Homework 4

Computational Biology course
Handed out: 14.11.2022
Due date: 12:00 on 28.11.2022

Theory questions

We encourage you to answer these questions only after you have completed the programming task. Please keep your answers short, most of the questions can be answered in one or two sentences. Submit your answers in a pdf file named following the format Lastname_Firstname_HW4.pdf.

1. Figure1, taken from lecture 6, shows likelihood calculations performed on a tree with pendant branch lengths of 0.2 and internal branch lengths of 0.1. The K80 model was used. The cells correspond, from left to right, to nucleotides T, C, A and G. Explain why there are two different probabilities at node 6 for A and G.

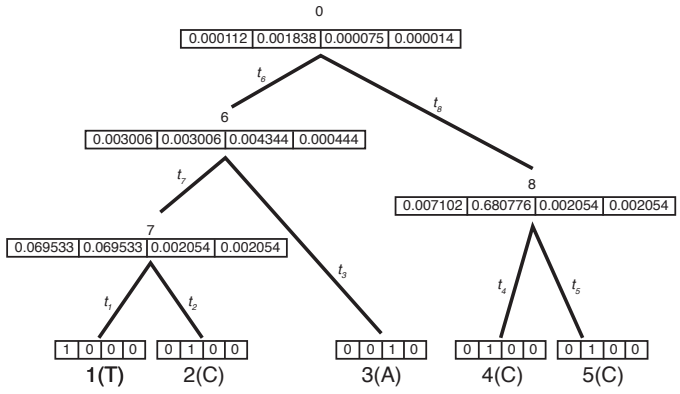

Figure 1: Likelihood calculation extracted from Lecture 6.

2. Imagine you have computed the likelihood of a tree with Felsenstein's pruning algorithm and have written down the results of intermediary calculations on internal nodes. You now alter the tree with a nearest-neighbour interchange move. If you want to calculate the likelihood of the new tree using the same algorithm, can you reuse: none, some, or all of the calculations you performed on the previous tree? Give a concise explanation.

3. List at least three key differences between the UPGMA tree reconstruction and the maximum-likelihood tree search.

4. By what factor would the runtime of Felsenstein's algorithm change if you had 5 nucleotides instead of 4? Here we are interested in how the actual number of calculations changes, not the asymptotic runtime. Shortly explain your answer.

5. In lecture 4 we learned about the condition of "time reversibility". Could you place the root anywhere in your tree and still obtain the same likelihood if the substitution model was not time reversible? Why (not)?

Programming task

Task description

In this homework you will implement Felsenstein's tree pruning algorithm to compute the likelihood of an *arbitrary* alignment on a given tree under the TN93 sequence evolution model. Your script should compute $\log L(\text{tree}|\text{alignment}) = \log P(\text{alignment}|\text{tree})$. The substitution rate matrix Q and the corresponding transition probability matrices P are defined in the "Substitution rate and transition probability matrices" section, and Felsenstein's pruning algorithm is presented in pseudo-code form in the section "Felsenstein's pruning algorithm pseudo-code".

As input your code should take:

1. `pi`: the vector of equilibrium frequencies of nucleotides;

2. `alpha1`: relative rate of C $\leftrightarrow$ T transitions;

3. `alpha2`: relative rate of A $\leftrightarrow$ G transitions;

4. `beta`: relative rate of transversions.

5. `sequences`: a list that holds the alignment of sequences with their names, e.g. `list(human = "AACTC", chimp = "AAGTC", orangutan = "TTAGT")`;

6. `newick_tree`: a tree in newick format where the tip labels match the given sequences, e.g `"(orangutan:10.25,(human:5.5,chimp:5.5):4.75);"`.

The input sequence alignment will contain sequences that are properly aligned and of the exact same length, and the alignment will not contain any gaps. As output, your script should return the log-likelihood of the given tree for the given input alignment under the TN93 sequence evolution model. A detailed step-by-step description and pseudocode overview of the algorithm are provided below.

Your script should follow the structure that we have laid out in the skeleton script called `CB_HW4_TreeLikelihood_skeleton.R`. This skeleton splits the assignment into smaller functions that are necessary to compute the final result. Each function is listed with the input parameters that it requires and the output that it must generate, as well as a short description of what it does. As in previous assignments, you only have to fill in the missing code (marked with `# ???`).

