Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science
6.7800 Inference and Information
Spring 2023

Project: Cipher Breaking using Markov Chain Monte Carlo

Issued: Wednesday, April 19, 2023 **Part I Due:** Wednesday, April 26, 2023
Part II Due: Friday, May 12, 2023

This project explores the use of a Markov Chain Monte Carlo (MCMC) method to decrypt text encoded with a secret substitution cipher.

Document structure The introduction provides some background on substitution ciphers. Part I guides you through a Bayesian framework for deciphering a substitution cipher and a basic MCMC-based approach for carrying it out. In Part II, you will further develop and refine your design from Part I.

Submission There will be three items you turn in for the project: 1) your writeup for Part I; 2) your writeup for Part II; and 3) your code submission. You will submit each to its corresponding assignment on Gradescope.

Time planning With a total amount of time roughly equivalent to two problem sets, you should be able to investigate the key concepts, get some interesting results, have the opportunity to be creative, and have some fun! Part I is quite structured, but Part II is open-ended. For Part I, we strongly recommend you start working on it as early as possible so that you have enough time to fully debug your code and produce accurate results. For Part II, you should only go beyond that approximate amount of effort to the extent that you have the time and are motivated to explore in more detail.

Collaboration policy As with the homework more generally, you are permitted to discuss concepts and possible approaches with one or two of your classmates. However, you must code, evaluate, and write-up your final solutions individually.

Don't forget! Completion of both parts of this project and the associated write-ups is a subject requirement.

Bonus We will share the best performing and most creative solutions on the last day of class in lecture.

Introduction

A *substitution cipher* is a method of encryption by which units of *plaintext* — the original message – are replaced with *ciphertext* – the encrypted message – according to a *ciphering function*. The “units” may be single symbols, pairs of symbols, triplets of symbols, mixtures of the above, and so forth. The decoder deciphers the text by inverting the cipher.

For example, consider a simple substitution cipher that operates on single symbols. The ciphering function $f(\cdot)$ is a one-to-one mapping between two finite and equal-size alphabets $\mathcal{A}$ and $\mathcal{B}$. The plaintext is a string of symbols, which can be represented as a length-n vector $\mathbf{x} = (x_1, \ldots, x_n) \in \mathcal{A}^n$. Similarly, the ciphertext can be represented as vector $\mathbf{y} = (y_1, \ldots, y_n) \in \mathcal{B}^n$ such that

$$y_k = f(x_k), \qquad k = 1, 2, \ldots, n. \tag{1}$$

Without loss of generality, we assume identical alphabets, $\mathcal{A} = \mathcal{B}$, so that a ciphering function $f(\cdot)$ is merely a permutation of symbols in $\mathcal{A}$. Throughout this project, we restrict our attention to the alphabet $\mathcal{A} = \mathcal{E} \cup \{$`" "`, `"."`$\}$, where $\mathcal{E} = \{$`"a"`, `"b"`, $\ldots$, `"z"`$\}$ is the set of lower-case English letters. That is, our alphabet for the project is the collection of letters `"a"` through `"z"`, space `" "`, and period `"."`. Henceforth any symbol to which we refer will be from this alphabet.

As an example, consider the short plaintext

`"this is a cool project."`

This plaintext is encrypted using a substitution cipher into the ciphertext

`".t qy qydywllpymnlsgw.b"`

where the ciphering function maps `"t"` into `"."`, `"h"` into `"t"`, `"i"` into `" "`, `"s"` into `"q"`, `" "` into `y`, `"."` into `"b"`, and so on.

Deciphering a ciphertext is straightforward if the ciphering function $f(\cdot)$ is known. Specifically, ciphertext $\mathbf{y} = (y_1, \ldots, y_n)$ is decoded as

$$x_k = f^{-1}(y_k), \qquad k = 1, 2, \ldots, n, \tag{2}$$

where the deciphering or decoding function $f^{-1}(\cdot)$ is the functional inverse of $f(\cdot)$. This inverse must exist because $f(\cdot)$ is a one-to-one function. However, when $f(\cdot)$ is secret (i.e., unknown), decoding a ciphertext is more involved and is more naturally framed as a problem of *inference*. In this project, you will develop an efficient algorithm for decoding such secret ciphertexts.

