Cipher Breaking using Markov Chain Monte Carlo

Project for 6.437: Inference and Information

Xinyu Wu

Student ID: 922490927
Email: xinyuwu1@mit.edu

1 Problem Formulation

In this project, we focus on ciphering functions, denoted by $f(\cdot)$, as one-to-one mappings between symbols in two alphabets $\mathcal{A}$ and $\mathcal{B}$, both containing 28 symbols: ['a', ..., 'z', ' ', '.'], where the last two symbols are space and dot. Specifically, for a plaintext with length n, denoted as $\mathbf{x} = \{x_1, \cdots, x_n\} \in \mathcal{A}^n$, the ciphertext, $\mathbf{y} = \{y_1, \cdots, y_n\}$, created by $f(\cdot)$, should satisfy $y_k = f(x_k), \forall k = 1, 2, \cdots, n$. As $f(\cdot)$ is one-to-one, the corresponding deciphering function is $f^{-1}(\cdot)$. This project intends to devise an efficient and robust mechanism to find the correct deciphering function based on Markov Chain Monte Carlo (MCMC).

2 Deciphering Mechanism Design

2.1 Methodology

We apply MCMC, specifically Metropolis-Hasting (MH) algorithm, to estimate the ciphering function. We do not use deterministic inference methods, such as searching the Maximum A Posterior (MAP) estimator, since it can be shown to be infeasible in practice.

The design of MH algorithm involves two main tasks: (i) Define the state; (ii) Specify the proposal distribution. For (i), we regard each possible ciphering function $f(\cdot)$ as a state, hence theoretically there will be $|\mathcal{A}|!$ possible states. For (ii), the proposal distribution $V(\cdot \mid f)$, we focus on ciphering functions that differ in exactly two symbol assignments with f. We denote the set of such functions as $\mathcal{F}_f$. Specifically, we set $V(f' \mid f) > 0$ only for $f' \in \mathcal{F}_f$, and selecting each possible f' in a uniform way. As the size of $\mathcal{F}_f$ is $\binom{|\mathcal{A}|}{2}$, we have

$$V(f' \mid f) = \begin{cases} \binom{|\mathcal{A}|}{2}^{-1}, & f' \in \mathcal{F}_f \\ 0, & \text{o.w.} \end{cases}$$

Now we can formulate the MH algorithm based on the specification of $V(\cdot \mid f)$ above, shown as Algorithm 1, with details discussed in next part.

2.2 Algorithm Details and Improvement

We first clarify how to do calculations in Algorithm 1. Specifically it only involves the calculation of $p_{\mathbf{y}|f}(\cdot \mid f)$, as we have already gained other information. Notice that the second equation in Line 7 holds as we sample f' in a uniform manner. Specifically,

$$\begin{aligned} p_{\mathbf{y}|f}(\cdot \mid f) &= \prod_{i=1}^{n} p_{y|f}(y_i \mid f) = \prod_{i=1}^{n} \sum_{(x_1, x_2, \cdots, x_n) \in \mathcal{A}^n} \mathbf{1}_{y_i = f(x_i)} P_{x_1} M_{x_2, x_1} M_{x_3, x_2} \cdot M_{x_n, x_{n-1}} \\ &= \prod_{i=1}^{n} P_{f^{-1}(y_1)} M_{f^{-1}(y_2), f^{-1}(y_1)} M_{f^{-1}(y_3), f^{-1}(y_2)} \cdot M_{f^{-1}(y_n), f^{-1}(y_{n-1})}, \end{aligned} \tag{1}$$

which can be easily computed as $\mathbf{y}$ is given and f is one-to-one mapping.

Meanwhile, there are some designing details and improvements based on the original MH algorithm.

Algorithm 1: Metropolis-Hasting Algorithm for Deciphering

Input: Number of Trials T; Number of Iterations for each Trial T_c; Acceptance Rate Calculation Window W; Proposal Function $V(f' \mid f)$; Likelihood function $p_{\mathbf{y}|f}$

Output: Prediction of Cipher Function f_*

1 **for** $t = 1$ *to* T **do**
2 Initialize an arbitrary assignment f.
3 $Count \leftarrow 0$.
4 **while** $Count \leq T_c$ **do**
5 $Count \leftarrow Count + 1$
6 Generate a proposed new state f' based on $V(\cdot \mid f)$.
7 Compute acceptance factor

$$a(f \to f') = \min\left\{1, \frac{p_{f'|\mathbf{y}}(f' \mid \mathbf{y})V(f \mid f')}{p_{f|\mathbf{y}}(f \mid \mathbf{y})V(f' \mid f)}\right\} = \min\left\{1, \frac{p_{\mathbf{y}|f'}(\mathbf{y} \mid f')}{p_{\mathbf{y}|f}(\mathbf{y} \mid f)}\right\}$$

8 Accept f' with probability $a(f \to f')$.
9 If accept f', then $f \leftarrow f'$. Otherwise keep f unchanged.
10 Check the acceptance rate $\beta_k, k = 1, 2, \cdots$ every W iterations, i.e., check the number of acceptance in time interval $[(k-1)W + 1, \cdots, kW]$ in terms of $Count$ value. break the loop when for some $\tilde{k}$, $\beta_{\tilde{k}} = 0$.
11 **end**
12 Compare the likelihood $p_{\mathbf{y}|f}(\mathbf{y} \mid f)$ and $p_{\mathbf{y}|f_*}(\mathbf{y} \mid f_*)$, where f_* is the maximizer of the likelihood in previous rounds : If $p_{\mathbf{y}|f}(\mathbf{y} \mid f) > p_{\mathbf{y}|f_*}(\mathbf{y} \mid f_*)$, then update $f_* \leftarrow f$.
13 **end**
14 **return** f_*.

