Data Analysis and Statistical Thinking: An R Workbook

Fernando Racimo, …

2021-03-02

Contents

Chapter 1

Introduction

Chapter 2

Getting Started with Data Analysis

Chapter 3

Probability Part 1

3.1 Tossing a coin

We can "virtually" toss a coin in our R console, using the rbinom() function:

```
rbinom(1, 1, 0.5)
```

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

Try copying the above chunk to your R console and running it multiple times. Do you always get the same result?

This function has 3 required input parameters: n, size and prob. The first parameter (n) determines the number of trials we are telling R to perform, in other words, the number of coin tosses we want to generate:

```
rbinom(20, 1, 0.5)
```

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

Here, we generate 20 toin cosses, and the zeroes and ones represent whether we got a heads or a tails in each trial. For now, we will ignore the second parameter (size) and fix it at 1, but we'll revisit it in a moment. The third parameter (prob) dictates how biased the coin is. If we set it to 0.9, we'll get the outcomes of a biased coin toss, in particular biased towards heads:

```
rbinom(20, 1, 0.9)
```

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

Exercise: What happens when you set prob to 0.1? Or 0.999? Why?

What we are really doing here is simulating outcomes of a random variable that is governed by a particular probability distribution - in this case, the Bernoulli

distribution. We can assign a name to this variable for storage and manipulation later on:

```
X <- rbinom(1, 1, 0.9)
```

If you type this in your console, X will now store the value of the outcome of a biased coin toss (either 0 or 1), which you can use later in your code.

How can we verify that R is really doing what we think it is doing? Well, if we think we have a fair coin and we throw it many times, then, on average, we should get the same number of heads and tails, right? This experiment should be more accurate the more trials we have. We can compute the average of our coin tosses by using the function sum(), which adds the elements of a vector, and then dividing by the total number of trials.

Let's create a new variable (n) that will determine how many trials we attempt, say 20.

```
n <- 20
sum(rbinom(n, 1, 0.5)) / n
```

```
## [1] 0.35
```

Exercise: Run the chunk of code above, in your own console. Do you get the same number as I do? Do you get exactly 0.5? If not, why not? Try the same exercise but with 100 trials, 1000 trials and 100000 trials. What happens as we increase the number of trials? This should illustrate how powerful R can be. We just threw 100 thousand coins into the air without even lifting our fingers! Try to repeat the exercise, but this time, set the Bernoulli prob parameter to be equal to a number of your choice (between 0 and 1). What is the average of all your coin tosses?

3.2 Adding up coin tosses

Let's say we are now not interested in any particular coin toss, but in the sum of several coin tosses. Each toss is represented by a 0 or a 1, so the sum of all our tosses cannot be smaller than 0 or larger than the total number of tosses we perform.

Try running the code below 5 times. What numbers do you obtain?

```
sum(rbinom(20, 1, 0.5))
```

```
## [1] 7
```

Turns out there's a short-hand way of performing the same experiment, i.e. tossing a bunch of coins - each a Bernoulli random variable - observing their outcomes and adding them up, without using the sum() function at all. Here's where the second input parameter - size - of the rbinom() function comes into

play. So far, we've always left it equal to 1 in all our command lines above, but we can set it to any positive integer:

```
rbinom(1, 20, 0.5)
```

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

The above code is equivalent to taking 20 Bernoulli trials, and then adding their outcomes up. The "experiment" we are running is now not a single coin toss, but 20 coin tosses together. The outcome of this experiment is neither heads nor tails, but the sum of all the heads in all those coin tosses. It turns out that this "experiment" is a probability distribution in its own right, and it is called the Binomial distribution. It has two parameters: the size of the experiment (how many tosses we perform) and the probability of heads for each toss (the prob parameter). The Bernoulli distribution is just a specific case of the Binomial distribution (the case in which we only toss 1 coin, i.e. size = 1). You can read more about this distribution if you go to the help menu for this function (type "?rbinom).

The Binomial and Bernoulli distributions are examples of distributions for discrete random variables, meaning random variables whose values can only take discrete values (0, 1, 2, 3, etc.). There are other types of distributions we'll study later, some of which can also take continuous values. For example, these could be any real number, or any real number between 2.4 and 8.3, or any positive number, etc. but we need not worry about these other distributions for now.

3.3 The expectation

We can compute the average of multiple Binomial trials. Let's try adding the results of 5 Binomial trials, each with size 20 (how many Bernoulli trials is this equivalent to?):

```
n <- 5
size <- 20
prob <- 0.5
X <- rbinom(n, size, prob)
X
```

```
## [1]  9 12  9 13  8
```

```
Xsum <- sum(X)
Xsum
```

```
## [1] 51
```

To get the average, we divide by the total number of trials. Remember here that the number of Binomial trials is 5:

```
Xave <- Xsum / n
Xave
```

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

A shorthand for obtaining the mean is the function mean(). This should give you the same result:

```
Xave <- mean(X)
Xave
```

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

Note that the mean need not be an integer, even though the outcome of each Binomial trial *must* be an integer.

Exercise: Try repeating the same exercise but using 100 Binomial trials, and then 100 thousand Binomial trials. What numbers do you get? What number do we expect to get as we increase the number of Binomial trials?

This number is called the *Expectation* of a random variable. For discrete random variables, it is defined as follows:

$$E[X] = \sum_i x_i P[X = x_i]$$

Here the sum is over all possible values that the random variable X can take. In other words, it is equal to the sum of each of these values, weighted by the probability that the random variable takes that value.

In the case of a variable that follows the Binomial distribution, the expectation happens to be equal to the product of size and prob:

$$E[X] = np$$

Note that the n here refers to the size of a single Binomial trial.

This should make intuitive sense: if we throw a bunch of coins and add up their results, the number we expect to get should be approximately equal to the probability of heads times the number of tosses we perform. Note that this equality only holds approximately: for any given Binomial trial, we can get any number between 0 and the size of the Binomial experiment. If we take an average over many Binomial experiments, we'll approach this expectation ever more accurately. The average (also called "sample mean") over a series of n experiments is thus an approximation to the expectation, which is often unknown in real life. The sample mean is often represented by a letter with a bar on top:

$$\bar{x} = \frac{\sum_{j=1}^{n} x_i}{n}$$

You can also think of the expectation as the mean over an infinite number of trials.

3.4 Our first probability mass function

Ok, all this talk of Bernoulli and Binomial is great. But what is the point of it? The nice thing about probability theory is that it allows us to better think about processes in nature, by codifying these processes into mathematical equations.

For example, going back to our coin tossing example, if someone asked you how many heads you'd expect among 20 tosses, your best bet would be to give the mean of a Binomial distribution with size 20 and probability of heads equal to 0.5: $0.5 * 20 = 10$.

But this is a fairly intuitive answer. You didn't need probability theory to tell you that about half the coins would turn out heads. Plus, we all know that one might not get 10 heads: we might get 9, 13 or even 0 if we're very unlucky. What is then, the probability that we would get 10 heads? In other words, if we were to repeat our Binomial experiment of 20 tosses a large number of times, how many of those experiments would yield exactly 10 heads? This is a much harder question to answer. Do you have a guess?

It turns out that probability theory can come to the rescue. The Binomial distribution has a very neat equation called its "Probability Mass Function" (or PMF, for short), which answers this question exactly:

$$P[X = k] = \binom{n}{k} p^k (1-p)^{n-k}$$

If we let k = 10, and plug in our values for the sample size and probability of heads, we get an exact answer:

$$P[X = 10] = \binom{20}{10} 0.5^{10} 0.5^{10} = 0.1762...$$

So in about 17% of all Binomial experiments of size 20 that we might perform, we should get that 10 out of the 20 tosses are heads.

Let's unpack this equation a bit. You can see that it has 3 terms, which are multiplied together. We'll ignore the first term for now. Let's focus on the second term: p^k. This is simply equivalent to multiplying our probability of heads k times. In other words, this means that we need k of the tosses to be heads, and the probability of this happening is just the product of the probability of heads in each of the total (n) tosses. In our case, $k = 10$, because we need 10 tosses, and $n = 20$ because we tossed the coin 20 times. So far, so good.

The third term is very similar. We not only need 10 heads, but also 10 tails (because we need exactly 10 of the tosses to be heads, no more, no less). The

probability of this happening is the product of the probability of getting tails $(1-p)$ multiplied $n-k$ times. In our case, $n-k$ happens to also be equal to 10.

But what about the first term: $\binom{n}{k}$? This is called a binomial coefficient. It is used to represent the ways in which one can choose an unordered subset of k elements from a fixed set of n elements. In our case, we need 10 of our 20 tosses to be heads, but we don't need to specify exactly which of the tosses will be heads. It could be that we get 10 heads followed by 10 tails, or 10 tails followed by 10 heads, or 1 head and 1 tail interspersed one after the other, or any other arbitrary combination of 10 heads and 10 tails. The binomial coefficient gives us the number of all these combinations. It is defined as:

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!}$$

where

$$a! = a(a-1)(a-2)(a-3)...1$$

Exercise: Plug in other values of k into the Probability Mass Function of the Binomial distribution. What probabilities do you get? How do these change as the numbers are closer or farther away from the expectation $(n*p=10)$?

Ok, this is very neat, but how can we check this equation is correct? Well, we can use simulations! We can generate a large number of Binomial trials in R, and check how many of those are exactly equal to our choice of k. The fraction of all trials that are equal to k should approximate $P[X=k]$. Let's try this for $k=10$ and 500 trials.

```
n <- 500
size <- 20
prob <- 0.5
binomvec <- rbinom(n, size, prob)
binomvec
```

```
##  [1]  9  9  8  6  8  8  8 10  9 11 10  7 11 10  7  9 11  8 12  6 11  8  9  9 10
## [26] 12  8 13 12  9  9  8  8  9 14  9 13  8  7  9  7 12 11  8 12 10 11 10 12  9
## [51]  9 12 11 10 12 15 16 11 13 11  9  8 11  8 11  8  6  6 11  8  9  9  9  9 14
## [76] 11 13 15 12  8 10 12  8  8  4  8 11  9 10  8 12  8  9 12 12  9  8 13 10  8
## [101]  7 12  5 10 11  9 13 10  7 11  8 11  8 10 11 10  8  7  8 11 13 11 14 11  9
## [126]  9 10  8 11 10 10  5 12 12  7 11  5 10 12  6 13  9 16  4 10 12  8  8 13  8
## [151] 10 10 12  5 11  8 11 11  8  7 12  8 10 10 11  7  9 11  7 11 10 12 10  9 11
## [176] 11 10 11 13  6  8  9 15  9  7  9  8  8 11  7 13  7 10 13  9 12 12  8  7 13
## [201] 13  7 10  9  9 11 15 15 12 11  7 12  9 10 12  9 11 14  9  8  8  7 11 10  9
## [226]  7 12 10 14 10 10 10 10 11  8  9  6  9 13  5  9  8 13  7  7  7 11 10  8 12
## [251] 12  9 10  8 12 10  8 11 12  9 10  9 15 11 10 12 14 13 11 10  6 12 13 11 14
## [276]  7  9 11  9  9  7  8  9  8 11  9  9 10 11  7  9 10  9  8  9 13 10  8  9  5
```

```
## [301] 11  8 12  8 12  7 10 10  8  8  8 14 10 12  7 16 11  8 12 11 14 10 10 13 10
## [326] 10 10 10  7  7  7  7 10 11 11  7 13  4 15 11 10 10 12 11 11 12  9  9 12 12
## [351] 12 11  8 11 11 10  9 10  5  7 11  8  9 10  8 11 11 11 10 11 11  8  9 12 11
## [376]  8 12 10 10  9 15  9  9  8 11  7  9 10  7  9 10 15  8 12  9  8  9 10 13 10
## [401] 11 11  7 10  9  9 11  9 14 10  9  9 11  7 10 10 12 14  9  5 11  8 10  8 15
## [426]  9 12 10 14 12  7  9  9 10 10  7  7  9 11  8  8 12  9 13  9  9  9 12  8  9
## [451] 10 11 10 10  7 10  6 10  8 11 10 11 10 10 11  8 13  8  9  8 11 11  8 10 10
## [476]  8 10 11  9 13  8  9  7 13  9 10  9  9  7 12  8  8 13 10 11 13  9  7 13 10
```

We can determine which of these trials was equal to 10 using the "==" symbol:

```
verify <- (binomvec == 10)
verify
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE  TRUE FALSE
## [13] FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [25]  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [37] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE  TRUE
## [49] FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE
## [61] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [73] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE
## [85] FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [97] FALSE FALSE  TRUE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE  TRUE
## [109] FALSE FALSE FALSE FALSE FALSE  TRUE FALSE  TRUE FALSE FALSE FALSE FALSE
## [121] FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE  TRUE  TRUE FALSE
## [133] FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE
## [145]  TRUE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE FALSE FALSE FALSE FALSE
## [157] FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE FALSE FALSE FALSE FALSE
## [169] FALSE FALSE  TRUE FALSE  TRUE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE
## [181] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [193]  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE
## [205] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE
## [217] FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE  TRUE
## [229] FALSE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [241] FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE
## [253]  TRUE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE
## [265]  TRUE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE
## [277] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE
## [289] FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE
## [301] FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE FALSE FALSE FALSE FALSE
## [313]  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE FALSE
## [325]  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE
## [337] FALSE FALSE FALSE FALSE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE
## [349] FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE  TRUE FALSE FALSE
## [361] FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE
## [373] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE
## [385] FALSE FALSE FALSE  TRUE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE
```

```
## [397] FALSE  TRUE FALSE  TRUE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE
## [409] FALSE  TRUE FALSE FALSE FALSE FALSE  TRUE  TRUE FALSE FALSE FALSE FALSE
## [421] FALSE FALSE  TRUE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE
## [433] FALSE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [445] FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE  TRUE  TRUE FALSE  TRUE
## [457] FALSE  TRUE FALSE FALSE  TRUE FALSE  TRUE  TRUE FALSE FALSE FALSE FALSE
## [469] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE FALSE  TRUE FALSE FALSE FALSE
## [481] FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE
## [493] FALSE  TRUE FALSE FALSE FALSE FALSE FALSE  TRUE
```

This returns a new vector in which each element is equal to TRUE if the corresponding element in "binomvec" is equal to 10, and FALSE otherwise. The nice thing is that R considers the value of TRUE to also be equal to 1, and the value of FALSE to also be equal to 0, so we can actually apply the function sum() to this vector!

```
how_many_tens <- sum(verify)
how_many_tens
```

```
## [1] 88
```

Finally, to get at the fraction of all trials that were equal to 10, we simply divide by the number of trials:

```
proportion_of_tens <- how_many_tens / n
proportion_of_tens
```

```
## [1] 0.176
```

You should have gotten a number pretty close to 17.62%. You can imagine that the more trials we perform, the more accurate this number will approximate the exact probability given by the PMF.

