Data Analysis and Statistical Thinking: An R Workbook

Fernando Racimo, Shyam Gopalakrishnan

2021-04-27

Contents

Chapter 1

Introduction

Chapter 2

Getting Started with Data Analysis

Chapter 3

Probability Part 1

3.1 Today's programme

We will try out different R functions for simulating probability distributions and computing summary statistics from the resulting data.

We will cover the following topics

- Bernoulli distribution
- Binomial distribution
- Simulating discrete random variables
- Expected value
- Variance
- Probability mass function

3.2 The Bernoulli distribution: tossing a coin

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

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

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

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:

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

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

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

```
## [1] 0.7
```

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?

3.3 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.

One way of doing this is by running 20 Bernoulli trials, and then adding them up using the function `sum()`:

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

```
## [1] 12
```

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, so the first parameter is now 1 (note: if we had set it to 20, we would have performed 20 20-toss experiments). 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.

3.4 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?):

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

```
## [1]  7 11 12 12 13
```

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

```
## [1] 55
```

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

```
Xave <- Xsum / nsims
Xave
```

```
## [1] 11
```

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

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

```
## [1] 11
```

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?

The number we "expect" our average to approach as we increase the number of trials 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. In the case of a variable Y that follows the Bernoulli distribution (which is just a Binomial with size 1), then:

$$E[Y] = p$$

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 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.

3.5 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

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

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.

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

```
##   [1] 12 12 13  8  6 10 10 13 10  9 11 11 10  9 11 10  6  8  8  9 10 11  8 14 13
##  [26] 10  6 10 10 12  7 10  9  7  8 15  9  8 10  9 13 10  9 11  8  6 10  8 14  6
##  [51]  9 12 12  9 11  8 10  7  9 10 15  9  9  7 10  5 10  9  9  8  8 14 10  6 12
##  [76] 10  9  9 15 10 12 11  8 10 10 10  7 10  9  9 13  8 15 10 11 17 12  8 13 10
## [101]  7  7  7 12 11 12 10  9  8 13 13 12 13 13  8 13 10  8  8 11  7 15  9 12 11
## [126] 12 12  7 10 10 12 10 11  7 12 12 13  9  8  9  6 13 12  7 12 13 10 13  9  9
## [151] 13  6  8 13 11  7 10 15  8 15  5 14  8 12 10 12 11 13 12  9 10 10 10 10  8
## [176] 12 10 13 14 11  8 10  8 13 13  9 12 11 14  9  6 11 11  9  9 16 11  9 11 12
## [201] 12  7 11  8  8 15  9  7  9  8  7  8 12 10 10  8  8 14  5 13 15 13 11 14 10
## [226] 13 10 11  6 11 10 10 10 12  9 13 11 14 10 10  6 10 14 12 12  9 12  9 14 13
## [251] 11 12 11  8  9  9  5 13 11  7  9 10  8 11 11 10 14  9  8  5 10 11 13  9  9
## [276] 10 11 11  9 11 12  7 12 15  9  9  7 13 11 11 13 11  8  8  9  9 12  6  8  8
## [301]  8 11  9  8  9 12  9 11  7  9  9 12 11  7 10 10 12  6 11 10  7 12 13  8  9
## [326]  7  8 12 11  8 15 16  9  8 10  8 10 12  6  4  9 11  7 12 13 14 11  7  8 12
## [351]  9 13  8 10 12 13  8 12 10  8 10 12  5  9  9  7  9 10 11  9 11  6 14 13 11
## [376] 12 13 10  9 11  9 12 14 10 11  8  8 11  8  8 12  8 11 11  8  8 10  9 10  7
## [401] 14  7 10  8 10  8 12 10  9  9 10  8  9 10 10 11 14  7 10 10 12 10  6  9  8
## [426] 11 11 11 11 15 12 12  9  7 11  9 14 10  9 11 15 13  9  6 12  8  8  9 14 12
## [451] 14 14 13 13  8  9 11 12 10 11 13 10 12 10 11 12  6  9  6 12 10 15  9  7 10
## [476] 14  8  5 10 11 10  9 12  8  8  8 10 11  9 12 11  9  9 10  7  8  9 12 13 10
```

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

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

```
## [1] 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] 87
```

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 / nsims
proportion_of_tens
```

```
## [1] 0.174
```

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?

3.6 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():

```
nsims <- 100000
size <- 10
prob <- 0.5
X <- rbinom(nsims, size, prob)
head(X)
```

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

```
mean(X)
```

```
## [1] 4.99437
```

```
var(X)
```

```
## [1] 2.492583
```

Exercise: Compute the variance of a set of 100,000 Binomial trials, each of size 10, for different values of the probability of heads, ranging from 0 to 1 (in steps of 0.1). This is equivalent to performing one hundred thousand 5-toss experiments, with different types of biased coins in each experiment. For what value of the binomial probability is the variance maximized? Afterwards, try overlaying the theoretical variance of the binomial distribuiton (`size*p*(1-p)`) on top of your plot.

Chapter 4

Probability Part 2

Chapter 5

Probability Catch-up

5.1 Discrete distributions

We'll start by reviewing a few commonly used discrete probability distributions.

There is a whole world of R functions we can use to sample random values from probability distributions or obtain probabilities from them. Generally, each of these functions is referenced with an abbreviation (`pois` for Poisson, `geom` for Geometric, `binom` for the Binomial, etc.) preceded by the type of function we want, referenced by a letter:

- `r`… for simulating from a distribution of interest
- `d`… for obtaining the probability mass function $P[X = x]$ (or probability density function for continuous random variables)
- `p`… for obtaining the cumulative distribution function, which is just a fancy way of saying $P[X \leq q]$ for any value of q.
- `q`… for the quantile function of a distribution of interest

For example, in the case of the Poisson distribution:

- `rpois(n, lambda)` generates n random samples from the Poisson distribution with rate `lambda`
- `dpois(x, lambda)` allow us to calculate the Poisson probability mass function (PMF) $P[X = x] = \lambda^x e^{-\lambda}/x!$ for a given value of x (an integer) and `lambda` (λ) - the rate of the Poisson process.
- `ppois(q,lambda)` allow us to calculate the Poisson cumulative distribution function $P[X \leq q]$ for any value of q (an integer) and rate `lambda` (λ).
- `qpois(p, lambda)` provides the pth quantile of the Poisson PMF with rate `lambda`.

The internet and the R help functions (e.g. `?rpois`) are your friends here. Use them whenever you're interested in understanding how to use these functions, and what types of inputs they require.

5.1.1 Poisson distribution

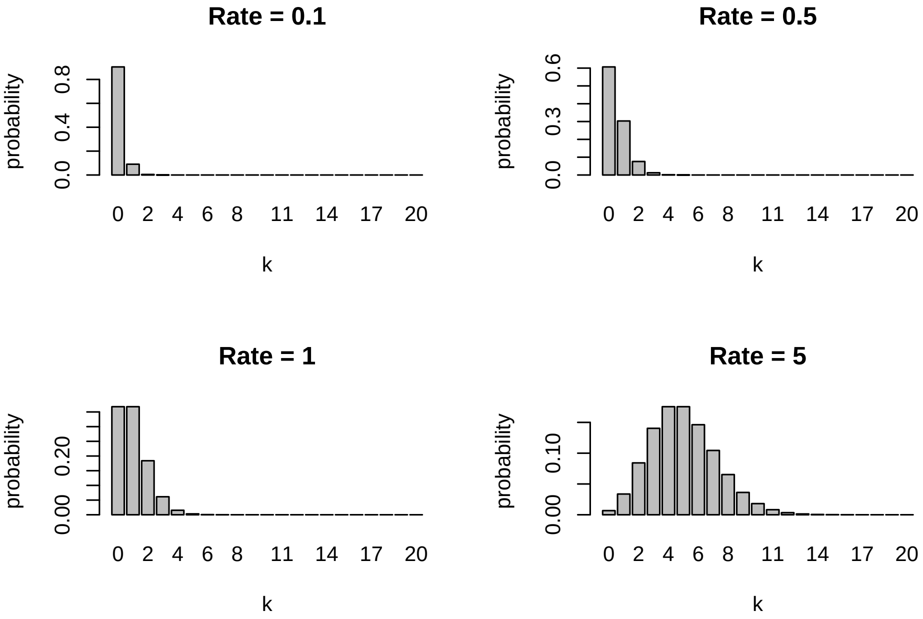
Let's take a stab at answering some discrete probability questions using the Poisson distribution and the above mentioned functions.

Recall that the Poisson distribution is used to model the number of events (or objects) that occur over a given period of time (or space), if these occur with a constant mean rate and independently of each other. It is generally applied to systems with a large number of possible events, each of which is rare.

Some (simplified) examples include:

- Number of rain droplets falling on a house roof
- Number of meteorites hitting a planet over a given period of time
- Number of buses arriving at a station over a given period of time
- Number of trees in a given area of land
- Number of fish in a given volume of water

Here are plots of Poisson PMFs with different mean rates λ:



Exercise: Let's assume the number of trees on any squared kilometer of land follows a Poisson distribution. If trees occur at a mean rate of 5 per squared kilometer, what is the probability that **3 or less trees** will occur on a given squared kilometer?

A nice property of the Poisson distribution is that if its rate λ is given over a particular unit of time (or space), but our period or area of study is larger or smaller than that unit, we can simply multiply that rate by the extent of the corresponding period or area to get the rate of the Poisson distribution for that period or area:

$$P[x\ events\ in\ interval\ of\ length\ 1] = \lambda^x e^{-\lambda}/x!$$

$$P[x\ events\ in\ interval\ of\ length\ t] = (\lambda t)^x e^{-\lambda t}/x!$$

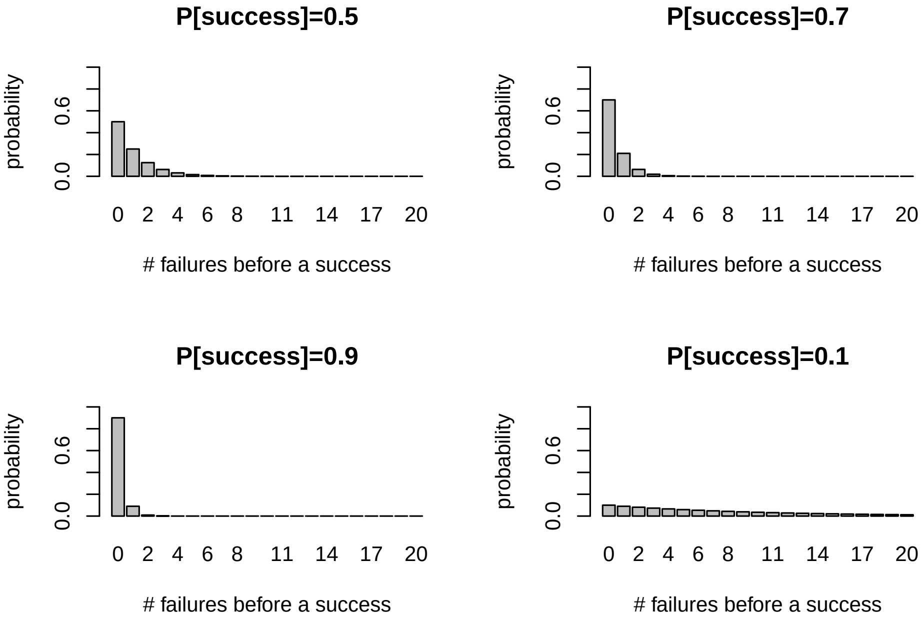
Exercise: Given the same per-unit rate as the previous exercise, what is the probability that **more than 10 trees** will occur on area that is **3 squared kilometers** large?

5.1.2 Geometric distribution

Another interesting distribution is the Geometric distribution. Recall that this distribution serves to model a sequence of trials, in which we are interested in computing the probability of a given number of failures, before we are succesful. The probability mass function depends on a single parameter, the per-trial probability of success, `p`. The probability I fail `k` times before I succeed is:

$$P[X = k] = (1 - p)^k p$$

Here are plots of Geometric PMFs with different probabilities of success:



While the probability mass function always decreases for increasing number of failures, the rate of this decrease strongly depends on the probability of success in eahc trial. For example, if I have a loaded coin that has 90% of probability of giving heads, then I won't have to wait for long until I get heads (success). Looking at the bottom left panel in the figures above, I can see that it's actually most likely I will get heads on the first try.

Exercise: I have a loaded coin that I use to trick people. Unlike normal coins, this coin results in heads 3 out of 4 times, on average. I start tossing the coin, and wonder: what is the probability I get at least 2 tails before I get a head? And what is the probability I get exactly 2 tails before I get a head? Use the geometric cumulative distribution function `pgeom` and the probability mass function `dgeom`, respectively, to answer these questions.

5.1.3 Sampling from arbitrary distributions

What if we want to create our own probability distribution? In R, we can, using the funciton `sample()`. Here's an example of a probability distribution I just decided to create, which serves to model a six-sided dice roll, where each side has equal probability (1/6) of being obtained:

```
sides=c(1,2,3,4,5,6) # vector of elements
probabilities=c(1/6,1/6,1/6,1/6,1/6,1/6) # vector of probabilities for each element
sample(sides,10,prob=probabilities,replace=TRUE) # roll a dice 10 times
```

```
##  [1] 5 5 4 2 5 2 5 6 2 1
```

Table 5.1: Dog litter size frequencies

litter_size	frequency
1	0.07
2	0.15
3	0.20
4	0.20
5	0.16
6	0.10
7	0.06
8	0.03
9	0.01
10	0.02

The option `replace=TRUE` ensures we can sample **with replacement** from the `sides` vector. Because each element has equal probability of occurring, the above is also calle a **discrete uniform distribution**. This is not necessary, however. Here's an example in which we roll a loaded dice where the side `6` is biased to appear more frequently than the others:

```
sides=c(1,2,3,4,5,6) # vector of elements
weights=c(1,1,1,1,1,4) # vector of unnormalized weights for each element, with the element '6' be
probabilities = weights / sum(weights) # to create a probability vector, we ensure all the weight
sample(sides,10,prob=probabilities,replace=TRUE) # roll a loaded dice 10 times
```

```
##  [1] 6 2 6 6 6 6 6 6 2 6
```

Let's put this function into practice.

Exercise: You're a dog breeder who's been given the following table of litter size frequencies, obtained from 10,000 previously recorded litter sizes:

```
library(knitr)
dogprobs <- round(c(dpois(seq(1,9),4)),2)
dogprobs <- c(dogprobs, 1-sum(dogprobs))
littertab <- cbind(c(1,2,3,4,5,6,7,8,9,10),dogprobs)
colnames(littertab) <- c("litter_size","frequency")
kable(littertab,caption="Dog litter size frequencies")
```

In this exercise, we'll assume that the probability of a litter size of 11 or higher is effectively zero. Simulate 100 new litter sizes using the above table as an approximation to the true distribution of litter sizes, by using the recorded frequencies as probabilities.

a) Compute the sample mean, sample variance and create a histogram (with the function `hist()`) of the simulated litter sizes.

b) Estimate the probability mass function of your simulated samples from part (a), using the help of the function `table()`. This should be very similar to the original probabilities that you simulated from.

5.2 The Normal distribution and the Central Limit Theorem

The Central Limit Theorem states that if $X_1, X_2, ..., X_m$ are random samples from a distribution (any distribution) with mean μ and finite variance σ^2, and the sample mean is $\bar{X}$, then, as m grows large, $\bar{X}$ converges in distribution to a Normal distribution with mean μ and standard deviation $\sqrt{\sigma^2/m}$.

In other words, for large values of m (large sample sizes), we can treat the average of our sample as being drawn from $N(\mu, \sigma^2/m)$.

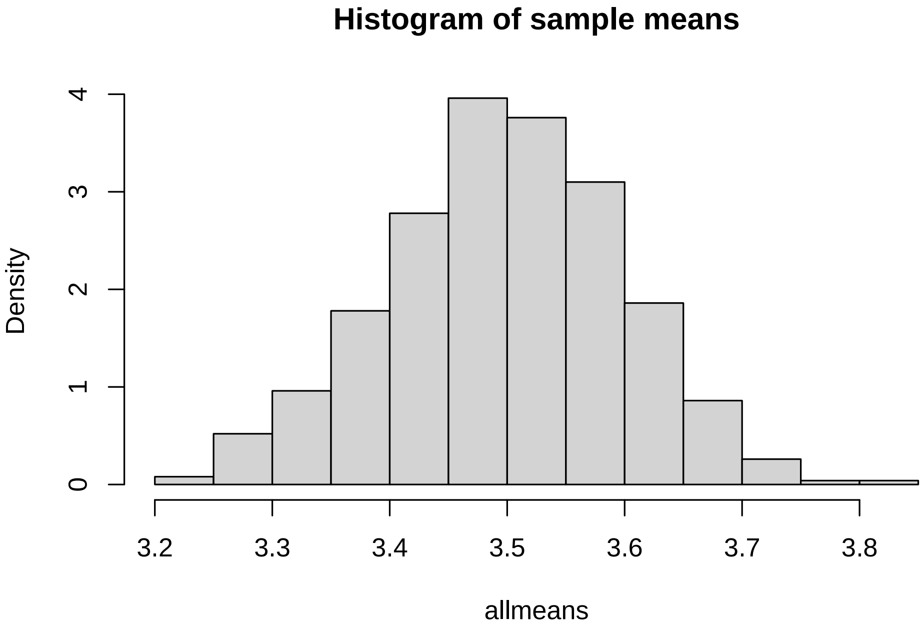
How can we verify this to be true? Well, one way to do this is to pretend (simulate) we have many sample sets, each composed of a large number of samples, compute their sample mean, and look at the histogram over all the sample means. If the CLT holds, that histogram should look very similar to a Normal distribution. Let's try that here!

We'll begin with a binomial distribution as our initial "sampling" distribution. We can, for example, draw $m = 100$ values from a binomial distribution with parameters $[p = 0.7, n = 5]$, and then compute their mean:

```r
m <- 100 # sample size
p <- 0.7 # binomial success parameter
n <- 5 # binomial size parameter
samp1 <- rbinom(m,n,p)  # simulation
mean1 <- mean(samp1) # sample average
```

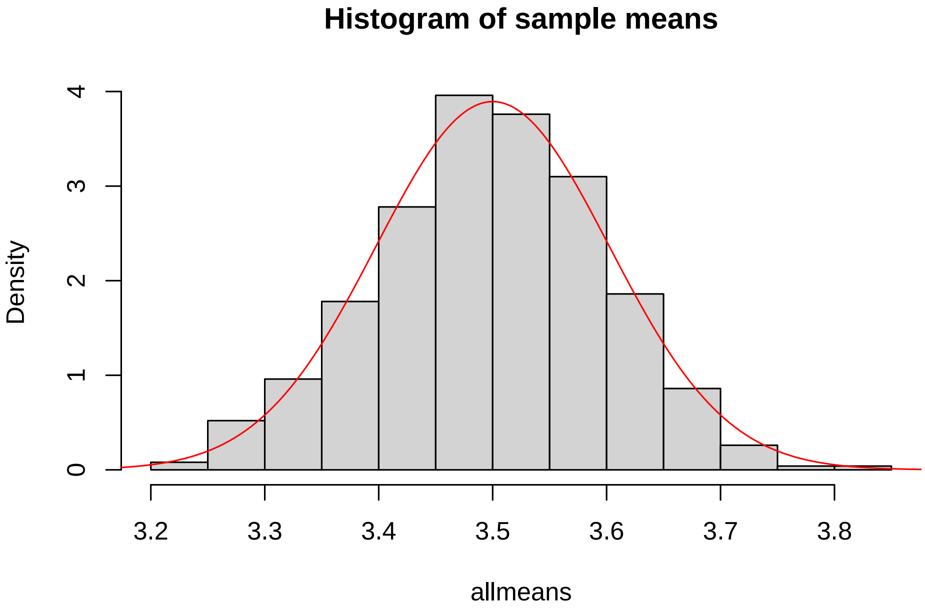
The CLT theorem is a statement about multiple means from multiple samples, so let's repeat the above exercise 1,000 times:

```r
allmeans <- sapply(seq(1,1000), function(i){mean(rbinom(m,n,p))})
hist(allmeans,freq=FALSE,main="Histogram of sample means")
```



The CLT states that this distribution should be very close to a Normal distribution with mean $\mu = np = 5 * 0.7 = 3.5$ and $\sigma^2 = (np(1-p))/m = 5 * 0.7 * 0.3/100 = 0.0105$. Let's verify that:

```
hist(allmeans,freq=FALSE,main="Histogram of sample means") # Histogram of sample means
mu <- n*p # mean of Normal distribution under the CLT
var <- n*p*(1-p)/m # variance of Normal distribution under the CLT
sd <- sqrt(var) # standard deviation of Normal distribution under the CLT
curve(dnorm(x,mean=mu,sd=sd),from=-5,to=5,n=5000,add=TRUE,col="red") # Normal distribution
```



Exercise: Repeat the exercise above but instead of a binomial distribution, use a Poisson distribution with parameter $\lambda = 5$ (considering what the mean and variance of the corresponding Normal distribution should be).

Exercise: Repeat the exercise above but instead of a binomial distribution, use a Geometric distribution with parameter $p = 0.8$ (considering what the mean and variance of the corresponding Normal distribution should be).

Note: if you know *a priori* that $X_1, X_2, ..., X_m$ are independent draws from a $Normal(\mu, \sigma^2)$ distribution, then you don't need to invoke the CLT: $\bar{X}$ will be normally distributed with mean μ and standard deviation $\sqrt{\sigma^2/m}$, even for low values of m!

5.3 The exponential distribution

Previously, we simulated the *quantity* of buses that would randomly arrive at a station over a particular period of time (an hour), given that the *rate* of this process is constant. For this, we used the Poisson distribution with parameter λ equal to this rate. Using the same rate assumption, we can also model the *waiting time* until the next bus arrives, using the exponential distribution. Take a look at the help page for `?rexp`.

Exercise: Buses arrive at a station at an average rate of 3 per hour. Using the `rexp` function, simulate 1,000 waiting times (in hours) of this random process, assuming the waiting time follows an exponential distribution. Create a histogram of these waiting times. Then, calculate their sample mean and compare it to the expected value of an exponential distribution with rate 3.

Exercise: I've arrived at a station. Assuming the same rate as above, use the `pexp` function to obtain the probability that I will have to wait less than 15 minutes till the next bus arrives.

Exercise: Assuming the same rate as above, use the `pexp` function to obtain the probability that I will have to wait more than 30 minutes till the next bus arrives.

Exercise: Assuming the same rate as above, use the `pexp` function to obtain the probability that I will have to wait between 33 and 48 minutes for the next bus to arrive.

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

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.

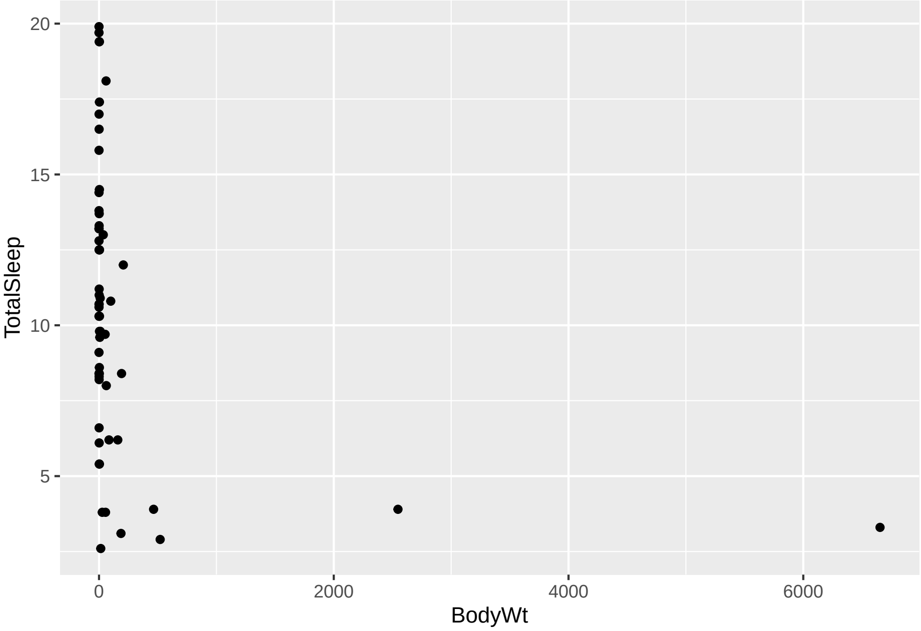
```
summary(allisontab)
```

```
##    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))
```

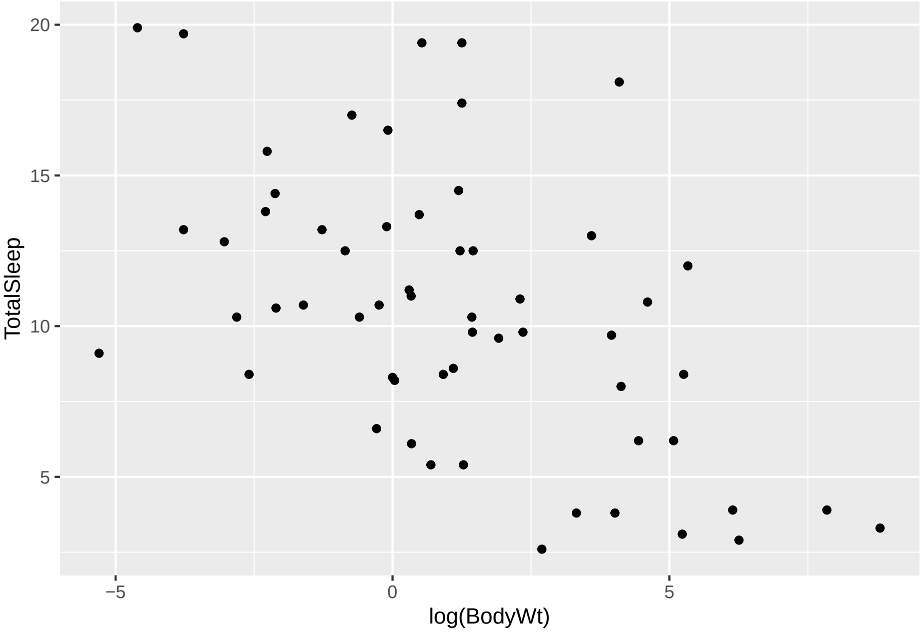
```
## Warning: Removed 4 rows containing missing values (geom_point).
```



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 `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.

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 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.

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 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 control the amount of error using a variable called `sigma.res`. We'll assume errors are homoscedastic (have the same variance) in this exercise.