Your code will be based on two main functions, `Felstensteins_pruning_loglikelihood` and `get_subtree_likelihood`. The `Felstensteins_pruning_loglikelihood` is the main function and will be called with the parameters described above. Then the function should call `get_subtree_likelihood` on the root of the given tree, which will in fact compute the likelihood of the whole tree for each site conditional on each possible nucleotide at a site at the root. Finally, the `Felstensteins_pruning_loglikelihood` should compute the final log likelihood of the alignment on the tree by summing up all of the log likelihoods per site.

The `get_subtree_likelihood` function is called get_**subtree**_likelihood, as it will be used recursively[1] to compute the conditional likelihoods at each node of the tree, going down from the root. You will have to traverse the tree (walk in the tree from the root node to the tips), as for each node the conditional likelihoods can only be computed when the conditional likelihoods of both of the child subtrees are known. The conditional likelihoods on the tips of the tree are known from the alignment, as at each tip each site of the alignment is defined. Thus, if the nucleotide at position i in a sequence is `A`, the likelihood conditioned on observing an `A` is 1 and the conditional likelihood of any other nucleotide is 0.

A few predefined functions are there to help you with the implementation. An important one is `get_sequence_at_tip_node(node, tree, sequences)`, which will return the appropriate sequence at the given tip in the form of a vector of numerical values corresponding to the nucleotide, e.g. when asking for the sequence at tip node 1, given
`sequences = list(human = "AACTC", chimp = "AAGTC", orangutan = "TTAGT")`
and
`tree = read.tree(text = "(orangutan:10.25,(human:5.5,chimp:5.5):4.75);")`,
you will get the vector `1 1 3 4 1`. Another important function is `get_node_children(node, tree)`, which will return the labels for the child nodes of the given node, and the branch lengths leading to the child nodes. Of course, the predefined functions will not be tested and graded.

Remember to pay careful attention to the comments in the skeleton code, as these give many useful hints on how to go about implementing the algorithm. It is a very good idea to try to read through all of the skeleton code, including these comments, before starting with your implementation.

> Please do not change the structure of the code and the inputs and outputs of functions. The code will be tested and graded automatically and any structural changes will result in your code failing the tests.

To help you understand the skeleton structure, we have provided a cross-reference

[1] Recursion is a method of problem solving, where the solution to a problem depends on the solutions to smaller instances of the exact same problem. In this case this means that in order to compute the likelihood of a tree we need to compute the likelihoods of the left and the right subtrees first. In your code this will also mean that the function will call itself on the child nodes of the given node. (See the "Recursive functions" subsection in the "Additional material and help" section of this document for a gentle introduction.)

table showing which functions in the skeleton should use which others (see Table 1).

Function name	Uses
`Felstensteins_pruning_loglikelihood`	`create_TN93_Q_matrix` `get_subtree_likelihood`

`get_subtree_likelihood`	`get_sequence_at_tip_node`
	`get_likelihood_from_sequence`
	`get_node_children`
	`get_subtree_likelihood`
	`calculate_node_likelihood_from_subtrees`

Table 1: Function cross-reference table

Testing

We provide you with a test suite which is very similar to the one we will use to grade the homework. You can use it to help verify that your code is functioning properly. However, be aware that you also need to do your own testing, as the test suite does not cover all the possible cases which might not be sufficient to ensure you code is bug-free. Moreover, note that the tests used to grade your final submission will differ from these, so you should make sure that your code behaves properly when different input values are used. A good first test is to try the short alignment and the tree given in the Example section above to see whether you get the same output.

Felsenstein's pruning algorithm pseudocode

A: given sequence alignment;

N: number of sites in the sequence alignment A;

$A[n, i]$: i-th site of the sequence at node n – only defined for the tips of the tree;

$root$: root node of the tree;

$child1$: first child node of the current node;

$child2$: second child node of the current node;

$L_X^{(i)}$(node): likelihood of the subtree starting at `node` in the tree for the i'th site of the alignment, conditioned on `node` having nucleotide X;

π_X: equilibrium frequency of the nucleotide X;

$|n, l|$: length of the branch between node n and node l;

$P(b)$: transition probability matrix P for the branch length b;

$P_{XY}(b)$: the transition probability from nucleotide X to nucleotide Y in b time units;

