up <- exp(qnorm(0.975, mn, pred$se.fit))
mn <- exp(mn)

par(mfrow=c(1,1), mar=c(5.1, 7.1, 1.1, 1.1),
    las=1, lend=1, bty="n",
    cex.axis=1.3, cex.lab=1.3)
# make the plot
plot(dat$age, dat$mass,
      pch=16, cex=1.2,
      xlab="Age (years)",
      ylab="", ylim=c(0,4000))
title(ylab="Mass (kg)", line=5)
```

```
points(px, lo, type="l", col="red", lwd=3, lty=2)
points(px, up, type="l", col="red", lwd=3, lty=2)
points(px, mn, type="l", col="red", lwd=3)
```



Selecting the optimal number of knots is bit of an art as well as a science. Using too many knots places your model at risk of overfitting. Using too few knots risks not capturing a biologically relevant pattern. The best option is probably to fit the model using different numbers of knots and comparing them using AIC or predictive performance. For example:

```
mod.1 <- gam(mass~s(age), data=dat, family=gaussian(link="log"))
mod.2 <- gam(mass~s(age, k=2), data=dat, family=gaussian(link="log"))

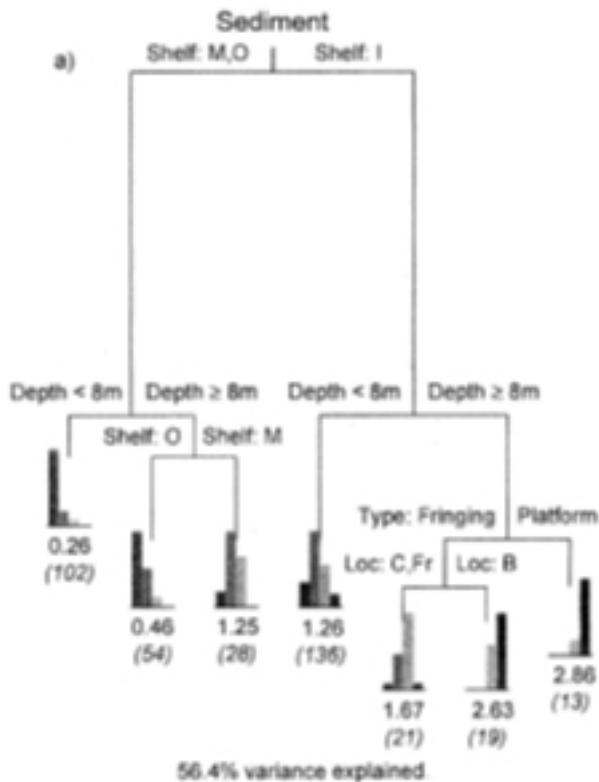
## Warning in smooth.construct.tp.smooth.spec(object, dk$data, dk$knots): basis dimensions
## do not match: 10 and 7
mod.3 <- gam(mass~s(age, k=3), data=dat, family=gaussian(link="log"))
mod.4 <- gam(mass~s(age, k=4), data=dat, family=gaussian(link="log"))
mod.5 <- gam(mass~s(age, k=5), data=dat, family=gaussian(link="log"))
mod.6 <- gam(mass~s(age, k=6), data=dat, family=gaussian(link="log"))
mod.7 <- gam(mass~s(age, k=7), data=dat, family=gaussian(link="log"))
AIC(mod.1, mod.2, mod.3, mod.4, mod.5, mod.6, mod.7)

##           df      AIC
## mod.1 7.491541 480.3155
## mod.2 3.984267 501.6689
```

```
## mod.3 3.984267 501.6689
## mod.4 4.936648 487.7807
## mod.5 5.896483 480.4869
## mod.6 6.851747 477.4232
## mod.7 7.161195 478.5131
```

6.6.4 Classification and regression trees (CART)

The final alternative to NLS that we will explore is **classification and regression trees (CART)**. CART is really a family of methods that use nonparametric partitioning to either place observations into categories (classification trees) or predict their numeric values (regression trees). A typical regression tree is shown below (De'ath and Fabricius 2000):



The tree predicts sediment thickness around coral reefs (as an ordinal variable) using two factors (shelf and reef type) and a numerical predictor (depth). The expected value for each observation is determined by going down the tree, selecting the correct statement about that observation. For example, an observation on the outer shelf (shelf = O), at 12 m depth, would have expected sediment thickness 0.46 (so values 0 or 1 would be most common). Each decision point is

called a “node”; the predicted values are at the terminal nodes (or “leaves”). In this tree, the distribution of response values at each leaf is shown.

Tree-based models can describe patterns where there is no simple data model (e.g., curve shape), or where there are complex interactions between predictor variables (interactions manifest as effects being nested within each other). Tree-based models have the disadvantage that they require large amounts of data and can be difficult to present and interpret. Worse, it can be difficult to distinguish between a tree-based model that explains a high proportion of variation in the data, and a model that is badly overfit (i.e., fitting random noise). Cross-validation can help with this problem but is no guarantee of a robust fit.

Another name for these kinds of tree-based models is “recursive partitioning”. This means that the models partition the data over and over until homogenous groups are found. The main R package for CART is `rpart` (Therneau and Atkinson 2019).

6.6.4.1 Classification tree example with real data

Hutcheon et al. (2002) reported data on brain morphology and lifestyle from 63 species of bats. Their dataset contains the following variables:

Variable	Meaning
<code>species</code>	Species
<code>family</code>	Taxonomic family
<code>diet</code>	Herbivore, gleaner, hawker, or vampire
<code>bow</code>	Body weight (g)
<code>brw</code>	Brain weight (μg)
<code>aud</code>	Auditory nuclei volume (mm^3)
<code>mob</code>	Main olfactory bulb volume (mm^3)
<code>hip</code>	Hippocampus volume (mm^3)

Import the dataset `bat_data_example.csv`. You can download it [here](#). The code below requires that you have the file in your R working directory.

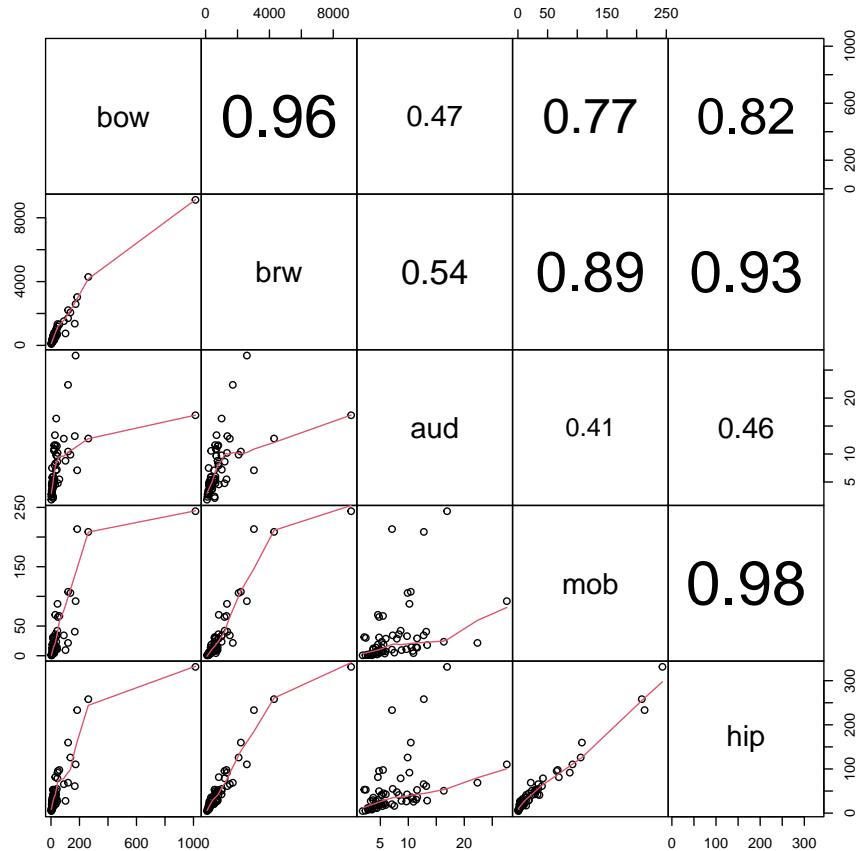
```
in.name <- "bat_data_example.csv"
dat <- read.csv(in.name, header=TRUE)
meas.cols <- 4:ncol(dat)
```

We are interested in whether the relative sizes of different brain components are related to diet. Statistically, we are testing whether any combination of brain component sizes can correctly classify diet type.

Let’s briefly explore the data. First, we can see how the variables are related to each other. We can do that with a scatterplot matrix.

```
panel.cor <- function(x, y, digits = 2,
                      prefix = "", cex.cor, ...){
  usr <- par("usr"); on.exit(par(usr))
  par(usr = c(0, 1, 0, 1))
  r <- cor(x, y, use="complete.obs")
  txt <- format(c(r, 0.123456789), digits = digits)[1]
  txt <- paste0(prefix, txt)
  if(missing(cex.cor)) cex.cor <- 0.8/strwidth(txt)
  text(0.5, 0.5, txt, cex = cex.cor * abs(r))
}

pairs(dat[,meas.cols],
      lower.panel = panel.smooth,
      upper.panel = panel.cor,
      gap=0)
```



What these correlations mean is that the variables are redundant with each other, and with overall size (body mass, `bow`). This makes a lot of sense for morphometric data. We need to keep this in mind as we analyze the data.

It might make sense to log-transform the data because we are working with measurements (which cannot be negative). Log-scale data will also be more normally distributed, which is not necessary for CART but can be more convenient.

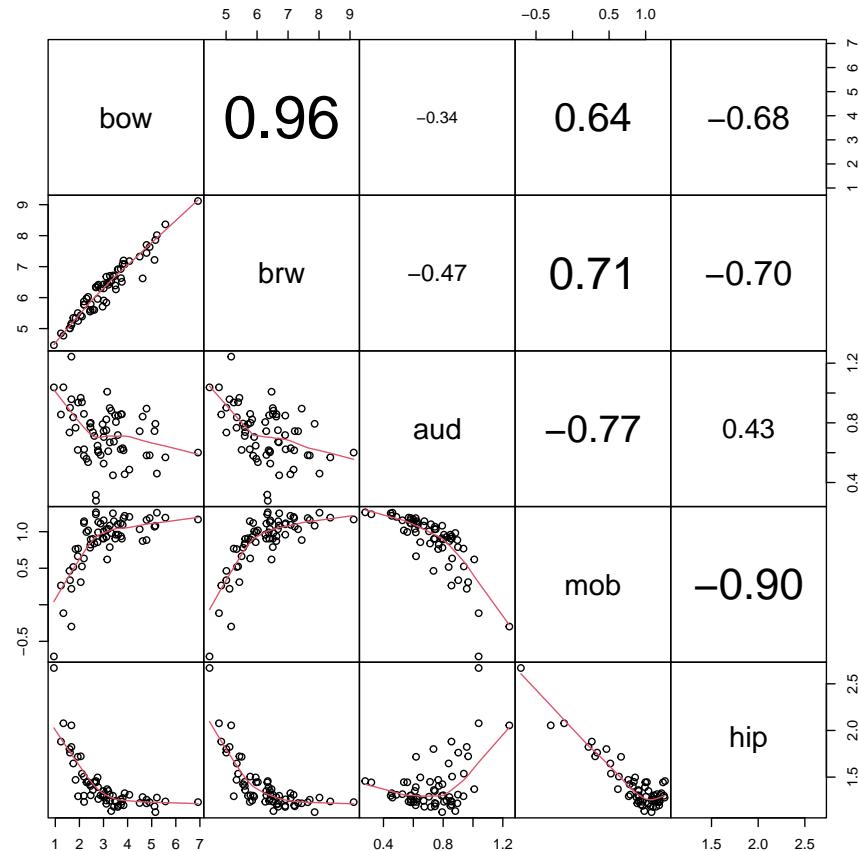
```
# log transform
dat[,meas.cols] <- apply(dat[,meas.cols], 2, log)
```

In addition, we should we can factor out the effect of overall brain size. This will make it so that any patterns we see are not just a result of brain size. Below we use the geometric mean of the three brain volumes as the “total brain size” because the variable `brw` contains many other components (and is a mass, not a volume). Note that the mean of the log-transformed values is really the geometric mean, not the arithmetic mean.

```
vols <- meas.cols[-c(1:2)]
dat$geom <- apply(dat[,vols], 1, mean)
dat[,vols] <- apply(dat[,vols], 2, function(x){x/dat$geom})
```

Remake the scatterplot matrix and observe that the three brain volumes are no longer so strongly correlated.

```
pairs(dat[,meas.cols],
      lower.panel = panel.smooth,
      upper.panel = panel.cor,
      gap=0)
```



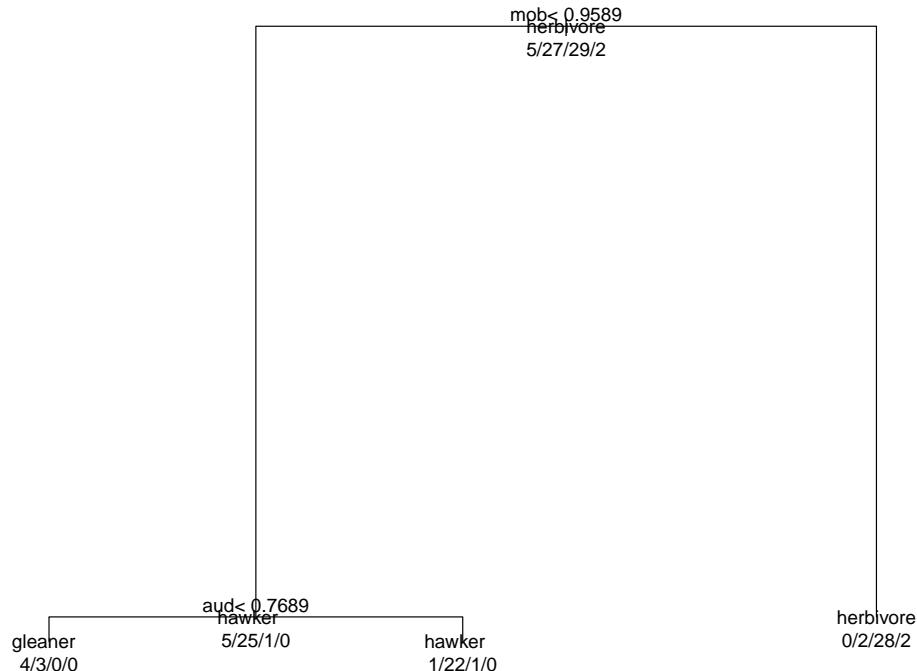
Now we should be ready to fit a classification tree. The syntax is simple (and similar to a GLM):

```

library(rpart)
set.seed(123)
mod1 <- rpart(diet~bow+aud+mob+hip, data=dat)

par(mfrow=c(1,1), mar=c(0.5, 0.5, 0.5, 0.5), xpd=NA)
plot(mod1)
text(mod1, use.n=TRUE, all=TRUE)

```



The default plot isn't very attractive, but it does the job. A nicer version is hard to make in R, but can be assembled in Powerpoint or your favorite drawing program very easily. The tree summarizes graphically how the algorithm partitioned the data:

- If $mob \geq 0.9589$, classify as herbivore.
- If $mob < 0.9589$ AND $aud < 0.7689$, classify as gleaner.
- If $mob < 0.9589$ AND $aud \geq 0.7689$, classify as hawker (aerial predator).

For each observation, start at the top and go LEFT if the statement at a node is true, and right if the statement is false. The numbers at each terminal node show the number of observations from each category that wound up at that node. For example, 32 observations were classified as herbivores, and 28 of them were correct (0/2/28/2). We can also see that 22 out of 24 observations classified as hawks by the model were actually hawks. The confusion matrix for the model is obtained by making a frequency table of the predicted classifications and the

actual classifications. The table below puts the predictions on the columns and the actual diets on the rows.

```
pred <- predict(mod1, type="class")
ftable(pred~dat$diet)
```

	pred	gleaner	hawker	herbivore	vampire
## dat\$diet					
## gleaner	4	1	0	0	
## hawker	3	22	2	0	
## herbivore	0	1	28	0	
## vampire	0	0	2	0	

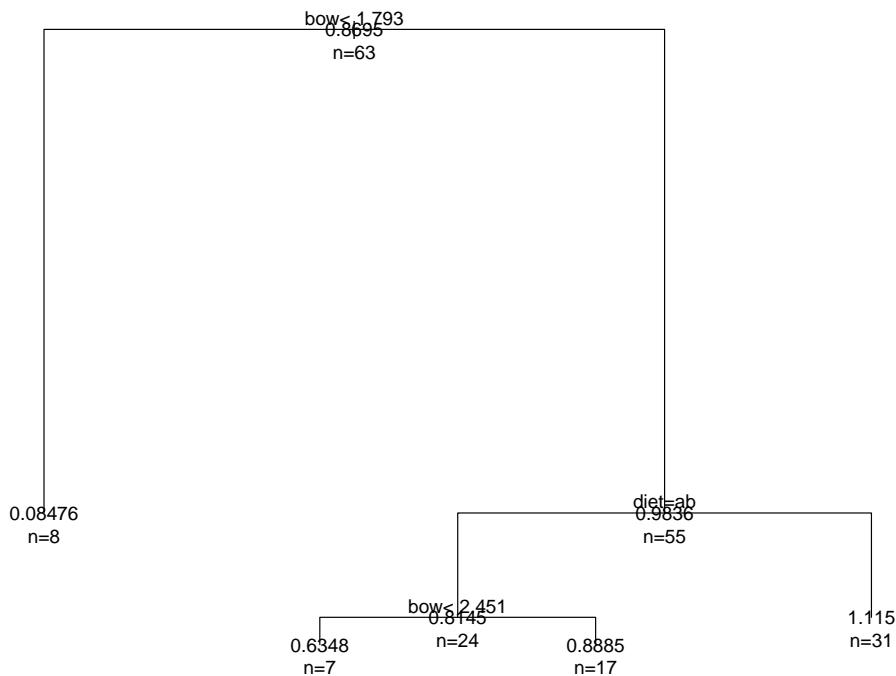
6.6.4.2 Regression tree example with real data

We'll use the bat data again to construct a simple regression tree. Can we predict main olfactory bulb volume from diet and other facts about a species? Load the bat data from above and go through the steps of log-transforming and relativizing to overall mass.

Now fit the model and plot it:

```
mod2 <- rpart(mob~diet+bow, data=dat)

par(mfrow=c(1,1), mar=c(0.5, 0.5, 0.5, 0.5), xpd=NA)
plot(mod2)
text(mod2, use.n=TRUE, all=TRUE)
```



The regression tree predicts the mean response variable at each terminal node. The logic of how to follow the tree to classify each observation is similar to before. At each node, go left when the statement is true and right when it is false. Factor levels are identified by letters corresponding to their order (usually alphabetical; check `levels(factor(dat$diet))`). Keep in mind that the values are all on the log scale, because we transformed the values. Another option would have been to standardize the values.

The interpretation of this tree is as follows:

- Overall mean = 0.8695, $n = 63$ (top node)
- $\text{bow} < 1.793 \rightarrow 8$ observations with mean = 0.08476
- $\text{bow} \geq 1.793$ and diet = gleaner (a) or hawker (b) $\rightarrow 31$ observations with mean = 1.115
- $\text{bow} \geq 1.793$ and diet \neq gleaner or hawker \rightarrow and $\text{bow} < 2.451 \rightarrow 7$ observations with mean = 0.6348
- $\text{bow} \geq 1.793$ and diet \neq gleaner or hawker \rightarrow and $\text{bow} \geq 2.451 \rightarrow 17$ observations with mean = 0.8885

We can visually inspect the effectiveness of the tree by plotting the predicted vs. observed values:

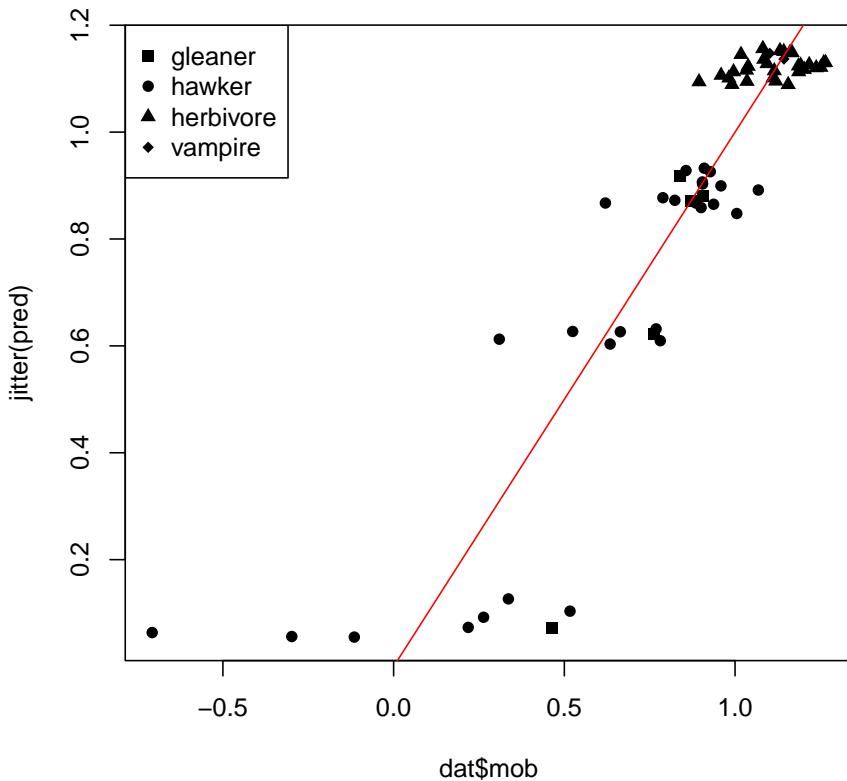
```

pred <- predict(mod2, type="vector")
pchs <- 15:18
  
```

```

diets <- sort(unique(dat$diet))
use.pch <- pchs[match(dat$diet, diets)]
plot(dat$mob, jitter(pred), pch=use.pch)
abline(a=0, b=1, col="red")
legend("topleft", legend=diets, pch=pchs)

```



Remember that regression trees only predict the mean within each group, not individual values. A good regression tree will have a relatively small spread within each group; the tree we just fit is not very good.

Better results can be obtained with random forests (Cutler et al. 2007, Brieuc et al. 2018) and boosted regression trees (BRT) (De'ath 2007, Elith et al. 2008). BRT use not one but many trees to obtain a more robust fit. Both of those techniques are powerful and underutilized in biology... but that is changing. At some point I'll produce a page demonstrating BRT. James et al. (2013) contains

primers on these and other machine learning techniques.

Chapter 7

Mixed models

7.1 Prelude (GLM)

So far in this course we have explored ways to model data by breaking them down into a deterministic part (that defines the expected value) and a stochastic part (that defines the variability or uncertainty). This culminated in the generalized linear model (GLM), a powerful framework for analyzing biological data.

Sometimes our data contain variability that we want to account for, without necessarily explaining. Some of this variability is described by **random effects**. Random effects describe variation due to unobserved or uncontrolled factors. As we will see below, random effects have a very different mathematical relationship to the data than the more familiar **fixed effects** such as intercept and slope. Models that contain both fixed and random effects are called **mixed effects models**, **hierarchical models**, or **multi-level models**. Just like GLMs, we will introduce mixed effects models as if they were extensions of ordinary linear models.

In the previous module we explored the generalized linear model (GLM), which unites many previously separate models into a single framework. A GLM is used to describe a system where there is a linear relationship between the predictor variables and some transformed version of the response variable, and a probability distribution that accounts for variability about the expected value of the response. This is a flexible generalization of the linear model. The linear model is presented below in the format of a GLM to illustrate the basic idea:

$$Y \sim Normal(\mu, \sigma^2)$$

$$\mu = \eta$$

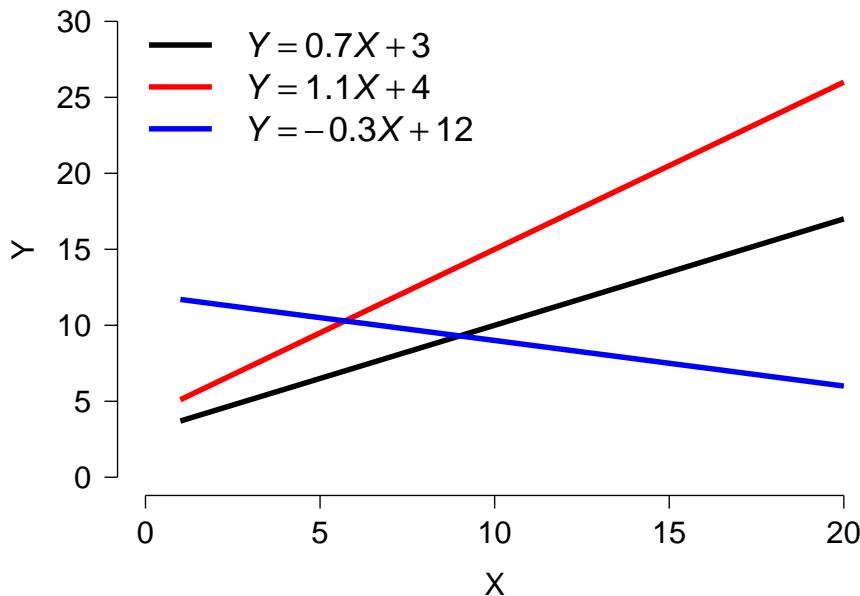
$$\eta = \beta_0 + \beta_1 X$$

In a GLM there is a linear predictor η (“eta”) that is a linear function of the predictor variable (or variables). The expected value of Y , μ , is related to η by a deterministic and invertible function (in this case, identity). The actual values of Y come from a probability distribution defined by the expected value μ and potentially other parameters (σ^2 , in this case). The equation for the linear predictor is also called the **deterministic part** of the model, and the expression for the distribution of the response variable is called the **stochastic part** of the model. Understanding how to think about statistical models in the language of their deterministic and stochastic parts is the key to understanding mixed models.

7.2 Linear mixed models (LMM)

A LM or GLM like the linear regression model shown above has exactly 1 stochastic component: the distribution of Y with parameters μ and σ^2 . In the absence of an interaction term, a LM or GLM implicitly assumes that the deterministic part of the model is the same for all observations, and for all groups of observations. In reality, this is not always the case. The relationship between a response Y and predictor X might have a different slope within different experimental populations, or cell strains, or plant cultivars... pick any grouping variable you like. Or, the intercept of the relationship might vary by group. Even worse, the intercept *and* slope might vary by group¹. The figure below shows a dataset where each group has a separate intercept and slope.

¹Theoretically, the intercept and slope could each vary by different grouping variables. However, in my opinion, you would have to work very hard to justify that on biological grounds.



As a researcher you may or may not be interested in how the slopes and intercepts differ between groups. If the slopes and intercepts associated with particular groups are of interest, then you should include the interaction between that grouping variable and other variables—i.e., an analysis of covariance (ANCOVA) with interaction.

However, sometimes the variation between groups is variation that we are not interested in. This might be because the groups themselves are not of primary interest. For example, a botanist who measures growth rates from five individual trees is probably not interested in fitting growth constants specific to those particular trees. But, she might still want to account for variability between trees in her dataset. In that case, “tree” would be a random effect in her analysis. In her model, the botanist would let the growth rate vary randomly between trees.

As noted by Bolker (2008), the distinction between fixed and random effects is often very murky and the subject of intense (and confusing) debates among statisticians. The table below summarizes some of the key differences between fixed and random effects. See the footnote² for some good online and offline discussions of the meanings of fixed and random effects.

²Discussion on Stack Overflow, with references. Blog post from Andrew Gelman, prominent statistician. Mixed models are presented and described thoroughly in Bolker (2008), Bolker et al. (2009), and Zuur et al. (2009a).

Condition	Fixed effects	Random effects
Research objective	Interested in predicting response for these specific levels of factor.	Interested in predicting response while accounting for variation between levels of factor.
Assumption about parameters	Coefficients fixed for all groups or individuals.	Coefficients come from a random distribution.
Hypothesis testing	Can get a P -value for coefficient estimates	Cannot formally test that coefficients different from 0.
Origin of variation	Values of factor set by experimenter	Values of factor drawn at random from underlying population.

When should you use a random effect in your analysis? The short answer is when some of the conditions in the right column apply to your data. As long as your choice is biologically defensible and statistically defensible, it is probably okay. There is no single agreed-upon standard for when a factor should be treated as a fixed or random effect.

There is one more aspect of random effects that is less widely appreciated than its implications for the variance structure. Fitting models with random effects between groups allows groups with fewer observations to “borrow” information from groups with more observations. The estimated group-level parameters (aka: BLUPs; see below) depend both on patterns within a group and on patterns that operate on all groups. This property of mixed models can be very useful for situations with unbalanced designs or where insufficient data are collected within all combinations of treatment factors³.

7.2.1 Formal definition and example

Mixed models can be described in several forms. Many biologists use a state-space format similar to the presentation of the GLM seen above. I prefer this form because it makes clear what the random effect is doing in the model: allowing the model intercept, slope, or both to vary random by a grouping factor. The form below also uses the observation-wise notation to make the model structure a little clearer.

$$Y_{ij} \sim \text{Normal}(\mu_{ij}, \sigma)$$

$$\mu_{ij} = \beta_0 + \beta_1 X_{ij} + a_i$$

³However, there is no such thing as a free lunch. Just because BLUPs, a.k.a. “conditional modes”, can be estimated for different levels of a factor does not mean that these estimates are equivalent to or as robust as analogous fixed effects.

$$a_i \sim \text{Normal}(0, \sigma_a)$$

In this model, response variable Y for observation j in group i is drawn from a normal distribution with mean μ_{ij} and standard deviation σ . The expected value is a linear function of a predictor variable X (measured for observation j in group i) and the effect of belonging to group i , a_i . The group effects are random, drawn from a normal distribution with mean 0 and SD σ_a . Notice that this model has the same slope for each group, but effectively different intercepts. The same model could also be written as:

$$\begin{aligned} Y_{ij} &\sim \text{Normal}(\mu_{ij}, \sigma) \\ \mu_{ij} &= \beta_{0,i} + \beta_1 X_{ij} \\ \beta_{0,i} &\sim \text{Normal}(\mu_{\beta_0}, \sigma_{\beta_0}) \end{aligned}$$

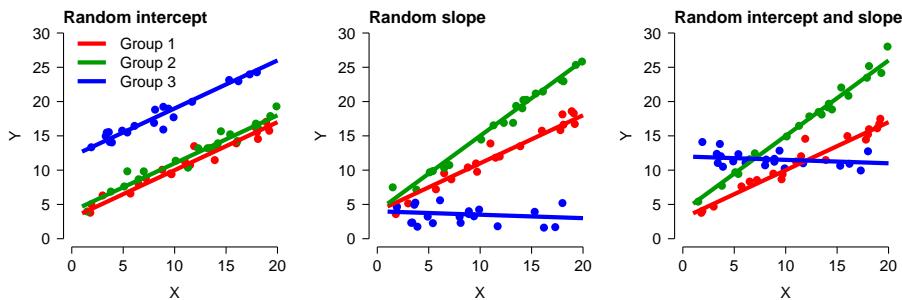
The model above, where the intercept varies randomly between groups, is sometimes called a **random intercept model**. A similar model could be defined that has a different slope in each group, but with a common intercept. Not surprisingly, this is called a **random slope model**:

$$\begin{aligned} Y_{ij} &\sim \text{Normal}(\mu_{ij}, \sigma) \\ \mu_{ij} &= \beta_0 + \beta_{1,i} X_{ij} \\ \beta_{1,i} &\sim \text{Normal}(\mu_{\beta_1}, \sigma_{\beta_1}) \end{aligned}$$

Or we can get crazy and have a model where both the intercept and slope are random variables:

$$\begin{aligned} Y_{ij} &\sim \text{Normal}(\mu_{ij}, \sigma) \\ \mu_{ij} &= \beta_{0,i} + \beta_{1,i} X_{ij} \\ \beta_{0,i} &\sim \text{Normal}(\mu_{\beta_0}, \sigma_{\beta_0}) \\ \beta_{1,i} &\sim \text{Normal}(\mu_{\beta_1}, \sigma_{\beta_1}) \end{aligned}$$

The figure below shows what these three types of models look like in practice.

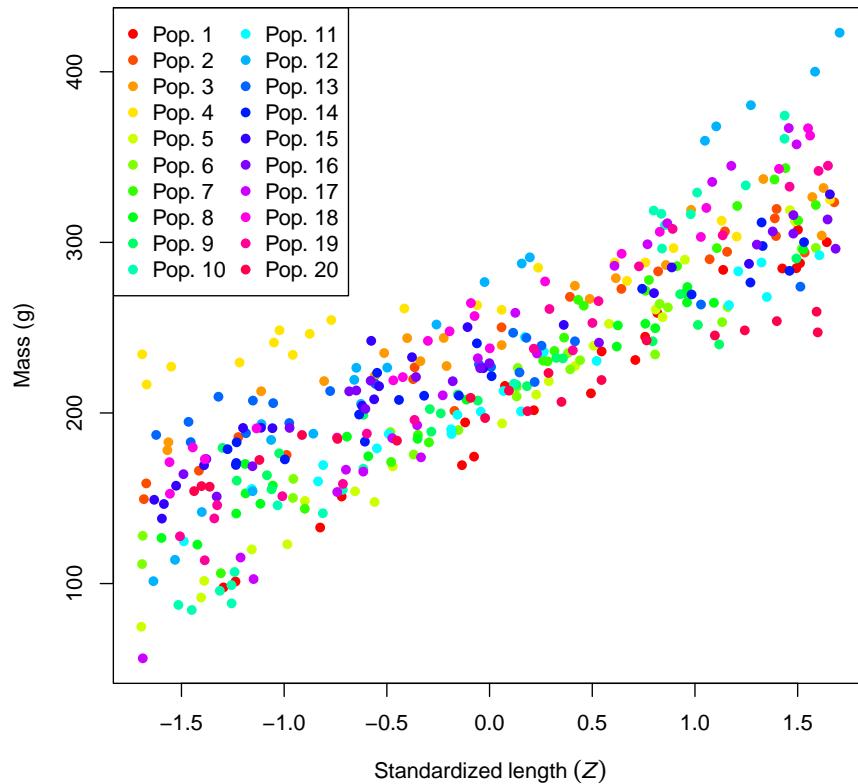


- In the random intercept model (left), all groups have the same slope but different intercepts. This looks visually like an ANCOVA without interaction, but there is a key difference: instead of being estimated for each group level by least squares, the intercepts of each group (i.e., the effects of being in each group) are assumed to come from a random distribution.
- In the random slope model (center), the groups all have the same intercept but have different slopes.
- In the random intercept and slope model (right), each group has its own intercept and slope. This looks visually like an ANCOVA with interaction, but with a key difference: instead of being estimated for each group level by least squares, the intercepts and slopes are assumed to come from a random distribution.

7.2.2 Example with simulated data

The best way to understand the structure of the LMM is to simulate some data suitable for this type of analysis. This example is adapted from one in chapter 12 of Kéry (2010), which was apparently adapted from Kéry (2002)⁴.

⁴The source of this example (Kéry 2010) used these data to illustrate fitting Bayesian mixed models with MCMC. I modified the data-generating parameters a bit to make the model more tractable by `lme4`. Kéry (2010) is one of the books I recommend to students interested in learning Bayesian statistics.



What a mess! It looks like population might affect mass, but it's hard to be sure. We can try fitting an ANCOVA model, but the sheer number of populations might make it difficult to get much out of it.

```
mod1 <- lm(mass ~ len.sc * factor(group), data=sim)
summary(mod1)

##
## Call:
## lm(formula = mass ~ len.sc * factor(group), data = sim)
##
## Residuals:
##      Min       1Q   Median       3Q      Max 
## -28.181  -5.755   0.236   6.404  31.993 
## 
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)    
## (Intercept) 230.000    1.000 230.000  <2e-16 ***
## len.sc      -1.000    0.000 -1.000  0.3173    
## factor(group)  0.000    0.000  0.000  0.0000    
##            ...
```

```

## (Intercept) 191.054 2.461 77.634 < 2e-16 ***
## len.sc       69.219 2.488 27.824 < 2e-16 ***
## factor(group)2 46.999 3.372 13.936 < 2e-16 ***
## factor(group)3 61.753 3.348 18.444 < 2e-16 ***
## factor(group)4 79.475 3.368 23.599 < 2e-16 ***
## factor(group)5 10.422 3.343 3.117 0.00197 **
## factor(group)6 16.211 3.361 4.824 2.08e-06 ***
## factor(group)7 22.669 3.596 6.304 8.48e-10 ***
## factor(group)8 21.630 3.347 6.461 3.37e-10 ***
## factor(group)9 22.693 3.341 6.793 4.56e-11 ***
## factor(group)10 34.157 3.342 10.220 < 2e-16 ***
## factor(group)11 17.508 3.348 5.230 2.89e-07 ***
## factor(group)12 78.423 3.372 23.255 < 2e-16 ***
## factor(group)13 39.791 3.486 11.414 < 2e-16 ***
## factor(group)14 38.247 3.338 11.458 < 2e-16 ***
## factor(group)15 48.387 3.493 13.851 < 2e-16 ***
## factor(group)16 42.141 3.338 12.625 < 2e-16 ***
## factor(group)17 35.137 3.338 10.526 < 2e-16 ***
## factor(group)18 66.286 3.338 19.857 < 2e-16 ***
## factor(group)19 36.231 3.338 10.853 < 2e-16 ***
## factor(group)20 14.415 3.343 4.312 2.09e-05 ***
## len.sc:factor(group)2 -18.619 3.195 -5.827 1.26e-08 ***
## len.sc:factor(group)3 -23.177 3.378 -6.861 3.00e-11 ***
## len.sc:factor(group)4 -40.542 3.320 -12.210 < 2e-16 ***
## len.sc:factor(group)5 6.079 3.429 1.773 0.07713 .
## len.sc:factor(group)6 -16.020 3.464 -4.625 5.23e-06 ***
## len.sc:factor(group)7 10.699 3.766 2.841 0.00475 **
## len.sc:factor(group)8 -15.346 3.347 -4.584 6.29e-06 ***
## len.sc:factor(group)9 -27.713 3.711 -7.468 6.22e-13 ***
## len.sc:factor(group)10 27.793 3.236 8.588 2.71e-16 ***
## len.sc:factor(group)11 -16.138 3.618 -4.460 1.09e-05 ***
## len.sc:factor(group)12 20.837 3.307 6.301 8.64e-10 ***
## len.sc:factor(group)13 -42.091 3.598 -11.699 < 2e-16 ***
## len.sc:factor(group)14 -22.964 3.369 -6.816 3.95e-11 ***
## len.sc:factor(group)15 -18.634 3.391 -5.495 7.40e-08 ***
## len.sc:factor(group)16 -21.612 3.237 -6.677 9.27e-11 ***
## len.sc:factor(group)17 27.873 3.527 7.902 3.38e-14 ***
## len.sc:factor(group)18 -7.946 3.327 -2.389 0.01743 *
## len.sc:factor(group)19 2.053 3.316 0.619 0.53630
## len.sc:factor(group)20 -36.396 3.376 -10.781 < 2e-16 ***
## ---
## Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 10.09 on 360 degrees of freedom
## Multiple R-squared: 0.9779, Adjusted R-squared: 0.9756
## F-statistic: 409.3 on 39 and 360 DF, p-value: < 2.2e-16

```

The fitted model above has 20 intercepts and 20 slopes. Such a model will be difficult to make sense of because of how each factor changes the slope and intercept, and the fact that there is no overall effect of length on mass in the model (at least not explicitly). Before accepting this monstrosity, it's worth asking whether we really care about the difference in parameters between populations. After all, these 20 populations are not the only 20 populations of this species. They are a random subsample of a much larger set of populations. If we want our results to apply to populations that are not in our dataset, we need to take a different approach.

We can account for population-level differences with a linear mixed model. For this system we can fit 3 different models: a random intercept model, a random slope model, and a random slope and intercept model. The code below will do this using R package `lme4`.

```
# package for fitting LMM
library(lme4)
##
## Attaching package: 'lme4'
## The following object is masked from 'package:nlme':
##
##      lmList
## The following object is masked from 'package:gmm':
##
##      checkConv

# fit models
mod1 <- lmer(mass~len.sc+(1|group), data=sim)
mod2 <- lmer(mass~len.sc+(0+len.sc|group), data=sim)
mod3 <- lmer(mass~len.sc+(len.sc|group), data=sim)
```

Below is the summary for model 3. This is different than the output for `lm()` or `glm()`. The summary separates the estimates into the fixed effects and random effects.

```
summary(mod3)

## Linear mixed model fit by REML ['lmerMod']
## Formula: mass ~ len.sc + (len.sc | group)
##   Data: sim
##
## REML criterion at convergence: 3151.3
##
## Scaled residuals:
##     Min      1Q  Median      3Q     Max
## -2.7807 -0.5698  0.0403  0.6265  3.1840
##
## Random effects:
```

```

## Groups      Name        Variance Std.Dev. Corr
## group      (Intercept) 484.3    22.01
##             len.sc      438.5    20.94   -0.09
## Residual          101.7    10.09
## Number of obs: 400, groups: group, 20
##
## Fixed effects:
##               Estimate Std. Error t value
## (Intercept) 227.702     4.948   46.02
## len.sc       58.604     4.712   12.44
##
## Correlation of Fixed Effects:
##           (Intr)
## len.sc -0.085

```

- The fixed effects are presented as estimates with SE and a t-value (but no P -value... we'll come back to that later).
- The random effects are presented as the variance and SD of the random effects. The output above shows that the intercepts in model 3 had mean 0 and SD 22.01. This is pretty close to the true value of 20. Likewise, the slopes had mean 0 and SD = 20.94. This is not very close to the true value of 30.
- Across groups, the intercepts and slopes are not correlated with each other ($r = -0.09$). In other words, variation in intercept appears unrelated to variation in slope. This is fine, because that is how the data were simulated.
- Finally, the output tells us that the residual SD was 10.09, very close to the true value of 10.

We can extract the results of a mixed model in several ways. We might ask what the random effects are. These are the differences between the group-specific parameters and the mean parameter. E.g., the difference between the overall estimated intercept (227.702) and the intercepts estimated for each group. The random effects can be accessed with the command `ranef()`. This returns a list with one element for each factor treated as a random effect. The elements are named according to the name of the factor, so the command below will get us a data frame of the random effects on the intercept and slope of each group.

```

re <- ranef(mod3)$group
head(re)

```

```

##   (Intercept)  len.sc
## 1 -36.16209 10.330202
## 2  10.23061 -7.919170
## 3  24.83656 -12.408188
## 4  42.44908 -29.716640
## 5 -25.97902 16.531595
## 6 -20.22579 -5.352089

```

The output shows us that group 1 had an intercept 36.16 smaller than the overall intercept, and a slope 10.33 greater than the overall slope; group 2 had an intercept 10.23 larger than the overall intercept and a slope 7.92 smaller than the overall slope; and so on.

A more useful output is the one containing the estimated intercepts and slopes for each group, not just their differences from the overall parameters. These can be accessed using the command `coef()`. As before we will extract the component of the list by name.

```
blup <- coef(mod3)$group
head(blup)

##   (Intercept)  len.sc
## 1    191.5397 68.93397
## 2    237.9324 50.68460
## 3    252.5384 46.19558
## 4    270.1509 28.88713
## 5    201.7228 75.13536
## 6    207.4760 53.25168
```

The output tells us that group 1 had an intercept of 191.54 and a slope of 68.93; that group 2 had an intercept of 237.93 and slope of 50.68; and so on. The name of the object, `blup`, is short for **best unbiased linear predictor**. These are the estimates of the parameters for each group. The name BLUP is retained for historical reasons, although these values are not guaranteed to be “best” or “unbiased”⁵. Some authors prefer to call them **conditional modes**. This name denotes that they are the most likely value (“mode”), conditional on the data.

7.2.3 *P*-values in LMM

The authors of the `lme4` package made a conscious decision to not present *P*-values for model parameters. Their reasons are highly technical and statistically sound, but these facts are often lost on journal reviewers. To make a long story short, `lme4` does not calculate *P*-values because it is not always clear what the correct number of degrees of freedom a model has, and so it is not clear what a *P*-value would even represent. On top of this, approximating *P*-values is computationally intractable in many situations, but that’s not a real excuse. That being said, in later work the `lme4` authors provide several suggestions for obtaining *P*-values. We will try the easiest method. In a later example, we try MCMC sampling with JAGS to get another angle on a significance test.

If all you need to know is whether a fixed effect is different from 0, you can calculate the 95% CI using function `confint()`. This will estimate the 95% CI of each fixed effect. If that CI does not include 0, then the parameter is

⁵A famous statistician once remarked that BLUPs are like the Holy Roman Empire, because just as that was neither holy, nor Roman, nor an empire, a BLUP is not necessarily “best”, “linear”, or “unbiased”. I can’t find the quote anymore.

“statistically significant” even though you don’t know the P -value. The result below shows that both the intercept and slope of the model are likely to be statistically significant.

```
confint(mod3)
```

```
## Computing profile confidence intervals ...

##           2.5 %    97.5 %
## .sig01    16.1187537 30.3787564
## .sig02    -0.4902622  0.3495054
## .sig03    15.3285043 28.9189751
## .sigma     9.3916449 10.8699356
## (Intercept) 217.7777253 237.6268196
## len.sc     49.1511687 68.0536264
```

If you, your advisor, or a reviewer insist on getting P -values for a mixed effects model, there are some ways. The package `lmerTest` (Kuznetsova et al. 2017) provides utility functions for working with mixed models. Included are methods for getting P -values from `lmer()` objects. This package extends the methods in `lme4`, so it needs to be loaded *after* `lme4` in your workspace. The model fitting functions and syntax are the same. But a model fit using `lmerTest` can output P -values for the fixed effects.

```
library(lmerTest) # AFTER loading lme4
```

```
## Warning: package 'lmerTest' was built under R version 4.1.2

## 
## Attaching package: 'lmerTest'

## The following object is masked from 'package:lme4':
## 
##     lmer

## The following object is masked from 'package:stats':
## 
##     step

# refit models using lmerTest::lmer
## lmerTest version of lmer() will mask lme4 version of lmer()
mod1 <- lmer(mass~len.sc+(1|group), data=sim)
mod2 <- lmer(mass~len.sc+(0+len.sc|group), data=sim)
mod3 <- lmer(mass~len.sc+(len.sc|group), data=sim)
#summary(mod1)
#summary(mod2)
summary(mod3)

## Linear mixed model fit by REML. t-tests use Satterthwaite's method [
## lmerModLmerTest]
```

```

## Formula: mass ~ len.sc + (len.sc | group)
##   Data: sim
##
## REML criterion at convergence: 3151.3
##
## Scaled residuals:
##   Min     1Q Median     3Q    Max
## -2.7807 -0.5698  0.0403  0.6265  3.1840
##
## Random effects:
##   Groups   Name        Variance Std.Dev. Corr
##   group    (Intercept) 484.3    22.01
##           len.sc      438.5    20.94    -0.09
##   Residual             101.7    10.09
## Number of obs: 400, groups: group, 20
##
## Fixed effects:
##   Estimate Std. Error    df t value Pr(>|t|)
## (Intercept) 227.702     4.948 19.000 46.02 < 2e-16 ***
## len.sc      58.604     4.712 18.999 12.44 1.41e-10 ***
## ---
## Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Correlation of Fixed Effects:
##   (Intr) 
## len.sc -0.085

```

The estimates for the fixed and random effects are basically the same as before. What is different is that the P -value for the t -statistics are provided. If you want the random effects or the BLUPs, the `ranef()` and `coef()` methods will work as before.

7.2.4 Specifying random effects

In R, you can specify many kinds of random effects. The syntax for random effects models differs slightly between the two most popular mixed models packages, `lme4` and `nlme`. The table below shows some example random effects structures in `lme4`:

Formula	Meaning
(1 g)	Random intercept by grouping variable g. Single fixed slope for all groups.

Formula	Meaning
$(1 g1/g2)$	Random intercepts among grouping variables $g1$ and $g2$, with $g2$ nested within $g1$. Single fixed slope for all groups.
$(1 g1)+(1 g2)$	Random intercepts among $g1$ and $g2$, not nested. Single fixed slope for all groups.
$(x g)$	Random slope (effect of x) and intercept by grouping variable g . Intercepts and slopes can be correlated with each other.
$(x g)$	Random slope (effect of x) and intercept by grouping variable g . Intercepts and slopes are not assumed to be correlated with each other. Try this if a model with correlated random effects $((x g))$ cannot converge.
$(0+x g)$	Random slope by grouping variable g . Single fixed intercept for all groups.

In `lme4`, the random effects are added to a model formula as terms after the fixed effects. So, a model of y with an effect of continuous predictor x and random slope by group g would have formula $y \sim x + (0 + x | g)$. You can define a huge variety of random effects structures, but probably shouldn't. As with defining potential regression models or GLMs, it is incumbent on you as a biologist to think critically about what models represent biologically reasonable hypotheses.

The syntax for random effects in package `nlme` works a little differently. In `nlme`, the fixed and random parts of the model are defined by separate arguments, rather than as part of the model formula. The `nlme` equivalents to the structures above are:

Formula	Meaning
$\sim 1 g$	Random intercept by grouping variable g . Single fixed slope for all groups.
$\sim 1 g1/g2$	Random intercepts among grouping variables $g1$ and $g2$, with $g2$ nested within $g1$. Single fixed slope for all groups.
<code>list(~1 g1, ~1 g2)</code>	Random intercepts among $g1$ and $g2$, not nested. Single fixed slope for all groups.

Formula	Meaning
$\sim x g$	Random slope (effect of x) and intercept by grouping variable g .
$\sim 0 + x g$	Random slope by grouping variable g . Single fixed intercept for all groups.

The `nlme` equivalents to the example analysis above are:

```
# not run

library(nlme)
mod1.nlme <- lme(mass~len.sc, random=~1|group, data=sim)
mod2.nlme <- lme(mass~len.sc, random=~0+len.sc|group, data=sim)
mod3.nlme <- lme(mass~len.sc, random=~len.sc|group, data=sim)
```

7.2.5 LMM example with real data

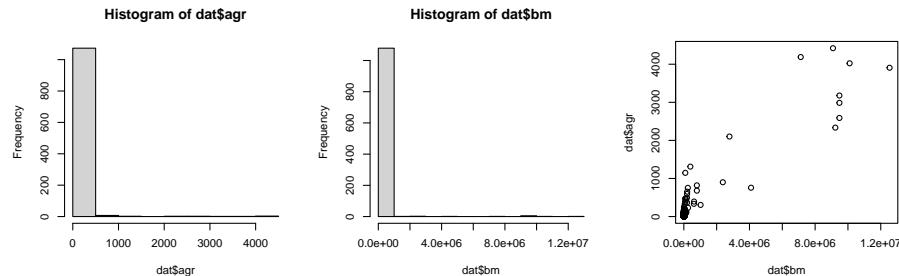
Despite a wealth of fossil remains and intensive study, many basic questions about the biology of the non-avian dinosaurs remain unanswered. Werner and Griebeler (2014) used data on growth rates in modern vertebrates to investigate whether dinosaurs were endothermic or ectothermic. They collected data on maximal growth rates and body size for 1072 extant species of birds, mammals, reptiles, and fishes, and compared these to 19 dinosaurs. They modeled maximum growth rate against the body size at which each species attained the maximal growth rate. The idea was that if dinosaurs were endothermic, the allometric relationship between their growth rate and body size would match that of modern-day endotherms. We will use their dataset and replicate part of their analysis.

Download the dataset and save it in your R home directory. Then import the data:

```
in.name <- "werner_2014_data_2021-10-19.csv"
dat <- read.csv(in.name, header=TRUE)
```

Next, let's take a look at the data.

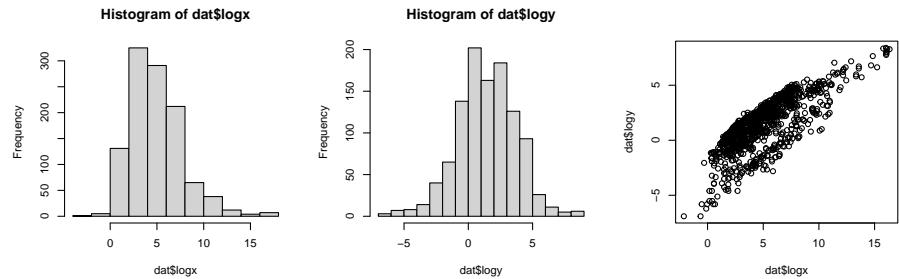
```
par(mfrow=c(1,3))
hist(dat$agr)
hist(dat$bm)
plot(dat$bm, dat$agr)
```



The histograms and the “dust bunny” pattern (i.e., little clump in the corner) in the scatterplot suggests that we should log-transform both variables.

```
dat$logr <- log(dat$agr)
dat$logx <- log(dat$bm)

par(mfrow=c(1,3))
hist(dat$logx)
hist(dat$logr)
plot(dat$logx, dat$logr)
```



The log transform did the trick. For the analysis, let’s work with the log-transformed data. If we want to see how the dinosaurs compare to the other taxa, we’ll need to identify them in the plot somehow. In the code below, we will select a color for each unique group, then manually set the color for dinosaurs to be black (so they stand out on the plot).

