Essentials of Mathematics and Statistics

Practical: Module 1

Anas A Rana

2021-10-12

Contents

1 Introduction

This is part of the practical for Module 1 - Essentials of Mathematics and Statistics part of the MSc Bioinformatics at the University of Birmingham.

This website hosts all practicals for **Module 1**, which covers:

- Linear regression
- Principal Component Analysis (PCA)
- Multivariate Regression
- Generalised Linear Models

1.1 How to use these resources

Section 2 covers some basic concepts on using `R` and `Rstudio`. You should have already covered it, but this is a refresher.

Section 3 contains a link to one of the locations you can download the data from.

The sections after that contain the content of the practical we will go through.

2 Getting started in R and Rstudio

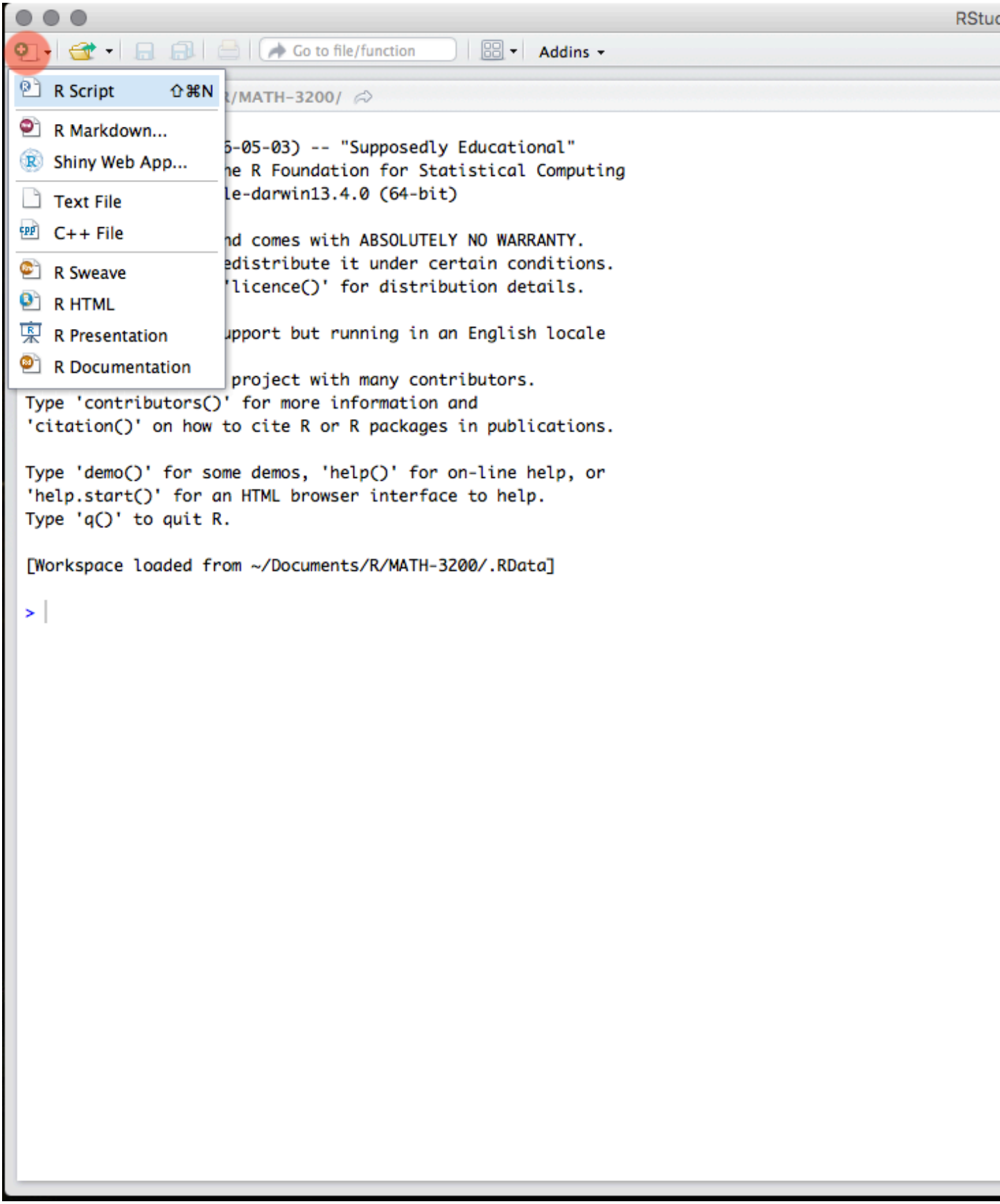
You have already started working with `R`, here are just some of the core principles revisited. Ensure you understand them as they are required knowledge and will not be revisited during the module.

2.1 R scripts

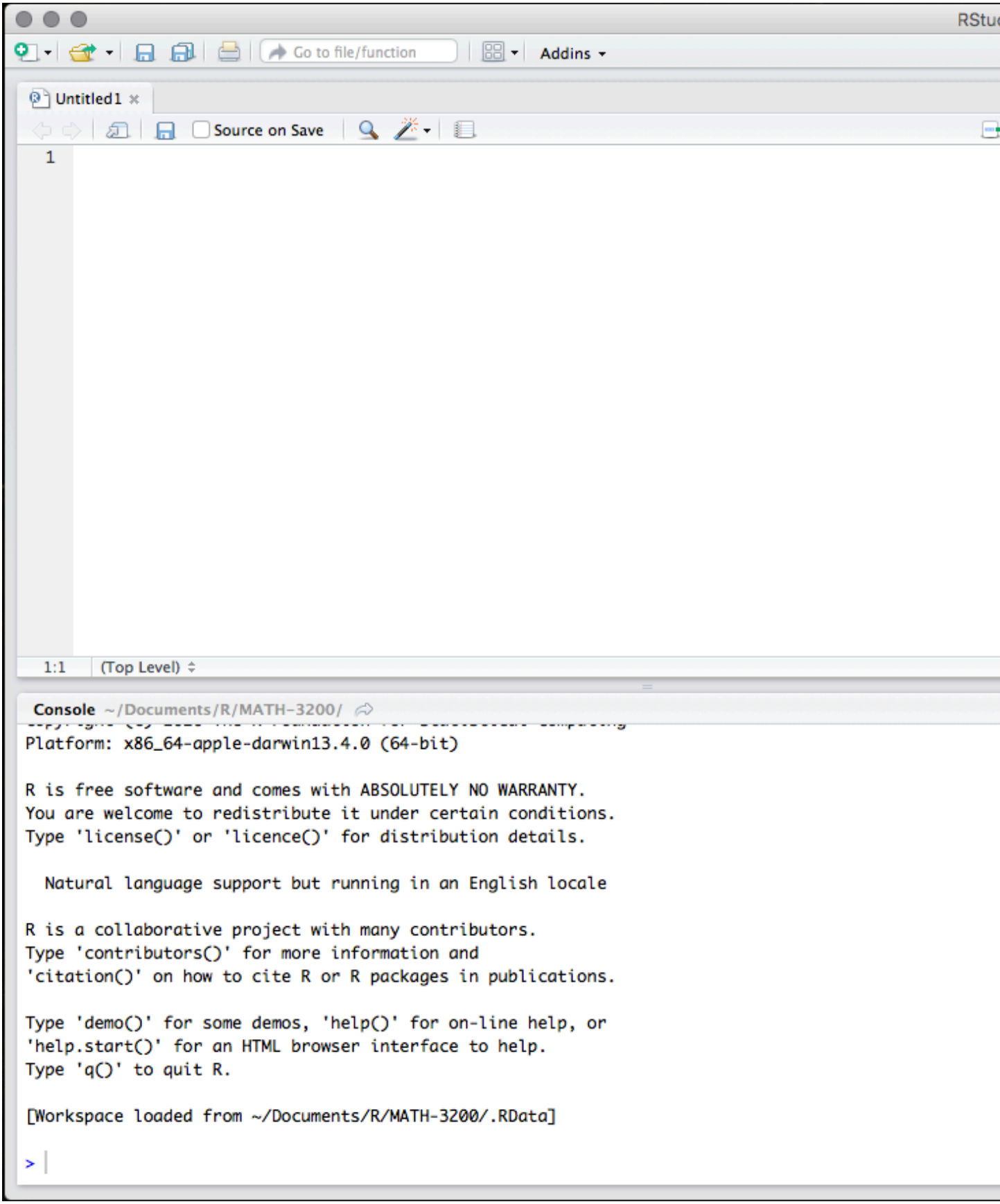
While entering and running your code at the R command line is effective and simple. This technique has its limitations. Each time you want to execute a set of commands, you have to re-enter them at the command line. Complex commands are potentially subject to typographical errors, necessitating that they be re-entered correctly. Repeating a set of operations requires re-entering the code stream. Fortunately, R and RStudio provide a method to mitigate these issues. R scripts are that solution. A script is simply a text file containing a set of commands and comments. The script can be saved and used later to re-execute the saved commands. The script can also be edited so you can execute a modified version of the commands.

2.2 Creating an R script

It is easy to create a new script in RStudio. You can open a new empty script by clicking the New File icon in the upper left of the main RStudio toolbar. This icon looks like a white square with a white plus sign in a green circle. Clicking the icon opens the New File Menu. Click the R Script menu option and the script editor will open with an empty script.

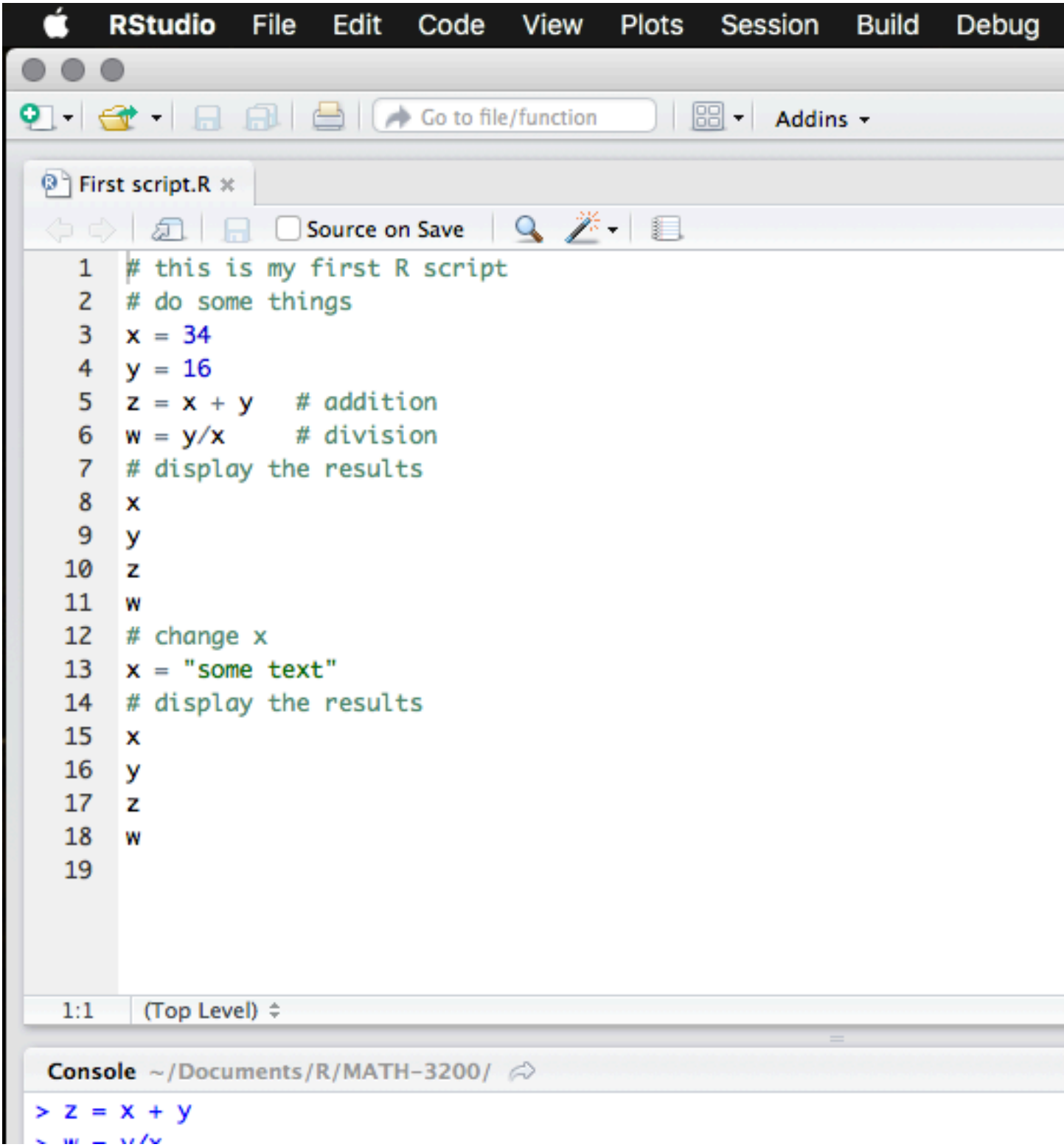

RStud
Go to file/function
Addins
R Script ⇧⌘N
R Markdown...
Shiny Web App...
Text File
C++ File
R Sweave
R HTML
R Presentation
R Documentation
/MATH-3200/
6-05-03) -- "Supposedly Educational"
he R Foundation for Statistical Computing
le-darwin13.4.0 (64-bit)
nd comes with ABSOLUTELY NO WARRANTY.
edistribute it under certain conditions.
'licence()' for distribution details.
upport but running in an English locale
project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.
Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.
[Workspace loaded from ~/Documents/R/MATH-3200/.RData]
>

Once the new script opens in the Script Editor panel, the script is ready for text entry, and your RStudio session will look like this.


RStu
Go to file/function
Addins
Untitled1
Source on Save
1
1:1
(Top Level)
Console ~/Documents/R/MATH-3200/
Platform: x86_64-apple-darwin13.4.0 (64-bit)
R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.
Natural language support but running in an English locale
R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.
Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.
[Workspace loaded from ~/Documents/R/MATH-3200/.RData]
>

Here is an easy example to familiarize you with the Script Editor interface. Type the following code into your new script *(later topics will explain the specific code components do)*.

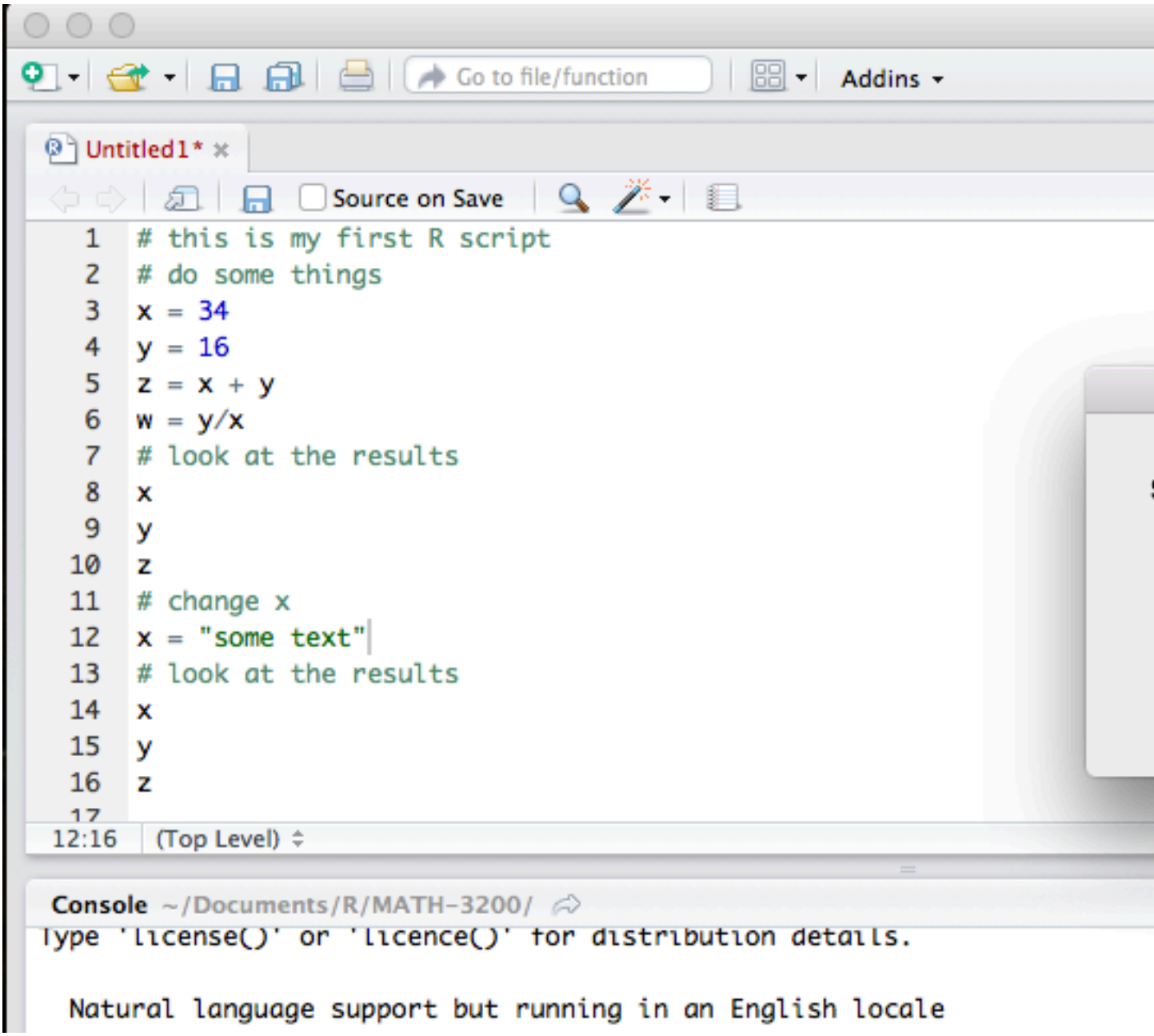
```
# this is my first R script
# do some things
x = 34
y = 16
z = x + y    # addition
w = y/x      # division
# display the results
x
y
z
w
# change x
x = "some text"
# display the results
x
y
z
w
```



There, you now have your first R script. Notice how the editor places a number in front of each line of code. The line numbers can be helpful as you work with your code. Before proceeding on to executing this code, it would be a good idea to learn how to save your script.

2.3 Saving an R script

You can save your script by clicking on the Save icon at the top of the Script Editor panel. When you do that, a Save File dialog will open.



2.4 Executing code in an R script

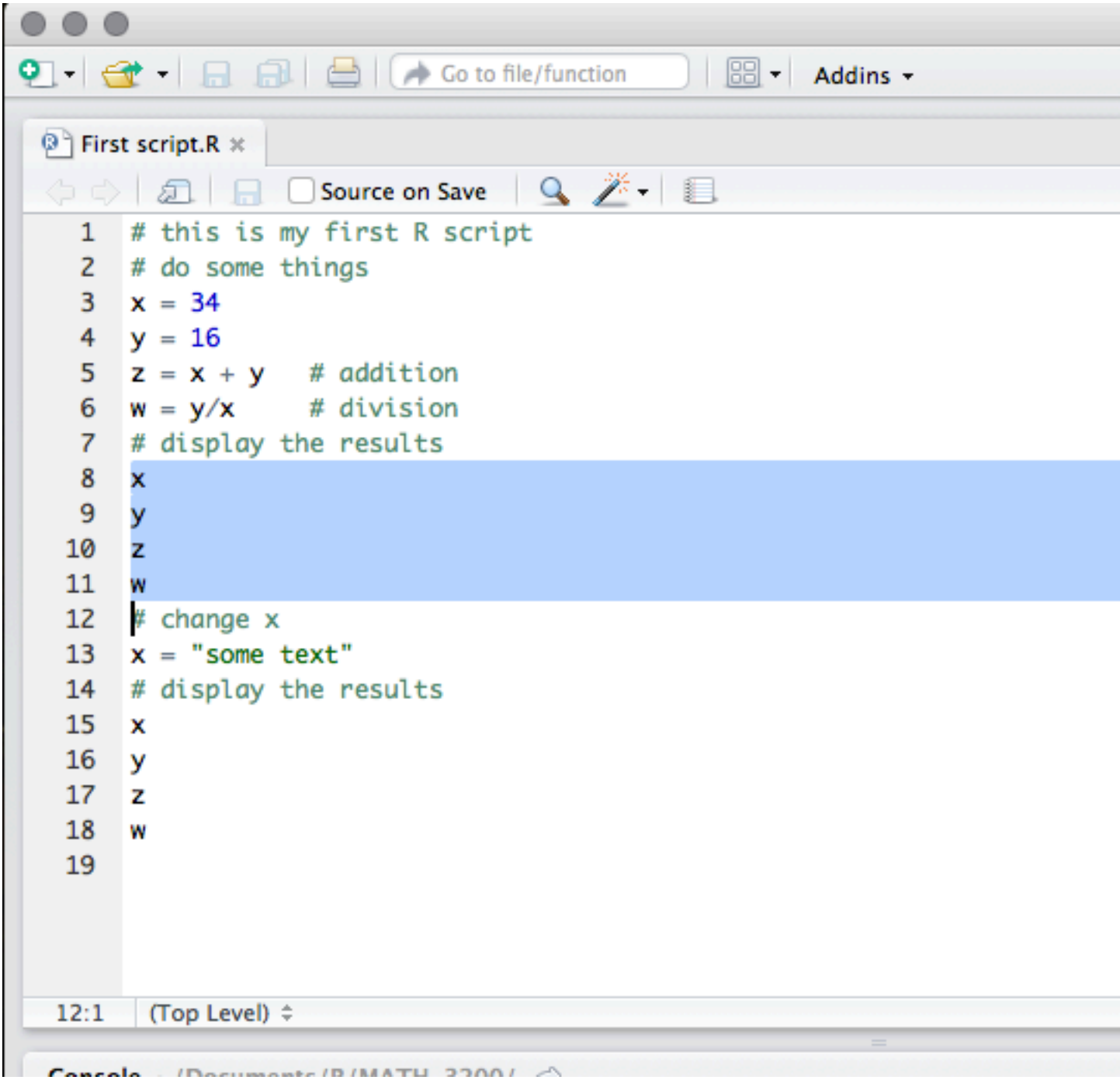
You can run the code in your R script easily. The Run button in the Script Editor panel toolbar will run either the current line of code or any block of selected code. You can use your First script.R code to gain familiarity with this functionality.

Place the cursor anywhere in line 3 of your script

$$x = 34$$

. Now press the Run button in the Script Editor panel toolbar. Three things happen: 1) the code is transferred to the command console, 2) the code is executed, and 3) the cursor moves to the next line in your script. Press the Run button three more times. RStudio executes lines 4, 5, and 6 of your script.

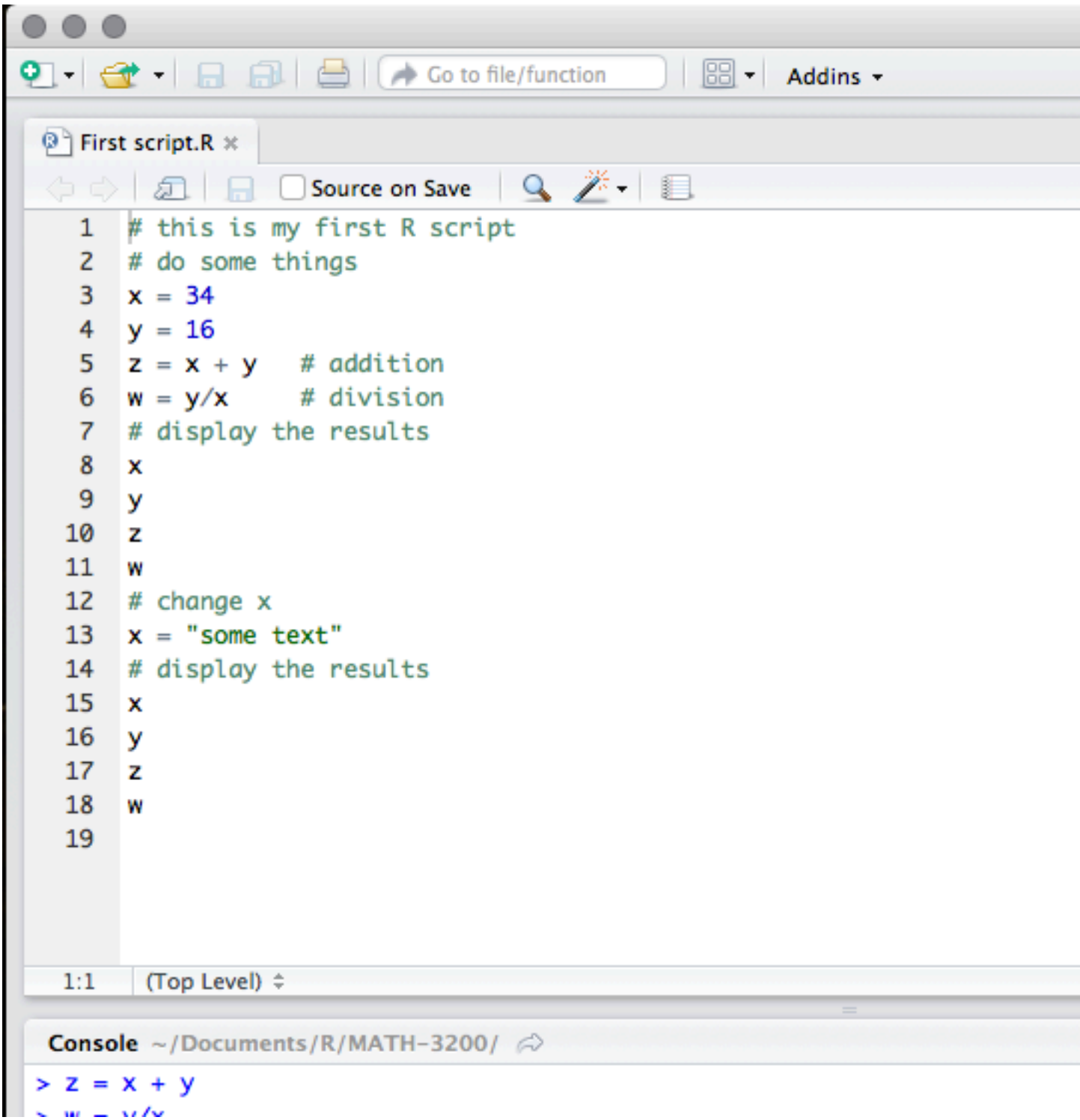
Now you will run a set of code commands all at once. Highlight lines 8, 9, 10, and 11 in the script.



Highlighting is accomplished similar to what you may be familiar with in word processor applications. You click your left mouse button and the beginning of the text you want to highlight, you hold the mouse button and drag the cursor to the end of the text and release the button. With those four lines of code highlighted, click the editor Run button. All four lines of code are executed in the command console. That is all it takes to run script code in RStudio.

2.4.1 Comments in an R script (documenting your code)

Before finishing this topic, there is one final concept you should understand. It is always a good idea to place comments in your code. They will help you understand what your code is meant to do. This will become helpful when you reopen code you wrote weeks ago and are trying to work with again. The saying, "Real programmers do not document their code. If it was hard to write, it should be hard to understand" is meant to be a dark joke, not a coding style guide.



A comment in R code begins with the # symbol. Your code in First script.R contains several examples of comments. Lines 1, 2, 7, 12, and 14 in the image above are all comment lines. Any line of text that starts with # will be treated as a comment and will be ignored during code execution. Lines 5 and 6 in this image contain comments at the end. All text after the # is treated as a comment and is ignored during execution.

Notice how the RStudio editor shows these comments colored green. The green color helps you focus on the code and not get confused by the comments.

Besides using comments to help make your R code more easily understood, you can use the # symbol to ignore lines of code while you are developing your code stream. Simply place a # in front of any line that you want to ignore. R will treat those lines as comments and ignore them. When you want to include those lines again in the code execution, remove the # symbols and the code is executable again. This technique allows you to change what code you execute without having to retype deleted code.

3 Data sets

For each practical there are some datasets required, you will find all data required for week two of practicals in the data folder here. Links to individual datasets requried can be found at the beginning of each practical or on Canvas.

4 Simulating random numbers

There are a number of functions in R that you can use to simulate random numbers according to different probability distributions.

The function sample allows you to take a sample of the specified size from the elements of a vector xusing sampling with or without replacement. You can use ?sample to read the documentation describing the command.

In the following, we will use the sample function to make 10,000 draws from the set of numbers 1, 2, 3 and 4 and display the distribution of the sampled values using a histogram.

First, we define a vector called x which contains the numbers 1, 2, 3, and 4. The function c allows us to combine these four numbers together into one vector:

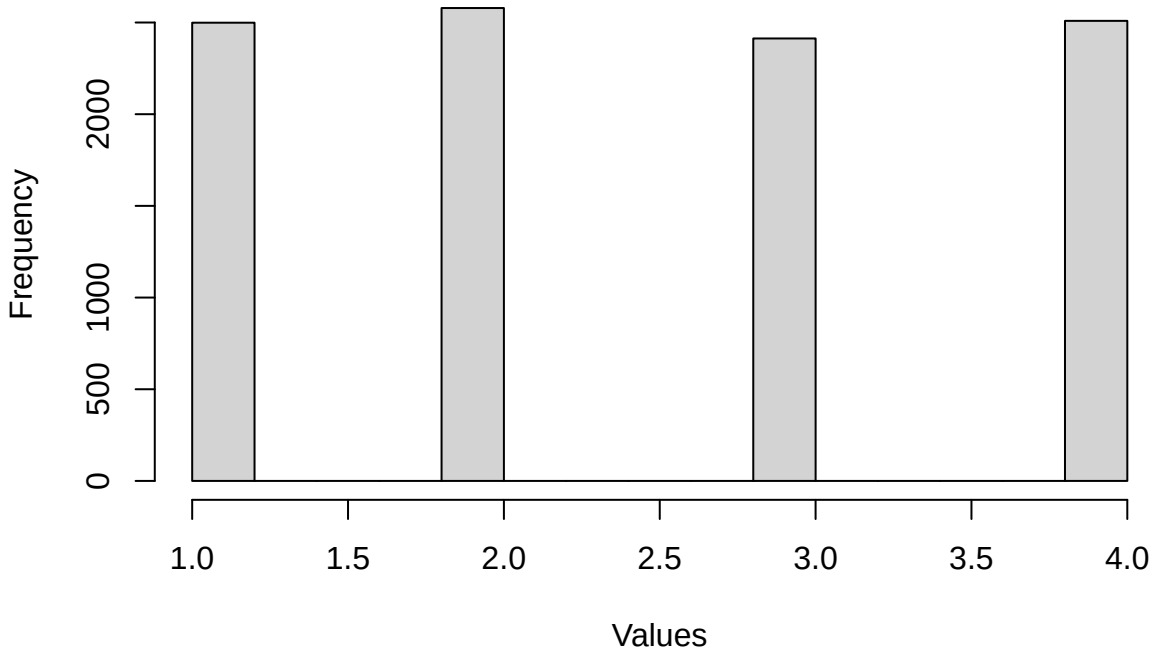
```
x <- c(1, 2, 3, 4)
```

We now use the function sample to pick from those four numbers in x 10,000 times. The result, the 10,000 numbers chosen, is store in out:

```
out <- sample(x, 10000, replace=TRUE)
```

Lets plot a histogram of the values picked:

```
hist_out <- hist(out, main = '', xlab = 'Values', ylab = 'Frequency')
```



We picked each number with equal probability so the histogram shows each number is equally likely to have been chosen.

Problem

What is the difference in the output `out1` and `out2` in the following piece of code?

```
x <- c( 1, 2, 2, 3, 4, 1, 6, 7, 8, 10, 5, 5, 1, 4, 9 )
out1 <- sample(x, 10, replace=FALSE)
out2 <- sample(x, 10, replace=TRUE)
```

The option `replace=TRUE` activates sampling with replacement (i.e. the numbers that are picked are put back and can be picked again).

The option `replace=FALSE` activates sampling without replacement (i.e. the numbers that are picked are not put back and cannot be picked again).

Problem

Use the `sample` or `sample.int` function to simulate values from rolls of an unbiased six-sided die. Show that the distribution of values you obtain is consistent with an unbiased die.

Hint 1: Type `?sample.int` in the console to get help on this function.

Hint 2: You may find it useful to use the function `table`. Type `?table` in the console to get help on this function.

```
rolls_from_sample = sample(c(1:6), size=5000, replace=TRUE)
rolls_from_sample.int = sample(6, size=5000, replace=TRUE)

table(rolls_from_sample)
```

```
## rolls_from_sample
##   1   2   3   4   5   6
## 868 831 822 835 882 762
```

```
table(rolls_from_sample.int)
```

```
## rolls_from_sample.int
```

```
##   1   2   3   4   5   6
## 848 819 780 868 818 867
```

Both gives a uniform distribution over the numbers 1-6. The function `sample.int` is a specialised version of sample for sampling integers. Many `R` libraries have specialised versions of more general functions to do specific tasks under certain conditions.

5 Markov Chains

We will now look at a **Markov Chain**. We have not covered it during lectures but based on the basic principles we have covered we will be able to use it for simulations.

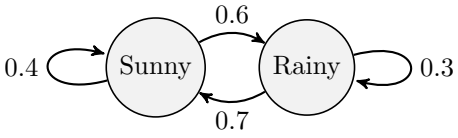
Any random process is known to have the *Markov property* (a Markov process) if the probability of going to the next state depends only on the current state and not on the past states. A Markov process is **memoryless property** in that it does not store any property or memory of its past states.

