MMG1345H: Foundational Computational Biology (FCB) II **Winter 2022**

<u>Assignment 1:</u>

In this assignment, you are going to implement an RNA motif-finding program. You will be implementing different versions of this program, and you will be implementing it in stages, so **please read all of the instructions before you start.**

The input to your program is the motif length, K, of the motif model that you are trying to fit, and the filename of a list of RNA sequences. This file contains sequences, one per line like this:

```
AGAAUCUA
GAUUCUA
GAUUCUAA
```

An example of an input file (*sample_input.txt*) is included in the assignment folder.

The output of your program will be a motif model represented as a position frequency matrix (PFM). A PFM has 4 rows and *K* columns. Each row *j* of this matrix *M* is associated with one of the four bases: *j = 1* corresponds to A, *j = 2* corresponds to C, *j = 3* corresponds to G, and *j = 4* corresponds to U.

We will use the base-to-number translation scheme to represent the input sequences (and their constituent k-mers) as arrays of numbers, so the 4-mer <u>UCUA</u> will be represented as [4,2,4,1], and the input sequence AGAA<u>UCUA</u> will be [1,3,1,1,4,2,4,1].

In the PFM, *M*, M_{jk} is the frequency with which the base corresponding to *j* appears in the *k*-th position of a *K*-mer in the RNA motif. Because they are frequencies (i.e. probabilities), all the values in *M* are positive, and each column sums to one.

To compute the probability, *P(**c**)*, that a PFM, *M*, 'generates' a *K*-mer represented by the array ***c***, we simply multiply together all of the elements of *M* corresponding to the bases indicated in ***c***:

$$P(c) = \prod_{k=1}^{K} M_{c_k k}$$

where c_k is the *k*-th element of the array **c**. Here we are using c_k to select which row of *M* to use when we get to the *k*-th column.

The PFM is a succinct way of representing the probabilities of different *K*-mers. The meaning of the PFM depends on what this motif model is used to represent. For example, if the PFM is used to model the binding preferences of an RNA-binding protein (RBP) – then the probabilities it assigns to K-mers correspond to how often those K-mer would appear as a binding site for the RBP, assuming that there was a limiting concentration of RBP and it had the opportunity to bind to a set of random sequences with an equal distribution of A's, C's, G's and U's.

Here, we are going to fit the PFM in a generative model framework which means that we need to correct the background distribution of the bases. We are going to represent that distribution with a 4-element vector ***b***, where b_j is the background probability of the base corresponding to *j*. Under the background model, the probability of a *K*-mer ***c***, *Q(**c**)*, is

$$Q(c) = \prod_{k=1}^{K} b_{c_k}$$

You will also output a set of offsets that indicate where the best match to the motif is in each of the input sequences. The offset o_i for the *i*-th sequence indicates the position in the input sequence that is the start of the *K*-mer which is the best match to the motif model. For this assignment, we will use "1-indexing" meaning that the first position in an input sequence is indicated by a "1" – beware, some programming languages use "0-indexing" where the first position in an array is indicated by "0". Differences between 1-indexing and 0-indexing are the source of many disastrous errors in bioinformatics so it is important to be sensitive to this issue early. We are using 1-indexing in this assignment so that $o_i = 0$ will indicate, in some of your answers, that there are no matches that are better than random in the sequence. Also note that in order for there to be a full *K*-mer at the given offset, the maximum offset that a given input sequence can have is $L_i - K + 1$ where L_i is the length of the sequence.

An example of an output file (*sample_output.txt*) is included in the assignment folder.

The goal of motif finding is to find a PFM, and a corresponding set of offsets, that maximizes:

$$E(M, o) = \prod_{i=1}^{I} \frac{P(s_i[o_i : o_i + K - 1])}{Q(s_i[o_i : o_i + K - 1])}$$

Where $s_i[o_i : o_i + K - 1]$ is the array representation of the *K*-mer in the *i*-th sequence (as indicated by s_i) that starts at position o_i and ends at position $o_i + K - 1$ inclusive. Here E is called an *objective function*.

