Individual-based models of cultural evolution

A step-by-step guide using R

Alberto Acerbi Alex Mesoudi Marco Smolla

2020-12-01

Contents

Note to the reader

Please notice this is a "living" book. It will be updated over time. The most updated html version, to which we suggest to refer (also to have better access to the simulations code), is available at: https://acerbialberto.com/IBM-cultevo/

If you would like to cite this book, you can use this reference:

Acerbi Alberto, Mesoudi Alex, and Smolla Marco (2020) *Individual-based models of cultural evolution. A step-by-step guide using R.* doi:110.31219/osf.io/32v6a

Introduction

Aim of the book

The field of cultural evolution has emerged in the last few decades as a thriving, interdisciplinary effort to understand cultural change and cultural diversity within an evolutionary framework and using evolutionary tools, concepts and methods. Given its roots in evolutionary biology, much of cultural evolution is grounded in, or inspired by, formal models. Yet many researchers interested in cultural evolution come from backgrounds that lack training in formal models, such as psychology, anthropology or archaeology.

The aim of this book is to partly address this gap by showing readers how to create individual-based models (IBMs, also known as agent-based models, or ABMs) of cultural evolution. We provide example code written in the programming language R, which has been widely adopted in the scientific community. We will go from very simple models of the basic processes of cultural evolution, such as biased transmission and cultural mutation, to more advanced topics such as the evolution of social learning, demographic effects, and social network analysis. Where possible we recreate existing models in the literature, so that readers can better understand those existing models, and perhaps even extend them to address questions of their own interest.

What is cultural evolution?

The theory of evolution is typically applied to genetic change. Darwin pointed out that the diversity and complexity of living things can be explained in terms of a deceptively simple process: (1) organisms vary in their characteristics, (2) these characteristics are inherited from parent to offspring, and (3) those characteristics that make an organism more likely to survive and reproduce will tend to increase in frequency over time. That's pretty much it. Since Darwin, biologists have filled in many of the details of this abstract idea. Geneticists have shown that heritable 'characteristics' are determined by genes, and worked out where genetic variation comes from (e.g. mutation, recombination) and how genetic inheritance works (e.g. via Mendel's laws, and DNA). The details of se-

lection have been explored, revealing the many reasons why some genes spread and others don't. Others realised that not all biological change results from selection, it can also result from random processes like population bottlenecks (genetic drift).

The modern theory of cultural evolution began from the observation that culture constitutes a similar evolutionary process to that outlined above. 'Culture' is defined as information that passes from one individual to another socially, rather than genetically. This could include things we colloquially call knowledge, beliefs, ideas, attitudes, customs, words, or values. These are all learned from others via various 'social learning' mechanisms such as imitation or spoken/written language. The key point is that social learning is an inheritance system. Cultural characteristics (or cultural traits) vary across individuals, they are passed from individual to individual, and in many cases some traits are more likely to spread than others. This is Darwin's insight, applied to culture. Cultural evolution researchers think that we can use similar evolutionary concepts, tools and methods to explain the diversity and complexity of culture, just as biologists have done for the diversity and complexity of living forms. The models in this book will help you to understand many of the above principles, by creating simulations of various aspects of cultural evolution.

Importantly, we do not need to assume that cultural evolution is identical to genetic evolution. Many of the details will be different. To take an obvious example, we get DNA only from our two parents, but we can get ideas from many sources: teachers, strangers on the internet, long-dead authors' books, or even our parents. Cultural evolution researchers seek to build models and do research to fill in these details. In the last part of the book we will also explore models that go beyond a strict analogy with biological evolution, and focus on features such as the fact that the 'rules' that regulate transmission can themselves culturally evolve, or that other processes than inheritance can create and stabilise culture.

Why model?

A formal model is a simplified version of reality, written in mathematical equations or computer code. Formal models are useful because reality is complex. We can observe changes in species or cultures over time, or particular patterns of biological or cultural diversity, but there are always a vast array of possible causes for any particular pattern or trend, and huge numbers of variables interacting in many different ways. A formal model is a highly simplified recreation of a small part of this complex reality, containing a few elements or processes that the modeller suspects are important. A model, unlike reality, can be manipulated and probed in order to better understand how each part works. No model is ever a complete recreation of reality. That would be pointless: we would have replaced a complex, incomprehensible reality with a complex, incomprehensible model. Instead, models are useful *because* of their simplicity.

Formal modelling is rare in the social sciences (with some exceptions, such as economics). Social scientists tend to be sceptical that very simple models can tell us anything useful about something as immensely complex as human culture. But the clear lesson from biology is that models are extremely useful in precisely this situation. Biologists face similarly immense complexity in the natural world. Despite this, models are useful. Population genetics models of the early 20th century helped to reconcile new findings in genetics with Darwin's theory of evolution. Ecological models helped understand interactions between species, such as predator-prey cycles. These models are hugely simplified: population genetics models typically make ridiculous assumptions like infinitely large populations and random mating. But they are useful because they precisely specify each part of a complex system, improving understanding of reality.

Another way to look at this is that all social scientists use models, but only some use *formal* models. Most theories in social sciences are verbal models, written in words. The problem is that words can be imprecise, and verbal models contain all kinds of hidden or unstated assumptions. The advantage of formal modelling is that we are forced to precisely specify every element and process that we propose, and make all of our assumptions explicit. Maths and code do not accept any ambiguity: they must be told absolutely everything.

Models can also help to understand the consequences of our theories. Social systems, like many others, are typically under the influence of several different interacting forces. In isolation the effects of these forces can be easy to predict. However, when several forces interact the resulting dynamics quickly become non-trivial. This is the basic idea behind defining these systems as 'complex' systems. With verbal descriptions, figuring out the effects of interactions is left to our insights. With formal models, we can set up systems with these forces and observe the dynamics of their interactions.

Why individual-based models?

There are several different types of formal models. Some models describe the behaviour of a system at the population-level, tracking overall frequencies or other descriptive statistics of traits without explicitly modelling individuals. For example, a model can specify that the frequency of a cultural trait A at time t depends on its frequency at time $t-1$. Perhaps it doubles at each time step. Other models, instead, describe the behaviour of a system at the individual-level, explicitly modelling the individual entities that possess the traits. Imagine the same question, but now we specify that, in a population of N individuals, each individual observes each time a random number of other individuals and, if at least one of them has trait A, it copies that trait.

Another distinction concerns models that are analytically tractable and models that are not. The former are mathematical models that consist of sets of equations that can be solved to find specific answers (e.g. equilibria points). Our

population-level model described above would fit this description. A big advantage of these models is that they can provide insight into the dynamics of a system for a wide range of parameters, or exact results for specific questions. However, this approach requires the studied dynamics to be rather simple. It would be more difficult (or perhaps impossible) to write and analytically solve the systems of equations necessary to describe the behaviours of the single individuals in the second model.

Often, when we want or need to describe the behaviour at the individual level - if, for example, individuals differ in their characteristics, exhibit learning or adaptation, or are embedded in social networks - trying to write a system of equations may not be the best strategy. Instead, we need to write code and let the computer program run. These are individual-based models (IBMs). These models are both individual-level (i.e. they specifies the characteristics of the individuals and some rules by which those individuals interact or change over time) and simulations (i.e. they are not solved analytically, but simulated through a computer program).

Simulations have greater flexibility than analytical models. Due to their structure they are often more intuitive to understand, especially for people with little training in mathematics. However, it is also important to be aware of their downsides. For example, generalisations are often not possible and statements only hold for parameters (or sets thereof) that have been simulated. Another potential downside is that the high flexibility of simulations can quickly lead to models that are *too* complex, and it can be hard to understand what is happening inside the model. That's why, hopefully, our IBMs are simple enough to understand, and provide a gateway into cultural evolution modelling.

How to use this book - the programming

All of the code in this book is written in R. Originally R had a strong focus on statistical data analysis. Its growing user-base has turned R into a more general-purpose programming language. While R is used less often for modelling, it is widely taught in many university departments and is the subject of lots of online tutorials and support forums. It is quite likely that many readers already have some experience in R for data analysis and visualisation which can be used also for IBMs, more easily than learning another programming language. Also, if your IBMs run in R, you can use the same language to analyse the output and plot the results.

We have used the bookdown package to create an html version of the book, which you may well be reading now. This is created from RMarkdown (.Rmd) files, which are a mix of regular text and code. As a reader, you can therefore read the online book and, alongside, run the code using an Rmd file. Of course you can just read the book, but running the code as you go will give you more direct experience of how the code executes, and will allow you to play around

with parameters and commands. The best way of learning - especially modelling! - is to try it out yourself.

We assume that the reader has basic knowledge of R (and RStudio, which provides a powerful user-interface for R), including installing it, setting it up, updating it, installing packages and running code. We strived to proceed from very simple to more complex code in a gradual way, and to explain all the non-obvious newly introduced programming techniques, but a basic knowledge of R as a programming language, e.g. the use of variables, dataframes, functions, subsetting and loops, will greatly facilitate the reading.

We use the tidyverse package and follow the underlying logic. For example, we use the tidyverse-typical data structures (tibbles rather than dataframes) and the ggplot graphic system (rather than the base R plot function). These are user-friendly and widely used, and they will make it easier to manipulate data and create professional-looking visualisations. The tidyverse, however, has not been created with IBMs in mind. We have therefore not religiously stuck to tidyverse, and we also use functions, data structures, and programming styles that go beyond the tidyverse (in chapter Chapter 7, for example, we show how matrices are more effective than tibbles in computationally-heavy simulations).

Beside the tidyverse package, we have limited as much as possible the number of additional packages needed to run the simulations. The few packages needed to compile some of the code are explicitly introduced in the book when needed.

How to use this book - the simulations

The book is intended - as the title says - as a step-by-step guide. If you are interested in modelling cultural evolution, or in modelling in general, and you do not have previous experience, you should go through the simulations we describe chapter by chapter. The chapters build in complexity both from the programming and from the conceptual point of view. Alternatively, if you are interested in specific models then you can go straight to the relevant chapter. In this case, however, you will need previous programming experience. (And you will have to figure out by yourself at least some of our programming choices!)

The book is organised as follows. We start by presenting IBM versions of some of the now-classic mathematical and population-level models described in the foundational cultural evolution books, such as Robert Boyd and Peter Richerson's *Culture and the Evolutionary Process* and Luigi-Luca Cavalli-Sforza and Marc Feldman's *Cultural Transmission and Evolution*. The models do not add conceptually to the original analytical treatments, but they show how to use them to develop IBMs, and they provide several basic tools to build models that describe cultural evolution. Some of the subsequent chapters develop aspects that are possible only with IBMs, for example, simulating cultural dynamics with many different traits (Chapter 7).

We then move to what we call 'Advanced topics'. These chapters deal with more recent work in cultural evolution and include different perspectives, or they concern analyses that are not customary in cultural evolution modelling (for example network analysis in Chapter 14).

The book does not present *new* models, views or findings on cultural evolution. We are trying to provide as much as possible an up-to-date reflection of the field and, mostly, to show some of the possibilities that IBMs offer to cultural evolutionists. If, while reading this book, you are suddenly struck by an idea for a new model or an alteration of one of the models we present here, we have succeeded in our mission.

Conventions and formatting

In general, we follow the tidyverse style guide for naming functions and variables, and code formatting.

Names of functions and variables use underscores to separate words and lowercase letters, e.g. `previous_population`, `biased_mutation`. If in the same chapter we have more than one function for the same model (for example because we gradually add parameters), they are numbered as `unbiased_transmission_1()`, `unbiased_transmission_2()`, etc.

For the text, we use the following conventions:

- names of functions and data structures: `unbiased_transmission()`, `population`, `output`
- technical terms: 'geoms', 'chr'
- names of variables: p, *generation*

Further reading

For some recent general books on cultural evolution, you can check Mesoudi [2011], Morin [2015], Henrich [2016], Laland [2017], and Acerbi [2019].

The 'foundational' books referred in the test above are Cavalli-Sforza and Feldman [1981] and Boyd and Richerson [1985].

For more on the virtues of formal models for social scientists, with a cultural evolution perspective, see Smaldino [2017]. Smaldino [2020] is dedicated to good practices to translate verbal theories into formal, especially individual-based, models.

A good introduction to R programming is Grolemund [2014]. Another general introduction, with a specific focus on the tidyverse logic, is Wickham and Grolemund [2017].

Basics

Chapter 1

Unbiased transmission

We start by simulating a simple case of unbiased cultural transmission. We will detail each step of the simulation and explain the code line-by-line. In the following chapters, we will reuse most of this initial model, building up the complexity of our simulations.

1.1 Initialising the simulation

Here we will simulate a case where N individuals each possess one of two mutually exclusive cultural traits. These alternative traits are denoted A and B. For example, A might be eating a vegetarian diet, and B might be eating a non-vegetarian diet. In reality, traits are seldom clear-cut (e.g. what about pescatarians?), but models are designed to cut away all the complexity to give tractable answers to simplified situations.

Our model has non-overlapping generations. In each generation, all N individuals are replaced with N new individuals. Again, this is unlike any real biological group but provides a simple way of simulating change over time. Generations here could correspond to biological generations, but could equally be 'cultural generations' (or learning episodes), which might be much shorter.

Each new individual of each new generation picks a member of the previous generation at random and copies their cultural trait. This is known as unbiased oblique cultural transmission. 'Unbiased' refers to the fact that traits are copied entirely at random. The term 'oblique' means that members of one generation learn from those of the previous, non-overlapping, generation. This is different from, for example, horizontal cultural transmission, where individuals copy members of the same generation, and vertical cultural transmission, where offspring copy their biological parents.

If we assume that the two cultural traits are transmitted in an unbiased way,

what does that mean for the average trait frequency in the population? To answer this question, we must track the proportion of individuals who possess trait A over successive generations. We will call this proportion p. We could also track the proportion who possess trait B, but this will always be $1 - p$ given that the two traits are mutually exclusive. For example, if 70% of the population have trait A ($p = 0.7$), then the remaining 30% must have trait B (i.e. $1 - p = 1 - 0.7 = 0.3$).

The output of the model will be a plot showing p over all generations up to the last generation. Generations (or time steps) are denoted by t, where generation one is $t = 1$, generation two is $t = 2$, up to the last generation $t = t_{\max}$.

First, we need to specify the fixed parameters of the model. These are quantities that we decide on at the start and do not change during the simulation. In this model these are `N` (the number of individuals) and `t_max` (the number of generations). Let's start with `N = 100` and `t_max = 200`:

```
N <- 100
t_max <- 200
```

Now we need to create our individuals. The only information we need to keep about our individuals is their cultural trait (A or B). We'll call `population` the data structure containing the individuals. The type of data structure we have chosen here is a tibble. This is a more user-friendly version of a dataframe. Tibbles, and the tibble command, are part of the `tidyverse` library, which we need to call before creating the tibble. We will use other commands from the `tidyverse` throughout the book.

Initially, we'll give each individual either an A or B at random, using the `sample()` command. This can be seen in the code chunk below. The `sample()` command takes three arguments (i.e. inputs or options). The first argument lists the elements to pick at random, in our case, the traits A and B. The second argument gives the number of times to pick, in our case N times, once for each individual. The final argument says to replace or reuse the elements specified in the first argument after they've been picked (otherwise there would only be one copy of A and one copy of B, so we could only give two individuals traits before running out). Within the tibble command, the word `trait` denotes the name of the variable within the tibble that contains the random As and Bs, and the whole tibble is assigned the name `population`.

```
library(tidyverse)
population <- tibble(trait = sample(c("A", "B"), N, replace = TRUE))
```

We can see the cultural traits of our population by simply entering its name in the R console:

```
population
```

```
## # A tibble: 100 x 1
```

```
##    trait
##    <chr>
##  1 A
##  2 A
##  3 A
##  4 A
##  5 B
##  6 A
##  7 B
##  8 A
##  9 A
## 10 B
## # ... with 90 more rows
```

As expected, there is a single column called `trait` containing As and Bs. The type of the column, in this case `<chr>` (i.e. character), is reported below the name.

A specific individual's trait can be retrieved using the square bracket notation in R. For example, individual 4's trait can be retrieved by typing:

```
population$trait[4]
```

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

This should match the fourth row in the table above.

We also need a tibble to record the output of our simulation, that is, to track the trait frequency p in each generation. This will have two columns with $t_{\max}$ rows, one row for each generation. The first column is simply a counter of the generations, from 1 to $t_{\max}$. This will be useful for plotting the output later. The other column should contain the values of p for each generation.

At this stage we don't know what p will be in each generation, so for now let's fill the `output` tibble with lots of NAs, which is R's symbol for Not Available, or missing value. We can use the `rep()` (repeat) command to repeat NA $t_{\max}$ times. We're using NA rather than, say, zero, because zero could be misinterpreted as $p = 0$, which would mean that all individuals have trait B. This would be misleading, because at the moment we haven't yet calculated p, so it's nonexistent, rather than zero.

```
output <- tibble(generation = 1:t_max, p = rep(NA, t_max))
```

We can, however, fill in the first value of `p` for our already-created first generation of individuals, held in `population`. The command below sums the number of As in `population` and divides by N to get a proportion out of 1 rather than an absolute number. It then puts this proportion in the first slot of `p` in `output`, the one for the first generation, $t = 1$. We can again write the name of the tibble, `output`, to see that it worked.

```
output$p[1] <- sum(population$trait == "A") / N
output
```

```
## # A tibble: 200 x 2
##    generation     p
##         <int> <dbl>
##  1          1  0.44
##  2          2 NA
##  3          3 NA
##  4          4 NA
##  5          5 NA
##  6          6 NA
##  7          7 NA
##  8          8 NA
##  9          9 NA
## 10         10 NA
## # ... with 190 more rows
```

This first value of `p` should be around 0.5, meaning that around 50 individuals have trait A, and 50 have trait B. Even though `sample()` returns either trait with equal probability, this does not necessarily mean that we will get exactly 50 As and 50 Bs. This happens with simulations and finite population sizes: they are probabilistic (or stochastic), not deterministic. Analogously, flipping a coin 100 times will not always give exactly 50 heads and 50 tails. Sometimes we will get 51 heads, sometimes 49, etc. To see this in our simulation, you can re-run all of the above code and you should get a different p.

1.2 Execute generation turn-over many times

Now that we have built the population, we can simulate what individuals do in each generation. We iterate these actions over $t_{\max}$ generations. In each generation, we need to:

- copy the current individuals to a separate tibble called `previous_population` to use as demonstrators for the new individuals; this allows us to implement oblique transmission with its non-overlapping generations, rather than mixing up the generations
- create a new generation of individuals, each of whose trait is picked at random from the `previous_population` tibble
- calculate p for this new generation and store it in the appropriate slot in `output`

To iterate, we'll use a for-loop, using `t` to track the generation. We've already done generation 1 so we'll start at generation 2. The random picking of models is done with `sample()` again, but this time picking from the traits held in

previous_population. Note that we have added comments briefly explaining what each line does. This is perhaps superfluous when the code is this simple, but it's always good practice. Code often gets cut-and-pasted into other places and loses its context. Explaining what each line does lets other people - and a future, forgetful you - know what's going on.

```
for (t in 2:t_max) {
  # Copy the population tibble to previous_population tibble
  previous_population <- population

  # Randomly copy from previous generation's individuals
  population <- tibble(trait = sample(previous_population$trait, N, replace = TRUE))

  # Get p and put it into the output slot for this generation t
  output$p[t] <- sum(population$trait == "A") / N
}
```

Now we should have 200 values of p stored in output, one for each generation. You can list them by typing output, but more effective is to plot them.

1.3 Plotting the model results

We use ggplot() to plot our data. The syntax of ggplot may be slightly obscure at first, but it forces us to have a clear picture of the data before plotting.

In the first line in the code below, we are telling ggplot that the data we want to plot is in the tibble output. Then, with the command aes() we declare the 'aesthetics' of the plot, that is, how we want our data mapped in our plot. In this case, we want the values of p on the y-axis, and the values of generation on the x-axis (this is why earlier we created, in the tibble output, a column to keep the count of generations).

We then use geom_line(). In ggplot, 'geoms' describe what kind of visual representation should be plotted: lines, bars, boxes and so on. This visual representation is independent of the mapping that we declared before with aes(). The same data, with the same mapping, can be visually represented in many different ways. In this case, we are asking ggplot to represent the data as a line. You can change geom_line() in the code below to geom_point() and see what happens (other geoms have less obvious effects, and we will see some of them in later chapters).

The other commands are mainly to make the plot look nicer. We want the y-axis to span all the possible values of p, from 0 to 1, and we use a particular 'theme' for our plot, in this case, a simple black and white (theme_bw) theme. With the command labs() we can provide a more informative label for the y-axis. ggplot automatically labels the axis with the name of the tibble columns that are plotted: this is good for generation, but less so for p.

```
ggplot(data = output, aes(y = p, x = generation)) +
  geom_line() +
  ylim(c(0, 1)) +
  theme_bw() +
  labs(y = "p (proportion of individuals with trait A)")
```

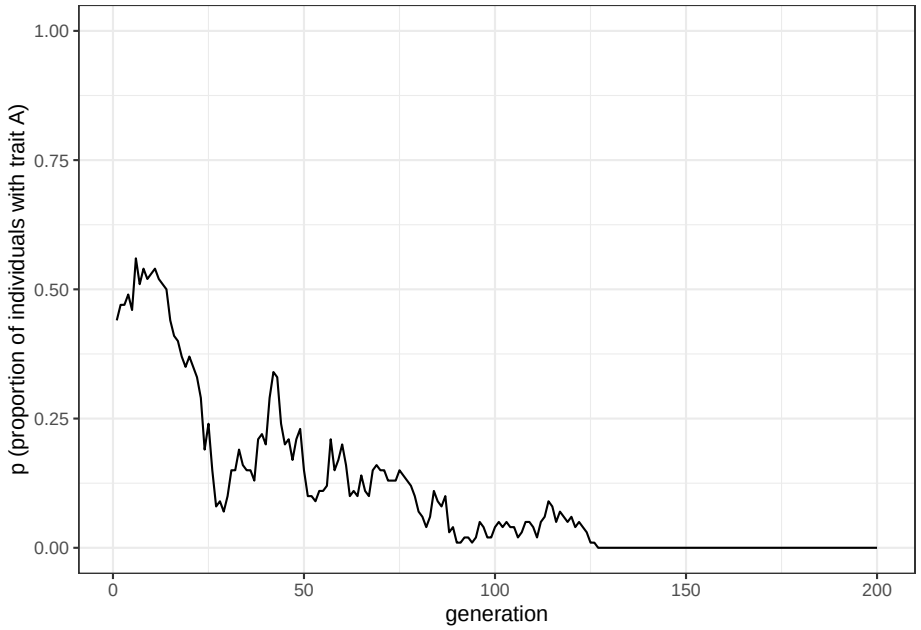


Figure 1.1: Random fluctuations of the proportion of trait A under unbiased cultural transmission

The proportion of individuals with trait A should start off hovering around 0.5, and then oscillate randomly (it may, in some cases, also reach 0, meaning that all As have disappeared, or 1, meaning that all Bs have disappeared). Unbiased transmission, or random copying, is by definition random, so different runs of this simulation will generate different plots. If you rerun all the code you will get something different. In all likelihood, p might go to 0 or 1 at some point. At $p = 0$ there are no As and every individual possesses B. At $p = 1$ there are no Bs and every individual possesses A. This is a typical feature of cultural drift, analogous to genetic drift: in small populations, with no selection or other directional processes operating, traits can be lost purely by chance after some generations.

1.4 Write a function to wrap the model code

Ideally, we would like to repeat the simulation to explore this idea in more detail, perhaps changing some of the parameters. For example, if we increase N, are we more or less likely to lose one of the traits? As noted above, individual-based models like this one are probabilistic or stochastic, thus it is essential to run simulations many times to understand what happens. With our code scattered about in chunks, it is hard to quickly repeat the simulation. Instead, we can wrap it all up in a function:

```
unbiased_transmission_1 <- function(N, t_max) {
  population <- tibble(trait = sample(c("A", "B"), N, replace = TRUE))

  output <- tibble(generation = 1:t_max, p = rep(NA, t_max))

  output$p[1] <- sum(population$trait == "A") / N

  for (t in 2:t_max) {
    # Copy individuals to previous_population tibble
    previous_population <- population

    # Randomly copy from previous generation
    population <- tibble(trait = sample(previous_population$trait, N, replace = TRUE))

    # Get p and put it into output slot for this generation t
    output$p[t] <- sum(population$trait == "A") / N
  }
  # Export data from function
  output
}
```

This is just all of the code snippets that we already ran above, but all within a function with parameters N and $t_{\max}$ as arguments to the function. In addition, `unbiased_transmission_1()` ends with the line `output`. This means that this tibble will be exported from the function when it is run. This is useful for storing data from simulations wrapped in functions, otherwise that data is lost after the function is executed.

Nothing will happen when you run the above code, because all you have done is define the function and not actually run it. The point is that we can now call the function in one go, easily changing the values of N and $t_{\max}$. Let's try first with the same values of N and $t_{\max}$ as before, and save the output from the simulation into `data_model`, as a record of what happened.

```
data_model <- unbiased_transmission_1(N = 100, t_max = 200)
```

We also need to create another function to plot the data, so we do not need to rewrite all the plotting instructions each time. Whereas this may seem imprac-

tical now, it is convenient to separate the function that runs the simulation and the function that plots the data for various reasons. With more complicated models, we do not want to rerun a simulation just because we want to change some detail in the plot. It also makes conceptual sense to keep separate the raw output of the model from the various ways we can visualise it, or the further analysis we want to perform on it. As above, the code is identical to what we already wrote:

```
plot_single_run <- function(data_model) {
  ggplot(data = data_model, aes(y = p, x = generation)) +
    geom_line() +
    ylim(c(0, 1)) +
    theme_bw() +
    labs(y = "p (proportion of individuals with trait A)")
}
```

At this point, we can visualise the results:

```
plot_single_run(data_model)
```

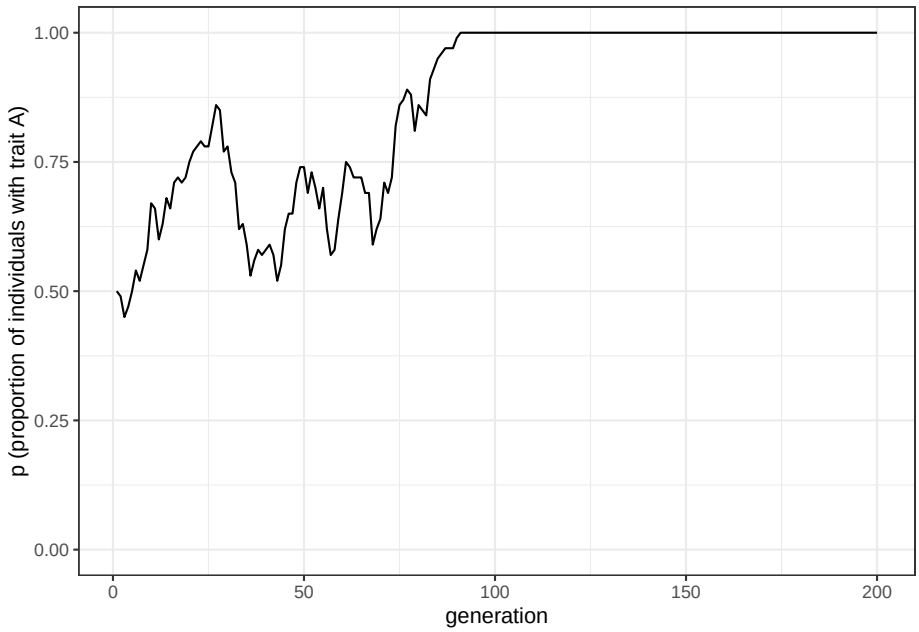


Figure 1.2: Random fluctuations of the proportion of trait A under unbiased cultural transmission

As anticipated, the plot is different from the simulation we ran before, even though the code is exactly the same. This is due to the stochastic nature of the simulation.

Now let's try changing the parameters. We can call the simulation and the plotting functions together. The code below reruns and plots the simulation with a much larger N.

```
data_model <- unbiased_transmission_1(N = 10000, t_max = 200)
plot_single_run(data_model)
```

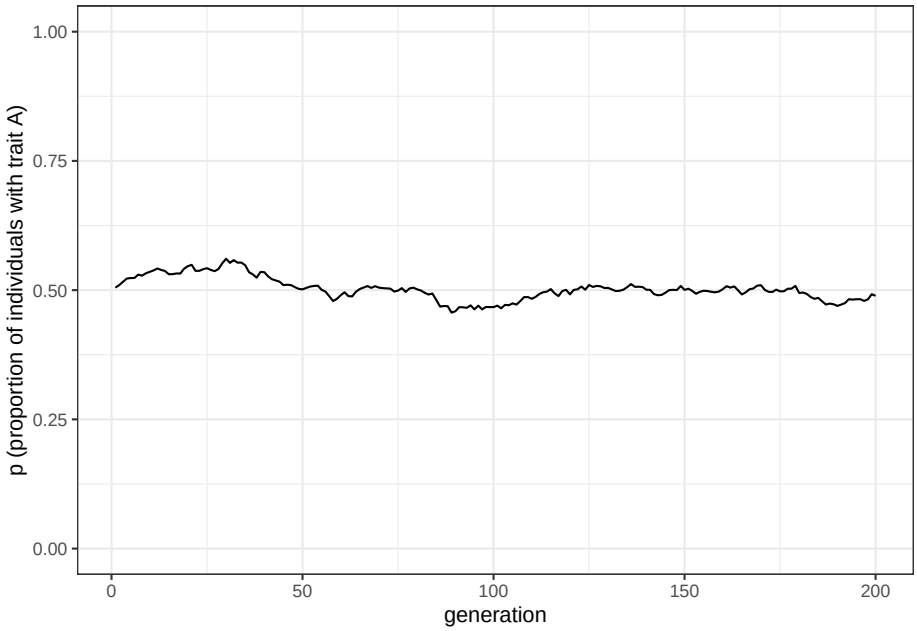


Figure 1.3: Random fluctuations of the proportion of trait A under unbiased cultural transmission and a large population size

You should see much less fluctuation. Rarely in a population of $N = 10000$ will either trait go to fixation. Try re-running the previous code chunk to explore the effect of N on long-term dynamics.

1.5 Run several independent simulations and plot their results

Wrapping a simulation in a function like this is good because we can easily re-run it with just a single command. However, it's a bit laborious to manually re-run it. Say we wanted to re-run the simulation 10 times with the same parameter values to see how many times A goes to fixation, and how many times B goes to fixation. Currently, we'd have to manually run the `unbiased_transmission_1()` function 10 times and record somewhere else what happened in each run. It would be better to automatically re-run the simulation several times and plot

each run as a separate line on the same plot. We could also add a line showing the mean value of p across all runs.

Let's use a new parameter $r_{\max}$ to specify the number of independent runs, and use another for-loop to cycle over the $r_{\max}$ runs. Let's rewrite the `unbiased_transmission_1()` function to handle multiple runs. We will call the new function `unbiased_transmission_2()`.

```
unbiased_transmission_2 <- function(N, t_max, r_max) {
  output <- tibble(generation = rep(1:t_max, r_max),
                   p = as.numeric(rep(NA, t_max * r_max)),
                   run = as.factor(rep(1:r_max, each = t_max)))
  # For each run
  for (r in 1:r_max) {
    # Create first generation
    population <- tibble(trait = sample(c("A", "B"), N, replace = TRUE))

    # Add first generation's p for run r
    output[output$generation == 1 & output$run == r, ]$p <-
      sum(population$trait == "A") / N

    # For each generation
    for (t in 2:t_max) {
      # Copy individuals to previous_population tibble
      previous_population <- population

      # Randomly copy from previous generation
      population <- tibble(trait = sample(previous_population$trait, N, replace = TRUE)

      # Get p and put it into output slot for this generation t and run r
      output[output$generation == t & output$run == r, ]$p <-
        sum(population$trait == "A") / N
    }
  }
  # Export data from function
  output
}
```

There are a few changes here. First, we need a different `output` tibble, because we need to store data for all the runs. For that, we initialise the same `generation` and `p` columns as before, but with space for all the runs. `generation` is now built by repeating the count of each generation $r_{\max}$ times, and `p` is NA repeated for all generations, for all runs.

We also need a new column called `run` that keeps track of which run the data in the other two columns belongs to. Note that the definition of `run` is preceded by `as.factor()`. This specifies the type of data to put in the `run` column. We

want `run` to be a 'factor' or categorical variable so that, even if runs are labelled with numbers (1, 2, 3…), this should not be misinterpreted as a continuous, real number: there is no sense in which run 2 is twice as 'runny' as run 1, or run 3 half as 'runny' as run 6. Runs could equally have been labelled using letters, or any other arbitrary scheme. While omitting `as.factor()` does not make any difference when running the simulation, it would create problems when plotting the data because ggplot would treat runs as continuous real numbers rather than discrete categories (you can see this yourself by modifying the definition of `output` in the previous code chunk). This is a good example of how it is important to have a clear understanding of your data before trying to plot or analyse them.

Going back to the function, we then set up a loop which executes once for each run. The code within this loop is mostly the same as before, except that we now use the `[output$generation == t & output$run == r, ]` notation to put p into the right place in `output`.

The plotting function is also changed to handle multiple runs:

```
plot_multiple_runs <- function(data_model) {
  ggplot(data = data_model, aes(y = p, x = generation)) +
    geom_line(aes(colour = run)) +
    stat_summary(fun = mean, geom = "line", size = 1) +
    ylim(c(0, 1)) +
    theme_bw() +
    labs(y = "p (proportion of individuals with trait A)")
}
```

To understand how the above code works, we need to explain the general functioning of ggplot. As explained above, `aes()` specifies the 'aesthetics', or how the data are mapped in the plot. This is independent from the possible visual representations of this mapping, or 'geoms'. If we declare specific aesthetics when we call `ggplot()`, these aesthetics will be applied to all geoms we call afterwards. Alternatively, we can specify the aesthetics in the geom itself. For example this:

```
ggplot(data = output, aes(y = p, x = generation)) +
  geom_line()
```

is equivalent to this:

```
ggplot(data = output) +
  geom_line(aes(y = p, x = generation))
```

We can use this property to make more complex plots. The plot created in `plot_multiple_runs()` has a first geom, `geom_line()`. This inherits the aesthetics specified in the initial call to `ggplot()` but also has a new mapping specific to `geom_line()`, `colour = run`. This tells ggplot to plot each run line with a different colour. The next command, `stat_summary()`, calculates the

mean of all runs. However, this only inherits the mapping specified in the initial `ggplot()` call. If in the aesthetic of `stat_summary()` we had also specified `colour = run`, it would separate the data by run, and it would calculate the mean of each run. This, though, is just the lines we have already plotted with the `geom_line()` command. For this reason, we did not put `colour = run` in the `ggplot()` call, only in `geom_line()`. As always, there are various ways to obtain the same result. This code:

```
ggplot(data = output) +
  geom_line(aes(y = p, x = generation, colour = run)) +
  stat_summary(aes(y = p, x = generation), fun = mean, geom = "line", size = 1)
```

is equivalent to the code we wrapped in the function above. However, the original code is clearer, as it distinguishes the global mapping, and the mappings specific to each visual representation.

`stat_summary()` is a generic ggplot function which can be used to plot different statistics to summarise our data. In this case, we want to calculate the mean of the data mapped in y, we want to plot them with a line, and we want this line to be thicker than the lines for the single runs. The default line size for geom_line is 0.5, so `size = 1` doubles the thickness.

Let's now run the function and plot the results for five runs with the same parameters we used at the beginning ($N = 100$ and $t_{\max} = 200$):

```
data_model <- unbiased_transmission_2(N = 100, t_max = 200, r_max = 5)
plot_multiple_runs(data_model)
```

You should be able to see five independent runs of our simulation shown as regular thin lines, along with a thicker line showing the mean of these lines. Some runs have probably gone to 0 or 1, and the mean should be somewhere in between. The data is stored in `data_model`, which we can inspect by writing its name.

```
data_model
```

```
## # A tibble: 1,000 x 3
##    generation     p run
##         <int> <dbl> <fct>
##  1          1  0.43 1
##  2          2  0.4  1
##  3          3  0.42 1
##  4          4  0.46 1
##  5          5  0.47 1
##  6          6  0.42 1
##  7          7  0.4  1
##  8          8  0.38 1
##  9          9  0.36 1
## 10         10  0.35 1
```

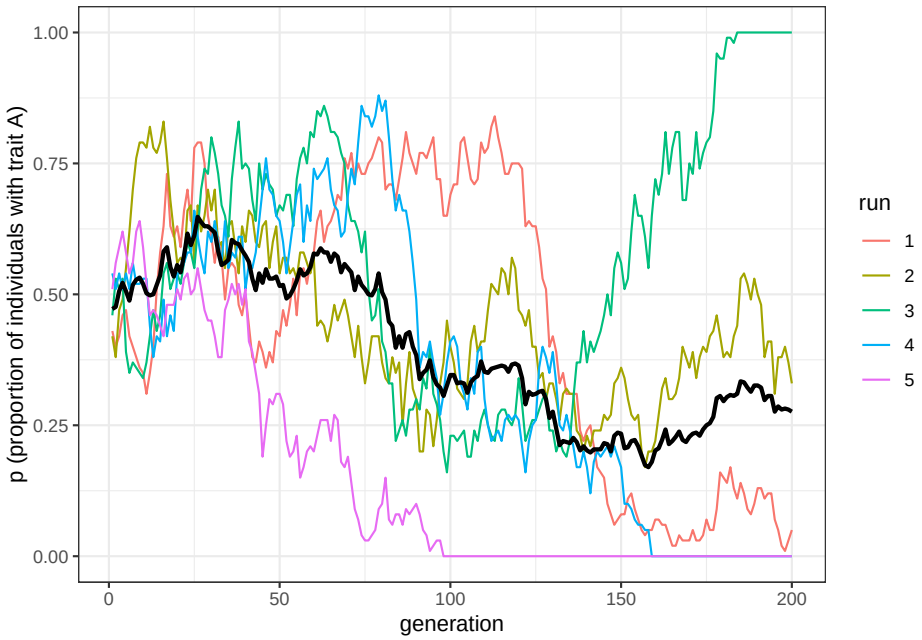


Figure 1.4: Unbiased cultural transmission generates different dynamics in multiple runs

```
## # ... with 990 more rows
```

Now let's run the `unbiased_transmission_2()` model with $N = 10000$, to compare with $N = 100$.

```
data_model <- unbiased_transmission_2(N = 10000, t_max = 200, r_max = 5)
plot_multiple_runs(data_model)
```

The mean line should be almost exactly at $p = 0.5$ now, with the five independent runs fairly close to it.

1.6 Varying initial conditions

Let's add one final modification. So far the starting frequencies of A and B have been the same, roughly 0.5 each. But what if we were to start at different initial frequencies of A and B? Say, $p = 0.2$ or $p = 0.9$? Would unbiased transmission keep p at these initial values, or would it go to $p = 0.5$ as we have found so far?

To find out, we can add another parameter, `p_0`, which specifies the initial probability of an individual having an A rather than a B in the first generation. Previously this was always `p_0 = 0.5`, but in the new function below we add it to the `sample()` function to weight the initial allocation of traits.

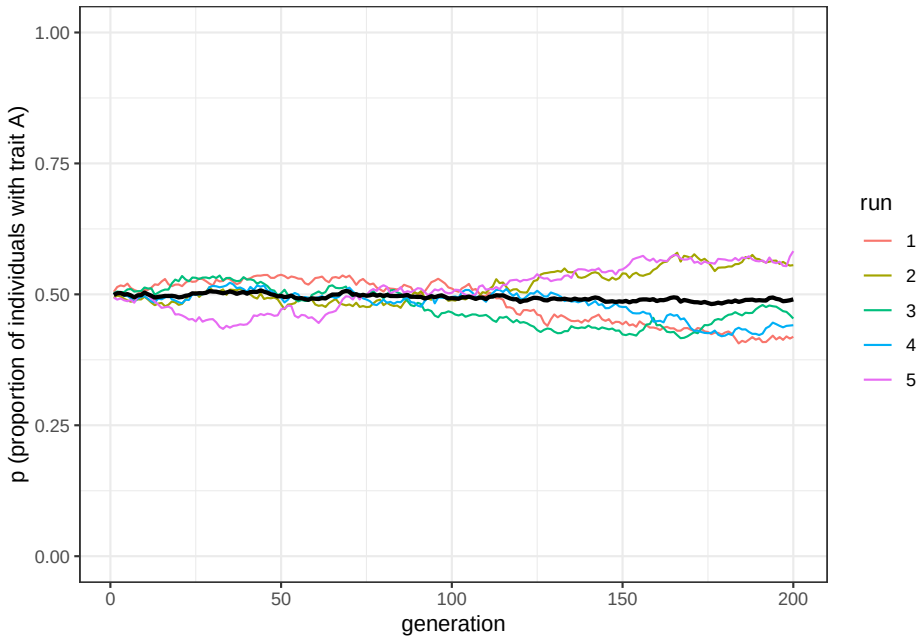


Figure 1.5: Unbiased cultural transmission generates similar dynamics in multiple runs when population sizes are very large

```
unbiased_transmission_3 <- function(N, p_0, t_max, r_max) {
  output <- tibble(generation = rep(1:t_max, r_max),
                   p = as.numeric(rep(NA, t_max * r_max)),
                   run = as.factor(rep(1:r_max, each = t_max)))
  # For each run
  for (r in 1:r_max) {
    # Create first generation
    population <- tibble(trait = sample(c("A", "B"), N, replace = TRUE,
                                        prob = c(p_0, 1 - p_0)))

    # Add first generation's p for run r
    output[output$generation == 1 & output$run == r, ]$p <-
      sum(population$trait == "A") / N

    for (t in 2:t_max) {
      # Copy individuals to previous_population tibble
      previous_population <- population

      # Randomly copy from previous generation
      population <- tibble(trait = sample(previous_population$trait, N, replace = TRUE
```

```
      # Get p and put it into output slot for this generation t and run r
      output[output$generation == t & output$run == r, ]$p <-
        sum(population$trait == "A") / N
    }
  }
  # Export data from function
  output
}
```

`unbiased_transmission_3()` is almost identical to the previous function. The only changes are the addition of p_0 as an argument to the function, and the *prob* argument in the `sample()` command. The *prob* argument gives the probability of picking each option, in our case A and B, in the first generation. The probability of A is now p_0, and the probability of B is now $1 - p_0$. We can use the same plotting function as before to visualise the result. Let's see what happens with a different value of p_0, for example $p_0 = 0.2$.

```
data_model <- unbiased_transmission_3(N = 10000, p_0 = 0.2, t_max = 200, r_max = 5)
plot_multiple_runs(data_model)
```

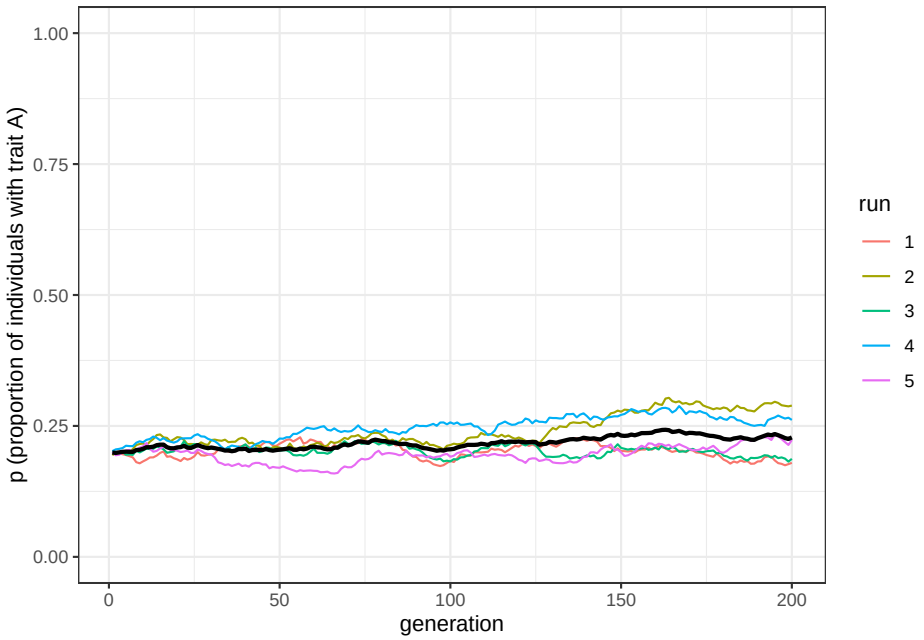


Figure 1.6: Unbiased transmission does not change trait frequencies from the starting conditions, barring random fluctuations

With $p_0 = 0.2$, trait frequencies stay at $p = 0.2$. Unbiased transmission is truly non-directional: it maintains trait frequencies at whatever they were in the

previous generation, barring random fluctuations caused by small population sizes.

1.7 Summary of the model

Even this extremely simple model provides some valuable insights. First, unbiased transmission does not in itself change trait frequencies. As long as populations are large, trait frequencies remain the same.

Second, the smaller the population size, the more likely traits are to be lost by chance. This is a basic insight from population genetics, known there as genetic drift, but it can also be applied to cultural evolution. Many studies have tested (and some supported) the idea that population size and other demographic factors can shape cultural diversity.

Furthermore, generating expectations about cultural change under simple assumptions like random cultural drift can be useful for detecting non-random patterns like selection. If we don't have a baseline, we won't know selection or other directional processes when we see them.

We have also introduced several programming techniques that will be useful in later simulations. We have seen how to use tibbles to hold characteristics of individuals and the outputs of simulations, how to use loops to cycle through generations and simulation runs, how to use `sample()` to pick randomly from sets of elements, how to wrap simulations in functions to easily re-run them with different parameter values, and how to use `ggplot()` to plot the results of simulations.

1.8 Further reading

Cavalli-Sforza and Feldman [1981] explored how cultural drift affects cultural evolution, which was extended by Neiman [1995] in an archaeological context. Bentley et al. [2004] present models of unbiased transmission for several cultural datasets. Lansing and Cox [2011] and commentaries explore the underlying assumptions of applying random drift to cultural evolution.

Chapter 2

Unbiased and biased mutation

Evolution doesn't work without a source of variation that introduces new variation upon which selection, drift and other processes can act. In genetic evolution, mutation is almost always blind with respect to function. Beneficial genetic mutations are no more likely to arise when they are needed than when they are not needed - in fact, most genetic mutations are neutral or detrimental to an organism. Cultural evolution is more interesting, in that novel variation may sometimes be directed to solve specific problems, or systematically biased due to features of our cognition. In the models below, we'll simulate both unbiased and biased mutation.

2.1 Unbiased mutation

First, we will simulate unbiased mutation in the same basic model as used in the previous chapter. We'll remove unbiased transmission to see the effect of unbiased mutation alone.

As in the previous model, we assume N individuals each of whom possesses one of two discrete cultural traits, denoted A and B. In each generation, from $t = 1$ to $t = t_{\max}$, the N individuals are replaced with N new individuals. Instead of random copying, each individual now gives rise to a new individual with the same cultural trait as them. (Another way of looking at this is in terms of timesteps, such as years: the same N individual live for $t_{\max}$ years and keep their cultural trait from one year to the next.)

At each generation, however, there is a probability μ that each individual mutates from their current trait to the other trait (the Greek letter Mu is the standard notation for the mutation rate in genetic evolution, and it has an

analogous function here). For example, vegetarian individuals can decide to eat animal products, and vice versa. Remember, this is not copied from other individuals, as in the previous model, but can be thought of as an individual decision. Another way to see this is that the probability of changing trait applies to each individual independently; whether an individual mutates has no bearing on whether or how many other individuals have mutated. On average, this means that μN individuals mutate each generation. Like in the previous model, we are interested in tracking the proportion p of agents with trait A over time.

We'll wrap this in a function called `unbiased_mutation()`, using much of the same code as `unbiased_transmission_3()`. As before, we need to call the tidyverse library in order to use the tibble command, and later commands like ggplot2.

```
library(tidyverse)

unbiased_mutation <- function(N, mu, p_0, t_max, r_max) {
  # Create the output tibble
  output <- tibble(generation = rep(1:t_max, r_max),
                   p = as.numeric(rep(NA, t_max * r_max)),
                   run = as.factor(rep(1:r_max, each = t_max)))

  for (r in 1:r_max) {
    population <- tibble(trait = sample(c("A", "B"), N, replace = TRUE,
                                        prob = c(p_0, 1 - p_0)))
    # Add first generation's p for run r
    output[output$generation == 1 & output$run == r, ]$p <-
      sum(population$trait == "A") / N
    for (t in 2:t_max) {
      # Copy individuals to previous_population tibble
      previous_population <- population

      # Determine 'mutant' individuals
      mutate <- sample(c(TRUE, FALSE), N, prob = c(mu, 1 - mu), replace = TRUE)

      # If there are 'mutants' from A to B
      if (nrow(population[mutate & previous_population$trait == "A", ]) > 0) {
        # Then flip them to B
        population[mutate & previous_population$trait == "A", ]$trait <- "B"
      }

      # If there are 'mutants' from B to A
      if (nrow(population[mutate & previous_population$trait == "B", ]) > 0) {
        # Then flip them to A
        population[mutate & previous_population$trait == "B", ]$trait <- "A"
```

```
      }

      # Get p and put it into output slot for this generation t and run r
      output[output$generation == t & output$run == r, ]$p <-
        sum(population$trait == "A") / N
    }
  }
  # Export data from function
  output
}
```

The only changes from the previous model are the addition of `mu`, the parameter that specifies the probability of mutation, in the function definition and new lines of code within the `for` loop on `t` which replace the random copying command with unbiased mutation. Let's examine these lines to see how they work.

The most obvious way of implementing unbiased mutation - which is *not* done above - would have been to set up another `for` loop. We would cycle through every individual one by one, each time calculating whether it should mutate or not based on `mu`. This would certainly work, but R is notoriously slow at loops. It's always preferable in R, where possible, to use 'vectorised' code. That's what is done above in our three added lines, starting from `mutate <- sample()`.

First, we pre-specify the probability of mutating for each individual. For this, we again use the function `sample()`, picking `TRUE` (corresponding to being a mutant) or `FALSE` (not mutating, i.e. keeping the same cultural trait) for N times. The draw, however, is not random: the probability of drawing `TRUE` is equal to μ, and the probability of drawing `FALSE` is $1-\mu$. You can think about the procedure in this way: each individual in the population flips a biased coin that has μ probability to land on, say, heads, and $1-\mu$ to land on tails. If it lands on heads they change their cultural trait.

In the subsequent lines we change the traits for the 'mutant' individuals. We need to check whether there are individuals that change their trait, both from A to B and vice versa, using the two `if` conditionals. If there are no such individuals, then assigning a new value to an empty tibble returns an error. To avoid this, we make sure that the number of rows is greater than 0 (using `nrow()>0` within the `if`).

To plot the results, we can use the same function `plot_multiple_runs()` we wrote in the previous chapter.

Let's now run and plot the model:

```
data_model <- unbiased_mutation(N = 100, mu = 0.05, p_0 = 0.5, t_max = 200, r_max = 5)
plot_multiple_runs(data_model)
```

Unbiased mutation produces random fluctuations over time and does not alter the overall frequency of A, which stays around $p = 0.5$. Because mutations from

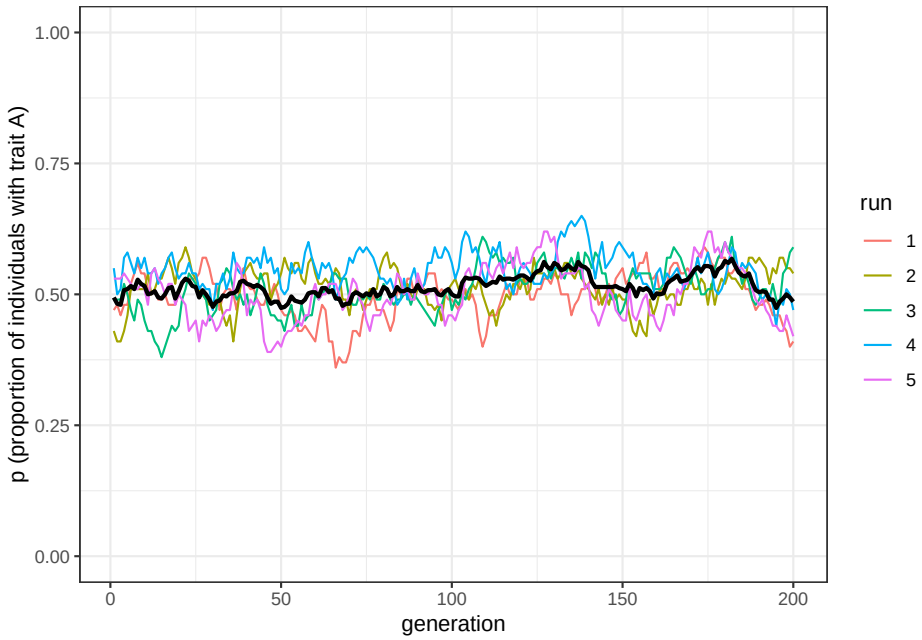


Figure 2.1: Trait frequencies fluctuate around 0.5 under unbiased mutation

A to B are as equally likely as B to A, there is no overall directional trend.

If you remember from the previous chapter, with unbiased transmission, when populations were small (e.g. $N = 100$) generally one of the traits disappeared after a few generations. Here, though, with $N = 100$, both traits remain until the end of the simulation. Why this difference? You can think of it in this way: when one trait becomes popular, say the frequency of A is equal to 0.8, with unbiased transmission it is more likely that individuals of the new generation will pick up A randomly when copying. The few individuals with trait B will have 80% probability of copying A. With unbiased mutation, on the other hand, since μ is applied independently to each individual, when A is common then there will be more individuals that will flip to B (specifically, μpN individuals, which in our case is 4) than individuals that will flip to A (equal to $\mu(1-p)N$ individuals, in our case 1) keeping the traits at similar frequencies.

But what if we were to start at different initial frequencies of A and B? Say, $p = 0.1$ and $p = 0.9$? Would A disappear? Would unbiased mutation keep p at these initial values, like we saw unbiased transmission does in Model 1?

To find out, let's change p_0, which specifies the initial probability of drawing an A rather than a B in the first generation.

```
data_model <- unbiased_mutation(N = 100, mu = 0.05, p_0 = 0.1, t_max = 200, r_max = 5)
plot_multiple_runs(data_model)
```

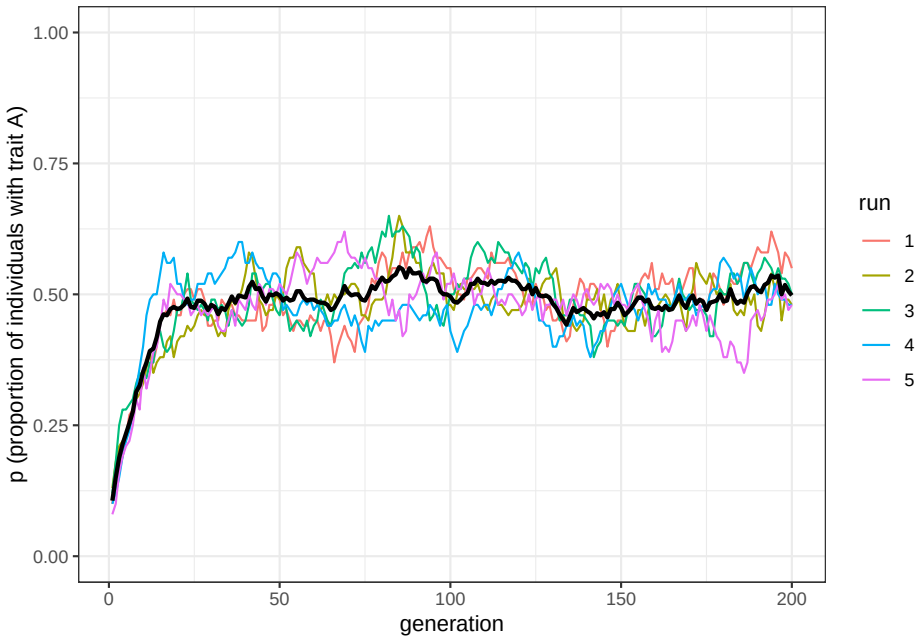


Figure 2.2: Unbiased mutation causes trait frequencies to converge on 0.5, irrespective of starting frequencies

You should see p go from 0.1 up to 0.5. In fact, whatever the initial starting frequencies of A and B, unbiased mutation always leads to $p = 0.5$, for the reason explained above: unbiased mutation always tends to balance the proportion of As and Bs.

2.2 Biased mutation

A more interesting case is biased mutation. Let's assume now that there is a probability μ_b that an individual with trait B mutates into A, but there is no possibility of trait A mutating into trait B. Perhaps trait A is a particularly catchy or memorable version of a story or an intuitive explanation of a phenomenon, and B is difficult to remember or unintuitive to understand.

The function `biased_mutation()` captures this unidirectional mutation.

```
biased_mutation <- function(N, mu_b, p_0, t_max, r_max) {
   # Create the output tibble
  output <- tibble(generation = rep(1:t_max, r_max),
                   p = as.numeric(rep(NA, t_max * r_max)),
                   run = as.factor(rep(1:r_max, each = t_max)))
```

```
  for (r in 1:r_max) {
    population <- tibble(trait = sample(c("A", "B"), N, replace = TRUE,
                                        prob = c(p_0, 1 - p_0)))
    # Add first generation's p for run r
    output[output$generation == 1 & output$run == r, ]$p <-
      sum(population$trait == "A") / N
    for (t in 2:t_max) {
      # Copy individuals to previous_population tibble
      previous_population <- population

      # Determine 'mutant' individuals
      mutate <- sample(c(TRUE, FALSE), N, prob = c(mu_b, 1 - mu_b), replace = TRUE)

      # If there are 'mutants' from B to A
      if (nrow(population[mutate & previous_population$trait == "B", ]) > 0) {
        # Then flip them to A
        population[mutate & previous_population$trait == "B", ]$trait <- "A"
      }
      # Get p and put it into output slot for this generation t and run r
      output[output$generation == t & output$run == r, ]$p <-
        sum(population$trait == "A") / N
    }
  }
  # Export data from function
  output
}
```

There are just two changes in this code compared to `unbiased_mutation()`. First, we've replaced `mu` with `mu_b` to keep the two parameters distinct and avoid confusion. Second, the line in `unbiased_mutation()` which caused individuals with A to mutate to B has been deleted.