If a Markov process operates within a specific (finite) set of states, it is called a ***Markov Chain***.

A Markov Chain is defined by three properties:

1. A state space: a set of values or states in which a process could exist
2. A transition matrix: defines the probability of moving from one state to another state
3. A current state probability distribution: defines the probability of being in any one of the states at the start of the process

Consider the following example where we have two states describing the weather on any particular day: (i) Sunny and (ii) Rainy. Each arrow denotes the probability of going from one state to itself or another over the course of a day. For example, if it is currently sunny, the probability of it raining the next day is 0.6. Conversely, if it is raining, the probability that it will become sunny the next day is 0.7 and 0.3 that it will continue raining.



The transition matrix can be written as the following in R:

```
transitionMatrix = matrix(c(0.4, 0.6, 0.7, 0.3), nrow=2, ncol=2, byrow=TRUE)
print(transitionMatrix)
```

```
##      [,1] [,2]
## [1,]  0.4  0.6
## [2,]  0.7  0.3
```

which creates a 2 x 2 matrix consisting of the transition probabilities shown in the diagram.

Suppose I want to simulate a sequence of 30 days and the weather patterns over those days. Assuming that on day 0 it is currently sunny, I can do the following:

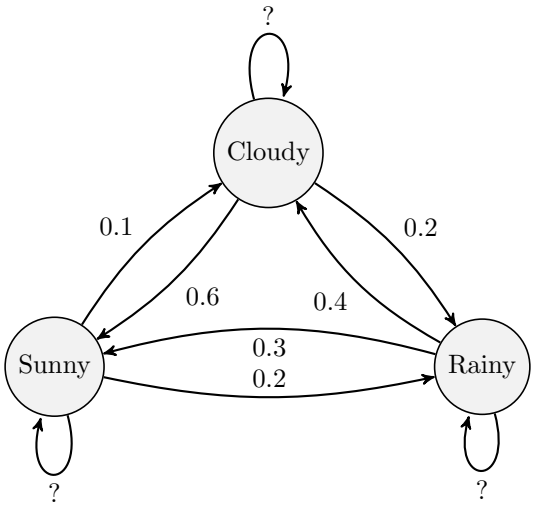
```
# initial state - it is [1] sunny or [2] rainy
state <- 1
weather_sequence <- rep(0, 30) # vector to store simulated values
for (day in 1:30) { # simulate for 30 days
  pr <- transitionMatrix[state, ] # select the row of transition probabilities

  # sample [1] or [2] based on the probs pr
  state = sample(c(1, 2), size = 1, prob = pr)
```

```
  weather_sequence[day] <- state # store the sampled state
}

# print the simulated weather sequence
print(weather_sequence)
```

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

Problem

Can you extend this example to a three-state model?



Note, the diagram (intentionally) misses out the self-transitions. You should be able to infer this because the probabilities given would otherwise not add up to one!

Model answers 3 state Markov Chain

Set up a 3x3 transition matrix:

```
transitionMatrix = matrix(c(0.7, 0.2, 0.1,
                            0.3, 0.3, 0.4,
                            0.6, 0.2, 0.2), nrow=3, ncol=3, byrow=TRUE)

# Check matrix set-up correctly
print(transitionMatrix)
```

```
##      [,1] [,2] [,3]
## [1,]  0.7  0.2  0.1
## [2,]  0.3  0.3  0.4
## [3,]  0.6  0.2  0.2
```

Note the ordering of the states is arbitrary but here we have used the convention that State 1 is Sunny, State 2 is Rainy and State 3 is Cloudy which means the probabilities are completed in that order in the transition matrix. We just need to be consistent.

```
state <- 1 # initial state - it is [1] sunny, [2] rainy and [3] cloudy
weather_sequence <- rep(0, 30) # vector to store simulated values

# simulate for 30 days
for (day in 1:30) {
```

```
  pr <- transitionMatrix[state, ] # select the row of transition probabilities

  # sample [1-3] based on the probs pr
  state <- sample(c(1, 2, 3), size = 1, prob = pr)
  weather_sequence[day] <- state # store the sampled state
}
print(weather_sequence)
```

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

6 A Monopoly simulation

Now you will use to simulate simplified games of Monopoly (https://en.wikipedia.org/wiki/Monopoly_(game)). In addition, there are also many tutorials and guides on the Web describing how to produce computer simulations for Monopoly. You are welcome to read and use these examples to inspire your work.

6.1 Moving around the board

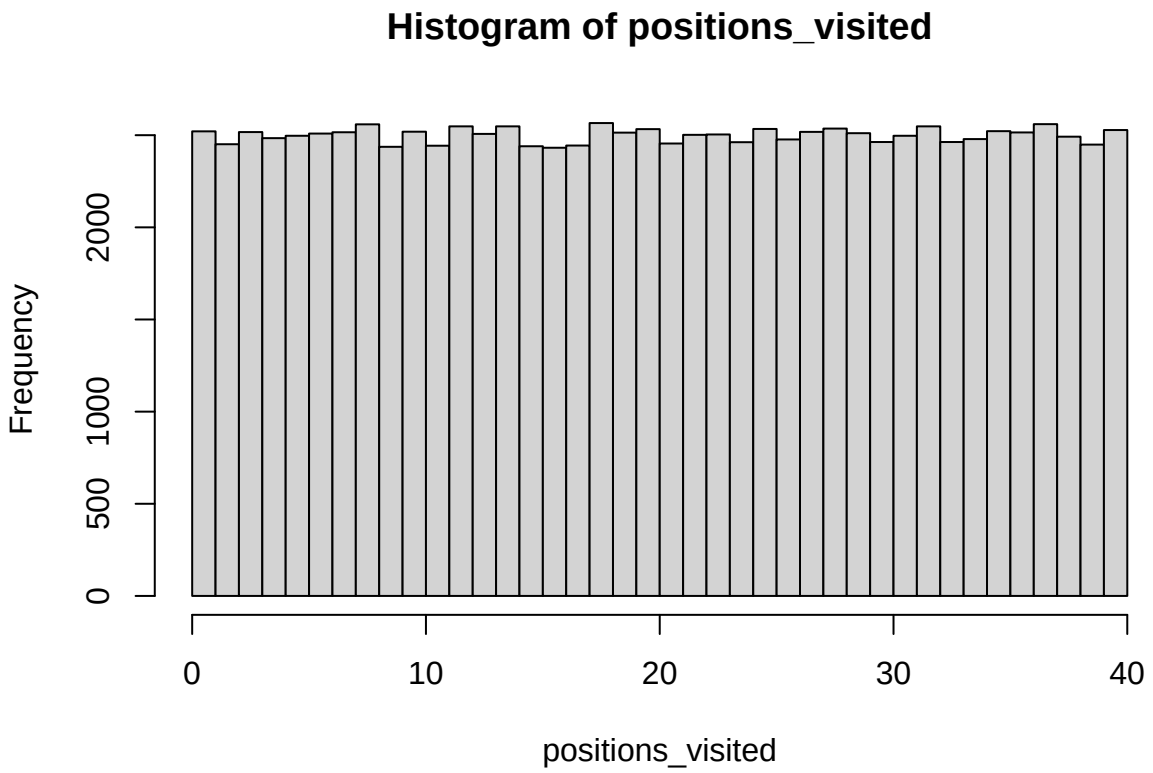
A Monopoly board has 40 spaces. Players take it in turns to roll two dice and traverse around the board according to the sum of the dice values.

Use the following code example to simulate turns of a single player:

```
num_turns <- 100000 # number of turns to take

current_board_position <- 0 # start on the GO space

move_size <- rep(0, num_turns)
positions_visited <- rep(0, num_turns)

# use a for loop to simulate a number of turns
for (turn in 1:num_turns) {

  # roll two dice
  die_values <- sample(c(1:6), 2, replace = TRUE)

  # move player position

  # number of positions to move
  plus_move <- sum(die_values)

  # compute new board position
  new_board_position <- current_board_position + plus_move

  # update board position (this corrects for the fact the board is circular)
  current_board_position <- (new_board_position %% 40)

  # store position visited
  positions_visited[turn] <- current_board_position

}
```

By increasing the number of turns taken, what distribution does the set of simulated board positions converge towards? Show this graphically using the histogram function.

```r
hist(positions_visited, breaks = seq(0, 40, len = 41), right = FALSE)
```



6.2 Going to Jail

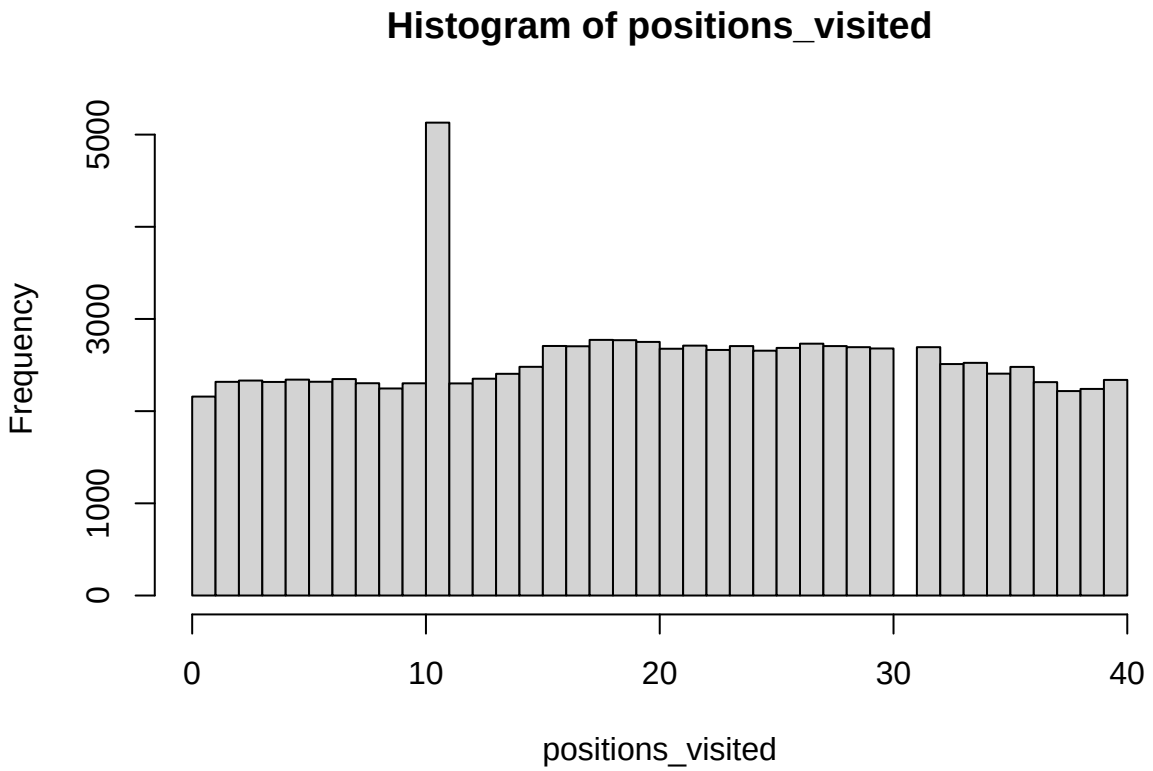
If a player lands on to Go To Jail space they must move immediately to the Jail space. Extend your code to include the possibility of going to jail. Here, assume that once in jail, the player continues as normal on the next turn.

```r
 num_turns <- 100000 # number of turns to take

current_board_position <- 0 # start on the GO space
go_to_jail_position <- 30 # the go to jail space
jail_position <- 10 # jail space

move_size <- rep(0, num_turns)
positions_visited <- rep(0, num_turns)

# use a for loop to simulate a number of turns
for (turn in 1:num_turns) {

  # roll two dice
  die_values <- sample(c(1:6), 2, replace = TRUE)

  # move player position

  # number of positions to move
  plus_move <- sum(die_values)

  # compute new board position
```

```
  new_board_position <- current_board_position + plus_move

  # if land on GO TO JAIL square, then go backwards to the JAIL square
  if (new_board_position == go_to_jail_position) {
    new_board_position <- jail_position
  }

  # update board position (this corrects for the fact the board is circular)
  current_board_position <- (new_board_position %% 40)

  # store position visited
  positions_visited[turn] <- current_board_position

}
```

What is the distribution of board positions during a long game?

```
hist(positions_visited, breaks = seq(0, 40, len = 41), right = FALSE)
```



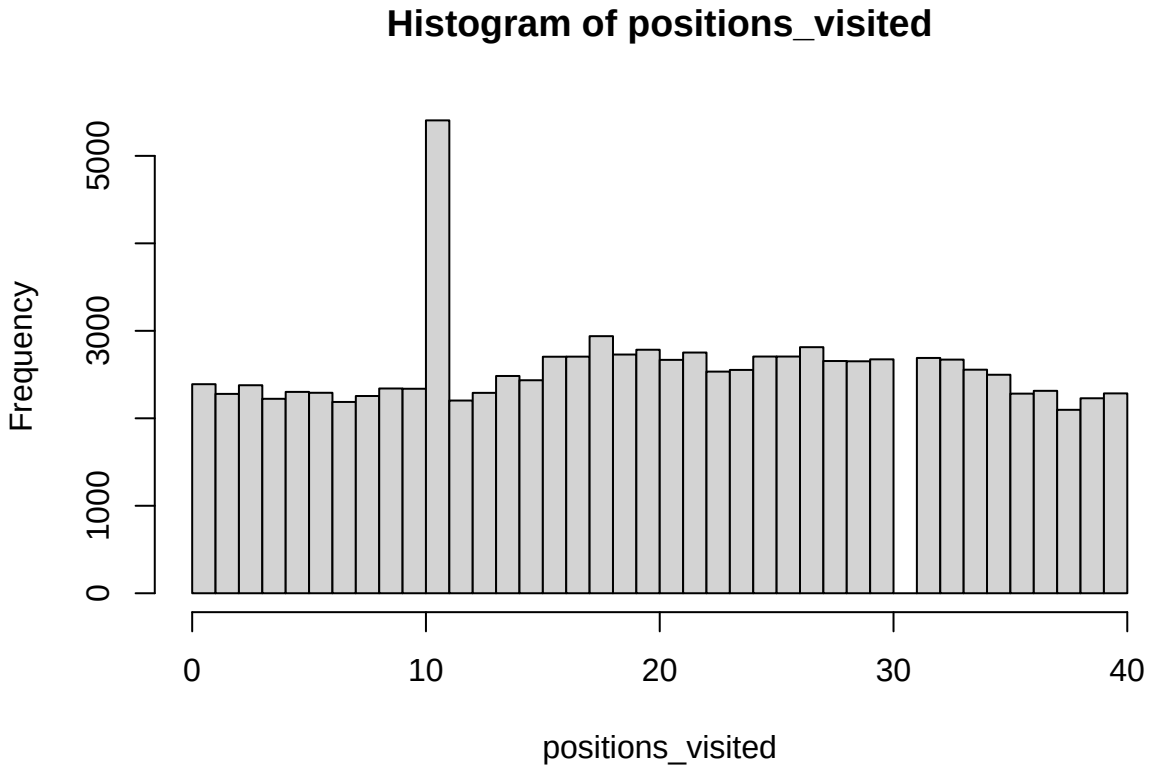
Can you explain this result qualitatively?

You can also go to jail, if you roll three doubles (both dice having the same value) in a row. Update your code to allow for the possibility of going to Jail with three doubles. How does the distribution of board positions change?

```
num_turns <- 100000 # number of turns to take

current_board_position <- 0 # start on the GO space
go_to_jail_position  <- 30 # the go to jail space
jail_position <- 10 # jail space
```

```r
move_size <- rep(0, num_turns)
positions_visited <- rep(0, num_turns)

# use a for loop to simulate a number of turns
for (turn in 1:num_turns) {

  # set double counter to zero
  double_counter <- 0

  # roll (max) three times
  for (j in 1:3){

    # roll two dice
    die_values <- sample(c(1:6), 2, replace = TRUE)

    # if we have rolled a double for the third time, we proceed straight to jail
    if ((die_values[1] == die_values[2]) & (double_counter == 2 )) {
      current_board_position <- jail_position
      break
    }

    # otherwise

    # move player position

    # number of positions to move
    plus_move <- sum(die_values)

    # compute new board position
    new_board_position <- current_board_position + plus_move

    # if land on GO TO JAIL square, then go backwards to the JAIL square
    if (new_board_position == go_to_jail_position) {
      new_board_position <- jail_position
    }

    # update board position (this corrects for the fact the board is circular)
    current_board_position <- (new_board_position %% 40)

    # break out of loop if we roll a non-double
    if (die_values[1] != die_values[2]) {
      break
    } else { # increment double counter
      double_counter <- double_counter + 1
    }

  }

  # store final position visited
  positions_visited[turn] <- current_board_position


}
```

```
hist(positions_visited, breaks = seq(0, 40, len = 41), right = FALSE)
```



Adding the rolling doubles feature doesn't seem to change much. We might expect this since rolling three doubles is a very unlikely event!

6.3 Further Exercises

Now consider building a more complex Monopoly simulation by incorporating more complex aspects of the game such as:

- the purchase of properties
- a ledger for each player
- chance and community cards

You will need to think carefully about the simplifying assumptions you will make to make the task achievable. Do not be over-ambitious. For example, you might initially assume that players will not build houses/hotels on properties.

Here are some questions to answer with your simulations:

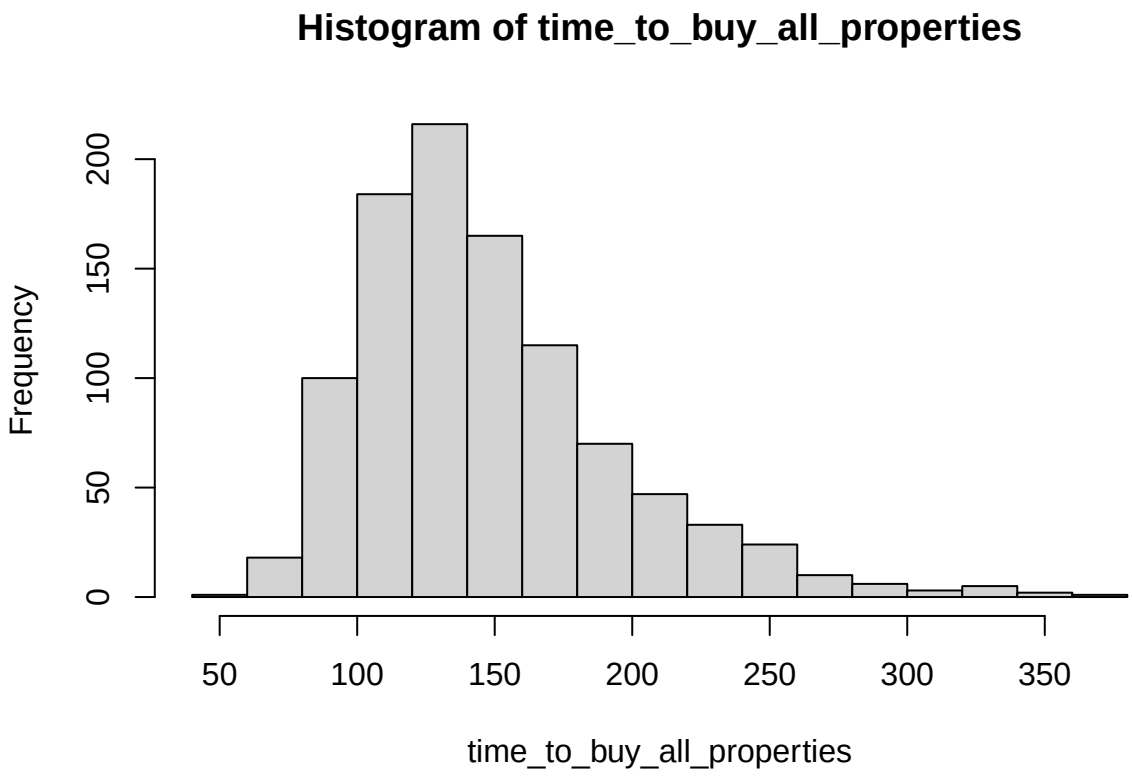
1. How many turns does it take before all properties are purchased?
2. What are the best properties to buy?
3. How long does it take for a winner to be determined?

For example, the following simple extension of the previous example adds some features to record properties being purchased. This simulation is constructed based on the assumption that a players always buys any free property that land on.

```
num_games <- 1000 # number of games to play
num_turns <- 1000 # number of turns to take
```

```
current_board_position <- 0 # start on the GO space
go_to_jail_position <- 30 # the go to jail space
jail_position <- 10 # jail space
# vector of squares containing properties
properties_that_can_be_bought <- c(1, 3, 5, 6, 8, 9, 11, 12, 13, 14, 15, 16,
   18, 19, 21, 23, 24, 25, 26, 27, 28, 29, 31, 32, 34, 35, 37, 39)


# vector to store number of turns to buy all properties
time_to_buy_all_properties <- rep(0, num_games)

# simulate multiple games
for (game in 1:num_games) {

  positions_visited <- rep(0, num_turns)
  positions_purchased <- rep(0, 40)
  properties_bought <- rep(0, num_turns)

  # use a for loop to simulate a number of turns
  for (turn in 1:num_turns) {

    # roll two dice
    die_values <- sample(c(1:6), 2, replace = TRUE)

    # move player position

    # number of positions to move
    plus_move <- sum(die_values)

    # compute new board position
    new_board_position <- current_board_position + plus_move

    # if land on GO TO JAIL square, then go backwards to the JAIL square
    if (new_board_position == go_to_jail_position) {
      new_board_position <- jail_position
    }

    # update board position (this corrects for the fact the board is circular)
    current_board_position <- (new_board_position %% 40)

    # if we can on a square that can be purchased and which has not been
    # purchased (note R uses 1-indexing for arrays)
    if (positions_purchased[current_board_position+1] == 0) {
      if (current_board_position %in% properties_that_can_be_bought) {
        positions_purchased[current_board_position + 1] <- 1
      }
    }

    # store position visited
    positions_visited[turn] <- current_board_position

    # store number of properties bought
    properties_bought[turn] <- sum(positions_purchased)
```

```
    # check if all properties are gone
    if (properties_bought[turn] == length(properties_that_can_be_bought)) {
      time_to_buy_all_properties[game] <- turn
      break
    }


  }

}

hist(time_to_buy_all_properties, breaks = 20)
```



7 Monte Carlo Methods

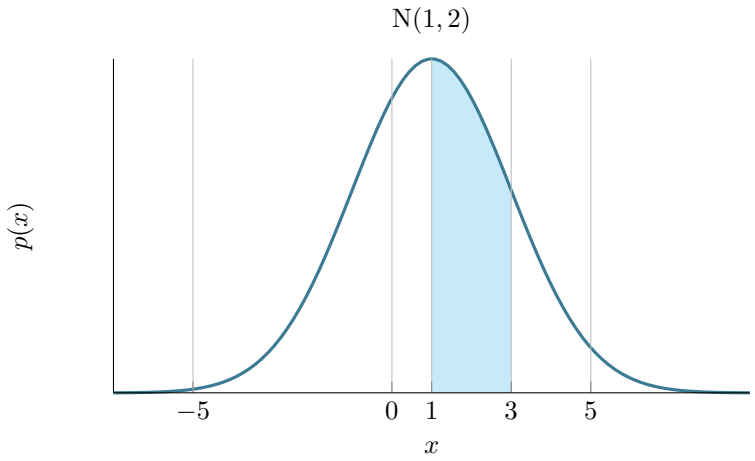
Monte Carlo (MC) simulations provide a means to model a problem and apply brute force computational power to achieve a solution - randomly simulate from a model until you get an answer. The best way to explain is to just run through a bunch of examples, so lets go!

7.1 Integration

We will start with basic integration. Suppose we have an instance of a Normal distribution with a mean of 1 and a standard deviation of 2 then we want to find the *integral* (area under the curve) from 1 to 3:

$$\int_1^3 \frac{1}{10\sqrt{2\,\pi}}\, e^{-\frac{(x-1)^2}{2\times 2^2}}\, dx$$

which we can visualise as follows:



If you have not done calculus before - do not worry. We are going to write a Monte Carlo approach for estimating this integral which does not require any knowledge of calculus!

The method relies on being able to generate samples from this distribution and counting how many values fall between 1 and 3. The proportion of samples that fall in this range over the total number of samples gives the area.

First, create a new `R` script in `Rstudio`. Next we define the number of samples we will obtain. Lets choose 10,000

```
n <- 100 # number of samples to take
```

Now we use the `R` function `rnorm` to simulate 100 numbers from a Normal distribution with mean 1 and standard deviation 2:

```
sims <- rnorm(n, mean = 1, sd = 2) # simulated normally distributed numbers
```

Lets estimate the integral between 1 and 3 by counting how many samples had a value in this range:

```
# find proportion of values between 1-3
mc_integral <- sum(sims >= 1 & sims <= 3) / n
```

The result we get is:

```
print(mc_integral)
```

```
## [1] 0.37
```

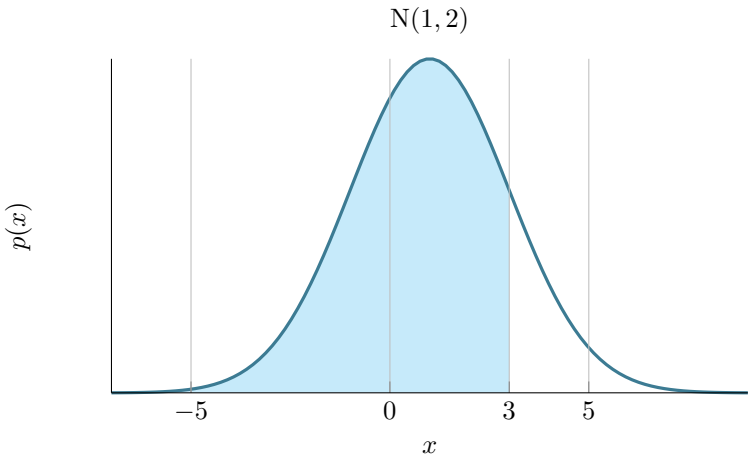
The exact answer given using the cumulative distribution function `pnorm` in R is given by:

```
mc_exact = pnorm(q=3, mean=1, sd=2) - pnorm(q=1, mean=1, sd=2)
print(mc_exact)
```

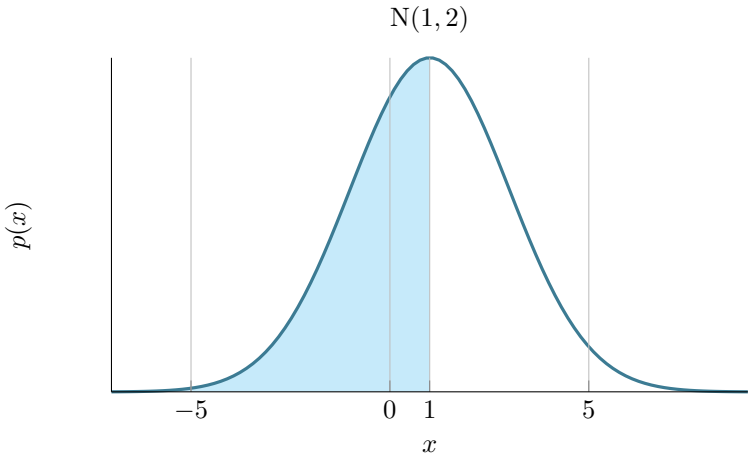
```
## [1] 0.3413447
```

The `pnorm` gives the integral under the Normal distribution (in this case with mean 1 and standard deviation 2) from negative infinity up to the value specified by `q`.

The first call to `pnorm(q=3, mean=1, sd=2)` gives us this integral:



The second call to `pnorm(q=1, mean=1, sd=2)` gives us this integral:



Therefore the difference between these gives us the integral of interest.

The Monte Carlo estimate is a fairly good approximation to the true value!

7.2 Problem: MC accuracy

1. Try increasing the number of simulations and see how the accuracy improves?
2. Can you draw a graph of number of MC samples vs accuracy?

Model answers are in the next section

7.3 Approximating the Binomial Distribution

We flip a coin 10 times and we want to know the probability of getting more than 3 heads. This is a trivial problem using the Binomial distribution but suppose we have forgotten about this or never learned it in the first place.

Lets solve this problem with a Monte Carlo simulation. We will use the common trick of representing tails with 0 and heads with 1, then simulate 10 coin tosses 100 times and see how often that happens.

```
runs <- 100 # number of simulations to run

greater_than_three <- rep(0, runs) # vector to hold outcomes

# run 100 simulations
```

```r
for (i in 1:runs) {

  # flip a coin ten times (0 - tail, 1 - head)
  coin_flips <- sample(c(0, 1), 10, replace = T)

  # count how many heads and check if greater than 3
  greater_than_three[i] <- (sum(coin_flips) > 3)
}

# compute average over simulations
pr_greater_than_three <- sum(greater_than_three) / runs
```

For our MC estimate of the probability $P(X > 3)$ we get

```r
print(pr_greater_than_three)
```

```
## [1] 0.85
```

which we can compare to R's built-in Binomial distribution function:

```r
print(pbinom(3, 10, 0.5, lower.tail = FALSE))
```

```
## [1] 0.828125
```

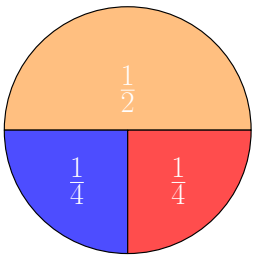
7.4 Problem: MC Binomial

1. Try increasing the number of simulations and see how the accuracy improves?
2. Can you plot how the accuracy varies as a function of the number of simulations? (hint: see the previous section)

Not bad! **The Monte Carlo estimate is close to the true value.**

7.5 Monte Carlo Expectations

Consider the following spinner. If the spinner is spun randomly then it has a probability 0.5 of landing on yellow and 0.25 of landing on red or blue respectively.



If the rules of the game are such that landing on 'yellow' you gain 1 point, 'red' you lose 1 point and 'blue' you gain 2 points. We can easily calculate the expected score.

Let X denote the random variable associated with the score of the spin then:

$$E[X] = \frac{1}{2} \times 1 + \frac{1}{4} \times (-1) + \frac{1}{4} \times 2 = 0.75$$

If we ask a more challenging question such as:

> **After 20 spins what is the probability that you will have less then 0 points?"**

How might we solve this?

Of course, there are methods to analytically solve this type of problem but by the time they are even explained we could have already written our simulation!

To solve this with a Monte Carlo simulation you need to sample from the Spinner 20 times, and return 1 if we are below 0 other wise we'll return 0. We will repeat this 10,000 times to see how often it happens!

7.6 Using Functions

First, we are going to introduce the concept of a function. This is a piece of code which is encapsulated so then we can refer to it repeated via the name of the function rather than repeatedly writing those lines of code. The function we will write will simulate one game as indicated above and return whether the number of points is less than zero.

```
# simulates a game of 20 spins
play_game <- function(){
    # picks a number from the list (1, -1, 2)
    # with probability 50%, 25% and 25% twenty times
  results <- sample(c(1, -1, 2), 20, replace = TRUE, prob = c(0.5, 0.25, 0.25))

  # function returns whether the sum of all the spins is < 1
  return(sum(results) < 0)
}
```

7.7 Simulating from function

Now we can use this function in a loop to play the game 100 times:

```
runs <- 100 # play the game 100 times

less_than_zero <- rep(0, runs) # vector to store outcome of each game
for (it in 1:runs) {
  # play the game by calling the function and store the outcome
  less_than_zero[it] <- play_game()
}
```

We can then compute the probability that, after twenty spins, we will have less than zero points:

```
prob_less_than_zero <- sum(less_than_zero)/runs
print(prob_less_than_zero)
```

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

The probability is very low. This is not surprising since there is only a 25% chance of getting a point deduction on any spin and a 75% chance of gaining points. Try to increase the number of simulation runs to see if you can detect any games where you do find a negative score.

7.8 Problem: MC Expectation

1. Modify your code to allow you to calculate the expected number of points after 20 spins.
2. Simulate a game in which you have a maximum of 20 spins but you go "bust" once you hit a negative score and take this into account when you compute the expected end of game score.