$logL$: log of the likelihood of the whole tree given the entire alignment;

Likelihood computation of τ given an alignment
Data: Sequence alignment A, tree τ, transition probability matrices $P_{TN93}(t)$ for each branch length t in τ
Result: $logL$
$logL \leftarrow 0$;
for $i \leftarrow 1$ **to** N **do**
  $L \leftarrow 0$;
  for x *in* $\{T, C, A, G\}$ **do**
    $L \leftarrow L + \pi_x L_x^{(i)}(\text{root})$;
  end
  $logL \leftarrow logL + \log(L)$;
end
return($logL$);

To calculate $L_x^{(i)}(\text{root})$ we define a recursive function that goes through all possible nucleotide combinations at the child nodes of the current node.

Likelihood computation for node n
Data: Node n**, sequence alignment A, tree τ, transition probability matrices $P_{TN93}(t)$ for each branch length t in τ**
Result: $L^{(i)}(\text{n})$
if n *is a tip* **then**
  for $i \leftarrow 1$ **to** N **do**
    $L^{(i)}(\text{n}) \leftarrow [0, 0, 0, 0]$;
    $L^{(i)}_{A[\text{n},i]}(\text{n}) \leftarrow 1$;
  end
else
  for $i \leftarrow 1$ **to** N **do**
    for X *in* $\{T, C, A, G\}$ **do**
      $L1 \leftarrow 0$;
      $L2 \leftarrow 0$;
      for Y *in* $\{T, C, A, G\}$ **do**
        $L1 \leftarrow L1 + P_{XY}(|\text{n}, \text{child1}|) \cdot L_Y^{(i)}(\text{child1})$;
        $L2 \leftarrow L2 + P_{XY}(|\text{n}, \text{child2}|) \cdot L_Y^{(i)}(\text{child2})$;
      end
      $L_x^{(i)}(\text{n}) = L1 \times L2$;
    end
  end
end
return($L^{(i)}(\text{n})$);

Additional material and help

Required Packages

This homework assignment requires that the following pair of packages be installed on your R system:

ape : a library providing data types and methods suitable for phylogenetics, and

Matrix : a library containing various linear algebra functions.

To install these packages, enter the following command at the R command line:

```
install.packages(c("ape", "Matrix"))
```

Remember that you only need to install a package once and then you can load it for use in you code using the `library("PackageName")`. The homework skeleton loads the two aforementioned packages for you.

Useful functions

We provide a list of R functions that you may find useful when writing your code (Table 2).

R function/operator	Description
`expm(matrix)`	Exponentiate the given `matrix`.
`vector1 * vector2`	Element-wise multiplication of vectors of the same lengths.
`sum(vector)`	Summation of all the elements in a given vector.

Table 2: Useful R functions.

Substitution rate and transition probability matrices

Under the TN93 model, the substitution rate matrix Q is defined as follows:

$$\mathrm{Q}_{TN93} = (q_{ij})_{i,j\in\{\mathrm{T,C,A,G}\}} = \begin{array}{c} \\ T \\ C \\ A \\ G \end{array} \begin{array}{c} \begin{array}{cccc} T & C & A & G \end{array} \\ \begin{pmatrix} - & \alpha_1\pi_C & \beta\pi_A & \beta\pi_G \\ \alpha_1\pi_T & - & \beta\pi_A & \beta\pi_G \\ \beta\pi_T & \beta\pi_C & - & \alpha_2\pi_G \\ \beta\pi_T & \beta\pi_C & \alpha_2\pi_A & - \end{pmatrix} \end{array}$$

The transition probability matrix P is defined as $\mathrm{P}(t) = e^{\mathrm{Q}t}$.

Recursive functions

Many programming languages allow functions to call themselves, either directly or indirectly. Such functions are referred to as "recursive". If you have never encountered them before, such recursive definitions may feel either useless (how is a definition that refers to itself useful?) or even paradoxical. However, recursive functions are not only practically very useful but can also be very intuitive once you grasp the basics of how they work.

To illustrate the idea behind recursive functions, consider the following function which recursively computes the factorial of an integer n:

```
factorial = function(n) {
  if (n == 0) {
    return(1)
  } else {
    return(n * factorial(n-1))
  }
}
```

The definition is composed of two cases: one in which the result is immediately known ($0! = 1$), and one in which the result depends on a call to the same function. The first of these cases is known as the *base case*.