Part I

Problem 1: Bayesian framework
In this part we address the problem of inferring the plaintext x from the observed ciphertext $\mathbf{y}$ in a Bayesian framework. For this purpose, we model the ciphering function f as random and drawn from the uniform distribution over the set of permutations of the symbols in alphabet $\mathcal{A}$.

We assume that characters in English text can be approximately modeled as a simple Markov chain, so that the probability of a symbol in some text depends only on the symbol that precedes it in the text. Enumerating the symbols in the alphabet $\mathcal{A}$ from 1 through $m = |\mathcal{A}|$ for convenience, the probability of transitioning from a symbol indexed with j to a symbol indexed with i is given by

$$\mathbb{P}\left(\mathsf{x}_k = i \mid \mathsf{x}_{k-1} = j\right) = M_{i,j}, \qquad i, j = 1, 2, \ldots, m, \qquad k \geq 2. \tag{3}$$

i.e., the element in row i, column j of matrix $\mathbf{M} = [M_{i,j}]$ is the probability of transition from symbol j to symbol i in one step.

Moreover, we assume that the probability of symbol i in $\mathcal{A}$ is given by

$$\mathbb{P}\left(\mathsf{x}_k = i\right) = P_i, \qquad i = 1, 2, \ldots, m. \tag{4}$$

The matrix $\mathbf{M}$ and vector $\mathbf{P} = [P_i]$ are known.

(a) Determine the likelihood of the (observed) ciphertext $\mathbf{y}$ under a ciphering function f, i.e., $p_{\mathsf{y}|\mathsf{f}}(\mathbf{y}|f)$.

(b) Determine the posterior distribution $p_{\mathsf{f}|\mathsf{y}}(f|\mathbf{y})$ and specify a MAP estimator $\hat{\mathsf{f}}_{\mathrm{MAP}}(\mathbf{y})$ of the ciphering function f.

(c) Explain why direct evaluation of a MAP estimator $\hat{\mathsf{f}}_{\mathrm{MAP}}(\mathbf{y})$ is computationally infeasible.

Problem 2: Markov Chain Monte Carlo method
Given the difficulty of directly evaluating MAP estimators of the ciphering function, we turn to stochastic inference methods. As we learned in class, the MCMC framework is a convenient method for sampling from complex distributions. This problem guides you through the construction of a Metropolis-Hastings algorithm for decoding.

(a) Modeling the ciphering function as uniformly distributed over the set of permutations of $\mathcal{A}$, determine the probability that two independently drawn ciphering functions f_1 and f_2 differ in exactly two symbol assignments.

(b) Using the Metropolis-Hastings algorithm, construct a Markov chain whose stationary distribution is the posterior found in Problem 1(b).
Hint: Use Problem 2(a) for constructing a proposal distribution.

(c) Fully specify a MCMC-based decoding algorithm, in the form of pseudo-code.

Problem 3: Implementation
You will now code up the MCMC decoding algorithm! We recommend you use Python.[1] To help you code and evaluate your algorithm, we have provided a file, `67800project.zip`, that includes:

- `src/decode.py`: a starter file for your Python implementation.
- `data/alphabet.csv`: a vector of the alphabet symbols.
- `data/letter_probabilities.csv`: a vector $\mathbf{P}$ of occurrence probabilities of alphabet symbols in a plaintext as defined in (4).
- `data/letter_transition_matrix.csv`: a matrix $\mathbf{M} = [M_{i,j}]$ of transition probabilities as defined in (3).
- `data/sample/plaintext.txt`: a sample plaintext.
- `data/sample/ciphertext.txt`: a ciphertext obtained by applying a cipher to `data/sample/plaintext.txt`.

You can ignore the remaining contents until Part II.

Run the algorithm on `data/sample/ciphertext.txt`. Use `data/sample/plaintext.txt` to explore the convergence behavior of your algorithm.

In each of the parts below, please be sure that you fully and clearly label all curves and axes in your plots. You are not required to turn in your code for Part I.