Model Answers: Monte Carlo

7.9 Problem: MC accuracy

First let's increase the number of simulations and out the accuracy

```
sample_sizes <- c(10, 50, 100, 250, 500, 1000) # try different sample sizes
n_sample_sizes <- length(sample_sizes) # number of sample sizes to try
rpts <- 100 # number of repeats for each sample size
accuracy <- rep(0, n_sample_sizes) # vector to record accuracy values
accuracy_sd <- rep(0, n_sample_sizes) # vector to record accuracy sd values

# for each sample size
for (i in 1:n_sample_sizes) {

  sample_sz <- sample_sizes[i] # select a sanmple size to use

  # vector to store results from each repeat
  mc_integral <- rep(0, rpts)
  for (j in 1:rpts){
    # simulated normally distributed numbers
    sims <- rnorm(sample_sz, mean = 1, sd = 2)
    # find proportion of values between 1-3
    mc_integral[j] <- sum(sims >= 1 & sims <= 3) / sample_sz
  }

  # compute average difference between integral estimate and real value
  accuracy[i] <- mean(mc_integral - mc_exact)
  # compute sd difference between integral estimate and real value
  accuracy_sd[i] <- sd(mc_integral - mc_exact)

}

print(accuracy)
```

```
## [1]  0.0046552539 -0.0023447461 -0.0067447461 -0.0042247461 -0.0012647461 -0.0007547461
```

```
print(accuracy_sd)
```

```
## [1] 0.14100648 0.05702206 0.04744843 0.02900132 0.02077084 0.01476420
```

```
print(accuracy + accuracy_sd)
```

```
## [1] 0.14566174 0.05467731 0.04070368 0.02477657 0.01950610 0.01400946
```

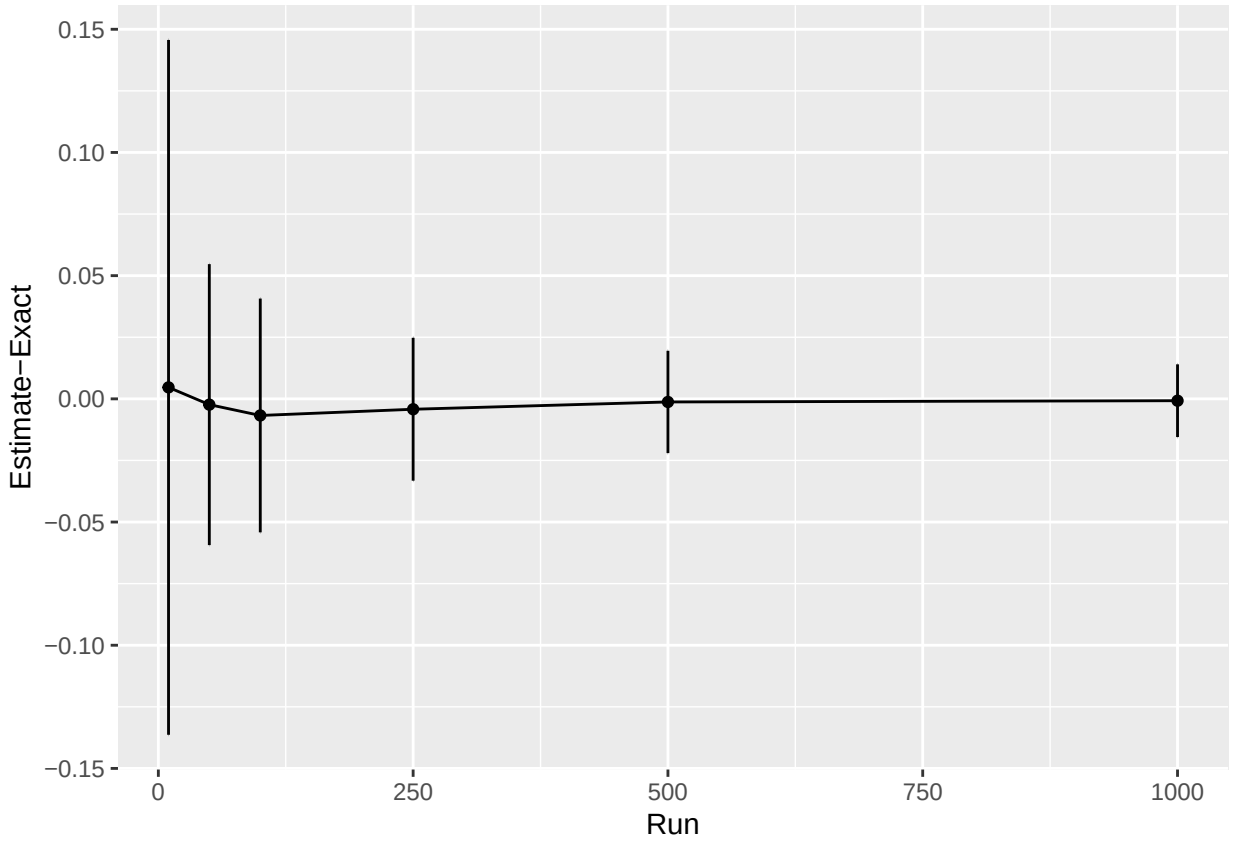
Next, we will plot the results. Here we will make use of `ggplot2` a library to create nice plots without much effort. The input need to be a `data.frame` so we will need to create one based on the data.

```
# load ggplot
library(ggplot2)

# create a data frame for plotting
df <- data.frame(sample_sizes, accuracy, accuracy_sd)

print(df)
```

```
##   sample_sizes      accuracy accuracy_sd
## 1           10  0.0046552539  0.14100648
```

```
## 2           50 -0.0023447461  0.05702206
## 3          100 -0.0067447461  0.04744843
## 4          250 -0.0042247461  0.02900132
## 5          500 -0.0012647461  0.02077084
## 6         1000 -0.0007547461  0.01476420
```

```
# use ggplot to plot lines for the mean accuracy and error bars
# using the std dev
ggplot(df, aes(x = sample_sizes, y = accuracy)) +
  geom_line() +
  geom_point() +
  geom_errorbar(
      aes(ymin = accuracy - accuracy_sd, ymax = accuracy + accuracy_sd),
          width = .2,
          position = position_dodge(0.05)) +
  ylab("Estimate-Exact") +
  xlab("Run")
```



This shows that as the number of Monte Carlo samples is increased, the accuracy increases (i.e. the difference between the estimated integral value and real values converges to zero). In addition, the variability in the integral estimates across different simulation runs reduces.

7.10 Problem: MC Expectation

7.10.1 Problem 1

```
# simulates a game of 20 spins
play_game <- function() {
```

```
    # picks a number from the list (1, -1, 2)
    #  with probability 50%, 25% and 25% twenty times
  results <- sample(c(1, -1, 2), 20, replace = TRUE, prob = c(0.5, 0.25, 0.25))
  return(sum(results)) # function returns the sum of all the spins
}

score_per_game = rep(0, runs) # vector to store outcome of each game
for (it in 1:runs) {
  score_per_game[it] <- play_game() # play the game by calling the function
}
expected_score = mean(score_per_game) # average over all simulations

print(expected_score)
```

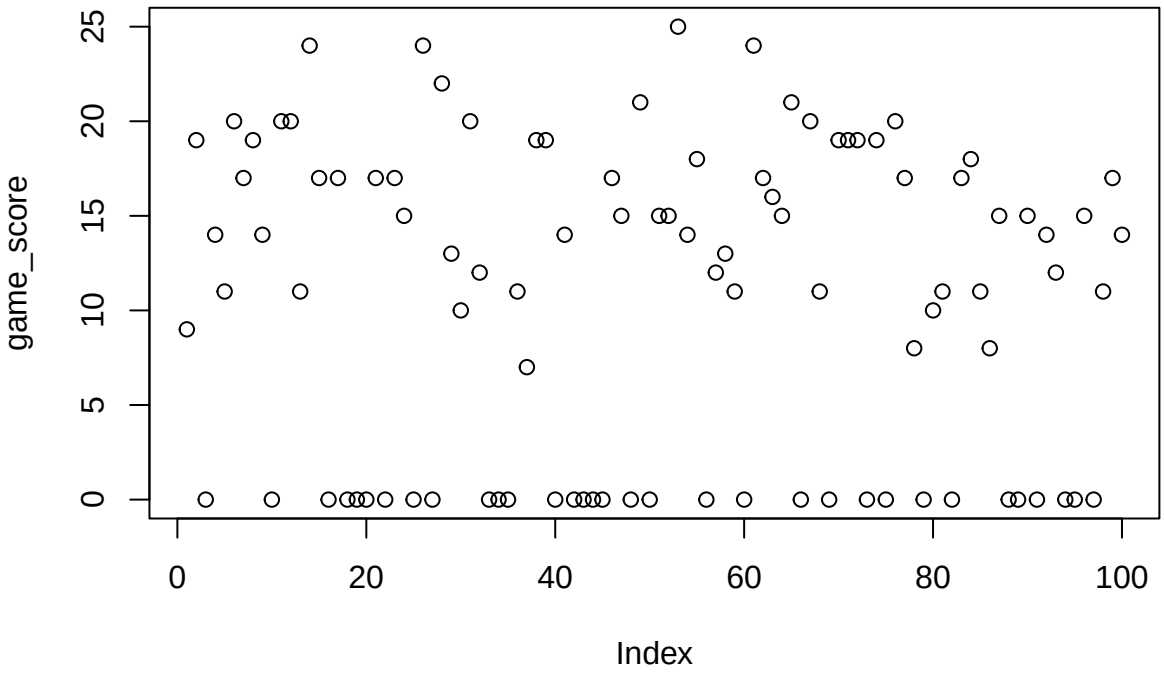
```
## [1] 15.37
```

7.10.2 Problem 2

```
# simulates a game of up to 20 spins
play_game <- function() {
    # picks a number from the list (1, -1, 2)
    #  with probability 50%, 25% and 25% twenty times
  results <- sample(c(1, -1, 2), 20, replace = TRUE, prob = c(0.5, 0.25, 0.25))
  results_sum <- cumsum(results) # compute a running sum of points
  # check if the game goes to zero at any point
  if (sum(results_sum <= 0)) {
    return(0) # return zero
  } else {
    return(results_sum[20]) # returns the final score
  }
}

game_score <- rep(0, runs) # vector to store scores in each game played

# for each game
for (it in 1:runs) {
  game_score[it] <- play_game()
}

print(mean(game_score))
```

```
## [1] 10.61
```

```
plot(game_score)
```



The games with score zero now corresponds to the number of games where we went bust (or genuinely ended the game with zero).

8 Maximum Likelihood

During the lectures, you saw how we could use a brute-force search of parameters to find the maximum likelihood estimate of an unknown mean for a Normal distribution given a set of data. In this exercise, we will now look at how we would do this more efficiently in real life.

8.1 The likelihood function

First, we are going to write a function to compute the log-likelihood function given parameters:

```
neglogLikelihood <- function(mu, x) {
  logF = dnorm(x, mean = mu, sd = 1, log = TRUE)
  return(-sum(logF))
}
```

Note that this function returns the `-sum(logF)` because the numerical optimisation algorithm we are going to use finds the *minimum* of a function. We are interested in the *maximum* likelihood but we can turn this into a minimisation problem by simply negating the likelihood.

Now, lets assume our data is captured in the following vector:

```
x = c(-0.5, 1.0, 0.2, -0.3, 0.5, 0.89, -0.11, -0.71, 1.0, -1.3, 0.84)
n = length(x)
```

8.2 Optimisation

Now, we will need to define an initial search value for the parameter, we will arbitrarily pick a value:

```
mu_init = 1.0
```

Now we will use the `R` function `optim` to find the maximum likelihood estimate. As mentioned above, `optim` finds the minimum value of a function so in this case we are trying to find the parameter that minimises the

negative log likelihood.

```
out <- optim(mu_init, neglogLikelihood, gr = NULL, x, method = "L-BFGS-B",
         lower = -Inf, upper = Inf)
```

Here, this says that we will start the search at `mu_init` using the function `logLikelihood` that we have defined above. The `optim` algorithm will use the `L-BFGS-B` search method. The parameter is allowed to take any value from `lower = -Inf` to `upper = Inf`. The result is stored in out.

Once the optimiser has run, we can see what parameter value it has found:

```
print(out$par)
```

```
## [1] 0.1372727
```

which we can compare against the sample mean

```
print(mean(x))
```

```
## [1] 0.1372727
```

It turns out that it is theoretically known that the maximum likelihood estimate, for this particular problem, is the sample mean which is why they coincide!

We can visualise this further. First we define an array of possible values for `mu` in this case between -0.1 and 0.3 with 101 values in-between:

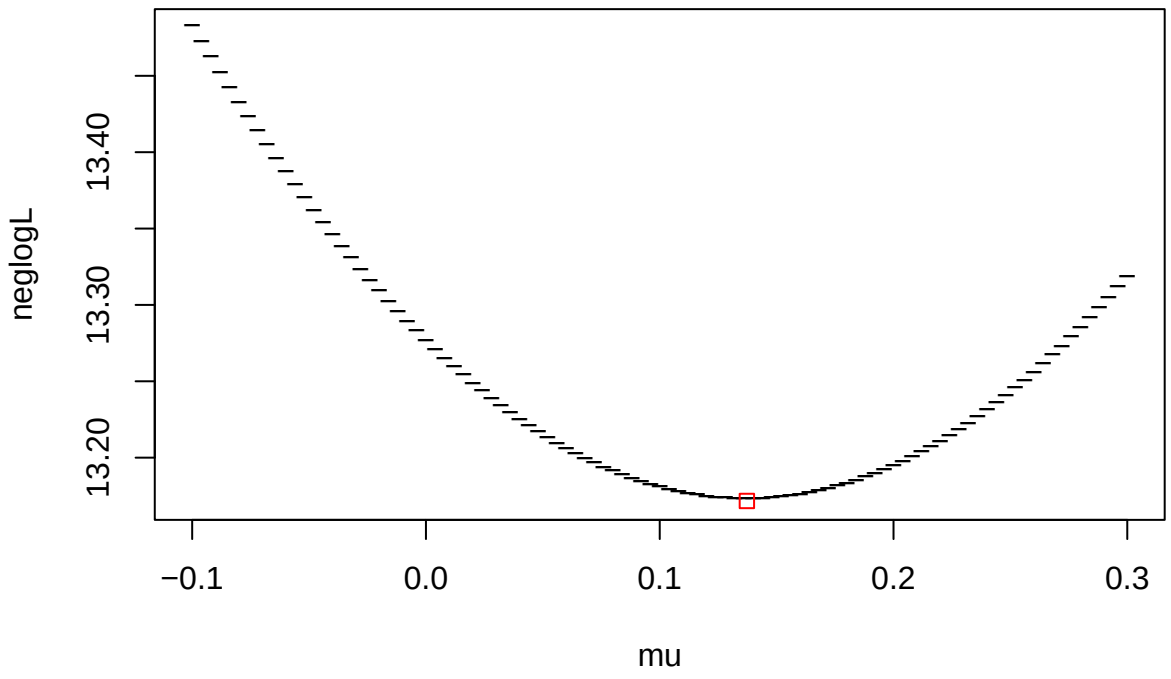
```
mu <- seq(-0.1, 0.3, length = 101)
```

We use the `apply` function to apply the `logLikelihood` function to each of the `mu` values we have defined. This means we do not need to use a for loop:

```
neglogL <- apply( matrix(mu), 1, neglogLikelihood, x)
```

We can then plot and overlay the maximum likelihood result:

```
plot(mu, neglogL, pch="-")
points(out$par, out$value, col="red", pch=0)
```



The plot shows that `optim` has found the `mu` which minimises the negative log-likelihood.

8.3 Two-parameter estimation

Now suppose both the mean and the variance of the Normal distribution are unknown and we need to search over two parameters for the maximum likelihood estimation.

We now need a modified negative log-likelihood function:

```
neglogLikelihood2 <- function(theta,x) {
  mu <- theta[1] # get value for mu
  sigma2 <- theta[2] # get value for sigma2

  # compute density for each data element in x
  logF <- dnorm(x, mean = mu, sd = sqrt(sigma2), log = TRUE)

  return(-sum(logF)) # return negative log-likelihood
}
```

Notice that we pass through one argument `theta` whose elements are the parameters for `mu` and `sigma2` which we unpack within the function.

Now we can run `optim` but this time the initial parameters values must be initialised with two values. Furthermore, as variance cannot be negative, we bound the possible lower values that `sigma2` can take by setting `lower = c(-Inf, 0.001)`. The second argument means `sigma2` cannot be lower than 0.001:

```
theta_init = c(1, 1)

out <- optim(theta_init, neglogLikelihood2, gr = NULL, x, method = "L-BFGS-B",
        lower = c(-Inf, 0.001), upper = c(Inf, Inf))
```

We can now visualise the results by creating a two-dimensional contour plot. We first need to generate a grid of values for `mu` and `sigma2`:

```
# one dimensional grid of values for mu
mu <- seq(-0.1, 1.0, length = 101)
# one dimensional grid of values for sigma2
sigma2 <- seq(0.1, 1.0, length = 101)

mu_xx <- rep(mu, each = 101) # replicate this 101 times
sigma2_yy <- rep(sigma2, times = 101) # replicate this 101 times

# generate grid of values (each row contains a unique combination
# of mu and sigma2 values)
mu_sigma_grid <- cbind(mu_xx, sigma2_yy)
```

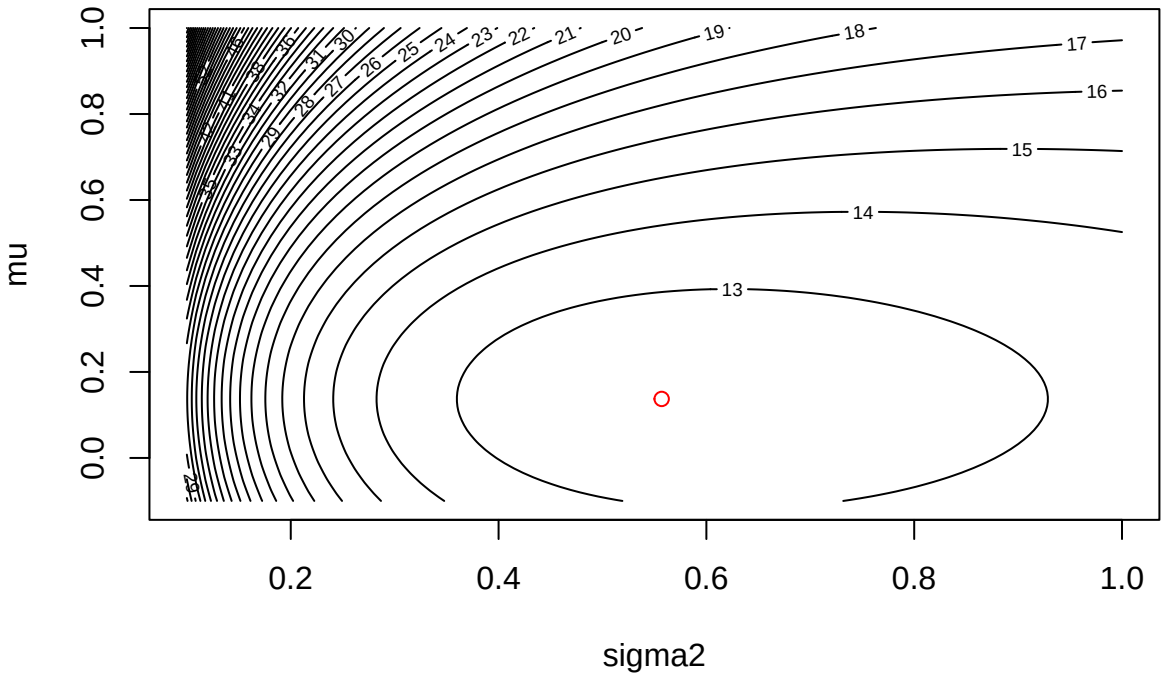
Now we apply our negative log-likehood function to this grid to generate a negative log-likelihood value for each position on the grid:

```
neglogL2 <- apply(mu_sigma_grid, 1, neglogLikelihood2, x)
```

We now use the `contour` function to plot our results:

```
# convert vector of negative log-likelihood values into a grid
neglogL2 <- matrix(neglogL2, 101)

# draw contour plot
contour(sigma2, mu, neglogL2, nlevels = 50, xlab = "sigma2", ylab = "mu")
```

```
# overlay the maximum likelihood estimate as a red circle
points(out$par[2], out$par[1], col="red")
```



Excellent! We have now found the maximum likelihood estimates for the unknown mean and variance for the Normal distribution that our data is assumed to be drawn from. Let's compare our estimates against the sample mean and variance. First, the estimates:

```
print(out$par[1]) # mu estimate
```

```
## [1] 0.1372727
```

```
print(out$par[2]) # sigma2 estimate
```

```
## [1] 0.5569665
```

Now, the sample mean and variances:

```
print(mean(x)) # sample mean
```

```
## [1] 0.1372727
```

```
print(var(x)) # sample variance (normalised by n-1)
```

```
## [1] 0.6126618
```

```
print(var(x)*(n-1)/n) # sample variance (normalised by n)
```

```
## [1] 0.5569653
```

Interesting! The maximum likelihood estimates return the sample mean and the **biased* sample variance estimate (where we normalise by n and not $n-1$). Indeed, it turns out that theoretically, the maximum likelihood estimate does give a biased estimate of the population variance.

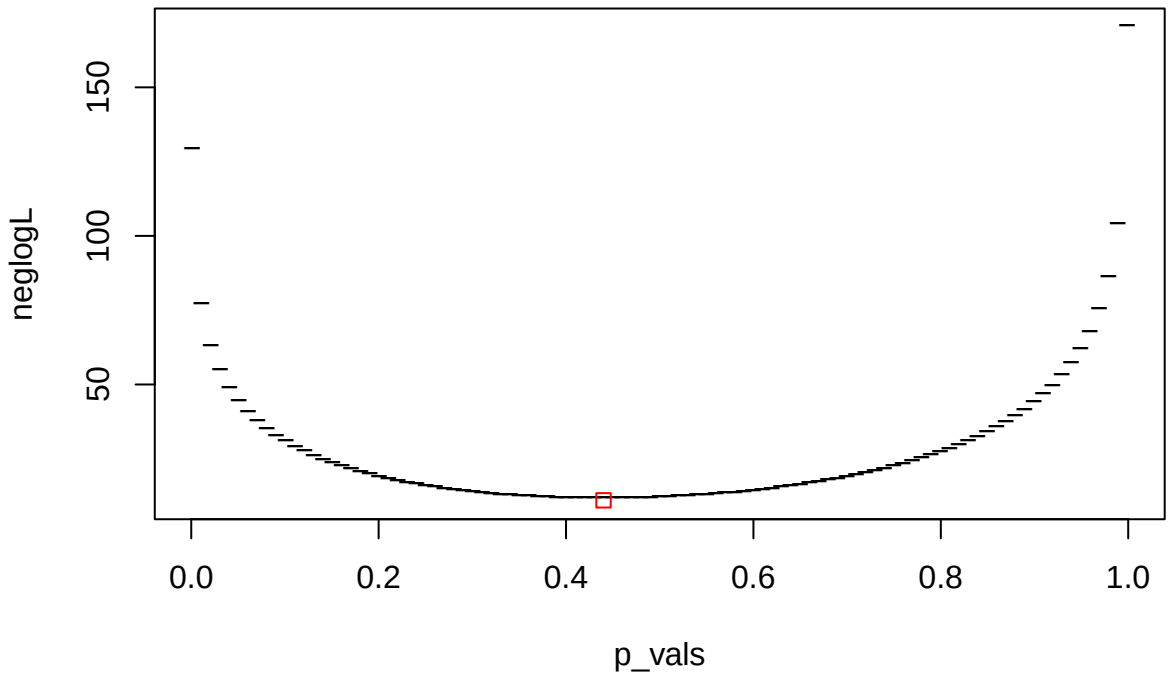
8.4 Problem: MLE

A potentially biased coin is tossed 10 times and the number of heads recorded. The experiment is repeated 5 times and the number of heads recorded was 3, 2, 4, 5 and 2 respectively. Can you

derive a maximum likelihood estimate of the probability of obtaining a head?

Model Answers: MLE

```r
neglogLikelihood <- function(p, n, x) {
  # compute density for each data element in x
  logF <- dbinom(x, n, prob = c(p, 1 - p), log = TRUE)
  return(-sum(logF)) # return negative log-likelihood
}

n <- 10 # number of coin tosses
x <- c(3, 2, 4, 5, 2) # number of heads observed

p_init <- 0.5 # initial value of the probability

# run optim to get maximum likelihood estimates
out <- optim(p_init, neglogLikelihood, gr = NULL, n, x, method = "L-BFGS-B",
  lower = 0.001, upper = 1-0.001)

# create a grid of probability values
p_vals <- seq(0.001, 1 - 0.001, length = 101)

# use apply to compute the negative log-likelihood for each probability value
neglogL <- apply(matrix(p_vals), 1, neglogLikelihood, n, x)

# plot negative log-likelihood function and overlay maximum (negative)
# log-likelihood estimate
plot(p_vals, neglogL, pch = "-")
points(out$par, out$value, col = "red", pch = 0)
```



9 Confidence Intervals

In this practical, you will learn how to derive confidence intervals for a particular problem using Monte Carlo simulations.

In a random experiment, a sample of data is collected from which we can estimate a population parameter of interest. This estimate can either be a point estimate or an *interval estimate* - a range of values.

A **confidence interval** is an interval estimate which has an associated *confidence level*. The confidence level tells us the probability that the *procedure* that is used to construct the confidence interval will result in the interval containing the true population parameter. It is *not* the probability that the population parameter lies in the range.

This is a very counter-intuitive concept which we shall now illustrate in this exercise.

9.1 Setup

First, create a new `R` script within Rstudio then we will start with some code preamble. We will use the package `ggplot2` for plotting.

```
library(ggplot2)
```

Use `install.packages("ggplot2")` in the console window if the package is not installed on your system.

Lets define the width of an interval, we will set this to 1 initially but we will change this later on:

```
interval_width = 1 # width of confidence interval
```

9.2 Simulating data

We are now going to generate some simulated data for our experiment. We will create 30 samples from a Normal distribution with mean 2.5 and variance 1. These are *true* values of the population parameters. In a real experiment, we would not know these values but using simulated data, we obviously control these.

Let define these first:

```
# number of data points to generate
n <- 30
# population mean
mu <- 2.5
# population standard deviation (square root of population variance)
sigma <- 1.0
```

Now, we generate some normally distributed data using the `R` function `rnorm`,

```
# generate n values from the  Normal distribution N(mu, sigma)
x <- rnorm(n, mean = mu, sd = sigma)
```

We now have 30 samples from a Normal distribution with population mean 2.5 and variance 1.

9.3 Constructing the confidence interval

We are going to pretend that we do not know the population mean value (2.5) used to generate this dataset and try to provide an interval estimate for it from the simulated sample data.

Remember, from lectures, that the sample mean $\bar{x}$ is a natural point estimate for the population mean μ.

```
x_bar = mean(x) # compute sample mean
```

so a suitable interval might be centred on the sample mean and extend out,

```
interval <- c(x_bar - interval_width / 2, x_bar + interval_width / 2)
```

Let's look at this interval:

```
print(interval)
```

```
## [1] 1.896618 2.896618
```

Q: Does the confidence interval contain the true parameter?

9.4 Experiment

The previous experiment only examined one simulated dataset so we cannot fully understand the probabilistic interpretation of the confidence interval just yet. At the moment, the interval you have calculated will either contain the population mean or not.

In order to understand the probabilistic interpretation, we will need to generate many data sets, construct confidence intervals as we have for each and then see across all generated data sets, how often those intervals cover the true population mean.

For a Monte Carlo simulation, we will need many repeats of the simulation. Lets define the number of repeats to be used:

```
nreps <- 1000 # number of Monte Carlo simulation runs
```

We will use 1000 simulations initially to make the code quick to run but you may want to make this higher later on for greater accuracy.

Now, let us define a series of interval widths to simultaneously test,

```
# define a series of interval widths
interval_width <- seq(0.1, 1.0, 0.1)
# store the number of interval widths generated
n_interval_widths <- length(interval_width)
```

This creates a sequence of values from 0.1 to 1.0 in steps of 0.1 in the vector `interval_width`:

```
print(interval_width)
```

```
##  [1] 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
```

Now, we will create a vector of zeros of the same length. We will use this to store the number of times that a confidence interval of those specific widths contain the true population mean

```
# create a vector to store the number of times the population mean is contained
mu_contained <- rep(0, n_interval_widths)
```

The hard work now begins. We use a `for` loop to repeat the simulation `nreps` times. Within each loop, we will simulate a new data set, compute a sample mean and then check if the confidence interval contains the true population mean. Since we are using more than one confidence width, we use a second `for` loop to cycle through the different widths.

```
for (replicate in 1:nreps) {

  x <- sigma * rnorm(n) + mu # simulate a data set

  xbar <- mean(x) # compute the sample mean

  # for each interval width that we are testing ...
  for (j in 1:n_interval_widths) {
    # check if the interval contains the true mean
```

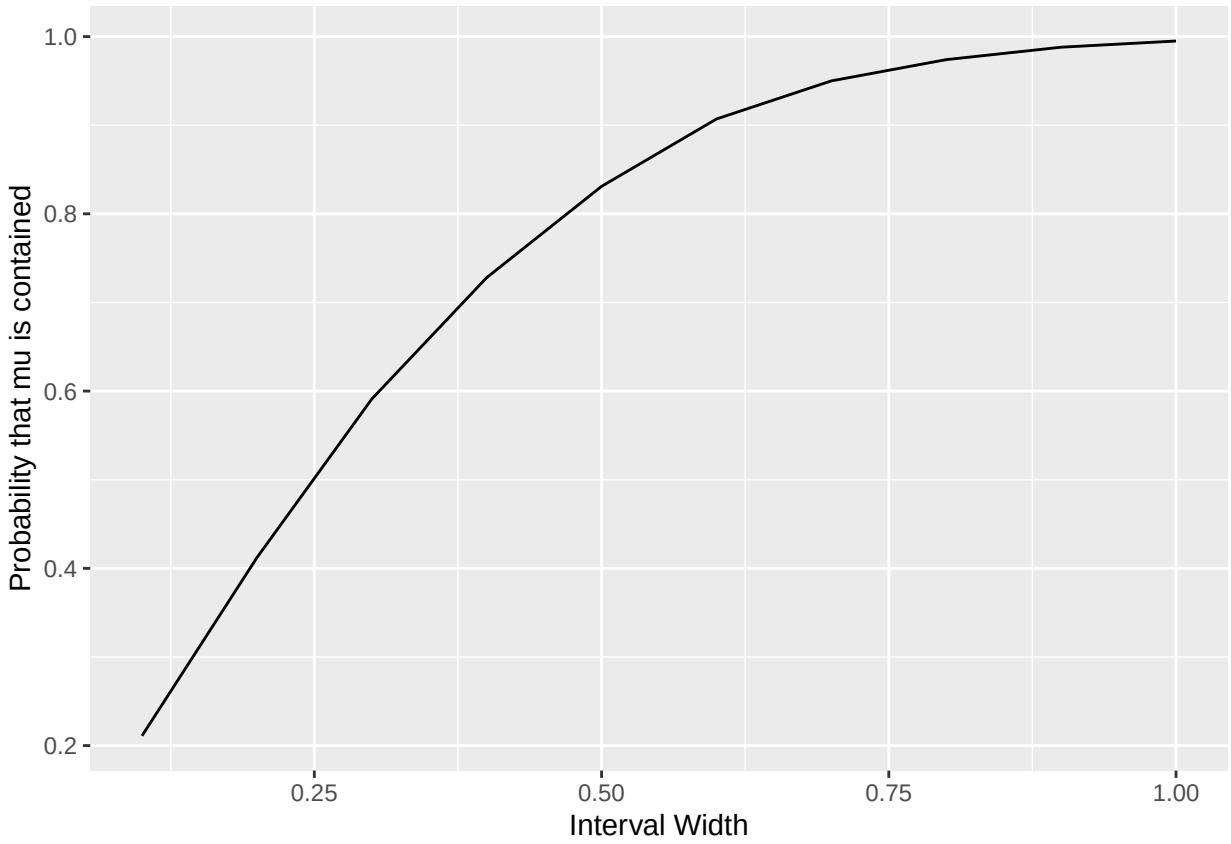
```
    if ((mu > xbar - 0.5 * interval_width[j]) &
        (mu < xbar + 0.5 * interval_width[j])) {
      # if it is, we increment the count by one for this width
      mu_contained[j] <- mu_contained[j] + 1
    }
  }

}
```

We can now calculate, for each width, an estimate of the probability that a confidence interval of that width will contain the population mean.

```
probability_mean_contained <- mu_contained / nreps
```

Let's use `ggplot2` to plot this relationship,

```
# create a data frame containing the variables we wish to plot
df <- data.frame(interval_width = interval_width,
                 probability_mean_contained = probability_mean_contained)

# initialise the ggplot
plt <- ggplot(df, aes(x = interval_width, y = probability_mean_contained))
# create a line plot
plt <- plt + geom_line()
# add a horizontal axis label
plt <- plt + xlab("Interval Width")
# create a vertical axis label
plt <- plt + ylab("Probability that mu is contained")

print(plt) # plot to screen
```



Can you see that an interval width of $0.6 \times (\bar{x} \pm 0.3)$ gives a confidence interval close to 90% probability of containing the population mean?

Remember from the lectures that we saw that the theory says $\bar{x} \pm 1.65 \frac{\sigma}{\sqrt{n}}$ gives a 90% confidence interval?

So, if we compute $2 \pm 1.65 \frac{\sigma}{\sqrt{n}}$, what do we get?

```
print(2 * 1.65 * sigma / sqrt(n))
```

```
## [1] 0.6024948
```

The Monte Carlo estimate matches up with the theory!

9.5 Problem: Confidence Interval

Can you devise a way to compute a confidence interval for the population variance?