Suppose we then execute `factorial(3)`. The interpreter will enter the function an immediately encounter the `if` statement. Because $3 \neq 0$ the else block will be executed, which essentially has the effect of replacing the original call with the expression `3*factorial(2)`. Following the execution in this way leads to the following sequence of replacements:

```
factorial(3)
3*factorial(2)
3*(2*factorial(1))
3*(2*(1*factorial(0)))
```

We now reach a point where the interpreter can execute the function without any further recursive calls, since `factorial(0)` results in the first block (the base case) within the if statement being executed, which just returns the numerical constant 1. The final replacement thus yields

```
3*(2*(1*1))
```

which is evaluated (from the inner-most parentheses outwards) to give the final answer 6.

While this may seem like a very strange way to evaluate factorials (and indeed it is not the most efficient), the recursive definition is elegant as it is directly compariable to the following recursion relation definition:

$$n! := \begin{cases} 1 & \text{if } n = 0 \\ (n-1)! \cdot n & \text{if } n > 0 \end{cases}$$

Furthermore, for trees, recursive functions provide a very natural way of performing an operation on every node of the tree. This is known as a *tree traversal*.

Example log likelihood computation

This section shows an example of the likelihood computation of a given tree given an alignment of length 1. As with all other assignments we assume the following order of nucleotides: T, C, A, and G. The parameters for this example are defined as follows:

```
beta = 0.00135
alpha1 = 0.5970915
alpha2 = 0.2940435
pi = c(0.22, 0.26, 0.33, 0.19)
newick_tree = "((one:2,two:2):1,(three:1,four:1):2);"
sequences = list(one = "C", two = "A", three = "T",
four = "G")
```

Figure 2 shows the tree and the sequences at the tips.

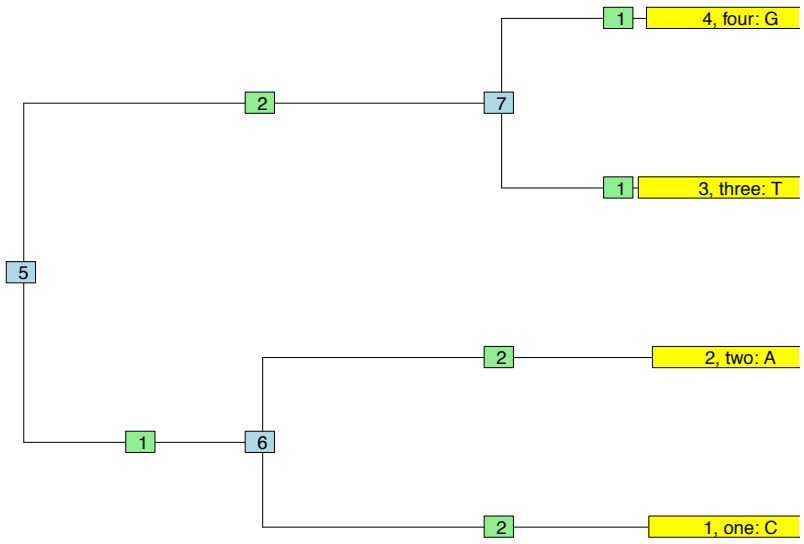

Figure 2: Example tree for the log likelihood computation. Blue labels correspond to the labels of internal nodes, green labels display branch lengths, and yellow labels display the labels of the tips, associated to their sequence.

Q and P matrices

To compute the likelihood one would need to compute the TN93 Q matrix, which for the given parameters should look as follows:

	T	C	A	G
T	−0.1559458	0.1552438	0.00044550	0.00025650
C	0.1313601	−0.1320621	0.00044550	0.00025650

A	0.0002970	0.0003510	−0.05651627	0.05586827
G	0.0002970	0.0003510	0.09703436	−0.09768235

The example tree only has 2 different branch lengths, thus only 2 different P matrices will be necessary for the computation. The P matrix for branch length 1 P(1) should look as follows:

	T	C	A	G
T	0.8644146411	0.1348838325	0.0004451994	0.0002563269
C	0.1141324737	0.8851659999	0.0004451994	0.0002563269
A	0.0002967996	0.0003507632	0.9475792317	0.0517732055
G	0.0002967996	0.0003507632	0.0899218832	0.9094305540

