Lab 1 — Simulated evolution

Evolution of Language and Music

October 22, 2018

Goals In today's computer lab you will experiment with simulated evolution and look at a simple model of the evolution of communication. The goals of this lab are:

- to better understand the concepts of genotype, genotype space, fitness, fitness landscape, selection, mutation, selection-mutation balance, frequency dependent selection;
- to see how these concepts can be formalised in a computer program;
- to appreciate both the power and the limits of natural selection.

1 Simulated Evolution

In the first part of this computer lab, we will use the programming language R to simulate the evolution of a (DNA) string under a particular fitness function. First, we will launch the interface for this programming language, and load the required packages:

- Start R or R-studio (depending on your operating system and preferences).
- Install the package stringr by typing `install.packages("stringr")`. As you (probably) have no rights to install the package globally, the computer will ask you if you want to install the package in a personal library, click okay and accept the default settings. If nothing seems to happen, it could be that the pop-up window appeared below another window.
- Load the library `stringr` by typing `library(stringr)` in the console.

We will represent DNA strings (the *genotype*) as a sequence of the letters 'A', 'G', 'C' and 'U'. In R, you can generate a random sequence of 10 of these letters using the following command:

```
sample(c('A','G','C','U'), size=10, replace=TRUE)
```

You can store the output under a name (for instance x), you can type:

```
x <- sample(c('A','G','C','U'), size=10, replace=TRUE)
```

You can then view the contents of a particular object by simply typing its name and pressing enter. Let's experiment a bit with this.

- Using the commands you just learned, generate a few random sequences of length 10 containing the characters A,G,C and U to confirm that it does what we want it to do. (hint: use the up-arrow key to scroll to previous commands).
- Generate a random sequence of length 50 containing the characters 'A', 'G', 'C' and 'U'.
- ⋆ The set of all possible sequences is called the *genotype space*. How big is this space? I.e., how many genotype strings are possible with our representation?

Now, let's create a *population* of DNA strings. To do this, we will make 100 genotype strings. We will store our population in a matrix (the *population matrix*), where each member of our population is represented as a row of the matrix.

- Let's start with creating a matrix filled with zero's that we can later fill:[a]

```
population <- matrix(rep(0, 100), 100, 50)}
```

- What do we find in column 30 of our population matrix?
- Fill your matrix by generating 100 population members in a for-loop and filling the matrix with them:[b]

```
population_size <- 100
for (i in 1:population_size) {
    population[i,] <- sample(c('A','G','C','U'),
      size=50, replace=TRUE)
}
```

Now we need to define a fitness function that computes the fitness of the individual members of our population. Imagine, for instance, that the string 'CAC' codes for some very useful aminoacid. The more CAC's in the genome, the higher the expected number of offspring. In our simulation of evolution, let's define the fitness as the number of times the substring 'CAC' appears in the genotype string.

To keep track of the fitness of *all* members in our population (which are represented as rows in the population matrix), we create a *vector* containing where each element of the vector represents the fitness value for one member of the population.

- Generate an empty vector to store the fitness values, and call it `fitness`:

```
fitness <- rep(0, population_size)
```

- Use a for-loop to fill the vector (created by the code above) with the fitness values:

[a] The command `matrix(x, height, width)` command transforms a vector `x` into a matrix with height `height` and width `width`.

[b] `x[i,]` accesses the ith row of the matrix `x`, which in our case thus corresponds with the ith member of our population

```
# loop over population size
for (i in 1:100) {
  # generate string representation
  member <- paste(population[i,], collapse='')

  # compute fitness member
  fitness_member <- str_count(member, "CAC")

  # store in fitness vector
  fitness[i] <- fitness_member
}
```

Note that R ignores everything that follows the character #. In programming terms, these texts are called *comments*.

? What is the highest possible fitness a member of this population can have?

- Compute the mean fitness of your population by using `mean(fitness)`.

? What is the average fitness of your population?