You can make use of the following as a point estimate of the sample variance:

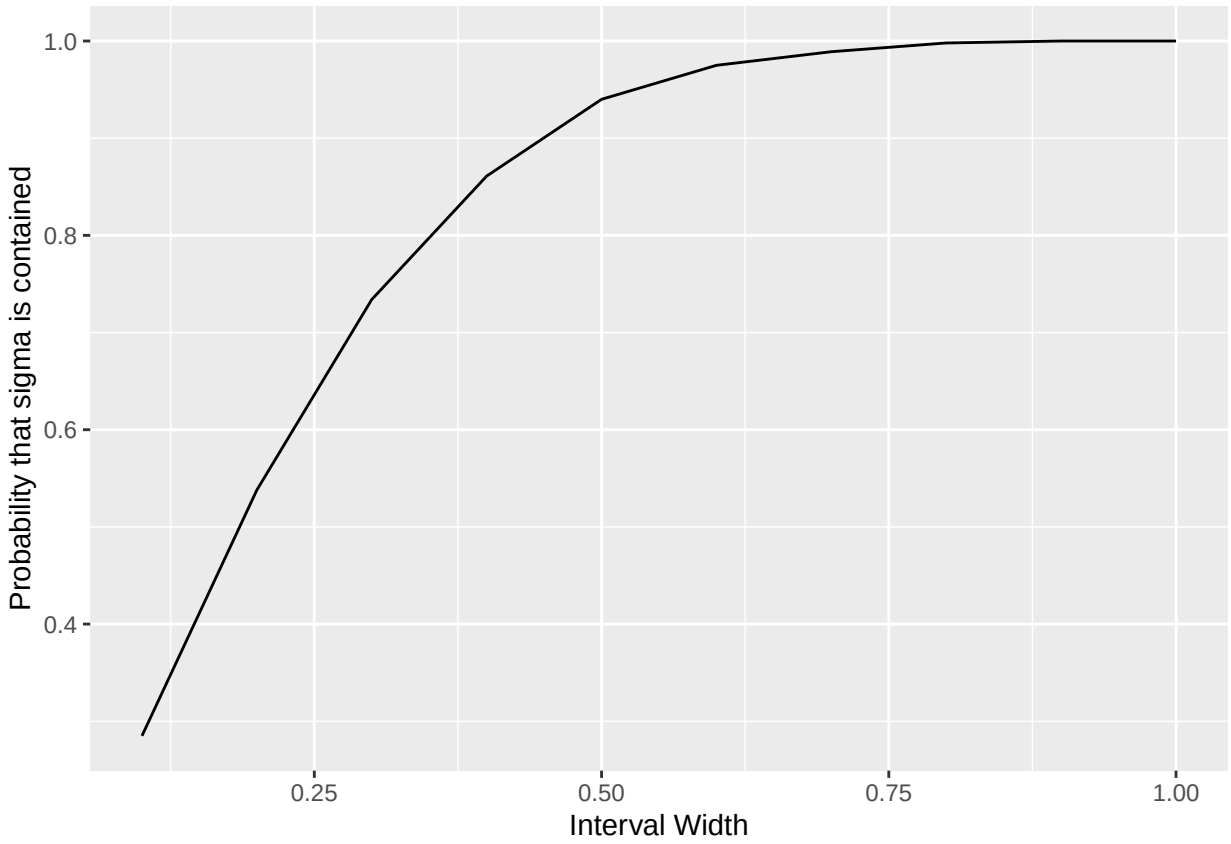
$$s^2 = \frac{1}{n-1} \sum_{i=1}^{n} (x - \bar{x})^2$$

which can be calculated using the `sd` function in `R`.

Model Answer: Confidence Interval

```
# create a vector to store the number of times
# the population variance is contained
```

```
sigma_contained <- rep(0, n_interval_widths)

for (replicate in 1:nreps) {

  x <- rnorm(n, mean = mu, sd = sigma) + mu # simulate a data set

  sigmabar <- sd(x) # compute the sample standard deviation

  # for each interval width that we are testing ...
  for (j in 1:n_interval_widths) {
    # check if the interval contains the true mean
    if ((sigma > sigmabar - 0.5 * interval_width[j]) &
        (sigma < sigmabar + 0.5 * interval_width[j])) {

      # if it is, we increment the count by one for this width
      sigma_contained[j] <- sigma_contained[j] + 1
    }
  }
}

probability_var_contained <- sigma_contained / nreps

# create a data frame containing the variables we wish to plot
df <- data.frame(interval_width = interval_width,
        probability_var_contained = probability_var_contained)

# initialise the ggplot
plt <- ggplot(df, aes(x = interval_width, y = probability_var_contained))
# create a line plot
plt <- plt + geom_line()
# add a horizontal axis label
plt <- plt + xlab("Interval Width")
# create a vertical axis label
plt <- plt + ylab("Probability that sigma is contained")

# plot to screen
print(plt)
```




```
print(df)
```

```
##    interval_width probability_var_contained
## 1             0.1                     0.285
## 2             0.2                     0.538
## 3             0.3                     0.734
## 4             0.4                     0.861
## 5             0.5                     0.940
## 6             0.6                     0.975
## 7             0.7                     0.989
## 8             0.8                     0.998
## 9             0.9                     1.000
## 10            1.0                     1.000
```

10 Computational Testing Techniques

In this practical we will look at various hypothesis testing problems that explores how we can perform hypothesis testing in R. Please make an attempt before looking at the models answers.

10.1 Problem 1

A process for filling milk cartons is claimed to fill each carton with an average of 260g.

The population fill weight is known to be normal, with a standard deviation of 1.65g.

A random sample of five cartons was collected, and the content weighed, yielding the following (in g.)

`263.9, 266.2, 266.3, 266.8, 265.0`

Construct a suitable hypothesis test, at the 1% significance level, to determine whether cartons are being over-filled.

10.2 Problem 2

The mean length of a certain type of component is assumed to be 100mm. Concerns are raised that the mean length is not 100mm.

A random sample of size 45 was obtained, yielding $\bar{x} = 103.11$ and $s = 53.5$.

Perform a hypothesis test, at the 5% level, to determine whether these concerns are justified.

10.3 Problem 3

Can you write your own `z_test` function to perform one-sided or two-sided location (z) tests?

10.4 Problem 4

A manufacturing process yields a product that has a quality control specification of $\mu_0 = 5.4$.

A random sample of size $n = 5$ had a sample mean of 5.64 and sample variance of 0.05.

Conduct a hypothesis test, at the 5 significance level, to assess whether the process is meeting specification.

10.5 Problem 5

Two methods of filling standard gas cylinders are claimed to be different.

In particular, process A is claimed to yield a higher pressure that process B.

> A random sample of 72 cylinders were filled using process A, yielding $\bar{x}_A = 88$ and $s_A^2 = 4.5$.
>
> A random sample of 48 cylinders were filled with process B, yielding $\bar{x}_B = 79$ and $s_B^2 = 4.2$.

Conduct a hypothesis test, at the 5% significance level, to investigate this claim. ## Problem 6

Two catalysts are available for a chemical process. Catalyst B is cheaper than catalyst A.

Provided catalyst B produces the same mean yields, it should be preferred.

To compare methods an experiment was conducted yielding the following data:

> A: 91.50 94.18 92.18 95.39 91.79 89.07 94.72 89.21
>
> B: 89.19 90.95 90.46 93.21 97.19 97.04 91.07 92.75

Is there evidence to say the two catalysts produce different mean yields? Test at the 5% significance level

10.6 Problem 7

In an animal behaviour experiment a group of 90 rats proceed down a ramp to one of three doors.

The observed counts for each door were:

> Door: 1, 2, 3
>
> Rats: 23, 36, 31

Is there evidence to suggest a preference for a specific door? Test at the 5% significance level.

10.7 Problem 8

The number of accidents at a junction per week, Y, was observed over a 50 week period, yielding:

y: $0, 1, 2, \geq 3$

O: $32, 12, 6, 0$

Test the hypothesis, at the 1% level, that $Y \sim \text{Poisson}(\lambda)$

Model Answers: Computational Testing

10.8 Problem 1

The data is:

```
x <- c(263.9, 266.2, 266.3, 266.8, 265.0)
```

Now construct some summary statistics and define some given parameters:

```
x_bar <- mean(x) # compute sample mean
sigma <- 1.65 # population standard deviation is given
mu <- 260 # population mean to be tested against
n <- length(x) # number of samples
```

Construct the z-statistic:

```
z <- (x_bar - mu) / (sigma / sqrt(n))
print(z)
```

```
## [1] 7.643287
```

Check if the z-statistic is in the critical range. First, work out what the z-value at the edge of the critical region is:

```
z_threshold <- qnorm(1 - 0.01, mean = 0, sd = 1)
print(z_threshold)
```

```
## [1] 2.326348
```

Thus, the z-statistic is much greater than the threshold and there is evidence to suggest the cartons are overfilled.

10.9 Problem 2

Parameters given by the problem:

```
x_bar <- 103.11
s <- 53.5
mu <- 100
n <- 45
```

Compute the z-statistic assuming large sample assumptions apply:

```
z <- ( x_bar - mu )/(s/sqrt(n))
print(z)
```

```
## [1] 0.3899535
```

Now, work out the thresholds of the critical regions:

```r
z_upper <- qnorm(1 - 0.025, mean = 0, sd = 1)
print(z_upper)
```

```
## [1] 1.959964
```

```r
z_lower <- qnorm(0.025, mean = 0, sd = 1)
print(z_lower)
```

```
## [1] -1.959964
```

The z-statistic is outside the critical regions and therefore we do not reject the null hypothesis.

10.10 Problem 3

```r
z_test <- function(x, mu, popvar){

  one_tail_p <- NULL

  z_score <- round((mean(x) - mu) / (popvar / sqrt(length(x))), 3)

  one_tail_p <- round(pnorm(abs(z_score),lower.tail = FALSE), 3)

  cat(" z =", z_score, "\n",
    "one-tailed probability =", one_tail_p, "\n",
    "two-tailed probability =", 2 * one_tail_p)

  return(list(z = z_score, one_p = one_tail_p, two_p = 2 * one_tail_p))
}

x <- rnorm(10, mean = 0, sd = 1) # generate some artificial data from a N(0, 1)
out <- z_test(x, 0, 1) # null should not be rejected!
```

```
##  z = -1.305
##  one-tailed probability = 0.096
##  two-tailed probability = 0.192
```

```r
print(out)
```

```
## $z
## [1] -1.305
##
## $one_p
## [1] 0.096
##
## $two_p
## [1] 0.192
```

```r
x <- rnorm(10, mean = 1, sd = 1) # generate some artificial data from a N(1, 1)
out <- z_test(x, 0, 1) # null should be rejected!
```

```
##  z = 1.356
##  one-tailed probability = 0.088
##  two-tailed probability = 0.176
```

```r
print(out)
```

```
## $z
```

```
## [1] 1.356
##
## $one_p
## [1] 0.088
##
## $two_p
## [1] 0.176
```

10.11 Problem 4

Define some parameters

```
mu <- 5.4
n <- 5
x_bar <- 5.64
s2 <- 0.05
```

Compute the t-statistic:

```
t <- (x_bar - mu) / sqrt(s2 / n)
print(t)
```

```
## [1] 2.4
```

Work out the thresholds of the critical regions:

```
t_upper <- qt(1 - 0.025, df = n - 1)
print(t_upper)
```

```
## [1] 2.776445
```

```
t_lower <- qt(0.025, df = n - 1)
print(t_lower)
```

```
## [1] -2.776445
```

The t-statistic is outside of the critical regions so we do not reject the null hypothesis.

10.12 Problem 5

Define the parameters:

```
x_bar_a <- 88
s2_a <- 4.5
n_a <- 72
x_bar_b <- 79
s2_b <- 4.2
n_b <- 48
mu_a <- 0
mu_b <- 0
```

Compute the z-statistic:

```
z <- ((x_bar_a - x_bar_b) - (mu_a - mu_b)) / sqrt(s2_a / n_a + s2_b / n_b)
print(z)
```

```
## [1] 23.2379
```

Work out for the 5% significance level, the critical values:

```r
z_upper <- qnorm(1 - 0.05, mean = 0, sd = 1)
print(z_upper)
```

```
## [1] 1.644854
```

There is evidence to support the claim that process A yields higher pressurisation.

10.13 Problem 6

```r
# Data vectors
x_A <- c(91.50, 94.18, 92.18, 95.39, 91.79, 89.07, 94.72, 89.21)
x_B <- c(89.19, 90.95, 90.46, 93.21, 97.19, 97.04, 91.07, 92.75)

# parameters based on data
x_bar_A <- mean(x_A)
s2_A <- var(x_A)
n_A <- length(x_A)
x_bar_B <- mean(x_B)
s2_B <- var(x_B)
n_B <- length(x_B)
```

Compute the pooled variance estimator:

```r
s2_p <- ((n_A - 1) * s2_A + (n_B - 1) * s2_B) / (n_A + n_B - 2)
print(s2_p)
```

```
## [1] 7.294654
```

Compute the t-statistic:

```r
t = ( x_bar_A - x_bar_B ) / sqrt( s2_p*(1/n_A + 1/n_B) )
print(t)
```

```
## [1] -0.3535909
```

Work out the critical values:

```r
t_upper <- qt(1 - 0.025, df = n_A + n_B - 2)
print(t_upper)
```

```
## [1] 2.144787
```

```r
t_lower <- qt(0.025, df = n_A + n_B - 2)
print(t_lower)
```

```
## [1] -2.144787
```

Since $|t| < 2.14$ we have no evidence to reject the null hypothesis that the mean yields are equal.

Now, let us use the built-in `t.test` command:

```r
  out <- t.test(x = x_A, y = x_B, paired = FALSE, var.equal = TRUE,
    conf.level = 0.95, mu = 0, alternative = "two.sided")
  print(out)
```

```
##
##  Two Sample t-test
##
## data:  x_A and x_B
## t = -0.35359, df = 14, p-value = 0.7289
```

```
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -3.373886  2.418886
## sample estimates:
## mean of x mean of y
##   92.2550   92.7325
```

The options `paired=FALSE` means this is an unpaired t-test, `var.equal=TRUE` forces the estimated variances to be the same (i.e. we are using a pooled variance estimator) and we are testing at 95% confidence level with an alternative hypothesis that the true difference in means is non-zero.

The p-value for the t-test is between 0 and 1. In this case, the value is around 0.72 which means the hypothesis should not be reject.

10.14 Problem 7

Define parameters:

```
x <- c(23, 36, 31)
p <- c(1 / 3, 1 / 3, 1 / 3)
n <- sum(x)
K <- length(x)
```

Compute the expected counts:

```
Ex = p*n
```

Compute the chi-squared statistic:

```
chi2 <- sum((x - Ex)^2 / Ex)
print(chi2)
```

```
## [1] 2.866667
```

Compute the critical value form the chi-squared distribution:

```
chi_upper <- qchisq(1 - 0.05, df = K-1)
print(chi_upper)
```

```
## [1] 5.991465
```

Thus there is no evidence to reject the null hypothesis. The data provides no reason to suggest a preference for a particular door.

Now, we could have done this in `R`:

```
out <- chisq.test(x, p = c(1 / 3, 1 / 3, 1 / 3))
print(out)
```

```
##
##  Chi-squared test for given probabilities
##
## data:  x
## X-squared = 2.8667, df = 2, p-value = 0.2385
```

10.15 Problem 8

```
y <- c( 0, 1, 2 )
x <- c( 32, 12, 6 )
```

You will need the `vcdExtra` package to use the `expand.dft` command:

```
install.packages("vcdExtra")
```

The `expand.dft` command allows one to convert the frequency table into a vector of samples:

```
library(vcdExtra)
```

```
## Loading required package: vcd
```

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

```
## Loading required package: gnm
```

```
samples <- expand.dft(data.frame(y,Frequency = x), freq = "Frequency")
```

```
## Warning in type.convert.default(as.character(DF[[i]]), ...): 'as.is' should be specified by the
## caller; using TRUE
```

Now we can use the `fitdistr` function in the `MASS` package to estimate the MLE of the Poisson distribution

```
# loading the MASS package
library(MASS)
```

```
##
## Attaching package: 'MASS'
```

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

```
# fitting a Poisson distribution using maximum-likelihood
lambda_hat <- fitdistr(samples$y, densfun = 'Poisson')
```

Let just solve this directly using R built in function. First compute the expected probabilities under the Poisson distribution using `dpois` to compute the Poisson pdf:

```
pr <- c(0, 0, 0)
pr[1] <- dpois(0, lambda = lambda_hat$estimate)
pr[2] <- dpois(1, lambda = lambda_hat$estimate)
pr[3] <- 1 - sum(pr[1:2])
```

Then apply `chisq.test`:

```
out <- chisq.test(x, p = pr)
```

```
## Warning in chisq.test(x, p = pr): Chi-squared approximation may be incorrect
```

```
print(out)
```

```
##
##  Chi-squared test for given probabilities
##
## data:  x
## X-squared = 1.3447, df = 2, p-value = 0.5105
```

Actually, in this case the answer is wrong(!), we need to apply an additional loss of degree of freedom to account for the use of the MLE. However, we can re-use values already computed by `chisq.test`:

```
chi2 <- out$statistic
print(chi2)
```

```
## X-squared
```

```
##   1.34466
```

```
chi2_lower <- qchisq(1 - 0.01, df = 1)
print(chi2_lower)
```

```
## [1] 6.634897
```

Hence, there is no evidence to reject the null hypothesis. There is no reason to suppose that the Poisson distribution is not a plausible model for the number of accidents per week at this junction.

11 Practical: Linear regression

In this practical you will go through some of the basics of linear modeling in `R` as well as simulating data. The practical contains the following elements:

- simulate linear regression model
- investigate parameters
- characterize prediction accuracy
- correlation of real world data

We will use `reshape2`, `ggplo2`, and `bbmle` packages. Run the following command to make sure they are installed and loaded

```
install.packages("ggplot2")
install.packages("reshape2")
install.packages("bbmle")
```

```
library(ggplot2)
library(reshape2)
library(bbmle)
```

11.1 Data

For this practical you will require three datasets:

- `stork.txt` (download)
- `lr_data1.Rdata` (download)
- `lr_data2.Rdata` (download).

11.2 Simulating data

You will simulate data based on the simple linear regression model:

$$y_i = \beta_0 + \beta_1 x_i + \epsilon_i,$$

where (x_i, y_i) represent the i-th measurement pair with $i = 1, \ldots, N$, β_0 and β_1 are regression coefficients representing intercept and slope respectively. We assume the noise term $\epsilon_i \sim N(0, \sigma^2)$ is normally distributed with zero mean and variance σ^2.

First we define the values of the parameters of linear regression $(\beta_0, \beta_1, \sigma^2)$:

```
b0 <- 10 # regression coefficient for intercept
b1 <- -8 # regression coefficient for slope
sigma2 <- 0.5 # noise variance
```

In the next step we will simulate $N = 100$ covariates x_i by randomly sampling from the standard normal distribution:

```r
set.seed(198) # set a seed to ensure data is reproducible
N <- 100 # no of data points to simulate
x <- rnorm(N, mean = 0, sd = 1) # simulate covariate
```

Next we simulate the error term:

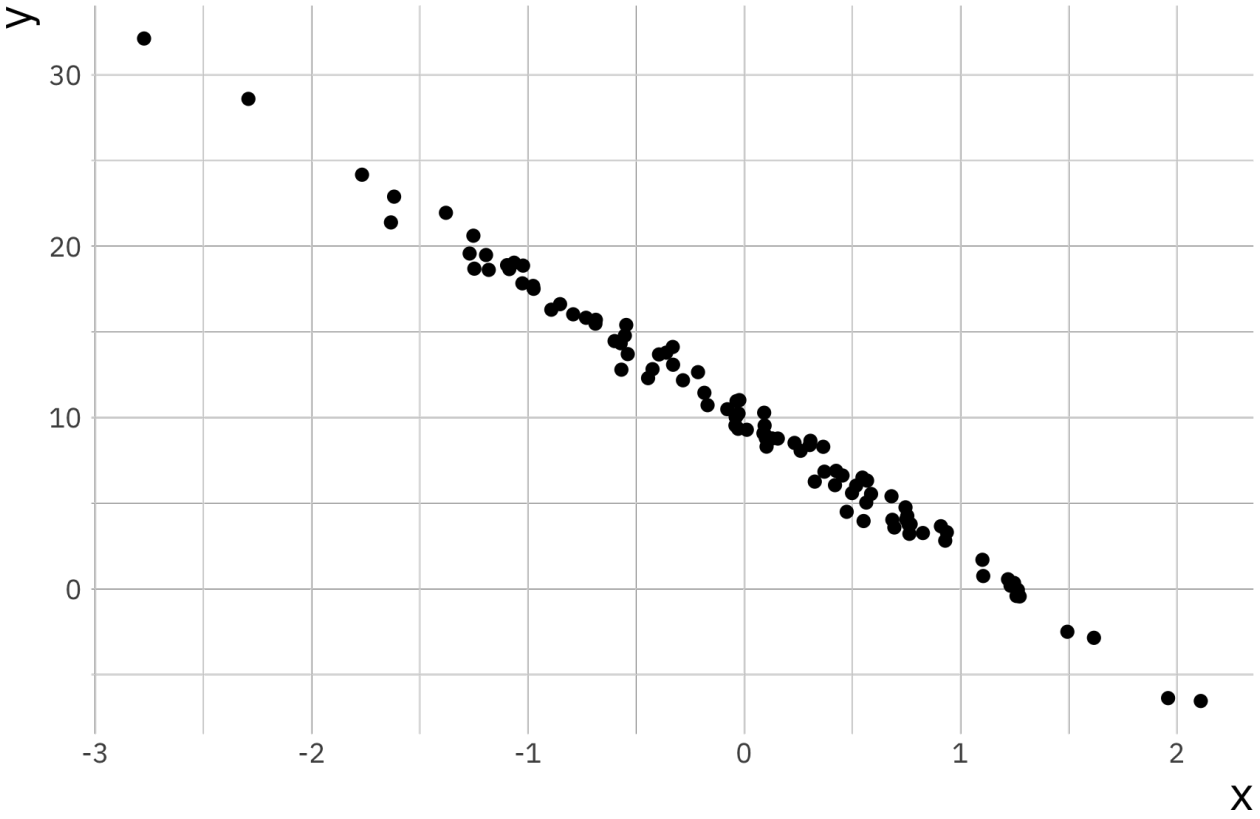
```r
# simulate the noise terms, rnorm requires the standard deviation
e <- rnorm(N, mean = 0, sd = sqrt(sigma2))
```

Finally we have all the parameters and variables to simulate the response variable y:

```r
# compute (simulate) the response variable
y = b0 + b1 * x + e
```

We will plot our data using `ggplot2` so the data need to be in a `data.frame` object:

```r
# Set up the data point
sim_data <- data.frame(x = x, y = y)

# create a new scatter plot using ggplot2
ggplot(sim_data, aes(x = x, y = y)) +
  geom_point()
```

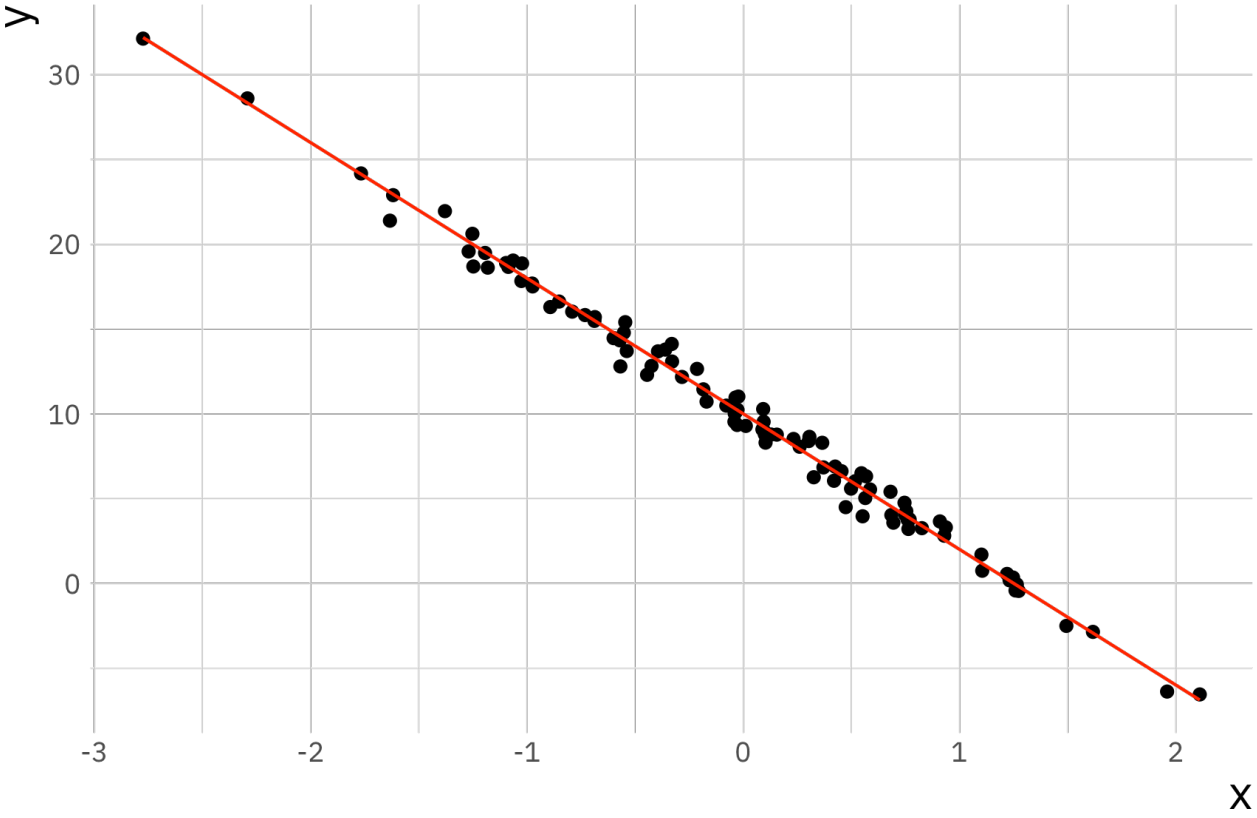


We define the true data `y_true` to be the true linear relationship between the covariate and the response without the noise.

```r
# Compute true y values
y_true <- b0 + b1 * x

# Add the data to the existing data frame
sim_data$y_true <- y_true
```

Now we will add the true values of y to the scatter plot:

```
lr_plot <- ggplot(sim_data, aes(x = x, y = y)) +
  geom_point() +
  geom_line(aes(x = x, y = y_true), colour = "red")

print(lr_plot)
```



11.3 Fitting simple linear regression model

11.3.1 Least squared estimation

Now that you have simulated data you can use it to regress y on x, since this is simulated data we know the parameters and can make a comparison. In `R` we can use the function `lm()` for this, by default it implements a least squares estimate:

```
# Use the lm function to fit the data
ls_fit <- lm(y ~ x, data = sim_data)

# Display a summary of fit
summary(ls_fit)
```

```
##
## Call:
## lm(formula = y ~ x, data = sim_data)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
```

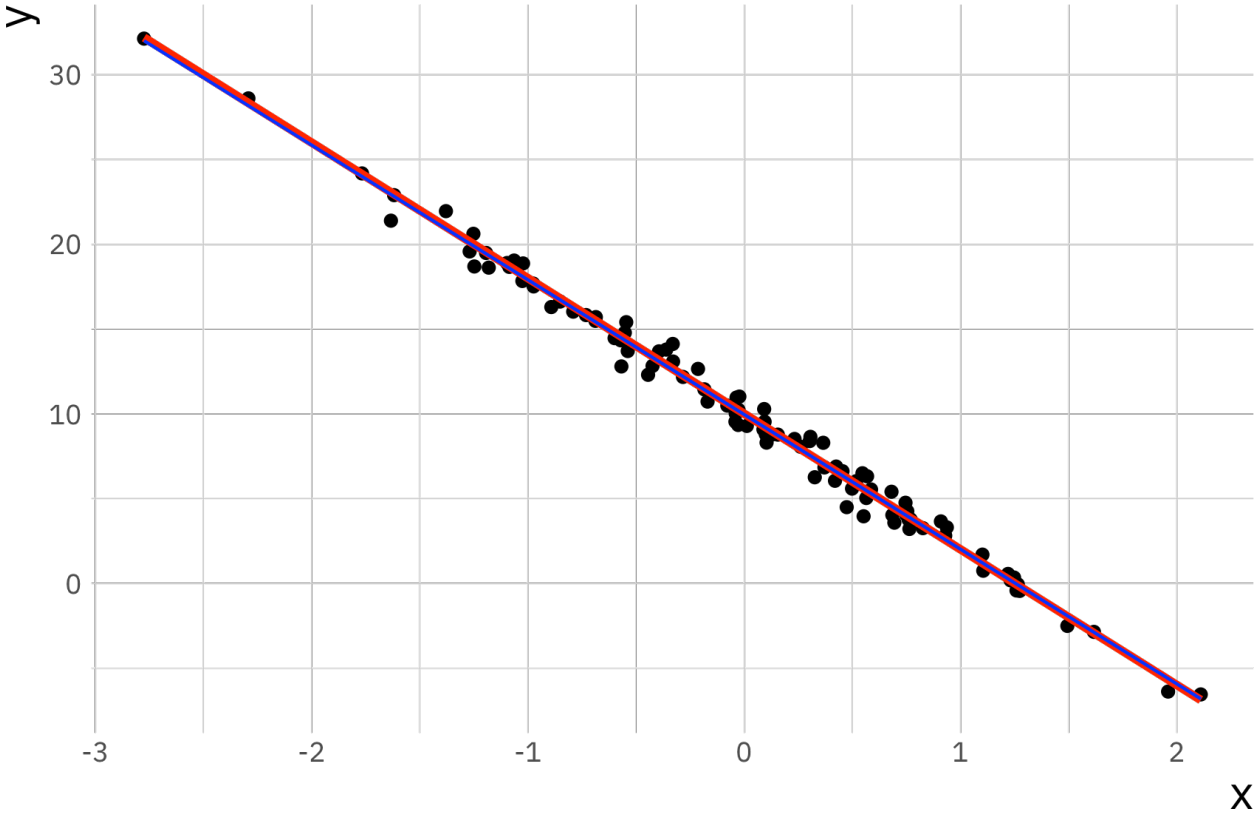
```
## -1.69905 -0.41534  0.02851  0.41265  1.53651
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  9.95698    0.06701   148.6   <2e-16 ***
## x           -7.94702    0.07417  -107.1   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.6701 on 98 degrees of freedom
## Multiple R-squared:  0.9915, Adjusted R-squared:  0.9914
## F-statistic: 1.148e+04 on 1 and 98 DF,  p-value: < 2.2e-16
```

The output for `lm()` is an object (in this case `ls_fit`) which contains multiple variables. To access them there are some built in functions, e.g. `coef()`, `residuals()`, and `fitted()`. We will explore these in turn:

```
# Extract coefficients as a named vector
ls_coef <- coef(ls_fit)

print(ls_coef)
```

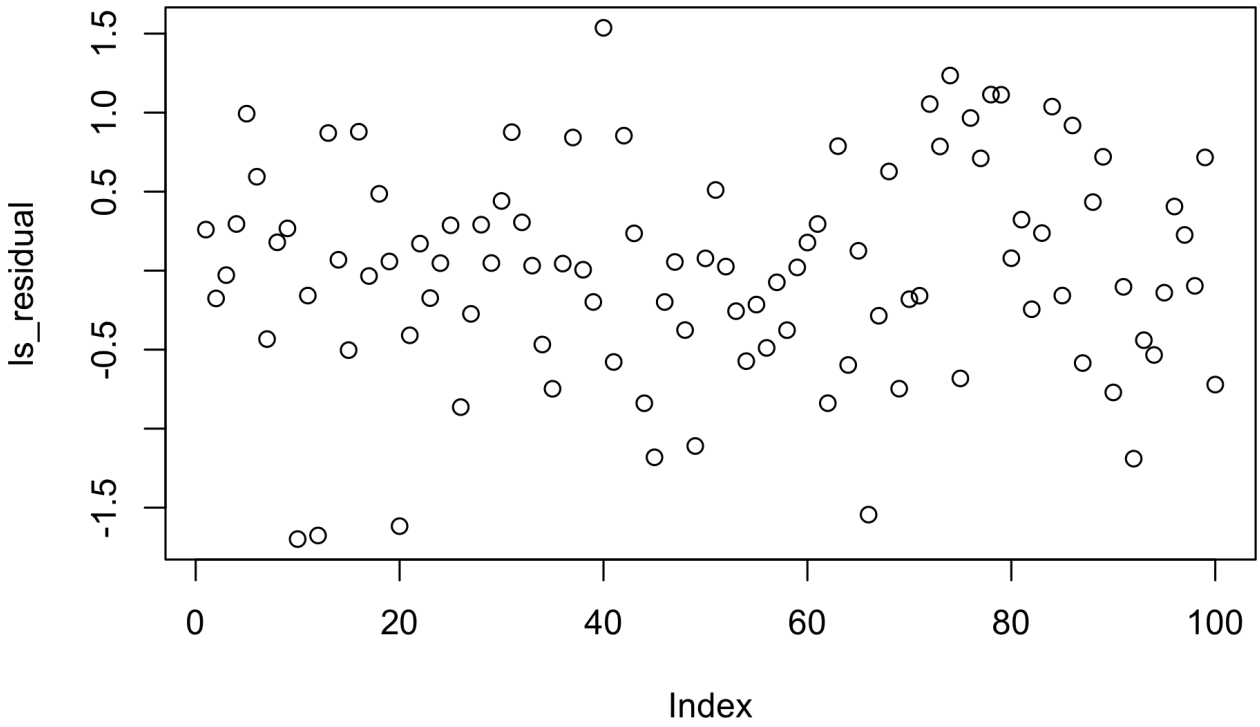
```
## (Intercept)           x
##    9.956981   -7.947016
```

```
# Extract intercept and slope
b0_hat <- ls_coef[1] # alternative ls_fit$coefficients[1]
b1_hat <- ls_coef[2] # alternative ls_fit$coefficients[2]

# Generate the predicted data based on estimated parameters
y_hat <- b0_hat + b1_hat * x
sim_data$y_hat <- y_hat # add to the existing data frame

# Create scatter plot and lines for the original and fitted
lr_plot <- ggplot(sim_data, aes(x = x, y = y)) +
  geom_point() +
  geom_line(aes(x = x, y = y_true), colour = "red", size = 1.3) +
  # plot predicted relationship in blue
  geom_line(aes(x = x, y = y_hat), colour = "blue")

# force Rstudio to display the plot
  print(lr_plot)
```



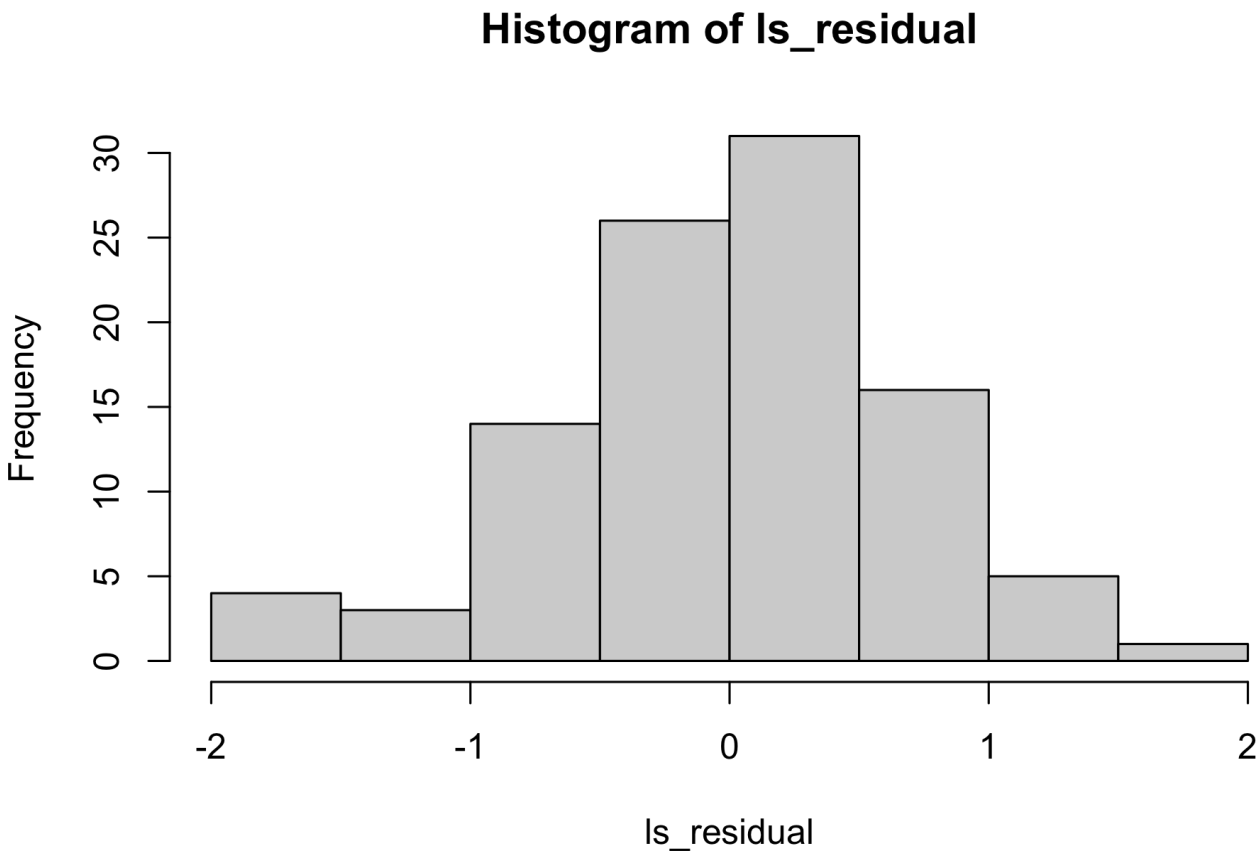
The estimated parameters and the plot shows a good correspondence between fitted regression parameters and the true relationship between y and x. We can check this by plotting the residuals, this data is stored as the `residuals` parameter in the `ls_fit` object.

```
# Residuals
ls_residual <- residuals(ls_fit) # can also be accessed via ls_fit$residuals

# scatter plot of residuals
plot(ls_residual)
```



A better way of summarising the data is to visualise them as a histogram:

```
hist(ls_residual)
```



We expect the mean and variance of the residuals to be close to the level used to generate the data.

```
print(mean(ls_residual))
```

```
## [1] -7.157903e-18
```

```
print(var(ls_residual))
```

```
## [1] 0.4444955
```

This is as expected since subtracting a good fit from the data leaves ϵ which has 0 mean and 0.5 variance.

