6.437 Project Part II

Andrew Wu

May 2021

1 Introduction

This report builds on the original Part I project report, detailing improvements made to the basic Metropolis-Hastings decoder to handle "breakpoints" within some encrypted text. In this setting, a breakpoint is defined as some arbitrarily chosen location in the ciphertext where the cipher changes, from cipher 1 to cipher 2. The major improvement to the algorithm was the implementation of a "secondary" Metropolis-Hastings step, taking advantage of the same property of expected convergence towards the most probable breakpoint and pair of ciphers.

2 Algorithm

Recall that the original algorithm for Part I used the following Metropolis-Hastings algorithm:

Algorithm 1: Part I M-H Algorithm

initialize an arbitrary cipher f **while** *iterating* **do**
  Draw a random f' from the proposal distribution V(f', f)
  Compute acceptance factor a = min(1, $\frac{P(y|f')}{P(y|f)}$)
  Draw x from Uniform[0, 1].
  if $x \leq a$ **then**
    f = f'
  else
end
Return f applied to ciphertext

The choice of V(f', f) as the uniform distribution from f over all f' such that f and f' differ by exactly two symbol assignments allowed for the observation that V(f', f) = V(f, f'), and thus the acceptance factor calculation was simplified.

To modify this algorithm for the breakpoint situation, we considered the new problem of computing the MAP estimation of the posterior $P(f_1, f_2, b|y)$, where y is the given ciphertext and f_1, f_2 are the ciphers for the first and second sections, respectively. It is simple to observe, however, that we can simply expand our M-H algorithm to proceed over this expanded set of possibilities. In the original algorithm, we moved over the 28! possible permutations of a single cipher f, but now we can move over the 28!28!$|Y|$ possiblities of f_1, f_2, and breakpoint, b. Unfortunately, even the Metropolis-Hastings algorithm has difficulty converging within a reasonable number of iterations for such a large search space, so numerous augmentations were implemented, and these are described in the following sections.

3 Augmentations

3.1 Independent Calculations

To begin, we observe that it is unrealistic to base our Metropolis-Hastings algorithm off a single, joint step probability. That is, if we modify the acceptance factor to $P(y|f_1', f_2', b')$, for some f_1', f_2', b' generated at each step, then it is possible that we can select some "good" f_1' and f_2', but a "bad" b' and end up failing to step altogether. Thus, we note that if we move b gradually, it is likely that ciphers with high likelihood for the first and second halves will still be good ciphers after a modification to b. Similarly, this is true in the reverse. Thus, we split the M-H step into two smaller steps: the first, where we update f_1 and f_2 by a M-H step using the current choice of breakpoint, and the second, where we update b by the current choice of ciphers f1, f2. Note that the steady state distribution is still the same, only that this converges much more quickly, since each component moves on its own.

3.2 Local movement of breakpoints

Inspired by our choice of proposal distribution in 1), as mentioned before, we chose a "local" proposal distribution for moving breakpoints. That is, the distribution is uniform over the breakpoints within 15 symbols, modulo the length of the input cipher text. This is done to speed-up convergence, since once an optimum is found, the small size of the uniform distribution prevents the algorithm from leaving quickly; however, at the same time, it allows for sufficient exploration in the area.

3.3 Doing Multiple Independent Trials

Unfortunately, the $M-H$ algorithm tends to get stuck in local optima, and often fails to find the true maximum likelihood choice of parameters (this in part is due to our separation assumption in 3.1). To fix this, we run the algorithm multiple times, tracking the maximum likelihood choice of parameters (f_1, f_2, b) throughout all the trials. At the end of the multiple runs, the algorithm returns the decoded text computed from the maximum likelihood choice over all runs. This greatly increases the accuracy of our algorithm, taking advantage of the speed of the other calculations. Because the original algorithm could converge quickly within 10 seconds, this allowed for 10 independent trials to be done, and their results aggregated, which severely increased performance on the test cases.

4 Final Algorithm

Fortunately, the aforementioned three augmentations were sufficient to produce a decoder with extremely high performance. After augmentation 3.3 was implemented for both Part I and Part II, the scores were 0.996 and 0.949 respectively. The final algorithm is presented below.

Algorithm 2: Part II M-H Algorithm

initialize breakpoint b, arbitrary ciphers f_1, f_2
initialize best likelihood, breakpoint, f_1, and f_2
while *iterating* **do**
  Draw random f_1', f_2' from the proposal distribution V(f', f)
  For both ciphers, compute acceptance factor a = min(1, $\frac{P(y|f_i',f_j,b)}{P(y|f_i,f_j,b)}$)
  Draw 2 values from Uniform[0, 1].
  if $x_1 \leq a_1$ **then**
    $f_1 = f_1'$
  else
  end
  if $x_2 \leq a_2$ **then**
    $f_2 = f_2'$
  else
  end
  Draw a random value from uniform(-15, 15) and add it to b to get b'
  Compute acceptance factor a' = min(1, $\frac{P(y|f_1,f_2,b')}{P(y|f_1,f_2,b)}$)
  Draw a value from Uniform[0, 1]
  if $x' \leq a'$ **then**
    $b = b'$
  else
  end
  if *current likelihood* $\geq$ *best likelihood* **then**
    Update the best b, f_1, f_2, and likelihood to the current values.
  else
end
Return f_1 applied to ciphertext[:best breakpoint] concatenated with f_2 applied to ciphertext[best breakpoint:]

5 Brief RD Discussion

The most pressing issue was deciding how to handle breakpoints. Initially, I planned on guessing breakpoints, optimizing f_1 and f_2 over the breakpoint, and comparing the likelihoods of guessed breakpoints, similar to binary search. However, the methodology for implementing this algorithm was not very clear, and moreover, the runtime would be large, since f_1 and f_2 would have to be re-optimized each time. It made much more sense to take advantage of the observation that f_1 and f_2 should almost always take into account the likelihoods of the beginning and end of the text, since those sections are always associated with f_1 and f_2. Once this notion was formalized, it made sense to perform M-H over the space of breakpoints as well, leading to augmentation 3.1. Once augmentation 3.1 was implemented, test.py gave extremely good results, but the decoder's performance on gradescope was somewhat poor; however, its convergence rate was extremely promising. This then lead to augmentation 3.3, allowing for multiple independent runs to collectively return the correct answer.