- **Multiple Initial Assignments**: The very outer *for* loop shows that we test with different initial assignments, as we observe that with different assignments, sometimes the corresponding final converged assignments do not match the correct one. This is because, we suppose, the existence of local extreme points such that whenever we achieve them, based on our local transition rule (i.e., randomly selecting the mapping that only differs in two symbol assignments) we can not jump out. The solution to this is that we compare the likelihood of the output estimated mapping for multiple initial assignments, and select the largest one, as Line 12 shows.
- **Stopping Rule for each Trial:** A trial under an initial assignment is shown from Line 3 to Line 11. Inside, there are 2 stopping rules we adopt. First, in Line 4 we set the upper bound of number of iterations of the state transitions, which is set to 5000 in our code as default. Second, More importantly, we examine the acceptance rate β, defined in Part I of the project, of every W iterations. Here we can view W as the window length for calculating the acceptance rate within this window. Normally, if such acceptance rate achieves 0, which means we *probably* (as there may exist multiple extreme points) find the correct estimation, then we break the *while* loop.
- **Efficient Way for Long Ciphertext:** As we observe, when the ciphertext is too long, for example the given ciphertext *warandpeace* containing more than $70,000$ characters, it will take more iterations for the algorithm to converge. Hence, we only use a small portion, for example the first $7,000$ characters, to feed into the MCMC procedure. After we get an estimation f from the MCMC, we test the likelihood $p_{\mathbf{y}|f}(\mathbf{y} \mid f)$. If the likelihood is 0, we discard this f, otherwise keep it.

These two modifications over the original MH algorithm make the estimation of cipher function much more efficient, and also ensure accuracy. Note that the parameters T, T_c, and W can be modified to balance the efficiency and accuracy.

3 Deciphering with Breakpoints

A *breakpoint* is a point that divides the text into two parts using different cipher functions, f_1 for left part and f_2 for right part. If such breakpoints exist, our idea follows 3 steps: (i) Identify the *left bound* of the breakpoint; (ii) Identify f_1 on the left part of such left bound; (iii) Extend the left bound to right step by step, and identify the place that the sudden decrease of likelihood increment happens conditioned on

f_1, which is our estimation of the place of breakpoint; (iv) Identify f_2 on the right part of the estimated breakpoint. We list details as follows.

1. We divide the ciphertext into two parts with equal length, and run our Algorithm 1 on each part. The one whose acceptance rate can achieve a fairly small value, say 0.3, indicates that it does NOT have a breakpoint, while the other one has[1]. If the one not having a breakpoint is on the left part, then set the middle place as the *left bound* of the breakpoint and stop this step, as the breakpoint must be on the right part of the left bound. Otherwise, we continue doing in this way on the part having a breakpoint (in fact always the left part), until we find that the part not having a breakpoint is on the left, and set such dividing point as the left bound.
2. Then we run our Algorithm 1 on the left side of the left bound, denoted as l_b to convergence, and output f_1.
3. Now we calculate $P_{\tilde{\mathbf{y}}|f_1}(\tilde{\mathbf{y}} \mid f_1)$ where $\tilde{\mathbf{y}} = \{y_1, \cdots, y_{l_b}\}$ initially. Then we extend $\tilde{\mathbf{y}}$ by appending characters on the right of the left bound in order one by one, i.e., $\tilde{\mathbf{y}} \leftarrow \{y_1, \cdots, y_{l_b}, \cdots, y_{l_{b+s}}\}$ at the s^{th} round of calculation, where $s = 1, 2, \cdots$. We examine when the likelihood value decrease significantly, or even to 0, this indicates that the breakpoint exists nearby, as f_2 is different from f_1 so deciphering by f_1 rather than f_2 on the right side of breakpoint will turns out some anomalies in likelihood calculation.
4. By (iii), we can identify the place of breakpoint approximately, so we can run Algorithm 1 on the right side of the estimated breakpoint to convergence and obtain f_2.

4 Evaluation Result

We test our algorithm on the given datasets including *test_ciphertext.txt*, *ciphertext.txt*, *ciphertext_feynman.txt*, *ciphertext_paradiselost.txt*,*ciphertext_warandpeace.txt*, all involving cases with and without breakpoints. If we set the parameters T, T_c, and W large enough, then all such cases can reach larger than 99% accuracy except the *ciphertext_paradiselost.txt* achieving 96%, as the transition probabilities at places around the breakpoint are small, thus difficult for our algorithm to distinguish the exact place of breakpoint.

However, to achieve higher efficiency, we test several combinations of parameter values, and identify that $T = 8$, $T_c = 6,000$, $W = 400$ is a good choice for both time and accuracy issues. This setting is the default value in our submitted code. However, sometimes if it is so unlucky that the algorithm randomly selects bad initial states in all T trials, or the method does not achieve a state close to the correct on in T_c iterations for each trial, it may still output a 'local' solution, achieving less accuracy, for example 80%, 73%, or even less than 50%.

5 Conclusion

In this project, we design a deciphering mechanism based on MCMC method, which can deal with ciphertexts with at most 1 breakpoint. We propose several improvements over the original MH algorithm, including trying different initial states to avoid being trapped in local extreme points, and a double stopping rule to ensure convergence and avoid running too long. We test our results on the given datasets from the Course 6.437.

[1] In fact the algorithm does not need to achieve convergence. If there is a breakpoint on one part, then the acceptance rate will keep around 0.8 to 0.9 based on our observation.