11.3.2 Maximum likelihood estimation

Next you will look at maximum likelihood estimation based on the same data you simulated earlier. This is a bit more involved as it requires you to explicitly write the function you wish to minimise. The function we use is part of the `bbmle` package.

```
# Loading the required package
library(bbmle)

# function that will be minimised. It takes as arguments all parameters
# Here we are helped by the way R works we don't have to explicitly pass x.
# The function will use the existing estimates in the environment
mle_ll <- function(beta0, beta1, sigma) {
  # first we predict the response variable based on the guess for our response
  y_pred = beta0 + beta1 * x

  # next we calculate the normal distribution based on the predicted value
  # the guess for sigma and return the log
  log_lh <- dnorm(y, mean = y_pred, sd = sigma, log = TRUE)

  #  We returnr the negative sum of the log likelihood
  return(-sum(log_lh))
}

# This is the function that actually performs the estimation
# The first variable here is the function we will use
# The second variable passed is a list of initial guesses of parameters
mle_fit <- mle2(mle_ll, start = list(beta0 = -1, beta1 = 20, sigma = 10))

# With the same summary function as above we can output a summary of the fit
summary(mle_fit)
```

```
## Maximum likelihood estimation
##
## Call:
## mle2(minuslogl = mle_ll, start = list(beta0 = -1, beta1 = 20,
##     sigma = 10))
##
## Coefficients:
##        Estimate Std. Error  z value     Pr(z)
## beta0  9.957019   0.066336  150.099 < 2.2e-16 ***
## beta1 -7.947005   0.073426 -108.231 < 2.2e-16 ***
## sigma  0.663347   0.046904   14.143 < 2.2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
```

```
## -2 log L: 201.7011
```

The estimated parameters using the maximum likelihood are also a very good estimate of the true values.

11.4 Effect of variance

Now investigate the quality of the predictions further by simulating more data sets and seeing how the variance affects the quality of the fit as indicated by the mean-squared error (mse).

To start you will define some parameter for the simulations, the number of simulations to run for each variance, and the variance values to try.

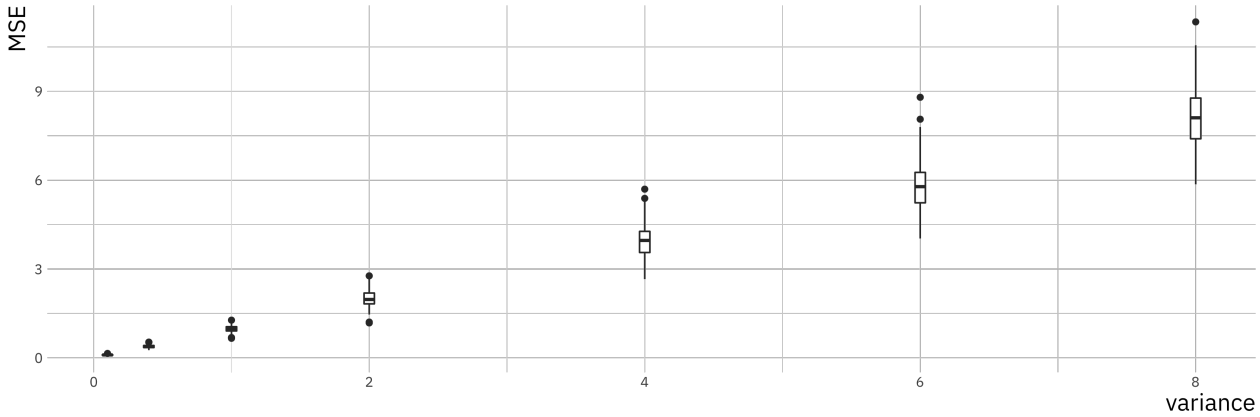
```
# number of simulations for each noise level
n_simulations <- 100

# A vector of noise levels to try
sigma_v <- c(0.1, 0.4, 1.0, 2.0, 4.0, 6.0, 8.0)
n_sigma <- length(sigma_v)

# Create a matrix to store results
mse_matrix <- matrix(0, nrow = n_simulations, ncol = n_sigma)

# name row and column
rownames(mse_matrix) <- c(1:n_simulations)
colnames(mse_matrix) <- sigma_v
```

Next we will write a nested `for` loop. The first loop will be over the variances and a second loop over the number of repeats. We will simulate the data, perform a fit with `lm()`. We can use the `fitted()` function on the resulting object to extract the fitted values $\hat{y}$ and use this to compute the mean-squared error from the true value y.

```
# loop over variance
for (i in 1:n_sigma) {
  sigma2 <- sigma_v[i]

  # for each simulation
  for (it in 1:n_simulations) {

    # Simulate the data
    x <- rnorm(N, mean = 0, sd = 1)
    e <- rnorm(N, mean = 0, sd = sqrt(sigma2))
    y <- b0 + b1 * x + e

    # set up a data frame and run lm()
    sim_data <- data.frame(x = x, y = y)
    lm_fit <- lm(y ~ x, data = sim_data)

    # compute the mean squared error between the fit and the actual y's
    y_hat <- fitted(lm_fit)
    mse_matrix[it, i] <- mean((y_hat - y)^2)

  }
}
```

We created a matrix to store the mse values, but to plot them using `ggplot2` we have to convert them to a `data.frame`. This can be done using the `melt()` function form the `reshape2` library. We can compare the results using boxplots.

```
library(reshape2)

# convert the matrix into a data frame for ggplot2
mse_df <- melt(mse_matrix)
# rename the columns
names(mse_df) <- c("Simulation", "variance", "MSE")

# now use a boxplot to look at the relationship between
# mean-squared prediction error and variance
mse_plt <- ggplot(mse_df, aes(x = variance, y = MSE)) +
  geom_boxplot(aes(group = variance))

print(mse_plt)
```



You can see that the variances of the mse and the value of the mse go up with increasing variance in the simulation.

What changes do you need to make to the above function to plot the accuracy of the estimated regression coefficients as a function of variance?

11.5 Exercise

11.5.1 Part I

Read in the data in `stork.txt`, compute the correlation and comment on it.

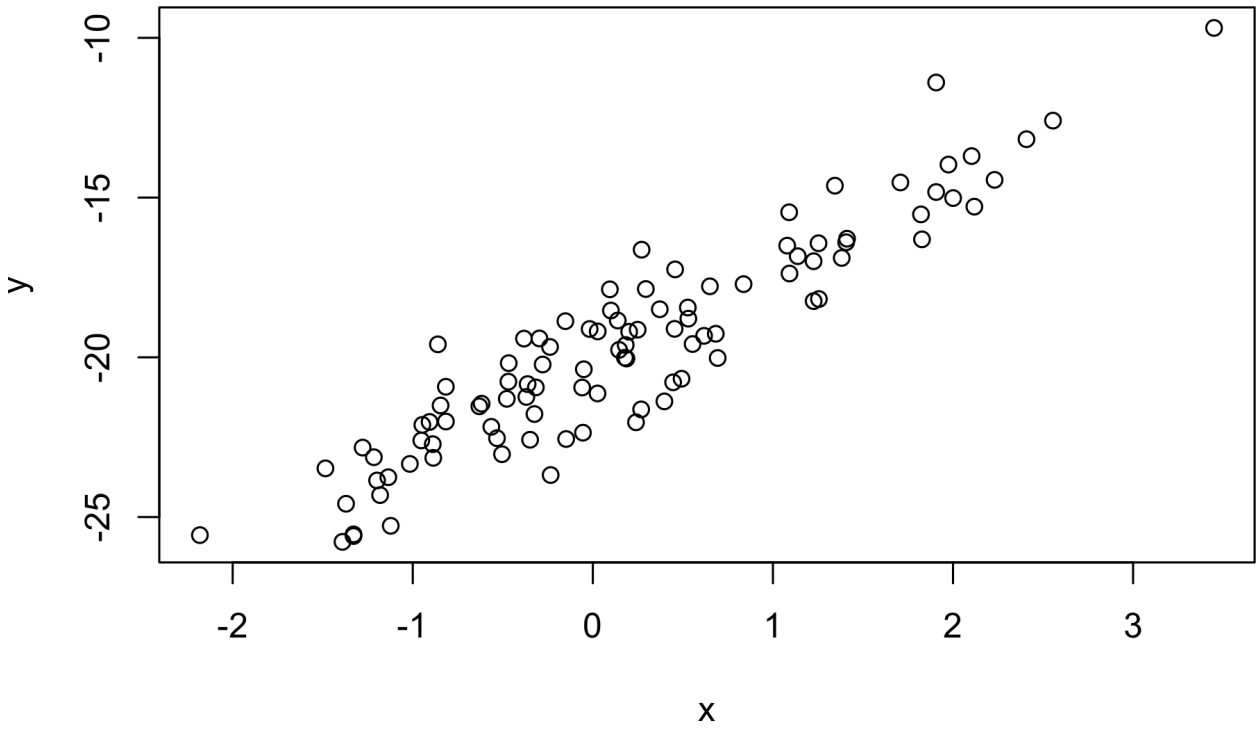
The data represents `no of storks` (column 1) in Oldenburg Germany from $1930 - 1939$ and the number of people (column 2).

11.5.2 Part II

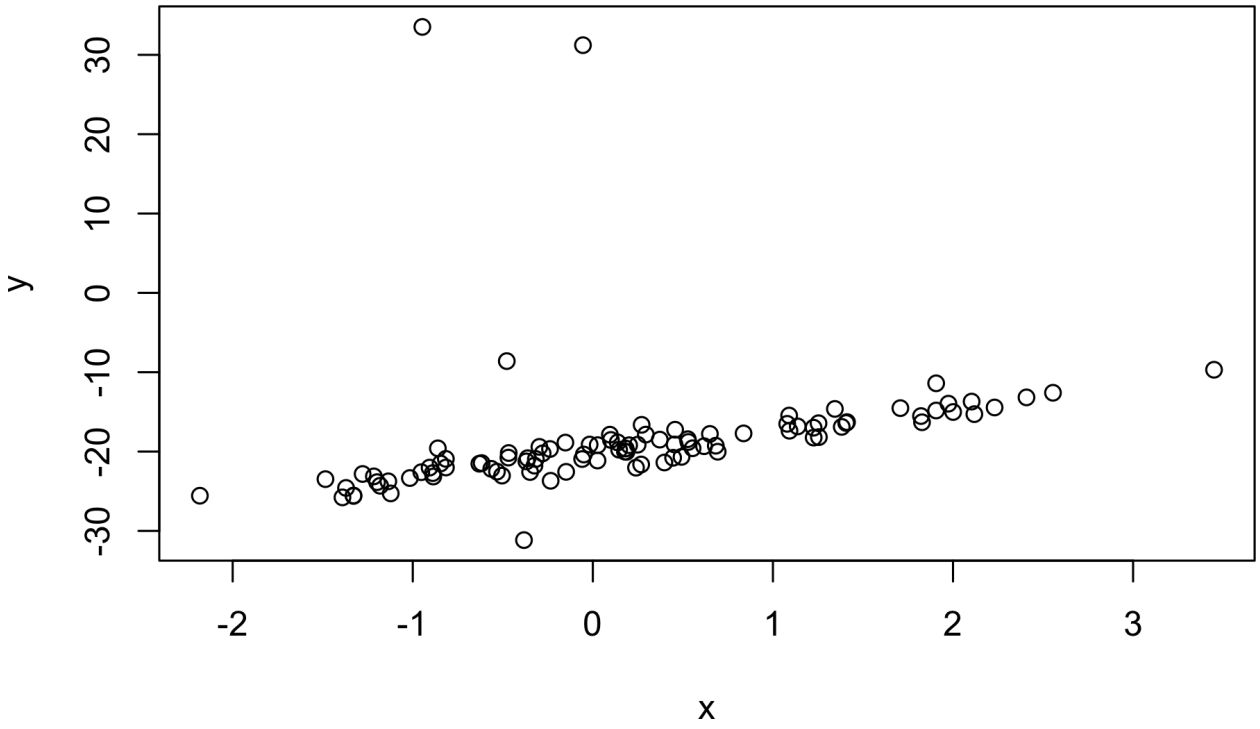
Fit a simple linear model to the two data sets supplied (`lr_data1.Rdata` and `lr_data2.Rdata`). In both files the (x, y) data is saved in two vectors, x and y.

Download the data from Canvas, you can read it into `R` and plot it with the following commands:

```
load("lr_data1.Rdata")
plot(x, y)
```




```
load("lr_data2.Rdata")
plot(x, y)
```



Fit the linear model and comment on the differences between the data.

11.5.3 Part III

Investigate how the sample size will affect the quality of the fit using mse, use the code for investigating the affect of variance as inspiration.

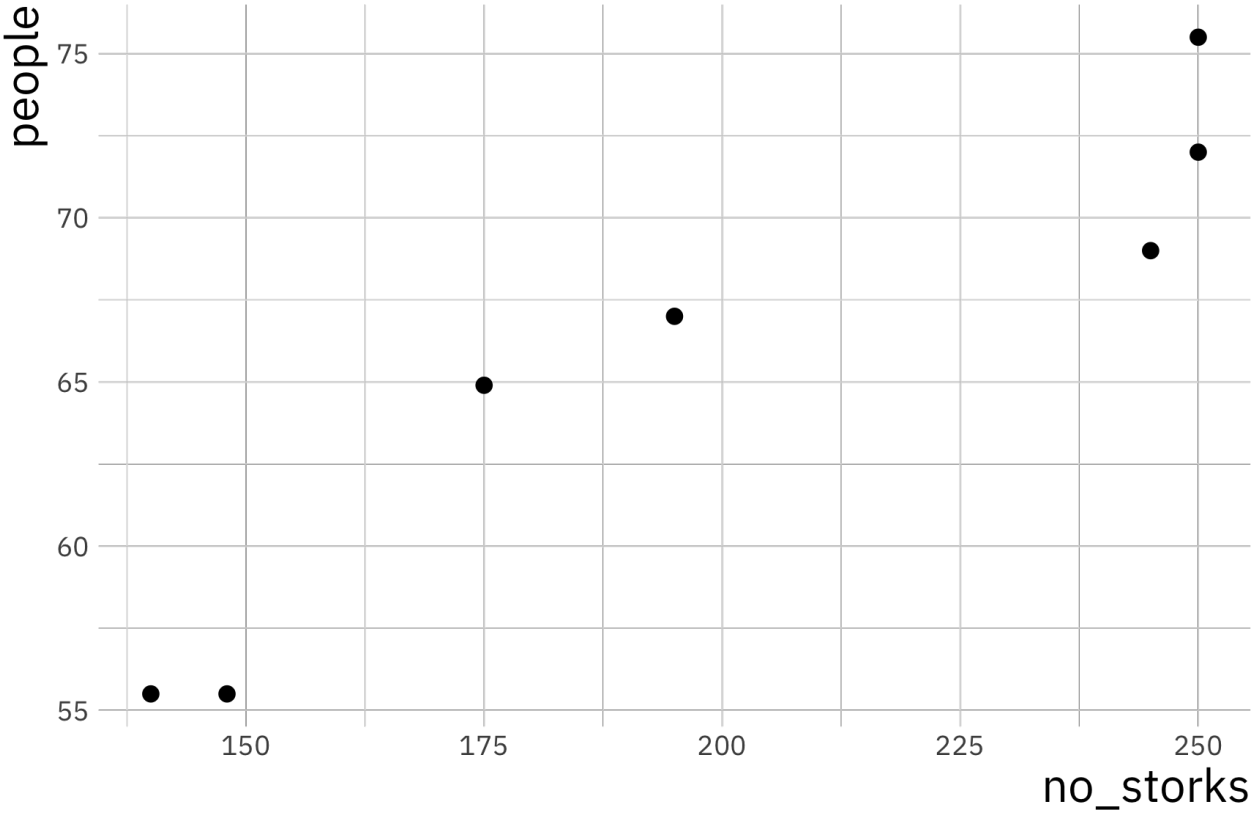
Model answers: Linear regression

11.6 Exercise I

```
library(ggplot2)
library(reshape2)
```

```
stork_dat <- read.table("stork.txt", hedaer = TRUE)
```

```
ggplot(stork_dat, aes(x = no_storks, y = people)) +
  geom_point(size = 2)
```



This is a plot of number of people in Oldenburg (Germany) against the number of storks. We can calculate the correlation in R

```
cor(stork_dat$no_storks, stork_dat$peopl)
```

```
## [1] 0.9443965
```

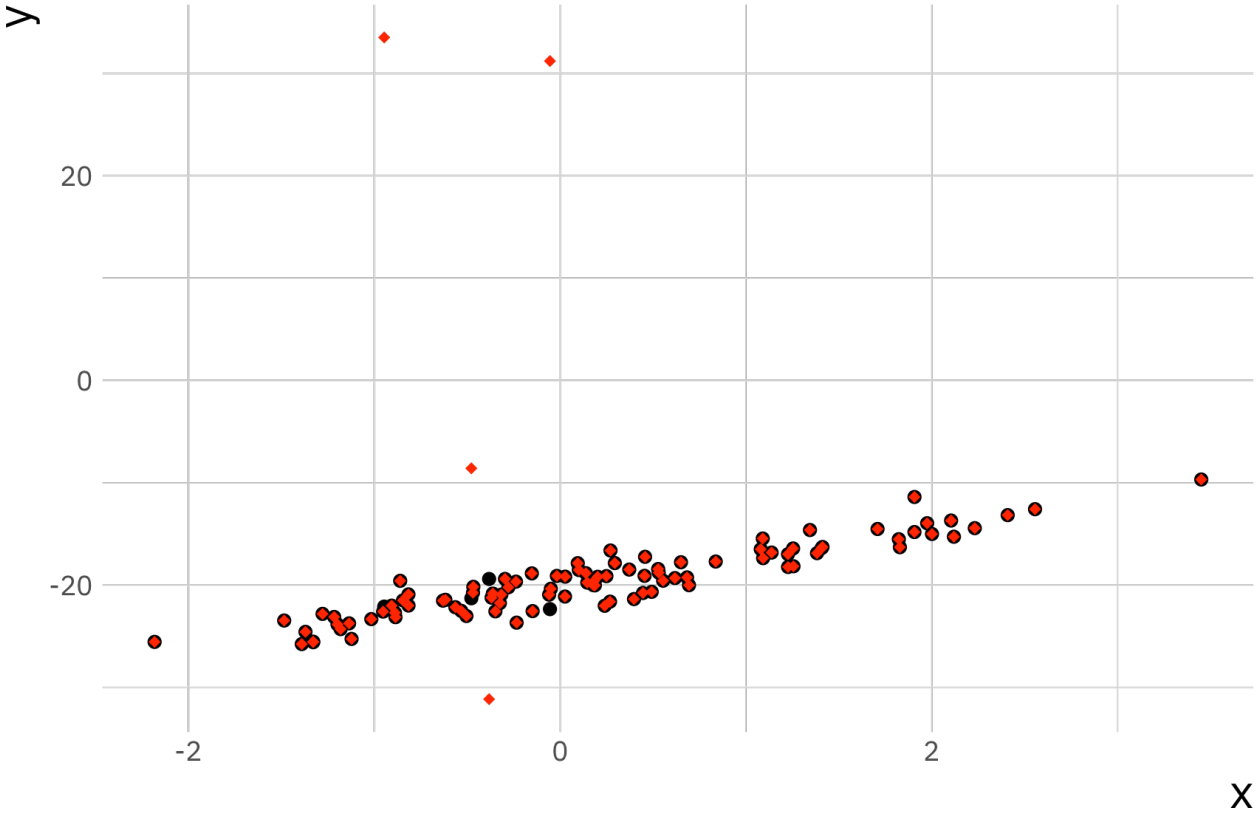
This is a very high correlation, and obviously there is no causation. Think about why there would be a correlation between these two random variables.

11.7 Exercise II

```
# load first data set and create data.frame
load("lr_data1.Rdata")
sim_data1 <- data.frame(x = x, y = y)

# load second data set and create data.frame
load("lr_data2.Rdata")
sim_data2 <- data.frame(x = x, y = y)

lr_fit1 <- lm(y ~ x, data = sim_data1)
lr_fit2 <- lm(y ~ x, data = sim_data2)
```

11.7.1 Comparison of data

```
ggplot(sim_data1, aes(x = x, y = y)) +
  geom_point(size = 1.5) +
  geom_point(data = sim_data2, color = "red", shape = 18)
```



If we plot the data on top of each other, the first data set in black and the second one in red, we can see a small number of points are different between the two data sets.

```
summary(lr_fit1)
```

```
##
## Call:
## lm(formula = y ~ x, data = sim_data1)
##
## Residuals:
```

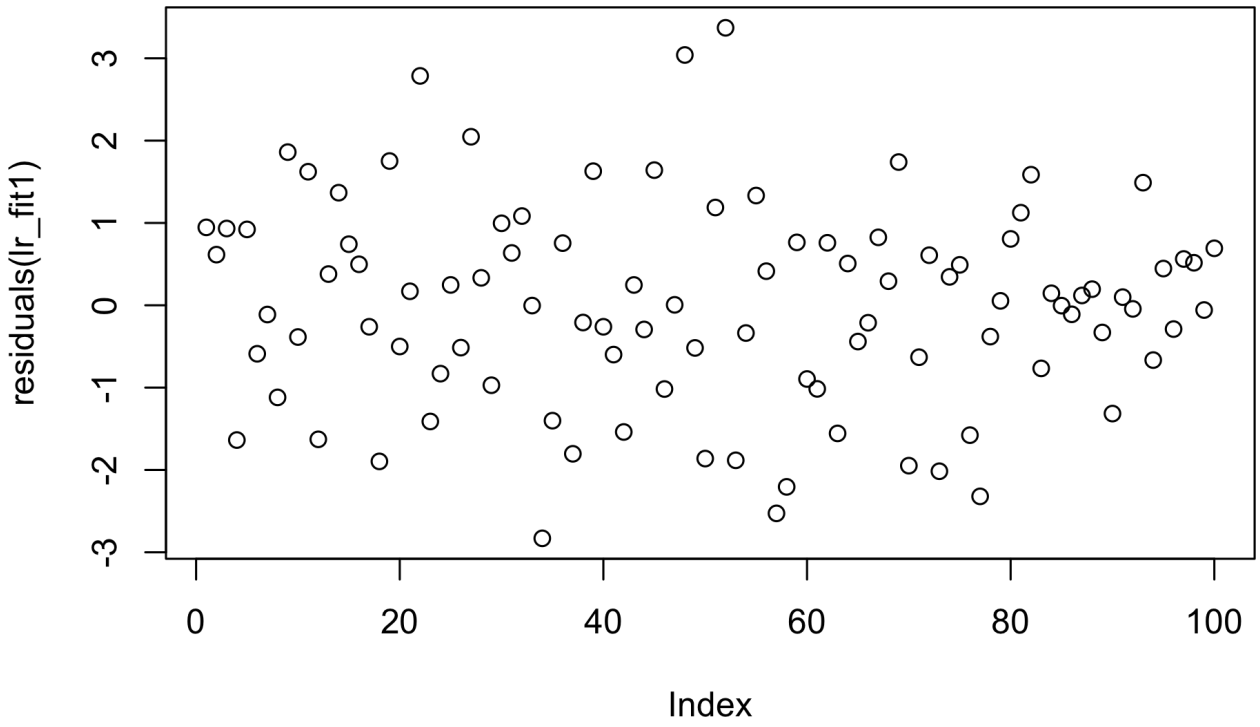
```
##     Min      1Q  Median      3Q     Max
## -2.8309 -0.6910  0.0296  0.7559  3.3703
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -20.1876     0.1250 -161.46   <2e-16 ***
## x             2.8426     0.1138   24.98   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.229 on 98 degrees of freedom
## Multiple R-squared:  0.8643, Adjusted R-squared:  0.8629
## F-statistic: 624.2 on 1 and 98 DF,  p-value: < 2.2e-16
```

```
summary(lr_fit2)
```

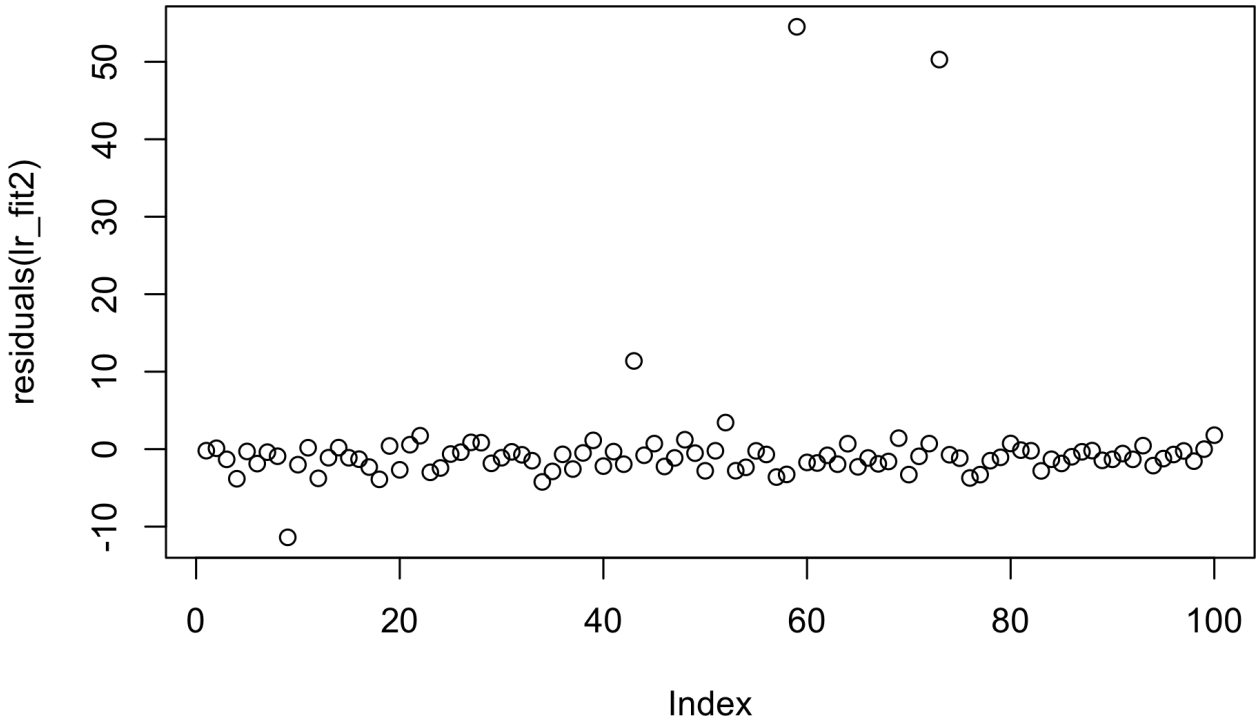
```
##
## Call:
## lm(formula = y ~ x, data = sim_data2)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -11.386  -1.960  -1.084  -0.206  54.516
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -18.9486     0.8006 -23.669  < 2e-16 ***
## x             2.1620     0.7285   2.968  0.00377 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 7.87 on 98 degrees of freedom
## Multiple R-squared:  0.08245,    Adjusted R-squared:  0.07309
## F-statistic: 8.806 on 1 and 98 DF,  p-value: 0.003772
```

From the summary data we can see a discrepancy between the two estimates in the regression coefficients (≈ 1), though the error in the estimate is quite large. The other thing to notice is that the summary of the residuals look quite different. If we investigate further and plot them we see:

```
plot(residuals(lr_fit1))
```

