6.047 Problem Set 3 Writeup

Matthew Feng

October 21, 2018

1 Gibbs sampling for motif discovery

(a) Gibbs sampling

My implementation of Gibbs sampling attempts to introduce randomization in as many locations as possible, with the goal to avoid becoming trapped in local maxima. As such, the sequence that is excluded is selected, but in a randomly shuffled order, so that every sequence is selected at the same frequency, but with randomization. Additionally, I incorporated the background probabilities of the bases as a "score correction", dividing the probability of observing motifs by the probability of the motif given the background distribution, to avoid focusing in on motifs that are simply more abundant, rather than more informative.

(b) Test cases

The most probable PWM for each of the datasets can be found in the `pwms/` folder. See Figures 1, 2, 3 and 4 for the most probable PWMs as sequence logos.

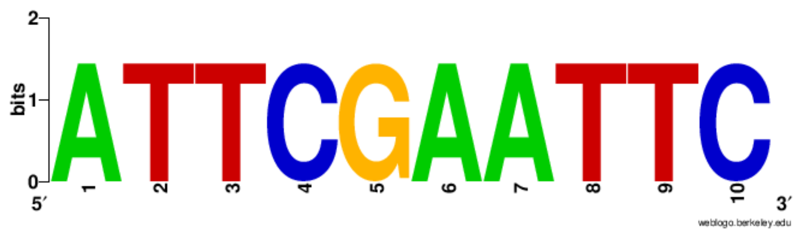

Figure 1: Most probable motif in `data1.txt`.

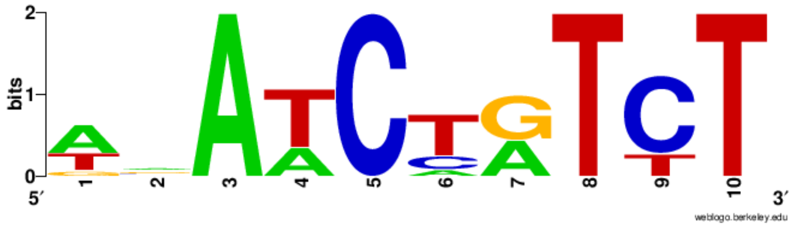

Figure 2: Most probable motif in `data2.txt`.

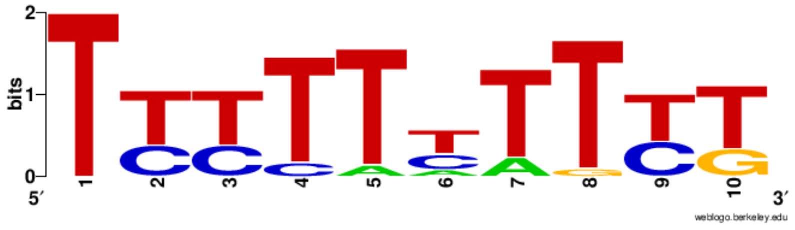

Figure 3: Most probable motif in `data3.txt`.

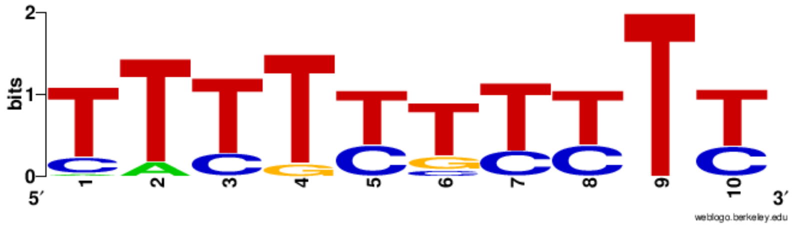

Figure 4: Most probable motif in `data4.txt`.

2 Evolutionary signatures of motifs

(a) Frequency vs. Conservation

See `kmers.py` for determining k-mer frequency and conservation ratio.
See `motif_match.py` for matching potential k-mer motifs against known motifs (i.e. testing for viability of the algorithm).
See `freq_vs_conserved.txt` for the top 50 most frequently occuring k-mers and the top 50 most highly conserved k-mers.