```
# identify groups of organisms
groups <- sort(unique(dat$group1))
ngroup <- length(groups)

# 40% opacity (alpha)
cols <- rainbow(ngroup, alpha=0.4)

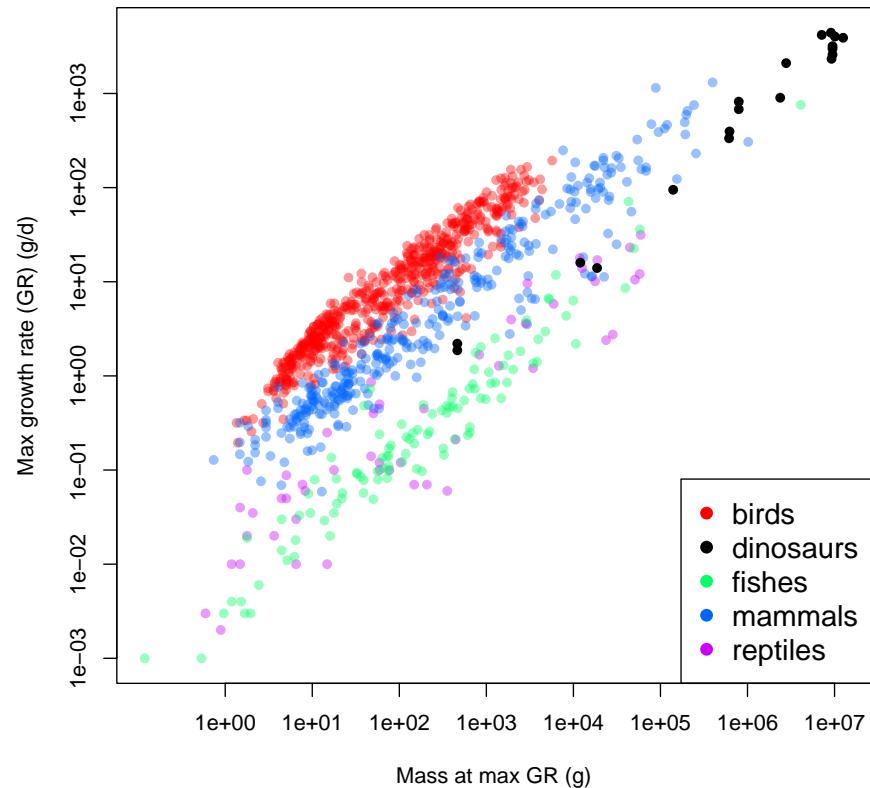
# same colors but solid (for legend)
```

```
cols2 <- rainbow(n group)

# change dinosaurs to black
cols[which(groups == "dinosaurs")] <- "black"
cols2[which(groups == "dinosaurs")] <- "black"

# assign colors to observations in data frame
dat$col <- cols[match(dat$group1, groups)]

# make the plot
par(mfrow=c(1,1))
plot(dat$bm, dat$agr,
      log="xy", # make x and y axes logarithmic
      xlab="Mass at max GR (g)",
      ylab="Max growth rate (GR) (g/d)",
      col=dat$col, pch=16)
legend("bottomright", legend=groups,
       pch=16, col=cols2, cex=1.3)
```



The figure suggests that, overall, larger animals have faster growth rates. This by itself is not too surprising. What is interesting is how the different groups (birds, dinosaurs, etc.) appear to form different parallel bands in the scatterplot. For any given body mass, birds had the highest growth rate, followed by mammals, then dinosaurs, then fishes and reptiles. Remember that we are interested in how the growth rates of dinosaurs compare to those of other groups. Statistically, this means that we want to see how the intercept and slope for dinosaurs compares to the same parameters for other taxa.

One approach would be to fit an ANCOVA model with mass as the predictor variable and taxon as a factor. The model could be fit with and without interaction, to see whether the intercept and slope really vary by taxon. However, before fitting the ANCOVA, it's worth asking whether these groups of vertebrates are really representative of vertebrate diversity. Sure, the list includes almost every extant class of Osteichthyes (bony fish). But, it ignores Chondrichthyes and Amphibia. It also ignores other extinct taxa, like Allotheria (extinct Mesozoic mammals), Temnospondyls (extinct large-bodied amphibians), and other groups

that might have been included. The groupings here also ignore other ways that the species might have been grouped. For example, should *Archaeopteryx* be counted as a bird or a dinosaur?⁶.

From these perspectives, the taxonomic groupings in this dataset might be considered a *random subsampling* of vertebrate groups. Emphasis on the “might” in the previous sentence. There are other, completely legitimate and scientific ways that the species could have been grouped. So, we probably don’t want to waste statistical power (degrees of freedom) estimating coefficients that are specific to these and only these groupings. A better strategy would be to let model parameters vary randomly by group, so we account for between-group variability, without tying our analysis to current opinions on taxonomy. So, the situation calls for mixed models rather than with ANCOVA.

The models to be fit are defined below. In each model G is growth rate, M is body mass at maximal growth rate, taxon is group, and β_0 and β_1 are the intercept and slope. ε represents the distribution of residuals with mean 0 and $SD = \sigma$.

Model	Description	Model formula
1	LMM with random intercept by taxon	$\log(G) = \beta_{0,Tax} + \beta_1 \log(M) + \varepsilon$
2	LMM with random slope by taxon	$\beta_{0,Tax} \sim Normal(\mu_{\beta_0}, \sigma_{\beta_0})$ $\log(G) = \beta_0 + \beta_{1,Tax} \log(M) + \varepsilon$
3	LMM with random intercept and slope by taxon	$\beta_{1,Tax} \sim Normal(\mu_{\beta_1}, \sigma_{\beta_1})$ $\log(G) = \beta_{0,Tax} + \beta_{1,Tax} \log(M) + \varepsilon$
4	LMM with uncorrelated random intercepts and slopes by taxon	$\beta_{0,Tax} \sim Normal(\mu_{\beta_0}, \sigma_{\beta_0})$ $\beta_{1,Tax} \sim Normal(\mu_{\beta_1}, \sigma_{\beta_1})$ $\log(G) = \beta_{0,Tax} + \beta_{1,Tax} \log(M) + \varepsilon$

⁶This is a bit like asking if humans should be counted as Primates or as Mammals.

The R code is straightforward. Notice that the first command (commented out) unloads package `lmerTest`. This package was loaded in the previous example so we could use its version of `lmer()`, which allowed us to calculate P -values. If you want the `lme4` version of `lmer()`, then you need to either unload `lmerTest`, or use the double-colon notation `::` to specify which package's `lmer()` function you want: `lme4::lmer()` or `lmerTest::lmer()`. This document was rendered with `lmerTest` still loaded, so `summary(mod4)` will return the `lmerTest` version with P -values.

```
# if desired:
# detach(package:lmerTest)

mod1 <- lmer(logy~logx+(1|group1), data=dat)
mod2 <- lmer(logy~logx+(0+logx|group1), data=dat)
mod3 <- lmer(logy~logx+(logx|group1), data=dat)

## Warning in checkConv(attr(opt, "derivs")), opt$par, ctrl = control$checkConv, :
## Model failed to converge with max|grad| = 0.00274806 (tol = 0.002, component 1)
mod4 <- lmer(logy~logx+(logx||group1), data=dat)
```

Notice that model 3 did not fit (failed to converge). This happens a lot in LMM with correlated random effects. If you think about it, assuming that random intercepts are correlated with random slopes “sneaks in” an extra parameter that has to be estimated: the correlation coefficient. There are plenty of situations where random effects should be correlated, but this may not be one of them. Because it did not converge, Model 3 can probably be safely discarded. That leaves us with models 1, 2, and 4. Model 4 is similar to model 3, but does *not* assume that the random intercepts and slopes are correlated with each other.

Use `summary()` to take a look at the models. We'll look at model 1..

```
summary(mod1)

## Linear mixed model fit by REML. t-tests use Satterthwaite's method [
## lmerModLmerTest]
## Formula: logy ~ logx + (1 | group1)
##   Data: dat
##
## REML criterion at convergence: 1850.6
##
## Scaled residuals:
##     Min      1Q  Median      3Q     Max 
## -4.0931 -0.5376  0.1150  0.6258  3.6284 
##
## Random effects:
##   Groups   Name        Variance Std.Dev. 
##   group1  (Intercept) 2.7725   1.6651 
```

```

## Residual           0.3079   0.5549
## Number of obs: 1091, groups: group1, 5
##
## Fixed effects:
##             Estimate Std. Error      df t value Pr(>|t|)
## (Intercept) -3.327e+00 7.466e-01 4.037e+00 -4.456    0.011 *
## logx        7.095e-01 6.596e-03 1.087e+03 107.574   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Correlation of Fixed Effects:
##       (Intr)
## logx -0.059

```

The `lmerTest` summary includes P -values for fixed effects: both the intercept and slope were statistically significant. The random intercepts had mean 0 and SD = 1.665.

Next, compare the models using AIC weights. Notice that model 3, which did not converge, was left out of the AIC weight calculation.

```

aic.df <- AIC(mod1, mod2, mod4)

# delta-AIC is difference from smallest AIC
aic.df$delta <- aic.df$AIC - min(aic.df$AIC)

# use delta-AIC to calculate AIC_wt
aic.df$wt <- exp(-0.5*aic.df$delta)
aic.df$wt <- aic.df$wt/sum(aic.df$wt)

# order to see models ranked by AIC_wt
aic.df <- aic.df[order(-aic.df$wt),]
aic.df

##      df      AIC      delta          wt
## mod4  5 1850.173  0.000000  9.856884e-01
## mod1  4 1858.638  8.464541  1.431159e-02
## mod2  4 2825.160 974.987148 1.896577e-212

```

The model with uncorrelated random effects of taxon, model 4, was by far the best-fitting model and the most likely of these 3 models to be the best-fitting one. We will interpret this model and plot its predictions in our write-up.

First, generate a data frame of taxon-specific X values at which to predict Y values. We'll generate the data frames in a loop and save them to a list. Then, the elements of the list will be combined to a single data frame with `do.call()`. This is a little extra work, but worth it because our figure will only show model predictions within the observed X range for each group.

```

# get X range for each group
agg <- aggregate(logx~group1, data=dat, range)

# number of values at which to predict
pn <- 30

# make a list containing a data frame for each group
px.list <- vector("list", ngroup)
for(i in 1:ngroup){
  imin <- agg[[2]][i,1]
  imax <- agg[[2]][i,2]
  ix <- seq(imin, imax, length=pn)
  px.list[[i]] <- data.frame(group1=groups[i], logx=ix)
}

# combine to a single data frame
dx <- do.call(rbind, px.list)

```

Next, generate predictions. The predict method for LMM does not generate SE; this turns out to be rather difficult mathematically. We'll plot the expected values first, then circle back and try to estimate some SE to get CI.

```

pred <- predict(mod4, newdata=data.frame(dx))
dx$y <- pred

```

The next block of code will produce some pretty axis tick marks. This is needed because the scales are logarithmic, but we want the *values* on the axes to be on the original scale. Manually constructing axes like this can be tedious (and a little confusing), but I think the payoff is worth it. The trick to remember that the plot is on the *natural log scale*, but the axes should be on the \log_{10} scale for interpretability. Most people intuitively understand 10^2 , 10^3 , etc., but e^2 , e^3 , and so on are not as clear.

The trick is to first define the values at which you want tick marks, THEN convert those values to natural log coordinates.

```

# main tick marks for powers of 10:
xax1 <- seq(-1, 7)
xax1 <- 10^xax1
xax1 <- log(xax1)
# manually specify some labels for tick marks
xticks <- c("0.1 g", "1 g", "10 g", "100 g",
           "1 kg", "10 kg", "100 kg", "1000 kg", "10000 kg")

# smaller tick marks for non-integer powers of 10
xax2 <- c(2:9/10, 2:9, 2:9*10, 2:9*100,
          2:9*1000, 2:9*1e4, 2:9*1e5, 2:9*1e6)

```

```
xax2 <- log(xax2)

# similar process for Y axis
yax1 <- seq(-3, 4, by=1)
yax1 <- log(10^yax1)
yax2 <- c(2:9/1000, 2:9/100, 2:9/10, 2:9,
        2:9*10, 2:9*100, 2:9*1000)
yax2 <- log(yax2)
yticks <- c(0.001, 0.01, 0.1, 1, 10, 100, 1000, 10000)
```

We should also change the group labels to be capitalized for the final figure. We could do this manually, but instead we'll write a custom R function to capitalize the words. There is a function provided on the help page for `tolower()` that can do this⁷:

```
capwords <- function(s, strict = FALSE) {
  cap <- function(s) paste(toupper(substring(s, 1, 1)),
                           {s <- substring(s, 2); if(strict) tolower(s) else s},
                           sep = "", collapse = " ")
  sapply(strsplit(s, split = " "), cap, USE.NAMES = !is.null(names(s)))
}

# now use the function
group.leg <- capwords(groups)
```

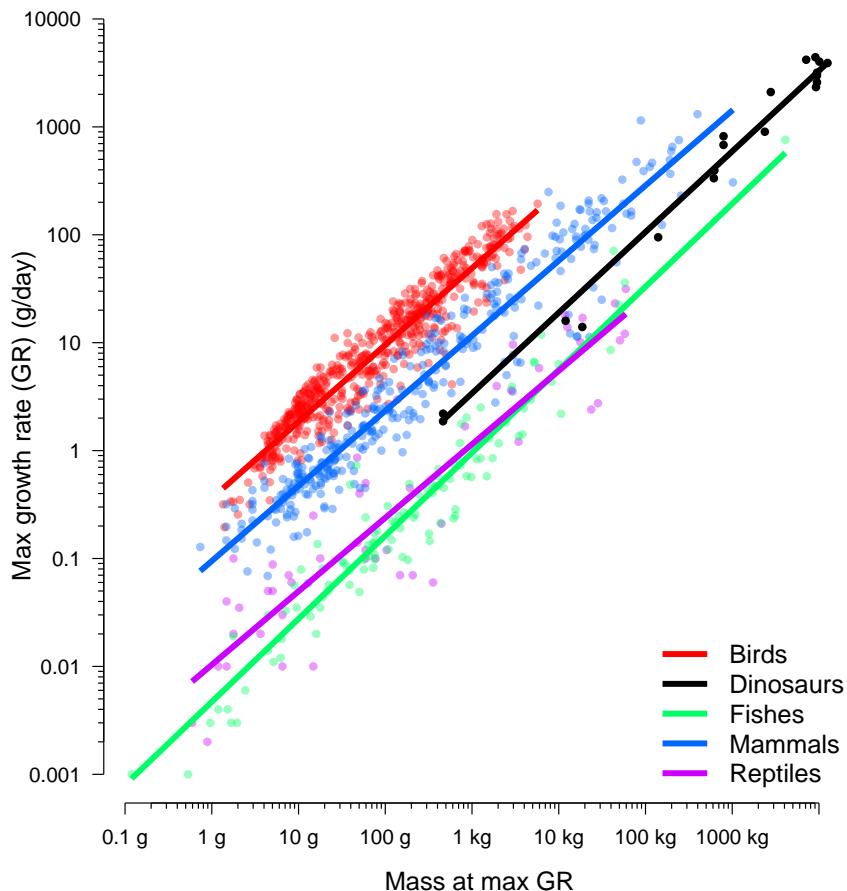
Now set some fancy plot options with `par()` and make the plot piece-by-piece⁸. Notice how the plot is set up with a log-log coordinate system (because x and y are given on log scale) and with no axes (`xaxt="n"` and `yaxt="n"`). The custom axes are added later. Then, a `for()` loop is used to plot the prediction lines for each group. Without that, we would have needed to write a separate `points()` command for each group. Finally, a legend shows the reader how to identify the groups.

```
par(mfrow=c(1,1), mar=c(5.1, 6.1, 1.1, 1.1),
    las=1, lend=1, bty="n", cex.axis=1.1, cex.lab=1.3,
```

⁷This goes to show why *reading the manual* is important. I can't count the number of times I've spent hours trying to program a custom solution to a problem, only to find that someone had already developed something much more elegant and robust and put it in the documentation.

⁸One of the things I love about R is the degree of control you have when making figures. Every single piece of a figure can be specified *exactly* the way you want it. This level of detail is very difficult (if not impossible) in Microsoft Excel. It can be approximated with graphics programs like Adobe Illustrator, but you'll never get the precision as you do when creating figure elements in R. Using R ensures that coordinate systems, coordinate values, colors, alignment, etc., are used precisely. Of course, all of this precision comes at a cost: complex figures can require a lot of code. In my workflow it's not unusual for really complicated figures to require >100 lines of code. This is partly because there really are a lot of elements to add individually, and partly because I'm a bit obsessive about getting figures just right.

```
xpd=NA)
plot(log(dat$bm), log(dat$agr), pch=16,
      col=dat$col,
      xaxt="n", yaxt="n",
      xlab="Mass at max GR",
      ylab="Max growth rate (GR) (g/day)")
axis(side=1, at=xax1,
      labels=xticks)
axis(side=1, at=xax2, labels=NA, tcl=-0.25)
axis(side=2, at=yax1,
      labels=yticks)
axis(side=2, at=yax2, labels=NA, tcl=-0.25)
for(i in 1:ngrup){
  flag <- which(dx$group == groups[i])
  points(dx$logx[flag], dx$y[flag], type="l", lwd=5, col=cols2[i])
}
legend("bottomright", legend=group.leg,
      lwd=5, col=cols2, cex=1.3, bty="n")
```



The final model lets us see that dinosaur growth rates are somewhere in between those of modern endotherms and modern ectotherms. We can compare the coefficients directly by examining the BLUPs. The result shows us that dinosaurs have an intercept and slope intermediate to the other groups.

```
coef(mod4)$group1
```

```
##              (Intercept)      logx
## birds      -1.019030  0.7111834
## dinosaurs -3.940582  0.7474059
## fishes     -5.360802  0.7693490
## mammals   -2.356117  0.6958912
## reptiles   -4.571142  0.6812206
```

If you want to examine the BLUPs or conditional modes and see how they vary between groups, you need to obtain the conditional SD as well as the conditional modes. The simplest way is to get them from the `raneff()` function

(short for “random effects”). The conditional SD are only returned when `as.data.frame()` is used on the results of `ranef()`, not when `ranef()` is used by itself.

The commands below will calculate the conditional mean and SD of each BLUP. We can use those results to estimate the 95% CI of each group-specific random parameter. This can also be used as a *heuristic* for interpreting the BLUPs, but **does not tell you whether or not each random effect is “significant”**. The framework of a “significant vs. nonsignificant” isn’t as well defined for random effects as it is for fixed effects. If the 95% CI of a random effect does not include 0, then its fine to say that that group’s random effect is not 0. Looking at which groups have nonzero random effects might give you some insight into what is driving the variability in your dataset. However, you should not use this method to say whether a random effect is statistically significant or not, because you haven’t performed a significance test. More generally, *you should not be making inferences based on conditional modes or BLUPs*. If you were interested in predicting values for these particular levels of a random factor, then you should have fit it as a fixed effect in the first place!

```
re <- as.data.frame(ranef(mod4, condVar=TRUE))

# 95% CI (normal approximation)
re$low <- qnorm(0.025, re$condval, re$condsd)
re$upp <- qnorm(0.975, re$condval, re$condsd)

# flag effects whose 95% CI includes 0
# (indicates BLUP not very different from
# fixed effect parameter)
re$inc.0 <- ifelse(0>=re$low & 0<=re$upp, 1, 0)

# round to 2 decimal places
re[,4:7] <- apply(re[,4:7], 2, round, 2)

# inspect results
re

##   grpvar      term      grp condval condsd    low    upp inc.0
## 1 group1 (Intercept)  birds    2.43   0.05  2.32  2.54    0
## 2 group1 (Intercept) dinosaurs -0.49   0.39 -1.26  0.27    1
## 3 group1 (Intercept) fishes   -1.91   0.10 -2.11 -1.71    0
## 4 group1 (Intercept) mammals  1.09   0.06  0.98  1.21    0
## 5 group1 (Intercept) reptiles -1.12   0.12 -1.36 -0.88    0
## 6 group1      logx     birds  -0.01   0.01 -0.03  0.01    1
## 7 group1      logx  dinosaurs  0.03   0.03 -0.03  0.08    1
## 8 group1      logx     fishes  0.05   0.02  0.01  0.08    0
## 9 group1      logx    mammals -0.03   0.01 -0.04 -0.01    0
## 10 group1     logx  reptiles -0.04   0.02 -0.08  0.00    0
```

The flag `inc.0` identifies which BLUPs have a 95% CI that includes 0. If the 95% CI includes 0, then we can infer that the BLUP for that group is not very different from the corresponding fixed effect—i.e., the overall estimate for that parameter. For example, the table above shows that the random effect on the intercept associated with dinosaurs, -0.49 with 95% CI [-1.26, 0.27], is similar to the overall intercept of -3.45 (see `summary(mod4)`). This is despite the fact that the BLUP for the intercept for dinosaurs was quite large: -3.94 (i.e., -3.45 + -0.49; see `coef(mod4)$group1`).

If you want the actual BLUPs for the groups and not just their differences from the mean, you can get them by adding estimates of the fixed effects to each parameter to the random effects associated with each group. Do this for every column except the SD. This works because one of the properties of the normal distribution is that its mean (i.e., location) is a separate parameter from its variance. So, adding a constant to the mean will shift the entire distribution, including its quantiles, by that constant while retaining the same variance.

```
coefs <- summary(mod4)$coefficients
add.val <- rep(coefs[,1], each=ngroup)
re$condval <- round(re$condval + add.val,2)
re$low <- round(re$low + add.val,2)
re$upp <- round(re$upp + add.val,2)

# inspect results
re

##   grpvar      term      grp condval condstd    low     upp inc.0
## 1 group1 (Intercept)  birds   -1.02  0.05 -1.13 -0.91  0
## 2 group1 (Intercept) dinosaurs -3.94  0.39 -4.71 -3.18  1
## 3 group1 (Intercept) fishes   -5.36  0.10 -5.56 -5.16  0
## 4 group1 (Intercept) mammals -2.36  0.06 -2.47 -2.24  0
## 5 group1 (Intercept) reptiles -4.57  0.12 -4.81 -4.33  0
## 6 group1      logx  birds    0.71  0.01  0.69  0.73  1
## 7 group1      logx dinosaurs  0.75  0.03  0.69  0.80  1
## 8 group1      logx  fishes   0.77  0.02  0.73  0.80  0
## 9 group1      logx mammals  0.69  0.01  0.68  0.71  0
## 10 group1     logx reptiles  0.68  0.02  0.64  0.72  0
```

There are other methods for getting confidence intervals on BLUPs that we will explore some of the other mixed model pages.

7.3 Generalized linear mixed models (GLMM)

7.3.1 Definition

Just as the linear mixed model (LMM) is an extension of the linear model, the **generalized linear mixed model (GLMM)** is an extension of the generalized

linear model (GLM). The difference between the LMM and GLMM is the same as the difference between the LM and GLM: the response values can come from distributions other than the normal, and the model can be linear on a different scale than of the original scale (i.e., the link scale).

The equations for a simple GLMM, with normal (Gaussian) family and identity link function, one continuous predictor variable X , and random intercepts and slope by a factor Z , are shown below. The normal distributions of the random effects are presented as parameterized by a mean μ and variance σ^2 (or sometimes SD σ)⁹.

Component	Expression
Stochastic part (Y values)	$y_i \sim Normal(\mu_i, \sigma^2)$
Link function	$E(y_i) = \mu_i = \eta_i$
Linear predictor	$\eta_i = \beta_{0,Z} + \beta_{1,Z}X_i$
Random intercepts	$\beta_{0,Z} \sim Normal(\mu_{\beta_0}, \sigma_{\beta_0}^2)$
Random slopes	$\beta_{1,Z} \sim Normal(\mu_{\beta_1}, \sigma_{\beta_1}^2)$

7.3.2 GLMM on simulated data

Let's explore the GLMM with a simulated example. We'll simulate data for a simulated study of the occurrence (i.e., presence/absence) of an imaginary bird species along a gradient of urbanization. The imaginary study was conducted over 12 years. In each year, between 9 and 19 sites were surveyed and the presence or absence of an endangered bird was recorded. The researchers assumed that variation in environmental conditions from year to year affected bird presence, so they used a mixed effects model to analyze their data. The explanatory variable, `urb`, is an index of urbanization and land use intensity derived from satellite data. Because they are interested in modeling bird presence/absence, they used a logistic regression model, with a random effect for year. In ecology this sort of thing is called an **occupancy model**.

The code below simulates the dataset described above. First the dataset structure is defined. Then, the **hyperparameters**, which define the distribution of random effects, are set. These are then used to define the group-level parameters and the individual observations.

```
set.seed(123)

# years in study
n.groups <- 12
```

⁹In principle, there is no reason why you couldn't have random effects following a non-normal distribution... such a model can be easily defined in BUGS or JAGS but I'm not aware of a way to do it in R.

```

# random number of sites each year
n.sample <- sample(8:20, n.groups, replace=TRUE)

# hyperparameters
mu.beta0 <- 1.5
sd.beta0 <- 1.7
mu.beta1 <- -6.8
sd.beta1 <- 0.1

# draw random effects
beta0 <- rnorm(n.groups, mu.beta0, sd.beta0)
beta1 <- rnorm(n.groups, mu.beta1, sd.beta1)

# set up data frame with explanatory variable.
dlist <- vector("list", n.groups)
for(i in 1:n.groups){
  dlist[[i]] <- data.frame(year=i, site=1:n.sample[i])
  dlist[[i]]$urb <- runif(n.sample[i], 0, 1)
}
sim <- do.call(rbind, dlist)

# add coefficients for each observation
sim$beta0 <- beta0[sim$year]
sim$beta1 <- beta1[sim$year]

# calculate linear predictor eta (logit scale)
# and probability p (probability scale)
sim$eta <- sim$beta0+sim$beta1*sim$urb
sim$p <- plogis(sim$eta)

# apply stochastic part of model (Bernoulli draws)
sim$y <- rbinom(nrow(sim), 1, sim$p)

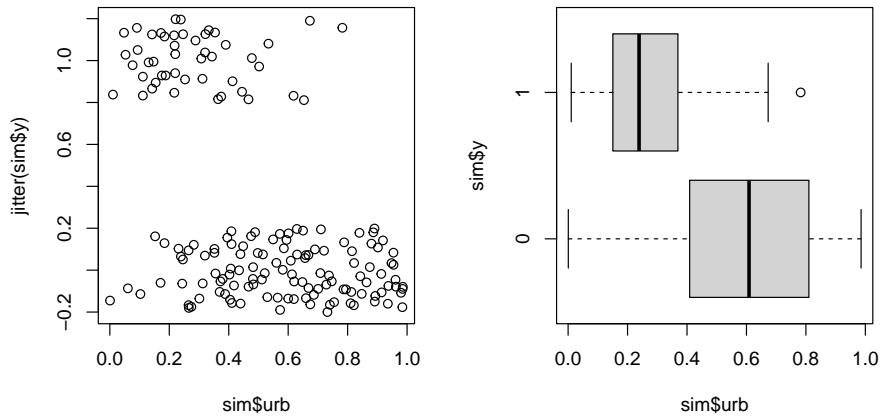
```

Let's start the analysis by examining the data:

```

par(mfrow=c(1,2), mar=c(5.1, 5.1, 1.1, 1.1))
plot(sim$urb, jitter(sim$y))
boxplot(sim$urb~sim$y, horizontal=TRUE)

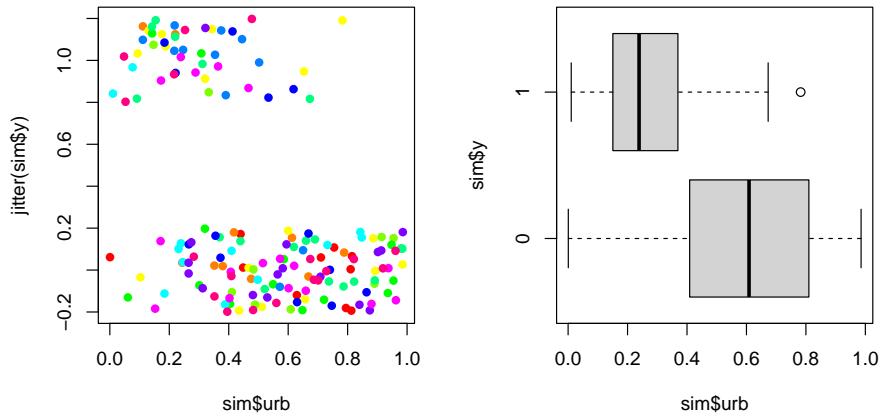
```



The figure suggests that increasing urbanization tends to decrease occupancy by this bird species. This makes sense, biologically, but it also appears that there is a lot of variation in urbanization within the sites where the bird occurs and the sites where it doesn't. Maybe there is some variation related to year?

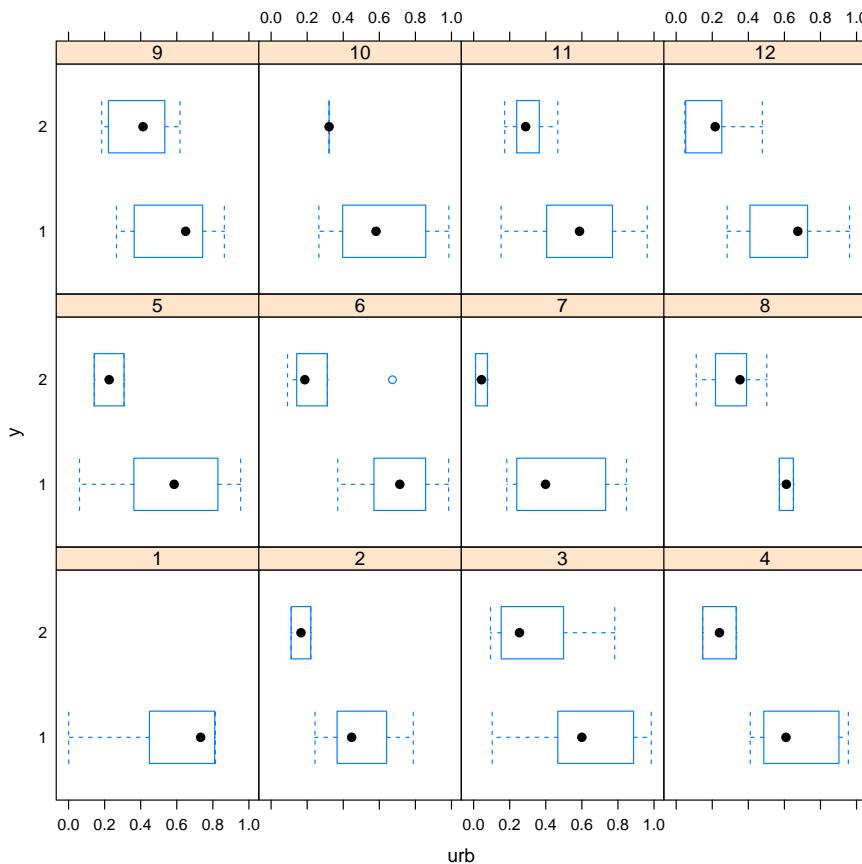
```
# add colors for each year
cols <- rainbow(n.groups)
sim$col <- cols[sim$year]

par(mfrow=c(1,2), mar=c(5.1, 5.1, 1.1, 1.1))
plot(sim$urb, jitter(sim$y), col=sim$col, pch=16)
boxplot(sim$urb~sim$y, horizontal=TRUE)
```



With so many groups (years) it's hard to tell if year might be affecting the relationship between occupancy and urbanization. Let's use package `lattice` to make a hierarchical plot that shows the groups separately.

```
library(lattice)
bwplot(y~urb|factor(year), data=sim)
```



The figure shows that urbanization appears to decrease occupancy in almost every year. The bird was not observed at all in year 1. In some years, occupancy dropped off at low levels of urbanization (e.g., years 2, 6, and 7); in other years, the bird was found in sites with much higher levels of urbanization (e.g., years 3 and 9). These findings support the idea that year needs to be accounted for. We'll fit three versions of the model, because we don't know if the random effect of year should affect the intercept, slope, or intercept and slope of the linear predictor.

```
library(lme4)
mod1 <- glmer(y~urb+(1|year), data=sim, family=binomial)
mod2 <- glmer(y~urb+(0+urb|year), data=sim, family=binomial)
mod3 <- glmer(y~urb+(urb|year), data=sim, family=binomial)
```

```
mod4 <- glmer(y~urb+(urb||year), data=sim, family=binomial)
```

All of the models converged¹⁰, so let's use AIC to determine which model is the best supported by the data.

```
aic.df <- AIC(mod1, mod2, mod3, mod4)
aic.df$delta <- aic.df$AIC - min(aic.df$AIC)
aic.df$wt <- exp(-0.5*aic.df$delta)
aic.df$wt <- aic.df$wt/sum(aic.df$wt)
aic.df <- aic.df[order(-aic.df$wt),]
aic.df

##      df      AIC     delta      wt
## mod1  3 139.7622 0.000000 0.50498148
## mod2  3 141.2993 1.537081 0.23415450
## mod4  4 141.7594 1.997213 0.18603135
## mod3  5 143.5807 3.818534 0.07483267
```

Interestingly, AIC identifies model 1 (random intercept) as the best fitting model, when we know that model 4 (random uncorrelated slopes and intercepts) is the truth. Looking back at the simulation parameters, can you guess why this happened? However, $\Delta(AIC)$ was <2 for models 1, 2, and 4, suggesting that any of them could be considered “best fitting”.

Inspect the outputs to see what the parameter estimates are. The estimated intercept and slope aren't very close to the true values (`mu.beta0` and `mu.beta1`), but this might have something do with the large variability in the intercepts (`sd.beta0`).

```
summary(mod1)
```

```
## Generalized linear mixed model fit by maximum likelihood (Laplace
## Approximation) [glmerMod]
## Family: binomial ( logit )
## Formula: y ~ urb + (1 | year)
##   Data: sim
##
##      AIC      BIC  logLik deviance df.resid
##    139.8    149.1   -66.9    133.8      161
##
## Scaled residuals:
##       Min     1Q Median     3Q    Max
## -2.8969 -0.4530 -0.1617  0.3846  4.1386
##
## Random effects:
##   Groups Name        Variance Std.Dev.
##
```

¹⁰Sometimes model 3 does not converge. If the random number seed is set to 7, only models 1, 2, and 4 will converge. (This was true on my machine. Your results may vary).

```

## year (Intercept) 1.137 1.066
## Number of obs: 164, groups: year, 12
##
## Fixed effects:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept) 1.959     0.602   3.254 0.00114 **
## urb        -7.312     1.330  -5.496 3.88e-08 ***
## ---
## Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Correlation of Fixed Effects:
##      (Intr) urb
## urb -0.753

```

The intercepts vary quite a bit between years:

```
coef(mod1)$year
```

```

##      (Intercept)      urb
## 1    0.8792071 -7.312053
## 2    1.4874052 -7.312053
## 3    2.8794661 -7.312053
## 4    2.0679683 -7.312053
## 5    1.3682659 -7.312053
## 6    2.5975640 -7.312053
## 7    0.9167481 -7.312053
## 8    3.6762635 -7.312053
## 9    2.6771235 -7.312053
## 10   1.1477449 -7.312053
## 11   2.0878343 -7.312053
## 12   2.0208604 -7.312053

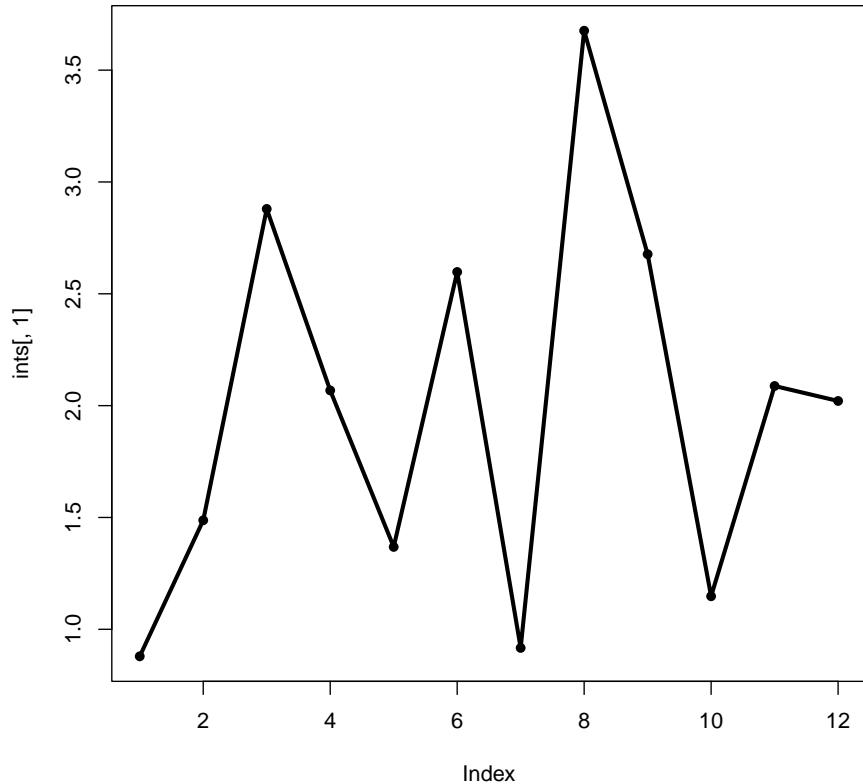
```

Plotting the intercept vs. year (i.e., time) might tell us something about how environmental factors varied over time.

```

ints <- coef(mod1)$year
par(mfrow=c(1,1))
plot(ints[,1], type="o", lwd=3, pch=16)

```



It appears that the intercept dropped precipitously 3 times: year 3, year 6, and years 8-10. Maybe something happened in those years that reduced the likelihood of the bird showing up. For example, maybe these were drought years. Or maybe those were the years when new cohorts of graduate students were recruited who weren't as effective at finding the birds. Whatever the cause of the variation was, the GLMM accounted for it. Explaining and interpreting that variation it is the job of the biologist.

We'll finish our example GLMM analysis by plotting the model predictions.

```
# number of points and x values
pn <- 50
px <- seq(0, 1, length=pn)

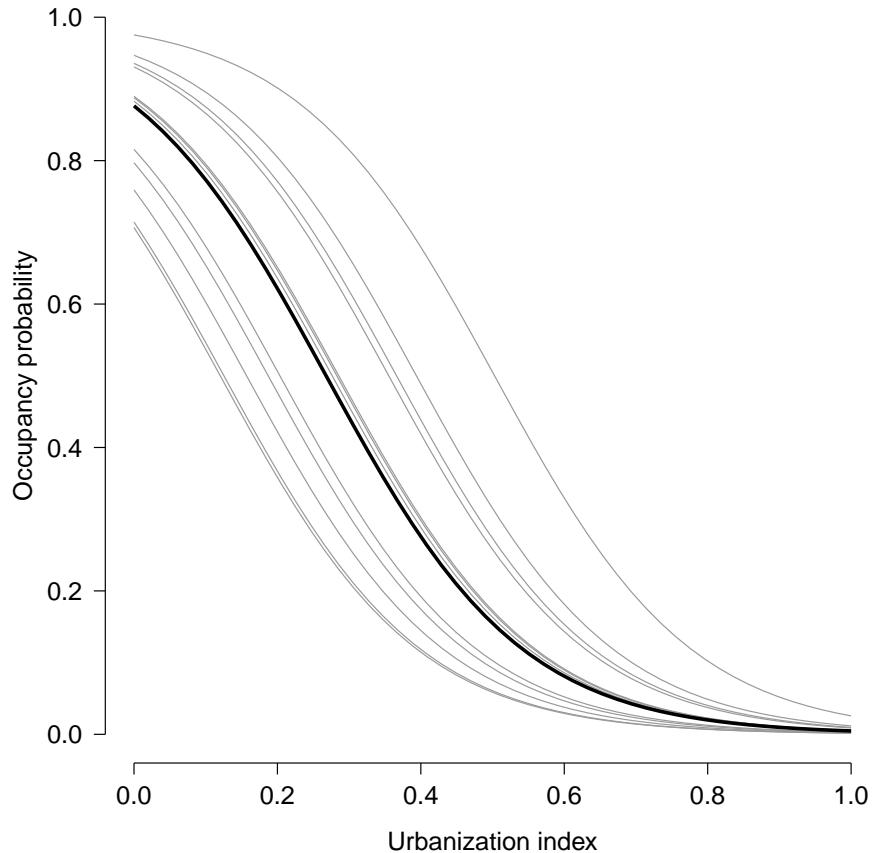
# predictions by level (year)
dx1 <- expand.grid(year=1:n.groups, urb=px)
dx1 <- dx1[order(dx1$year, dx1$urb),]
```

```
# coefficients (include random beta0)
dx1$y <- predict(mod1, newdata=data.frame(dx1), type="link")
dx1$y <- plogis(dx1$y)
dx1$col <- cols[dx1$year]

# predictions for "average"
dx2 <- data.frame(year=NA, urb=px)
dx2$y <- predict(mod1, newdata=data.frame(dx2),
  type="link", re.form=NA, allow.new.levels=TRUE)
dx2$y <- plogis(dx2$y)
```

Now make the plot.

```
par(mfrow=c(1,1), mar=c(5.1, 5.1, 1.1, 1.1),
  lend=1, las=1, bty="n",
  cex.axis=1.3, cex.lab=1.3)
plot(dx1$urb, dx1$y, type="n",
  ylim=c(0,1), ylab="Occupancy probability",
  xlab="Urbanization index")
for(i in 1:n.groups){
  flag <- which(dx1$year==i)
  points(dx1$urb[flag], dx1$y[flag], type="l", col="grey60")
}
points(dx2$urb, dx2$y, type="l", lwd=3)
```

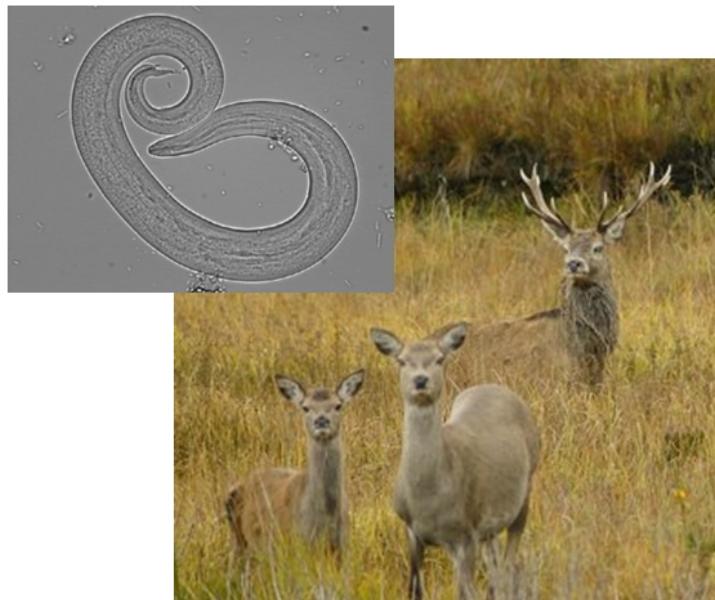


7.3.3 GLMM on real data

This example is adapted from Zuur et al. (2009a). Vicente et al. (2005) studied the distribution of the parasitic nematode *Elaphostrongylus cervi* in red deer (*Cervus elephas*) in Spain (below)¹¹. Red deer are the primary hosts of *E. cervi*, which primarily affect the hosts' lung tissue. Vicente et al. (2005) surveyed deer feces on many farms in Spain and tested droppings for *E. cervi* larvae. We will analyze part of their dataset, to see if red deer size and sex affect the occurrence of *E. cervi* larvae. We will use GLMM to account for potential variation in parasite occurrence between farms. GLMM is appropriate in this situation because we are not interested in the effects of particular farms, which are a subsample of the locations where red deer occur.

¹¹Nematode image: Alberti et al. (2011). Deer image: O. Deme (https://upload.wikimedia.org/wikipedia/commons/d/df/Red_deer.jpg).

Elaphostrongylus cervi



Red deer (*Cervus elaphus*)

Download the dataset and save it to your R home directory. This is a tab-delimited file (.txt), so you need to use `read.table()` instead of `read.csv()`.

```
dat.name <- "DeerEcervi.txt"
dat <- read.table(dat.name, sep="\t", header=TRUE)
```

The dataset has the following variables:

Variable	Meaning
Farm	Farm (i.e., sampling site)
Sex	Sex of deer (1 = female, 2 = male)
Length	Deer body length (cm)
Ecervi	Parasite load in fecal sample

Because we are interested in parasite occurrence, not abundance, we should use logistic regression. That is, a binomial GLM with a logit link. We will include a random effect associated with `farm`. Before running the analysis, let's modify the variables in the dataset. First, define a response variable `y` that is 1 when parasites are present and 0 when parasite are absent. Then, center the deer lengths by subtracting the mean length. Finally, recode the sexes as `f` and `m`.

instead of 1 and 2. This is optional but will make things easier later.

```
dat$y <- ifelse(dat$Ecervi > 0, 1, 0)
dat$len <- dat$Length-mean(dat$Length)
dat$sex <- ifelse(dat$Sex==1, "f", "m")
```

Now we can fit a model. Like the original authors, we will investigate the effects of sex, size, and farm. The model for parasite presence y in individual j on farm i , with probability p_{ij} , varies with length L_{ij} , sex S_{ij} , the interaction between sex and length $S_{ij}L_{ij}$, and farm F_i as:

$$\begin{aligned} y_i &\sim \text{Bernoulli}(p_{ij}) \\ \text{logit}(p_{ij}) &= \beta_0 + \beta_1 L_{ij} + \beta_2 S_{ij} + \beta_3 L_{ij}S_{ij} + \gamma_i \\ \gamma_i &\sim \text{Normal}(\mu_\gamma, \sigma_\gamma) \end{aligned}$$

In this model farm has a random effect on the intercept, γ_i . The random effects on the intercepts are assumed to come from a normal distribution with mean μ_γ and SD σ_γ . Some people prefer to write the model a slightly different way, which combines the overall intercept β_0 with the random effects on the intercept γ_i as a single parameter $\beta_{0,i}$:

$$\begin{aligned} \text{logit}(p_{ij}) &= \beta_{0,i} + \beta_1 L_{ij} + \beta_2 S_{ij} + \beta_3 L_{ij}S_{ij} + \gamma_i \\ \beta_{0,i} &\sim \text{Normal}(\mu_{\beta_0}, \sigma_{\beta_0}) \end{aligned}$$

In this format, the combined intercept $\beta_{0,i}$, is equivalent to $(\beta_0 + \gamma_i)$ for each group i . This is the way I prefer to write GLMM formulas because it makes it more clear what the random effects are really doing.

GLMM can be fit with `lme4::glmer()`:

```
library(lme4)
mod1 <- glmer(y~len*sex+(1|Farm), data=dat, family=binomial)
summary(mod1)

## Generalized linear mixed model fit by maximum likelihood (Laplace
## Approximation) [glmerMod]
## Family: binomial ( logit )
## Formula: y ~ len * sex + (1 | Farm)
## Data: dat
##
##      AIC      BIC  logLik deviance df.resid
##     832.6    856.1   -411.3    822.6      821
##
## Scaled residuals:
##      Min      1Q  Median      3Q      Max
## -3.000 -0.800 -0.100  0.600  2.000
```

```

## -6.2678 -0.6090  0.2809  0.5022  3.4546
##
## Random effects:
## Groups Name           Variance Std.Dev.
## Farm   (Intercept) 2.391    1.546
## Number of obs: 826, groups: Farm, 24
##
## Fixed effects:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept) 0.938969  0.356004  2.638  0.00835 **
## len         0.038964  0.006917  5.633 1.77e-08 ***
## sexm       0.624487  0.222938  2.801  0.00509 **
## len:sexm    0.035859  0.011409  3.143  0.00167 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Correlation of Fixed Effects:
##          (Intr) len     sexm
## len      -0.107
## sexm     -0.189  0.238
## len:sexm  0.091 -0.514  0.232

```

The model is essentially an ANCOVA model (continuous predictor that interacts with a factor, **sex**) fit as a binomial GLM (logistic regression) with a random effect of another factor (**Farm**). Interpreting the fixed effects length (**len**) and **sex** is a little tricky because of the interaction. We have to be careful and think about which observations each coefficient applies to:

- The printed intercept, **(Intercept)**, of 0.939 is the intercept for the baseline **sex**, females, regardless of farm effects.
- The printed coefficient for length (**len**), 0.038, is the slope for the baseline **sex** (females).
- The intercept for males is the baseline intercept plus the effect of being male: $0.9389 + 0.6244 = 1.5634$.
- The slope for males is the baseline slope plus the effect of being male on the slope: $0.0389 + 0.0358 = 0.0748$.

Thus, the fixed effects for females are:

$$\text{logit}(p_j) = 0.939 + 0.038L_j$$

And the fixed effects for males are:

$$\text{logit}(p_j) = (0.939 + 0.624) + (0.039 + 0.036)L_j = 1.563 + 0.075L_j$$

Now let's plot the model predictions against the original data. We'll assemble a

new dataset that covers the domain of x values (centered lengths), both sexes, and the 24 farms.

```
farms <- sort(unique(dat$Farm))
nfarm <- length(farms)
agg <- aggregate(len~sex, data=dat, range)
agg <- agg[[2]]
n <- 50
dx1 <- expand.grid(
  sex="f",
  len=seq(agg[1,1], agg[1,2], length=n),
  Farm=farms)
dx2 <- expand.grid(
  sex="m",
  len=seq(agg[2,1], agg[2,2], length=n),
  Farm=farms)
dx <- rbind(dx1, dx2)
```

Now generate the predictions:

```
pred <- predict(mod1, newdata=data.frame(dx), type="link")
dx$y <- pred

# inverse link function
dx$py <- plogis(dx$y)
```

The predictions we just calculated are specific to each farm. The “average” probability, which ignores farm, can be obtained either predicting without random effects (safest), by manual calculation using the fixed effects (more trouble but generally okay) or by averaging across levels of the random effect (not recommended). The first two methods are demonstrated below, but we’re going to use the outputs from the first.

```
# method 1: predict with re.form=NA
dx3 <- dx[which(dx$Farm == farms[1]),]
dx3$y <- predict(mod1,
  newdata=data.frame(dx3),
  type="link", re.form=NA)
dx3$py <- plogis(dx3$y)

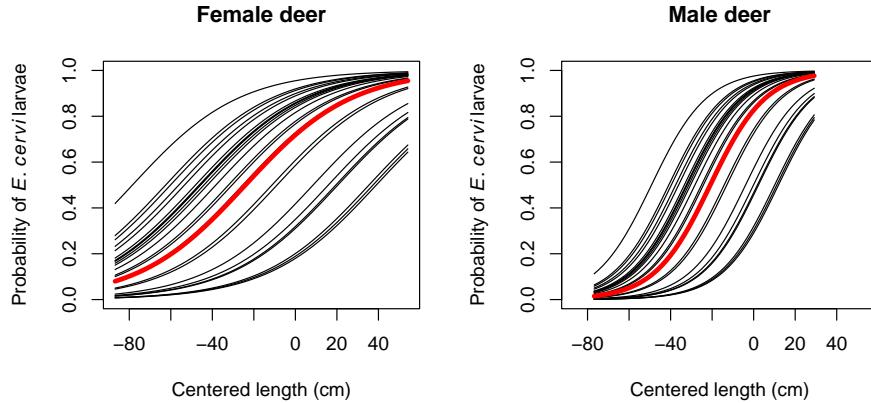
# method 1 ver. 2: predict to a new level of the random effect
dx5 <- dx[which(dx$Farm == farms[1]),]
dx5$Farm <- NA
dx5$y <- predict(mod1,
  newdata=data.frame(dx5),
  type="link", re.form=NA,
  allow.new.levels=TRUE)
dx5$py <- plogis(dx5$y)
```

```
# method 2: manually calculate
dx4 <- dx3
flag <- which(dx4$sex == "f")
dx4$y[flag] <- 0.939+0.038*dx4$len[flag]
dx4$y[-flag] <- 1.563+0.075*dx4$len[-flag]
```

For clarity, we will plot females and males on separate plots. Also for clarity, we will not plot the data and instead will plot the predicted probabilities.

