PSTAT 160B Final Project

Harrison Hansen, Jiayue Chen, Jack Liu, Evan Pei, Hunter Marshall

March 28, 2023

1 Introduction

Markov Chain Monte Carlo (MCMC) methods are used to solve difficult combinatorial problems, often formulated in terms of optimization. Given a probability distribution π, the goal of MCMC is to simulate a random variable X whose distribution is π. The MCMC algorithm constructs an ergodic Markov chain whose limiting distribution is the desired π. Once the chain has been run long enough to converge, or nearly converge, to its limiting distribution, the algorithm outputs the final element or elements of the Markov sequence as a sample from π. In this project, we will proposing modifications to the decryption Python code given to us to try and optimize unscrambling a random message.

2 MCMC algorithms

2.1 Metropolis Hastings Algorithm Background

The Metropolis Hastings algorithm is the most common Markov Chain Monte Carlo method, and it forms the basis of this project. As an example we can let $\pi = (\pi_1, \pi_2, ...)$ be a discrete probability distribution. The Metropolis Hastings algorithm constructs a reversible Markov chain $X_0, X_1, ...$ whose stationary distribution is π. We can let T be a transition matrix for some irreducible Markov chain with the same state space as π which is used as a proposal chain, generating elements that the algorithm accepts/rejects. Assuming that at time n, the chain is at state i, the next step X_{n+1} in the transition mechanism for X_0, X_1 can be determined by a two step process:

1) Choose a new state according to T, choose j with probability T_{ij}. Where state j is now called the proposal state.

2) Decide whether to accept or reject state j. We can let

$$a(i,j) = \frac{\pi_j T_{ji}}{\pi_i T_{ij}}$$

This new function a is the acceptance function where if $a(i,j) < 1$ then j is accepted only with probability $a(i,j)$. If $a(i,j) \geq 1$, then j is accepted and is now the next state of the chain. However if j is not accepted, then i is kept as the next step.

X constructed by the Metropolis–Hastings algorithm is a reversible Markov chain whose stationary distribution is π. For $i \neq j$,

$$P_{ij} = \begin{cases} T_{ij}a(i,j) & \text{if } \pi_j T_{ji} \leq \pi_i T_{ij} \\ T_{ij} & \text{if } \pi_j T_{ji} > \pi_i T_{ij} \end{cases}$$

Now, we need to show local balance holds. Let's assume $\pi_j T_{ji} \leq \pi_i T_{ij}$.

$$\pi_i P_{ij} = \pi_i T_{ij} a(i,j) = \pi_i T_{ij} \frac{\pi_j T_{ji}}{\pi_i T_{ij}} = \pi_j T_{ji} = \pi_j P_{ji}$$

Let's assume now that $\pi_j T_{ji} > \pi_i T_{ij}$.

$$\pi_i P_{ij} = \pi_i T_{ij} = \pi_j T_{ji} \frac{\pi_i T_{ij}}{\pi_j T_{ji}} = \pi_j T_{ji} a(j,i) = \pi_j P_{ji}$$

Thus local balance holds, so the chain is reversible.

2.2 Cryptography

For this project, we use the Metropolis Hastings algorithm to decrypt a string in the English alphabet. Let Σ denote the alphabet, which is the set of all letters and symbols used to write in a language. Permutations can encrypt messages in the language on Σ.

Let $T = (t_0, t_1, ..., t_n)$ be a message of length $n+1$.

1) t_k is the kth character of the message
2) all characters belong to the defined alphabet σ
3) permutation σ maps t_k to σ_{t_k}
4) permutation σ maps T to σ_T

For example Σ = e,n,o,p,r,t,y and words that can be formed include: entropy, rope, tree, try, no, etc. In this example, Σ = e,n,o,p,r,t,y and the permutation σ = "n y t r o p e" so our message is "no rope try tree". The decrypted message T = "yt otrn poe ponn" can be translated correctly given the correct permutation. The permutation σ = "y e r t p o n" can decrypt the message T giving σ_T = "no rope try tree". Giving an incorrect permutation σ "p y t e r n o" would translate T into σ_T = "on tnry etp etyy".

Now, let $\ell(a,b)$ denote the prior probability of seeing the letter b given that the previous letter was a. (These may be estimated from some large body of text). This results in a $|\Sigma| \cdot |\Sigma|$ matrix of probabilities.