Exercise: Try repeating the above procedure but using a different value of k, between 0 and 20. Is your resulting probability lower or higher than P[X=10]?

Exercise: Plot a histogram of the vector "binomvec" using the function hist(). What do you observe?

3.5 The variance

There is another important property of a distribution: its *Variance*. This reflects how much variation we expect to get among different instances of an experiment:

$$Var[X] = E[(X - E[X])^2]$$

The variance is the expectation of $(X - E[X])^2$. This term represents the squared difference between the variable and it expectation, and so the variance is the expected value of this squared difference.

It turns out that the expectation of a function of a random variable is simply the sum of the function of each value the random variable can take, weighted by the probability that the random variable actually takes that value:

$$E[f(x)] = \sum_i f(x_i)P[X = x_i]$$

For a discrete random variable, we can thus write the variance as:

$$Var[X] = \sum_i (x_i - E[X])^2 P[X = x_i]$$

In the particular case of a discrete random variable that follows the Binomial distribution, the variance is a simple function of n and p:

$$Var[X] = np(1 - p)$$

A measurable approximation to the *variance* is called the "sample variance" and can be computed from n samples of an experiment as follows:

$$s = \frac{\sum_{j=1}^{n}(x_j - \bar{x})^2}{n - 1}$$

Just as we can compute the sample mean of a set of trials using the function mean(), we can easily compute the variance of a set of trials using the function var():

```
n <- 5
size <- 100
prob <- 0.5
X <- rbinom(n, size, prob)
X
```

```
## [1] 48 46 52 55 49
```

```
mean(X)
```

```
## [1] 50
```

```
var(X)
```

```
## [1] 12.5
```

Exercise: Compute the variance of a set of 5 Binomial trials of size 100, for different values of the probability of heads. This is equivalent to performing 5 100-toss experiments, with different types of biased coins in each experiment. For what value of the binomial probability is the variance maximized? Does this agree with the variance equation for a binomially-distributed random variable?

Chapter 4

Probability Part 2

Chapter 5

Probability Part 3

Chapter 6

Linear Models

We describe linear models in this chapter. First we need to load some libraries (and install them if necessary).

```
if (!require("tidyverse")) install.packages("tidyverse") # Library for data analysis
```

```
## Loading required package: tidyverse
```

```
## -- Attaching packages --------------------------------------- tidyverse 1.3.0 --
```

```
## v ggplot2 3.3.2     v purrr   0.3.4
## v tibble  3.0.4     v dplyr   1.0.2
## v tidyr   1.1.2     v stringr 1.4.0
## v readr   1.4.0     v forcats 0.5.0
```

```
## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

```
if (!require("stargazer")) install.packages("stargazer") # Library for producing pretty tables o
```

```
## Loading required package: stargazer
```

```
##
## Please cite as:
```

```
##  Hlavac, Marek (2018). stargazer: Well-Formatted Regression and Summary Statistics Tables.
```

```
##  R package version 5.2.2. https://CRAN.R-project.org/package=stargazer
```

```
if (!require("devtools")) install.packages("devtools")
```

```
## Loading required package: devtools
```

```
## Loading required package: usethis
```

```
if (!require("report")) devtools::install_github("easystats/report") # Library for pro
```

```
## Loading required package: report
```

```
## report is in alpha - help us improve by reporting bugs on github.com/easystats/report/is
```

6.1 Fitting a simple linear regression

We'll use a dataset published by Allison and Cicchetti (1976). In this study, the authors studied the relationship between sleep and various ecological and morphological variables across a set of mammalian species: https://science.sciencemag.org/content/194/4266/732

Let's start by loading the data into a table:

```
allisontab <- read.csv("Data_allison.csv")
```

This dataset contains several variables related to various body measurements and measures of sleep in different species. Note that some of these are continuous, while others are discrete and ordinal.

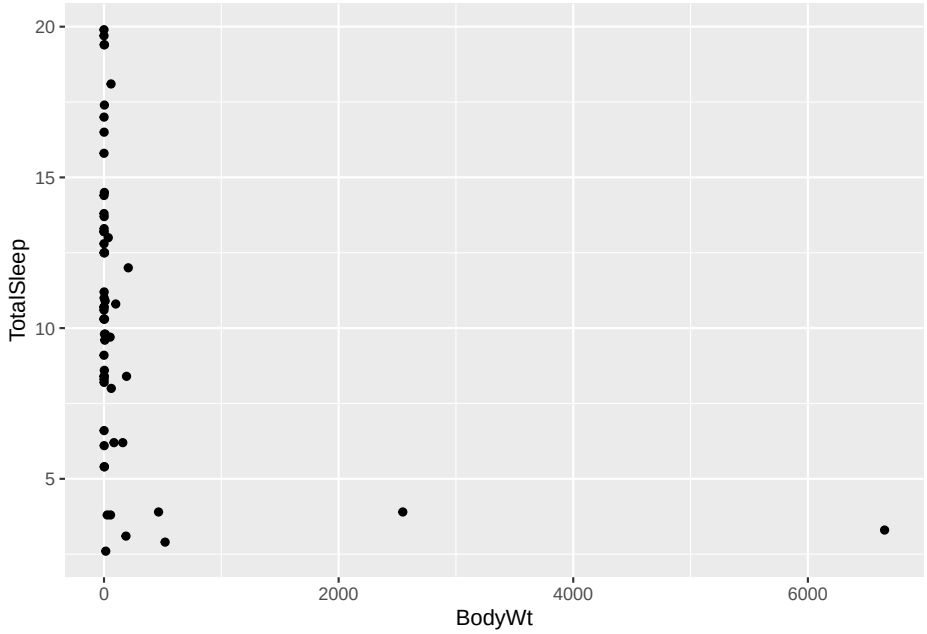
```
summary(allisontab)
```

```
##    Species            BodyWt          BrainWt        NonDreaming
##  Length:62        Min.   :   0.005   Min.   :   0.14   Min.   : 2.100
##  Class :character   1st Qu.:   0.600   1st Qu.:   4.25   1st Qu.: 6.250
##  Mode  :character   Median :   3.342   Median :  17.25   Median : 8.350
##                     Mean   : 198.790   Mean   : 283.13   Mean   : 8.673
##                     3rd Qu.:  48.202   3rd Qu.: 166.00   3rd Qu.:11.000
##                     Max.   :6654.000   Max.   :5712.00   Max.   :17.900
##                                                          NA's   :14
##     Dreaming       TotalSleep       LifeSpan         Gestation
##  Min.   :0.000   Min.   : 2.60   Min.   :  2.000   Min.   : 12.00
##  1st Qu.:0.900   1st Qu.: 8.05   1st Qu.:  6.625   1st Qu.: 35.75
##  Median :1.800   Median :10.45   Median : 15.100   Median : 79.00
##  Mean   :1.972   Mean   :10.53   Mean   : 19.878   Mean   :142.35
##  3rd Qu.:2.550   3rd Qu.:13.20   3rd Qu.: 27.750   3rd Qu.:207.50
##  Max.   :6.600   Max.   :19.90   Max.   :100.000   Max.   :645.00
##  NA's   :12      NA's   :4       NA's   :4         NA's   :4
##    Predation        Exposure         Danger
##  Min.   :1.000   Min.   :1.000   Min.   :1.000
##  1st Qu.:2.000   1st Qu.:1.000   1st Qu.:1.000
##  Median :3.000   Median :2.000   Median :2.000
##  Mean   :2.871   Mean   :2.419   Mean   :2.613
##  3rd Qu.:4.000   3rd Qu.:4.000   3rd Qu.:4.000
##  Max.   :5.000   Max.   :5.000   Max.   :5.000
##
```

We'll begin by focusing on the relationship between two of the continuous variables: body size (in kg) and total amount of sleep (in hours). Let's plot these to see what they look like:

```
ggplot(allisontab) + geom_point(aes(x=BodyWt,y=TotalSleep))
```

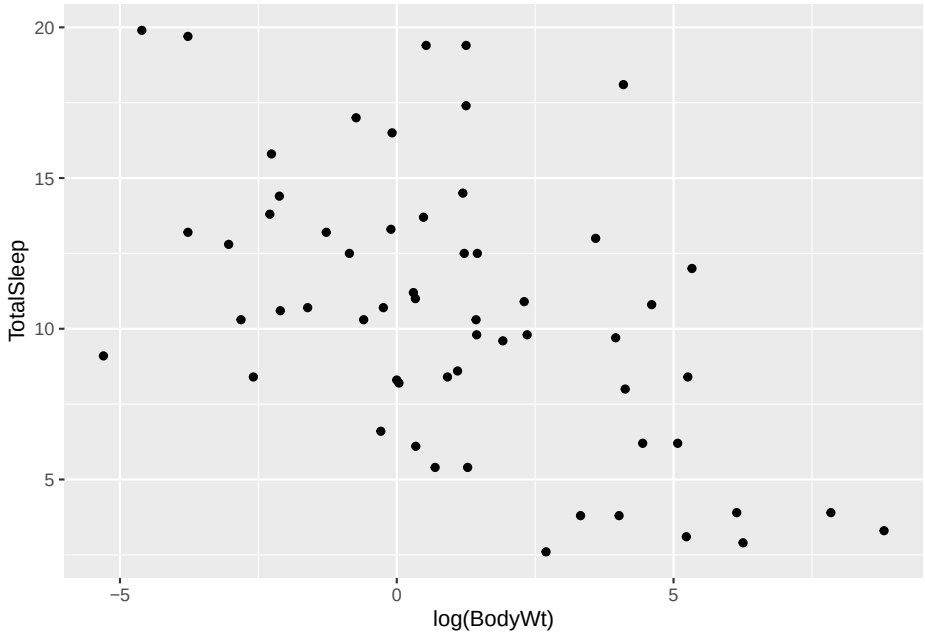
```
## Warning: Removed 4 rows containing missing values (geom_point).
```



Hmmm this looks weird. We have many measurements of body weight around 0 (small values) and a few very large values of thousands of kilograms. This is not surprising: given that this dataset spans several different species, the measurements spans several orders of magnitude (from elephants to molerats). To account for this, variables involving body measurements (like weight or length) are traditionally converted into a log-scale when fitted into a linear model. Let's see what happens when we log-scale the body weight variable:

```
ggplot(allisontab) + geom_point(aes(x=log(BodyWt),y=TotalSleep))
```

```
## Warning: Removed 4 rows containing missing values (geom_point).
```



A pattern appears to emerge now. There seems to be a negative correlation between the log of body weight and the amount of sleep a species has. Indeed, we can measure this correlation using the `cor()` function:

```
cor(log(allisontab$BodyWt), allisontab$TotalSleep, use="complete.obs")
```

```
## [1] -0.5328345
```

Let's build a simple linear model to explain total sleep, as a function of body weight. In R, the standard way to fit a linear model is using the function `lm()`. We do so by following the following formula:

```
fit <- lm(formula, data)
```

The formula within an `lm()` function for a simple linear regression is:

$$y \sim x_1$$

Where y is the response variable and x_1 is the predictor variable. This formula is a shorthand way that R uses for writing the linear regression formula:

$$Y = \beta_0 + \beta_1 x_1 + \epsilon$$

In other words, R implicitly knows that each predictor variable will have an associated β coefficient that we're trying to estimate. Note that here y, x_1, ϵ, etc. represent lists (vectors) of variables. We don't need to specify additional terms for the β_0 (intercept) and ϵ (error) terms. The `lm()` function automatically accounts for the fact that a regression should have an intercept, and that there

will necessarily exist errors (residuals) between our fit and the the observed value of Y.

We can also write this exact same equation by focusing on a single (example) variable, say y_i:

$$y_i = \beta_0 + \beta_1 x_{1,i} + \epsilon_i$$

In general, when we talk about vectors of variables, we'll use boldface, unlike when referring to a single variable.

In our case, we'll attempt to fit total sleep as a function of the log of body weight, plus some noise:

```
myfirstmodel <- lm(TotalSleep ~ log(BodyWt), data=allisontab)
myfirstmodel
```

```
##
## Call:
## lm(formula = TotalSleep ~ log(BodyWt), data = allisontab)
##
## Coefficients:
## (Intercept)  log(BodyWt)
##     11.4377      -0.7931
```

This way, we are fitting the following model:

$$TotalSleep = \beta_0 + \beta_1 log(BodyWt) + \epsilon$$

Remember that the β_0 coefficient is implicitly assumed by the `lm()` function. We can be more explicit and incorporate it into our equation, by simply adding a value of 1 (a constant). This will result in exactly the same output as before:

```
myfirstmodel <- lm(TotalSleep ~ 1 + log(BodyWt), data=allisontab)
myfirstmodel
```

```
##
## Call:
## lm(formula = TotalSleep ~ 1 + log(BodyWt), data = allisontab)
##
## Coefficients:
## (Intercept)  log(BodyWt)
##     11.4377      -0.7931
```

Exercise: the function `attributes()` allows us to unpack all the components of the object outputted by the function `lm()` (and many other objects in R). Try inputting your model output into this function. We can observe that one of the attributes of the object is called `coefficients`. If we type `ourfirstmodel$coefficients`, we obtain a vector with the value of our two fitted coefficients (β_0 and β_1). Using the values from this vector, try plotting

the line of best fit on top of the data. Hint: use the `geom_abline()` function from the `ggplot2` library.

6.2 Interpreting a simple linear regression

We can obtain information about our model's fit using the function `summary()`:

```
summary(myfirstmodel)
```

```
##
## Call:
## lm(formula = TotalSleep ~ 1 + log(BodyWt), data = allisontab)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -6.6990 -2.6264 -0.2441  2.1700  9.9095
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  11.4377     0.5510  20.759  < 2e-16 ***
## log(BodyWt)  -0.7931     0.1683  -4.712 1.66e-05 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 3.933 on 56 degrees of freedom
##   (4 observations deleted due to missingness)
## Multiple R-squared:  0.2839, Adjusted R-squared:  0.2711
## F-statistic:  22.2 on 1 and 56 DF,  p-value: 1.664e-05
```

The `summary()` function provides a summary of the output of `lm()` after it's been given some data and a model to fit. Let's pause and analyze the output here. The first line just re-states the formula we have provided to fit our model. Below that, we get a summary (min, max, median, etc.) of all the residuals (error terms) between our linear fit and the observed values of $TotalSleep$.

Below that, we can see a table with point estimates, standard errors, and a few other properties of our estimated coefficients: the intercept (β_0, first line) and the slope (β_1, second line). The standard error is a measure of how confident we are about our point estimate (we'll revisit this in later lectures). The "t value" corresponds to the statistic for a "t-test" which serves to determine whether the estimate can be considered as significantly different from zero. The last column is the P-value from this test. We can see that both estimates are quite significantly different from zero ($P < 0.001$), meaning we can reject the hypothesis that these estimates are equivalent to zero.

Finally, the last few lines are overall measures of the fit of the model. 'Multiple R-squared' is the fraction of the variance in $TotalSleep$ explained by the fitted

model. Generally, we want this number to be high, but it is possible to have very complex models with very high R-squared but lots of parameters, and therefore we run the risk of "over-fitting" our data. 'Adjusted R-squared' is a modified version of R-squared that attempts to penalize very complex models. The 'residual standard error' is the sum of the squares of the residuals (errors) over all observed data points, scaled by the degrees of freedom of the linear model, which is equal to n − k − 1 where n = total observations and k = total model parameters. Finally, the F-statistic is a test for whether *any* of the explanatory variables included in the model have a relationship to the outcome. In this case, we only have a single explanatory variable ($log(BodyWt)$), and so the P-value of this test is simply equal to the P-value of the t-test for the slope of $log(BodyWt)$.

We can use the function `report()` from the library `easystats` (https://github.com/easystats/report) to get a more verbose report than the `summary()` function provides.

```
report(myfirstmodel)
```

```
## Formula contains log- or sqrt-terms. See help("standardize") for how such terms are standardized.
## Formula contains log- or sqrt-terms. See help("standardize") for how such terms are standardized.

## We fitted a linear model (estimated using OLS) to predict TotalSleep with BodyWt (formula: TotalSle
##
##   - The effect of BodyWt [log] is significantly negative (beta = -0.79, 95% CI [-1.13, -0.46], t(56)
##
## Standardized parameters were obtained by fitting the model on a standardized version of the dataset
```

Note that this function "standardizes" the input variables before providing a summary of the output, which makes the estimates' value to be slightly different than those stored in the output of `lm()`. This makes interpretation of the coefficients easier, as they are now expressed in terms of standard deviations from the mean.

Another way to summarize our output is via a summary table in , which can be easily constructed using the function `stargazer()` from the library `stargazer` (https://cran.r-project.org/web/packages/stargazer/index.html).

```
stargazer(myfirstmodel, type="text")
```

```
##
## ================================================
##                          Dependent variable:
##                      ---------------------------
##                              TotalSleep
## ------------------------------------------------
## log(BodyWt)                   -0.793***
##                                (0.168)
##
```

```
## Constant                      11.438***
##                                (0.551)
##
## ------------------------------------------------
## Observations                     58
## R2                              0.284
## Adjusted R2                     0.271
## Residual Std. Error        3.933 (df = 56)
## F Statistic           22.203*** (df = 1; 56)
## ================================================
## Note:                *p<0.1; **p<0.05; ***p<0.01
```

This package also supports LaTeX and HTML/CSS format (see the `type` option in `?stargazer`), which makes it very handy when copying the output of a regression from R into a working document.

Exercise: try fitting a linear model for $TotalSleep$ as a function of brain weight ($BrainWt$). Keep in mind that this is a size measurement that might span multiple orders of magnitude, just like body weight. What are the estimated slope and intercept coefficients? Which coefficients are significantly different from zero? What is the proportion of explained variance? How does this compare to our previous model including $BodyWt$?

Exercise: Plot the linear regression line of the above exercise on top of your data.

6.3 Simulating data from a linear model

It is often useful to simulate data from a model to understand how its parameters relate to features of the data, and to see what happens when we change those parameters. We will now create a function that can simulate data from a simple 1-parameter linear model. We will then feed this function different values of the parameters, and see what the data simulated under a given model looks like.

Let's start by first creating the simulation function. We'll simulate data from a linear model. The model simulation function needs to be told: 1) The number (n) of data points we will simulate 1) How the explanatory variables are distributed: we'll use a normal distribution to specify this. 2) What the intercept (β_0) and slope (β_1) for the linear relationship between the explanatory and response variables are 3) How departures (errors) from linearity for the response variables will be modeled: we'll use another normal distribution for that as well, and assume errors are heteroscedastic for now.

```
linearmodsim <- function(n=2, beta_0=0, beta_1=1, sigma.res=1, mu.explan=5, sigma.expla
  # Simulate explanatory variables
  explan <- r_explan(n,mu.explan,sigma.explan)
  # Sort the simulated explanatory values from smallest to largest
```

```
  explan <- sort(explan)
  # Standardize the response variables so that they are  mean-centered and scaled by their standa
  explan.scaled <- scale(explan)
  # OPTIONAL: If errors are not heteroscedastic (hetero does not equal 0), then their standard de
  sdev.err <- sapply(sigma.res + explan.scaled*hetero,max,0)
  # Simulate the error values using the above-specified standard deviation
  error <- rerror(n,0,sdev.err)
  # Simulate response variables via the linear model
  response <- beta_0 + beta_1 * explan + error
  # Output a table containing the explanatory values and their corresponding response values
  cbind(data.frame(explan,response))
}
```

Exercise:

a) Carefully read the code for the function above. Make sure you understand every step in the function.

b) Plot the output of a simulated linear model with 40 data points, an intercept of 1.5 and a slope of 3. Simulate from the same model one more time, and plot the output again.

c) Now, fit the data from your latest simulation using the `lm()` function. Does your fit match your simulations?

d) Try increasing the sample size (say, to 200 data points), and repeat the `lm()` fitting. How does this influence the accuracy of your fitted model? Try simulating and fitting multiple times to get an idea of how well you can estimate the parameters.

e) Try changing the standard deviation of the simulated errors (make it smaller or larger), and repeat the `lm()` fitting. How does this influence the accuracy of your fitted model?

Chapter 7

Properties of Estimators and Inference

7.1 Properties of point estimators

7.2 Builidng confidence intervals

7.3 ROC curve

Chapter 8

Frequentist inference

Chapter 9

Bayesian Inference

Chapter 10

Classification

Chapter 11

Model Assessment

Chapter 12

Resampling

```
library("tidyverse")
```

12.1 The bootstrap

We'll work with a subset of the Allison et al. data. We'll start by using the body and brain weight measurements from all the species, after log-scaling them. Later on, we'll also use the TotalSleep variable as well, so let's remove any rows that have missing data for any of these 3 variables.

```
# Load table
allisontab <- tibble(read.csv("Data_allison.csv"))
# Remove rows with missing data in columns of interest
allisontab <- filter(allisontab,!is.na(BrainWt) & !is.na(BodyWt) & !is.na(TotalSleep))
# Log-scale body and brain weight
allisontab <- mutate(allisontab,logBody=log10(BodyWt), logBrain=log10(BrainWt))
```

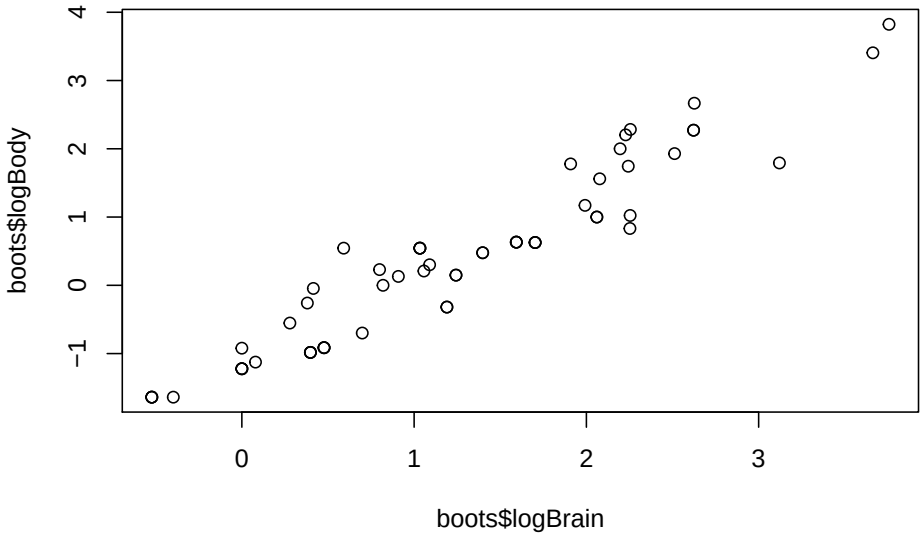
Below is a function to obtain a single bootstrapped sample from an input data. Take a close look at each step.

```
bootstrap <- function(tab){
  # Preliminary check: if the table is a vector with a single variable, turn it into a matrix
  if(is.null(dim(tab))){tab <- matrix(tab,ncol=1)}
  # Count the number of elements in our data
  numelem <- nrow(tab)
  # Sample indexes with replacement
  bootsidx <- sample(1:numelem, replace=TRUE)
  # Obtain a boostrapped sample by selecting the bootstrapped indexes from the original sample
  final <- tab[bootsidx,]
  # Produce bootstrapped sample as output
```

```
  return(final)
}
```

Let's see what happens when we run this function on our data.

```
boots <- bootstrap(allisontab)
plot(boots$logBrain,boots$logBody)
```



Repeat the above command lines multiple times. What happens?

Let's estimate a parameter: the slope coefficient in a linear regression of log brain weight on log body weight:

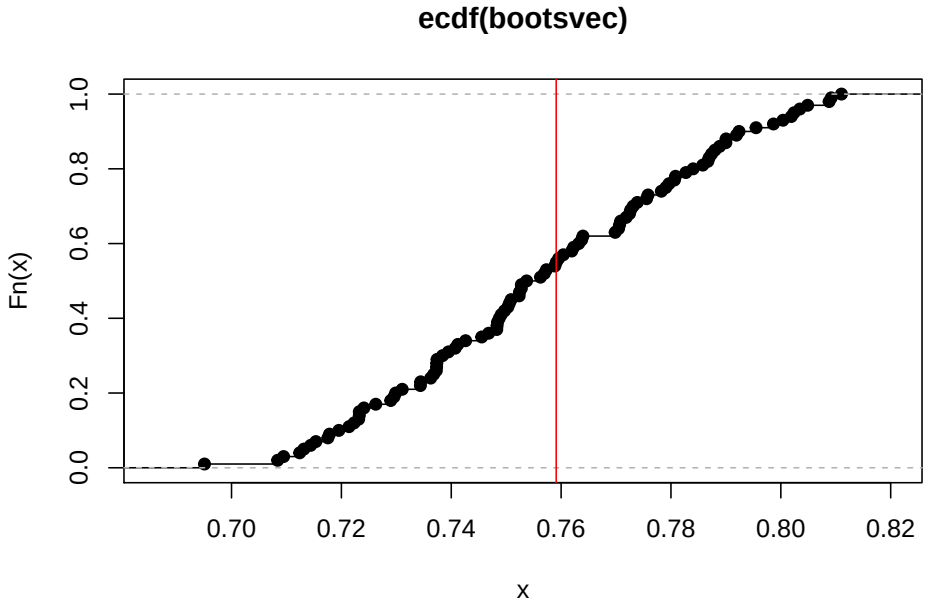
```
estimate <- lm(logBrain ~ logBody, data=allisontab)$coeff[2]
estimate
```

```
##   logBody
## 0.7591064
```

Exercise: try estimating the same parameter from a series of 100 bootstrapped samples of our original data, and collecting each of the bootstrapped parameters into a vector called "bootsvec". Hint: you might want to use a for loop or a vectorized sapply() function.

Let's plot the ecdf of all our estimates, using the function ecdf().