```
par(mfrow=c(1,2))
plot(dx$len, dx$py, type="n", ylim=c(0,1),
     xlab="Centered length (cm)",
     ylab=expression(Probability~of~italic(E)*.~italic(cervi)~larvae))
use.dx <- dx[which(dx$sex == "f"),]
for(i in 1:nfarm){
  flag <- which(use.dx$Farm == farms[i])
  points(use.dx$len[flag], use.dx$py[flag], type="l")
}
flag <- which(dx3$sex == "f")
points(dx3$len[flag], dx3$py[flag], type="l", lwd=4, col="red")
title(main="Female deer")

plot(dx$len, dx$py, type="n", ylim=c(0,1),
     xlab="Centered length (cm)",
     ylab=expression(Probability~of~italic(E)*.~italic(cervi)~larvae))
use.dx <- dx[which(dx$sex == "m"),]
for(i in 1:nfarm){
  flag <- which(use.dx$Farm == farms[i])
  points(use.dx$len[flag], use.dx$py[flag], type="l")
}
flag <- which(dx3$sex == "m")
points(dx3$len[flag], dx3$py[flag], type="l", lwd=4, col="red")
title(main="Male deer")
```



It is worth asking what the 95% CI of the predicted probability for an “average” farm would be. We can calculate this using some information from the summary of model 1 (see above). The random effects on the intercept for each farm have mean 0 and SD = 1.546. Because this model has only a random intercept, and not a random slope, we can express uncertainty about the predicted value (on the logit scale) by shifting the predicted value up or down. This means that the predicted value (on the logit scale) has a confidence interval defined by a normal distribution with the predicted value as the mean, and a standard deviation of 1.546. For any predicted value $E(\text{logit}(p))$, the 95% CI can be approximated as $E(\text{logit}(p)) \pm 1.96 \times 1.546$.

In R we can calculate the CI directly using the normal distribution quantile functions:

```
dx5 <- dx[which(dx$Farm == farms[1]),]
dx5$Farm <- NA
dx5$y <- predict(mod1,
  newdata=data.frame(dx5),
  type="link", re.form=NA,
  allow.new.levels=TRUE)
dx5$lo <- qnorm(0.025, dx5$y, 1.546)
dx5$up <- qnorm(0.975, dx5$y, 1.546)

# inverse link function
dx5$py <- plogis(dx5$y)
dx5$plo <- plogis(dx5$lo)
dx5$pup <- plogis(dx5$up)
```

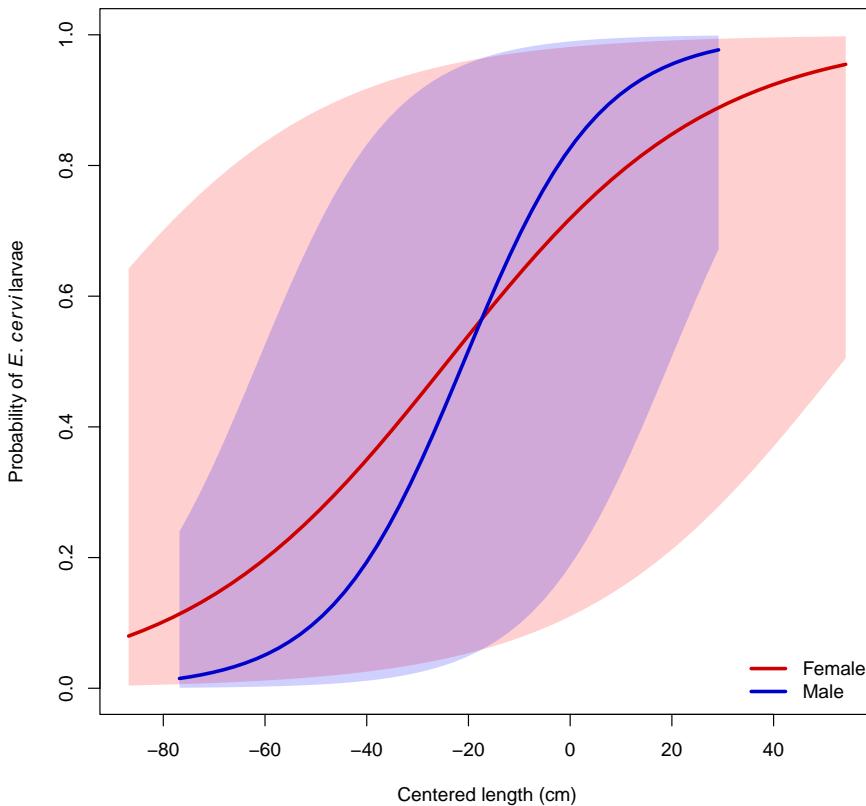
Finally, we can make the plot with the approximated 95% CI:

```
par(mfrow=c(1,1))
plot(dx$len, dx$py, type="n",
  ylim=c(0,1),
```

```

xlab="Centered length (cm)",
ylab=expression(Probability~of~italic(E)*.~italic(cervi)~larvae))
flag1 <- which(dx5$sex == "f")
flag2 <- which(dx5$sex == "m")
polygon(x=c(dx5$len[flag1], rev(dx5$len[flag1])),
        y=c(dx5$plo[flag1], rev(dx5$pup[flag1])),
        border=NA, col="#FF000030")
polygon(x=c(dx5$len[flag2], rev(dx5$len[flag2])),
        y=c(dx5$plo[flag2], rev(dx5$pup[flag2])),
        border=NA, col="#0000FF30")
points(dx5$len[flag1], dx5$py[flag1], type="l", lwd=3, col="red3")
points(dx5$len[flag2], dx5$py[flag2], type="l", lwd=3, col="blue3")
legend("bottomright", legend=c("Female", "Male"),
       lwd=3, col=c("red3", "blue3"), bty="n")

```



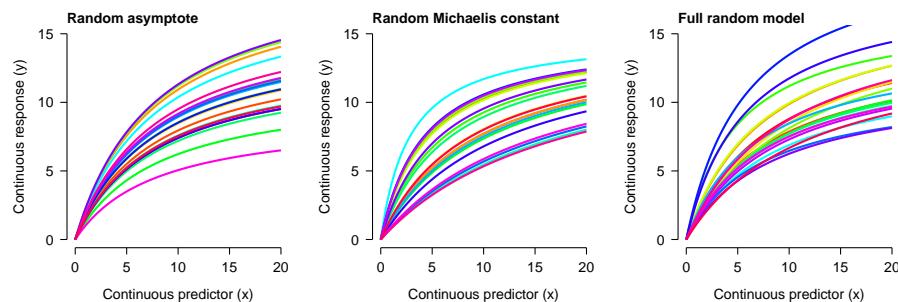
Because the GLMM we fit was a logistic regression model, you could also calculate

an ROC curve and AUC) to express the predictive accuracy of the model. For other GLMM, you could perform a cross-validation analysis to see how well the model predicts out-of-sample data.

7.4 Nonlinear mixed models (NLME)

7.4.1 Definition and background

Nonlinear mixed effects models (NLME) extend the nonlinear model (NLS) to allow parameters to vary randomly by a factor. In an LMM or GLMM, the parameters that could vary randomly were the intercept and slope (or other coefficients). Because NLS models don't always have intercepts and slopes, the random effects can potentially apply to any parameter in the model. For example, a mixed effects Michaelis-Menten model might have an asymptote, Michaelis constant, or both that vary randomly between levels of a factor:



As you might expect, fitting NLME can be tricky: it combines all of the difficulties with nonlinear models with those of mixed models! One of the most commonly used options is the `nlmer()` function from the `lme4` package. For complicated models you may want to consider using Bayesian inference with JAGS because of the added flexibility afforded by the Markov chain Monte Carlo (MCMC) paradigm.

7.4.2 NLME on simulated data

Let's simulate data that might have come from an enzyme kinetics study. We'll model reaction rate y as a function of concentration x for some family of substrates. NLME might be a better choice than NLS for such a problem because the selection of substrates is only a sampling of the potential substrates, and the researcher is interested in how reaction rate might vary with concentration for other, untested substrates.

```
set.seed(123)

# data structure: number of substrates (nlev)
```

```

# and replicates per concentration
# and substrate (nrep)
nlev <- 8
nrep <- 3

# concentrations:
x <- 10^seq(-2, 2, by=0.5)

# hyperparameters for a and b
avec <- rnorm(nlev, 129, 12)
bvec <- rnorm(nlev, 22, 12)

# assemble dataframe of predictors and parameters
dat <- expand.grid(subs=1:nlev, conc=x, rep=1:nrep)
dat$a <- avec[dat$subs]
dat$b <- bvec[dat$subs]

# calculate expected value
dat$mu <- (dat$a*dat$conc)/(dat$b+dat$conc)

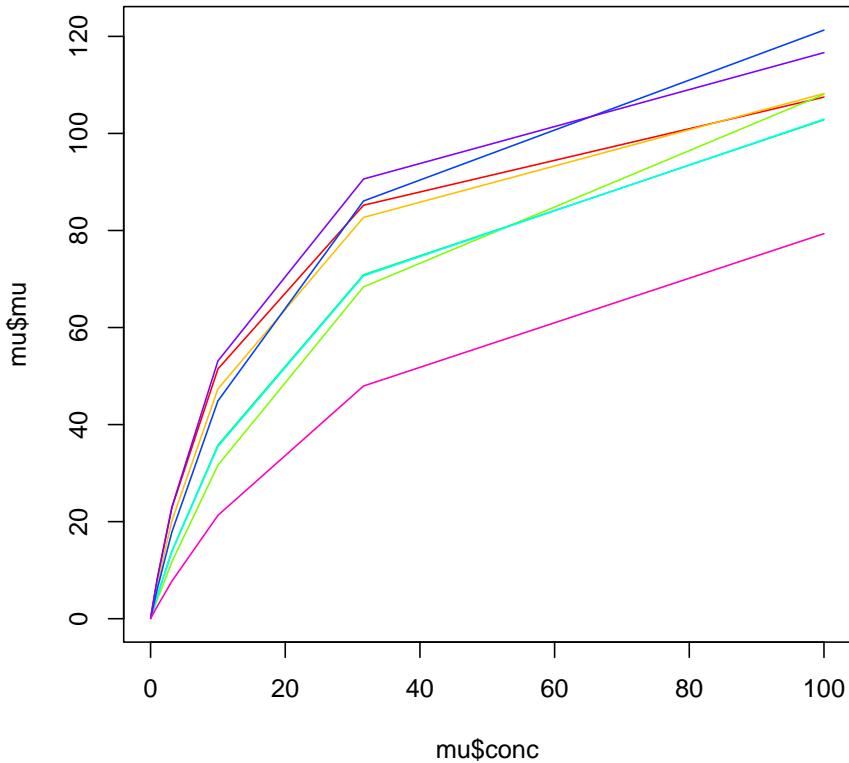
# residual SD and residuals
sigma <- 10
dat$y <- rnorm(nrow(dat), dat$mu, sigma)

# can't have negative values
dat$y <- pmax(0, dat$y)

# add colors for plotting
cols <- rainbow(nlev)
dat$col <- cols[dat$subs]

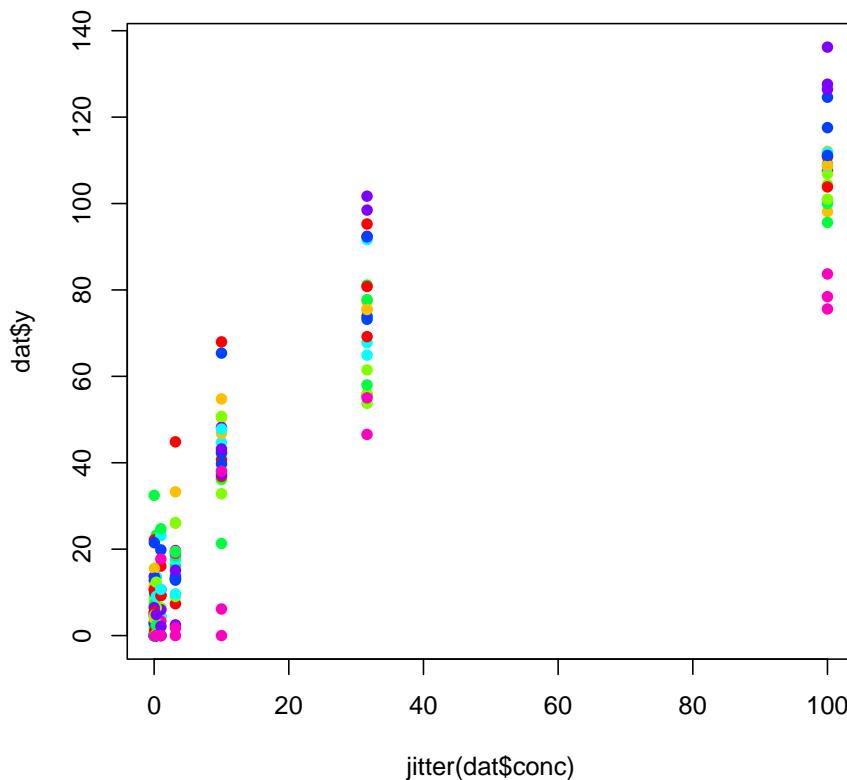
```

Make a quick plot of expected value in each group (substrate):



We can also see the actual data:

```
plot(jitter(dat$conc), dat$y, pch=16, col=dat$col)
```



Faced with these data, we might suspect that the y variable (reaction rate) could be modeled by a Michaelis-Menten (MM) function. However, we might also suspect that there is a lot of variation due to substrate that is not accounted for by a garden variety MM model. If we are not interested in these substrates in particular, but only as representatives of all substrates, then a mixed model is called for. What is not clear is whether the asymptote, Michaelis constant, or both should be random. Fortunately, it is straightforward to test all three variations.

To use `nlmer()`, you need to specify the model in 3 parts: response, function, and random effects. The parts are separated by tildes (~). Unlike `nls()`, where the model formula can be provided directly, the model formula in `nlmer()` must be a function that takes in a predictor and uses named parameters to produce a numeric output. The self-start functions we saw in the NLS unit can be used for this, but unlike in `nls()` you *always* need to provide starting values. The starting values can be estimated or guessed at by inspecting the scatterplot and thinking about the meaning of each parameter (example). Fortunately, the

random effects syntax works in much the same way as it does in `lmer()`.

```
library(lme4)

# named vector of starting values
startvec <- c(a=120, b=25)

# model 1: random asymptote
mod1 <- nlmer(y~SSmicmen(conc, a, b)~a|subs,
               data=dat, start=startvec)

# model 2: random Michaelis constant
mod2 <- nlmer(y~SSmicmen(conc, a, b)~b|subs,
               data=dat, start=startvec)

# model 3: random both parameters
mod3 <- nlmer(y~SSmicmen(conc, a, b)~(a|subs)+(b|subs),
               data=dat, start=startvec)
```

The parameter estimates of all three models are close to the true values. We can compare the fit of the models using AIC and find that model 3 has the best fit. This should not be surprising because model 3 is the correct one.

```
AIC(mod1, mod2, mod3)
```

```
##      df      AIC
## mod1  4 1563.909
## mod2  4 1573.510
## mod3  5 1561.939
```

Unfortunately, generating predictions for `nlmer()` models is not straightforward¹². So, let's use a different method of model fitting: Markov chain Monte Carlo (MCMC). You'll need a separate program, JAGS (Plummer 2003), and the R packages `rjags` (Plummer 2021) and `R2jags` (Su and Masanao Yajima 2021). The general procedure for using JAGS is to first write the model to a text file using `sink()`, then call JAGS from within R. JAGS will then read the text file and fit the model. The code block below will write the model file to your current R working directory.

```
# write model file
mod.name <- "nlme_mod03.txt"

sink(mod.name)
cat("
model{
  # priors
  ## hyperparameters
```

¹²I couldn't get it to work at all.

```

mu.a ~ dunif(1, 200)
sd.a ~ dunif(1, 20)
mu.b ~ dunif(1, 200)
sd.b ~ dunif(1, 20)

## precision of a and b
tau.a <- 1/(sd.a * sd.a)
tau.b <- 1/(sd.b * sd.b)

## parameters
for(i in 1:nlev){
  a.vec[i]~dnorm(mu.a, tau.a)
  b.vec[i]~dnorm(mu.b, tau.b)
}

## residual SD and precision (tau)
sigma ~ dunif(0, 1e4)
tau.y <- 1 / (sigma * sigma)

# likelihood
for(i in 1:N){
  y[i] ~ dnorm(eta[i], tau.y)
  eta[i] <- (a.vec[levs[i]] *
    x[i])/(b.vec[levs[i]]+x[i])
}
}# i for N
}#model
", fill=TRUE)
sink()

```

Next, package the data up for JAGS. Any value or set of values used in the model must be included by name in a list.

```

# package data for JAGS
in.data <- list(y=dat$y,
                 x=dat$conc,
                 levs=dat$sub,
                 N=nrow(dat),
                 nlev=nlev
                 )

```

Any value that is not fixed (i.e., part of the input data) must be supplied with an initial value. It doesn't matter much what this value is, as long as it is within the proper domain of the variable. For example, initial values for the λ parameter of a Poisson distribution must be ≥ 0 . The initial values must be stored in a list that has one element for each MCMC chain. I prefer to define a function that generates initial values automatically, in case I ever want to change the number

of chains.

```
# define initial values for MCMC chains
init.fun <- function(nc){
  res <- vector("list", length=nc)
  for(i in 1:nc){
    res[[i]] <- list(
      mu.a=runif(1, 1, 200),
      sd.a=runif(1, 1, 20),
      mu.b=runif(1, 1, 200),
      sd.b=runif(1, 1, 20),
      a.vec=runif(nlev, 1, 200),
      b.vec=runif(nlev, 1, 20),
      sigma=runif(1, 0.1, 20))
  }
  return(res)
}
```

Finally, set some parameters that control how JAGS will run. For actual inference, use a number of iterations $\geq 100,000$ (`n.iter`) and a burn-in length (`n.burn`) of about 20% of the total number of iterations. A large thinning interval (`n.thin`) is needed to mitigate autocorrelation of the MCMC chains; 100 is commonly used. These values are probably larger than needed for JAGS, but not larger than needed for OpenBUGS or WinBUGS¹³. The final size of your posterior samples—i.e., the number of draws from the posterior distribution that can be used for inference—is equal to:

$$n_{post} = n_{chains} \left(\frac{n_{iter} - n_{burn}}{n_{thin}} \right)$$

Set the last few parameters and send the model to JAGS:

```
library(rjags)
library(R2jags)

nchains <- 3
inits <- init.fun(nchains)

# parameters to monitor
params <- c("mu.a", "sd.a", "mu.b", "sd.b", "sigma",
           "a.vec", "b.vec")
```

¹³OpenBUGS and WinBUGS are two alternative programs that R can call to conduct MCMC sampling. WinBUGS is older and closed-source; OpenBUGS is newer and open-source. The modeling languages of JAGS, WinBUGS, and OpenBUGS are mostly identical. I learned Bayesian modeling with WinBUGS, but since 2020 have switched to JAGS. In my experience, JAGS is faster and more stable, while OpenBUGS is slower and less stable. Both are far preferable to WinBUGS

```

# MCMC parameters
n.ITER <- 5e3
n.burnin <- 1e2
n.thin <- 100

mod3j <- jags(data=in.data, inits=inits,
                 parameters.to.save=parms,
                 model.file=mod.name,
                 n.chains=nchains, n.ITER=n.ITER,
                 n.burnin=n.burnin, n.thin=n.thin) #jags

## Compiling model graph
## Resolving undeclared variables
## Allocating nodes
## Graph information:
##   Observed stochastic nodes: 216
##   Unobserved stochastic nodes: 21
##   Total graph size: 909
##
## Initializing model
##
## | | |
## | | |

```

The most important part of the output is here obtained by printing the model object to the console:

```
mod3j
```

```

## Inference for Bugs model at "nlme_mod03.txt", fit using jags,
## 3 chains, each with 5000 iterations (first 100 discarded), n.thin = 100
## n.sims = 147 iterations saved
##          mu.vect sd.vect    2.5%     25%     50%     75%   97.5%   Rhat
## a.vec[1] 128.513  6.888 116.049 123.847 128.003 133.146 142.769 0.995
## a.vec[2] 124.844  7.369 112.265 119.553 123.998 129.427 141.065 1.027
## a.vec[3] 131.011  8.392 117.081 125.895 129.974 135.282 149.705 1.002
## a.vec[4] 128.561  7.784 114.009 123.471 128.797 133.397 143.299 1.018
## a.vec[5] 130.787  8.030 116.861 125.303 129.991 135.742 146.610 1.034
## a.vec[6] 141.115  7.310 128.492 136.745 139.739 145.799 157.006 1.030
## a.vec[7] 157.374  8.199 142.347 152.461 156.953 161.871 173.935 1.023
## a.vec[8] 112.327  9.903  94.144 105.211 111.992 118.570 132.351 0.999
## b.vec[1] 17.655  3.048 12.552 15.310 17.434 19.692 23.174 0.998
## b.vec[2] 17.503  3.452 12.110 14.843 17.549 19.415 24.899 1.045
## b.vec[3] 30.481  5.581 21.453 27.065 29.868 33.323 43.953 1.004
## b.vec[4] 25.801  4.611 18.427 22.587 25.446 28.601 35.398 0.993
## b.vec[5] 22.084  3.786 15.758 19.649 21.693 24.311 29.921 1.026

```

```

## b.vec[6]    22.279   3.304   17.169   19.847   21.889   24.332   29.111 1.015
## b.vec[7]    23.264   3.522   17.293   20.870   22.635   25.248   30.658 1.036
## b.vec[8]    38.488   8.071   25.234   32.215   37.642   44.289   55.795 1.007
## mu.a      131.686   5.700  120.606  128.297  131.977  135.282  143.156 0.999
## mu.b      24.845   3.978   18.438   22.266   24.084   27.565   32.609 1.018
## sd.a      14.226   3.392   7.310    11.971   14.170   16.999   19.760 1.064
## sd.b      9.087    3.613   3.604    6.349    8.695    11.196   17.040 1.037
## sigma     8.244    0.411   7.515    7.919    8.222    8.496    9.086 1.017
## deviance  1523.397  6.499  1513.634  1518.874  1522.689  1526.226  1537.916 1.080
##
## n.eff
## a.vec[1]    150
## a.vec[2]     75
## a.vec[3]    150
## a.vec[4]    150
## a.vec[5]     68
## a.vec[6]    150
## a.vec[7]    140
## a.vec[8]    150
## b.vec[1]    150
## b.vec[2]     58
## b.vec[3]    150
## b.vec[4]    150
## b.vec[5]     67
## b.vec[6]    140
## b.vec[7]     76
## b.vec[8]    120
## mu.a      150
## mu.b      150
## sd.a       83
## sd.b       75
## sigma     150
## deviance   35
##
## For each parameter, n.eff is a crude measure of effective sample size,
## and Rhat is the potential scale reduction factor (at convergence, Rhat=1).
##
## DIC info (using the rule, pD = var(deviance)/2)
## pD = 20.2 and DIC = 1543.5
## DIC is an estimate of expected predictive error (lower deviance is better).

```

Unlike `lmer()`, `glmer()`, and `n1mer()`, an MCMC output automatically contains estimates of the region in which value of each parameter is likely to fall. However, instead of being called “confidence intervals” (CI), these intervals are called **credible intervals (CRI)**. The difference between a CI and a CRI is in how they are calculated. CI are calculated using the estimated mean, SE, and properties of the normal distribution. CRI, on the other hand, are simply quantiles of the

posterior distribution of the parameter. This means that CRI are inherently Bayesian and only defined in Bayesian inference; CI are frequentist and used in non-Bayesian context. CRI have the distinct advantage of not assuming anything about the appropriate distribution for the parameter.

Other outputs of MCMC can be used to calculate model predictions and their 95% CRI. Let's use the posterior draws to calculate model predictions. The posteriors are stored in several ways within the JAGS output. For generating predictions and CRI, we should use the values in the `sims.matrix` part of the output, which have already been permuted (this reduces bias in the predictions).

```

po <- mod3j$BUGSoutput$sims.matrix
nsim <- nrow(po)
a.cols <- grep("a.vec", colnames(po))
b.cols <- grep("b.vec", colnames(po))

pn <- 100
x <- seq(min(dat$conc), max(dat$conc), length=pn)

# version 1: full random effects
dx1 <- expand.grid(subs=1:nlev, conc=x)
dy1 <- matrix(NA, nrow=nrow(dx1), ncol=nsim)
for(i in 1:nrow(dx1)){
  ilev <- dx1$subs[i]
  use.ac <- a.cols[ilev]
  use.bc <- b.cols[ilev]
  for(j in 1:nsim){
    use.a <- po[j, use.ac]
    use.b <- po[j, use.bc]
    dy1[i,j] <- (use.a*dx1$conc[i])/(use.b+dx1$conc[i])
  }
}
dx1$lo <- apply(dy1, 1, quantile, 0.025)
dx1$mn <- apply(dy1, 1, quantile, 0.5)
dx1$up <- apply(dy1, 1, quantile, 0.975)

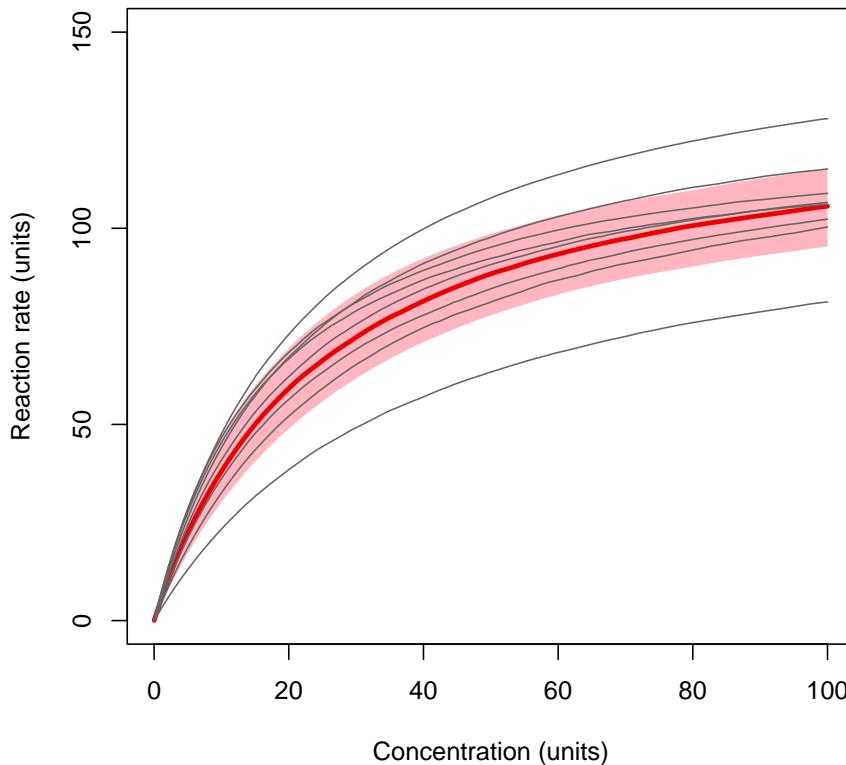
# version 2: no random effects
dx2 <- data.frame(conc=x)
dy2 <- matrix(NA, nrow=nrow(dx2), ncol=nrow(po))
for(i in 1:nrow(dx2)){
  for(j in 1:nsim){
    dy2[i,j] <- po[j, "mu.a"]*dx2$conc[i]/(po[j, "mu.b"]+dx2$conc[i])
  }
}
dx2$lo <- apply(dy2, 1, quantile, 0.025)

```

```
dx2$mn <- apply(dy2, 1, quantile, 0.5)
dx2$up <- apply(dy2, 1, quantile, 0.975)
```

Now make the plot.

```
plot(dx2$conc, dx2$mn, type="n",
      xlab="Concentration (units)",
      ylab="Reaction rate (units)",
      ylim=c(0, 150))
polygon(x=c(dx2$conc, rev(dx2$conc)),
        y=c(dx2$lo, rev(dx2$up)),
        border=NA, col="lightpink")
points(dx2$conc, dx2$mn, type="l", lwd=3, col="red2")
for(i in 1:nlev){
  flag <- which(dx1$subs==i)
  points(dx1$conc[flag], dx1$mn[flag],
         type="l", col="grey40")
}
```

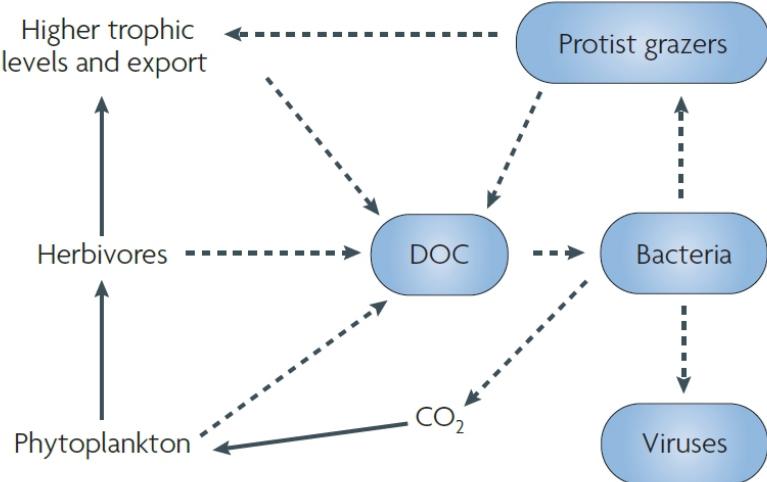


The plot shows the curves for individual substrates as thin lines, and the overall prediction with 95% CRI as a red line and shaded area. The highlighted curve and CRI is the expectation for any level of the random effect factor not in the original dataset. In other words, it is what the model predicts given the variability in the levels of the random factor that have already been observed. The ability to make robust predictions about unobserved levels of the random factor is one of the biggest advantages of mixed models.

7.4.3 NLME on real data

Kirchman et al. (2009) investigated the role of temperature and organic carbon availability on growth and secondary production by heterotrophic bacteria in the oceans. Heterotrophic bacteria are key components of marine food webs and can exert enormous control over ecosystem cycling of nutrients. The figure below shows the place of heterotrophic bacteria in the marine “microbial loop” that relates dissolved CO₂ and dissolved organic carbon (DOC) (Kirchman et al.

2009).

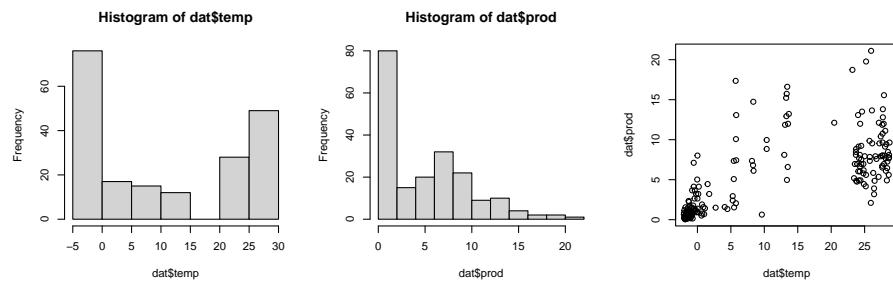


To better understand how these bacteria affect ocean ecosystems, the authors examined bacterial growth (d^{-1}) and productivity ($\text{mmol C m}^{-2} \text{ d}^{-1}$) varied with latitude ($^{\circ}\text{N}$ or $^{\circ}\text{S}$), nutrient availability ($\mu\text{M C}$), and mean annual temperature ($^{\circ}\text{C}$). We will re-examine part of their dataset to practice using nonlinear mixed effects models. The data are stored in this text file. Download this file and save it to your R working directory.

```
in.name <- "kirchman_data_2021-10-21.csv"
dat <- read.csv(in.name, header=TRUE)
```

As usual, we start by exploring the distributions of the variables and a basic scatterplot.

```
par(mfrow=c(1,3))
hist(dat$temp)
hist(dat$prod)
plot(dat$temp, dat$prod)
```

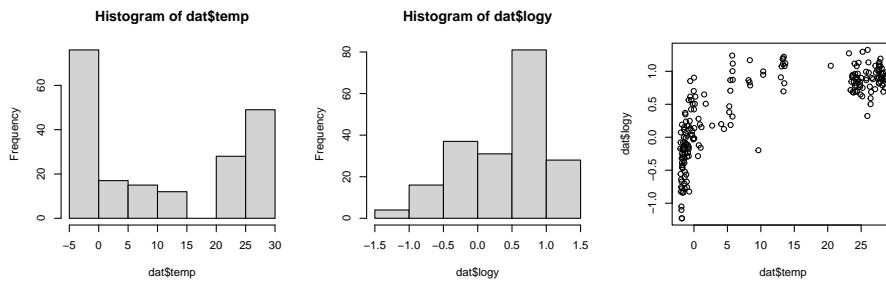


The figure suggests that the response variable, productivity (`prod`), is right-skewed. So, it might benefit from a log transform. The scatterplot backs this up—notice the “dust bunny” in the lower right. It’s not clear whether or not the predictor variable temperature (`temp`) needs to be transformed. Let’s leave it alone for now.

Make a log-transformed version of productivity and a new set of exploratory figures. We’ll use \log_{10} to save ourselves some trouble later.

```
dat$logy <- log10(dat$prod)

par(mfrow=c(1,3))
hist(dat$temp)
hist(dat$logy)
plot(dat$temp, dat$logy)
```



There appears to be a nonlinear relationship between log-transformed productivity and temperature. The values start small and increase quickly until about 2 °C, after which the productivity levels off. Can you think of a biological interpretation of this curve shape?

There seem to be some distinct clumps in the scatterplot which might correspond to the grouping variable `sea`. Let’s add some colors to the dataset and then to the plot. Just for fun, let’s make a color ramp from colder to warmer sites.

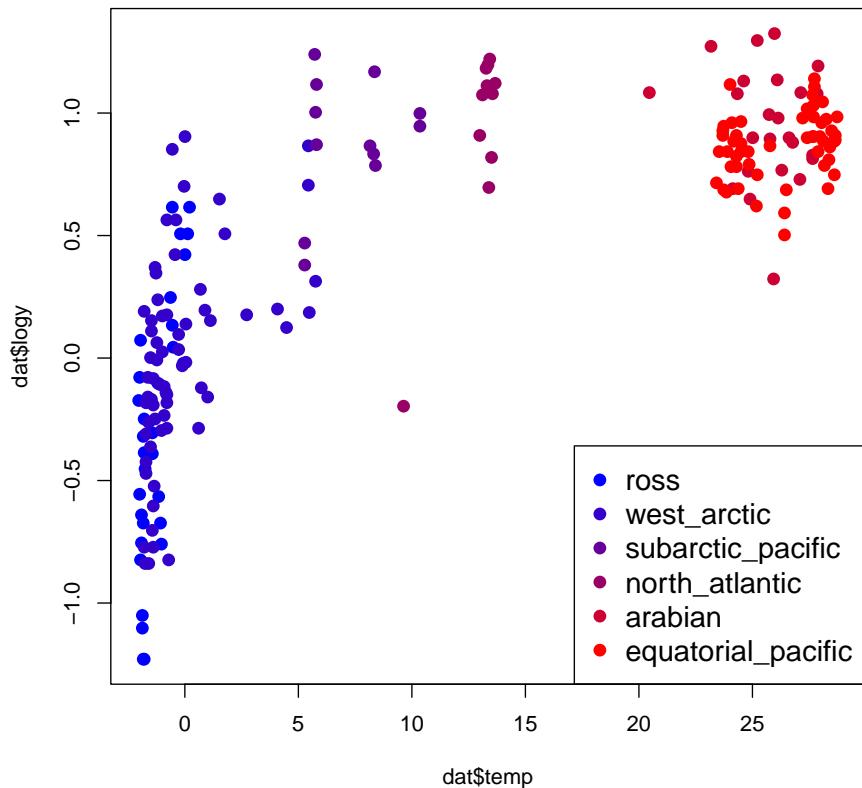
```
agg <- aggregate(temp~sea, data=dat, median)
agg <- agg[order(agg$temp),]
nsea <- nrow(agg)
seas <- agg$sea

# color ramp from colder (blue) to warmer (red)
col.fun <- colorRampPalette(c("blue", "red"))
cols <- col.fun(nsea)
agg$col <- cols

matchx <- match(dat$sea, agg$sea)
```

```
dat$col <- agg$col[matchx]

par(mfrow=c(1,1))
plot(dat$temp, dat$logy,
     pch=16, col=dat$col, cex=1.3)
legend("bottomright", legend=seas,
       pch=16, col=cols, cex=1.3)
```

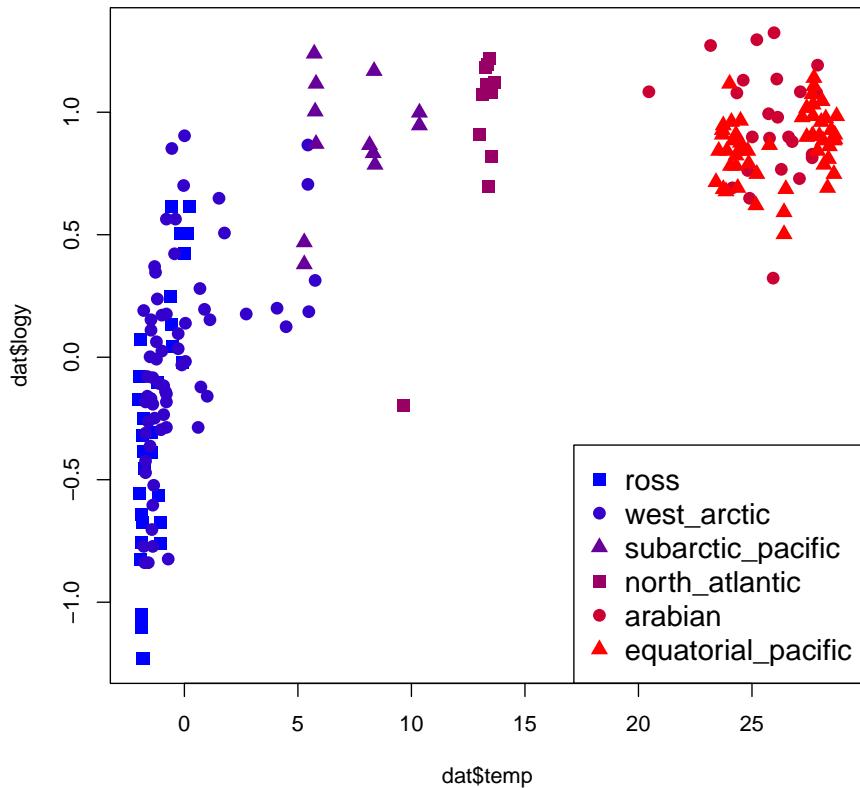


As nice as the color ramp effect is, the seas are still hard to tell apart. Let's add some shapes as well.

```
agg$shp <- 15:17
dat$shp <- agg$shp[matchx]

par(mfrow=c(1,1))
```

```
plot(dat$temp, dat$logy,
  pch=dat$shp, col=dat$col, cex=1.3)
legend("bottomright", legend=seas,
  pch=agg$shp, col=cols, cex=1.3)
```



There appears to be an effect of temperature and possibly of sea. Biologically, the bacteria within a sea are probably more similar to each other than they are to bacteria in other seas. So, we should expect there to be some effect of sea just based on genetic relatedness or some other kind of similarity. However, we want a model that is applicable to all seas, not just the 6 in this dataset. For example, this dataset is missing the Indian Ocean, the Mediterranean, the Caribbean, the Sea of Japan, and many others. So, we need a nonlinear *mixed model*. We don't know exactly what curve to use, so we'll try a few. The table below shows the equations for each model.

Model	Function	Fixed effects	Random effects
1	Monod, random a	$\mu_i = \frac{a_j x_i}{b+x_i}$	$a_j \sim Normal(\mu_a, \sigma_a^2)$
2	Monod, random b	$\mu_i = \frac{ax_i}{b_j+x_i}$	$b_j \sim Normal(\mu_b, \sigma_b^2)$
3	Monod, random a and b	$\mu_i = \frac{a_j x_i}{b_j+x_i}$	$a_j \sim Normal(\mu_a, \sigma_a^2)$ $b_j \sim Normal(\mu_b, \sigma_b^2)$
4	Holling type III, random a	$\mu_i = \frac{a_j x_i^2}{b^2+x_i^2}$	$a_j \sim Normal(\mu_a, \sigma_a^2)$
5	Holling type III, random b	$\mu_i = \frac{ax_i^2}{b_j^2+x_i^2}$	$b_j \sim Normal(\mu_b, \sigma_b^2)$
6	Holling type III, random a and b	$\mu_i = \frac{a_j x_i^2}{b^2+x_i^2}$	$a_j \sim Normal(\mu_a, \sigma_a^2)$ $b_j \sim Normal(\mu_b, \sigma_b^2)$
7	Monomolecular, $\mu_i =$ random a	$a_j (1 - e^{-bx})$	$a_j \sim Normal(\mu_a, \sigma_a^2)$
8	Monomolecular, $\mu_i =$ random b	$a (1 - e^{-b_j x})$	$b_j \sim Normal(\mu_b, \sigma_b^2)$
9	Monomolecular, $\mu_i =$ random a and b	$a_j (1 - e^{-b_j x})$	$a_j \sim Normal(\mu_a, \sigma_a^2)$ $b_j \sim Normal(\mu_b, \sigma_b^2)$

Fitting the **Monod** (aka: Michaelis-Menten) model with mixed effects is simple, because there is a self-start version of the function designed for nonlinear modeling (we've used it before). We just need to remember to supply reasonable starting values.

```
library(lme4)
# Monod (aka: Michaelis-Menten) with
# different random structures
startvec <- c(a=1.3, b=1)
mod01 <- nlmer(logy~SSmicmen(temp, a, b)~a|sea,
                 data=dat, start=startvec)
mod02 <- nlmer(logy~SSmicmen(temp, a, b)~b|sea,
                 data=dat, start=startvec)
mod03 <- nlmer(logy~SSmicmen(temp, a, b)~(a|sea)+(b|sea),
                 data=dat, start=startvec)
```

The other two curves are less simple to fit because there is no self-start model available. In `nlmer()`, the fixed part of the model needs to be provided as a

special type of function that takes in input values and parameter values, and outputs a vector of response values. Additionally, the function needs to have some internal attributes that let `nlmer()` evaluate its likelihood function and how the likelihood function changes. The procedure below seems to work well.

```
# Holling type III (Bolker 2008 p. 92)
startvec <- c(a=1.3, b=1)

# define model as function
holl <- ~(a*x^2)/(b^2 + x^2)

# prepare version of function for nlmer()
hollfun <- deriv(holl, namevec=c("a", "b"),
  function.arg=c("x", "a", "b"))

# fit models
mod04 <- nlmer(logy~hollfun(temp, a, b)~a|sea,
  data=dat, start=startvec)
mod05 <- nlmer(logy~hollfun(temp, a, b)~b|sea,
  data=dat, start=startvec)

## Warning in (function (fn, par, lower = rep.int(-Inf, n), upper = rep.int(Inf, :
## failure to converge in 10000 evaluations

## Warning in optwrap(control$optimizer[[1]], devfun, rho$pp$theta, lower =
## rho$lower, : convergence code 4 from Nelder_Mead: failure to converge in 10000
## evaluations

## Error in fn(nM$xeval()): prss{Update} failed to converge in 'maxit' iterations
mod06 <- nlmer(logy~hollfun(temp, a, b)~(a|sea)+(b|sea),
  data=dat, start=startvec)

## Error in fn(nM$xeval()): prss{Update} failed to converge in 'maxit' iterations
# monomolecular (Bolker p. 95)
# special case of von Bertalanffy curve

# define model as function
mono <- ~a*(1-exp(-b*x))

# prepare version of function for nlmer()
monofun <- deriv(mono, namevec=c("a", "b"),
  function.arg=c("x", "a", "b"))

mod07 <- nlmer(logy~monofun(temp, a, b)~a|sea,
  data=dat, start=startvec)
mod08 <- nlmer(logy~monofun(temp, a, b)~b|sea,
  data=dat, start=startvec)
```

```
mod09 <- nlmer(logy~monofun(temp, a, b)~(a|sea)+(b|sea),
  data=dat, start=startvec)
```

R was not able to find solutions for models 5 and 6, so we can discard those. Let's examine the outputs of the models that did fit. First, we can use AIC to get a sense of which models might have worked best:

```
aic.df <- AIC(mod01, mod02, mod03, mod04, mod07, mod08, mod09)
aic.df$delta <- aic.df$AIC - min(aic.df$AIC)
aic.df$wt <- exp(-0.5*aic.df$delta)
aic.df$wt <- aic.df$wt/sum(aic.df$wt)
aic.df <- aic.df[order(-aic.df$wt),]
aic.df
```

	df	AIC	delta	wt
## mod08	4	105.4030	0.000000	6.545609e-01
## mod09	5	107.4030	1.999977	2.408023e-01
## mod07	4	109.0699	3.666938	1.046368e-01
## mod04	4	156.6238	51.220848	4.937240e-12
## mod01	4	164.5428	59.139802	9.416842e-14
## mod03	5	166.5428	61.139802	3.464263e-14
## mod02	4	376.3158	270.912825	9.726972e-60

The AICs suggest that model 8 is the best-fitting. This was the monomolecular curve with the growth constant (b) varying by sea. The Monod functions were the worst fitting (by AIC); and the only Holling curve that fit was somewhere in the middle. Let's generate and plot model predictions as usual. We'll have to calculate the predictions manually because the `predict()` method for `nlmer()` objects is not well documented.

```
# number of points and x values
pn <- 50
px <- seq(min(dat$temp), max(dat$temp), length=pn)

# predictions by level (sea)
dx1 <- expand.grid(sea=seas, temp=px)
dx1 <- dx1[order(dx1$sea, dx1$temp),]

## coefficients table (including random b)
cofs <- coef(mod08)$sea
matchx <- match(dx1$sea, rownames(cofs))
dx1$a <- cofs$a[matchx]
dx1$b <- cofs$b[matchx]

## calculate predictions
dx1$y <- dx1$a*(1-exp(-dx1$b*dx1$temp))
```

```
# prediction of overall mean (applicable to other seas)
dx2 <- data.frame(temp=px)
fixe <- fixef(mod08)
dx2$y <- fixe["a"]*(1-exp(-fixe["b"]*dx2$temp))
```

Now we can assemble a few pieces and make our nice plot.

```
# 40% opacity (alpha) to the colors for
# clearer plot (pun intended)
dat$col2 <- paste0(dat$col, "40")

# nicer version of names for legend
leg.labs <- c(
  "Ross Sea", "Western Arctic",
  "Subarctic Pacific", "North Atlantic",
  "Arabian Sea", "Equatorial Pacific")

# graphical parameters for attractive plot
par(mfrow=c(1,1), mar=c(5.1, 5.1, 1.1, 1.1),
  lend=1, las=1, bty="n",
  cex.axis=1.3, cex.lab=1.3)
# set up plot area
## note expression() used for proper axis labels
## and y axis suppressed (yaxt="n") so a better
## y axis can be added manually
plot(dat$temp, dat$logy, type="n",
  xlab=expression(Temperature~(degree*C)),
  ylab=expression(Productivity~(mmol~C~m^{-2}~d^{-1})),
  yaxt="n",
  ylim=c(-1.5, 2))

# add sea-specific lines in a loop
for(i in 1:nsea){
  flag <- which(dx1$sea == seas[i])
  points(dx1$temp[flag], dx1$y[flag],
    type="l", col="grey60")
}

# add fixed effect line
points(dx2$temp, dx2$y, type="l", lwd=3)

# original data, with semi-transparent colors
points(dat$temp, dat$logy,
  pch=dat$shp, col=dat$col2, cex=1.3)

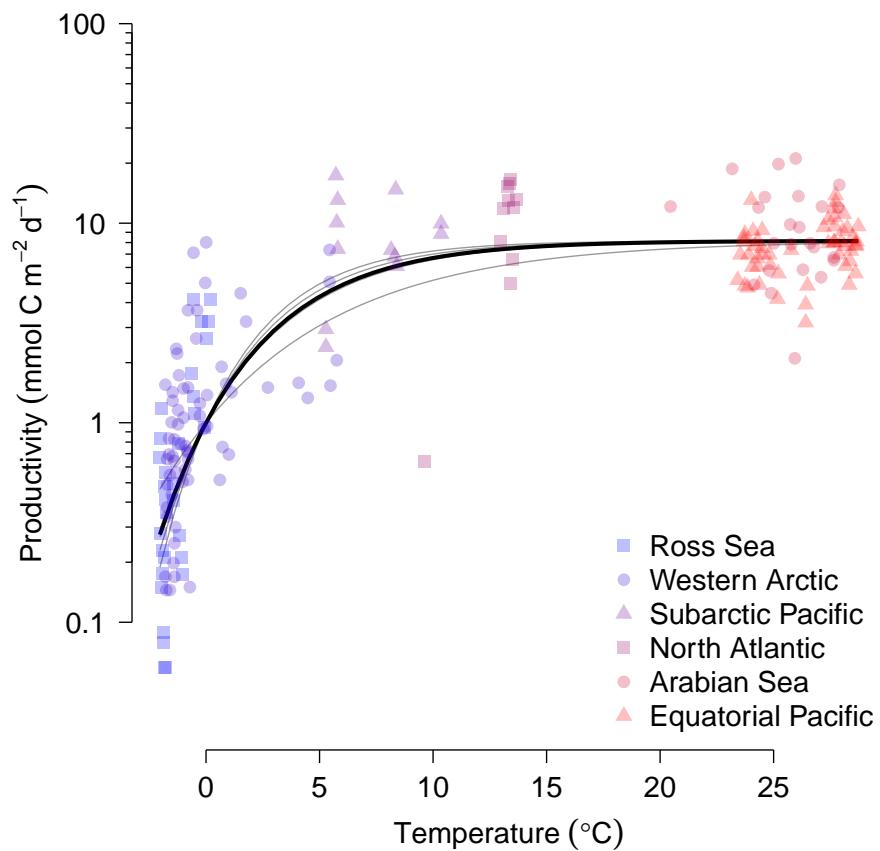
# fancy y axis with logarithmic tick marks
```

```

axis(side=2, at=seq(-1, 2, by=1),
     labels=c(0.1, 1, 10, 100))
axis(side=2, at=log10(c(2:9/10, 2:9, 2:9*10)),
     labels=NA, tcl=-0.25)

# legend
legend("bottomright", legend=leg.labs, pch=agg$shp,
       col=paste0(cols, "40"), cex=1.3, bty="n")

```



7.5 (Generalized) additive mixed models (AMM/GAMM)

The last category of mixed models that we will cover are **generalized additive mixed models (GAMM)**. GAMM extend generalized additive models (GAM) by allowing for observations in different groups to have different intercepts or different smoothing functions. If that sounds complicated, that's because it is.

- **Additive models (AM)** are a generalization of linear models, where a **smoother** of arbitrary shape is fit instead of a straight line. Smoothers can describe nonlinear patterns (very common in biology), but the process of fitting smoothers is sensitive to many assumptions and methodological details.
- **Generalized additive models (GAM)** generalize AM by relaxing the assumption of normal residuals and introducing a link function to additive models. This adds a layer of complexity to additive modeling¹⁴.
- **Generalized additive mixed models (GAMM)** generalize GAM by allowing different groups of observations to have different smoothers, or different coefficients on those smoothers. This adds even more complexity to the model.

Going from a linear model to a generalized linear model adds a layer of complication. Going from a fixed effects model to a mixed effects model adds more complication. And going from a linear model to an additive one compounds all of these complications. I'm not trying to warn you away from trying GAMM—just advising you to be prepared for frustration.

7.5.1 Example GAMM with real data

Pethybridge et al. (2018) compiled data on stable isotope concentrations in 3 species of tuna throughout the Atlantic, Pacific, and Indian Oceans. They used these concentrations to estimate the trophic position of each species. The data were made available in a separate publication (Bodin et al. 2021). Download this adapted version of the dataset and save it to your R home directory (or other favorite folder). Import the data and take a look.

```
in.name <- "bodin_data_2021-10-22.csv"
dat <- read.csv(in.name, header=TRUE)
# remove observations missing trophic position
dat <- dat[!is.na(dat$tp),]

# inspect dataset
head(dat)

##          species      common       date      lat      long ocean fork   d15n
#> 1  Thunnus thynnus Thunnus thynnus 2010-01-01 30.000 150.000  Atlantic  100  -15.5
#> 2  Thunnus thynnus Thunnus thynnus 2010-01-01 30.000 150.000  Atlantic  100  -15.5
#> 3  Thunnus thynnus Thunnus thynnus 2010-01-01 30.000 150.000  Atlantic  100  -15.5
#> 4  Thunnus thynnus Thunnus thynnus 2010-01-01 30.000 150.000  Atlantic  100  -15.5
#> 5  Thunnus thynnus Thunnus thynnus 2010-01-01 30.000 150.000  Atlantic  100  -15.5
#> 6  Thunnus thynnus Thunnus thynnus 2010-01-01 30.000 150.000  Atlantic  100  -15.5
```

¹⁴Going from additive models to generalized additive models is similar to going from linear models to generalized linear models.

