CSC 395 Homework 4 Spring 2024

Due: Friday, April 5

In this homework assignment, you will be implementing a version of Metropolis-Hastings to help crack a substitution cipher. You should put your solutions in a file `hw4.py`, which will be submitted through the course gradescope.

Part 1. **Cracking a Cipher**

A substitution cipher is a cipher where letters are substituted for other letters in an attempt to scramble a message. The cipher takes the form of a mapping of original letters to scrambled letters, i.e., it is a permutation of the alphabet.

For example, if our original alphabet is $\{a, b, c, d, e\}$, then the cipher $\{c, d, b, e, a\}$ will result in the following mappings:

$$a \to c, \quad b \to d, \quad c \to b, \quad d \to e, \quad e \to a.$$

With this mapping, the original message "bad" would become the scrambled message "dce."

To unscramble the text and retrieve the original message, we will need a *reverse cipher*, the inverse of the cipher's mapping. In this case, the reverse cipher would be $\{e, c, a, b, d\}$. This would allow us to read the scrambled text "dce" and see that the 'd' becomes a 'b', the 'c' becomes an 'a', and the 'e' becomes a 'd', to return us to the original message "bad."

Our goal: Given a scrambled message, determine a correct reverse cipher to retrieve the original message.

The idea: If we have some random reverse cipher, we would like to design a measure for how well this reverse cipher unscrambles the text. This way reverse ciphers that do a good job of unscrambling the message will be more probable. We can then use Metropolis-Hastings to sample according to this measure to hopefully find a reverse cipher that unscrambles the message.

But how can we measure how well the reverse cipher unscrambles the text? The idea here is to find a way to measure how close the unscrambled message is to being "proper English."

One strategy to do this would be to count the letter frequencies in the scrambled text and compare that to the letter frequencies of "proper English." To obtain these "proper English" frequencies, we will use a large reference text (you have been provided with the full text of Tolstoy's War and Peace in the file `WarAndPeace.txt`.

Unfortunately, just comparing letter frequencies does not work very well in this context. The problem is that messages tend to be short (compared to War and Peace), and the odds that a few letters appear more or less frequently in the message than in typical English is fairly high: a message talking about pizza would have many more 'z's than a typical text.

So instead of comparing letter frequencies, we will compare the frequencies of letter transitions, i.e., the rate at which an ‘a’ is followed by a ‘b’, etc. We will choose to store these letters in a matrix M, where rows and columns correspond to letters, and the (i, j) entry in the matrix is the probability that the ith letter is followed by the jth letter. Again, we will create this matrix using a large reference text.

Now we need to determine how we should use this matrix to form our measure of “proper English.” On one hand, we could compute a transition matrix for our attempt at unscrambling the message (using our current guess of a reverse cipher to unscramble the message), then define our measure as a matrix norm between this transition matrix and the “proper English” transition matrix from our reference text. This could work, but again has problems due to the short length of most messages: a message talking about pizza will record many more $i \to z$, $z \to z$, and $z \to a$ transitions than you would expect.

So instead, we will define our measure as *the probability that the unscrambled text was generated randomly from the “proper English” transition matrix.* This probability will be the product of the probability that the first letter starts a message, the probability that the second letter follows the first, the probability that the third letter follows the second, etc. So if our unscrambled message is “Pizza”, then we would calculate the product of:

- The probability that the message starts with ‘P’.
- The probability that ‘P’ is followed by ‘i’.
- The probability that ‘i’ is followed by ‘z’.
- The probability that ‘z’ is followed by ‘z’.
- The probability that ‘z’ is followed by ‘a’.

This measure will give us much better results with a few slight modifications:

- First, we will drop the probability that the message starts with a given letter. The sequence of the letters is much more impactful than which letter starts the sequence, so dropping this term will not affect our accuracy. But keeping this term could have dangers: what if someone maliciously started their message with a ‘z’ (i.e., “zHello World”)? Since ‘z’ will have a very low probability of starting the word, we will end up assigning a very low score to the correct reverse cipher! So it is better to simply drop this term and rely on the sequence of transitions within the text.
- Second, we will artificially remove any 0 probability events. For example, the probability that a ‘q’ is followed by a ‘T’ in “proper English” should be 0. But that would mean any reverse cipher that creates an unscrambled message with a “qT” in it will have a measure of 0. On one hand, this seems desirable, since that reverse cipher is clearly wrong. But, the issue is that having a measure of 0 can result in Metropolis-Hastings getting stuck and being unable to proceed. To fix this issue, we will replace all 0s in our transition matrix M with a very small, but nonzero value (try e^{-20}).
- Third, when computing our measure, we will instead compute the exponentiation of the sum of the logs of the transition probabilities, rather than directly computing the product of the probabilities. Our messages can be long enough, and the probabilities small enough, that computing the direct product can lead to numerical errors. It is safer to compute the sum of the logs of the probabilities, then exponentiate.

Now that we have fully described our idea, let's formalize this into a specific Metropolis-Hastings approach.