```
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 heteroscedastic (hetero does not equal 0), then their stan
  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 residual errors (make `sigma.res` smaller or larger), and repeat the `lm()` fitting. How does this influence the accuracy of your fitted model?

Chapter 7

Properties of Estimators and Hypothesis Testing

7.1 Properties of point estimators

In this exercise, we'll draw many simulated samples from a known distribution with known parameters. We will then consider these as instances of real datasets, and estimate parameters of the original distributions using different estimators applied to the datasets. Our goal will be to analyze properties of the estimators, namely their sampling distribution, bias and variance.

Recall the definitions of two important properties of point estimators:

Bias:

$$B(\hat{\theta}_n) = E[\hat{\theta}_n(D) - \theta]$$

The bias reflects the average difference between our estimator and the true parameter.

Variance:

$$Var(\hat{\theta}_n) = E[(\hat{\theta}_n(D) - E[\hat{\theta}_n(D)])^2] = SE(\hat{\theta}_n)^2$$

The variance reflects the variation of the estimator's sampling distribution around its own mean (regardless of what the true parameter is). It is also the square of the standard error.

> In real life, we will generally not know the sampling distribution of our test statistic, but we can attempt to approximate it. As we'll see below, simulations can be of great help here.

We will first create a function that draws a user-specified number (`nsim`) of independent data samples of size `sampsize` from a distribution of choice. By default this distribution is the Normal distribution (`rnorm`), but it can be changed by the user:

```
simsamps <- function(sampsize, nsim = 10000, rsim = rnorm, ...){
  sims <- rsim(sampsize*nsim,...) # generate draws from a specified distribution
  simmat <- matrix(sample(sims), nrow=nsim, ncol=sampsize) # organize those draws into
  return(simmat) # provide the matrix as output
}
```

The output of this function is a matrix, with row number equal to the number of generated simulations, and column number equal to the size of each simulation. For example, to create 10,000 simulations, each of size 30, from a normal distribution, we would write:

```
testmat <- simsamps(30,10000,rnorm) # generate matrix of 10,000 Normal data sets, each
dim(testmat) # dimensions of matrix
```

```
## [1] 10000    30
```

What type of Normal distribution are we sampling from though? By default, the function rnorm samples from a standard Normal distribution with mean equal to 0 and standard deviation equal to 1. In our function `simsamps`, we are only feeding one argument to the normal distribution (the number of draws we want to obtain from it). The three dots placed in both the argument of the `simsamps` function and in its internal call to `rsim` allows us to feed more parameters to `rsim`:

```
testmat <- simsamps(30,100,rnorm,mean=3,sd=5) # generate matrix of 1000 Normal(3,5) da
```

We can obtain the sample mean and sample median from each of these datasets, using the `apply` function:

```
testmean <- apply(testmat,1,mean)
testmedian <- apply(testmat,1,median)
```

Exercise: Create 20,000 simulated data sets of size 100, each drawn from a Normal distribution with expected value equal to 4 and standard deviation equal to 10. Create a histogram of the sample means from all the data sets. Draw a blue line where the average of all these means is located, and a red line where the **expected value** of the original source distribution is located. Next, do the same for all the sample medians.

Now, try answering these questions: Is the sample mean a biased estimator of the expected value of a normal distribution? Is the sample median a biased estimator of the expected value of a normal distribution? Which estimator has the highest variance? Which estimator would you use if given a dataset like one of the ones we simulated, in order to estimate the expected value of this Normal distribution?

Exercise: Create 20,000 simulated data sets of size 100, each drawn from an Exponential distribution with rate equal to 2. Create a histogram of the sample means from all the data sets. Draw a blue line where the average of all these means is located, and a red line where the **expected value** of the original source distribution is located. Recall that the expected value of this distribution is 1/rate = 0.5.

Now, try answering these questions: Is the sample mean a biased estimator of the expected value of an exponential distribution? Is the sample median a biased estimator of the expected value of an exponential distribution? Which of these two estimators would you use if you were trying to estimate the rate of an exponential process (e.g. the rate at which buses arrive at a station), from a dataset like one of the ones we just simulated?

The histograms we've built reflect the **sampling distributions of estimators**. They serve to visualize the spread of an estimator's distribution around its own mean, and allow us to determine whether that estimator's mean is equal to the expected value of our distribution of interest.

> The 'sampling distribution' is the distribution of a particular test statistic, like the sample mean or the sample median. This is different from the distribution of the data itself. For example, a data set may come from a particular distribution, say Poisson, Normal or Exponential. The sample mean of such a dataset is a single number. If we had different data sets of equal size, we would be able to obtain different sample means. The distribution of those means (in the limit of infinite datasets) is the sampling distribution, which will often be Normal, if the conditions of the Central Limit Theorem apply. For other statistics, for example, the sample median, the Central Limit Theorem might not apply.

7.2 Obtaining confidence intervals

Confidence intervals (CIs) denote how sure we are about the value of a parameter. Importantly, CIs are statements made about an infinite number of datasets. For example, let's imagine a parameter of interest called θ, which has a value that we don't know. If we had an infinite number of data sets ($i = 1, 2, 3, ..., \infty$), the lower and upper 95% confidence intervals for each dataset i are the two values that would bound (contain) the unknown parameter θ in 95% of those datasets.

Generally, we only have **one set of data points**, so this statement might sound a bit confusing. How can we make statements about infinite data sets, when we only have one set? In practice, the confidence interval is an approximation based on the **spread (variance) of an estimator's ($\hat{\theta}$) sampling distribution around the expected value of the unknown parameter (θ)**.

If a given estimator $\hat{\theta}$ of a data set of size n:

1. has a known standard error:

$$SE(\hat{\theta}_n) = \sqrt{Var(\hat{\theta}_n)}$$

2. is unbiased:

$$E[\hat{\theta}_n - \theta] = 0$$

and

3. has a normal sampling distribution:

$$\hat{\theta}_n \sim Normal$$

then the confidence intervals that will contain the true value of the parameter 95% of the time are:

$$(\hat{\theta} + SE(\hat{\theta}_n)q_{2.5\%},\ \hat{\theta} + SE(\hat{\theta}_n)q_{97.5\%})$$

Here, $q_{x\%}$ is the $x\%$ quantile function of a standard Normal(0,1) distribution. This value marks a limit such that $x\%$ of the probability mass of a distribution is to the left (lower than the value), and $1 - x\%$ is to the right (higher than the value).

Because the standard normal(0,1) distribution is symmetric and centered around 0, $q_{2.5\%} = -q_{97.5\%}$, so we can also write the CI as:

$$(\hat{\theta} - SE(\hat{\theta}_n)q_{97.5\%},\ \hat{\theta} + SE(\hat{\theta}_n)q_{97.5\%})$$

If we compute these boundaries for a particular data set we study, and the estimator we're applying satisfies the 3 conditions above, we can be sure that these boundaries will contain the true parameter 95% of the time (out of an infinite number of possible data sets that we could have obtained, in theory, from the phenomenon of interest).

> Note that the sampling distribution of the mean will tend to be normal and unbiased for large sample sizes, because of the Central Limit Theorem! However, the sampling distribution of other statistics (like the median) may be neither normal nor unbiased. For example, as we saw in the previous section, the sampling distribution of the median of a data set that is exponentially distributed is biased! Thus, it would be inappropriate to compute the 95% CIs using the equation above in that case. Later on in this class, we'll figure out ways to obtain more general CIs that do not depend on the strict assumptions of unbiasedness and normality of the estimator's sampling distribution.

The mean of a normally distributed dataset is both unbiased and normally distributed, so the stated assumptions hold in that case. Let's verify that:

Exercise: Create 20,000 simulated data sets of size 5, each drawn from a Normal distribution with expected value equal to 4 and standard deviation equal to 10. For each dataset, compute the sample mean. Then, compute the standard error of the mean (standard deviation over all means). Finally, use the standard error to compute the 95% confidence intervals for each data set, using the formula stated above. How often do these confidence intervals contain the expected value? Hint: to obtain the quantiles of a Normal distribution, you can use the function `qnorm`. For example, the 35% quantile of a standard Normal(0,1) distribution is equal to `qnorm(0.35,mean=0,sd=1)`.

Here, because we are using simulations, we can obtain the standard error of the mean by generating many data sets. Recall that the standard error of the mean is equal to a scaled version of the standard deviation:

$$SE(\hat{\theta}_n) = \frac{\sqrt{\sigma^2}}{\sqrt{n}}$$

In practice, when working with a single data set, the estimator's standard error $SE(\hat{\theta}_n)$ is generally unknown. However, it can be approximated using the sample standard deviation s of the dataset we are studying:

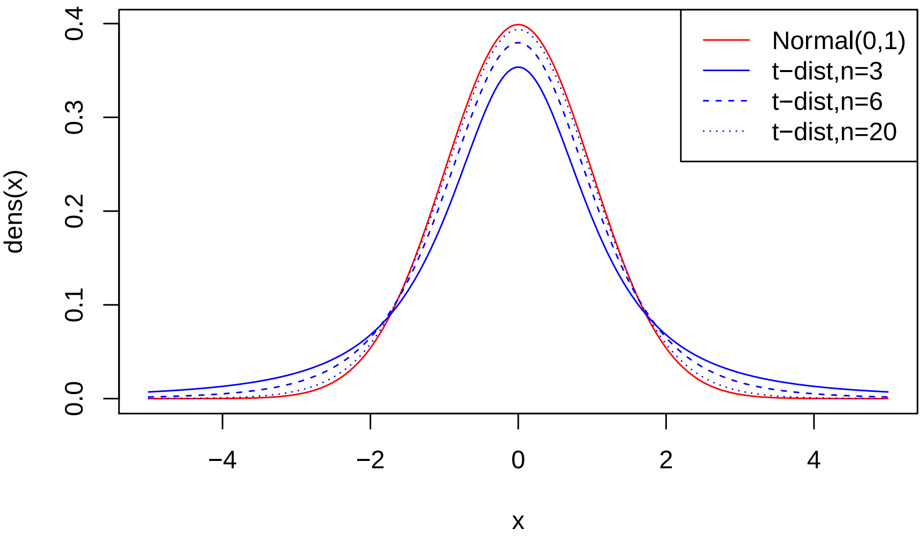
$$SE(\hat{\theta}_n) \approx \frac{s}{\sqrt{n}}$$

Thus, we can obtain CIs by replacing $SE(\hat{\theta}_n)$ above with $s/\sqrt{n}$:

$$(\hat{\theta} + \frac{s}{\sqrt{n}} q_{2.5\%}, \hat{\theta} + \frac{s}{\sqrt{n}} q_{97.5\%})$$

Exercise: Repeat the exercise above, but instead of using the standard error, approximate this value by using the standard deviation (s) of each data set (via the function `sd()`). Plug that standard deviation into the formula above for obtaining confidence intervals. How often do these confidence intervals contain the expected value? Repeat this exercise multiple times. Is the proportion of bounded means as accurate as in the previous exercise?