Your program will fit the motif model using an iterative "coordinate ascent" procedure that switches between two phases: 1) updating *o* for a given *M*, and 2) finding the best *M* for a given *o*. These two phases are sometimes the E-step and M-step, because this algorithm is similar to an Expectation-Maximization (EM) algorithm.

With this iterative procedure, you need to make either an initial guess for *o*, or an initial guess for *M*. We will start the procedure with *o*. Usually, people randomly select a starting array for *o*, but here, so that everyone gets the same answer, we will set all of the initial values of *o* to 1.

Pro-tip: if you have a lot of sequences, *E(M, o)* might get quite big and cause numerical overflow – because of this, it is better to compute, and report, *log E(M, o)*. You can compute this by summing the logs of the probability ratios rather than computing *E(M, o)* by taking the product of those ratios.

In the output file, for each of the four models that you will implement below, report *log E(M, o)*, the final score that your motif achieved, together with 1) the associated *M* and *o* and 2) the runtime of your algorithm.

Model 1 - HardOOPS: OOPS model, hard E-step

First, we will assume that there is exactly one copy of the motif per input sequence (OOPS), and simply choose the best offset. This is sometimes called a “hard E-step” or hard assignment (as opposed to a soft assignment).

For the M-step, you can update M given o as follows:

Set n_{jk} to be the number of times that the base *j* appears in the *k*-th position of the *K*-mers indicated (in the input sequences) by the offset array *o*.

$$M_{jk} = \frac{n_{jk}+a_j}{\Sigma_{l=1}^{4} n_{lk}+a_l}$$

where the array a = [a_1, a_2, a_3, a_4] are pseudo-counts representing our prior. Here let’s just use *a = b*. For ease, also set all of the elements of ***b*** to 0.25 but in general, *b* should be set based on the counts of A, C, G, and U in the input sequences or the genome of interest.

For the E-step, for each input sequence s_i, choose o_i to maximize:

$$\frac{P(s_i[o_i:o_i + K - 1])}{Q(s_i[o_i:o_i + K - 1])}$$

Note that once you finish the E-step, you have all the information you need to compute *E(M, o)*. If your program is working correctly, *E(M, o)* should always increase every iteration until your algorithm converges, then it will stop increasing. You can also check for convergence by checking whether or not the offset vector changes from the previous iteration.

Model 2 - HardZOOPS: ZOOPS model, hard E-step

Instead of forcing each sequence to contain a copy of the motif, you can sometimes generate a better motif model by leaving some input sequences out when you build it. You might do this, for example, if there was noise in the binding assay or some outlier sequences. To do this, you use the ZOOPS model which permits the motif model to ignore some sequences.

Implementing ZOOPS requires a simple change to your code. In the E-step, if, for some sequence *i*, none of the probability ratios are greater than 1 — in other words, the PFM does not explain any *K*-mers in the sequence better than the background — then set o_i to 0 and set the ratio for this sequence to 1. No *K*-mers from this sequence contribute to the motif in the M-step.

Model 3 - GibbsZOOPS: ZOOPS model, Gibbs sampling

Hard E-steps can get stuck in a bad local optimum. There are two ways to address this problem – restart with a different initial guess for ***o*** or, as we will do here, sometimes generate less than optimal offset arrays in the E-step. Now we are going to replace the hard E-step with a Gibbs sampling step (Frith *et al.*, 2008).

What is Gibbs sampling?

Gibbs sampling (Wikipedia page) is a Monte Carlo Markov Chain (MCMC) method that iteratively draws an instance from the distribution of each variable, conditional on the current values of the other variables in order to estimate complex joint distributions. It is a very useful way of sampling from distributions that are difficult to sample directly. We will learn more about Gibbs sampling later in this course.