Let's see what effect this has by running `biased_mutation()`. We'll start with the population entirely composed of individuals with B, i.e. $p_0 = 0$, to see how quickly and in what manner A spreads via biased mutation.

```
data_model <- biased_mutation(N = 100, mu_b = 0.05, p_0 = 0, t_max = 200, r_max = 5)
plot_multiple_runs(data_model)
```

The plot shows a steep increase that slows and plateaus at $p = 1$ by around generation $t = 100$. There should be a bit of fluctuation in the different runs, but not much. Now let's try a larger sample size.

```
data_model <- biased_mutation(N = 10000, mu_b = 0.05, p_0 = 0, t_max = 200, r_max = 5)
plot_multiple_runs(data_model)
```

With $N = 10000$ the line should be smooth with little (if any) fluctuation

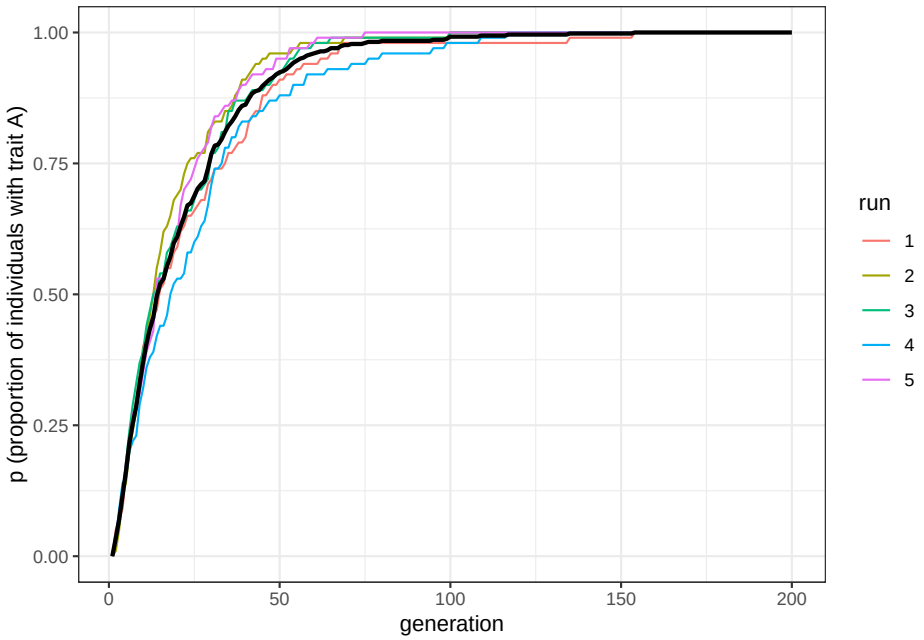


Figure 2.3: Biased mutation causes the favoured trait to replace the unfavoured trait

across the runs. But notice that it plateaus at about the same generation, around $t = 100$. Population size has little effect on the rate at which a novel trait spreads via biased mutation. μ_b, on the other hand, does affect this speed. Let's double the biased mutation rate to 0.1.

```
data_model <- biased_mutation(N = 10000, mu_b = 0.1, p_0 = 0, t_max = 200, r_max = 5)
plot_multiple_runs(data_model)
```

Now trait A reaches fixation around generation $t = 50$. Play around with N and μ_b to confirm that the latter determines the rate of diffusion of trait A, and that it takes the same form each time - roughly an 'r' shape with an initial steep increase followed by a plateauing at $p = 1$.

2.3 Summary of the model

With this simple model, we can draw the following insights. Unbiased mutation, which resembles genetic mutation in being non-directional, always leads to an equal mix of the two traits. It introduces and maintains cultural variation in the population. It is interesting to compare unbiased mutation to unbiased transmission from the previous chapter. While unbiased transmission did not change p over time, unbiased mutation always converges on $p = 0.5$, irrespective

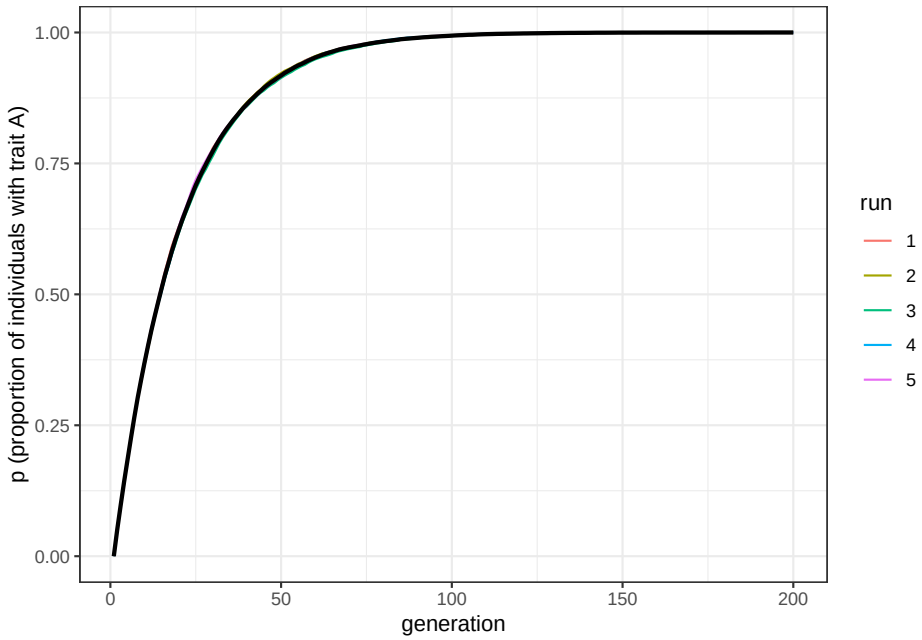


Figure 2.4: Increasing the population size does not change the rate at which biased mutation causes the favoured trait to increase in frequency

of the starting frequency. (NB $p = 0.5$ assuming there are two traits; more generally, $p = 1/v$, where v is the number of traits.)

Biased mutation, which is far more common - perhaps even typical - in cultural evolution, shows different dynamics. Novel traits favoured by biased mutation spread in a characteristic fashion - an r-shaped diffusion curve - with a speed characterised by the mutation rate μ_b. Population size has little effect, whether $N = 100$ or $N = 10000$. Whenever biased mutation is present ($\mu_b > 0$), the favoured trait goes to fixation, even if it is not initially present.

In terms of programming techniques, the major novelty in this model is the use of `sample()` to determine which individuals should undergo whatever the fixed probability specifies (in our case, mutation). This could be done with a loop, but vectorising code in the way we did here is much faster in R than loops.

2.4 Further reading

Boyd and Richerson [1985] model what they call 'guided variation', which is equivalent to biased mutation as modelled in this chapter. Henrich [2001] shows how biased mutation / guided variation generates r-shaped curves similar to those generated here.

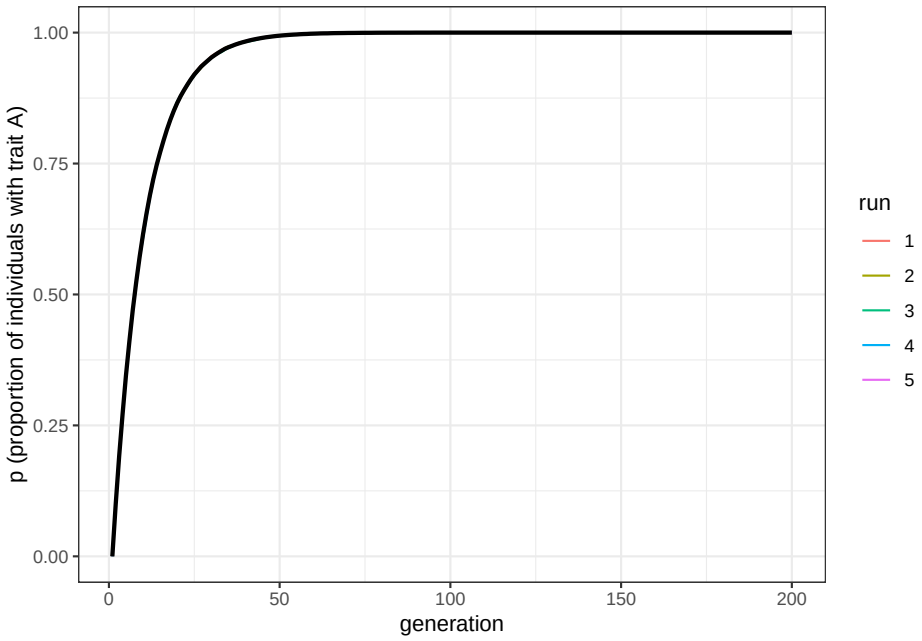


Figure 2.5: Increasing the mutation rate increases the rate at which biased mutation causes the favoured trait to increase in frequency

Chapter 3

Biased transmission: direct bias

So far we have looked at unbiased transmission (Chapter 1) and mutation, both unbiased and biased (Chapter 2). Let's complete the set by looking at biased transmission. This occurs when one trait is more likely to be copied than another trait. When the choice depends on the features of the trait, it is often called 'direct' or 'content' bias. When the choice depends on features of the demonstrators (the individuals from whom one is copying), it is often called 'indirect' or 'context' bias. Both are sometimes also called 'cultural selection' because one trait is selected to be copied over another trait. In this chapter, we will look at trait-based (direct, content) bias.

(As an aside, there is a confusing array of terminology in the field of cultural evolution, as illustrated by the preceding paragraph. That's why models are so useful. Words and verbal descriptions can be ambiguous. Often the writer doesn't realise that there are hidden assumptions or unrecognised ambiguities in their descriptions. They may not realise that what they mean by 'cultural selection' is entirely different from how someone else uses it. Models are great because they force us to precisely specify exactly what we mean by a particular term or process. We can use the words in the paragraph above to describe biased transmission, but it's only really clear when we model it, making all our assumptions explicit.)

3.1 A simple model of directly biased transmission

To simulate biased transmission, following the simulations in Chapter 1, we assume there are two traits A and B, and that each individual chooses another

individual from the previous generation at random. This time, however, we give the traits two different probabilities of being copied: we can call them s_a and s_b respectively. When an individual encounters another individual with trait A, they will copy them with probability s_a. When they encounter an individual with trait B, they will copy them with probability s_b.

With $s_a = s_b$, copying is unbiased, and individuals switch to the encountered alternative with the same probability. This reproduces the results of the simulations when the transmission is unbiased. If $s_a = s_b = 1$, the model is exactly the same as in Chapter 1. The relevant situation in this chapter is when $s_a > s_b$ (or $s_a < s_b$) so that we have biased transmission. Perhaps A (or B) is a more effective tool, a more memorable story, or a more easily pronounced word.

Let's first write the function, and then explore what happens in this case. Below is a function `biased_transmission_direct()` that implements all of these ideas.

```
library(tidyverse)

biased_transmission_direct <- function (N, s_a, s_b, p_0, t_max, r_max) {

  output <- tibble(generation = rep(1:t_max, r_max),
                   p = as.numeric(rep(NA, t_max * r_max)),
                   run = as.factor(rep(1:r_max, each = t_max)))

  for (r in 1:r_max) {
    # Create first generation
    population <- tibble(trait = sample(c("A", "B"), N,
                                        replace = TRUE, prob = c(p_0, 1 - p_0)))

    # Add first generation's p for run r
    output[output$generation == 1 & output$run == r, ]$p <-
      sum(population$trait == "A") / N

    for (t in 2:t_max) {
      # Copy individuals to previous_population tibble
      previous_population <- population

      # For each individual, pick a random individual from the previous generation
      demonstrator_trait <-
        tibble(trait = sample(previous_population$trait, N, replace = TRUE))

      # Biased probabilities to copy:
      copy_a <- sample(c(TRUE, FALSE), N, prob = c(s_a, 1 - s_a), replace = TRUE)
      copy_b <- sample(c(TRUE, FALSE), N, prob = c(s_b, 1 - s_b), replace = TRUE)

      # If the demonstrator has trait A and the individual wants to copy A, then copy
```

```
      if (nrow(population[copy_a & demonstrator_trait$trait == "A", ]) > 0) {
        population[copy_a & demonstrator_trait$trait == "A", ]$trait <- "A"
      }

      # If the demonstrator has trait B and the individual wants to copy B, then copy B
      if (nrow(population[copy_b & demonstrator_trait$trait == "B", ]) > 0) {
        population[copy_b & demonstrator_trait$trait == "B", ]$trait <- "B"
      }
      # Get p and put it into output slot for this generation t and run r
      output[output$generation == t & output$run == r, ]$p <-
        sum(population$trait == "A") / N
    }
  }
  # Export data from function
  output
}
```

Most of `biased_transmission_direct()` is recycled from the previous models. As before, we initialise the data structure `output` from multiple runs, and in generation $t = 1$, we create a `population` tibble to hold the trait of each individual.

The major change is that we now include biased transmission. We first select at random the demonstrators from the previous generation (using the same code we used in `unbiased_transmission()`) and we store their trait in `demonstrator_trait`. Then we get the probabilities for copying A and for copying B for the entire population, using the same code used in `biased_mutation()`. Again using the same code as in `biased mutation()`, we have the individuals copy the trait at hand with the desired probability.

Let's run our function `biased_transmission_direct()`. As before, to plot the results, we can use the function `plot_multiple_runs()` we wrote in the first chapter.

As noted above, the interesting case is when one trait is favoured over the other. We can assume, for example, $s_a = 0.1$ and $s_b = 0$. This means that when individuals encounter another individual with trait A they copy them 1 out every 10 times, but when individuals encounter another individual with trait B, they never switch. We can also assume that the favoured trait, A, is initially rare in the population ($p_0 = 0.01$) to see how selection favours this initially-rare trait (Note that p_0 needs to be higher than 0; since there is no mutation in this model, we need to include at least some As at the beginning of the simulation, otherwise it would never appear).

```
data_model <- biased_transmission_direct(N = 10000, s_a = 0.1, s_b = 0 ,
                                         p_0 = 0.01, t_max = 150, r_max = 5)
plot_multiple_runs(data_model)
```

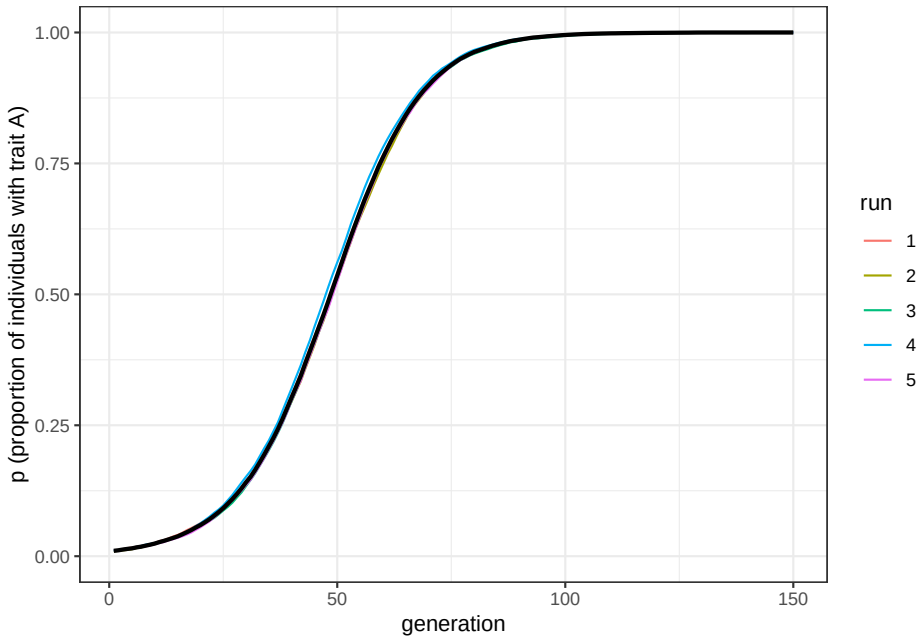


Figure 3.1: Biased transmission generates an s-shaped diffusion curve

With a moderate selection strength, we can see that A gradually replaces B and goes to fixation. It does this in a characteristic manner: the increase is slow at first, then picks up speed, then plateaus.

Note the difference from biased mutation. Where biased mutation was r-shaped, with a steep initial increase, biased transmission is s-shaped, with an initial slow uptake. This is because the strength of biased transmission (like selection in general) is proportional to the variation in the population. When A is rare initially, there is only a small chance of picking another individual with A. As A spreads, the chances of picking an A individual increases. As A becomes very common, there are few B individuals left to switch. In the case of biased mutation, instead, the probability of switching is independent of the variation in the population.

3.2 Strength of selection

On what does the strength of selection depend? First, the strength is independent of the specific values of s_a and s_b. What counts is their relative difference, which in the above case is $s_a - s_b = 0.1$. If we run a simulation with, say, $s_a = 0.6$ and $s_b = 0.5$, we see the same pattern, albeit with slightly more noise. That is, the single runs are more different from one another compared to the previous simulation. This is because switches from A to B are now also possible.

```
data_model <- biased_transmission_direct(N = 10000, s_a = 0.6, s_b = 0.5 ,
                                         p_0 = 0.01, t_max = 150, r_max = 5)
plot_multiple_runs(data_model)
```

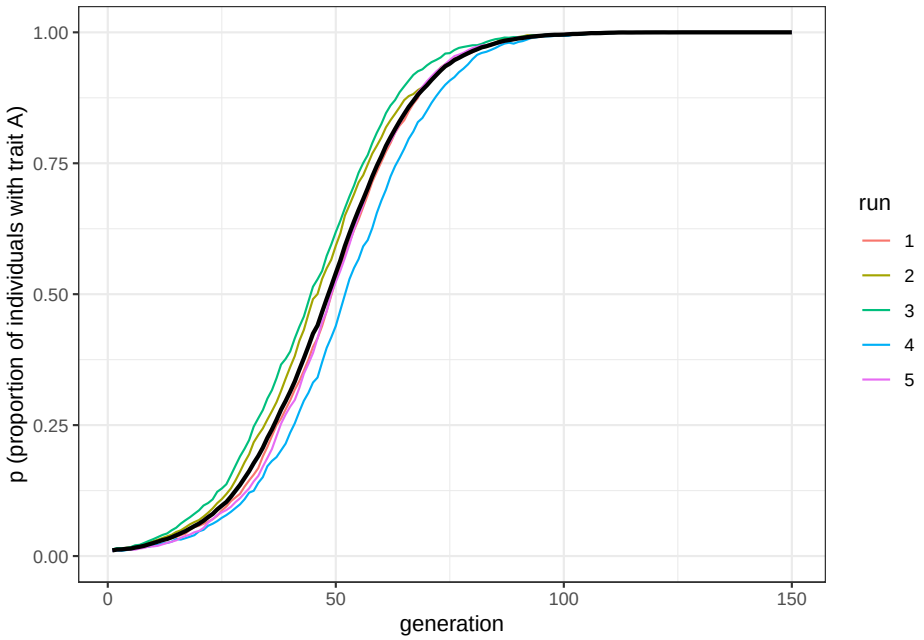


Figure 3.2: Biased transmission depends on the relative difference between the transmission parameters of each trait

To change the selection strength, we need to modify the difference between s_a and s_b. We can double the strength by setting $s_a = 0.2$, and keeping $s_b = 0$.

```
data_model <- biased_transmission_direct(N = 10000, s_a = 0.2, s_b = 0 ,
                                         p_0 = 0.01, t_max = 150, r_max = 5)
plot_multiple_runs(data_model)
```

As we might expect, increasing the strength of selection increases the speed with which A goes to fixation. Note, though, that it retains the s-shape.

3.3 Summary of the model

We have seen how biased transmission causes a trait favoured by cultural selection to spread and go to fixation in a population, even when it is initially very rare. Biased transmission differs in its dynamics from biased mutation. Its action is proportional to the variation in the population at the time at which it acts. It is strongest when there is lots of variation (in our model, when there

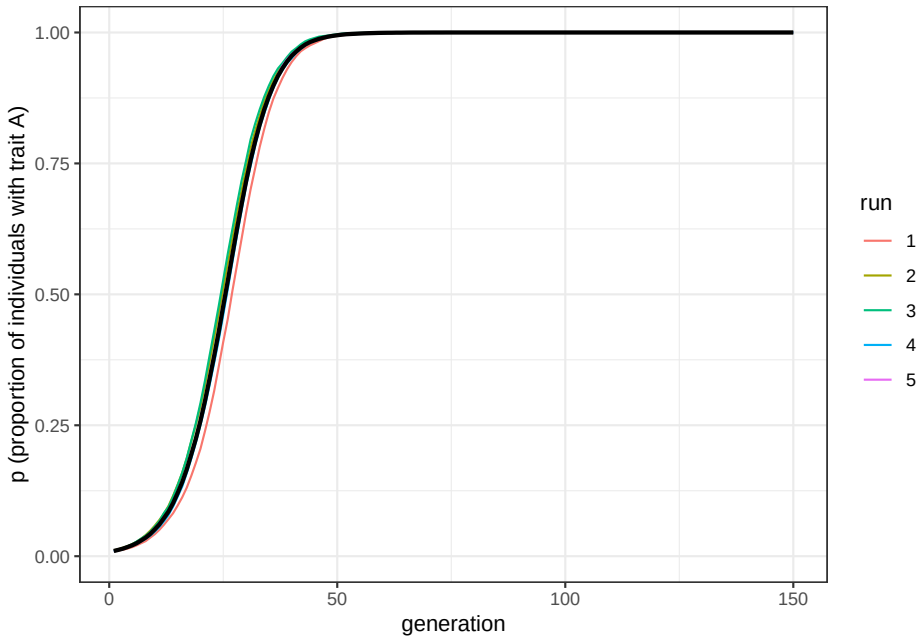


Figure 3.3: Increasing the relative difference between transmission parameters increases the rate at which the favoured trait spreads

are equal numbers of A and B at $p = 0.5$), and weakest when there is little variation (when p is close to 0 or 1).

3.4 Further reading

Boyd and Richerson [1985] modelled direct bias, while Henrich [2001] added directly biased transmission to his guided variation / biased mutation model, showing that this generates s-shaped curves similar to those generated here. Note though that subsequent work has shown that s-shaped curves can be generated via other processes (e.g. Reader [2004]), and should not be considered definite evidence for biased transmission.

Chapter 4

Biased transmission: frequency-dependent indirect bias

In Chapter 3 we looked at the case where one cultural trait is intrinsically more likely to be copied than another trait. Here we will start looking at the other kind of biased transmission when traits are equivalent, but individuals are more likely to adopt a trait according to the characteristics of the population, and in particular which other individuals already have it. (As we mentioned previously, these are often called 'indirect' or 'context' biases).

4.1 The logic of conformity

A first possibility is that we may be influenced by the frequency of the trait in the population, i.e. how many other individuals already have the trait. Conformity (or 'positive frequency-dependent bias') has been most studied. Here, individuals are disproportionately more likely to adopt the most common trait in the population, irrespective of its intrinsic characteristics. (The opposite case, anti-conformity or negative frequency-dependent bias is also possible, where the least common trait is more likely to be copied. This is probably less common in real life.)

For example, imagine trait A has a frequency of 0.7 in the population, with the rest possessing trait B. An unbiased learner would adopt trait A with a probability exactly equal to 0.7. This is unbiased transmission and is what happens the model described in (Chapter 1: by picking a member of the previous generation at random, the probability of adoption is equal to the frequency of that trait among the previous generation.

A conformist learner, on the other hand, would adopt trait A with a probability greater than 0.7. In other words, common traits get an 'adoption boost' relative to unbiased transmission. Uncommon traits get an equivalent 'adoption penalty'. The magnitude of this boost or penalty can be controlled by a parameter, which we will call D.

Let's keep things simple in our model. Rather than assuming that individuals sample across the entire population, which in any case might be implausible in large populations, let's assume they pick only three demonstrators at random. Why three? This is the minimum number of demonstrators that can yield a majority (i.e. 2 vs 1), which we need to implement conformity. When two demonstrators have one trait and the other demonstrator has a different trait, we want to boost the probability of adoption for the majority trait, and reduce it for the minority trait.

We can specify the probability of adoption as follows:

Table 1: Probability of adopting trait A for each possible combination of traits amongst three demonstrators

Demonstrator 1	Demonstrator 2	Demonstrator 3	Probability of adopting trait A
A	A	A	1
A	A	B	$2/3 + D/3$
A	B	A	$2/3 + D/3$
B	A	A	$2/3 + D/3$
A	B	B	$1/3 - D/3$
B	A	B	$1/3 - D/3$
B	B	A	$1/3 - D/3$
B	B	B	0

The first row says that when all demonstrators have trait A, then trait A is definitely adopted. Similarly, the bottom row says that when all demonstrators have trait B, then trait A is never adopted, and by implication trait B is always adopted.

For the three combinations where there are two As and one B, the probability of adopting trait A is $2/3$, which it would be under unbiased transmission (because two out of three demonstrators have A), plus the conformist adoption boost specified by D. As we want D to vary from 0 to 1, it is divided by three, so that the maximum probability of adoption is equal to 1 (when $D = 1$).

Similarly, for the three combinations where there are two Bs and one A, the probability of adopting A is $1/3$ minus the conformist adoption penalty specified

by D.

Let's implement these assumptions in the kind of individual-based model we've been building so far. As before, assume N individuals each of whom possesses one of two traits A or B. The frequency of A is denoted by p. The initial frequency of A in generation $t = 1$ is p_0. Rather than going straight to a function, let's go step by step.

First, we'll specify our parameters, N and p_0 as before, plus the new conformity parameter D. We also create the usual `population` tibble and fill it with As and Bs in the proportion specified by p_0, again exactly as before.

```
library(tidyverse)

N <- 100
p_0 <- 0.5
D <- 1

# Create first generation
population <- tibble(trait = sample(c("A", "B"), N,
                                    replace = TRUE, prob = c(p_0, 1 - p_0)))
```

Now we create another tibble called `demonstrators` that picks, for each new individual in the next generation, three demonstrators at random from the current population of individuals. It therefore needs three columns/variables, one for each of the demonstrators, and N rows, one for each individual. We fill each column with randomly chosen traits from the`population` tibble. We can have a look at `demonstrators` by entering its name in the R console.

```
# Create a tibble with a set of 3 randomly-picked demonstrators for each agent
demonstrators <- tibble(dem1 = sample(population$trait, N, replace = TRUE),
                        dem2 = sample(population$trait, N, replace = TRUE),
                        dem3 = sample(population$trait, N, replace = TRUE))

# Visualise the tibble
demonstrators
```

```
## # A tibble: 100 x 3
##    dem1  dem2  dem3
##    <chr> <chr> <chr>
##  1 A     B     B
##  2 B     B     B
##  3 A     B     B
##  4 A     B     B
##  5 B     A     B
##  6 A     A     B
##  7 B     B     A
##  8 A     B     A
```

```
##  9 B     B     B
## 10 B     B     B
## # ... with 90 more rows
```

Think of each row here as containing the traits of three randomly-chosen demonstrators chosen by each new next-generation individual. Now we want to calculate the probability of adoption of A for each of these three-trait demonstrator combinations.

First we need to get the number of As in each combination. Then we can replace the traits in `population` based on the probabilities in Table 1. When all demonstrators have A, we set to A. When no demonstrators have A, we set to B. When two out of three demonstrators have A, we set to A with probability $2/3 + D/3$ and B otherwise. When one out of three demonstrators have A, we set to A with probability $1/3 - D/3$ and B otherwise.

```
# Get the number of As in each 3-demonstrator combinations
num_As <- rowSums(demonstrators == "A")

# For 3-demonstrator combinations with all As, set to A
population$trait[num_As == 3] <- "A"
# For 3-demonstrator combinations with all Bs, set to B
population$trait[num_As == 0] <- "B"

prob_majority <- sample(c(TRUE, FALSE),
                        prob = c((2/3 + D/3), 1 - (2/3 + D/3)), N, replace = TRUE)
prob_minority <- sample(c(TRUE, FALSE),
                        prob = c((1/3 - D/3), 1 - (1/3 - D/3)), N, replace = TRUE)

# 3-demonstrator combinations with two As and one B
if (nrow(population[prob_majority & num_As == 2, ]) > 0) {
  population[prob_majority & num_As == 2, ] <- "A"
}
if (nrow(population[prob_majority == FALSE & num_As == 2, ]) > 0) {
  population[prob_majority == FALSE & num_As == 2, ] <- "B"
}

# 3-demonstrator combinations with one A and two Bs
if (nrow(population[prob_minority & num_As == 1, ]) > 0) {
  population[prob_minority & num_As == 1, ] <- "A"
}
if (nrow(population[prob_minority == FALSE & num_As == 1, ]) > 0) {
  population[prob_minority == FALSE & num_As == 1, ] <- "B"
}
```

To check it works, we can add the new `population` tibble as a column to `demonstrators` and have a look at it. This will let us see the three demonstra-

tors and the resulting new trait side by side.

```
demonstrators <- add_column(demonstrators, new_trait = population$trait)

# Visualise the tibble
demonstrators
```

```
## # A tibble: 100 x 4
##    dem1  dem2  dem3  new_trait
##    <chr> <chr> <chr> <chr>
##  1 A     B     B     B
##  2 B     B     B     B
##  3 A     B     B     B
##  4 A     B     B     B
##  5 B     A     B     B
##  6 A     A     B     A
##  7 B     B     A     B
##  8 A     B     A     A
##  9 B     B     B     B
## 10 B     B     B     B
## # ... with 90 more rows
```

Because we set $D = 1$ above, the new trait is always the majority trait among the three demonstrators. This is perfect conformity. We can weaken conformity by reducing D. Here is an example with $D = 0.5$. All the code is the same as what we already discussed above.

```
D <- 0.5

# create first generation
population <- tibble(trait = sample(c("A", "B"), N,
                                    replace = TRUE, prob = c(p_0, 1 - p_0)))

# Create a tibble with a set of 3 randomly-picked demonstrators for each agent
demonstrators <- tibble(dem1 = sample(population$trait, N, replace = TRUE),
                        dem2 = sample(population$trait, N, replace = TRUE),
                        dem3 = sample(population$trait, N, replace = TRUE))

# Get the number of As in each 3-demonstrator combinations
num_As <- rowSums(demonstrators == "A")

# For 3-demonstrator combinations with all As, set to A
population$trait[num_As == 3] <- "A"
# For 3-demonstrator combinations with all Bs, set to B
population$trait[num_As == 0] <- "B"

prob_majority <- sample(c(TRUE, FALSE),
```

```
                          prob = c((2/3 + D/3), 1 - (2/3 + D/3)), N, replace = TRUE)
prob_minority <- sample(c(TRUE, FALSE),
                          prob = c((1/3 - D/3), 1 - (1/3 - D/3)), N, replace = TRUE)

# 3-demonstrator combinations with two As and one B
if (nrow(population[prob_majority & num_As == 2, ]) > 0) {
  population[prob_majority & num_As == 2, ] <- "A"
}
if (nrow(population[prob_majority == FALSE & num_As == 2, ]) > 0) {
  population[prob_majority == FALSE & num_As == 2, ] <- "B"
}

# 3-demonstrator combinations with one A and two Bs
if (nrow(population[prob_minority & num_As == 1, ]) > 0) {
  population[prob_minority & num_As == 1, ] <- "A"
}
if (nrow(population[prob_minority == FALSE & num_As == 1, ]) > 0) {
  population[prob_minority == FALSE & num_As == 1, ] <- "B"
}

demonstrators <- add_column(demonstrators, new_trait = population$trait)

# Visualise the tibble
demonstrators
```

```
## # A tibble: 100 x 4
##    dem1  dem2  dem3  new_trait
##    <chr> <chr> <chr> <chr>
##  1 A     A     A     A
##  2 B     B     A     B
##  3 B     A     B     B
##  4 B     A     A     A
##  5 B     A     A     B
##  6 A     A     A     A
##  7 B     A     B     B
##  8 B     A     B     A
##  9 A     B     A     A
## 10 B     B     B     B
## # ... with 90 more rows
```

Now that conformity is weaker, sometimes the new trait is not the majority amongst the three demonstrators.

4.2 Testing conformist transmission

As in the previous chapters, we can put all this code together into a function to see what happens over multiple generations and in multiple runs. There is nothing new in the code below, which is a combination of the code we already wrote in (Chapter 1) and the new bits of code for conformity introduced above.

```
conformist_transmission <- function (N, p_0, D, t_max, r_max) {

  output <- tibble(generation = rep(1:t_max, r_max),
                   p = as.numeric(rep(NA, t_max * r_max)),
                   run = as.factor(rep(1:r_max, each = t_max)))

  for (r in 1:r_max) {
    # Create first generation
    population <- tibble(trait = sample(c("A", "B"), N,
                                        replace = TRUE, prob = c(p_0, 1 - p_0)))

    # Add first generation's p for run r
    output[output$generation == 1 & output$run == r, ]$p <-
      sum(population$trait == "A") / N

    for (t in 2:t_max) {

      # Create a tibble with a set of 3 randomly-picked demonstrators for each agent
      demonstrators <- tibble(dem1 = sample(population$trait, N, replace = TRUE),
                              dem2 = sample(population$trait, N, replace = TRUE),
                              dem3 = sample(population$trait, N, replace = TRUE))

      # Get the number of As in each 3-demonstrator combinations
      num_As <- rowSums(demonstrators == "A")

      # For 3-demonstrator combinations with all As, set to A
      population$trait[num_As == 3] <- "A"
      # For 3-demonstrator combinations with all Bs, set to B
      population$trait[num_As == 0] <- "B"

      prob_majority <- sample(c(TRUE, FALSE),
                              prob = c((2/3 + D/3), 1 - (2/3 + D/3)), N, replace = TRUE)
      prob_minority <- sample(c(TRUE, FALSE),
                              prob = c((1/3 - D/3), 1 - (1/3 - D/3)), N, replace = TRUE)

      # 3-demonstrator combinations with two As and one B
      if (nrow(population[prob_majority & num_As == 2, ]) > 0) {
        population[prob_majority & num_As == 2, ] <- "A"
      }
```

```
      if (nrow(population[prob_majority == FALSE & num_As == 2, ]) > 0) {
        population[prob_majority == FALSE & num_As == 2, ] <- "B"
      }
      # 3-demonstrator combinations with one A and two Bs
      if (nrow(population[prob_minority & num_As == 1, ]) > 0) {
        population[prob_minority & num_As == 1, ] <- "A"
      }
      if (nrow(population[prob_minority == FALSE & num_As == 1, ]) > 0) {
        population[prob_minority == FALSE & num_As == 1, ] <- "B"
      }

      # Get p and put it into output slot for this generation t and run r
      output[output$generation == t & output$run == r, ]$p <-
        sum(population$trait == "A") / N
    }
  }
  # Export data from function
  output
}
```

We can test the function with perfect conformity ($D = 1$) and plot it (again we use the function `plot_multiple_runs()` we wrote in Chapter 1).

```
data_model <- conformist_transmission(N = 1000, p_0 = 0.5, D = 1, t_max = 50, r_max = 1
plot_multiple_runs(data_model)
```

Here we should see some lines going to $p = 1$, and some lines going to $p = 0$. Conformity acts to favour the majority trait. This will depend on the initial frequency of A in the population. In different runs with $p_0 = 0.5$, sometimes there will be slightly more As, sometimes slightly more Bs (remember, in our model, this is probabilistic, like flipping coins, so initial frequencies will rarely be precisely 0.5).

What happens if we set $D = 0$?

```
data_model <- conformist_transmission(N = 1000, p_0 = 0.5, D = 0, t_max = 50, r_max = 1
plot_multiple_runs(data_model)
```

This model is equivalent to unbiased transmission. As for the simulations described in Chapter 1, with a sufficiently large N, the frequencies fluctuate around $p = 0.5$. This underlines the effect of conformity. With unbiased transmission, majority traits are favoured because they are copied in proportion to their frequency (incidentally, it is for this reason that 'copying the majority' is not a good description of conformity in the technical sense used in the field of cultural evolution: even with unbiased copying the majority trait is copied more than the minority one). However, they reach fixation only in small populations. With conformity, instead, the majority trait is copied with a probability higher

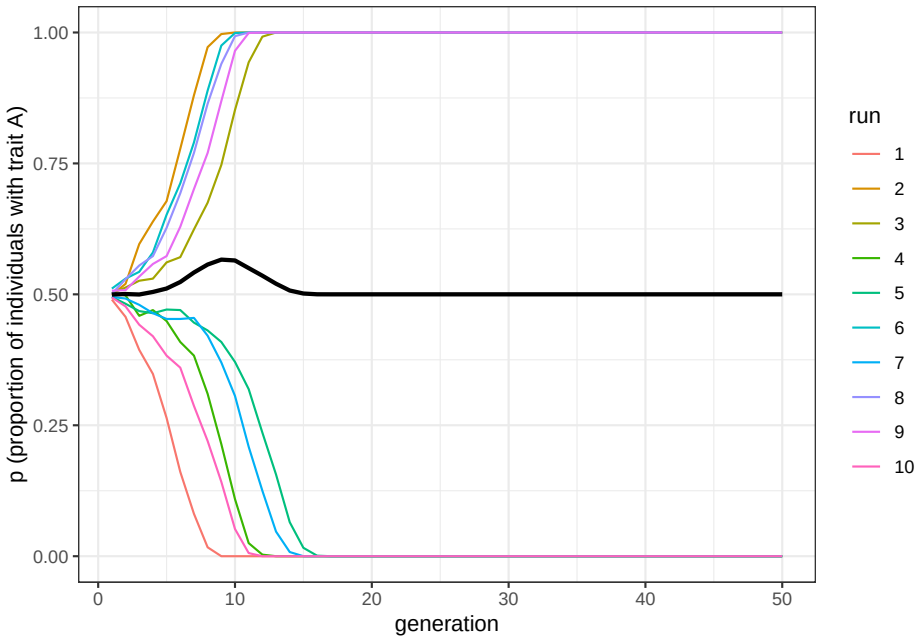


Figure 4.1: Conformity causes one trait to spread and replace the other by favouring whichever trait is initially most common

than its frequency, so that conformity drives traits to fixation as they become more and more common.

As an aside, note that the last two graphs have roughly the same thick black mean frequency line, which hovers around $p = 0.5$. This highlights the dangers of looking at means alone. If we hadn't plotted the individual runs and relied solely on mean frequencies, we might think that $D = 0$ and $D = 1$ gave identical results. But in fact, they are very different. Always look at the underlying distribution that generates means.

Now let's explore the effect of changing the initial frequencies by changing p_0, and adding conformity back in.

```
data_model <- conformist_transmission(N = 1000, p_0 = 0.55, D = 1, t_max = 50, r_max = 10)
plot_multiple_runs(data_model)
```

When A starts with a slight majority ($p_0 = 0.55$), all (or almost all) of the runs result in A going to fixation. Now let's try the reverse.

```
data_model <- conformist_transmission(N = 1000, p_0 = 0.45, D = 1, t_max = 50, r_max = 10)
plot_multiple_runs(data_model)
```

When A starts off in a minority ($p_0 = 0.45$), all (or almost all) of the runs result in A disappearing. These last two graphs show how initial conditions

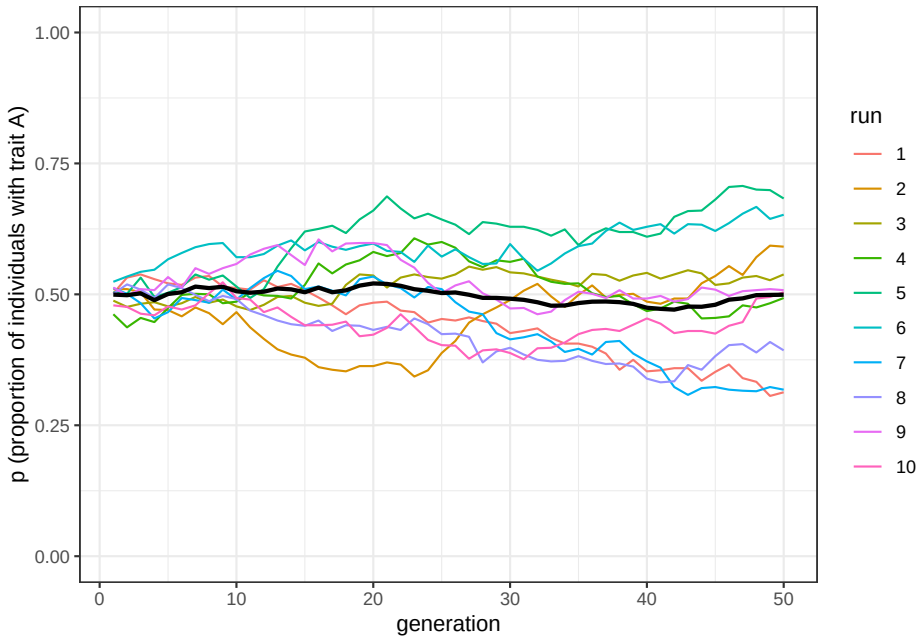


Figure 4.2: Removing conformist bias recreates unbiased transmission, and does not systematically change trait frequencies

affect conformity. Whichever trait is more common is favoured by conformist transmission.

4.3 Summary of the model

In this chapter, we explored conformist biased cultural transmission. This is where individuals are disproportionately more likely to adopt the most common trait among a set of demonstrators. We can contrast this indirect bias with the direct (or content) biased transmission from [Chapter 3][Biased transmission (direct bias)], where one trait is intrinsically more likely to be copied. With conformity, the traits have no intrinsic difference in attractiveness and are preferentially copied simply because they are common.

We saw how conformity increases the frequency of whichever trait is more common. Initial trait frequencies are important here: traits that are initially more common typically go to fixation. This, in turn, makes stochasticity important, which in small populations can affect initial frequencies.

We also discussed the subtle but fundamental difference between unbiased copying and conformity. In both, majority traits are favoured, but it is only with conformity that they are *disproportionately* favoured. In large populations, unbiased transmission rarely leads to trait fixation, whereas conformist transmission

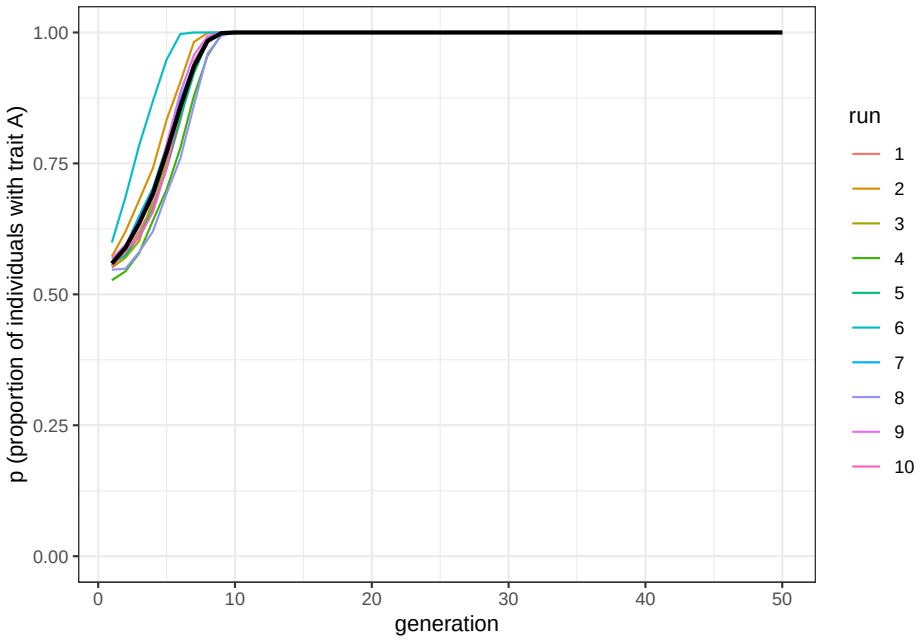


Figure 4.3: When trait A is always initially in the majority, it is always favoured by conformity

often does. Furthermore, as we will see later, conformity also makes majority traits resistant to external disturbances, such as the introduction of other traits via innovation or migration.

4.4 Further readings

Boyd and Richerson [1985] introduced conformist or positive frequency-dependent cultural transmission as defined here, and modelled it analytically with similar methods. Henrich and Boyd [1998] modelled the evolution of conformist transmission, while Efferson et al. [2008] provided experimental evidence that at least some people conform in a simple learning task.

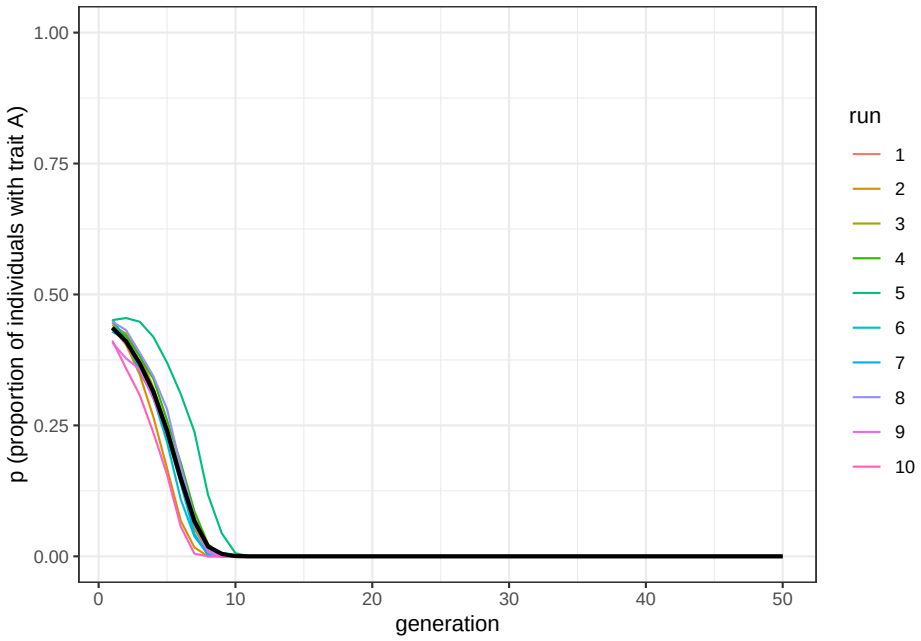


Figure 4.4: When trait B is always initially in the majority, it is always favoured by conformity

Chapter 5

Biased transmission: demonstrator-based indirect bias

In the previous two chapters we examined two forms of biased transmission, one where the bias arises due to characteristics of the traits (or direct bias) and another where the bias arises due to the characteristics of the population (or indirect bias). In the previous chapter we examined frequency-dependent indirect bias which takes into account the frequency of the trait (or conformity). Here we examine indirect bias that takes into account specific features of the demonstrators. This demonstrator-based bias is also called 'model bias' or 'context bias' in the cultural evolution literature.

Whereas the simulations we created previously are fairly standard, indirect demonstrator-based biases can be implemented in several ways. Demonstrator biases result whenever individuals decide whether or not to copy by taking into account any features of the demonstrators, as long as it is not directly tied to the traits. The most studied demonstrator bias is prestige bias, where individuals are more likely to copy from demonstrators who are considered more 'prestigious' or high in subjective social status, for example because other individuals show deference to them. Alternatively, individuals can copy demonstrators who are more successful according to some objective criterion (e.g. wealth) independently from how others judge them, or they can copy individuals that are more similar to themselves, or older (or younger) than themselves, and so on. The key point is that the decision is not directly linked to the cultural trait itself, and relates to some characteristic of the demonstrator(s) from whom one is copying.

5.1 A simple demonstrator bias

To implement a simple version of demonstrator-biased cultural transmission, we first need to assume that there are some intrinsic differences between individuals within the population. Up until now, our individuals have only been described by the traits they possess. We now want individuals to have some additional feature which others can use when deciding whether to copy that individual. We call this feature 'status'. For simplicity, an individual's status is a binary variable that could stand for whether they are prestigious or not, successful or not, and so on. We define a parameter p_s that determines the probability that an individual has high status, as opposed to low status.

```
library(tidyverse)

N <- 100
p_0 <- 0.5
p_s <- 0.05

population <- tibble(trait = sample(c("A", "B"), N,
                                    replace = TRUE, prob = c(p_0, 1 - p_0)),
                     status = sample(c("high", "low"), N,
                                     replace = TRUE, prob = c(p_s, 1 - p_s)))
```

We can inspect the tibble by typing its name in the R console

```
population
```

```
## # A tibble: 100 x 2
##    trait status
##    <chr> <chr>
##  1 B     low
##  2 B     low
##  3 B     low
##  4 B     low
##  5 A     high
##  6 A     low
##  7 A     low
##  8 A     low
##  9 B     low
## 10 A     low
## # ... with 90 more rows
```

With $p_s = 0.05$ around 5 individuals in a population of 100 will have high status.

We now need to make it so that these rare high status individuals are more likely to be copied. One way of doing this is to assume that the probabilities of picking high-status and low-status individuals as demonstrators are different. So far, when using the function `sample()` to select demonstrators, we did not

include any specific probability. This meant that each individual of the previous generation had the same likelihood of being selected and copied. Instead, now we pass to the function a vector of probabilities to weight the choice.

We assume that the probability if selecting low-status individuals is given by a further parameter, p_{low}, that gives the proportion between the probabilities of choosing a low-status individual versus an high-status individual. When $p_{\text{low}} = 1$, the simulations correspond to unbiased transmission, as everybody has the same probability of being chosen. When $p_{\text{low}} = 0$, there is a strict status-based demonstrator bias, where only high-status individuals are ever selected as demonstrators.

To implement this, we first store in `p_demonstrator` the probabilities of being copied for each member of the population:

```
p_low <- 0.01
p_demonstrator <- rep(1, N)
p_demonstrator[population$status == "low"] <- p_low
```

Then we sample the traits in the population using these probabilities. Notice the condition `if(sum(p_demonstrator) > 0)`. This is necessary in case there are no high-status individuals (for example when $p_s \approx 0$) and the probability of selecting a low status demonstrator to copy is 0 ($p_{\text{low}} = 0$). This would make the overall probability equal to 0, and without including this control the model would generate an error.

```
if(sum(p_demonstrator) > 0){

  demonstrator_index <- sample (N, prob = p_demonstrator, replace = TRUE)
  population$trait <- population$trait[demonstrator_index]

}
```

As usual, we can wrap everything in a function.

```
biased_transmission_demonstrator <- function(N, p_0, p_s, p_low, t_max, r_max) {

  output <- tibble(generation = rep(1:t_max, r_max),
                   p = as.numeric(rep(NA, t_max * r_max)),
                   run = as.factor(rep(1:r_max, each = t_max)))

  for (r in 1:r_max) {
    # Create first generation
    population <- tibble(trait = sample(c("A", "B"), N,
                                        replace = TRUE, prob = c(p_0, 1 - p_0)),
                         status = sample(c("high", "low"), N,
                                         replace = TRUE, prob = c(p_s, 1 - p_s)))

    # Assign copying probabilities based on individuals' status
```

```
    p_demonstrator <- rep(1,N)
    p_demonstrator[population$status == "low"] <- p_low

    # Add first generation's p for run r
    output[output$generation == 1 & output$run == r, ]$p <-
      sum(population$trait == "A") / N

    for (t in 2:t_max) {
      # Copy individuals to previous_population tibble
      previous_population <- population

      # Copy traits based on status
      if(sum(p_demonstrator) > 0){
        demonstrator_index <- sample (N, prob = p_demonstrator, replace = TRUE)
        population$trait <- previous_population$trait[demonstrator_index]
      }
      # Get p and put it into output slot for this generation t and run r
      output[output$generation == t & output$run == r, ]$p <-
        sum(population$trait == "A") / N
    }

  }
  # Export data from function
  output
}
```

We can now test our simulation, assuming a very low, but not zero, probability of selecting low-status individuals as demonstrators. We are using the usual `plot_multiple_runs()` function to plot the results of the simulations.

```
data_model <- biased_transmission_demonstrator(N = 100, p_s = 0.05, p_low=0.0001,
                                               p_0 = 0.5, t_max = 50, r_max = 10)
plot_multiple_runs(data_model)
```

The results are similar to what we saw in the previous chapter for conformity: one of the two traits quickly reaches fixation. In the case of conformity, however, the trait reaching fixation was the one that happened to have a slightly higher frequency at the beginning, because of the random initialisation. With a demonstrator bias, this is not the case.

From this perspective, an indirect demonstrator-based bias is more similar to unbiased transmission. If you remember from the first chapter, simulations with unbiased transmission also generally ended up with one trait reaching fixation in small populations ($N = 100$), but in bigger ones ($N = 10000$) the frequencies of the two traits remained around $p = 0.5$. What happens with demonstrator-based bias?

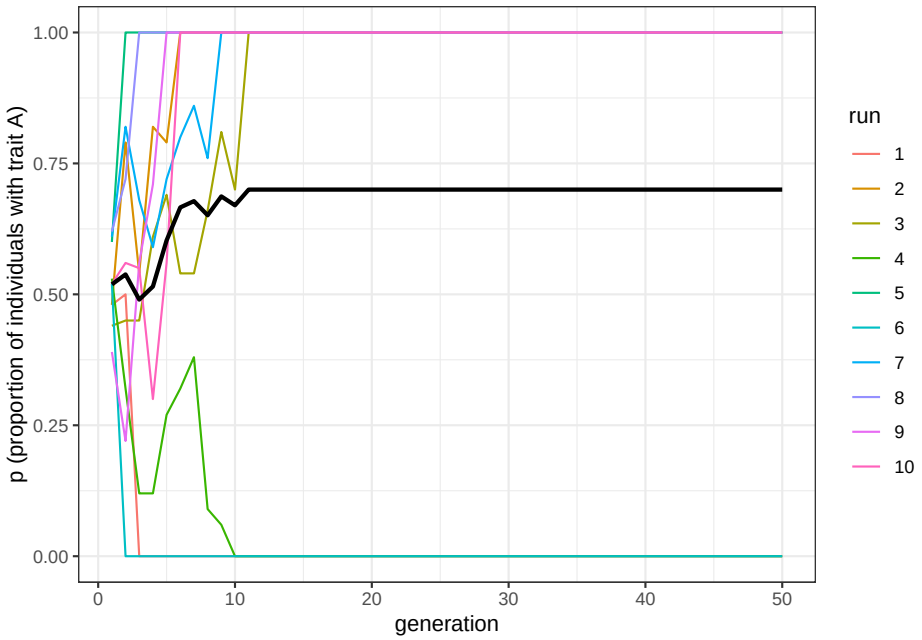


Figure 5.1: Indirectly biased transmission causes one trait to spread and the other to be lost.

```
data_model <- biased_transmission_demonstrator(N = 10000, p_s = 0.005, p_low=0.0001,
                                               p_0 = 0.5, t_max = 200, r_max = 10)
plot_multiple_runs(data_model)
```

Even with $N = 10000$, if the number of high-status individuals is sufficiently low, as in this case ($p_s = 0.005$ means that, on average, 50 individuals are high-status in each run), traits reach fixation. By reducing the pool of demonstrators, demonstrator-based bias makes drift more important for the overall dynamics. The pool of high-status demonstrators (equal to Np_s) is the effective population size, which is much smaller than the actual population size (N).

You can experiment with different values of p_s and p_{low}. How big can the pool of high-status demonstrators be before the dynamics become indistinguishable from unbiased transmission?

5.2 Predicting the 'winning' trait

With conformity, as just mentioned, the trait that reaches fixation is the one starting out in the majority. With unbiased transmission the trait that goes to fixation cannot be predicted at the beginning of the simulation. With a demonstrator-based bias, a reasonable guess would be that the 'winning' trait

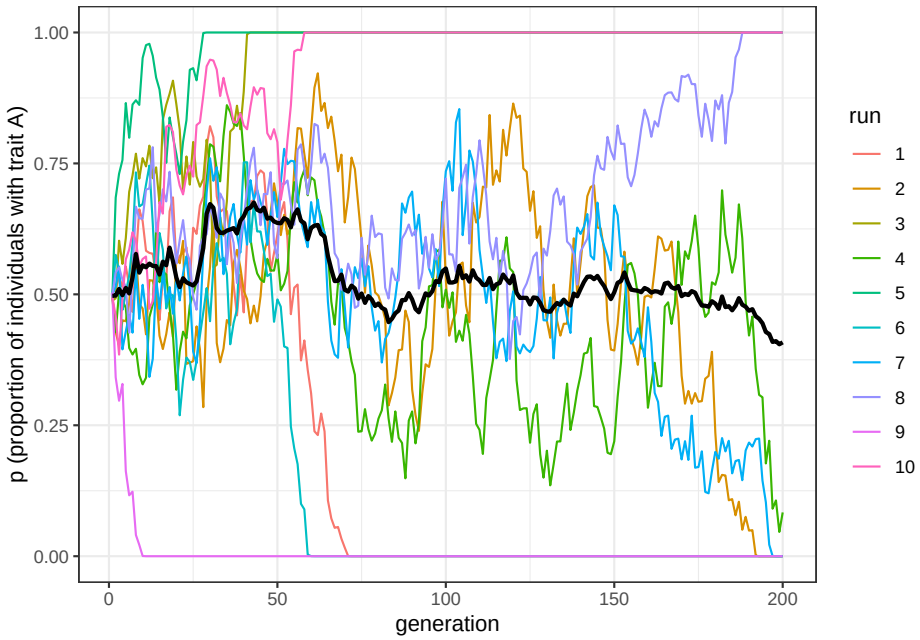


Figure 5.2: Indirectly biased transmission depends on the effective population size, not the overall population size.

is the one that is, at the beginning, most common among the high-status individuals. Can we check this intuition with our model?

Currently the output we obtain from the simulations is not suitable for this purpose. On the one hand, we do not have the crucial piece of information that we need: the proportion of each trait among the high-status individuals when the population is initialised. On the other hand, we have much information that we do not need, such as the frequency of the two traits at each time step. However, we just want to know which traits reach fixation. We can therefore rewrite the `biased_transmission_demonstrator()` function and change the `output` tibble to suit our needs.

```
biased_transmission_demonstrator_2 <- function(N, p_0, p_s, p_low, t_max, r_max) {

  output <- tibble(status_A = as.numeric(rep(NA, r_max)),
                   p = as.numeric(rep(NA, r_max)))

  for (r in 1:r_max) {

    # Create first generation
    population <- tibble(trait = sample(c("A", "B"), N,
                                        replace = TRUE, prob = c(p_0, 1 - p_0)),
```

```
                          status = sample(c("high", "low"), N,
                                          replace = TRUE, prob = c(p_s, 1 - p_s)))

    # Assign copying probabilities based on individuals' status
    p_demonstrator <- rep(1,N)
    p_demonstrator[population$status == "low"] <- p_low

    # Add first generation's frequency of high-status individuals with traits A for run r
    output[r, ]$status_A <-
      sum(population$status == "high" & population$trait == "A") /
      sum(population$status == "high")

    for (t in 2:t_max) {

      # Copy individuals to previous_population tibble
      previous_population <- population

      # Copy traits based on status
      if(sum(p_demonstrator) > 0){
        demonstrator_index <- sample (N, prob = p_demonstrator, replace = TRUE)
        population$trait <- previous_population$trait[demonstrator_index]
      }
    }
  # Get p at the end of the run
  output[r, ]$p <- sum(population$trait == "A") / N
  }
  output # export data from function
}
```

Here, `status_A` gives the starting frequency of A among the high status individuals. p, as before, gives the frequency of A in the entire population, but we only record this value at the very end of the simulation, to see if one trait has gone to fixation. The tibble `output`, as a consequence, has now only $r_{\max}$ rows.

Let's run the new function, `biased_transmission_demonstrator_2()`, for 50 runs (setting `r_max = 50`) so that we have more independent data points, and inspect the output.

```
data_model <- biased_transmission_demonstrator_2(N = 100, p_s = 0.05, p_low=0.0001,
                                                 p_0 = 0.5, t_max = 50, r_max = 50)
data_model
```

```
## # A tibble: 50 x 2
##    status_A     p
##       <dbl> <dbl>
##  1    0         0
##  2    0.5       0
```

```
##  3    1         1
##  4    0.333     1
##  5    0.667     1
##  6    1         1
##  7    1         1
##  8    0.167     1
##  9    0.5       0
## 10    0.667     1
## # ... with 40 more rows
```

Each line of the output is a run of the simulation. The first column (labelled `status_A`) gives the frequency of A in the high-status individuals, and the second (labelled `p`) the frequency of A at the end of the simulation. From a cursory inspection of the output, you should be able to see that our guess was correct, and when `status_A` is higher than 0.5, `p` should be generally 1, and when `status_A` is less than 0.5, `p` should be generally 0. But let's visualise all the data to be sure.