As you've probably noticed, this will lead to overly confident boundaries, because we've replaced the standard error with a rough approximation to it. To correct for this, we must replace the quantiles of the Normal distribution with the quantiles of a distribution with slightly bigger tails: a 'more uncertain' distribution, called the t-distribution. This serves to correct for the extra uncertainty that we are bringing in by using our sample's standard deviation s instead of the true $SE(\hat{\theta}_n)$ of the sampling distribution:



The t-distribution has a single parameter (called the 'degrees of freedom'), and represents how much information we have about the shape of the sampling distribution. In our case, this parameter should be set to $n-1$ where n is the size of our data set. The larger our dataset, the bigger this parameter, and the closer the t-distribution will become to a Normal distribution.

Thus, to compute the 95% confidence intervals when working with a real data set with a normally distributed estimator, we can use this formula:

$$(\hat{\theta} + \frac{s}{\sqrt{n}} t_{2.5\%}, \hat{\theta} + \frac{s}{\sqrt{n}} t_{97.5\%})$$

where $t_{x\%}$ is the $x\%$ quantile function of the t-distribution with $n-1$ degrees of freedom.

Exercise: Repeat the exercise above, but instead of using the standard normal(0,1) quantiles, use the quantiles from a t-distribution with $n-1$ degrees of freedom, where n is the size of each data set. Hint: You can obtain the quantile of a t-distribution using the function `qt()`. For example, the 30% quantile of a t-distribution with 4 degrees of freedom is equal to `qt(0.3,4)`.

This is a useful formula, as the sampling distribution of the sample mean of large data sets (large n) will tend to be normal and unbiased due to the Central Limit Theorem. Thus, we can readily use this formula whenever we've obtained a large dataset and have calculated its sample mean and standard deviation!

A good rule of thumb is to use the above formula to compute CIs using the sample mean $\bar{X}$ as the estimator $\hat{\theta}$ if:

1. The data points $X_1, X_2, ..., X_n$ of your data set are known to come from a Normal distribution, so $\bar{X}$ is guaranteed to be Normally distributed, OR...

2. The data points $X_1, X_2, ..., X_n$ of your data set might not come from a Normal distribution, but n is large enough that the Central Limit Theorem kicks in, and so $\bar{X}$ is approximately Normal (generally around 30 data points).

If these assumptions don't hold, you might need to use other methods, like permutation and resampling, which we'll cover in later lectures.

7.3 Hypothesis testing

We often need to use the sampling distribution of an estimator $\hat{\theta}$ to determine whether the value of the estimator is "far enough" from a given value θ_0 to reject the hypothesis that the true expected value is equal to θ_0. How "far enough" the estimator needs to be from θ_0 - the null hypothesis - will depend on how wide the sampling distribution is.

If we're dealing with unbiased, normally distributed estimators - like the sample mean when n is large - then under the hypothesis that the true expected value $\theta = \theta_0$:

$$\hat{\theta} \sim Normal(\theta_0, Var(\hat{\theta}))$$

In other words,

$$\hat{\theta} - \theta_0 \sim Normal(0, Var(\hat{\theta}))$$

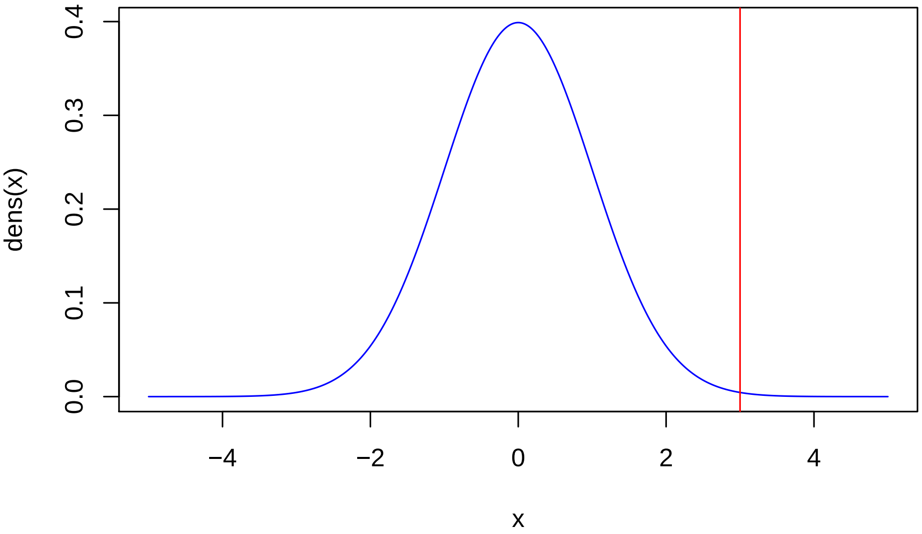
If we move the variance to the other side, we obtain:

$$\frac{\hat{\theta} - \theta_0}{\sqrt{Var(\hat{\theta})}} \sim Normal(0, 1)$$

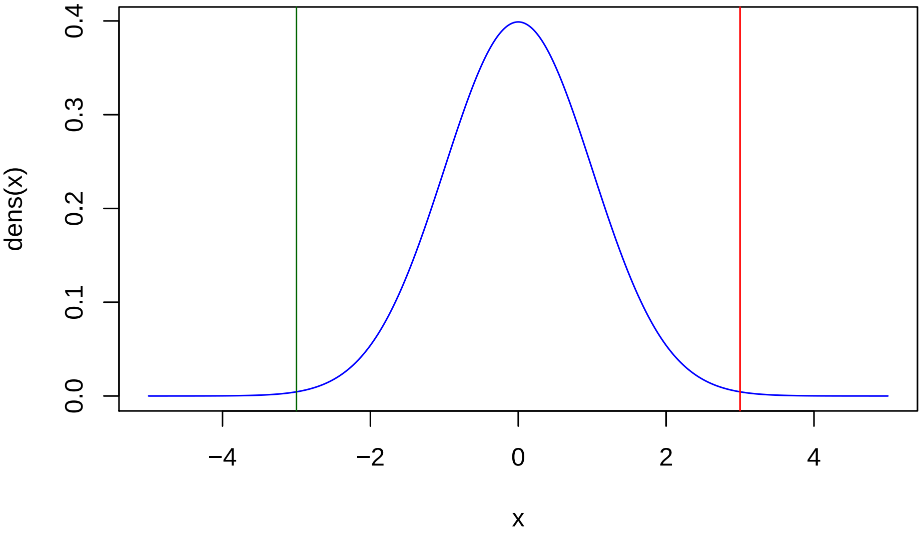
Recalling that the standard error (SE) is equal to the square root of the variance of the estimator, we can also re-write the above formula as:

$$\frac{\hat{\theta} - \theta_0}{SE(\hat{\theta})} \sim Normal(0, 1)$$

For example, in the plot below, we show a $Normal(0, 1)$ distribution, and an observed value of the difference $\hat{\theta} - \theta_0$. If the magnitude of this difference is far enough from zero, then we can reject the hypothesis that $\hat{\theta} = \theta_0$. We need to define an arbitrary cutoff for what "far enough" means. This cutoff is traditionally set such that values as extreme or more extreme than the one observed have a probability less than 5% (in other words, P-value < 0.05), but this cutoff is just a tradition.



We also need to be aware that we often don't know whether the true expected value of our distribution is *a priori* lower or higher than the value stated by our null hypothesis. In that case, we need to look at the two tails of the sampling distribution, e.g. the probability mass to the left of the green line and to the right of the red line in the plot below:



Annoyingly, to determine the "width" of the sampling distribution of the estimator, we need its standard error. If our test statistic is the sample mean and we have a large dataset (generally $n > 30$), then can again re-use the Central Limit Theorem, which states the standard error of the sample mean is equal to the standard deviation scaled by the square root of the sample size:

$$SE(\hat{\theta}) = \sqrt{\sigma^2}/\sqrt{n}$$

Thus:

$$\frac{\hat{\theta} - \theta_0}{\sqrt{\sigma^2/n}} \sim Normal(0, 1)$$

This standardized version of the sample mean is called a **z-score** and follows a standard normal distribution. However, the standard deviation in the denominator ($\sqrt{\sigma^2}$) is generally unavailable because we would only be able to know this with certainty if we knew the parameters of the distribution from which our data comes from. Instead, just as we did when computing confidence intervals, $\sqrt{\sigma^2}$ can be replaced by the standard deviation **of the data set at hand** (s), scaled by the square root of the sample size:

$$SE(\hat{\theta}) \approx s/\sqrt{n}$$

When this happens, we can no longer say that our test statistic follows a Normal distribution: the use of the sample standard deviation as a "stand-in" for the true standard deviation causes the sampling distribution of our test statistic to become wider. Our new test-statistic - called a **t-statistic** - now follows a new distribution called the **t-distribution** with $n - 1$ degrees of freedom:

$$\frac{\hat{\theta} - \theta_0}{s/\sqrt{n}} \sim t_{n-1}$$

The t-distribution is a handy distribution when performing hypothesis testing, particularly because it arises whenever we use the standard deviation of the data as a replacement for the standard error.

> The t-statistic is an example of a 'pivot statistic': like other statistics, it can be a function of observations, but its key feature is that its probability distribution does not depend on unknown parameters. You can be sure that the distribution of a t-statistic will always be the t-distribution with n-1 degrees of freedom, even if we don't know the true standard deviation of the sampling distribution of $\hat{\theta}$.

As before, a good rule of thumb is to use the above formula for performing hypothesis testing using the sample mean $\bar{X}$ as the estimator $\hat{\theta}$ if:

1. The data points $X_1, X_2, ..., X_n$ of your data set are known to come from a Normal distribution, so $\bar{X}$ is guaranteed to be Normally distributed, OR…
2. The data points $X_1, X_2, ..., X_n$ of your data set might not come from a Normal distribution, but n is large enough that the Central Limit Theorem kicks in, and so $\bar{X}$ is approximately Normal (generally around 30 data points).

Exercise: Create a single simulated data set of size 50, where each data point is drawn from a Normal distribution with expected value equal to 4 and standard deviation equal to 10. Let's pretend like you don't know the expected value from which you're simulating. Using a P-value cutoff of 5%, can you reject the null hypothesis that the expected value is equal to 5? The test will need to be `two.sided` as we don't know whether we expect the true expected value to be larger than or smaller than 5 *a priori*. Hint: you can do this in two ways, which should yield exactly the same P-values:

1. computing a t-statistic and then using twice the CDF of a t-distribution with 'n-1' degrees of freedom, to obtain the symmetric 2-tailed P-values: `2*pt()`, or…
2. using the handy function `t.test()`. You'll need to set the option `x` to be equal to your vector of data points, and the option `mu` to the particular hypothesis value you might want to test.

Exercise: Repeat the above exercise, but this time simulate a data set of size 500 instead. Using a P-value cutoff of 5%, can you now reject the null hypothesis that the expected value is equal to 5?

This exercise illustrates that the ability to reject a null hypothesis does not only depend on the difference between a statistic and its null expectation, but also in the amount of data we have, i.e. the power we have to reject the hypothesis.

> You might notice lots of similarities between the computation of confidence intervals and the testing of a hypothesis: same assumptions, same approximations, same use of the Normal distribution. Under the "frequentist" school of statistical thought, hypothesis testing and confidence interval estimation are indeed intimately related: they are two sides of the same coin!

Chapter 8

Likelihood-based inference

To understand and experiment with frequentist inference, we will focus our efforts on the Maximum Likelihood Estimates (MLE). To accomplish this, we will use two different approaches. First, we will use calculus to calculate the MLE of parameters of the exponential distribution using some meteorite landing data from NASA. We will also use grid search to narrow down the parameter estimate. Second, we will use a linear regression model to use both calculus and numerical methods to get the MLE of the parameters of a simple linear regression. Let's get started.

8.1 Meteorite data

First of all, download the meteorite data from the absalon home page under files for today's lecture. Note that you have a choice of 2 data sets, one is a pre-processed data set with only the location and year, and the second one is the full data set which you can use to play a bit more with, if you are so inclined.

Exercise 1. Let us first process the data, and plot it to visualize how many meteorites fell per year in the period 1890-1970. Also plot a histogram of this number (meteorites/year). What does it look like? Given what you know about the data, what distribution would you say works best for this type of data?

Exercise 2. Write the log-likelihood of the Poisson distribution? Using this log-likelihood, use calculus to figure out what the MLE of the rate parameter would be? Use this to get an estimate of the rate parameter for the number of meteorites that fall on the earth every year.

Exercise 3. Using the log-likelihood equation from above, plot the log-likelihood for our data over a grid of values from 0 to 25, with a spacing of 0.1. Looking at the log-likelihood values over this range, what would your estimate of the rate parameter be (Hint: Where does the log-likelihood hit its maximum?)