Now we will generate the next generation. We assume that each member of the next generation inherits the genome of one of the members of the previous generation. The probability of inheriting each genome is proportional to the genome's fitness: a child is most likely to inherit the genome of the fittest member of the previous population. This simulates selection.

- Compute the average fitness of the population and store it in a variable using:

```
av_fitness <- mean(fitness)
```

- Generate 100 new children, using the built-in function `sample` (the same one we used before):[a]

```
indices <- sample(100, size=100, replace=TRUE,
                  prob=fitness/sum(fitness))
new_population <- population[indices,]
```

? If one population member has fitness 10 and all the other population members have fitness 1, what is the probability that a child will inherit its genome from this one population member? What do you expect to happen with the population?

- To simulate the evolution of the population, we want to repeat this process several times and plot the average fitness over time. If you like programming, you can try to do the implementation yourself, but we also provided a script called `lab-2.R` that does the trick. The next bullet point contains some instructions for implementation, if you use the script you can thus skip them.
- To repeat the previous process 100 times, you should create a for-loop that executes the previous bits of code 100 times, storing the fitness of every population in a vector. You can plot your results using

```
plot(seq(1,100,1), av_fitness, type="l", ann=FALSE)
```

(Assuming you stored the fitness values in av_fitness).
To label the axes and title of the plot use:

```
title(main="title", xlab="x_label", ylab="y_label")
```

- To run a script in R, you type `source('scriptname')` in the command line. To be able to run a script, it should be available to the interpreter. You should make sure that both the file `lab-2.R` and the file `auxiliary_functions.R` are accessible. If you are working in R-studio, the easiest way to do this, is to put them both in the same folder and set this folder as the *working directory* of R-studio. Look at this website to find out how to do that for your version of R-studio. You can use tab for auto completion.
- ⋆ You will notice the fitness stops increasing quite early in the simulation. Why is this? (note that lab-2.R will create a new random population matrix)

In the previous simulation, we looked at selection *without* mutation. Let's now look at the case where every child's nucleotide has a probability μ to change into a random other nucleotide.

- ? If $\mu = 0.01$, what is the chance that no changes occur in a genome. What is the chance that no changes occur in an entire population? And if $\mu = 0.001$?
- Use the provided script to do the same simulation, but with a mutation level $\mu = 0.001$. You can chance the values of the parameters at the top of the script. After changing them, save the file and run the script again by typing `source('lab-2.R')`. Adapt the length of the simulation to a number you think is suitable.
- Now repeat the simulation with $\mu = 0.01$, plot the fitness. This shows the mutation-selection balance.
- ⋆ Why does the fitness with relatively high mutation rate level stop increasing a slightly lower level?

2 Evolution of communication

In the second part of this assignment, we will model the evolution of a communication system. A possible way of representing a communication system is by using matrices that describe a mapping from a set of meanings to a set of forms (or signals). For instance, the well known alarm call system of Vervet monkeys **seyfarth1980monkey** in its usual idealisation, can be described as follows:

[a] We first draw 100 random numbers between 1 and 100 (repetitions possible). If population member 2 has a very high fitness, it will have a very high chance of being drawn. Then we use the drawn numbers to create a new population of the members corresponding to the numbers.

$$S = \left(\begin{array}{c|ccc} & \text{chip} & \text{grunt} & \text{chutter} \\ \hline \text{leopard} & 0.8 & 0.2 & 0 \\ \text{eagle} & 0.1 & 0.9 & 0 \\ \text{snake} & 0.05 & 0.1 & 0.85 \end{array}\right)$$

$$R = \left(\begin{array}{c|ccc} & \text{leopard} & \text{eagle} & \text{snake} \\ \hline \text{chirp} & 0.9 & 0 & 0.1 \\ \text{grunt} & 0 & 1 & 0 \\ \text{chutter} & 0.2 & 0 & 8 \end{array}\right)$$

The S matrix represents the sender: the first column contains the meanings (or situations) that the sender may want to express, the first row the signals that it can use to express these meanings. The numbers in the matrix represent the probabilities that the sender will use a certain signal to express a certain meaning. The matrix R describes the behaviour of the receiver in a similar way: the numbers in the matrix are the probabilities that the receiver will interpret a certain signal (first column) as having a certain meaning (first row).

