Cipher Breaking using Markov Chain Monte Carlo

Roshni Sahoo, 6.437

April 27, 2019

1 Introduction

In this paper, we explore the use of the Markov Chain Monte Carlo (MCMC) method to decrypt text encoded with a secret substitution cipher. A *substitution cipher* is a method of encryption by which units of plaintext, the original message, are replaced with ciphertext, the encrypted message, according to an invertible ciphering function. The class of ciphering functions that we consider in this paper map a permutation of a 28-symbol alphabet to another permutation of the alphabet.

We discuss describe an MCMC algoritm to decode text encoded by a single substitution cipher. We also describe how to generalize this algorithm to decode text encrypted with *breakpoint-substitution cipher*, where a portion of the text is encoded by one substitution cipher and the remaining portion is encoded by a different substitution cipher. There is a tradeoff between accuracy and speed in this algorithm, and we justify our decisions for these tradeoffs. In addition, we detail the accuracy and performance improvements for our approach, as well as our testing procedure.

2 Decoding Ciphertext Encrypted with Substitution Cipher

Let M be a two-dimensional array of transition probabilities and P be a one-dimensional array of frequency probabilities. The likelihood of the (observed) ciphertext y under a ciphering function f, i.e.,

$$p_{y|f}(y|f) = P_{f^{-1}(y_1)} \cdot \prod_{i=1}^{N-1} M_{f^{-1}(y_{i+1})}, M_{f^{-1}(y_i)}$$

We can use the Metropolis Hasting algorithm to approximate the cipher function. We can create a proposal distribution as follows.

$$q(f'|f) = \mathbb{1}_{f,f' \text{ differ by 2 symbol assignments}} \cdot \frac{1}{\binom{28}{2}}.$$

The proposal distribution is uniform over all distributions that differ from f' in two symbol assignments.

Now we can run the Metropolis Hasting algorithm as follows.

1. Start with an arbitrary initial state f_0.
2. At time n, suppose a sample $f_n = f$ has been generated.
3. Generate a proposed new state f' as a sample of the random variable f' distributed according to the proposal distribution $q(\cdot|f)$.
4. Compute the acceptance factor
$$a(f \to f') = \min\left(1, \frac{p_{y|f}(y|f')}{p_{y|f}(y|f)}\right)$$
5. With probability $a(f \to f')$, set $f_{n+1} = f'$, else set $f_n = f$.
6. Iterate.

We can specify the pseudocode for the MCMC based decoding algorithm as follows. We can define

Algorithm 1 Decode Single Substitution Cipher

```
1:  procedure DECODE($y$)
2:      $f_0 \leftarrow f \sim$ Uniform(set of permutations of A)
3:      for $i \in 0, 1, 2, \ldots$ k do
4:          $f'_i \leftarrow f \sim q(\cdot|f_{(i)})$
5:          $a(f_i \to f'_i) = \min\left(1, \frac{p_{y|f}(y|f'_i)}{p_{y|f}(y|f_i)}\right)$
6:          $x \leftarrow x \sim$ Uniform$(0, 1)$
7:          if $x < a(f_i \to f'_i)$ then
8:              $f_{i+1} \leftarrow f'_i$
9:          if $x \geq a(f_i \to f'_i)$ then
10:             $f_{i+1} \leftarrow f_i$
11:     x $\leftarrow f_{\mathrm{k}}(y)$
```

3 Decoding Ciphertext Encoded with Breakpoint-Substitution Cipher

We explain how to generalize the earlier algorithm to decode text encrypted with *breakpoint-substitution cipher*, where a portion of the text is encoded by one substitution cipher and the remaining portion is encoded by a different substitution cipher. We also detail our stopping condition.

3.1 Algorithm

Let f_1, f_2 be the two ciphering functions for different portions of the text. Let b be the breakpoint. We can approximate the likelihood of the observed ciphertext y as follows

$$p_{y|f_1,f_2,b} = P_{f_1^{-1}(y_1)} \prod_{i=1}^{b-1} M_{f_1^{-1}(y_{i+1}), M_{f_1^{-1}(y_i)}} \cdot P_{f_2^{-1}(y_b)} \prod_{i=b}^{N-1} M_{f_2^{-1}(y_{i+1}), M_{f_2^{-1}(y_i)}}.$$

We extend our state-space to include the permutations for f_1 and f_2 and possible locations for b. We initialize f_1 and f_2 to be random permutations and set the breakpoint to be the midpoint of the text initially. Every iteration, we sequentially update f_1, f_2, and b in a Metropolis-Hastings step. Until the stopping criteria is met, we repeat.

In a Metropolis-Hastings step for f_i, we generate a f_i' from the proposal distribution of f_i'. As in the previous algorithm, the proposal distribution is stated as follows

$$q(f_i'|f_i) = \mathbb{1}_{f_i, f_i' \text{ differ by 2 symbol assignments}} \cdot \frac{1}{\binom{28}{2}}.$$

We decide to accept or reject the generated sample based on the acceptance factor, which we can write as follows. Let the ciphertext until the breakpoint be y_1 and after the breakpoint be y_2. Then,

$$a(f_i \to f_i') = \min\left(1, \frac{p_{y_i|f}(y|f_i)}{p_{y_i|f}(y_i|f_i')}\right).$$

In a Metropolis-Hastings step for b, we generate a b' from the proposal distribution, which we define as a Normal distribution about the previous breakpoint b with standard deviation of 20 characters. In practice, we sample from a normal distribution about the previous breakpoint and then take that value modulus the length of the ciphertext.