2.3 Guiding Principle

Given any message T and a (possibly wrong) decoding permutation σ, the likelihood of σ_T being in our language is

$$\mu_T(\sigma) = \sum_{k=1}^{n+1} \ell(\sigma_{t_{k-1}}, \sigma_{t_k})$$

The goal is to find the permutation σ on Σ for which $\mu(\sigma_T)$ is largest. Let $\mathbb{S}$ be the space of all permutations on Σ. For each $\sigma \in \mathbb{S}$ the value $\mu(\sigma_T)$ is nonnegative. So,

$$\pi(\sigma) = \frac{\mu_T(\sigma)}{\sum_{z \in S} \mu_T(z)} \quad \forall \sigma \in S$$

forms a distribution of nonnegative numbers on $\mathbb{S}$ that sum to 1. We want to construct a Markov Chain with stationary distribution π, or equivalently with stationary measure μ_T. As the chain runs, its entropy increases, gets closer closer to the stationary distribution, and closer to the permutations that decrypt T.

2.4 Base Algorithm

Our state space, $\mathbb{S}$, is given by all permutations on Σ. Now, randomly walk on the state space. At each state $\sigma \in S$ we apply σ to the message T. $\mu_T(\sigma)$ represents the likelihood that σ decrypts the message T and eventually $\sigma \in S$ that decrypts T. We can reduce the amount of transitions from each state by allowing the transition $\sigma \implies \sigma'$ only if they differ by two character positions. We can also improve this method by biasing our transitions toward the more "likely" states using $\frac{\mu_T(\sigma')}{\mu_T(\sigma)}$.

We consider the transition probability to be zero, or $\sigma \implies \sigma'$ with probability

$$p(\sigma, \sigma') = \binom{|\Sigma|}{2}^{-1} \min(1, \frac{\mu_T(\sigma')}{\mu_T(\sigma)})$$

for $\sigma, \sigma' \in S$ that differ by a switch of 2 characters.

$$q(\sigma, \sigma') = \binom{|\Sigma|}{2}^{-1}$$

is known as the proposal probability.

3 Modifications

3.1 Character Selection

A possible way to reduce the runtime of the code could be to limit the size of the state space, $\mathbb{S}$, which is given by all permutations on Σ. Potentially, many of the entries in the character transition frequency matrix obtained from our reference texts could be very small, to the point of insignificance. So, there might not be enough information regarding transitions between these characters. Let's say the sequence "Cz" is not contained within our reference texts, but it is in the text in which we are attempting to decode. Then the entry corresponding to "Cz" in the character transition frequency matrix could be zero, so our code would have difficulty decoding this part of the text.

One potential solution to this problem could be to combine certain entries in the matrix obtained from the reference text, such that the amount of information available is increased. For example, certain forms of punctuation tend to be more infrequent in text, in comparison to other characters. So, we could combine these rarely used characters into one "chaos character". For example, if a character is used at a percentage level below a previously defined α, we can assign that character to be a chaos character. Now, the transition probabilities can be arranged so as to incorporate all of the information arising from each form of punctuation being mapped to it.

Taking this idea of simplification one step further, we could convert all upper case characters into lower case. Upper case characters in formal text are only used in the beginning of sentences and for nouns, so they are less frequent. This lack of information could potentially skew the "true"

character transition frequency matrix. From a Linguistics perspective, this change might not be deemed proper, but in practice may lead to better results as the same word could have a capitalized initial letter just because of its placement in the sentence and nothing else.

3.2 Non-random starting state

In the given code, the initial state σ_{t_0} is chosen at random. However, if we were to choose an "optimal" starting state, the chain would be more likely to converge to the true permutation. If we match up the character frequencies in the text we are trying to decode with those in our reference text, this should give us a more optimal starting permutation.

Here is a python function that finds an optimal starting permutation.

```
def starter(code, freq1, freq2):
    if not isinstance(code, np.ndarray) or not isinstance(freq1, np.ndarray) or not
        raise ValueError("All inputs must be numpy arrays")

    temp1 = np.zeros((len(freq2), 1), dtype = np.object)
    temp2 = np.zeros((len(freq2), 1), dtype = np.object)

    for tt in range(len(freq2)):
        temp1[tt] = np.insert(freq1[tt], 1, 'Alph'[tt])
        temp2[tt] = np.insert(freq2[tt], 1, 'Alph'[tt])

    help1 = np.sort(temp1[:,1])
    help2 = np.sort(temp2[:,1])

    final = np.zeros(len(help1), dtype=np.int64)

    for hh in range(len(final)):
        xxx = np.where(help1 == 'Alph'[hh])[0].flatten()
        final[hh] = help2[xxx]

    return [help1, help2, final.flatten()]
```