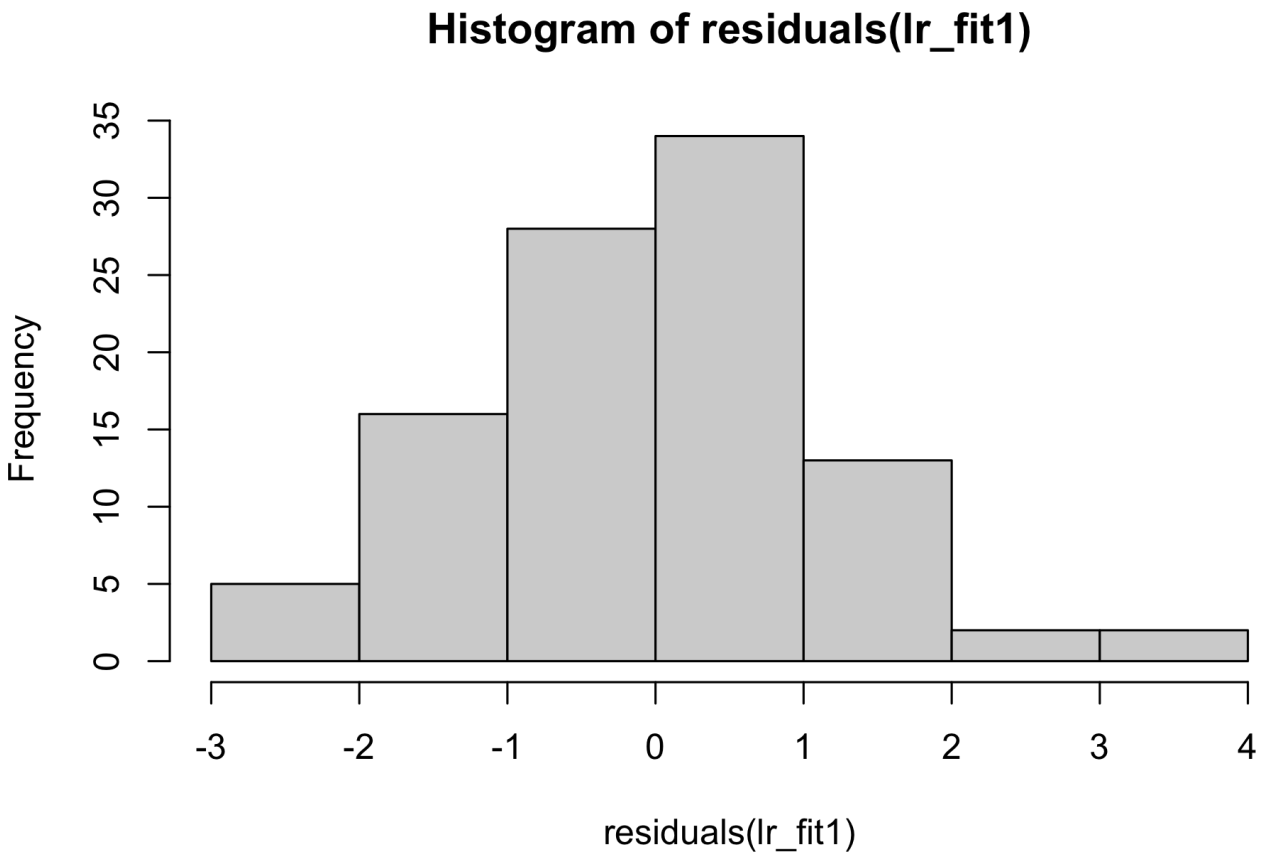



```
plot(residuals(lr_fit2))
```

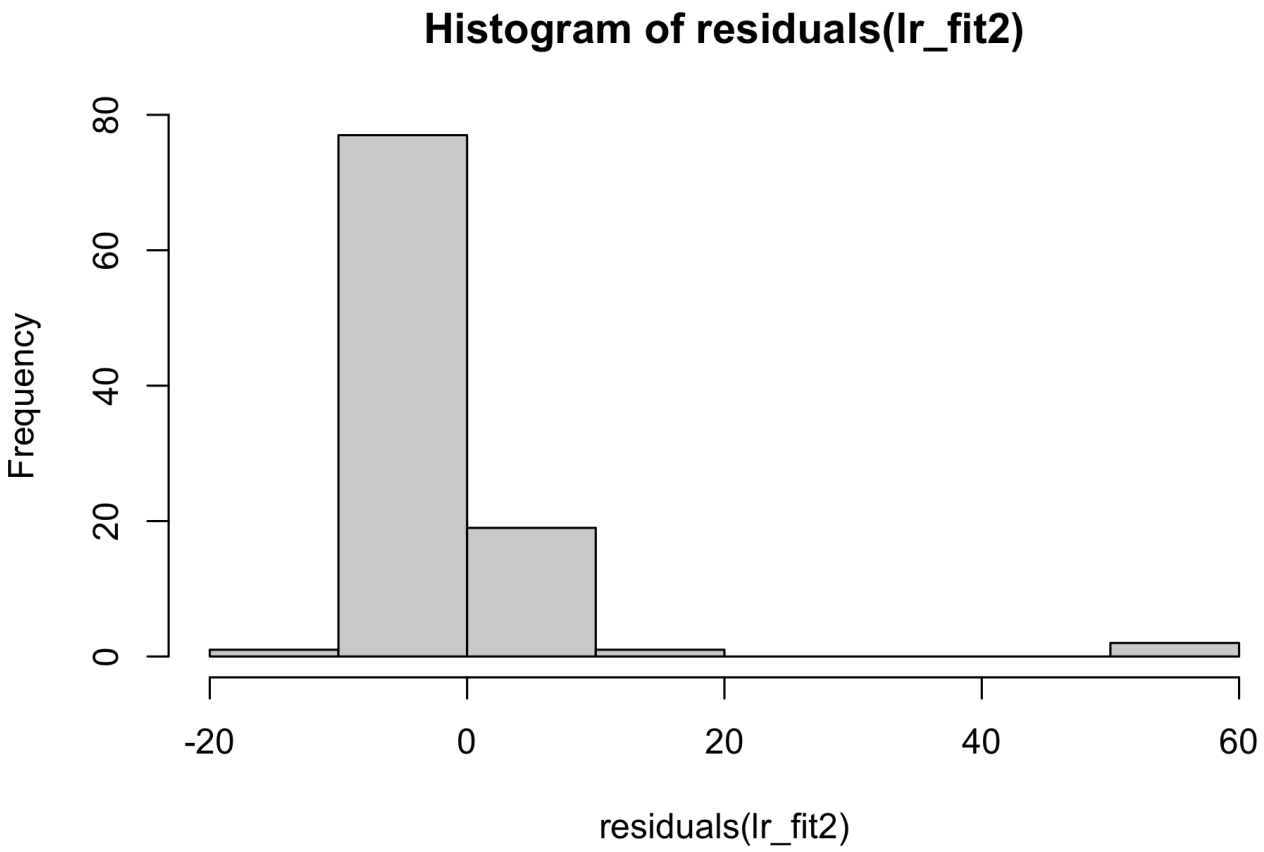


Here we can once again see the outliers in the second data set which affect the estimation. We now plot the histogram and boxplots for comparison:

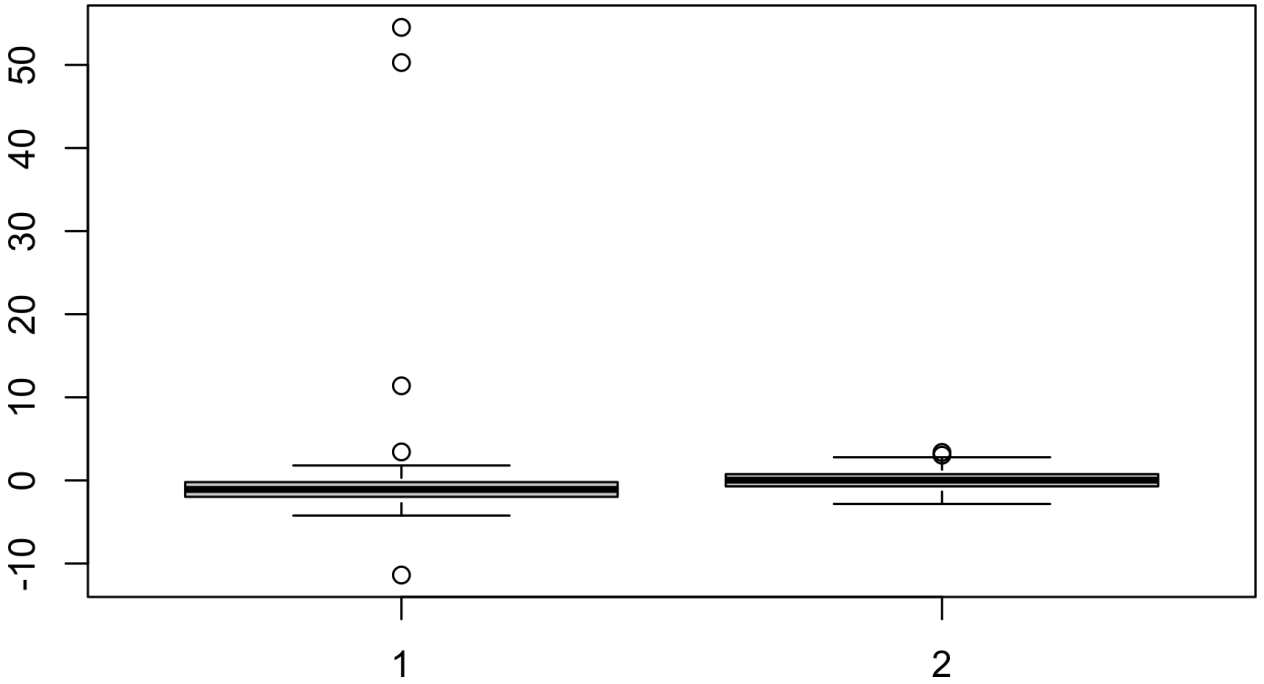
```
hist(residuals(lr_fit1))
```




```
hist(residuals(lr_fit2))
```




```
boxplot(residuals(lr_fit2), residuals(lr_fit1))
```



Her we can see that the distribution of the residuals has significantly changed in data set 2.

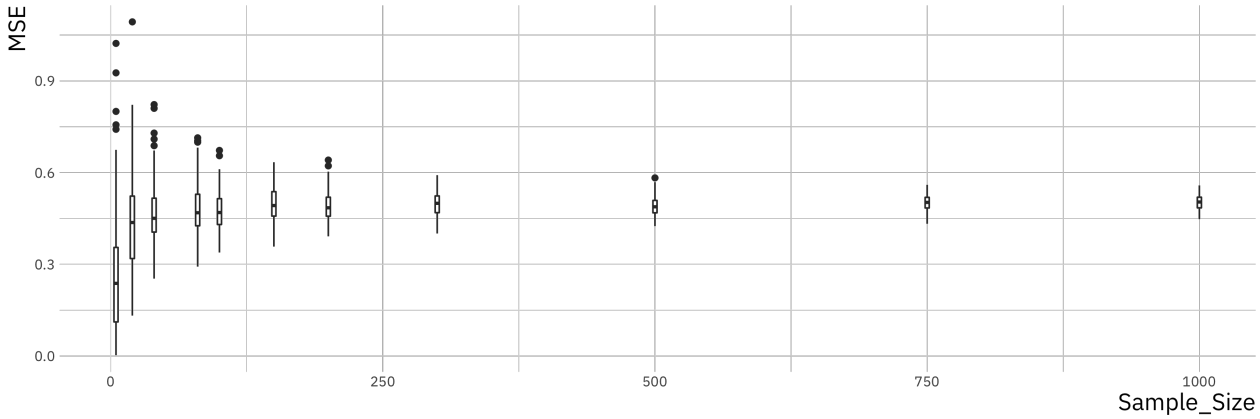
A change in only 4 data points was sufficient to change the regression coefficients.

11.8 Exercise II

```
b0 <- 10 # regression coefficient for intercept
b1 <- -8 # regression coefficient for slope
sigma2 <- 0.5 # noise variance

# number of simulations for each sample size
n_simulations <- 100

# A vector of sample sizes to try
sample_size_v <- c( 5, 20, 40, 80, 100, 150, 200, 300, 500, 750, 1000 )

n_sample_size <- length(sample_size_v)

# Create a matrix to store results
mse_matrix <- matrix(0, nrow = n_simulations, ncol = n_sample_size)

# name row and column
rownames(mse_matrix) <- c(1:n_simulations)
colnames(mse_matrix) <- sample_size_v
```

```
# loop over sample size
for (i in 1:n_sample_size) {
  N <- sample_size_v[i]

  # for each simulation
  for (it in 1:n_simulations) {

    x <- rnorm(N, mean = 0, sd = 1)
    e <- rnorm(N, mean = 0, sd = sqrt(sigma2))
    y <- b0 + b1 * x + e

    # set up a data frame and run lm()
    sim_data <- data.frame(x = x, y = y)
    lm_fit <- lm(y ~ x, data = sim_data)

    # compute the mean squared error between the fit and the actual y's
    y_hat <- fitted(lm_fit)
    mse_matrix[it, i] <- mean((y_hat - y)^2)

  }
}
```

```
library(reshape2)

mse_df = melt(mse_matrix) # convert the matrix into a data frame for ggplot
names(mse_df) = c("Simulation", "Sample_Size", "MSE") # rename the columns

# now use a boxplot to look at the relationship between mean-squared prediction error and sample size
mse_plt = ggplot(mse_df, aes(x=Sample_Size, y=MSE))
mse_plt = mse_plt + geom_boxplot( aes(group=Sample_Size) )
print(mse_plt)
```



You should see that the variance of the mean-squared error goes down as the sample size goes up and converges towards a limiting value. Larger sample sizes help reduce the variance in our estimators but do not make the estimates more accurate.

Can you do something similar to work out the relationship between how accurate the regression coefficient estimates are as a function of sample size?

12 Practical: Principal component analysis

In this practical we will practice some of the ideas outlined in the lecture on Principal Component Analysis (PCA), this will include computing principal components, visualisation techniques and an application to real data.

12.1 Data

For this practical we will use some data that is built into R and we require two additional files you will download:

- `Pollen2014.txt` (download)
- `SupplementaryLabels.txt` (download)

12.2 Introduction

We use PCA in order to explore complex datasets. By performing dimensionality reduction we can better visualize the data that has many variables. This technique is probably the most popular tool applied across bioscience problems (e.g. for gene expression problems).

In many real-world dataset we deal with a high dimensional data, e.g. for a number of individuals we can take a number of health related measurement (called variables). This is great, however having a large number of variables also means that it is difficult to plot the data as it is (in its "*raw*" format), and in turn it might be difficult to understand if this dataset contains any interesting patterns/trends/relationships across individuals. Using PCA we visualize such data in a more "*human friendly*" fashion.

Recall:

- PCA performs a linear transformation to data.
- This means that any input data can be visualized in a new coordinate system. The first coordinate (PC 1) variance is found on the first coordinate; each subsequent coordinate is orthogonal to the previous one and contains the larges variance from what was left.
- Each principal component is associated with certain percentage of the total variation in the dataset.
- If variables are strongly correlated with one another, a first few principal components will enable us to visualize the relationships present in any dataset.

- Eigenvectors describe new directions, whereas accompanying eigenvalues tell us how much variance there is in the data in given direction.
- The eigenvector with the highest eigenvalue is called the first principal component. The second highest eigenvalue would correspond to a second principle component and etc.
- There exist a d number of eigenvalues and eigenvectors; d is also equal to the size of the data (number of dimensions).
- For the purpose of visualization we preselect the first q components, where $q < d$.

12.3 Exercise I

There are many datasets built into `R`. Wed will look at the `mtcars` dataset. Type `?mtcars` to get a description of data. Then use `head()` function to have a look at the first few rows; and `dim()` to get the dimensions of the data.

```
library(ggplot2)
head(mtcars)
```

```
##                   mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
## Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

```
dim(mtcars)
```

```
## [1] 32 11
```

In this case we have 32 examples (cars in this case), and 11 features. Now we can perform a principal component analysis, in `R` it is implemented as the `prcomp()` function. We can type `?prcomp` to see a description of the function and some help on possible arguments. Here we set `center` and `scale` arguments to `TRUE`, recall from the lecture why this is important. We can try to perform PCA without scaling and centering and compare.

```
cars_pca <- prcomp(mtcars, center = TRUE, scale = TRUE)
```

We can use the `summary()` function to summarise the results from PCA, it will return the standard deviation, the proportion of variance explained by each principal component, and the cumulative proportion.

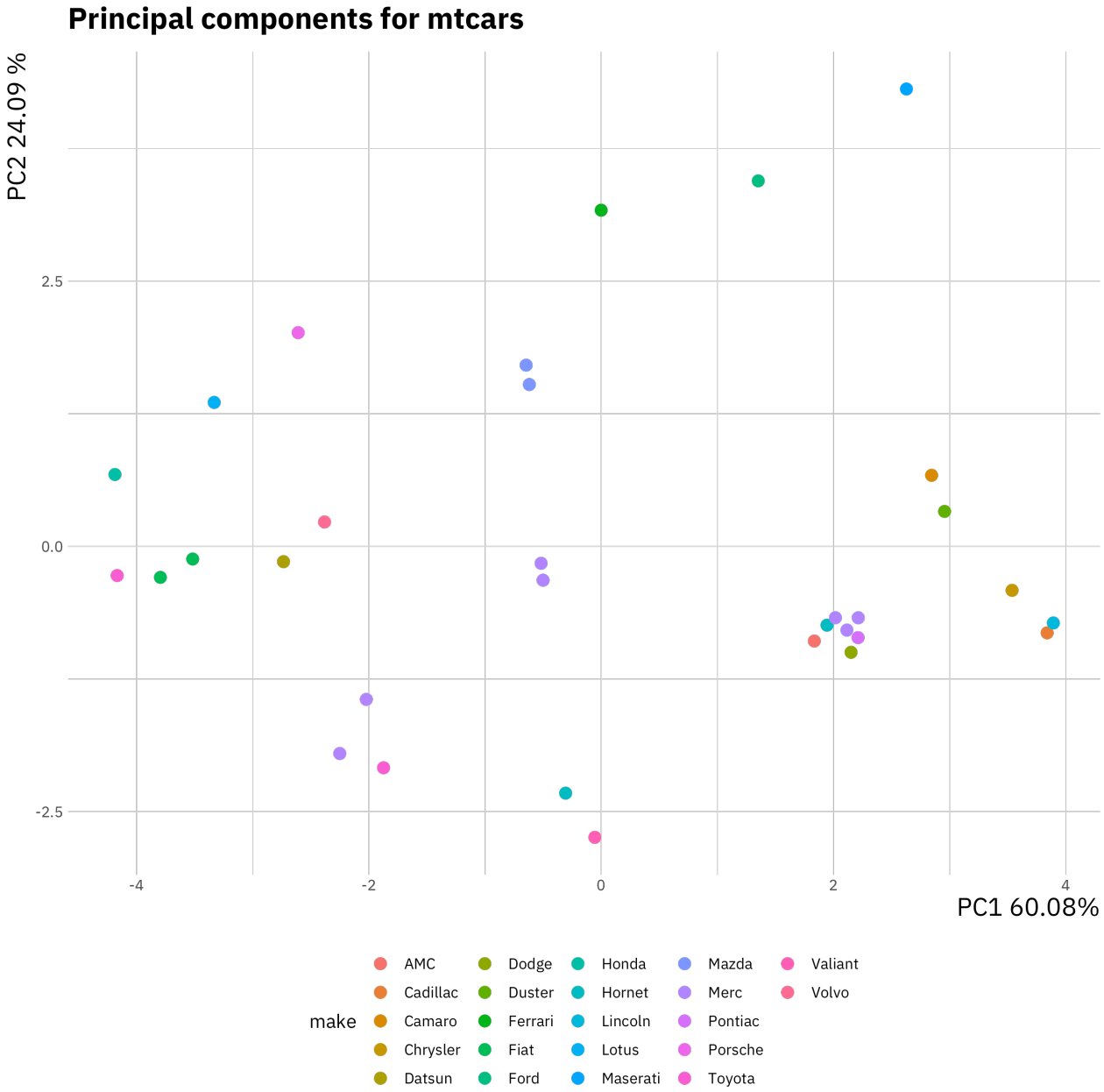
```
pca_summary <- summary(cars_pca)
print(pca_summary)
```

```
## Importance of components:
##                           PC1    PC2     PC3     PC4     PC5     PC6    PC7     PC8    PC9    PC10
## Standard deviation     2.5707 1.6280 0.79196 0.51923 0.47271 0.46000 0.3678 0.35057 0.2776 0.22811
## Proportion of Variance 0.6008 0.2409 0.05702 0.02451 0.02031 0.01924 0.0123 0.01117 0.0070 0.00473
## Cumulative Proportion  0.6008 0.8417 0.89873 0.92324 0.94356 0.96279 0.9751 0.98626 0.9933 0.99800
##                           PC11
## Standard deviation     0.1485
## Proportion of Variance 0.0020
## Cumulative Proportion  1.0000
```

Note, `Proportion of Variance` will always add up to 1. Here the `PC1` explain 60.08 of the variance, and `PC2` explains 24.09, which means together `PC1` and `PC2` account for 84.17 of the variance.

Using the `str()` function we can see the full structure of an `R` object, or alternatively using `?prcomp` in the *Value* section. In this case the `cars_pca` variable is a list containing several variables, `x` is the data represented using the new principal components. We can now plot the data in the first two principal components:

```
pca_df <- data.frame(cars_pca$x, make = stringr::word(rownames(mtcars), 1))

ggplot(pca_df, aes(x = PC1, y = PC2, col = make)) +
geom_point(size = 3) +
labs(x = "PC1 60.08%",
     y = "PC2 24.09 %",
     title = "Principal components for mtcars") +
theme(legend.position = "bottom")
```



Here we added a color based on the make of each car. We can observe which samples (or cars) cluster together. Have a look at these variables and decide why certain cars or models would cluster together.

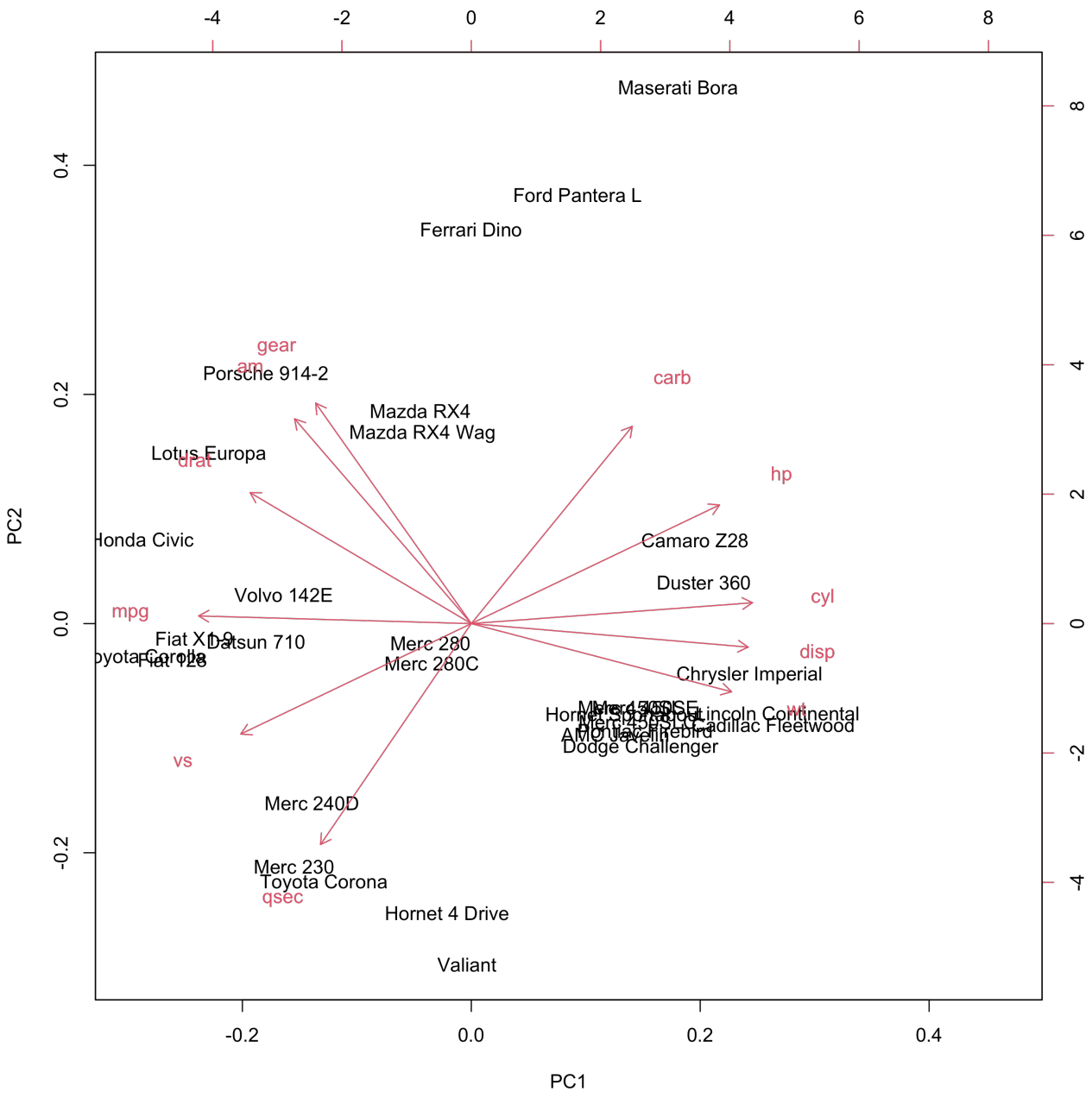
We created this plot using the `ggplot2` package, it is also possible to do this using base plot if you prefer.

```
plot(pca_df$PC1, pca_df$PC2)
```

12.4 Exercise II

Next we look at another representation of the data, the *biplot*. This is a combination of a PCA plot of the data and a *score plot*. We saw the PCA plot in the previous section in a *biplot* we add the original axis as arrows.

```
biplot(cars_pca)
```



We can see the original axis starting from the origin. Therefore we can make observations about the original variables (e.g. `cyl` and `mpg` contribute to PC1) and how the data points relates to these axes.

12.5 Exercise III

Now try to perform a PCA on the `USArrests` data also build into `R`. Typing `?USArrests` will give you further information on the data. Perform the analysis on the subset `USArrests[, -3]` data.

12.6 Exercise IV: Single cell data

We can now try to apply what we learned above on a more realistic datasets. You can download the data either on *canvas* or using these links `Pollen2014.txt` and `SupplementaryLabels.txt`. Her we will be dealing with single cell RNA-Seq (scRNA-Seq) data, which consist of 300 single cells measured across 8686 genes.

```
pollen_df <-read.table("Pollen2014.txt", sep=',', header = T,row.names=1)

label_df <-read.table("SupplementaryLabels.txt", sep=',', header = T)

pollen_df[1:10, 1:6]
```

```
##          Cell_2338_1 Cell_2338_10 Cell_2338_11 Cell_2338_12 Cell_2338_13 Cell_2338_14
## MTND2P28          78           559          811          705          384          447
## MTATP6P1        2053          1958         4922         4409         2610         3709
## NOC2L              1           125          126            0          487           66
## ISG15           2953          4938          580          523         2609            1
## CPSF3L             2            42           19            0           37           12
## MXRA8              0             0            0            0            0            0
## AURKAIP1         302           132           64          492           11          182
## CCNL2              0           235            0           84           13           11
## MRPL20           330           477          288          222           44          282
## SSU72            604           869         2046          158          530          272
```

```
dim(pollen_df)
```

```
## [1] 8686  300
```

Measurements of scRNA-Seq data are integer counts, this data does not have good properties so we perform a transformation on the data. The most commonly used transformation on RNA-Seq count data is $\log_2$. We will also transpose the data matrix to rows representing cells and columns representing genes. This is the data we can use to perform PCA.

```
# scRNA-Seq data transformation
pollen_mat <- log2(as.matrix(pollen_df) + 1)
# transpose the data
pollen_mat <- t(pollen_mat)
```

We will now use information that we read into the `label_df` variable to rename cells.

```
# Check which columns we have available
colnames(label_df)
```

```
## [1] "Cell_Identifier"        "Population"             "Cell_names"
## [4] "TrueLabel_CellLevel"    "Tissue_name"            "TrueLabel_TissuelLevel"
```

```
# rename rows
rownames(pollen_mat) <- label_df$Cell_names
```

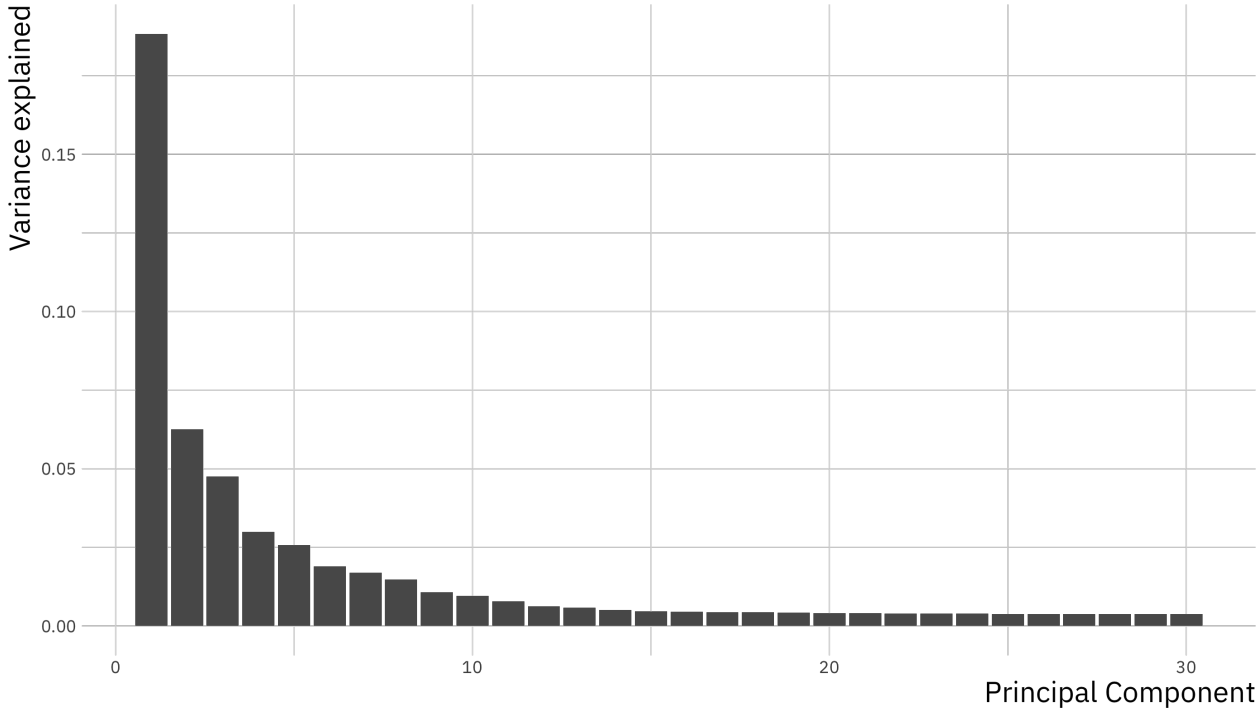
Next we perform PCA on the data and extract the proportion of variance explained by each component.

```
sc_pca <- prcomp(pollen_mat)
```

```
# variance is the square of the standard deviation
pr_var <- sc_pca$sdev^2

# compute the variance explained by each principal component
prop_var_exp <- pr_var / sum(pr_var)
```

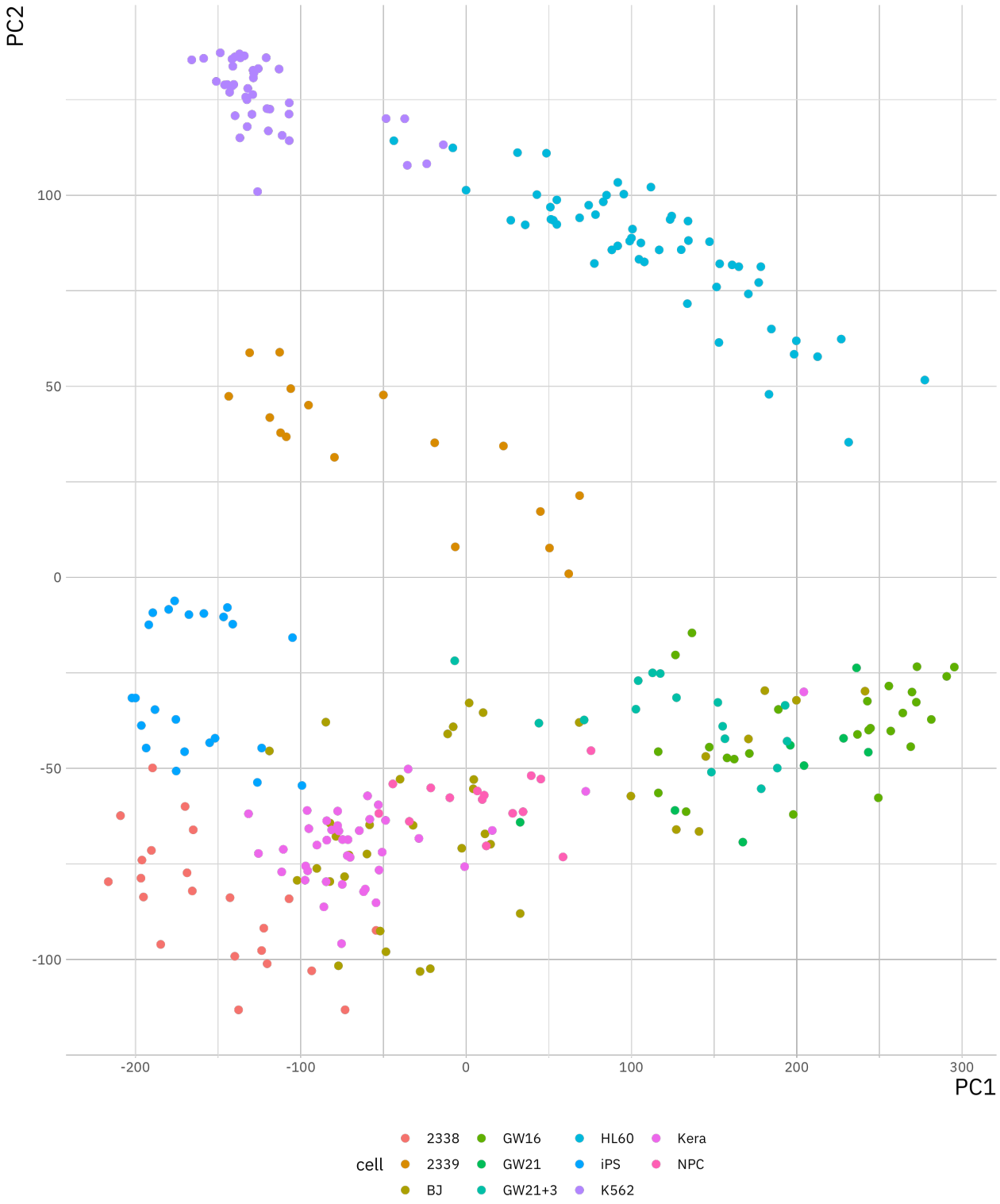
Think about the calculation and what exactly it means. We can visualise this

```
var_exp <- data.frame(variance = prop_var_exp, pc = 1:length(prop_var_exp))

ggplot(var_exp[1:30, ], aes(x = pc, y = variance)) +
    geom_bar(stat = "identity") +
    labs(x = "Principal Component",
         y  = "Variance explained")
```



We see that the first few principal components explain significant variance, but after about the PC10, there is very limited contribution. Next we will plot the data using the first two Principal components as before.

