Algorithms in Computational Biology, 2020
Exercise 2 - Programming - Motif finding

Due date: 3/12/20

Submission should include a tar file named $ex2.tar$, containing the following files:

- motif_find.py
- Possibly other python files

ZOOPS model

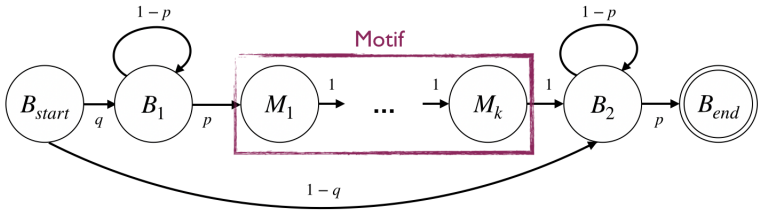
In class we discussed using HMMs for finding motifs in biological sequences. We presented three different models, OOPS (One Occurance Per Sequence), ZOOPS (Zero Or One Occurance Per Sequence) and TCM (Two Component Mixture).

In this exercise you will implement the ZOOPS variant for finding motifs where a sequence contains at most one occurrence of a motif of size k. In each sequence a motif can begin at a different position, if at all.

Recall an HMM model with k states ($S = \{s_i\}_{i=1}^{n}$) that emits data from an alphabet of $\Sigma = \{\sigma_j\}_{j=1}^{m}$ can be described with parameters $\theta = (\tau, e)$ where:

- τ - transition probability matrix ($n \times n$), where $\tau_{i,j}$ is the probability of moving from state s_i to state s_j
- e - emission probability matrix ($n \times m$), where $e_{i,j}$ is $e_{s_i}(\sigma_j)$: the probability of s_i to emit letter σ_j.

Consider the following ZOOPS model:



States:

- B_{start}, B_{end} are the start/end states. The model is forced to begin in B_{start} and to end in B_{end}. These special states do not emit any value (see details below).
- B_1, B_2 are background (non-motif) states.
- $m_1, ..., m_k$ are states denoting each position of the motif.

Transitions:

- p is the probability of transition from a background state (B_1, B_2) to the next state (M_1, B_{end}). Typically, p will be small, as the background states are the most abundant.
- q is the probability of transition from B_{start} to B_1. q represents a prior knowledge about the probability a sequence contains a motif. For $q = 1$, we get an OOPS model.

Emissions:

- Fix the emission probabilities for the background states (B_1, B_2) to be uniform $(\frac{1}{4})$.
- Emission probabilities for the motif states will be given as input (or learned, in EM part).

Note: according to this specific model, a motif cannot begin at the first position, and must end before the position of the sequence.

Implementing Start/End states: In order to enforce the model to begin/end in B_{start}/B_{end}, we recommend the following trick:

- End: add the special character \$ to the alphabet, and append it to the end of the input sequence. Edit the emissions matrix such that

$$e_{S_i}(\$) = \begin{cases} 1 & S_i = B_{end} \\ 0 & S_i \neq B_{end} \end{cases}$$

  Don't forget to remove the special character from your output.
- Start: Similary, use the ˆ character for B_{start}.

HMM implementation

In this exercise you will implement the motif_find.py program, with 4 possible algorithms: Forward, Backward, Posterior decoding and Viterbi. Write a

program that reads the model probabilities, initializes a new HMM model, and runs these algorithms.

Input:

- The code receives an initial emission matrix: initial_emission.tsv - defines the emission probabilities of the states for each nucleotide. Columns correspond to A, C, G, T in order, left to right. The rows describe the motif states in order $1, ..., k$.
- Transition probabilities are passed as command line arguments p, q.
- The sequence, as a string (seq).

1 Probability of Observations

Using the HMM model it is possible to evaluate the (log-)likelihood that some sequence of observations came from the model.

Implement the **Forward** algorithm. The function should receive a sequence, and transition and emission probabilities, and generate the Forward table. Use this table to find the log-likelihood of the sequence.

It should be possible to invoke the forward algorithm from the command line using the following syntax:

```
python3 motif_find.py --alg forward seq initial_emission.tsv p q
```

The program should print the log-likelihood of the sequence. The Forward table should not be printed.

Similarly, implement the **Backward** algorithm.

Running the following command should print the same output as the forward command (i.e. the log-likelihood):

```
python3 motif_find.py --alg backward seq initial_emission.tsv p q
```

For the example above, the output is simply (output at least 2 digits precision):

-108.21

2 Posterior Decoding

Using the Forward and Backward algorithms, you can now compute posterior probability for each state of the i'th position in the sequence X, $P(S_i = s|X)$. Use this to find the most likely state for each index, given the sequence.

Running the following command:

```
python3 motif_find.py --alg posterior seq initial_emission.tsv p q
```

Should print the most probable state for each index, found by the posterior decoding.

The output format should be as follows (for $p = 0.1, q = 0.8$ and the provided *initial_emission.tsv* file):

BBBBBBBBBBBBBBMMMMMMMMBBBBBBBBBBBBBBBBBBBBBBBBBBBB
TCGAATCCGTACGGTTAGCTTTACGGCGCCTCGAATTCGCCCCCCATATG

BBB
CTT

Blocks of 2 lines of length up to 50 characters followed by an empty line. Bottom row should be the given sequence and top row the corresponding hidden state ('B' - Background, 'M' - Motif). Do not print the special states (B_{start}, B_{end}) or characters (^, $).

3 Viterbi Decoding

Given the model parameters (i.e. transition probabilities p, q and emission probabilities for motif states) and a sequence of observations $X = (x_i)_{i=1}^{n}$, we want to find the most likely assingment for the hidden states that emitted X. That is, which positions in the sequence are part of a motif. In this section you are required to retrieve the most probable path for the hidden states given the emitted sequence.

It should be possible to invoke the decoder from the command line using the following syntax:

```
python3 motif_find.py --alg viterbi seq initial_emission.tsv p q
```

The program should print the most probable path of hidden states that could emit the sequence seq, using **Viterbi** algorithm. The output **format** is identical to the Posterior decoding format above.

Are the results identical to the posterior decoding? Why?

Tips and implementation issues

Running time

- Your program should run fast. Your Forward & Backward methods will run many times on the next exercise. Use vectorized numpy operations instead of nested loops as often as possible.

Underflow

- The probabilities in the Viterb/Forward/Backward tables decrease with the length of the input sequence, and it might cause underflow. If the sequence is long enough (e.g. >200bp), It's likely the last columns of the

tables will be all zeros, and the output will be incorrect. The solution we suggest for this problem is working in log space. Replace multiplication with addition and addition with logsumexp.

- It is recommended to first implement the algorithms in a straightforward way, with nested loops and without logspace. Make sure it's correct for short sequences (5-10 characters), and then change it to log space and vectorized operations.
- You are allowed to use non-standard libraries for reading / writing files (e.g. pandas, Bio).
- Optional: You may find it useful to print colored text in your terminal (use it for debugging, but do not submit it). Try this, for example:

```
>>> print("\033[91m" + "Red text!" + "\033[00m")
```