3.3 Text Changes

Another change that aims at improving accuracy would be to deal with a target text size, as the text length increases we cannot expect the accuracy to improve. Thus, if we have a large sample, such as a novel, partitioning the text into smaller sections would likely yield a more accurate output when compared to inputting the raw length. Having a smaller text length would lead to multiple choices for the permutation of σ^{-1}. Then the program would choose one of these permutations with the lowest negative log-likelihood to be accepted and applied to encoded text for the final decryption. However, the target text also cannot be too small since there may not be enough information about the transitions to form any letter transition probabilities. Therefore a good start to achieve optimal accuracy would be to split the target text into 2 parts and decipher from there since it would increase the speed of convergence as it decreases the number of iterations to reach equilibrium.

3.4 Different Types of Text

Since we may have some prior knowledge or assumptions about the plain text we want, such as the language or the structure of the text. This knowledge can be used to inform the MCMC algorithm and improve the efficiency of the decryption process. For example, if we are using the novel War and Peace as the basis of the matrix, then the program would perform well on similar text such as novel from same author in the same era. Similarly, if we want to decipher more modern language message such as the encoded messages between prisoners, we need to have some prior knowledge of modern text to increase the accuracy of deciphering.

A method where this could be accomplished would be pulling the frequency matrix for characters from the text we are given to unscramble, and matching that up with a similar data set that we already have. The main idea here is that most likely similar texts are going to have similar word frequencies, and we could for example have a list of possible data sets ready to match with. Additionally, it would also work to run 2 or 3 different data sets for unscrambling, and at some future time K we choose whichever message has the lowest entropy and keep that one as our main message. This would work to ensure that regardless of which initial training set we use, we can ensure that it ends up working better for our message.

In this project, we are deciphering text written by college students about a statistics course's final project. So, if we were to use a text database of student reports relating to statistics, perhaps this would improve the final decryption results.

3.5 Chat GPT

Nothing written in 2023 would be complete without mentioning Chat GPT, and this one is no different. Our group noticed that after using the code to make an initial guess, Chat GPT does very well at unscrambling the rest of the message. We could include an API Call in the python script that sends our initial guess to Chat GPT, or something with similar programming, in order to unscramble the rest of the message and to fix any errors.

Unfortunately, access to OpenAI's API is restricted by a paywall, so we will not be able to demonstrate a complete implementation of this workflow. However, we do provide a function below that would be added to "deshuffling_utils.py", and called by the main script at the end.

```
import openai
openai.api_key = "your_api_key"

def decrypt_string(encoded_string):

    prompt = "Please decrypt the following message: " + encoded_string

    # Set up the GPT-3 API parameters
    completions = openai.Completion.create(
        engine = "davinci",
        prompt = prompt,
        max_tokens = 1024,
        n = 1,
        stop = None,
        temperature = 0.5,
```

```
    )

    # Extract the decrypted string from the GPT-3 API response
    decrypted_string = completions.choices[0].text.strip()

    return decrypted_string
```

As a proof of concept, we use the Python code to make an initial guess on deciphering the encrypted message. We will then include a screenshot of Chat GPT deciphering the remaining message, given the partially deciphered message from Bernstein's code. This will show that if we were to include an API call to Chat GPT in the code, it will be able to decipher the rest of the message, thus improving the overall functionality of the code. This does not work perfectly unless you already have the message somewhat unscrambled, as shown below. With a fully scrambled messsage, it is unable to produce a correct response. As such, once you are able to achieve some entropy threshold, ChatGPT is able to almost perfectly unscramble it. There needs to be some testing done to determine at what point can ChatGPT optimally decode some given message, but at the very least it can be used to polish up an already made guess.

Summary

The given code is efficient enough to decrypt and encrypt text, but small improvements can be made to enhance the accuracy and speed of the program. Suggestions range from limiting the state space, choosing an optimal starting state as opposed to a random σ_{t_0}, reducing the target text size to a length more manageable, and using a relative reference text. The most important optimization we could reasonably do would be enhancing our prior, which in this case is the training data set that our acceptance probabilities are based on. Additionally, implementing an API Call in the Python script would also be extremely efficient as the AI is able to fix a lot errors and unscramble text, a further point of optimization here would involve testing to find the optimal entropy before we can call ChatGPT to finish the rest of it.