We decide to accept or reject the generated breakpoint based on the acceptance factor, which we can write as follows

$$a(b \to b') = \min\left(1, \frac{p_{y|f_1,f_2,b}(y|f_1, f_2, b)}{p_{y|f_1,f_2,b}(y|f_1, f_2, b')}\right).$$

3.2 Stopping Rule

We repeat iterations and return the best f_1, f_2, b after we have not improved best log likelihood in 1500 iterations or if we complete 10000 iterations.

4 Performance Improvements

Initially, it took over 382 seconds to run an iteration of Metropolis Hastings to find the breakpoint and the pair of ciphering functions. We found significant performance improvements by compacting the data structures used to store the ciphering functions and vectorizing the calculation of the log likelihood. We reduced the time for a single trial to 4 seconds on average.

4.1 Use Log Likelihood for Acceptance Factor

We can calculate the acceptance factor by using the log likelihood instead of the likelihood. This simplifies calculations because we can to sum the log of the transition probabilities instead of multiply.

$$\log p_{y|f_1,f_2,b} = \log P_{f_1^{-1}(y_1)} + \sum_{i=1}^{b-1} \log M_{f_1^{-1}(y_{i+1})}, M_{f_1^{-1}(y_i)} + \\ \log P_{f_2^{-1}(y_b)} + \sum_{i=b}^{N-1} M_{f_2^{-1}(y_{i+1})}, M_{f_2^{-1}(y_i)}$$

To do so, we store the log probabilities in M and P.

4.2 Replace Dictionaries with Numpy Arrays

Initially, we represented the ciphering functions with Python dictionaries mapping a permutation of the symbols of the alphabet to a different permutation of symbols. Not only does this require more operations to calculate the likelihood and more space, but it also prevents us from taking advantage of Numpy's vectorized calculations.

We replace the dictionaries with Numpy arrays storing a permutation of integers in the range from 0 to 27 inclusive.

This change did not cause a large speedup by itself, but it is useful because it allows us to vectorize with Numpy.

4.3 Vectorize Calculation of the Log Likelihood

In order to vectorize the calculation of the log likelihood, we must represent the ciphertext as an numpy array of indices into the ordered alphabet. For instance, $'a' = 0,' b' = 1$, etc. Using Numpy arrays, we can greatly improve the speed of the computation of the log likelihood using SIMD instructions (single instruction multiple data) by broadcasting the computation. This reduces the length of a trial from 70 seconds to 7 seconds, a 10x improvement.

4.4 Store Transition Counts of the Ciphertext for Single Substitution Ciphertexts

In the situation where the given ciphertext does not have a breakpoint, we can compute the transition counts of letters within the ciphertext to aid in computing the log likelihood. This allows us to compute the log likelihood with a simple matrix multiply calculation. For example,

$$\log p_{y|f} = \sum_{i}^{28} \sum_{j}^{28} T_{i,j} \cdot M_{f^{-1}(i), f^{-1}(j)}$$

This can be completed with a simple matrix multiply computation. Here, computing the log likelihood takes $O(|M|^2)$ time were $|M|$ is the size of the alphabet.

Note that we do not precompute the transition counts of letters when decoding ciphertext with a breakpoint because the transition counts shift for every new value of the breakpoint. In that case, it is more useful to just vectorize the iteration through the ciphertext, which is $O(n)$ where n is the length of the ciphertext.

4.5 Use a Gaussian distribution as the proposal distribution for the breakpoint

Initially, our implementation of the proposal distribution of the breakpoint was a discrete uniform distribution over the entire length of the ciphertext. We altered this to a Gaussian distribution centered at the previous breakpoint because as the algorithm converges, it is more likely that the best breakpoint is in a small neighborhood around the previous breakpoint. This yields slightly faster convergence, a 1 second improvement per trial.

5 Accuracy Improvements

We benchmarked that our algorithm for decrypting ciphertext with a breakpoint had an 30% accuracy rate across 50 trials. With the following changes with increase the accuracy rate to 99.9%.

5.1 Removing Zero-Probability Events

We notice that the Metropolis-Hastings algorithm can get stuck at a suboptimal local maximum. It is difficult to leave the local maximum if all transitions out of the state are equally unfavorable. Unless we run the algorithm for enough iterations, it outputs an incorrect answer. To prevent this from happening, we can perturb the zero-valued entries with a small random positive value. This reduces the number of local maxima that are difficult to transition out of by smoothening the log-likelihood space. This change yields a increases the accuracy rate to 60%.

5.2 Repeated Independent Trials

We define the correct answer to be 99% accuracy decoding on the ciphertext.

We measured that the Metropolis Hastings algorithm to learn f_1, f_2, b converges to the correct answer in 31 trials out of 50. We can increase the probability of success by running independent repeated trials and returning the ciphering functions and breakpoint that result in the best log likelihood. The probability of a suboptimal result across n trials is 0.38^n. We want to reduce the probability to 0.001, which is approximately 8 trials (rounding up). Thus, if we run this algorithm 10 times, we are likely return the correct answer 99.99% of the time.

Similarly, we also measured that the Metropolis Hastings algorithm to learn a single ciphering function converges to the correct answer in 43 trials out of 50. We can increase the probability of success by repeating this algorithm 5 times.

6 Testing

We benchmarked performance improvements and accuracy results on the encoded texts provided, but we also generated our own using an online substitution cipher. We used passages from *The Adventures of Huckleberry Finn* and from the Markov Chain Monte Carlo wikipedia page. We verified that the timing and accuracy conditions were consistent on these examples.

7 Acknowledgements

I would like to acknowledge that I discussed ideas and testing procedures with Ajay Jain and Bhavik Nagda, but we implemented the MCMC and writeup individually.