```

## 1 Thunnus alalunga Albacore tuna 2002-10-10 -16.64 -150.79   PO  91 15.17
## 2 Thunnus alalunga Albacore tuna 2004-16-09 -25.85  153.98   PO  87  9.55
## 3 Thunnus alalunga Albacore tuna 2004-16-09 -25.85  153.98   PO  99  9.48
## 4 Thunnus alalunga Albacore tuna 2004-20-09 -28.17  155.82   PO  78 10.46
## 5 Thunnus alalunga Albacore tuna 2004-23-09 -28.37  160.13   PO  76 12.23
## 6 Thunnus alalunga Albacore tuna 2004-23-09 -28.37  160.13   PO  80 10.35
##      tp
## 1 7.32
## 2 4.98
## 3 4.95
## 4 5.36
## 5 6.10
## 6 5.31

```

The dataset has the following variables:

Variable	Units	Definition
species	N/A	Scientific name (binomen) of species.
common	N/A	Common name of species.
date	Days	Date of observation in YYYY-MM-DD format.
lat	°N	Latitude of observation. Positive values indicate °N; negative values indicate °S.
long	°E	Longitude of observation. Positive values indicate °E; negative values indicate °W.
ocean	N/A	Ocean where observation was taken. AO = Atlantic Ocean; IO = Indian Ocean; PO = Pacific Ocean.
fork	cm	Fork length (FL) of individual fish (distance from tip of snout to middle caudal fin rays). Slightly less than total length (TL).
d15n	Parts per thousand	Ratio of ¹⁵ N to ¹⁴ N in sample (aka: $\delta^{15}\text{N}$ or “delta fifteen N”)

Variable	Units	Definition
tp	Tropic levels	Estimated trophic level of individual based on $\delta^{15}\text{N}$ in tissue samples.

The researchers were interested in how the trophic level of tuna varied with body size and geography. The trophic level expresses how high in the food chain an animal is. Primary producers such as plants and phytoplankton have a trophic level of 1. Animals that consume primary producers have a trophic level of 2. For example, grasses have trophic level 1, zebras have trophic level 2, and lions have trophic level 3.

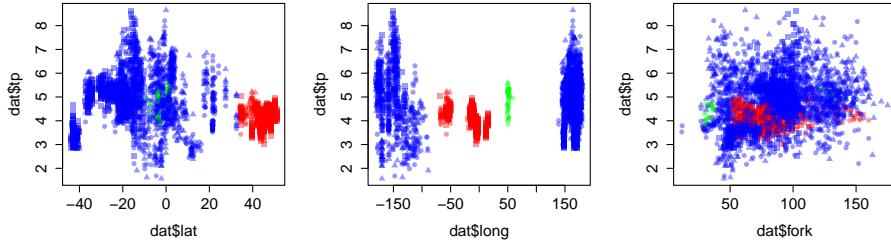
The variables in play are animal size (fork length), latitude, longitude, ocean, and species. Fork length, latitude, and longitude can be modeled as continuous predictors. Ocean and species are factors. Ocean has 3 levels (Atlantic, Indian, and Pacific), as does species (*Thunnus albacares*, *T. obesus*, and *T. alalunga*). Which of these factors should be treated as fixed? Which as random? It depends on the researchers' goals. If the objective is to predict the trophic position of particular species, or to predict the trophic position of species within particular oceans, then those factors should be fixed. If the sets of oceans and species in the dataset are just taken as representing other, unexamined oceans and/or species, then these factors should be treated as random. We will follow the example of the original authors and treat ocean as a random factor, with species as fixed.

Now that we have decided on a mixed model approach, we need to decide whether to use linear models or additive models. A quick plot of the data, with species and ocean overlaid as shapes and colors, respectively, suggests that linear models are unlikely to describe the data very well. So, we can try an additive model.

```
# color by ocean
ocs <- sort(unique(dat$ocean))
cols <- rainbow(length(ocs), alpha=0.4)
dat$col <- cols[match(dat$ocean, ocs)]

# shape by species
spps <- sort(unique(dat$species))
pchs <- 15:17
dat$pch <- pchs[match(dat$species, spps)]

par(mfrow=c(1,3), cex.lab=1.4, cex.axis=1.4)
plot(dat$lat, dat$tp, col=dat$col, pch=dat$pch)
plot(dat$long, dat$tp, col=dat$col, pch=dat$pch)
plot(dat$fork, dat$tp, col=dat$col, pch=dat$pch)
```



Next, we need to consider what kind of additive model to use. The decision of whether to use an additive mixed model (AMM) or generalized additive mixed model (GAMM) hinges on the distribution of the response variable. Trophic positions cannot be negative (or <1), so we might think of the values as coming from a lognormal distribution. This would be a GAMM with Gaussian family and log link function.

The main R package for fitting GAMM is the same package for fitting GAM: `mgcv` (Wood 2017). It should be noted that there are several ways within `mgcv` to fit GAMM. We'll use the function `gamm()`; function `gam()` can also be used by manipulating some of the arguments to the smoother functions `s()`.

For this example we will fit 3 models, each with smoothers for a single continuous predictor. Each species will have its own smoother: this is what the `by=spp` argument in `s()` does. The random intercept of `ocean` is specified as a list.

```
library(mgcv)

# very important: make a factor version of species to be
# used by gamm()
dat$spp <- factor(dat$species)

# fit models
mod01 <- gamm(tp~s(lat, by=spp), data=dat,
               random=list(ocean=~1), family=gaussian(link="log"))

##
## Maximum number of PQL iterations: 20
## iteration 1
## iteration 2
## iteration 3
## iteration 4
mod02 <- gamm(tp~s(long, by=spp), data=dat,
               random=list(ocean=~1), family=gaussian(link="log"))
```

```

##  

## Maximum number of PQL iterations: 20  

## iteration 1  

## iteration 2  

## iteration 3  

## iteration 4  

## iteration 5  

mod03 <- gamm(tp~s(fork, by=spp), data=dat,  

  random=list(ocean=~1), family=gaussian(link="log"))

```

```

##  

## Maximum number of PQL iterations: 20  

## iteration 1  

## iteration 2  

## iteration 3  

## iteration 4

```

Outputs from `gamm()` are very complicated and intimidating, even to experts. The most important part of the output is in the `$gam` component.

```

summary(mod01$gam, cor=FALSE)

##  

## Family: gaussian  

## Link function: log  

##  

## Formula:  

## tp ~ s(lat, by = spp)  

##  

## Parametric coefficients:  

##             Estimate Std. Error t value Pr(>|t|)  

## (Intercept) 1.64874    0.01467   112.4   <2e-16 ***  

## ---  

## Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1  

##  

## Approximate significance of smooth terms:  

##                      edf Ref.df      F p-value  

## s(lat):sppThunnus alalunga 7.257 7.257 122.11 <2e-16 ***  

## s(lat):sppThunnus albacares 8.743 8.743 27.61 <2e-16 ***  

## s(lat):sppThunnus obesus    8.661 8.661 20.05 <2e-16 ***  

## ---  

## Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

```

```
##  
## R-sq.(adj) =  0.365  
##   Scale est. = 0.68458   n = 3177
```

The output contains only one parametric component, the estimated intercept. The overall or mean intercept was estimated as 1.648 ± 0.015 , and was statistically significant. The “approximate significance” of smooth terms is presented in a table separate from the parametric components. The “approximate” part of the name is very important: take estimates of P -values, R^2 , and other such measures from GAMMs with a grain of salt. Even the help page for `summary.gam()` states that the P -value presented are likely to be too small.

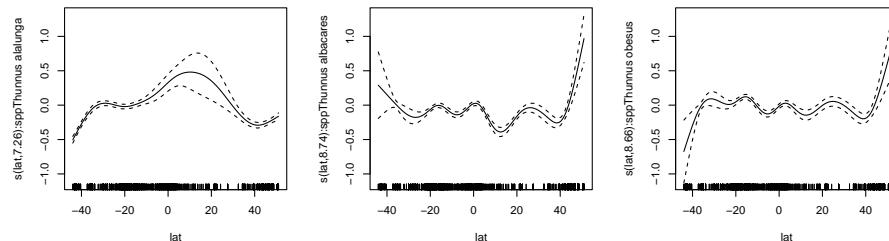
If you have a factor with >2 levels, you can print an ANOVA for the level-specific smoothers:

```
anova(mod01$gam)
```

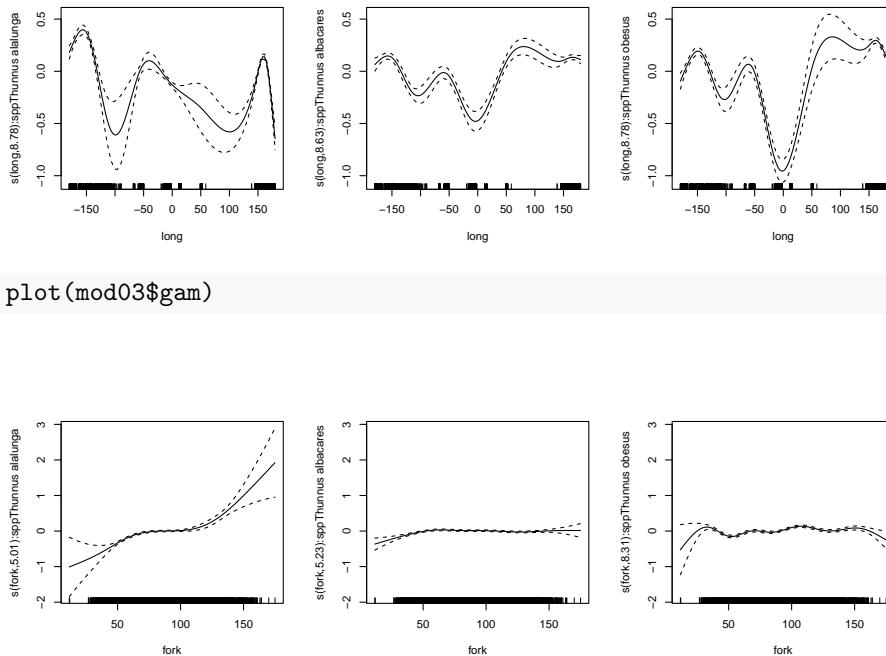
```
##  
## Family: gaussian  
## Link function: log  
##  
## Formula:  
## tp ~ s(lat, by = spp)  
##  
## Approximate significance of smooth terms:  
##  
##          edf Ref.df      F p-value  
## s(lat):sppThunnus alalunga 7.257 7.257 122.11 <2e-16  
## s(lat):sppThunnus albacares 8.743 8.743 27.61 <2e-16  
## s(lat):sppThunnus obesus    8.661 8.661 20.05 <2e-16
```

You can see the estimated smoothers using `plot()` on the `$gam` output.

```
par(mfrow=c(1,3)) # 1x3 panels, because 3 smoothers  
plot(mod01$gam)
```



```
plot(mod02$gam)
```



Some of the smoothers for the predictor variables have quite different shapes for each species! The original study authors fit a separate model for each species and continuous predictor combination. But, with this many observations (`nrow(dat)`), we might be able to fit a single model that includes all predictors.

```
# can we fit all the predictors in one model?
mod04 <- gamm(tp~s(lat, by=spp)+s(long, by=spp)+s(fork, by=spp),
  data=dat,
  random=list(ocean=~1),
  family=gaussian(link="log"))

## 
## Maximum number of PQL iterations: 20
## iteration 1
## iteration 2
## iteration 3
## iteration 4
# yes!
summary(mod04$gam)

##
```

```

## Family: gaussian
## Link function: log
##
## Formula:
## tp ~ s(lat, by = spp) + s(long, by = spp) + s(fork, by = spp)
##
## Parametric coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 1.8114     0.1406   12.88 <2e-16 ***
## ---
## Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Approximate significance of smooth terms:
##                               edf Ref.df      F p-value
## s(lat):sppThunnus alalunga    6.581  6.581 66.471 < 2e-16 ***
## s(lat):sppThunnus albacares   8.518  8.518 20.780 < 2e-16 ***
## s(lat):sppThunnus obesus     8.848  8.848 32.363 < 2e-16 ***
## s(long):sppThunnus alalunga   8.100  8.100 40.762 < 2e-16 ***
## s(long):sppThunnus albacares  8.020  8.020 23.551 < 2e-16 ***
## s(long):sppThunnus obesus    8.010  8.010 52.744 < 2e-16 ***
## s(fork):sppThunnus alalunga   3.660  3.660  8.713 1.64e-05 ***
## s(fork):sppThunnus albacares  6.447  6.447 14.814 < 2e-16 ***
## s(fork):sppThunnus obesus     7.625  7.625  4.212 4.67e-05 ***
## ---
## Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## R-sq.(adj) = -0.0545
## Scale est. = 0.48822 n = 3168

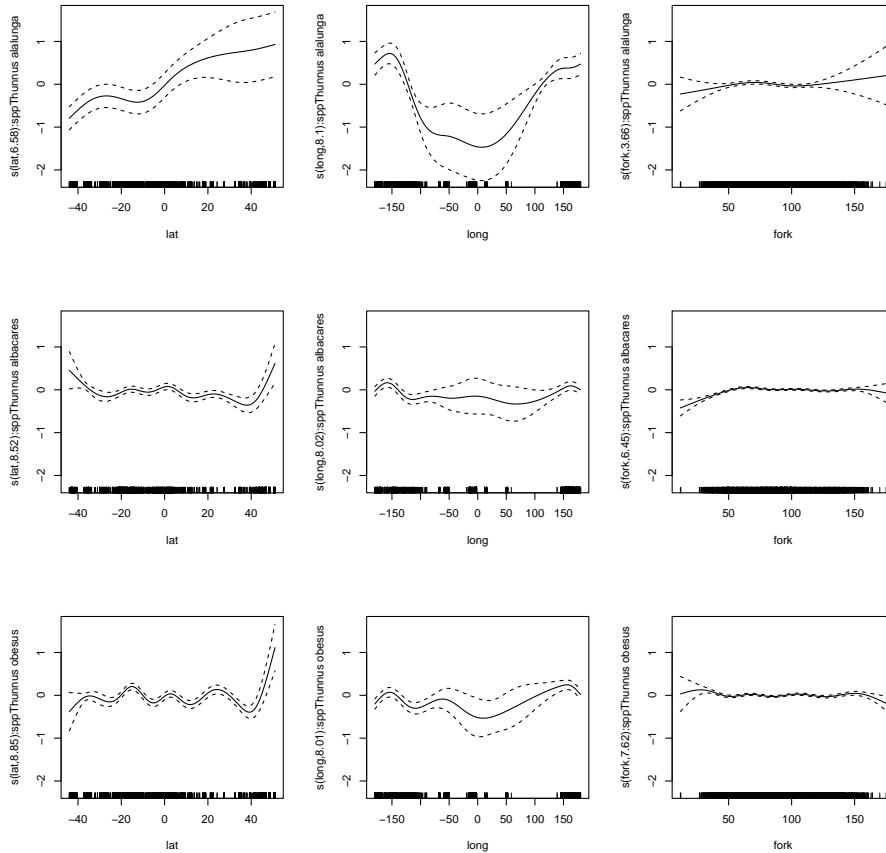
```

Examine the smoothers:

```

# 3 predictors * 3 species = 9 panels
# note use of mfcoll to fill by column,
# so species are rows and predictors are columns
par(mfcoll=c(3,3))
plot(mod04$gam)

```



The plots produced by plotting a `$gam` output show the effect of each smoother, not the actual predicted y value. The predicted value for an observation is obtained by adding the intercept to the effect of each smoother (this is what the “additive” in “additive model” means). For your manuscript, you probably want to show model predictions on the original data scale. The method below should work for most situations.

In the past, we’ve predicted response variables with single continuous predictor variables. For this example, we have 3 continuous predictors. The usual strategy a situation with >1 continuous predictor is to present model predictions for each predictor with the other predictors held at their median. This makes the bookkeeping a little complicated, but the final graph is well worth the effort.

```
# number of x values for prediction
pn <- 50

# sequence of each x value across its domain
```

```

px1 <- seq(min(dat$lat), max(dat$lat), length=pn)
px2 <- seq(min(dat$long), max(dat$long), length=pn)
px3 <- seq(min(dat$fork, na.rm=TRUE),
            max(dat$fork, na.rm=TRUE), length=pn)

# data frame of all factors and an index for row number (z)
dx <- expand.grid(ocean=NA, spp=spps, z=1:pn)
dx <- dx[order(dx$ocean, dx$spp, dx$z),]

# add median of each predictor to dx
dx$lat <- median(dat$lat)
dx$long <- median(dat$long)
dx$fork <- median(dat$fork, na.rm=TRUE)

# need separate data frame of predictions for
# each predictor
dx1 <- dx
dx2 <- dx
dx3 <- dx

# in each prediction data frame, assign ONE of
# the predictors its sequence of values, leaving
# other predictors at their median.
## works because of recycling rule:
dx1$lat <- px1
dx2$long <- px2
dx3$fork <- px3

# calculate predictions
pred1 <- predict(mod04$gam, newdata=data.frame(dx1),
                  type="link", se.fit=TRUE)
pred2 <- predict(mod04$gam, newdata=data.frame(dx2),
                  type="link", se.fit=TRUE)
pred3 <- predict(mod04$gam, newdata=data.frame(dx3),
                  type="link", se.fit=TRUE)

# add predictions and SE to data frames
dx1$mu <- pred1$fit
dx2$mu <- pred2$fit
dx3$mu <- pred3$fit

dx1$se <- pred1$se.fit
dx2$se <- pred2$se.fit
dx3$se <- pred3$se.fit

```

```
# normal approximation for 95% CI
dx1$lo <- qlnorm(0.025, dx1$mu, dx1$se)
dx2$lo <- qlnorm(0.025, dx2$mu, dx2$se)
dx3$lo <- qlnorm(0.025, dx3$mu, dx3$se)

dx1$mn <- qlnorm(0.5, dx1$mu, dx1$se)
dx2$mn <- qlnorm(0.5, dx2$mu, dx2$se)
dx3$mn <- qlnorm(0.5, dx3$mu, dx3$se)

dx1$up <- qlnorm(0.975, dx1$mu, dx1$se)
dx2$up <- qlnorm(0.975, dx2$mu, dx2$se)
dx3$up <- qlnorm(0.975, dx3$mu, dx3$se)

# add variable "panel" to help organize the plot later
dx1$panel <- rep(1:3, each=pn)
dx2$panel <- rep(4:6, each=pn)
dx3$panel <- rep(7:9, each=pn)
dz <- rbind(dx1, dx2, dx3)
dz <- dz[order(dz$panel),]
```

Next we make the plot.

```
# helps with bookkeeping
pan.df <- data.frame(panel=1:9,
  con=rep(c("lat", "long", "fork"), each=3),
  spp=rep(spps, times=3))

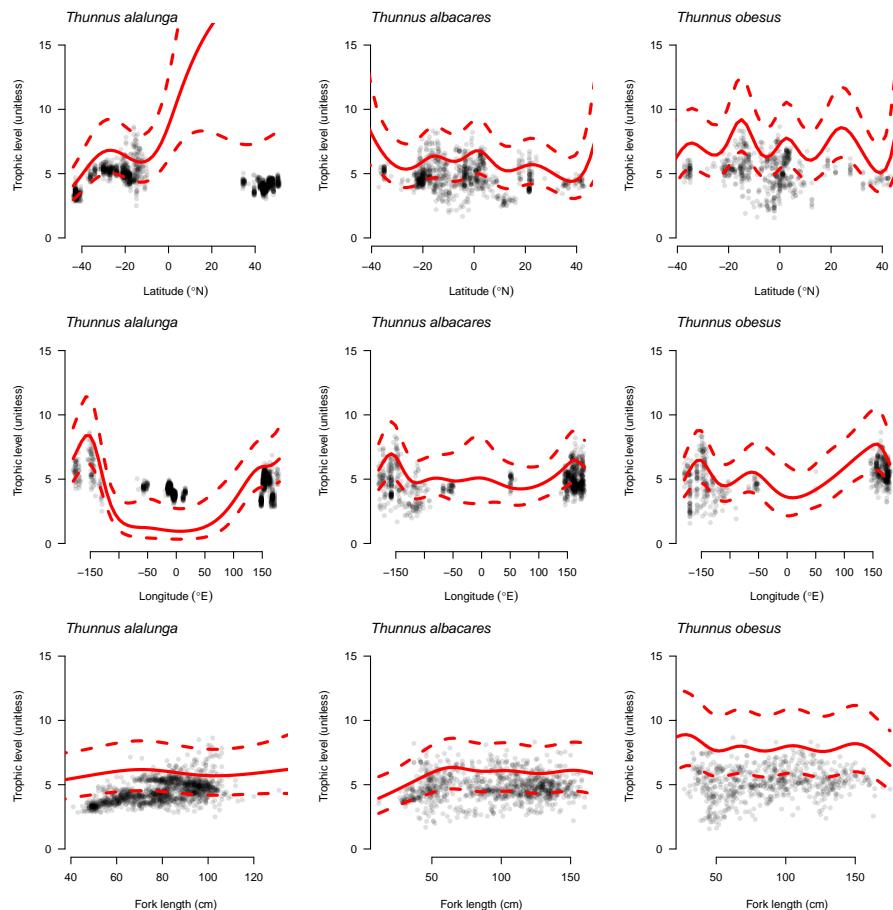
par(mfrow=c(3,3), mar=c(5.1, 5.1, 1.1, 1.1),
  bty="n", las=1)
for(i in 1:9){
  flag <- which(dz$panel == i)
  zi <- dz[flag,]
  xvar <- pan.df$con[i]
  x <- zi[,xvar]
  sppi <- pan.df$spp[i]
  di <- dat[which(dat$species == sppi),]

  if(xvar == "lat"){
    use.xlab <- expression(Latitude~(degree*N))
  }
  if(xvar == "long"){
    use.xlab <- expression(Longitude~(degree*E))
  }
  if(xvar == "fork"){
    use.xlab <- "Fork length (cm)"
  }
}
```

```

plot(di[,xvar], di$tp, pch=16, col="#00000020",
      ylim=c(0,16), xlab=use.xlab,
      ylab="Trophic level (unitless)")
points(x, zi$mn, type="l", lwd=3, col="red")
points(x, zi$lo, type="l", lwd=3, lty=2, col="red")
points(x, zi$up, type="l", lwd=3, lty=2, col="red")
title(main=pan.df$spp[i], adj=0, font.main=3)
}

```



The fitted smoothers are pretty good for some species and predictors, but not so much for others. Refinement of the analysis should involve revisiting what distribution to use and what random effects structure to include in the model. Fiddling with the options for fitting smoothers (see `?s`) might help as well.

Chapter 8

Multivariate data analysis

8.1 Multivariate data

8.1.1 Univariate vs. multivariate data

Things are not always one or two dimensional. Oftentimes in biology we need to consider many variables at once. This is the realm of **multivariate statistics**. The “multi” in “multivariate” refers to the fact that there are multiple variables or attributes of each sample that we want to understand simultaneously. The terminology can be confusing sometimes because other statistical methods have the word “multiple” in their names. For example, “multiple linear regression” is not considered a multivariate method because there is only one measurement from each sample (the response variable) that is being modeled.

The figure below shows some differences between a univariate analysis and a multivariate one. In the univariate analysis, one and only one variable is the response variable and is assumed to be predicted by (aka: depend on) the others. Frequently, the observations are divided *a priori* into groups by a factor encoded by one of the predictor variables. This means that the goal of a univariate analysis is to explain patterns in one variable relative to other variables. In a univariate analysis, the emphasis is on discovering relationships between variables, and the observations are a means to that end.

In a multivariate analysis, none of the variables are singled out and all are of interest for explaining differences or similarities between the observations. The observations may come from *a priori* groups defined in a separate matrix (see below), or the researcher may want to discover how the observations naturally separate into clusters based on the variables (i.e., *a posteriori* groups). Or, the researcher may want to know which variables, if any, drive most of the variation between observations. The key is that in a multivariate analysis the emphasis is often on discovering patterns between observations, and the variables are a

means to that end.

Univariate analysis (single response)					
Observation	Response	Predictor 1	Predictor 2	...	Predictor k
1	y_1	$X_{1,1}$	$X_{2,1}$...	$X_{k,1}$
2	y_2	$X_{1,2}$	$X_{2,2}$...	$X_{k,2}$
3	y_3	$X_{1,3}$	$X_{2,3}$...	$X_{k,3}$
...	\vdots	\vdots
n	y_n	$X_{1,n}$	$X_{2,n}$...	$X_{k,n}$

→ $E(y_i) = f(X_1, X_2, \dots, X_k; \theta)$

Multivariate analysis (many “responses”)					
Observation	Variable 1	Variable 2	Variable 3	...	Variable k
1	$X_{1,1}$	$X_{2,1}$	$X_{3,1}$...	$X_{k,1}$
2	$X_{1,2}$	$X_{2,2}$	$X_{3,2}$...	$X_{k,2}$
3	$X_{1,3}$	$X_{2,3}$	$X_{3,3}$...	$X_{k,3}$
...	\vdots	\vdots
n	$X_{1,n}$	$X_{2,n}$	$X_{3,n}$...	$X_{k,n}$

→ How do X_1, X_2, \dots, X_k vary together?
Do observations cluster in terms of X_1, X_2, \dots, X_k ?
What variables explain variation between observations?
And so on...

8.1.2 Components of multivariate data

When planning and executing a multivariate analysis it is very important to keep in mind which components of the dataset are which. These components are usually stored in matrices with a well-defined structure. The key concept is that of the **sample unit** or **observation**. These are individual observations about which many measurements or attributes are recorded. Sample units correspond to rows of the data matrix. Each sample unit has several **attributes** or **variables** associated with it. Variables correspond to columns of the data matrix. The figure below shows the layout of a typical multivariate dataset. The example has 12 observations and 6 variables . In some fields, samples are also called records and variables are called features.

	Variables					
	Var. 1	Var. 2	Var. 3	Var. 4	Var. 5	Var. 6
Observation 1						
Observation 2						
Observation 3						
Observation 4						
Observation 5						
Observation 6						
Observation 7						
Observation 8						
Observation 9						
Observation 10						
Observation 11						
Observation 12						

Complex datasets often contain data of different types, or sets of variables that capture information about different aspects of the system. For example, you might have variables that contain species abundances, optical densities, concentrations, femur lengths, treatment groups, and so on. When planning a multivariate analysis you need to consider whether and which variables should be grouped together. In an exploratory analysis, perhaps all of the variables are of interest and no subgroups are needed. In many situations, some of the variables will be the “response” variables. These contain the patterns that you are interested in explaining. Other variables might be considered “explanatory variables”. These are variables that may or may not help explain some of the patterns in the first set of variables. Other variables might contain information about the response variables. For all variables except the first set of “response” variables, the key is whether the values are associated with rows of the main data matrix shown above (i.e., with observations), or whether the values are associated with columns of the main data matrix (i.e., with other variables).

The figure below shows the relationship between three interrelated matrices typical in community ecology. Similar matrices can be defined for other fields...just replace “Environmental variables” with another label that helps describe variation

among the sample units.

- The data matrix **A** contains data on the abundance of 6 species at each of 12 sites.
- The explanatory matrix **E** contains 4 environmental variables (e.g., temperature, latitude, etc.) for each of the sites. These data describe relationships between observations in **A**.
- The trait matrix **S** contains data about traits (e.g., maximum size, longevity, etc.) of the species in the main matrix. These data describe relationships between variables in **A**.

The figure displays three matrices arranged horizontally. Matrix **A** (Species abundances) has rows labeled 'Observations = Sites' (Site 1 to Site 12) and columns labeled 'Spp. 1' to 'Spp. 6'. Matrix **E** (Environmental variables) has rows unlabeled and columns labeled 'Env. 1' to 'Env. 4'. Matrix **S** (Species traits) has rows labeled 'Species traits' (Trait 1 to Trait 4) and columns unlabeled. The letter labels **A**, **E**, and **S** are placed in their respective matrices: **A** is in the center of its grid, **E** is in the center of its grid, and **S** is in the center of its grid.

Species abundances						
	Spp. 1	Spp. 2	Spp. 3	Spp. 4	Spp. 5	Spp. 6
Site 1						
Site 2						
Site 3						
Site 4						
Site 5						
Site 6						
Site 7						
Site 8						
Site 9						
Site 10						
Site 11						
Site 12						

Environmental variables				
	Env. 1	Env. 2	Env. 3	Env. 4

Species traits			
Trait 1			
Trait 2			
Trait 3			
Trait 4			

A biological system of interest might have relationships among all three of these matrices:

- **Patterns within A:** Explaining groupings or trends among the sites in terms of the species present or absent at each site (or, the abundances of species at each site).
- **Patterns within E:** Discovering groupings among the sites in terms of their environmental variables.
- **Relationships between A and E:** Relating patterns in environmental variables to patterns in species abundances across sites.
- **Relating patterns between A and E in terms of S:** Relating species

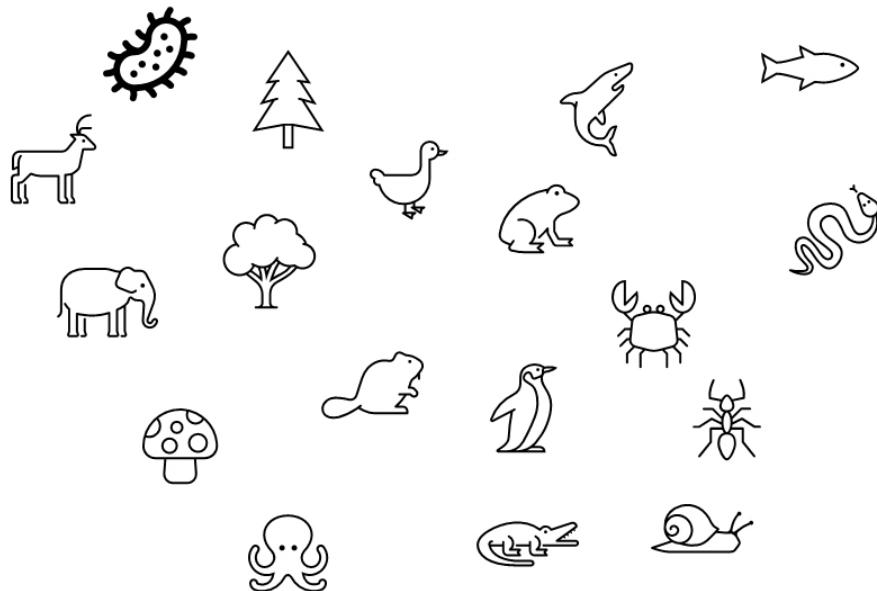
traits to environmental conditions.

- **Patterns within S:** Identifying clusters of similar species.

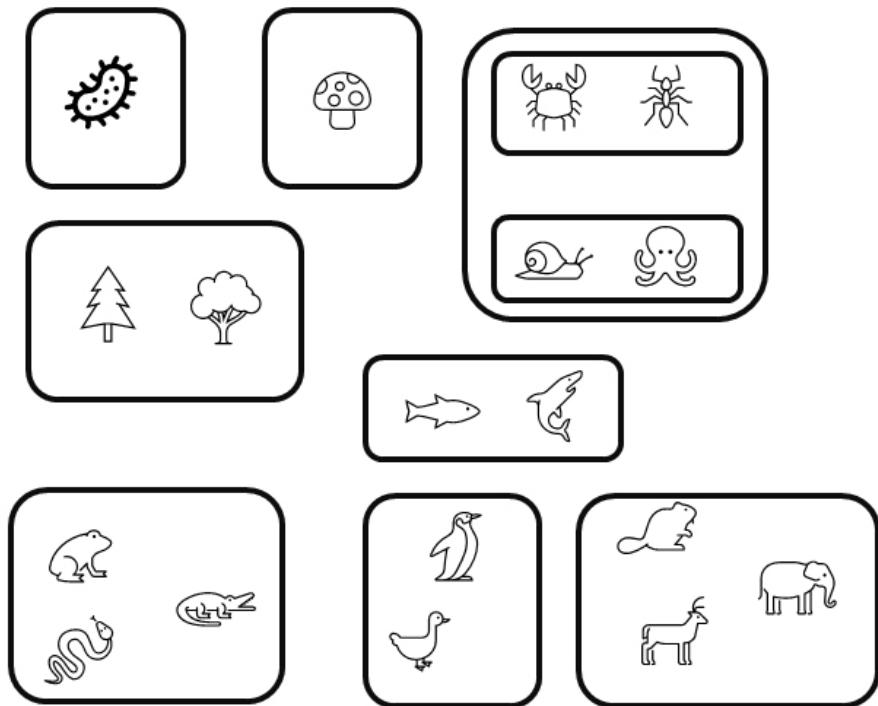
McCune et al. (2002) give a thorough treatment of the relationships between **S**, **E**, **A**, and other matrices derived from them. For this course, we will focus on analyzing **A** and sometimes **E**. There is a related set of terminology used to describe multivariate analyses in terms of what is being compared. Analyses that seek relationships among samples are said to be in **Q mode**; analyses that seek relationships among variables are in **R mode** (Legendre and Legendre 2012). These terms appear to be particular to ecology, but offer a helpful way to distinguish between different multivariate analysis pathways.

8.2 Distance metrics: biological (dis)similarity

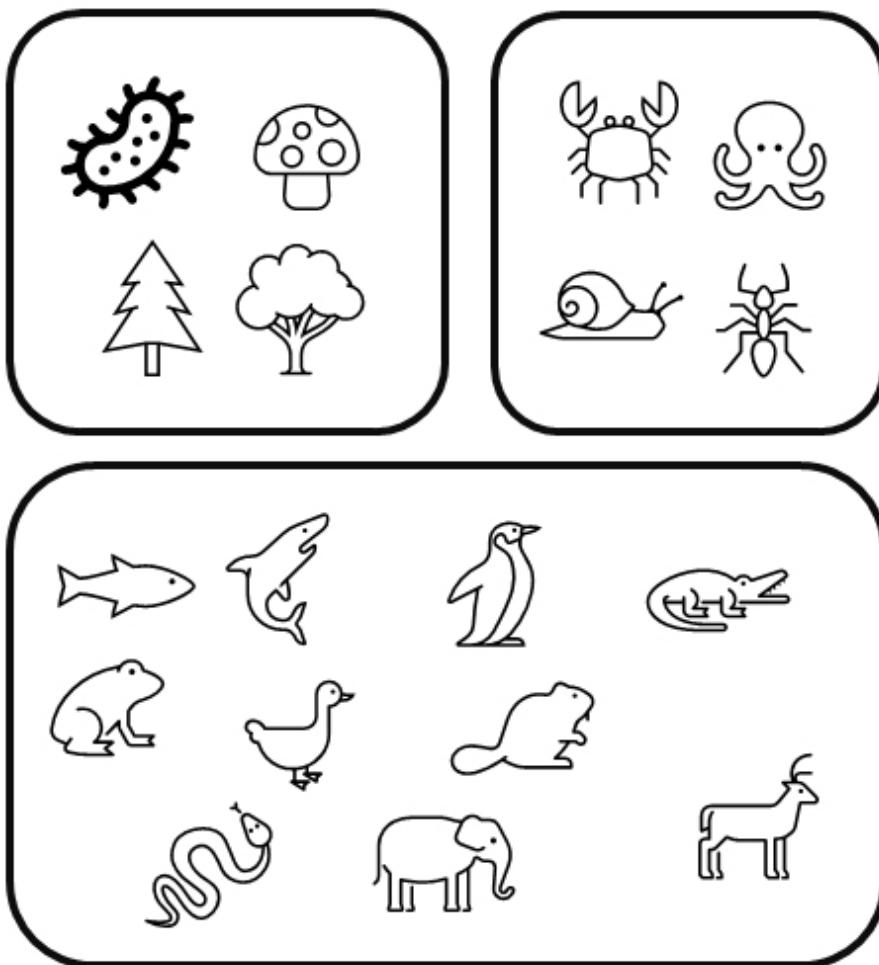
One of the central questions in a multivariate analysis is, “How similar to each other are my samples?” This is a more complicated question than it appears. It’s worth considering for a moment what “similar” even means. Consider the set of organisms below. How would you group them by “similarity”?



Here is one way:



You might have come up with a slightly different set of groupings. This one is just as valid:



These examples illustrate how we might classify the organisms based on qualitative characteristics. But do our choices weight each characteristic equally? After all, fungi and plants are far more different from each other than ants and fish, but the second scheme lumps all non-animals together. Which characteristics of the organisms are really driving our classification decisions?

One way around this problem is to assign numeric scores to characters: 1 if an organism has the trait, and 0 if it doesn't. The table below illustrates this for a few organisms and characteristics. Note that bacteria have NA for heterotrophy, because many bacteria are also autotrophs.

Taxon	Multicel.	Heterotroph.	Flower.	Vert.	Gills	Amnion	Endotherm.
Bacteria	0	NA	0	0	0	0	0
Maple	1	0	1	0	0	0	0

Taxon	Multicel.	Heterotroph.	Flower.	Vert.	Gills	Ammion	Endotherm.
Pine	1	0	0	0	0	0	0
Octopus	1	1	0	0	1	0	0
Shark	1	1	0	1	1	0	0
Mushroom	1	1	0	0	0	0	0
Frog	1	1	0	1	0	0	0
Elephant	1	1	0	1	0	1	1
Penguin	1	1	0	1	0	1	1
...							

The full table for our organisms has more columns and rows (we'll get to that later).

The next step is to somehow quantify the similarities between the organisms. It turns out that it's easier to define **dissimilarity** than it is to define similarity. This is because the range of a similarity metric is ill-defined: there is no natural value to assign to identical samples. I.e., what value should identical samples have? Infinity? Some other arbitrary value? The first isn't very useful, and the second is highly subjective. Dissimilarity, on the other hand, has a natural value to assign to identical samples: 0. Dissimilarity is often expressed as a **distance** through a **hyperspace**. Hyperspace in this context means a coordinate system with one axis (dimension) for each variable in the data.

To quantify distances between taxa, we might try adding up the differences between each pair. We should also square each individual difference, so $1 - 0$ counts the same as $0 - 1$. Then, we add up the squared differences and take the square root to get back to the original scale. For example, the distance between maple and pine in the table above would be:

$$D(\text{maple}, \text{pine}) = \sqrt{(1 - 1)^2 + (0 - 0)^2 + (1 - 0)^2 + (0 - 0)^2 + (0 - 0)^2 + (0 - 0)^2 + (0 - 0)^2} = 1$$

The difference between octopus and penguin would be:

$$D(\text{octopus}, \text{penguin}) = \sqrt{(1 - 1)^2 + (1 - 1)^2 + (0 - 0)^2 + (0 - 1)^2 + (1 - 0)^2 + (0 - 1)^2 + (0 - 1)^2} = 1$$

And the difference between mushrooms and elephants is:

$$D(\text{mushroom}, \text{elephant}) = \sqrt{(1 - 1)^2 + (1 - 1)^2 + (0 - 0)^2 + (0 - 1)^2 + (1 - 0)^2 + (0 - 1)^2 + (0 - 1)^2} \approx 1.41$$

So, for this set of characteristics, octopuses and penguins are twice as different from each other as are maple trees and pine trees, while mushrooms and elephants

are not quite as different from each other as octopuses and penguins. Interestingly, mushrooms and elephants are more different from each other than are maples and pines, even though maples and pines are in the same kingdom while mushrooms and plants are not. Does that make sense biologically?

These comparisons were chosen to make a couple of points about quantifying differences. First, the choice of characteristics (or measurements, or metrics, etc.) has a large influence over the calculation. Look over the list again and ask yourself if those characteristics are applicable to all of the organisms. Four of the eight traits are specific to just the animals! Second, the choice of how to combine the differences makes a big difference. There are other methods we could have chosen (see below) that might have measured dissimilarity differently.

8.2.1 Euclidean distance

The most basic distance metric is the **Euclidean distance**. This is a generalization of the Pythagorean theorem to an arbitrary number of dimensions. The calculations above were presented in a cumbersome way to show how they work, but the Euclidean distance metric is usually written in a more compact format:

$$D(x_i, x_h) = \sqrt{\sum_{j=1}^p (x_{i,j} - x_{h,j})^2}$$

In this expression the difference between observation i and observation h ($x[i]$ and $x[h]$) is the square root of the sum of squared differences between x_i and x_h in terms of each variable j , up to the number of variables p . When $p = 2$, this formula reduces to the Pythagorean theorem:

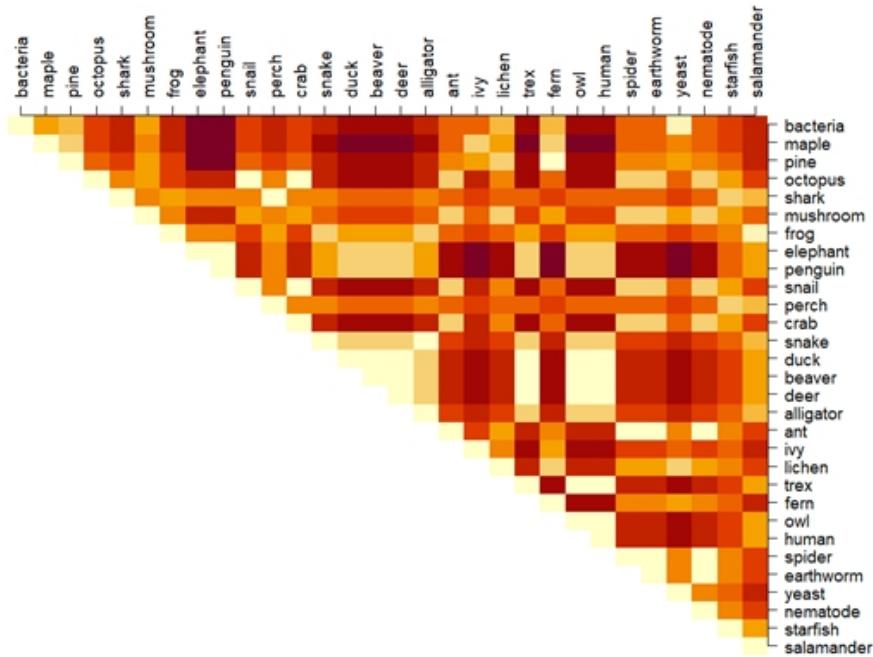
$$D(x_i, x_h) = \sqrt{\sum_{j=1}^2 (x_{i,j} - x_{h,j})^2} = \sqrt{(x_{i,1} - x_{h,1})^2 + (x_{i,2} - x_{h,2})^2}$$

Again, the Euclidean distance is just the Pythagorean theorem generalized to p dimensions rather than the usual 2. One way to think of the Pythagorean theorem is as the distance between points on the 2-d plane, with the x and y coordinates as the side lengths of the triangle.

Here and for the rest of this course, “dimension” means the same as “variable”. When thinking about multivariate statistics, it can be useful to think of your data as defining a **hyperspace** or **hypervolume**, with one dimension for each variable in your data. Thus a dataset with 2 variables defines a space with 2 dimensions (i.e., a plane); a dataset with 3 dimensions defines a space with 3 dimensions (i.e., a volume), and so on. When discussing multivariate differences between sample units, we usually refer to the differences as distances through

these hyperspaces. Each distance metric calculates that distance a slightly different way.

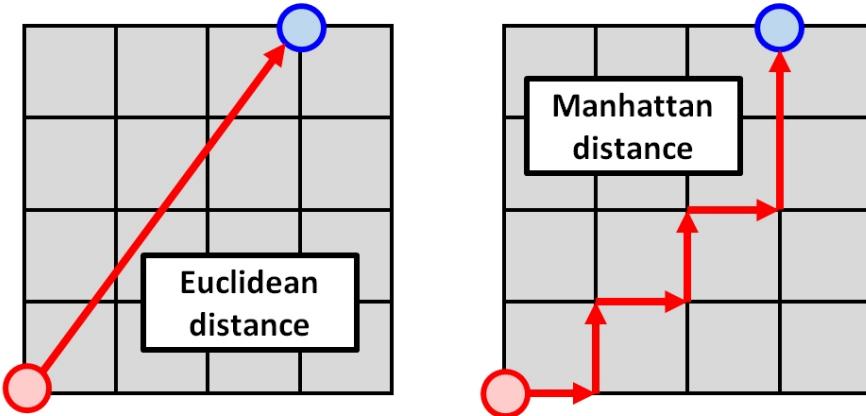
Back to our organism classification example, the figure below shows Euclidean distances for 30 organisms across 12 traits. The distances are distances between each of the 435 unique pairs of taxa in a 12-dimensional hyperspace. Darker colors indicate greater distance—i.e., greater dissimilarity or smaller similarity.



If you examine the chart above you will find that some of the distances are silly: for example, octopuses are presented as more different from owls than they are from bacteria. This suggests that the Euclidean distance metric was maybe not the right one.

8.2.2 Bray-Curtis and other distance metrics

Another distance metric is called the **Manhattan** or **city block** distance. It gets its name from the way that distances are added up by assuming that samples can only be connected by paths that move along one axis (i.e., dimension) at a time. This is analogous to how people in cities can walk along the sidewalks of the city street grid, but cannot cut through blocks. The figure below illustrates this.



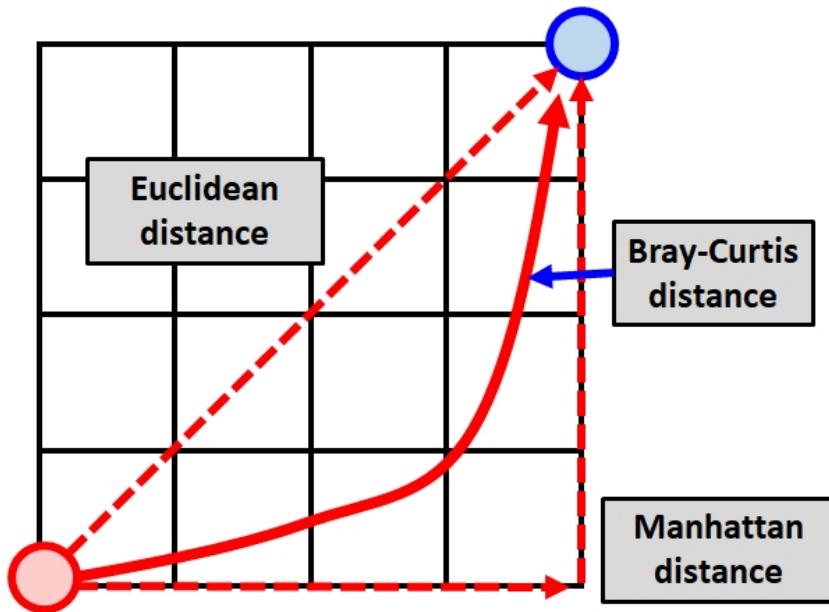
The Manhattan distance is longer than the Euclidean distance, but it can sometimes be a better representation of the differences between sample units in terms of many variables. The Manhattan distance is calculated as:

$$D(x_i, x_h) = \sum_{j=1}^p |x_{i,j} - x_{h,j}|$$

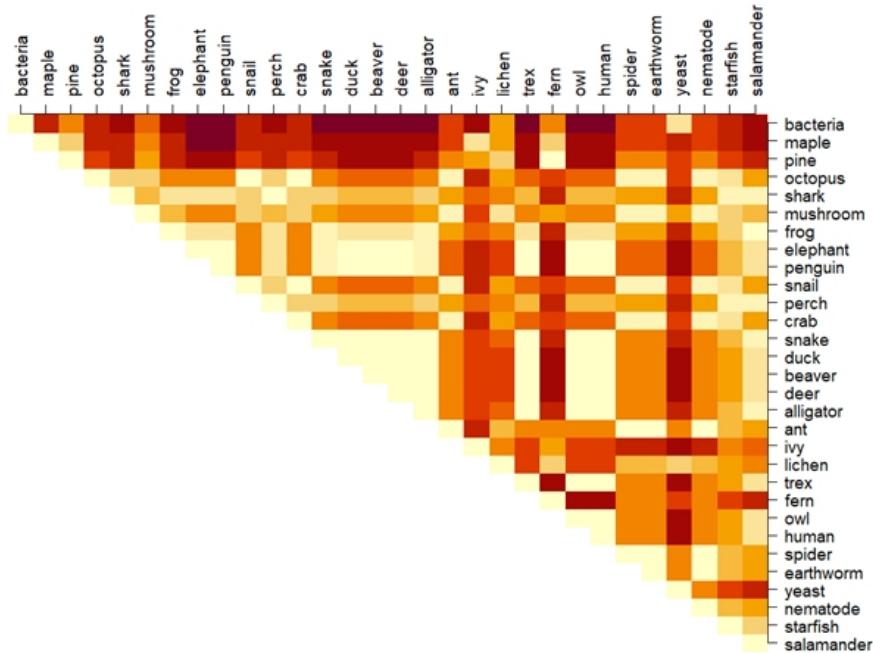
The Manhattan distance is not often used by itself, but a relativized version of it is extremely common in biology: the **Bray-Curtis distance**, also known as the **Sørenson distance**. This distance measure is called the Sørenson index when used with binary data (such as 1/0 for presence/absence). The name Bray-Curtis is used when the same formula is applied to continuous data (often proportions or percentages). The Bray-Curtis distance is calculated as:

$$D_{BC}(x_i, x_h) = \frac{\sum_{j=1}^p |x_{i,j} - x_{h,j}|}{\sum_{i=1}^p x_{i,j} + \sum_{i=1}^p x_{h,j}} = 1 - \frac{2 \sum_{j=1}^p \text{MIN}(x_{i,j}, x_{h,j})}{\sum_{i=1}^p x_{i,j} + \sum_{i=1}^p x_{h,j}}$$

In the second expression, $\text{MIN}()$ is a function that returns the smaller of two values. Notice that the Bray-Curtis distance is essentially the Manhattan distance divided by the shared total values in both samples. This ratio can be thought of as the shared values divided by the total of values. The division makes this value “relativized” (i.e., relative to something else, in the same way that percentages are relative to 100). The figure below shows the approximate relationship between the Euclidean distance, Manhattan distance.



If we recalculate the distance matrix between the taxa in our example, the distances now look like this:



The new distances are an improvement but there is still a lot of room to get better. For example, salamanders are more similar to pines than they are to maples. I'll leave the interpretation of that finding up to you (hint: there isn't one... this is a silly example).

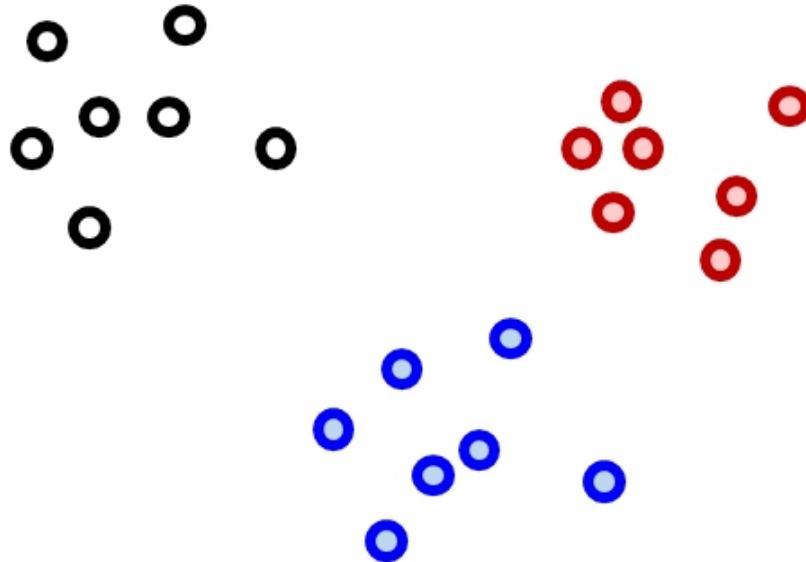
There are lots of other distance metrics, but a full exploration is beyond the scope of this course. The table below, adapted from McCune et al. (2002), gives some characteristics of some common measures. The range of input data x and distance measures d is provided. For most biological situations where data are nonnormal and relationships are nonlinear, the Bray-Curtis distance is likely to be the most appropriate. When many variables contain lots of 0s, a modified version can be used that adds a "dummy species" present in every sample to stabilize the distances (Clarke et al. 2006). The Euclidean metric has strong requirements of multivariate normality and collinearity among variables; the Bray-Curtis metric does not. The other metrics aren't as commonly used as the Euclidean and Bray-Curtis, but are included here for reference.

Metric	Domain of x	Range of d	Comments
Bray-Curtis (Sørenson)	$x \leq 0$	$0 \leq d \leq 1$	Preferred in many biological situations
Relative Sørenson (Kulczynski)	$x \leq 0$	$0 \leq d \leq 1$	Relativized by sample totals
Jaccard	$x \leq 0$	$0 \leq d \leq 1$	Related to Manhattan distance
Euclidean (Pythagorean)	$x \in \mathbb{R}$	$0 \leq d$	Often requires multivariate normality
Relative Euclidean (chord distance)	$x \in \mathbb{R}$	$0 \leq d \leq \sqrt{2}$ or $0 \leq d \leq 2$	Euclidean distance on a hypersphere
Chi-square	$x \geq 0$	$0 \leq d$	Euclidean but weighted by sample and variable totals
Squared Euclidean	$x \in \mathbb{R}$	$0 \leq d$	Square of Euclidean distance
Mahalanobis	$x \in \mathbb{R}$	$0 \leq d$	Distance between groups weighted by intragroup variance

Remember that distances express differences between samples with respect to several variables. This can be visualized as a distance through a hyperspace with as many dimensions as you have variables. As we'll see in the next section, we can use distances to explore similarities and differences between samples.

8.3 Clustering

Now that we have a way of quantifying multivariate differences between samples as distance, we can use those measures to explore patterns in data. One application of distance metrics is to use them to organize data into clusters or groups consisting of samples that are more similar to each other than they are to other samples. In other words, the **within-group** distances should be smaller than the **between-group** distances. The figure below illustrates this concept with an imaginary example. The clusters shown cleanly separate black, red, and blue points. For example, the average distance between the black points is smaller than the average distance between the black points and the red or blue points. Likewise, the average distance between the red points is smaller than the average distance between the red points and the black points or the blue points.



There are many methods for clustering observations. Many methods are **hierarchical**, with groups being nested within other groups. Hierarchical methods

are often the most useful for biologists, because so many of the phenomena we study have some hierarchical nature to them. Other methods are nonhierarchical. Both approaches work by minimizing within-group distances relative to between group distances. The difference is that hierarchical methods have the constraint that larger groups are formed from subgroups.

Clustering methods also differ in their manner of group construction. Many methods are **agglomerative**, which means that groups are formed by combining samples. Other methods are **divisive**, which means that all samples start in a single group which is then broken up. Examples of each class of clustering method are shown below.

Group nature	Group formation	Common methods
Nonhierarchical	Agglomerative	??
Nonhierarchical	Divisive	<i>K</i> -means clustering
Hierarchical	Agglomerative	Ward clustering (Ward 1963)
Hierarchical	Divisive	Association analysis (Ludwig et al. 1988, Legendre and Legendre 2012); TWINSPAN (Hill 1979, Roleček et al. 2009)

8.3.1 *K*-means clustering

K-means clustering is a method for dividing a dataset with n observations, with p variables, into k categories such that each observation belongs to the cluster with the most similar mean. The “mean” for each category (i.e., group) is really a position in a p -dimensional hyperspace called a **centroid**. In this context, centroids are calculated as the point at the central coordinate of each dimension. The mean is most often used, but the median can be a useful alternative.

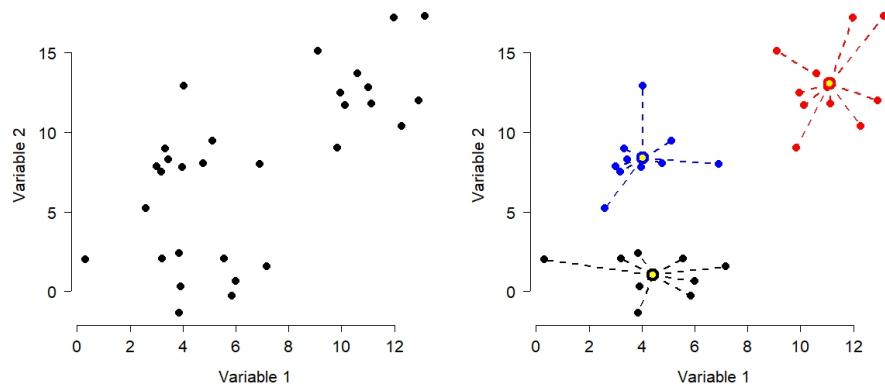
There are several algorithms for k -means clustering. One of the most common works like this:

1. Select the desired number of clusters k .
2. Make an initial guess of the mean of each cluster. I.e., Start with k random positions in the p -dimensional hyperspace. These positions are known as centroids.
3. Calculate the Euclidean distance between each sample and each centroid.
4. Assign each sample to the category corresponding to the nearest centroid.
5. Update the centroids for each cluster by calculating the new centroid for the samples assigned to each cluster.
6. Repeat steps 3, 4, and 5 until cluster assignments no longer change.

This algorithm is not guaranteed to find the best possible solution, but it usually

works. Distance metrics other than the Euclidean can be used, but Euclidean is most common. Transforming, scaling, and/or standardizing variables so that they are all on the same scale can improve the results.

The figure below shows the effects of k -means clustering on a pretend dataset with 2 variables with $k = 3$. The group assignments were determined because groups with those centroids had the smallest possible set of Euclidean distances between the points and the group centroids (right panel). Group centroids are added to the plot in yellow.



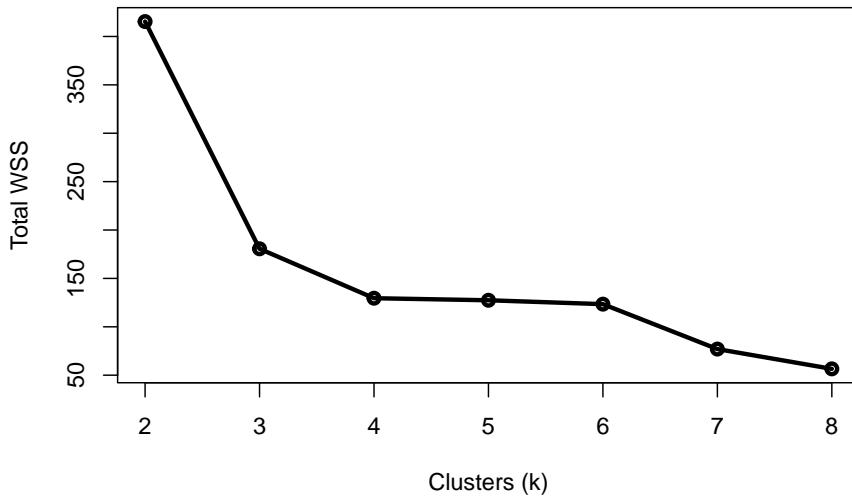
How many clusters are appropriate? That's a good question to ask, and sometimes a tricky question to answer. One common way to assess the optimal k is to try several k and calculate the within-cluster sum of squared errors (WSS) for all points. The optimal value of k is the value at which a plot of WSS vs. k starts to bend. The example below illustrates this method for the data above.

