Decoding Substitution Ciphers

Patrick Peng

February 22, 2022

Instructor: Professor Menon

Abstract

In this project, the objective was to decode sample texts using the Metropolis-Hasting algorithm. This report describes the process of the decoding, from mining to employing the algorithm to tuning hyperparameters.

1 Objective

To decode the three samples of text using the Metropolis-Hasting scheme. Each sample is composed of 26 lowercase characters and the space character, and they have been scrambled using a substitution cipher.

1.1 Definitions

Subsitution Cipher A cipher where each letter is substituted for another unique letter.

MCMC Markov Chain Monte Carlo describes algorithms for methodically yet randomly selecting from a high-dimensional distribution based on the current sample. It can be likened to a systematic random walk that approaches some minimum.

2 Experimental Data

Scrambled Text H	4426 characters
Scrambled Text J	4279 characters
Scrambled Text F	4570 characters
Pride and Prejudice (Mining Text)	660264 characters

3 Methodology

The overall steps of the methodology are as follows:

a. Extract and clean text

b. Estimate $P_{true}(a)$ and $Q^2(a_1, a_2)$ for $a, a_1, a_2 \in S_{27}$

c. Apply MCMC with a random starting permutation to scrambled text until equilibrium is reached

d. Tune hyperparameters to optimize performance

In more detail, we begin by extracting the text from our PDF with the package `PyMuPDF`. As the text has many extraneous characters and still retains PDF formatting, we lowercase it, replace newlines with spaces, remove characters not in S_{27}, and remove extraneous whitespace.
To estimate $P_{true}(a)$, we count the total occurrences of each symbol and divide by the length of our cleaned text. $Q^2(a_1, a_2)$ is also generated by iteration through our text; for this we count the number of occurrences of each pair of symbols and then divide the elements of each row by the sum of the elements in that row to ensure the sum is 1. Before normalizing to the sum, though, we add 1 in each category with no occurrences. As our dataset is significantly large, this does not affect the overall probabilities significantly but ensures we do not have any errors with a possible $\ln 0$ in energy calculations.
I represented a permutation on S_{27} as a string σ of length 27 where $p_k \in S_{27} \notin \{p_0, ..., p_{k-1}\}$. For convenience, $\sigma_1 = c$ represented a mapping from a to c, $\sigma_2 = c$ represented a mapping from b to c, and so on, with σ_{26} and σ_{27} representing the elements z and space mapped to, respectively. Uniform random transformations to new permutations τ were achieved by swapping any two distinct characters in the string. $E(\sigma)$ and $E(\tau)$ were calculated by applying the permutation f to the scrambled text b and computing

$$-\ln P_{true}(f^{-1}(b_1)) - \sum_{j=1}^{n-1} \ln Q(f^{-1}(b_j), f^{-1}(b_{j+1})) \tag{1}$$

Following with Metropolis-Hasting's acceptance conditions, we accept if ΔE is negative and accept with $P = \exp(-\beta * \Delta E)$. To determine if convergence had been achieved, I counted how many epochs had gone without an accepted permutation change; if this exceeded a set bound then the algorithm would stop. However, I also implemented a large maximum epoch count just in case.
To achieve better efficiency, I tuned β, the number of random swaps between a proposed permutation and the original, and the number of epochs the permutation had to stay unchanged to declare convergence.

4 Results

I found that this algorithm performed well with $\beta = 0.7$, one random swap between every proposed permutation and the original, and 1000 epochs without accepted changes to determine convergence. I believe 1000 epochs performed

the best because there is a good chance most possible transformations will be tested in that duration.
Here are the correct permutations as provided by the algorithm:

a. For `f_29.txt`, an excerpt from Feynman's Lectures on Computation, this was the permutation:

   σ("ohfyazsplnqmdbexc kiwvjtugr") = "abcdefghijklmnopqrstuvwxyz "

b. For `h_29.txt`, an excerpt from Harry Potter, this was the permutation:

   σ("lnvtzmhawgosiujfxckqrbpdey " = "abcdefghijklmnopqrstuvwxyz "

c. For `j_29.txt`, an excerpt from Finnegan's Wake, this was the permutation:

   σ(" rdkqhugsaxyeptbmwjlcfozivn") = "abcdefghijklmnopqrstuvwxyz "

The texts will be submitted in a separate attachment.

5 Discussion

These results seem correct–there do not appear to be any typos and I was able to find the actual texts they corresponded to. I believe the algorithm had the most trouble with Harry Potter and Finnegan's Wake, as these books both employed strange words not commonly found in English, especially the latter (I originally thought my algorithm had converged incorrectly for Finnegan's Wake, but it turns out that's just how the book works.) In terms of runtime, the algorithm consistently achieved convergence within a few minutes, if not less than one. Assuming the probability matrix and values are precomputed, the main factor in this is the energy calculation, taking $O(n)$ time where $O(n)$ is the length of the text to be decoded. This step then has to be run m times in a loop. I believe m is somehow inversely dependent on n, given that the sample is reasonable English, but I don't have a solid reason beyond intuition.