We want to know how the initial proportion of high-status individuals is related to the two possible outcomes (trait A reaches fixation or trait B reaches fixation). A convenient way is to use a boxplot. In the code below, we first eliminate the runs where the traits did not reach fixation (if they exist) using the new function `filter()`, and, for clarity, we assign the trait name A or B to each run according to which trait reached fixation. We can then plot our output.

The main novelties in this code are the new ggplot 'geoms' `geom_boxplot()` and `geom_jitter()`. Whereas boxplots are useful to detect aggregate information on our simulations, `geom_jitter()` plots all of the data points, so we can get a better idea of how the proportions of high-status individuals are distributed in the various runs. We could have done this with our usual `geom_point()`, but `geom_jitter()` scatters randomly the points in the plot (at a distance specified by the parameter `width`). This avoids the overlapping of individual data points (known as overplotting).

```
# Filter only lines where p is equal to 1 or to 0
data_model <- filter(data_model, p == 1 | p == 0)

data_model$p <- as.character(data_model$p)
# Call "A" the runs where p is equal to 1
data_model[data_model$p==1, ]$p <- "A"
# Call "B" the runs where p is equal to 0
data_model[data_model$p==0, ]$p <- "B"

ggplot(data = data_model, aes(x = p, y = status_A, fill = p)) +
  geom_boxplot() +
  geom_jitter(width = 0.05) +
  labs(y = "proportion of high-status individuals with trait A",
```

```
       x = "winning trait") +
  theme_bw() +
  theme(legend.position = "none")
```

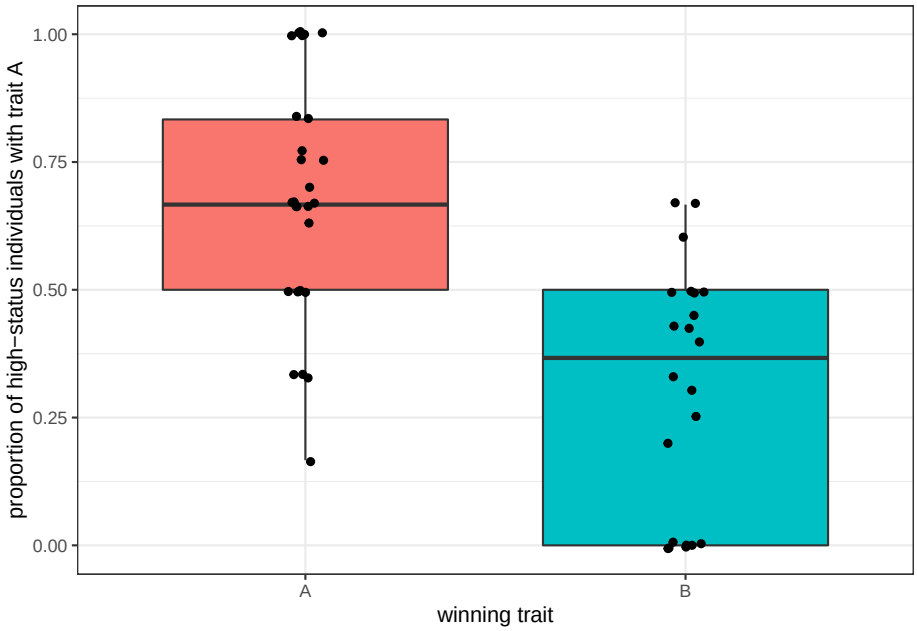


Figure 5.3: The trait reaching fixation tends to be the trait that was in majority among high-status individuals.

The plot shows that when trait A reaches fixation there are more high-status individuals with trait A at the beginning, and vice versa for B, confirming our intuition. However, this is far from being a safe bet. Runs with only a quarter of high-status individuals with A ended up with all As in the population and, conversely, runs with 80% of high-status individuals with A ended up with the fixation of B. With bigger populations (e.g. with $N = 10000$), it is even worse.

```
data_model <- biased_transmission_demonstrator_2(N = 10000, p_s = 0.005, p_low=0.0001,
                                                 p_0 = 0.5, t_max = 200, r_max = 50)

# Filter only lines where p is equal to 1 or to 0
data_model <- filter(data_model, p == 1 | p == 0)

data_model$p <- as.character(data_model$p)
# Call "A" the runs where p is equal to 1
data_model[data_model$p==1, ]$p <- "A"
# Call "B" the runs where p is equal to 0
data_model[data_model$p==0, ]$p <- "B"
```

```
ggplot(data = data_model, aes(x = p, y = status_A, fill = p)) +
  geom_boxplot() +
  geom_jitter(width = 0.05) +
  labs(y = "proportion of high-status individuals with trait A",
       x = "winning trait") +
  ylim(c(0,1)) +
  theme_bw() +
  theme(legend.position = "none")
```

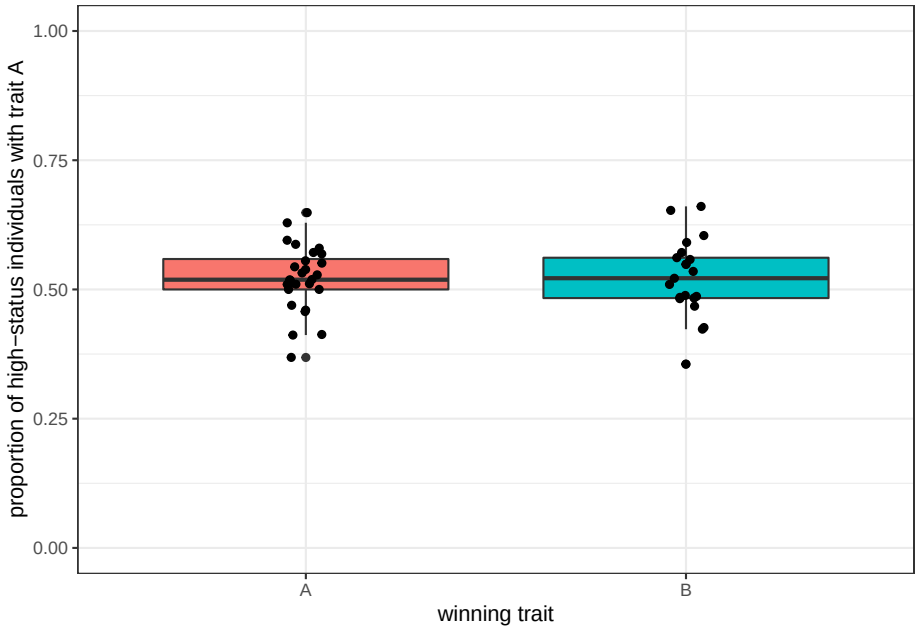


Figure 5.4: With bigger populations - and bigger pools of high-status demonstrators is more difficult to predict the winning trait.

With $N = 10000$ and around 50 high-status individuals, the traits are more equally distributed among 'influential' demonstrators at the beginning, and there is hardly any difference in the two outcomes.

5.3 Summary of the model

In this chapter we modeled an example of indirectly-biased or demonstrator-biased transmission. We assumed that a fraction of individuals in the population were 'high-status' and thus more likely to be selected as demonstrators. The results show that in this situation a trait is likely to become predominant even when populations are large. This is due to the fact that a demonstrator

bias effectively reduces the pool of demonstrators and accelerates convergence through a similar process as drift / unbiased transmission.

We also saw that the possibility of predicting which trait will become predominant depends on the number of high-status demonstrators. When there are few high-status demonstrators, then the most common trait amongst these high-status demonstrators will likely go to fixation. When their number increases, it is more difficult to make such a prediction.

We also saw how it is important to modify the output of a model depending on the question we are interested in. We used a novel ggplot aesthetic to produce a boxplot, a convenient way of displaying the distribution of data among different groups.

5.4 Further readings

Examples of simulation models implementing indirect, demonstrator-based, biased transmission include Mesoudi [2009], an individual-based model that explores how prestige bias can generate clusters of recurring behaviours, applied to the case of copycat suicides. Henrich Joseph et al. [2015] presents a population-level model that links prestige to the emergence of within-group cooperation. Henrich [2004] describes an analytical, population-level, model, where individuals copy the most successful demonstrator in the population.

An earlier analytical treatment of demonstrator-based bias, with extensions on the evolution of symbolic traits that may be associated to demonstrators is in Chapter 8 of Boyd and Richerson [1985].

Finally, Henrich and Gil-White [2001] is the classic treatment of prestige bias, and a recent review of the empirical evidence supporting it is Jiménez and Mesoudi [2019].

Chapter 6

Vertical and horizontal transmission

An important distinction in cultural evolution concerns the pathway of cultural transmission. Vertical cultural transmission occurs when individuals learn from their parents. Oblique cultural transmission occurs when individuals learn from other (non-parental) members of the older generation, such as teachers. Horizontal cultural transmission occurs when individuals learn from members of the same generation.

These terms (vertical, oblique and horizontal) are borrowed from epidemiology, where they are used to describe the transmission of diseases. Cultural traits, like diseases, are interesting in that they have multiple pathways of transmission. While genes spread purely vertically (at least in species like ours; horizontal gene transfer is common in plants and bacteria), cultural traits can spread obliquely and horizontally. These latter pathways can increase the rate at which cultural traits can spread, compared to vertical transmission alone.

In this chapter we will simulate and test this claim, focusing in particular on horizontal cultural transmission: when and why does horizontal transmission increase the rate of spread of a cultural trait compared to vertical cultural transmission?

6.1 Vertical cultural transmission

To simulate vertical cultural transmission we need to decide how people learn from their parents, assuming those two parents possess different combinations of cultural traits. As in previous models, we assume two discrete traits, A and B. There are then four combinations of traits amongst two parents: both parents have A, both parents have B, mother has A and father has B, and mother has

B and father has A.

For simplicity, we can assume that when both parents have the same trait, the child adopts that trait. When parents differ, the child faces a choice. To make things more interesting, let's assume a bias for one trait over the other in such situations (otherwise we would be back to unbiased transmission, and no trait would reliably spread - remember we are interested in how quickly traits spread under vertical vs horizontal transmission).

Hence we assume a probability b that, when parents differ in their traits such that there is some uncertainty, the child adopts A. With probability $1 - b$ they adopt trait B. When $b = 0.5$, transmission is unbiased. When $b > 0.5$, A should be favoured; when $b < 0.5$, B should be favoured. Let's simulate this and test these predictions.

The following function `vertical_transmission()` is very similar to previous simulation functions. The explanation follows.

```
library(tidyverse)

vertical_transmission <- function(N, p_0, b, t_max, r_max) {
  output <- tibble(generation = rep(1:t_max, r_max),
                   p = as.numeric(rep(NA, t_max * r_max)),
                   run = as.factor(rep(1:r_max, each = t_max)))

  for (r in 1:r_max) {
    # Create first generation
    population <- tibble(trait = sample(c("A", "B"), N,
                                        replace = TRUE, prob = c(p_0, 1 - p_0)))

    # Add first generation's p for run r
    output[output$generation == 1 & output$run == r, ]$p <-
      sum(population$trait == "A") / N

    for (t in 2:t_max) {
       # Copy individuals to previous_population tibble
      previous_population <- population

      # Randomly pick mothers and fathers
      mother <- tibble(trait = sample(previous_population$trait, N, replace = TRUE))
      father <- tibble(trait = sample(previous_population$trait, N, replace = TRUE))

      # Prepare next generation
      population <- tibble(trait = as.character(rep(NA, N)))

      # Both parents are A, thus child adopts A
      both_A <- mother$trait == "A" & father$trait == "A"
```

```
      if (sum(both_A) > 0) {
        population[both_A, ]$trait <- "A"
      }

      # Both parents are B, thus child adopts B
      both_B <- mother$trait == "B" & father$trait == "B"
      if (sum(both_B) > 0) {
        population[both_B, ]$trait <- "B"
      }
      # If any empty NA slots (i.e. one A and one B parent) are present
      if (anyNA(population)) {
        # They adopt A with probability b
        population[is.na(population)[,1],]$trait <-
          sample(c("A", "B"), sum(is.na(population)), prob = c(b, 1 - b), replace = TRUE)
      }

      # Get p and put it into output slot for this generation t and run r
      output[output$generation == t & output$run == r, ]$p <-
        sum(population$trait == "A") / N
    }
  }
  # Export data from function
  output
}
```

First we set up an output tibble to store the frequency of A (p) over $t_{\max}$ generations and across $r_{\max}$ runs. As before we create a `population` tibble to store our N traits, one per individual.

This time, however, in each generation we create two new tibbles, `mother` and `father`. These store the traits of two randomly chosen individuals from the `previous_population`, one pair for each new individual. Note that we are assuming random mating here: parents pair up entirely at random. Alternative mating rules are possible, such as assortative cultural mating, where parents preferentially assort based on their cultural trait. We will leave it to readers to create models of this.

Once the `mother` and `father` tibbles are created, we can fill in the new individuals' traits in `population`. `both_A` is used to mark with `TRUE` whether both mother and father have trait A, and (assuming some such cases exist), sets all individuals in `population` for whom this is true to have trait A. `both_B` works equivalently for parents who both possess trait B.

The remaining cases (identified as still being NA in the `population` tibble, with the function `anyNA()`) must have one A and one B parent. We are not concerned with which parent has which in this simple model, so in each of these cases we set the individual's trait to be A with probability b and B with probability $1-b$.

Again, we leave it to readers to modify the code to have separate probabilities for maternal and paternal transmission.

Once all generations are finished, we export the `output` tibble as our data. We can use our existing function `plot_multiple_runs()` from previous chapters to plot the results.

And now run both functions to see what happens. Remember we are interested in how fast the favoured trait spreads, so let's start it off at a low frequency ($p_0 = 0.01$) so we can see it spreading from rarity. We use a small transmission bias $b = 0.6$ favouring A.

```
data_model <- vertical_transmission(N = 10000, p_0 = 0.01, b = 0.6,
                                    t_max = 50, r_max = 5)
plot_multiple_runs(data_model)
```

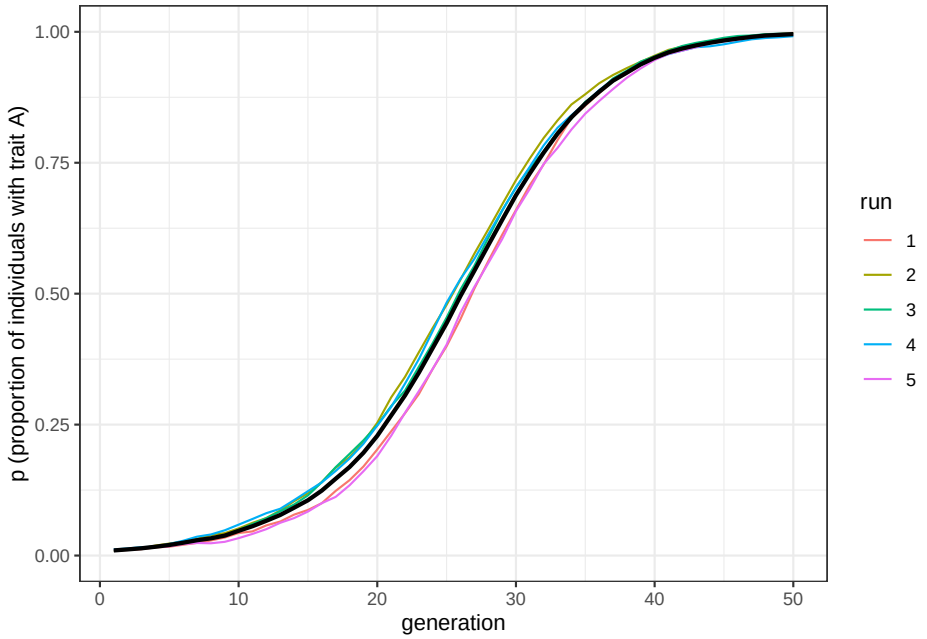


Figure 6.1: The favourite trait, A, spreads in the population under vertical transmission.

Here we can see a gradual spread of the favoured trait A from $p = 0.01$ to $p = 1$. As in our directly biased transmission model, the diffusion curve is s-shaped. To obtain the same result with two different models is encouraging! We can also test our prediction that when $b = 0.5$, we recreate our unbiased transmission model from Chapter 1:

```
data_model <- vertical_transmission(N = 10000, p_0 = 0.1, b = 0.5,
                                    t_max = 50, r_max = 5)
```

```
plot_multiple_runs(data_model)
```

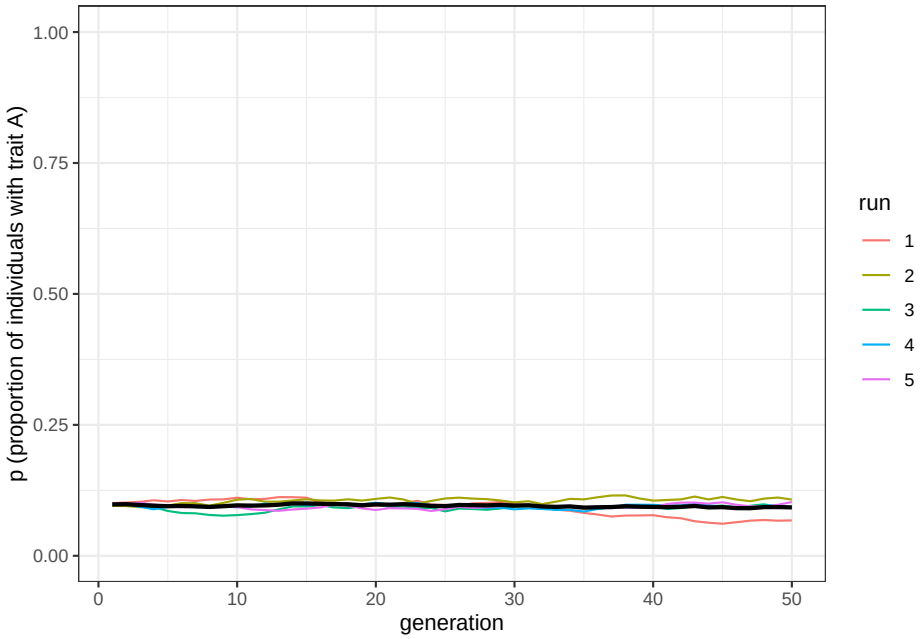


Figure 6.2: When no trait is favoured, there is no change in the frequency of trait A under vertical transmission.

As predicted, there is no change in starting trait frequencies when $b = 0.5$. If you reduce the sample size, you will see much more fluctuation across the runs, with some runs losing A altogether.

6.2 Horizontal cultural transmission

Now let's add horizontal cultural transmission to our model. We will add it to vertical cultural transmission, rather than replace vertical with horizontal, so we can compare both in the same model.

First there is vertical transmission as above, with random mating and the parental bias b, to create a new generation. Then, the new generation learns from each other. The key difference between vertical and horizontal transmission is that horizontal cultural transmission can occur from more than two individuals. Let's assume individuals pick n other individuals from their generation. We also assume a bias in favour of A during horizontal transmission. If the learner is B, then for each of the n demonstrators who have A, there is an independent probability g that the learner switches to A. If the learner is already A, or if the demonstrator is B, then nothing happens.

The following code implements this horizontal transmission in a new function `vertical_horizontal_transmission()`.

```
vertical_horizontal_transmission <- function(N, p_0, b, n, g, t_max, r_max) {
  output <- tibble(generation = rep(1:t_max, r_max),
                   p = as.numeric(rep(NA, t_max * r_max)),
                   run = as.factor(rep(1:r_max, each = t_max)))

  for (r in 1:r_max) {
    # Create first generation
    population <- tibble(trait = sample(c("A", "B"), N,
                                        replace = TRUE, prob = c(p_0, 1 - p_0)))

    # Add first generation's p for run r
    output[output$generation == 1 & output$run == r, ]$p <-
      sum(population$trait == "A") / N

    for (t in 2:t_max) {
      # Vertical transmission ------------------------------------------------

      # Copy individuals to previous_population tibble
      previous_population <- population

      # Randomly pick mothers and fathers
      mother <- tibble(trait = sample(previous_population$trait, N, replace = TRUE))
      father <- tibble(trait = sample(previous_population$trait, N, replace = TRUE))

      # Prepare next generation
      population <- tibble(trait = as.character(rep(NA, N)))

      # Both parents are A, thus child adopts A
      both_A <- mother$trait == "A" & father$trait == "A"
      if (sum(both_A) > 0) {
        population[both_A, ]$trait <- "A"
      }

      # Both parents are B, thus child adopts B
      both_B <- mother$trait == "B" & father$trait == "B"
      if (sum(both_B) > 0) {
        population[both_B, ]$trait <- "B"
      }
      # If any empty NA slots (i.e. one A and one B parent) are present
      if (anyNA(population)) {
        # They adopt A with probability b
        population[is.na(population)[,1],]$trait <-
          sample(c("A", "B"), sum(is.na(population)), prob = c(b, 1 - b), replace = TRU
```

```
      }

      # Horizontal transmission ------------------------------------------------

      # Previous_population are children before horizontal transmission
      previous_population <- population

      # N_B = number of Bs
      N_B <- length(previous_population$trait[previous_population$trait == "B"])

      # If there are B individuals to switch, and n is not zero
      if (N_B > 0 & n > 0) {
        # For each B individual...
        for (i in 1:N_B) {
          # Pick n demonstrators
          demonstrator <- sample(previous_population$trait, n, replace = TRUE)
          # Get probability g
          copy <- sample(c(TRUE, FALSE), n, prob = c(g, 1-g), replace = TRUE)
          # if any demonstrators with A are to be copied
          if ( sum(demonstrator == "A" & copy == TRUE) > 0 ) {
            # The B individual switches to A
            population[previous_population$trait == "B",]$trait[i] <- "A"
          }

        }
      }
      # Get p and put it into output slot for this generation t and run r
      output[output$generation == t & output$run == r, ]$p <-
        sum(population$trait == "A") / N
    }
  }
  # Export data from function
  output
}
```

The first part of this code is identical to `vertical_transmission()`. Then there is horizontal transmission. We put `population` into `previous_population` again, but now `population` contains the individuals after horizontal transmission, and `previous_population` contains the individuals before. `N_B` holds the individuals in `previous_population` who are B, as they are the only ones we need to concern ourselves with (A individuals do not change). If there are such individuals ($N_B > 0$), and individuals are learning from at least one individual ($n > 0$), then for each individual we pick n demonstrators, and if any of those demonstrators are A plus probability g is fulfilled, we set the individual to A.

Running horizontal transmission with $n = 5$ and $g = 0.1$ and without vertical

transmission bias ($b = 0.5$) causes, as expected, A to spread.

```
data_model <- vertical_horizontal_transmission(N = 5000, p_0 = 0.01, b = 0.5, n = 5,
                                               g = 0.1, t_max = 50, r_max = 5)
plot_multiple_runs(data_model)
```

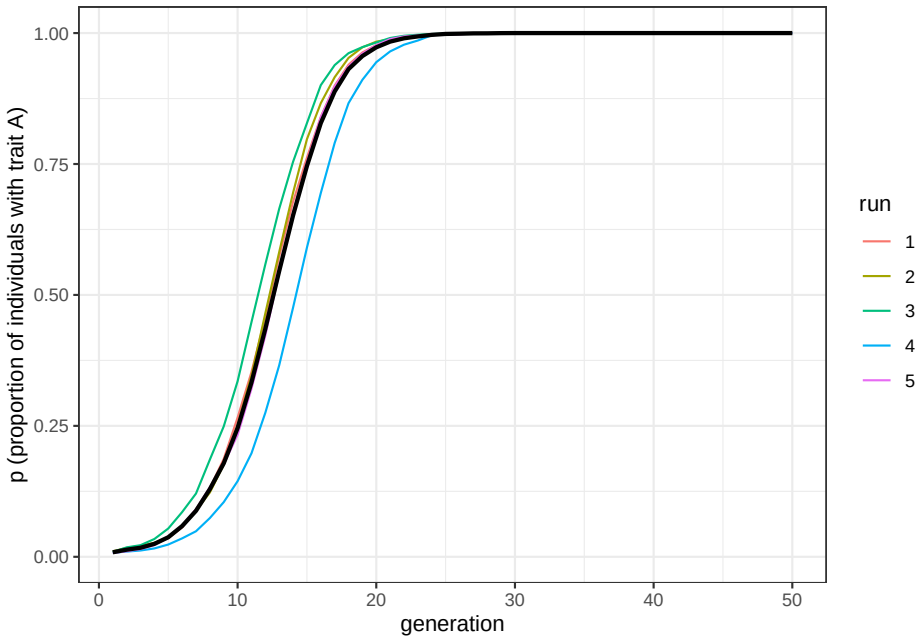


Figure 6.3: The favourite trait, A, spreads in the population under horizontal transmission.

This plot above confirms that horizontal cultural transmission, with some direct bias in the form of g, again generates an s-shaped curve and causes the favoured trait to spread. But we haven't yet done what we set out to do, which is compare the speed of the different pathways. The following code generates three datasets, one with only vertical transmission and $b = 0.6$, one with only horizontal transmission with $n = 2$ and $g = 0.1$ which is roughly equivalent to two parents and a bias of $b = 0.6$ (0.1 higher than unbiased), and one with only horizontal transmission with $n = 5$ and $g = 0.1$.

```
data_model_v <- vertical_horizontal_transmission(N = 5000, p_0 = 0.01, b = 0.6, n = 0,
                                                 g = 0, t_max = 50, r_max = 5)

data_model_hn2 <- vertical_horizontal_transmission(N = 5000, p_0 = 0.01, b = 0.5, n = 1
                                                   g = 0.1, t_max = 50, r_max = 5)

data_model_hn5 <- vertical_horizontal_transmission(N = 5000, p_0 = 0.01, b = 0.5, n = 5
                                                   g = 0.1, t_max = 50, r_max = 5)
```

```
plot_multiple_runs(data_model_v)
```

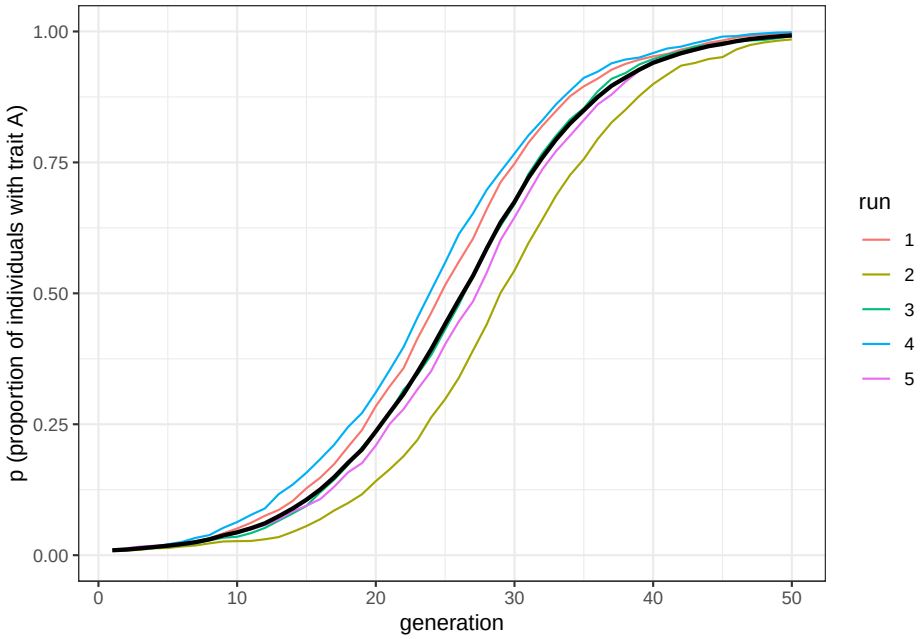


Figure 6.4: The favourite trait, A, spreads in the population under vertical transmission only.

```
plot_multiple_runs(data_model_hn2)
```

```
plot_multiple_runs(data_model_hn5)
```

The first two plots should be very similar. Horizontal cultural transmission from $n = 2$ demonstrators is equivalent to vertical cultural transmission, which of course also features two demonstrators, when both pathways have similarly strong direct biases. The third plot shows that increasing the number of demonstrators makes favoured traits spread more rapidly under horizontal transmission, without changing the strength of the biases. Of course, changing the relative strength of the vertical and horizontal biases (b and g respectively) also affects the relative speed. But all else being equal, horizontal transmission with $n > 2$ is faster than vertical transmission.

6.3 Summary of the model

This model has combined directly biased transmission with vertical and horizontal transmission pathways. The vertical transmission model recreates the patterns from our previous unbiased and directly biased transmission, but explicitly

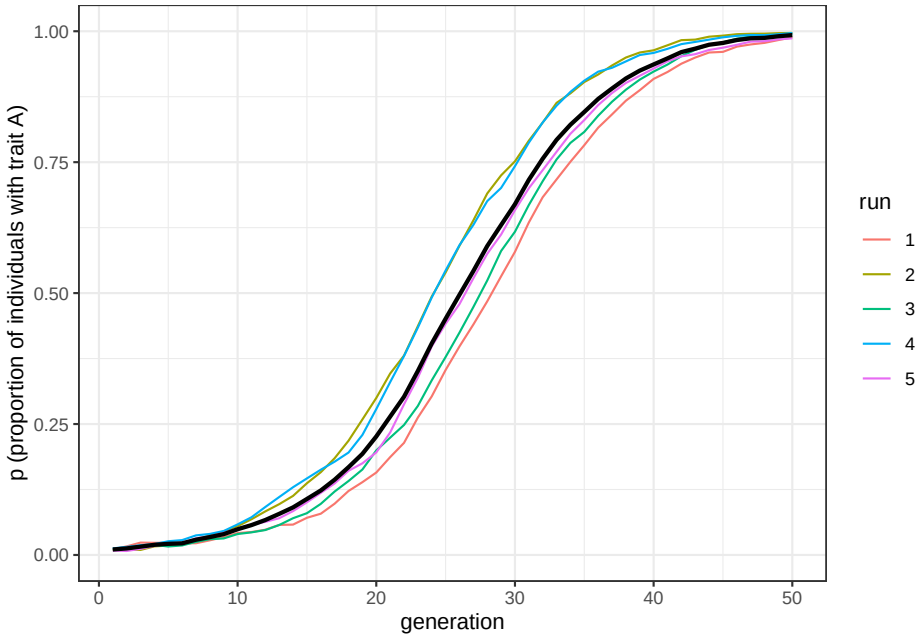


Figure 6.5: Given an equivalent bias strength and two demonstrators, the favourite trait, A, spreads under horizontal transmission at the same speed than in the vertical transmission scenario.

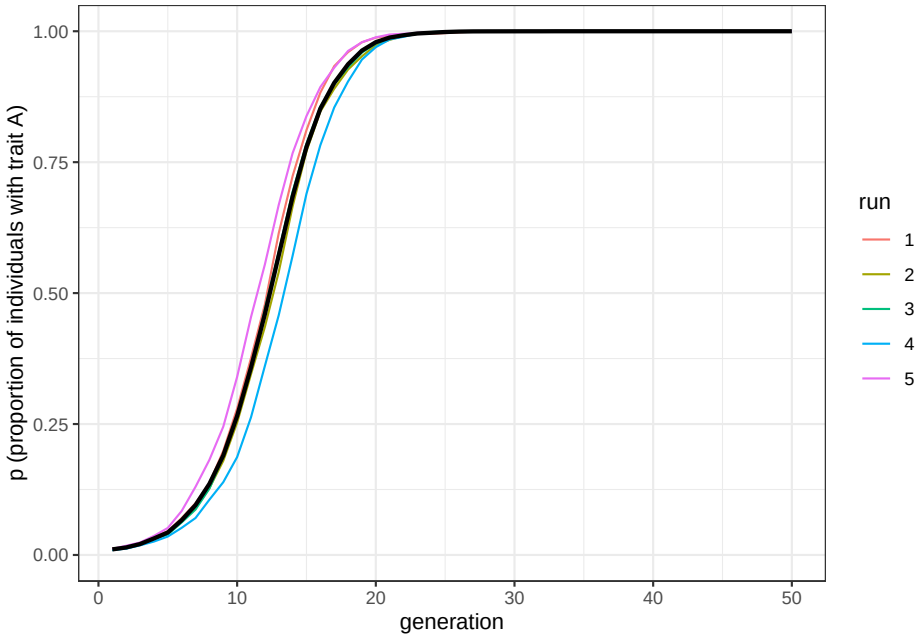


Figure 6.6: Given an equivalent bias strength and five demonstrators, the favourite trait, A, spreads under horizontal transmission faster than in the vertical transmission scenario.

modelling parents and their offspring. Although there were no differences, our vertical transmission model could be modified easily to study different kinds of parental bias (e.g. making maternal influence stronger than paternal influence), or different types of non-random mating.

Our horizontal transmission model is similar to the conformist bias simulated in Chapter 4, but slightly different - there is no disproportionate majority copying, and instead one trait is favoured when learning from n demonstrators. Comparing the two pathways, we can see that horizontal cultural transmission is faster than vertical cultural transmission largely because it allows individuals to learn from more than two demonstrators.

6.4 Further reading

The above models are based on those by Cavalli-Sforza and Feldman [1981]. Their vertical cultural transmission models feature bias parameters for each combination of matings (b_0, b_1, b_2 and b_3); our b is their b_1 and b_2. Their horizontal transmission model also features n and g, which have the same definitions as here. Subsequent models in that volume examine assortative cultural mating and oblique transmission, although the latter is similar to horizontal transmission.

Chapter 7

Multiple traits models

In all previous models, individuals could possess one of only two cultural traits, A or B. This is a useful simplification, and it represents cases in which cultural traits can be modeled as binary choices, such as voting Republican or Democrat, driving on the left or the right, or being vegetarian or meat-eating. In other cases, however, there are many options: in many countries there are multiple political parties to vote for, there may be many dietary choices (vegan, pescatarian, vegetarian, etc), and so on. What happens when we copy others' choices given more than two alternatives? To simplify this question, we again assume unbiased copying as in Chapter 1: all traits are functionally equivalent and other individuals are copied at random.

7.1 Unbiased transmission with multiple traits

The first modification we need to make in the code concerns how traits are represented. Since we have an undetermined number of possible traits we cannot use the two letters A and B. Instead we will use numbers, referring to trait "1", trait "2", trait "3", etc. How can we distribute the traits in the initial population? We can assume that there are m possible traits at the beginning, with $m \leq N$ (as usual, N is the population size). In all the following simulations, we will fix $m = N$, and effectively initialise each individual with a trait randomly chosen between "1" and "100".

```
library(tidyverse)
N <- 100
population <- tibble(trait = sample(1:N, N, replace = TRUE))
```

You can inspect the `population` tibble by writing its name.

```
population
```

```
## # A tibble: 100 x 1
##    trait
##    <int>
##  1   100
##  2    88
##  3    97
##  4    70
##  5    70
##  6    61
##  7    62
##  8    53
##  9    81
## 10    72
## # ... with 90 more rows
```

The basic code of the simulation is similar to the code in the first chapter, but what should the `output` be? Until now, we generally just needed to save the frequency of one of the two traits, because the frequency of the other was always one minus the first's frequency. Now we need the frequencies of all N traits. (Technically, we only need to track $N - 1$ frequencies, with the last inferred by substracting the other frequencies from 1. But for simplicity we'll track all of the frequencies.)

Second, how do we measure the frequency of the traits in each generation? The base R function `tabulate()` does this for us. `tabulate()` counts the number of times each element of a vector (`population$trait` in our case) occurs in the bins that we also pass to the function. In our case the bins are 1 to N. Since we want the frequencies, and not the absolute number, we divide the result by N.

```
multiple_traits <- function(N, t_max) {

  output <- tibble(trait = as.factor(rep(1:N, each = t_max)),
                   generation = rep(1:t_max, N),
                   p = as.numeric(rep(NA, t_max * N)))

  # Create first generation
  population <- tibble(trait = sample(1:N, N, replace = TRUE))

  # Add first generation's p for all traits
  output[output$generation == 1, ]$p <- tabulate(population$trait, nbins = N) / N

  for (t in 2:t_max) {
    # Copy individuals to previous_population tibble
    previous_population <- population

    # Randomly copy from previous generation
```

```
    population <- tibble(trait = sample(previous_population$trait, N, replace = TRUE))

    # Get p for all traits and put it into output slot for this generation t
    output[output$generation == t, ]$p <- tabulate(population$trait, nbins = N) / N
  }
  # Export data from function
  output
}
```

Finally, the function to plot the output is similar to what we have already done when plotting multiple runs. The one difference is that now the colored lines do not represent different runs, but different traits, as indicated below by `aes(colour = trait)`. The new line `theme(legend.position = "none")` simply tells ggplot to not include the legend in the graph, as it is not informative. It would just show 100 or more colors, one for each trait.

```
plot_multiple_traits <- function(data_model) {
  ggplot(data = data_model, aes(y = p, x = generation)) +
    geom_line(aes(colour = trait)) +
    ylim(c(0, 1)) +
    theme_bw() +
    theme(legend.position = "none")
}
```

As usual, we can call the function and see what happens:

```
data_model <- multiple_traits(N = 100, t_max = 200)
plot_multiple_traits(data_model)
```

Usually, only one or two traits are still present in the population after 200 generations, and, if we increase $t_{\max}$ for example to 1000, virtually all runs end up with only a single trait reaching fixation:

```
data_model <- multiple_traits(N = 100, t_max = 1000)
plot_multiple_traits(data_model)
```

This is similar to what we saw with only two traits, A and B: with unbiased copying and relatively small populations, drift is a powerful force and quickly erodes cultural diversity.

As we already discussed, increasing N reduces the effect of drift. You can experiment with various values for N and $t_{\max}$. However, the general point is that variation is gradually lost in all cases. How can we counterbalance the homogenizing effect that drift has in small and isolated population, such as the one we are simulating?

Figure 7.1: With small populations, the majority of traits disappear after few generations in a model with multiple traits and unbiased transmission.

7.2 Introducing innovation

One option is to introduce new traits via innovation. We can imagine that, at each time step, a proportion of individuals, μ, introduces a new trait in the population. We use the same notation that we used for mutation in Chapter 2: you can think that 'mutation' is when an individual change its trait for one that is already present, whereas an 'innovation' happens when an individual introduces a new trait never seen before. The remaining proportion of individuals, $1 - \mu$, copy at random from others, as before. We can start with a small value, such as $\mu = 0.01$. Since $N = 100$, this means that in each generation, on average, one new trait will be introduced into the population.

The following code adds innovation to the multiple-trait code from above:

```
mu <- 0.01
# Record the last trait introduced in the population
last_trait <- max(population)
# Copy the population tibble to previous_population tibble
previous_population <- population
# Randomly copy from previous generation
population <- tibble(trait = sample(previous_population$trait, N, replace = TRUE))
# Identify the innovators
```

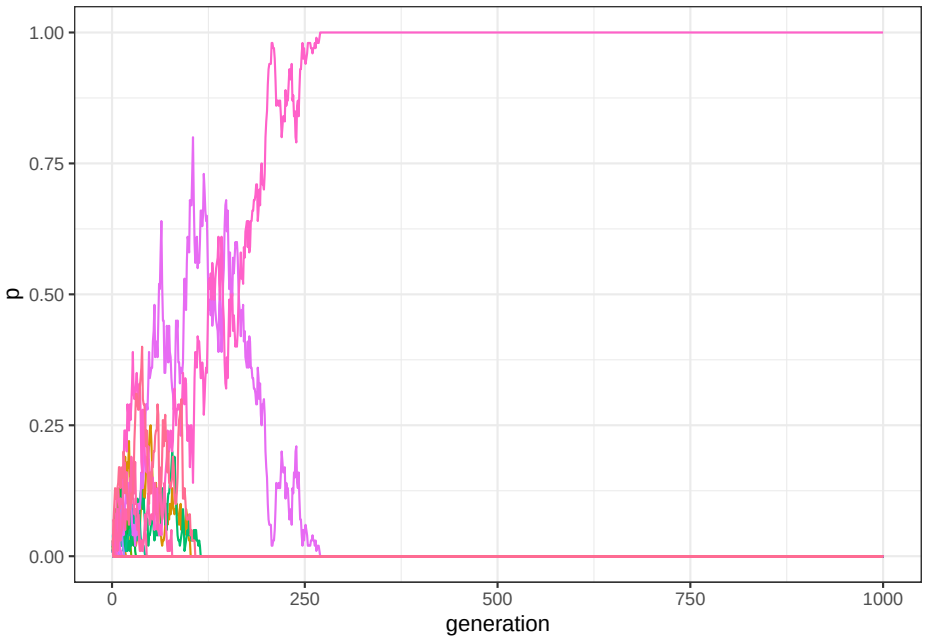


Figure 7.2: With small populations, a single traits reach fixation if there are enough generations in a model with multiple traits and unbiased transmission.

```
innovators <- sample(c(TRUE, FALSE), N, prob = c(mu, 1 - mu), replace = TRUE)
# If there are innovators
if( sum(innovators) > 0){
  # Replace innovators' traits with new traits
  population[innovators,]$trait <- (last_trait + 1):(last_trait + sum(innovators))
}
```

There are two modifications here. First, we need to select who are the innovators. For that, we use again the function `sample()`, biased by μ, picking `TRUE` (corresponding to being an innovator) or `FALSE` (keeping the copied cultural trait) N times.

Second, we need to actually introduce the new traits, with the correct number labels. First we record at the beginning of each generation the label of the last trait introduced (at the beginning, with $N = 100$, it will likely be 100 because we initialise each individual's traits by choosing randomly between 1 and 100). When new traits are introduced, we give them consecutive number labels: the first new trait will be called 101, the second 102, and so on. The code above adds all of the new traits into the innovator slots all in one go, which is more efficient than doing it one innovator at a time.

We can now, as usual, wrap everything in a function:

```
multiple_traits_2 <- function(N, t_max, mu) {
  max_traits <- N + N * mu * t_max

  output <- tibble(trait = as.factor(rep(1:max_traits, each = t_max)),
                   generation = rep(1:t_max, max_traits),
                   p = as.numeric(rep(NA, t_max * max_traits)))

  # Create first generation
  population <- tibble(trait = sample(1:N, N, replace = TRUE))
  # Add first generation's p for all traits
  output[output$generation == 1, ]$p <-
    tabulate(population$trait, nbins = max_traits) / N

  for (t in 2:t_max) {
    # Record what is the last trait introduced in the population
    last_trait <- max(population)

    # Copy individuals to previous_population tibble
    previous_population <- population

    # Randomly copy from previous generation
    population <- tibble(trait = sample(previous_population$trait, N, replace = TRUE))

    # Select the innovators
    innovators <- sample(c(TRUE, FALSE), N, prob = c(mu, 1 - mu), replace = TRUE)
    if ((last_trait + sum(innovators)) < max_traits) {
      if(sum(innovators) > 0){
        # Replace innovators' traits with new traits
        population[innovators,]$trait <-
          (last_trait + 1):(last_trait + sum(innovators))
      }
    }
    # Get p for all traits and put it into output slot for this generation t
    output[output$generation == t, ]$p <-
      tabulate(population$trait, nbins = max_traits) / N
  }
  # Export data
  output
}
```

You should now be familiar with more or less everything within this function, with one exception: the new quantity *max_traits*. This is a trick we are using to avoid making the code too slow to run. Our `output` tibble, as you remember, records all the frequencies of all traits. When programming, a good rule-of-thumb is to avoid dynamically modifying the size of your data structures, such as

adding new rows to a pre-existing tibble during the simulation. Where possible, set the size of a data structure at the start, and then modify its values during the simulation. So rather than creating a tibble that is expanded dynamically as new traits are introduced via innovation, we create a bigger tibble from the start. How big should it be? We do not know for sure, but a good estimate is that we will need space for the initial traits (N), plus around $N\mu$ traits that are added each generation.

To be absolutely sure we do not exceed this estimate, we wrap the innovation instruction within the `if ((last_trait + sum(innovators)) < max_traits)` condition. This prevents innovation when the tibble has filled up. This might prevent innovation in the last few generations, but this should have negligible consequences for our purposes.

Let's now run the function with an innovation rate $\mu = 0.01$, a population of 100 individuals, and for 200 generations.

```
data_model <- multiple_traits_2(N = 100, t_max = 200, mu = 0.01)
plot_multiple_traits(data_model)
```

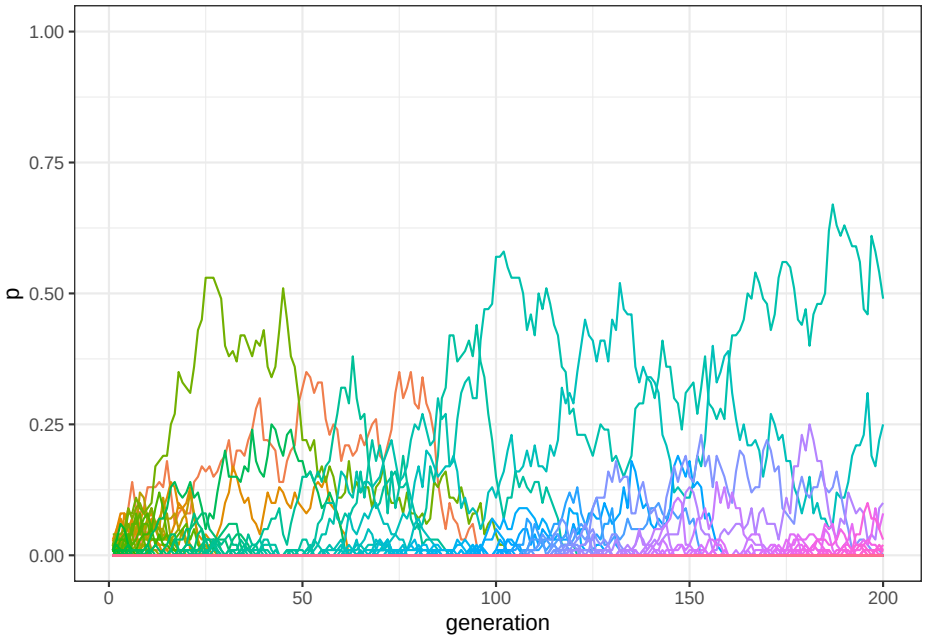


Figure 7.3: By adding innovations, more traits can be preserved in the population.

With innovation, there should now be more traits at non-zero frequency at the end of the simulation than when innovation was not possible. We can check the exact number, by inspecting how many frequencies are higher than 0 in the last

row of our matrix:

```
sum(filter(data_model, generation==200)$p > 0)
```

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

What happens if we increase the number of generations, or time steps, to 1000, as we did before?

```
data_model <- multiple_traits_2(N = 100, t_max = 1000, mu = 0.01)
plot_multiple_traits(data_model)
```

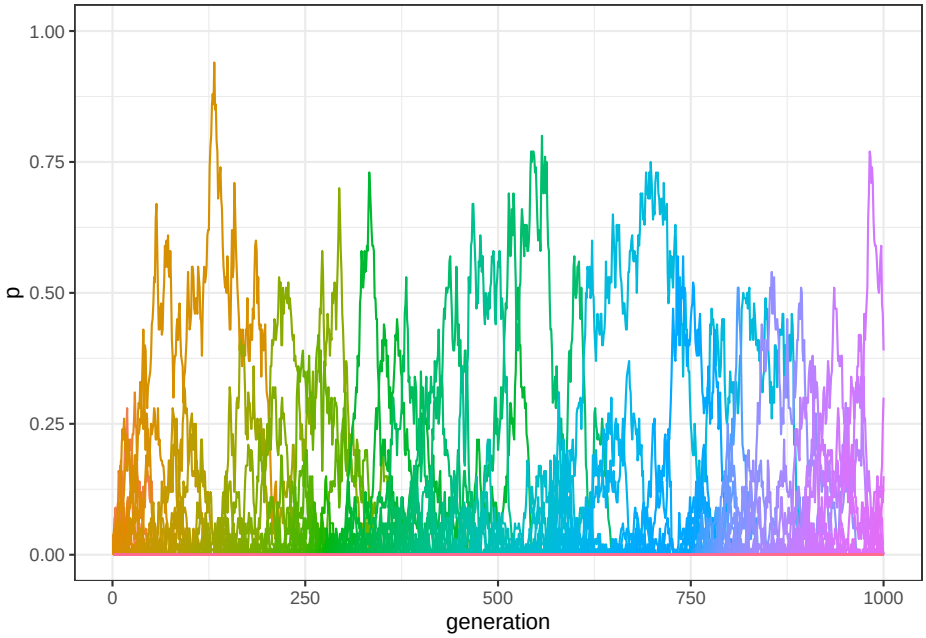


Figure 7.4: By adding innovations, traits are preserved even when the model runs for several generations.

As you can see in the plot, there should still be several traits that have frequencies higher than 0, even after 1000 generations. Again, we can find the exact number in the final generation:

```
sum(filter(data_model, generation==1000)$p > 0)
```

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

Innovation, in sum, allows the maintenance of variation even in small populations.

7.3 Optimising the code

Now for a short technical digression. You may have noticed that running the function `multiple_traits_2()` is quite time consuming with a population of 1000 individuals. There is a quick way to check the exact time needed, using the function `Sys.time()`. This returns the current time at the point of its execution. Let's run the function again and calculate how long it takes.

```
start_time <- Sys.time()
data_model <- multiple_traits_2(N = 100, t_max = 1000, mu = 0.01)
end_time <- Sys.time()
end_time - start_time
```

```
## Time difference of 21.44886 secs
```

While this varies from computer to computer, it may take several seconds to finsh. To store the output, we are using a tibble with 1100000 data points, as *max_traits* is equal to 1100, which needs to be updated in each of the 1000 generations. One way of speeding up the simulation is to record our output in a different data structure.

So far, we have been using tibbles to store our simulation output. R, as with all programming languages, can store data in different structures. Depending on what the data are and what one wants to do with them, different structures are more or less suitable. The advantage of tibbles is that they can contain heterogeneous data, depending on what we need to store: for example, in our `output` tibble, the `trait` column was specified as a factor, whereas the others two columns, `generation` and `p`, were numeric.

An alternative is to use vectors and matrices. A vector is a list of data points that are all of the same type, e.g. logical (TRUE/FALSE), integer (whole numbers), numeric (any numbers), or character (text). Matrices are just two-dimensional vectors: they must also contain all the same type of data, but they have rows and columns similar to a tibble, dataframe or Excel spreadsheet. The advantage of vectors and matrices is efficiency: they make simulations much faster than identical code running with tibbles.

Let's rewrite our multiple trait function that runs exactly the same simulation, but using matrices instead of tibbles. The output is now a matrix with `t_max` rows and `max_traits` columns. This is initialised with NAs at the beginning. The population is a vector of integers, representing the trait held by each individual.

```
multiple_traits_matrix <- function(N, t_max, mu) {

  max_traits <- N + N * mu * t_max

  output <- matrix(data = NA, nrow = t_max, ncol = max_traits)
```

```
  # Create first generation
  population <- sample(1:N, N, replace = TRUE)
  output[1, ] <- tabulate(population, nbins = N) / N

  # Add first generation's p for all traits
  for (t in 2:t_max) {
    # Record what is the last trait introduced in the population
    last_trait <- max(population)

    # Copy individuals to previous_population tibble
    previous_population <- population

    # Randomly copy from previous generation
    population <- sample(previous_population, N, replace = TRUE)

    # Select the innovators
    innovators <- sample(c(TRUE, FALSE), N, prob = c(mu, 1 - mu), replace = TRUE)
    if ((last_trait + sum(innovators)) < max_traits) {
      # Replace innovators' traits with new traits
      population[innovators] <- (last_trait + 1):(last_trait + sum(innovators))
    }
    # Get p for all traits and put it into output slot for this generation t
    output[t, ] <- tabulate(population, nbins = max_traits) / N
  }
  # Export data
  output
}
```

To plot the output, we re-convert it into a tibble so that it can be handled by `ggplot()`. We first create a column that explicitly indicates the number of generations, and then we use the function `pivot_longer()` from the tidyverse to reassemble the columns of the matrix in trait-frequency pairs, the 'tidy' format consistent with the functioning of ggplot.

```
plot_multiple_traits_matrix <- function(data_model) {
  generation <- rep(1:dim(data_model)[1], each = dim(data_model)[2])

  data_to_plot <- as_tibble(data_model) %>%
    pivot_longer(everything(), names_to = "trait", values_to = "p") %>%
    add_column(generation)

  ggplot(data = data_to_plot, aes(y = p, x = generation)) +
    geom_line(aes(colour = trait)) +
    ylim(c(0, 1)) +
    theme_bw() +
    theme(legend.position = "none")
```

```
}
```

We can now run the new function, checking that it gives the same output as the tibble version, and again calculating the time needed.

```
start_time <- Sys.time()
data_model <- multiple_traits_matrix(N = 100, t_max = 1000, mu = 0.01)
end_time <- Sys.time()
plot_multiple_traits_matrix(data_model)
```

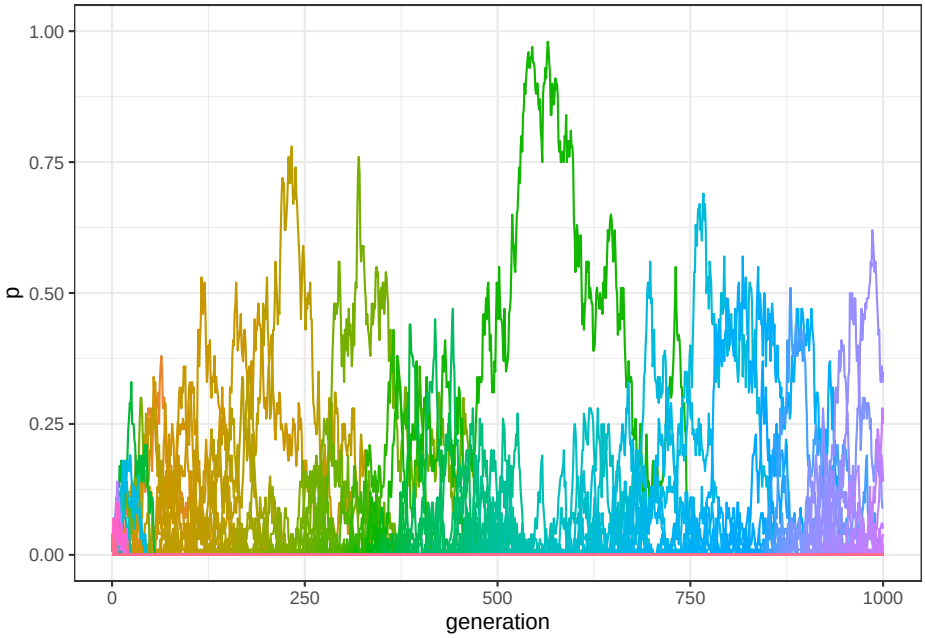


Figure 7.5: We obtain qualitatively the same results of the previous code, in a model with unbiased transmission, multiple traits, and innovation.

```
end_time - start_time
```

```
## Time difference of 0.183202 secs
```

The results are equivalent, but the simulation should be hundreds of times faster! This shows that implementation details are very important when building individual based models. When one needs to run the same simulation many times, or test many different parameter values, implementation choices can make drastic differences.

7.4 The distribution of popularity

An interesting aspect of these simulations is that, even if all traits are functionally equivalent and transmission is unbiased, a few traits, for random reasons, are more successful than the others. A way to visualise this is to plot their cumulative popularity, i.e. the sum of their quantities over all generations. Given our matrix, it is easy to calculate this by summing each column and multiplying by N (remember they are frequencies, whereas now we want to visualise their actual quantities). We also need to keep only the values that are higher than zero: values equal to zero are in fact the empty slots created in the initial matrix that were never filled with actual traits.

```
cumulative <- colSums(data_model) * N
cumulative <- cumulative[cumulative > 0]
```

Let's sort them from the most to the least popular and plot the results.

```
data_to_plot <- tibble(cumulative = sort(cumulative, decreasing = TRUE))

ggplot(data = data_to_plot, aes(x = seq_along(cumulative), y = cumulative)) +
  geom_point() +
  theme_bw() +
  labs(x = "trait label", y = "cumulative popularity")
```

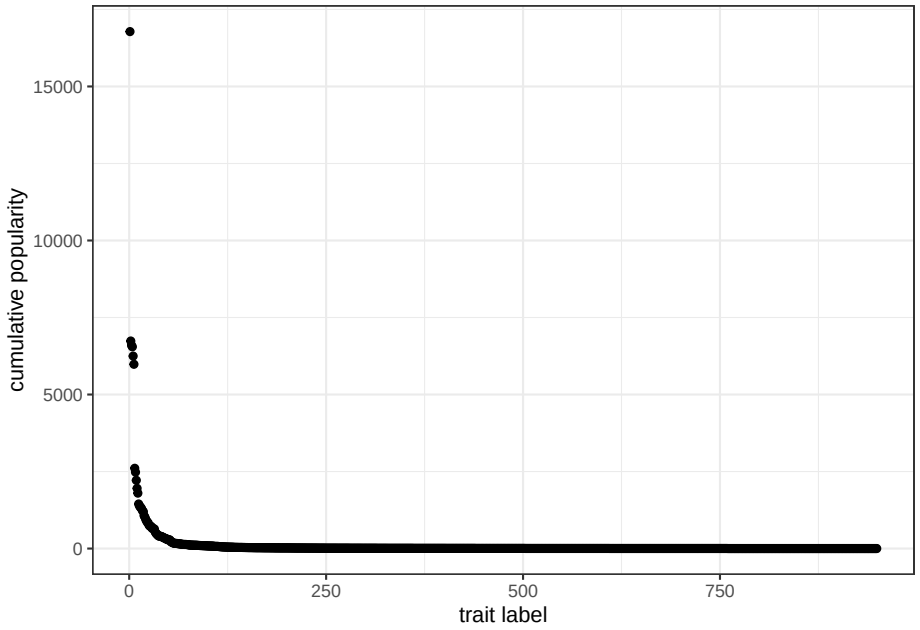


Figure 7.6: The popularity distribution of traits with unbiased copying is long-tailed, with few very successful traits and many relatively unsuccessful ones.

This is an example of a long-tailed distribution. The great majority of traits did not spread in the population, and their cumulative popularity is very close to one. Very few of them—the ones on the left side of the plot—were instead very successful. Long-tailed distributions like the one we just produced are very common for cultural traits: a small number of movies, books, or first names are very popular, while the great majority is not. In addition, in these domains, the popular traits are *much* more popular than the unpopular ones. The average cumulative popularity of the data we plotted is 106.43, but the most successful trait has a popularity of 16781.