```
# generate a random dataset with 3 clusters
set.seed(1234)
x <- c(rnorm(20, 5, 2), rnorm(10, 12, 2))
y <- c(rnorm(10, 3, 2), rnorm(10, 10, 2), rnorm(10, 14, 2))
dat <- data.frame(x,y)

# perform k-means clustering with k from 2 to 8
kvec <- 2:8
nk <- length(kvec)
k.list <- vector("list", nk)
for(i in 1:nk){
  k.list[[i]] <- kmeans(dat, kvec[i])
}

# gather WSS for each k
wss <- sapply(k.list, function(x){x$tot.withinss})
```

```
# plot WSS vs. k
par(mfrow=c(1,1))
plot(kvec, wss, type="o", lwd=3, pch=1.5,
     xlab="Clusters (k)",
     ylab="Total WSS")
```



Notice that total WSS drops off much more slowly above 3 clusters. That is, going from 3 to 4 clusters reduces WSS much less than going from 2 to 3 clusters. The result above could reasonably justify 3 or 4 clusters. The figure below shows how the data are divided into 3 and 4 clusters.

```
# assemble values needed for plots
## get clusters
dat$k3 <- k.list[[which(kvec == 3)]]$cluster
dat$k4 <- k.list[[which(kvec == 4)]]$cluster

## coordinates of group centroids
km3 <- k.list[[which(kvec == 3)]]
km4 <- k.list[[which(kvec == 4)]]

dat$cenx3 <- km3$centers[dat$k3,1]
dat$ceny3 <- km3$centers[dat$k3,2]
dat$cenx4 <- km4$centers[dat$k4,1]
dat$ceny4 <- km4$centers[dat$k4,2]
```

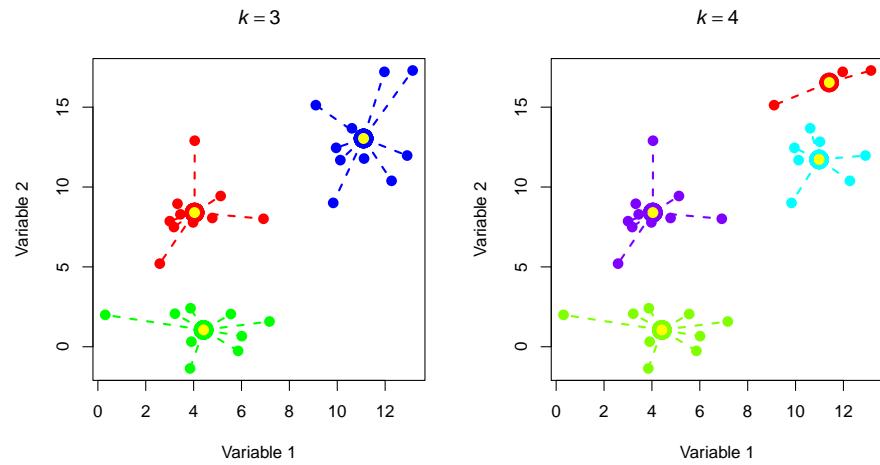
```

## colors for each group
dat$col3 <- rainbow(3)[dat$k3]
dat$col4 <- rainbow(4)[dat$k4]

# make plot
par(mfrow=c(1,2))
plot(x,y, pch=16, col=dat$col3, cex=1.4,
      xlab="Variable 1", ylab="Variable 2",
      main=expression(italic(k)==3))
segments(dat$cenx3, dat$ceny3, dat$x, dat$y,
      col=dat$col3, lwd=2, lty=2)
points(dat$cenx3, dat$ceny3, pch=21, cex=2,
      bg="yellow", col=dat$col3, lwd=4)

plot(x,y, pch=16, col=dat$col4, cex=1.4,
      xlab="Variable 1", ylab="Variable 2",
      main=expression(italic(k)==4))
segments(dat$cenx4, dat$ceny4, dat$x, dat$y,
      col=dat$col4, lwd=2, lty=2)
points(dat$cenx4, dat$ceny4, pch=21, cex=2,
      bg="yellow", col=dat$col4, lwd=4)

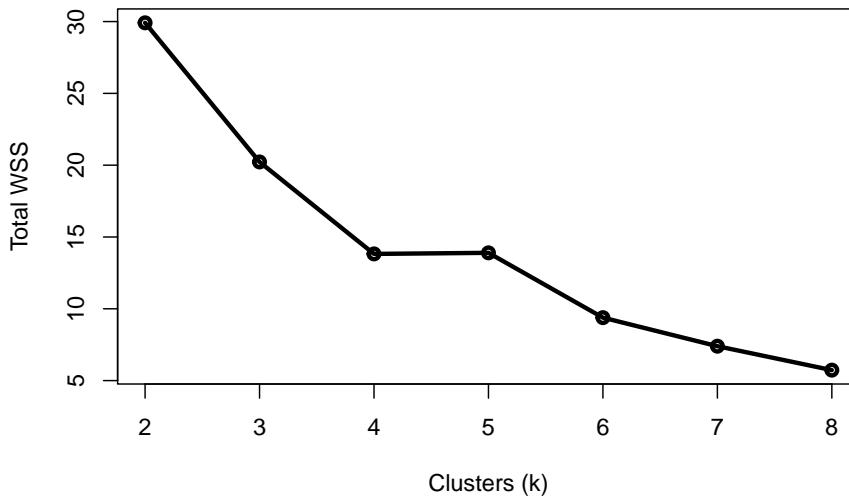
```



Most of the group assignments stayed the same, but the algorithm split the blue group when k increased from 3 to 4. Accepting the 3 cluster or 4 cluster solution is as much of a biological decision as it is a statistical one. When you get to this point, you need to think about whether the groups make sense or not. Notice that k -means clustering does not necessarily result in groups of equal sizes. This may or may not be a problem depending on the structure of your dataset and the question you are trying to answer.

Below is another example of k -means clustering applied to the taxonomy dataset from above. Examine the figure and ask yourself whether the groupings make sense from a biological perspective. Note that the code block below requires that you have the data file `tax_example_2021-10-27.csv` in your R home directory. Download it here. Note that this is a completely made-up dataset, so you should not use it for anything important.

```
set.seed(123)
in.name <- "tax_example_2021-10-27.csv"
dat <- read.csv(in.name, header=TRUE)
rownames(dat) <- dat$organism
dat$organism <- NULL
kvec <- 2:8
nk <- length(kvec)
k.list <- vector("list", nk)
for(i in 1:nk){
  k.list[[i]] <- kmeans(dat, kvec[i])
}
wss <- sapply(k.list, function(x){x$tot.withinss})
par(mfrow=c(1,1))
plot(kvec, wss, type="o", lwd=3, pch=1.5,
  xlab="Clusters (k)",
  ylab="Total WSS")
```



Four clusters appears to be the “elbow” of the curve, but this varied from run to run (try changing the random number seed and seeing if you get different

results). Let's use $k = 4$.

```
# assemble values needed for plots
## get clusters
dat$k4 <- k.list[[which(kvec == 4)]]$cluster

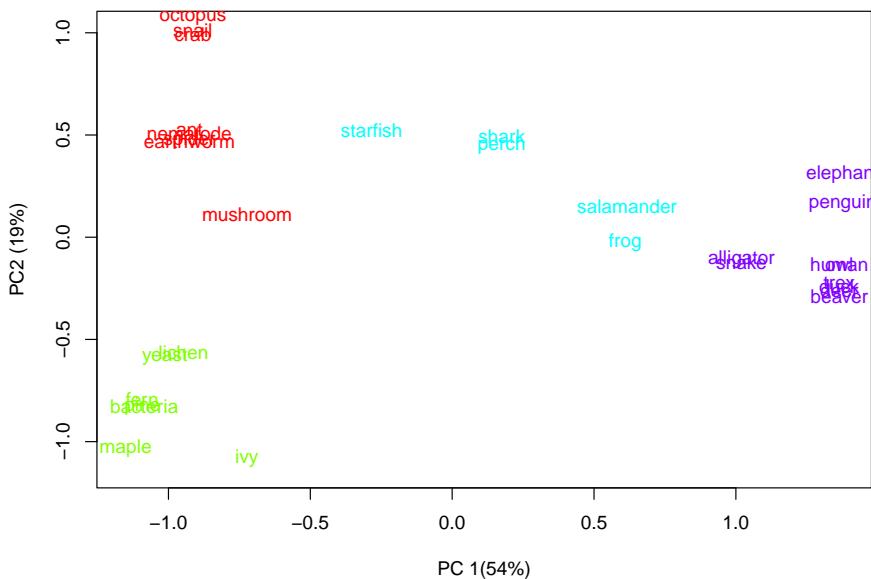
## coordinates of group centroids
km4 <- k.list[[which(kvec == 4)]] 

dat$cenx4 <- km4$centers[dat$k4,1]
dat$ceny4 <- km4$centers[dat$k4,2]

## colors for each group
dat$col4 <- rainbow(4)[dat$k4]

# use PCA to define a reduced dimensional space
# (we'll cover PCA in detail later)
pr1 <- prcomp(dat[,1:12])
dat$x <- pr1$x[,1]
dat$y <- pr1$x[,2]

# make plot
par(mfrow=c(1,1))
plot(dat$x, dat$y, type="n",
      xlab="PC 1(54%)", ylab="PC2 (19%)")
text(dat$x, jitter(dat$y, amount=0.1),
     rownames(dat), col=dat$col4)
```



It's a little hard to see some of the names, even with the jittering of the Y coordinates. How did the algorithm do? The groups appear to mostly make sense, but there are some oddities. For example:

- Elephants and penguins are more similar than either is to other mammals or birds, respectively (although they are in the same group).
- Salamanders and frogs were grouped with fish and echinoderms, despite being tetrapods.
- Mushrooms are clustered with the protostome invertebrates, despite being more closely related to yeast (which are grouped with the microbes and plants).

So, the results of the k -means clustering are mostly okay, but there is definitely plenty of room for improvement.

8.3.2 Hierarchical agglomerative clustering

One key disadvantage of k -means clustering is that it is not hierarchical: all groups are assumed to be homogenous, with no within-group structure or subgroups. In biological systems this is rarely the case. Many biological phenomena can be understood as hierarchical: for example, phylogeny and taxonomy. Hierarchical clustering can help discover, or at take advantage of, these relationships within your data.

There are many methods of hierarchical clustering, just as there were many

methods of k -means clustering. One of the most common is **Ward's method** (Ward 1963), which has many variations. The basic procedure is:

1. Start with every sample separate (i.e., in its own cluster).
2. Find a pair of clusters to combine that leads to the smallest increase in total within-cluster variance
3. Repeat step 2 until a stopping point is reached (varies by method).

The within-cluster variance is an example of an **objective function**, which measures how effectively a statistical model represents the data. Different versions of Ward's method use different objective functions. The most common is the Euclidean distance between cluster centroids. Squared Euclidean distance and other metrics are also seen in the literature.

The example below applies hierarchical clustering to the taxonomy dataset seen above. The base R function for hierarchical clustering is `hclust()`. The data are provided as a distance matrix rather than as raw values. We'll use the `vegdist()` function from package `vegan` instead of the base R `dist()` function because it offers more distance metrics. Note that the code below assumes that you have the data file `tax_example_2021-10-27.csv` (here) in your R home directory.

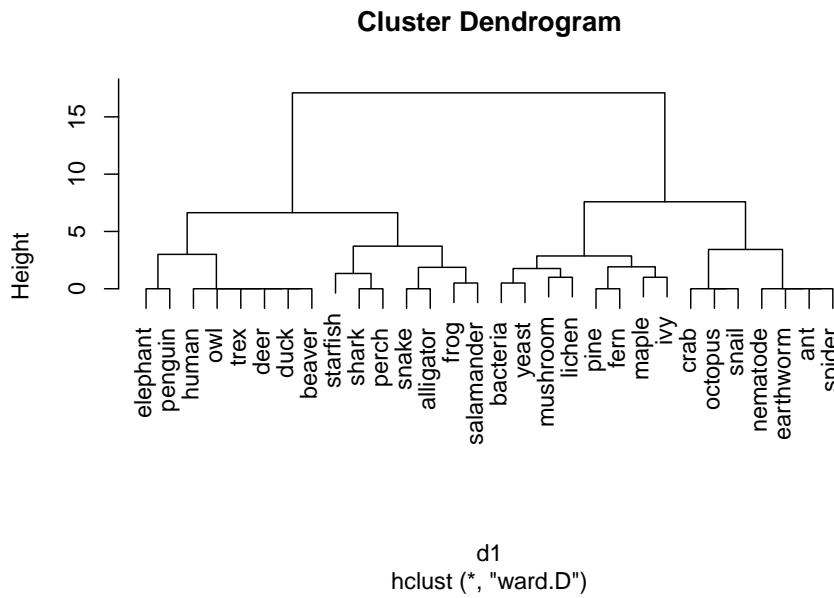
```
set.seed(123)
in.name <- "tax_example_2021-10-27.csv"
dat <- read.csv(in.name, header=TRUE)
rownames(dat) <- dat$organism
dat$organism <- NULL

library(vegan)

## Loading required package: permute

## This is vegan 2.5-7
d1 <- vegdist(dat, method="euclidean")
d2 <- vegdist(dat) # default bray-curtis

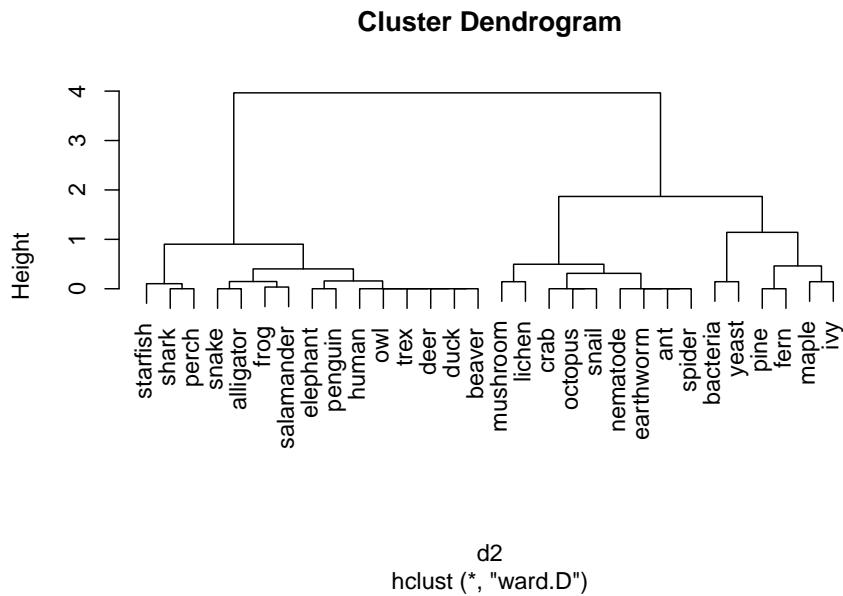
h1 <- hclust(d1, method="ward.D")
plot(h1)
```



The results of hierarchical clustering are usually presented as a **dendrogram**. The word root *dendro-* means “tree”, which is a good way to think of a dendrogram. The branches show the relationship between the clusters. In the result above, pines and ferns form a cluster, as do maple and ivy. The “pine-fern” and “maple-ivy” clusters together form a bigger cluster.

The clustering based on Bray-Curtis distances is slightly different:

```
h2 <- hclust(d2, method="ward.D")
plot(h2)
```



The results using the Euclidean and Bray-Curtis distance metrics are similar, but both dendograms make some “interesting” choices. For example, one would expect the most basal (toward the root of the tree) division between animals and non-animals, or between prokaryotes and eukaryotes. Is that the case?

Not at all. The method clustered the taxa in a way that doesn’t match their real-life phylogenetic relationships. This is partly because of the characteristics that were used (a very vertebrate animal-centric set!). This also illustrates the difference between taxonomy, which seeks to combine organisms by shared characteristics; and phylogeny, which results from how lineages divide over time.

8.4 Analyzing dissimilarity

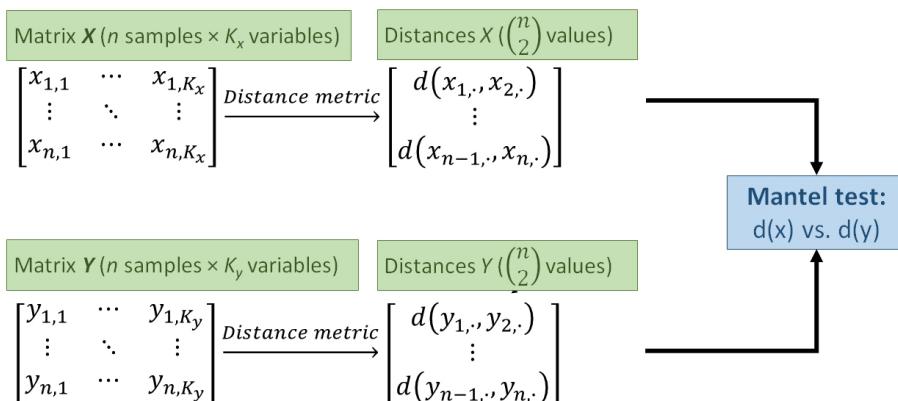
A metric like the Bray-Curtis or Euclidean metric can quantify how different samples are from each other in terms of many variables simultaneously. The natural next question to ask is, “So what?”. Just as we can compare means between groups, or measure whether two variables are correlated, we can compare distances between groups or test whether two sets of distances are correlated. This document demonstrates some ways to use distance metrics to generalize common univariate and bivariate tests to patterns in higher dimensions.

8.4.1 Mantel tests: distance vs. distance

One way to use distance metrics in a biological investigation is to calculate different distances between samples using different sets of variables, and then see

if those distances are related. For example, an ecologist might calculate distances between sites using plant cover data, and a separate set of distances using soil chemistry data. She would then have one distance metric describing differences between the sites in terms of their plant cover, and a second distance metric describing differences between the sites in terms of their soil characteristics. A natural question to ask then is, “is distance in terms of plant cover related to distance in terms of soil chemistry?”. In other words, “do sites that are more similar in terms of their soil chemistry tend to be more similar in terms of their plant cover?”

Answering these questions is tantamount to calculating a **correlation coefficient between two vectors of distance metrics**. This procedure is called a **Mantel test** (Mantel 1967). The Mantel correlation coefficient r answers the question, “How is variation in one set of variables related to variation in another set of variables?”.



The example below uses the `iris` dataset to test whether variation in petal morphology is related to variation in sepal morphology. Note that while the Euclidean distance metric is used here because data are known to be multivariate normal, the Bray-Curtis or other metrics may be more appropriate in other situations.

```

# get sepal and petal variables from iris
x1 <- iris[,grep("Sepal", names(iris))]
x2 <- iris[,grep("Petal", names(iris))]

# calculate distance metrics
library(vegan)
d1 <- vegdist(x1, method="euclidean")
d2 <- vegdist(x2, method="euclidean")
d3 <- vegdist(x1) # vegdist() default metric = Bray-Curtis
d4 <- vegdist(x2)

```

```
# Mantel test on Euclidean distances
mantel(d1, d2)

##
## Mantel statistic based on Pearson's product-moment correlation
##
## Call:
## mantel(xdis = d1, ydis = d2)
##
## Mantel statistic r: 0.7326
## Significance: 0.001
##
## Upper quantiles of permutations (null model):
## 90% 95% 97.5% 99%
## 0.0242 0.0330 0.0409 0.0490
## Permutation: free
## Number of permutations: 999
```

The Mantel statistic r is exactly the same as the Pearson product moment correlation r between the two sets of distance metrics (verify this with the command `cor(d1, d2)`). The significance of the Mantel r is usually calculated by a permutation test instead of the more conventional methods . A **permutation test** for significance works by rearranging, or permuting, the observations many times and comparing a statistic (r) to the distribution of the statistic across the permutations. The P -value is calculated as the proportion of permutations that had a statistic with a magnitude at least as great as the original statistic. In the example above, $P = 0.001$ because at most 1 out of 999 permutations had an $r \geq 0.7326$.

The function below will replicate the permutation test done for the Mantel test above. This function is provided for didactic purposes only. You should use the methods built into R and `vegan` instead. But, studying the code below can help you understand how a permutation test works.

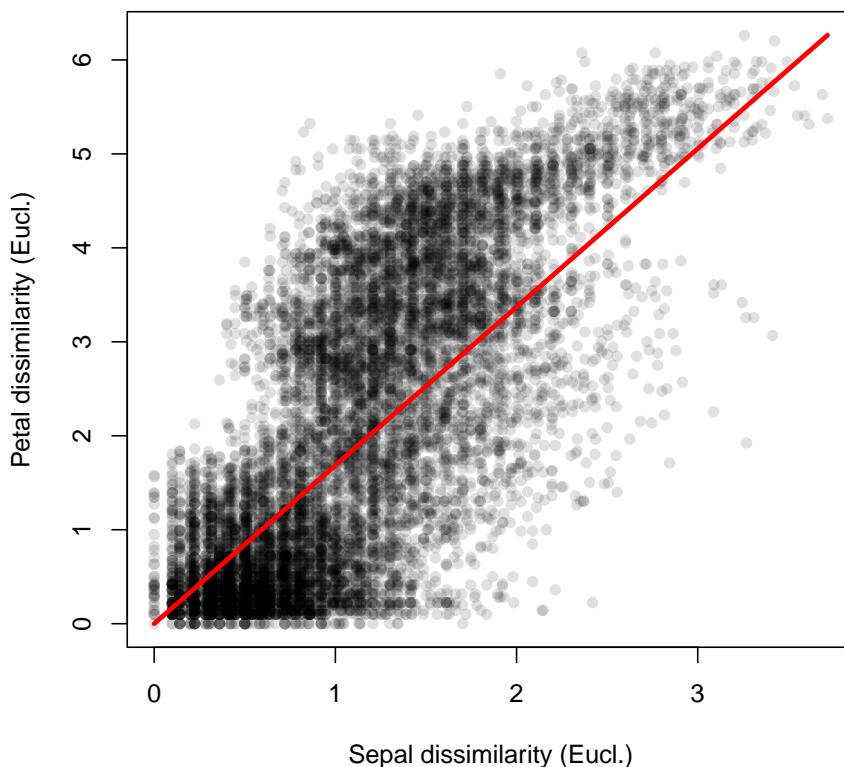
```
simple.ptest <- function(dm1, dm2, n=999){
  r <- cor(dm1, dm2)
  dn <- length(dm1)
  rvec <- numeric(dn)
  for(i in 1:dn){
    i1 <- dm1[sample(1:dn, replace=FALSE)]
    i2 <- dm2[sample(1:dn, replace=FALSE)]
    rvec[i] <- cor(i1, i2)
  }
  res <- length(which(rvec >= r))/dn
  return(res)
}
```

```
simple.ptest(d1, d2, 999)
```

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

Mantel tests are usually reported by their numerical outputs, but you can plot the results if the test is central to your presentation. The simplest way is to plot the distance metrics on a scatterplot. The example below uses partial transparency to better show the overlapping points. A diagonal red line shows where the two dissimilarities are equal.

```
plot(d1, d2, pch=16, col="#00000020",
      xlab="Sepal dissimilarity (Eucl.)",
      ylab="Petal dissimilarity (Eucl.)")
segments(0, 0, max(d1), max(d2),
      lwd=3, col="red")
```



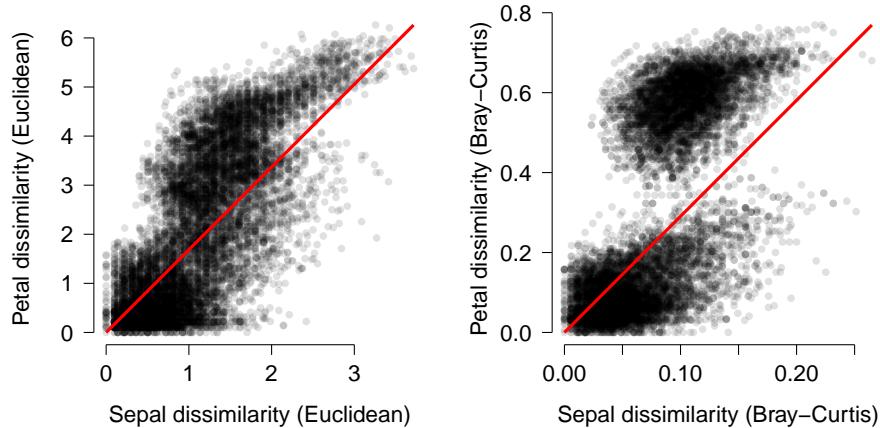
The figure shows what the Mantel r statistic already indicated: that flowers

with similar sepal morphology tend to have similar petal morphology.

Just for fun, the figure below visualizes the Mantel correlation based on the Euclidean (left) and Bray-Curtis (right) distances.

```
d3 <- vegdist(x1) # default Bray-Curtis
d4 <- vegdist(x2)
mantel(d3, d4)
##
## Mantel statistic based on Pearson's product-moment correlation
##
## Call:
## mantel(xdis = d3, ydis = d4)
##
## Mantel statistic r: 0.6552
##      Significance: 0.001
##
## Upper quantiles of permutations (null model):
##    90%    95%   97.5%    99%
## 0.0260 0.0365 0.0459 0.0578
## Permutation: free
## Number of permutations: 999
par(mfrow=c(1,2), mar=c(5.1, 5.1, 1.1, 1.1),
    bty="n", lend=1, las=1,
    cex.axis=1.3, cex.lab=1.3)
plot(d1, d2, pch=16, col="#00000020",
      xlab="Sepal dissimilarity (Euclidean)",
      ylab="Petal dissimilarity (Euclidean)")
segments(0, 0, max(d1), max(d2),
         lwd=3, col="red")

plot(d3, d4, pch=16, col="#00000020",
      xlab="Sepal dissimilarity (Bray-Curtis)",
      ylab="Petal dissimilarity (Bray-Curtis)")
segments(0, 0, max(d3), max(d4),
         lwd=3, col="red")
```



Interestingly, the Bray-Curtis distances seemed to fall into two clusters in terms of the petal dissimilarity. That probably has something to do with the difference between *I. setosa* and the other species, but I haven't investigated it.

The Mantel test can also be performed with correlation coefficients other than the linear correlation coefficient. This makes sense if the correlation appears nonlinear. For example, if one dissimilarity metric increases consistently with the other dissimilarity metric, but not in a straight line. The most common alternative is the Spearman's rank correlation coefficient ρ ("rho").

```
# euclidean distance metric
mantel(d1, d2, method="spearman")

##
## Mantel statistic based on Spearman's rank correlation rho
##
## Call:
## mantel(xdis = d1, ydis = d2, method = "spearman")
##
## Mantel statistic r: 0.7258
##      Significance: 0.001
##
## Upper quantiles of permutations (null model):
##      90%    95%   97.5%    99%
## 0.0197 0.0253 0.0298 0.0362
## Permutation: free
## Number of permutations: 999

# bray-curtis metric
mantel(d3, d4, method="spearman")

##
## Mantel statistic based on Spearman's rank correlation rho
```

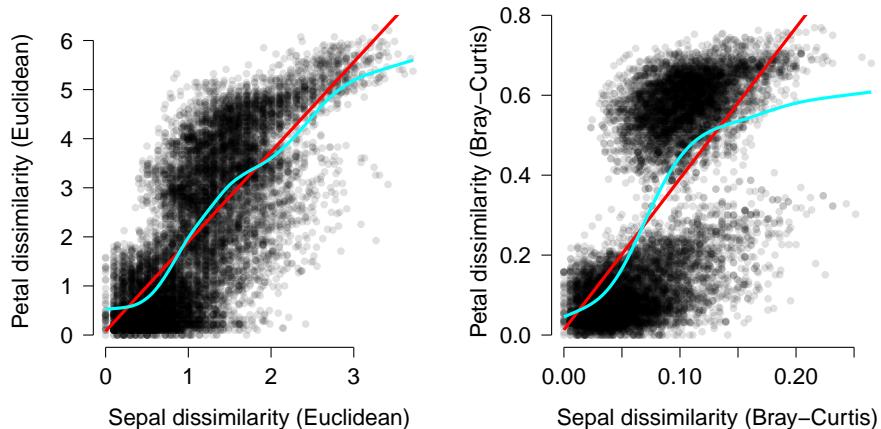
```

## 
## Call:
## mantel(xdis = d3, ydis = d4, method = "spearman")
## 
## Mantel statistic r:  0.69
##      Significance: 0.001
## 
## Upper quantiles of permutations (null model):
##    90%   95%  97.5%   99%
## 0.0247 0.0331 0.0392 0.0540
## Permutation: free
## Number of permutations: 999

```

Changing from a linear to nonlinear correlation coefficient slightly decreased the Mantel correlation for the Euclidean distances (0.7326 to 0.7258), and slightly increased the correlation for the Bray-Curtis distances (0.6552 to 0.69). Examine the figure above and see if you can work out why this is.

The figure below shows the two sets of distances with two lines: a straight line to represent the linear correlation r and a spline curve to represent the nonlinear correlation ρ .



8.4.2 Comparing dissimilarity between groups

Just as the Mantel test generalizes the idea of correlation from “one variable vs. one variable” to “many variables vs. many variables”, other techniques generalize the idea of group differences from “difference in means between groups” to “differences in centroids between groups”. This section demonstrates three common tests for multivariate differences between groups.

8.4.2.1 Analysis of similarity (ANOSIM): distances between groups

The **analysis of similarities (ANOSIM)** is a nonparametric, ANOVA-like test that tests whether similarity between groups is greater than or equal to the similarity within groups (Clarke 1993). This is analogous to how ANOVA tests whether the difference in means (i.e., variance) between groups is greater than or equal to the variance within groups. What makes the test nonparametric is that it operates on the rank-transformed distance matrix rather than on the actual distance metrics.

The **vegan** function `anosim()` is used for ANOSIM. The function takes the original data matrix, a grouping variable, and can use any distance metric available in `vegan::vegdist()`. The example below uses the Euclidean metric because the data are known to be multivariate normal; you may need to use the Bray-Curtis or another metric with your own data.

```
a1 <- anosim(iris[,1:4],
  grouping=iris$Species,
  distance="euclidean")
a1

##
## Call:
## anosim(x = iris[, 1:4], grouping = iris$Species, distance = "euclidean")
## Dissimilarity: euclidean
##
## ANOSIM statistic R: 0.8794
##      Significance: 0.001
##
## Permutation: free
## Number of permutations: 999
```

Like the Mantel test, ANOSIM estimates a P -value by permutation. In this case, the permutation is of group membership. The underlying idea is that if there are no differences between groups (i.e., if the null hypothesis were true), then group membership is irrelevant and changing group memberships should not affect any test statistics. The ANOSIM statistic R expresses the relationship between the difference in mean ranks between groups ($mr_{between}$) and the within groups (mr_{within}).

$$R = \frac{mr_{between} - mr_{within}}{\frac{1}{4}n(n-1)}$$

The denominator scales the difference in mean ranks from -1 to 1. $R = 0$ means that grouping is unrelated to differences in mean ranks; greater R values indicate that differences between groups are greater than differences within groups; smaller R values indicate the reverse. The R value in our test above

suggests that about 87.9% of the variation in the rank order of the distance matrix can be attributed to differences between species of *Iris*.

8.4.2.2 MANOVA and PERMANOVA: distances between centroids

Multivariate analysis of variance (MANOVA) is an extension of ANOVA with multiple response variables. The underlying theory and matrix algebra is very similar to ANOVA, and so many of the same assumptions apply (just in many dimensions instead of one). For that reason, MANOVA is often not appropriate for real biological data without careful experimental design and exploratory data analysis to confirm that the test's assumptions are met.

```

Y <- as.matrix(iris[,1:4])
mod1 <- manova(Y~iris$Species)

# (M)ANOVA table
summary(mod1, test="Pillai")

##           Df Pillai approx F num Df den Df    Pr(>F)
## iris$Species   2 1.1919   53.466      8     290 < 2.2e-16 ***
## Residuals     147
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
# coefficients for each response and level of predictor
coef(mod1)

##                               Sepal.Length Sepal.Width Petal.Length Petal.Width
## (Intercept)                  5.006       3.428      1.462      0.246
## iris$Speciesversicolor      0.930      -0.658      2.798      1.080
## iris$Speciesvirginica       1.582      -0.454      4.090      1.780

```

Permutational analysis of variance (PERMANOVA) is a nonparametric version of MANOVA (Anderson 2001). It is nonparametric because it uses permutation of group membership to determine statistical significance instead of calculating an F statistic based on probability theory. The most common R function for PERMANOVA is `adonis()` from the `vegan` package.

```

mod2 <- adonis(iris[,1:4]~iris$Species, method="euclidean")

# (perm)anova table
mod2

##
## Call:
## adonis(formula = iris[, 1:4] ~ iris$Species, method = "euclidean")
##
## Permutation: free
## Number of permutations: 999

```

```

## 
## Terms added sequentially (first to last)
##
##          Df SumsOfSqs MeanSqs F.Model      R2 Pr(>F)
## iris$Species  2    592.07 296.037 487.33 0.86894 0.001 ***
## Residuals    147     89.30   0.607           0.13106
## Total        149    681.37           1.00000
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
# coefficients for each response and level of predictor
mod2$coefficients

##          Sepal.Length Sepal.Width Petal.Length Petal.Width
## (Intercept)  5.84333333  3.0573333     3.758   1.1993333
## iris$Species1 -0.83733333  0.3706667    -2.296  -0.9533333
## iris$Species2  0.09266667 -0.2873333      0.502   0.1266667

```

8.4.2.3 MRPP: differences in location

Multiple response permutation procedures (MRPP) is a permutational test for location that is very similar to ANOSIM (McCune et al. 2002). The practical difference between MRPP and ANOSIM is that MRPP is typically used on coordinates within an ordination space (usually NMDS), while ANOSIM is usually used with all variables (i.e., on the original data).

In this example, we first fit an NMDS ordination to the `iris` data, then use MRPP to test whether the 3 species differ from each other. NMDS will be described in detail in its own page, but for now just understand that proximity in NMDS coordinates represents proximity in terms of the original variables. The question addressed by the MRPP is basically the same as asking whether the 3 clouds of points in the ordination space overlap or not.

```

# NMDS ordination
library(vegan)
n1 <- metaMDS(iris[,1:4])

## Run 0 stress 0.03775523
## Run 1 stress 0.03775524
## ... Procrustes: rmse 9.172761e-06 max resid 5.394399e-05
## ... Similar to previous best
## Run 2 stress 0.03775524
## ... Procrustes: rmse 8.541023e-06 max resid 7.918737e-05
## ... Similar to previous best
## Run 3 stress 0.05313096
## Run 4 stress 0.04367533
## Run 5 stress 0.04355785
## Run 6 stress 0.03775522

```

```

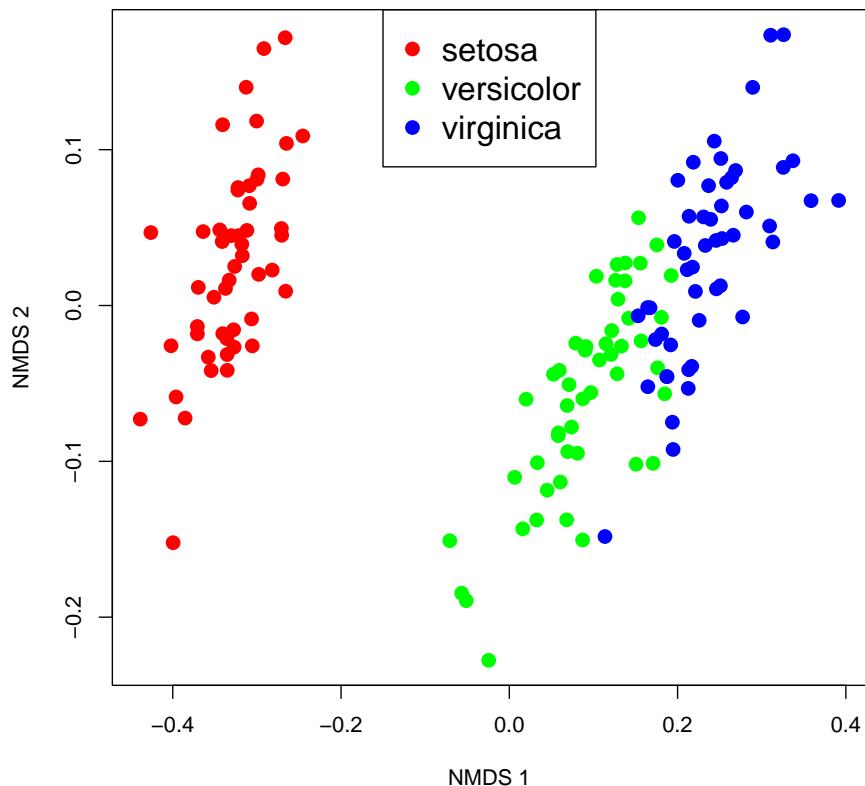
## ... New best solution
## ... Procrustes: rmse 7.683165e-06 max resid 7.674469e-05
## ... Similar to previous best
## Run 7 stress 0.03775524
## ... Procrustes: rmse 6.562437e-06 max resid 2.98595e-05
## ... Similar to previous best
## Run 8 stress 0.04367538
## Run 9 stress 0.0505973
## Run 10 stress 0.04804019
## Run 11 stress 0.05059737
## Run 12 stress 0.03775521
## ... New best solution
## ... Procrustes: rmse 8.597169e-06 max resid 4.289104e-05
## ... Similar to previous best
## Run 13 stress 0.03775522
## ... Procrustes: rmse 2.949822e-06 max resid 1.237819e-05
## ... Similar to previous best
## Run 14 stress 0.03775523
## ... Procrustes: rmse 1.114005e-05 max resid 5.032798e-05
## ... Similar to previous best
## Run 15 stress 0.06096633
## Run 16 stress 0.0436752
## Run 17 stress 0.03775525
## ... Procrustes: rmse 1.186399e-05 max resid 7.094844e-05
## ... Similar to previous best
## Run 18 stress 0.04709617
## Run 19 stress 0.04367538
## Run 20 stress 0.04713709
## *** Solution reached

# extract scores (coordinates)
nx <- scores(n1)[,1]
ny <- scores(n1)[,2]

# set up some colors for the plot
cols <- rainbow(3)
use.cols <- cols[as.numeric(iris$Species)]

# make the plot
plot(nx, ny, pch=16, col=use.cols,
      cex=1.5,
      xlab="NMDS 1", ylab="NMDS 2")
legend("top", legend=unique(iris$Species),
      pch=16, cex=1.5, col=cols)

```



The figure suggests that there is good separation between at least *I. setosa* and the other species, and likely separation between *I. versicolor* and *I. virginica*. The MRPP will test whether or not the clouds of points overlap.

```
mrpp(scores(n1), iris$Species)
```

```
##
## Call:
## mrpp(dat = scores(n1), grouping = iris$Species)
##
## Dissimilarity index: euclidean
## Weights for groups: n
##
## Class means and counts:
##
##      setosa  versicolor virginica
## delta 0.09269  0.1086     0.1035
```

```

## n      50      50      50
##
## Chance corrected within-group agreement A: 0.6635
## Based on observed delta 0.1016 and expected delta 0.3018
##
## Significance of delta: 0.001
## Permutation: free
## Number of permutations: 999

```

The test statistics are δ (the weighted mean within-group distance); and A , the chance-corrected within-group agreement or agreement statistic.

- The δ value for each group with expresses how homogenous the group is.
 - When $\delta = 0$, all members of a group are identical (i.e., have identical location in the NMDS space).
 - The mean of the group-level δ is reported as the “observed delta”.
- The agreement statistic A scales δ to the within-group homogeneity expected by chance if group membership was unrelated to location (expected delta in the output above).
 - Greater values of A indicate higher agreement within groups.
 - A ranges from 0 (heterogeneity within groups equal to heterogeneity expected by chance) to 1 (items within groups identical).
 - In practice, values of $A > 0.3$ are rather high.

The significance of the MRPP is calculated by permutation, just as in the Mantel test or ANOSIM. As with any statistical test, the P -value is sensitive to not only the effect size, but also the sample size. Large sample sizes can make tiny differences appear statistically significant, so it is up to you as a biologist to interpret the output of the test.

8.5 Ordination

In the last section we saw an application of **ordination**: representing high-dimensional relationships between objects in a 2-d space. This is done in such a way as to represent important patterns in the data in few enough dimensions for our 3-d brains to handle.

Ordination is literally **ordering observations along two or more axes**. That is, coming up with a new coordinate system that shows relationships between observations. How that ordering is done varies wildly among techniques. One broad class of techniques, collectively called **eigenvector** based methods, use the power of linear algebra to place the data into a new coordinate system that better captures the patterns in the data. The other class uses **Monte Carlo sampling** to find arrangements of samples that retain the important patterns in reduced dimension space.

No matter how they work, all ordination methods have the same goal: repre-

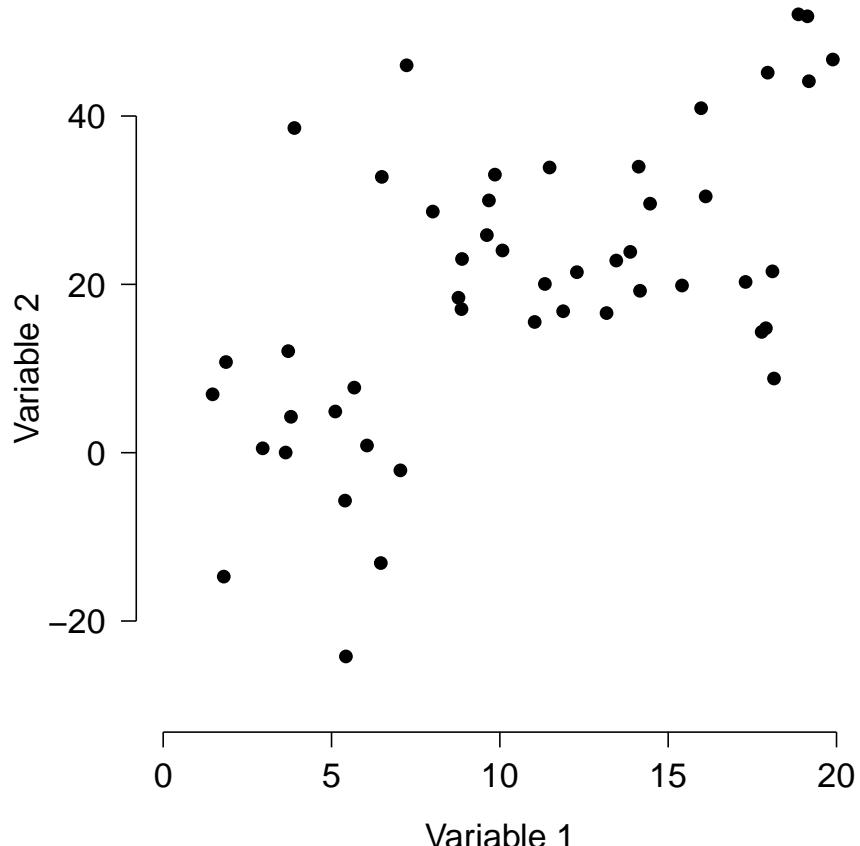
senting patterns found in many dimensions in fewer dimensions. For this course our focus will be on interpretation rather than calculation. The applications of ordination overlap with those of the multivariate techniques that we have already seen.

- **Cluster identification:** observations that are closer to each other in the ordination space are more similar to each other
- **Dimension reduction:** the axes of an ordination are estimates of synthetic variables that combine information about many variables at once.

8.5.1 Principal components analysis (PCA)

8.5.1.1 PCA intro

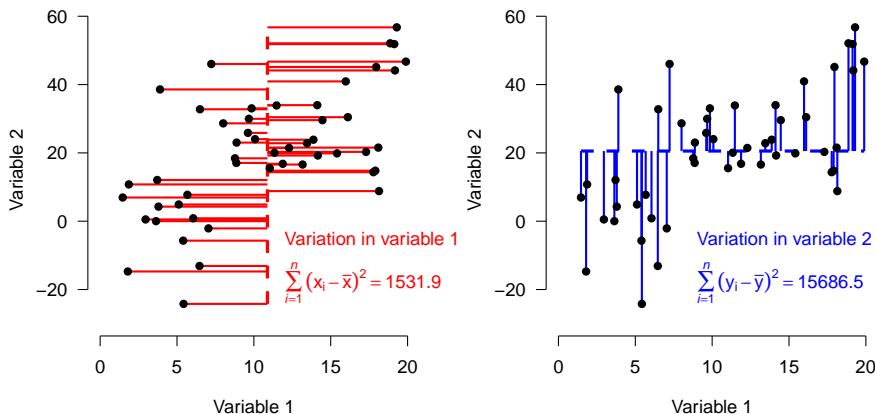
Principal components analysis (PCA) is a method for extracting synthetic gradients from a multivariate dataset that capture most of the variation in that dataset. These gradients are calculated by finding linear combinations of the variables that minimize sums of squared deviations from the gradient. This means that PCA has a lot in common with linear regression, and many of the same assumptions apply. If you've never heard of principal components analysis or ordination, it might be worth watching a video that explains and shows the basic ideas. Here is one that only takes 5 minutes and has a nice theme song (accessed 2021-08-10) . Imagine a dataset with 2 variables, x and y . You could capture and display all the information about this dataset in a 2-d scatterplot, by simply plotting y vs. x . Likewise, you could capture and display all of the information about a 3-dimensional dataset with a 3-d plot. For 4 or more dimensions, a true scatterplot can't be rendered sensibly or even understood by our pathetic meat-based brains.



One way to describe the variation in the dataset above is to think about how the samples vary along each axis. The figure below shows how the variation among samples can be broken down into the variation in Variable 1 and the variation in Variable 2. When we say “variation in variable 1”, we mean “deviation between the values of variable 1 and the mean of variable 1”. That is what is shown in the figure below.

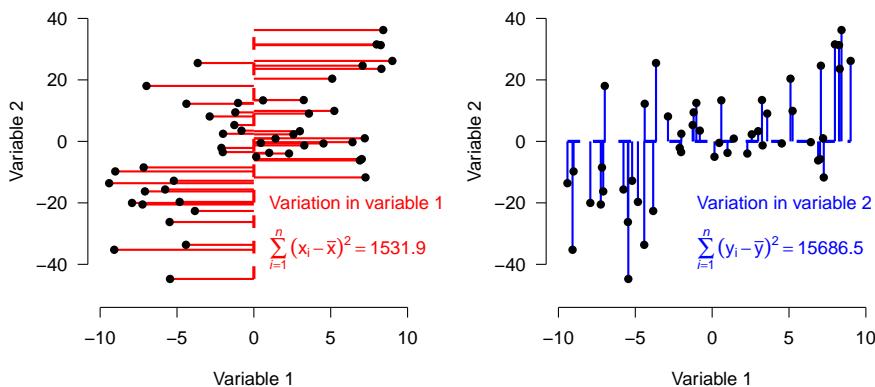
- Each point, or sample, has a Variable 1 coordinate and a Variable 2 coordinate.
- Each point’s value for Variable 1 can be thought of as the difference between that value and the mean of Variable 1.
- Likewise, each point’s value for Variable 2 can be thought of as the difference between that value and the mean of Variable 2.
- The position of each point can thus be reframed as its deviation with respect to the Variable 1, and its deviation with respect to Variable 2. This reframing is the same as centering each variable (i.e., subtracting the mean).

- The total variance among the samples is equal to the variance with respect to Variable 1 plus the variance with respect to Variable 2.



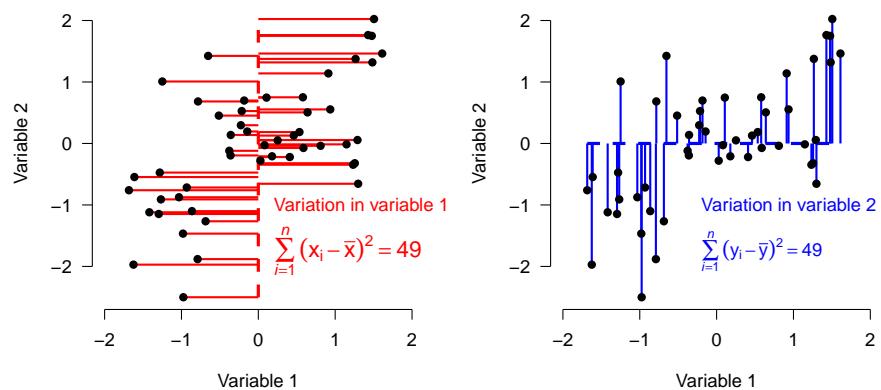
The figure above shows how the total variation in the dataset is split up (“partitioned”) into variation in Variable 1 and variation in Variable 2.

Describing each observation as its deviation from the mean of each variable has the effect of **centering** the points at the origin. Notice that the variation, expressed as sums of squared deviations, is unchanged.

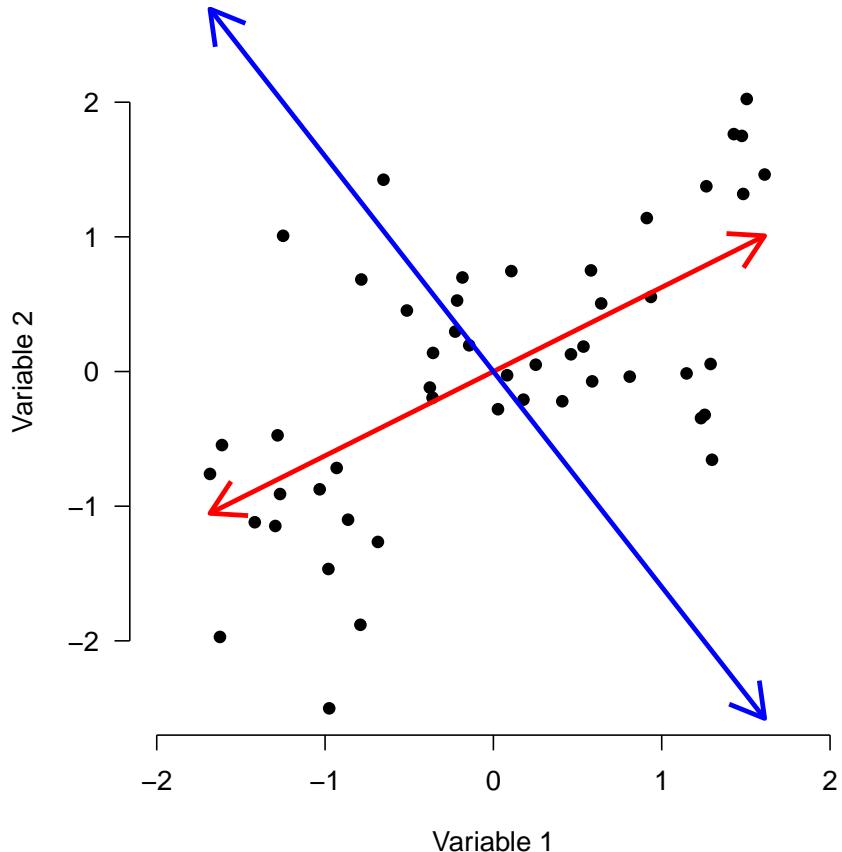


If variables have different ranges, it is a good idea to scale them as well as center them (aka: standardizing or Z -scaling). This means that the original values are converted to Z -scores by subtracting the mean and dividing by the SD. Trying to use PCA or other eigenvector-based methods without standardizing variables will distort the results. Standardization puts variation along any axis on equal footing.

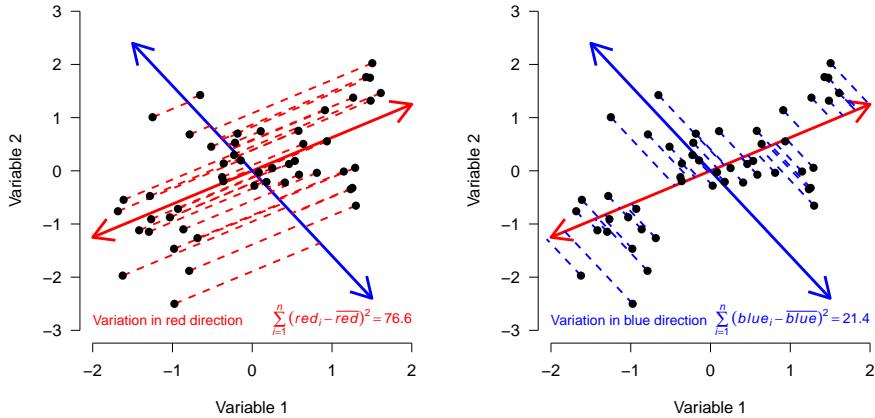
The figure below shows the data as deviations from variable means after standardization. Note that the sums of squares are equal in both directions now.



If our goal is to describe variation in the data as succinctly as possible, then using Variable 1 and Variable 2 as axes might not be the best approach. Notice that most of the variation in the points doesn't line up exactly with either of the variables, but along the red arrow shown below. The rest of the variation is along the direction perpendicular to the red arrow, illustrated by the blue arrow. (Note that the arrows may only appear perpendicular if the plot is precisely square).



The red arrow is a vector that contains information about both Variable 1 and Variable 2. It is tricky, but not too hard, to calculate each observation's deviations from the mean of that vector (left panel below). We can also calculate the deviations along a perpendicular vector, because even after describing the variation in the “red” direction, there is still some variation in the “blue” direction (right panel).



We know that the variation on the red and blue axes is the same as the variation on the X and Y axes, because the sums of squares are the same. The red and blue axes are just different ways of orienting the data to express the same patterns. All we really did was **rotate** the original coordinate system (defined by Variables 1 and 2) to a new coordinate system (defined by red and blue).

You may have guessed that the red and blue axes have special names: they are the **principal components** of this dataset. Principal components (PCs) are **synthetic or composite axes** that capture most of the variation. A PC is a **linear combination** of each of the original variables. This is easy to see in the figure above, because PC1 is defined by its coordinates. The same logic is true with more variables; it's just harder to visualize.

8.5.1.2 PCA—more details

The procedure above illustrates the geometric interpretation of PCA. The general procedure is:

1. Begin with the samples as a cloud of n points in a p -dimensional space.
2. Center (and usually scale) the axes in the point cloud. This will place the origin of the new coordinate system at the p -dimensional centroid of the cloud.
3. Rotate the axes to maximize the variance along the axes. As the angle of rotation θ changes, the variance σ^2 will also change.
4. Continue to rotate the axes until the variance along each axis is maximized. Because the data are changed only by rotation, the Euclidean distances between them are preserved.

The other way PCA can be understood is in the language of linear algebra. The procedure is roughly given by:

1. Calculate the variance-covariance matrix \mathbf{S} of the data matrix \mathbf{A} . This is a $p \times p$ square matrix with the variances of each variable on the diagonal,

and covariances between pairs of variables in the upper and lower triangles.

2. Find the eigenvalues λ of \mathbf{S} . Each eigenvalue represents a portion of the original total variance—the proportion corresponding to a particular principal component.
3. Find the eigenvectors. For each eigenvalue λ , there is an eigenvector that contains the coefficients of the linear equation for that principal component. Together the eigenvectors form a $p \times p$ matrix, \mathbf{Y} .
4. Find the scores for the original samples on each principal component as $\mathbf{X} = \mathbf{AY}$. The linear algebra method of PCA in R is illustrated below.