```
plot(ecdf(bootsvec))
abline(v=estimate,col="red")
```



We are now ready to obtain confidence intervals (CIs) of our original parameter estimate, using our bootstrapped distribution. There are multiple ways to obtain CIs from a bootstrapped distribution. Some of these assume that the ECDF has particular properties, while others are more generally applicable:
a) Standard error approach - assumes ECDF is normal
b) Percentile approach - assumes ECDF is symmetric and median-unbiased
c) Pivotal approach - most general, makes very few assumptions.
These three approaches generally result in very similar CIs, but they differ (slightly) in methodology. The most widely used method is the pivotal approach, though the motivation for its construction is a bit long-winded. In the interest of time, we'll demonstrate how to run the first two approaches in R. We'll leave the third approach as an exercise you can do at home (read Box 8-1 in the Edge book for an explanation of it, and a code example).

12.2 Permutation test

Let's evaluate the relationship that there is no relationship between logBrain and logBody. Recall that one way to do it would be by using a linear model, and testing whether the value of the fitted slope is significantly different from zero, using a t-test:

```
summary(lm(logBrain ~ logBody, data=allisontab))
```

```
##
## Call:
## lm(formula = logBrain ~ logBody, data = allisontab)
```

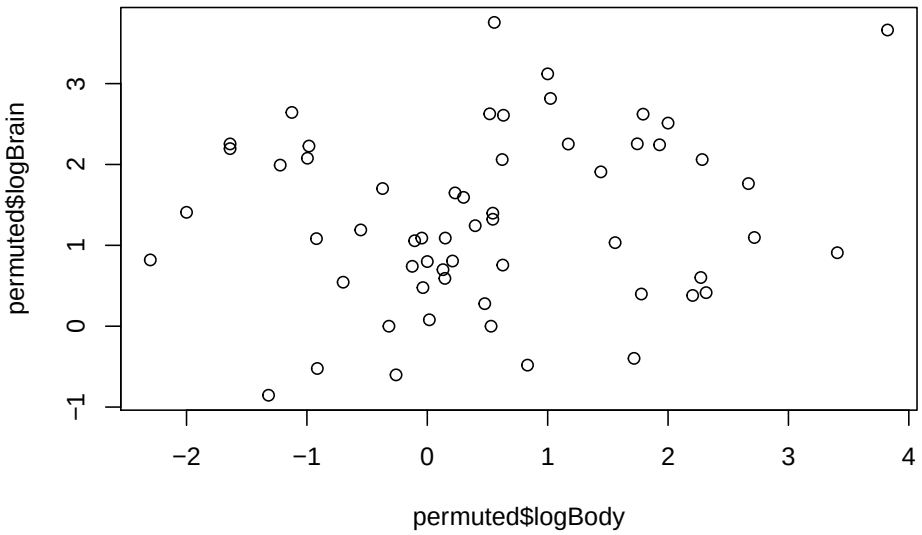
```
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -0.75701 -0.21266 -0.03618  0.19059  0.82489
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  0.93507    0.04302   21.73   <2e-16 ***
## logBody      0.75911    0.03026   25.09   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.3071 on 56 degrees of freedom
## Multiple R-squared:  0.9183, Adjusted R-squared:  0.9168
## F-statistic: 629.2 on 1 and 56 DF,  p-value: < 2.2e-16
```

This test, however, makes assumptions on our data that sometimes may not be warranted, like large sample sizes and homogeneity of variance. We can perform a more general test that makes less a priori assumptions on our data - a permutation test - as long as we are careful in permuting the appropriate variables for the relationship we are trying to test. In this case, we only have two variables, and we are trying to test whether there is a significant relationship between them. If we randomly shuffle one variable with respect to the other, we should obtain a randomized sample of our data. We can use the following function, which takes in a tibble and a variable of interest, and returns a new tibble in which that particular variable's values are randomly shuffled.

```
permute <- function(tab,vartoshuffle){
  # Extract column we wish to shuffle as a vector
  toshuffle <- unlist(tab[,vartoshuffle],use.names=FALSE)
  # The function sample() serves to randomize the order of elements in a vector
  shuffled <- sample(toshuffle)
  # Replace vector in new table (use !! to refer to a dynamic variable name)
  newtab <- mutate(tab, !!vartoshuffle := shuffled )
  return(newtab)
}
```

Now we can obtain a permuted version of our original data, and compute the slope estimate on this dataset instead:

```
permuted <- permute(allisontab, "logBrain")
plot(permuted$logBody,permuted$logBrain)
```




```
permest <- lm(logBrain ~ logBody, data=permuted)$coeff[2]
permest
```

```
##   logBody
## 0.1269916
```

Exercise: try estimating the same parameter from a series of 100 permuted versions of our original data, and collecting each of the permuted parameters into a vector called "permvec".

We now have a distribution of the parameter estimate under the assumption that there is no relationship between these two variables:

Exercise: obtain an empirical one-tailed P-value from this distribution by counting how many of the permuted samples are as large as our original estimate, and dividing by the total number of permuted samples we have. Note: you should add a 1 to both the denominator and the numerator of this ratio, in case there are no permuted samples that are as large as the original estimate, so as not to get an infinite number.

12.3 Validation

We'll perform a validation exercise to evaluate the error of various models on the data. In this case, we'll create a predictor for TotalSleep as a function of logBody, using a linear model, and then test how well it does. We'll first divide our data into a "training" partition - which we'll use to fit our model - and a separate "test" partition - which we'll use to test how well our model is doing, and avoid over-fitting. Each partition will be one half of our original data.

```
# Obtain the number of data points we have
numdat <- dim(allisontab)[1]
# For the training set, randomly sample 50% of the data indexes
trainset <- sample(numdat, round(numdat*0.5))
# For the test set, obtain all indexes that are not in training set
testset <- seq(1,numdat)[-trainset]
```

Let's begin by calculating the mean squared error (MSE) between our observatiosn and our predictions in our test partition, after fitting a linear model to our training partition:

```
# Fit the linear model to the training subset of the data
fit1 <- lm(TotalSleep ~ logBody, data=allisontab,subset=trainset)
# Predict all observations using the fitted linear model
predall <- predict(fit1,allisontab)
# Compute mean squared differences between observations and predictions
sqdiff <- (allisontab$logBrain - predall)^2
# Extract the differences for the test partition
sqdiff.test <- sqdiff[testset]
# Compute the mean squared error
mse1 <- mean(sqdiff.test)
```

Now, we'll try to fit our data to two more complex models: a quadratic model and a cubic model, using the function `poly`:

```
fit2 <- lm(TotalSleep ~ poly(logBody,2), data=allisontab,subset=trainset)
mse2 <- mean(((allisontab$logBrain - predict(fit2,allisontab))^2)[testset])

fit3 <- lm(TotalSleep ~ poly(logBody,3), data=allisontab,subset=trainset)
mse3 <- mean(((allisontab$logBrain - predict(fit3,allisontab))^2)[testset])
```

We can see that the MSE appears to increase for the more complex models. This suggests a simple linear fit performs better at predicting values that were not included in the training set.

Exercise: compute the MSE on the training partition. Compare the resulting values to the MSE on the test partition. What do you observe? Is the difference in errors between the three models as large as when computing the MSE on the test partition? Why do you think this is?

12.4 Cross-validation

We'll now perform a cross-validation exercise. If you haven't installed it, you'll need to install the library `boot` before loading it.

```
if(require("boot") == FALSE){install.packages("boot")}
```

```
## Loading required package: boot
```

```
library("boot")
```

The function `cv.glm()` from the library `boot` can be used to compute a cross-validation error. This function is designed to work with the `glm()` function for fitting generalized linear models in R, but we can compute a simple linear regression using `glm()` as well, and then feed the result into `cv.glm()`:

```
fit1=glm( TotalSleep ~ logBody, data=allisontab )
# The LOOCV error is computed using the function cv.glm()
cv.err=cv.glm(allisontab,fit1)
```

The value of the cross-validation error is stored in the second element of the attribute `delta` of the output of `cv.glm`. By default, this is a "leave-one-out" cross-validation (LOOCV) error, meaning it computes error by leaving 1 data point out of the fitting and evaluating the error at that data point. The process is iterated over all data points, and the errors are then averaged together. We can obtain the value of the LOOCV error by writing:

```
cv.err$delta[1]
```

```
## [1] 15.97798
```

Now, let's compute the LOOCV error for increasingly complex polynomial models (linear, quadratic, cubic, etc.):

```
CVerr=rep(0,5)
for(m in 1:5){
  fit=glm(TotalSleep ~ poly(logBody,m), data=allisontab)
  CVerr[m]=cv.glm(allisontab,fit)$delta[1]
}
```

Exercise: Plot the results of this cross-validation exercise. Which model has the lowest LOOCV error?

Exercise: Take a look at the help function for `cv.glm`. Which argument would you modify to be able to compute the 10-fold cross-validation error, instead of the LOOCV error. Can you do this for the five models we tested above?

Chapter 13

Mixed Models

Chapter 14

Ordination

14.1 Libraries and Data

Today, we will work with the package vegan (useful for ordination techniques) and the packages ggplot2 and ggbiplot (useful for fancy plotting). Make sure all these libraries are installed before you begin.

Let's begin by installing and loading the necessary libraries:

```
if (!require("vegan")) install.packages("vegan")
```

```
## Loading required package: vegan
```

```
## Loading required package: permute
```

```
##
## Attaching package: 'permute'
```

```
## The following object is masked _by_ '.GlobalEnv':
##
##     permute
```

```
## The following object is masked from 'package:recipes':
##
##     check
```

```
## The following object is masked from 'package:devtools':
##
##     check
```

```
## Loading required package: lattice
```

```
##
## Attaching package: 'lattice'
```

```
## The following object is masked from 'package:boot':
##
##     melanoma
```

```
## Registered S3 method overwritten by 'vegan':
##   method          from
##   print.nullmodel parsnip
```

```
## This is vegan 2.5-6
```

```
##
## Attaching package: 'vegan'
```

```
## The following object is masked from 'package:parsnip':
##
##     nullmodel
```

```
if (!require("devtools")) install.packages("devtools")
if (!require("ggplot2")) install.packages("ggplot2")
if (!require("ggbiplot")){ library("devtools"); install_github("vqv/ggbiplot") }
```

```
## Loading required package: ggbiplot
```

```
## Loading required package: plyr
```

```
## ------------------------------------------------------------------------------
```

```
## You have loaded plyr after dplyr - this is likely to cause problems.
## If you need functions from both plyr and dplyr, please load plyr first, then dplyr:
## library(plyr); library(dplyr)
```

```
## ------------------------------------------------------------------------------
```

```
##
## Attaching package: 'plyr'
```

```
## The following objects are masked from 'package:dplyr':
##
##    arrange, count, desc, failwith, id, mutate, rename, summarise,
##     summarize
```

```
## The following object is masked from 'package:purrr':
##
##     compact
```

```
## Loading required package: grid
```

We will use a dataset on measurements of particular parts of the iris plant, across individuals from three different species.

```
data(iris)
```

Exercise: Take a look at the iris data matrix. How many samples does it have? How many variables? What happens when you run the function `plot()` on this matrix? Which variables seem to be strongly correlated? (you can use the

function cor() to compute the strength of correlations). Speculate as to why some of these variables could be strongly correlated.

14.2 Principal component analysis (PCA)

We'll perform a PCA of the data. The function prcomp() performs the PCA, and we can assign the result of this function to a new variable (let's call it "fit"). We must first remove the last column to whatever we give as input to prcomp, as the species names are a non-linear (categorical) variable and we don't have (for now) any natural measures of distance for species. The option scale=T standardizes the variables to the same relative scale, so that some variables do not become dominant just because of their large measurement unit.

```
fit<-prcomp(iris[-5], scale=TRUE)
```

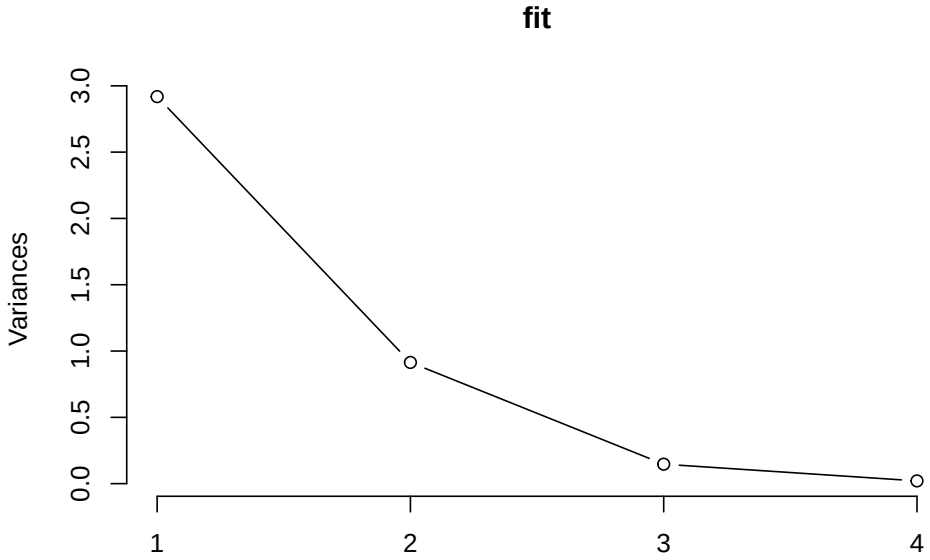
If we run the summary function on fit, it indicates that four PCs where created: the number of possible PCs always equals the number of original variables.

```
summary(fit)
```

```
## Importance of components:
##                           PC1    PC2     PC3     PC4
## Standard deviation     1.7084 0.9560 0.38309 0.14393
## Proportion of Variance 0.7296 0.2285 0.03669 0.00518
## Cumulative Proportion  0.7296 0.9581 0.99482 1.00000
```

How much of the variance is explained by PC1? How much is explained by PC2?

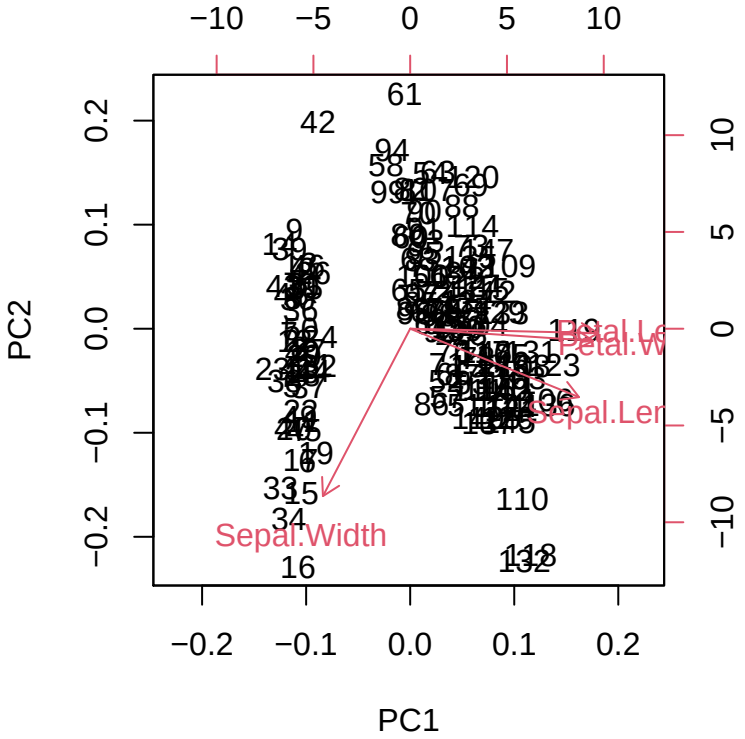
```
plot(fit,type="lines")
```



The "Rotation" matrix (fit[2]) contains the "loadings" of each of the original variables on the newly created PCs. Take a look at this matrix. The larger the absolute value of a variable in each PC, the more that variable contributes to that PC.

We can use the function "biplot" to plot the first two PCs of our data. The plotted arrows provide a graphical rendition of the loadings of each of the original variables on the two PCs.

