COMPUTER LAB

Evolution & phylogenetic reconstruction

Evolution of Language and Music

Practicalities In this computer lab you will experiment with simulated evolution and phylogenetic reconstruction: the reconstruction of the evolutionary history of species, simulated genes, or — in the end — language and music. We will work on this computer lab during several lab sessions throughout the course, and indicate which sections you are expected to finish in every lab session. If you don't manage to finish those sections, you can work on them at home, and ask questions by e-mail or during the next lab.

1 Getting started

R In this course we assume you have some basic understanding of programming and R. You will not have to write a lot of code yourself, but you should be able to read and run the scripts that we provide you with. If you have never seen any R code before, it might be useful for you to work through a tutorial, such as the first three pages of this one: `http://projects.illc.uva.nl/LaCo/CLAS/clc13/assignments/deboer13rtutorial.pdf`. Of course, you can always ask us for help if you are stuck on a particular piece of code!

RStudio We strongly recommend you use a particular environment for running your R code, called RStudio. If you haven't installed RStudio yet, you should do so by finding the appropriate download of RStudio Desktop (Open Source License) for your operating system on `www.rstudio.com` and following the instructions until you can open the program.

Console & Scripts When you open RStudio, you will find your screen divided into a few different parts. The most important ones that allow you to interact with R are on the left. The bottom left panel, called the *Console*, is where you can enter commands directly, and where the results of your command will be shown immediately after pressing Enter. When R is ready to accept commands, the *Console* shows a `>` prompt. In the *Console*, you can retrieve commands that you entered previously by using the up arrow key. However, when you close RStudio, all the commands you entered in the *Console* will be forgotten. When you are using a sequence of multiple commands and/or would like to run your

Last updated on October 24, 2018

code more often than once, it is better to save them in a *Script* (a plain text file that contains your code). The top left panel in your RStudio window is the *Script editor*.

Working directory

EXERCISE 1

- Start R-studio (depending on your operating system and preferences).
- **Todo: create new file**
- **Todo: working directory** 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.
- 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.

2 Simulated Evolution

Goals 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.

Prerequisites It is highly recommended to use R Studio. You furthermore need the two R files in the `materials` folder: `auxiliary_functions.R` and `lab-2.R`.

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. 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.

EXERCISE 2

- 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.

EXERCISE 3

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

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

  So what do we find in column 30 of our population matrix, and what does this number mean?
- 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)
  }
  ```

Defining a fitness function 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 (without overlapping, so 'CACAC' contains *one* copy). 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*

[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

containing where each element of the vector represents the fitness value for one member of the population.

EXERCISE 4

- 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:

```
# 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.

EXERCISE 5

- 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")
  ```

⋆ 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)

Evolution with mutation 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.

EXERCISE 6

- 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?

Goals Last week we implemented computer simulations of evolution in R. This week, we extend this simulation, and think about the patterns of genetic variation the evolutionary process leaves in a population. We look at methods for *phylogenetic tree reconstruction*. These methods use the genetic variation in the most recent (current) generation to reconstruct the evolutionary history of a population or a set of species.

[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.

We use a simple clustering algorithm[1] for phylogenetic tree reconstruction. The goal of this lab is to learn about the possibilities and difficulties that are involved in phylogenetic tree reconstruction.

3 Simulated evolution with ancestry

So far, we have simulated the evolution of strings of symbols, and looked at the effect of different fitness functions. Now we will repeat this simulation, but during the evolutionary process we will keep track of ancestry, so that we can reconstruct family trees of different individuals.

We will start with a very simple simulation. The script `lab-3.R` runs the same simulation we saw in the previous part of this lab. This time, however, the script also generates a matrix called `parent_matrix` that specifies the parent of each member in each generation. Where the parent is the individual of the previous generation whose genetic material was inherited. At the end of the simulation, a plot illustrating the development of both the average population fitness and the diversity of the population is generated.

EXERCISE 7

- Change the parameters at the top of the file `lab-3.R`: Set both `population_size` and `simulation_length` to 10. What values do you expect on the y-axes of these plots? What do you think the curves of average population fitness and population diversity look like? At what point do you expect the curves to start and finish?
- Run the script by executing the following command in the console:

  ```
  source('lab-3.R')
  ```

  Are the results as you expected?
- Visualise the parent matrix by running

  ```
  print_parent_matrix(parent_matrix)
  ```

  Where in this plot can you find the first generation?
- ⋆ Follow some paths up and down. Why do downward paths often end in dead ends, whereas upward paths always go all the way up?

EXERCISE 8

- Change the parameters back to their original settings:

  ```
  population_size <- 100
  simulation_length <- 1000
  ```
- Run the simulation again (this may take a while).

Printing the parent matrix for such large simulations is not very helpful (you may try if you want), because the network is too dense to properly visualise.

[1] An algorithm is a description of a series of steps to do arrive at a certain end result or perform a calculation. Algorithms can for example be implemented by a computer program.

Rather than looking the parent matrix, we will use the parent matrix to reconstruct a *family tree* for only the last generation (that is, we only look at the members of previous generations whose offspring appears in the last generation.) To generate a visual representation of the tree, we will use an online tree viewer.

EXERCISE 9

- From the data you just generated, generate a family tree with the function `reconstruct_tree` and print it with the function `print_tree`, which will generate a textual representation of the phylogenetic tree:

```
tree <- reconstruct_tree(parent_matrix)
print_tree(tree)
```

- Copy everything between the double quotes in the output of `print_tree`.
- Go to `http://evolangmus.knownly.net/newick.html`.
- On the website, change the tree type from *Cladogram* to *Rectangular cladogram*.
- Paste the tree representation you copied into the text area.
- Click "show". Now, you can zoom in by scrolling and move the tree around by dragging the mouse.
- ⋆ As far as you can judge, how many generations ago did the LCA of the current population live?
- ⋆ Which aspects of evolution leave traces that we can detect in the current generation and which aspects do not?

4 Phylogenetic reconstruction of biological data

In evolutionary research, the elaborate ancestry information represented by the parent matrix isusually not available. To reconstruct family trees we have to resort to different methods. Information about when species branched of ('speciated') can be deduced from genetic variation in the current population. For instance, horses and genetically more similar to donkeys than to, say, frogs. So, the last common ancestor of horses and frogs most likely lived much further in the past than the last common ancestor of horses and donkeys. In other words, the branch that would eventually evolve into frogs split off from the branch that would eventually evolve into horses earlier than the branch that would eventually evolve into donkeys This type of analysis is called phylogenetic reconstruction. It's based on genetic similarity between members of the current generation, which we measure using a *distance measure*. There are R packages that can automatically perform this reconstruction. Lets start with installing these packages:

EXERCISE 10

- Create a new R script for section 3.
- Install the packages `ape` and `phangorn` by typing in your *console*:

```
install.packages("ape")
```

```
install.packages("phangorn")
```

Load them in your *script* using:

```
library(ape)
library(phangorn)
```

The `phangorn` package comes with a dataset that contains real genetic data (i.e., RNA samples) from many different species. You can load this dataset by typing:

```
data(Laurasiatherian)
```

To show a summary of the data you can type `str(Laurasiatherian)`. The data originates from `http://www.allanwilsoncentre.ac.nz/`. If you want to find out more about this data, have a look at that website.

We will try to reconstruct a phylogenetic tree for these species. That is, we will try to reconstruct when different species branched off from each other, based only on genetic information of the current population (the last generation). The first step is to measure 'genetic distance' between the genetic samples for each species. For simplicity, we assume that (1) all species ultimately originate from a single common ancestor (an uncontroversial assumption in evolutionary biology), and (2) that species have diverged genetically by picking up mutations at a roughly constant rate (a more problematic assumption). (Try and convince yourself that the phylogenetic tree reconstruction method we described requires the second assumption — and that this assumption is problematic when considering evolution in the real world.)

The distance between strings of DNA or RNA is typically measured by counting the number of mutations required to change one into the other. Because of the second assumption, the genetic distance between two species is proportional to the time that has passed since their last common ancestor.

EXERCISE 11

- Select five species from the Laurasiatherian dataset (for instance three that you think are closely related and two that are more distantly related)
- Create a subset of the data containing just these five species using:

  ```
  mysubset <- subset(Laurasiatherian, subset=c(19,20,28,29,30))
  ```

  The numbers correspond to the position of the species in the list printed by `str(Laurasiatherian)`, i.e. Platypus = 1, Possum = 3, etc.) You have to replace these numbers by the numbers corresponding to the species that you chose.
- Verify that your subset contains the right species using:

  ```
  str(mysubset)
  ```

- Compute the *pairwise distance* between all elements in the set using the function `dist.ml` and print it