```
sc_pca_df <- data.frame(sc_pca$x, cell = rownames(sc_pca$x),
                        var_exp = prop_var_exp)

ggplot(sc_pca_df, aes(x = PC1, y = PC2, col = cell)) +
    geom_point(size = 2) +
    theme(legend.position = "bottom")
```



Why is it not useful to create biplot for this example?

13 Practical: Multiple regression

Previously we have only considered simple linear regression with one response variable and one feature. In this practical we will go through examples with multiple features:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \epsilon$$

For this practical we will use data that is already inbuilt in R or is part of the `MASS` package. The only thing we need to do to make the data available is load the `MASS` package.

13.1 Multiple regression

For this part we will use the inbuilt `trees` dataset containing `Volume`, `Girth` and `Height` data for 31 trees.

First we revisit linear regression on this example, recall the function to fit a linear model `lm()`. Consider `Volume` to be the response variable and `Girth` to be the covariate.

```
lr_fit <- lm(Volume ~ Girth, data = trees)
summary(lr_fit)
```

```
##
## Call:
## lm(formula = Volume ~ Girth, data = trees)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -8.065 -3.107  0.152  3.495  9.587
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -36.9435     3.3651  -10.98 7.62e-12 ***
## Girth         5.0659     0.2474   20.48  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 4.252 on 29 degrees of freedom
## Multiple R-squared:  0.9353, Adjusted R-squared:  0.9331
## F-statistic: 419.4 on 1 and 29 DF,  p-value: < 2.2e-16
```

We will now consider a linear regression example with multiple covariates, `Girth` as well as `Height`. In this case of course we know that they are related so we do expect both covariates to be significant.

```
mr_fit <- lm(Volume ~ Girth + Height, data = trees)


summary(mr_fit)
```

```
##
## Call:
## lm(formula = Volume ~ Girth + Height, data = trees)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -6.4065 -2.6493 -0.2876  2.2003  8.4847
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -57.9877     8.6382  -6.713 2.75e-07 ***
## Girth         4.7082     0.2643  17.816  < 2e-16 ***
## Height        0.3393     0.1302   2.607   0.0145 *
## ---
```

```
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 3.882 on 28 degrees of freedom
## Multiple R-squared:  0.948,  Adjusted R-squared:  0.9442
## F-statistic:   255 on 2 and 28 DF,  p-value: < 2.2e-16
```

Note, in the formula you only enter the covariates and not the regression coefficients or any information regarding the noise.

Let us now look at RSS values, we can calculate the RSS for the `lf_fit` object by using `sum(residuals(lr_fit)^2)`. We see that the RSS for LR = 524.3 and the RSS for MR = 421.92. Therefore the fit has improved but the regression coefficient for `Height` is very small and not significant.

One reason for this is that the in the relationship between `Volume`, `Girth`, and `Height` is not additive but rather `Girth` and `Height` are multiplied. Using the fact that $\log(a * b) = \log(a) + \log(b)$ we can consider the log-transformed data in a linear model.

```
mrl_fit <- lm(log(Volume) ~ log(Girth) + log(Height), data = trees)

summary(mrl_fit)
```

```
##
## Call:
## lm(formula = log(Volume) ~ log(Girth) + log(Height), data = trees)
##
## Residuals:
##       Min        1Q    Median        3Q       Max
## -0.168561 -0.048488  0.002431  0.063637  0.129223
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -6.63162    0.79979  -8.292 5.06e-09 ***
## log(Girth)   1.98265    0.07501  26.432  < 2e-16 ***
## log(Height)  1.11712    0.20444   5.464 7.81e-06 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.08139 on 28 degrees of freedom
## Multiple R-squared:  0.9777, Adjusted R-squared:  0.9761
## F-statistic: 613.2 on 2 and 28 DF,  p-value: < 2.2e-16
```

Now we see that the regression coefficient is large and both covariates are significant. This shows that we need to ensure we understand the relationship between covariates before we construct our model.

13.2 Categorical covariates

Recall from the lecture that covariates don't need to be numerical but can also be *categorical*. We will now explore regression with a categorical variable. Load a new dataset which is included in the `MASS` package, you won't be able to load this dataset if package isn't installed. Load the dataset explore what the data looks like.

```
library(MASS)
data("birthwt")

head(birthwt)
```

```
##    low age lwt race smoke ptl ht ui ftv  bwt
## 85   0  19 182    2     0   0  0  1   0 2523
```

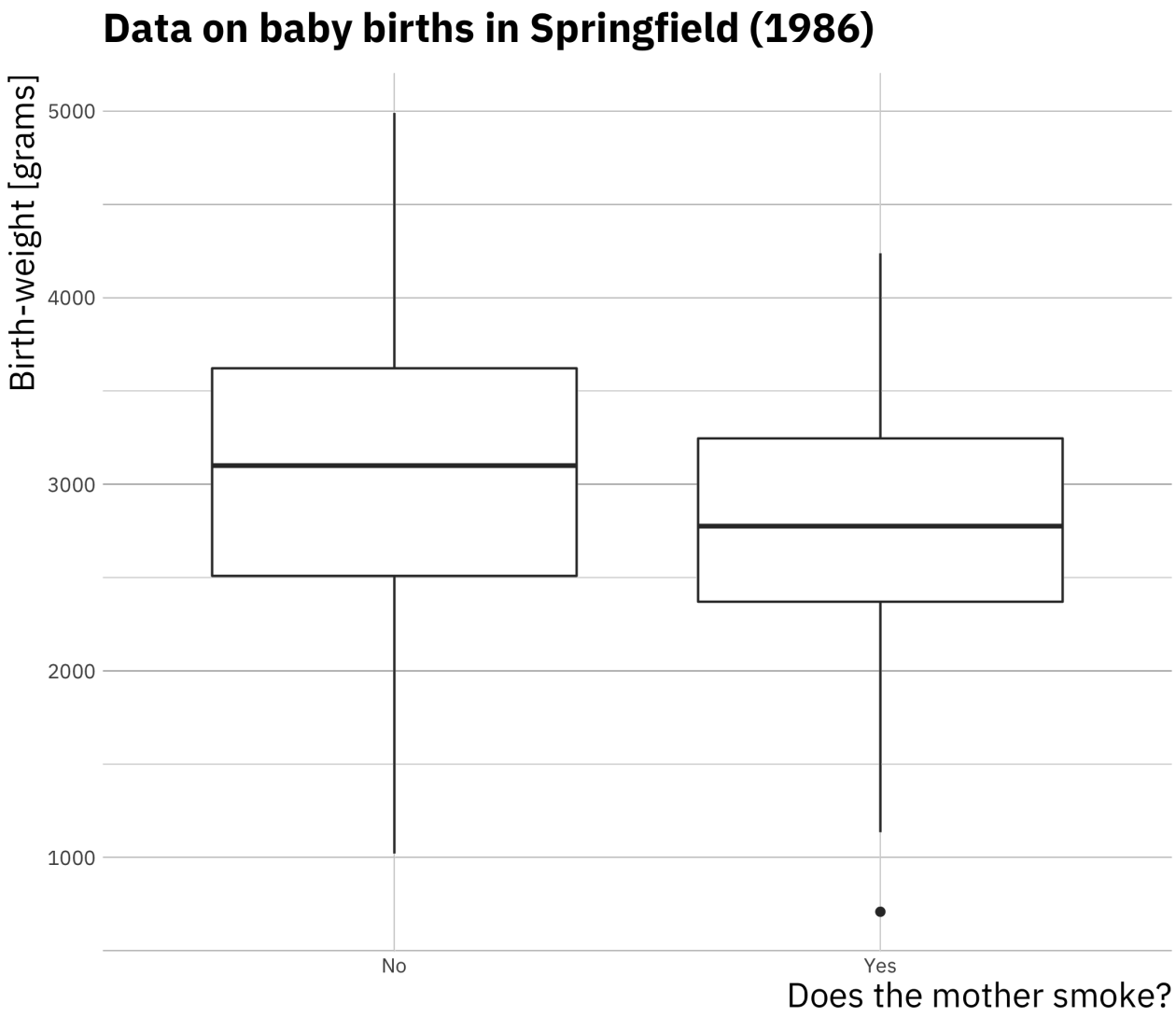
```
## 86   0  33 155    3     0   0  0  0   3 2551
## 87   0  20 105    1     1   0  0  0   1 2557
## 88   0  21 108    1     1   0  0  1   2 2594
## 89   0  18 107    1     1   0  0  1   0 2600
## 91   0  21 124    3     0   0  0  0   0 2622
```

```
summary(birthwt)
```

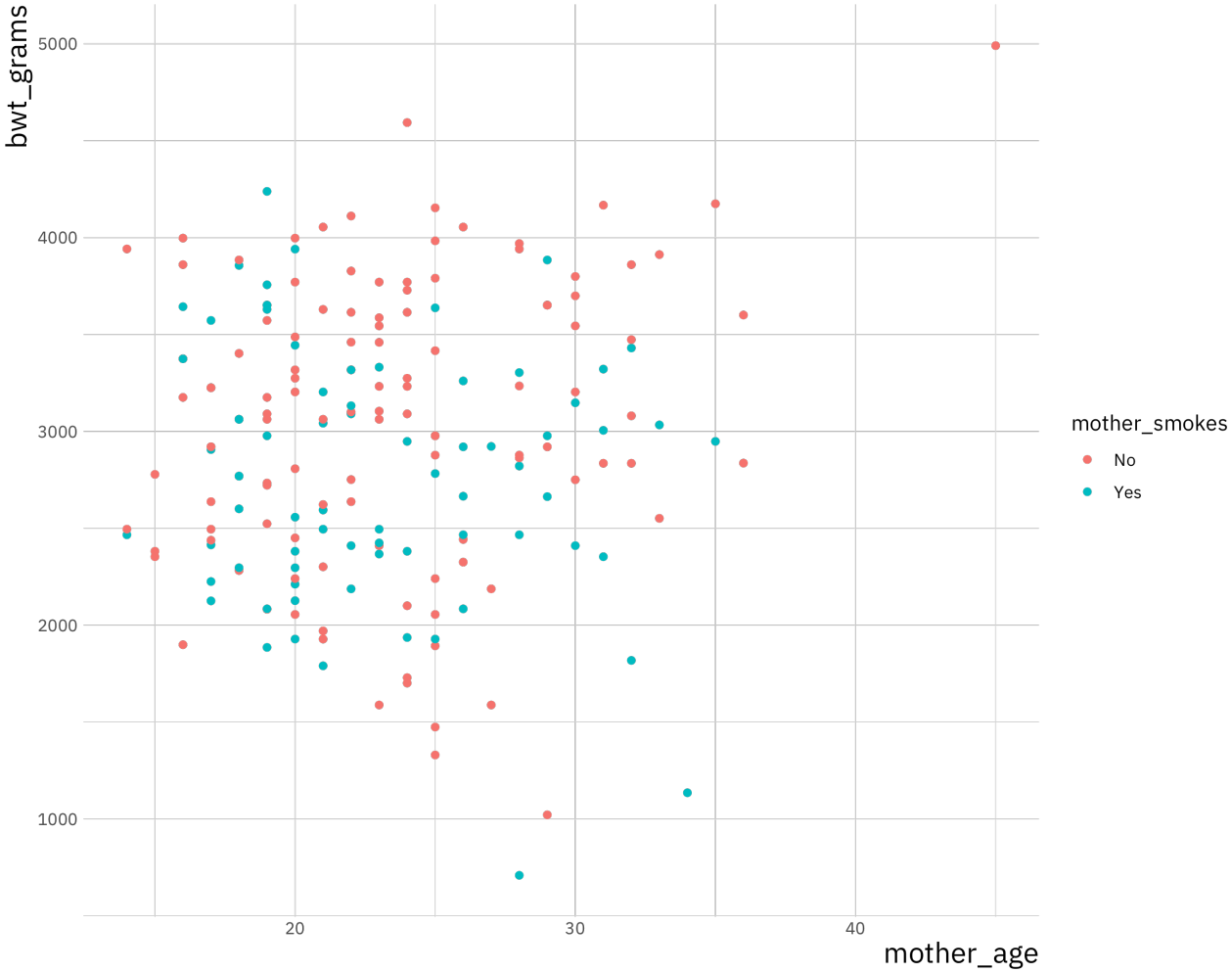
```
##       low              age             lwt             race           smoke
##  Min.   :0.0000   Min.   :14.00   Min.   : 80.0   Min.   :1.000   Min.   :0.0000
##  1st Qu.:0.0000   1st Qu.:19.00   1st Qu.:110.0   1st Qu.:1.000   1st Qu.:0.0000
##  Median :0.0000   Median :23.00   Median :121.0   Median :1.000   Median :0.0000
##  Mean   :0.3122   Mean   :23.24   Mean   :129.8   Mean   :1.847   Mean   :0.3915
##  3rd Qu.:1.0000   3rd Qu.:26.00   3rd Qu.:140.0   3rd Qu.:3.000   3rd Qu.:1.0000
##  Max.   :1.0000   Max.   :45.00   Max.   :250.0   Max.   :3.000   Max.   :1.0000
##       ptl               ht                ui              ftv              bwt
##  Min.   :0.0000   Min.   :0.00000   Min.   :0.0000   Min.   :0.0000   Min.   : 709
##  1st Qu.:0.0000   1st Qu.:0.00000   1st Qu.:0.0000   1st Qu.:0.0000   1st Qu.:2414
##  Median :0.0000   Median :0.00000   Median :0.0000   Median :0.0000   Median :2977
##  Mean   :0.1958   Mean   :0.06349   Mean   :0.1481   Mean   :0.7937   Mean   :2945
##  3rd Qu.:0.0000   3rd Qu.:0.00000   3rd Qu.:0.0000   3rd Qu.:1.0000   3rd Qu.:3487
##  Max.   :3.0000   Max.   :1.00000   Max.   :1.0000   Max.   :6.0000   Max.   :4990
```

We will give the data more interpretable names and generally cleanup the data a little bit.

```
# rename columns
colnames(birthwt) <- c("bwt_below_2500", "mother_age", "mother_weight", "race",
                       "mother_smokes", "previous_prem_labor", "hypertension",
                       "uterine_irr", "physician_visits", "bwt_grams")

birthwt$race <- factor(c("white", "black", "other")[birthwt$race])
birthwt$mother_smokes <- factor(c("No", "Yes")[birthwt$mother_smokes + 1])
birthwt$uterine_irr <- factor(c("No", "Yes")[birthwt$uterine_irr + 1])
birthwt$hypertension <- factor(c("No", "Yes")[birthwt$hypertension + 1])

ggplot(birthwt, aes(x = mother_smokes, y = bwt_grams)) +
    geom_boxplot() +
    labs(title = "Data on baby births in Springfield (1986)",
         x = "Does the mother smoke?",
         y = "Birth-weight [grams]")
```




```
ggplot(birthwt, aes(x = mother_age, y = bwt_grams, col = mother_smokes)) +
    geom_point()
```



Now we perform linear regression using the categorical variable, it is no different than performing linear regression on numeric data. The difference is in interpretation.

```
bwt_fit <- lm(bwt_grams ~ mother_smokes, data = birthwt)

summary(bwt_fit)
```

```
##
## Call:
## lm(formula = bwt_grams ~ mother_smokes, data = birthwt)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -2062.9  -475.9    34.3   545.1  1934.3
##
## Coefficients:
##                  Estimate Std. Error t value Pr(>|t|)
## (Intercept)       3055.70      66.93  45.653  < 2e-16 ***
## mother_smokesYes  -283.78     106.97  -2.653  0.00867 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 717.8 on 187 degrees of freedom
```

```
## Multiple R-squared:  0.03627,    Adjusted R-squared:  0.03112
## F-statistic: 7.038 on 1 and 187 DF,  p-value: 0.008667
```

When you put a categorical variable in the formula for `lm` as in this case `bwt_grams ~ mother_smokes` where we have two levels in the categorical variable. If we consider this model as $y = \beta_0 + \beta_1 x + \epsilon$ The coefficients in the model can be interpreted as follows:

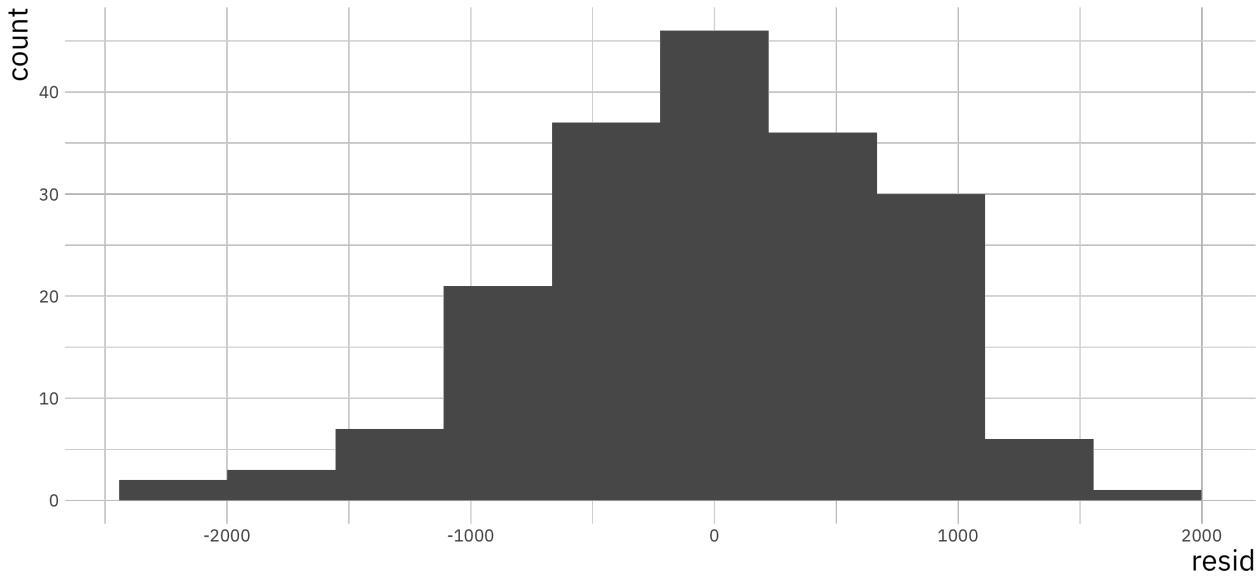
- β_0 is average birth weight where the mother was a non smoker
- $\beta_0 + \beta_1$ is the average birth weight where the mother is a smoker
- β_1 is the average difference in birth weight for babies between mother that were smokers and mothers that were non smokers.

Categorical variables can also have more than two levels and in those cases each additional level can be interpreted in the same way.

13.3 Residuals

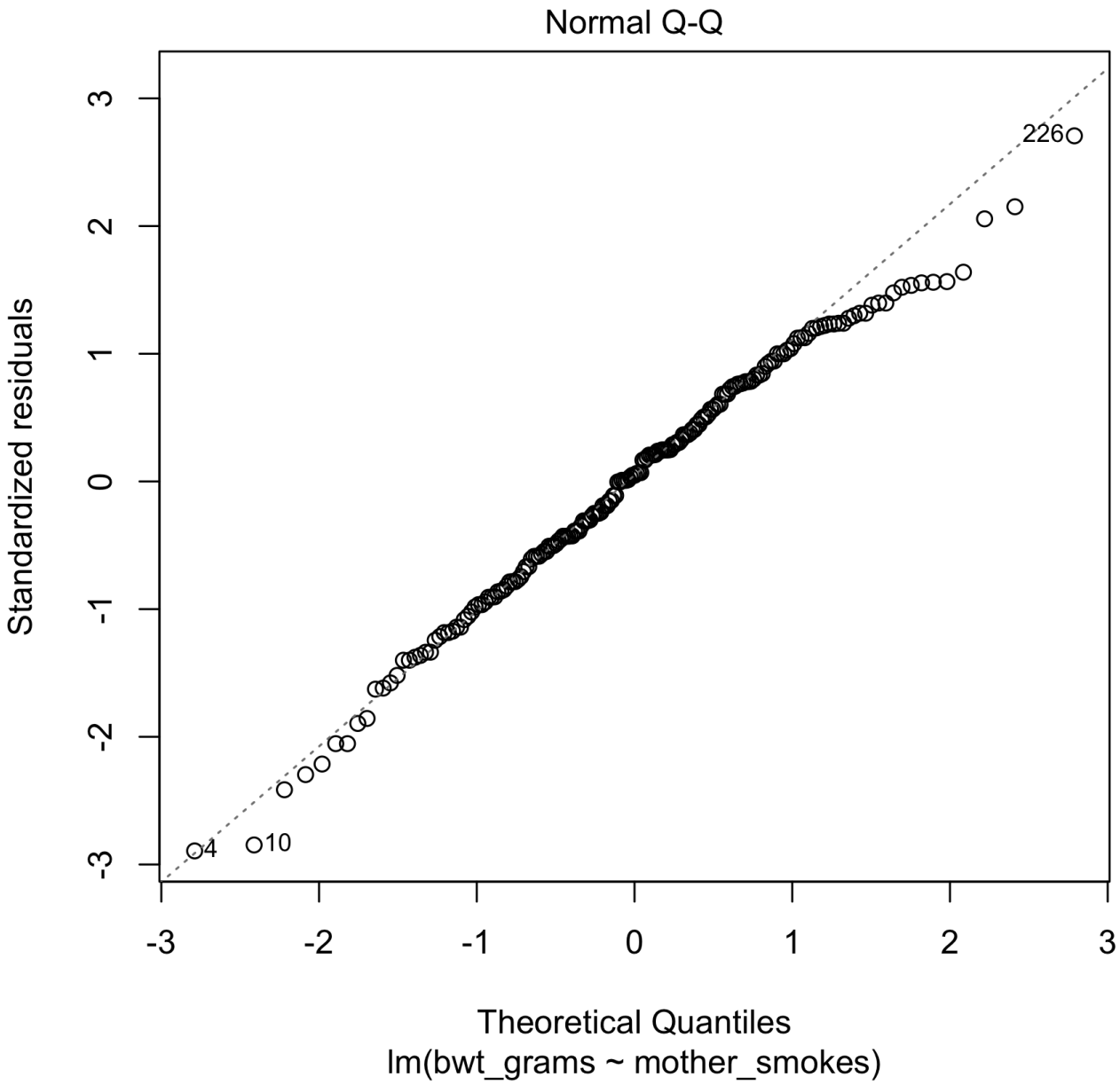
Recall from the lectures the residuals are the differences between the observed data y and the fitted values $\hat{y}$. One of the assumptions we make in the simple linear regression model is that the residuals should be normally distributed. To extract residuals from an `lm` object we will use the `residuals()` function.

```
residuals_df <-
data.frame(resid = residuals(bwt_fit))

ggplot(residuals_df, aes(x = resid)) +
    geom_histogram(bins = 10)
```



Even if we consider that these residuals look like they are normally distributed we need to get better understanding of this we will use the Q-Q Plot. You can take a look at the wiki to get a better understanding (Q–Q plot - Wikipedia). In simple terms if the residuals are normally distributed we expect them to be on the diagonal straight line on a Q-Q plot. The simplest way to get such a plot is using the `plot()` function and specifically for an `lm` object it has an option `which =` that takes a numeric value depending on which plot you want to plot.

```
plot(bwt_fit, which = 2)
```



As we can see in this example the residuals are very close to normal with some outliers especially towards larger values of the residual. This would indicate that the model as it stands does not fulfill that assumption fully but comes close.

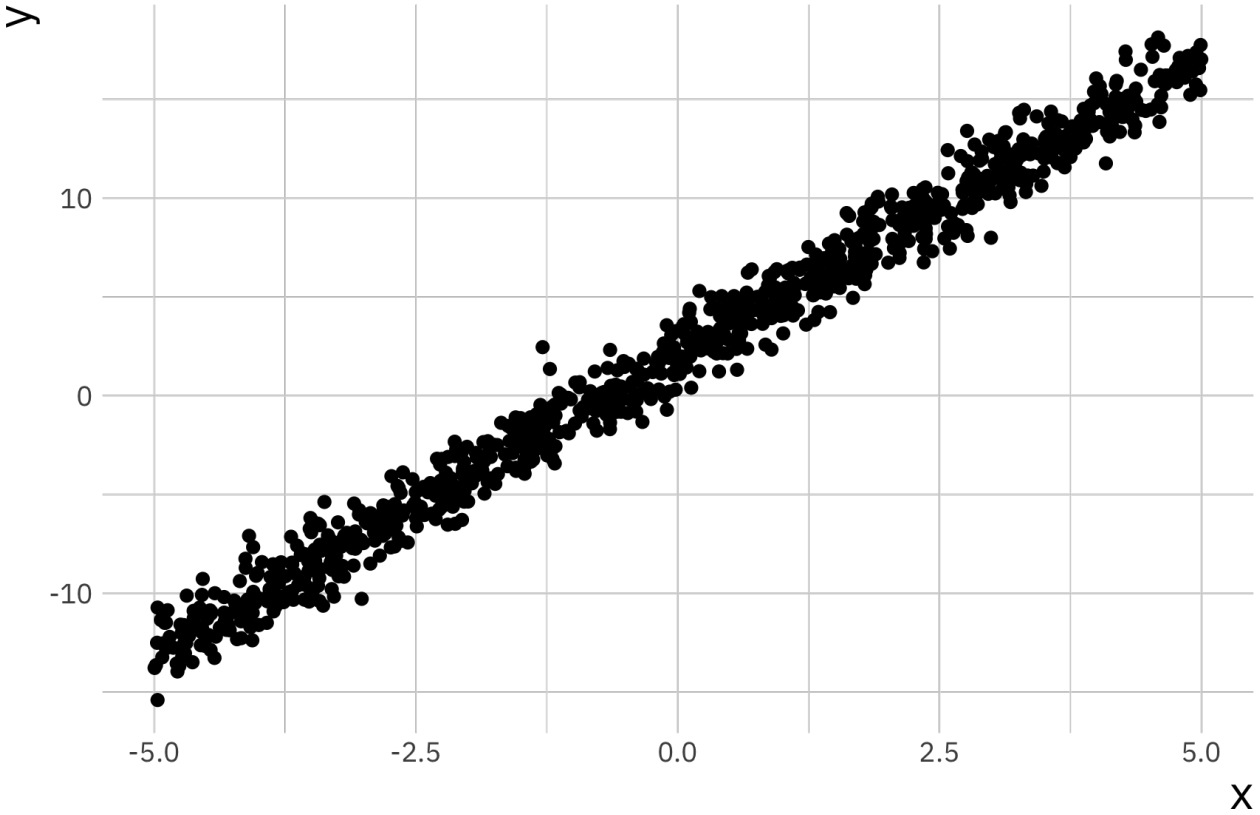
13.4 Gradient descent algorithm (+)

Finally, in todays practical we will implement the *gradient descent algorithm* which we discussed in the lecture.

For simplicity we will only consider the case with one covariate. In this section we will use simulated data and compare the results with `lm()`. The model we will simulate from is:

$$y = 2 + 3x + \epsilon$$

```r
# setting seed to be able to reproduce the simulation
set.seed(200)

# number of samples
n_sample <- 1000

# We sample x values from a uniform distribution in the range [-5, 5]
x <- runif(n_sample, -5, 5)
# Next we compute y
y <- 3 * x + 2 + rnorm(n = n_sample, mean = 0, sd = 1)

sim_df <- data.frame(x = x, y = y)

ggplot(sim_df, aes(x = x, y = y)) +
    geom_point()
```



Recall that in gradient descent we want to minimise the Mean Squared Error ($J(\beta)$) which is the cost function. The first step is to write this cost function in R. For simplicity we will use matrix multiplication, which in R is implemented as `%*%`. (*Note*, to get help on these function with special characters you can't simply run the command `?%*%` instead you have to put it in quotes `?"%*%"`.)

```r
cost_fn <- function(X, y, coef) {
    sum( (X %*% coef - y)^2 ) / (2*length(y))
}
```

To perform an optimisation we will have to initialise parameters, in general optimisation algorithms won't always produce the same results for all choices of initialisations.

```
# First we set alpha and the number of iterations we will perform
alpha <- 0.2
num_iters <- 100

# next we will initialise regression coefficients
coef <- matrix(c(0,0), nrow=2)
X <- cbind(1, matrix(x))
res <- vector("list", num_iters)
```

We now write a for loop to compute the optimisation, where we store the full history of the opmtimisation.

```
for (i in 1:num_iters) {
  error <- (X %*% coef - y)
  delta <- t(X) %*% error / length(y)
  coef <- coef - alpha * delta
  res_df <- data.frame(itr = i , cost = cost_fn(X, y, coef),
                       b0 = coef[1], b1 = coef[2])

  res[[i]] <- res_df
}
```

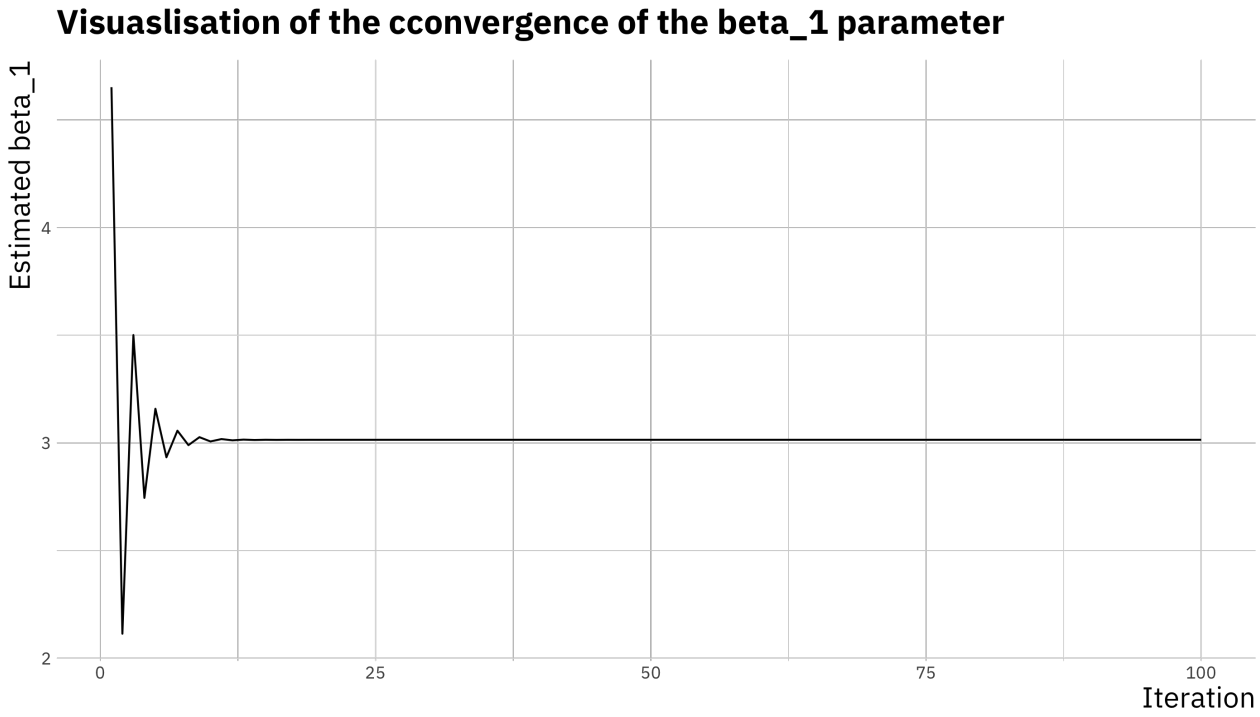
We created a list to store results `res` it is possible to combine all results into a simple `data.frame` using the `bind_rows()` function from the `dplyr` package. If we look at the final values in the resulting variable we will

```
library(dplyr)
res_df <- bind_rows(res)
tail(res_df)
```

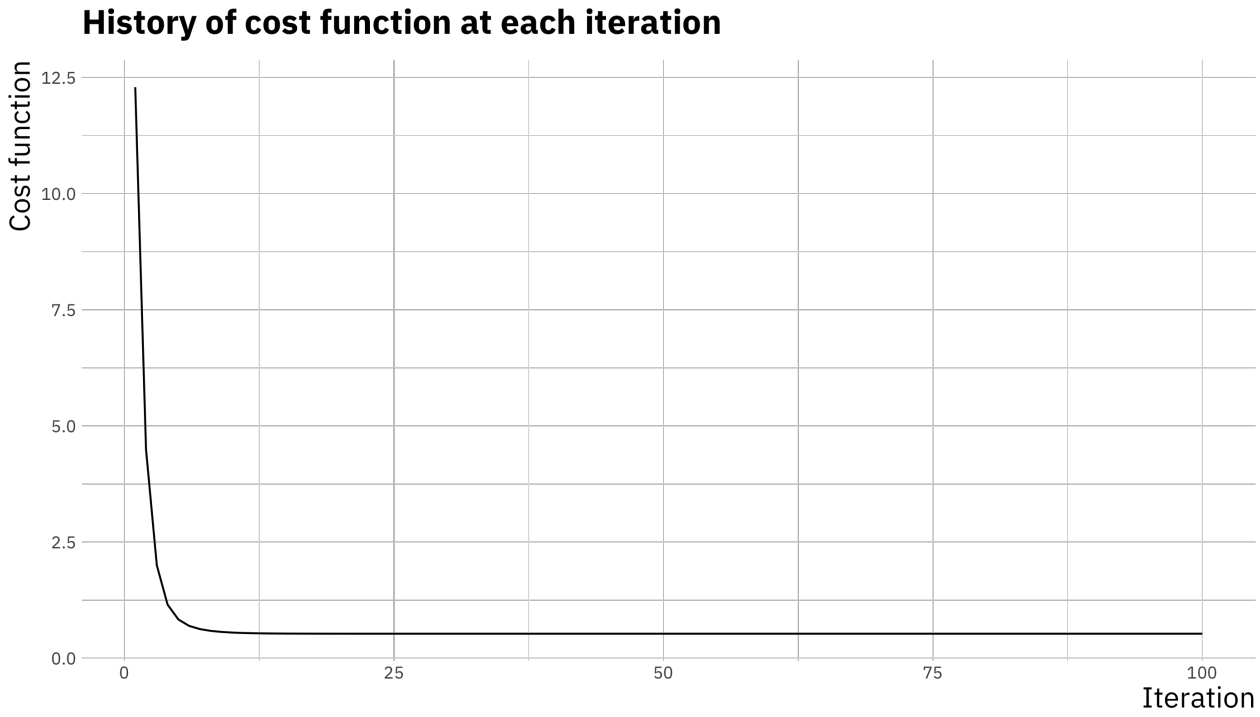
```
##     itr      cost       b0       b1
## 95   95 0.5275707 2.034285 3.014512
## 96   96 0.5275707 2.034285 3.014512
## 97   97 0.5275707 2.034285 3.014512
## 98   98 0.5275707 2.034285 3.014512
## 99   99 0.5275707 2.034285 3.014512
## 100 100 0.5275707 2.034285 3.014512
```

We can see that $\beta_0 = 2$ and $\beta_1 = 3$ are reproduced faithfully. There are a few ways to visualise the optimisation. We can look at the convergence of the parameters, the cost function itself or even the estimated y at each step of the optimisation.

