Cipher Breaking

Stephen Andrews

May 12, 2025

1 Initial Method

We start with an overview of the initial method that we implemented with no breakpoint. For this, we use a fairly standard implementation of the Metropolis Hastings algorithm based on Algorithm 20.2 in the notes. The states of our Markov Chain will be the possible permutations of the letters corresponding to the ciphers. Since our objective is the posterior $p_{f|y}(f|y)$, we define $p_\circ(f) = p_{y|f}(y|f)$. Next, our proposal function from any permutation will be a uniform distribution over all other permutations with exactly two letters flipped. Therefore, we get that for all $f \neq f'$ such that f and f' have two letters flipped,

$$v(f|f') = \frac{1}{\binom{|\mathcal{A}|}{2}} = v(f'|f).$$

Therefore, we get that our acceptance function is defined as follows:

$$a(f \to f') = \min(1, \frac{p_\circ(f')v(f|f')}{p_\circ(f)v(f'|f)}) = \min(1, \frac{p_\circ(f')}{p_\circ(f)}).$$

This guarantees that our converging distribution is the desired posterior function by using Algorithm 20.2. Turning this Markov Chain into a method for developing a decoded text is not too challening. One way to do it is as follows:

1. Pick some initial permutation f_i
2. Define $f^* = f_i$, $L_* = p_{y|f}(y|f_i)$, and $f_p = f_i$
3. Generate f' according to $v(f'|f_p)$ as defined above
4. Perform the Bernoulli sampling as in Algorithm 20.2 steps 5 and 6.
5. If we accept the new permutation update $f_p = f'$. If we also had that $p_{y|f}(y|f') > L^*$, then update f^* and L^* accordingly.
6. Go back to step 3 and repeat for some predetermined amount of iterations.

2 Breakpoint Adaptations

2.1 Brute Force

My first thought when approaching the breakpoint adaptation of this problem was to simply to test each possible gap in letters as the breakpoint. For each possible breakpoint location, we would run the algorithm defined in the previous section on each part of the text and record each of the log likelihoods and sum them to get the log likelihood

for the entire text. We would record the highest log likelihood and keep track of the corresponding breakpoint and permutations for each half. However, there are some clear issues with this implementation.

1. First off, this is extremely slow and would not run in the allotted 2 minutes given on Gradescope.
2. Second, is that as we saw in the first part of the project, the MH algorithm performs worse on shorter texts. As a result, when our breakpoint is either very early or late on in the text the algorithm above would perform well on the shorter chunk of text.

2.2 Optimizations

In the brute force implementation, we would evaluate every possible split point in the text, running the full Metropolis–Hastings decoder on each half. To reduce this cost, my first optimization was a simple striding scheme: rather than testing all N positions, I only considered breakpoints at regular intervals (for example, every 50 characters). This cuts the number of full decodings from N down to $N/50$, yielding a constant-factor speedup. However, since each candidate still requires two complete runs of the decoder, the overall time complexity remains linear in N, and the benefit is primarily a modest, text-specific improvement rather than a fundamental algorithmic advance.

A more substantial gain came from a binary-search–inspired approach augmented by a local refinement step. In the coarse phase, we repeatedly compare log-likelihoods at midpoints to cut the search range in half, reducing the number of full decoder runs from $O(N)$ to $O(\log N)$. Because each likelihood estimate is noisy, however, these large jumps can occasionally send the search off course. To stabilize the result, whenever we have a new candidate breakpoint, we locally shift the breakpoint to the left a few characters and to the right a few characters to see how drastically performance changes from local shifts. This fine-grained, local search corrects any misdirection introduced by randomness in the binary phase, preserving almost all of the speedup while more reliably pinpointing the true breakpoint.

2.3 R&D

There are a few design choices I made when testing my algorithm. None were particularly rigorous, but they allowed me to gain a good feel of the problem nonetheless. The first was deciding how many iterations to run the Metropolis Hastings algorithm, ie the "predetermined amount of iterations" in step 6 of the algorithm above. This was done almost entirely empirically. I made sure to balance between trying to get as close as possible to the converged permutation while also trying to save as much time as possible.

The other problem I had to solve was trying to reduce instability in the results of my algorithm. As mentioned in the previous section, an off the shelf implementation of Binary Search would have high variance in results depending on how far we skipped over the true breakpoint location. This was pinpointed as an issue by running test.py numerous times and seeing large changes in performance. In future testing, I would experiment by starting with a more intentionally chosen starting permutation as opposed to a completely random one as this would probably also decrease noise.

3 Results

My unadapted part 1 solution on the breakpoint test performed with about 58% accuracy. After changes, the binary search solution consistently achieved between 75-85% accuracy.