```
biplot(fit)
```

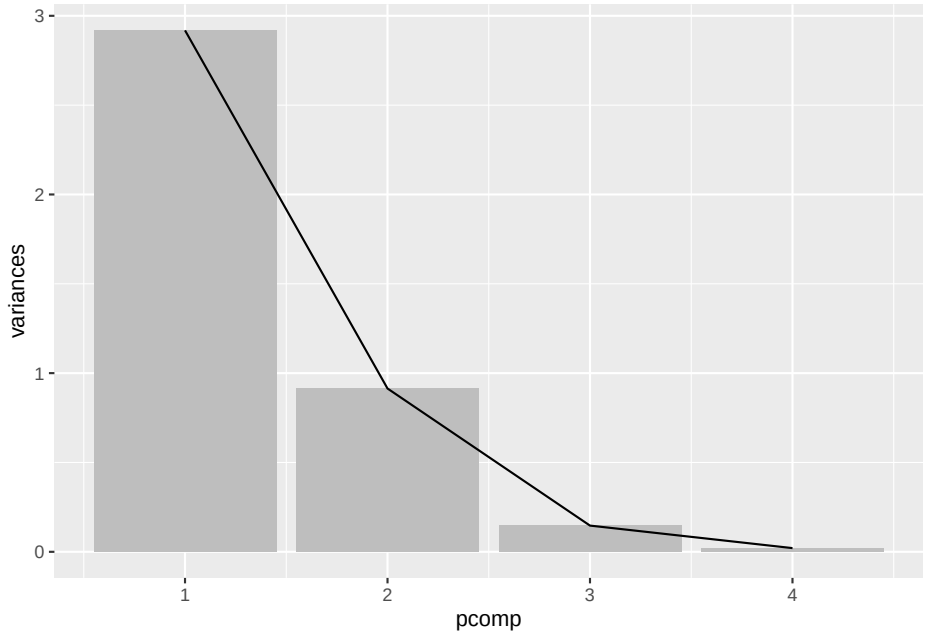


Exercise: Across this reduced dimensional space, we can see that particular variables tend to co-vary quite strongly. Which ones? We can also see a separation into two groups on PC1. Which variables do you think would be most different between samples in one group and in the other?

We can make prettier plots using ggplot2 and ggbiplot.

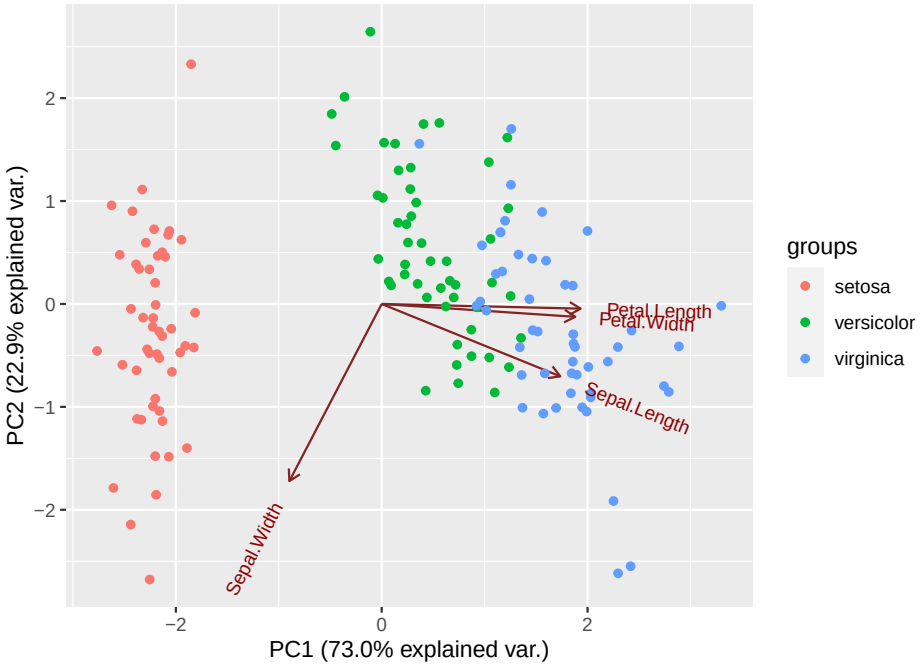
We first extract the variances of the principal components and then plot them:

```
variances <- data.frame(variances=fit$sdev**2, pcomp=1:length(fit$sdev))
varPlot <- ggplot(variances, aes(pcomp, variances)) + geom_bar(stat="identity", fill="gray") + ge
varPlot
```



We can also plot the first two PCs, like we had done before in base R, but now coloring the samples by their corresponding species. How are the species distributed along PC1?

```
Species<-iris$Species
iris_pca <- ggbiplot(fit,obs.scale = 1,
         var.scale=1,groups=Species,ellipse=F,circle=F,varname.size=3)
iris_pca
```



14.3 PCA under the hood

Rather than just using a ready-made function to compute a PCA, let's take a longer route to understand exactly what's happening under the hood of the prcomp() function.

First, let's standardize each column of our data so that each column has mean 0 and variance 1

```
irisdat <- iris[-5]
irisstandard <- apply(irisdat,2,function(x){(x-mean(x))/sd(x)})
```

Now, calculate the covariance matrix. Because the data has been standardized, this is equivalent to calculating the correlation matrix of the pre-standardized data.

```
cormat <- cov(irisstandard)
```

Then, extract the eigenvalues and eigenvectors of correlation matrix:

```
myEig <- eigen(cormat)
```

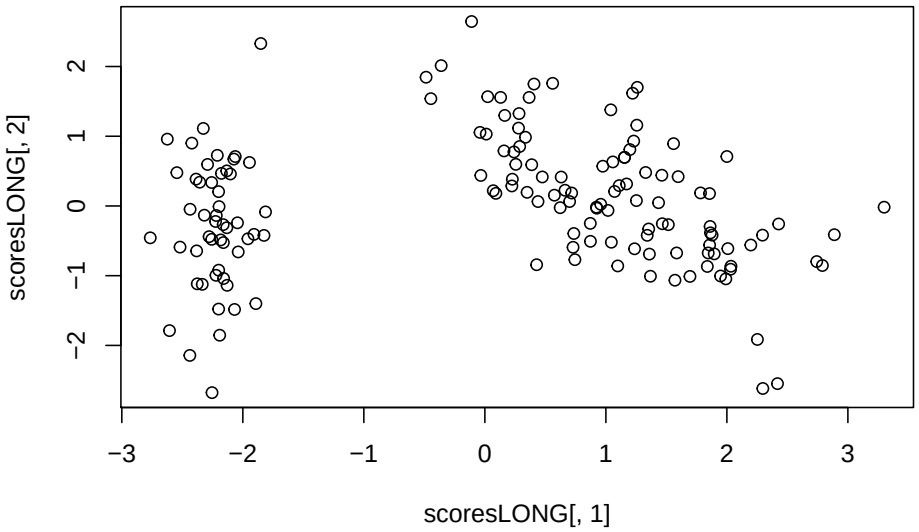
Now, we'll manually obtain certain values that were automatically computed by the prcomp function when we ran it earlier. We'll calculate the singular values (square root of eigenvalues) and also obtain the eigenvectors, also called

loadings.

```
sdLONG <- sqrt(myEig$values)
loadingsLONG <- myEig$vectors
rownames(loadingsLONG) <- colnames(irisstandard)
```

Using the loadings, we can plot our original (standardized) data matrix into the new PC-space, by multiplying the data matrix by the matrix of loadings. Plotting the first two rows of the resulting product should reveal the location of our data points in the first two principal components (like we had before):

```
scoresLONG <- irisstandard %*% loadingsLONG
plot(scoresLONG[,1],scoresLONG[,2])
```



You can compare the results from the first section (using the ready-made function prcomp) and this section (taking a longer road), to check that the results are equivalent: the minimum and maximum differences in values for the loadings, the scores and the standard deviations of the PCs are all infinitesimally small (effectively zero).

```
range(fit$sdev - sdLONG)
```

```
## [1] -6.661338e-16  2.220446e-16
```

```
range(fit$rotation - loadingsLONG)
```

```
## [1] -6.661338e-16  7.771561e-16
```

```
range(fit$x - scoresLONG)
```

```
## [1] -2.359224e-15  3.108624e-15
```

14.4 NMDS

We'll now perform non-metric multidimensional scaling. Let's first take a look at the raw data we will use. This is a data matrix containing information about dune meadow vegetation. There are 30 species and 20 sites. Each cell corresponds to the number of specimens of a particular species that has been observed at a particular site (Jongman et al. 1987). As one can see, there are many sites where some species are completely absent (the cell value equals 0):

```
data(dune)
dune
```

```
##   Achimill Agrostol Airaprae Alopgeni Anthodor Bellpere Bromhord Chenalbu
## 1        1        0        0        0        0        0        0        0
## 2        3        0        0        2        0        3        4        0
## 3        0        4        0        7        0        2        0        0
## 4        0        8        0        2        0        2        3        0
## 5        2        0        0        0        4        2        2        0
## 6        2        0        0        0        3        0        0        0
## 7        2        0        0        0        2        0        2        0
## 8        0        4        0        5        0        0        0        0
## 9        0        3        0        3        0        0        0        0
## 10       4        0        0        0        4        2        4        0
## 11       0        0        0        0        0        0        0        0
## 12       0        4        0        8        0        0        0        0
## 13       0        5        0        5        0        0        0        1
## 14       0        4        0        0        0        0        0        0
## 15       0        4        0        0        0        0        0        0
## 16       0        7        0        4        0        0        0        0
## 17       2        0        2        0        4        0        0        0
## 18       0        0        0        0        0        2        0        0
## 19       0        0        3        0        4        0        0        0
## 20       0        5        0        0        0        0        0        0
##   Cirsarve Comapalu Eleopalu Elymrepe Empenigr Hyporadi Juncarti Juncbufo
## 1        0        0        0        4        0        0        0        0
## 2        0        0        0        4        0        0        0        0
## 3        0        0        0        4        0        0        0        0
## 4        2        0        0        4        0        0        0        0
## 5        0        0        0        4        0        0        0        0
## 6        0        0        0        0        0        0        0        0
## 7        0        0        0        0        0        0        0        2
## 8        0        0        4        0        0        0        4        0
## 9        0        0        0        6        0        0        4        4
## 10       0        0        0        0        0        0        0        0
## 11       0        0        0        0        0        2        0        0
## 12       0        0        0        0        0        0        0        4
```

```
## 13       0       0       0       0       0       0       0       3
## 14       0       2       4       0       0       0       0       0
## 15       0       2       5       0       0       0       3       0
## 16       0       0       8       0       0       0       3       0
## 17       0       0       0       0       0       2       0       0
## 18       0       0       0       0       0       0       0       0
## 19       0       0       0       0       2       5       0       0
## 20       0       0       4       0       0       0       4       0
##    Lolipere Planlanc Poaprat Poatriv Ranuflam Rumeacet Sagiproc Salirepe
## 1         7       0       4       2       0       0       0       0
## 2         5       0       4       7       0       0       0       0
## 3         6       0       5       6       0       0       0       0
## 4         5       0       4       5       0       0       5       0
## 5         2       5       2       6       0       5       0       0
## 6         6       5       3       4       0       6       0       0
## 7         6       5       4       5       0       3       0       0
## 8         4       0       4       4       2       0       2       0
## 9         2       0       4       5       0       2       2       0
## 10        6       3       4       4       0       0       0       0
## 11        7       3       4       0       0       0       2       0
## 12        0       0       0       4       0       2       4       0
## 13        0       0       2       9       2       0       2       0
## 14        0       0       0       0       2       0       0       0
## 15        0       0       0       0       2       0       0       0
## 16        0       0       0       2       2       0       0       0
## 17        0       2       1       0       0       0       0       0
## 18        2       3       3       0       0       0       0       3
## 19        0       0       0       0       0       0       3       3
## 20        0       0       0       0       4       0       0       5
##    Scorautu Trifprat Trifrepe Vicilath Bracruta Callcusp
## 1         0        0        0        0        0        0
## 2         5        0        5        0        0        0
## 3         2        0        2        0        2        0
## 4         2        0        1        0        2        0
## 5         3        2        2        0        2        0
## 6         3        5        5        0        6        0
## 7         3        2        2        0        2        0
## 8         3        0        2        0        2        0
## 9         2        0        3        0        2        0
## 10        3        0        6        1        2        0
## 11        5        0        3        2        4        0
## 12        2        0        3        0        4        0
## 13        2        0        2        0        0        0
## 14        2        0        6        0        0        4
## 15        2        0        1        0        4        0
## 16        0        0        0        0        4        3
```

```
## 17        2        0        0        0        0        0
## 18        5        0        2        1        6        0
## 19        6        0        2        0        3        0
## 20        2        0        0        0        4        3
```

Note that this data is non-linear, so our first instinct should not be to perform PCA on it. Because NMDS relies on "distances", we need to specify a distance metric that we'll use. The function for performing NMDS in the package 'vegan' is called metaMDS() and its default distance metric is "bray", which corresponds to the Bray-Curtis dissimilarity: a statistic used to quantify the compositional dissimilarity between two different sites, based on counts at each site

Let's perform NMDS ordination using the Bray-Curtis dissimilarity. Remember that, unlike PCA, NMDS requires us to specify the number of dimensions (k) a priori (the default in vegan is 2). It also performs a series of transformations on the data that are appropriate for ecological data (default: autotransform=TRUE). The trymax option ensures that the algorithm is started from different points (in our case, 50) to avoid local minima.

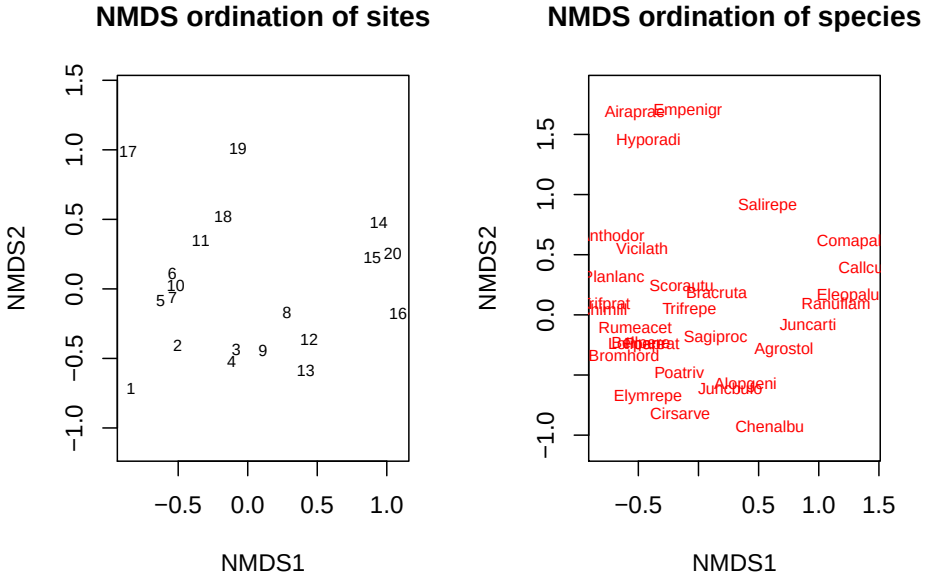
```
ord <- metaMDS(dune, k=2, autotransform = TRUE, trymax=50)
```

