Decoding Substitution Ciphers Using the Markov Chain Monte Carlo Algorithm

Michael Lu

February 2023

1 Introduction

Students were given three scrambled texts with lengths around 5,000 characters. These texts only included 26 lowercase letters and the space character (S_{27}). These texts had been scrambled using a hidden **substitution cipher**-an algorithm that substitutes each character with another unique character. The algorithm used to decode the ciphers is the Metropolis-Hastings algorithm, a variant of the **Markov Chain Monte Carlo (MCMC)** algorithm. MCMC describes a set set of algorithms that randomly sample from a high dimensional space based on the current state of a sample. This is analogous to a random walk across a large graph with nodes representing unique states of the sample.

2 Methodology

Scrambled texts were selected and prepared before hand using a hidden substitution cipher. As mentioned before, these texts were each approximately 5000 characters long and contained a non-scrambled header "The coded text: `\n`".

A number of texts were mined to estimate $P_{true}(x)$ and $Q^2(x_1, x_2)$ for $x, x_1, x_2 \in S_{27}$. These texts can be found in the `\text_data` folder in code repository (https://github.com/mdlu02/MCMC). Some of these texts include *War and Peace* and *Pride and Prejudice.*

The first step of the algorithm was to extract and clean the text from the input data files. To read in `.pdf` files, I used the Python library `PyPDF2`. Texts were first converted to lower case before all extraneous punctuation and symbols were removed or replaced with spaces. After cleaning, only the characters in S_{27} were left in prepared texts. All cleaned texts were then concatenated into one string to be used later to calculate digram probabilities.

$P_{true}(x)$ was estimated by counting all occurrences of each character within the full string and dividing by the total length of the cleaned text. $Q^2(x_1, x_2)$

was calculated through iteration where each number of occurrences of each pair of characters was counted and stored in a matrix. Then, to generate the probabilities, these sums were divided by the total number of elements in each row. In order to help our algorithm generalize to potentially unseen sequences, all pairs of characters without a count received +1 before normalization. Since the data set used for mining was sufficiently large, the impact of this should be negligible to the overall distribution of the probabilities. Furthermore, this ensures that we do not run into issues with the negative log likelihood computation (see below).

I represented the current substitution cipher permutation on S_{27} using a string *perm* of length 27 with no duplicates in characters. The element $perm_i$ at each index i of *perm* represents the mapping from the element at index i in a reference string to the element at $perm_i$. The reference string used for this project was " abcdefghijklmnopqrstuvwxyz". At each iteration, a new permutation *curr* of *perm* were generated sudo-randomly by randomly sampling two integers on the range [0, 26] and swapping the corresponding elements of *perm* at the chosen indices. Both mappings *perm* and *curr* were applied to a given text to generate decoding which were then used to calculate their respective energies E using the following formula:

$$E(a)\text{=-}\ln P_{true}(f^{-1}(a_1)) - \sum_{i=1}^{n-1} \ln Q(f^{-1}(a_i), f^{-1}(a_{i+1}))$$

curr would be accepted as the new permutation if $E(curr) < E(prem)$ (i.e. $\Delta E < 0$) or with the random probability $P = e^{-\beta * \Delta E}$.

3 Methodology

I tuned β, the number of random swaps each epoch, maximum epochs without a change (for convergence), and the total maximum number of epochs. I found that my implementation of the algorithm performed best with two swaps per epoch and $\beta = 0.63$. I fit my model for 20,000 epochs with a 2,000 epoch convergence limit. I also implemented an additional check to prevent the algorithm from visiting the same permutation more than once between accepted changes. Below are the results from my decoding algorithm.

1. The permutation for `student_20_text1.txt` was σ("iwnbherascugfoz tpykvqjmxld") = " abcdefghijklmnopqrstuvwxyz". I identified this to be an excerpt from Nikolai Gogol's *Dead Souls.*

2. The permutation for `student_219_text2.txt` was σ("waduehiycfslb pvtokjgmqrnxz") = " abcdefghijklmnopqrstuvwxyz". Although I couldn't determine the exact source of this text excerpt, I believe that it is education material regarding medicare services.

3. The permutation for `student_102_text3.txt` was σ("calvwjde kzpxqbrmyigostunfh") = " abcdefghijklmnopqrstuvwxyz". I identified this to be an excerpt from James Joyce's *Ulysses.*

(Final decoded files submitted separately on Canvas)

4 Discussion

Overall, the decoded results appear qualitatively correct. There are minimal typos and the inconsistencies that are present appear mostly with less frequently used letters. While running the MCMC algorithm, I noticed that the model had a harder time with *Dead Souls*. Decoded results would occasionally take on very different forms and would get stuck in local minima corresponding to unreadable texts. Intuitively, this makes sense as this text borrows many names and words from Russian. If I had mined data from other novels by Russian writers, results for the text might have been more consistent. For the other texts, decoded results were much more consistent over multiple runs. Convergence generally took about 2-3 minutes running on my MacBook Pro although roughly readable results were generally achieved within the first 1-2 minutes. Computing my digram probability matrix took around 4 minutes given the large corpus of files I used. Each epoch of the algorithm has an $O(n)$ runtime where n is the length of the encoded text. This can be manually adjusted to only consider the first x characters in a text for the purposes of calculating energy. Thus, the total runtime is $O(nm)$ where m is the number of epochs. From discussion with other member of this class, I have a rough intuition that m is inversely related to n as the more letters we consider from the encoded text, the more accurate our energy calculation will be.