  ```
  distance_matrix <- dist.ml(mysubset)
  print(distance_matrix)
  ```

The results are stored in a *distance matrix*. How can you read off the distance between two species from this matrix? Why are the numbers on the diagonal of this matrix zero?

- Do the computed distances correspond to your intuitions about the selected species' relatedness?
- ⋆ Using pen and paper, or your favourite drawing software, reconstruct a phylogenetic tree that describes the evolutionary relations between your selected species. Use the principles described earlier. You shouldn't need to do any calculations.

5 Phylogenetic trees with hierarchical clustering

We can use a simple method called 'hierarchical clustering' to build such phylogenetic trees automatically. Hierarchical clustering can be done with a simple algorithm that follows these steps:

1. treat each datapoint (for example, a RNA sample of a species) as a separate "cluster" containing just one datapoint;
2. compute the distances between all clusters (using some distance measure; for example, genetic distance);
3. merge the two clusters that are nearest to each other into a new cluster;
4. repeat steps 2 and 3 until only all datapoints are in cluster.

To construct a phylogenetic tree, we can think of each merging of clusters as the joining of two branches. In the simplest version of this algorithm, we define 'distance' between a cluster A and a cluster B as the average distance between any datapoint in A and any datapoint in B (a slightly more complicated method, Ward's clustering, uses the square root of the average of the squared point-to-point distances).

EXERCISE 12

- ⋆ Using the distances between species in `mysubset`, manually perform three cycles of the hierarchical clustering algorithm with pen and paper.

The `phangorn` package we installed earlier provides pre-defined functions implementing different hierarchical clustering methods.

EXERCISE 13

- Generate a phylogenetic tree for your subset and plot it using:

  ```
  tree <- upgma(distance_matrix, method='average')
  plot(tree)
  ```

  Is the tree the same as the one that you created before with pen and paper?
- Create a tree for the entire dataset. Does it agree with your expectations?
- *Optional:* Try different methods for computing the distance between clusters by changing the parameter `method` (options are, for instance, *ward.D*, *single* and *median*). Do you notice any changes in the resulting phyloge-

netic trees?

We will now investigate what happens if we perform phylogenetic analysis on the population resulting from our own simulated evolution. Remember that, since this is a simulation over which we have full control, we can reconstructed the *actual* phylogenetic tree using the information that we stored in the parent matrix.

EXERCISE 14

- Run the script `lab-3.R` again to generate a new population and parent matrix
- Generate a distance matrix of the last generation from your simulation using the function `compute_distance_matrix`:

  ```
  distance_matrix <- compute_distance_matrix(population)
  ```

- Reconstruct a phylogenetic tree with the upgma function (choose your own *method*) and plot it:

  ```
  tree <- upgma(distance_matrix, method='ward.D')
  plot(tree,cex=0.3)
  ```

  The parameter *cex* sets the font size of the plot, adjust it if the numbers are illegible.
- Now generate the *actual* family tree of the simulation by running

  ```
  gold_standard_tree <- reconstruct_tree(parent_matrix)
  print_tree(gold_standard_tree)
  ```

  and plot it using the online tree visualiser we have used before. How well does the reconstruction produced by the hierarchical clustering algorithm match the actual family tree?
- ⋆ How can you explain the differences between the reconstructed and the actual family tree?

6 Phylogenetic reconstruction of language & music

EXERCISE 15

- We preprocessed the dataset for you so it can be loaded into R. To do this, type

  ```
  load('language_data.Rdata')
  ```

This will create an object called `mydata` containing the dataset, you can check the languages in the data by typing `names(mydata)`

EXERCISE 16

- If you are working from a university computer, reinstall the packages

`ape` and `phangorn` with the command `install.packages` (put quotes around the name of the package);

- Load the packages `ape` and `phangorn` by typing `library(ape)` and `library(phangorn)` in the console;
- Generate a list of all the languages in the dataset by typing `names(mydata)`
- Choose a subset of the list of languages. We will initially build a phylogenetic tree of this subset.
- Define your subset with the `subset` function. For instance, if you want to select language 40,41,42,58 and 60 you type:

  ```
  mysubset <- subset(mydata,c(40:42,58,60))
  ```

- Create a distance matrix of your subset, by letting the computer count the number of feature values that differ between two languages ("hamming distance"):

  ```
  distance_matrix <- dist.hamming(mysubset)
  ```

- Pick your favourite clustering algorithm and method and generate a tree, for instance:

  ```
  tree <- upgma(distance_matrix, method='ward.D')
  ```

- Plot your tree:

  ```
  plot(tree, use.edge.length=FALSE, cex=2)
  ```

- Do the same thing for the entire dataset (you might want to adapt the `cex` parameter, that sets the fontsize of the plot). Be aware of the influence the clustering algorithm and method for computing the distances between clusters can have.

⋆ What are the nine main language families you can distinguish within the Indo-European family, and in which regions of the world are they spoken (before colonial times)?