(a) Plot the log-likelihood of the accepted state in the MCMC algorithm as a function of the iteration count.

(b) Plot, as a function of the iteration count t, the value over a sliding window of length T of the *acceptance rate* – the fraction of proposed transitions that are accepted. To be more precise, at iteration t, the acceptance rate over a sliding window of length T is

$$\frac{\#\text{ of accepted transitions from iteration } t-T+1 \text{ to } t}{T}.$$

Select the window length T appropriately.

[1]More specifically Python 3. This will allow you to reuse your code in Part II in preparing your code submission. The autograder that grades your submitted code only supports Python 3 (see later section for details).

(c) Plot the *decoding accuracy* as a function of the iteration count. The *decoding accuracy* at iteration t is defined as the ratio of the number of correctly deciphered symbols at iteration t and the length n of the plaintext.

(d) Experiment with partitioning the ciphertext into segments and running your algorithm independently on different segments. Specifically, experiment with different segment lengths. How is the accuracy affected? Why?

(e) How does the log-likelihood per symbol, in bits, evolve over iterations? How does its steady-state value compare to the entropy of English? Explain.
Note: The empirical entropy of English text depends on how the alphabet is modeled; for some insights, see, e.g., the classic paper

> C. E. Shannon, "Prediction and Entropy of Printed English," *Bell System Technical Journal*, 1951.

Part II

The goal of Part II is to extend the functionality of your decoder to support a *breakpoint*, as well as to improve the overall efficiency of your decoder, in terms of both computation time and the amount of text necessary for accurate decoding.

This part is intended to be open-ended. Beyond the required functionality extension, feel free to focus on the aspects of the challenge most interesting to you. We encourage you to be creative in applying what you have learned.

Please carefully read the submission guidelines and ensure that your submission adheres to them strictly. Since the evaluation is automated, it will fail if our scripts cannot find your files, for instance.

Introducing the breakpoint

In Part II, you need to extend your design from Part I to handle an adversarial scenario where the ciphertext includes a *breakpoint*, i.e., a location at which the cipher changes. As an example, consider the plaintext

`"this is a cool project."`

and its encryption

`".t qy qydywlzdemkzsvq.t"`

generated by a substitution cipher identical to that described in Part I up to and including the first `"o"` in `"cool"`, which then changes to the second cipher that maps `"o"` to `"z"`, `"l"` to `"d"`, and so on.

You can assume that every ciphertext has at most one breakpoint, which is placed at random. You will be told if the ciphertext has a breakpoint.

While developing your solutions...

You may choose to make use of the transition probabilities **M** and marginal probabilities **P** from Part I if you like, but you are not required to. If you do not make use of them, be aware that all evaluation plaintexts satisfy the following constraints:

- The plaintext is English.
- The plaintext starts with a letter from $\mathcal{E}$.
- A period (`"."`) is always preceded by a letter and followed by a space (if not at the end of the text).
- A space is always followed by a letter (if not at the end of the text).

It is also important for you to address how to stop your program; it should not iterate indefinitely. You need to develop a criterion for terminating your program when you are sufficiently confident of the decoded text. Problem 3(c) of Part I may, for example, provide some insight into this.

A lot of information about the English language that may be useful to you can be found at `http://norvig.com/mayzner.html`. This has (among other things) n-gram tables (i.e., frequency of occurrence of each possible combination of n letters at a time) for the English language, computed using a large repository.[2] Of course, you do not have to use this information.

To assist you in developing your decoder, the following additional files are provided in `67800project.zip`:

- `data/sample/ciphertext_breakpoint.txt`: ciphertext obtained by applying a cipher with a breakpoint to `data/sample/plaintext.txt`.
- `data/sample/short_*`: shorter tests, used in `test.py`.
- `data/texts/*`: some sample English texts.
- `src/encode.py`: a file for cleaning text and generating ciphertext with and without breakpoints.
- `decode-cli`: the script the autograder calls.
- `make_submission.py`: a script for zipping your code for Gradescope submission.
- `test.py`: a script for testing your submission.

The sample English texts are provided as convenient resources. The plaintexts that that we will use to evaluate your solutions do **not** come from these texts.