```
ggplot(res_df, aes(x = itr, y = b1)) +
    geom_line() +
    labs(x = "Iteration",
         y = "Estimated beta_1",
         title = "Visuaslisation of the cconvergence of the beta_1 parameter")
```

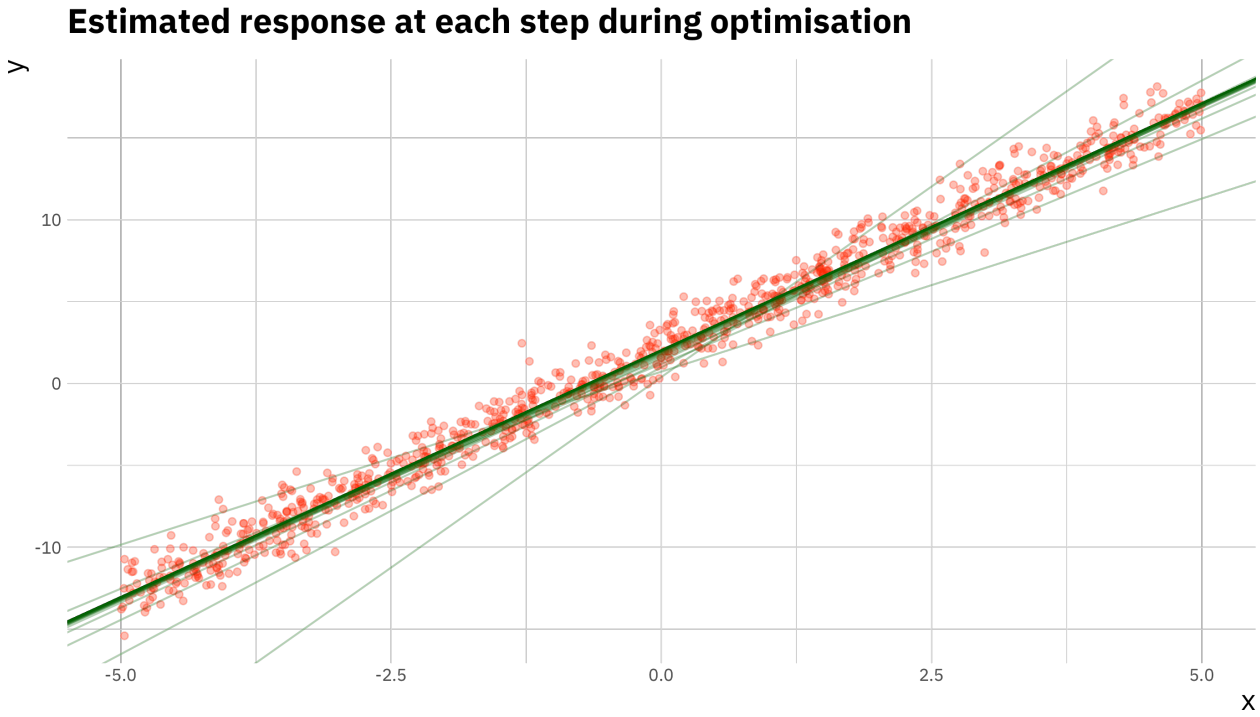



```
ggplot(res_df, aes(x = itr, y = cost)) +
    geom_line() +
    labs(x = "Iteration",
         y = "Cost function",
         title = "History of cost function at each iteration")
```




```
ggplot(sim_df, aes(x = x, y = y)) +
    geom_point(color = "red", alpha = 0.3) +
    geom_abline(data = res_df, aes(intercept = b0, slope = b1),
```

```
              alpha = 0.3, col = "darkgreen", size = 0.5) +
  labs(x = "x", y = "y",
       title = "Estimated response at each step during optimisation")
```



Now compare these results to the ones obtained by fitting a linear model in R using the function `lm()`, how different are the results. Try to reproduce these plots with $\alpha = (0.02, 0.1, 0.5)$, and different number of iterations in the optimisation and compare the estimated $\hat{\beta}_0$, and $\hat{\beta}_1$ to the values you use during the simulation step. This will give you an idea how important the right choice of these two parameters is.

14 Practical: Generalised linear models

In a genome-wide association study, we perform an experiment where we select n individuals with a disease (cases) and n individuals without the diseases (controls) and look for genetic differences between these two groups. In particular, we are interested in specific genetic variants (SNPs) that might induce some predisposition towards the disease.

Suppose I observe the following genotypes for a SNP in 4,000 individuals (2,000 cases, 2,000 controls):

- Genotypes: `AA Aa aa`
- Controls: `3 209 1788`
- Cases: `83 621 1296`

The cases seem to have relatively more `A` alleles than the controls. This might make us suspect that having `A` alleles at this SNP is associated with the disease.

14.1 Data

For this practical we will use two files you can use these links to download them:

- `gwas-cc-ex1.Rdata` (download)
- `gwas-cc-ex2.Rdata` (download)
- `nb_data.Rdata` (download)

14.2 Detecting SNP associations

We have seen in lectures that we can do statistical tests for this type of contingency table using Chi Squared Tests. Let's load example data set and and prepare

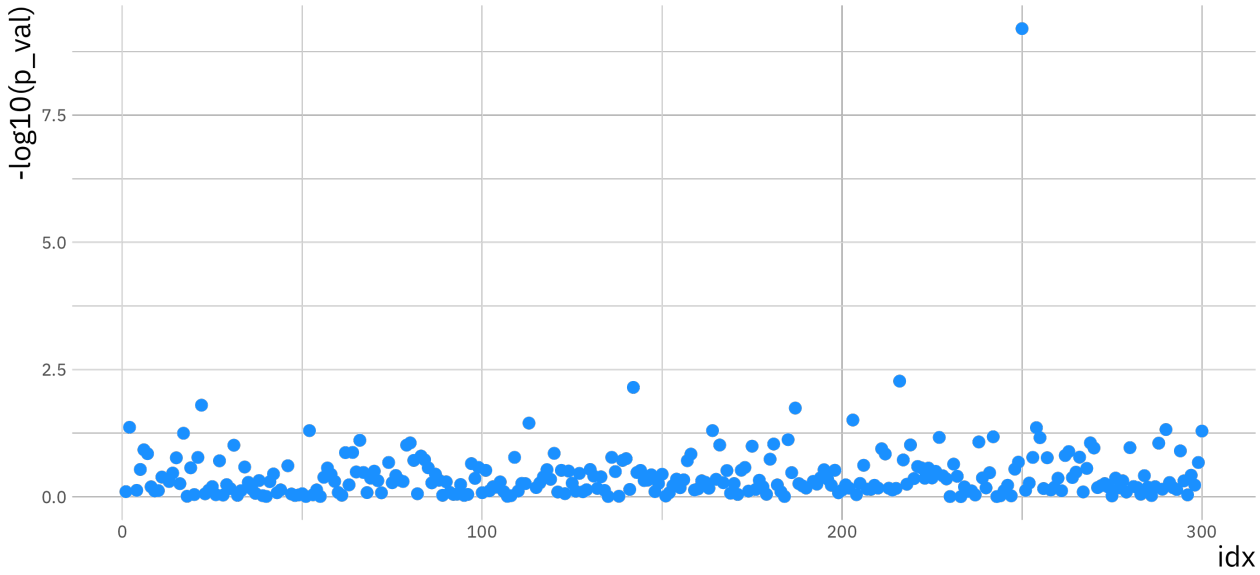
```
library(ggplot2) # for plots later
load("gwas-cc-ex1.Rdata")

# how many individuals are there
n <- length(y)
# How many SNPs do we have data for
p <- nrow(X)

# samples that are controls are encoded as 0 in y
control <- which(y == 0)
# disease cases are encoded as 1 in y
cases <- which(y == 1)
```

Now we need to write a loop that scans through all, p, SNPs:

```
# create a vector where p-values will be stored
p_vals <- rep(0, p)

# Loop over SNPs
for (i_p in 1:p) {
    # 1. obtain genotype counts
    counts <- matrix(0, nrow = 2, ncol = 3)
    counts[1, ] <- c(sum(X[i_p, control] == 0),
                     sum(X[i_p, control] == 1),
                     sum(X[i_p, control] == 2))

    counts[2, ] <- c(sum(X[i_p, cases] == 0),
                     sum(X[i_p, cases] == 1),
                     sum(X[i_p, cases] == 2))

    # 2. expected probability of AA
    # (assuming no dependence on case/control status)
    expected_pr_AA <- sum(counts[, 1]) / n
    # expected probability of Aa
    expected_pr_Aa <- sum(counts[, 2]) / n
    # expected probability of aa
    expected_pr_aa <- sum(counts[, 3]) / n

    expected_probs <- c(expected_pr_AA, expected_pr_Aa, expected_pr_aa )

    # 3. do my chi-squared test
    out <- chisq.test(counts, p = expected_probs)
    # extract p value of test and store
    p_vals[i_p] <- out$p.value
}
```

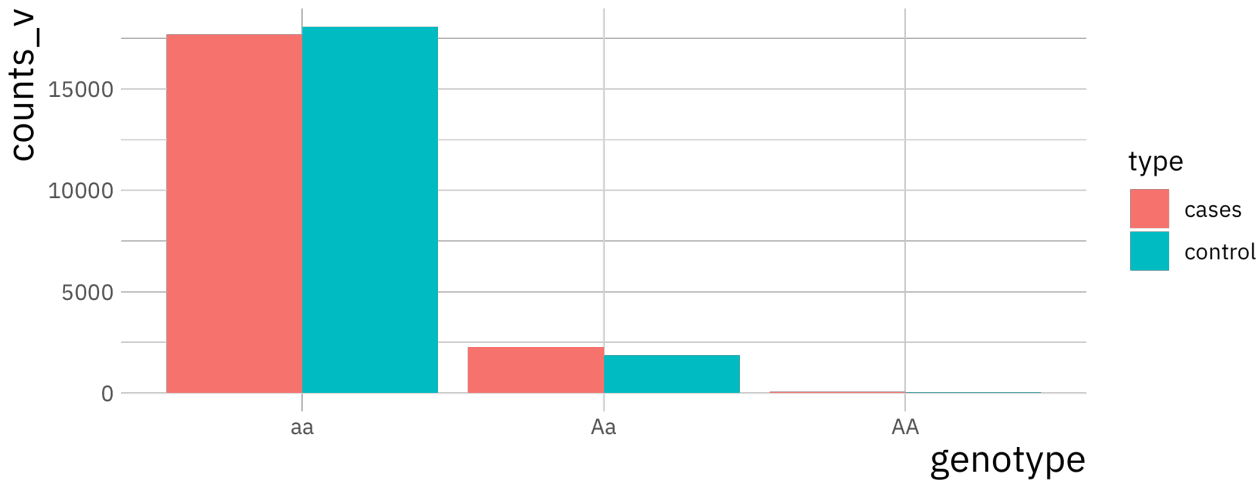
We went through each SNP (rows in matrix `X`), extracted the counts of each genotype (marked `1.` in code) for cases and controls, then we compute expected probability (marked `2.` in code). Finally, we perform a chi-squared contingency table test comparing those observed counts to expected probabilities assuming that genotype is not related to disease status (marked `3.` in code).

```
p_val_df <- data.frame(p_val = p_vals, idx = 1:p)

ggplot(p_val_df, aes(x = idx, y = -log10(p_val))) +
    geom_point(size = 2.5, col = "dodgerblue1")
```



This plot is knows as a *Manhattan* plot. One SNP (`i_p = 250`) will pop out as being highly associated with the disease process. Look at the genotype counts (or MAF) for this SNP in the cases and controls to see for yourself that there is large difference in the distribution of genotypes (or MAF).

```
i_p <- 250
counts_v <- c(sum(X[i_p, control] == 0), sum(X[i_p, control] == 1),
              sum(X[i_p, control] == 2), sum(X[i_p, cases] == 0),
              sum(X[i_p, cases] == 1), sum(X[i_p, cases] == 2))

snp_procs <- data.frame(counts_v, type = rep(c("control", "cases"), each = 3),
           genotype = rep(c("AA", "Aa", "aa"), 2))

ggplot(snp_procs, aes(x = genotype, y = counts_v, fill = type)) +
    geom_bar(stat = "identity", position = "dodge")
```



14.3 GWAS and logistic regression

Now lets approach this problem using Generalised Linear Models. Lets load a data set containing genotypes in `X` and case-control status in `y`:

```
# load an example data set (genotypes in X, case-control (1/0) status in y)
load("gwas-cc-ex2.Rdata")

n <- length(y) # how many individuals do we have in total?
p <- nrow(X) # how many SNPs do I have data for?
```

For each of the `p` SNPs we are going to call the `R` GLM function `glm` using the `binomial` family option with the `logit` link function because my outcomes are binary. We will then extract the p-value associated with the regression coefficient for the genotype. This is obtained from applying a hypothesis test (the *Wald Test*) on whether the coefficient has a null value zero.

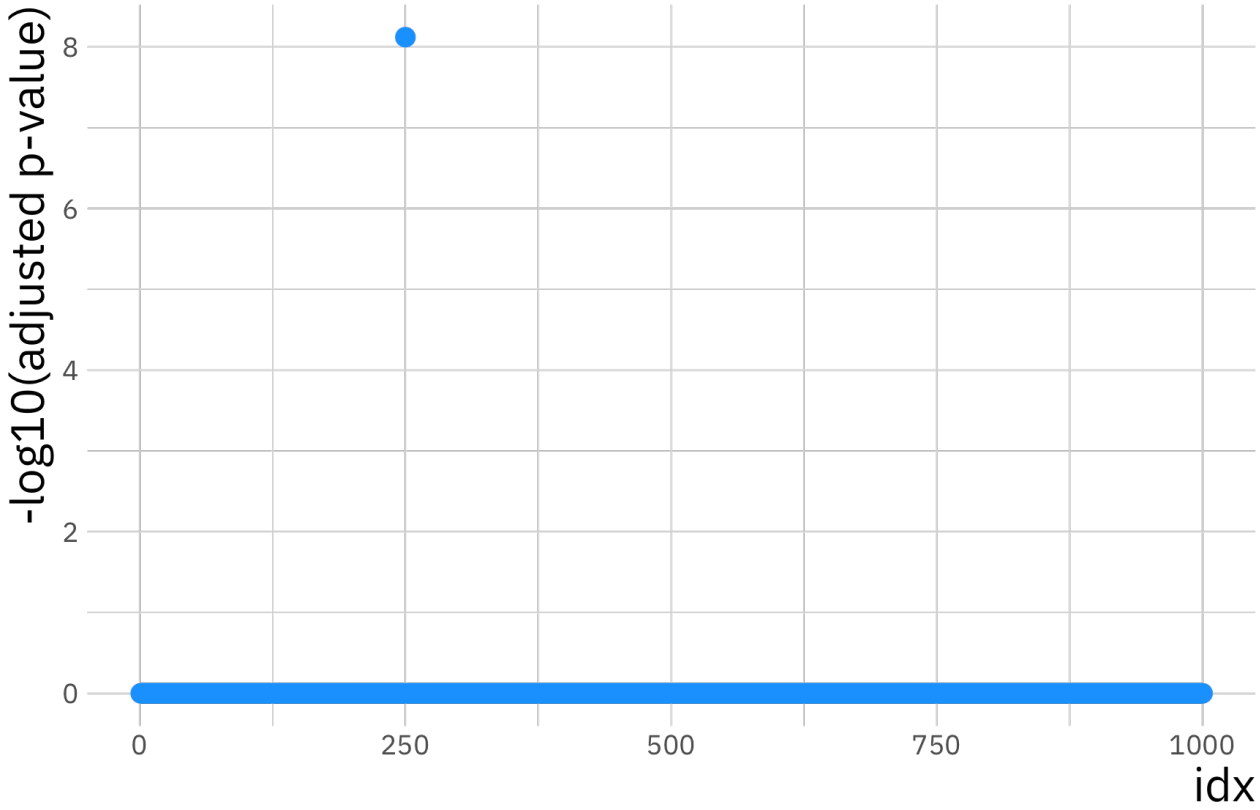
```
p_vals <- rep(0, p)
for ( j in 1:p ) {
  snp_data <- data.frame(y = y, x = X[j, ])
  glm.fit <- glm(y ~ x, family = binomial(link = logit), data = snp_data  )
  p_vals[j] <- summary(glm.fit)$coefficients[2,4]
}
```

We are testing 1,000 SNPs so lets use Bonferroni correction to adjust these p-values to take into account multiple testing:

```
adj_p_vals <- p.adjust(p_vals, "bonferroni")
```

Lets use the adjusted -log10 p-values to plot a Manhattan plot:

```
# create data.frame with p-values for plotting with ggplot
p_val_df <- data.frame(p_val = adj_p_vals, idx = 1:p)

ggplot(p_val_df, aes(x = idx, y = -log10(p_val))) +
    geom_point(size = 2.5, col = "dodgerblue1") +
    labs(y = "-log10(adjusted p-value)")
```



You should see a single SNP showing a strong association with disease status.

14.4 Negative binomial and Poisson regression

Molecular biologists study the behavior of protein expression in normal and cancerous tissues. The hypothesis is that the total number of over-expressed proteins depends on the histopathological-derived tumor subtype and an immune cell contexture measure.

You are provided with data on 314 tumours in the file `nb_data.Rdata`. The file contains one `data frame` with the following variables:

- **`overexpressed_proteins`**: response variable of interest.
- **`immunoscore`**: gives a standardized measure of immune cell contexture.
- **`tumor_subtype`**: three-level nominal variable indicating the histopathological sub-type of the tumour. The three levels are Unstable, Stable, and Complex

Let's load some prerequisite R libraries and the data to produce some summary statistics (*install if required using `install.package()` command*):

```
# required libraries
library(MASS)
library(foreign)


load("nb_data.Rdata")

# print summary statistics to Console
summary(dat)
```

```
##    sample_id      gender     immunoscore    overexpressed_proteins  tumor_subtype
##  1001   :  1   female:160   Min.   : 1.00   Min.   : 0.000         Complex : 40
```

```
##  1002   :  1   male  :154   1st Qu.:28.00   1st Qu.: 1.000         Unstable:167
##  1003   :  1                 Median :48.00   Median : 4.000         Stable  :107
##  1004   :  1                 Mean   :48.27   Mean   : 5.955
##  1005   :  1                 3rd Qu.:70.00   3rd Qu.: 8.000
##  1006   :  1                 Max.   :99.00   Max.   :35.000
##  (Other):308
```

14.4.1 Count-based GLMs

The `overexpressed_proteins` measurements are counts. This implies we should use a Poisson based GLM.

In Poisson regression models, the conditional variance is by definition equal to the conditional mean. This can be limiting.

Negative binomial regression can be used for over-dispersed count data, that is when the conditional variance exceeds the conditional mean.

It can be considered as a generalization of Poisson regression since it has the same mean structure as Poisson regression but it has an extra parameter to model the over-dispersion. If the conditional distribution of the outcome variable is over-dispersed, the confidence intervals for the Poisson regression are likely to be narrower as compared to those from a Negative Binomial regression model.

In the following we will try both models to see which fits best.

14.4.2 Fitting a GLM

Below we use the `glm.nb` function from the `MASS` package to estimate a negative binomial regression. The use of the function is similar to that of `lm` for linear models but with the additional requirement of a link function. As count data is always positive, a log link function is useful here.

```
glm_1 <- glm.nb(overexpressed_proteins ~ immunoscore + tumor_subtype + gender, data = dat, link=log)

# print summary statistics of glm.nb output object to Console
summary(glm_1)
```

```
##
## Call:
## glm.nb(formula = overexpressed_proteins ~ immunoscore + tumor_subtype +
##     gender, data = dat, link = log, init.theta = 1.047288915)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -2.1567  -1.0761  -0.3810   0.2856   2.7235
##
## Coefficients:
##                         Estimate Std. Error z value Pr(>|z|)
## (Intercept)             2.707484   0.204275  13.254  < 2e-16 ***
## immunoscore            -0.006236   0.002492  -2.502   0.0124 *
## tumor_subtypeUnstable -0.424540   0.181725  -2.336   0.0195 *
## tumor_subtypeStable   -1.252615   0.199699  -6.273 3.55e-10 ***
## gendermale             -0.211086   0.121989  -1.730   0.0836 .
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for Negative Binomial(1.0473) family taken to be 1)
##
##     Null deviance: 431.67  on 313  degrees of freedom
```

```
## Residual deviance: 358.87  on 309  degrees of freedom
## AIC: 1740.3
##
## Number of Fisher Scoring iterations: 1
##
##
##               Theta:  1.047
##           Std. Err.:  0.108
##
##  2 x log-likelihood:  -1728.307
```

R first displays the call and the deviance residuals. Next, we see the regression coefficients for each of the variables, along with standard errors, z-scores, and p-values. The variable `immunoscore` has a coefficient of -0.006, which is statistically significant at the 5% level (`Pr(>|z|) = 0.0124*`). This means that for each one-unit increase in `immunoscore`, the expected log count of the number of `overexpressed_proteins` decreases by 0.006.

The indicator variable shown as `tumor_subtypeUnstable` is the expected difference in log count between group Unstable and the reference group (`tumor_subtype=Complex`). The expected log count for the Unstable type is approximately 0.4 lower than the expected log count for the Complex type.

The indicator variable for Stable type is the expected difference in log count between the Stable type and the reference Complex group. The expected log count for Stable is approximately 1.2 lower than the expected log count for the Complex type.

14.4.3 Comparing nested models

To determine if `tumor_subtype` itself, overall, is statistically significant, we can compare a model with and without `tumor_subtype`. The reason it is important to fit separate models is that, unless we do, the overdispersion parameter is held constant and it would not be a fair comparison.

```
glm_2 <- glm.nb(overexpressed_proteins ~ immunoscore + gender, data = dat, link = log)
```

We use the `anova` function to compare models using a likelihood ratio test (LRT):

```
anova(glm_1, glm_2, test = "LRT")
```

```
## Warning in anova.negbin(glm_1, glm_2, test = "LRT"): only Chi-squared LR tests are implemented
```

```
## Likelihood ratio tests of Negative Binomial Models
##
## Response: overexpressed_proteins
##                                    Model     theta Resid. df    2 x log-lik.   Test    df LR stat.
## 1                 immunoscore + gender 0.8705939       311       -1772.074
## 2 immunoscore + tumor_subtype + gender 1.0472889       309       -1728.307 1 vs 2     2 43.76737
##         Pr(Chi)
## 1
## 2 3.133546e-10
```

The two degree-of-freedom chi-square test indicates that `tumor_subtype` is a statistically significant predictor of `overexpressed_proteins` (`Pr(Chi) = 3.133546e-10`).

The `anova` function performs a form of *LRT*. It computes the likelihood of the data under the two models being compared and then uses the ration of these likelihood values as a test statistic.

Theory tells us that, for large samples sizes, the (2x) log likelihood ratio has a chi-squared distribution with degrees of freedom equal to the difference in the number of free parameters between the two models being compared. The LRT only applies to *nested models*, i.e. a pair of models where one is a less complex subset of the other.

14.5 Negative-Binomial vs Poisson GLMs

Negative binomial models assume the conditional means are not equal to the conditional variances. This inequality is captured by estimating a dispersion parameter (not shown in the output) that is held constant in a Poisson model. Thus, the Poisson model is actually nested in the negative binomial model. We can then use a likelihood ratio test to compare these two models.

To do this, we will first fit a GLM Poisson regression:

```
glm_3 <- glm(overexpressed_proteins ~ immunoscore + tumor_subtype + gender, family = "poisson", data = d
```

Now, lets do our likelihood ratio test, we can extract the log-likelihood using `logLik()` and then use `pchisq()` to extract the probability of getting a statistic at least as extreme as this:

```
pchisq(2 * (logLik(glm_1) - logLik(glm_3)), df = 1, lower.tail = FALSE)
```

```
## 'log Lik.' 3.847622e-198 (df=6)
```

Note that the more complex model goes first because more complex models always have the larger likelihood.

In this example the associated chi-squared value estimated from `2*(logLik(m1) - logLik(m3))` is around 900 with one degree of freedom. This strongly suggests the negative binomial model, estimating the dispersion parameter, is more appropriate than the Poisson model.

14.6 Further understanding the model (OPTIONAL)

For assistance in further understanding the model, we can look at predicted counts for various levels of our predictors. Below we create new datasets with values of `immunoscore` and `tumor_subtype` and then use the predict command to calculate the predicted number of overexpressed proteins

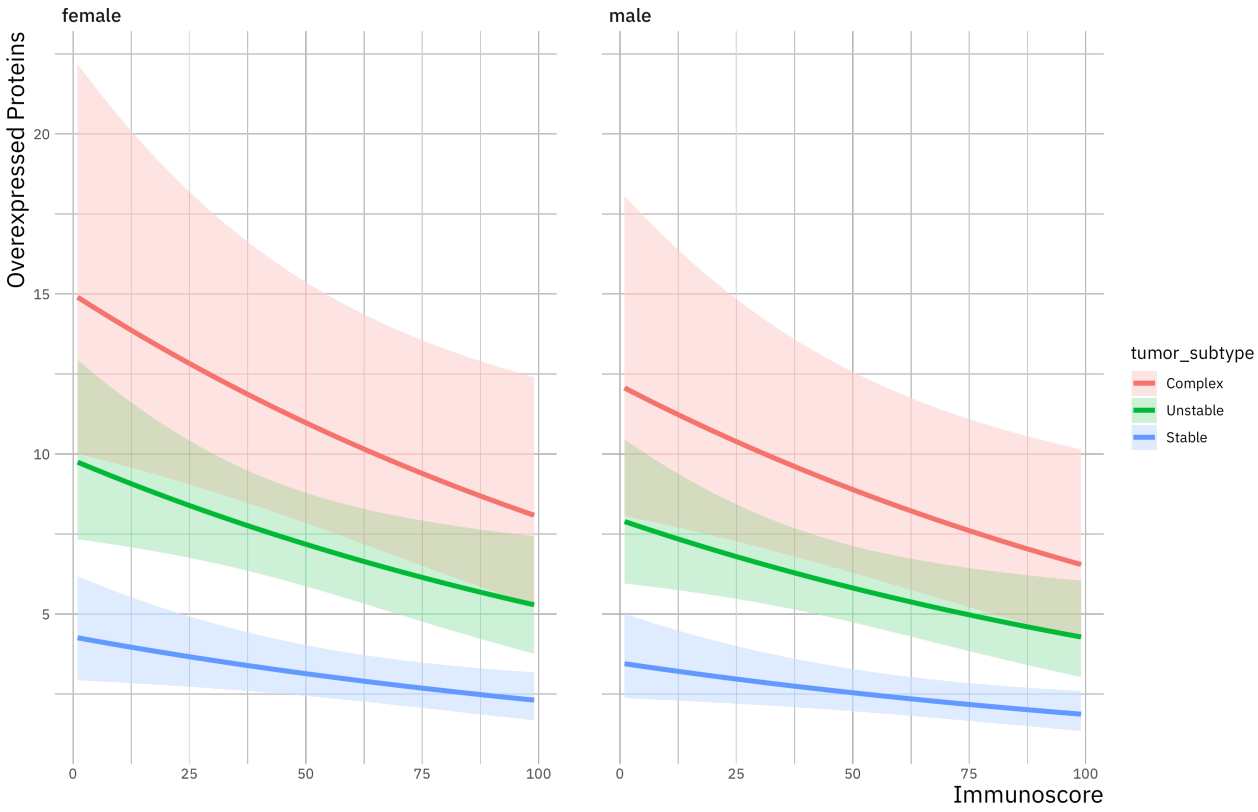
First, we can look at predicted counts for each value of `tumor_subtype` while holding `immunoscore` at its mean. To do this, we create a new dataset with the combinations of `tumor_subtype` and `immunoscore` for which we would like to find predicted values, then use the `predict()` command.

```
newdata_1 <-
data.frame(
    immunoscore = mean(dat$immunoscore),
    tumor_subtype = factor(c("Complex", "Unstable", "Stable"), labels = levels(dat$tumor_subtype)),
    gender="male")

newdata_2 <-
data.frame(
    immunoscore = mean(dat$immunoscore),
    tumor_subtype = factor(c("Complex", "Unstable", "Stable"), labels = levels(dat$tumor_subtype)),
    gender="female")

new_data <- rbind(newdata_1, newdata_2)

new_data$phat <- predict(glm_1, new_data, type = "response")

print(new_data)
```

```
##   immunoscore tumor_subtype gender      phat
## 1    48.26752       Complex   male  8.983829
## 2    48.26752        Stable   male  2.567187
## 3    48.26752      Unstable   male  5.876060
## 4    48.26752       Complex female 11.095193
## 5    48.26752        Stable female  3.170523
## 6    48.26752      Unstable female  7.257042
```

```
newdata_3 <-
data.frame(
    immunoscore = rep(seq(from = min(dat$immunoscore), to = max(dat$immunoscore), length.out = 100), 3)
    tumor_subtype = rep(factor(c("Complex", "Unstable", "Stable"), labels = levels(dat$tumor_subtype)),
    gender="male")

newdata_4 <-
data.frame(
    immunoscore = rep(seq(from = min(dat$immunoscore), to = max(dat$immunoscore), length.out = 100), 3)
    tumor_subtype = rep(factor(c("Complex", "Unstable", "Stable"), labels = levels(dat$tumor_subtype)),
    gender="female")

new_data <- rbind(newdata_3, newdata_4)

new_data <- cbind(new_data, predict(glm_1, new_data, type = "link", se.fit=TRUE))

new_data <- within(new_data, {
  overexpressed_proteins <- exp(fit)
  LL <- exp(fit - 1.96 * se.fit)
  UL <- exp(fit + 1.96 * se.fit)
})
```

```
library(ggplot2)

ggplot(new_data, aes(immunoscore, overexpressed_proteins)) +
    geom_ribbon(aes(ymin = LL, ymax = UL, fill = tumor_subtype), alpha = 0.2) +
    geom_line(aes(colour = tumor_subtype), size = 1.5) +
    labs(x = "Immunoscore",
         y = "Overexpressed Proteins") +
    facet_wrap(~ gender)
```


female
male
Overexpressed Proteins
20
15
10
5
0
25
50
75
100
Immunoscore
tumor_subtype
Complex
Unstable
Stable