It is common to plot these distributions by binning the data in intervals of exponentially increasing size. In other words, we want to know how many traits have a cumulative popularity between 1 and 2, then between 2 and 4, then between 4 and 8, and so on, until we reach the maximum value of cumulative popularity. The code below does that, using a `for` cycle to find how many traits fall in each bin and further normalising according to bin size. The size is increased 50 times, until an arbitrary maximum bin size of 2^{50}, to be sure to include all cumulative popularities.

```
bin <- rep(NA, 50)
x <- rep(NA, 50)

for(i in 1:50){
  bin[i] <- sum(cumulative >= 2^(i - 1) & cumulative < 2^i)
  bin[i] <- (bin[i] / length( cumulative)) / 2^(i - 1);
  x[i] <- 2^i
}
```

We can now visualise the data on a log-log plot, after filtering out the empty bins. A log-log plot is a graph that uses logarithmic scales on both axes. Using logarithmic axes is useful when, as in this case, the data are skewed towards large values. In the previous plot, we were not able to appreciate visually any difference in the great majority of data points, for example points that had cumulative popularity between 1 and 10, as they were all bunched up close to the x-axis.

```
data_to_plot <- tibble(bin = bin, x = x)
data_to_plot <- filter(data_to_plot, bin > 0)

ggplot(data = data_to_plot, aes(x = x, y = bin)) +
  geom_point() +
  labs(x = "cumulative popularity", y = "proportion of traits") +
  scale_x_log10() +
  scale_y_log10() +
  stat_smooth(method = "lm") +
  theme_bw()
```

On a log-log scale, the distribution of cumulative popularity produced by un-

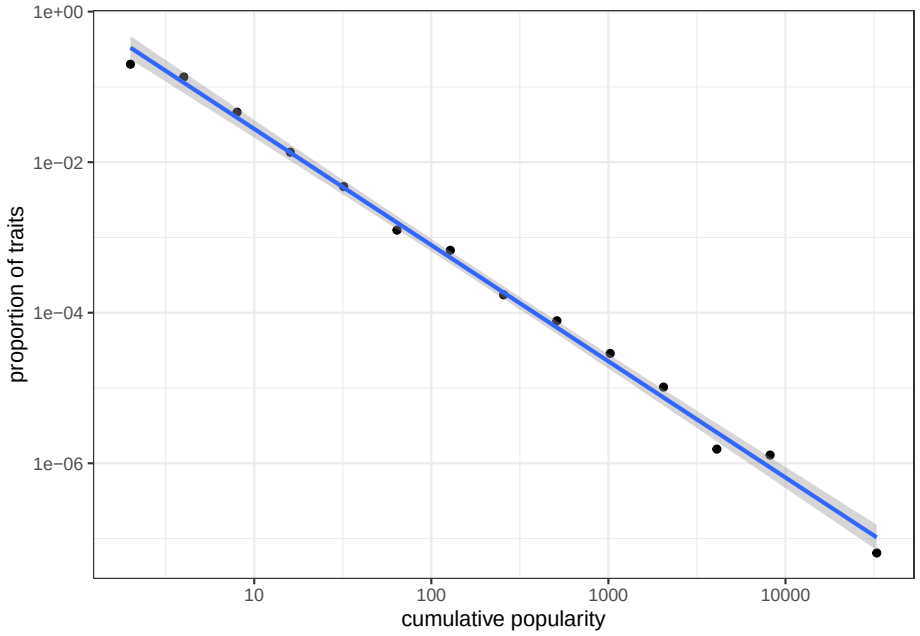


Figure 7.7: Popularity distribution of traits on a log-log scale.

biased copying lies approximately on a straight line (this linear best-fit line is produced using the command `stat_smooth(method = "lm")`). This straight line on a log-log plot is known as a "power law" frequency distribution. The goodness of fit and the slope of the line can be used to compare different types of cultural transmission. For example, what would happen to the above power law if we added some degree of conformity? What about demonstrator-based bias? We can also generate equivalent plots for real-world cultural datasets to test hypotheses about the processes that generated these distributions in the real world.

7.5 Summary of the model

In this chapter we simulated the case where individuals can possess one of more than two traits. We explored the simplest case of unbiased transmission. We also implemented the possibility of innovation, where individuals introduce, with some probability, new traits into the cultural pool of the population. Individual innovations counterbalance the homogenizing effect of drift, and replace the traits that are gradually lost.

To simulate multiple traits and innovation we also needed to deal with a few technical details such as how to keep track of an initially unknown number of new traits. We learned that it is best to create data structures of the desired

size at the outset, rather than changing their size dynamically during the simulation. We also saw the importance of using appropriate data structures when simulations start to become more complex. Replacing tibbles with matrices, we were able to make our simulation 100 times faster.

Our results showed that unbiased copying produces long-tailed distributions where very few traits are very popular and the great majority are not. An interesting insight from this model is that these extreme distributions do not necessarily result from extreme tendencies at the individual level. Some traits become hugely more popular than others without individuals being biased, for example, towards popular traits. Cultural transmission generates these distributions without biases, but simply because popular traits have the intrinsic advantage of being more likely to be randomly copied. We also introduced a new technique, the log-log plot of binned popularity distributions, to visualise this outcome.

7.6 Further readings

Neiman [1995] first introduced a model of unbiased copying with multiple traits to explain popularity distributions in assemblages of Neolithic pottery. Bentley et al. [2004] elaborated on this idea, presenting a 'random copying' model (equivalent to the one developed in this chapter) and comparing the popularity distributions produced with real datasets, including the frequency distributions of first names in the US and the citations of patents. Mesoudi and Lycett [2009] explored how adding transmission biases (e.g. conformity) to the basic model changes the resulting power-law frequency distribution.

Advanced topics - The evolution of cultural evolution

Chapter 8

Rogers' Paradox

The previous chapters all concerned cultural evolutionary dynamics: how different biases and transmission pathways affect the frequency of cultural traits in a population over time. Equally important, though, is to step back and consider where the possibility of culture came from in the first place. That is, we need to also consider the evolution of culture, and the evolution of cultural evolution.

The most basic question we can ask here is why a capacity for social learning (learning from others) evolved, relative to individual learning (learning directly from the environment, on one's own). An intuitive answer to this question is that social learning is less costly than individual learning. Imagine trying out different foods, some of which may be poisonous. One could try each one, and see if they make you ill. A less risky strategy would be to observe one's neighbour, and eat what they are eating. Unless they look sickly all the time, this will likely lead to a palatable (and evolutionarily adaptive) choice. Consequently, social learning should increase the mean adaptation of a population.

However, this intuition can be misleading. This was shown in 1988 by Alan Rogers in a now-classic model of the evolution of social learning (Rogers [1988]). This model is often called "Rogers' paradox", because it shows that under certain conditions, social learning does not lead to increased adaptation, even when it is less costly than individual learning. More precisely, the mean fitness of a population containing social learners does not exceed the mean fitness of a population composed entirely of individual learners. Here we will recapitulate Rogers' mathematical model in an individual-based simulation, to see when and why this counter-intuitive result holds.

8.1 Modelling Rogers' Paradox

In Rogers' model there are N individuals. Each individual has a fixed learning strategy: they are either an individual learner, or a social learner. Each indi-

vidual also exhibits a behaviour, which we will represent, as the traits in the previous chapter with an integer (e.g. "5", or "32"). ("Trait" and "behaviour" are often used interchangeably in cultural evolution literature.) There is also an environmental state, E, which is also represented with an integer. When an individual's behaviour matches the environment, they receive increased fitness, compared to when it does not match. A match might represent 'palatable food', while a mismatch might represent 'poisonous food'.

In each generation, individual learners directly sample the environment, and have a probability p of acquiring the 'correct', adaptive behaviour that matches the environment (and therefore a probability $1 - p$ of adopting the incorrect, maladaptive behaviour). Social learners choose a member of the previous generation at random and copy their behaviour, just like for unbiased transmission considered in Chapter 1.

Unlike previous models, we are interested here not in the behaviours or traits, but in how the learning strategies evolve over time. We therefore want to track the proportion of social learners in the population, which we call p_{SL} (with $1-p_{SL}$ being the proportion of individual learners). We assume these strategies are inherited (perhaps genetically, possibly culturally) from parent to offspring, and are affected by the fitness of the bearers of the strategies. Hence we need to specify fitness parameters.

Each individual starts with a baseline fitness, w. This is typically set at 1, to avoid tricky-to-handle negative fitnesses. Individuals who have behaviour that matches the environment receive a fitness boost of $+b$. Individuals who have behaviour that does not match the environment receive a fitness penalty of $-b$. Explicit in the above verbal outline is that social learning is less costly than individual learning. Therefore, individual learners receive a fitness cost of $-b * c$, and social learners receive a fitness cost of $-b * s$, where $c > s$. For simplicity, we can set $s = 0$ (social learning is free) and set $c > 0$, so we only have to change one parameter.

The fitness of each individual is then totted up based on the above, and the next generation is created. Each individual reproduces in proportion to the fitness of their strategy, relative to other strategies.

We also assume some mutation during reproduction. With probability μ, the new individual 'mutates' to the other learning strategy. Because we are interested here in how social learning evolves from individual learning, we start with a first generation entirely made up of individual learners. Social learning then appears from the second generation onwards via mutation.

Finally, Rogers was interested in the effect of environmental change. Each generation, there is a probability u of the environment changing to a new state. In Rogers' original model, the environment flipped between the same two states, back and forth. However, this is problematic when environmental change is very fast, because an individual with out-dated behaviour can receive a fitness benefit if the environment flips back to the previous state. Hence we assume

that when environments change, they change to a new value never previously experienced by any individual.

This is a complex model but let's go step by step. First we create and initialise tibbles to store the output and the population of individuals, just like in previous chapters. The output here needs to be big enough to store data from r_{max} runs and t_{max} generations, like before. We then need to create NA placeholders for p_{SL} (the proportion of social learners) and W (the mean population fitness). The `population` tibble stores the characteristics of the individuals: learning strategy ('individual' or 'social'), behaviour (initially all NA) and fitness (initially all NA). Finally, we initialise the environment E at zero, which will subsequently increment, meaning that the environment changes.

```
library(tidyverse)

N <- 100
r_max <- 1
t_max <- 10

# Create the output tibble
output <- tibble(generation = rep(1:t_max, r_max),
                 run = as.factor(rep(1:r_max, each = t_max)),
                 p.SL = as.numeric(rep(NA, t_max * r_max)),
                 W = as.numeric(rep(NA, t_max * r_max)))

# Create the population tibble
population <- tibble(learning = rep("individual", N),
                     behaviour = rep(NA, N),
                     fitness = rep(NA, N))

# Initialise the environmental state to 0
E <- 0
```

Now let's go through each event that happens during a single generation. Later we will put it all inside a loop. It's useful to write out the events that we need:

1. Social learning
2. Individual learning
3. Calculate fitnesses
4. Store population characteristics in output tibble
5. Reproduction
6. Potential environmental change

First, social learning. The following code picks random individuals from the `previous_population` tibble (which we have yet to create, but will do later), to put into the social learner individuals in the current `population` tibble. This is similar to what we did in the previous chapters. It only does this if there is at least one social learner. As noted above, we start in the first generation with

all individual learners and no social learners, so this will not be fulfilled until the second generation. For now, nothing happens.

```
if (sum(population$learning == "social") > 0) {
  population$behaviour[population$learning == "social"] <-
    sample(previous_population$behaviour, sum(population$learning == "social"), replac
}
```

The following code implements individual learning. This *does* apply to the first generation. We first create a vector of `TRUE` and `FALSE` values dependent on p, the probability of individual learning resulting in a correct match with the environment. With this probability, individual learners have their behaviour set to the correct E value. Otherwise, they are given the incorrect behaviour $E-1$. Note the use of the `!` before `learn_correct` to give a match when this vector is `FALSE` (i.e. they do *not* learn the correct behaviour).

```
learn_correct <- sample(c(TRUE, FALSE), N, prob = c(p, 1 - p), replace = TRUE)
population$behaviour[learn_correct & population$learning == "individual"] <- E
population$behaviour[!learn_correct & population$learning == "individual"] <- E - 1
```

Now we obtain the fitnesses for each individual. First we give everyone the baseline fitness, w. Then we add or subtract b, based on whether the individual has the correct or incorrect behaviour. Finally we impose costs, which are different for social and individual learners.

```
# Baseline fitness
population$fitness <- w

# For individuals with behaviour matched to the environment, add b
population$fitness[population$behaviour == E] <-
  population$fitness[population$behaviour == E] + b
# For individuals with behaviour not matched to the environment, subtract b
population$fitness[population$behaviour != E] <-
  population$fitness[population$behaviour != E] - b

# Impose cost b*c on individual learners:
population$fitness[population$learning == "individual"] <-
  population$fitness[population$learning == "individual"] - b*c
# Impose cost b*s (i.e. 0) on social learners:
population$fitness[population$learning == "social"] <-
  population$fitness[population$learning == "social"] - b*s
```

The fourth stage is recording the resulting data into the `output` tibble. First we calculate p_{SL} as the number of social learners divided by the total population size. Then we calculate W, the mean fitness in the entire population. All of these are done with the standard R mean command.

```
output[output$generation == t & output$run == r, ]$p_SL <- mean(population$learning == "social")
output[output$generation == t & output$run == r, ]$W <- mean(population$fitness)
```

The fifth stage is reproduction. Here we put the current `population` tibble into a new tibble, called `previous_population`, as we have done before. This acts as both a record to now calculate fitnesses, as well as a source of demonstrators for the social learning stage we covered above. After doing this, we reset the behaviour and fitness of the current population. We then over-write the learning strategies based on fitness.

First we get `fitness_IL`, the fitness of individual learners relative to the fitness of the entire population (assuming there are any individual learners, otherwise we set this to zero). This then serves as the probability of setting new individuals as individual learners in the next generation. We use gain the function `sample()` to create `mutation`, denoting the probability of an individual mutating their learning strategy. Finally, we change the learning strategy of the 'mutant' individuals. Notice we need to create a temporary new tibble, `previous_poulation2`, to avoid to mutate twice the individuals that are changed from `individual` to `social` learning in the first mutation instruction.

```
previous_population <- population
population$behaviour <- NA
population$fitness <- NA

# Relative fitness of individual learners (if there are any)
if (sum(previous_population$learning == "individual") > 0) {
  fitness_IL <- sum(previous_population$fitness[previous_population$learning == "individual"]) /
    sum(previous_population$fitness)
} else {
  fitness_IL <- 0
}

# Create the new population
population$learning <- sample(c("individual", "social"), size = N,
             prob = c(fitness_IL, 1 - fitness_IL), replace = TRUE)

# Also add mutation, chance of switching learning types
mutation <- sample(c(TRUE, FALSE), N, prob = c(mu, 1 - mu), replace = TRUE)

# Store current population in a tibble to avoid mutating twice
previous_population2 <- population
# If an individual is an individual learner plus mutation, then they're a social learner
population$learning[previous_population2$learning == "individual" & mutation] <- "social"
# If an individual is a social learner plus mutation, then they're an individual learner
population$learning[previous_population2$learning == "social" & mutation] <- "individual"
```

The final stage is the easiest. With probability u, we increment the environmen-

tal state E by one. Otherwise, it stays as it is. To do this we pick a random number between 0 and 1 using the *runif* command, and if u exceeds this, we increment E.

```
if (runif(1) < u) E <- E + 1
```

That covers the six stages that occur in each generation. We can now put them all together into a loop tracking runs, and a loop tracking generations. We can also put all this inside a function. This should all be familiar from previous chapters. Almost all the code is taken from above, and we numbered the different stages (you can find the comments to the specific lines of codes in the chunks above). We also add a parameter check at the start, to make sure that we don't get negative fitnesses. This uses the new function `stop()`, that tells R to terminate the execution of the function, and print on screen the message in the parenthesis. Another novelty is that we already set some of the parameters (w, b and s) in the function call. In this way, the parameters are set to these default values if not specified when the function is called. The other parameters need to be instead specified.

```
rogers_model <- function(N, t_max, r_max, w = 1, b = 0.5, c, s = 0, mu, p, u) {

  # Check parameters to avoid negative fitnesses
  if (b * (1 + c) > 1 || b * (1 + s) > 1) {
    stop("Invalid parameter values: ensure b*(1+c) < 1 and b*(1+s) < 1")
  }

  # Create output tibble
  output <- tibble(generation = rep(1:t_max, r_max),
                   run = as.factor(rep(1:r_max, each = t_max)),
                   p_SL = as.numeric(rep(NA, t_max * r_max)),
                   W = as.numeric(rep(NA, t_max * r_max)))

  for (r in 1:r_max) {

    # Create a population of individuals
    population <- tibble(learning = rep("individual", N),
                         behaviour = rep(NA, N), fitness = rep(NA, N))

    # Initialise the environment
    E <- 0

    for (t in 1:t_max) {

      # 1. Social learning
      if (sum(population$learning == "social") > 0) {
        population$behaviour[population$learning == "social"] <-
          sample(previous_population$behaviour, sum(population$learning == "social"),
```

```
    }

    # 2. individual learning
    learn_correct <- sample(c(TRUE, FALSE), N, prob = c(p, 1 - p), replace = TRUE)
    population$behaviour[learn_correct & population$learning == "individual"] <- E
    population$behaviour[!learn_correct & population$learning == "individual"] <- E - 1

    # 3. Calculate fitnesses
    population$fitness <- w

    population$fitness[population$behaviour == E] <-
      population$fitness[population$behaviour == E] + b

    population$fitness[population$behaviour != E] <-
      population$fitness[population$behaviour != E] - b

    population$fitness[population$learning == "individual"] <-
      population$fitness[population$learning == "individual"] - b*c

    population$fitness[population$learning == "social"] <-
      population$fitness[population$learning == "social"] - b*s

    # 4. Store population characteristics in output
    output[output$generation == t & output$run == r, ]$p_SL <-
      mean(population$learning == "social")
    output[output$generation == t & output$run == r, ]$W <-
      mean(population$fitness)

    # 5. Reproduction
    previous_population <- population
    population$behaviour <- NA
    population$fitness <- NA

    if (sum(previous_population$learning == "individual") > 0) {
      fitness_IL <- sum(previous_population$fitness[previous_population$learning == "individual
        sum(previous_population$fitness)
    } else {
      fitness_IL <- 0
    }

    population$learning <- sample(c("individual", "social"), size = N,
           prob = c(fitness_IL, 1 - fitness_IL), replace = TRUE)

    mutation <- sample(c(TRUE, FALSE), N, prob = c(mu, 1 - mu), replace = TRUE)
```

```
      previous_population2 <- population
      population$learning[previous_population2$learning == "individual" & mutation] <-
      population$learning[previous_population2$learning == "social" & mutation] <- "ind

      # 6. Potential environmental change
      if (runif(1) < u) E <- E + 1

    }
  }
  # Export data from function
  output
}
```

Now we can run the simulation for 10 runs, and 200 generations, with a population of 1000 individuals. The other parameters we set are the cost associated to individual learning (`c = 0.9`); the mutation rate, i.e. the probability that an individual that inherit learning strategy (individual or social) will switch to the other (`mu = 0.001`); the accuracy of individual learning (`p = 1`); and, finally, the probability of environmental change (`u = 0.2`). We will later explore other values of these parameters, but feel free to change them and see what happens!

```
data_model <- rogers_model(N = 1000, t_max = 200, r_max = 10, c = 0.9, mu = 0.01, p =
```

You can inspect the `data_model` tibble, but so much data is hard to make sense of. Let's write a plotting function like in previous chapters. The only difference from our usual `plot_multiple_runs()` is that instead of plotting the frequency of traits, we want to visualise p_{SL}, the frequency of social learners, so we

```
plot_multiple_runs_p_SL <- function(data_model) {
  ggplot(data = data_model, aes(y = p_SL, x = generation)) +
    geom_line(aes(colour = run)) +
    stat_summary(fun = mean, geom = "line", size = 1) +
    ylim(c(0, 1)) +
    theme_bw() +
    labs(y = "proportion of social learners")
}

plot_multiple_runs_p_SL(data_model)
```

Here we can see that, for these parameter values, the mean proportion of social learners quickly goes to 0.5, but then remains fluctuating around this value. However, each run is quite erratic, with a large spread. More important for our understanding of Rogers' paradox, however, is the mean fitness of the population, and how this compares with a population entirely composed of individual learners. Consequently, we need to plot the mean population fitness over time. This is `W` in the output of the `rogers_model()` function. The function below plots this, along with a dotted line denoting the fitness of an individual learner,

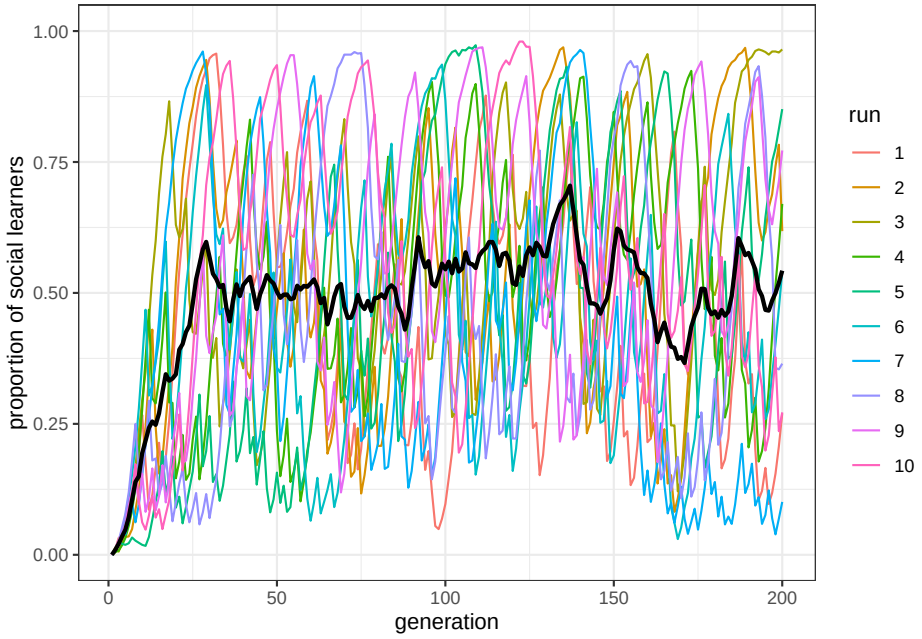


Figure 8.1: On average the proportion of social learners fluctuates around 0.5 (black line). However, individual runs have a larger spread around this mean (overlapping colored runs).

which by extension will be the same as the mean fitness of a population entirely composed of individual learners. We do not need to extract this from the output of the simulation: the fitness of individual learners is fixed, known a-priori, and can be calculated knowing the values of some of the parameters of the simulation. There are a few new elements in the plotting functions. First, we want to pass to the function, together with the `data_model` tibble, some other information on the parameters of out simulations, so that that fitness line for individual learners can be draw. As in the main `rogers_model()` function, `w` and `b` are hard coded, and we need to specify `c` and `p`. Second, we use the function `geom_hline()`. This is another ggplot 'geom' that plots, as the name suggest, an horizontal line that intercept the y axis where indicated by `yintervept`, in our case the fitness of individual learners. Finally, we set the upper limit of the y axis to NA, which ggplot interprets as the limit from the range of the data.

```
plot_W <- function(data_model, w = 1, b = 0.5, c, p) {
  ggplot(data = data_model, aes(y = W, x = generation)) +
    geom_line(aes(color = run)) +
    stat_summary(fun = mean, geom = "line", size = 1) +
    geom_hline(yintercept = w + b * (2 * p - c - 1), linetype = 2) +
    ylim(c(0, NA)) +
    theme_bw() +
    labs(y = "mean population fitness")
}

plot_W(data_model, c = 0.9, p = 1)
```

This is Rogers' paradox. Even though social learning is less costly than individual learning (i.e. $s < c$), our population of roughly 50% social learners do not consistently exceed the dotted line that indicates the fitness of a population of individual learners. Social learning does not increase adaptation. This also runs counter to the common claim that culture - with social learning at its heart - has been a key driver of our species' ecological success.

The reason for this result is that social learning is frequency-dependent in a changing environment. Individual learners undergo costly individual learning and discover the correct behaviour, initially doing well. Social learners then copy that behaviour, but at lower cost. Social learners therefore then do better than, and outcompete, individual learners. But when the environment changes, the social learners do badly, because they are left copying outdated behaviour. Individual learners then do better, because they can detect the new environmental state. Individual learners increase in frequency, and the cycle continues. This is what the large oscillations of the single runs show. Analytically, it can be shown that they reach an equilibrium at which the frequency of social and individual learners is the same but, by definition, this equilibrium must have the same mean fitness as a population entirely composed of individual learners. Hence, the 'paradox'.

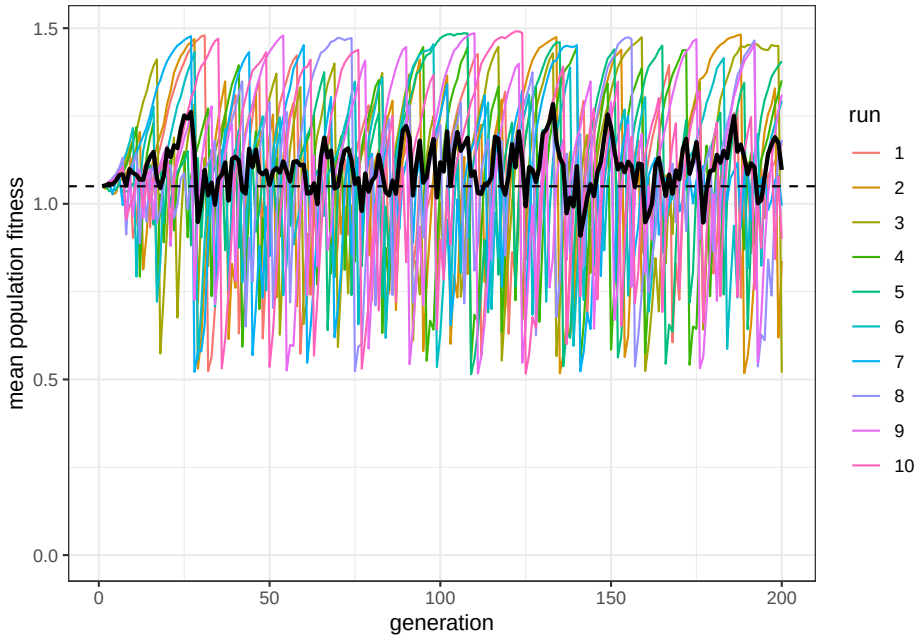


Figure 8.2: Populations with roughly 50% social learners have on average the same fitness (black line) as populations with only individual learners (dashed line). Even though populations with social learners sometimes exceed the average fitness of all-individual learner populations they also sometimes fall far below it.

To explore this further, we can alter the parameters. First, we can reduce the cost of individual learning, from $c = 0.9$ to $c = 0.4$.

```
data_model <- rogers_model(N = 1000, t_max = 200, r_max = 10, c = 0.4, mu = 0.01, p = 1
plot_multiple_runs_p_SL(data_model)
```

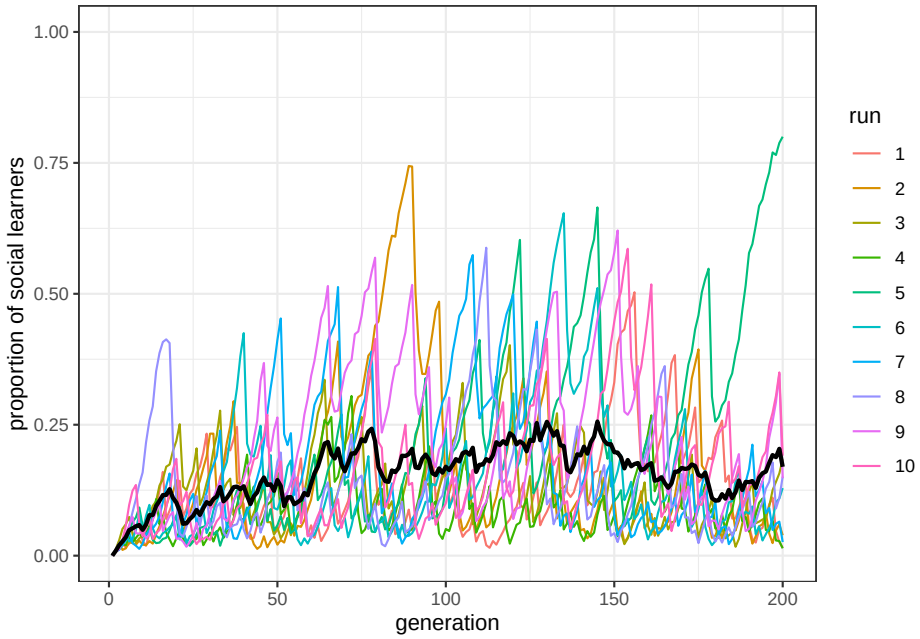


Figure 8.3: There are fewer social learners in a population where the cost of individaul learning is lower.

```
plot_W(data_model, c = 0.4, p = 1)
```

As we might expect, this reduces the proportion of social learners, by giving individual learners less of a penalty for doing their individual learning. Also as expected, the paradox remains. In fact it is even more obvious, given that there are many more individual learners.

We can also reduce the accuracy of individual learning, reducing p from 1 to 0.7.

```
data_model <- rogers_model(N = 1000, t_max = 200, r_max = 10, c = 0.9, mu = 0.01, p = 0
plot_multiple_runs_p_SL(data_model)
```

```
plot_W(data_model, c = 0.9, p = 0.7)
```

Now there are a majority of social learners. Yet the paradox remains: the mostly social learners still do not really exceed the pure individual learning fitness line.

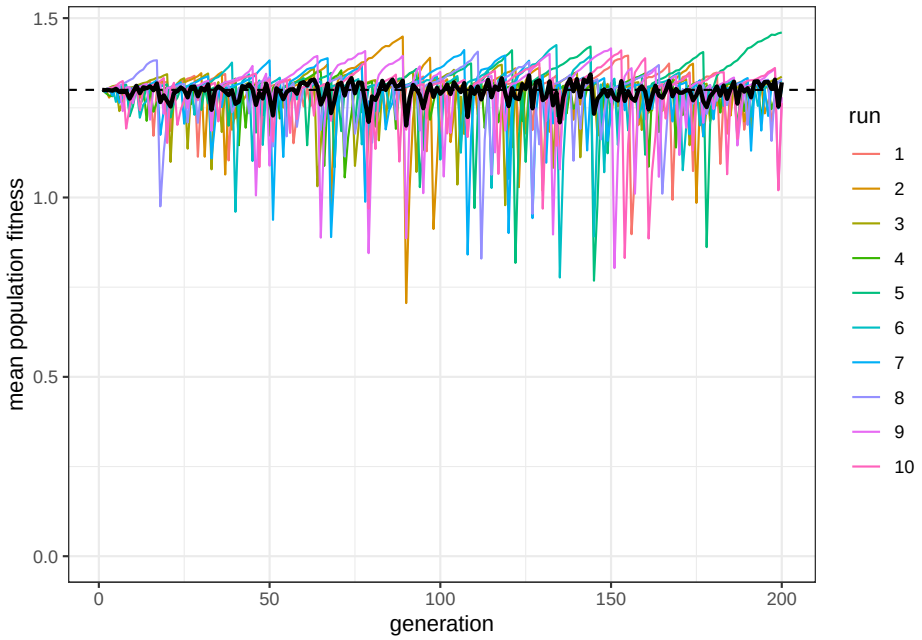


Figure 8.4: The fitness of the mixed population remains equal to the fitness of individual learners in a population where the cost of individaul learning is lower.

If our explanation above is correct, then making the environment constant should remove the paradox. If the environment stays the same, then behaviour can never be outdated, and individual learners never regain the upper hand. Setting $u = 0$ shows this.

```
data_model <- rogers_model(N = 1000, t_max = 200, r_max = 10, c = 0.9, mu = 0.01, p = 1, u = 0)
plot_multiple_runs_p_SL(data_model)
```

```
plot_W(data_model, c = 0.9, p = 1)
```

Now the paradox has disappeared: social learners clearly outperform the individual learners after the latter have gone to the trouble of discovering the correct behaviour, and the social learners have higher mean fitness than the individual learning dotted line. (Notice also the oscillations within each run disappeared.) This is just as we would expect. Rogers' paradox crucially depends on a changing environment. However, nature rarely provides a constant environment. Food sources change location, technology accumulates, languages diverge, and climates change.

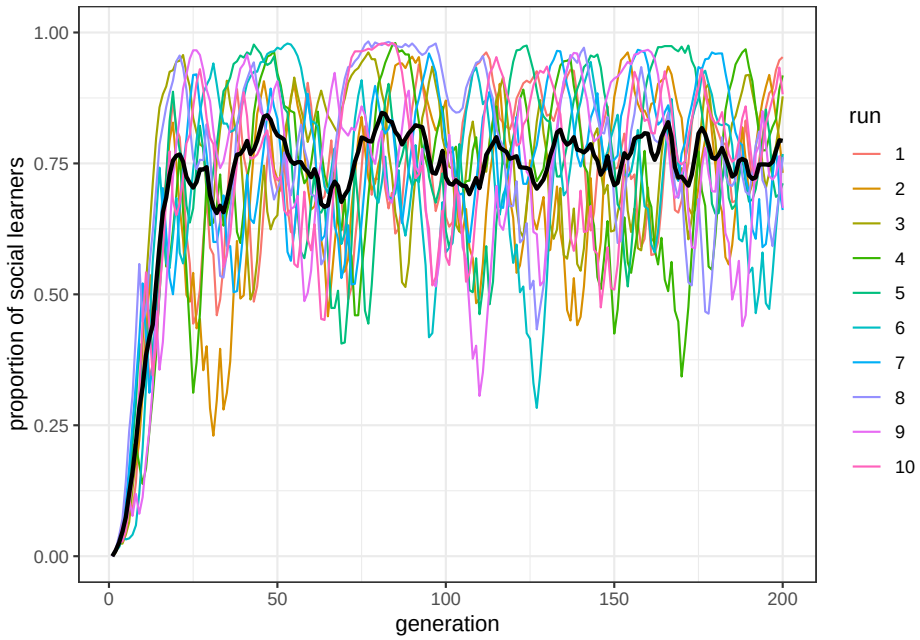


Figure 8.5: When individual learning is accurate, there are more social learners in populations.

8.2 Summary of the model

Rogers' model is obviously a gross simplification of reality. However, as discussed in earlier chapters, realism is often not the aim of modelling. Models - even simple and grossly unrealistic ones - force us to think through assumptions, and challenge verbal theorising. Rogers' model is a good example of this. Even though it sounds reasonable that social learning should increase the mean fitness, or adaptation, of a population, in this simple model with these assumptions it does not. We saw one situation in which social learning *does* increase mean fitness: when environments do not change. This, however, is not very plausible. Environments always change. We therefore need to examine the other assumptions of Rogers' model. We will do this in the next chapter.

8.3 Further reading

An early example of the claim that social learning is adaptive because it reduces the costs of learning can be found in Boyd and Richerson [1985]. Rogers [1988] then challenged this claim, as we have seen in this chapter. In the next chapter we will consider subsequent models that have examined 'solutions' to Rogers' paradox.

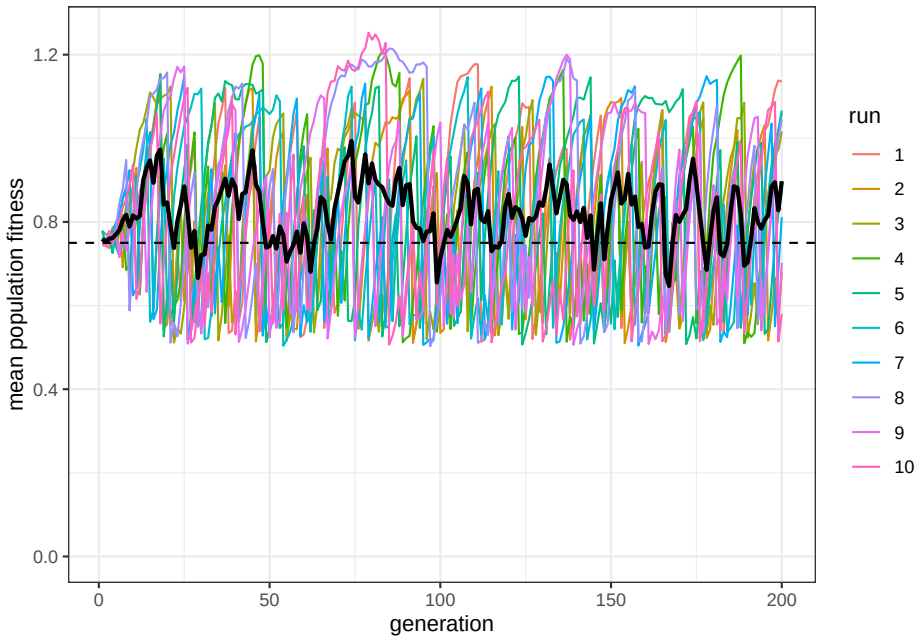


Figure 8.6: Even when individual learning is more accurate, the average fitness of mixed populations is close to the fitness of pure individual learners.

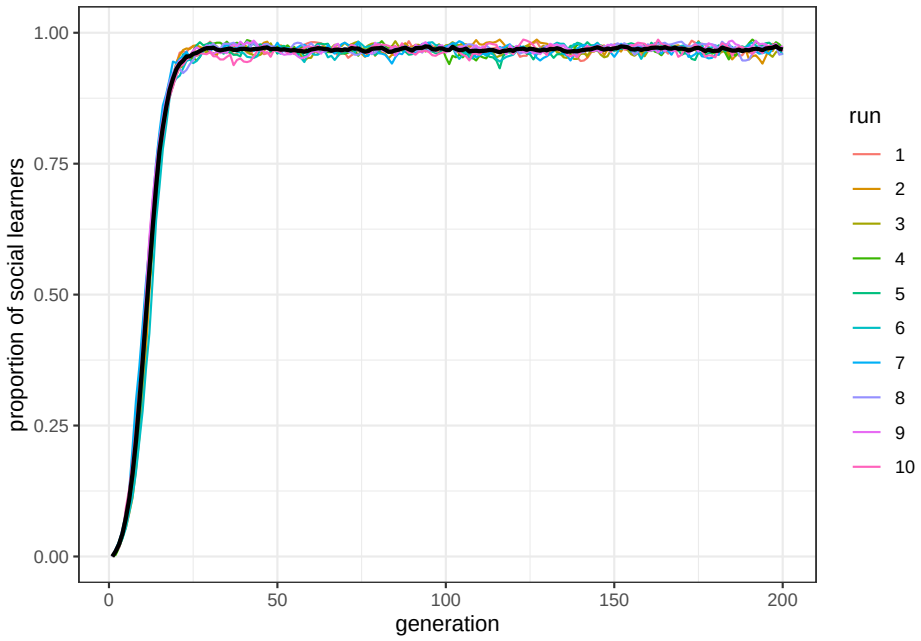


Figure 8.7: When the environment is unchanging, social learners will outperform individual learners and take over in populations.

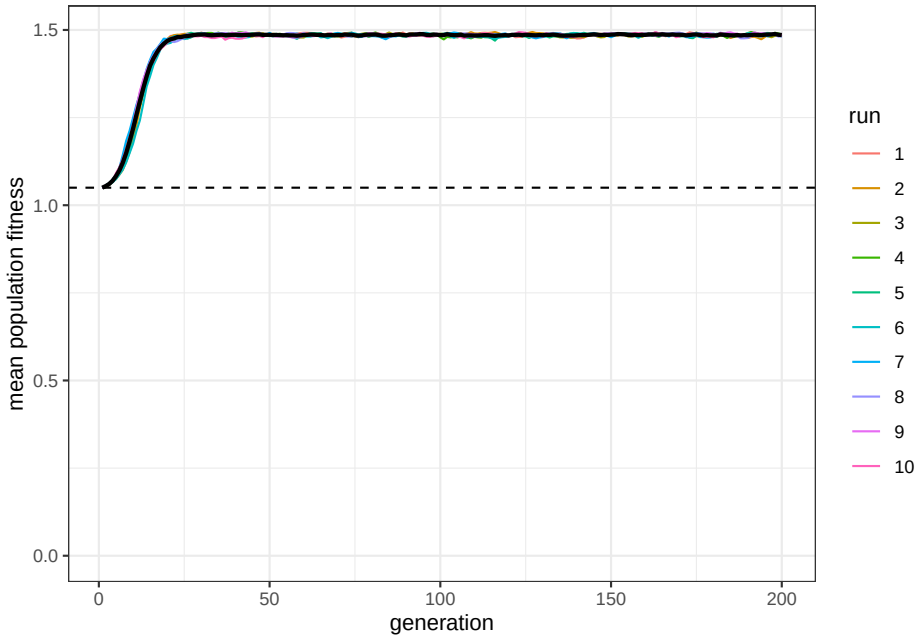


Figure 8.8: Their average fitness (black line) is now much higher than that of indiviudal learners (dashed line).

Chapter 9

Rogers' Paradox: A Solution

In the previous chapter we saw how social learning does not increase the mean fitness of a population relative to a population entirely made up of individual learners, at least in a changing environment. This is colloquially known as Rogers' paradox, after Alan Rogers' model which originally showed this. It is a 'paradox' because it holds even though social learning is less costly than individual learning, and social learning is often argued to underpin our species' ecological success. The paradox occurs because social learning is frequency dependent: when environments change, the success of social learning depends on there being individual learners around to copy. Otherwise social learners are left copying each others' outdated behaviour.

Several subsequent models have explored 'solutions' to Rogers' paradox. These involve relaxing the obviously unrealistic assumptions. One of these is that individuals in the model come in one of two fixed types: social learners (who always learn socially), and individual learners (who always learn individually). This is obviously unrealistic. Most organisms that can learn individually can also learn socially, and the two capacities likely rely on the same underlying mechanisms (e.g. associative learning, see e.g. Heyes [2012]).

9.1 Modelling critical social learners

To explore how a mixed learning strategy would compete with pure strategies (only social or only individual learning), Enquist et al. [2007] added another type of individual to Rogers' model: a critical social learner. These individuals first try social learning, and if the result is unsatisfactory, they then try individual learning.

The following function modifies the `rogers_model()` function from the last chapter to include critical learners. We need to change the code in a few places, but the modifications should be all easy to understand at this point. To start with, in the output tibble, we need to take track also of the number of individual learners (before they were simply the individuals that were *not* social learners) and of the number of the individuals adopting the new strategy, critical social learning. We have now two more variables for this: `P_IL` and `P_CL`. Next, we need to add a learning routine for critical learners. This involves repeating the social learning code originally written for the social learners. We then apply the individual learning code to those critical learners who copied the incorrect behaviour, i.e. if their behaviour is different from `E` (this makes them 'unsatisfied'). To make it easier to follow, we now insert the fitness updates into the learning section. This is because only those critical learners who are unsatisfied will suffer the costs of individual learning. If we left it to afterwards, it's easy to lose track of who is paying what fitness costs.

Reproduction and mutation are changed to account for the three learning strategies. We now need to get the relative fitness of social and individual learners, and reproduce based on those fitnesses. Individuals left over become critical learners. We could calculate the relative fitness of critical learners, but it's not really necessary given that the proportion of critical learners will always be 1 minus the proportion of social and individual learners. Similarly, mutation now needs to specify that individuals can mutate into either of the two other learning strategies. We assume this mutation is unbiased, and mutation is equally likely to result in the two other strategies. Notice the use of the function `sample()` when we set the learning strategies of the new population. So far we always used for binary choices, now we are using it with three elements (`c("individual", "social", "critical")` and three probabilities (`prob = c(fitness_IL, fitness_SL, 1 - (fitness_SL + fitness_IL))`).

```
library(tidyverse)

rogers_model2 <- function(N, t_max, r_max, w = 1, b = 0.5, c, s = 0, mu, p, u) {

  # Check parameters to avoid negative fitnesses
  if (b * (1 + c) > 1 || b * (1 + s) > 1) {
    stop("Invalid parameter values: ensure b*(1+c) < 1 and b*(1+s) < 1")
  }

  # Create output tibble
  output <- tibble(generation = rep(1:t_max, r_max),
                   run = as.factor(rep(1:r_max, each = t_max)),
                   p_SL = as.numeric(rep(NA, t_max * r_max)),
                   p_IL = as.numeric(rep(NA, t_max * r_max)),
                   p_CL = as.numeric(rep(NA, t_max * r_max)),
                   W = as.numeric(rep(NA, t_max * r_max)))
```

```r
for (r in 1:r_max) {

  # Create a population of individuals
  population <- tibble(learning = rep("individual", N),
                       behaviour = rep(NA, N), fitness = rep(NA, N))

  # Initialise the environment
  E <- 0

  for (t in 1:t_max) {

    # Now we integrate fitnesses into the learning stage
    population$fitness <- w

    # 1. Social learning
    if (sum(population$learning == "social") > 0) {
      # Subtract cost b*s from fitness of social learners
      population$fitness[population$learning == "social"] <-
        population$fitness[population$learning == "social"] - b*s
      # Update behaviour
      population$behaviour[population$learning == "social"] <-
        sample(previous_population$behaviour, sum(population$learning == "social"), replace = T

    }

    # 2. Individual learning
    # Subtract cost b*c from fitness of individual learners
    population$fitness[population$learning == "individual"] <-
      population$fitness[population$learning == "individual"] - b*c
    # Update behaviour
    learn_correct <- sample(c(TRUE, FALSE), N, prob = c(p, 1 - p), replace = TRUE)
    population$behaviour[learn_correct & population$learning == "individual"] <- E
    population$behaviour[!learn_correct & population$learning == "individual"] <- E - 1


    # 3. Critical social learning
    if (sum(population$learning == "critical") > 0) {

      # Subtract b*s from fitness of socially learning critical learners
      population$fitness[population$learning == "critical"] <-
        population$fitness[population$learning == "critical"] - b*s

      # First critical learners socially learn
      population$behaviour[population$learning == "critical"] <-
        sample(previous_population$behaviour,
```

```
              sum(population$learning == "critical"), replace = TRUE)

    # Subtract b*c from fitness of individually learning critical learners
    population$fitness[population$learning == "critical" & population$behaviour !=
      population$fitness[population$learning == "critical" & population$behaviour

    # Individual learning for those critical learners who did not copy correct beh
    population$behaviour[learn_correct & population$learning == "critical" & popula
    population$behaviour[!learn_correct & population$learning == "critical" & popul
  }

  # 4. Calculate fitnesses (now only need to do the b bonus or penalty)
  population$fitness[population$behaviour == E] <-
    population$fitness[population$behaviour == E] + b
  population$fitness[population$behaviour != E] <-
    population$fitness[population$behaviour != E] - b

  # 5. store population characteristics in output
  output[output$generation == t & output$run == r, ]$p_SL <-
    mean(population$learning == "social")
  output[output$generation == t & output$run == r, ]$p_IL <-
    mean(population$learning == "individual")
  output[output$generation == t & output$run == r, ]$p_CL <-
    mean(population$learning == "critical")
  output[output$generation == t & output$run == r, ]$W <-
    mean(population$fitness)

  # 6. Reproduction
  previous_population <- population
  population$behaviour <- NA
  population$fitness <- NA

  # Individual learners
  if (sum(previous_population$learning == "individual") > 0) {
    fitness_IL <- sum(previous_population$fitness[previous_population$learning ==
      sum(previous_population$fitness)
  } else {
    fitness_IL <- 0
  }

  # Social learners
  if (sum(previous_population$learning == "social") > 0) {
    fitness_SL <- sum(previous_population$fitness[previous_population$learning ==
      sum(previous_population$fitness)
  } else {
```

```
        fitness_SL <- 0
      }

      population$learning <- sample(c("individual", "social", "critical"), size = N,
             prob = c(fitness_IL, fitness_SL, 1 - (fitness_SL + fitness_IL)), replace = TRUE)

      mutation <- sample(c(TRUE, FALSE), N, prob = c(mu, 1 - mu), replace = TRUE)

      previous_population2 <- population

      population$learning[mutation & previous_population2$learning == "individual"] <-
        sample(c("critical", "social"),
               sum(mutation & previous_population2$learning == "individual"),
               prob = c(0.5, 0.5), replace = TRUE)

      population$learning[mutation & previous_population2$learning == "social"] <-
        sample(c("critical", "individual"),
               sum(mutation & previous_population2$learning == "social"),
               prob = c(0.5, 0.5), replace = TRUE)

      population$learning[mutation & previous_population2$learning == "critical"] <-
        sample(c("individual", "social"),
               sum(mutation & previous_population2$learning == "critical"),
               prob = c(0.5, 0.5), replace = TRUE)

      # 7. Potential environmental change
      if (runif(1) < u) E <- E + 1
    }
  }
  # Export data from function
  output
}
```

Now we can run `rogers_model2()`, with the same parameter values as we initially ran `rogers_model()` in the last chapter.

```
data_model <- rogers_model2(N = 1000, t_max = 200, r_max = 10, c = 0.9, mu = 0.01, p = 1, u = 0.2
```

As before, it's difficult to see what's happening unless we plot the data. The following function `plot_prop()` now plots the proportion of all three learning strategies. To do this we need to convert our wide `data_model` tibble (where each strategy is in a different column) to long format (where all proportions are in a single column, and a new column indexes the strategy). To do this we use `pivot_longer()`, similarly to what we did in Chapter 7. For visualisation purposes, we also rename the variables that keep track of the frequencies of the strategies (`p_IL`, `p_SL`, `p_CL`) with full words. For this plot, we only visualise

the averages of all run with the `stat_summary()` function.

```
plot_prop <- function(data_model) {

  names(data_model)[3:5] <- c("social", "individual", "critical")
  data_model_long <- pivot_longer(data_model, -c(W, generation, run),
                                  names_to = "learning",
                                  values_to = "proportion")

  ggplot(data = data_model_long, aes(y = proportion, x = generation, colour = learning
    stat_summary(fun = mean, geom = "line", size = 1) +
    ylim(c(0, 1)) +
    theme_bw() +
    labs(y = "proportion of learners")
}

plot_prop(data_model)
```

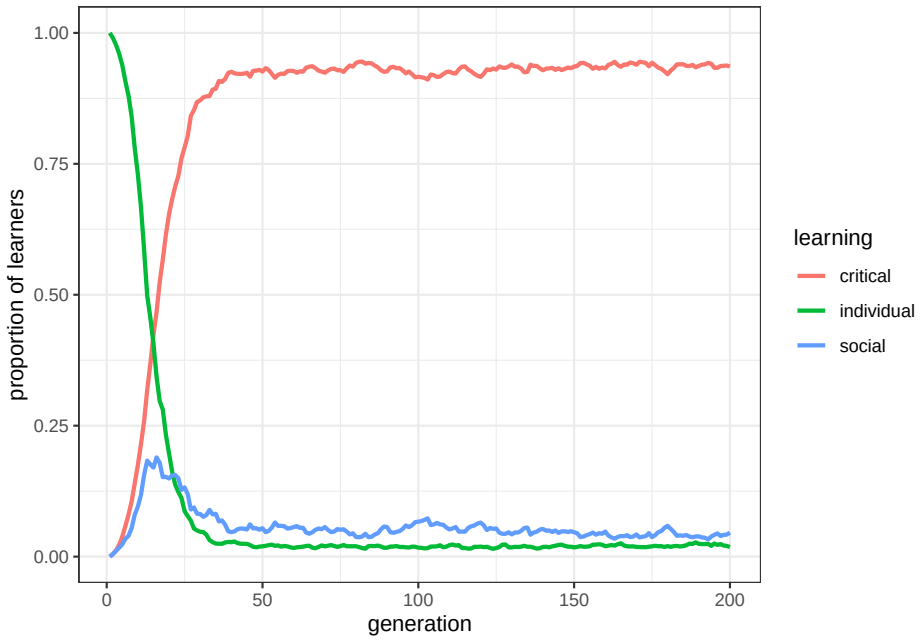


Figure 9.1: Critical learners quickly spread in populations of social learners and indiviual learners.

Here we can see that critical learners have a clear advantage over the other two learning strategies. Critical learners go virtually to fixation, barring mutation which prevents it from going to 100%. It pays off being a flexible, discerning learner who only learns individually when social learning does not work.

What about Rogers' paradox? Do critical learners exceed the mean fitness of a population entirely composed of individual learners? We can use the `plot_W()` function from the last chapter to find out:

```
plot_W(data_model, c = 0.9, p = 1)
```

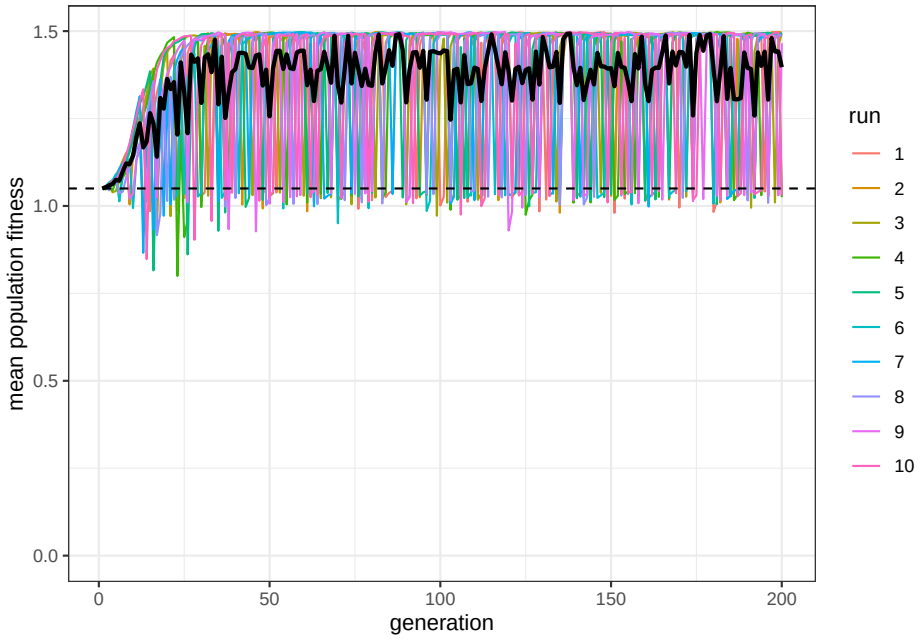


Figure 9.2: The average fitness of critical learners (black line) clearly exceeds the average fitness of populations entirely comprised of individual learners.

Yes: Even if there is still some noise, critical learners clearly outperform the dotted line indicating a hypothetical 100% individual learning population. Rogers' paradox is solved.

9.2 Summary of the model

Several 'solutions' have been demonstrated to Rogers' paradox. Here we have explored one of them. Critical learners can flexibly employ both social and individual learning, and do this in an adaptive manner (i.e. only individually learn if social learning is unsuccessful). Critical learners outperform the pure individual learning and pure social learning strategies. They therefore solve Rogers' paradox by exceeding the mean fitness of a population entirely composed of individual learning.

One might complain that all this is obvious. Of course real organisms can learn both socially and individually, and adaptively employ both during their lifetimes.

But hindsight is a wonderful thing. Before Rogers' model, scholars did not fully recognise this, and simply argued that social learning is adaptive because it has lower costs than individual learning. We now know this argument is faulty. But it took a simple model to realise it, and to realise the reasons why.

9.3 Further reading

There are several other solutions to Rogers' paradox in the literature. Boyd and Richerson [1995] suggested individuals who initially learn individually and then if unsatisfied learn socially - the reverse of Enquist et al. [2007]'s critical learners. Boyd and Richerson [1995] also suggested that if culture is cumulative, i.e. each generation builds on the beneficial modifications of the previous generations, then Rogers' paradox is resolved.

Advanced topics - Cultural inheritance

Chapter 10

Reproduction and transformation

To be considered 'cultural', ideas, behaviours, and artefacts need to be sufficiently stable in time. The version of *Little Red Riding Hood* we heard now is part of a long cultural transmission chain that includes all the other versions of the tale because they all share *enough* features to be considered the same tale. Similarly, the lasagne I cooked yesterday are part of a long, intricate, chain of cultural transmission events, where all the products are stable enough that we can consider all of them as one cultural trait: *lasagne*. Boundaries are muddled for many traits: while some artefacts can be the exact replica of each other, no two identical lasagne exist. In any case, the question we explore in this chapter is: how does this stability is brought about? In the models so far, as much as in the majority of models in cultural evolution, we assumed that traits are copied from one (cultural) generation to another with enough fidelity to assure a relative stability. This is a useful assumption and, in many case, a good approximation of what happens in reality.

However, cultural traits can be stable not because they are copied with high-fidelity, but because, when passing from an individual to another, they are independently reconstructed in the same way or, another way to say it, they become similar to each other through a process of convergent transformation. Think about whistling. We do learn to whistle from each other through a process of cultural transmission (we want to reproduce what others do), but the configuration of the muscles in the mouth is not something that we copy directly. Still, there are few ways to effectively whistle, so that we likely end up with the same - or similar - configuration. (Notice we can also actually copy the exact configuration and, indeed, there are specialised whistling techniques for which it is required. As we will mention again later, copying and reconstructing are not two alternative processes, but they both concur to cultural evolution.)

Evolutionary psychologists and some anthropologists emphasise how certain cultural traditions are similar in many societies, such as supernatural beliefs, types of musics, or what people find or not disgusting. These similarities do not need to be produced by genetically encoded preferences, but it suffices that some general tendencies make more likely that people everywhere will converge on these quasi-universal forms. A psychological tendency to interpret the behaviour of an entity as intentional (even if this entity is an inanimate object) could give rise to similarity in supernatural beliefs, as much as the physical property of the mouth give rise to similarity in how people whistle everywhere.

10.1 Copying and selection

To have a better grasp of the consequence of this idea we can, as usual, try to model a very simple case, where cultural stability can be obtained with a process of copying and selection of a model, as we did in many of the previous chapters, or with convergent transformation, where individuals are not very good at copying, or at selecting models, but they tend to transform the trait in the same way.

Let's imagine a population with a single trait, a continuous trait P, that can have values between 0 and 1. At the beginning of the simulations, P is uniformly distributed in the population. Let's say the optimal value of P is 1 (this is convenient for the code, but the exact value is not important). You can think to P as, for example, how sharp is a knife: the sharper the better.

```
library(tidyverse)
N <- 1000
population <- tibble(P = runif(N))
```

Now, we can write the familiar function where individuals copy the trait from the previous generation with one of the biases we explored earlier in the book. In Chapter 3, for example, we showed how a direct bias for one of two discrete cultural traits could make it spread and go to fixation. We can do something similar here, with the difference that the trait is continuous and the bias needs to be a preference for traits close to the optimal value. (Notice the code would be equivalent - and we would obtain the same effect of convergence to optimal value - thinking in terms of other methods of cultural selection, e.g. an indirect bias towards successful demonstrators, that are successful as they have a P close to the optimal).

```
reproduction <- function(N, t_max, r_max, mu) {
  output <- tibble(generation = rep(1:t_max, r_max),
                   p = as.numeric(rep(NA, t_max * r_max)),
                   run = as.factor(rep(1:r_max, each = t_max)))
  for (r in 1:r_max) {
    # Create first generation
    population <- tibble(P = runif(N))
```

```
    # Add first generation's p for run r
    output[output$generation == 1 & output$run == r, ]$p <- sum(population$P) / N

    for (t in 2:t_max) {
      # Copy individuals to previous_population tibble
      previous_population <- population

      # Select a pair of demonstrator for each individual
      demonstrators <- tibble(P1 = sample(previous_population$P, N, replace = TRUE),
                              P2 = sample(previous_population$P, N, replace = TRUE))

      # Copy the one with the trait value closer to 1
      copy <- pmax(demonstrators$P1, demonstrators$P2)

      # Add mutation
      population$P <- copy + runif(N, -mu, +mu)

      # Keep the  traits' value in the boundaries [0,1]
      population$P[population$P > 1] <- 1
      population$P[population$P < 0] <- 0

      # Get p and put it into output slot for this generation t and run r
      output[output$generation == t & output$run == r, ]$p <-
        sum(population$P) / N
    }
  }
  # Export data from function
  output
}
```

The function is similar to what we have already done several times. Let's have a look at the few differences. First, we find again the parameter μ, as done in various previous chapters. Similarly, it implements here the error in copying: with respect to the P of the demonstrator chosen, the new trait will vary of maximum of μ, through the instruction `runif(N, -mu, +mu)`. The two following lines just keep the traits in the boundaries between 0 and 1. The second difference is in the selection of the trait to copy. Here each individual sample two traits (or demonstrators) from the previous generation, and simply copies the one with the trait closer to the optimal value of 1.