```
# generate some random data for PCA
set.seed(123)
n <- 50
x <- runif(n, 1, 20)
y <- 1.2 + 1.7*x + rnorm(n, 0, 15)

# produces data with a linear relationship between x and y
# data matrix: n rows * p columns
A <- cbind(x,y)

# standardize each variable
A <- apply(A, 2, scale)

# calculate variance-covariance matrix S
S <- cov(A)
```

The variance-covariance matrix \mathbf{S} contains the variances of the variables on the diagonal. Both variances are 1 because we scaled the variables (compare to `cov(A)` to see the difference). This matrix \mathbf{S} is symmetric because the covariance function is reflexive; i.e., $\text{Cov}(x,y) = \text{Cov}(y,x)$. The variance-covariance matrix is useful because it contains information about both the spread (variance) and orientation (covariance) in the data. For a dataset like ours with 2 variables, the variance-covariance matrix has 2 dimensions (one for each variable).

$$\mathbf{S} = \begin{bmatrix} \text{Var}(x) & \text{Cov}(x,y) \\ \text{Cov}(y,x) & \text{Var}(y) \end{bmatrix}$$

```
# calculate eigenvalues and eigenvectors
eigens <- eigen(S)
evals <- eigens$values
evecs <- eigens$vectors
```

PCA is all about defining a new coordinate system for the data that preserves Euclidean distances, but maximizes the variance captured on the axes of the new system. The axes of the new system are found as linear combinations of the original variables. This means that the new coordinate system will have as many dimensions as the original coordinate system.

The data in our example can be thought of as a matrix with two columns; each sample is defined by a vector of length two (one for each dimension). That vector simply contains the x and y coordinates of the sample. If the dataset is matrix \mathbf{X} , then each point is a vector v_i where $i = 1, 2, \dots, n$ (n = number of samples). For example, the vector $[1, 2]$ corresponds to an observation with $x = 1$ and $y = 2$ (this vector is also a row of \mathbf{X}).

To transform the points in the original coordinate system to a new coordinate system, we multiply each vector v_i by a $p \times p$ transformation matrix \mathbf{T} (recall that p is the number of dimensions) to get the transformed coordinates b_i .

$$\mathbf{T}v_i = b_i$$

Or written out fully:

$$\begin{bmatrix} \mathbf{T}_{1,1} & \mathbf{T}_{1,2} \\ \mathbf{T}_{2,1} & \mathbf{T}_{2,2} \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix} = \begin{bmatrix} \mathbf{T}_{1,1}x_i & \mathbf{T}_{1,2}y_i \\ \mathbf{T}_{2,1}x_i & \mathbf{T}_{2,2}y_i \end{bmatrix}$$

It turns out that some vectors v_i have a very interesting property: when transformed by \mathbf{T} , they change length but not direction. These vectors are called **eigenvectors**. The **scalar** (aka: constant) that defines the change in their length (aka: magnitude) is called an **eigenvalue**. This property is expressed compactly as:

$$\mathbf{T}v_i = \lambda b_i$$

The interpretation of this expression is that transforming a coordinate v_i into a new coordinate system with matrix \mathbf{T} is the same as multiplying v_i by the eigenvalue λ . If the eigenvectors are collected into a new matrix \mathbf{V} , and the eigenvalues are collected into a vector \mathbf{L} , then

$$\mathbf{S}\mathbf{V} = \mathbf{L}\mathbf{V}$$

Put in terms of our 2-d example,

$$\begin{bmatrix} \text{Var}(x) & \text{Cov}(x, y) \\ \text{Cov}(y, x) & \text{Var}(y) \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} \lambda_1 \\ \lambda_2 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}$$

Once the terms are solved (by the computer!), we can put the eigenvectors in descending order of their eigenvalues to get the principal components.

```
# calculate scores by matrix operations
pc <- A %*% evecs

# check that variances = eigenvalues and
```

```
# covariances = 0 (must be, by definition of PCA)
## close enough (<1e-16), probably a numerical limit involved
cov(pc)

## [,1]      [,2]
## [1,] 1.625446e+00 9.690887e-17
## [2,] 9.690887e-17 3.745539e-01
# variance explained by each PC
round(evals/sum(evals),3)

## [1] 0.813 0.187
```

The result shows us that about 81% of the variation is explained by PC1 alone, with the remainder explained by PC2. Notice that all of the variation in the dataset is associated with 1 and only 1 PC. This is part of the definition of PCA—the total variation among the samples does not change. All that changes is how we describe the variation (i.e., how we orient the axes used to describe the samples).

Why go through all of this trouble? Besides the benefit of understanding what PCA is doing, we can use the linear algebra to translate between the coordinate system of the original data, and the coordinates system defined by the PCs. The algebra is complicated, but it boils down to multiplying the matrix of PC scores by the transpose of the eigenvector matrix:

```
backxy <- pc %*% t(evecs)
range(A-backxy)

## [1] -4.440892e-16 4.440892e-16
```

The “back-calculated” coordinates are off by at most 4.5×10^{-16} . The differences should technically be 0, but they are approximated to a very small number by R.

8.5.1.3 PCA in R (package vegan)

There are two methods for PCA in base R: `prcomp()` and `princomp()`. Both of these methods produce similar outputs, and the only major difference between them is the syntax for extracting results. For this course we are going to use the ordination functions in package `vegan`. Although designed for community ecology, `vegan` is widely used for ordination and multivariate analysis in many fields. Importantly, `vegan` is actively maintained and updated with new techniques as they are developed. Also importantly, `vegan` offers a common interface for many kinds of ordination.

The example below uses the bat brain dataset that we have used before. Load the dataset into R and take a look. Hutcheon et al. (2002) reported data on brain morphology and lifestyle from 63 species of bats. Their dataset contains the following variables:

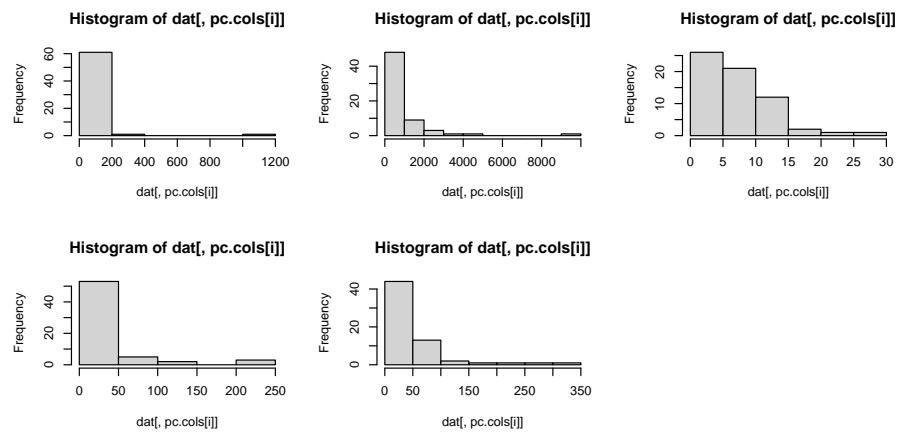
Variable	Meaning
species	Species
family	Taxonomic family
diet	Herbivore, gleaner, hawker, or vampire
bow	Body weight (g)
brw	Brain weight (μg)
aud	Auditory nuclei volume (mm^3)
mob	Main olfactory bulb volume (mm^3)
hip	Hippcampus volume (mm^3)

Import the dataset `bat_data_example.csv`. You can download it [here](#). The code below requires that you have the file in your R working directory.

```
library(vegan)
library(rgl)
in.name <- "bat_data_example.csv"
dat <- read.csv(in.name, header=TRUE)
```

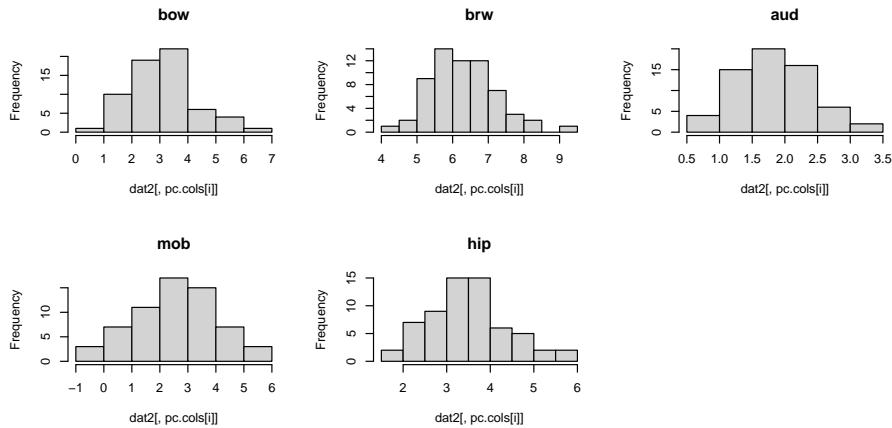
We are going to explore how brain morphology, measured as volumes of various brain parts, varies with taxonomy and lifestyle. First, we need to make sure that our data are suitable for PCA. We can inspect distributions of the variables with histograms.

```
pc.cols <- 4:ncol(dat)
par(mfrow=c(2,3))
for(i in 1:length(pc.cols)){hist(dat[,pc.cols[i]])}
```



The histograms suggest that a log transform would be appropriate, because PCA requires multivariate normality.

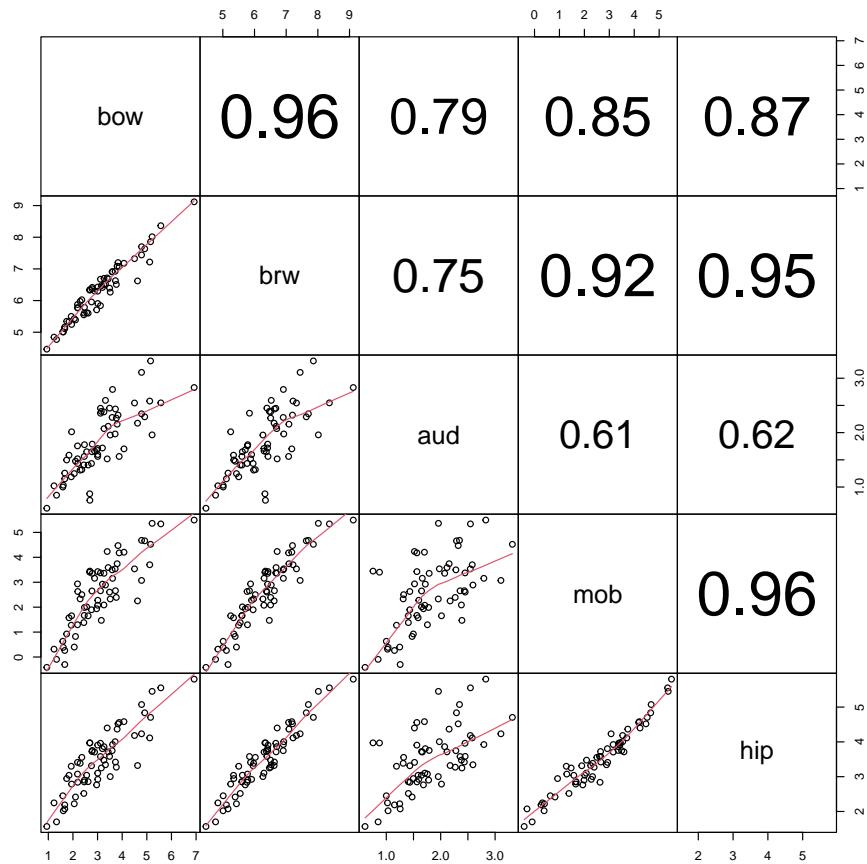
```
dat2 <- dat
dat2[,pc.cols] <- apply(dat2[,pc.cols], 2, log)
par(mfrow=c(2,3))
for(i in 1:length(pc.cols)){
  hist(dat2[,pc.cols[i]],
       main=names(dat2)[pc.cols[i]])
}
```



Much better. We should also check to see if any of the variables are related to each other. It's okay if they are, but such relationships need to be kept in mind when interpreting the PCA (or any ordination, for that matter). To explore relationships between the variables, we will use a `pairs()` plot and a function borrowed from the `pairs()` help page.

```
# borrowed from ?pairs examples
panel.cor <- function(x, y, digits = 2, prefix = "", cex.cor, ...)
{
  usr <- par("usr"); on.exit(par(usr))
  par usr = c(0, 1, 0, 1)
  r <- abs(cor(x, y))
  txt <- format(c(r, 0.123456789), digits = digits)[1]
  txt <- paste0(prefix, txt)
  if(missing(cex.cor)) cex.cor <- 0.8/strwidth(txt)
  text(0.5, 0.5, txt, cex = cex.cor * r)
}

# make our plot:
pairs(dat2[,pc.cols],
      lower.panel = panel.smooth,
      upper.panel = panel.cor,
      gap=0)
```



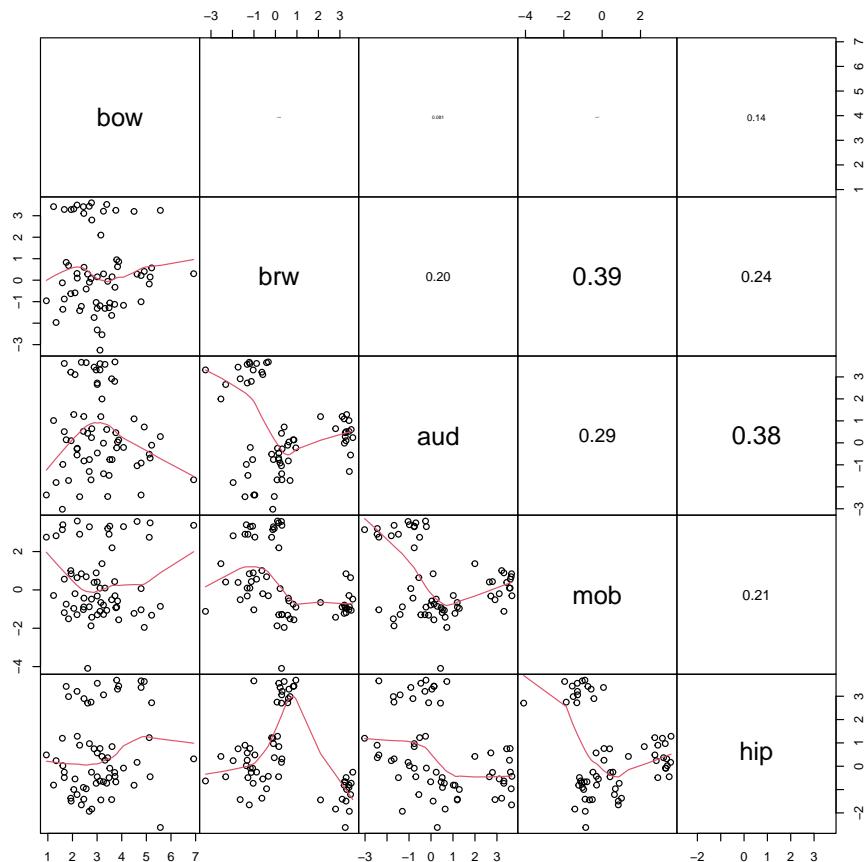
The scatterplot matrix suggests that every variable is closely related to every other variable. This makes sense for morphological data: the sizes of different body parts tend to scale with overall body size. We are interested in brain morphology independent of size, we can factor out size by dividing the volumes of each brain part by the body weight. Because the data are already on the log scale, the division is accomplished by subtracting the log-transformed body weight. The new values of `brw`, `aud`, `mob`, and `hip` are the sizes of those brain components with body size factored out. Neglecting to factor out body size or overall size prior to an ordination will result in an ordination dominated by body size.

```
dat3 <- dat2
dat3[,pc.cols[-1]] <- apply(dat3[,pc.cols], 1,
  function(x){x[-1]-x[1]})
```

`pairs(dat3[,pc.cols],`

```
  lower.panel = panel.smooth,
  upper.panel = panel.cor,
```

```
gap=0)
```



Next, use the function `rda()` from `vegan` to calculate the PCA. In this PCA we scale the variables (aka: standardize) by subtracting the mean and dividing by the SD. This has the effect of making each variable have mean = 0 and SD = 1. Scaling is not required for PCA but is **highly** recommended. If values are not scaled, then the PCA will be dominated by the variables with the greatest values or by the observations with extreme values.

```
p1 <- rda(dat3[,pc.cols], scale=TRUE)
summary(p1)

##
## Call:
## rda(X = dat3[, pc.cols], scale = TRUE)
##
## Partitioning of correlations:
```

```

##                  Inertia Proportion
## Total           5       1
## Unconstrained  5       1
##
## Eigenvalues, and their contribution to the correlations
##
## Importance of components:
##                  PC1    PC2    PC3    PC4    PC5
## Eigenvalue      1.4884 1.3471 1.0867 0.9389 0.13888
## Proportion Explained 0.2977 0.2694 0.2173 0.1878 0.02778
## Cumulative Proportion 0.2977 0.5671 0.7844 0.9722 1.00000
##
## Scaling 2 for species and site scores
## * Species are scaled proportional to eigenvalues
## * Sites are unscaled: weighted dispersion equal on all dimensions
## * General scaling constant of scores: 4.196048
##
##
## Species scores
##
##                  PC1    PC2    PC3    PC4    PC5
## bow   0.6502 -0.3804 0.3746 1.6772 -0.0256
## brw  -0.8926 -1.2116 -1.0353 0.2617  0.3410
## aud  -1.2718  0.7591  1.0657 0.2616  0.3513
## mob   0.9822  1.2887 -0.8596 0.1665  0.3596
## hip   1.1999 -0.9455  0.8601 -0.5732  0.3456
##
##
## Site scores (weighted sums of species scores)
##
##                  PC1    PC2    PC3    PC4    PC5
## sit1  -0.4822 -0.47110 -0.51367 0.032283 0.474975
## sit2  -0.1875  0.46735  0.51711 0.369969 -0.003147
## sit3   0.7140  0.21390 -0.80651 -0.818336 -0.071516
## sit4   0.4038 -0.72892  0.44826 -0.063108 0.690554
## sit5  -0.4484 -0.47079 -0.46877 0.246659 -0.080258
## sit6  -0.1740  0.54116  0.76904 -0.090556 0.443921
## sit7   0.7314  0.13147 -0.08352 0.791550 0.556946
## sit8   0.4378 -0.68647  0.29135 -0.318361 -0.389457
## sit9  -0.6786 -0.40593 -0.49047 -0.055734 0.047201
## sit10 -0.5471  0.46826  0.39227 -0.277305 0.021265
## sit11  0.5047  0.45175 -0.68715 -1.046120 -0.554421
## sit12  0.2737 -0.76045  0.56922 -0.009404 0.423659
## sit13 -0.7074 -0.40835 -0.46360 -0.442119 0.136189
## sit14 -0.4174  0.54986  0.56501 0.390291 -0.996710
## sit15  0.7238  0.23208 -0.69268 0.086960 -0.220235

```

```

## sit16  0.4023 -0.71503  0.56602  0.622207 -0.030857
## sit17 -0.5113 -0.39390 -0.48222  1.449440 -0.874003
## sit18 -0.5161  0.65950  0.41766 -0.464633  0.546833
## sit19  0.5582  0.59672 -0.21665  0.456523  0.041643
## sit20  0.7217 -0.57541  0.37694  0.361859  0.367978
## sit21 -0.7814 -0.18490 -0.59536 -0.546239  0.624799
## sit22 -0.5743  0.86534  0.31746 -0.031015 -0.046946
## sit23  0.6501  0.30104 -0.50589  0.704037  0.450911
## sit24  0.1322 -0.48776  0.29655 -0.905271  0.829062
## sit25 -0.6016 -0.09952 -0.25379  0.285644 -0.292755
## sit26  0.1762  0.68850  0.53632  0.011641 -0.162855
## sit27  0.7698  0.39111 -0.48473 -0.523310 -0.586544
## sit28  0.9450 -0.72682  0.51129  0.189683 -0.972058
## sit29 -0.5400 -0.44113 -0.89254  0.077095 -1.008438
## sit30 -0.2476  0.41802  0.76914  0.033110  0.585923
## sit31  0.4579  0.43434 -0.42246 -0.427813  1.254583
## sit32  0.3163 -0.48116  0.14197 -0.583475  0.258704
## sit33 -0.7874 -0.22881 -0.42449  0.020886  0.036846
## sit34 -0.3562  0.67117  0.40279  0.130779  0.254196
## sit35  0.5619  0.57716 -0.36416  0.198927 -0.096652
## sit36  0.3383 -0.69771  0.27202 -0.543479 -0.917745
## sit37 -0.5503 -0.04613 -0.87845 -0.183795  0.366676
## sit38 -0.5610  0.76767  0.14607 -0.233642  0.203234
## sit39  0.5757  0.53175 -0.44112  0.176223 -0.783513
## sit40  0.1957 -0.53875  0.38645 -0.748866  0.152175
## sit41 -0.5793 -0.50413 -0.38612  0.342061  0.134688
## sit42 -0.4700  0.34255  0.60210  0.442490  0.268606
## sit43  0.4381  0.72725 -0.50710 -0.652534 -0.132495
## sit44  0.3847 -0.57265  0.44835 -0.003145  0.642012
## sit45 -0.5225 -0.50720 -0.16657  0.831065  0.048770
## sit46 -0.1615  0.68519  0.58968  0.026128 -0.556304
## sit47  0.9713  0.04069 -0.36804  1.647929  0.094091
## sit48  0.2589 -0.69440  0.08940 -0.926043 -0.419415
## sit49 -0.4445 -0.20883 -0.92458 -0.132031  0.220466
## sit50 -0.5502  0.63824  0.32205  0.014305  0.458476
## sit51  0.5413  0.65064 -0.46422 -0.868767 -0.806164
## sit52  0.3340 -0.79973  0.88265  0.412611  0.351906
## sit53 -0.8137 -0.28059 -0.41296 -0.184942  0.305652
## sit54 -0.2019  0.32853  0.67470 -0.050061 -0.030891
## sit55  0.5214  0.37534 -0.43319 -0.221992  0.717535
## sit56  0.3656 -0.62963  0.49570 -0.443295  0.124316
## sit57 -0.4750 -0.49025 -0.34624  0.481810  0.037603
## sit58 -0.2769  0.47637  0.73519  0.163663  0.314753
## sit59  0.4670  0.13039 -0.27637  0.167998  0.281708
## sit60 -0.1991 -0.86051  0.80864 -0.561143 -0.932744
## sit61 -0.7639 -0.28665 -0.42427  0.151890 -0.689890

```

```
## sit62 -0.3285  0.65774  0.99773  0.038309 -1.346359
## sit63  0.5844  0.37256 -0.46123  1.000511  0.233515
```

The summary of the output contains most of what we need. Of greatest interest are the relative contributions of each PC. That is, how much of the overall variation is associated with each PC.

The overall variation is the sum of the variances of the original variables. The PCA was fit using centered and scaled values, so we need to scale the original data to see the variances that were used in the PCA. The scaling meant that each variable ended up with mean = 0 and variance = 1, so the total variance was equal to 5.

```
apply(scale(dat3[,pc.cols]), 2, var)
```

```
## bow brw aud mob hip
##   1   1   1   1   1
```

That total variance of 5 was partitioned into 5 PCs. In PCA, data are ordinated on one PC for each of the original variables. Each PC is a combination of the original variables. The summary table for the PCA shows how the total variance was split up.

- The first row, **Eigenvalue**, is the variance associated with each PC. The sum of the eigenvalues equals the total variance, 5.
- The **Proportion Explained** row shows the proportion of total variance captured on each PC. PC1 captures 29.8%, PC2 captured 26.9%, and so on. For example, the proportion of variance explained by PC1, 0.2977, is equal to the eigenvalue of PC1 divided by the total variance ($1.4884 / 5$).
- The **Cumulative Proportion** row is the running total of proportion of variance explained, starting with PC1. The rule of thumb for PCA is that you should present and interpret enough PCs to capture $\geq 80\%$ of the variation. In this example, it takes 3 PCs to get up to about 80% of the variation explained). Two PCs is easier to deal with, but sometimes you need 3.

```
summary(p1)$cont$importance
```

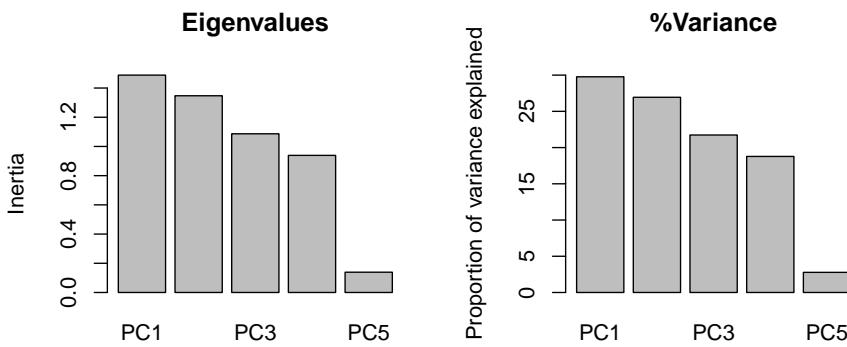
```
## Importance of components:
##                               PC1      PC2      PC3      PC4      PC5
## Eigenvalue            1.4884  1.3471  1.0867  0.9389  0.13888
## Proportion Explained 0.2977  0.2694  0.2173  0.1878  0.02778
## Cumulative Proportion 0.2977  0.5671  0.7844  0.9722  1.00000
```

An alternative strategy for deciding how many PCs to interpret is to look at a screeplot, which shows the relative contributions of each PC to the overall variance. The variance is expressed as “Inertia”—the eigenvalues of the PCs. The proportion explained by each axis (seen in the table above), is simply the eigenvalues of the axes divided by the total of all eigenvalues. Some people prefer

to present a screeplot that shows proportion of variance explained rather than the eigenvalues.

```
par(mfrow=c(1,2))
screeplot(p1, main="Eigenvalues")

# alternative version with proportion of variance explained
# instead of eigenvalues (variances)
prx <- 100*summary(p1)$cont$importance[2,]
barplot(prx, ylim=c(0, 30),
        main="%Variance",
        ylab="Proportion of variance explained")
```



The **loadings** of the variables express how much each variable is associated with each PC. These values have two interpretations:

- First, they are the correlations between the variables and the PCs.
- Second, they are the coordinates for the biplot vectors (see below), which help us see the relationships between the ordination and the variables.
 - Note: the biplot vectors implied by these coordinates are sometimes rescaled to more faithfully represent the relationships between variables. See Legendre and Legendre (2012) for a thorough explanation.

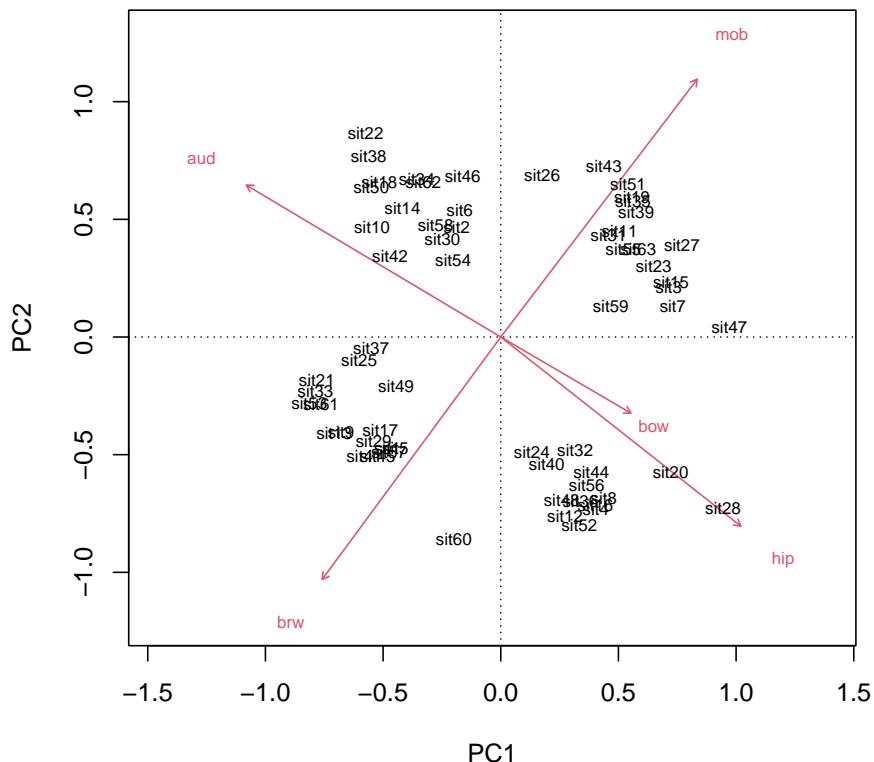
```
scores(p1, choices = 1:4, display = "species", scaling = 0)

##          PC1          PC2          PC3          PC4
## bow  0.2840062 -0.1746469  0.1914720  0.92240824
## brw -0.3898723 -0.5563058 -0.5292561  0.14392406
## aud -0.5555135  0.3485255  0.5448121  0.14388004
## mob  0.4290023  0.5917029 -0.4394228  0.09156649
## hip  0.5241200 -0.4340967  0.4396748 -0.31521900
## attr(",const")
## [1] 4.196048
```

PC1 is most strongly correlated with `aud` ($r = -0.55$) and `hip` ($r = 0.52$). PC2 is most strongly correlated with `mob` ($r = 0.59$) and `brw` ($r = -0.55$). Ideally, each PC would have a few variables strongly correlated with it ($r > 0.7$), but that isn't the case here. Interestingly, most of the variables are moderately correlated with first three axes. This suggests that none of the variables is strongly driving any of the PCs. We can check this with a biplot.

An ordination **biplot** is probably the most important tool for interpreting the relationships in the data captured by the ordination. It is called a biplot because it presents two kinds of data: *similarity between the samples* indicated by proximity in the ordination space; and *relationships between some set of quantitative variables and the ordination axes*. The samples are plotted as points; the variables are plotted as vectors radiating from the origin.

```
par(mfrow=c(1,1))  
biplot(p1)
```

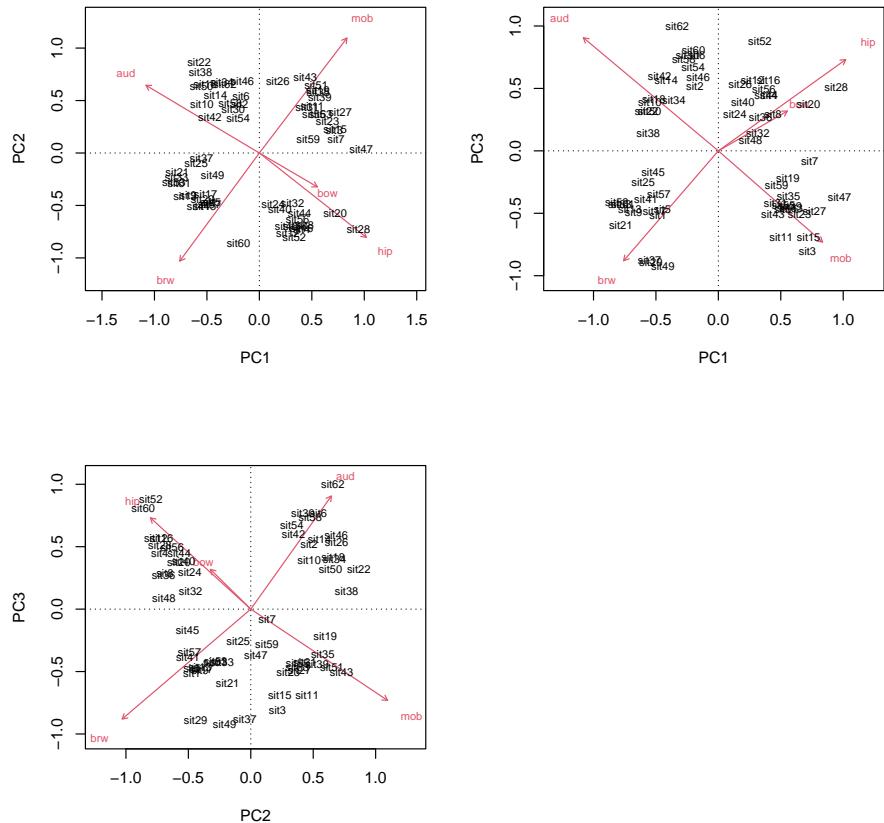


The biplot shows the points (by row name/number) and the five numeric variables that went into the PCA. Each vector shows the projection of a variable into the ordination space.

- The direction of a vector shows the direction (in ordination space) in which a variable increases. For example, samples in the upper left have increased `aud`; samples in the upper right have increased `mob`, and so on. A variable decreases in the direction opposite its vector: samples in the lower right have decreased `aud`.
 - Compare the variable loadings with the biplot arrows. Do these values make sense?
- Relative angles of vectors reflect the correlation between the underlying variables.
 - Variables whose vectors point in the same direction are positively correlated with each other; the smaller the angle between two vectors, the stronger the correlation (r approaching 1).
 - Vectors perpendicular to each other are uncorrelated (r close to 0).
 - Vectors pointing in opposite directions are negatively correlated with each other (r approaching -1).
 - Correlation coefficients cannot be inferred directly from angles because of how the coordinates on the plot are scaled, but the angles do give a *rough* idea.
- The length of a vector indicates the strength of the correlation with the ordination space. Longer vectors indicate stronger correlations ($|r|$).
- Each variable vector represents an axis of the original coordinate system.
 - Shorter vectors have most of their length off the plane of the biplot.
 - Longer vectors have more of their length near the plane of the biplot.
 - The biplot is really a plane within the original data coordinate system defined by the PCs.

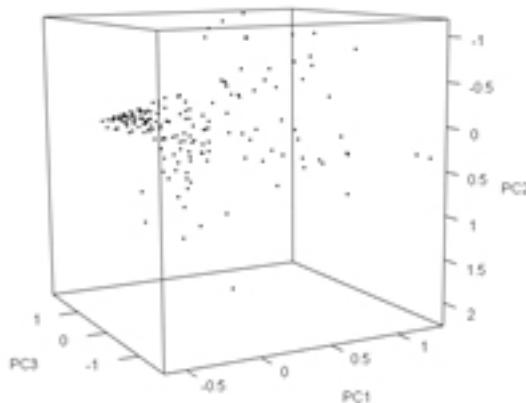
Because it took 3 PCs to get up to about 80% of the variation, we should present and interpret the first 3 PCs. We can plot other variables with the `biplot()` command.

```
par(mfrow=c(2,2))
biplot(p1, choices=c(1,2))
biplot(p1, choices=c(1,3))
biplot(p1, choices=c(2,3))
```



Even better, we can make a 3-d plot using the `rgl` package. Note that the code block below was not run to make this page; try running it on your machine. The plots made by `plot3d()` are cool because you can rotate them with the mouse. Exporting `rgl` figures to static image formats like `.jpg` can be tricky because you must specify the rotation angles, which can pretty much only be done by trial and error.

```
library(rgl)
par(mfrow=c(1,1))
px <- scores(p1, choices=1:3)$sites
plot3d(px[,1], px[,2], px[,3],
      xlab="PC1", ylab="PC2", zlab="PC3")
```



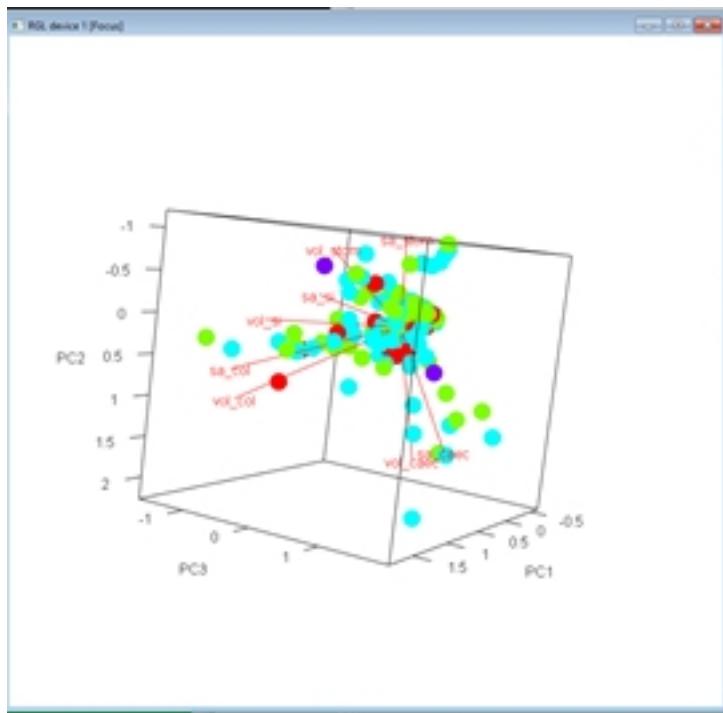
We can add more information to the biplots to help us make sense of the data. Let's color-code the diets.

```

diets <- sort(unique(dat3$diet))
cols <- rainbow(length(diets))
use.cols <- cols[match(dat3$diet, diets)]

par(mfrow=c(1,1))
px <- scores(p1, choices=1:3)$sites
vx <- scores(p1, choices=1:3)$species
plot3d(px[,1], px[,2], px[,3],
      xlab="PC1", ylab="PC2", zlab="PC3",
      col=use.cols, size=50)
for(i in 1:nrow(vx)){
  segments3d(c(0, vx[i,1]), c(0, vx[i,2]), c(0, vx[i,3]), col="red")
  text3d(vx[i,1], vx[i,2], vx[i,3], rownames(vx)[i], col="red")
}

```



The `biplot()` command in `vegan` isn't very flexible, so we if we want a nicer-looking plot we will need to construct it manually.

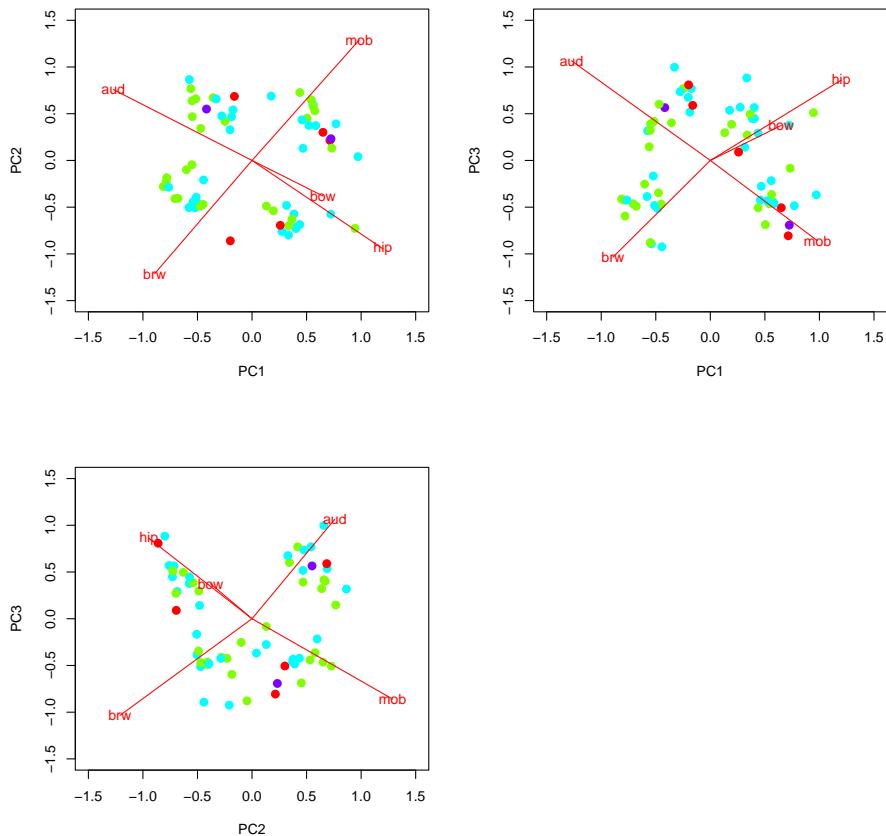
```
px <- scores(p1, choices=1:3)$sites
vx <- scores(p1, choices=1:3)$species

diets <- sort(unique(dat3$diet))
cols <- rainbow(length(diets))
use.cols <- cols[match(dat3$diet, diets)]

par(mfrow=c(2,2))
plot(px[,1], px[,2], pch=16, cex=1.4, col=use.cols,
     xlab="PC1", ylab="PC2", xlim=c(-1.5, 1.5), ylim=c(-1.5, 1.5))
segments(0, 0, vx[,1], vx[,2], col="red")
text(vx[,1], vx[,2], rownames(vx), col="red")
plot(px[,1], px[,3], pch=16, cex=1.4, col=use.cols,
     xlab="PC1", ylab="PC3", xlim=c(-1.5, 1.5), ylim=c(-1.5, 1.5))

segments(0, 0, vx[,1], vx[,3], col="red")
text(vx[,1], vx[,3], rownames(vx), col="red")
plot(px[,2], px[,3], pch=16, cex=1.4, col=use.cols,
     xlab="PC2", ylab="PC3", xlim=c(-1.5, 1.5), ylim=c(-1.5, 1.5))
segments(0, 0, vx[,2], vx[,3], col="red")
```

```
text(vx[,2], vx[,3], rownames(vx), col="red")
```



8.5.1.4 Application of PCA: PC regression

Because PCs capture information about multiple variables at once, they can be used to represent those variables *in other statistical methods*. For example, a PC that represents many measurements of body parts or tree species composition or gene expression can be used as a predictor variable in a logistic regression or as a response variable. This practice is sometimes called **PC regression**. The example below uses the `iris` dataset to illustrate using principal components of flower morphology to predict species identity.