```
## Run 0 stress 0.1192678
## Run 1 stress 0.1192681
## ... Procrustes: rmse 0.0002886995  max resid 0.0008843855
## ... Similar to previous best
## Run 2 stress 0.1183186
## ... New best solution
## ... Procrustes: rmse 0.02027201  max resid 0.06496324
## Run 3 stress 0.1889653
## Run 4 stress 0.2075713
## Run 5 stress 0.1183186
## ... New best solution
## ... Procrustes: rmse 1.602892e-05  max resid 4.319392e-05
## ... Similar to previous best
## Run 6 stress 0.1183186
## ... Procrustes: rmse 1.078983e-05  max resid 3.531281e-05
## ... Similar to previous best
## Run 7 stress 0.1809578
## Run 8 stress 0.1183186
## ... Procrustes: rmse 1.790996e-05  max resid 5.68235e-05
## ... Similar to previous best
## Run 9 stress 0.1192684
## Run 10 stress 0.1192678
## Run 11 stress 0.1192678
## Run 12 stress 0.1183186
## ... Procrustes: rmse 2.814207e-05  max resid 9.006326e-05
## ... Similar to previous best
```

```
## Run 13 stress 0.1192682
## Run 14 stress 0.1192685
## Run 15 stress 0.1192678
## Run 16 stress 0.1192679
## Run 17 stress 0.1192679
## Run 18 stress 0.1183186
## ... Procrustes: rmse 3.520508e-05  max resid 0.000108158
## ... Similar to previous best
## Run 19 stress 0.2930675
## Run 20 stress 0.1192678
## *** Solution reached
```

As you can see, the function goes through a series of steps until convergence is reached. Let's plot the results:

```
par(mfrow=c(1,2))
plot(ord,display="sites",main="NMDS ordination of sites",type="t")
plot(ord,display="species",main="NMDS ordination of species",type="t")
```




```
par(mfrow=c(1,1))
```

Exercise: What do these plots tell you about the distribution of species across sites? Which species tend to co-occcur with each other? Which sites tend to have similar species compositions?

Exercise: Try changing the number of dimensions or the distance metric used. You can take a look at the list of possible distances and their definitions using "?vegdist". Do the results change? Why?

Chapter 15

Clustering

Today, we'll perform a clustering of the iris dataset, using the K-means clustering method.

15.1 Libraries and Data

We'll use the following libraries for clustering:

```
if (!require("cluster")) install.packages("cluster")
```

```
## Loading required package: cluster
```

```
if (!require("NbClust")) install.packages("NbClust")
```

```
## Loading required package: NbClust
```

```
if (!require("tidyverse")) install.packages("tidyverse")
```

Now, load the iris dataset using the `data()` function.

```
data(iris)
```

15.2 Distances

The variables we will use to cluster our data are the four flower measurements in the iris dataset. They all represent the measured length of a segment between two points (e.g. sepal length, petal width), the Euclidean distance is an obvious choice of distance for clustering our observation. The clustering method we will apply to our data (and the ordination methods we applied before) implicitly use the function `dist()` to calculate distances between observations.

```
d <- dist(iris[,1:4], method="euclidean")
d <- as.matrix(d)
```

Exercise: Under the chosen distance metric, what is the most disparate observation from observation 1? What is the distance between this observation and observation 1? Do they belong to different species?

Exercise: Under the chosen distance metric, what is the pair of observations that are most disparate from each other, among all pairs? What is the distance between them? Do they belong to different species?

15.3 K-means clustering

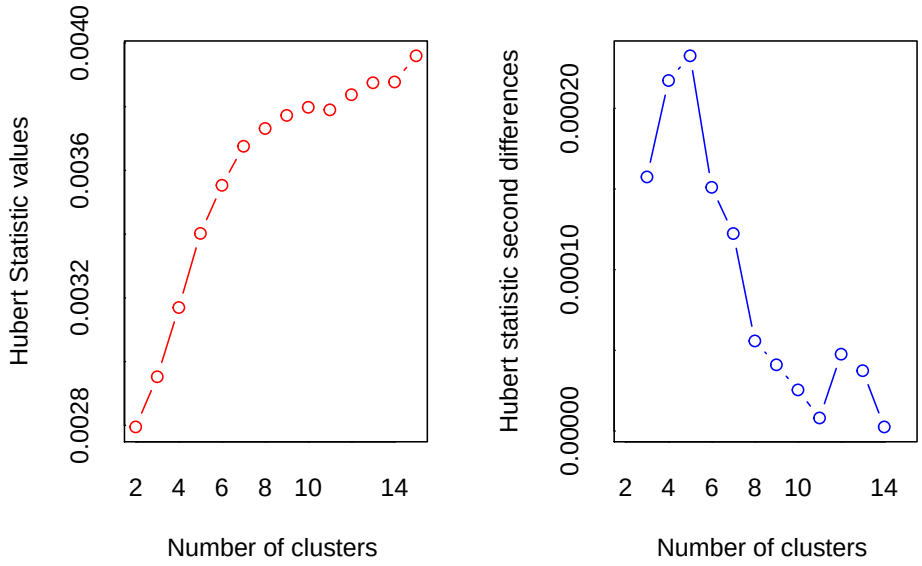
Ok, we're almost ready to cluster our data. There are just a few preliminary details to keep in mind. The first of these details is that our variables lie on different scales: some span a wider range of variation than others. A common practice in clustering is to scale all variables in our data such that they are mean-centered and have a standard deviation of 1, before performing the clustering. This ensures that all variables have the same "vote" in the clustering. If we don't scale, then variables that have large amounts of variation (large standard deviations) will disproportionately affect the Euclidean distance, and therefore our clustering will be highly influenced by those variables to the detriment of other variables. This is not inherently wrong, but it is important to keep in mind that unscaled data might result in different clusters than scaled data. To scale our data, we use the function `scale()`.

```
df <- scale(iris[,1:4])
```

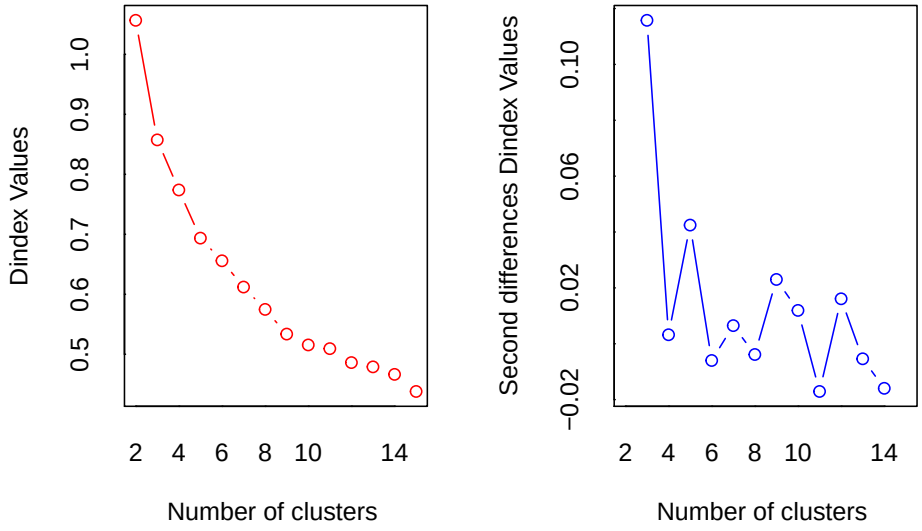
Exercise: what's the mean and standard deviation of each of the four variables in the iris dataset before scaling? what is their mean and standard deviation after scaling?

Ok, now on to the second preliminary detail. Clustering methods require the specification of the number of clusters that we a priori choose to fit the data to. Deciding on what is the "best" number of clusters depends on a number of criteria (e.g. minimizes the total within-cluster variance, homogenizing per-cluster variance, etc), and there are many methods with different criteria. We'll use the function `NbClust` which runs 26 of these different methods on our data for assessing the "best" number of clusters. We'll then choose to use the number of clusters that is recommended by the largest number of methods.

```
numclust <- NbClust(df, min.nc=2, max.nc=15, distance="euclidean", method="kmeans")
```




```
## *** : The Hubert index is a graphical method of determining the number of clusters.
##             In the plot of Hubert index, we seek a significant knee that corresponds to a
##             significant increase of the value of the measure i.e the significant peak in Hubert
##                   index second differences plot.
##
```




```
## *** : The D index is a graphical method of determining the number of clusters.
##             In the plot of D index, we seek a significant knee (the significant peak in Dindex
##             second differences plot) that corresponds to a significant increase of the value of
##                   the measure.
##
```

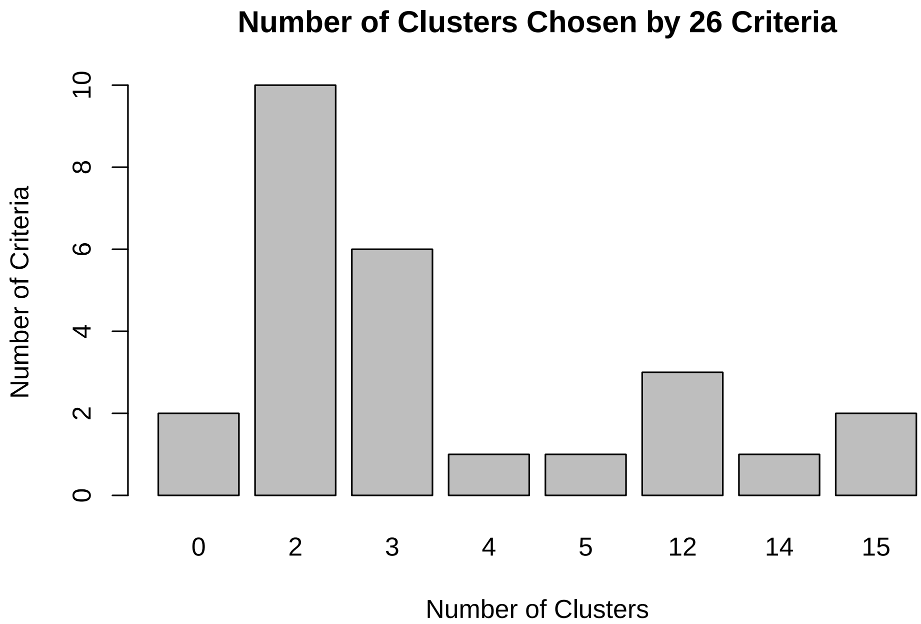
```
## *****************************************************************
## * Among all indices:
## * 10 proposed 2 as the best number of clusters
## * 6 proposed 3 as the best number of clusters
## * 1 proposed 4 as the best number of clusters
## * 1 proposed 5 as the best number of clusters
## * 3 proposed 12 as the best number of clusters
## * 1 proposed 14 as the best number of clusters
## * 2 proposed 15 as the best number of clusters
##
##                     ***** Conclusion *****
##
## * According to the majority rule, the best number of clusters is  2
##
##
## *****************************************************************
```

```
table(numclust$Best.n[1,])
```

```
##
##  0  2  3  4  5 12 14 15
##  2 10  6  1  1  3  1  2
```

It seems like ten of the methods suggest that the best number of clusters should be 2. There is also a considerable (but smaller) number of methods (six), that suggest it should be 3.