We can now run the simulation, and plot it with a slightly modified function `plot_multiple_runs_p()` (we just need to change the label for y-axis). We use a low value for the copying error, such as $\mu = 0.05$.

```
plot_multiple_runs_p <- function(data_model) {
  ggplot(data = data_model, aes(y = p, x = generation)) +
```

```
    geom_line(aes(colour = run)) +
    stat_summary(fun = mean, geom = "line", size = 1) +
    ylim(c(0, 1)) +
    theme_bw() +
    labs(y = "p (average value of P)")
}
```

```
data_model <- reproduction(N = 1000, t_max = 20, r_max = 5, mu = 0.05)
plot_multiple_runs_p(data_model)
```

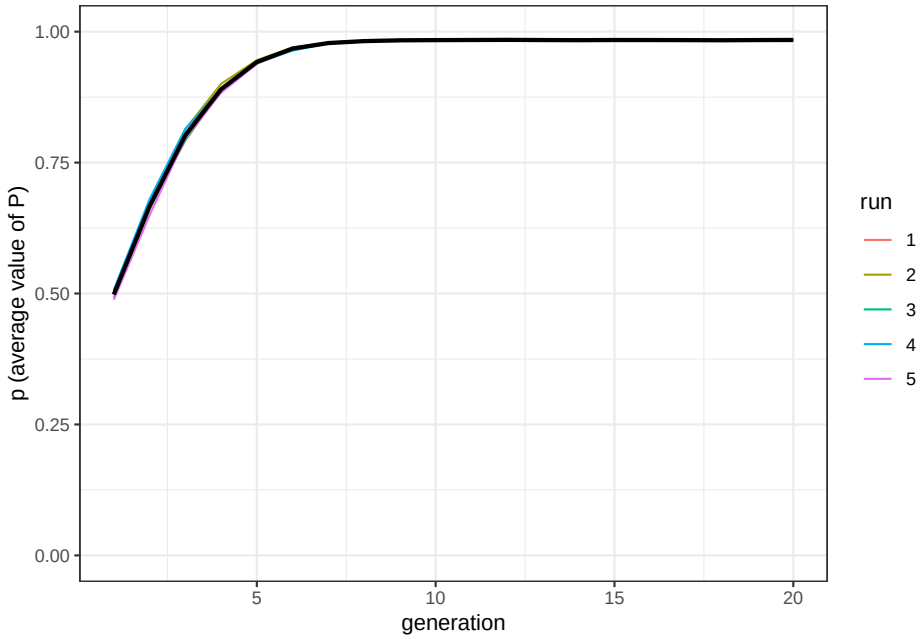


Figure 10.1: The populations reach the optimal trait value with cultural selection.

Even with a weak form of selection (sampling two traits and choosing the better one) the population converges on the optimal value quickly, in only around ten cultural generations.

10.2 Convergent transformation

Now we can write another function where convergent transformation produces the same effect.

```
transformation <- function(N, t_max, r_max) {
  output <- tibble(generation = rep(1:t_max, r_max),
                   p = as.numeric(rep(NA, t_max * r_max)),
```

```
                    run = as.factor(rep(1:r_max, each = t_max)))

  for (r in 1:r_max) {
    # Create first generation
    population <- tibble(P = runif(N))

    # Add first generation's p for run r
    output[output$generation == 1 & output$run == r, ]$p <- sum(population$P) / N

    for (t in 2:t_max) {
       # Copy individuals to previous_population tibble
      previous_population <- population

      # Only one demonstrator is selected at random for each individual
      demonstrators <- tibble(P = sample(previous_population$P, N, replace = TRUE))

      # The new P is in between the demonstrator's value and 1
      population$P <- demonstrators$P + runif(N, max = 1 - demonstrators$P)

      # Get p and put it into output slot for this generation t and run r
      output[output$generation == t & output$run == r, ]$p <- sum(population$P) / N
    }
  }
  # Export data from function
  output
}
```

There is only a line we need to pay attention to: the values of P of the new generation are calculated as `demonstrators$P + runif(N, max = 1-demonstrators$P)`. This means that the individuals of the new population copy the old generation, but they are not particularly good. They take a value of P randomly drawn between the value of the demonstrator and the optimal $P = 1$. Thus, if they attempt to copy a demonstrator with $P = 0.1$, their 'error' can be as large as 0.9. While modifications can be big, they are all in the same direction, contributing to increase P. Let's run the simulations.

```
data_model <- transformation(N = 1000, t_max = 20, r_max = 5)
plot_multiple_runs_p(data_model)
```

As the transformations tend to all converge in the same direction, the results are equivalent to the previous model. It does not matter when exactly the population reaches stability at $P = 1$, as this depends on the specific implementation choices, for example the strength of cultural selection in the first model, or how big can be the 'jump' of the transformation in the second model (you can try to modify those by yourself).

When we see cultural traits in real life, we are observing systems in a state

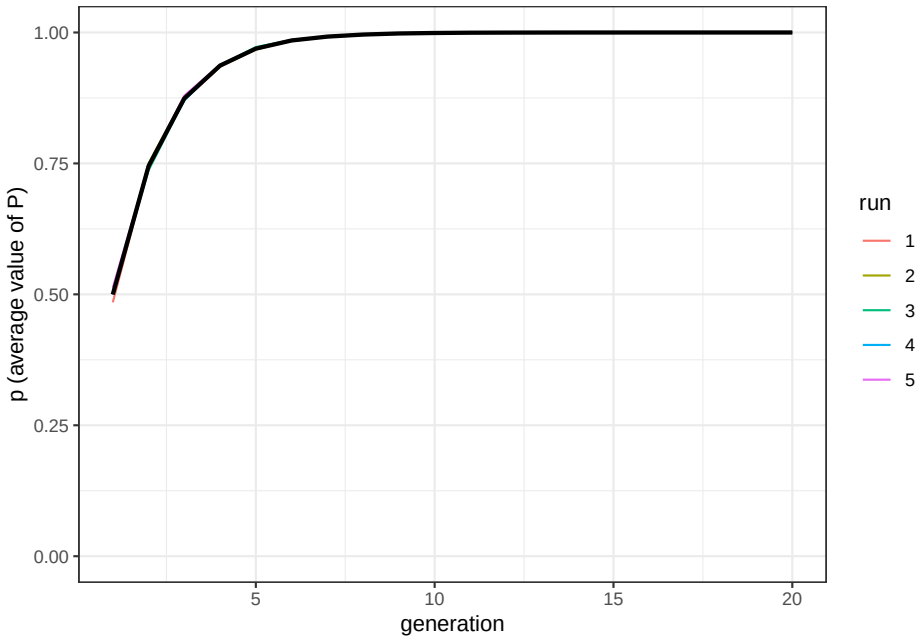


Figure 10.2: The population reach the optimal trait value when transformations converge towards the optimal value.

analogous to what happens on the right side of the two plots, where individuals reproduce traits one similar to the other. As we touched earlier in the chapter, both faithful copying coupled with selection and transformation can be important for culture, and their importance can depend from the specific features we are interested to track, or from the cultural domain. As we have just shown, however, the spread of the traits in the population looks similar in both cases. Are there ways to distinguish the relative importance of the two processes?

10.3 Emergent similarity

One possibility is to track how similar are the traits that the observers reproduce, with respect to the traits they use as a starting point. If reproduction is the driving force, they should be fairly similar and the measure should be always the same, as the distance between the trait produced and the trait copied is fixed, and given by the parameter μ. If transformation is the driving force, instead, we should expect similarity being lower when traits are far from the optimal value (i.e. at the beginning of the simulations), and higher when traits are close to the optimal value.

We can rewrite the `reproduction()` and `transformation()` functions adding as a further output this measure of similarity, that is, how distant are the traits

that the new generation show versus the traits that they had copied from the previous. We thus add a variable `d` (as in 'distance') in our `output` tibble, and we calculate this value at the end of each generation as `sum(abs(population$P - copy)) / N`.

```
reproduction <- function(N, t_max, r_max, mu) {
  output <- tibble(generation = rep(1:t_max, r_max),
                   p = as.numeric(rep(NA, t_max * r_max)),
                   run = as.factor(rep(1:r_max, each = t_max)),
                   d = as.numeric(rep(NA, t_max * r_max)))
  for (r in 1:r_max) {
    # Create first generation
    population <- tibble(P = runif(N))

    # Add first generation's p for run r
    output[output$generation == 1 & output$run == r, ]$p <- sum(population$P) / N

    for (t in 2:t_max) {
      # Copy individuals to previous_population tibble
      previous_population <- population

      demonstrators <- tibble(P1 = sample(previous_population$P, N, replace = TRUE),
                              P2 = sample(previous_population$P, N, replace = TRUE))

      copy <- pmax(demonstrators$P1, demonstrators$P2)

      population$P <- copy + runif(N, -mu, +mu)
      population$P[population$P > 1] <- 1
      population$P[population$P < 0] <- 0

      # Output:
      output[output$generation == t & output$run == r, ]$p <- sum(population$P) / N
      output[output$generation == t & output$run == r, ]$d <- sum(abs(population$P - copy)) / N
    }
  }
  # Export data from function
  output
}
```

We do the same for the `transformation()` function, and we can run again both simulations.

```
transformation <- function(N, t_max, r_max) {
  output <- tibble(generation = rep(1:t_max, r_max),
                   p = as.numeric(rep(NA, t_max * r_max)),
                   run = as.factor(rep(1:r_max, each = t_max)),
                   d = as.numeric(rep(NA, t_max * r_max)))
```

```
  for (r in 1:r_max) {
    # Create first generation
    population <- tibble(P = runif(N))

    # Add first generation's p for run r
    output[output$generation == 1 & output$run == r, ]$p <- sum(population$P) / N

    for (t in 2:t_max) {
      # Copy individuals to previous_population tibble
      previous_population <- population

      demonstrators <- tibble(P = sample(previous_population$P, N, replace = TRUE))

      population$P <- demonstrators$P + runif(N, max = 1-demonstrators$P)

      # Output
      output[output$generation == t & output$run == r, ]$p <- sum(population$P) / N
      output[output$generation == t & output$run == r, ]$d <- sum(abs(population$P - de
    }
  }
  # Export data from function
  output
}
```

```
data_model_reproduction <- reproduction(N = 1000, t_max = 20, r_max = 5, mu = 0.05)
data_model_transformation <- transformation(N = 1000, t_max = 20, r_max = 5)
```

We already know the results with respect to the value of P, but now we are interested in comparing if and how the values for d change in time in the two conditions. For this, we write an *ad hoc* plotting function, that takes the data from the two outputs and plot them in the same graph. Notice the `na.omit()` function in the first line: the data on `d` is NA for the first generation, because there is no previous generation from which to take the measure, so we want to exclude it from our plot, and start from generation number 2. For this reason, all the other values are rescaled and, in particular, the variable `generation` starts from 2.

```
data_to_plot <- tibble(distance = c(na.omit(data_model_reproduction$d),
                                    na.omit(data_model_transformation$d)),
                       condition = rep(c("reproduction", "transformation"), each = 95)
                       generation = rep(2:20,10),
                       run = as.factor(rep(1:10, each = 19)))
ggplot(data = data_to_plot, aes(y = distance, x = generation, group = run, color = cond
  geom_line() +
  geom_point() +
  theme_bw() +
```

```
  labs(y = "d (average distance observer/demonstrator)")
```

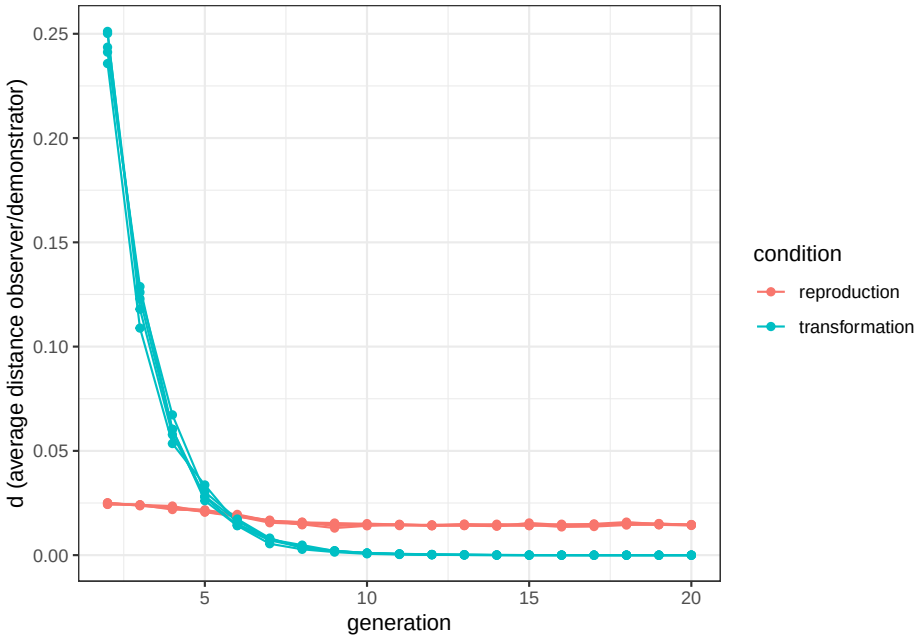


Figure 10.3: When convergent transformation is the driving force, the similarity between original and copied items starts high and decreases with time. When cultural selection is the driving force, the similarity is constant.

As predicted, in the 'transformation' condition distance is higher at the beginning of the simulation, and reaches zero when all individuals have the optimal value. In the "reproduction" condition, instead, the distance is approximately constant. In fact, it slightly decreases after the first few generations. This is due to the fact that at the beginning, with P randomly distributed in the population, the mutation is effectively drawn between $P - \mu$ and $P + \mu$, but after a while, when demonstrators have a value close to the optimal $P = 1$, the mutation is only drawn in $P - \mu$, as values of P higher than 1 are not allowed.

10.4 Cultural fitness

Another way to look at the difference between the two conditions, focusing on the process of selection, is to look at the 'cultural fitness' of the individuals in the population. More in detail, one can look at how this metric covaries with how good they actually are, as given by their value of P. We can define W, a measure of cultural fitness, as the number of "cultural offspring" that the individual i has in the next generation. If individual i has been copied by, say, four individuals, its fitness will be $W_i = 4$.

If individuals, or their traits, are selected, as happens in the "reproduction" condition, we expect that individuals with higher values of P have more cultural offspring, thus, that to higher Ps correspond higher Ws. We expect, in other words, the covariance between W and P being positive i.e. $cov(W, P) > 0$. On the other hand, in the condition "transformation" there is no selection, and there are no reasons why an individual with higher P produces more cultural offspring. In this case, the covariance should be zero, i.e. $cov(W, P) = 0$.

To calculate cultural fitness, and how it covaries with P, we need to modify again our functions. Let's start, as before, with `reproduction()`:

```
reproduction <- function(N, t_max, r_max, mu) {
  output <- tibble(generation = rep(1:t_max, r_max),
                   p = as.numeric(rep(NA, t_max * r_max)),
                   run = as.factor(rep(1:r_max, each = t_max)),
                   cov_W_P = as.numeric(rep(NA, t_max * r_max)))
  for (r in 1:r_max) {
    # Create first generation
    population <- tibble(P = runif(N))

    # Add first generation's p for run r
    output[output$generation == 1 & output$run == r, ]$p <- sum(population$P) / N

    for (t in 2:t_max) {
      # Copy individuals to previous_population tibble
      previous_population <- population

      # Sample the demonstrators using their indexes
      demonstrators <- cbind(sample(N, N, replace = TRUE), sample(N, N, replace = TRUE)
      # Retrieve their traits from the indexes
      copy <- max.col(cbind(previous_population[demonstrators[,1],],
                            previous_population[demonstrators[,2],]))
      # Save the demonstrators
      demonstrators <- demonstrators[cbind(1 : N, copy)]
      fitness <- tabulate(demonstrators, N)

      population$P <- previous_population[demonstrators,]$P + runif(N, -mu, +mu)
      population$P[population$P > 1] <- 1
      population$P[population$P < 0] <- 0

      # Output
      output[output$generation == t & output$run == r, ]$p <-
        sum(population$P) / N
      output[output$generation == t & output$run == r, ]$cov_W_P <-
        cov(fitness, previous_population$P)
    }
```

```
  }
  # Export data from function
  output
}
```

The function produces the usual output, but in a different way. To measure W, we need to know the actual individuals that are copied, not only their P values, as we were doing previously. For this reason, the sampling of the demonstrators is done on their indexes with the instruction `sample(N, N, replace = TRUE)`. Then, the indexes are used to retrieve their P, in the two following lines. Finally, we count how many times each index, that is, each individual, is used as demonstrator, using the function `tabulate()`, introduced in Chapter 7.

Now we need to do the same for `transformation()`:

```
transformation <- function(N, t_max, r_max) {
  output <- tibble(generation = rep(1:t_max, r_max),
                   p = as.numeric(rep(NA, t_max * r_max)),
                   run = as.factor(rep(1:r_max, each = t_max)),
                   cov_W_P = as.numeric(rep(NA, t_max * r_max)))
  for (r in 1:r_max) {
    # Create first generation
    population <- tibble(P = runif(N))

    # Add first generation's p for run r
    output[output$generation == 1 & output$run == r, ]$p <- sum(population$P) / N
    for (t in 2:t_max) {
      # Copy individuals to previous_population tibble
      previous_population <- population

      # Choose demonstrators and calcualte  their fitness
      demonstrators <- sample(N, N, replace = TRUE)
      fitness <- tabulate(demonstrators, N)

      population$P <- previous_population[demonstrators,]$P +
        runif(N, max = 1 - previous_population[demonstrators,]$P)

      # Output
      output[output$generation == t & output$run == r, ]$p <- sum(population$P) / N
      output[output$generation == t & output$run == r, ]$cov_W_P <- cov(fitness,previous_populati
    }
  }
  # Export data from function
  output
}
```

The logic is exactly the same, only we do not need to use the indexes to retrieve

the P values, as they are not needed to choose demonstrators. At this point, as before, we can run the simulations in the two conditions, and plot the results (the code for the plot is also the same, only the label for the y-axes changes).

```
data_model_reproduction <- reproduction(N = 1000, t_max = 20, r_max = 5, mu = 0.05)
data_model_transformation <- transformation(N = 1000, t_max = 20, r_max = 5)

data_to_plot <- tibble(covariance = c(na.omit(data_model_reproduction$cov_W_P),
                                      na.omit(data_model_transformation$cov_W_P)),
                       condition = rep(c("reproduction", "transformation"), each = 95),
                       generation = rep(2:20,10),
                       run = as.factor(rep(1:10, each = 19)))
ggplot(data = data_to_plot, aes(y = covariance, x = generation,
                                group = run, color = condition)) +
  geom_line() +
  geom_point() +
  theme_bw() +
  labs(y = "covariance between cultural fitness and P")
```

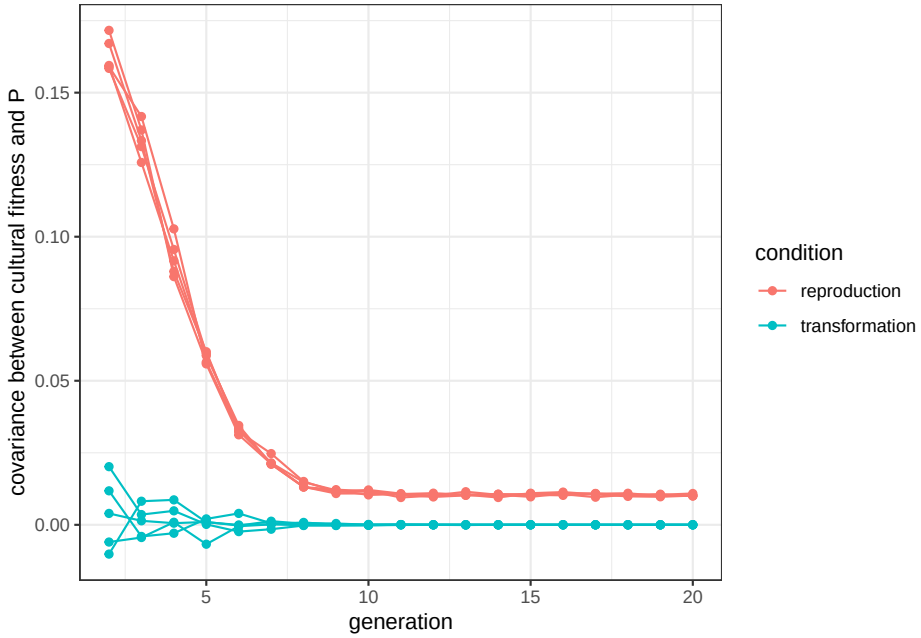


Figure 10.4: When cultural selection is the driving force, better cultural items are more likely to be copied (until the population converges to optimal values). When convergent transformation is the driving force, there is no relationship between quality and cultural success.

In the 'reproduction' condition, the covariance is indeed positive, and decreases

gradually close to zero, when all the individuals converge to $P = 1$, as there is no more variation on which selection can act. Notice it does not reach zero, as mutation keeps some variation, and individuals that muted to lower Ps are less likely to be selected. As expected, the covariance is equal to zero in the 'transformation' condition. This is hardly a surprising result as demonstrators are selected fully at random in the model, but it is important to compare this with what happens with empirical data of real cultural dynamics, where we can be able to distinguish different underlying stabilising forces.

10.5 Summary of the model

Cultural traditions can survive intact through long and wide transmission chains because cultural traits are copied faithfully, because some of them are copied more than others (cultural selection), and because everybody involved in the episodes of transmission tend to reproduce them in a similar way. All these forces are likely to be important, to a various degree, in different domains and for different features of cultural traits. While in the rest of the book we have focused on copying and selection, in this chapter we have considered transformation. We have shown that both copying plus selection and convergent transformation create stable cultural systems, where all individuals in the population converge on similar cultural traits. We also explored how these forces can be distinguished. One possible way is to chart the similarity between the cultural traits observed and the cultural traits reproduces: in the case of transformation depends on the specific feature of the traits (the closer to the end-point of the convergence, the higher the similarity), whereas for reproduction we should expect similarity to be constant, depending on how, generally, copying is precise. Another way is to detect whether demonstrators with certain traits (or certain features of a trait) are copied more: this is the sign of selection, that characterised our 'reproduction' model, but not the 'transformation' one.

10.6 Further readings

The model comparing reproduction and transformation is a simplified version of the model described in Acerbi et al. [2019]. The analysis to detect signs (similarity and cultural fitness) of the two forces are inspired also by the application of the Price Equation to cultural evolution in Nettle [2020]. An early account of the importance of convergent transformation ('cultural attraction') in cultural evolution is Sperber [1997]. A discussion of the relative importance of transformation and reproduction in cultural evolution, and of the necessity to consider both is Acerbi and Mesoudi [2015].

Chapter 11

Social learning of social learning rules

In most of the models we explored so far, individuals decide whether to copy or not according to various rules, often called 'transmission biases' in cultural evolution jargon. They may have a tendency to copy common traits, or to copy a subset of the population, or to prefer certain cultural traits with respect to others by virtue of their intrinsic characteristics, and so on.

A feature of all these models is that these rules were considered stable, or changing very slowly (perhaps because of genetic evolution) in comparison to the timescale of the model, so that we effectively treated them as fixed. However, cultural evolution can also influence its own rules, that is, we can learn from others when, what, or from whom to learn. This is far from being a rare instance: parents, at least in modern western societies, invest much effort to transmit to children that learning from schoolteachers is important, or teenagers groups may discourage learning from other groups, or from adults in general. Educational systems in countries such as Korea or Japan are thought to encourage pupils to learn and trust teachers almost unconditionally, whereas, in countries like UK and USA, the emphasis is on individual creativity and critical thinking.

11.1 Openness and conservatism

How can we approach the social learning of social learning rules with simple models? To start with, we can imagine that individuals learn from others whether to copy others or not. We can imagine the simplest possible dynamic, where a single trait, P, both regulate the probability to copy from others and is the trait that is actually copied. When an individual has $P = 1$ always copies others (we will call it a completely 'open' individual), and when it has $P = 0$ never copies

others (we will call it a completely 'conservative' individual). All intermediate values of P are possible.

```
library(tidyverse)
N <- 1000
population <- tibble(P = runif(N))
```

After initialising the population with a random uniform sample of values of P, we can write the function to run the simulations.

```
openness_conservatism <- function(N, t_max, r_max) {
  output <- tibble(generation = rep(1:t_max, r_max),
                   p = as.numeric(rep(NA, t_max * r_max)),
                   run = as.factor(rep(1:r_max, each = t_max)))
  for (r in 1:r_max) {
    # Create first generation
    population <- tibble(P = runif(N))

    # Add first generation's p for run r
    output[output$generation == 1 & output$run == r, ]$p <-
      sum(population$P) / N

    for (t in 2:t_max) {
      # Copy individuals to previous_population tibble
      previous_population <- population

      # Choose demonstrators at random
      demonstrators <- tibble(P = sample(previous_population$P, N, replace = TRUE))

      # Choose individuals that copy, according to their P
      copy <- previous_population$P > runif(N)

      # Copy
      population[copy, ]$P <- demonstrators[copy, ]$P

      # Get p and put it into output slot for this generation t and run r
      output[output$generation == t & output$run == r, ]$p <-
        sum(population$P) / N
    }
  }
  # Export data from function
  output
}
```

Everything should be familiar in this function. The only new instruction is in the line `copy <- previous_population$P > runif(N)`. This simply compares each individual's P value with a random number extracted between 0 and 1. If

the P value is higher, the individual will copy, otherwise it will not.

We can now run the simulation, and plot it with the `plot_multiple_runs_p()` function for continuous traits we wrote in the previous chapter.

```
plot_multiple_runs_p <- function(data_model) {
  ggplot(data = data_model, aes(y = p, x = generation)) +
    geom_line(aes(colour = run)) +
    stat_summary(fun = mean, geom = "line", size = 1) +
    ylim(c(0, 1)) +
    theme_bw() +
    labs(y = "p (average value of P)")
}
```

```
data_model <- openness_conservatism(N = 1000, t_max = 50, r_max = 5)
plot_multiple_runs_p(data_model)
```

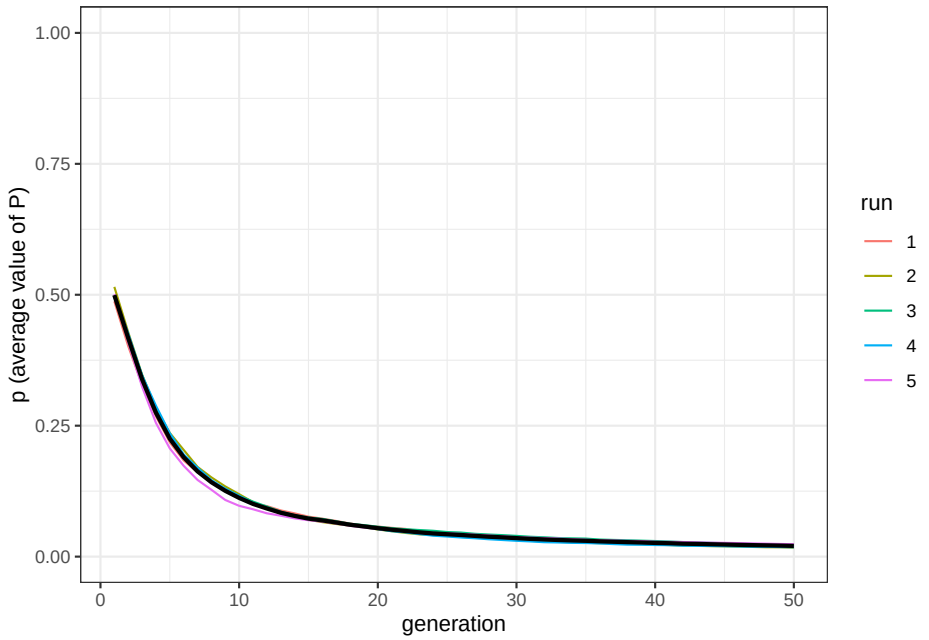


Figure 11.1: After few generations, the popualtion is composed by conservative individuals.

The average value of P in the population quickly converges towards 0 (in fact, towards the lower initial value, as there are no mutations) in all runs. At this point of the book, you should be able to introduce mutations, as well as initialising the population with different values of P. What would happen, for example, if individuals start with values of P clustering around 1, that is, they are all initially very open? Another possible modification is that, instead of

comparing the copier's P value with a random number, when two individuals are paired, the individual with the higher P (that is, the most open of the two) copies the other one.

At the risk of ruining the surprise, the main result of populations converging towards maximum conservatism is robust to many modifications (but you should try your own, this is what models are about). The result seems at first sight counterintuitive: the outcome of social transmission is to eliminate social transmission! A way to understand this result is that conservative individuals, exactly because they are conservative, change less than open individuals and, in general, transitions from open to conservative happen more frequently than transitions from conservative to open. Imagine a room where people are all copying the t-shirt colours of each other, but one stubborn individual, with a red t-shirt, never changes. If there are not other forces acting, at some point all individuals will wear red t-shirts.

11.2 Maintaining open populations

The result above highlights a possibly interesting aspect of what could happen when social learning rules are themselves subject to social learning, but it does not represent, of course, what happens in reality. Some models, such as the Rogers' model we explored in [chapter 8][Rogers' model], are useful exactly because they force us to think how reality differs from the modelled situation. Individuals, in real life, remain open because learning from others is, on average, effective, and increases their fitness.

However, even without considering the possible fitness advantages of copying from others, there may be other reasons why individuals remain open to cultural influences. We can add a bit of complexity to the previous model and see what happens. For example, instead of having a single P value, individuals can be "open" or "conservative" depending on the specific cultural trait they observe. One can be open to try exotic recipes, while another may like only its local cuisine; one want to know everything about combat sports, while another prefers watching them in TV. We can say that, instead of a single P, we have many preferences associated to cultural traits and, as before, they can be transmitted from one individual to another. Second, we decide to copy other individuals depending on our preferences for the traits they show us.

Finally, differently from the models we explored in the previous chapters, individuals in the population are replaced through a birth/death process. They are born without cultural traits, and they acquire them during the course of their life, by copying them from others, or by introducing them through innovation. The new function `openness_conservatims_2()` does all the above.

```
openness_conservatism_2 <- function(N, M, mu, p_death, t_max, r_max){
  output <- tibble(generation = rep(1:t_max, r_max),
                   p = as.numeric(rep(NA, t_max * r_max)),
```

```
                  m = as.numeric(rep(NA, t_max * r_max)),
                  run = as.factor(rep(1:r_max, each = t_max)))

for (r in 1:r_max) {

  # Initialise population
  population_preferences <- matrix( runif(M * N), ncol = M, nrow = N)
  population_traits <- matrix(0, ncol = M, nrow = N)

  # Write first output
  output[output$generation == 1 & output$run == r, ]$p <- mean(population_preferences)
  output[output$generation == 1 & output$run == r, ]$m <- sum(population_traits) / N

  for(t in 2:t_max){
    # Innovations
    innovators <- sample(c(TRUE, FALSE), N, prob = c(mu, 1 - mu), replace = TRUE)
    innovations <- sample(1:M, sum(innovators), replace = TRUE)
    population_traits[cbind(which(innovators == TRUE), innovations)] <- 1

    # Copying
    previous_population_preferences <- population_preferences
    previous_population_traits <- population_traits

    demonstrators <- sample(1:N, replace = TRUE)
    demonstrators_traits <- sample(1:M, N, replace = TRUE)

    copy <- previous_population_traits[cbind(demonstrators,demonstrators_traits)] == 1 &
      previous_population_preferences[cbind(1:N, demonstrators_traits)] > runif(N)

    population_traits[cbind(which(copy), demonstrators_traits[copy])] <- 1

    population_preferences[cbind(which(copy), demonstrators_traits[copy])] <-
      previous_population_preferences[cbind(demonstrators[copy], demonstrators_traits[copy])]

    # Birth/death
    replace <- sample(c(TRUE, FALSE), N, prob = c(p_death, 1 - p_death), replace = TRUE)
    population_traits[replace, ] <- 0
    population_preferences[replace, ] <- runif(M * sum(replace))

    # Write output
    output[output$generation == t & output$run == r, ]$p <- mean(population_preferences)
    output[output$generation == t & output$run == r, ]$m <- sum(population_traits) / N
  }
}
# Export data from function
```

```
  output
}
```

The population is now described by two matrices, `population_preferences` and `population_traits`, that are initialised, respectively, with random number between 0 and 1 and with all 0s, respectively, meaning that at the beginning there are no traits in the population. The same happens for newborns. A parameter of the simulation, M, gives the maximum possible number of traits. At each time step, a proportion μ of innovators introduce a trait at random.

The main novelties of the code are in the copying procedure. After selecting random demonstrators and, for each of them, a random trait-slot, we record in the variable `copy` whether or not the individuals that will copy the demonstrator. For this to happen, the demonstrator needs to actually possess the trait randomly selected (`previous_population_traits[cbind(demonstrators,demonstrators_traits)]==1`) and the preference of the observer for that trait should be sufficiently high (`previous_population_preferences[cbind(1:N, demonstrators_traits)] > runif(N)`). If these two conditions are satisfied, the observer copies both the trait and the preference of the demonstrator.

We can start with a situation similar to the previous model, with only a single trait ($M = 1$). We set a relatively high innovation rate ($\mu = 0.1$) so that the initial population is quickly populated by cultural traits, and $p_{\text{death}} = 0.01$, meaning that, with a population of 100 individuals, every time step there will be on average one newborn. (As usual, you are invited to explore the effect of these parameters.)

```
data_model <- openness_conservatism_2(N = 1000, M = 1, mu = 0.1,
                                      p_death = 0.01, t_max = 50, r_max = 5)
plot_multiple_runs_p(data_model)
```

The plot is fairly similar to what we saw before. The average openness of the population converges towards lower values in few generations, in all runs. The descent is less steep since at the beginning of the simulations individuals need to acquire cultural traits to kick start social transmission. We can now try with an higher number of possible traits, for example $M = 10$.

```
data_model <- openness_conservatism_2(N = 1000, M = 10, mu = 0.1,
                                      p_death = 0.01, t_max = 50, r_max = 5)
plot_multiple_runs_p(data_model)
```

Now the convergence seems slower. We can try with longer simulations, fixing $t_{\max} = 1000$.

```
data_model <- openness_conservatism_2(N = 1000, M = 10, mu = 0.1,
                                      p_death = 0.01, t_max = 1000, r_max = 5)
plot_multiple_runs_p(data_model)
```

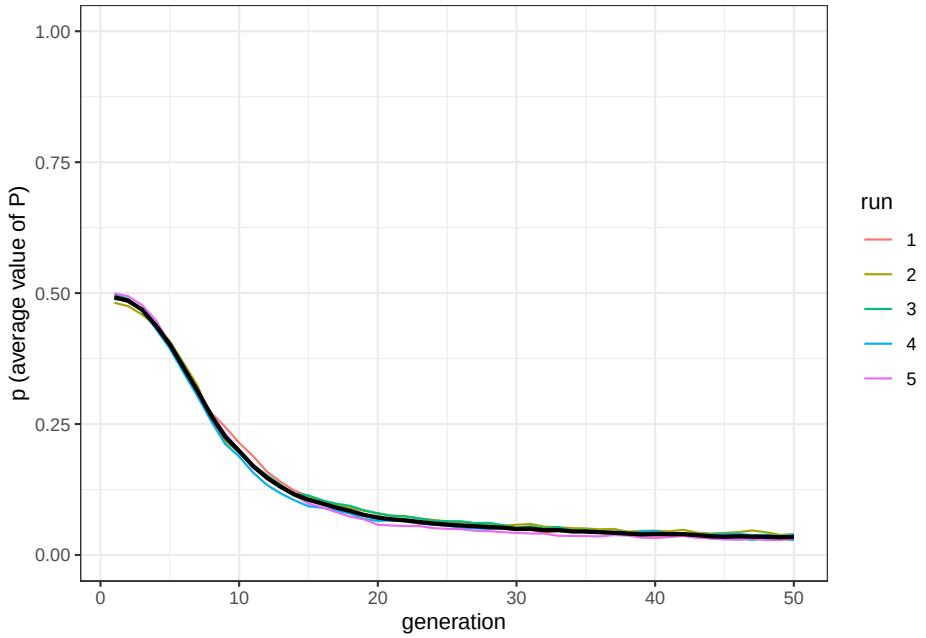


Figure 11.2: Simlarly to the previous model, the popualtion converges to conservatism, even if the descent is less steep as individuals need some time to acquire traits.

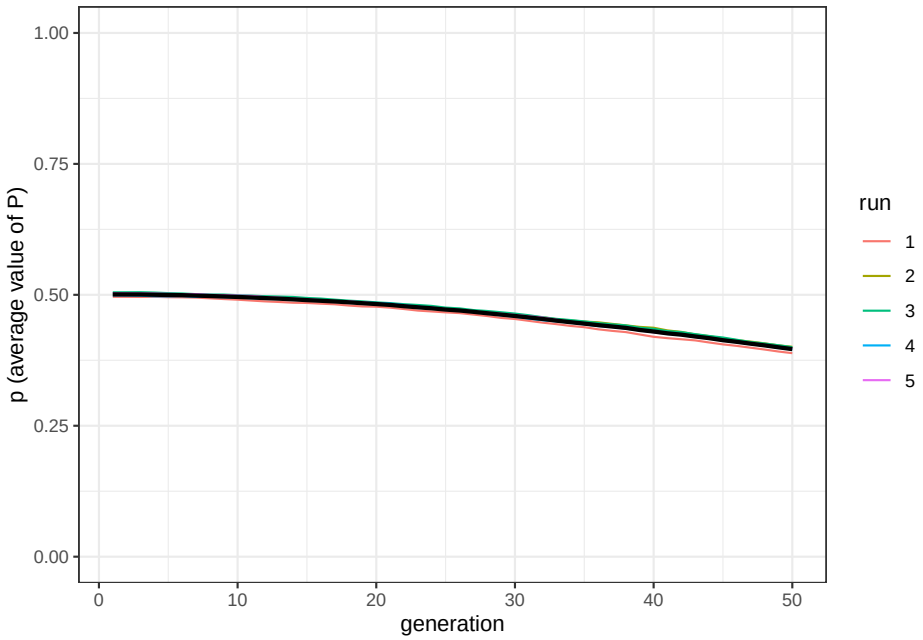


Figure 11.3: With 10 possible traits, convergence to conservatism is slower.

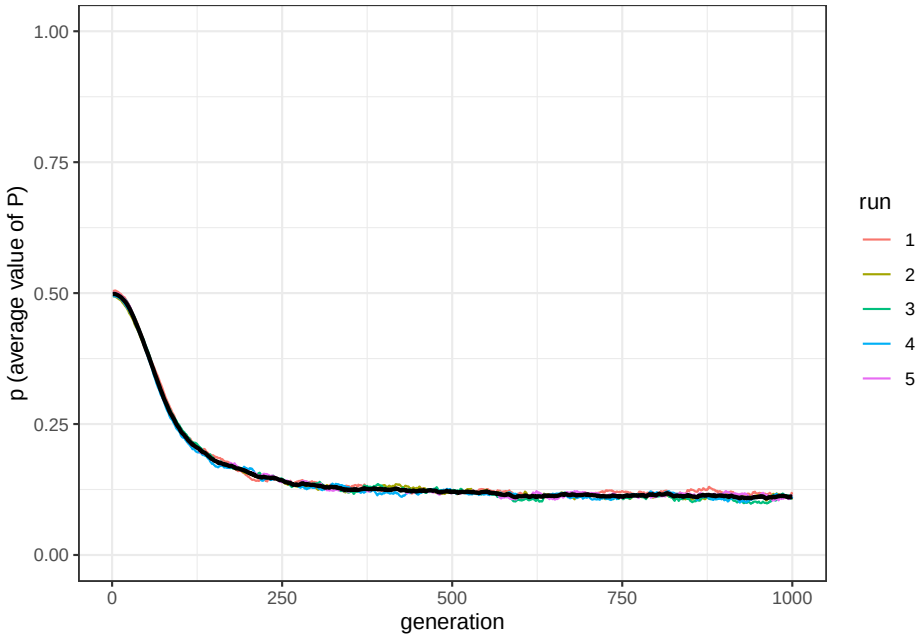


Figure 11.4: Even after 1,000 generations, with 10 possible traits, individuals are not completely conservative.

Even after 1000 generations, population openness did not go to 0, but it stabilises on a value of around 0.12. To understand what happens it is interesting to plot the other value we are recording in the output of the simulation, that is the average number of traits that individuals possess. The function below is equivalent to the usual `plot_multiple_runs()`, but with a different y-axis label, and it takes M (the maximum number of traits) as a parameter, so that we can set the y-axis to span from 0 to M, to have a better visual estimate of the proportion of traits present with respect to the maximum possible.

```
plot_multiple_runs_m <- function(data_model, M) {
  ggplot(data = data_model, aes(y = m, x = generation)) +
    geom_line(aes(colour = run)) +
    stat_summary(fun = mean, geom = "line", size = 1) +
    ylim(c(0, M)) +
    theme_bw() +
    labs(y = "m (average number of traits)")
}
```

```
plot_multiple_runs_m(data_model, M = 10)
```

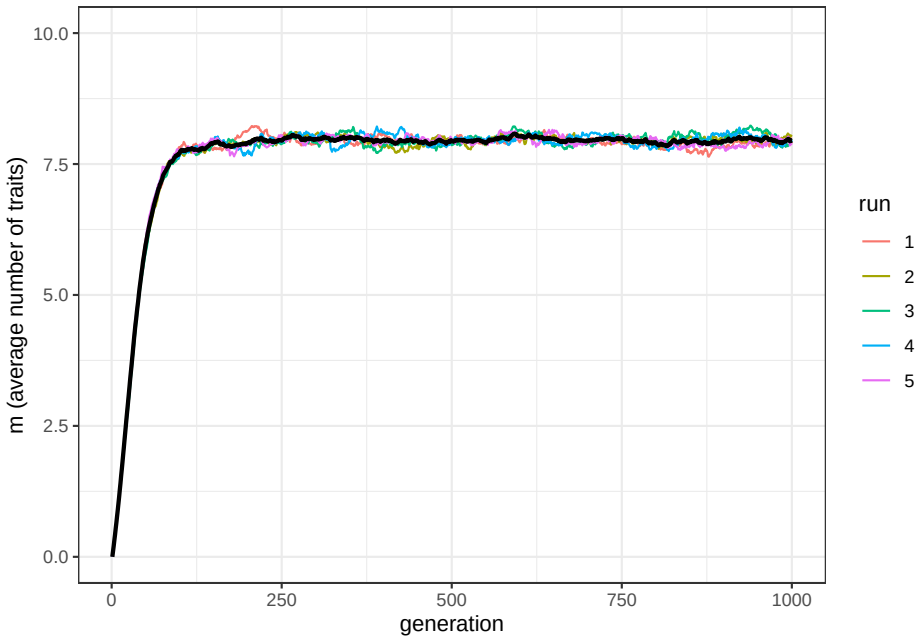


Figure 11.5: Individuals, on average, do not acquire all the possible traits during the lifetime.

On average, individuals do not have all 10 possible traits. Remember that individuals are replaced with a birth/death process, and they are born with no cultural traits so that they need time to acquire them. Let's try now with a

bigger possible cultural repertoire, say $M = 50$, and plot the average openness as well as the average number of traits.

```
data_model <- openness_conservatism_2(N = 1000, M = 50, mu = 0.1, p_death = 0.01, t_max
plot_multiple_runs_p(data_model)
```

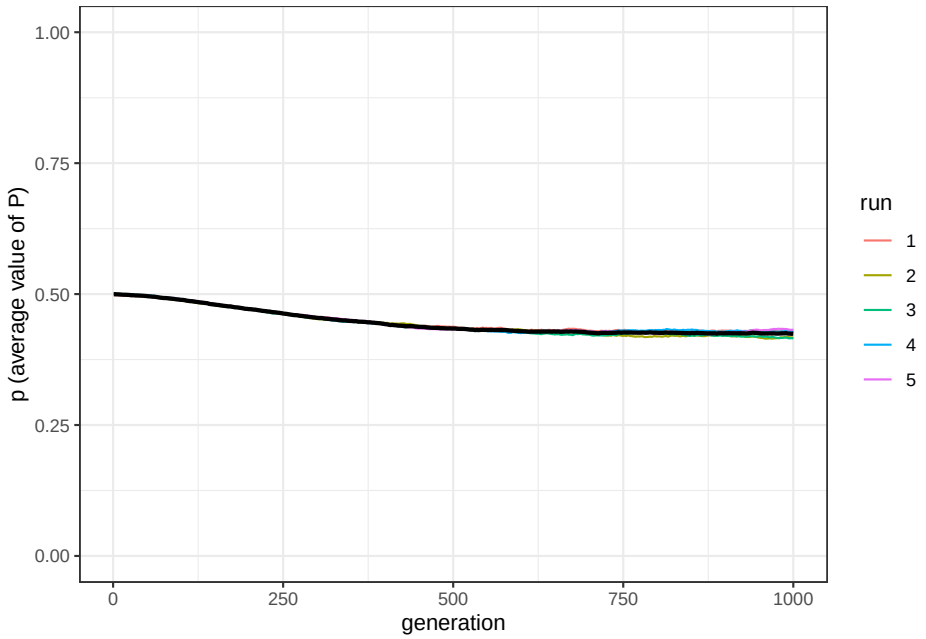


Figure 11.6: Individuals, on average, acquire less then half of the available traits, when there are 50 possible traits.

```
plot_multiple_runs_m(data_model, M = 50)
```

This time the average openness stabilises to an even higher value (around 0.4), and the number of cultural traits is below 20, lower than half of all possible traits.

We can explicitly visualise the relationship between M and population openness after 1000 generations for few representative values of M. We consider only a single run for each condition as, from the previous results, we know that different runs give very similar results.

```
test_openness <- tibble(M = c(1,5,10,20,50,100), p = as.numeric(rep(NA, 6)))
for(condition in test_openness$M){
  data_model <- openness_conservatism_2(N = 1000, M = condition, mu = 0.1,
                                        p_death = 0.01, t_max = 1000, r_max = 1)
  test_openness[test_openness$M == condition, ]$p <-
    data_model[data_model$generation == 1000, ]$p
```

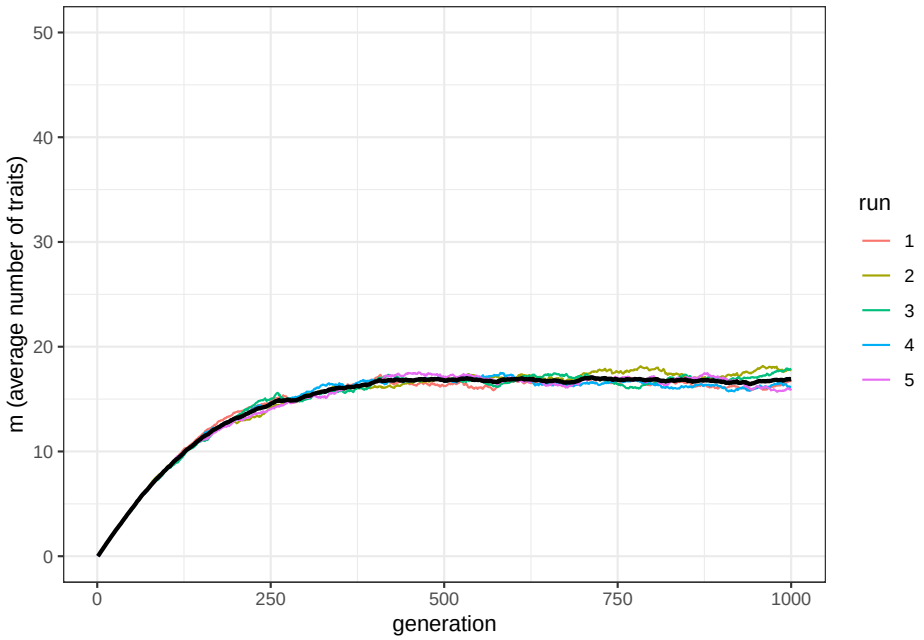


Figure 11.7: Individuals, on average, acquire less then half of the available traits, when there are 50 possible traits.

```
}
ggplot(data = test_openness, aes(x = M, y = p)) +
  geom_line(linetype = "dashed") +
  geom_point() +
  theme_bw() +
  labs(x = "Maximum possible number of traits", y = "p (final average value of p)")
```

The more cultural traits that are possible to acquire, the more individuals remain open. Why is that the case? As we saw before, a conservative individual will be able to spread its traits because they are more stable (remember the red t-shirt example). On the other hand, to be copied, an individual needs to showcase its traits. As the traits are chosen at random, it is better for an individual - from the point of view of its cultural success - to have many traits. These two requirements are in conflict: to acquire many traits an individual needs to remain relatively open. For this reason, when the cultural repertoire is big, individuals will remain open longer.

You can easily check by yourself that decreasing p_{death} has a similar effect of decreasing M. Individual living longer will generate more conservative populations. With a bit of work to the code, the same effect can be produced if individuals can learn faster. You can add a parameter to the model that tells

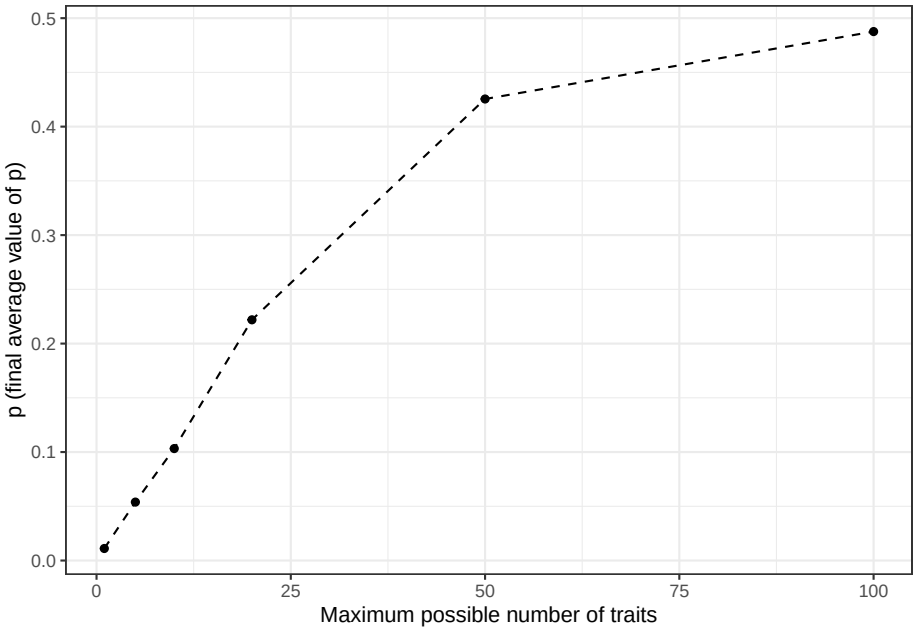


Figure 11.8: Relationhsip between the number of possible cultural traits and the average openness of the population: when there are more traits possible to acquire, populations remain more open.

how many traits an observer copies from the demonstrator at each interaction (in the case above is as this parameter would have been fixed to 1). The more effective is cultural transmission, the more conservative the populations. All depends on whether individuals have time to to optimise *both* openness and conservatism: big repertories, short lifespans, and ineffective cultural transmission all maintain relatively open populations.

11.3 Summary of the model

In this chapter we explored the idea that we can learn from others not only beliefs and skills, but also the rules that govern how and when we learn from others. The models we presented just scratch the surface of what the consequences of the 'social learning of social learning rules' could be, and we invite the readers to explore other possibilities. The models still provide some interesting insights: successful cultural models need to integrate openness (to acquire cultural traits liked by others) and, at the same time, conservativeness (to remain stable and repeatedly show the same traits to copy). This also suggests that successful cultural traits should not only be liked by many, but they also should promote conservativeness, as we defined it here, in their bearers. After all, the first commandment in the Abrahamic religions is 'Thou shalt have no other gods before me' rather then 'Check the other gods, and you'll see I am the better one;! Regardless of the particular results, however, these models mostly highlight how unexpected cultural dynamics can emerge from systems in which the rules governing social learning are not fixed, but they are themselves subject of cultural evolution.

11.4 Further readings

The models above are based on the models described in Ghirlanda et al. [2006] and Acerbi et al. [2009]. Acerbi et al. [2009] investigates possible variants of the main model of this chapter, such as continuous traits, innovations possible for preferences too, and various degrees of effectiveness of cultural transmission. It also explores how the basic dynamics affects individual characteristics (young individuals are more open than older individuals, older individuals are more effective cultural models than younger individuals, and so on). Acerbi et al. [2014] summarises these models, and provides a more general perspective on the 'social learning of social learning rules' topics, including other simulated scenarios. Mesoudi et al. [2016] is a review of the individual and cultural variation in social learning, pointing to various references, including empirical evidences of cultural variation in social learning in humans.

Chapter 12

Traits inter-dependence

In real life, the relationship a cultural trait has with other, coexisting, cultural traits, is important to determine its success. Nails will enjoy much cultural success in a world where hammers are present, and less in a world where they are not. Being against abortion in contemporary US is strongly correlated to being religious, which, in turn, is (less strongly) correlated with not supporting same-sex marriage. Of course, not all these relationships are stable in time, and they can also be themselves subject to cultural change. In this chapter, we will explore how simple relationships between traits can be modelled, and how they can influence cultural evolution.

12.1 Compatible and incompatible traits

We can start by assuming that, when an observer meets a demonstrator, the observer evaluates the relationships of the traits of the demonstrator with its own traits, and use this information to decide whether to copy or not. For example, if the observer has the trait 'being religious' and the demonstrator the trait 'being pro abortion', copying will be less likely to happen than if the demonstrator has the trait "being against abortion".

We can imagine a simple scenario when there are only two possible relationships between two traits: they are compatible, meaning that the presence of one trait will reinforce the presence of the other, or incompatible, meaning the opposite. In addition, the relationship is symmetric: if trait A favours trait B, and conversely if trait B favours trait A (the same holds for the case of incompatibility). Finally, we assume that each trait is compatible with itself, simply meaning that, if both the observer and the demonstrator have trait A the probability to copy trait B will increase.

In a simple world with only four traits, we can represent trait relationships with a symmetric matrix, as the one below, where $+1$ denotes compatibility, and -1

denotes incompatibility.

Traits	A	B	C	D
A	+1	+1	-1	-1
B	+1	+1	-1	-1
C	-1	-1	+1	+1
D	-1	-1	+1	+1

In this case, traits A and B are compatible with each other but incompatible with C and D. The same is true for C and D, which are compatible with each other but incompatible with A and B.

We can construct this matrix in R, by indicating one-by-one the values we want to fill in, and the number of rows and columns the matrix needs to have. As usual, we can check the resulting matrix by writing its name and hitting the return key.

```
library(tidyverse)
my_world <- matrix(c(1,1,-1,-1,1,1,-1,-1,-1,-1,1,1,-1,-1,1,1), nrow = 4, ncol = 4)
my_world
```

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

Given this simple way of representing a world of compatibilities among traits, we can write our model.

```
traits_inter_dependence <- function(N, t_max, k, mu, p_death, world){
  output <- tibble(trait = rep(c("A","B","C","D"), each = t_max),
                   generation = rep(1:t_max, 4),
                   p = as.numeric(rep(NA, t_max * 4)))

  population <- matrix(0, ncol = 4, nrow = N)

  output[output$generation == 1 ,]$p <- colSums(population) / N

  for(t in 2:t_max){

    # Innovations
    innovators <- sample(c(TRUE, FALSE), N, prob = c(mu, 1 - mu), replace = TRUE)
    innovations <- sample(1:4, sum(innovators), replace = TRUE)
    population[cbind(which(innovators == TRUE), innovations)] <- 1
```

```
    # Copying
    demonstrators <- sample(1:N, replace = TRUE)
    demonstrators_traits <- sample(1:4, N, replace = TRUE)

    for(i in 1:N){
      if(population[demonstrators[i], demonstrators_traits[i]]){
        compatibility_score <- sum(world[demonstrators_traits[i], population[i, ] != 0])
        copy <- (1 / (1 + exp(-k*compatibility_score))) > runif(1)
        population[i,demonstrators_traits[i]] <- 1 * copy
      }
    }

    # Birth/death
    replace <- sample(c(TRUE, FALSE), N, prob = c(p_death, 1 - p_death), replace = TRUE)
    population[replace, ] <- 0

    # Output
    output[output$generation == t ,]$p <- colSums(population) / N
  }
  # Export data from function
  output
}
```

As in the previous chapter, the simulation starts with no traits, and individuals introduce them with random innovations, the rate of which is regulated by the parameter μ. Individuals are replaced by culturally-naive newborns with a probability p_{death}. There are two major differences from the previous model. One is that the function accepts a parameter, called `world`, a four-by-four matrix of compatibilities between traits (thus the compatibilities can change, but not the actual number of traits). The second is in the copying procedure.

As in the previous chapter, one trait is randomly selected to be observed from a demonstrator and, if the demonstrator i has it (`population[demonstrators[i], demonstrators_traits[i]]`), we calculate the 'compatibility score'. The compatibility score is the sum of the compatibilities of all the traits of the observer towards the traits of the demonstrator, using the `world` matrix. If, for example, both observer and demonstrator have A and B (and only A and B), the compatibility would be 2, if the observer has A and B and the demonstrator C and D, the compatibility would be -2 and so on. In the next line, the compatibility score is transformed in the actual probability to copy with a logistic function. This is a useful trick to transform possibly unbounded positive and negative values to be between 0 and 1:

$$P_{\text{copy}} = \frac{1}{1 + e^{-kC}} \tag{15.1}$$

where C represents the compatibility score between observer and demonstrator,

and k is a parameter of the simulation, that controls the steepness of the logistic curve, i.e. how fast positive values of the compatibility score produce a probability to copy equal to 1, and negative values a probability equal to 0.

We can now run the function, using the `plot_multiple_traits()` function to plot the result. We use a value of $k = 10$, and a small probability of innovation $\mu = 0.0005$, so that the dynamics are mainly generated by cultural transmission.

```
my_world <- matrix(c(1,1,-1,-1,1,1,-1,-1,-1,-1,1,1,-1,-1,1,1), nrow = 4, ncol = 4)
data_model <- traits_inter_dependence(N = 100, t_max = 1000, k = 10,
                                      mu = 0.0005, p_death = 0.01, world = my_world)
plot_multiple_traits(data_model)
```

Figure 12.1: Frequency of traits in a world with four traits and pairwise compatibilities.