- Solutions: Solutions to our problem will take the form of a reverse cipher, which is a permutation of our alphabet Γ (**for this assignment**, you should take the alphabet Γ to include all lower case letters, all upper case letters, the space character, the comma character, and the period character - **your code should be written to simply ignore any characters not in this alphabet**).
- Sampling Algorithm: If we currently have a reverse cipher X, we will create a proposed reverse cipher Y by randomly selecting two letters in X and swapping their locations. (Note: we could generate a new reverse cipher by randomly permuting the alphabet, but this would not create a proper Markov chain. It is necessary for the new proposed solution to depend on the current proposed solution.)
- $Q(X \to Y)$: Note that our sampling algorithm randomly chooses two letters in the alphabet and swaps them. So the probability of making that specific choice is the probability of choosing those two letters out of all of the letters in the alphabet Γ, which gives us

  $$Q(X \to Y) = \binom{|\Gamma|}{2}.$$

  Note that this Q is the same for all X and Y, so the Q terms will cancel entirely!
- Measure $\mu(X)$: For a given reverse cipher X, we first obtain its corresponding unscrambled text (this can be done by 'enciphering' the scrambled message using the reverse cipher). We then compute the following quantity (where $M(c_i, c_j)$ is the entry in matrix M corresponding to the probability that the letter c_i is followed by the letter c_j)

  $$\sum_{i=0}^{n-2} \log M(c_i, c_{i+1}).$$

  (Note that this expression is indexing from 0, and we are taking n to be the length of the message.) We can then exponentiate this sum to obtain $\mu(X)$.
- Acceptance Probability: The acceptance probability for Metropolis-Hastings is given by

  $$A(X \to Y) = \min\left\{\frac{\mu(Y)Q(Y \to X)}{\mu(X)Q(X \to Y)}, 1\right\}.$$

  But as we previously noted, $Q(X \to Y) = Q(Y \to X)$, so those terms will cancel, leaving us to compute the ratio $\mu(Y)/\mu(X)$. To compute this ratio, we again must be careful of numerical errors. Instead of computing $\mu(X)$ and $\mu(Y)$, then dividing, we will use logarithms and exponentiate at the end. If the reverse cipher X gives the unscrambled string $x_0x_1x_2\ldots x_{n-1}$, and the reverse cipher Y gives the unscrambled string $y_0y_1y_2\ldots y_{n-1}$, then we can compute the acceptance probability as follows:

  $$A(X \to Y) = \min\left\{\exp\left(\sum_{i=0}^{n-2} \log M(y_i, y_{i+1}) - \sum_{i=0}^{n-2} \log M(x_i, x_{i+1})\right), 1\right\}.$$

  This will avoid the numerical errors that can arise from the more direct computation.

- Stopping Condition: For this problem, we will simply use a maximum number of iterations as our stopping condition. For most reasonable messages, around 10,000 iterations should work.

Your task for the first part of this assignment is to write the code necessary to implement this Metropolis-Hastings algorithm. You should put all of your code in the file `hw4.py`. You will need to write code to perform the following:

- Randomly permute the alphabet (this is used to give a random initial reverse cipher to start the chain).
- Encipher a message using a given cipher, where each letter of the message is replaced by the corresponding letter in the cipher. Note: if you have a scrambled message and a reverse cipher, you can unscramble the message by 'enciphering' using the reverse cipher.
- Generate a new reverse cipher Y given a current reverse cipher X by randomly selecting two letters and swapping them.
- Create the transition matrix M from the provided reference text (`WarAndPeace.txt`). You should consider saving this matrix to a file and simply reading the matrix from the file when you go to perform the full algorithm. That way you avoid having to re-process the full reference text every time you use the algorithm.
- Calculate the acceptance probability for the proposed move from X to Y (note: you never have to actually compute $\mu(X)$ by itself).
- Generate the chain of 10,000 reverse ciphers using Metropolis-Hastings. Note: the algorithm might not terminate at the best solution, so you should track the best solution as you go and return that reverse cipher when you have hit the maximum number of iterations.

Once you have written this code, you should set up a test. Create your own message and cipher. Scramble the message, then use your code to decipher it.

Note: a good message should be roughly a paragraph or two in length. Longer is fine, but if the message is too short, you might have difficulty deciphering it.

As a final note, remember that this method is random, so it might fail to decipher the text even if implemented correctly. If it fails, try again several times. You should be able to decipher the message most of the time, so a few attempts should be sufficient.

Part 2. **Tricking a Metropolis Attacker**

Now that you have written your own Metropolis-Hastings algorithm to crack a substitution cipher, I thought it would be fun to consider the other side of the problem: suppose you are trying to send a message safely using a substitution cipher, what can you do to try to defeat a Metropolis attacker?

Now obviously one approach would be to use a stronger method for encryption. Our Metropolis attacker couldn't handle RSA encryption. But that feels like cheating. Let's think up ways to trick our attacker with modifications to the idea of the substitution cipher.

There are three classic ways to edit a string: substitution, insertion, and deletion. The first of these forms the basis of the substitution cipher. But the other two edits might give us something to work with.

Using only these three edits, try to think up some methods for altering your message (before or after the substitution cipher is applied) so that it is more secure against a Metropolis attacker. The key idea is to find a way to ensure that even the correct reverse cipher is scored very low according to the measure $\mu(X)$.

One thing to keep in mind: your method must be reversible. Remember that you are attempting to send your message to someone else. Your recipient must be able to undo your changes to obtain the original message.

In your `hw4.py` file, add a comment describing at least two methods for altering your message to improve the security and trick a Metropolis attacker. Then compare your different methods based on the following two criteria:

- Maximum Security: when pushed to their extremes, which method do you think will be able to provide the highest level of security?
- Efficient Security: which method provides the greatest amount of increased security for the least amount of work?

Once you have discussed these methods, choose your favorite and implement it in the code (you should also implement a procedure for your recipient to undo the changes given the correct reverse cipher). Test it against the Metropolis attacker you created in Part 1. How well does it seem to work?

As a final note about this exercise: there are no correct or incorrect answers here. Off the top of my head, I can think of over a dozen different approaches to confuse the Metropolis attacker, and I can think of some pros and cons of each. I am not looking for you to find the best method, and it is fine if your method is not able to trick your Metropolis attacker. I am mainly interested in how you think about Metropolis-Hastings and how your knowledge of the algorithm has helped shape your ideas here. And remember: this part is supposed to be fun. So have some fun with it!