```
barplot(table(numclust$Best.n[1,]),
xlab="Number of Clusters", ylab="Number of Criteria", main="Number of Clusters Chosen
```



Let's go with 2 clusters then. We are now finally ready to perform a K-means clustering. We do so as follows:

```
fit.kmeans <- kmeans(df, 2)
```

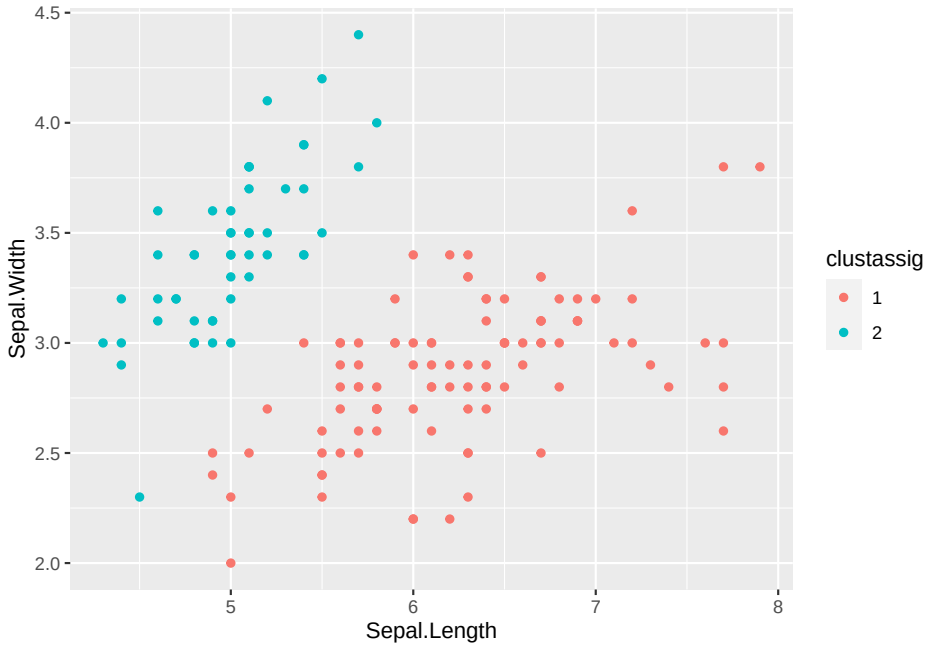
The cluster assignments for all observations are stored in the "cluster" attribute of the resulting object:

```
clustassig <- fit.kmeans$cluster
clustassig
```

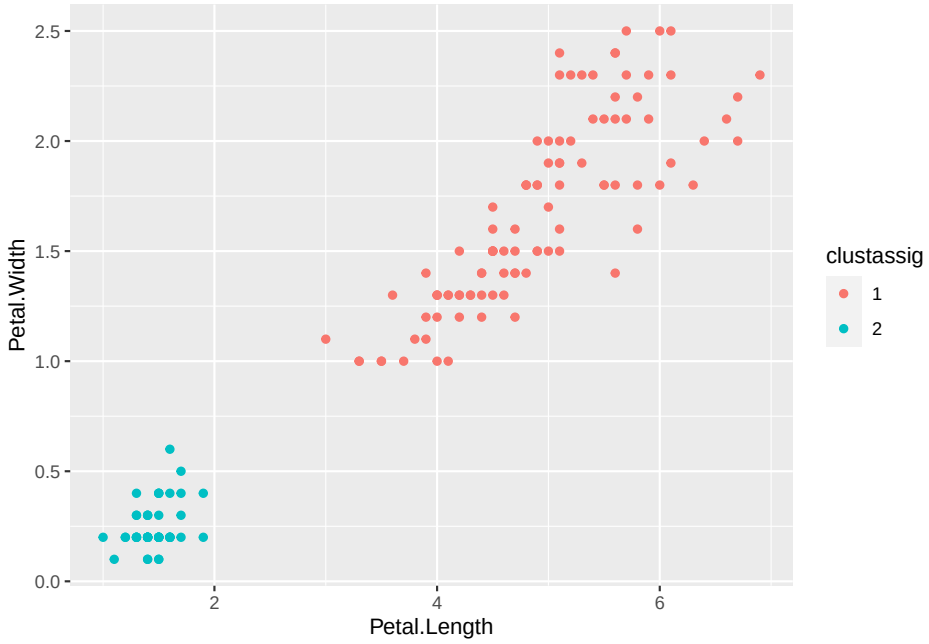
```
##   [1] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
##  [38] 2 2 2 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##  [75] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [112] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [149] 1 1
```

We can use this vector to plot our data, colored by the resulting clusters.

```
clustassig <- as.factor(clustassig) # Treat the cluster assignments as discrete factors
irisclust <- data.frame(iris,clustassig) # Combine data with cluster assignments
ggplot(data=iris) + geom_point(aes(x=Sepal.Length,y=Sepal.Width,color=clustassig)) # Plot sepal
```




```
ggplot(data=iris) + geom_point(aes(x=Petal.Length,y=Petal.Width,color=clustassig)) # P
```



Exercise: Now try computing a K-means clustering on the data yourself, this time using 3 clusters. Plot the result using a different color for each cluster.

Exercise: How well do the clusters from the previous exercise correspond to the 3 iris species? Are they perfectly matched? Why? Why not?

Chapter 16

REcoStats: Linear Models

We describe linear models in this chapter. First we need to load some libraries (and install them if necessary).

```
if (!require("tidyverse")) install.packages("tidyverse") # Library for data analysis
if (!require("stargazer")) install.packages("stargazer") # Library for producing pretty tables o
if (!require("devtools")) install.packages("devtools")
if (!require("report")) devtools::install_github("easystats/report") # Library for producing nice
```

16.1 Fitting a simple linear regression

We'll use a dataset published by Allison and Cicchetti (1976). In this study, the authors studied the relationship between sleep and various ecological and morphological variables across a set of mammalian species: https://science.sciencemag.org/content/194/4266/732

Let's start by loading the data into a table:

```
allisontab <- read.csv("Data_allison.csv")
```

This dataset contains several variables related to various body measurements and measures of sleep in different species. Note that some of these are continuous, while others are discrete and ordinal.

```
summary(allisontab)
```

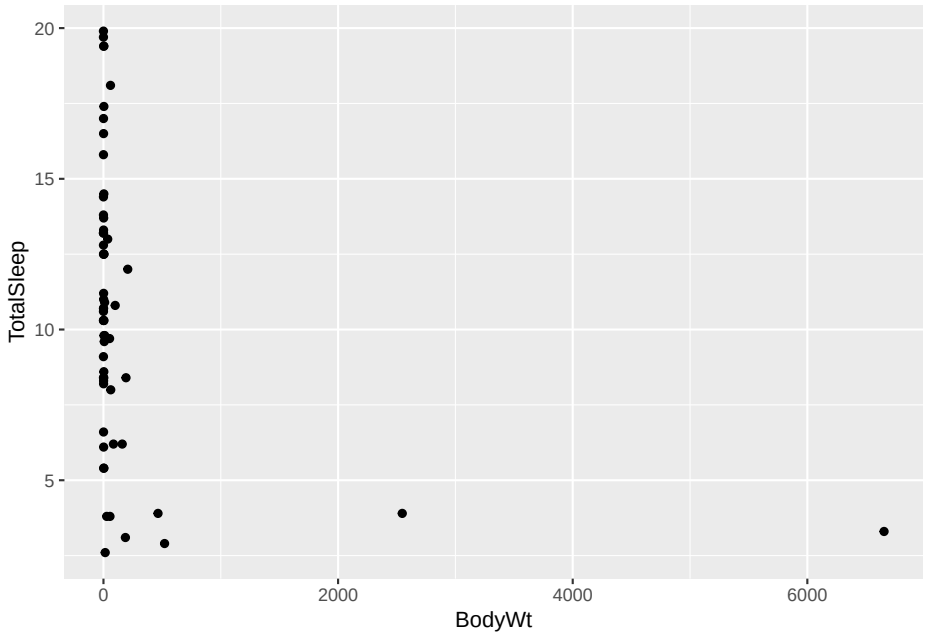
```
##    Species            BodyWt          BrainWt        NonDreaming
##  Length:62        Min.   :  0.005   Min.   :  0.14   Min.   : 2.100
##  Class :character  1st Qu.:  0.600   1st Qu.:  4.25   1st Qu.: 6.250
##  Mode  :character  Median :  3.342   Median :  17.25   Median : 8.350
##                    Mean   : 198.790   Mean   : 283.13   Mean   : 8.673
##                    3rd Qu.:  48.202   3rd Qu.: 166.00   3rd Qu.:11.000
```

```
##                Max.   :6654.000   Max.   :5712.00   Max.   :17.900
##                                                        NA's   :14
##     Dreaming       TotalSleep       LifeSpan         Gestation
##  Min.   :0.000   Min.   : 2.60   Min.   :  2.000   Min.   : 12.00
##  1st Qu.:0.900   1st Qu.: 8.05   1st Qu.:  6.625   1st Qu.: 35.75
##  Median :1.800   Median :10.45   Median : 15.100   Median : 79.00
##  Mean   :1.972   Mean   :10.53   Mean   : 19.878   Mean   :142.35
##  3rd Qu.:2.550   3rd Qu.:13.20   3rd Qu.: 27.750   3rd Qu.:207.50
##  Max.   :6.600   Max.   :19.90   Max.   :100.000   Max.   :645.00
##  NA's   :12      NA's   :4       NA's   :4         NA's   :4
##    Predation        Exposure         Danger
##  Min.   :1.000   Min.   :1.000   Min.   :1.000
##  1st Qu.:2.000   1st Qu.:1.000   1st Qu.:1.000
##  Median :3.000   Median :2.000   Median :2.000
##  Mean   :2.871   Mean   :2.419   Mean   :2.613
##  3rd Qu.:4.000   3rd Qu.:4.000   3rd Qu.:4.000
##  Max.   :5.000   Max.   :5.000   Max.   :5.000
##
```

We'll begin by focusing on the relationship between two of the continuous variables: body size (in kg) and total amount of sleep (in hours). Let's plot these to see what they look like:

```
ggplot(allisontab) + geom_point(aes(x=BodyWt,y=TotalSleep))
```

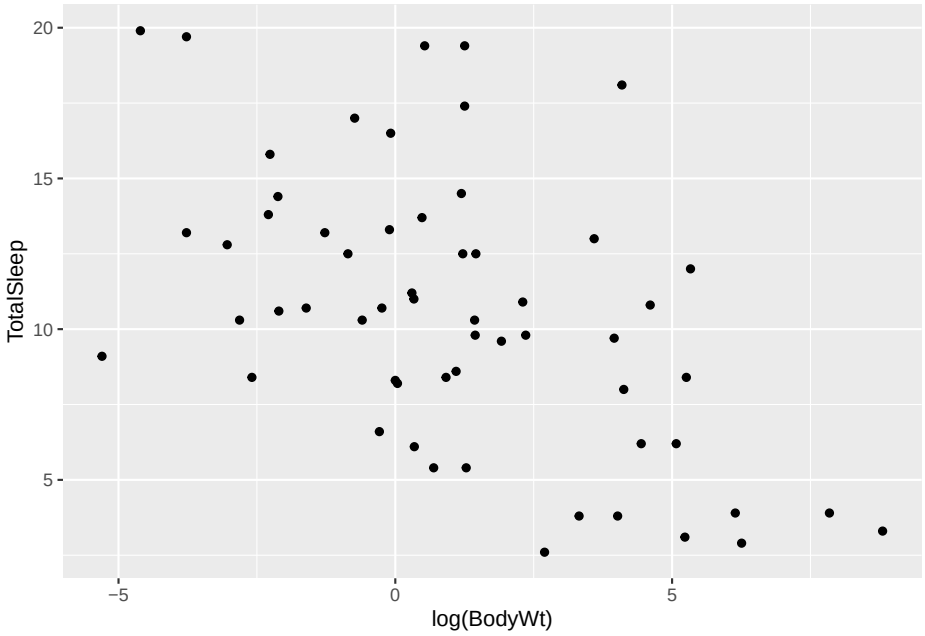
```
## Warning: Removed 4 rows containing missing values (geom_point).
```



Hmmm this looks weird. We have many measurements of body weight around 0 (small values) and a few very large values of thousands of kilograms. This is not surprising: given that this dataset spans several different species, the measurements spans several orders of magnitude (from elephants to molerats). To account for this, variables involving body measurements (like weight or length) are traditionally converted into a log-scale when fitted into a linear model. Let's see what happens when we log-scale the body weight variable:

```
ggplot(allisontab) + geom_point(aes(x=log(BodyWt),y=TotalSleep))
```

```
## Warning: Removed 4 rows containing missing values (geom_point).
```



A pattern appears to emerge now. There seems to be a negative correlation between the log of body weight and the amount of sleep a species has. Indeed, we can measure this correlation using the `cor()` function:

```
cor(log(allisontab$BodyWt), allisontab$TotalSleep, use="complete.obs")
```

```
## [1] -0.5328345
```

Let's build a simple linear model to explain total sleep, as a function of body weight. In R, the standard way to fit a linear model is using the function `lm()`. We do so by following the following formula:

```
fit <- lm(formula, data)
```

The formula within an `lm()` function for a simple linear regression is:

$$y \sim x_1$$

Where y is the response variable and x_1 is the predictor variable. This formula is a shorthand way that R uses for writing the linear regression formula:

$$Y = \beta_0 + \beta_1 x_1 + \epsilon$$

In other words, R implicitly knows that each predictor variable will have an associated β coefficient that we're trying to estimate. Note that here y, x_1, ϵ, etc. represent lists (vectors) of variables. We don't need to specify additional terms for the β_0 (intercept) and ϵ (error) terms. The `lm()` function automatically accounts for the fact that a regression should have an intercept, and that there will necessarily exist errors (residuals) between our fit and the the observed value of Y.

We can also write this exact same equation by focusing on a single (example) variable, say y_i:

$$y_i = \beta_0 + \beta_1 x_{1,i} + \epsilon_i$$

In general, when we talk about vectors of variables, we'll use boldface, unlike when referring to a single variable.

In our case, we'll attempt to fit total sleep as a function of the log of body weight, plus some noise:

```
myfirstmodel <- lm(TotalSleep ~ log(BodyWt), data=allisontab)
myfirstmodel
```

```
##
## Call:
## lm(formula = TotalSleep ~ log(BodyWt), data = allisontab)
##
## Coefficients:
## (Intercept)  log(BodyWt)
##     11.4377      -0.7931
```

This way, we are fitting the following model:

$$TotalSleep = \beta_0 + \beta_1 log(BodyWt) + \epsilon$$

Remember that the β_0 coefficient is implicitly assumed by the `lm()` function. We can be more explicit and incorporate it into our equation, by simply adding a value of 1 (a constant). This will result in exactly the same output as before:

```
myfirstmodel <- lm(TotalSleep ~ 1 + log(BodyWt), data=allisontab)
myfirstmodel
```

```
##
## Call:
## lm(formula = TotalSleep ~ 1 + log(BodyWt), data = allisontab)
```

```
##
## Coefficients:
## (Intercept)  log(BodyWt)
##     11.4377      -0.7931
```

Exercise: the function `attributes()` allows us to unpack all the components of the object outputted by the function `lm()` (and many other objects in R). Try inputting your model output into this function. We can observe that one of the attributes of the object is called `coefficients`. If we type `myfirstmodel$coefficients`, we obtain a vector with the value of our two fitted coefficients (β_0 and β_1). Using the values from this vector, try plotting the line of best fit on top of the data. Hint: use the `geom_abline()` function from the `ggplot2` library.

16.2 Interpreting a simple linear regression

We can obtain information about our model's fit using the function `summary()`:

```
summary(myfirstmodel)
```

```
##
## Call:
## lm(formula = TotalSleep ~ 1 + log(BodyWt), data = allisontab)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -6.6990 -2.6264 -0.2441  2.1700  9.9095
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  11.4377     0.5510  20.759  < 2e-16 ***
## log(BodyWt)  -0.7931     0.1683  -4.712 1.66e-05 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 3.933 on 56 degrees of freedom
##   (4 observations deleted due to missingness)
## Multiple R-squared:  0.2839, Adjusted R-squared:  0.2711
## F-statistic:  22.2 on 1 and 56 DF,  p-value: 1.664e-05
```

The `summary()` function provides a summary of the output of `lm()` after it's been given some data and a model to fit. Let's pause and analyze the output here. The first line just re-states the formula we have provided to fit our model. Below that, we get a summary (min, max, median, etc.) of all the residuals (error terms) between our linear fit and the observed values of $TotalSleep$.

Below that, we can see a table with point estimates, standard errors, and a few

other properties of our estimated coefficients: the intercept (β_0, first line) and the slope (β_1, second line). The standard error is a measure of how confident we are about our point estimate (we'll revisit this in later lectures). The "t value" corresponds to the statistic for a "t-test" which serves to determine whether the estimate can be considered as significantly different from zero. The last column is the P-value from this test. We can see that both estimates are quite significantly different from zero ($P < 0.001$), meaning we can reject the hypothesis that these estimates are equivalent to zero.

Finally, the last few lines are overall measures of the fit of the model. 'Multiple R-squared' is the fraction of the variance in $TotalSleep$ explained by the fitted model. Generally, we want this number to be high, but it is possible to have very complex models with very high R-squared but lots of parameters, and therefore we run the risk of "over-fitting" our data. 'Adjusted R-squared' is a modified version of R-squared that attempts to penalize very complex models. The 'residual standard error' is the sum of the squares of the residuals (errors) over all observed data points, scaled by the degrees of freedom of the linear model, which is equal to n − k − 1 where n = total observations and k = total model parameters. Finally, the F-statistic is a test for whether *any* of the explanatory variables included in the model have a relationship to the outcome. In this case, we only have a single explanatory variable ($log(BodyWt)$), and so the P-value of this test is simply equal to the P-value of the t-test for the slope of $log(BodyWt)$.

We can use the function `report()` from the library `easystats` (https://github.com/easystats/report) to get a more verbose report than the `summary()` function provides.

```
report(myfirstmodel)
```

```
## Formula contains log- or sqrt-terms. See help("standardize") for how such terms are stan
## Formula contains log- or sqrt-terms. See help("standardize") for how such terms are stan

## We fitted a linear model (estimated using OLS) to predict TotalSleep with BodyWt (formul
##
##   - The effect of BodyWt [log] is significantly negative (beta = -0.79, 95% CI [-1.13, -0
##
## Standardized parameters were obtained by fitting the model on a standardized version of
```

Note that this function "standardizes" the input variables before providing a summary of the output, which makes the estimates' value to be slightly different than those stored in the output of `lm()`. This makes interpretation of the coefficients easier, as they are now expressed in terms of standard deviations from the mean.

Another way to summarize our output is via a summary table in , which can be easily constructed using the function `stargazer()` from the library `stargazer` (https://cran.r-project.org/web/packages/stargazer/index.html).

```
stargazer(myfirstmodel, type="text")
```

```
##
## ================================================
##                         Dependent variable:
##                     ---------------------------
##                              TotalSleep
## ------------------------------------------------
## log(BodyWt)                   -0.793***
##                                (0.168)
##
## Constant                      11.438***
##                                (0.551)
##
## ------------------------------------------------
## Observations                     58
## R2                              0.284
## Adjusted R2                     0.271
## Residual Std. Error        3.933 (df = 56)
## F Statistic            22.203*** (df = 1; 56)
## ================================================
## Note:                *p<0.1; **p<0.05; ***p<0.01
```

This package also supports LaTeX and HTML/CSS format (see the `type` option in `?stargazer`), which makes it very handy when copying the output of a regression from R into a working document.

Exercise: try fitting a linear model for $TotalSleep$ as a function of brain weight ($BrainWt$). Keep in mind that this is a size measurement that might span multiple orders of magnitude, just like body weight. What are the estimated slope and intercept coefficients? Which coefficients are significantly different from zero? What is the proportion of explained variance? How does this compare to our previous model including $BodyWt$?

Exercise: Plot the linear regression line of the above exercise on top of your data.

16.3 Simulating data from a linear model

It is often useful to simulate data from a model to understand how its parameters relate to features of the data, and to see what happens when we change those parameters. We will now create a function that can simulate data from a simple 1-parameter linear model. We will then feed this function different values of the parameters, and see what the data simulated under a given model looks like.

Let's start by first creating the simulation function. We'll simulate data from a

linear model. The model simulation function needs to be told: 1) The number (n) of data points we will simulate 1) How the explanatory variables are distributed: we'll use a normal distribution to specify this. 2) What the intercept (β_0) and slope (β_1) for the linear relationship between the explanatory and response variables are 3) How departures (errors) from linearity for the response variables will be modeled: we'll use another normal distribution for that as well, and assume errors are heteroscedastic for now.

```
linearmodsim <- function(n=2, beta_0=0, beta_1=1, sigma.res=1, mu.explan=5, sigma.expla
  # Simulate explanatory variables
  explan <- r_explan(n,mu.explan,sigma.explan)
  # Sort the simulated explanatory values from smallest to largest
  explan <- sort(explan)
  # Standardize the response variables so that they are  mean-centered and scaled by t
  explan.scaled <- scale(explan)
  # OPTIONAL: If errors are not heteroscedastic (hetero does not equal 0), then their
  sdev.err <- sapply(sigma.res + explan.scaled*hetero,max,0)
  # Simulate the error values using the above-specified standard deviation
  error <- rerror(n,0,sdev.err)
  # Simulate response variables via the linear model
  response <- beta_0 + beta_1 * explan + error
  # Output a table containing the explanatory values and their corresponding response
  cbind(data.frame(explan,response))
}
```

Exercise:

a) Carefully read the code for the function above. Make sure you understand every step in the function.

b) Plot the output of a simulated linear model with 40 data points, an intercept of 1.5 and a slope of 3. Simulate from the same model one more time, and plot the output again.

c) Now, fit the data from your latest simulation using the `lm()` function. Does your fit match your simulations?

d) Try increasing the sample size (say, to 200 data points), and repeat the `lm()` fitting. How does this influence the accuracy of your fitted model? Try simulating and fitting multiple times to get an idea of how well you can estimate the parameters.

e) Try changing the standard deviation of the simulated errors (make it smaller or larger), and repeat the `lm()` fitting. How does this influence the accuracy of your fitted model?

16.4 Hypothesis testing and permutation testing

Let's evaluate again the hypothesis that there is no relationship between TotalSleep and log(BodyWt). Recall that one way to do it would be by using a linear model, and testing whether the value of the fitted slope is significantly different from zero, using a t-test:

```
summary(lm(TotalSleep ~ log(BodyWt), data=allisontab))
```

```
##
## Call:
## lm(formula = TotalSleep ~ log(BodyWt), data = allisontab)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -6.6990 -2.6264 -0.2441  2.1700  9.9095
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  11.4377     0.5510  20.759  < 2e-16 ***
## log(BodyWt)  -0.7931     0.1683  -4.712 1.66e-05 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 3.933 on 56 degrees of freedom
##   (4 observations deleted due to missingness)
## Multiple R-squared:  0.2839, Adjusted R-squared:  0.2711
## F-statistic:  22.2 on 1 and 56 DF,  p-value: 1.664e-05
```

Take a look at the P-values for the intercept and the slope. If you look at the help page `?summary.lm`, you can see that the P-values from these values come from a two-sided t-test. t-tests are usually deployed to compare the means of two populations, or to assess whether the mean of a population has a value specified by a hypothesis. In the case of the slope, for example, we're assessing whether our parameter estimate for the slope has a value specified by the null hypothesis, which in our case is zero. In other words, we're testing whether the value of the slope is consistent with there being no relationship between the two variables (such that if we had an infinite number of data points, their estimated slope would be zero)

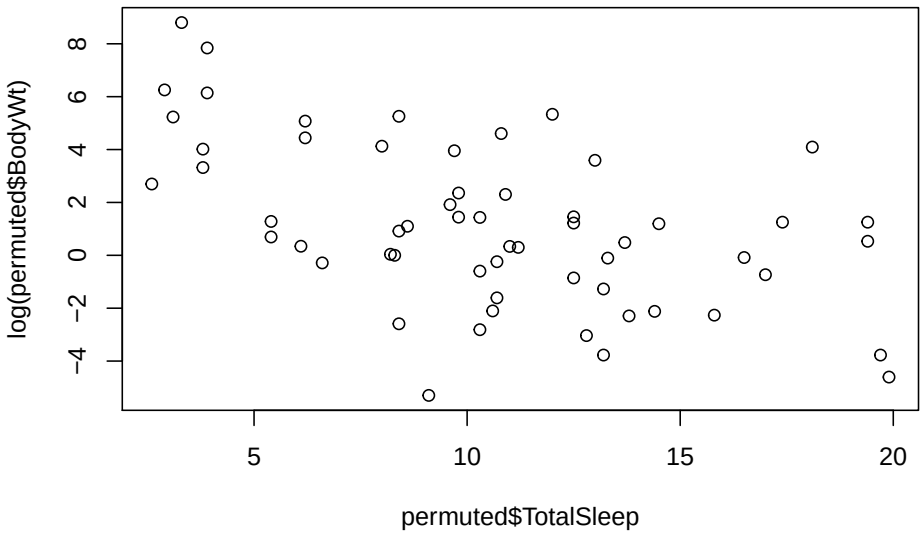
The above t-test makes assumptions on our data that sometimes may not be warranted. Most importantly, the t-test assumes we have a large number of samples, which might not always be the case. We can perform a more robust test that makes less a priori assumptions on our data - a permutation test. To do so, we need to be careful to permute the appropriate variables relevant to the relationship we are trying to test. In this case, we only have two variables

(TotalSleep and log(BodyWt)), and we are trying to test whether there is a significant relationship between them. If we randomly shuffle one variable with respect to the other, we should obtain a randomized sample of our data. We can use the following function, which takes in a tibble and a variable of interest, and returns a new tibble in which that particular variable's values are randomly shuffled.

```
permute <- function(tab,vartoshuffle){
  # Extract column we wish to shuffle as a vector
  toshuffle <- unlist(tab[,vartoshuffle],use.names=FALSE)
  # The function sample() serves to randomize the order of elements in a vector
  shuffled <- sample(toshuffle)
  # Replace vector in new table (use !! to refer to a dynamic variable name)
  newtab <- mutate(tab, !!vartoshuffle := shuffled )
  return(newtab)
}
```

Now we can obtain a permuted version of our original data, and compute the slope estimate on this dataset instead:

```
permuted <- permute(allisontab, "TotalSleep")
plot(permuted$TotalSleep,log(permuted$BodyWt))
```




```
permest <- lm(TotalSleep ~ log(BodyWt), data=permuted)$coeff[2]
permest
```

```
## log(BodyWt)
##  -0.7931139
```

Exercise: try estimating the same parameter from a series of 100 permuted versions of our original data, and collecting each of the permuted parameters

into a vector called "permvec".

We now have a distribution of the parameter estimate under the assumption that there is no relationship between these two variables:

Exercise: obtain an empirical one-tailed P-value from this distribution by counting how many of the permuted samples are as extreme or more extreme (in the negative or positive direction, than our original estimate, and dividing by the total number of permuted samples we have. Note: you should add a 1 to both the denominator and the numerator of this ratio, in case there are no permuted samples that are as large as the original estimate, so as not to get an infinite number.

The R package `coin` provides a handy way to apply permutation tests to a wide variety of problems.

```
if (!require("coin")) install.packages("coin")
```

```
## Loading required package: coin
```

```
## Loading required package: survival
```

```
##
## Attaching package: 'survival'
```

```
## The following object is masked from 'package:boot':
##
##     aml
```

```
##
## Attaching package: 'coin'
```

```
## The following object is masked from 'package:infer':
##
##     chisq_test
```

```
## The following object is masked from 'package:scales':
##
##     pvalue
```

```
library("coin") # Library with pre-written permutation tests
```

The `spearman_test()` function runs a permutation test of independence between two numeric variables, like the one in the `permute()` function we coded above. The advantage is that we don't need to actually code the function, we can just run the pre-made function in the `coin` package directly, as long as we know what type of dependency we're testing. In this case, we perform a test using 1000 permutations (the more permutations, the more exact the test):

```
spearman_test(TotalSleep ~ log(BodyWt), data=allisontab, distribution=approximate(nresample=1000)
```

```
##
##  Approximative Spearman Correlation Test
```

```
##
## data:  TotalSleep by log(BodyWt)
## Z = -3.8188, p-value < 0.001
## alternative hypothesis: true rho is not equal to 0
```

Exercise: Perform a permutation test to assess whether there is a significant relationship between log(BrainWt) and TotalSleep. Compare this to a t-test testing the same relationship.

Let's perform a different type of permutation test. In this case, we'll test whether the mean scores of two categories (for example, the math exam scores from two classrooms) are equal to each other.

```
mathscore <- c(80, 114, 90, 110, 116, 114, 128, 110, 124, 130)
classroom <- factor(c(rep("X",5), rep("Y",5)))
scoretab <- data.frame(classroom, mathscore)
```

The standard way to test this is using a t-test, which assumes we have many observations from the two classrooms (do we?) and that these observations come from distributions that have the same variance:

```
t.test(mathscore~classroom, data=scoretab, var.equal=TRUE)
```

```
##
##  Two Sample t-test
##
## data:  mathscore by classroom
## t = -2.345, df = 8, p-value = 0.04705
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -38.081091  -0.318909
## sample estimates:
## mean in group X mean in group Y
##           102.0           121.2
```

Exercise: look at the help menu for the `oneway_test` in the `coin` package and find a way to carry out the same type of statistical test as above, but using a permutation procedure. Apply it to the `scoretab` data defined above. Do you see any difference between the P-values from the t-test and the permutation-based test. Why?