In the great majority of the runs, two out of the four traits diffuse in the population. We can check whether these are in fact one of the couple of compatible traits, having a look at the last line of the output produced by the simulation.

```
data_model[data_model$generation==1000, ]
```

```
## # A tibble: 4 x 3
##   trait generation     p
##   <chr>      <int> <dbl>
## 1 A           1000  0.95
## 2 B           1000  0.96
```

```
## 3 C            1000  0.01
## 4 D            1000  0.01
```

Depending on random factors, the successful traits will be A and B or C and D. If you run the simulation again and again you would see that in around half of the simulations A and B are the successful traits, and in another half C and D are, and very few other possible cases with this "world" of compatibilities.

What happens if we change the world? We can run a new simulation where the traits A, B, and C are all compatible, but not D (remember, you can visualise the matrix of compatibilities by typing its name to be sure to have entered the compatibilities correctly).

```
my_world <- matrix(c(1,1,1,-1, 1,1,1,-1,1,1,1,-1,-1,-1,-1,1), nrow = 4, ncol = 4)
data_model <- traits_inter_dependence(N = 100, t_max = 1000, k = 10,
                                      mu = 0.0005, p_death = 0.01, world = my_world)
plot_multiple_traits(data_model)
```

Figure 12.2: Frequency of traits in a world with four traits and three traits compatibile with each other, but not compatible with the fourth.

As expected, now three traits have high frequencies in the population, while one is unsuccessful. As before, you can check that the unsuccessful trait is actually D by inspecting manually the last line of the output.

Not surprisingly, if all traits are compatible, they all spread equally in the population.

```
my_world <- matrix(c(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1), nrow = 4, ncol = 4)
data_model <- traits_inter_dependence(N = 100, t_max = 1000, k = 10,
                                      mu = 0.0005, p_death = 0.01, world = my_world)
plot_multiple_traits(data_model)
```

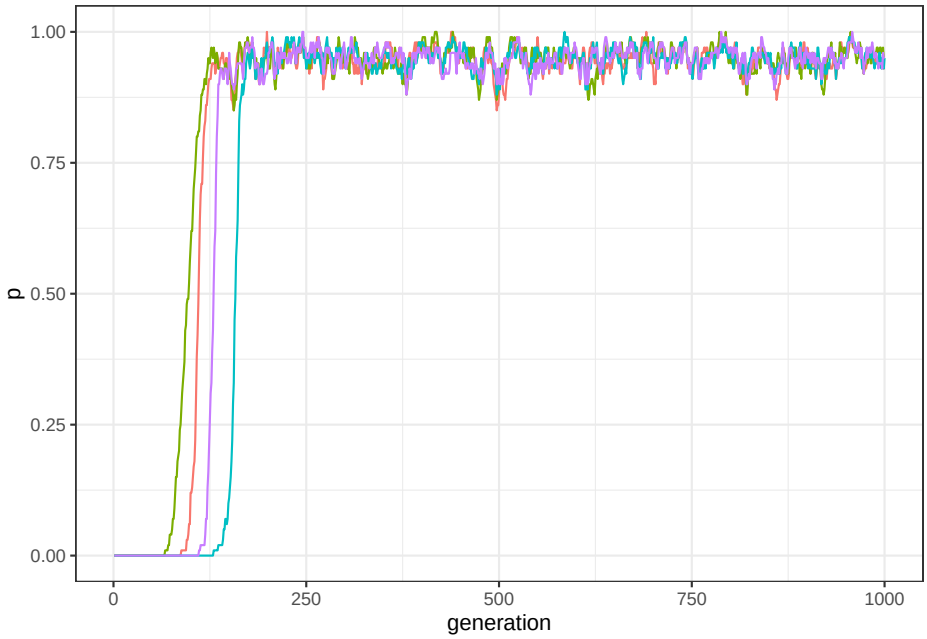


Figure 12.3: Frequency of traits in a world with four traits all compatibles with each other.

12.2 Many-traits model

Building the compatibility matrix by hand is not very practical, especially if we want to test our model with more traits. We will now extend the basic model above in order to be able to customise the maximum number of traits and to automatically generate the compatibility worlds.

```
M <- 7
gamma <- 0.5
my_world <- matrix( rep(1, M * M), nrow = M)
compatibilities <- sample(c(1, -1), choose(M, 2), prob = c(gamma, 1 - gamma), replace =
my_world[upper.tri(my_world)] <- compatibilities
my_world <- t(my_world)
my_world[upper.tri(my_world)] <- compatibilities
```

We have now two parameters we use to build the matrix: the maximum number

of traits M, and the probability that two traits are compatible with each other, γ (or `gamma` in the code). To build the matrix, we create a M by M matrix filled with 1s, then a vector of compatibilities randomly generated with probability γ (the length of the vector is the number of entries above the main diagonal of the matrix, given by `choose(M, 2)`), and finally we copy the values the lower triangle to the upper triangle of the matrix (in practice, to make it symmetric, we copy it twice in the upper triangle, transposing the matrix after the first copy).

Have a look at the the matrix we just generated.

```
my_world
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7]
## [1,]    1   -1   -1    1   -1    1   -1
## [2,]   -1    1   -1    1    1   -1   -1
## [3,]   -1   -1    1   -1   -1   -1   -1
## [4,]    1    1   -1    1    1    1   -1
## [5,]   -1    1   -1    1    1    1    1
## [6,]    1   -1   -1    1    1    1   -1
## [7,]   -1   -1   -1   -1    1   -1    1
```

The function to run the simulation is very similar to the previous one, once we account for the difference in how the compatibility matrix is created and for the two new parameters (M and γ) needed in the function call. Another difference is that the output data structure is a matrix and not a tibble. Since we want to be able to run the simulations with an arbitrary large number of traits, we need to speed up the computation, exactly in the same way as we did for the multiple traits model in chapter 7.

```
traits_inter_dependence_2 <- function(N, M, t_max, k, mu, p_death, gamma){
  output <- matrix(data = NA, nrow = t_max, ncol = M)

  # Initialise the traits' world
  world <- matrix( rep(1, M * M), nrow = M)
  compatibilities <- sample(c(1, -1), choose(M,2), prob = c(gamma, 1 - gamma), replace = TRUE)
  world[upper.tri(world)] <- compatibilities
  world <- t(world)
  world[upper.tri(world)] <- compatibilities

  # Initialise the population
  population <- matrix(0, ncol = M, nrow = N)
  output[1, ] <- colSums(population) / N

  for(t in 2:t_max){
    # Innovations
    innovators <- sample(c(TRUE, FALSE), N, prob = c(mu, 1 - mu), replace = TRUE)
    innovations <- sample(1:M, sum(innovators), replace = TRUE)
```

```
    population[cbind(which(innovators == TRUE), innovations)] <- 1

    # Copying
    demonstrators <- sample(1:N, replace = TRUE)
    demonstrators_traits <- sample(1:M, N, replace = TRUE)

    for(i in 1:N){
      if(population[demonstrators[i], demonstrators_traits[i]]){
        compatibility_score <- sum(world[demonstrators_traits[i], which(population[i,]
        copy <- (1 / (1 + exp(-k*compatibility_score))) > runif(1)
        if(copy){
          population[i,demonstrators_traits[i]] <- 1
        }
      }
    }

    # Birth/death
    replace <- sample(c(TRUE, FALSE), N, prob = c(p_death, 1 - p_death), replace = TRU
    population[replace, ] <- 0

    # Write output
    output[t, ] <- colSums(population) / N
  }
  # Export data from function
  output
}
```

We can now use the `plot_multiple_traits_matrix()` function (see chapter 7) to visualise the model results. Let's have a look at what happens when we have 20 traits and an intermediate probability of compatibility.

```
data_model <- traits_inter_dependence_2(N = 100, M = 20, t_max = 2000, k = 10,
                                        mu = 0.001, p_death = 0.01, gamma = .5)
plot_multiple_traits_matrix(data_model)
```

The simulation generates a complex dynamic in which some of the traits spread in the population, while others do not. The success of a trait depends on its general compatibility with other traits but also on which traits are present at a certain point in time in the population. Some traits succeed to spread only after the traits they are compatible with have sufficiently spread in the population.

Let's change γ to be 1, i.e. when all traits are compatible with each other.

```
data_model <- traits_inter_dependence_2(N = 100, M = 20, t_max = 2000, k = 10,
                                        mu = 0.001, p_death = 0.01, gamma = 1)
plot_multiple_traits_matrix(data_model)
```

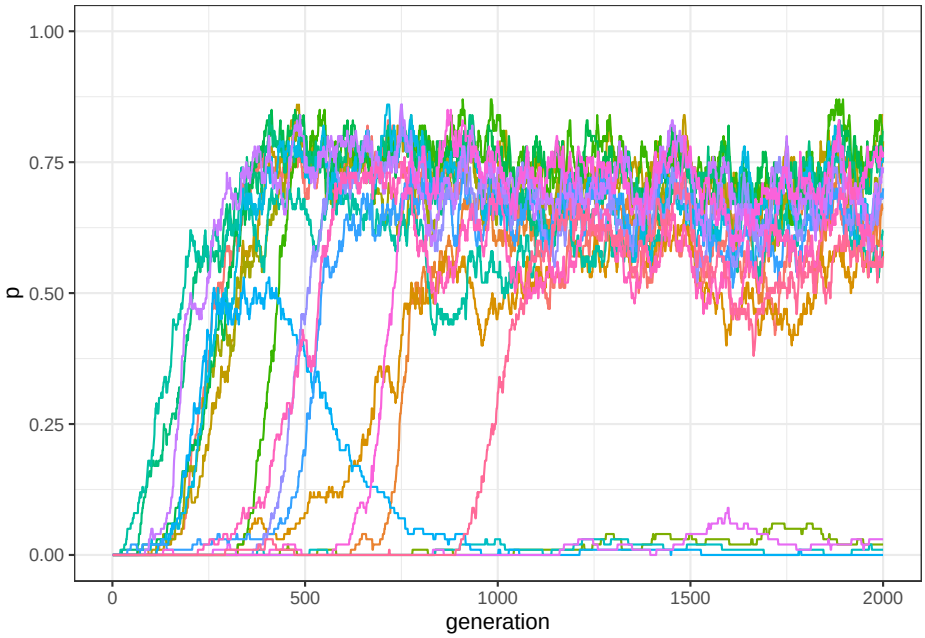


Figure 12.4: Frequency of traits in a world with 20 traits and compatiblity randomly generated. Each trait has 50% of probability of being compatible with each other trait.

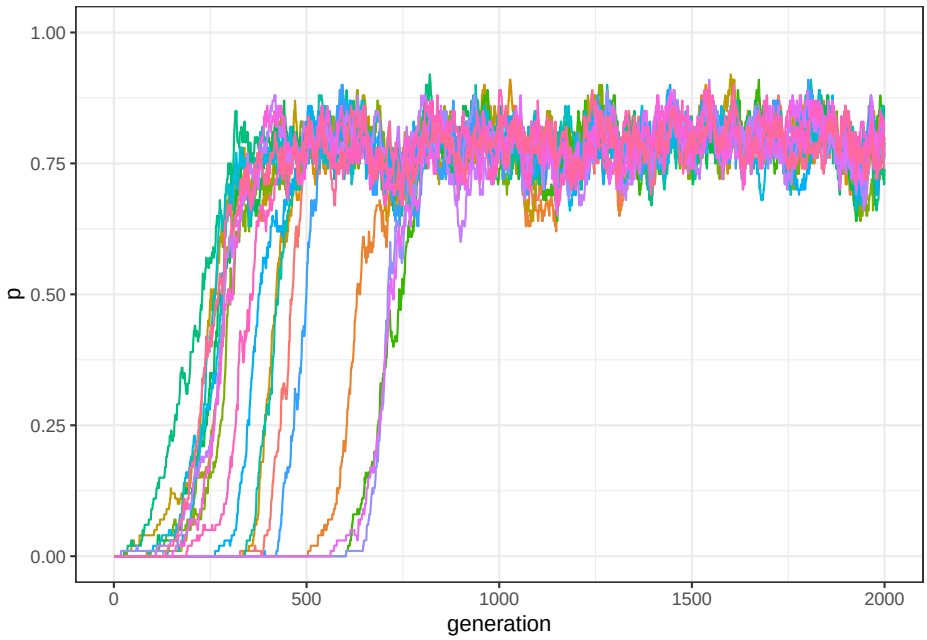


Figure 12.5: Frequency of traits in a world with 20 traits and compatiblity randomly generated. Each trait is compatible with all other traits.

As expected, all traits spread in the population.

We leave to the reader to explore further what drives the dynamic, especially in the interesting cases with intermediate values of γ. In order to do this, we suggest to add further outputs to the function `traits_inter_dependence_2()`, such as the compatibility world generated by the simulation, or the actual composition of the population, perhaps only at the end, or every 100, or 500, time steps.

As an example, we can try to confirm the intuition that the final number of traits depends on the value of γ: if it is more likely that traits are compatible with each other, we expect more traits to spread in the population. We can choose values of γ between 0 and 1 (with steps of 0.1), and use a `for` cycle to run our function for each value. In fact, we are running another `for` cycle within the main one, as to have more runs for each value of γ (an alternative would be to rewrite the function `traits_inter_dependence_2()` to accept an additional argument that indicates the number of simulation repetitions, as we did in previous chapters).

Finally, we store the number of "successful" traits at the end of the simulation, that is, the traits that spread in at least half of the population (`data_model[2000,]>.5`).

```
r_max = 10
test_inter_dependence <- tibble(gamma = as.factor(rep(seq(0, 1, by = .1), r_max)),
                                run = as.factor(rep(1:r_max, each = 11)),
                                C = as.numeric(NA))
for(condition in seq(0, 1, by = .1)){
  for(r in 1:r_max) {
    data_model <- traits_inter_dependence_2(N = 100, M = 20, t_max = 2000, k = 10,
                                            mu = 0.001, p_death = 0.01, gamma = condition)
    test_inter_dependence[test_inter_dependence$gamma == condition &
                          test_inter_dependence$run == r, ]$C <-
      sum(data_model[2000,]>.5)
  }
}
```

To plot the results, we combine boxplots and `geom_jitter()` so we can see the actual data points. (We introduced `geom_jitter()` in chapter 5.)

```
ggplot(data = test_inter_dependence, aes(x = gamma, y = C)) +
  geom_boxplot() +
  geom_jitter(width = 0.1, height = 0, alpha = 0.5) +
  theme_bw() +
  labs(x = "Average compatibility", y = "C (number of common traits)")
```

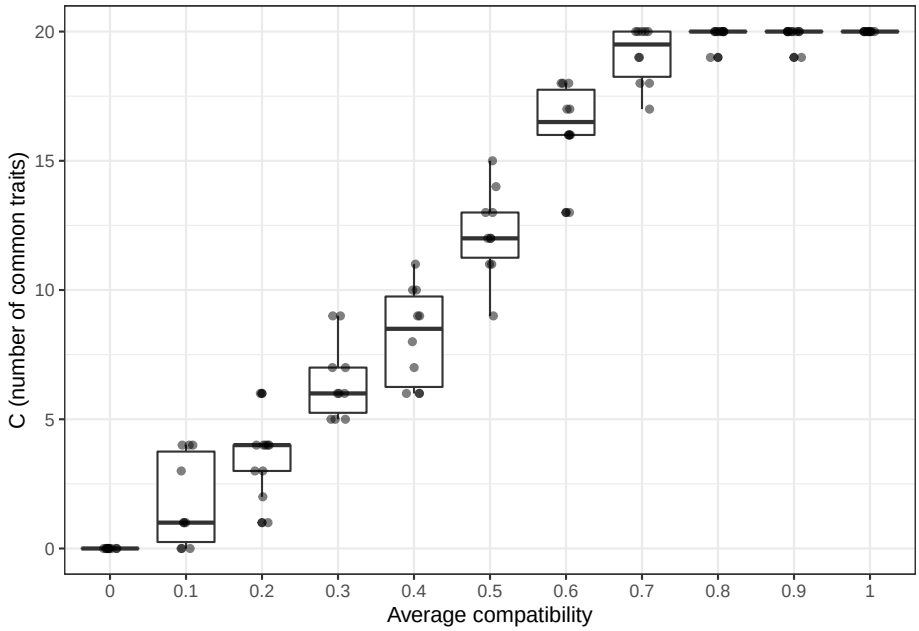


Figure 12.6: Increasing the probability of traits being compatible with each others produces bigger populations.

The results broadly confirm our intuitions. Interestingly, we do not need that all traits are compatible with each other to produce the outcome of all traits being successful. Already with $\gamma = 0.8$, all the 20 traits spread in more than half of the population in almost all runs of the simulation. Even with few incompatibilities, the "compatibility score" between observers and demonstrators is, given a sufficient number of compatible traits, a positive number, resulting in high probabilities to copy.

12.3 Summary of the model

Using simple models, we formalised the intuitive idea that cultural traits (can) have meaningful (for us) relationships with each other. When we decide whether to support a policy or not, to adopt a behaviour or not, or to participate in the latest fad, the decision may depend on how well the policy, the behaviour, or the fad fit with our pre-existing ideas. We used a simple rule for which traits can be compatible or incompatible with the other traits, and we showed that the success of traits depend on their compatibility. We also showed that, quite intuitively, populations where many traits are compatible with each other, will generate bigger cultures.

We also introduced a few new modelling devices, such as the logistic function, that transform unbounded positive and negative values into probabilities ranging from 0 to 1, and the ggplot geom `geom_jitter()`, which is useful to visualise overlapping data points.

12.4 Further readings

A broader treatment of models of cultural evolution where the outcomes depend on the relationships between traits, defined by the authors 'cultural systems', is in Buskell et al. [2019]. More complex relationships between traits, including the possibility that some trait will facilitate the appearance of new traits, generating cascade of innovations are explored in Enquist et al. [2011].

Advanced topics - Culture and populations

Chapter 13

Demography

In the previous chapters, we have looked at the transmission of information between individuals. We have seen that relatively simple mechanisms at the individual level can lead to population-level outcomes (e.g. the fixation of a rare cultural trait). We have also seen the importance of the characteristics of individuals (e.g. for success and prestige bias) in cultural processes. What we have not yet looked at is how the characteristics of the population may affect the outcome of cultural dynamics. In the following three chapters we will have a closer look at how population size (demography, this chapter), population structure (social networks, Chapter 14), and group structured populations with migration (Chapter 15) can influence cultural evolution.

Why would demography matter to cultural evolution? As long as information is transmitted among individuals and between generations, the size of the population should not play a role. In theory, this statement is true but it relies on a crucial assumption: information transfer is not only complete (all information from the previous generation is transmitted to the next generation) but also error-free. However, from many lab experiments, we know that copying information is an error-prone process. In this chapter, we will look at how those errors affect information accumulation and how population size is augmenting this process.

Several studies have looked at population effects. A well-known study is that by Joseph Henrich [2004]. His model takes inspiration from the archaeological record of Tasmania, which shows a deterioration of some cultural traits and the persistence of others after Tasmania was cut-off from Australia at the end of the last ice age. Henrich develops a compelling analytical model to show that the same adaptive processes in cultural evolution can result in the improvement and retention of simple skills, but also the deterioration and even loss of complex skills. In the following section, we will take a closer look at this model.

13.1 The Tasmania Case

The main idea of Henrich's model is the following: information transmission from one generation to another (or from one individual to another, here it does not make a difference) has a random component (error rate) that will lead to most individuals failing to achieve the same skill level (denoted with z) as their cultural model, whereas a few will match and - even fewer - exceed that skill level. Imagine a group of students who try to acquire the skills to manufacture a spear. As imitation is imperfect, and memorizing and recalling action sequences is error-prone, some students will end up with a spear that is inferior to the one of their cultural model. A few individuals might achieve a similar or even higher skill level than their cultural model.

To simulate imperfect imitation, Henrich's model uses random values from a Gumbel distribution. This distribution is commonly used to model the distribution of extreme values. Its shape is controlled by two parameters: μ (location) and σ (scale, sometimes also denoted as β). Varying μ affects how tricky it is to acquire a given skill. If we subtract an amount α from μ we move the distribution to the left, and so fewer individuals will acquire a skill level that is larger than that of the cultural model. The larger α the harder it is to acquire a given skill. Varying σ on the other hand affects the width of the distribution, and so whether imitators make very similar or systematic mistakes (small σ, narrow distribution) or whether errors are very different from each other (large σ, wide distribution). By using different values for α and σ, we can simulate different skill complexity and imperfect imitation. Intuitively, whether the average skill level of a population increases, persists, or decreases depends on how likely it is that some imitators will achieve a skill that exceeds the current cultural model. An illustration of Gumbel distributions for a complex and a simple skill is provided in the figure below.

Additional to the skill complexity, this also depends on how many individuals try to imitate the skill (how many values are drawn from the distribution). The smaller the pool of imitators, the fewer individuals will achieve a higher skill level and so, over time the skill level will decrease. Henrich provides an analytical model to explain how societies below a critical size (of cultural learners) might lose complex (or even simple) cultural skills over time. We will attempt to re-create his results using an individual-based model.

13.2 Modelling the Tasmania Case

Our model looks like this: we simulate a population with N individuals. Each individual has a skill level z. In each round, we determine the highest skill level in the population, $z_{\max}$. We will then draw new values of z for each individual in the population. We draw these values from Gumbel distribution where the new mean is the same as the skill level of the most skilled individual minus α, i.e. $\mu = z_{\max} - \alpha$. To keep track of the simulation we will store the average

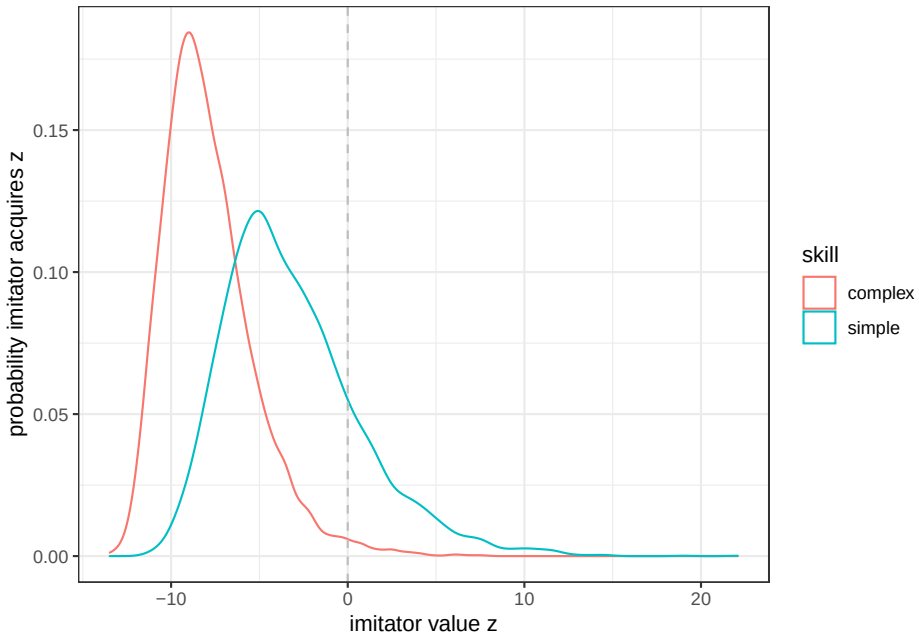


Figure 13.1: Shown are the probability distributions to acquire a specific skill level (z, x-axis) for two different skills (a simple one that is easy to learn, and a compelx one that is harder to learn). Given that learning is error-prone more individuals will acquire a skill level that is lower than that of a cultural model (its level is indicated by the vertical dashed line) through imitation (left of the dashed line). A few individauls will achieve higher skill levels (right of the dashed line). For the complex skill the probability to be above the skill level of the cultural model is lower (smaller area under the curve) than for simple skills.

proficiency $\bar{z}$ and the change in average proficiency $\Delta\bar{z}$.

We begin by loading the packages we will need. We will load the `extraDistr` package that gives us access to the `rgumbel()` function, which draws random values from a Gumbel distribution. We will have to define the shape of the distribution by providing two values, μ (location) and σ (scale).

Next, we set the some of the parameters that we need to run the simulation, that is, population size `N` and the number of simulation turns `t_max`. We also create some data structures to store the skill level `z` for each individual, and the reporting variables `z_bar` and `z_delta_bar` for average skill level and the change of the average skill level, respectively.

We also set the parameters for the Gumbel distribution, here $\sigma = 3$ and $\alpha = 5$.

Finally, we write down a very basic learning loop. The first step in this `for()` loop is to draw new values of `z` and store them in `z_new`. We then calculate the mean of the new skill levels and the change compared to the previous time step and finally update all values stored in `z`.

```
library(tidyverse)
library(extraDistr)

# Set population size
N <- 1000
# Set number of simulation rounds
t_max <- 5000
# Draw random values from a uniform distribution to initialise z
z <- rep(1, N)
# Set up variable to store average z
z_bar <- rep(NA, t_max)
# Set up variable to store change in average z
z_delta_bar <- rep(NA, t_max)

# Set parameters for Gumbel distribution
sigma <- 3
alpha <- 5

for(r in 1:t_max){
  # Calculate new z
  z_new <- rgumbel(n = N, mu = max(z) - alpha, sigma = sigma)
  # Record average skill level
  z_bar[r] <- mean(z_new)
  # Record average change in z
  z_delta_bar[r] <- mean(z_new - z)
  # Update z
  z <- z_new
}
```

Let us now plot the result of this simulation run. We first transform the output data structures in a tibble, so that can be conveniently be plotted with `ggplot`:

```
z_delta_bar_val <- tibble(x = 1:length(z_delta_bar), y = z_delta_bar)
ggplot(z_delta_bar_val) +
  geom_line(aes(x = x, y = y)) +
  xlab("time") +
  ylab("change in z") +
  geom_hline(yintercept = mean(z_delta_bar_val$y), col = "grey", linetype = 2) +
  theme_bw()
```

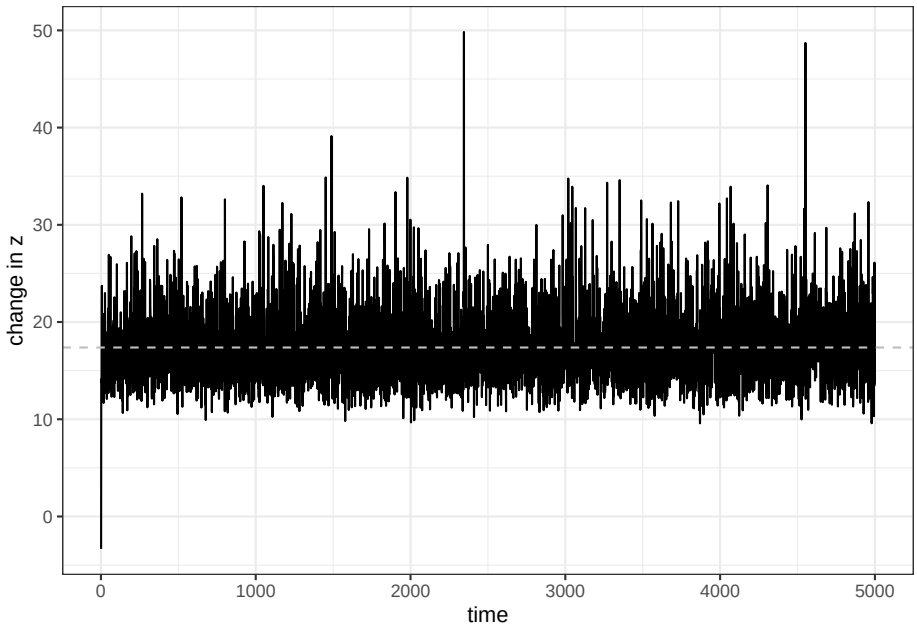


Figure 13.2: While $\bar{z}$ is sometimes above and sometimes below 0, it is on average postive (dashed line), which indicated that the average skill level of the population increases.

We find that $\Delta\bar{z}$ quickly plateaus at about 17.4 (grey dashed line). As this is > 0, on average the population will improve its skill over time. We can see that this is the case when we plot the average skill level over time:

```
z_bar_val <- tibble(x = 1:length(z_bar), y = z_bar)
ggplot(z_bar_val) +
  geom_line(aes(x = x, y = y)) +
  xlab("time") +
  ylab("average skill-level") +
  theme_bw()
```

As in the previous chapters, we can now write a wrapper function that allows us

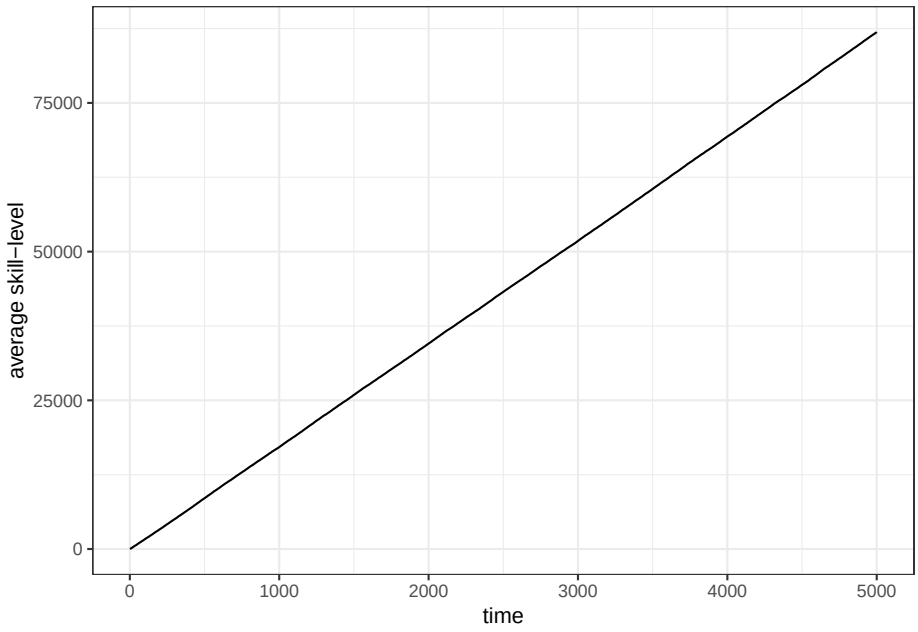


Figure 13.3: For the given parameter ($\alpha = 5$, $\sigma = 3$) the average skill-level increases continously.

to execute this model repeatedly and for different parameters. In the following, we will use a new function: `lapply()`. There is a series of apply functions in the R programming language that 'apply' a function to the elements of a given data object. Generally, these functions take an argument `X` (a vector, matrix, list, etc.) and then apply the function `FUN` to each element. We use `lapply` here on a vector `1:R_MAX`, that is, a vector of the length of the number of repetitions that we want. What will happen is that `lapply()` will execute the function that we will provide exactly `R_MAX` times, and then return the result of each calculation in a list at the end. We could also use a `for()` loop just as we have done it in the previous chapters. However, the advantage of using the apply function over the loop is that each simulation can run independently from each other. That is because the second simulation does not have to wait for the first to be finished. In contrast, we could not use the apply function for the individual turns. Here the second simulation step *does* rely on the results of the first step. In this case, all simulation steps have to be calculated in sequence.

Have a look at our `demography_model()` wrapper function:

```
demography_model <- function(T_MAX, N, ALPHA, SIGMA, R_MAX){
  res <- lapply(1:R_MAX, function(repetition){
                  z <- rep(1, N)
                  z_delta_bar <- rep(NA, T_MAX)
```

```
                  for(turn in 1:T_MAX){
                    z_new <- rgumbel(n = N, mu = max(z) - ALPHA, sigma = SIGMA)
                    z_delta_bar[turn] <- mean(z_new - z)
                    z <- z_new
                  }
                  return(mean(z_delta_bar))
                })
  mean(unlist(res))
}
```

We begin by initiating a function called `demography_model()` that is taking a set of parameters (note, it can be useful to capitalize arguments of a function to differentiate between those values that are calculated within a function (not capitalized) and those that have been provided with the function call). When we execute `demography_model()` it will first run an `lapply()` function for `R_MAX` number of rounds. The `lapply()` function will now run independent simulations which we have discussed above (i.e. setting up a population of individuals with skill level `z`, updating these values, and calculating the change in average skill level). The last step is to calculate the mean of `z_delta_bar`, i.e. the average of the change of the mean skill level. This is value is calculated for each repetition. `lapply()` returns all of these values in a list called `res`. As we are interested in the average change of the skill level across all repetitions, we first turn this list into a vector (using the `unlist()` function) and then calculate the mean.

Let us now use the `demography_model()` function to run repeated simulations for different population sizes and different skill complexity. Here, we use the following parameters for the skill complexities: $\alpha = 7, \sigma = 1$ (simple) and $\alpha = 9, \sigma = 1$ (complex).

We first define a variable, `sizes`, for the different population sizes. We are then again relying on the magic of the `lapply()` function. As above, the reason is that we can let simulations with different population sizes run independently from each other. Note that we provide `sizes` as our `X` argument, and `demography_model()` as the `FUN` function argument. Our `demography_model()` itself requires further arguments to run. In the `lapply()` function we can simply add them at the end. They will be directly handed over to `demography_model()` when we execute the `lapply()` function.

In the last line of this chunk, we create a tibble that will hold the final results of the simulations for each skill and the different population sizes.

```
sizes <- c(2, seq(from = 100, to = 6100, by = 500))

simple_skill <- lapply(X = sizes, FUN = demography_model,
                       T_MAX = 200, ALPHA = 7, SIGMA = 1, R_MAX = 20)

complex_skill <- lapply(X=sizes, FUN=demography_model,
```

```
                          T_MAX = 200, ALPHA = 9, SIGMA = 1, R_MAX = 20)

data <- tibble(N = rep(sizes, 2),
               z_delta_bar = c(unlist(simple_skill),
                               unlist(complex_skill)),
               skill = rep(c("simple","complex"), each = length(sizes)))
```

Let us now plot the results:

```
ggplot(data) +
  geom_line(aes(x = N, y = z_delta_bar, color = skill)) +
  xlab("effective population size") +
  ylab("change in average skill level, delta z bar") +
  geom_hline(yintercept = 0) +
  theme_bw()
```

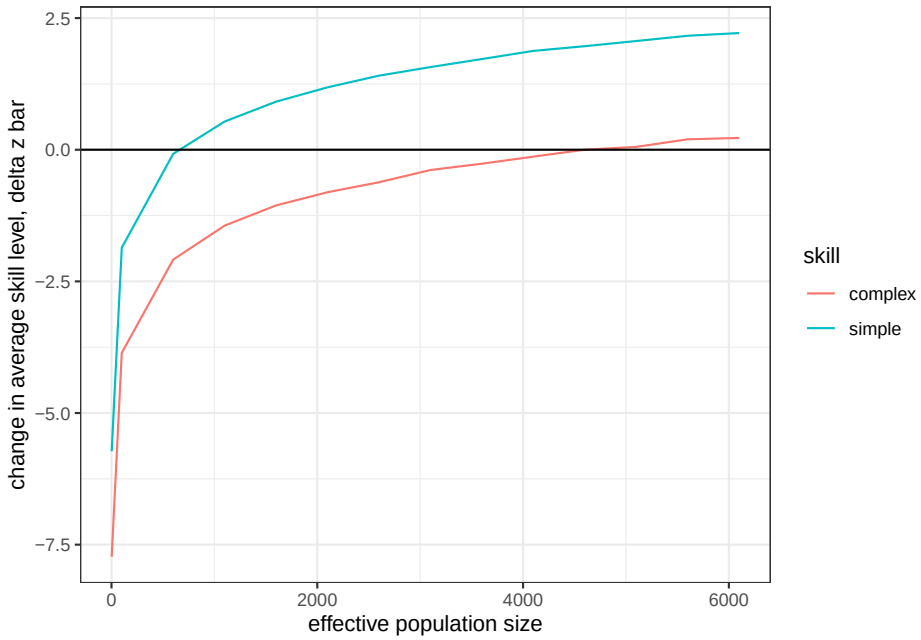


Figure 13.4: For a simple skill, effective populaton size (at which the skill can be just maintained in a population) is much smaller than the population that is required to maintain a complext skill.

In the figure above we can see that, for simple skills, the change in average skill level becomes positive (see where it intercepts the x-axis at 0) at much smaller population sizes than the complex skill. This means a simple skill can be maintained by much smaller populations, whereas larger populations of imitators are required for complex skills.

13.3 Calculating critical population sizes based on skill complexity

Henrich calls the minimum population size required to maintain a skill the critical population size, $N^\star$. How can we calculate $N^\star$ for different skill complexities? We could run simulations for many more population sizes and find the one where $\Delta\bar{z}$ is closest to zero. Alternatively, here is a more elegant and less computationally intensive method, that we will use below.

When we plot the previous results over logarithmic population size the resulting graphs are almost linear.

```
ggplot(data) +
  geom_line(aes(x = log(N), y = z_delta_bar, color=skill)) +
  xlab("log(effective population size)") +
  ylab("change in average skill level, delta z bar") +
  geom_hline(yintercept = 0) +
  theme_bw()
```

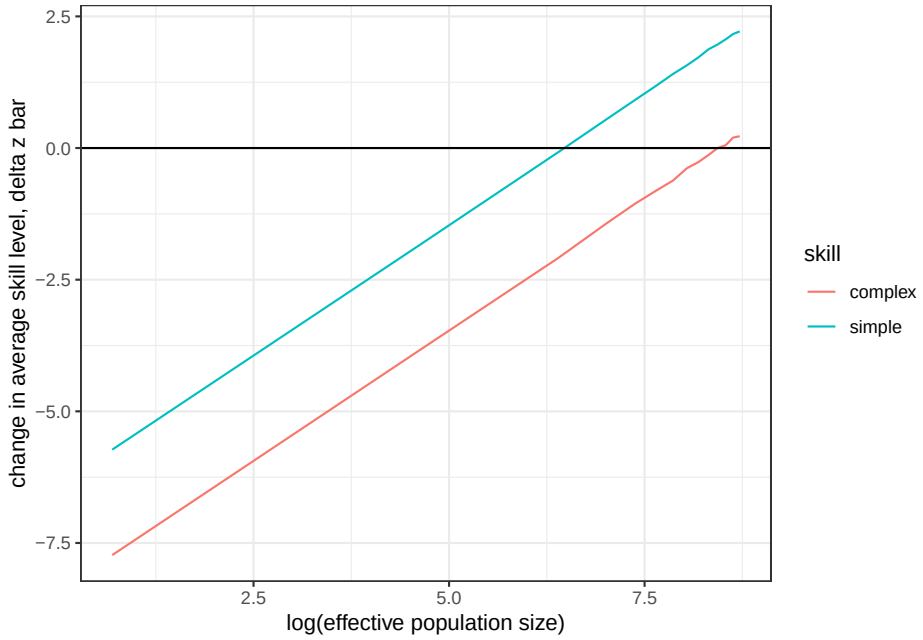


Figure 13.5: The same results as in Figure 13.4 but using log on population sizes.

Thus, we could use a linear fit and then solve for $y = 0$ to calculate $N^\star$. To do this we use the `lm()` function, which fits a linear model to the data that we provide. It takes a `formula` argument that identifies the 'response variable' (here `z_delta_bar`) and the 'predictor variable' (here `log(N)`). The two variables are

separated with a ~ sign. To calculate a fit just for the data points of the simple skill simulation, we only hand over that part of the `data` where the `skill` column contains the term `simple`. As usual, we can inspect the results simply writing the variable name where we saved the results of the function, i.e. `fit`.

```
# Create linear regression for the change in average skill level in response to popula
fit <- lm(formula = z_delta_bar ~ log(N),
          data = data[data$skill == "simple",])
fit
```

```
##
## Call:
## lm(formula = z_delta_bar ~ log(N), data = data[data$skill ==
##     "simple", ])
##
## Coefficients:
## (Intercept)       log(N)
##     -6.4257       0.9946
```

The result is a list of information from the linear regression. Here, we are interested in the intercept with the y-axis and the inclination of the linear regression, both of which are displayed under `Coefficients`. We can calculate the point at which our regression line crosses the x-axis using the linear function $y = mx + b$, setting $y = 0$ and then transforming, such that $x = -\frac{b}{m}$, where b is the y-axis intercept, and m is the inclination.

```
# Solve for y = 0 by using the coefficients of the linear regression:
b <- fit$coefficients[1]
m <- fit$coefficients[2]
N_star_simple <- exp(-(b / m))

# And the same calculation for the complex skill
fit <- lm(formula = z_delta_bar ~ log(N),
          data = data[data$skill == "complex",])
N_star_complex <- exp(-(fit$coefficients[1] / fit$coefficients[2]))
```

Note that we need to take the exponent (`exp()`) of the resulting value to revert the log function that we applied to the population size. We see that a simple skill with a low alpha to sigma ratio requires a minimum population size of about 639, whereas a much large population size is required to maintain a complex trait (about 4712). (You can visualise those results by writing `N_star_simple` and `N_star_complex`.) When you go back to Figure 13.4 you can see that these points correspond with the graphs of the simple and complex skill crossing the x-axis.

Let us now calculate the $N^\star$ values for different skill complexities and different population sizes. We first set up the parameter space using `expand.grid()`. This function essentially creates all possible combinations of input variables. In

our case, we want all possible combinations of the different population sizes `N` and skill complexities, which we will vary using different values for α. Therefore, executing this function will return a two column (for `N` and `alpha`) data structure, stored in `simulations`. We visualise the first lines with the function `head()`:

```
# Run simulation for the following population sizes
sizes <- seq(from = 100, to = 6100, by = 500)

# Run simulation for the following values of alpha
alphas <- seq(from = 4, to = 9, by = .5)

simulations <- expand.grid(N = sizes, alpha = alphas)
head(simulations)
```

```
##      N alpha
## 1  100     4
## 2  600     4
## 3 1100     4
## 4 1600     4
## 5 2100     4
## 6 2600     4
```

Now we can run simulations for all combinations of population sizes and skill compexities:

```
z_delta_bar <- lapply(X = 1:nrow(simulations), FUN = function(s){
                  demography_model(T_MAX = 200,
                                   N = simulations[s, "N"],
                                   ALPHA = simulations[s, "alpha"],
                                   SIGMA = 1,
                                   R_MAX = 5)
  })
# Add results to population size and skill complexity
data <- cbind(simulations, z_delta_bar=unlist(z_delta_bar))
```

Finally, let us fit a linear regression to each skill complexity to determine the according critical population size $N^\star$:

```
n_stars <- lapply(X = unique(data$alpha), FUN = function(alpha){
                    # Only use the results with identical value for alpha
                    subset <- data[data$alpha == alpha,]
                    # Fit regression
                    fit <- lm(formula = z_delta_bar ~ log(N), data = subset)
                    # Solve for n star
                    n_star <- exp(solve(coef(fit)[-1], -coef(fit)[1]))
                    return(n_star)
                  })
```

```
# Combine all results in a single tibble
results <- tibble(n_star = unlist(n_stars), alpha = unique(data$alpha))
```

Now, we plot the critical population size as a function of the skill complexity α over σ. Note, that the x-axis label contains Greek letters. There are at least two ways to get `ggplot` to display Greek letters. The most simple way is to type the unicode equivalents of the symbols and letters. In our case this would look like this: `xlab("\u03b1 / \u03C3")`. Alternatively, we can use the `expression()` function and type the names of the Greek letters (see below). This will be parsed and translated into the according letters:

```
ggplot(results, aes(x = alpha, y = n_star)) +
  geom_line() +
  xlab(expression(alpha/sigma)) +
  ylab("critical populaton size, N*") +
  theme_bw()
```

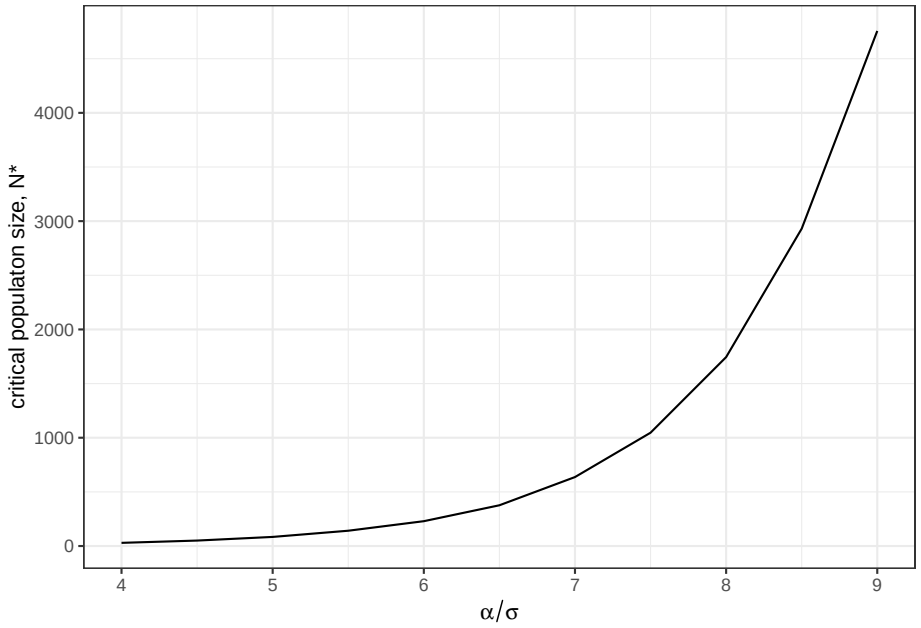


Figure 13.6: The critical population size, $N^\star$, increases exponentially as skill complexity increases.

It is interesting to observe that the critical population size increases exponentially with skill complexity. This also suggests that, all being equal, very high skill levels will never be reached by finite population sizes. However, different ways of learning (e.g. teaching) could considerably decrease α and σ over time and so allow high skill levels.

13.4 Summary of the model

Similar to the model in the chapter on Rogers' paradox, the present model is very simple and is making many simplifications. Nevertheless, it provides an intuitive understanding of how changes (up and down) in population size can affect the cultural repertoire of a population, and how it can be that simple skills thrive, while complex ones disappear. In the next chapter, we will discuss the importance of social networks, i.e. who can interact with whom. We will see that this will also have an effect (additional to the population size).

In this chapter we also introduced several new R functions and programming styles. Most important, we used `lapply()` instead of the usual `for` loops to run multiple runs of the simulations. We used a different notation for function parameters (all capital letters) to distinguish them from the same values that are calculated within a function). At this point of the book, as long oyu know what you are doing, you may want to experiment with different options and choose the ones that works better for you!

13.5 Further readings

Henrich [2004] provides a detailed analytical model of the simulation described in this chapter. Powell et al. [2009] is an extension to Henrich's model that incorporates sub-populations with varying density. A criticism of Henrich's Tasmanian model is Vaesen et al. [2016]. Shennan [2001] is another modelling paper that suggests that innovations are far more successful in larger compared to smaller populations. Ghirlanda et al. [2010] investigates the interplay between cultural innovations and cultural loss.

Shennan [2015] provides a good overview of a variety of approaches and questions in studies of population effects in cultural evolution.

Chapter 14

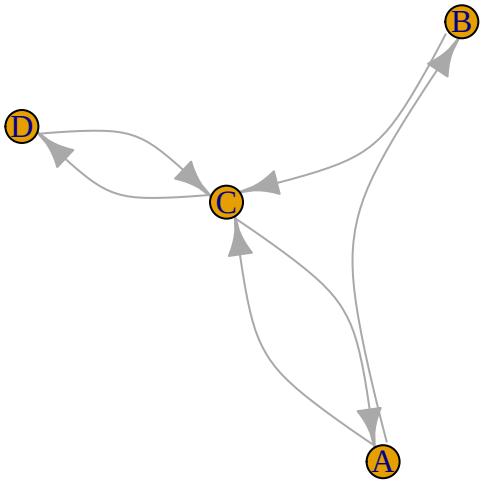
Social network structure

For mathematical tractability models often assume well-mixed populations, where any individual is equally likely to interact with any other individual. These models can provide good approximations of the real-world. Henrich's model from the previous chapter, for example, has shown that a population's ability to maintain and accumulate cultural traits depends on its size, whereby larger populations are more likely to retain and improve more complex cultural traits than smaller ones. This model provides useful insights into the role of population-level characteristics (here, demography) on cultural dynamics. However, cultural transmission is a social process that happens at the individual level, and for some questions, it is important to take an individual-level perspective. There is a growing number of studies showing that the structure of interactions (commonly represented as social networks) with other individuals can affect the population-wide transmission of behaviours or information. For example, a study on the spread of health behaviour has shown that information spreads faster and further in networks that are more clustered than in less clustered networks (Centola [2010]). To study the role of network characteristics (size, density, clustering, etc.) we need to be able to implement, modify, visualise, and analyse networks. We will cover all of these points in the first half of this chapter. In the second half, we will use these skills to study how social network structure affects cultural dynamics.

14.1 Network basics

Networks are a popular tool to visualise relationships between different actors. For example, co-authorship networks typically visualise which authors frequently publish articles together, ecological networks demonstrate trophic interactions between different species, and kinship networks show how individuals are related to each other. All networks are comprised of at least two components: 'nodes' (also referred to as vertex, *pl.* vertices), which represent actors,

and 'ties' (also referred to as edges) between any two nodes, which represent the existence of a relationship (e.g. co-authorship, kinship, cooperation, etc.).



The above figure shows a simple graph with four nodes (A-D), and their relationship edges (arrows). Note that some nodes have an incoming and an outgoing edge to another node (e.g. B and D) but some nodes have more incoming than outgoing edges (here C). This means that the relationship is not mutual, or reciprocal, like in the case of friendship or donations. In other cases, e.g. kinship ties, edges are always reciprocal. In this case, arrowheads are often omitted and a straight line is drawn between the nodes.

To work with networks in R, we need to find a way to represent nodes and edges. One option is to use an adjacency matrix (the other is an edge list, which we will not discuss here). An adjacency matrix is a square matrix where every possible relationship between any two actors (also called a dyad), is represented as a value (typically a 0 if there is no relationship, or a 1 if there is one). Generally, rows and columns represent the 'from' and 'to' nodes. As an example, assume we have N actors, then our adjacency matrix A will be of the size $N \times N$. If individual i and j have a reciprocal relationship, then $A_{i,j} = A_{ji} = 1$. However, if, say, i has donated to j at some point but j not to i, then we would write $A_{ij} = 1$ and $A_{ji} = 0$. Consequentially, if there is no relationship between the two then $A_{i,j} = A_{ji} = 0$. Let us translate this into R code.

We start by setting up a simple square matrix with a few zeros and ones. Let us also name the rows and columns with capital letters, using the `LETTERS` constant, built into R. These will be the names of the nodes.

```
m <- matrix(c(0,1,1,0, 0,0,1,0, 1,0,0,1, 0,0,1,0), nrow = 4, byrow = TRUE)
row.names(m) <- LETTERS[1:4]
colnames(m) <- LETTERS[1:4]
m
```

```
##   A B C D
```

```
## A 0 1 1 0
## B 0 0 1 0
## C 1 0 0 1
## D 0 0 1 0
```

Let's have a look at the resulting matrix `m`. It describes the relationships between four individuals: A, B, C, and D. When we look at the first row, we see that A has no interaction with itself (indicated by the zero), but interacts both with B and C (indicated by the ones). From the next row, we see that B is only interacting with C, and so forth. You might notice that this is the adjacency matrix that we used for the network graph above. This adjacency matrix is also called an asymmetric adjacency matrix because not all interactions are reciprocal. We can see this when we plot the network but we can also test using a bit of code.

Let's think, under which condition are all interactions reciprocal? The answer is when for all i and j it is true that $A_{ij} = A_{ji}$. In other words, if the entries of A's upper triangle are identical to the entries of its lower triangle. To return only the values of the upper triangle of our matrix, we can use the `upper.tri()` function. It uses the matrix itself as an argument to return a matrix of the same size with `TRUE` for entries that are part of the upper triangle, and `FALSE` for all other entries. We use this to select the correct values from our matrix. We can then compare these values with the upper triangle of the transposed version of the matrix (we transpose using the `t()` function). If `all()` entries are identical, we know that we have a symmetric matrix.

```
all(m[upper.tri(m)] == t(m)[upper.tri(m)])
```

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

As we have already determined, A is asymmetric. If you want a short cut to test whether a matrix is symmetric, you can also use the generic function `isSymmetric()`.

```
isSymmetric(m)
```

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

Sometimes, you might want to create a random network using a symmetric adjacency matrix. A simple way to create a symmetric adjacency matrix is the following: create a random square matrix and then copy the entries from the transposed upper triangle into the entries of the lower triangle. This ensures that $A_{ij} = A_{ji}$.

```
# Creating a random matrix
random_matrix <- matrix(sample(x = c(0,1), size = 100, replace = T), ncol = 10)
random_matrix
```

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

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

```
# It is not symmetric
isSymmetric(random_matrix)
```

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

```
# Replace the upper triangle with the transposed upper triangle
random_matrix[upper.tri(random_matrix)] <- t(random_matrix)[upper.tri(random_matrix)]
random_matrix
```

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

```
# Now it is symmetric
isSymmetric(random_matrix)
```

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

14.2 Generating networks

Networks can vary widely depending on what they represent, or what their purpose is if they are simulate from scratch. The figure below shows a few different types of networks that require different methods to create. A simple way to generate certain types of networks with, say, different number of nodes and edges, is to use one of the many functions provided by the `igraph` package. For example, the ring graph below can be generated with this simple command: `make_ring(n = 10)`.

For certain projects, these standard networks might not be sufficient, for example, if you have a certain mechanism in mind. Let us write our own network

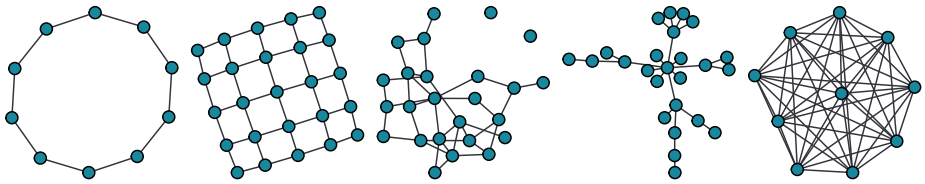


Figure 14.1: Different graph types (from left: circle, lattice, random, scale free, fully connected) vary in the density and distribution of edges between nodes.

generating function to suit our needs. As an example, let us write a function that creates a random network of N individuals. Let's assume that the edges we will add between nodes represent friendships, with a variable friendship proportion f (for $f = 1$ all individuals are friends, and for $f = 0$, sadly no one is friends with each other).

```
create_network <- function(N, f){
  # Set up an empty adjacency matrix of size NxN
  A <- matrix(0, ncol = N, nrow = N)
  # Set up a friendship counter
  friends <- 0

  # We will add friendships until we reach the desired number
  while(friends < round((((N^2) - N) / 2) * f)){
    dyad <- sample(x = N, size = 2, replace = FALSE)
    i <- dyad[1]
    j <- dyad[2]
    if(A[i, j] == 0){
      A[i, j] <- A[j, i] <- 1
      friends <- friends + 1
      }
  }
  return(A)
}
```

In our new function `create_network()` we first set up an empty adjacency matrix and a friendship counter. Then, we use a `while` loop to continuously select a random dyad in our group (i.e. individual i and j), test whether they are already friends. If this is not the case, we set $A_{ij} = A_{ji} = 1$ and increase the friendship counter. However, if they already are friends, we simply continue selecting another random dyad from our network. We stop when we reach $f(N^2 - N)/2$ friendships (the total number of possible connections is N^2, however, we assume that individuals cannot be friends with themselves and that friendships are reciprocal).

Our new function allows us to create a variety of different networks. One can generate, for example, a network with 10 individuals, that have 50% of proba-

bility of being friend of each other.

```
adjm <- create_network(N = 10, f = .5)
adjm
```

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

Let us now use additional functions from the `igraph` package to turn our adjacency matrices in networks and then to visualise them. This will give us a better intuition of what our groups look like when we change the two parameters.

14.3 Plotting networks

To visualise a network nodes and edges have to be placed in the correct relation to each other on an empty canvas. Because there is a lot of calculation going into this, we will use the plotting features that come with the `igraph` package. This package requires the network to be an `IGRAPH` object. So first, we will have to turn our adjacency matrix into a network. The function `graph_from_adjacency_matrix()` is going to do this for us:

```
library(igraph)
net <- graph_from_adjacency_matrix(adjm)
net
```

```
## IGRAPH 7610305 D--- 10 44 --
## + edges from 7610305:
##  [1]  1-> 4  1-> 5  1-> 6  1-> 8  1-> 9  1->10  2-> 4  2-> 9  3-> 6  3-> 9
## [11]  4-> 1  4-> 2  4-> 5  4-> 6  4-> 7  4-> 8  4-> 9  5-> 1  5-> 4  5-> 8
## [21]  5->10  6-> 1  6-> 3  6-> 4  6-> 8  6-> 9  7-> 4  7->10  8-> 1  8-> 4
## [31]  8-> 5  8-> 6  8-> 9  8->10  9-> 1  9-> 2  9-> 3  9-> 4  9-> 6  9-> 8
## [41] 10-> 1 10-> 5 10-> 7 10-> 8
```

If we enter the name of the network `net` we are provided with a lot of information, for example, that `net` is an `IGRAPH` object and that there are 10 vertices and 44 edges (for more information on reading this output take a look at this short igraph introduction). In addition to general information about the network, we receive a series of 'from to' pairs, e.g. `1->3`, indicating the edges between the different nodes. This is essentially an edge list, an alternative to our adjacency

matrix to describe relationships. Sometimes, it might be better to work with an edgelist, especially if there are many more nodes than edges. In this case, we would require a very large matrix that is mostly filled with zeros and only very few ones. You can use `get.edgelist(net)` to return the edgelist of your network.

Now that we have our network in the correct shape, we can use the plot function to visualise it:

```
plot(net)
```

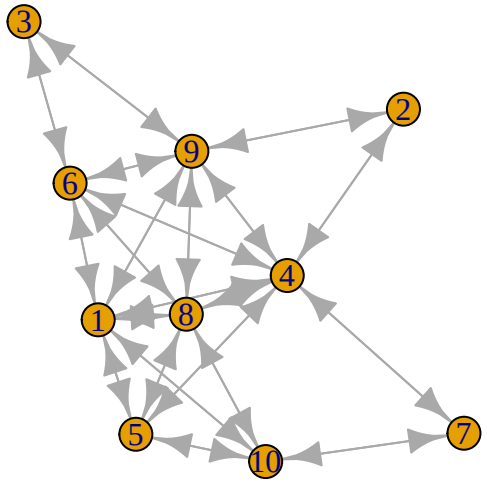


Figure 14.2: A simple network with 10 vertices, which are connected by edges (arrows).

This is the most basic network plot where each node (with the numbers 1 to 10) and their edges are plotted such that nodes that receive more connections are more central and those that receive less are more peripheral.