```
library(vegan)
# grab numeric variables
dat <- iris[,1:4]
# standardize each variable
```

```

dat <- apply(dat, 2, scale)
# fit PCA with vegan::rda()
p1 <- rda(dat)

# examine output
summary(p1)

## 
## Call:
## rda(X = dat)
##
## Partitioning of variance:
##           Inertia Proportion
## Total          4            1
## Unconstrained 4            1
##
## Eigenvalues, and their contribution to the variance
##
## Importance of components:
##             PC1     PC2     PC3     PC4
## Eigenvalue   2.9185  0.9140  0.14676  0.020715
## Proportion Explained 0.7296  0.2285  0.03669  0.005179
## Cumulative Proportion 0.7296  0.9581  0.99482  1.000000
##
## Scaling 2 for species and site scores
## * Species are scaled proportional to eigenvalues
## * Sites are unscaled: weighted dispersion equal on all dimensions
## * General scaling constant of scores: 4.940963
##
##
## Species scores
##
##             PC1     PC2     PC3     PC4
## Sepal.Length 2.199 -0.89142  0.6810  0.09290
## Sepal.Width -1.137 -2.18073 -0.2313 -0.04392
## Petal.Length 2.450 -0.05785 -0.1345 -0.28497
## Petal.Width  2.384 -0.15811 -0.6003  0.18617
##
##
## Site scores (weighted sums of species scores)
##
##             PC1     PC2     PC3     PC4
## sit1    -0.534807 -0.202559  0.134486  0.067744
## sit2    -0.491417  0.284467  0.247065  0.288729
## sit3    -0.558311  0.144276 -0.046548  0.079541

```

```

## sit4   -0.542997  0.252085 -0.096137 -0.184874
## sit5   -0.564359 -0.272948 -0.016574 -0.100692
## sit6   -0.490158 -0.628394 -0.028400  0.018523
## sit7   -0.577155 -0.020105 -0.353282 -0.103082
## sit8   -0.527285 -0.094163  0.093405 -0.068988
## sit9   -0.551323  0.470639 -0.152779 -0.075287
## sit10  -0.515827  0.197911  0.267239 -0.111838
## sit11  -0.511572 -0.440410  0.282946  0.046898
## sit12  -0.549314 -0.056156 -0.098737 -0.374156
## sit13  -0.523885  0.307482  0.243171  0.006797
## sit14  -0.621804  0.405731 -0.190395 -0.053861
## sit15  -0.519231 -0.784896  0.498009  0.545836
## sit16  -0.534220 -1.133542 -0.032147  0.141647
## sit17  -0.521320 -0.626044  0.005628  0.529258
## sit18  -0.517249 -0.206277  0.046563  0.260934
## sit19  -0.448346 -0.592881  0.394218  0.171253
## sit20  -0.553384 -0.475923 -0.139672 -0.105832
## sit21  -0.452066 -0.172526  0.443660  0.030613
## sit22  -0.521184 -0.389955 -0.168353  0.167052
## sit23  -0.655159 -0.193409 -0.349815  0.055075
## sit24  -0.429477 -0.036103 -0.036320  0.423650
## sit25  -0.525943 -0.057918 -0.124258 -0.757207
## sit26  -0.460927  0.263995  0.321868  0.122104
## sit27  -0.484377 -0.102187 -0.090949  0.189708
## sit28  -0.512107 -0.222443  0.217797  0.028802
## sit29  -0.505256 -0.132170  0.285546  0.236180
## sit30  -0.534939  0.142514 -0.072069 -0.303510
## sit31  -0.505388  0.212903  0.078991 -0.135074
## sit32  -0.432529 -0.178788  0.284827  0.672360
## sit33  -0.617518 -0.756842 -0.049736 -0.642551
## sit34  -0.577663 -0.907551  0.087057 -0.135146
## sit35  -0.498269  0.194193  0.179315  0.081352
## sit36  -0.521372  0.086972  0.237411  0.473451
## sit37  -0.482959 -0.279160  0.510263  0.550396
## sit38  -0.596827 -0.249932 -0.020468 -0.382624
## sit39  -0.573755  0.381540 -0.203515 -0.027297
## sit40  -0.512375 -0.113460  0.185222  0.019754
## sit41  -0.539949 -0.186392 -0.036748  0.299875
## sit42  -0.438794  0.986328  0.215078  0.812492
## sit43  -0.603039  0.202168 -0.322000 -0.186685
## sit44  -0.463903 -0.199310 -0.326039  0.496394
## sit45  -0.504665 -0.481991 -0.261624 -0.423377
## sit46  -0.488768  0.300046  0.067324  0.393177
## sit47  -0.563153 -0.472792 -0.060256 -0.426705
## sit48  -0.565430  0.162986 -0.146873 -0.136884
## sit49  -0.526482 -0.421113  0.191129 -0.041844

```

```
## sit50 -0.520433 -0.003889  0.161154  0.138390
## sit51  0.260185 -0.364152  0.720933  0.097640
## sit52  0.172705 -0.250912  0.099119  0.013745
## sit53  0.293056 -0.260062  0.583421  0.026414
## sit54  0.096227  0.740313  0.024328  0.184351
## sit55  0.253972  0.087948  0.419400  0.293579
## sit56  0.091788  0.250350 -0.130785 -0.675053
## sit57  0.176292 -0.326194 -0.156879 -0.216869
## sit58 -0.115081  0.781677 -0.262499 -0.113579
## sit59  0.219123 -0.013599  0.627822 -0.083753
## sit60  0.002698  0.436329 -0.567511 -0.079777
## sit61 -0.026023  1.119949  0.049275  0.038572
## sit62  0.104069  0.026709 -0.215962  0.112474
## sit63  0.132741  0.744668  0.806428  0.128185
## sit64  0.169924  0.078582  0.072304 -0.461957
## sit65 -0.007877  0.185248 -0.205282  0.305664
## sit66  0.206727 -0.214812  0.530244  0.294159
## sit67  0.082712  0.082838 -0.516935 -0.536803
## sit68  0.037503  0.334244  0.318073 -0.575452
## sit69  0.289305  0.684545  0.507911  0.633200
## sit70  0.038945  0.549667  0.182014 -0.144991
## sit71  0.174203 -0.167343 -0.649260 -0.233447
## sit72  0.112475  0.176098  0.279019  0.318333
## sit73  0.291450  0.393839  0.387972 -0.027875
## sit74  0.149449  0.175705  0.307394 -0.768643
## sit75  0.165933  0.026758  0.469708  0.121814
## sit76  0.206459 -0.105828  0.497670  0.285111
## sit77  0.296723  0.032600  0.765762  0.111247
## sit78  0.320786 -0.139805  0.274675  0.187320
## sit79  0.156992  0.095336 -0.090423 -0.102141
## sit80 -0.009507  0.446752  0.336540  0.181602
## sit81  0.030887  0.659238  0.157947 -0.026355
## sit82  0.005538  0.663544  0.254377 -0.091862
## sit83  0.057039  0.327982  0.159240  0.066295
## sit84  0.250577  0.267465 -0.110903 -0.515666
## sit85  0.052892  0.121433 -0.700570 -0.714287
## sit86  0.101341 -0.356814 -0.474560 -0.307421
## sit87  0.247656 -0.220292  0.416800  0.104297
## sit88  0.246665  0.583585  0.724840  0.383552
## sit89  0.016433  0.092625 -0.307060 -0.412448
## sit90  0.066943  0.560940 -0.094157  0.024963
## sit91  0.065904  0.472623 -0.099504 -0.758656
## sit92  0.147492 -0.010517  0.021569 -0.413967
## sit93  0.079471  0.417081  0.209976  0.018305
## sit94 -0.085529  0.852066 -0.111439  0.054857
## sit95  0.068149  0.361096 -0.137839 -0.301050
```

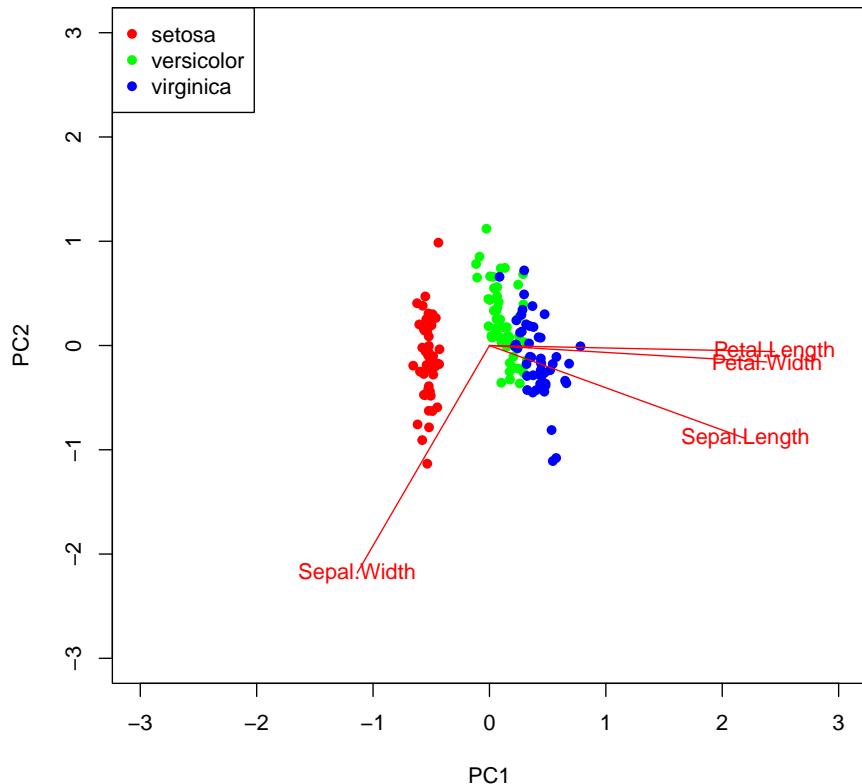
```
## sit96  0.021575  0.076458 -0.135826 -0.644580
## sit97  0.053775  0.162426 -0.164507 -0.371696
## sit98  0.136114  0.065353  0.286073 -0.055670
## sit99  -0.105716  0.651440 -0.200510  0.560453
## sit100 0.060627  0.252700 -0.096757 -0.164318
## sit101 0.435593 -0.367295 -1.058781 -0.138048
## sit102 0.273433  0.294905 -0.558308 -0.113580
## sit103 0.520772 -0.237154  0.213687  0.165894
## sit104 0.340090  0.019828 -0.172318 -0.660867
## sit105 0.441082 -0.124501 -0.416633 -0.045684
## sit106 0.649852 -0.337752  0.613226 -0.284182
## sit107 0.086671  0.658913 -1.039290 -0.373148
## sit108 0.543719 -0.177257  0.686307 -0.667233
## sit109 0.473877  0.300209  0.414909 -0.242490
## sit110 0.533644 -0.810616 -0.418659  0.293865
## sit111 0.322149 -0.292325 -0.299723  0.302334
## sit112 0.378471  0.177947 -0.024417  0.163504
## sit113 0.444881 -0.176912 -0.027737  0.410403
## sit114 0.297575  0.490444 -0.611056  0.277939
## sit115 0.346583  0.186627 -1.057168  0.772676
## sit116 0.375495 -0.285358 -0.672325  0.537795
## sit117 0.347477 -0.107866 -0.039419 -0.435393
## sit118 0.572975 -1.078844  0.134671 -0.767485
## sit119 0.781818 -0.007503  0.740646  0.126664
## sit120 0.298437  0.720202  0.281741 -0.182703
## sit121 0.481205 -0.384193 -0.247266  0.470770
## sit122 0.230949  0.241270 -0.872095  0.077799
## sit123 0.684278 -0.174546  0.902945 -0.356926
## sit124 0.314842  0.203312  0.005717  0.392307
## sit125 0.401627 -0.427849 -0.314296 -0.172788
## sit126 0.461512 -0.425256  0.442283 -0.612005
## sit127 0.277500  0.133510 -0.136837  0.351555
## sit128 0.241097 -0.027152 -0.355646 -0.024258
## sit129 0.422317  0.079062 -0.285028  0.087139
## sit130 0.440099 -0.237272  0.753629 -0.583630
## sit131 0.575248 -0.109411  0.766458 -0.050239
## sit132 0.544307 -1.108240  0.519674 -0.593330
## sit133 0.439876  0.075343 -0.372952  0.280329
## sit134 0.263105  0.123606  0.193230 -0.522324
## sit135 0.283963  0.342354  0.173469 -1.372029
## sit136 0.660927 -0.361549  0.571731  0.829359
## sit137 0.372231 -0.450913 -0.996072  0.099803
## sit138 0.317926 -0.178255 -0.190479 -0.603829
## sit139 0.218397 -0.007268 -0.438957  0.014683
## sit140 0.437359 -0.285308  0.013345  0.547135
## sit141 0.475796 -0.259044 -0.451074  0.693853
```

```
## sit142  0.449105 -0.290983 -0.136981  1.316566
## sit143  0.273433  0.294905 -0.558308 -0.113580
## sit144  0.481876 -0.366071 -0.356097  0.126660
## sit145  0.471861 -0.442722 -0.665990  0.599972
## sit146  0.441718 -0.163290 -0.269880  1.091092
## sit147  0.369474  0.378378  0.027771  0.617201
## sit148  0.359223 -0.113540 -0.189745  0.334038
## sit149  0.324183 -0.426723 -0.982952  0.073239
## sit150  0.226858  0.010267 -0.556295 -0.457110

# make some colors to label species
use.col <- rainbow(3)[match(iris$Species,levels(iris$Species))]

# extract scores of samples (px) and biplot vectors (vx)
px <- scores(p1, display="sites")
vx <- scores(p1,display="species")

# make a plot of the flowers in PCA space
plot(px[,1], px[,2], col=use.col, pch=16,
      xlim=c(-3, 3), ylim=c(-3, 3),
      xlab="PC1", ylab="PC2")
segments(0, 0, vx[,1], vx[,2], col="red")
text(vx[,1], vx[,2], rownames(vx), col="red")
legend("topleft", legend=levels(iris$Species),
       pch=16, col=rainbow(3))
```



The ordination reveals that 73% of variation is explained by PC1. The figure shows that the species fall out very cleanly along PC1, which is associated with petal morphology. Let's use PC1 as a predictor for species in a logistic regression.

```
dat2 <- data.frame(y=iris$Species, pc1=px[,1])
dat2$z <- ifelse(dat2$y == "virginica", 1, 0)
mod1 <- glm(z~pc1, data=dat2, family=binomial)
summary(mod1)

##
## Call:
## glm(formula = z ~ pc1, family = binomial, data = dat2)
##
## Deviance Residuals:
##      Min        1Q    Median        3Q       Max
## -1.83252 -0.10909 -0.00016  0.10887  2.83405
##
```

```

## Coefficients:
##              Estimate Std. Error z value Pr(>|z|)
## (Intercept) -6.023     1.393  -4.323 1.54e-05 ***
## pc1         23.366     5.162   4.526 6.00e-06 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
## Null deviance: 190.954  on 149  degrees of freedom
## Residual deviance: 50.108  on 148  degrees of freedom
## AIC: 54.108
##
## Number of Fisher Scoring iterations: 9

```

The logistic regression results suggest that for every unit increase in PC1, the odds ratio of a flower being *Iris virginica* increases by 23. That's a very strong signal. Just for fun, below are the model predictions of probability of being *I. virginica* and the ROC curve. Both confirm visually what the coefficients table above suggested, that PC1 is a very reliable predictor of *Iris* species (at least in this dataset).

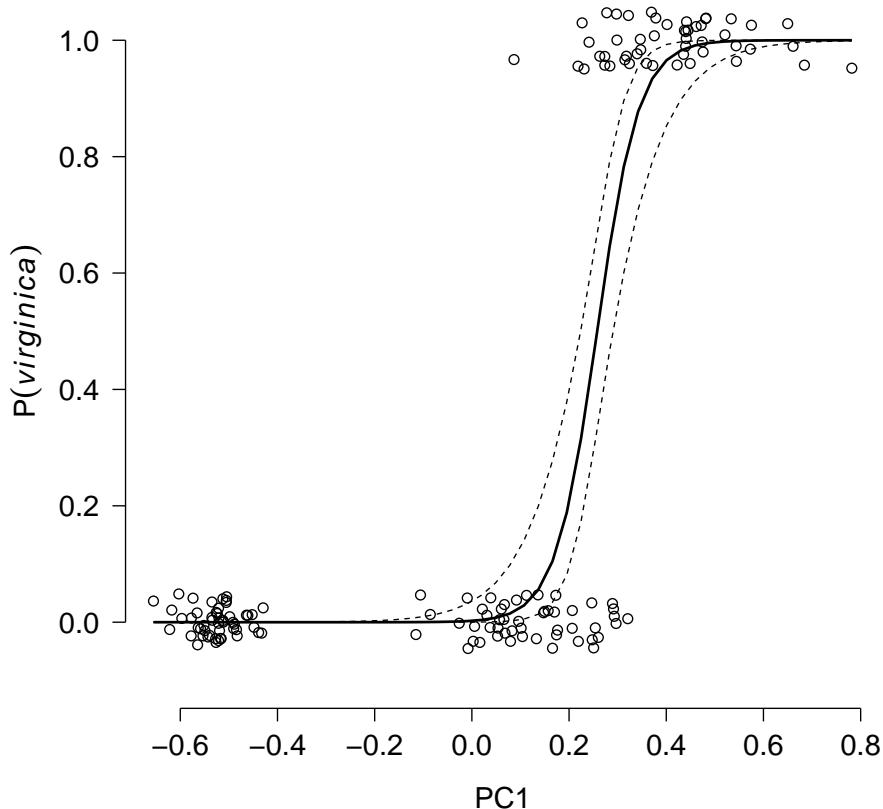
```

n <- 50
prx <- seq(min(dat2$pc1), max(dat2$pc1), length=n)
dx <- data.frame(pc1=prx)
pred <- predict(mod1, newdata=data.frame(dx),
                 type="link", se.fit=TRUE)

mn <- plogis(pred$fit)
ll <- plogis(qnorm(0.025, pred$fit, pred$se.fit))
uu <- plogis(qnorm(0.975, pred$fit, pred$se.fit))

par(mfrow=c(1,1), mar=c(5.1, 5.1, 1.1, 1.1),
    lend=1, las=1, cex.axis=1.3, cex.lab=1.3,
    bty="n")
plot(prx, mn, type="n", xlab="PC1",
      ylab=expression(P(italic(virginica))),
      ylim=c(-0.1,1.1))
points(prx, ll, type="l", lty=2)
points(prx, uu, type="l", lty=2)
points(prx, mn, type="l", lwd=2)
points(dat2$pc1, jitter(dat2$z, amount=0.05), xpd=NA)

```



```

library(pROC)

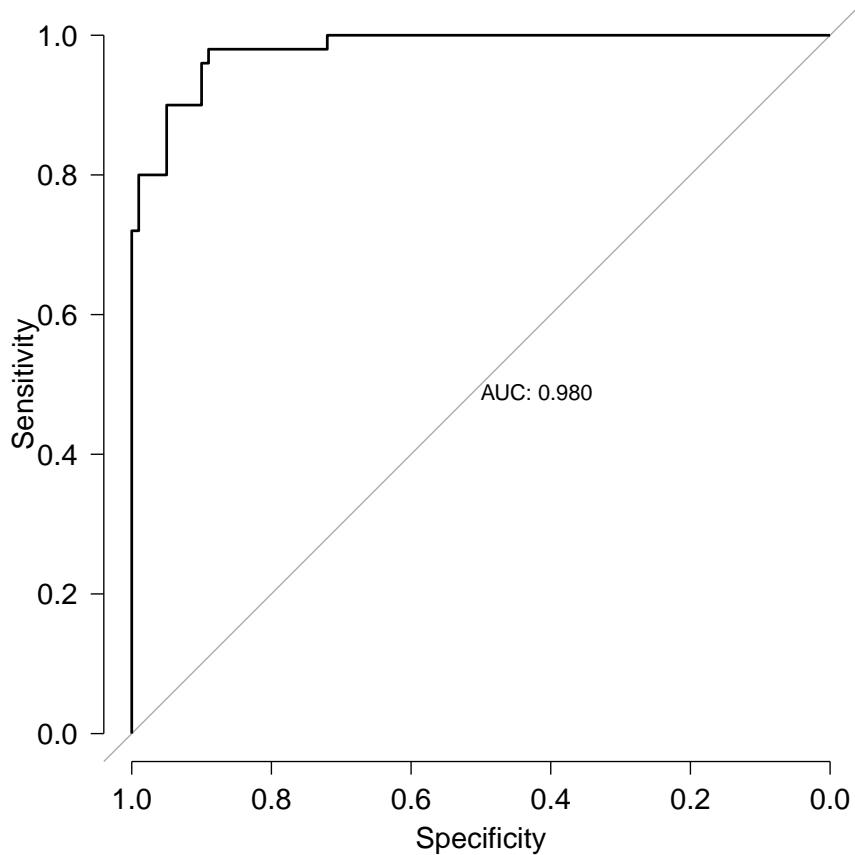
## Type 'citation("pROC")' for a citation.

##
## Attaching package: 'pROC'

## The following objects are masked from 'package:stats':
## 
##      cov, smooth, var
p1 <- predict(mod1, type="response")
roc1 <- roc(dat2$z ~ p1, plot = TRUE, print.auc = TRUE)

## Setting levels: control = 0, case = 1
## Setting direction: controls < cases

```



```
roc1

##
## Call:
## roc.formula(formula = dat2$z ~ p1, plot = TRUE, print.auc = TRUE)
##
## Data: p1 in 100 controls (dat2$z 0) < 50 cases (dat2$z 1).
## Area under the curve: 0.9804
```

Going the other way, modeling PC scores in response to predictor variables, is tricky but it can be done. Usually this is only acceptable when there is a *clear* relationship between the modeled PC and several of the original variables that went into the PCA. Treating a PC as a *dependent variable* is an elegant way to get around the problem of having multiple collinear response variables. Conversely, treating a PC as a predictor variable is a way of dealing with multiple collinear predictor variables. Axes from other ordination techniques can be used in this manner, but require *careful* biological and statistical justification as well

as very cautious interpretation.

8.5.2 NMDS and other ordination methods

PCA and many related techniques are based on linear algebra and eigenvalues. This is fine for datasets where variables are mostly linearly related to each other, or can be transformed to be linearly related. Most of the eigenvalue-based techniques also require data to be (mostly) multivariate normal distributed.

If relationships between variables are non-linear, or if the many other assumptions of eigenvalue-based ordination cannot be met, then the next best option is often a non-parametric ordination technique called **nonmetric multidimensional scaling (NMDS or NMS)**. Unlike PCA, which solves linear algebra problems to extract synthetic gradients (“principal components”), NMDS works by trying iteratively to arrange the samples into a reduced dimensional space that preserves the rank order of the distance matrix.

8.5.2.1 NMDS simple explanation

Let's start with a simple example. Download the dataset state_caps.csv, put it in your R home directory, and load it into R. This dataset contains the latitudes and longitudes of the state capitals of the US. For simplicity, we will use only the lower 48 states.

```
library(vegan)

in.name <- "state_caps.csv"
dat <- read.csv(in.name, header=TRUE)
dat <- dat[which(!dat$state %in% c("Alaska", "Hawaii")),]

# set rownames so the city names will carry through to the
# distance matrix
rownames(dat) <- dat$cap
dat <- dat[,c("long", "lat")]
```

Use function `vegdist()` in package `vegan` to calculate Euclidean distances between the cities¹. Then print the object `d1` to the console and take a look (not shown).

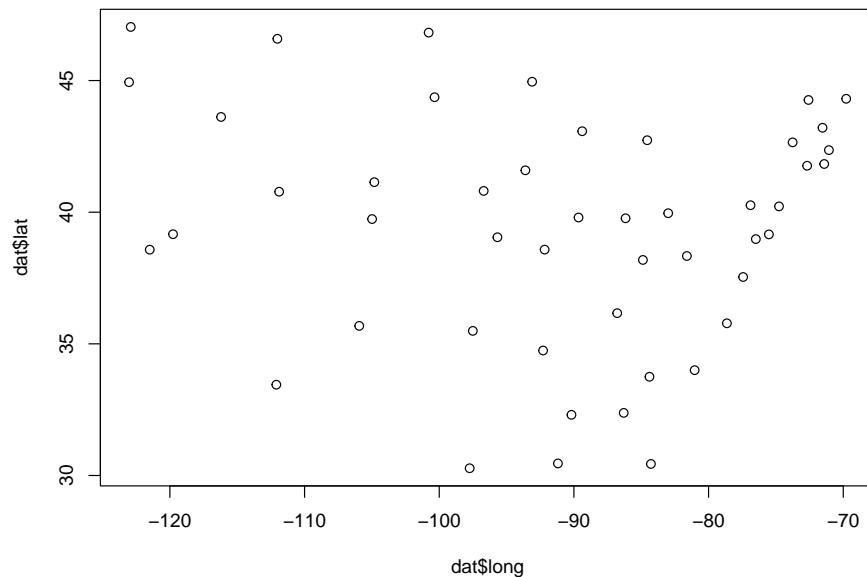
```
d1 <- vegdist(dat[,c("long", "lat")], method="euclidean")
```

¹Because position is expressed in latitude and longitude, it is implicit that these are positions on a sphere. Euclidean distance between coordinates assumes that the coordinates are on a plane, and so will be distorted. The distances are also in nonsensical units that combine degrees latitude and longitude, which are not interchangeable. If you ever need to calculate exact distances between geographic locations, transform the coordinates into a projected coordinate system like UTM (which will be in meters, by definition). Or, use the GIS functions in R which can work in spherical coordinates. Despite being incorrect from a GIS perspective, the distances calculated here are close enough for demonstrating NMDS.

The distance matrix gives the distance between each pair of cities, identified by row labels.

If you plot latitude vs. longitude, you will get a reasonable map of the continental US (CONUS).

```
plot(dat$long, dat$lat)
```



Wouldn't it be neat if we could recover the actual arrangement of capitals shown above just using their distance matrix? It turns out, we can, with NMDS. The main function for NMDS is `metaMDS()` in package `vegan`. NMDS involves random sampling, so set the random number seed for reproducibility.

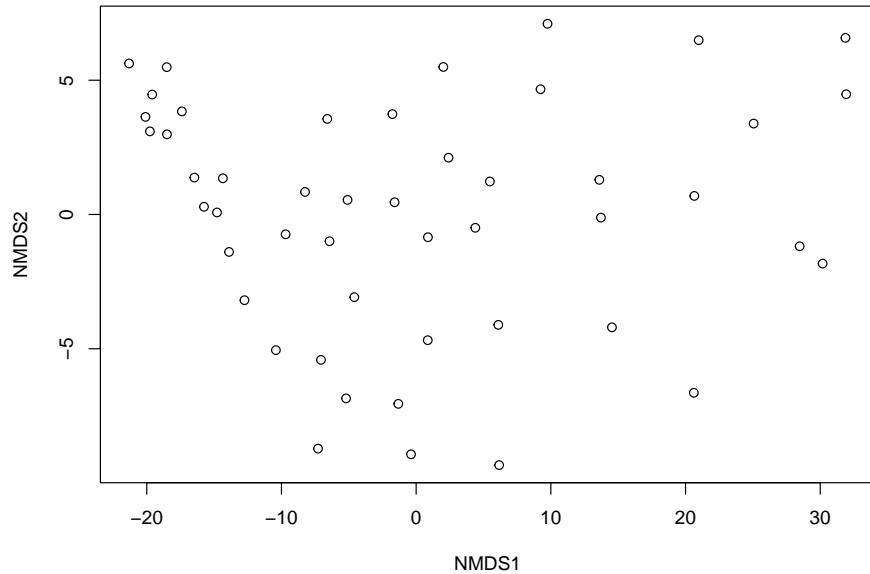
```
set.seed(123)
n1 <- metaMDS(dat, distance="euclidean")

## 'comm' has negative data: 'autotransform', 'noshare' and 'wascores' set to FALSE

## Run 0 stress 0
## Run 1 stress 0.02824041
## Run 2 stress 9.688201e-05
## ... Procrustes: rmse 0.0003258308 max resid 0.0008680344
## ... Similar to previous best
## Run 3 stress 9.741634e-05
## ... Procrustes: rmse 0.0003391501 max resid 0.0009861631
## ... Similar to previous best
```

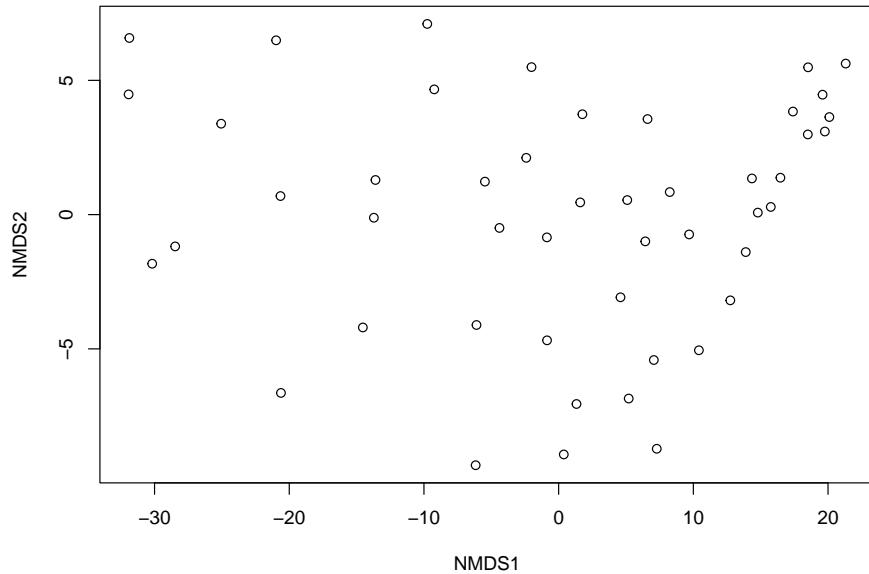
```
## Run 4 stress 9.614493e-05
## ... Procrustes: rmse 0.0003316108 max resid 0.00101616
## ... Similar to previous best
## Run 5 stress 9.839223e-05
## ... Procrustes: rmse 0.0003366584 max resid 0.001019179
## ... Similar to previous best
## Run 6 stress 9.938774e-05
## ... Procrustes: rmse 0.0003421181 max resid 0.001056077
## ... Similar to previous best
## Run 7 stress 9.747501e-05
## ... Procrustes: rmse 0.0003420786 max resid 0.001042044
## ... Similar to previous best
## Run 8 stress 9.688222e-05
## ... Procrustes: rmse 0.0003407193 max resid 0.001040716
## ... Similar to previous best
## Run 9 stress 9.042907e-05
## ... Procrustes: rmse 0.0003277819 max resid 0.0009954318
## ... Similar to previous best
## Run 10 stress 9.585724e-05
## ... Procrustes: rmse 0.0003388215 max resid 0.001045013
## ... Similar to previous best
## Run 11 stress 9.623688e-05
## ... Procrustes: rmse 0.0003381601 max resid 0.001029097
## ... Similar to previous best
## Run 12 stress 0.04810304
## Run 13 stress 9.271431e-05
## ... Procrustes: rmse 0.0003274582 max resid 0.000996862
## ... Similar to previous best
## Run 14 stress 9.821812e-05
## ... Procrustes: rmse 0.0003434694 max resid 0.001046526
## ... Similar to previous best
## Run 15 stress 9.886879e-05
## ... Procrustes: rmse 0.000344009 max resid 0.001069766
## ... Similar to previous best
## Run 16 stress 9.538052e-05
## ... Procrustes: rmse 0.0003364742 max resid 0.001031398
## ... Similar to previous best
## Run 17 stress 0.02824038
## Run 18 stress 9.633286e-05
## ... Procrustes: rmse 0.0003317048 max resid 0.000937519
## ... Similar to previous best
## Run 19 stress 0.02824036
## Run 20 stress 9.744164e-05
## ... Procrustes: rmse 0.0003414834 max resid 0.001047844
## ... Similar to previous best
## *** Solution reached
```

```
## Warning in metaMDS(dat, distance = "euclidean"): stress is (nearly) zero: you
## may have insufficient data
plot(scores(n1))
```



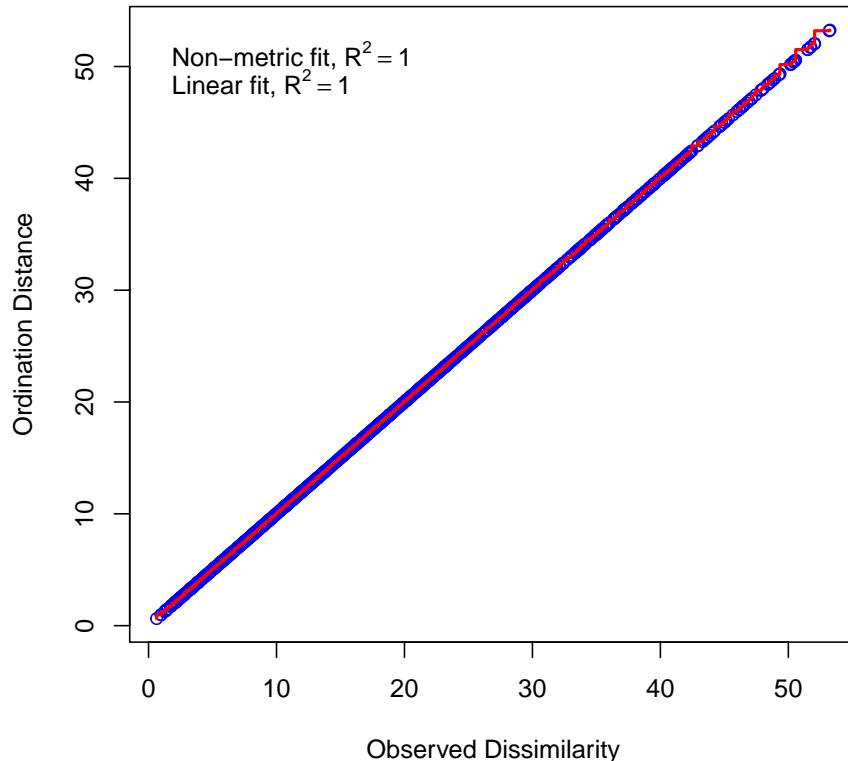
Notice anything odd? The *arrangement* of the cities, or relative positions, are mostly correct. However, the cities are flipped east to west! This is because NMDS only considers relative position, not absolute position. NMDS positions are arbitrary, so we can rotate or flip the axes to make better sense of them. We'll flip the first axis (NMDS1) by multiplying it by -1.

```
sn <- scores(n1)
sn[,1] <- -1*sn[,1]
plot(sn)
```



Much better. The coordinates aren't quite the same as the original data, but the relative positions of the cities are correct. We can confirm this with a stress plot.

```
stressplot(n1)
```



This chart shows that the distances between points in the NMDS space (Ordination Distance) are nearly identical to the distances between points in the original data space (Observed Dissimilarity). A good NMDS fit will have nearly all of the points on the red line of equality.

8.5.2.2 NMDS more complicated explanation

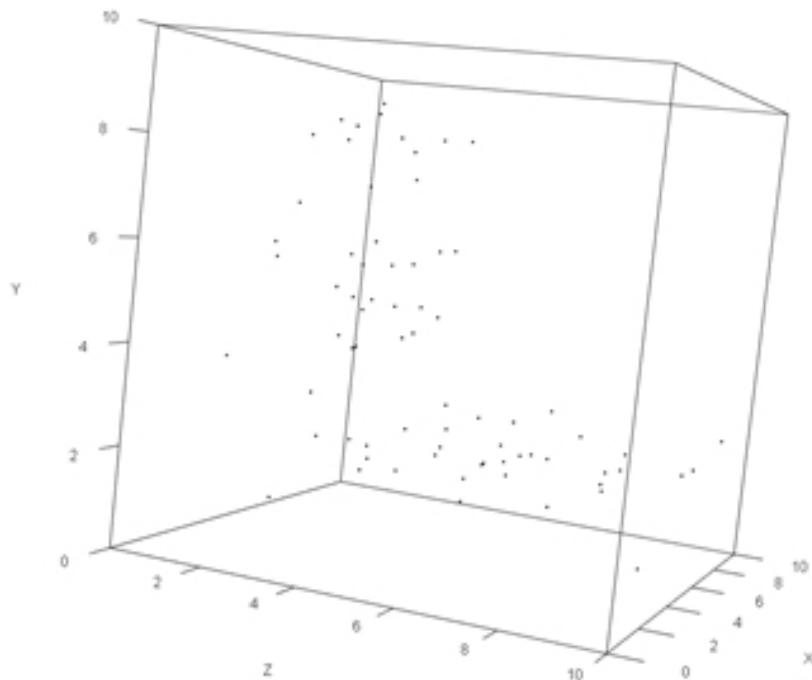
Like PCA, NMDS is a technique for reducing the dimensionality of data. The general approach of NMDS is to start with a cloud of n samples (points) in p dimensions (variables), and try to position the points into $< p$ dimensions in a way that preserves the relative distances between the points. Consider the example below, which shows a 3-d dataset with variables X , Y , and Z .

```
set.seed(123)
n <- 36
x <- runif(n, 1, 10)
y <- runif(n, 1, 3)
```

```
z <- runif(n, 1, 10)
```

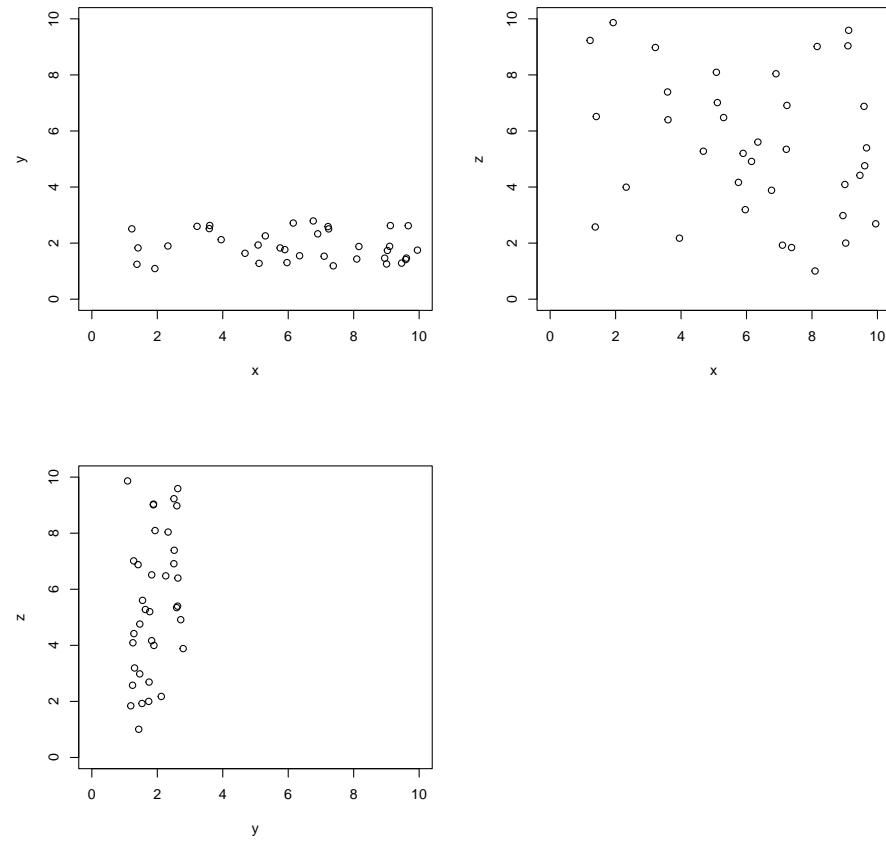
The code below was not run for this tutorial, but a screenshot of the result is shown below.

```
library(rgl)
par(mfrow=c(1,1))
plot3d(x, y, z,
       xlab="X", ylab="Y", zlab="Z",
       xlim=c(0,10), ylim=c(0,10), zlim=c(0,10))
```



Or more prosaically:

```
par(mfrow=c(2,2))
plot(x, y, xlim=c(0, 10), ylim=c(0, 10))
plot(x, z, xlim=c(0, 10), ylim=c(0, 10))
plot(y,z, xlim=c(0, 10), ylim=c(0, 10))
```



The scatterplots suggest that most of the variation is tied up in X and Z , with very little in Y . If we wanted to present the data in two dimensions, we could use NMDS to arrange the points in a 2-d space that preserves the relative distances. Data can be passed directly to `metaMDS()`, in which case it will calculate the distance matrix for you. Alternatively, you can supply a distance matrix. The function will figure out what to do based on what your input.

As always, set the random number seed for reproducibility.

```
set.seed(42)
dat <- cbind(x,y,z)
n2 <- metaMDS(dat)

## Wisconsin double standardization
## Run 0 stress 0.02846602
## Run 1 stress 0.02846602
## ... Procrustes: rmse 4.316137e-06 max resid 1.726271e-05
```

```
## ... Similar to previous best
## Run 2 stress 0.02846602
## ... Procrustes: rmse 1.118155e-05 max resid 4.018889e-05
## ... Similar to previous best
## Run 3 stress 0.02846602
## ... New best solution
## ... Procrustes: rmse 6.792159e-06 max resid 2.56478e-05
## ... Similar to previous best
## Run 4 stress 0.02846602
## ... Procrustes: rmse 7.699264e-06 max resid 2.638313e-05
## ... Similar to previous best
## Run 5 stress 0.02846602
## ... Procrustes: rmse 9.363497e-06 max resid 3.897864e-05
## ... Similar to previous best
## Run 6 stress 0.02846602
## ... New best solution
## ... Procrustes: rmse 2.322942e-06 max resid 7.365495e-06
## ... Similar to previous best
## Run 7 stress 0.02846602
## ... Procrustes: rmse 3.750166e-06 max resid 1.250864e-05
## ... Similar to previous best
## Run 8 stress 0.02846602
## ... Procrustes: rmse 6.108855e-06 max resid 2.128185e-05
## ... Similar to previous best
## Run 9 stress 0.02846602
## ... Procrustes: rmse 4.354076e-06 max resid 1.548638e-05
## ... Similar to previous best
## Run 10 stress 0.02846602
## ... Procrustes: rmse 1.755597e-06 max resid 5.736735e-06
## ... Similar to previous best
## Run 11 stress 0.02846602
## ... Procrustes: rmse 5.436306e-06 max resid 2.010142e-05
## ... Similar to previous best
## Run 12 stress 0.02846602
## ... Procrustes: rmse 1.012847e-06 max resid 3.324696e-06
## ... Similar to previous best
## Run 13 stress 0.02846602
## ... Procrustes: rmse 4.858257e-06 max resid 1.845511e-05
## ... Similar to previous best
## Run 14 stress 0.11564
## Run 15 stress 0.11564
## Run 16 stress 0.02846602
## ... Procrustes: rmse 3.74378e-06 max resid 1.643431e-05
## ... Similar to previous best
## Run 17 stress 0.02846602
## ... Procrustes: rmse 3.719871e-06 max resid 1.260728e-05
```

```

## ... Similar to previous best
## Run 18 stress 0.02846602
## ... Procrustes: rmse 5.596345e-06 max resid 2.38461e-05
## ... Similar to previous best
## Run 19 stress 0.11564
## Run 20 stress 0.11564
## *** Solution reached

n2

##
## Call:
## metaMDS(comm = dat)
##
## global Multidimensional Scaling using monoMDS
##
## Data:      wisconsin(dat)
## Distance: bray
##
## Dimensions: 2
## Stress:     0.02846602
## Stress type 1, weak ties
## Two convergent solutions found after 20 tries
## Scaling: centring, PC rotation, halfchange scaling
## Species: expanded scores based on 'wisconsin(dat)'

```

The output tells us a lot about the fit. We got a 2-d fit (the default), used the Bray-Curtis distance metric (the default), and variables were centered, scaled, and rotated (also default). The rotation referred to here means that the final arrangement of points was rotated so that NMDS axis 1 captured as much variation as possible (aka: “varimax” rotation; McCune et al. (2002)).

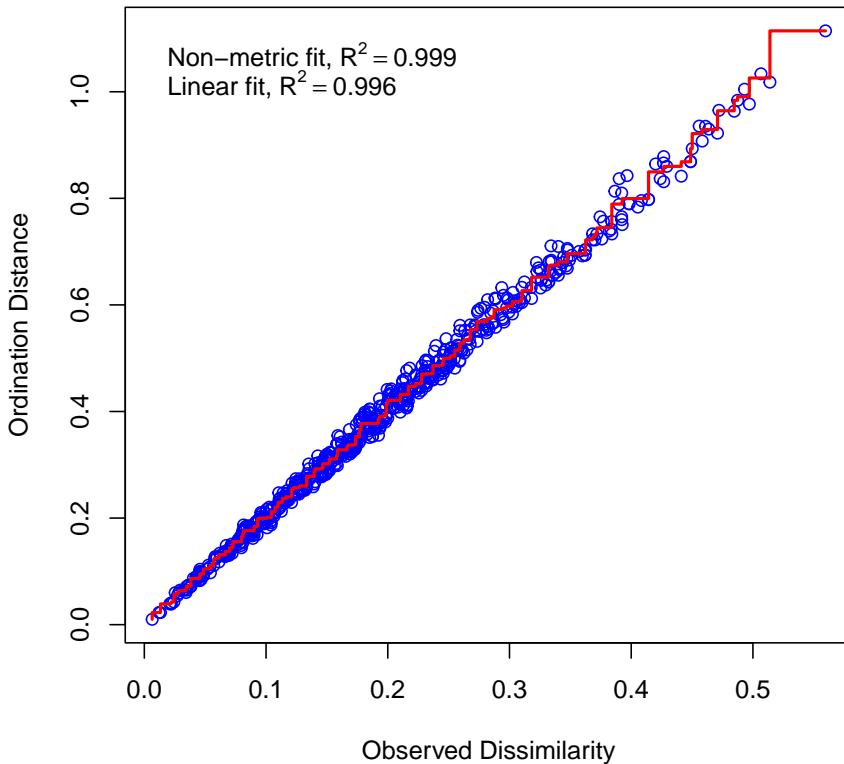
The **stress** expresses how well the NMDS represented the **rank order of the distance matrix**. Lower stress indicates a more faithful representation. Stress increases as distances between samples are increased or decreased relative to the same distances in the original, full-dimensional space. If you picture the full-dimensional distances as ropes connecting the samples, and the amount of stretching or coiling of the ropes that happens as the samples are squashed into lower-dimensional space as the stress, then you pretty much have the idea.

There is no hard and fast rule, but in practice stress values should be >0 and <0.2 . Some authors suggest that NMDS fits with stress ≥ 0.2 are suspect and should not be interpreted; other authors use 0.1 as the cutoff. Some authors and software packages scale stress by 100, so the cutoff becomes 20 or 10 instead of 0.2 or 0.1 (McCune et al. 2002). Like P -values, stress values are a heuristic for inference and so you should not get too hung up on them. The stress plot and the screeplot are arguably more meaningful evaluations of your NMDS fit.

The stressplot below shows that the NMDS did a very good job representing the

distance matrix in the reduced space, because the points are clustered near the line and the R^2 values very high.

```
par(mfrow=c(1,1))
stressplot(n2)
```

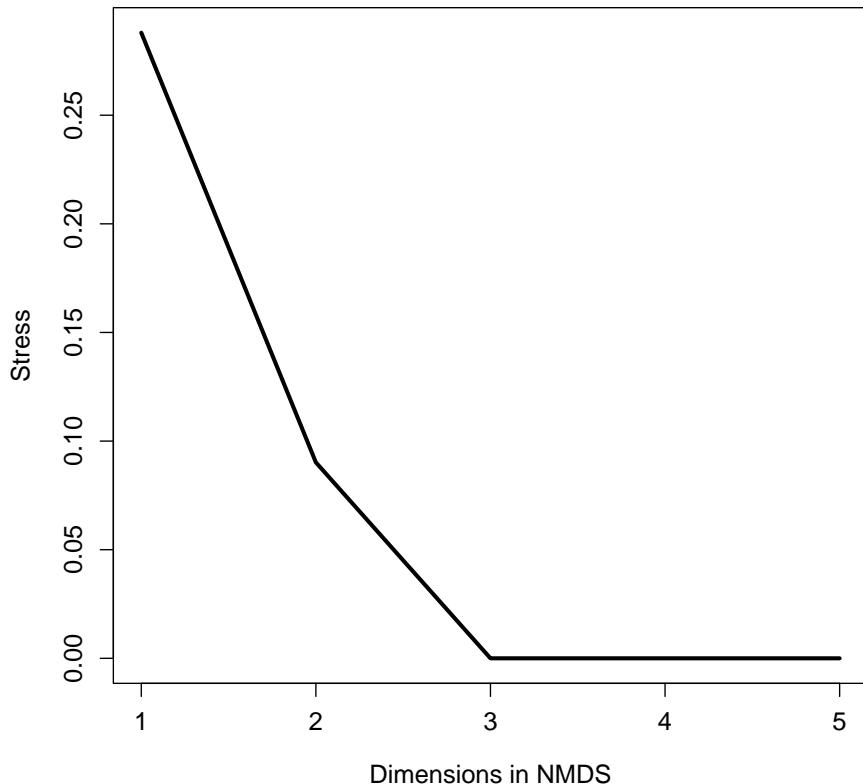


The **screeplot** shows how the stress of the ordination changes as more dimensions are added. An NMDS ordination with the same number of dimensions as the original data would have stress 0. Generally, adding dimensions decreases stress, but there is usually a point of diminishing returns.

For the example below, I added a few more dimensions to the dataset `dat` with no information so that a better screeplot could be constructed. You don't need to do this to your own data—the only reason it's being done here is because our dataset only had 3 dimensions to start with. With your data, you should fit NMDS ordinations with more dimensions than you think you need, to see how stress decreases with dimensionality.

```
# extra dimensions with random numbers (no information)
ext <- matrix(runif(n*1, 0, 0.1), nrow=n, ncol=2)
# combine with original data
dat2 <- cbind(dat, ext)
# fit NMDS ordinations in a loop
# for each element of nlist, k = number of dimensions
nlist <- vector("list", ncol(dat2))
for(i in 1:length(nlist)){
  nlist[[i]] <- metaMDS(dat2, distance="euclidean", k=i)
}
# extract stress values for each NMDS fit
strs <- sapply(nlist, function(x){x$stress})

# fancy plot
par(mfrow=c(1,1), cex.lab=1.2, cex.axis=1.2)
plot(1:length(nlist), strs, type="l", lwd=3,
  xlab="Dimensions in NMDS",
  ylab="Stress")
```



The screeplot shows that the stress decreases much more when going from 1 to 2 dimensions than it does for going from 2 to 3 dimensions. Stress becomes negligible for ≥ 3 dimensions, because the original data had 3 dimensions and the two additional dimensions had almost 0 variance. The bend in the curve at $k = 2$ indicates that 2 is probably the optimal number of dimensions. If you're not sure how many dimensions to use, examine biplots for both k and try to determine which plot is more interpretable biologically.

We can plot the scores of the samples much in the same way as we did the PCA scores in the previous example. We can also produce a biplot that works much the same way as the PCA biplot. We'll explore those methods in the next section.

8.5.2.3 NMDS real data example

Load the `dune` dataset that comes with package `vegan`. This dataset contains cover class values for 30 species of plants on 20 dune meadow sites in the

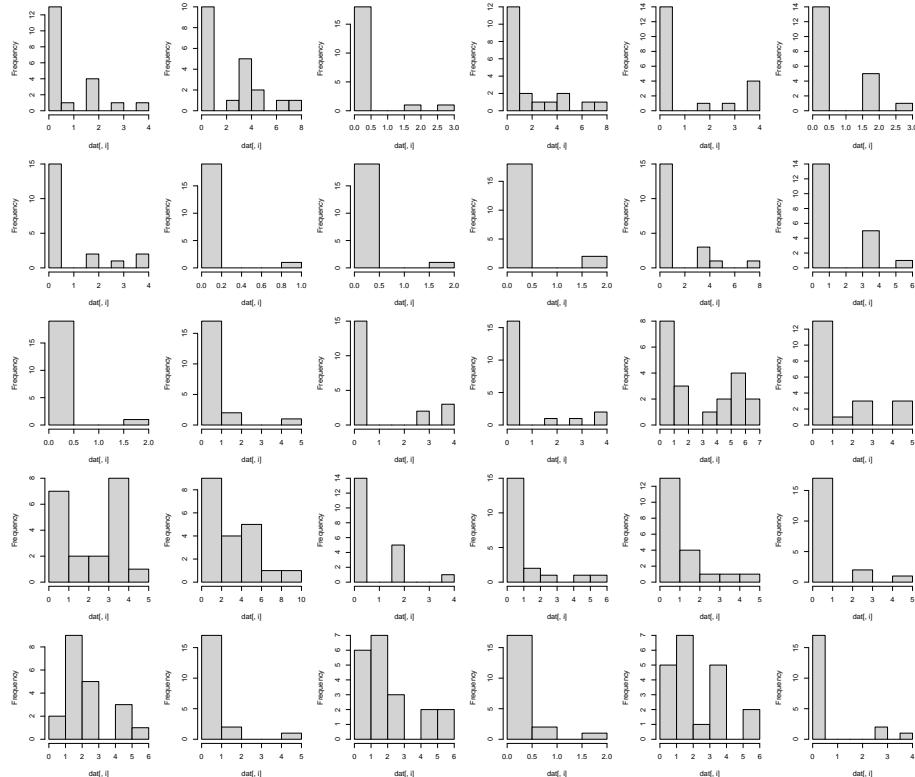
Netherlands. We'll also load `dune.env`, the environmental data that describe each site.

```
library(vegan)
data(dune)
data(dune.env)

dat <- dune
env <- dune.env
```

If we are interested in plant diversity among the sites, we could use multivariate methods to describe variation in plant cover. First, make some histograms of the cover of each plant species.

```
par(mfrow=c(5, 6), mar=c(4.1, 4.1, 1.1, 1.1))
for(i in 1:ncol(dat)){hist(dat[,i], main="")}
```



The cover of most species is clearly non-normal. This means that PCA would be difficult to fit. What's more, the response variable (canopy cover class) isn't really quantitative. Instead, it is ordinal. For example, category 2 represents greater proportion of plant cover than category 1, but it's not clear how much more. So, an analysis based on rank order might make more sense than one based on actual

values. That suggests using NMDS instead of PCA. Use `vegan::metaMDS()` to fit the NMDS, using the default Bray-Curtis distance metric.

```
set.seed(456)
p1 <- metaMDS(dat)

## Run 0 stress 0.1192678
## Run 1 stress 0.1183186
## ... New best solution
## ... Procrustes: rmse 0.02027121 max resid 0.06496598
## Run 2 stress 0.1900911
## Run 3 stress 0.1192679
## Run 4 stress 0.19015
## Run 5 stress 0.1192678
## Run 6 stress 0.1183186
## ... New best solution
## ... Procrustes: rmse 9.438674e-06 max resid 3.090521e-05
## ... Similar to previous best
## Run 7 stress 0.1183186
## ... Procrustes: rmse 2.811111e-06 max resid 7.842694e-06
## ... Similar to previous best
## Run 8 stress 0.1192679
## Run 9 stress 0.1192678
## Run 10 stress 0.1192678
## Run 11 stress 0.1812933
## Run 12 stress 0.1889647
## Run 13 stress 0.1192679
## Run 14 stress 0.1192678
## Run 15 stress 0.1183186
## ... Procrustes: rmse 1.256982e-05 max resid 3.657271e-05
## ... Similar to previous best
## Run 16 stress 0.1192679
## Run 17 stress 0.1192678
## Run 18 stress 0.1192678
## Run 19 stress 0.1192678
## Run 20 stress 0.192224
## *** Solution reached

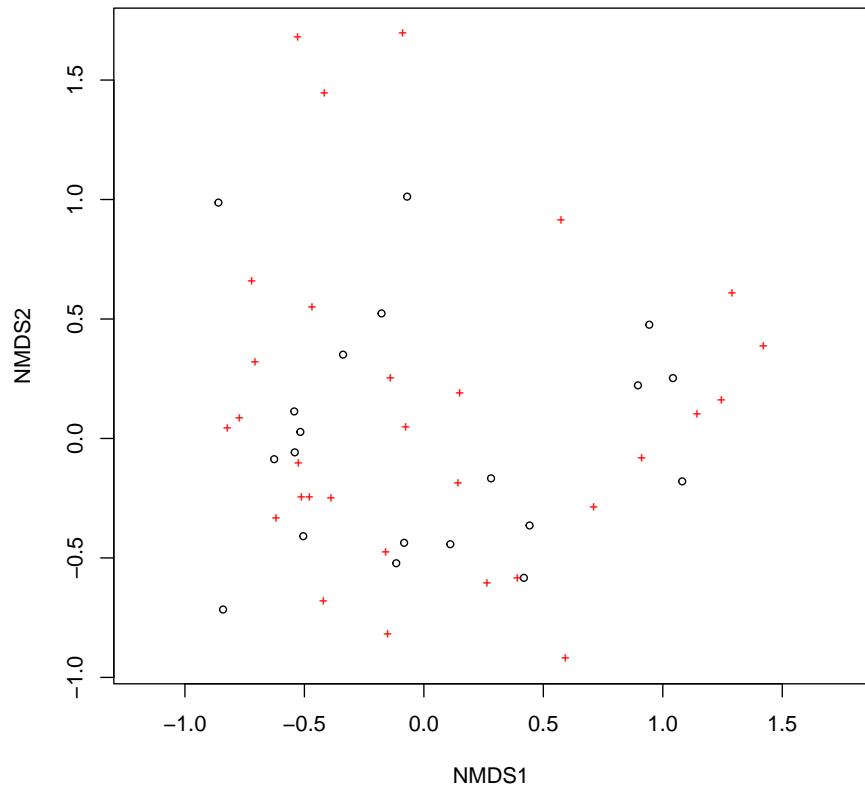
p1

##
## Call:
## metaMDS(comm = dat)
##
## global Multidimensional Scaling using monoMDS
##
## Data:      dat
```

```
## Distance: bray
##
## Dimensions: 2
## Stress: 0.1183186
## Stress type 1, weak ties
## Two convergent solutions found after 20 tries
## Scaling: centring, PC rotation, halfchange scaling
## Species: expanded scores based on 'dat'
```

The NMDS fit has stress = 0.118, which is not too bad. Let's take a look at the ordination.