Exercise 4. Use the rate parameter you obtained from exercise 2 to plot the density of the Poisson distribution. Compare this to the density from the data. Is it a good fit?

Exercise X. (BONUS) Instead of using the range 1890-1970, use the range 1890-2000. What do your exploratory plots show? Find the MLE estimate for this data, and repeat the part from exercise 4. Is this a good fit now?

8.2 Simple linear regression

In this example, we will first simulate data from a linear model, then we will use inbuilt functions to estimate the parameters for our linear model, and finally we will use a numerical solver to figure out what estimates we get from maximizing the log-likelihood function for our data.

Recall that the simple linear regression model is represented by

$$Y_i = \alpha + \beta x_i + \epsilon_i$$

Also, recall that $\epsilon_i \sim N(0, \sigma^2)$, and that the observations are i.i.d. (independent and identically distributed), which just means that for all i, the ϵ_i has the same distribution and that these "error terms" are not correlated between the different i.

Since we assume that α, β and x_i are fixed quantities (and not random variables), we can compute the distribution of Y_i. We can compute the expectation of Y_i as

$$E[Y_i] = E[\alpha + \beta x_i + \epsilon_i]E[Y_i] = \alpha + \beta x_i + E[\epsilon_i]E[Y_i] = \alpha + \beta x_i$$

Similarly the variance of Y_i can be computed as

$$Var(Y_i) = Var(\alpha + \beta x_i + \epsilon_i)Var(Y_i) = Var(\epsilon_i) = \sigma^2$$

So we can conclude that $Y_i \sim N(\alpha + \beta x_i, \sigma^2)$.

Now that we know the distribution of Y_i, we can write the likelihood of the parameters (α and β) for a single observation as

$$L(\alpha, \beta) = f(Y_i|\alpha, \beta) = \frac{1}{\sigma\sqrt{2\pi}} e^{((Y_i - \alpha - \beta x_i)^2)/2\sigma^2}$$

We are ready to start the exercises now.

Exercise 5. Assume that you have n observations of Y_i and their corresponding x_i values. Use the equation above to write the joint log likelihood of n observations. (Hint: Remember that we assumed that the ϵ_i are i.i.d.) [**BONUS**: Use calculus to get MLEs of α and β.]

Exercise 6. Let us simulate some data. We will use $n = 100$, $\alpha = 1.5$, $\beta = 3$ and $\sigma^2 = 0.81$. Start with simulating 100 values for x: $x_1, \dots, x_{100}$ by using a standard uniform in the range (0,20) using the `runif` function, then simulation $\epsilon_1, \dots, \epsilon_{100}$ from $N(0, 0.81)$. Now calculate Y_is using the first equation of this section. Plot the distribution of Y_i. What does it look like?

Exercise 7. Now we can fit our simple linear regression using the inbuilt R function `lm`. After you fit the model, use the summary function to get details on the model. What do your parameters look like? Are they close enough estimates of our choice of 1.5 and 3? Can you compute the 95% confidence interval of these estimates from the output of the summary of the model?

Exercise 8. (semi-difficult) Using the log-likelihood equation you calculated in step 5, and using the inbuilt function `optim`, find the α and β parameters that maximize the log-likelihood. Do these estimates agree with the values from `lm`? (Hint: Write a R function that takes in the values of α and β `-` as a single vector `-` and returns the log-likelihood. Use this function in `optim`.)

Exercise X. (BONUS) Repeat exercise 6 and 7 with $n = 1000$. How does this affect the outcome? What happened to our estimates of the parameters `-` in terms of precision and accuracy?

Chapter 9

Bayesian Inference

9.1 Using Bayes' rule: Covid-19

For our first taste of Bayesian inference, we are going to use covid-19 quicktest data to use and understand Bayes' rule and how to use it. As you might know, there is a lot of debate in the media about the efficacy of the antigen based test (quicktest) in identifying and preventing the spread of the virus. First let us gather all the information we have on the quicktest - we will use this paper as the source of the information (https://www.medrxiv.org/content/10.1101/2021.01.22.21250042v1) - note that we do not have the whole data.

First consider that C is the event that a patient has covid-19 (C=1) or not (C=0), and T is the event that the test is positive (T=1) or negative (T=0). From the paper, we know that the sensitivity of the test, $P(T = 1|C = 1) = 0.697$, while the specificity of the test, $P(T = 0|C = 0) = 0.995$, and finally the prevalence of covid-19, $P(C = 1) = 0.046$. Using this information, we can compute the positive and negative predictive values of the test, which is $P(C = 1|T = 1)$ and $P(C = 0|T = 0)$.

Remember Bayes' rule

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

Use this to compute the two values asked for above. How does the test perform? Do you think it is useful as a standalone tracker of covid in the population?

9.2 First foray into Bayesian inference

We are going to delve into Bayesian inference by using a simple dice problem. We have three identical coins, but they are not identical in terms of their probability of tossing a head. So let us call these three coins A, B and C, and their heads probabilities are

$$A : P(A\,gives\,heads) = 0.5 B : P(B\,gives\,heads) = 0.6 C : P(C\,gives\,heads) = 0.9$$

.

We are going to design a Bayesian inference method to help us identify which coin we have using the results of a coin toss experiment. You pick a coin blindfolded, and then you toss the coins a number of times, and record how many heads and tails you have. We will use this as input data for our model.

So A is a fair coin, while B and C are biased. And *a priori* there is no way for us to know which coin you have for your toss experiments.

Now let us start by specifying the parts of our Bayesian inference framework. Again, recall the Bayesian framework - we want to compute the posterior probabilities; in this case, that would be the probability that the coin we chose is A, B or C **given** the number of heads and tails observed. Let us denote the number of heads as H and number of tails as T. So our data D consists of just the number of heads and the total number of tosses $N = H + T$. What kind of model would you choose for this data? How would you write the likelihood of this data given the probability of heads, p (Remeber the Binomial distribution).

The second step: time for us to choose a prior - What prior would you choose? You have no information for the three coins - so a uniform prior on all three coins sounds like a good idea! What would a uniform prior look like?

Putting it all together

We are ready to apply Bayesian inference to identify which coin we have. Let us start by simulating some data. Choose a coin of your liking - I am choosing the one with the heads probability of 0.9.

Exercise 1. With the coin with 0.9 probability of heads, simulate $N = 10$ coin tosses. Note the number of heads as H. Use this as input data. Now use Bayes' rule together with the Uniform prior on the 3 coins to get the posterior on which coin was chosen? What does the posterior probability look like? Are you convinced about which coin was chosen?

Exercise 2. Same as exercise 1, but with $N = 100$ and $N = 1000$ coin tosses. How does the posterior change? Did your "confidence" in which coin you chose increase, and how much?

Exercise 3. Now repeat for $N = 10, 100, 1000$ coin tosses, but use the coin with heads probability of 0.6. How did your results change? Was it easier or more difficult to separate the three coin cases?

9.3 Conjugate prior, Rejection sampling and MCMC*

In this exercise, we will try and solve problem 10.1 from the Edge book. But we will not use the rejection sampler that comes with the book, we will write our own. First, let us set up the problem. We have n independent observations $x_1, x_2, \ldots, x_n$ drawn from a Normal(θ, σ^2), and we assume that the variance σ^2 is known. We will also assume the conjugate prior for the mean θ. So the prior for θ is Normal(γ, τ^2). Choose whatever values of θ, σ^2 γ, τ^2 and n you want. Just note that if the prior is very far from the data (i.e. θ is very different than γ), your rejection sampler will be very inefficient and slow. In our example, as in the book, we will use $\theta = 2$, $\gamma = 0$, $\sigma^2 = \tau^2 = 1$, and $n = 20$.

Exercise 4. First generate your data using `rnorm` using the θ and the σ^2 given.

Exercise 5. We know that the setup we have will lead to a posterior that has the same form as the prior (Normal). And in class and in the book, we know what the expectation and variance of this posterior is. Compute these values and use them to sample 10000 points from the posterior using `rnorm`. Plot the histogram of these points.

Exercise 6. In R, first install and load the library `MCMCpack`. Then using the function `MCnormalnormal()`, draw 10000 samples from the posterior distribution. Compute the mean and variance of this sample set, and plot the distribution using a histogram.

Exercise 7. Rejection sampling. We know our prior: Normal(γ, τ^2), and our model (likelihood): Normal(θ, σ^2). So we can use the recipe for a rejection sampler to write a function for rejection samplling to sample from the posterior. Keep track of your accpetance rate (what proportion of your samples were not rejected). Use your own function to draw 10000 samples from the posterior. Use these samples to compute the mean and variance, and plot their density using a histogram.

Exercise 8. Change your γ to 4 and run your rejection sampler again. What happened to the acceptance rate? How much slower did your sampler get? Is there any way to make this faster?

Exercise X. (BONUS) Can you write a rejection sampler for a coin toss experiment. Assume that the probability of heads of a coin is p. Assume a $Beta(\alpha = 5, \beta = 5)$ prior on this probability. For generating your data, use `rbinom` with a success probability of 0.7, and draw 1000 samples. Use both the conjugate prior and rejection sampling method to estimate the parameters of the Beta distribution that will be the posterior in this case.

9.4 Bayesian point estimates and credible intervals

In the last section, we used sampling from the posterior to understand the distribution and compute statistics such as the mean and variance of the posterior distribution. We will use the results from Exercise 5-7, to compute the point estimates and the credible interval.

9.4.1 Point estimates

Remember that there are three point estimates in a Bayesian inference.

Posterior mean: $\theta_{mean} = E[\theta|D]$

Posterior median: $\theta_{med}; P(\theta < \theta_{med}|D) = 0.5$

Posterior mode: $\theta_{mode} = argmax\ f(\theta|D)$

Exercise 9. Using the samples from exercises 5-7, compute the three point estimates for your problem? How well did these three perform? Why are they all the same in our case? (Note: You can compute these for problem 5, above using that you would expect them to be for a Normal distribution given the parameters of the posterior Normal distribution.) What happened to the point estimates when you used a prior of exercise 8.

Exercise X. (BONUS) If you did the last bonus exercise, then compute the point estimates for the coin toss problem.

9.4.2 Credible intervals

Recall that the $(1 - \alpha)$ credible interval for a parameter is any interval (a, b) such that $P(a < \theta < b|D) = 1 - \alpha$.

Exercise 10. Using the samples from exercise 7 and 8, compute the 95% credible interval for our posterior. How many such intervals exist? Can you give a couple of examples of 95% credible intervals? How would you choose one?

Exercise X. (BONUS) If you did this for the beta and binomial setup in the BONUS problems, then compute the 95% credible interval for p. Is the quantile based credible interval the highest posterior density interval?

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.