14.3.1 Network layout

There are also ways to plot networks in entirely different layouts. In the following graph, for example, we put nodes in a ring layout or on a grid. We use the different `layout.x()` functions in `igraph` in combination with the `layout=` argument of the plotting function. Notice we are not using the usual `ggplot` function to plot, as we make use of the specific functionalities of `igraph`.

```
# Set the plotting environment to allow 2 plots next to each other
par(mfrow = c(1,2))
plot(net, layout = layout.circle(net), main = "Ring layout")
plot(net, layout = layout.grid(net), main = "Grid layout")
```

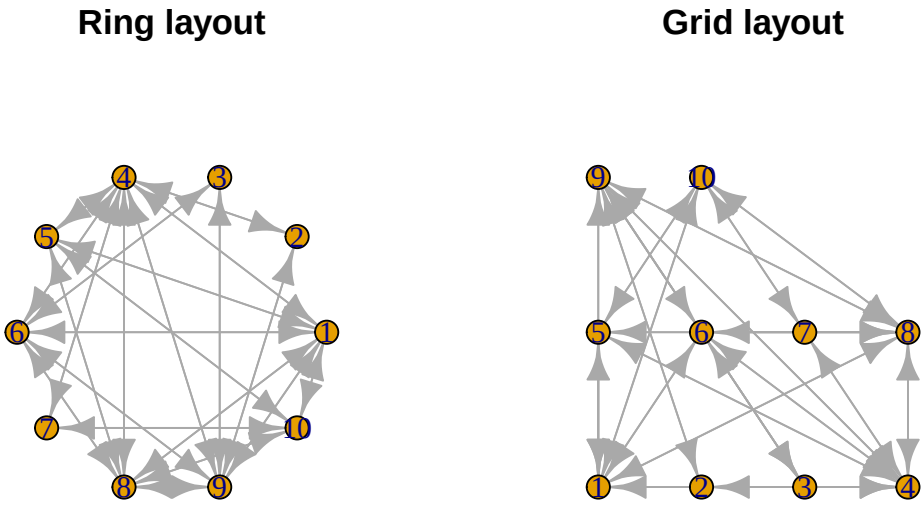


Figure 14.3: Example for two different network layouts, the grid and the ring.

```
# Reset the plotting environment to a single plot
par(mfrow = c(1,1))
```

There are many more layout functions that will result in slightly different visualisations, based on how degree or centrality are weighted (bringing nodes closer together or keeping them further apart). All of these functions start with the term `layout`.

Another important use-case for using `layout_` functions is to retain the position of nodes when plotting the same network repeatedly. If we do not store the layout of a graph, the plotting command will generate slightly different network visualisations every time we run it. To preserve the node position we can return each vertices coordinate and then hand this over to the plotting function whenever we plot the same network:

```
par(mfrow = c(1,3))
coords <- layout.fruchterman.reingold(net)
plot(net, layout = coords, main = "Original")
plot(net, main = "Without layout")
plot(net, layout = coords, main = "With layout")
```

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

14.3.2 Network styling

In addition to the layout, which affects the placing of nodes, we can also change the actual appearance of nodes and edges, such as their size and width, colour,

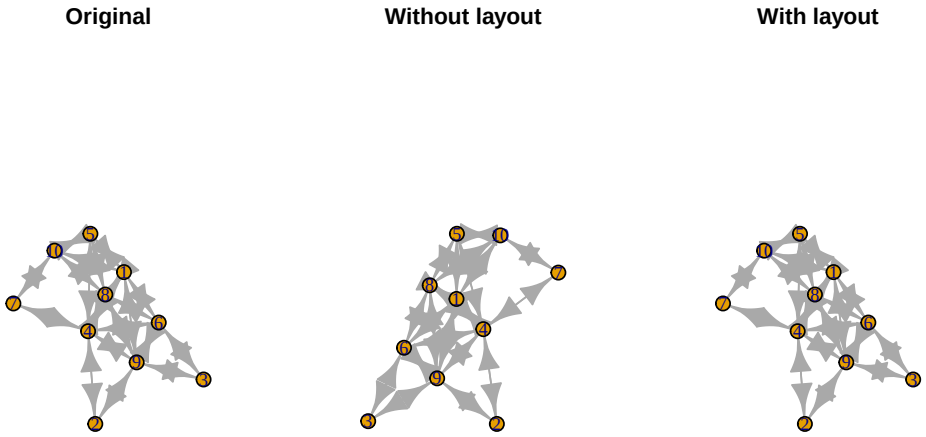


Figure 14.4: Using vertex coordinates preserves the exact layout of a network.

labelling, and more. While the igraph manual is a good reference to figure out all the possibilities, here is the general principle: any attribute we want to change needs either the `vertex.` or `edge.` prefix. For example, to change the colour of all vertices, we would use `vertex.color=`. Have a look at the following example for more ideas of what we can change:

```
plot(net,
     vertex.color = "dodgerblue",
     vertex.label.color = "white",
     vertex.size = 20,
     edge.color = "black",
     edge.width = 1,
     edge.arrow.size = 0.5,
     layout = coords,
     main = "Default layout with styling")
```

As you can see in the graph above, we have changed the colour of the nodes and text, and their overall size (attributes starting with `vertex.`). We have also changed the colour of the edges, their width, and the size of the arrow tips (attributes starting with `edge.`).

Additional to colour our network, we can also use colour to indicate additional information. Say, we know the age of each individual that is represented by a node in out network. How can we instruct `plot()` to use a different colour for each node depending on its age? To do this, we need to first select colours to represent different ages. Then, we can hand this over to the `vertex.color` argument. Below, we use a function called `heat.colors()`. It creates a vector of n contiguous colours that span the 'heat' colour space (from white over yellow to red). We generate 80 different shades and then select 10 colours (we use the `vcount()` function to count the number of vertices in our network) according

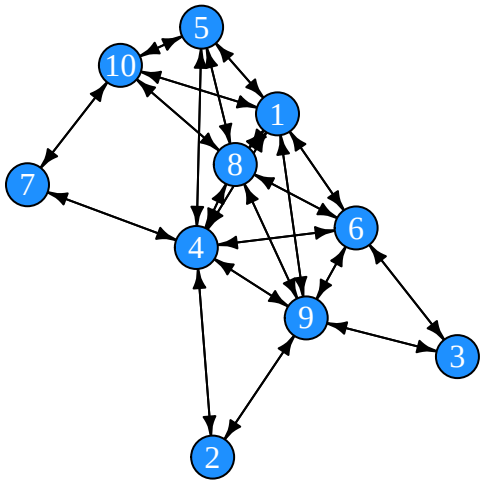


Figure 14.5: Using syling arguments with the `plot()` function allows to manipulate the appearence of vertices, edges, and labels.

to the `age` vector:

```
# Generate random ages
age <- sample(x = 18:80, size = vcount(net), replace = TRUE)
# Select colours
col <- heat.colors(n = 80, rev = TRUE)[age]
# Plot network with col
plot(net,
     vertex.color = col,
     vertex.label = age,
     vertex.label.color = "black",
     vertex.size = 20,
     edge.color = "black",
     edge.width = 1,
     edge.arrow.size = 0.5,
     layout = coords,
     main = "Default layout with styling")
```

Compared to the previous plot, three things have changed: (1) the colour of the nodes, (2) the labels of the nodes (now indicating the correct age), and (3) the colour of the vertex text (black is easier to read on these colours).

It might be a good idea to keep both the `age` and the `col` vector attached to the vertices of our network. We can add them as attributes to the vertices

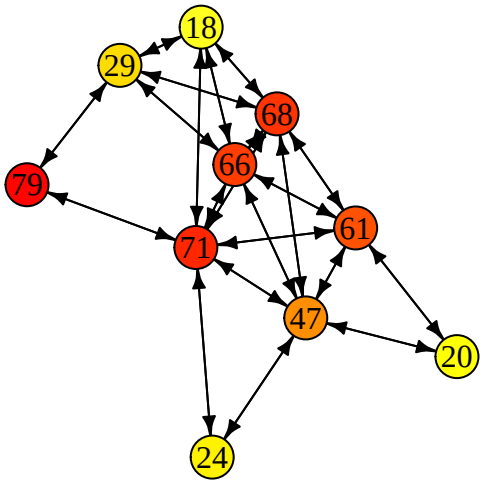


Figure 14.6: In this plot we indicate a node's age by colouring the node with a different colour (pale yellow: younger, red: older)

of our network. Using the `get.vertex.attribute()` function we can see that currently there are no attributes stored at all:

```
get.vertex.attribute(net)
```

```
## list()
```

Let us add the two attributes to the vertices using the `V()` function:

```
# Add an attribute called 'age' and assign the values of the age vector
V(net)$age <- age
# The same for the 'col' vector
V(net)$col <- col
# Return attributes
get.vertex.attribute(net)
```

```
## $age
##  [1] 68 24 20 71 18 61 79 66 47 29
##
## $col
##  [1] "#FF3400" "#FFF200" "#FFFF06" "#FF2700" "#FFFF20" "#FF5200" "#FF0400"
##  [8] "#FF3D00" "#FF8F00" "#FFDC00"
```

```
# Return the network object
net
```

```
## IGRAPH 7610305 D--- 10 44 --
## + attr: age (v/n), col (v/c)
## + edges from 7610305:
##  [1]  1-> 4  1-> 5  1-> 6  1-> 8  1-> 9  1->10  2-> 4  2-> 9  3-> 6  3-> 9
## [11]  4-> 1  4-> 2  4-> 5  4-> 6  4-> 7  4-> 8  4-> 9  5-> 1  5-> 4  5-> 8
## [21]  5->10  6-> 1  6-> 3  6-> 4  6-> 8  6-> 9  7-> 4  7->10  8-> 1  8-> 4
## [31]  8-> 5  8-> 6  8-> 9  8->10  9-> 1  9-> 2  9-> 3  9-> 4  9-> 6  9-> 8
## [41] 10-> 1 10-> 5 10-> 7 10-> 8
```

We now have two attributes that are associated with our network. To use them in plotting you can simply replace `age` in the previous plot with `V(net)$age` and `col` with `V(net)$col`. Also, have a look at the `net` object. It now also tells us that there are two attributes (one is called age, with numeric values, and one is called col, with character values).

Because we have colours representing the age, we may want to remove the labels in each node and make the nodes smaller. This becomes even more important when networks become large. For this, we can simply set `vertex.label` to `NULL`. Also, given that we have a symmetric network (all relationships are reciprocal), we can get rid of the arrow tips. We can do this by turning our network into an 'undirected' network, using the `as.undirected()` function:

```
net <- as.undirected(net)
plot(net,
     vertex.color = V(net)$col,
     vertex.label = NA,
     vertex.size = 9,
     edge.width = 1,
     layout = coords,
     edge.arrow.size = 0.5)
```

Similar to what we did with the nodes (using colour to represent additional information), we can also let edges represent additional information, for example, the width of the stroke can represent the strength of a relationship. Fatter lines could represent stronger friendships. In real life, you will probably have actual values (from experiments or simulations) that you want to use for each edge. Here, we draw 22 random values (using the `ecount()` function to count the number of edges) from a Uniform Distribution using the `runif()` function:

```
# Create random strength values
strength <- runif(n = ecount(net), min = 0, max = 1)
# Assign values to edges
E(net)$weight <- strength
net
```

```
## IGRAPH f371763 U-W- 10 22 --
## + attr: age (v/n), col (v/c), weight (e/n)
## + edges from f371763:
```

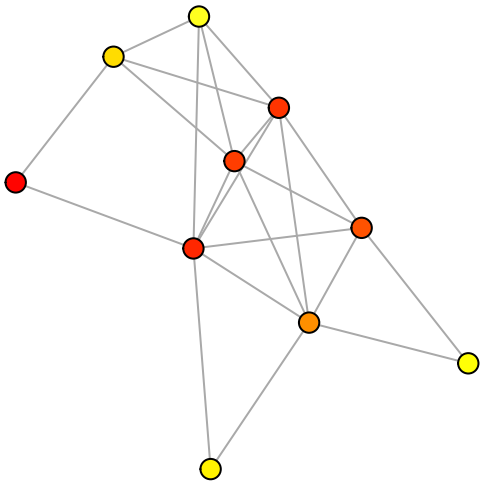


Figure 14.7: A cleaner version of the same network.

```
##  [1] 1-- 4 2-- 4 1-- 5 4-- 5 1-- 6 3-- 6 4-- 6 4-- 7 1-- 8 4-- 8 5-- 8 6-- 8
## [13] 1-- 9 2-- 9 3-- 9 4-- 9 6-- 9 8-- 9 1--10 5--10 7--10 8--10
```

We assigned the friendship strength values to an attribute called `weight`. Take a look at the `net` object. It now says `U-W-`, indicating that it is both an **u**ndirected as well as a **w**eighted network. To plot the edge weights, we instruct the plotting function to use the `E(net)$weight` values for the `edge.width` parameter (note that we multiply the values by some value to make the strokes bigger in the final plot):

```
plot(net,
     vertex.color = V(net)$col,
     vertex.label = NA,
     vertex.size = 9,
     edge.width = E(net)$weight*5,
     edge.arrow.size = 0.5,
     layout = coords)
```

You can now observe strong and weak relationships between individuals, their location relative to each other, and how they cluster. Let us now look at how to quantify the observed network characteristics.

14.4 Analyse social networks

To describe networks, a set of specific terms and measures are used. Let us take a look at the most common measures. In principle, we distinguish between two different levels to describe properties that are associated with networks:

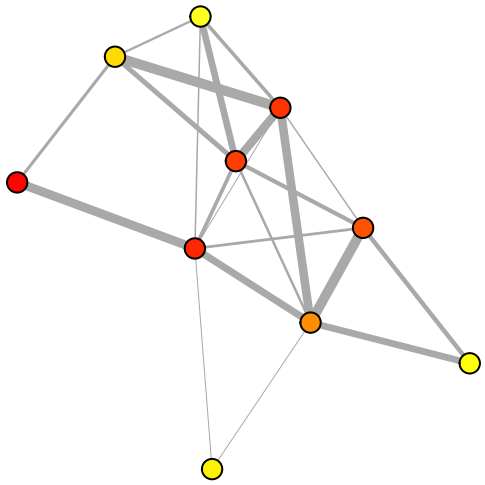


Figure 14.8: To indicate different strength in the relationship of two nodes, we can vary the width of edge between them.

- Population-level network properties
- Individual-level vertex properties

14.4.1 Network properties and characteristics

To retrieve the most basic information about our network we can use the `V()` and `E()` function for vertices and edges of a given network.

```
V(net)
```

```
## + 10/10 vertices, from f371763:
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
E(net)
```

```
## + 22/22 edges from f371763:
##  [1] 1-- 4 2-- 4 1-- 5 4-- 5 1-- 6 3-- 6 4-- 6 4-- 7 1-- 8 4-- 8 5-- 8 6-- 8
## [13] 1-- 9 2-- 9 3-- 9 4-- 9 6-- 9 8-- 9 1--10 5--10 7--10 8--10
```

If our network has attributes associates for its vertices or edges, we can retrieve them with the following two functions:

```
get.vertex.attribute(graph = net)
```

```
## $age
##  [1] 68 24 20 71 18 61 79 66 47 29
##
## $col
```

```
##  [1] "#FF3400" "#FFF200" "#FFFF06" "#FF2700" "#FFFF20" "#FF5200" "#FF0400"
##  [8] "#FF3D00" "#FF8F00" "#FFDC00"
```

```
get.edge.attribute(net)
```

```
## $weight
##  [1] 0.001295885 0.012496948 0.356248695 0.152634992 0.143698542 0.448286853
##  [7] 0.278506543 0.889052924 0.843522300 0.373944373 0.668397761 0.422108888
## [13] 0.854927049 0.098408939 0.676915249 0.708571775 0.987267465 0.253735373
## [19] 0.996833770 0.228852503 0.309746238 0.495430453
```

As you can see, there are two attributes associates with the vertices (`age` and `col`) and one with the edges (`weight`).

As mentioned earlier, `vcount()` and `ecount()` are functionts that return the number of vertices and edges of our network:

```
vcount(net)
```

```
## [1] 10
```

```
ecount(net)
```

```
## [1] 22
```

Additional to these descriptive measures, there is a series of measures that can be calculated:

'Diameter' is a measure for the longest (geodesic) path, i.e. the largest number of steps that are necessary to reach two vertices in a network (using `farthest_vertices()` we can return the ID of the two vertices).

```
diameter(graph = net)
```

```
## [1] 1.284515
```

'Average path length' is the average number of steps that need to be traversed between any two vertices (aka as dyad). We can also use the `distance()` function to return a distance matrix similar to the adjacency matrix.

```
mean_distance(graph = net)
```

```
## [1] 1.6
```

'Edge density' is the proportion of edges present in the graph relative to the number of possible edges (i.e. in a fully connected network with the same number of nodes). This is one of the easier calculations, which we could alo write as `sum(adjm>0) / (length(adjm)-ncol(adjm))` using our adjacency matrix.

```
edge_density(graph = net)
```

```
## [1] 0.4888889
```

'Reciprocity' calculates the proportion of mutual edges relative to all existing edges. This is relevant for directed graphs. As we have an undirected graph, this value is one.

```
reciprocity(graph = net)
```

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

'Clustering coefficient' (also referred to as transitivity, or cliquishness) is the probability that the two neighbours of a vertex are neighbours of each other. This is also called a triangle. You can also imagine it as 'my friends are friends with each other'.

```
transitivity(graph = net)
```

```
## [1] 0.5934066
```

14.4.2 Vertex properties

Additional to these high-level measures, we can use a series of vertex-level measures.

'Degree centrality' refers to the number of (incoming/outgoing/both) edges of a vertex. We can use the `degree()` function to determine the degree centrality of each node:

```
# Number of edges that connected with each node
degree(graph = net)
```

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

```
# The mean of all degree centralities is a general measure of network connectivity
mean(degree(graph = net))
```

```
## [1] 4.4
```

'Strength' is similar to degree centrality but relevant for weighted networks. It is the sum of all adjacent edge weights (a node might have many edges but with very low weights and so with high degree centrality but low strength). In the case of an unweighted network, `degree()` and `strength()` would return the same result.

```
sort(strength(graph = net))
```

```
##  [1] 0.1109059 1.1252021 1.1987992 1.4061340 2.0308630 2.2798683 2.4165034
##  [8] 3.0571391 3.1965262 3.5798258
```

'Closeness centrality' represents the number of steps it takes from a given vertex to any other vertex in the network. It is a measure of how long information on average takes to arrive at this node.

```
closeness(graph = net)
```

```
##  [1] 0.2130991 0.2736146 0.1213209 0.2414627 0.2089982 0.1859112 0.1139896
##  [8] 0.1784780 0.1460121 0.1765328
```

Note that the values are < 1. This is because `igraph` defines closeness centrality as 'the inverse of the average length of the shortest paths to/from all the other vertices in the graph.'

'Betweenness centrality' is the number of shortest paths between nodes that pass through a particular node. It is often seen as a measure for a node's gatekeeping or brokerage potential:

```
betweenness(graph = net)
```

```
##  [1] 11  9  0 22 12  7  0  0  4  8
```

'Eigenvector centrality' is the eigenvector of the adjacency matrix. Vertices with a high eigenvector centrality are connected to many individuals who are connected to many individuals, and so on (see also page rank, `page_rank()`, and authority, `authority_score()`, score functions).

```
eigen_centrality(graph = net)$vector
```

```
##  [1] 1.00000000 0.04180454 0.39619796 0.62662060 0.47443678 0.73405173
##  [7] 0.29928079 0.89975077 0.99650048 0.64895575
```

Sometimes it is good to visualise these characteristics as colour in your network. For example, betweenness centrality can be hard to see by just looking at the graph. Instead of plotting the age of our vertices, we could also plot their network metrics. Here is an example for colouring nodes based on their betweenness centrality. The closer to red colour, the more a node has higher betweenness centrality.

```
between <- betweenness(graph = net)
col <- heat.colors(n = max(between) + 1, rev = TRUE)[between + 1]
# Plot network with col
plot(net,
     vertex.color = col,
     vertex.label = NA,
     vertex.size = 20,
     edge.color = "black",
     edge.width = 1,
     edge.arrow.size = 0.5,
     layout = coords,
     main = "Betweenness centrality")
```

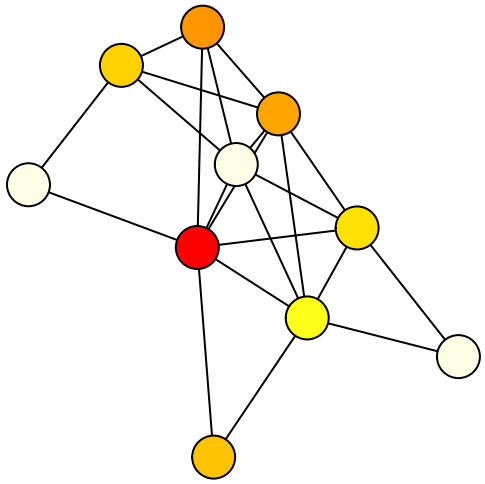


Figure 14.9: Example graph where nodes are coloured based on their betweenness centrality.

14.5 Modelling information transmission in social networks

Now that we know how to generate, plot, and analyse networks, we can move on to use them in a social learning context.

The diffusion of information in social networks differs from the diffusion in well-mixed populations (see our earlier chapters) in that the individual does only have access to the information of her direct network neighbours, i.e. those they share edges with. In comparison, in well-mixed populations (equivalent to a fully connected network) every individual is equally likely to interact with any other individual, and so has access to information from the entire population. Thus, when modelling transmission in social networks, we have to take into account that an individual can only sample from its social environment and not from the entire population (which we have done in the earlier chapters, like the biased and unbiased transmission). Instead, we have to simulate neighbourhood sampling for each node individually.

In this part of the chapter, we will first develop a function to simulate the spread of gossip in networks of varying degree centrality. This will give us a better understanding of the effect of edge density on the diffusion speed. This is a simple model that you can alter to test other network characteristics (e.g. diameter or betweenness centrality). With the second model, we will simulate different

ways of information diffusion (i.e. simple versus complex contagion) and test how network transitivity and the mode of transmission interact.

14.5.1 Gossip diffusion in networked populations

Let us develop a model that simulates the spread of gossip in a group of people. As before, we will assume that a proportion f of the population are friends and thus share an (undirected) edge. Gossip can spread between connected individuals. Gossip spreads from individuals that have gossip to those who do not. Eventually, all individuals that are in some way connected to an individual with gossip will possess the gossip.

Which elements do we need for this model? First, we need to keep track which individual has previously received the gossip (we will use a vector called `gossip` of length N where `TRUE` indicates the possession of gossip). Second, we need an adjacency matrix that describes the connections in our network (this is stored in `adjm` and created using the `create_network()` function that we set up earlier in this chapter). Third, we will need a reporting variable to keep track how the proportion of the population in possession of gossip changes over time. This can be a simple vector of length `r_max`, i.e. the number of rounds. We will call this reporting vector `proportion`. And finally, we need a simulation loop that executed the following three steps:

1. In random sequence go through all individuals
2. If the focal individual has at least one neighbour, select one random neighbour
3. If that neighbour has gossip, change the focal individual's gossip indicator to `TRUE`

We repeat these steps for `r_max` number of rounds. At the end of all rounds, we return a `tibble` where we will store the proportion of the population with gossip at each round, as well as the value of f and the `sim` argument. This is the counter of our simulation. At the moment we only run one simulation at a time, so we set this argument to be one, i.e. `sim = 1` in the function definition. We will see that this counter can be useful when we run repeated simulations later.

```
library(tidyverse)
gossip_model <- function(N, f, r_max, sim = 1){
  # Create a vector indicating possession of gossip and set one entry to TRUE
  gossip <- rep(FALSE, N)
  gossip[sample(x = N, size = 1)] <- TRUE
  # Create a network
  adjm <- create_network(N = N, f = f)
  # Create a reporting variable
  proportion <- rep(0, r_max)

  # Loop over r_max rounds
```

```
  for(r in 1:r_max){
    # In random sequence go through all individuals
    for(i in sample(N)){
      # Select i's neighbourhood
      nei <- adjm[i,] > 0
      # Proceed if there is at least one neighbour
      if(sum(nei) > 0){
        # Choose one random neighbour, j
        j <- sampling(x = which(nei), size = 1)
        # Set i's gossip indicator to TRUE if j's indicator is TRUE
        if(gossip[j]){
          gossip[i] <- TRUE
        }
      }
    }
    # Record proportion of the population with gossip
    proportion[r] <- sum(gossip) / N
    # Increment the round counter
    r <- r + 1
  }
  # Return a tibble with simulation results
  return(tibble(time = 1:r_max, proportion = proportion, f = f, sim = sim))
}
```

Going through this function, you will find two functions that are not part of R. The first is `create_network()`, which we created earlier. The second one is called `sampling()`. This is a wrapper function for the generic `sample()` function. What we want the sample function to do is to return one random value of a vector x of length n. This works well as long as $n > 1$. If $n = 1$, i.e. if we pass to `sample()` a single number, we would expect `sample(x)` to return `x`. However, what you will find if you try this is that `sample()` will return a random sequence of the values `1:x`. So, if `x = 5`, `sample(x)` will return e.g. 2, 4, 5, 1, 3. What we could do to receive the desired outcome is to write a function that returns `x` whenever $n = 1$ and in all other cases uses the `sample()` function. This function could look like the following:

```
sampling <- function(x, size = length(x), prob = NULL){
  if(length(x) == 1){
    return(x)
  } else {
    return(sample(x = x, size = size, prob = prob))
  }
}
```

We can now run our simulations for networks with different degree centrality. We will vary the average number of friends an individual has in the population.

If there are N individuals and an individual has on average two friends then the probability of two individuals sharing an edge is $f = 2/N$, and so on.

```
N <- 1000
set.seed(3)
data <- lapply(X = c(1/N, 2/N, 3/N, 5/N, 8/N, 10/N),
               FUN = function(f) gossip_model(N = N, f = f, r_max = 50))
data_bnd <- bind_rows(data)
```

As in the previous chapter, we use the `lapply()` function to run several independent simulations. The resulting `data` object is a list of tibbles. It is easier to plot the results if we could stack them all on top of each other into a single large tibble. We can do this using the `bind_rows()` function.

Also note the use of the `set.seed()` function here (and in some of the following simulations). This function affects how your computer generates random numbers. For example, repeatedly running `runif(1)` will draw a different number from a uniform distribution every time. However, if we repeatedly run `set.seed(1); runif(1)` we will always receive the same number. By setting a seed we can recover the same (pseudo) random process between different executions. Due to the stochastic nature of our simulations, results might look different between different runs. And so, for illustrative purposes we use a seed here.

We now can use a simple `ggplot()` line plot to see how the frequency of gossip in populations with different degree centrality changes over time:

```
ggplot(data_bnd) +
  geom_line(aes(x = time, y = proportion, col = factor(round(f * N)))) +
  ylab("proportion of individuals with gossip") +
  labs(col = "average number of friends") +
  theme_bw()
```

We can see that there is a big difference between 1 and 2 friends on average but very small between 8 and 10. It is also good to see that 50 turns are sufficient for our system to reach an equilibrium.

We should run the simulation more than once and average the results. That way, we can say more definitely how average degree affects the speed and level of spread. We can also measure how long it takes for some gossip to spread in more than, say, 75% of the population. Again, we will use an `lapply()` function to run our simulations. Note that this time, we use an indicator `i` to number the current simulation run (by setting `sim = i`). This way, we can keep the results from repeated simulation runs with the same starting parameters separate from each other.

```
N <- 100
f <- rep(c(1/N, 2/N, 3/N, 5/N, 8/N, 10/N), each = 10)
```

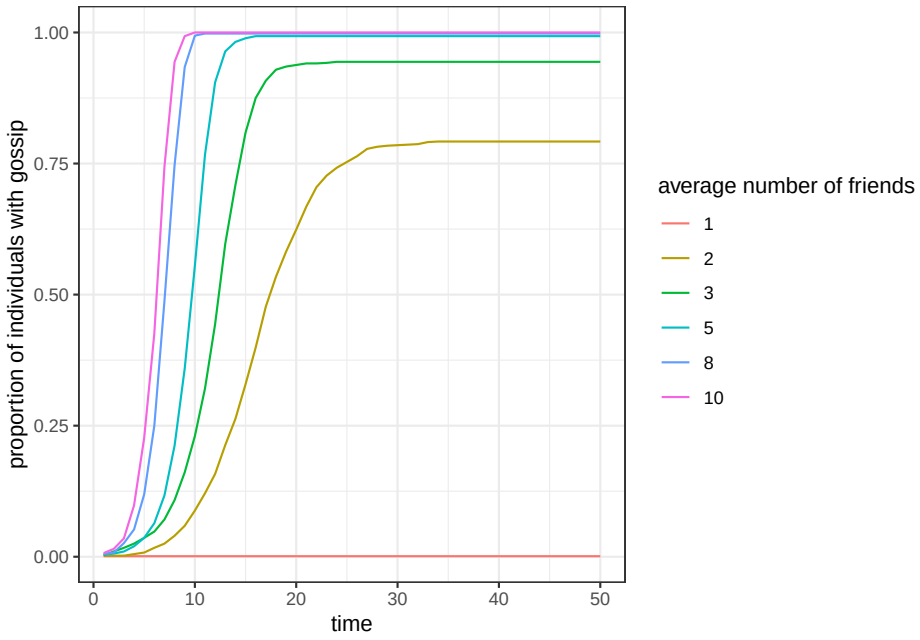


Figure 14.10: Gossip spreads first quickly and then more slowly throughout a population.

```
set.seed(5)
data <- lapply(X = 1:length(f),
               FUN = function(i) gossip_model(N = N, f = f[i], r_max = 50, sim = i))
```

Now we bind the resulting tibbles into a single object and plot the simulation results. Note that we are now using the `sim` counter as input for the `group` argument in `geom_line()`. This tells `ggplot` to draw a line only between those points that belong to the same group, i.e. the same simulation run:

```
data_bnd <- bind_rows(data)
ggplot(data_bnd) +
  geom_line(aes(x = time, y = proportion, col = factor(f * N), group = factor(sim))) +
  ylab("proportion of individual with gossip") +
  labs(col = "average number of friends") +
  theme_bw()
```

As you can see, each simulation run is a bit different even of they have the same starting parameter. Let us plot the proportion of population with gossip in the last simulation step for each degree centrality:

```
data_bnd_last <- data_bnd[data_bnd$time == max(data_bnd$time), ]
ggplot(data_bnd_last) +
```

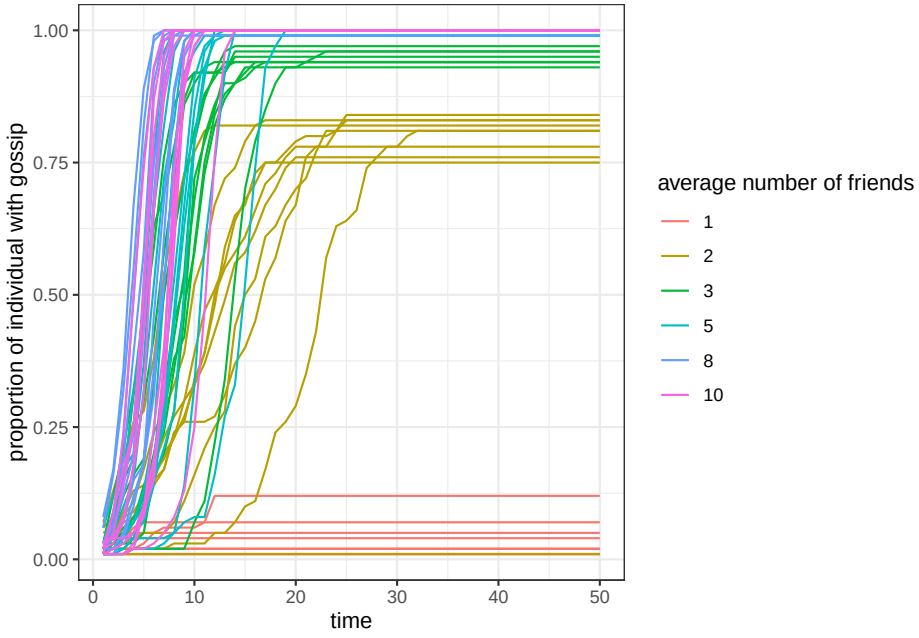


Figure 14.11: Repeated runs of `gossip_model()` show that on average gossip spreads to less than 0.2 of the population if individuals have on average 1 friend. For 2 and more friends gossip spreads to more than three quarter of the population.

```
  geom_boxplot(aes(x = factor(f * N), y = proportion)) +
  xlab("average number of friends") +
  ylab("proportion of individuals with gossip") +
  theme_bw()
```

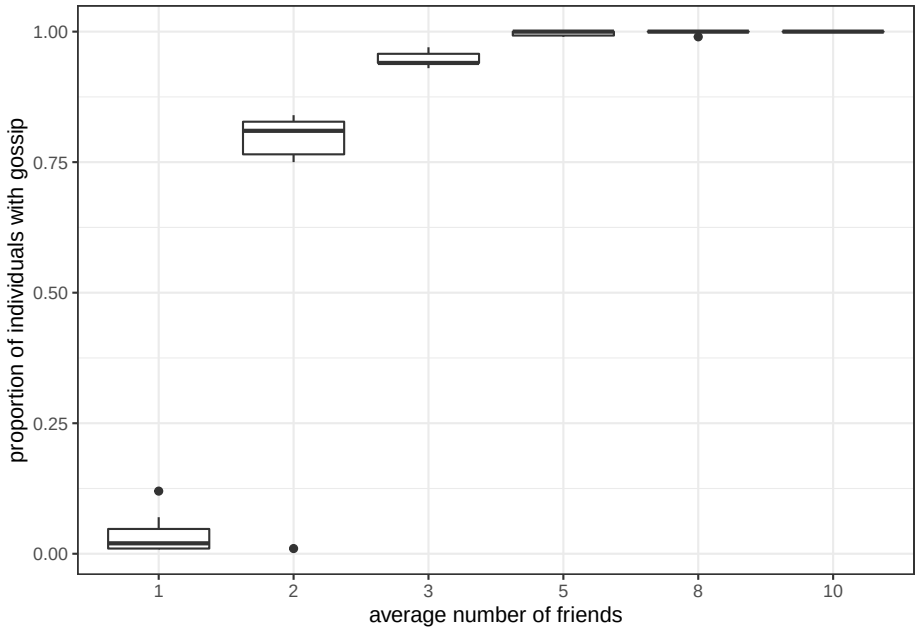


Figure 14.12: Here we plot the end points of each simulation from the previous plot as a boxplot. This, again, shows that there is a dramatic difference of diffusion in networks with 1 to 3 friends, but far less in those with 5 and more friends.

You can see that as the number of friends increase a larger proportion of the population will have gossip after 50 simulation rounds. The increase is very strong initially and then levels off quickly after an average degree of 3.

Another metric we can look at is the number of rounds until 75% of the population own gossip. For this, we will go through each simulation run, that is, each list element stored in `data`, and select the first timestep where `proportion >= 0.75`. Note that some simulations do not reach this 75% and the resulting value will be `NA`. One way to deal with these is to disregard these simulations. Another one is to set all `NA` values to `r_max`, in our case 50, as we will do here.

```
data_bnd_time <- lapply(data, function(dat){
  tibble(dat[1,], time_to_x = which(dat[,"proportion"] >= 0.75)[1])
}) %>% bind_rows()

data_bnd_time$time_to_x[is.na(data_bnd_time$time_to_x)] <- 50
```

```
ggplot(data_bnd_time) +
  geom_boxplot(aes(x = factor(f * N), y = time_to_x)) +
  xlab("average number of friends") +
  ylab("time to spread to 75% of population") +
  theme_bw()
```

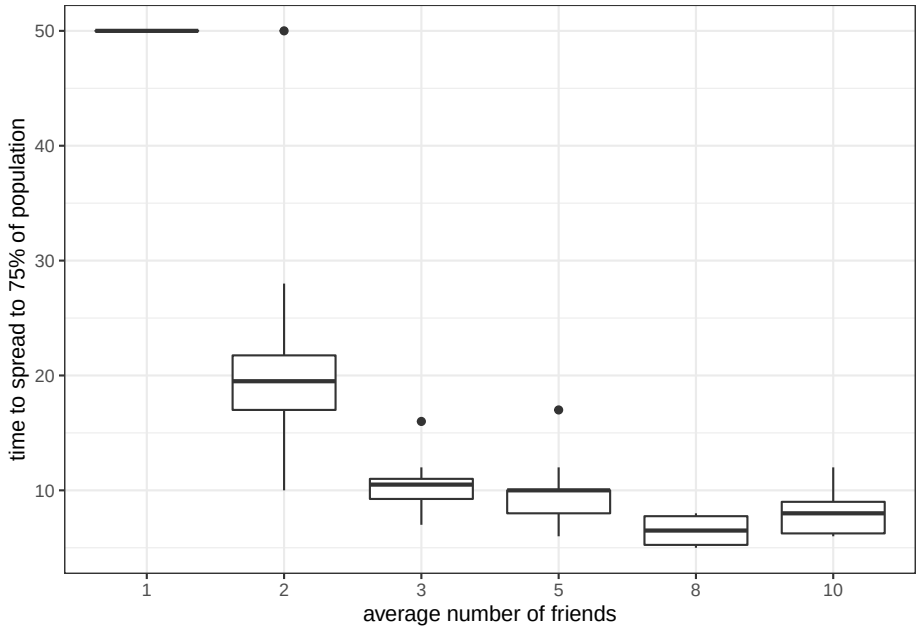


Figure 14.13: Speed at which gossip spreads depending on the average number of friends.

Now we can say that with more friends, information (or gossip) does not only spread to larger proportions of a population but it does so faster. This is true for the random networks that we have tested here. However, as mentioned earlier, there are many different network categories that differ in degree distribution, average path length, clustering, and others. With the next model, we will take a closer look at the effect of clustering.

14.5.2 Complex versus simple contagion information transmission

Another factor that affects the spread of information in networks is the mode of information transmission. That is, often information is not transmitted from one individual to another in a simple contagion-like manner, where exposure to one informed individual is sufficient, but instead requires increased social facilitation. In other words, often we are more likely to acquire behaviours from

others if this behaviour is more frequent in our neighbourhood. This kind of transmission is called 'complex contagion'. A study on the spread of health behaviour reported that a new behaviour spread faster in clustered than in random networks (Centola [2010]). The author explained the result by pointing to the increased social feedback a focal individual can receive in clustered networks, where neighbours are more likely also neighbours with each other, as compared to random networks. Let us model complex contagion to better understand how the mode of transmission affects the speed of diffusion in different networks.

In the previous iteration of `gossip_model()`, we selected a random neighbour, j, of a focal individual i. If j had gossip then i acquired the gossip with certainty. For example, if i had 3 neighbours of which only one had gossip, his probability to acquire gossip within the next round, p_g, was 1/3, as there is a 1 in 3 chance that we randomly pick the neighbour with gossip. So, instead of writing:

```
j <- sampling(x = which(nei), size = 1)
if(gossip[j]){
  gossip[i] <- TRUE
}
```

We could also write:

```
p_g <- sum(gossip * nei) / length(nei)
if(runif(n = 1, min = 0, max = 1) <= p_g){
  gossip[i] <- TRUE
}
```

Here, the `if` statement is true if a value that is randomly drawn from a uniform distribution (that is what `runif()` is doing), is smaller or equal to the probability to encounter an individual with gossip, p_g (i.e. number of neighbours with gossip, `sum(gossip * nei)`, divided by the number of neighbours, `length(nei)`). Note that `gossip * nei` returns a boolean vector that is only `TRUE` for individuals that are both a neighbour and have gossip. As you can see, the probability that i acquires gossip scales linearly with the proportion of neighbours with gossip. This is a good approximation for the spread of information following simple contagion.

In the case of complex contagion, instead, we assume that occasional interactions (or exposures) are less important than more frequent ones. We might require an individual to be exposed to gossip repeatedly. This could be, say, the repeated encounter with a gossiper or encounter of more than one gossiper. We can then write:

```
if(runif(n = 1, min = 0, max = 1) <= (sum(nei) / length(nei))^e){
  gossip[i] <- TRUE
}
```

where the exponent `e` affects the shape of the function that describes the probability to acquire gossip:

```
prop_nei_info <- seq(from = 0, to = 1, length.out = 10)
e <- c(1,2)
contagion <- tibble(
  contagion = rep(c("simple", "complex"), each = 10),
  x = rep(prop_nei_info, 2),
  y = c(prop_nei_info^e[1], prop_nei_info^e[2]))

ggplot(contagion, aes(x = x, y = y, col = contagion)) +
  geom_point() +
  geom_line() +
  xlab("proportion of neighbours with gossip") +
  ylab("probability to acquire gossip") +
  theme_bw()
```

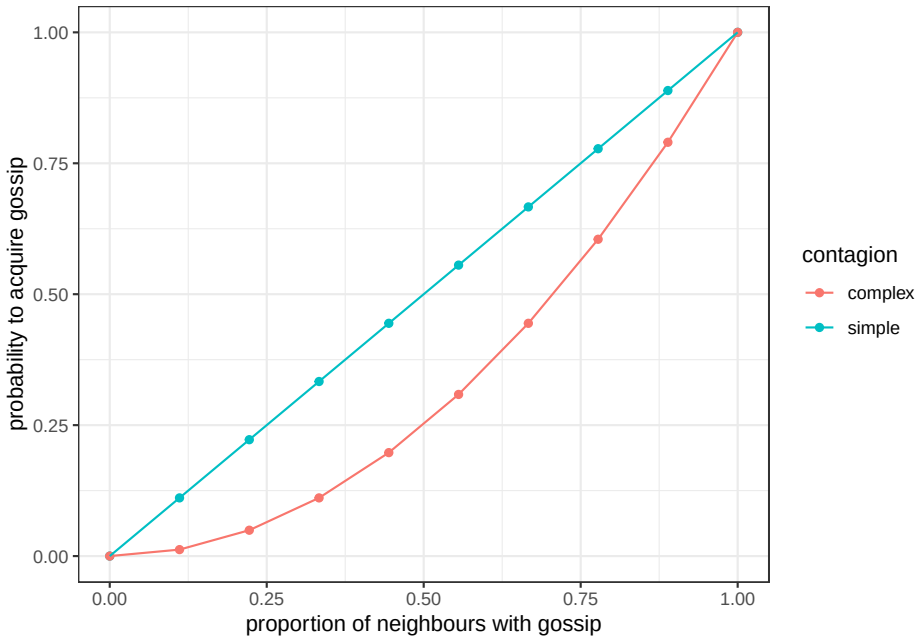


Figure 14.14: In simple contagion the probability to acquire gossip scales linearly with the proportion of neighbours with gossip, whereas it increases superlinearly (here exponentially) in the case of complex contagion.

The figure shows that in the case of simple contagion ($e = 1$), the probability to acquire gossip increases linearly with the proportion of gossiping neighbours. For complex contagion (e.g. $e = 2$), however, acquisition increases exponentially.

Let us test whether we can simulate the empirical results from the study we mentioned earlier, i.e. that information spreading in a complex contagion-like manner spreads faster in clustered networks. To simulate networks that are as close

as possible to the ones used by Centola [2010] we will use the `make_lattice()` function in igraph. The function sets up simple lattice graphs of varying dimensions and lengths along each dimension. In our case, we want a two-dimensional graph, of 100 individuals, and a neighbourhood size of two, i.e. an individual is not only connected to the direct neighbours in a square lattice but also the direct neighbour of each of their neighbours. Additionally, we set the `circular` argument to be true. This connects the nodes of one edge of the lattice with the nodes on the opposite side. The resulting graph looks like a torus.

```
net_clust <- make_lattice(length = 10, dim = 2, circular = T, nei = 2)
```

This is a regular graph (i.e. each neighbourhood looks the same). We can use the `rewire()` function to turn it into a random graph. In combination with `keeping_degseq()`, the function will take two random edges (say between nodes A, B and C,D) and rewire them (to A,D and C,B). This shuffling will reduce the clustering but keep the degree of each node the same. This is good because we do not want to change too many characteristics of the network, which would make it more complicated to explain differences in the simulation results.

```
net_rand <- rewire(graph = net_clust, with = keeping_degseq(loops = F, niter = 10^3))
```

Let us have a look at the network characteristics of the clustered and the random network:

```
head(degree(net_clust))
```

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

```
head(degree(net_rand))
```

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

```
transitivity(net_clust)
```

```
## [1] 0.4545455
```

```
transitivity(net_rand)
```

```
## [1] 0.09818182
```

While the degree centrality remains unchanged, the clustering coefficient of the random network is only about a quarter of the lattice network. We will use these two networks in the following simulations.

Now that we have the networks, let us modify the `gossip_model()` function. We will change the function such that (1) we can hand over the network directly (`net` argument), (2) this network will be rewired if the additional argument `rewire` is set to `TRUE`, and (3) contagion can be simple or complex, which we will set with one more argument (`e`).

```r
gossip_model_2 <- function(net, rewire, e = 1, r_max, sim = 1){
  # Rewire network if random is set to TRUE
  if(rewire){
    net <- rewire(graph = net, with = keeping_degseq(loops = F, niter = 10^3))
  }
  # Get adjacency matrix from network
  adjm <- get.adjacency(net, sparse = F)
  # Turn adjacency matrix into boolean (TRUE / FALSE)
  adjm_bool <- adjm > 0
  # Set number of individuals based adjacency matrix
  N <- vcount(net)

  # Create a vector indicating possession of gossip and set one entry to TRUE
  gossip <- rep(FALSE, N)
  gossip[sample(x = N, size = 1)] <- TRUE

  # Create a reporting variable
  proportion <- rep(0, r_max)

  # Rounds
  for(r in 1:r_max){
    # In random sequence go through all individuals without gossip
    for(i in sample(N)){
      # Select i's neighbourhood (boolean)
      nei <- adjm_bool[i,]
      # Proceed if there is at least one neighbour
      if(sum(nei) > 0){
        # Simple contagion for e = 1 and complex contagion for e = 2
        if(runif(n = 1, min = 0, max = 1) <= (sum(gossip * nei) / length(nei))^e){
          gossip[i] <- TRUE
        }
      }
    }
    # Record proportion of the population with gossip
    proportion[r] <- sum(gossip) / N
    # Increment the round counter
    r <- r + 1
  }
  # Return a tibble with simulation results
  return(tibble(time = 1:r_max,
                proportion = proportion,
                time_to_max = which(proportion == max(proportion))[1],
                e = e,
                network = ifelse(test = rewire, yes = "random", no = "clustered"),
                sim = sim))
```

```
}
```

Note that in this function we also updated the output. Now, we return not only the proportion of individuals with gossip at each simulation round but also the time at which the maximum proportion was reached (`time_to_max`). To make plotting easier, we also return the `e` argument, whether the network was rewired (if `TRUE` we return `random`, if `FALSE` we return `clustered`), and the simulation count `sim`.

You might have also noticed that we turned the numeric adjacency matrix into a boolean matrix that only contains `TRUE` and `FALSE` values (using `adjm > 0`). This is a little trick to speed up simulations. Instead of repeatedly asking R to identify which value in an individual's neighbourhood is a `1`, we can get the same result instantly by turning all `1`s into `TRUE`s.

Let us now run the simulation for random and clustered networks, and for simple ($e = 1$) and complex ($e = 2$) contagion:

```
set.seed(1)
res <- bind_rows(
  gossip_model_2(net = net_clust, rewire = TRUE, e = 1, r_max = 500),
  gossip_model_2(net = net_clust, rewire = FALSE, e = 1, r_max = 500),
  gossip_model_2(net = net_clust, rewire = TRUE, e = 2, r_max = 5000),
  gossip_model_2(net = net_clust, rewire = FALSE, e = 2, r_max = 5000)
  )

ggplot(res) +
  geom_line(aes(x = time, y = proportion, col = network)) +
  facet_wrap("e", labeller = label_both, scales = "free_x") +
  theme_bw()
```

As you can see, while there is no major difference in the spread of information in clustered and random networks for simple contagion (left), we find that information spreads faster in clustered networks if the transmission follows complex contagion dynamic. The reason for this is that in clustered networks an individual's neighbours are more likely to also be connected. This increases the likelihood that the neighbours also share the same information, and, in turn, increases the individual's exposure to this information.

14.6 Summary of the model

In this chapter, we have explored individual-level effects on population-level outcomes. That is, how the structure of individual interactions affect the spread of information in a population. We have seen that both network characteristics (degree and clustering) but also the mode of information transmission (simple versus complex) can have strong effects on how efficiently information travels through a population.

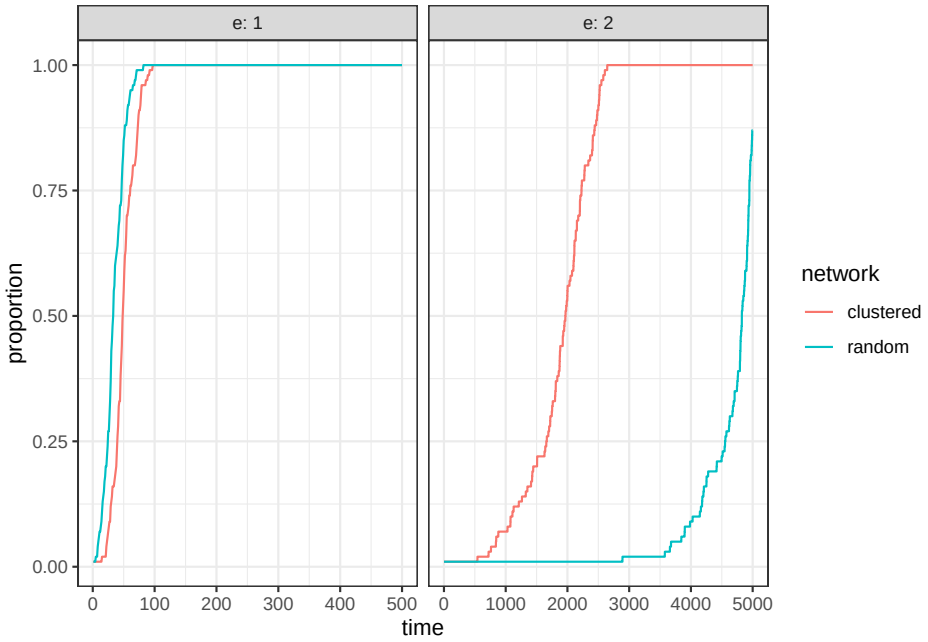


Figure 14.15: There are no differences in the spread of gossip in clustered and random networks if it spreads based on simple contagion (left, $e = 1$). However, if gossip spreads based on complex contagion (right, $e = 2$), it spreads faster in clustered than in random networks.

14.7 Further Reading

There is an increasing number of theoretical models that are looking at the effect of network characteristics on the spread of cultural traits, e.g. O'Sullivan et al. [2015]. Several empirical studies with humans (Hill et al. [2014]) and non-human animals (Aplin et al. [2012]), have recorded network structures and reported on the effects of network structure on the spread of novel information. It is also interesting to ask, how the network structure itself might be the result of cultural dynamics (see e.g. Smolla and Akçay [2019]). A good overview of relevant literature is provided in a review by Derex and Mesoudi [2020].

Chapter 15

Group structured populations and migration

Many simulations assume well-mixed populations, that is, populations of individuals that have an equal chance of encountering each other. In the previous chapter, we have looked at the effects of structured interactions on the transmission of cultural traits. Both structured and unstructured interactions can be good approximations of the real world, depending on the context and research question. What about structured populations, a combination of the two? That is, a large population of individuals is divided into subsets, where individuals are more likely to encounter individuals from the same subset but much less likely to encounter individuals from a different one. Here learning would almost exculsively occur within each subset. However, individuals may migrate between subsets, bringing along their own selection of cultural traits, or might visit another subset and then return with a new cultural trait. In this chapter we will take a closer look at these two scenarios.

15.1 Modelling migration and contact between population subsets

Before we model subset populations, let us begin by developing a simple transmission model, similar to what we have done in the first chapter of this book. We start with a population of size n, and b instances of a behavioural trait. That is, an individual will only ever have one version of a behaviour, for example, greeting another individual with a handshake or by bumping elbows. We choose this to keep the example simple. Other possibilities would be, for example, to model the migration of distinct behaviours.

Let us set up a population. Because we are only interested in which instance

of a particular behaviour an individual expresses, each individual can be fully described by that instance, and so, we can represent the entire population as a vector of expressed behaviours (`behaviours`).

```
n <- 100
b <- 2

behaviours <- sample(x = b, size = n, replace = TRUE)
table(behaviours)
```

```
## behaviours
##  1  2
## 51 49
```

The `table()` function counts each element of a given vector. It is similar to `tabulate()` which we used previously, however, `table()` returns a named vector, so that you know exactly which element is present how many times.

Each individual in our population expresses only one behaviour. Occasionally, one of the individuals will copy the behaviour of another individual. To simulate this behavioural updating, we select a random individual, which copies a randomly selected behaviour from the population. This approach is identical to the unbiased learning in our earlier chapters. We can simulate repeated updating events by wrapping a `for` loop around the code. Additionally, we will add a record variable (`rec_behav`) to store the frequency of each behaviour.

```
library(tidyverse)
r_max <- 1000
rec_behav <- tibble(time = 1:r_max, b1 = 0, b2 = 0)

for(round in 1:r_max){
  # Unbiased copying of a behaviour by a random individual
  behaviours[ sample(x = n, size = 1) ] <- sample(x = behaviours, size = 1)
  # Record the frequency of each trait in each round
  rec_behav[round, "b1"] <- sum(behaviours==1)
  rec_behav[round, "b2"] <- sum(behaviours==2)
}
rec_behav
```

```
## # A tibble: 1,000 x 3
##     time    b1    b2
##    <int> <dbl> <dbl>
##  1     1    50    50
##  2     2    49    51
##  3     3    50    50
##  4     4    50    50
##  5     5    50    50
##  6     6    50    50
```

```
##  7     7    50    50
##  8     8    50    50
##  9     9    50    50
## 10    10    50    50
## # ... with 990 more rows
```

To plot the results with `ggplot()`, it would be bst to turn the 'wide' data (where repeated measures/different categories are in the same line) into 'long' data (where each line has only one observation/category). To 'rotate' our data we can use the `pivot_longer()` function. In its arguments we tell it to combine the results of columns `b1` and `b2` into `beaviour`, which indicates the name of the behaviour, and `freq`, which is the frequency of that behaviour. The function will repeat the entries of the `time` column accordingly:

```
rec_behav_l <- pivot_longer(data = rec_behav, names_to = "behaviour",
                            values_to = "freq", cols=c("b1","b2"))
rec_behav
```

```
## # A tibble: 1,000 x 3
##     time    b1    b2
##    <int> <dbl> <dbl>
##  1     1    50    50
##  2     2    49    51
##  3     3    50    50
##  4     4    50    50
##  5     5    50    50
##  6     6    50    50
##  7     7    50    50
##  8     8    50    50
##  9     9    50    50
## 10    10    50    50
## # ... with 990 more rows
```

```
rec_behav_l
```

```
## # A tibble: 2,000 x 3
##     time behaviour  freq
##    <int> <chr>     <dbl>
##  1     1 b1           50
##  2     1 b2           50
##  3     2 b1           49
##  4     2 b2           51
##  5     3 b1           50
##  6     3 b2           50
##  7     4 b1           50
##  8     4 b2           50
##  9     5 b1           50
## 10     5 b2           50
```

```
## # ... with 1,990 more rows
```

Now, we can plot the frequency of each behaviour over time:

```
ggplot(rec_behav_l) +
  geom_line(aes(x = time, y = freq/n, col = behaviour)) +
  scale_y_continuous(limits = c(0,1)) +
  ylab("proportion of population with behaviour") +
  theme_bw()
```

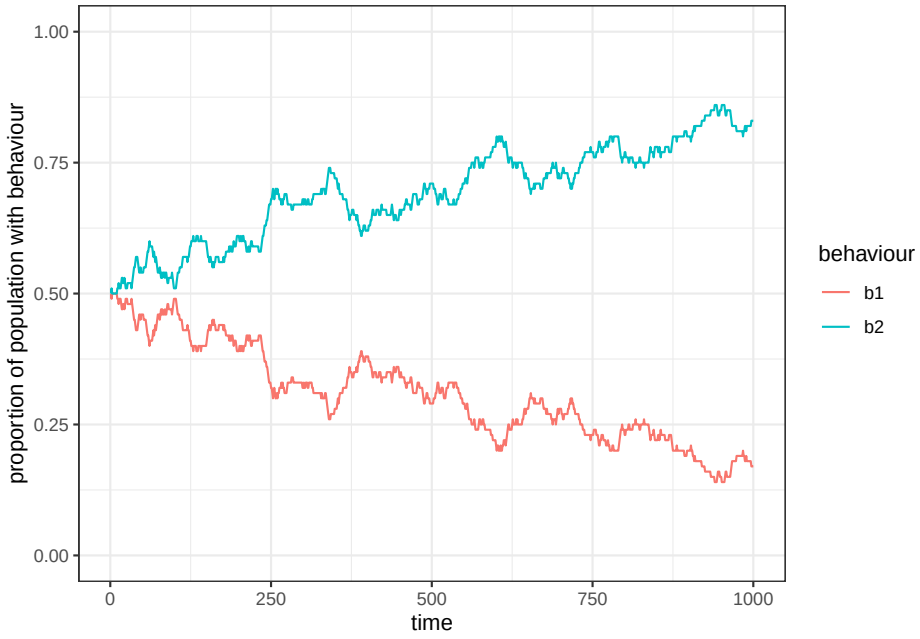


Figure 15.1: On average the frequenca of two behaviours that are transmitted without a bias will fluctuate around 0.5.

As we would expect from an unbiased transmission, the frequency of the two traits will move around 0.5.

We can wrap this code in a function to make it easier to use in the future:

```
structured_population_1 <- function(n, b, r_max){
  behaviours <- sample(x = 1:b, size = n, replace = TRUE)
  rec_behav <- matrix(NA, nrow = r_max, ncol = b)

  for(round in 1:r_max){
    # Unbiased copying of a behaviour by a random individual
    behaviours[ sample(x = n, size = 1) ] <- sample(x = behaviours, size = 1)
    rec_behav[round,] <- unlist(lapply(1:b, function(B) sum(behaviours==B))) / n
  }
```

```
  # Turn matrix into tibble
  rec_behav_tbl <- as_tibble(rec_behav)
  # Set column names to b1, b2 to represent the behaviours
  colnames(rec_behav_tbl) <- paste("b" ,1:b, sep = "")
  # Add a column for each round
  rec_behav_tbl$time <- 1:r_max
  # Turn wide format into long format data
  rec_behav_tbl_l <- pivot_longer(data = rec_behav_tbl, names_to = "behaviour", values_to = "freq
  # Return result
  return(rec_behav_tbl_l)
}
```

Note that we initially create the matrix `rec_behav`, which we then convert to a `tibble` object. The advantage here is that we can quickly create a matrix of a certain size (here, the number of rounds times number of behaviours), in which we can record the frequencies of each behaviour. The `tibble`, however, will make handling and plotting our data easier.

Let us test what would happen if we ran this simulation with a very small population size of $n = 20$:

```
# Run simulation
res <- structured_population_1(n = 20, b = 2, r_max = 1000)

# Plot results
ggplot(res) +
  geom_line(aes(x = time, y = freq, col = behaviour)) +
  scale_y_continuous(limits = c(0,1)) +
  ylab("proportion of population with behaviour") +
  theme_bw()
```

We observe that the two behaviours fluctuate around 0.5 until, by chance, one behaviour is completely replaced by the other one. This is simply due to drift, which affects all small populations or very long time scales, as we already saw in the first chapter.