```
plot(p1)
```



The default NMDS biplot is not very informative, so let's build the plot manually. While we're at it, let's use the environmental dataset to assign some category labels.

```
# management regimes
mans <- sort(unique(env$Management))
nman <- length(mans)

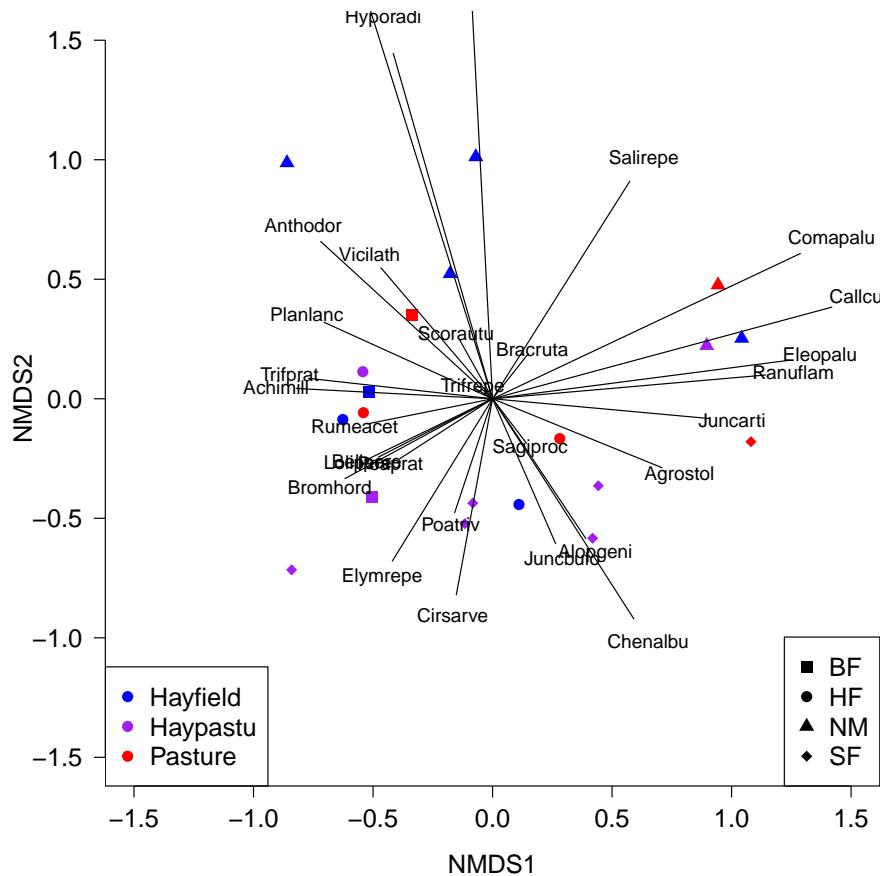
# land uses
uses <- sort(unique(env$Use))
nuse <- length(uses)

# colors and symbols
cols <- c("blue", "purple", "red")
pchs <- 15:18

env$col <- cols[match(env$Use, uses)]
env$pch <- pchs[match(env$Management, mans)]

# get site and species scores
px <- scores(p1)
vx <- scores(p1, display="species")

# fancy plot
par(mfrow=c(1,1), mar=c(5.1, 5.1, 1.1, 1.1),
     bty="n", las=1, lend=1,
     cex.axis=1.3, cex.lab=1.3)
plot(px, pch=env$pch, col=env$col, cex=1.3,
      xlim=c(-1.5, 1.5), ylim=c(-1.5, 1.5))
segments(0, 0, vx[,1], vx[,2])
mult <- 1.1
text(mult*vx[,1], mult*vx[,2], rownames(vx))
legend("bottomright", legend=mans, pch=pchs, cex=1.3)
legend("bottomleft", legend=uses, col=cols, pch=16, cex=1.3)
```



There's a lot of information in that figure, so let's break it down:

- Points show samples (sites).
- Proximity of two points reflects their dissimilarity in terms of the plant variables (closer = more similar).
- Point color encodes land use (hayfield, pasture, or both).
- Point shape encodes land management: BF (Biological farming), HF (Hobby farming), NM (Nature Conservation Management), and SF (Standard Farming).
- Vectors show how the species are associated with the ordination space. Species cover increases in the direction of the vector. For example, species "Juncarti" increases to the right and decreases to the left.
- Magnitude of each vector indicates the strength of the association. Vectors that are parallel to one of the NMDS axes are particularly interesting because they describe important gradients among the sites.

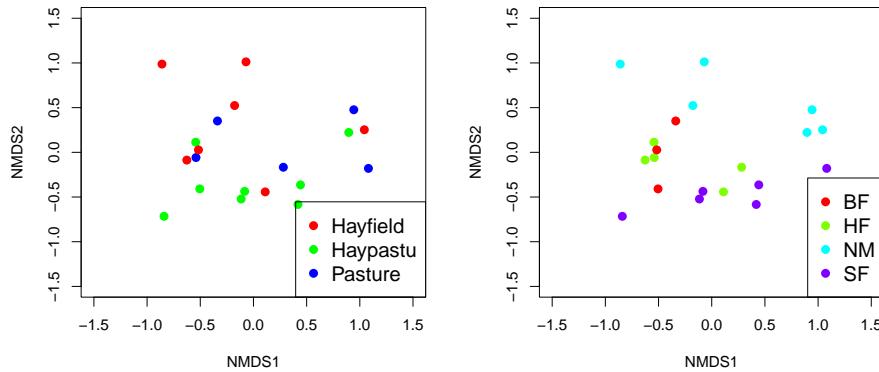
The figure is a bit of a mess, so we should try some different representations to

help us make sense of it. Code one factor at a time:

```
par(mfrow=c(1,2))
cols1 <- rainbow(nuse)
cols1x <- cols1[match(env$Use, uses)]
plot(px, pch=16, col=cols1x, cex=1.3,
      xlim=c(-1.5, 1.5), ylim=c(-1.5, 1.5))
legend("bottomright", legend=uses,
       pch=16, col=cols1, cex=1.3)

cols2 <- rainbow(nman)
cols2x <- cols2[match(env$Management, mans)]

plot(px, pch=16, col=cols2x, cex=1.3,
      xlim=c(-1.5, 1.5), ylim=c(-1.5, 1.5))
legend("bottomright", legend=mans,
       pch=16, col=cols2, cex=1.3)
```



When only one factor is plotted, it's a little easier to see the potential grouping by management style in the right panel. Styles SF, HF, and NM fall out mostly along NMDS2, but BF does not separate cleanly from the others. Let's investigate the differences related to management styles.

We can add ellipses to help illustrate group centers (aka: “centroids”, or average positions in NMDS space) and the 95% CI of the centroids.

```
# define some colors
cols <- rainbow(nman)
colsx <- cols2[match(env$Management, mans)]

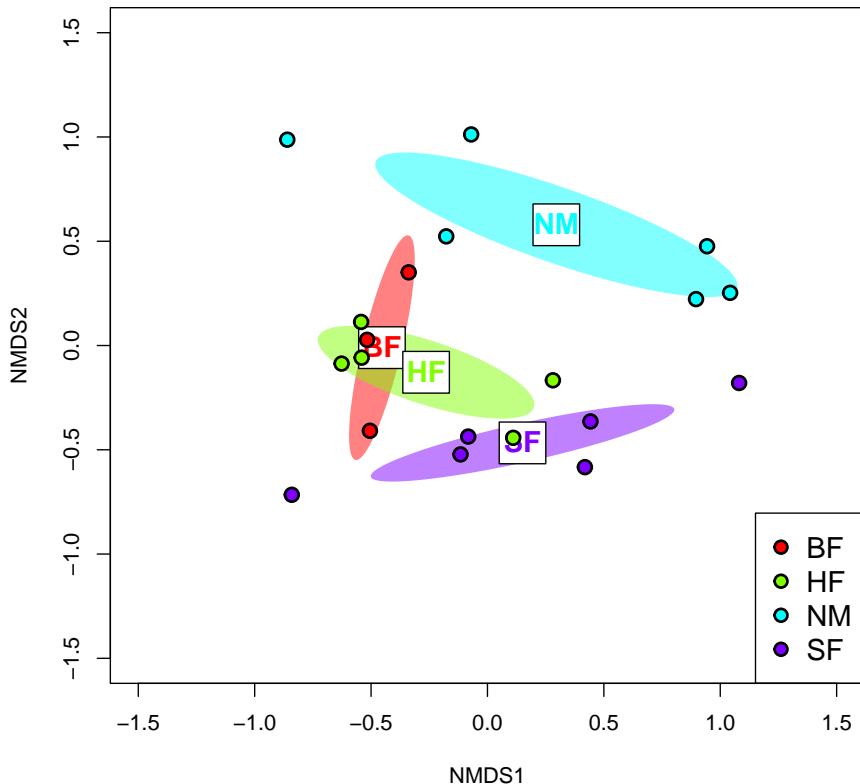
# make plot
par(mfrow=c(1,1))
```

```
plot(px, pch=16, col=colsx, cex=1.3,
      xlim=c(-1.5, 1.5), ylim=c(-1.5, 1.5))
# add ellipse
ordiellipse(p1, env$Management,
            kind="se", conf=0.95,
            draw="polygon", col=cols, border=NA)

# plot group labels with white box behind at centroids
centx <- aggregate(p1$points[,1],
                     by=list(env$Management), FUN=mean)$x
centy <- aggregate(p1$points[,2],
                     by=list(env$Management), FUN=mean)$x

box.len <- 0.1 # fiddle with this to get boxes right size
rect(centx-box.len, centy-box.len,
      centx+box.len, centy+box.len,
      col="white", border="black")
text(centx, centy, mans, cex=1.3,
     col=cols, font=2, adj=c(0.5, 0.45))

# add points again to make sure they show up
points(px, pch=21, bg=colsx, fg="black", lwd=2, cex=1.4)
legend("bottomright", legend=mans,
       pch=21, pt.bg=cols, col="black", pt.lwd=2, cex=1.3)
```



The default colors produced by `rainbow()` can be horrendous, but you get the idea. There are better ways to get color palettes that you can explore. The figure shows that it is quite likely that at least some of the management groups differ from each other in terms of their plant cover. We can test this directly with MRPP.

```
mrpp(px, env$Management)

##
## Call:
## mrpp(dat = px, grouping = env$Management)
##
## Dissimilarity index: euclidean
## Weights for groups: n
##
## Class means and counts:
##
```

```

##      BF      HF      NM      SF
## delta 0.5279 0.5845 1.055 0.8339
## n      3       5       6       6
##
## Chance corrected within-group agreement A: 0.2216
## Based on observed delta 0.792 and expected delta 1.017
##
## Significance of delta: 0.004
## Permutation: free
## Number of permutations: 999

```

The MRPP shows significant separation between the management groups, but the relatively low A indicates that there is still a lot of the positioning left unexplained.

We can also use NMDS to test for relationships between the NMDS coordinates and continuous variables using `envfit()` (from the `vegan` package). Note that only columns 1, 2, and 5 are used in the routine because those are the columns that contain quantitative (or ordinal) potential predictors.

```

ef <- envfit(p1, env[,c(1,2,5)])
ef

##
## ***VECTORS
##
##      NMDS1      NMDS2      r2 Pr(>r)
## A1 0.96474 0.26322 0.3649  0.014 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
## Permutation: free
## Number of permutations: 999
##
## ***FACTORS:
##
## Centroids:
##      NMDS1      NMDS2
## Moisture1 -0.5101 -0.0403
## Moisture2 -0.3938  0.0139
## Moisture4  0.2765 -0.4033
## Moisture5  0.6561  0.1476
## Manure0   0.2958  0.5790
## Manure1   -0.2482 -0.0215
## Manure2   -0.3079 -0.1866
## Manure3   0.3101 -0.2470
## Manure4   -0.3463 -0.5583
##

```

```

## Goodness of fit:
##                  r2 Pr(>r)
## Moisture  0.5014  0.002 **
## Manure    0.4247  0.018 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
## Permutation: free
## Number of permutations: 999

```

The output shows that all three of A1 soil horizon thickness (A1), moisture category, and manure category were significantly related to the NMDS configuration. The plots below show some ways to illustrate these relationships. First, we can add vectors representing continuous variables to the ordination. These vectors are exactly like biplot arrows for variables that went into the ordination, and are interpreted the same way . If you include both ordinated variables and other variables fitted with `envfit()`, you should make them different colors.

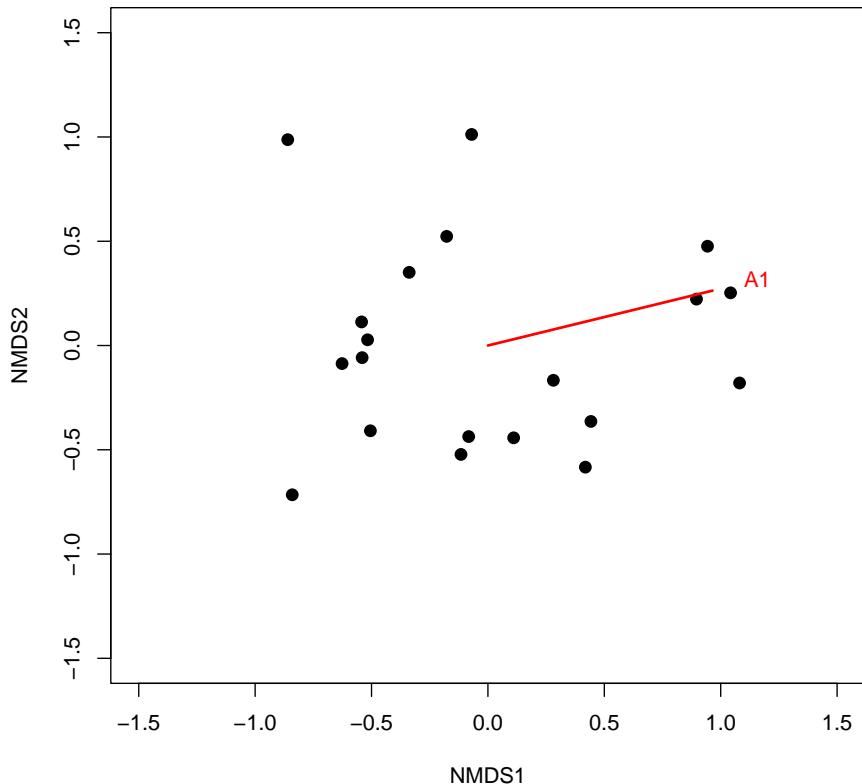
```

# get coordinates of vectors (matrix)
ef.vec <- ef$vectors$arrows

# plot points
par(mfrow=c(1,1))
plot(px, pch=16, cex=1.3,
      xlim=c(-1.5, 1.5), ylim=c(-1.5, 1.5))

# add envfit vector
segments(0, 0, ef.vec[1,1], ef.vec[1,2], col="red", lwd=2)
# multiplier to move label outwards (should be >1)
mult <- 1.2
text(mult*ef.vec[1,1], mult*ef.vec[1,2],
      rownames(ef.vec)[1], col="red")

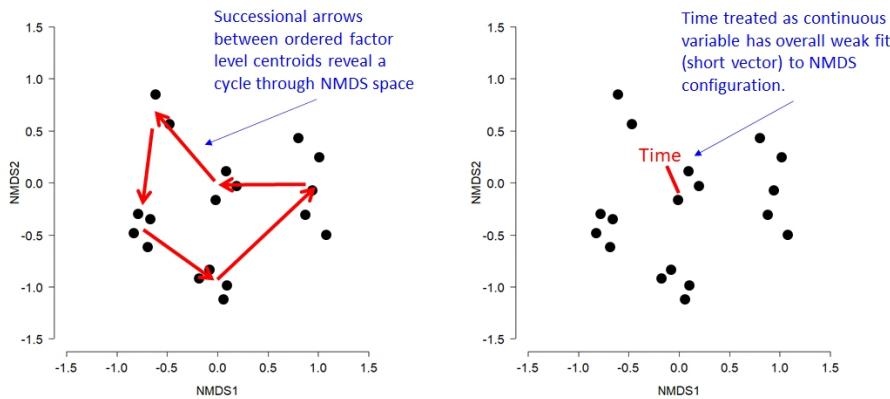
```



```
# not run:
# if more than one continuous envfit to plot:
#mult <- 1.2
#for(i in 1:nrow(ef.vec)){
#  segments(0, 0, ef.vec[i,1], ef.vec[i,2],
#          col="red", lwd=2)
#  text(mult*ef.vec[i,1], mult*ef.vec[i,2],
#       rownames(ef.vec)[i], col="red")
#}
```

Factors that define groups, like the farming types above, can be encoded in the points by shape and/or color, and their uncertainty expressed with ellipses. Factors that are ordered, like Moisture and Manure, can be added to the plot as arrows. Adding arrows to connect centroids of the levels of an ordered factor is also a great way to show changes over time or space. For example, you could use season or year as an ordered factor to show changes over time; or, use latitude or elevation intervals as an ordered factor. Even if one of these variables

is treated continuously, showing arrows by ordered groups (levels) can help capture nonmonotonic relationships that would be masked by a single `envfit()` vector. The figure below illustrates the difference between treating a continuous predictor, time, as an ordered factor vs. as a continuous variable.



The code below shows how to extract the coordinates for the arrows showing levels of an ordered factor.

```
# get matrix of level centroids (has all factors)
ef.fac <- ef$fac$centroids

# isolate moisture and manure rows of matrix
ef.moi <- ef.fac[grep("Moist", rownames(ef.fac)),]
ef.man <- ef.fac[grep("Manure", rownames(ef.fac)),]

# number of levels of each factor
nmois <- nrow(ef.moi)
nmanu <- nrow(ef.man)

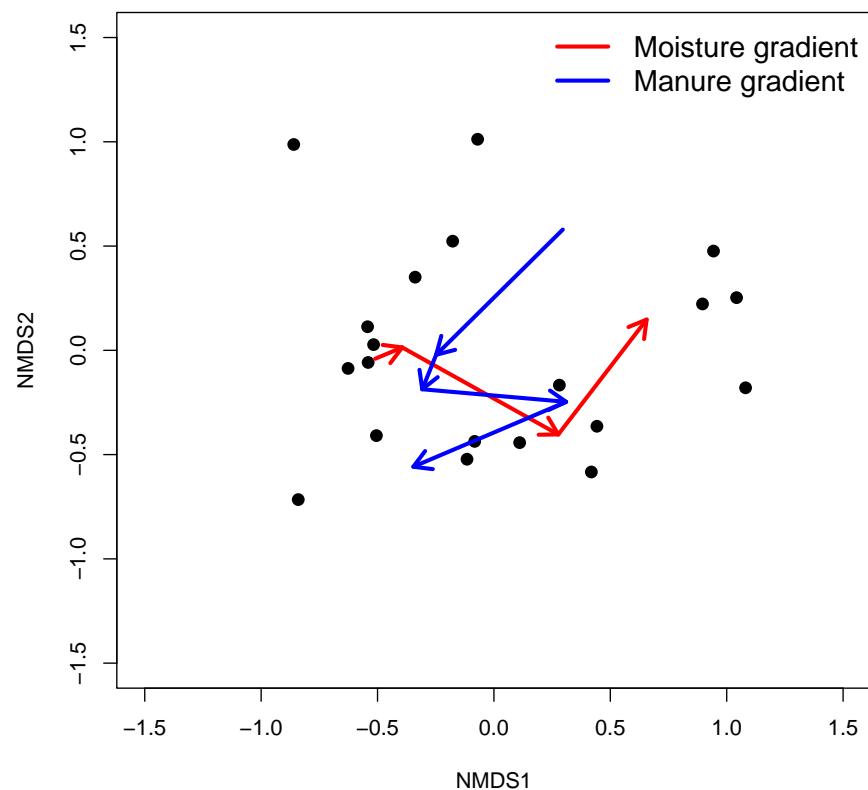
par(mfrow=c(1,1))
plot(px, pch=16, cex=1.3,
      xlim=c(-1.5, 1.5), ylim=c(-1.5, 1.5))

# add arrows
# notice how arrows are specified from the matrix as
# beginning x, beginning y, ending x, and ending y, and
# how bracket notation is used.
arrows(ef.moi[-nmois,1], ef.moi[-nmois,2],
       ef.moi[-1, 1], ef.moi[-1, 2], col="red",
       lwd=3, length=0.15)
arrows(ef.man[-nmanu ,1], ef.man[-nmanu ,2],
       ef.man[-1, 1], ef.man[-1, 2], col="blue",
```

```

lwd=3, length=0.15)
legend("topright",
       legend=c("Moisture gradient", "Manure gradient"),
       lwd=3, col=c("red", "blue"), bty="n", cex=1.3)

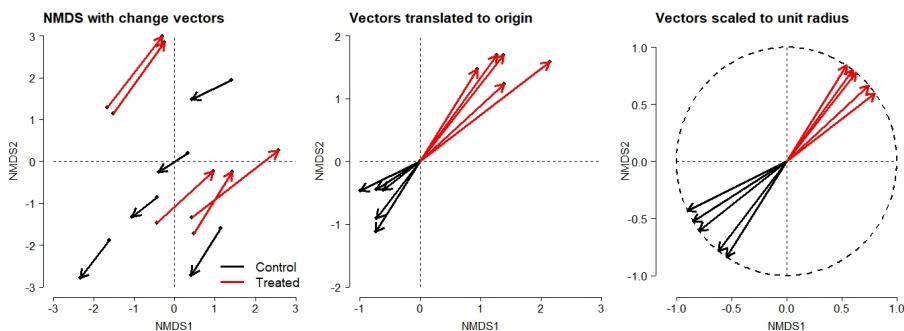
```



8.5.2.4 Hypothesis testing using NMDS

NMDS ordinations can also be used as the basis for hypothesis tests about the magnitude and direction of multivariate trends in experiments. If control and treated samples (or before and after samples) can be matched in pairs or groups, then the paths from the control/before to the treated/after samples can be thought of as the multivariate “response” to whatever the treatment was (McCune et al. 2002). This is because differences in position in NMDS space represent differences in terms of the original variables. The example below uses simulated data to illustrate the method.

- **Left panel:** NMDS ordination of samples. Arrows show change from “before” to “after”. Color of arrows signifies control treatment (black) or treated samples (red).
- **Center panel:** “Change vectors” translated so that they emanate from the origin. Magnitude of vector indicates magnitude of change from before to after treatment. Direction of vector indicates direction of change.
- **Right panel:** Change vectors translated to origin and scaled to unit length. Position of end points on unit circle indicate direction of change. In ecology these are also called “composition vectors” because they indicate changes in species composition regardless of magnitude of change.



The positions of the vector endpoints in the center panel can be used to test hypotheses about the overall change in the response variable matrix in response to treatment. An MRPP or other multivariate test such as PERMANOVA can be used to compare the locations of treatment groups in NMDS space. Differences in location in the center plot reflect both *magnitude* and *direction* of change.

Similarly, the positions of the composition vectors in the right panel can be used to test hypotheses about changes in the response variable matrix in response to treatment. Unlike the center panel, the differences in position in the right panel reflect *direction of change* only.

8.5.3 Other ordination techniques

PCA and NMDS were explored in detail because they are two of the most commonly used and applicable ordination techniques for biologists. There are other techniques out there that you may find useful but are beyond the scope of the course.

Principal coordinates analysis (PCoA) is a variation of PCA that can use any kind of distance metric. In one sense PCA uses the Euclidean distance metric because of its reliance on least squares. PCoA can use any distance metric, which may be desirable for data that do not meet the linearity and normality assumptions of PCA. PCoA works by converting distances into the elements of the crossproducts matrix \mathbf{S} , which basically “pretends” that non-Euclidean distances are Euclidean.

Canonical ordination techniques simultaneously analyze two data matrices. Canonical ordination methods can be symmetric or asymmetric. In **asymmetric** methods, there is a response dataset and a predictor dataset. Reversing the roles of the matrices will change the outcome of the analysis. In contrast, **symmetric** methods treat both matrices the same. That is, in a symmetric ordination analysis, neither matrix is considered the “response” or “predictor” dataset. Reversing the roles of the matrices will not change the outputs.

8.5.3.1 Canonical asymmetric methods

Canonical asymmetric methods produce ordinations of a response matrix **Y** that are constrained to be linearly related to a second set of variables, the predictor matrix **X**.

Redundancy analysis (RDA) is an extension of multiple regression that models multivariate rather than univariate response values. Each ordination axis in RDA is called a “canonical axis” and is similar to a principal component. Each canonical axis in the response matrix **Y** is maximally related to a linear combination of variables in the predictor matrix **X**. RDA preserves the Euclidean distances among samples; this is a consequence of the relationship between RDA and multiple linear regression.

Canonical correspondence analysis (CCorA) is similar to RDA, but preserves a different distance metric (χ^2 or “chi-squared”) among the samples. CCorA may thus be more appropriate than RDA when data are nonnormal or when relationships are nonlinear.

Linear discriminant analysis (LDA) identifies linear combinations of variables in a predictor matrix **X** that classify observations in a response matrix **Y**. LDA can be thought of loosely as a multivariate analogue to multiple logistic or multinomial regression. Classification trees may present a more robust alternative to LDA in many situations.

8.5.3.2 Canonical symmetric methods

Canonical correlation analysis (CCA) is a constrained ordination technique where the ordination of one matrix is informed by multiple regression on another matrix. Given two data matrices **A** and **B**, with correlations among variables in **A** and **B**, CCA fits an ordination for both **A** and **B** where the canonical axes of each ordination are correlated with variables in the other matrix. In other words, CCA finds gradients in one matrix and tries to explain them in terms of information from another matrix. CCA is a powerful tool for exploratory analysis but can be confusing to fit and interpret.

Co-inertia analysis (CoIA) and **Procrustes analysis (Proc)** both seek ordinations of two data matrices **A** and **B** that represent relationships in both **A** and **B** simultaneously. CoIA and Proc work very differently but answer a

similar question: how can samples be arranged in a reduced dimension space that preserves distances in terms of two sets of variables?

These techniques are described by Legendre and Legendre (2012) and McCune et al. (2002). The former reference has a more thorough mathematical treatment; the latter is more introductory. Both are excellent. Several newer references (Legendre and Legendre 2012, Borcard et al. 2018) include R code for implementing the methods.

Chapter 9

Planning your analysis (what test?)

One of the main questions that biologists have after collecting data is, “What statistical test do I use?” Ignoring the fact that they should have thought about that **before collecting data**, the answer to that question depends on the answers to two other questions:

1. What question are you trying to answer?
2. What kind of data do you have?

After exploring the data and gaining an understanding of the patterns and problems in a dataset, it is time to run the main analysis. This guide is designed to help you use what you learned in exploring the data to choose an appropriate analytical strategy.

9.1 How to use this guide

The sections below list some common analytical strategies that you might use for some common types of biological questions, and guides you through the selection process. This document is meant to serve as a primer and rough guide; it is not meant to be exhaustive. This document is also very much a work in progress. Not every situation is covered, and there might be issues in your dataset that make the recommendations below problematic. Each section is structured as a dichotomous key, because biologists love dichotomous keys. At each step, choose the option that best applies to your data. If you don’t know the answer to a question or which option to choose, you may need to go back and do some more exploratory data analysis. R functions are provided for most methods.

9.2 What question are you trying to answer?

The first thing to nail down is what question you are trying to answer. This does not mean what biological question you are investigating, but the two are related. An example biological question might be:

How do small mammals (rodents) adapt to live in urban environments?

This is a good biological question, but it is not a question that you can answer with statistics. In order to answer the overarching biological question, we usually need to break it down into smaller chunks: hypotheses that can either be true or false, and predictions (patterns in the data) that would be observed if the hypothesis is true and not observed if a hypothesis is false. For the question above, we might have some working hypothesis like:

Rodents living in urban environments shift their diet to take advantage of anthropogenic food sources.

This is closer to a statistical hypothesis, but we're not quite there yet. What does it mean to "shift their diet"? Compared to what? How do we quantify a "shift"? To make a statistical hypothesis we need to think of specific quantitative or categorical comparisons that would support or contradict the hypothesis. For example, here are a few of the potential predictions implied by the hypothesis above:

- Rodents in urban environments will have a greater proportion of artificial foods in their feces than conspecific rodents in rural or natural environments.
- Rodents in urban environments will consume a smaller number of different kinds of food than rodents in rural environments because of the availability of calorie-dense foods associated with humans.
- Rodents in urban environments will spend less time foraging and make shorter foraging movements than rodents in rural or natural environments.

These three predictions are statements of the kinds of patterns that we should observe if the hypothesis was true. Critically, they include comparison phrases like **greater than** or **smaller number of** or **less time than**. Those comparisons are the sorts of patterns that a statistical model can actually evaluate. Knowing what kind of pattern you are testing is a key step in planning your data analysis.

Broadly speaking, statistical hypotheses in biology tend to fall into just a few categories:

1. Testing for a difference in mean or location between groups.
2. Testing for a continuous relationship between a response variable (or multiple variables) and one or more predictor variables. Also referred to as "predicting" or "modeling" a continuous variable.
3. Classifying observations based on predictor variables.

These categories are not mutually exclusive. Many questions phrased as one category can be reframed into another. How you frame your question depends on many factors and at some point will involve some discretion on your part. *There is often more than one right way analyze a dataset*, but the right ways are a vanishingly small subset of the possible ways to analyze your data.

9.3 Testing for a difference in mean or location

Dichotomous key for testing for a difference in mean or location:

1. Single response variable (or, one variable tested at a time): 5
- 1'. Multiple response variables tested at once: 2
 2. Data multivariate normal: 3
 - 2'. Data not multivariate normal: 4
3. All conditions for MANOVA met: try MANOVA. R function `manova()`.
- 3'. MANOVA conditions not all met: 4.
 4. Try permutational MANOVA (PERMANOVA) (R function `vegan::adonis()`)
or:
 - 4'. Ordinate using NMDS, then try MRPP in NMDS-space. R functions `vegan::metaMDS()` followed by `vegan::mrpp()`.
 5. Comparing between 2 groups (i.e., a single factor with 2 levels): 6
 - 5'. Comparing between >2 groups: 8
 6. Data meet assumptions of t-test: use *t*-test. R function `t.test()`.
 - 6'. Data do not meet assumptions of *t*-test: 7.
 7. Need to predict values of response variable: use GLM. R function `glm()`.
 - 7'. Need only to test for difference in location: use Mann-Whitney test (aka: Wilcoxon rank sum test). R function `wilcox.test()`.
 8. Response variables normal, variance homogeneous: use ANOVA. R function `lm()` followed by `anova()` and/or `aov()`.
 - 8'. Response variables nonnormal OR variance heterogeneous: 9.
 9. Need to predict values of response variable: use GLM. R function `glm()`.
 - 9'. Need to test for difference in location only: use Kruskal-Wallis test. R function `kruskal.test()`.

9.3.1 Additional considerations

Once you reach a decision from the key above, there are a few more options to consider:

- Are there any interactions between your predictor variables? This means that the effects of two predictors are not additive; i.e., the effect of one variable affects the effect of another variable. Most of the methods above can accommodate interactions. You can use R function `interaction.plot()` to explore potential interactions visually.
- Is there a hierarchical structure to your data, with the higher-level groups representing variation that you are interested in explaining? This is often

- called “blocking” and needs to be included in your model.
- Is there a hierarchical structure to your data, with the higher-level groups representing variance you are not interested in but still need to account for? If so, you may want to consider adding random effects to your model. This creates a **mixed model** (see Module 7).

9.4 Testing for a continuous relationship between two or more variables

Dichotomous key for testing continuous relationships:

1. Need to test whether two variables vary together, not predict actual values: 2
- 1'. Need to predict values of response variables: 3
2. Both variables normally distributed, suspected relationship linear: use Pearson’s product moment correlation r . R function `cor()` or `cor.test()`.
- 2'. One or both variables nonnormal and/or suspected relationship nonlinear: use Spearman’s rank correlation coefficient ρ (“rho”). R function `cor.test()`.
3. Suspected relationship linear, or can be made linear by transform: 4
- 3'. Suspected relationship nonlinear and cannot be made linear: 5.
4. Response variable normally distributed or can be transformed to be normal: use linear regression (on transformed scale, if needed). R function `lm()`.
- 4'. Response variable nonnormal and cannot be transformed to normality: use GLM. R function `glm()`.
5. Response variable and residuals normal: use NLS. R function `nls()`.
- 5'. Response variable and residuals nonnormal: try GAM, GNM, or regression trees (in that order). R functions `mgcv:::gam()`, `gnm:::gnm()`, and `rpart:::rpart()`, respectively.

9.4.1 Additional considerations

Once you reach a decision from the key above, there are a few more options to consider:

- Is there autocorrelation in your data? If so, you may want to consider including an autocorrelation structure.
- Is there a hierarchical structure to your data, with the higher-level groups representing variance you are not interested in but still need to account for? If so, you may want to consider adding random effects to your model. This creates a **mixed model**:
 - Linear models become linear mixed models (LMM). R function `lme4:::lmer()`.
 - Generalized linear models become generalized linear mixed models (GLMM). R function `lme4:::glmer()`.
 - Additive models become generalized additive mixed models (GAMM). R function `mgcv:::gamm()`.

- NLS models become nonlinear mixed models (NLME). R function `nlme::nlme(#mod-07-nlme)` or `lme4::nlmer()`.
- Do your data meet most of the assumptions for linear regression, but with heteroscedasticity (non-constant variance in residuals)? You might consider using generalized least squares (GLS) with a different variance structure. R function `nlme::gls()`.
- Is there severe multicollinearity among your predictor variables? If so, try dropping some of the collinear predictors or try a regularization technique such as ridge or lasso regression. Various R packages.
- Do you not have a clear idea of the form of the relationship between your continuous response variable and the predictors? Or, do you not want to assume a data model and just let the data speak for themselves? Try a machine learning technique like boosted regression trees. R function `gbm::gbm()`.

9.5 Classifying observations

Dichotomous key for classification problems:

1. Outcome binary: 2
- 1'. Outcome not binary (>2 options): 3.
 2. Logistic regression works: use logistic regression. R function `glm()`.
 - 2'. Logistic regression doesn't work: use classification trees. R function `rpart::rpart()`.
3. Try multinomial logistic regression. Several R packages, including `nnet` and `mlogit`.
- 3'. Try classification trees. R function `rpart::rpart()`.

9.6 Conclusions

This guide was designed to cover a lot of possible pathways for biological data analysis, but of course there are some situations that aren't covered. By the time you have deconstructed your problem enough to answer the diagnostic questions above, you should be well on your way to figuring out what analysis to use. Whatever you do, make sure that you document your decision and its justification...this will be very helpful later!

Literature Cited

- Alberti, E., G. Gioia, G. Sironi, S. Zanzani, P. Riccaboni, M. Magrini, and M. Manfredi. 2011. *Elaphostrongylus cervi* in a population of red deer (*Cervus elaphus*) and evidence of cerebrospinal nematodiasis in small ruminants in the province of Varese, Italy. *Journal of Helminthology* 85:313–318.
- Anderson, M. J. 2001. A new method for non-parametric multivariate analysis of variance. *Austral Ecology* 26:32–46.
- Bodin, N., H. Pethybridge, L. M. Duffy, A. Lorrain, V. Allain, J. M. Logan, F. Ménard, B. Graham, C. A. Choy, C. J. Somes, R. J. Olson, and J. W. Young. 2021. Global data set for nitrogen and carbon stable isotopes of tunas. *Ecology* 102:e03265.
- Bolker, B. M. 2008. Ecological models and data in R. Princeton University Press, Princeton, New Jersey, USA.
- Bolker, B. M., M. E. Brooks, C. J. Clark, S. W. Geange, J. R. Poulsen, M. H. H. Stevens, and J.-S. S. White. 2009. Generalized linear mixed models: A practical guide for ecology and evolution. *Trends in Ecology & Evolution* 24:127–135.
- Borcard, D., F. Gillet, and P. Legendre. 2018. Numerical ecology with R. Springer, New York.
- Box, G. E. P., and D. R. Cox. 1964. An analysis of transformations. *Journal of the Royal Statistical Society: Series B (Methodological)* 26:211–243.
- Brieuc, M. S. O., C. D. Waters, D. P. Drinan, and K. A. Naish. 2018. A practical introduction to random forest for genetic association studies in ecology and evolution. *Molecular Ecology Resources* 18:755–766.
- Broman, K. W., and K. H. Woo. 2018. Data organization in spreadsheets. *The American Statistician* 72:2–10.
- Bubrig, L. T., J. M. Sutton, and J. L. Fierst. 2020. *Caenorhabditis elegans* dauers vary recovery in response to bacteria from natural habitat. *Ecology and Evolution* 10:9886–9895.
- Burnham, K. P., and D. R. Anderson. 2004. Multimodel inference: Understanding AIC and BIC in model selection. *Sociological methods & research* 33:261–304.
- Burnham, K., and D. Anderson. 2002. Model selection and multi-model inference: a practical information-theoretic approach. Springer-Verlag, New York.
- Button, K. S., J. P. Ioannidis, C. Mokrysz, B. A. Nosek, J. Flint, E. S. Robinson,

- and M. R. Munafò. 2013. Power failure: why small sample size undermines the reliability of neuroscience. *Nature Reviews Neuroscience* 14:365–376.
- Cade, B. S., and B. R. Noon. 2003. A gentle introduction to quantile regression for ecologists. *Frontiers in Ecology and the Environment* 1:412–420.
- Carey, V. J. 2019. gee: Generalized estimation equation solver.
- Carnell, R. 2019. Triangle: Provides the standard distribution functions for the triangle distribution.
- Chivers, D. J., and C. M. Hladik. 1980. Morphology of the gastrointestinal tract in primates: Comparisons with other mammals in relation to diet. *Journal of Morphology* 166:337–386.
- Clarke, K. R. 1993. Non-parametric multivariate analyses of changes in community structure. *Australian journal of ecology* 18:117–143.
- Clarke, K. R., P. J. Somerfield, and M. G. Chapman. 2006. On resemblance measures for ecological studies, including taxonomic dissimilarities and a zero-adjusted bray-curtis coefficient for denuded assemblages. *Journal of Experimental Marine Biology and Ecology* 330:55–80.
- Cutler, D. R., T. C. Edwards, K. H. Beard, A. Cutler, K. T. Hess, J. Gibson, and J. J. Lawler. 2007. Random forests for classification in ecology. *Ecology* 88:2783–2792.
- Dalgaard, P. 2002. Introductory statistics with R. Springer Science+Business Media.
- De'ath, G. 2007. Boosted trees for ecological modeling and prediction. *Ecology* 88:243–251.
- De'ath, G., and K. E. Fabricius. 2000. Classification and regression trees: A powerful yet simple technique for ecological data analysis. *Ecology* 81:3178–3192.
- De Solla, S. R., P. A. Martin, K. J. Fernie, B. J. Park, and G. Mayne. 2006. Effects of environmentally relevant concentrations of atrazine on gonadal development of snapping turtles (*chelydra serpentina*). *Environmental Toxicology and Chemistry: An International Journal* 25:520–526.
- Dormann, C. F., J. Elith, S. Bacher, C. Buchmann, G. Carl, G. Carré, J. R. G. Marquéz, B. Gruber, B. Lafourcade, P. J. Leitão, T. Münkemüller, C. McClean, P. E. Osborne, B. Reineking, B. Schröder, A. K. Skidmore, D. Zurell, and S. Lautenbach. 2013. Collinearity: A review of methods to deal with it and a simulation study evaluating their performance. *Ecography* 36:27–46.
- Elith, J., and J. Leathwick. 2017. Boosted regression trees for ecological modeling. R Documentation. Available online: <https://cran.r-project.org/web/packages/dismo/vignettes/brt.pdf> (accessed on 2021-12-23).
- Elith, J., J. R. Leathwick, and T. Hastie. 2008. A working guide to boosted regression trees. *Journal of Animal Ecology* 77:802–813.
- Faurby, S., M. Davis, R. Ø. Pedersen, S. D. Schowanek, A. Antonelli, and J.-C. Svenning. 2018. PHYLACINE 1.2: The phylogenetic atlas of mammal macroecology. *Ecology* 99:2626.
- Foucault, Q., A. Wieser, A.-M. Waldvogel, B. Feldmeyer, and M. Pfenninger. 2018. Rapid adaptation to high temperatures in *Chironomus riparius*. *Ecol-*

- ogy and Evolution 8:12780–12789.
- Gelfand, A. E., and P. Vounatsou. 2003. Proper multivariate conditional autoregressive models for spatial data analysis. *Biostatistics* 4:11–15.
- Gould, S. J. 1974. The origin and function of 'bizarre' structures: Antler size and skull size in the 'irish elk,' *Megaloceros giganteus*. *Evolution* 28:191–220.
- Green, N. S., M. L. Wildhaber, and J. L. Albers. 2019. Potential responses of the Lower Missouri River shovelnose sturgeon (*Scaphirhynchus platorynchus*) population to a commercial fishing ban. *Journal of Applied Ichthyology* 35:370–377.
- Green, N. S., M. L. Wildhaber, and J. L. Albers. 2021. Effectiveness of a distance sampling from roads program for white-tailed deer in the National Capital Region parks. Natural Resource Report, National Park Service, Fort Collins, Colorado.
- Green, N. S., M. L. Wildhaber, J. L. Albers, T. W. Pettit, and M. J. Hooper. 2020. Efficient mammal biodiversity surveys for ecological restoration monitoring. Integrated Environmental Assessment and Management.
- Grolemund, G., and H. Wickham. 2011. Dates and times made easy with lubridate. *Journal of Statistical Software* 40:1–25.
- Head, M. L., L. Holman, R. Lanfear, A. T. Kahn, and M. D. Jennions. 2015. The extent and consequences of p-hacking in science. *PLoS Biology* 13:e1002106.
- Hill, M. O. 1979. A FORTRAN program for arranging multivariate data in an ordered two-way table by classification of the individuals and attributes. TWINSPLAN.
- Hobbs, N. T., and R. Hilborn. 2006. Alternatives to statistical hypothesis testing in ecology: a guide to self teaching. *Ecological Applications* 16:5–19.
- Højsgaard, S., U. Halekoh, and J. Yan. 2006. The R package geepack for generalized estimating equations. *Journal of statistical software* 15:1–11.
- Hurlbert, S. H. 1984. Pseudoreplication and the design of ecological field experiments. *Ecological Monographs* 54:187–211.
- Hutcheon, J. M., J. A. W. Kirsch, and T. G. Jr. 2002. A comparative analysis of brain size in relation to foraging ecology and phylogeny in the Chiroptera. *Brain, behavior and evolution* 60:165–180.
- Inderjit, J. C. Streibig, and M. Olofsdotter. 2002. Joint action of phenolic acid mixtures and its significance in allelopathy research. *Physiologia Plantarum* 114:422–428.
- James, G., D. Witten, T. Hastie, and R. Tibshirani. 2013. An introduction to statistical learning. 7th edition. Springer.
- Jones, D. T. 2019. Setting the standards for machine learning in biology. *Nature Reviews Molecular Cell Biology* 20:659–660.
- Kan, A. 2017. Machine learning applications in cell image analysis. *Immunology and Cell Biology* 95:525–530.
- Kéry, M. 2002. Inferring the absence of a species: a case study of snakes. *The Journal of Wildlife Management*:330–338.
- Kéry, M. 2010. Introduction to WinBUGS for ecologists: Bayesian approach to regression, ANOVA, mixed models and related analyses. Academic Press.
- Kirchman, D. L., X. A. G. Morán, and H. Ducklow. 2009. Microbial growth

- in the polar oceans—role of temperature and potential impact of climate change. *Nature Reviews Microbiology* 7:451–459.
- Kirchner, B. N., N. S. Green, D. A. Sergeant, J. N. Mink, and K. T. Wilkins. 2011. Responses of small mammals and vegetation to a prescribed burn in a tallgrass blackland prairie. *The American Midland Naturalist* 166:112–125.
- Koenker, R. 2021. Quantreg: Quantile regression.
- Kuehn, I., and C. F. Dormann. 2012. Less than eight (and a half) misconceptions of spatial analysis. *Journal of Biogeography* 39:995–998.
- Kunz, B. K., J. H. Waddle, and N. S. Green. 2019. Amphibian monitoring in hardwood forests: Optimizing methods for contaminant-based compensatory restorations. *Integrated environmental assessment and management*.
- Kuznetsova, A., P. B. Brockhoff, and R. H. B. Christensen. 2017. lmerTest package: Tests in linear mixed effects models. *Journal of Statistical Software* 82:1–26.
- Lazic, S. E., C. J. Clarke-Williams, and M. R. Munafò. 2018. What exactly is ‘N’ in cell culture and animal experiments? *PLoS Biology* 16:e2005282.
- Lebreton, J.-D., K. P. Burnham, J. Clobert, and D. R. Anderson. 1992. Modeling survival and testing biological hypotheses using marked animals: a unified approach with case studies. *Ecological Monographs* 62:67–118.
- Legendre, P., and L. Legendre. 2012. *Numerical ecology*. Elsevier, Amsterdam.
- Leonard, E. M., and C. M. Wood. 2013. Acute toxicity, critical body residues, Michaelis–Menten analysis of bioaccumulation, and ionoregulatory disturbance in response to waterborne nickel in four invertebrates: *Chironomus riparius*, *Lymnaea stagnalis*, *Lumbriculus variegatus* and *Daphnia pulex*. *Comparative Biochemistry and Physiology Part C: Toxicology & Pharmacology* 158:10–21.
- Liang, K.-Y., and S. L. Zeger. 1986. Longitudinal data analysis using generalized linear models. *Biometrika* 73:13–22.
- Link, W. A., E. Cam, J. D. Nichols, and E. G. Cooch. 2002. Of BUGS and birds: Markov chain Monte Carlo for hierarchical modeling in wildlife research. *The Journal of Wildlife Management*:277–291.
- Ludwig, J. A., J. F. Reynolds, L. QUARTET, and J. Reynolds. 1988. *Statistical ecology: A primer in methods and computing*. John Wiley & Sons.
- Lunn, D., D. Spiegelhalter, A. Thomas, and N. Best. 2009. The BUGS project: evolution, critique and future directions. *Statistics in Medicine* 28:3049–3067.
- Makin, T. R., and J.-J. O. de Xivry. 2019. Science forum: Ten common statistical mistakes to watch out for when writing or reviewing a manuscript. *Elife* 8:e48175.
- Mangiafico, S. 2021. rcompanion: Functions to support extension education program evaluation.
- Mantel, N. 1967. The detection of disease clustering and a generalized regression approach. *Cancer research* 27:209–220.
- McCarthy, M. A. 2007. *Bayesian methods for ecology*. Cambridge University Press, Cambridge, United Kingdom.
- McCune, B., J. B. Grace, and D. L. Urban. 2002. *Analysis of ecological communities*. MjM software design Gleneden Beach, Oregon, USA.

- Nelder, J. A., and R. W. M. Wedderburn. 1972. Generalized linear models. *Journal of the Royal Statistical Society: Series A (General)* 135:370.
- Penone, C., A. D. Davidson, K. T. Shoemaker, M. Di Marco, C. Rondinini, T. M. Brooks, B. E. Young, C. H. Graham, and G. C. Costa. 2014. Imputation of missing data in life-history trait datasets: Which approach performs the best? *Methods in Ecology and Evolution* 5:961–970.
- Pethybridge, H., C. A. Choy, J. M. Logan, V. Allain, A. Lorrain, N. Bodin, C. J. Somes, J. Young, F. Ménard, C. Langlais, L. Duffy, A. J. Hobday, P. Kuhnert, B. Fry, C. Menkes, and R. J. Olson. 2018. A global meta-analysis of marine predator nitrogen stable isotopes: Relationships between trophic structure and environmental conditions. *Global Ecology and Biogeography* 27:1043–1055.
- Plummer, M. 2003. JAGS: A program for analysis of Bayesian graphical models using Gibbs sampling. Pages 1–10 Proceedings of the 3rd international workshop on distributed statistical computing. Vienna, Austria.
- Plummer, M. 2021. rjags: Bayesian graphical models using MCMC.
- Ritz, C., F. Baty, J. C. Streibig, and D. Gerhard. 2015. Dose-response analysis using R. *PLoS ONE* 10:e0146021.
- Roleček, J., L. Tichý, D. Zelený, and M. Chytrý. 2009. Modified TWINSPLAN classification in which the hierarchy respects cluster heterogeneity. *Journal of Vegetation Science* 20:596–602.
- Schabenberger, O., and F. J. Pierce. 2001. Contemporary statistical models for the plant and soil sciences. CRC press, Boca Raton, Florida, USA.
- Smith, F. A., and S. K. Lyons. 2013. Animal body size: Linking pattern and process across space, time, and taxonomic group. University of Chicago Press, Chicago.
- Spector, P. 2008. Data manipulation with R. Springer Science+Business Media, New York.
- Spiess, A.-N. 2018. propagate: Propagation of uncertainty.
- Spiess, A.-N., and N. Neumeyer. 2010. An evaluation of R^2 as an inadequate measure for nonlinear models in pharmacological and biochemical research: A Monte Carlo approach. *BMC Pharmacology* 10.
- Su, Y.-S., and Masanao Yajima. 2021. R2jags: Using R to run 'JAGS'.
- Tarca, A. L., V. J. Carey, X. Chen, R. Romero, and S. Drăghici. 2007. Machine learning and its applications to biology. *PLoS Computational Biology* 3:e116.
- Therneau, T., and B. Atkinson. 2019. rpart: Recursive partitioning and regression trees.
- Tufte, E. 2001. The visual display of quantitative information. Cheshire: Graphic Press.–2001.–213 p.
- Turner, H., and D. Firth. 2020. Generalized nonlinear models in R: An overview of the gnm package.
- Venables, W. N., and B. D. Ripley. 2002. Modern applied statistics with S. Fourth. Springer, New York.
- Vicente, J., Y. Fierro, and C. Gortazar. 2005. Seasonal dynamics of the fecal excretion of *Elaphostrongylus cervi* (Nematoda, Metastrengyoidea) first-stage larvae in Iberian red deer (*Cervus elaphus hispanicus*) from southern

- Spain. Parasitology Research 95:60–64.
- Voit, E. O. 2019. Perspective: dimensions of the scientific method. PLoS Computational Biology 15:e1007279.
- Ward, J. H. 1963. Hierarchical grouping to optimize an objective function. Journal of the American Statistical Association 58:236–244.
- Warton, D. I., and F. K. C. Hui. 2011. The arcsine is asinine: The analysis of proportions in ecology. Ecology 92:3–10.
- Wasserstein, R. L., and N. A. Lazar. 2016. The ASA statement on p-values: Context, process, and purpose. Taylor & Francis.
- Werner, J., and E. M. Griebeler. 2014. Allometries of maximum growth rate versus body mass at maximum growth indicate that non-avian dinosaurs had growth rates typical of fast growing ectothermic sauropsids. PloS ONE 9:e88834.
- White, J. W., A. Rassweiler, J. F. Samhouri, A. C. Stier, and C. White. 2014. Ecologists should not use statistical significance tests to interpret simulation model results. Oikos 123:385–388.
- Whytock, R. C., J. Świeżewski, J. A. Zwerts, T. Bara-Słupski, A. F. Koumba Pambo, M. Rogala, L. Bahaa-el-din, K. Boekee, S. Brittain, A. W. Cardoso, and others. 2021. Robust ecological analysis of camera trap data labelled by a machine learning model. Methods in Ecology and Evolution.
- Wickham, H. 2014. Tidy data. Journal of Statistical Software 59.
- Wickham, H., M. Averick, J. Bryan, W. Chang, L. D. McGowan, R. François, G. Grolemund, A. Hayes, L. Henry, J. Hester, M. Kuhn, T. L. Pedersen, E. Miller, S. M. Bache, K. Müller, J. Ooms, D. Robinson, D. P. Seidel, V. Spinu, K. Takahashi, D. Vaughan, C. Wilke, K. Woo, and H. Yutani. 2019. Welcome to the tidyverse. Journal of Open Source Software 4:1686.
- Wood, S. N. 2017. Generalized additive models: An introduction with R. 2nd edition. CRC press.
- Woodward, H. N., E. A. F. Fowler, J. O. Farlow, and J. R. Horner. 2015. *Maiasaura*, a model organism for extinct vertebrate population biology: A large sample statistical assessment of growth dynamics and survivorship. Paleobiology 41:503–527.
- Zuur, A. F., E. N. Ieno, and C. S. Elphick. 2010. A protocol for data exploration to avoid common statistical problems. Methods in ecology and evolution 1:3–14.
- Zuur, A. F., E. N. Ieno, and G. M. Smith. 2007. Analysing ecological data. Springer Science+Business Media, New York.
- Zuur, A. F., E. N. Ieno, N. J. Walker, A. A. Saveliev, G. M. Smith, and others. 2009a. Mixed effects models and extensions in ecology with R. Springer, New York.
- Zuur, A., E. N. Ieno, and E. Meesters. 2009b. A beginner’s guide to R. Springer Science+Business Media, New York.