The P matrix for branch length 2 P(2) should look as follows:

	T	C	A	G
T	0.7626075054	0.2359903883	0.0008897982	0.0005123081
C	0.1996841747	0.7989137190	0.0008897982	0.0005123081
A	0.0005931988	0.0007010532	0.9025622328	0.0961435152
G	0.0005931988	0.0007010532	0.1669861054	0.8317196426

Tip likelihoods

For tree tips, the conditional likelihoods only depend on the tip sequences. I.e. for node 1, for which position 1 has nucleotide C, the conditional likelihoods per nucleotide at this site look as follows:

$$\left(L_T^{(1)}(1) = 0, \quad L_C^{(1)}(1) = 1, \quad L_A^{(1)}(1) = 0, \quad L_G^{(1)}(1) = 0\right)$$

Similarly, the conditional likelihood vectors for the three remaining tips are:

$$L^{(1)}(2) = (0, 0, 1, 0)$$
$$L^{(1)}(3) = (1, 0, 0, 0)$$
$$L^{(1)}(4) = (0, 0, 0, 1)$$

Internal node likelihoods

To compute the conditional likelihoods at an internal node we need to know the conditional likelihood values at the child nodes of the current node. For example, computing the likelihood at node 6, conditioned on having a nucleotide T, requires to know the conditional likelihoods at tips 1 and 2:

$$\begin{aligned}
L_T^{(1)}(6) &= \left(\sum_{X \in \{\text{T,C,A,G}\}} P_{TX}(2) L_X^{(1)}(1)\right) \times \left(\sum_{X \in \{\text{T,C,A,G}\}} P_{TX}(2) L_X^{(1)}(2)\right) \\
&= (0 \times P_{TT}(2) + 1 \times P_{TC}(2) + 0 \times P_{TA}(2) + 0 \times P_{TG}(2)) \\
&\quad \times (0 \times P_{TT}(2) + 0 \times P_{TC}(2) + 1 \times P_{TA}(2) + 0 \times P_{TG}(2)) \\
&= (0 \times 0.7626075054 + 1 \times 0.2359903883 + 0 \times 0.0008897982 + 0 \times 0.0005123081) \\
&\quad \times (0 \times 0.7626075054 + 0 \times 0.2359903883 + 1 \times 0.0008897982 + 0 \times 0.0005123081) \\
&\approx 0.0002099838
\end{aligned}$$

Computing the conditional likelihoods for the other 3 nucleotides is done in exactly the same way:

$$
\begin{aligned}
L_C^{(1)}(6) &= \left(\sum_{X\in\{\mathrm{T,C,A,G}\}} P_{CX}(2)L_X^{(1)}(1)\right) \times \left(\sum_{X\in\{\mathrm{T,C,A,G}\}} P_{CX}(2)L_X^{(1)}(2)\right) \\
&= (1 \times P_{CC}(2)) \times (1 \times P_{CA}(2)) \\
&= 0.7989137190 \times 0.0008897982 \approx 0.000710872 \\
L_A^{(1)}(6) &= \left(\sum_{X\in\{\mathrm{T,C,A,G}\}} P_{AX}(2)L_X^{(1)}(1)\right) \times \left(\sum_{X\in\{\mathrm{T,C,A,G}\}} P_{AX}(2)L_X^{(1)}(2)\right) \\
&= (1 \times P_{AC}(2)) \times (1 \times P_{AA}(2)) \\
&= 0.0007010532 \times 0.9025622328 \approx 0.0006327441 \\
L_G^{(1)}(6) &= \left(\sum_{X\in\{\mathrm{T,C,A,G}\}} P_{GX}(2)L_X^{(1)}(1)\right) \times \left(\sum_{X\in\{\mathrm{T,C,A,G}\}} P_{GX}(2)L_X^{(1)}(2)\right) \\
&= (1 \times P_{GC}(2)) \times (1 \times P_{GA}(2)) \\
&= 0.0007010532 \times 0.1669861054 \approx 0.0001170661
\end{aligned}
$$

The vector of conditional likelihoods at node 6 is:

$\mathrm{L}_1(6)\mathrm{:}\ \ 0.0002099838\ \ 0.000710872\ \ 0.0006327441\ \ 0.0001170661$