15.2 Subdivided population with limited contact

Let us now move on from the single population to a population that is divided into subsets (we will call them `clusters`, as `subset()` is a generic function in R). For simplicity, we will assume only two instances of a behaviour. That way, we only need to track the frequency of one behaviour, p, in each subset, as the frequency of the other is simply $1 - p$. We will also assume that there are the same number, n, of individuals in each cluster, c.

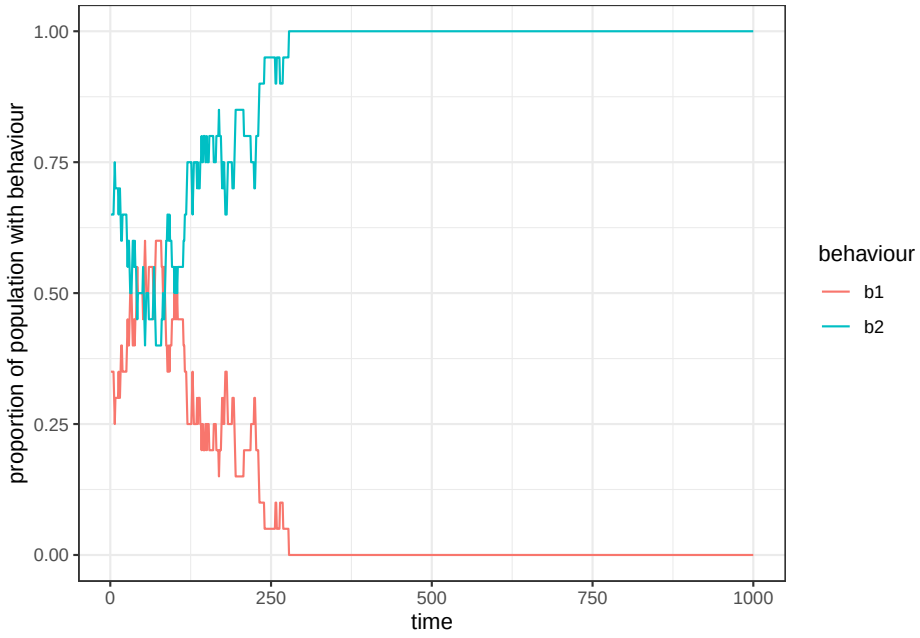


Figure 15.2: In smaller populations (here, $n = 20$), drift might lead to the sudden exclusion of one of the two behaviours from the group, such that the other behaviour becomes fixed.

```
structured_population_2 <- function(n, c, r_max){
  total_pop <- c * n
  cluster <- rep(1:c, each = n)
  behaviours <- sample(x = 2, size = total_pop, replace = TRUE)
  rec_behav <- matrix(NA, nrow = r_max, ncol = c)

  for(round in 1:r_max){
    behaviours[ sample(x = total_pop, size = 1) ] <- sample(x = behaviours, size = 1)
    # Recalculate p for each cluster
    for(clu in 1:c){
      rec_behav[round, clu] <- sum(behaviours[cluster == clu] == 1) / n
    }
  }
  rec_behav_tbl <- as_tibble(rec_behav)
  # Set column names to c1, c2 to represent each cluster
  colnames(rec_behav_tbl) <- paste("c", 1:c, sep = "")
  rec_behav_tbl$time <- 1:r_max
  rec_behav_tbl_l <- pivot_longer(data = rec_behav_tbl, names_to = "cluster",
                                  values_to = "p", !time)
  return(rec_behav_tbl_l)
}
```

This function is very similar to `migration_model_1()` but accounts for the additional clusters. For example, the columns of `rec_behav` now store p for each cluster. We calculate these values in every time step. We could also just calculate it for the cluster of the observing individual, however, if we keep the calculation more general, we will not have to change the code should we want individuals to be able to permanently move from one cluster to another. In that case we would need to calculate p for both clusters.

Also, note that this time we did not tell `pivot_longer()` which columns to pivot but which one we want to keep unchanged (`!time`). This is because we do not know the numbers of columns before we run the simulation. Therefore, it is easier to just make an exception for the time column.

Let us run the simulation for 4 subsets with each 50 individuals:

```
res <- structured_population_2(n = 50, c = 4, r_max = 1000)

ggplot(res) +
  geom_line(aes(x = time, y = p, col = cluster)) +
  scale_y_continuous(limits = c(0,1)) +
  ylab("proportion of behaviour 1 in cluster") +
  theme_bw()
```

Executing this code repeatedly will show you two things. First, on average the frequency of each behaviour will still be around 0.5, and second that the

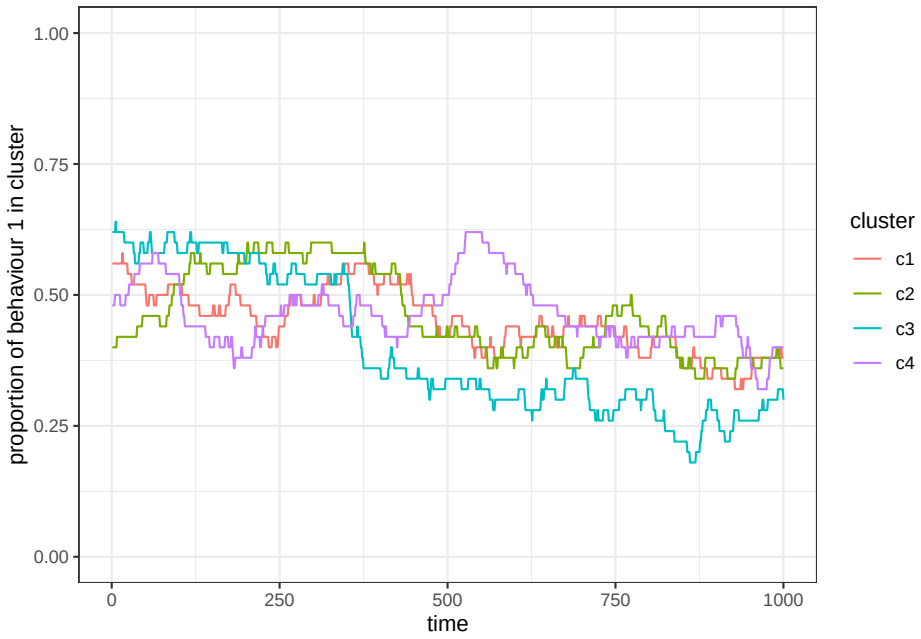


Figure 15.3: The frequency of one out of two behaviours in two subsets will fluctuate around 0.5 if individuals are equally likely to learn from individuals in both subsets (or clusters).

frequency changes are correlated between all subsets. This is expected because, with the current version of our model, individuals do not distinguish between or have different access to individuals of either cluster. In fact, the fluctuations we observe here are purely stochastic and based on the relatively small subsets (try running the code with e.g. $N = 1000$).

Let us now move on to the case where members of a subset preferentially learn from others within their cluster. This might be the case where individuals spent most of their time in their subsets and only occasionally interact with individuals from other subsets. To simulate this, we can use most of the code from `structured_population_2()` with a few small changes. First, we will change the `sample()` function. Instead of sampling the entire population, we want the observer to sample preferentially (or exclusively) from its own cluster. To achieve this, we will use the `prob` argument in the `sample()`. This argument gives a weight (or probability) with which an element of a provided set is chosen. By default, each element has a weight of 1 (or a probability of $1/N$) and thus is equally likely to be selected. To limit our scope to individuals within the same cluster, we can simply set the weight to 0 for all individuals that are in a different cluster and to 1 for those that are in the same cluster. Assuming an individual is in cluster 2 then we select all other individuals in the same cluster using `cluster == cluster_id`, where `cluster_id` is 2. This will return a vector with `TRUE` and `FALSE` values. We can turn this into weights (i.e. 0s and 1s) simply by multiplying the vector with 1, R will then automatically turn the boolean into a numeric vector. Additionally, we will select two individuals from the cluster, an observer, and a demonstrator (or model). Of course, we can only perform this, `if` there are at least 2 individuals in the cluster. Take a look at the new function:

```
structured_population_3 <- function(n, c, r_max){
  total_pop <- c * n
  cluster <- rep(1:c, each = n)
  behaviours <- sample(x = 2, size = total_pop, replace = TRUE)
  rec_behav <- matrix(NA, nrow = r_max, ncol = c)

  for(round in 1:r_max){
    # Choose a random cluster
    cluster_id <- sample(c, 1)
    # If there are at least two individuals in this cluster
    if(sum(cluster == cluster_id)>1){
      # Choose a random observer and a random individual to observe within the same cluster
      observer_model <- sample(x = total_pop, size = 2, replace = F,
                               prob = (cluster == cluster_id)*1)
      behaviours[ observer_model[1] ] <- behaviours[ observer_model[2] ]
    }

    for(clu in 1:c){
```

```
      rec_behav[round, clu] <- sum(behaviours[cluster == clu] == 1) / n
    }
  }
  return(matrix_to_tibble(m = rec_behav))
}
```

You might have noticed a new function at the very end: `matrix_to_tibble()`. This is a little helper function. As we keep turning matrices into tibbles, changing their column names, and adding a time column at the end of our simulation, we can also separate this process in its own function and re-cycle it in the future versions of our simulation function. This is generally useful whenever you have a piece of code that you keep replicating. This is what the helper function looks like:

```
matrix_to_tibble <- function(m){
  m_tbl <- as_tibble(m)
  colnames(m_tbl) <- paste("c" ,1:ncol(m), sep = "")
  m_tbl$time<- 1:nrow(m)
  m_tbl_l <- pivot_longer(data = m_tbl, names_to = "cluster", values_to = "p", !time)
  return(m_tbl_l)
}
```

Let us run a simple example with three subsets and each $n = 20$:

```
res <- structured_population_3(n = 20, c = 3, r_max = 1000)
ggplot(res) +
  geom_line(aes(x = time, y = p, col = cluster)) +
  scale_y_continuous(limits = c(0,1)) +
  ylab("proportion of behaviour 1 in cluster") +
  theme_bw()
```

When we run this simulation repeatedly, you will see that sometimes behaviour 1 gets lost in one, both, or neither of the clusters. Because in this iteration of our simulation there are no interactions between individuals of different clusters, we are essentially simulating three (small) independent populations.

Let us change the code so that we can alter the rate at which individuals from different subsets might encounter each other. In mathematical terms, let $\omega = s$ be the probability to observe another individual, with $s = 0$ for individuals of another subset and $s = 1$ for individuals of the same subset. We can alter ω to allow interaction with individuals from another subset by adding a contact probability, p_c, such that $\omega = \frac{s+p_c}{1+p_c}$. We divide by $1 + p_c$ to keep $0 \leq \omega \leq 1$. With this change the probability to encounter an individual from any subset is at least p_c. Take a look at the updated function:

```
structured_population_4 <- function(n, c, p_c, r_max){
  total_pop <- c * n
  cluster <- rep(1:c, each = n)
```

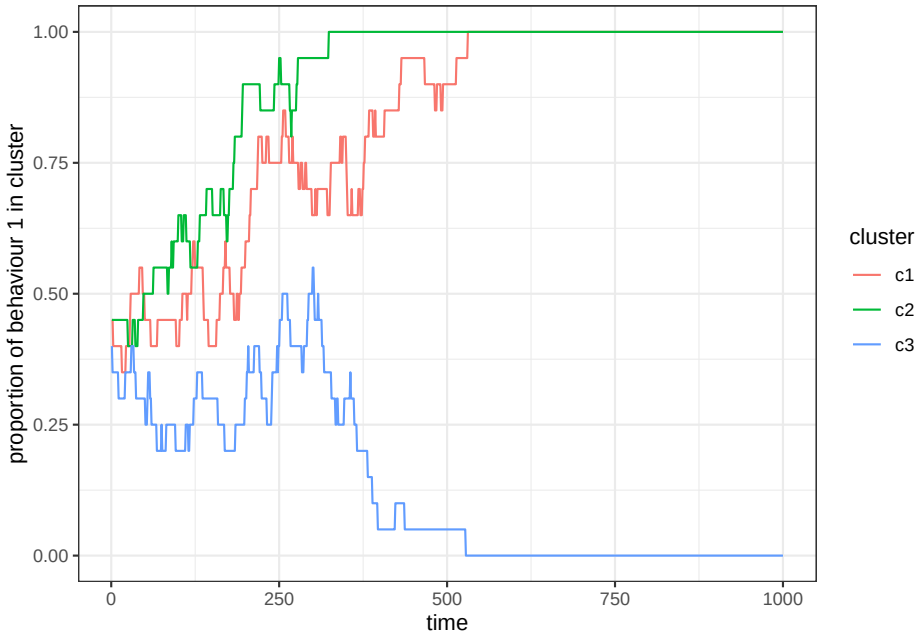


Figure 15.4: When individuals copy behaviours only from individuals within their own subset, we find that the frequency of behaviour 1 becomes uncorrelated between the two subsets. In this example, behaviour 1 is lost in cluster 2, whereas it is still present in cluster 1 at the end of the simulation.

```
  behaviours <- sample(x = 2, size = total_pop, replace = TRUE)
  rec_behav <- matrix(NA, nrow = r_max, ncol = c)

  for(round in 1:r_max){
    cluster_id <- sample(c, 1)
    s <- cluster == cluster_id
    if(sum(s)>1){
      # Choose a random observer and a random individual to observe within the same cl
      observer_model <- sample(x = total_pop, size = 2, replace = F,
                               prob = (s + p_c) / (1 + p_c))
      behaviours[ observer_model[1] ] <- behaviours[ observer_model[2] ]
    }

    for(clu in 1:c){
      rec_behav[round, clu] <- sum(behaviours[cluster == clu] == 1) / n
    }
  }
  return(matrix_to_tibble(m = rec_behav))
}
```

Let us now run simulations for different contact probabilities. We will simulate five subsets and use $p_c \in \{0, 0.1, 1\}$, i.e. no contact, some contact, and full contact:

```
res_0  <- structured_population_4(n = 20, c = 5, p_c = 0, r_max = 1000)
res_01 <- structured_population_4(n = 20, c = 5, p_c = 0.1, r_max = 1000)
res_1  <- structured_population_4(n = 20, c = 5, p_c = 1, r_max = 1000)

ggplot(res_0) +
  geom_line(aes(x = time, y = p, col = cluster)) +
  scale_y_continuous(limits = c(0,1)) +
  ylab("proportion of behaviour 1 in cluster") +
  theme_bw()
```

```
ggplot(res_01) +
  geom_line(aes(x = time, y = p, col = cluster)) +
  scale_y_continuous(limits = c(0,1)) +
  ylab("proportion of behaviour 1 in cluster") +
  theme_bw()
```

```
ggplot(res_1) +
  geom_line(aes(x = time, y = p, col = cluster)) +
  scale_y_continuous(limits = c(0,1)) +
  ylab("proportion of behaviour 1 in cluster") +
  theme_bw()
```

With $p_c = 0$, the subsets (again) act as independent populations that fluctuate

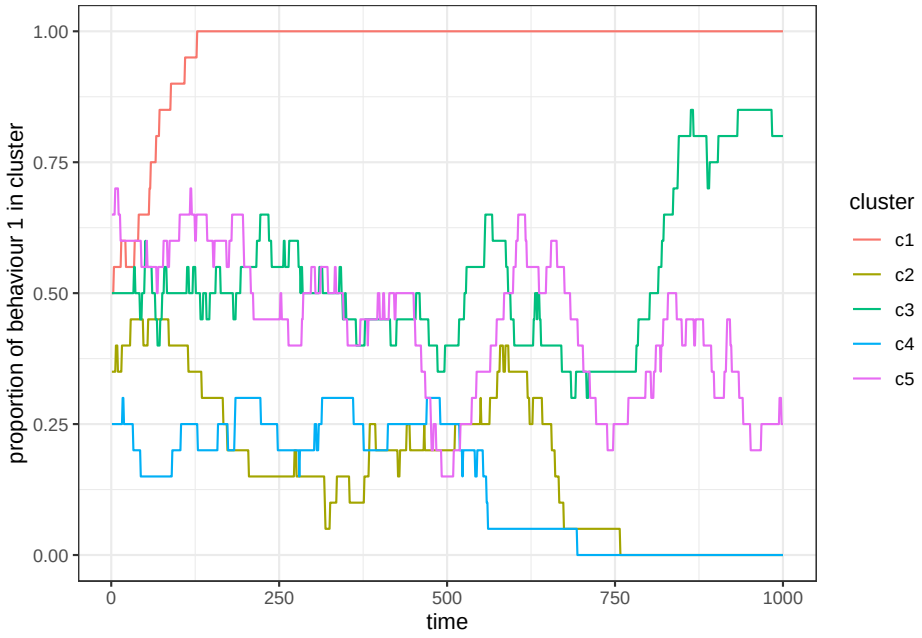


Figure 15.5: Simulation with no contact, $p_c = 0$.

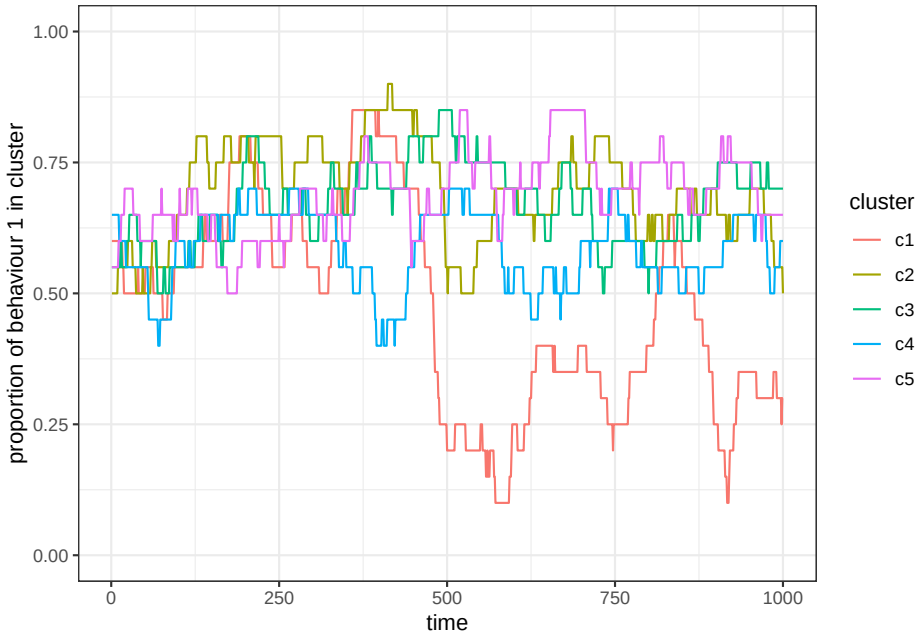


Figure 15.6: Simulation with some contact, $p_c = 0.1$.

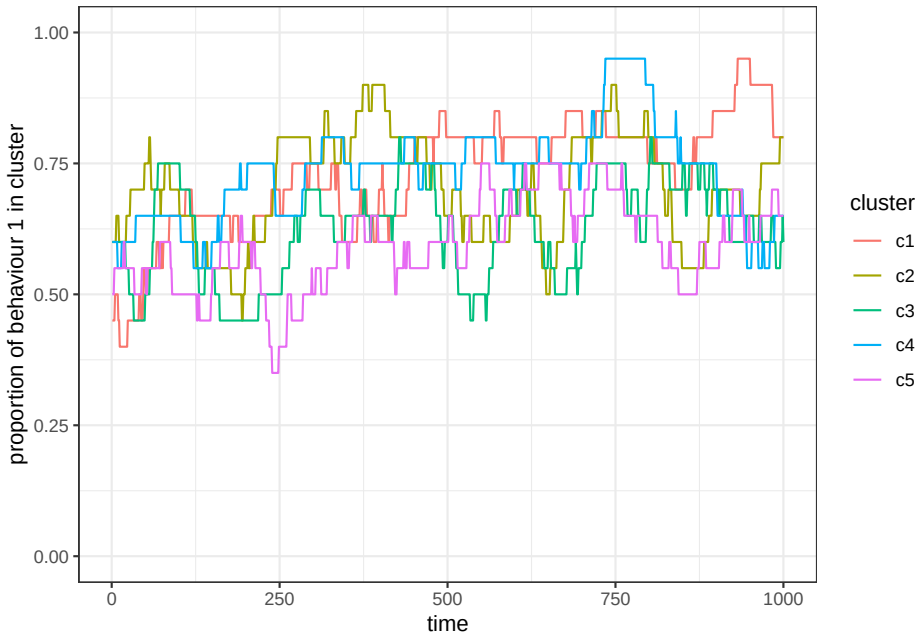


Figure 15.7: Simulation with full contact, $p_c = 1$.

stochastically. As we increase p_c the subsets become more correlated in the proportion of behaviour 1. For $p_c = 1$ we recover a population without subsets.

15.3 Subdivided populations with migration

In the previous section we have modelled the movement of cultural traits between subsets due to occasional interactions. Let us now simulate the movement of individuals (and their cultural trait) between subsets. To model migration, we will add a migration probability p_m to the model (with no migration where $p - m = 0$, and always migrating to a random subset where $p_m = 1$):

```
structured_population_5 <- function(n, c, p_c, p_m, r_max){
  total_pop <- c * n
  cluster <- rep(1:c, each = n)
  behaviours <- sample(x = 2, size = total_pop, replace = TRUE)
  rec_behav <- matrix(NA, nrow = r_max, ncol = c)

  for(round in 1:r_max){
    cluster_id <- sample(c, 1)
    s <- cluster == cluster_id
    if(sum(s)>1){
      observer_model <- sample(x = total_pop, size = 2, replace = F,
```

```
                                prob = (s + p_c) / (1 + p_c))
      behaviours[ observer_model[1] ] <- behaviours[ observer_model[2] ]
    }

    # Migration to another cluster with probability p_m and if there is more than
    # one subset
    if((runif(1,0,1) <= p_m) & (c > 1)){
      # Set cluster id that is different from the current one
      cluster[ observer_model[1] ] <- sample((1:c)[-cluster_id], 1)
    }

    for(clu in 1:c){
      rec_behav[round, clu] <- sum(behaviours[cluster == clu] == 1) / sum(cluster == clu)
    }
  }
  return(matrix_to_tibble(m = rec_behav))
}
```

The migration code chunk is doing two things. First, we make sure that migration only happens with the migration probability p_m (for that, we compare a random value from a uniform distribution with `p_m`: `runif(1,0,1) <= p_m`) and only if the population is actually subset (`c>1`). And second, if the statement is `TRUE`, we choose one new cluster ID among all cluster IDs but without the current one (`sample((1:c)[-cluster_id], 1)`).

Let us run the simulation for three different migration probabilities, $p_m \in \{0, 0.1, 1\}$. Let us also run the simulations much longer so that we will get to a case where a behaviour might get fixed in the subsets or the population:

```
res_0  <- structured_population_5(c = 5, n = 50, r_max = 10000, p_m = 0, p_c=0)
res_1  <- structured_population_5(c = 5, n = 50, r_max = 10000, p_m = 1, p_c=0)
res_01 <- structured_population_5(c = 5, n = 50, r_max = 10000, p_m = 0.1, p_c=0)

ggplot(res_0) +
  geom_line(aes(x = time, y = p, col = cluster)) +
  scale_y_continuous(limits = c(0,1)) +
  ylab("relative frequency of behaviour 1") +
  theme_bw()
```

For $p_m = 0$ we find that the subsets act independently and fix either on behaviour 1 or 2.

```
ggplot(res_1) +
  geom_line(aes(x = time, y = p, col = cluster)) +
  scale_y_continuous(limits = c(0,1)) +
  ylab("relative frequency of behaviour 1") +
  theme_bw()
```

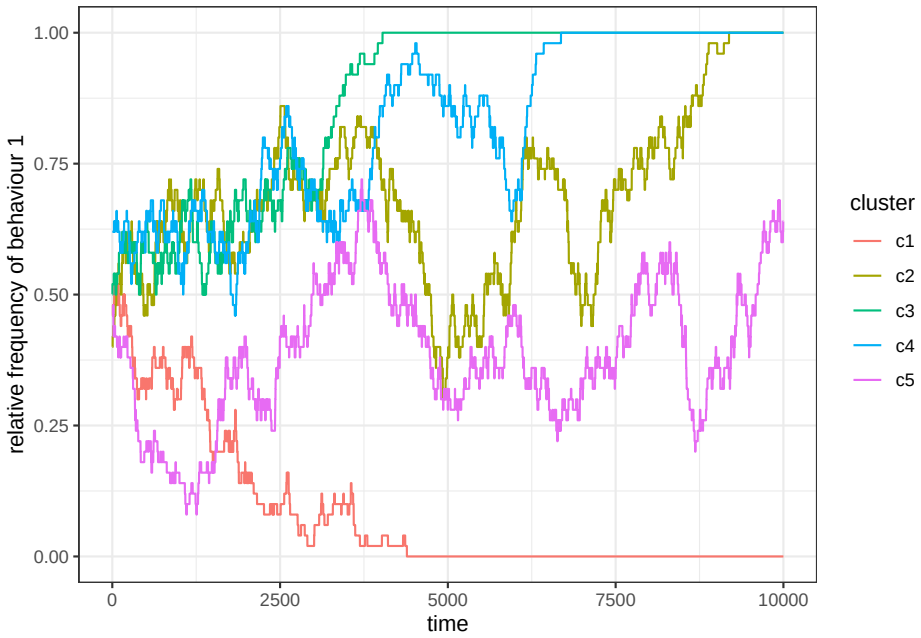


Figure 15.8: Without migration between clusters, there is no learning outside a subset, and so subsets act as independent populations.

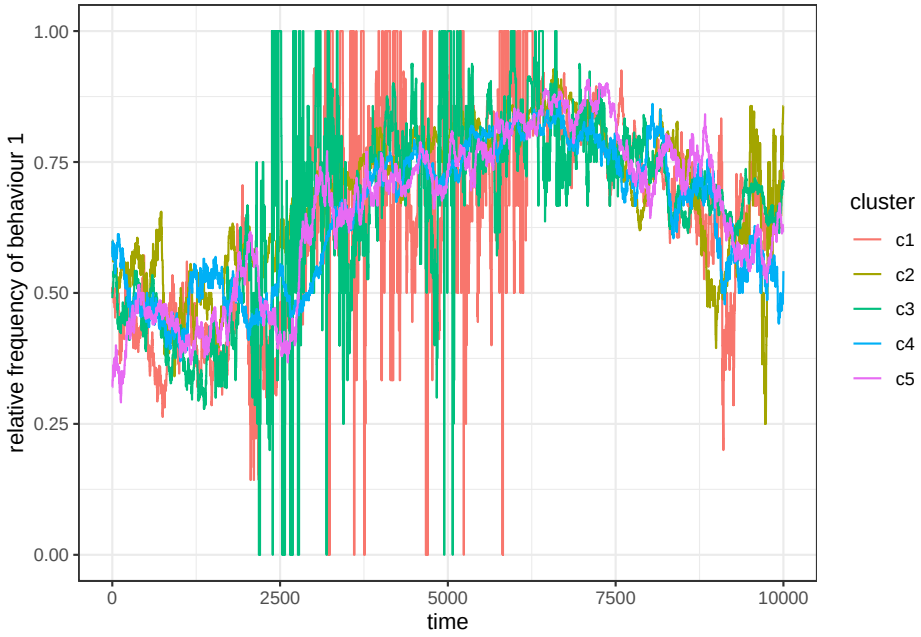


Figure 15.9: When $p_m = 1$ the subsets act again as a single population.

For $p_m = 1$, we find that the frequency of the behaviours become correlated as more and more individuals keep moving between the clusters. Eventually, all subsets will settle on the same behaviour. You might have also noticed that sometimes the curve for one or more subsets jump between 0 and 1. This is the case when a subset is almost empty. Imagine a subset with only one individual with behaviour 1, then $p = 1$. If this individual leaves the subset the curve jumps to $p = 0$. We are less likely to observe empty subsets if we let them start out bigger (say $N = 100$).

```
ggplot(res_01) +
  geom_line(aes(x = time, y = p, col = cluster)) +
  scale_y_continuous(limits = c(0,1)) +
  ylab("relative frequency of behaviour 1") +
  theme_bw()
```

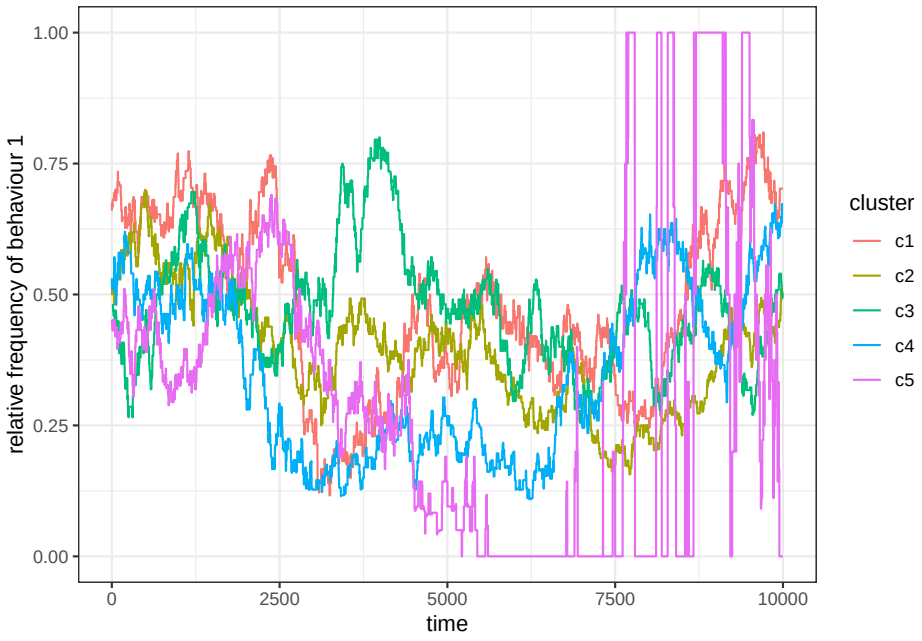


Figure 15.10: When migration is rare, the frequency of behaviour 1 changes occassionally but quickly bounes back to the original value in the subset.

For rare migration ($p_m = 0.1$), we sometimes find the population to fix on one behaviour, on two, or neither.

15.4 Varying contact and migration probability for repeated simulation runs

We have seen that both the migration and the contact probability affect the distribution of behaviour 1 and 2 in the population. In this last section let us run the model for different pairs of p_c and p_m to more systematically analyse the individual and combined effect of contact and migration probability. To do this, let us change our simulation function so that it returns a measure for how similar (or different) the proportion of behaviour 1 is among the subsets. We could, for example, calculate the variance as a measure for the variability between subsets (using `var(rec_behav)`). In this case, we do not need to store the frequency of behaviour 1 in each subset and for each round (in `rec_behav`), instead we only need to calculate it for the current round to calculate the variance:

```
structured_population_6 <- function(n, c, p_c, p_m, r_max, sim = 1){
  total_pop <- c * n
  cluster <- rep(1:c, each = n)
  behaviours <- sample(x = 2, size = total_pop, replace = TRUE)
  rec_behav <- rep(NA, times = c)
  # Adding a reporting variable for the similarity of clusters
  rec_var <- rep(NA, r_max)

  for(round in 1:r_max){
    cluster_id <- sample(c, 1)
    s <- cluster == cluster_id
    if(sum(s)>1){
      observer_model <- sample(x = total_pop, size = 2, replace = F,
                               prob = (s + p_c) / (1 + p_c))
      behaviours[ observer_model[1] ] <- behaviours[ observer_model[2] ]
    }

    if((runif(1,0,1) <= p_m) & (c > 1)){
      cluster[ observer_model[1] ] <- sample((1:c)[-cluster_id], 1)
    }

    for(clu in 1:c){
      rec_behav[clu] <- sum(behaviours[cluster == clu] == 1) / sum(cluster == clu)
    }
    # Calculating variance in behaviour 1 between clusters
    rec_var[round] <- var(rec_behav)
  }
  # Preparing a reporting table to return
  rec_var <- bind_cols(time = 1:r_max, var = rec_var, sim = sim, p_c = p_c, p_m = p_m)
  return(rec_var)
}
```

Now let us run this simulation for different values of p_m and p_c. Also, let us repeat each set of parameters 20 times. As in previous chapters, we set up a table that contains all the individual simulations that we want to run. The `expand.grid()` function creates a `data.frame` with all possible combinations of our parameters. We will use $p_m = \{0, 0.01, 0.1, 1\}$ and $p_c = \{0, 0.01, 0.1, 1\}$. So, this would result in a combination matrix of $4 \times 4 = 16$ simulations. Additionally we add `rep` as a counter of our repetitions (here, `1:20`), and so we receive $4 \times 4 \times 20 = 320$ individual simulation runs.

```
grid <- expand.grid(rep = 1:20,
                    pm = c(0, .01, .1, 1),
                    pc = c(0, .01, .1, 1))
head(grid)
```

```
##   rep pm pc
## 1   1  0  0
## 2   2  0  0
## 3   3  0  0
## 4   4  0  0
## 5   5  0  0
## 6   6  0  0
```

```
dim(grid)
```

```
## [1] 320   3
```

We will again use the `lapply()` function to run many independent simulations in parallel. We combine the `list` element that `lapply()` returns into a single object by binding the rows of each result together, by wrapping `bind_rows()` around the function. The `lapply()` function will execute `structured_population_6()` with fixed arguments for the number of subsets (`c`), subset size (`n`), and number of rounds (`r_max`). Arguments `p_m` and `p_c` are selected from the `grid` table that we just created. To get the right parameters for the right simulation run, `lapply()` is handing over a variable that we called `i` (this is an arbitrary name and you could also choose any other name here). `i` is an element of the data that we handed over, here `1:nrow(grid)`, i.e. values from 1:20 (note, we could have also used `1:20` but should you ever change the number of repetitions in your grid object, you would have to also make this change manually in the `lapply()` function, using `nrow(grid)` is taking care of this automatically). We will also use `i` as our `sim` argument, which will help us later to tell individual simulations apart.

```
res <- bind_rows(lapply(1:nrow(grid), function(i)
  structured_population_6(c = 5,
                          n = 20,
                          r_max = 2000,
                          p_m = grid[i, "pm"],
                          p_c = grid[i, "pc"],
```

```
                          sim = i)
  ))
res
```

```
## # A tibble: 640,000 x 5
##     time     var   sim   p_c   p_m
##    <int>   <dbl> <int> <dbl> <dbl>
##  1     1 0.00675     1     0     0
##  2     2 0.00575     1     0     0
##  3     3 0.00575     1     0     0
##  4     4 0.00675     1     0     0
##  5     5 0.00700     1     0     0
##  6     6 0.00575     1     0     0
##  7     7 0.00425     1     0     0
##  8     8 0.0075      1     0     0
##  9     9 0.0075      1     0     0
## 10    10 0.008       1     0     0
## # ... with 639,990 more rows
```

Now, let us plot how variance changes over time for each simulation. We could just use `geom_line(aes(x = time, y = var, group = sim), alpha=.5)` to plot the individual simulation runs (not that we have grouped the results by their `sim` indicator). However, the results for the different p_m and p_c values would be all in the same plot. Ggplot2 allows us to quickly separate the simulations based on these two values with the `facet_grid()` function. And so, we will add `facet_grid(p_m ~ p_c, labeller = label_both)` which will separate the results based on p_m and p_c. The `labeller` argument will add the name of the variable to the columns and rows of the plotted grid (without this argument the column and row titles would only show the numeric values). This makes it easier to identify the pairs of parameters for each plot:

```
ggplot(res) +
  geom_line(aes(x = time, y = var, group = sim), alpha=.5) +
  facet_grid(p_m ~ p_c, labeller = label_both) +
  ylab("average variance between subsets") +
  theme_bw()
```

In accordance with our previous results in the absence of migration and contact the variance is highest, whereas it is lowest when both p_m and p_c are 1. However, we also see that variance drops faster as p_c increases compared to the same increase in p_m (compare $p_c = 0.1, p_m = 0$ and $p_c = 0, p_m = 0.1$).

Sometimes we will not have the space to plot a grid like this. In that case, we could condense the results further down by averaging, say, the last 20% of the simulation rounds over all simulations and then plot a single value for each parameter pair. A good visualisation for this is the raster, which ggplot provides with the `geom_raster()` function.

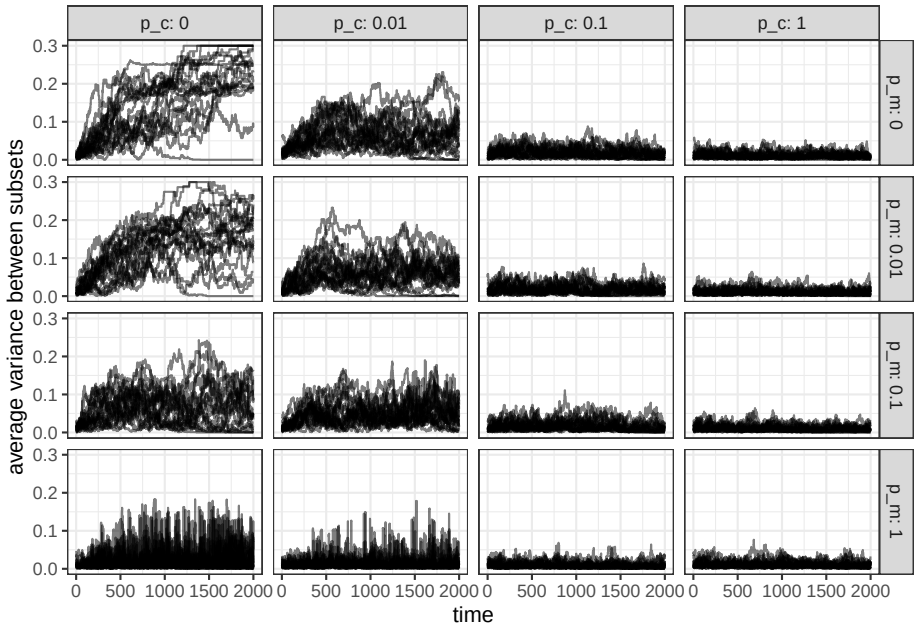


Figure 15.11: In the absence of contact and migration ($p_m = p_c = 0$) the variance is highest between subsets. The variance decreases more quickly as p_c increases compared to p_m.

But first, we need to summarise our data using the following `tidyverse` functions: we use `filter()` to select only the rows of the last 20% of our simulation turns, handing this over (`%>%`) to the `group_by()` function, which will create groups where `p_m` and `p_c` are identical, and then `summarise()`, which will add a new column with the average `var` values from our results object. We will store the sumamrised result in `res_summ`:

```
res_summ <- filter(res, time >= (max(time)*.8)) %>%
              group_by(p_m, p_c) %>%
                summarise(mean_var = mean(var), .groups = "keep")
```

We can now use `res_summ` to plot our raster where we set the z-value (the colour of each raster rectangle) using the `fill` argument:

```
ggplot(res_summ) +
  geom_raster(aes(x = factor(p_m), y = factor(p_c), fill = mean_var)) +
  xlab("p_m") + ylab("p_c") +
  theme_bw() +
  theme(legend.title = element_blank())
```

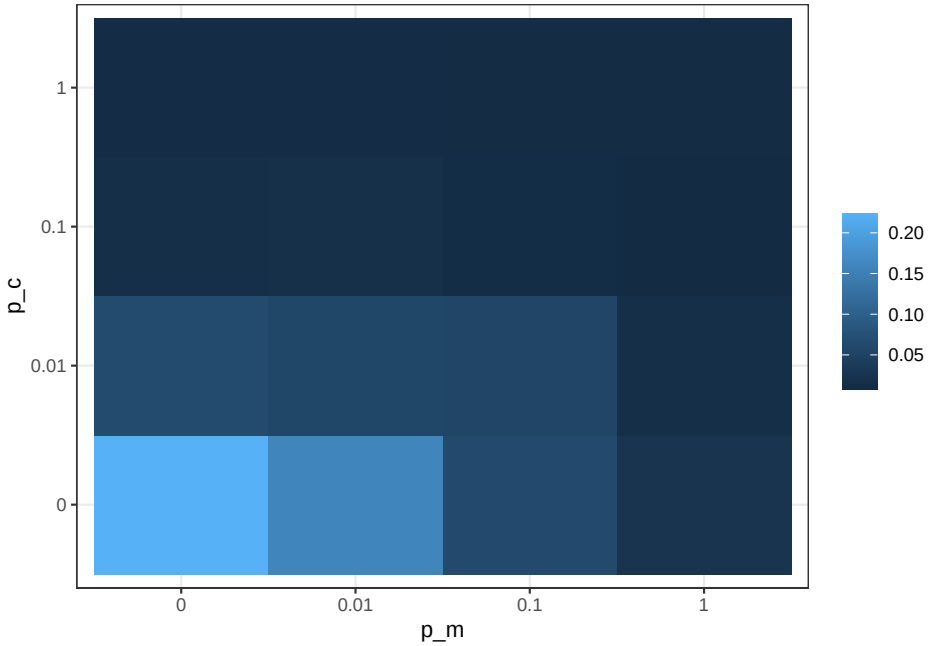


Figure 15.12: Summarised simulation results showing the effect of contact and migration on the distribution of behaviour 1 among subsets of a population.

Our results show that contact and migration are not interchangeable or symmetric in their behaviour. With this model, we could now ask many more interesting questions, for example, how migration and contact probability affect

trait distribution in populations with unevenly sized subsets, or how the number of clusters, behaviours, or population size affects the results. Below, we have a selection of further model extensions to consider.

15.5 Model extensions

Innovation or mutation

An interesting extension to this model is the addition of innovation or mutation. For example, individuals could invent completely new behaviours (in this case we would not work with a fixed behaviour number b), or with a certain probability, an individual might try to copy behaviour 2 but acquires behaviour 1 (akin to mutation). These are all mechanisms that would add diversity to the model.

Copy m models

So far, an individual is changing its behaviour based on observing one other individual. However, instead of choosing from a single model, we can change the code such that the individual is considering the behaviours of m other individuals. As we have seen in previous chapters, as the difference between n and m becomes smaller the more the dynamics will look like frequency biased copying.

To achieve this behaviour we can change the `observer_model` part of our model to:

```
if(sum(cluster == cluster_id) > m){
  observer_model <- sample(x = total_pop, size = m + 1, replace = FALSE,
                           prob = (cluster == cluster_id) * 1)
  behaviours[ observer_model[1] ] <-
    behaviours[ sample(observer_model[2:(m + 1)], size = 1) ]
}
```

where m is the number of models to observe. (Note, due to the peculiarities of the `sample()` function this code only works for $m > 1$).

Variable migration probability among subsets

Finally, this model can be extended to accommodate different population structures. In this chapter, we have only looked at symmetric connections between subsets (all subsets are connected and migration between them is equally likely). But the structure could also be a line, a circle, a star, and others, where not all subsets are connected (missing links) or where migration probability is low (using weighted connections). This can be useful to generally better understand how population structure will affect transmission. But it can also be used to model specific scenarios if there is existing data on population structure.

For this iteration of the base model, we need to change the `migration` section. Instead of choosing randomly among other clusters, we would provide a probability vector to the `sample()` function that reflects the probabilities to move from one to another subset. As an example, let us assume we are looking at three subsets $\{A, B, C\}$. A simple structure is a line, where A is connected with B, and B is connected with C, or `A--B, B--C`. We can use an adjacency matrix to describe the probability to move from one subset to another:

```
adj <- matrix(c(0,1,0,  1,1,1,  0,1,0), nrow = 3)
adj
```

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

If rows are the starting and columns the end subsets, then this matrix tells us that an individual in subset A (first row) can move to B (second column entry is 1) but not to C (last column entry is 0). Now, when we determine that an individual is moving to a different group, we can simply recall the correct row of the `adj` matrix based on the individual's `cluster_id`:

```
if((runif(1,0,1) <= p_m) & (c > 1)){
  cluster[ observer_model[1] ] <- sample((1:c), size = 1, prob = adj[cluster_id,])
}
```

Also, this piece of code allows us to use non-binary values, where small values represent a low probability to move from one to another subset, and asymmetric matrices where the probability going from A to B can be different from the probability for the reverse movement.

15.6 Summary of the model

In this chapter, we have used a simple model to simulate the effect of population sub-structuring. We have seen how contact (movement of cultural traits) and migration (movement of individuals) affect the frequency of a behaviour in each subset. When migration and contact probability are low, the frequency of individual behaviours become uncorrelated between subsets. However, as both parameters increase, subsets behave more and more like a single large population.

15.7 Further reading

There are a couple of interesting empirical studies on migration and the social transmission of locally adaptive behaviours in animals. For example, the study by Luncz and Boesch [2014] reports on the stability of tool traditions in neighbouring chimpanzee communities.

There are also a few theoretical studies on the persistence or change of local traditions. Boyd and Richerson [2009], for example, focus on how adaptive a behaviour is, whereas Mesoudi [2018] focuses on the strength of acculturation that is required to maintain cultural diversity between groups.

References

Bibliography

Alberto Acerbi. *Cultural Evolution in the Digital Age*. Oxford University Press, Oxford, New York, December 2019. ISBN 978-0-19-883594-3.

Alberto Acerbi and Alex Mesoudi. If we are all cultural Darwinians what's the fuss about? Clarifying recent disagreements in the field of cultural evolution. *Biology & Philosophy*, 30(4):481–503, July 2015. ISSN 1572-8404. doi: 10.1007/s10539-015-9490-2. URL https://doi.org/10.1007/s10539-015-9490-2.

Alberto Acerbi, Magnus Enquist, and Stefano Ghirlanda. Cultural evolution and individual development of openness and conservatism. *Proceedings of the National Academy of Sciences*, 106(45):18931–18935, November 2009. ISSN 0027-8424, 1091-6490. doi: 10.1073/pnas.0908889106. URL https://www.pnas.org/content/106/45/18931.

Alberto Acerbi, Stefano Ghirlanda, and Magnus Enquist. Regulatory Traits: Cultural Influences on Cultural Evolution. In Stefano Cagnoni, Marco Mirolli, and Marco Villani, editors, *Evolution, Complexity and Artificial Life*, pages 135–147. Springer, Berlin, Heidelberg, 2014. ISBN 978-3-642-37577-4. doi: 10.1007/978-3-642-37577-4_9. URL https://doi.org/10.1007/978-3-642-37577-4_9.

Alberto Acerbi, Mathieu Charbonneau, Helena Miton, and Thom Scott-Phillips. Cultural stability without copying. preprint, Open Science Framework, March 2019. URL https://osf.io/vjcq3.

L. M. Aplin, D. R. Farine, J. Morand-Ferron, and B. C. Sheldon. Social networks predict patch discovery in a wild population of songbirds. *Proceedings of the Royal Society B: Biological Sciences*, 279(1745):4199–4205, October 2012. doi: 10.1098/rspb.2012.1591. URL https://royalsocietypublishing.org/doi/full/10.1098/rspb.2012.1591. Publisher: Royal Society.

R. Alexander Bentley, Matthew W. Hahn, and Stephen J. Shennan. Random drift and culture change. *Proceedings of the Royal Society of London. Series B: Biological Sciences*, 271(1547):1443–1450, July 2004. doi: 10.1098/rspb.2004.2746. URL https://royalsocietypublishing.org/doi/abs/10.1098/rspb.2004.2746.

Robert Boyd and Peter J. Richerson. *Culture and the evolutionary process*. Culture and the evolutionary process. University of Chicago Press, Chicago, IL, US, 1985. ISBN 978-0-226-06931-9 978-0-226-06933-3.

Robert Boyd and Peter J. Richerson. Why does culture increase human adaptability? *Ethology and Sociobiology*, 16(2):125–143, March 1995. ISSN 0162-3095. doi: 10.1016/0162-3095(94)00073-G. URL http://www.sciencedirect.com/science/article/pii/016230959400073G.

Robert Boyd and Peter J. Richerson. Voting with your feet: Payoff biased migration and the evolution of group beneficial behavior. *Journal of Theoretical Biology*, 257(2):331–339, March 2009. ISSN 0022-5193. doi: 10.1016/j.jtbi.2008.12.007. URL http://www.sciencedirect.com/science/article/pii/S0022519308006371.

Andrew Buskell, Magnus Enquist, and Fredrik Jansson. A systems approach to cultural evolution. *Palgrave Communications*, 5:131, 2019. URL https://www.nature.com/articles/s41599-019-0343-5.

Luigi Luca Cavalli-Sforza and Marcus W. Feldman. *Cultural transmission and evolution*. Princeton Univ. Press, Princeton, 1981.

Damon Centola. The Spread of Behavior in an Online Social Network Experiment. *Science*, 329(5996):1194–1197, September 2010. ISSN 0036-8075, 1095-9203. doi: 10.1126/science.1185231. URL https://science.sciencemag.org/content/329/5996/1194. Publisher: American Association for the Advancement of Science Section: Report.

Maxime Derex and Alex Mesoudi. Cumulative Cultural Evolution within Evolving Population Structures. *Trends in Cognitive Sciences*, 24(8):654–667, August 2020. ISSN 1364-6613. doi: 10.1016/j.tics.2020.04.005. URL http://www.sciencedirect.com/science/article/pii/S1364661320301078.

C. Efferson, Rafael Lalive, Peter J. Richerson, R. McElreath, and Mark Lubell. Conformists and mavericks: the empirics of frequency-dependent cultural transmission. *Evolution and Human Behavior*, 29(1):56–64, 2008. doi: 10.1016/j.evolhumbehav.2007.08.003.

Magnus Enquist, Kimmo Eriksson, and Stefano Ghirlanda. Critical Social Learning: A Solution to Rogers's Paradox of Nonadaptive Culture. *American Anthropologist*, 109(4):727–734, 2007. Type: Journal Article.

Magnus Enquist, Stefano Ghirlanda, and Kimmo Eriksson. Modelling the evolution and diversity of cumulative culture. *Philosophical Transactions of the Royal Society B: Biological Sciences*, 366(1563):412–423, February 2011. doi: 10.1098/rstb.2010.0132. URL https://royalsocietypublishing.org/doi/full/10.1098/rstb.2010.0132.

Stefano Ghirlanda, Magnus Enquist, and Mayuko Nakamaru. Cultural Evolution Develops Its Own Rules: The Rise of Conservatism and Persua-

sion. *Current Anthropology*, 47(6):1027–1034, 2006. ISSN 0011-3204. doi: 10.1086/508696. URL https://www.jstor.org/stable/10.1086/508696.

Stefano Ghirlanda, Magnus Enquist, and Matjaž Perc. Sustainability of culture-driven population dynamics. *Theoretical Population Biology*, 77(3):181–188, May 2010. ISSN 0040-5809. doi: 10.1016/j.tpb.2010.01.004. URL http://www.sciencedirect.com/science/article/pii/S0040580910000110.

Garrett Grolemund. *Hands-On Programming with R*. Sebastopol, CA, 2014. URL https://rstudio-education.github.io/hopr/.

Joseph Henrich. Cultural transmission and the diffusion of innovations. *American Anthropologist*, 103(4):992–1013, 2001.

Joseph Henrich. Demography and Cultural Evolution: How Adaptive Cultural Processes can Produce Maladaptive Losses: The Tasmanian Case. *American Antiquity*, 69(2):197–214, 2004. ISSN 0002-7316. doi: 10.2307/4128416. URL https://www.jstor.org/stable/4128416.

Joseph Henrich. *The Secret of Our Success*. Princeton University Press, Princeton, NJ, 2016. Type: Book.

Joseph Henrich and Robert Boyd. The evolution of conformist transmission and the emergence of between-group differences. *Evolution and Human Behavior*, 19(4):215–241, 1998. doi: 10.1016/S1090-5138(98)00018-X.

Joseph Henrich and Francisco J Gil-White. The evolution of prestige: freely conferred deference as a mechanism for enhancing the benefits of cultural transmission. *Evolution and Human Behavior*, 22(3):165–196, May 2001. ISSN 1090-5138. doi: 10.1016/S1090-5138(00)00071-4. URL http://www.sciencedirect.com/science/article/pii/S1090513800000714.

Henrich Joseph, Chudek Maciej, and Boyd Robert. The Big Man Mechanism: how prestige fosters cooperation and creates prosocial leaders. *Philosophical Transactions of the Royal Society B: Biological Sciences*, 370(1683):20150013, December 2015. doi: 10.1098/rstb.2015.0013. URL https://royalsocietypublishing.org/doi/full/10.1098/rstb.2015.0013.

Cecilia Heyes. What's social about social learning? *Journal of Comparative Psychology*, 126(2):193–202, 2012. ISSN 1939-2087, 0735-7036. doi: 10.1037/a0025180. URL http://doi.apa.org/getdoi.cfm?doi=10.1037/a0025180.

Kim R. Hill, Brian M. Wood, Jacopo Baggio, A. Magdalena Hurtado, and Robert T. Boyd. Hunter-Gatherer Inter-Band Interaction Rates: Implications for Cumulative Culture. *PLOS ONE*, 9(7):e102806, July 2014. ISSN 1932-6203. doi: 10.1371/journal.pone.0102806. URL https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0102806. Publisher: Public Library of Science.

Ángel V. Jiménez and Alex Mesoudi. Prestige-biased social learning: current evidence and outstanding questions. *Palgrave Communications*, 5(1):1–12,

February 2019. ISSN 2055-1045. doi: 10.1057/s41599-019-0228-7. URL https://www.nature.com/articles/s41599-019-0228-7.

Kevin N. Laland. *Darwin's Unfinished Symphony: How Culture Made the Human Mind*. Princeton University Press, Princeton, NJ, 2017. Type: Book.

J. Stephen Lansing and Murray P. Cox. The domain of the replicators. *Current Anthropology*, 52(1):105–125, February 2011. ISSN 0011-3204. doi: 10.1086/657643. URL https://www.journals.uchicago.edu/doi/abs/10.1086/657643.

Lydia V. Luncz and Christophe Boesch. Tradition over trend: Neighboring chimpanzee communities maintain differences in cultural behavior despite frequent immigration of adult females. *American Journal of Primatology*, 76(7): 649–657, July 2014. ISSN 1098-2345. doi: 10.1002/ajp.22259.

Alex Mesoudi. The Cultural Dynamics of Copycat Suicide. *PLOS ONE*, 4(9): e7252, September 2009. ISSN 1932-6203. doi: 10.1371/journal.pone.0007252. URL https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0007252.

Alex Mesoudi. *Cultural Evolution: How Darwinian Theory Can Explain Human Culture and Synthesize the Social Sciences*. University of Chicago Press, Chicago, 2011. Type: Book.

Alex Mesoudi. Migration, acculturation, and the maintenance of between-group cultural variation. *PLOS ONE*, 13(10):e0205573, October 2018. ISSN 1932-6203. doi: 10.1371/journal.pone.0205573. URL https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0205573. Publisher: Public Library of Science.

Alex Mesoudi and Stephen J. Lycett. Random copying, frequency-dependent copying and culture change. *Evolution and Human Behavior*, 30(1):41–48, January 2009. ISSN 1090-5138. doi: 10.1016/j.evolhumbehav.2008.07.005. URL http://www.sciencedirect.com/science/article/pii/S1090513808000810.

Alex Mesoudi, Lei Chang, Sasha R. X. Dall, and Alex Thornton. The Evolution of Individual and Cultural Variation in Social Learning. *Trends in Ecology & Evolution*, 31(3):215–225, March 2016. ISSN 0169-5347. doi: 10.1016/j.tree.2015.12.012. URL http://www.sciencedirect.com/science/article/pii/S0169534715003237.

Olivier Morin. *How Traditions Live and Die*. Oxford University Press, London & New York, November 2015. ISBN 978-0-19-021050-2.

Fraser D. Neiman. Stylistic Variation in Evolutionary Perspective: Inferences from Decorative Diversity and Interassemblage Distance in Illinois Woodland Ceramic Assemblages. *American Antiquity*, 60(1):7–36, January 1995. ISSN 0002-7316, 2325-5064. doi: 10.2307/282074. URL https://www.cambridge.org/core/journals/american-antiquity/article/stylistic-variation-in-evolutionary-perspective-inferences-from-decorative-diversity-and-

interassemblage-distance-in-illinois-woodland-ceramic-assemblages/69A948F595717901C75F84D033F1A1C9.

Daniel Nettle. Selection, adaptation, inheritance and design in human culture: The view from the Price equation. *Philosophical Transactions of the Royal Society B*, 375:20190358, 2020.

David J. P. O'Sullivan, Gary James O'Keeffe, Peter G. Fennell, and James P. Gleeson. Mathematical modeling of complex contagion on clustered networks. *Frontiers in Physics*, 3, 2015. ISSN 2296-424X. doi: 10.3389/fphy.2015.00071. URL https://www.frontiersin.org/articles/10.3389/fphy.2015.00071/full. Publisher: Frontiers.

Adam Powell, Stephen Shennan, and Mark Thomas. Late Pleistocene Demography and the Appearance of Modern Human Behavior. *Science*, 324:1298–1301, 2009. Type: Journal Article.

S. M. Reader. Distinguishing social and asocial learning using diffusion dynamics. *Learning and Behavior*, 32:90–104, 2004.

Alan R. Rogers. Does Biology Constrain Culture? *American Anthropologist*, 90(4):819–831, 1988. ISSN 1548-1433. doi: 10.1525/aa.1988.90.4.02a00030. URL https://anthrosource.onlinelibrary.wiley.com/doi/abs/10.1525/aa.1988.90.4.02a00030.

Stephen Shennan. Demography and Cultural Innovation: a Model and its Implications for the Emergence of Modern Human Culture. *Cambridge Archaeological Journal*, 11(1):5–16, 2001. Type: Journal Article.

Stephen Shennan. Demography and Cultural Evolution. In *Emerging Trends in the Social and Behavioral Sciences*, pages 1–14. American Cancer Society, 2015. ISBN 978-1-118-90077-2. doi: 10.1002/9781118900772.etrds0073. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/9781118900772.etrds0073. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/9781118900772.etrds0073.

Paul E. Smaldino. Models Are Stupid, and We Need More of Them, May 2017. URL https://www.taylorfrancis.com/. Pages: 311-331 Publisher: Routledge.

Paul E. Smaldino. How to translate a verbal theory into a formal model. *Social Psychology*, 51(4):207–218, 2020. ISSN 2151-2590(Electronic),1864-9335(Print). doi: 10.1027/1864-9335/a000425. Place: Germany Publisher: Hogrefe Publishing.

Marco Smolla and Erol Akçay. Cultural selection shapes network structure. *Science Advances*, 5(8):eaaw0609, August 2019. ISSN 2375-2548. doi: 10.1126/sciadv.aaw0609. URL https://advances.sciencemag.org/content/5/8/eaaw0609. Publisher: American Association for the Advancement of Science Section: Research Article.

Dan Sperber. Selection and Attraction in Cultural Evolution. In *Explaining Culture. A naturalistic approach*, pages 409–426. Blackwell, Oxford, 1997.

Krist Vaesen, Mark Collard, Richard Cosgrove, and Wil Roebroeks. Population size does not explain past changes in cultural complexity. *Proceedings of the National Academy of Sciences of the United States of America*, 113(16): E2241–2247, April 2016. ISSN 1091-6490. doi: 10.1073/pnas.1520288113.

Hadley Wickham and Garrett Grolemund. *R for Data Science: Import, Tidy, Transform, Visualize, and Model Data*. O'Reilly Media, Sebastopol, CA, 1st edition edition, January 2017. ISBN 978-1-4919-1039-9.