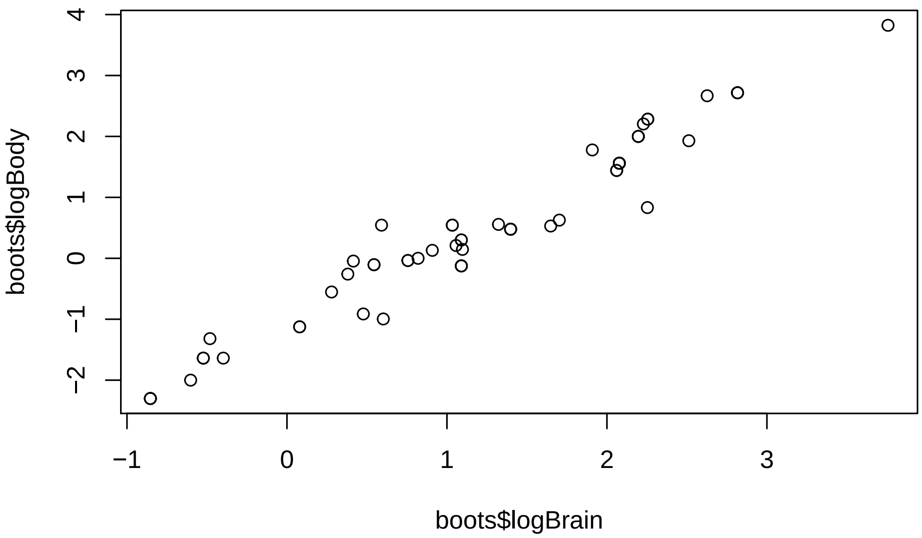
```
# Load table
allisontab <- tibble(read.csv("Data_allison.csv"))
# Remove rows with missing data in columns of interest
allisontab <- filter(allisontab,!is.na(BrainWt) & !is.na(BodyWt) & !is.na(TotalSleep))
# Log-scale body and brain weight
allisontab <- mutate(allisontab,logBody=log10(BodyWt), logBrain=log10(BrainWt))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

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:

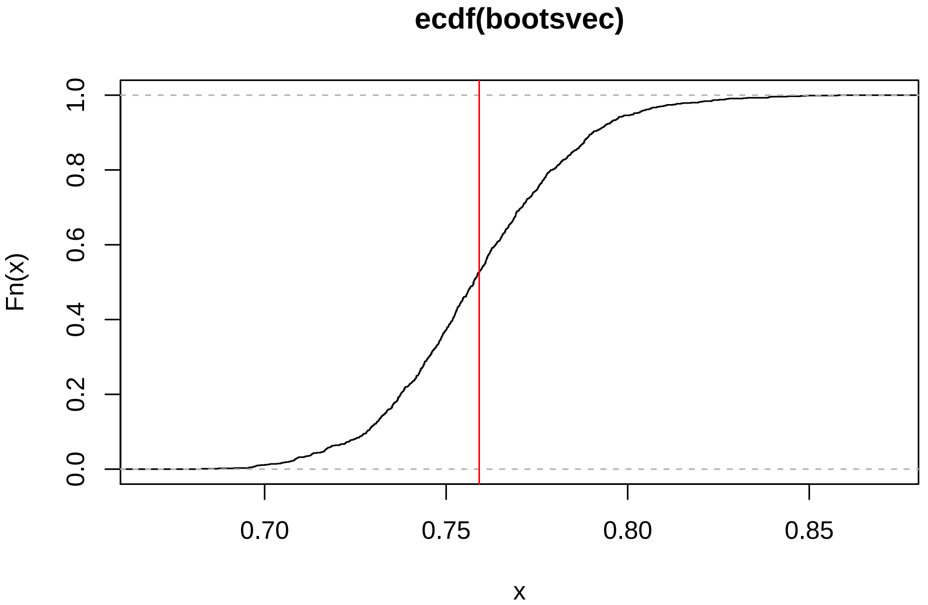
```
estimate <- lm(logBrain ~ logBody, data=allisontab)$coeff[2]
estimate
```

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

Exercise: try estimating the same parameter from a series of 1000 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. 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).

The standard approach assumes that the bootstrap distribution is a Normal distribution. Recall from the lecture on properties of estimators that we can obtain CIs from a Normally-distributed summary statistic θ by using the standard error of the statistic: $SE(\hat{\theta}_n)$:

$$(\hat{\theta} - SE(\hat{\theta}_n)q_{97.5\%}, \hat{\theta} + SE(\hat{\theta}_n)q_{97.5\%})$$

Where $q_{97.5\%}$ is a quantile from a standard N(0,1) Normal distribution, and $SE(\hat{\theta}_n)$ is the standard error (the standard deviation of our estimator). The standard error approach to obtain boostrap confidence intervals proceeds by replacing the standard error $SE(\hat{\theta}_n)$ with the standard deviation of the bootstrap distribution as a stand-in for it. Let's call this last value $SE_b(\hat{\theta}_n)$. Then:

$$(\hat{\theta} - SE_b(\hat{\theta}_n)q_{97.5\%}, \hat{\theta} + SE_b(\hat{\theta}_n)q_{97.5\%})$$

In R, we can thus obtain the CI around the mean as follows:

```
# Let bootsvec be our vector of boostrapped parameters from the previous exercise
bootsmean <- mean(bootsvec)
SEb <- sd(bootsvec)
NormalQ <- qnorm(0.975,mean=0, sd=1)
CIs <- c( bootsmean - SEb*NormalQ, bootsmean + SEb*NormalQ )
CIs
```

```
## [1] 0.7079987 0.8092793
```

The percentile approach involves obtaining percentiles of the bootstrap distributions. If we are interested in a 95% CI, for example, we just look for the values of the distribution that correspond to the 2.5% percentile and the 97.5% percentile, so that the central boostrap distribution mass between them adds up to 95%.

```
# Let bootsvec be our vector of boostrapped parameters from the previous exercise
CIs <- quantile(bootsvec, c(0.025,0.975))
CIs
```

```
##      2.5%     97.5%
## 0.7082542 0.8125138
```

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
##
```

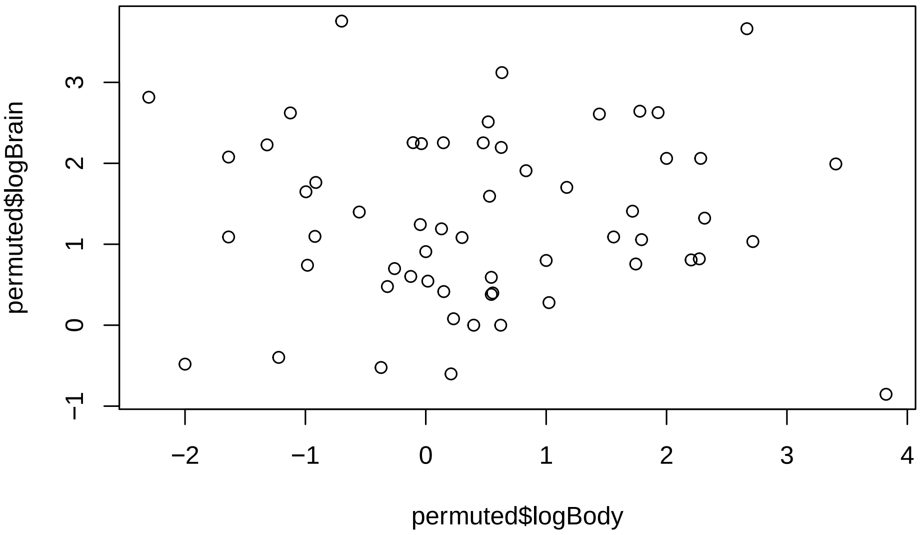
```
## Residual standard error: 0.3071 on 56 degrees of freedom
## Multiple R-squared:  0.9183, Adjusted R-squared:  0.9168
## F-statistic: 629.2 on 1 and 56 DF,  p-value: < 2.2e-16
```

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
## 0.02403355
```

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.

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:

```r
if (!require("vegan")) install.packages("vegan")
if (!require("devtools")) install.packages("devtools")
if (!require("ggplot2")) install.packages("ggplot2")
```

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

```r
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. We use

```
irisnumvar <- iris[-5] # Remove the categorical variable
fit<-prcomp(irisnumvar, scale=TRUE) # Perform PCA
```

Exercise: Try using the `summary()` and `plot()` functions to obtain a summary of the resulting PCA object How many principal components were created? (note that the number of PCs always equals the number of original variables). How much variance does the first principal component serve to explain in our data? How much variance does the second component explain? How many PCs would we need to be able to explain at least 95% of the variation in our data?

The "Rotation" matrix is included inside the `fit` object we just constructed. You can retrieve it by typing `fit[2]$rotation`. This matrix contains the "loadings" of each of the original variables on the newly created PCs.

Exercise: Take a look at the rotation matrix. The larger the absolute value of a variable in each PC, the more that variable contributes to that PC. For each component, use the function `barplot()` to plot the loadings (contributions) of each variable into that component. Which variables contribute most to each component?

Exercise: 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. 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. Based on the previous exercise (looking at the rotation matrix), which variables do you think would be most different between samples in one group and in the other?

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:

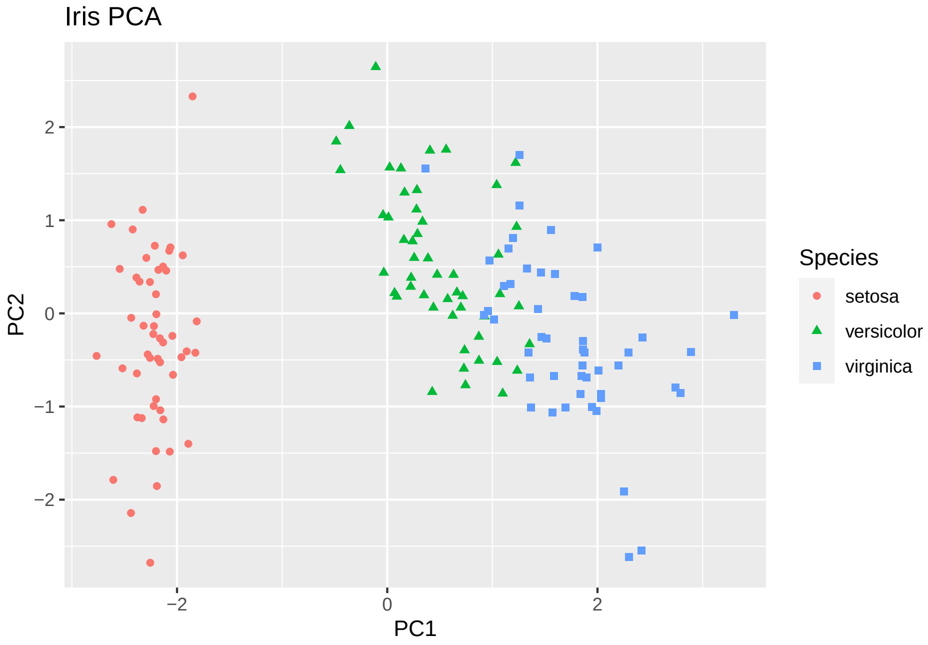
```
myEig <- eigen(cormat)
```

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
iristoplot <- data.frame(scoresLONG,iris$Species)
colnames(iristoplot) <- c("PC1","PC2","PC3","PC4","Species")
ggplot(iristoplot, aes(PC1, PC2)) +
  geom_point(aes(color=Species, shape = Species)) +
  xlab("PC1") +
  ylab("PC2") +
  ggtitle("Iris PCA")
```



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 function `range()` returns a vector containing the minimum and maximum of a given vector. Using this function, we can observe that 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 Principal components as explanatory variables

We can use principal components as explanatory variables to any linear model. In this case, we'll use the first two principal components of the PCA we performed above, to perform a logistic regression on the probability that an individual belongs to the species 'virginica'. First, let's create a new variable that is equal to 1 if an individual belongs to this species, and is 0 otherwise. We'll use this variable as the response variable

```
isvirginica <-  as.numeric(iris[,5] == "virginica")
```

We now collate the principal components from the exercise above into a new dataframe that also includes the response variable we just created.

```
# The PC scores are stored in the fifth element of fit. Here, we could have also used
PC.scores <- fit[5]
newiris <- data.frame(PC.scores,isvirginica)
colnames(newiris) <- c("PC1","PC2","PC3","PC4","isvirginica")
head(newiris)
```

```
##         PC1        PC2         PC3          PC4 isvirginica
## 1 -2.257141 -0.4784238  0.12727962  0.024087508           0
## 2 -2.074013  0.6718827  0.23382552  0.102662845           0
## 3 -2.356335  0.3407664 -0.04405390  0.028282305           0
## 4 -2.291707  0.5953999 -0.09098530 -0.065735340           0
## 5 -2.381863 -0.6446757 -0.01568565 -0.035802870           0
## 6 -2.068701 -1.4842053 -0.02687825  0.006586116           0
```

Exercise: use the `glm()` function on the newly created `newiris` data-frame, to perform a logistic regression for the probability that an individual belongs to the species virginica, using the first two principal components (PC1 and PC2) as explanatory variables. Do both components have fitted effects that are significantly different from 0? Do these results make sense in light of the PCA biplots created in the sections above?

Exercise: Compare the logistic model to another logistic model, this time using only PC1 as the explanatory variable. Which model has the highest AIC score?

14.5 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):