The vector of conditional likelihoods at node 7 are:

$\mathrm{L}_1(7)\mathrm{:}\ \ 0.0002215728\ \ 2.925523\mathrm{e}{-}05\ \ 1.536627\mathrm{e}{-}05\ \ 0.0002699186$

The conditional likelihoods for internal node 5 (which is the root of the tree) are computed in exactly the same way. E.g. for nucleotide T the computation looks as follows:

$$
\begin{aligned}
L_T^{(1)}(5) &= \left(\sum_{X\in\{\mathrm{T,C,A,G}\}} P_{TX}(1)L_X^{(1)}(6)\right) \times \left(\sum_{X\in\{\mathrm{T,C,A,G}\}} P_{TX}(2)L_X^{(1)}(7)\right) \\
&= (0.0002099838 \times 0.8644146411 + 0.000710872 \times 0.1348838325 \\
&\quad + 0.0006327441 \times 0.0004451994 + 0.0001170661 \times 0.0002563269) \\
&\quad \times (0.0002215728 \times 0.7626075054 + 2.925523e-05 \times 0.2359903883 \\
&\quad + 1.536627e-05 \times 0.0008897982 + 0.0002699186 \times 0.0005123081) \\
&\approx 4.888499e-08
\end{aligned}
$$

Similarly, for all the other nucleotides the conditional likelihoods are as follows:

$$\begin{aligned}
L_C^{(1)}(5) &= \left(\sum_{X\in\{\mathrm{T,C,A,G}\}} P_{CX}(1)L_X^{(1)}(6)\right) \times \left(\sum_{X\in\{\mathrm{T,C,A,G}\}} P_{CX}(2)L_X^{(1)}(7)\right) \\
&\approx 4.428818e-08 \\
L_A^{(1)}(5) &= \left(\sum_{X\in\{\mathrm{T,C,A,G}\}} P_{AX}(1)L_X^{(1)}(6)\right) \times \left(\sum_{X\in\{\mathrm{T,C,A,G}\}} P_{AX}(2)L_X^{(1)}(7)\right) \\
&\approx 2.422087e-08 \\
L_G^{(1)}(5) &= \left(\sum_{X\in\{\mathrm{T,C,A,G}\}} P_{GX}(1)L_X^{(1)}(6)\right) \times \left(\sum_{Y\in\{\mathrm{T,C,A,G}\}} P_{GX}(2)L_X^{(1)}(7)\right) \\
&\approx 3.718882e-08
\end{aligned}$$

The vector of conditional likelihood values at node 5 (root node) is:

$\mathrm{L_1(5):}\ \ 4.888499\mathrm{e}{-}08\ \ 4.428818\mathrm{e}{-}08\ \ 2.422087\mathrm{e}{-}08\ \ 3.718882\mathrm{e}{-}08$

Final tree log likelihood computation

Once the vectors of conditional likelihoods for each position of the sequence at the root are computed, we can compute the final log likelihood of the tree by using the equilibrium distribution π as follows:

$$\begin{aligned}
logL &= \log\left(\prod_{i=1}^{N}\left(\sum_{X\in\{\mathrm{T,C,A,G}\}} \pi_X L_X^{(i)}(5)\right)\right) \\
&= \sum_{i=1}^{N} \log\left(\sum_{X\in\{\mathrm{T,C,A,G}\}} \pi_X L_X^{(i)}(5)\right)
\end{aligned}$$

Given that in this example there is only one site, the computation boils down to:

$$\begin{aligned}
logL &= \log\left(\sum_{X\in\{\mathrm{T,C,A,G}\}} \pi_X L_X^{(1)}(5)\right) \\
&= \log(\pi_{\mathrm{T}} L_T^{(1)}(5) + \pi_{\mathrm{C}} L_C^{(1)}(5) + \pi_{\mathrm{A}} L_A^{(1)}(5) + \pi_{\mathrm{G}} L_G^{(1)}(5)) \\
&= \log(0.22 \times 4.888499e-08 + 0.26 \times 4.428818e-08 + \\
&\qquad 0.33 \times 2.422087e-08 + 0.19 \times 3.718882e-08) \\
&= \log(3.732839e-08) \\
&\approx -17.1035117087
\end{aligned}$$