To incorporate Gibbs sampling into our RNA-motif finding program, in the E-step, instead of selecting the best ratio, we will sample the value of o_i from a probability distribution over the possible offsets, including $o_i = 0$. This is a type of Gibbs sampling. Here's how to do it:

First, for each sequence *i* compute

$$r_{pi} = \frac{P(s_i[p{:}p + K - 1])}{Q(s_i[p{:}p + K - 1])}$$

for each possible offset *p* in sequence *i*, including 0, and set $r_{0i} = 1$. Then compute

$$g(p) = \frac{r_{pi}}{\sum_{q=0}^{L_i - K + 1} r_{qi}}$$

Now *g(p)* is a probability distribution over the possible values for o_i, sample o_i from this distribution. Continue doing this until you have generated the array *o*.

Unlike the hard E-steps, this will not converge, **so you need to decide how many iterations to run – make it a parameter and set it to 1000 for this assignment**. Let's call this value T_{max}. Also, there is no guarantee that *log E(M, o)* will always increase. So you need to save the offset vector *o* that achieves the highest value of E. At the end of T_{max} iterations, you should output the best motif model along with the value of E that it achieved — note that you can always recompute *M* given *o*, so you can just save *o* and recompute *M* at the end if you want.

Model 4 - AnnealZOOPS: ZOOPS model, Gibbs sampling, Simulated annealing

Sometimes Gibbs sampling alone isn't enough to knock your program out of bad local optima. For example, if many different offsets need to change at once to get your PFM to a better one, this is very unlikely to happen quickly with Gibbs sampling. To speed up convergence to a better optimum, you can use a technique called Simulated Annealing.

What is simulated annealing?

Simulated annealing (Wikipedia page) is a very useful technique for approximating the global optimum of a given function. Originally inspired by the process of annealing in metalwork, simulated annealing involves "heating" and "cooling" a temperature variable: We initially set it high and then allow it to slowly "cool" as the algorithm runs. While this temperature variable is high the algorithm will be allowed, with more frequency, to accept solutions that are worse than our current solution. This gives the algorithm the ability to jump out of any local optima in which it finds itself early on in the execution. As the temperature is reduced so is the chance of accepting worse solutions, therefore allowing the algorithm to gradually focus on an area of the search space in which hopefully, a close-to-optimal solution can be found.

To perform simulated annealing, you "heat up" the E-step so that you take more unlikely steps in earlier iterations and gradually "cool down" the E-step in later iterations. In your last iteration, you sample *o* from the correct distribution.

Once again, this is a simple change to your code. Simply replace *g(p)* with *h(p,T)*.

$$h(p,T) = \frac{(r_{pi})^T}{\Sigma_{q=0}^{L_i - K + 1} (r_{qi})^T}$$

where $T = t/T_{max} - 1$ and t is the index of the current iteration. T is inversely proportional to the 'temperature' of the system in this case. t starts at 0 and ends at T_{max}. If $t = 0$, then $T = 0$ and $h(p,T)$ gives a uniform distribution over p – this is the most random you can get. When $t = T_{max} - 1$, then $T = 1$ and $h(p,T) = g(p)$.

Here, unlike Gibbs sampling, you should take the final setting for *o*, because it is the only setting sampled from the correct distribution. What happens in the earlier iterations is a gentle nudging of *o* toward a better optimum than those nearest its starting point.

Compare and contrast

Apply your four motif finding models to the test datasets (*test_input_<i>.txt*) in the assignment folder. For each dataset, use ***K = 6*** as input and generate a text output file. Also, plot *log E(M, o)* over iterations**.** What method is the best? Why?

One sample plot (*sample_hardoops.png*) was included in the assignment folder.

Submission

Please submit a zip file (first_lastname.zip) with

1. Program source code (in .R or .py format) for your four motif finding models.
2. A README file on how to run your models.
3. Output text files, one per test input file.
4. Plots, one set (4 plots) per test input file.
5. A PDF write-up. You may embed plots into the write-up.

References

Frith, Martin C., Neil F. W. Saunders, Bostjan Kobe, and Timothy L. Bailey. 2008. "Discovering Sequence Motifs with Arbitrary Insertions and Deletions." *PLoS Computational Biology* 4 (4): e1000071.