```
require(vegan)
data(dune)
head(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
##   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
##   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
##    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
```

Note that here linearity is not a good assumption to make for our ordination: a difference between a site containing 0 specimens (absence) and a site containing 1 specimen is conceptually larger than a difference between a site containing 1 specimen and another site containing 2. In other words, the difference between presence and absence is more important than a difference in quantity of specimens. Thus, 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

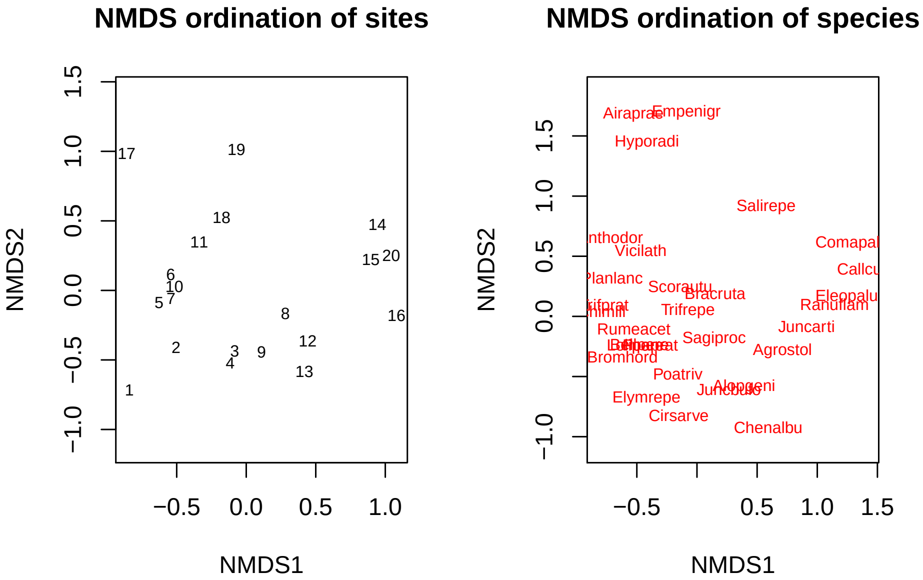
Let's perform NMDS ordination using the Bray-Curtis dissimilarity. Remember that, unlike PCA, NMDS requires us to specify the number of dimensions (k) a priori (the default in vegan is 2). It also performs a series of transformations on the data that are appropriate for ecological data (default: autotransform=TRUE). 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, distance="bray")
```

```
## Run 0 stress 0.1192678
## Run 1 stress 0.1183186
## ... New best solution
## ... Procrustes: rmse 0.02026759  max resid 0.06494227
## Run 2 stress 0.1183186
## ... New best solution
## ... Procrustes: rmse 3.285599e-05  max resid 0.000108675
## ... Similar to previous best
## Run 3 stress 0.1183186
## ... Procrustes: rmse 2.054674e-05  max resid 6.534918e-05
## ... Similar to previous best
## Run 4 stress 0.1183186
## ... Procrustes: rmse 7.252381e-05  max resid 0.0002360155
## ... Similar to previous best
## Run 5 stress 0.1192679
## Run 6 stress 0.1183186
## ... Procrustes: rmse 3.356249e-05  max resid 7.659093e-05
## ... Similar to previous best
## Run 7 stress 0.1192678
## Run 8 stress 0.1183186
## ... Procrustes: rmse 4.807861e-06  max resid 1.495597e-05
## ... Similar to previous best
## Run 9 stress 0.1183186
## ... New best solution
## ... Procrustes: rmse 1.292393e-05  max resid 3.601861e-05
## ... Similar to previous best
## Run 10 stress 0.2035424
## Run 11 stress 0.1192679
## Run 12 stress 0.1183186
## ... Procrustes: rmse 1.99821e-05  max resid 5.482749e-05
## ... Similar to previous best
## Run 13 stress 0.1183186
## ... Procrustes: rmse 3.48063e-05  max resid 0.0001079752
## ... Similar to previous best
## Run 14 stress 0.1183186
## ... New best solution
## ... Procrustes: rmse 3.47904e-06  max resid 8.511745e-06
## ... Similar to previous best
## Run 15 stress 0.1183187
## ... Procrustes: rmse 5.767978e-05  max resid 0.0001773045
## ... Similar to previous best
## Run 16 stress 0.1183186
## ... Procrustes: rmse 3.648247e-05  max resid 0.0001157266
## ... Similar to previous best
## Run 17 stress 0.1192682
## Run 18 stress 0.1192678
## Run 19 stress 0.1192682
## Run 20 stress 0.1192681
## *** 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,choices=c(1,2),display="sites",main="NMDS ordination of sites",type="t")
plot(ord,choices=c(1,2),display="species",main="NMDS ordination of species",type="t")
```



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

Here, the option `choices` determines which NMDS dimensions are being plotted. We only have two dimensions, so there's only two dimensions that we can plot here. In turn, the option `display` allows us to plot an ordination of the sites (assuming species are properties of each site), or of the species (assuming sites are properties of each species).

Exercise: Take a look at the plots. Which species tend to co-occcur with each other? Which sites tend to have similar species compositions?

Exercise: Change the number of dimensions and re-run the ordination. Note that you'll have to create multiple plots to observe all the dimensions if there are more than 2 of them. How do the results change?

Exercise: Change the distance metric used and re-run the ordination with k=2 (e.g. try using the Euclidean distance instead). 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")
if (!require("NbClust")) install.packages("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: Take a look at the first row of this matrix (`d[1,]`). Under the chosen distance metric, what is the most disparate observation from specimen

1? What is the distance between this specimen and specimen 1? Do these specimens belong to different species? (Hint: the species ID is stored in the origin `iris` data frame)

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.

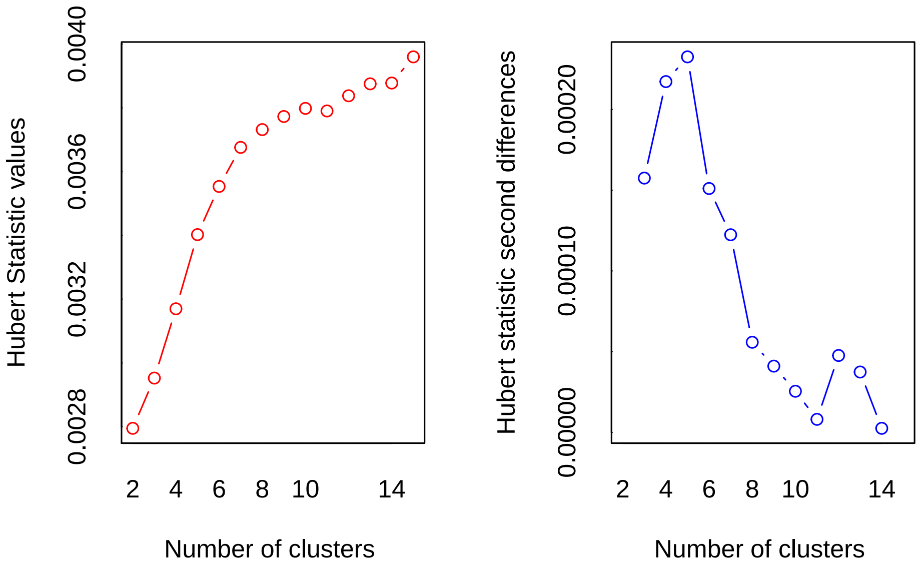
The first of these details is that our variables lie on different scales: some span a wider range of variation than others. A common practice in clustering is to scale all variables in our data such that they are mean-centered and have a standard deviation of 1, before performing the clustering. This 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])
```

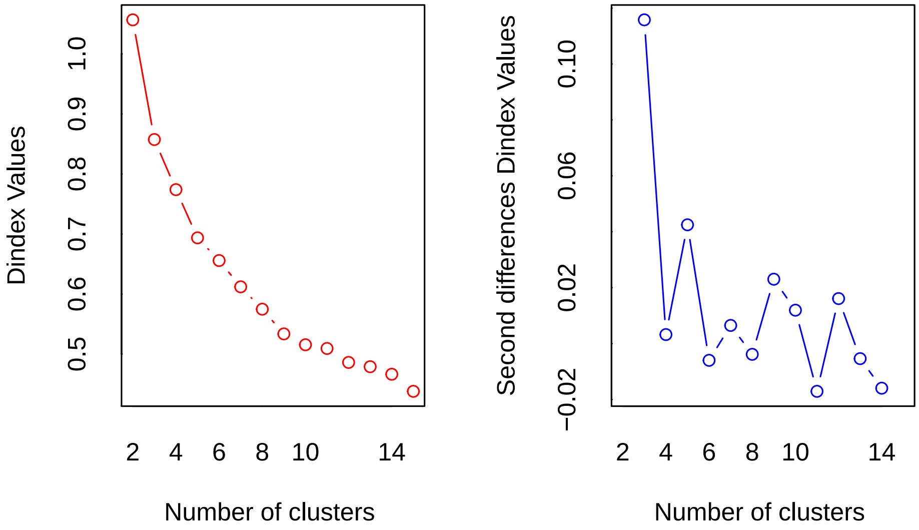
Exercise: what's the mean and standard deviation of each of the four variables in the iris dataset before scaling? what is their mean and standard deviation after scaling?

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.
##
```

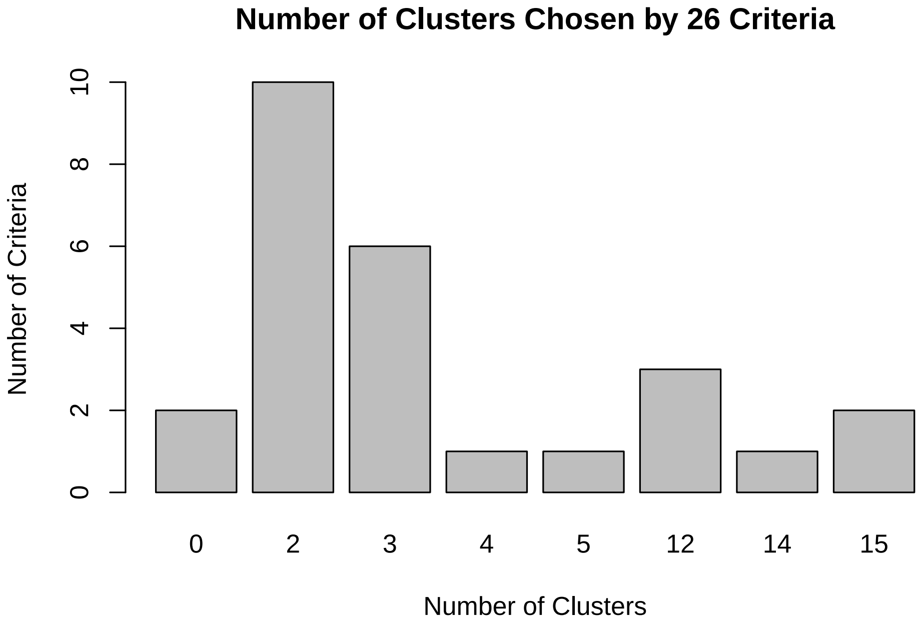
```
## *******************************************************************
## * Among all indices:
## * 10 proposed 2 as the best number of clusters
## * 6 proposed 3 as the best number of clusters
## * 1 proposed 4 as the best number of clusters
## * 1 proposed 5 as the best number of clusters
## * 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)
```

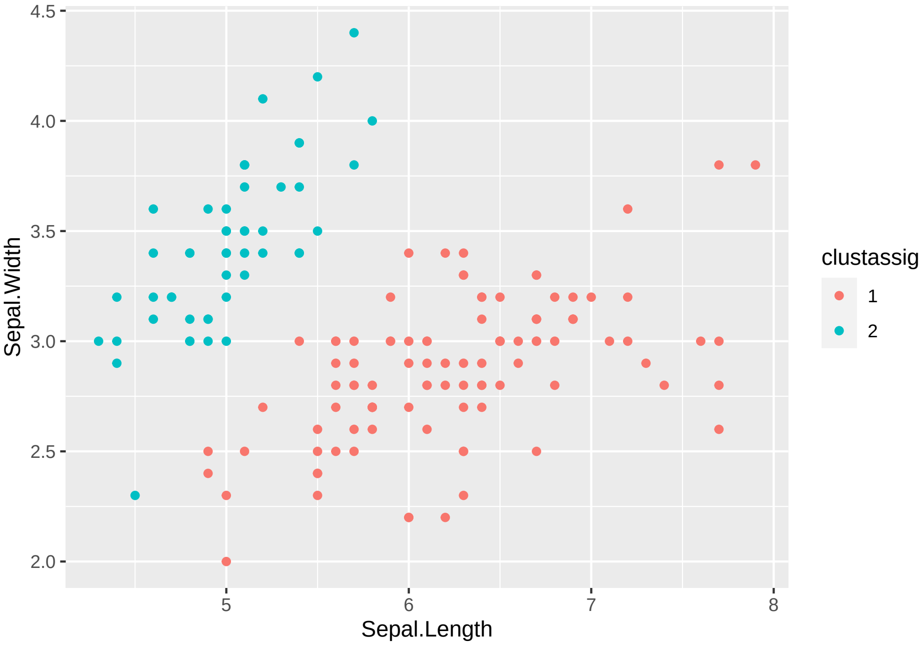
The cluster assignments for all observations are stored in the "cluster" attribute of the resulting object:

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

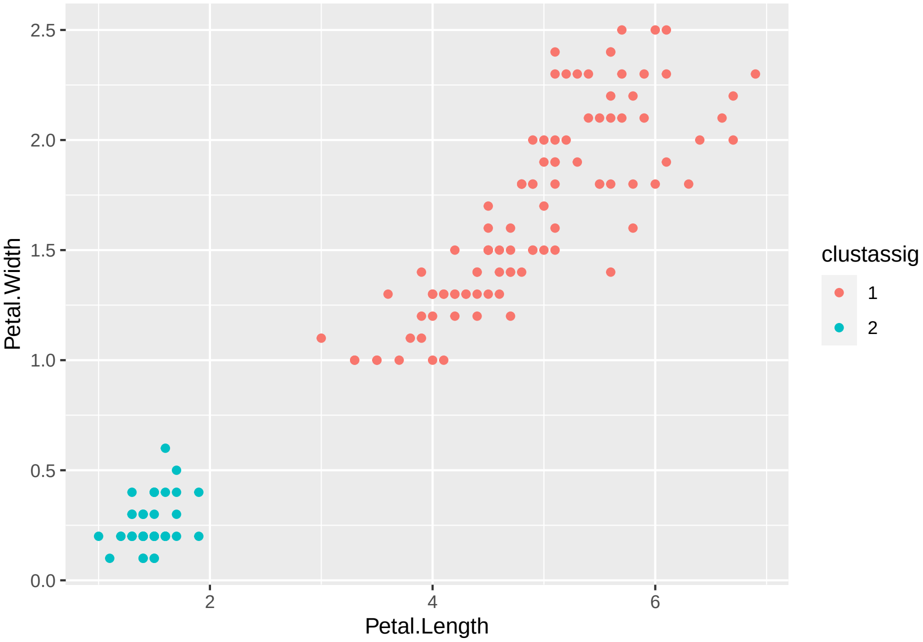
```
##   [1] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
##  [38] 2 2 2 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##  [75] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [112] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1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

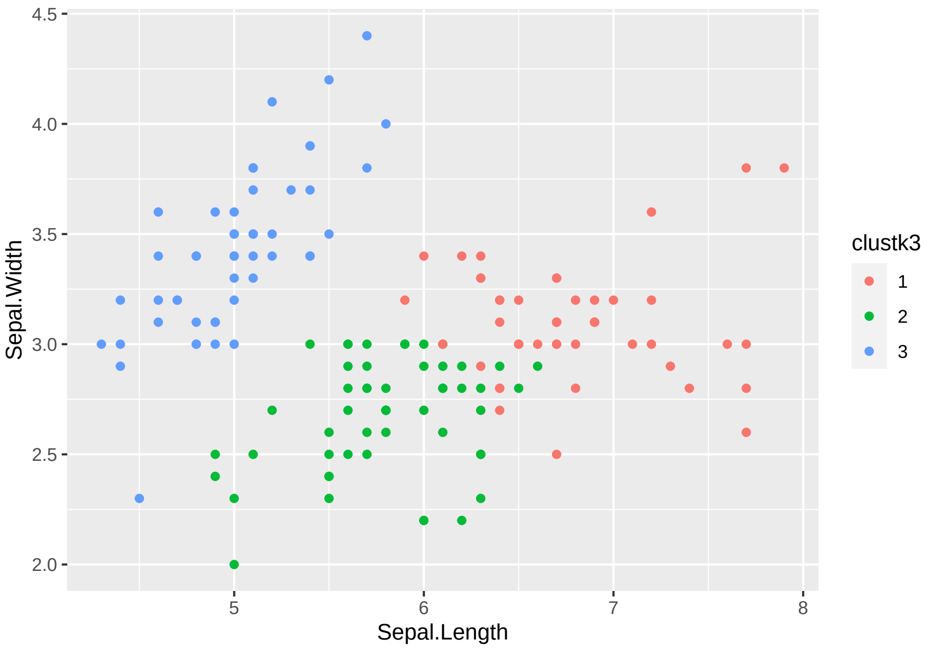
```
clustassig <- as.factor(clustassig) # Treat the cluster assignments as discrete factors
irisclust <- data.frame(iris,clustassig) # Combine data with cluster assignments
ggplot(data=iris) + geom_point(aes(x=Sepal.Length,y=Sepal.Width,color=clustassig)) # Plot sepal
```

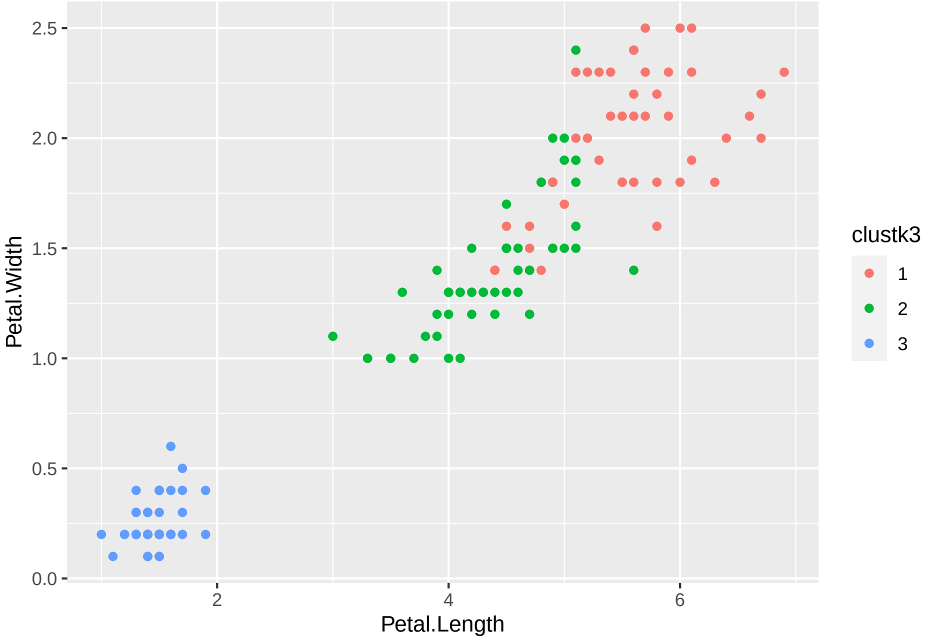


```
ggplot(data=iris) + geom_point(aes(x=Petal.Length,y=Petal.Width,color=clustassig)) # P
```



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 nic
```

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.

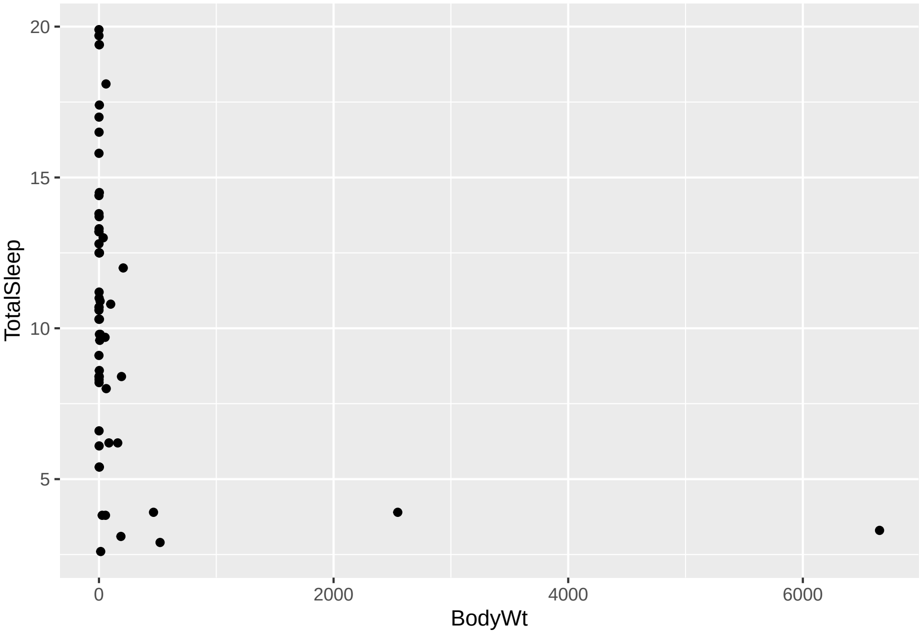
```
summary(allisontab)
```

```
##    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))
```

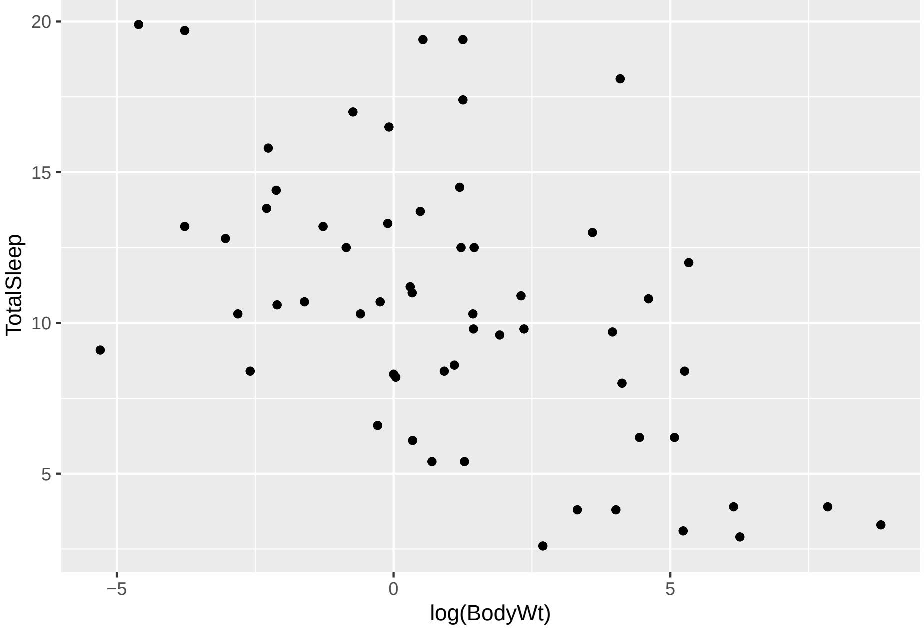
```
## Warning: Removed 4 rows containing missing values (geom_point).
```



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 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 control the amount of error using a variable called `sigma.res`. We'll assume errors are homoscedastic (have the same variance) in this exercise.

```
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)
  # Simulate the error values using the specified standard deviation for the residuals
  error <- rerror(n,0,sigma.res)
  # 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 residual errors (make `sigma.res` 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)

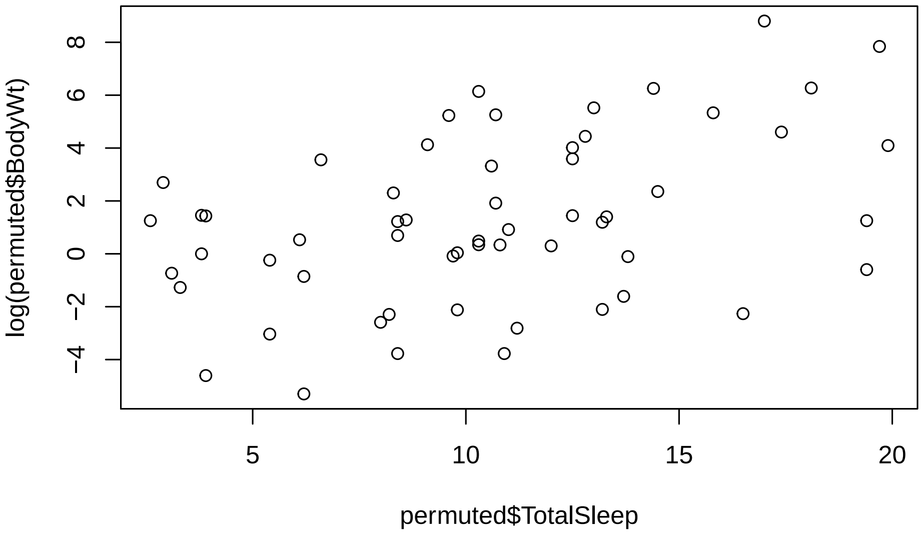
The above t-test makes assumptions on our data that sometimes may not be warranted. Most importantly, the t-test assumes we have a large number of samples, which might not always be the case. We can perform a more robust test that makes less a priori assumptions on our data - a permutation test. To do 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)
##   0.6247139
```

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 the p-value form the t-statistic testing the same relationship in your fitted linear model.

```
##
##  Approximative Spearman Correlation Test
##
## data:  TotalSleep by log(BrainWt)
## Z = -4.2069, p-value < 0.001
## alternative hypothesis: true rho is not equal to 0
```

```
##
## Call:
## lm(formula = TotalSleep ~ log(BrainWt), data = allisontab)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -6.7029 -3.2406  0.0498  2.5162  9.0210
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)   13.7245     0.8046  17.058  < 2e-16 ***
## log(BrainWt)  -1.0571     0.2076  -5.092  4.3e-06 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 3.842 on 56 degrees of freedom
##   (4 observations deleted due to missingness)
## Multiple R-squared:  0.3165, Adjusted R-squared:  0.3043
## F-statistic: 25.93 on 1 and 56 DF,  p-value: 4.297e-06
```

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?