6.7800 Final Project

Annie Feng `azf@mit.edu`

Part II

High-level overview

I had two subroutines that handled the cases with and without a breakpoint. Sections 1 and 2 go into these two subroutines. Section 3 talks about the R&D process to arrive at these methods.

Algorithm 1 Decode

```
1: procedure DECODE_MAIN(ciphertext, has_breakpoint)
2:     if has_breakpoint then
3:         return decode_breakpoint(ciphertext)
4:     else
5:         return decode_enhanced_part1(ciphertext)
```

1 Enhancements to Part I

In my part I solution, I took advantage of numpy's optimized vector operations to reduce runtime. From previously, I handled computations in log-space when computing acceptance probabilities; I vectorized the implementations to sum over the tensor instead of using a for-loop over the length of the ciphertext.

I represented the substitution-cipher-permutation as a permuted array of 0 to N-1; so for example, 210 corresponds to a code from plaintext to ciphertext: 'a' -> 'c', 'b' -> 'b', and 'c' -> 'a'.

To vectorize my implementation, I used 'numpy.vectorize' to wrap the alphabet lookups in my ciphertext-to-plaintext operation (my 'inverse_vec' function); this alone doesn't give speed improvement, but this allows to index the tokenized substitution-cipher-permutation with a numpy array, hence a speed improvement: the ciphertext-to-plaintext operation was a frequent operation in my code.

1.1 Convergence

I used the steady-state value I found in part (e) to create my convergence criteria: once the log likelihood increases above a reasonable threshold to get within .8 accuracy (for me, experimentally around -3.45) or there have been 9000 iterations (large, compared to 3000 iters for length 2000 text in part I), I use the substitution cipher at that iteration to return the plaintext.

2 Handling Breakpoints

Since there is only one breakpoint, I initialize and run MCMC for two ciphers on each segment of the text. To select the breakpoint, I am using a uniform random search on either the left or right of the breakpoint, depending which of the two log-likelihoods was greater: if the cipher on the left segment has a lower log-likelihood, then should this breakpoint fail to converge, the next candidate breakpoint should be chosen closer to N to increase the size of the left segment; a lower log-likelihood indicates more over-fitting, which means the true breakpoint is likely to the right. Similarly, if the left segment has higher log-likelihood, the next breakpoint candidate should be to the left.

2.1 Convergence

My convergence criteria was for the sum of the log-likelihoods (per character) from both segments to be between -3.65 and -3.3; this seemed reasonable since my log-likelihood per character from part I, 3e, was -3.43, and the entropy of English should be around -3.23 for the bigram model.

2.2 Initializing the breakpoint and two ciphers

I use a while loop to find a breakpoint that gives rise to ciphers that get nonzero probability on the decoded sequence. I check that there are ciphers that have nonzero probability on the sequence by uniformly randomly selecting from the possible set of ciphers (permutations of length N), and deciding there are none after some threshold of ciphers sampled this way have been checked.

3 R&D Process

I used git to track progress on implementation of my methods (pure random search of breakpoint and guided random search by relative log-likelihoods). Ultimately the latter method performed better. In the end, I realized a third method: it'd be interesting to use MCMC to guess the breakpoint: breakpoints that had sum(ll per char of 1st segment + ll per char of 2nd segment) closer to the entropy of English should be weighted higher; I want to sample the next breakpoint closer to these ones. However, I'm still developing this third idea; the guided random-search method is sufficient for now.

Timeline / Progress Log for the 2 completed methods

First I optimized my implementation for Part I; I used the time library to benchmark each of my functions and found the most time-intensive computations to start working on; I also prioritized the subroutines that were called most often.

After that, I implemented the breakpoint code. I started out with replicating exactly my Part I code but with two ciphers, on two segments of the text. Then, I implemented shifting the breakpoint left or right on each iteration, but realized that this idea was half-baked and hinges a lot on the initialization. I then decided to randomly pick over all breakpoints each time I required a new breakpoint to try. However, this didn't work very well. I thought for a bit and combined the best of the two approaches into my final approach: the random search guided by relative log-likelihoods, as described in **'Handling Breakpoints'** above.

I also realized that for badly selected breakpoints, it's very hard to sample a cipher that gives nonzero probability for the decoded sequence. I utilized this to set a threshold for which if the number of consecutive uniformly selected sequences that have zero probability crosses, I will select a new breakpoint. I also discard these bad breakpoints from the candidate pool when selecting the next breakpoint.

Threshold tuning and convergence criteria

After the algorithm was settled, I played around with three thresholds. The two mportant ones are for convergence: the log-likelihood threshold, loosely around the entropy of English, for the breakpoint method and the same for the non-breakpoint method. Once the log-likelihood increases to this threshold, the decoded text is returned. The trade-off for a higher threshold closer to the theoretical one (-3.23) is that there may not be convergence possible for some texts, as seen in part I (-3.43 was achieved instead of Shannon's -3.23), or it will take too long to run. The third threshold to tune is the one that selects how to initialize the breakpoints and ciphers; a larger value would be ideal, but at the cost of runtime.

The non-breakpoint method had a threshold of -3.45. The convergence criteria was set to 9000 iterations or getting the log-likelihood above -3.45. The breakpoint method had thresholds -3.65 and -3.3; the log-likelihood had to be in this range to converge. Practically, the upper threshold isn't used, since the log-likelihood generally is below -3.3 for the correct ciphers, at least on the provided texts I tested on.

$\hat{f}_{MAP}(y) = argmax_{f \in F} \ \frac{p_{y|f}(y|f)}{\sum_{i=1}^{m} p_{y|f}(y|f_i)} p_{\mathbf{y}|f} = P(x_1 = f^{-1}(y_1))) \cdot P(x_2 = f^{-1}(y_2)|y_1) \cdots P(x_n = f^{-1}(y_n)|y_{n-1})$