⋆ What are the optimal S and R, for maximal communicative success in a population?
⋆ How is ambiguity (i.e., one signal with multiple meanings) reflected in S and R matrices? And synonymity (two signals that have the same meaning).

By using a bit of a trick, we can study the evolution of such a communication system using the same protocol as in the first part of this assignment. The S and R matrices of an individual are uniquely defined by 18 numbers. Assume that we model this by saying that every individual is characterized by a genome of length 18, where each nucleotide codes for one value in S and R. Let's say $A = 3, G = 2, C = 1$ and $U = 0$. To construct the S and R matrices, we put the numbers corresponding to the nucleotides in two matrices and normalise the rows, such that the probabilities add up to 1.

? What would a genome corresponding to the S and R matrix depicted above look like?
? Can you think of two strings that have a different genotype but the same phenotype?

Of course our previous fitness function — the count of the substring "CAC" — does not make much sense in this case. We will have to define a new one. We can compute the chance of successful communication between two agents by summing up the chance of success for each individual meaning-signal combination. For instance, let's assume the sender wants to convey the meaning "leopard". We multiply the probabilities for all signals the sender could use for this meaning (the row in S starting with "leopard") with the chance that the receiver will interpret this signal as having the meaning "leopard" (the "leopard" column in R). In this case as the sender only uses the signal chirp to express the meaning leopard, and the receiver interprets this signal as having the meaning

leopard with probability 1, the chance of successfully conveying the meaning "leopard" in this system is 1. Due to the fact that agents have both a sender and receiver matrix, it is possible that the communication in one direction runs flawlessly, but any communication in the other direction is unsuccesfull. We define the fitness as the sum of the chances of success for all meanings in both directions.

? Is it possible to compute the fitness of one individual without taking into account who he is communicating with? Why (not)?

We implemented some fitness functions that you can find in the file `auxiliary_functions.R`:

- `CAC_count`: This is the fitness function you used before, that counts the number of occurrences of the substring "CAC" in the genome;
- `communication_fixed_target`: This fitness function captures how well the population member can communicate with a fixed target with S and R matrices that allow perfect communication (i.e., this other population member does not use the same signal for different meanings, or assign different meanings to the same signal).
- `communication_random_target`: This fitness function describes the more realistic situation, in which the fitness of a population member is determined based on its communication with a random other member of the population.

- Load the auxiliary functions library by typing

  ```
  source('auxiliary_functions.R')
  ```

  in the terminal. Leave the file `auxiliary_functions.R` untouched, you don't have to change anything there.

You can change the fitness function - like the rest of the parameters - at the top of the file `lab-2.R`, by uncommenting the line with the preferred fitness function (and commenting out all other fitness function lines). As you may have guessed, you can (un)comment a line in an R script by placing (removing) a '#' at the beginning.

? What is the maximal fitness that an individual can have?

- Change the fitness function in the script to `communication_fixed_target`. Run an evolutionary simulation with a low mutation rate for 100 iterations. What is the average fitness and most frequent communication system at the end of it? You can check the population at the end by typing `population` in your command line.

? Can the members of the resulting population also communicate with each other or only with the preset fixed target?

? What would happen if the target was fixed, but not perfect? You can test your assumption by changing the target matrices in the `auxiliary_functions file`.

A more realistic situation is the one in which the members of the population do not all communicate with the same fixed target, but with other members of the population, that have their own (evolved) S and R matrix.

- Run some evolutionary simulations for this scenario (compute the fitness by using the function `communication_random_target`). What is the average fitness and most frequent communication system at the end of it? Experiment with the learning rate.
- ⋆ This is frequency dependent selection. Why does it not always evolve to the optimal communication system?
- ⋆ What do you expect to happen if only successfully *receiving* but not *sending* contributes to fitness?
- Test your assumption by using the fitness function `sending_random_target`.