6.437 Project Part II Report
Michael Everett
May 10, 2019

Computational Efficiency

I identified a major computational bottleneck by profiling the code and finding 95.7% of the time was spent on computing likelihood.

In the original implementation, my system computed the likelihood of a proposed cipher function by iterating through the full ciphertext. For each pair in the ciphertext, I decoded the two characters and looked up the likelihood of that pair in the pre-loaded M matrix, then computed the log likelihood, and incremented a running sum for the proposed and previous likelihoods.

In the new version, I treated the count of each bigram as a sufficient statistic for the full ciphertext, at least for the task of computing likelihood of a mapping as a product of bigram probabilities. Before the first iteration, I iterated through the full ciphertext to build up a dictionary of (bigram, count) pairs (e.g., {'qx': 200, ' p': 6, …}). Because there are so many repeated bigrams (for common pairs like "th"), the number of bigrams in the ciphertext was much lower than the number of characters in the ciphertext, which provided a way to reduce the length of the for loop in each iteration's likelihood computation.

Given the ciphertext's bigram-count dictionary, d, and the English bigram log probabilities, a, the likelihood, L, is computed as,

$$L = \sum_{b \, \varepsilon \, ciphertext \, pairs} a[decode(b)] = \sum_{b \, \varepsilon \, bigrams} d[b] \ * \ a[decode(b)] \, .$$

The first summation expression for L is equivalent to the second because the ciphertext pairs are pre-grouped into bigrams instead of summing d[b] identical terms.

I measured the improvement in computation time from the new likelihood computation by running the test.py script, which dropped from 223 seconds to 11 seconds for the same number of iterations. This enabled me to use a larger number of iterations under the same time constraint, which increased the possible number of proposed cipher functions.

Handling Breakpoints

The implementation for Part I did not consider a breakpoint in the cipher function. My first implementation looked at the first/last 500 characters of the ciphertext as two different messages, and tried to decode each message individually. Iterations were split into three groups: sample to fix early cipher function, sample to fix later cipher function, and sample to adjust the breakpoint. After one of these samples was selected, the likelihood of the whole decoded message was computed to accept/reject that sample. The issue with this

implementation is if the breakpoint is wrong, the likelihood will be heavily skewed by all the wrong characters in the middle, causing small improvements in the cipher function at either end to be rejected.

Therefore, I stopped computing likelihood of the whole decoded ciphertext as the accept/reject condition, and I stopped sampling a new breakpoint every third iteration. Instead, I computed the most likely period/space mappings, for the early/late ciphertext blocks (still split by first/last 500 characters), then starting from the start/end of the full ciphertext, looked how far along the ciphertext each cipher function could decode “letter-period-space” tuples, without running into a violation of the rule, such as “space space” or “space period”. With that hard constraint from the rules, I could estimate a lower/upper bound on the breakpoint for the early/late ciphertext segments. Then, for long ciphertexts with >>1000 characters, at least one of the early/late ciphertexts would have a lot of characters, so I could estimate likelihood much more accurately, and reject/accept cipher function sample more successfully.

However, this approach provided only a coarse estimate of the breakpoint, in the form of a lower/upper bound. To get a point estimate, at the very end of the decoding, after both early/late cipher functions had converged, I computed the likelihood of each bigram of the full ciphertext using the early cipher function the whole time, and again using the late cipher function the whole time. Then I used np.cumsum to get the accumulated likelihood for each string (in forward direction for early ciphertext section, in reverse for late ciphertext section). I then added these two cumulative sums which creates a v-shaped total likelihood estimate (Figure 1), for each possible breakpoint value along the entire ciphertext. In practice, the curve always had a clear v-shape in the long passages I tested, so it was possible to select the breakpoint as the argmax of that curve.

Figure 1 shows a representative example of the obvious choice of breakpoint after both cipher functions have converged. There is a clear change in slope of the accumulated log likelihood in the forward and reverse directions (blue, green), which is even more clear in the sum (red).

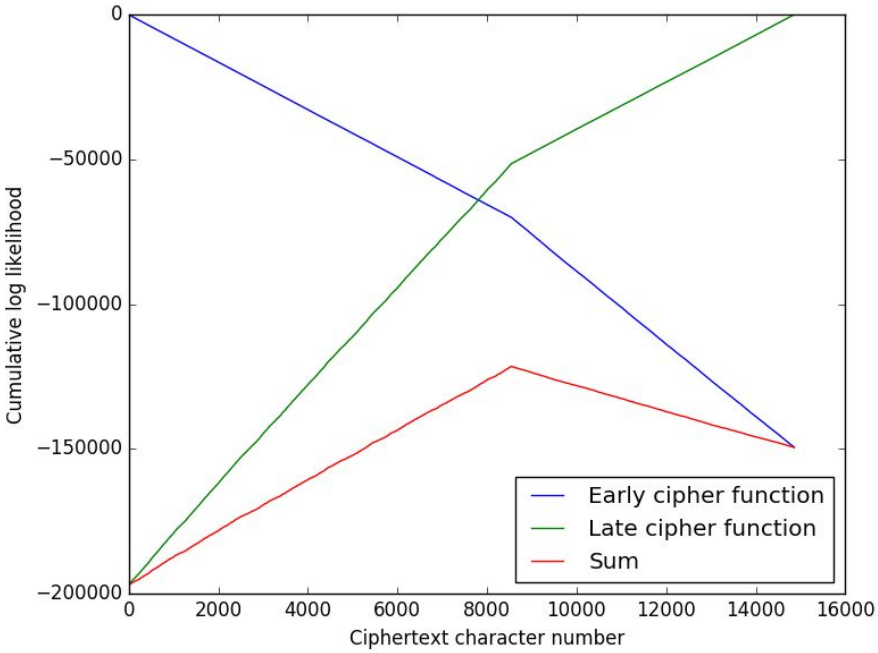

Figure 1: Breakpoint estimation

Lastly, when experimenting with setting the breakpoint very early/late in the message, my approach failed because it was computing the likelihoods from an arbitrary first/last 500 characters. Thus, I introduced the breakpoint estimator to occur every 1000 iterations (instead of only at the end), and updated the definition of the early/late ciphertext sections based on the latest breakpoint estimate. Occasionally, if the initial breakpoint estimate was so bad it caused poor choice for space/period, there was little chance of success to recover. So, I estimated whether the space/period choice was correct by computing the average word length and percent of words longer than 20 letters. If these exceeded well-known values for typical English passages (4.79 letters/word, <0.03% words > 20 letters [1]), I re-ran the period/space finder using the new breakpoint estimate. This led to better performance on ciphertexts with breakpoints very early or late in the message.

My system achieves high (but not perfect) accuracy on the test ciphertexts with and without a breakpoint:
Elapsed time: 23.5758538246 s
Score (no breakpoint): 5282 out of 5287
Score (with breakpoint): 5259 out of 5287

[1] http://norvig.com/mayzner.html
All code is posted at: https://github.com/mfe7/6.437/project