Code submission

To submit, zip your code and upload the zip file to Gradescope, where it will be autograded. You can use the `make_submission.py` helper script to zip your code. Run it via the command:

```
python3 make_submission.py
```

This will generate a file `submission.zip` that you can then upload to Gradescope. Manual zipping is also fine. If you go this route, make sure that `decode-cli` is at the top level of your archive.

Prior to zipping, we recommend you run the script `test.py` via the command

```
python3 test.py
```

[2] Some of the links to the n-gram tables are broken. However, the link to the zip file still works.

This tests your code locally. If these tests pass, your code has a good chance of working on Gradescope. Also as a rule of thumb, to properly zip your code you should run `python3 make_submission.py` from the same directory you ran `python3 test.py`.

If you are working on a Unix (e.g., MacOS or Linux) operating system, the above steps should work locally without too much fuss. If you don't have access to a Unix operating system, you can try using `https://replit.com` to test and zip your code. Just ask the course staff if you need any assistance with testing or submitting code.

Autograder details

The autograder calls `decode-cli` to evaluate your decoder. The file `decode-cli` should be a script which takes two command-line arguments:

- `ciphertext` is a ciphertext to be decoded, given as a string.
- `has_breakpoint` is a string, where `true` (case-insensitive) indicates that the ciphertext should be decoded as if it has a single breakpoint, and any other value indicates that the ciphertext should be decoded as if it has no breakpoint (i.e., the same cipher function is used for the whole text).

The script `decode-cli` should print the decoded text to `stdout` (e.g., using the standard `print` function in Python). Nothing else should be printed by your code since everything that is printed will be interpreted as part of the output.

A default version of `decode-cli` is provided that calls the `decode` function in `src/decode.py`.

Make sure that your submission contains all the files needed by the decoder, including any files from Part I. Remember to use *relative* pathnames so that your code can find the resources that it needs on the evaluation server.

Additionally, be sure to *remove all code related to figures* in your submission, such as calls to the functions `figure` and `plot`. These functions are not supported by the autograder and may lead to your code crashing.

Autograder environment

Your decoder will be evaluated on a Gradescope server with 4 virtual CPUs, 6GB of RAM, and an Ubuntu 18.04 OS. Your decoder will be run with Python 3.6.9 and will have access to the NumPy library.

Code scoring

The autograder will run your decoder on a collection of ciphertexts created from different plaintexts and ciphers. The evaluation ciphertexts range in length from 210 to 2100 characters. Some of the ciphertexts will have a breakpoint and some will not.

Your `raw_score` is the sum of the number of characters you correctly decipher across all test cases. Your `score` is your `raw_score` divided by sum of the lengths of all test cases (thus your score lies in $[0, 1]$). Your decoder performance grade (a number in $[0, 1]$) will be computed using the formula

$$\texttt{decoder\_perf\_grade} = \min\left(1, \frac{\texttt{score}}{0.80}\right).$$

In other words, you will get full points here if you decode 80% or more of the characters in the test cases.

For each test case, we will allot your decoder 2 minutes to finish executing. If your decoder takes longer than 2 minutes on a test case, you will receive a raw score of zero for that test case.

You will be able to see your score and decoder performance grade as soon as the Gradescope autograder finishes grading your submission. You will also be able to see more detailed scores for a number of public test cases. Sometimes the Gradescope servers may be under load and your submission may only run a while after you submit. The course staff have found that resubmitting sometimes speeds things up.

There will also be a global leaderboard of decoder performances. A pseudonym that you get to pick will be displayed in place of your name on the leaderboard.

Part II Writeup Guidelines

You will also need to submit a short (approximately 2-3 page) report of your decoder in the form of a PDF to Gradescope.

The report should include:

- A high-level overview of your decoding algorithm.
- A discussion of enhancements made to your part I solution.
- A discussion of how you handled breakpoints.
- A discussion of your R&D process, which could include things like how you tested your decoder, how you measured your decoder's performance, and how you decided what parts of your decoder to focus your efforts on.

Finally...

Have fun! We will highlight some of the best performing and most interesting solutions in class!