Biological sequence analysis

Hidden Markov Model assignment March 12, 2025

I am making the function definition suggestions general below (arbitrary output alphabet, arbitrary number of hidden states) because for future assignments you can re-use them. But right now test it on the fair/biased coin tossing problem:
Example probabilities (you can play with these):
Hidden states 1 = fair, 2 = biased, 0 = start/end
Fair coin: probability of heads = tails = 0.5
Biased coin: probability of heads = 0.8, tails = 0.2
Switching probability: 0.05, probability of staying in same hidden state: 0.94, probability of terminating (from either H or T): 0.01

Start with Viterbi and send that to me before doing the rest.

1. Implement a hidden Markov model sequence generator. Write a function called `hmm_gen` that takes an alphabet `S`, a hidden state space `T`, a matrix of transition rates `a` (including from begin and to end states), a matrix of emission rates `e`, and generates a sequence according to the HMM specified by those parameters. Make it output both the sequence `x` and the hidden state sequence `pi`

   Here `T` can be an integer indicating the number of hidden states (excluding 0). For example, in the coin example above,

$$
\begin{aligned}
\texttt{S} &= \{H, T\} \equiv \{1, 2\} \\
\texttt{T} &= \{0, F, B\} \equiv \{0, 1, 2\} \\
\texttt{a} &= \begin{pmatrix} 0 & 0.5 & 0.5 \\ 0.01 & 0.94 & 0.05 \\ 0.01 & 0.05 & 0.94 \end{pmatrix} \\
\texttt{e} &= \begin{pmatrix} 0.5 & 0.5 \\ 0.8 & 0.2 \end{pmatrix}
\end{aligned}
$$

   Here, the first row and column of `a` have index 0. The `e` matrix has indices 1 and 2 (F and B for rows, H and T for columns).

2. Implement the following:

   (a) A function called `viterbi` that takes a sequence `x`, an alphabet `S`, a hidden state space `T` (ie number of hidden states), a matrix of transition rates `a`, a matrix of emission rates `e`, and implements the Viterbi algorithm. Run this function on the sequence output `x` of your previous function, for the fair/biased coin problem for various choices of transition probabilities, and see if it recovers the (known) hidden states.

   Use log probabilities to avoid underflows. Also, in python you can implement the pointers as a matrix (or list of lists) of tuples, eg if `ptr[3,2] == (3,1)` that means the (3,2) element was calculated from the (3,1) element.

   (b) A function called `forward` that takes a sequence `x`, an alphabet `S`, a hidden state space `pi`, a matrix of transition rates `a`, a matrix of emission rates `e`, and implements the "forward algorithm" for the subsequence `x[1..n]`. Use log probabilities, and the log-sum-exp trick to calculate sums of probabilities.

   (c) Similarly, a function called `backward` that takes the same parameters and implements the backward algorithm for the subsequence `x[n..L]` where `L` is the length of `x`.

   (d) A function `posterior` that takes the same parameters and returns a vector of posterior probabilities for all possible hidden states at site `[n]`. Again, see if it recovers the known hidden states, for various choices of the transition probabilities.

   You should find that the performance is fairly good if transition probabilities from fair $\rightarrow$ unfair and vice versa are small.