(b) Validation

Simply counting the frequency of a 6-mer as a motif has its bias in that the more frequently appearing nucleotides will be overrepresented; in other words, common sequences that do not have particular meaning will tend to be the most frequent k-mers.

As such, we should use the most percentage conserved motifs instead, as conservation usually implies that certain sequences have been important throughout evolution.

From the most conserved 6-mers, we identified the following motifs:

1. `CACGTG` $\rightarrow$ PHO4
2. `ACGCGT` $\rightarrow$ SWI6
3. `GATGAG` $\rightarrow$ ESR1

3 RNA secondary structure

(a) Nussinov algorithm

See `nussinov.py` for the implementation details. The program uses dynamic programming using a bottom-up approach for greater efficiency.

(b)

The average Nussinov score across 1000 randomly generated RNA sequences is -44.155.

(c)

As length increases, the score decreases (i.e. more pairings).

See Figure 5 for reference.

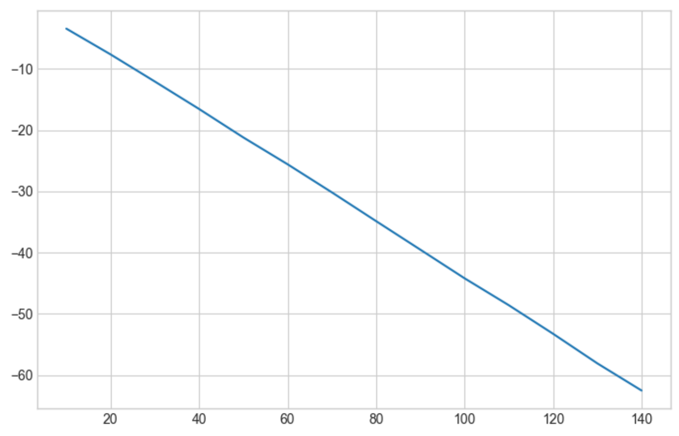

Figure 5: As sequence length increases, the score decreases (i.e. more bases are paired). Sequence length is along the horizontal axis, while the Nussinov score is along the vertical axis.

(d)

As GC content approaches 0.5, number of pairs decreases (i.e. score becomes less negative), and then increases again (i.e. score becomes more negative) as the GC content approaches 1.0. The score function is symmetric about GC content of 0.5 because as GC content decreases, the number of A-U pairs is likely to increase, but when GC content increases, the number of G-C pairs is likely to increases, either of which compensates for the other.

See Figure 6 for reference.

(e)

Since both length and percentage of GC content influence the score output by the Nussinov algorithm, it may be useful to normalize the score (correct the score) by dividing by the length and $\alpha + |0.5 - p|$, where p is the percentage of GC content and α is a hyperparameter to be tuned. The division essentially "cancels out" any effect of the length and percentage had in the numerator.

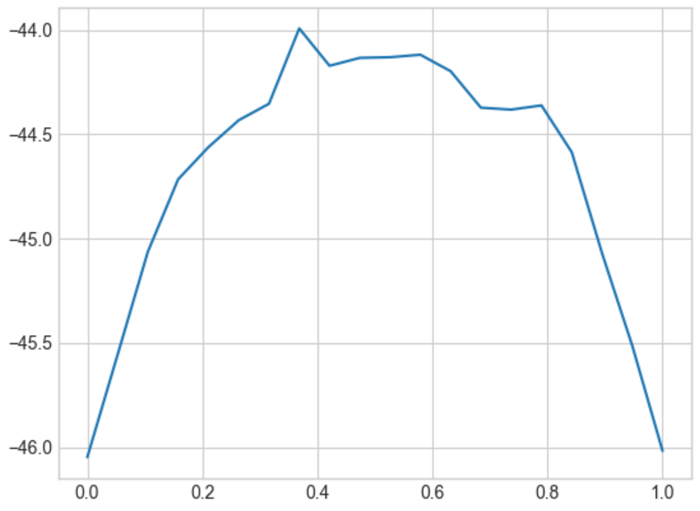

Figure 6: As percentage of GC content increases, the score initially increases, and then decreases after passing 50%. The Nussinov score is along the vertical axis, while the fraction of GC content is along the horizontal axis.