Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science
6.437 Inference and Information
Spring 2019

Project: Cipher Breaking using Markov Chain Monte Carlo

Issued: Tuesday, April 23, 2019 **Part I Due:** Tuesday, April 30, 2019
Part II Due: Friday, May 10, 2019

This project explores the use of the 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 to carrying it out. In Part II, you will further develop and refine your design from Part I.

Submission Part I is to be submitted in class on the due date. Part II is to be submitted via Stellar by the specified due date. Be sure to follow submission guidelines in Part II *carefully*, since some key parts of the evaluation are automated. Later in the document, we describe a test API provided to help you ensure your code passes our evaluation routine. You are expected to verify your code passes the provided tests before submitting.

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 Special distinction will go to the best deciphering code submissions, measured in terms of accuracy and efficiency.

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 $\mathcal{A} = \mathcal{E} \cup \{\text{␣}, .\}$, where $\mathcal{E} = \{\text{a}, \text{b}, \ldots, \text{z}\}$ 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 a 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 $\mathbf{x}$ 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 to $m = |\mathcal{A}|$ for convenience, the probability of transitioning from a symbol indexed with j to a symbol indexed with i is given by

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

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

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

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., $p_{\mathbf{y}|\mathsf{f}}(\mathbf{y} \mid f)$.

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

(c) Why is direct evaluation of the MAP estimate $\hat{\mathsf{f}}_{\mathrm{MAP}}$ computationally infeasible?

Problem 2: Markov Chain Monte Carlo Method

Given the difficulty of directly evaluating the MAP estimate of the ciphering function, we use stochastic inference methods. As we learned in class, the MCMC framework is a convenient method for sampling from complex distributions. This part guides you through the construction of a Metropolis-Hastings algorithm for the problem.

(a) Modeling the ciphering function f as uniformly distributed over the set of permutations of symbols from $\mathcal{A}$, find the probability that two 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 the MCMC based decoding algorithm, in the form of pseudo-code.

Problem 3: Implementation

This problem will require that you implement the decoding algorithm developed in Problem 2 in `Python`. To help you test, debug, and refine your code, and to analyze and evaluate how the algorithm performs, we have provided a file `6437project.zip` that includes:

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

In each csv file, the entries are separated by a comma. You can ignore the remaining contents until Part II.

Run the algorithm on `ciphertext`. Use the `plaintext` to explore the convergence behavior of your algorithm.

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

(b) Plot the acceptance rate of state transitions in the MCMC algorithm as a function of the iteration count. The acceptance rate $a(t)$ at iteration t is defined as the ratio between the number of accepted transitions between iterations $t - T$ and t and the overall number of proposed transitions, where window length T is appropriately chosen.

(c) Plot the accuracy rate as a function of the iteration count. The accuracy rate $\beta(t)$ at iteration t is defined as the ratio between the number of correctly deciphered symbols with the ciphering function at iteration t and the overall length 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 the English text? Explain. *Note:* The empirical entropy of English text depends on how the alphabet is modeled; for insights, see, e.g.,

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

Please make sure that you fully and clearly label all curves and axes in all your plots! Note: You are not required to turn in your code for (this) Part I.

Part II

In this part, your goal is to extend your design from Part I in two ways:

1. Improve the efficiency of your decoder, in terms of both computation time and the amount of text necessary for efficient decoding.

2. Extend your decoder to handle adversarial scenarios where the ciphertext includes a *breakpoint*, i.e., a location at which the cipher changes.

For 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.

As a reminder, this part is open-ended, giving you the opportunity to be thoughtful and creative in applying what you have learned.

Please read carefully 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.

While Developing Your Solutions...

You may choose to make use of the transition probabilities $\mathbf{M}$ and marginal probabilities $\mathbf{P}$ from Part I if you like, but you are not required to. If you do not make use of them, be aware that all text we will encrypt to test your algorithm will be formatted to satisfy the following four constraints:

- All text will be English.

- The text always starts with a letter from the set $\mathcal{E}$.

- A period ('.') is always followed by a space and always preceded by a letter.

- A space is always followed by a letter.

You may not assume anything else about the structure of the plaintext.

It is 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. Of course, you do not have to use this information.

To assist you in developing your decoder, the following additional files are provided in `6437project.zip`:

- `test_ciphertext_breakpoint.txt`: ciphertext obtained by applying a cipher with a breakpoint to `test_plaintext.txt`
- `data/plaintext_feynman.txt`: an example plaintext.
- `data/plaintext_warandpeace.txt`: an example plaintext.
- `data/plaintext_paradiselost.txt`: an example plaintext.
- `encode.py`: a file for generating ciphertext with and without breakpoints.

This is provided as a convenient resource in your development, and are *not* the plaintexts that we will use to evaluate your solutions.

Code Testing Guidelines

To confirm that your code can be evaluated by our automated platform, we **strongly recommend** that you test your code following the steps below before making your submission via Stellar. You should be able to perform this test from your own computer. We emphasize that this test only detects problems with your code that will lead to the system failing to evaluate your submission; thus, passing this test does not guarantee the correctness of the deciphered result.

(a) **Copy to Athena**: Copy the folder containing your code onto Athena. If you don't already have your favorite way of doing this, possible approaches for this step include using file transfer tools such as `SecureFX` for Windows and `Fetch` for Mac, which can be downloaded from

- 64-bit Windows: `http://ist.mit.edu/securecrt-fx/win64/recommended`
- 32-bit Windows: `http://ist.mit.edu/securecrt-fx/win32/recommended`
- MacOS X: `http://ist.mit.edu/fetch/5x/mac`

These tools from the IST website have been pre-configured with a session profile with which you can directly login to your home directory on Athena. Run these tools, login with your Kerberos account, and put the folder in your home directory on Athena.

An alternative method for copying that does not require any downloads is to use the terminal command `scp`, for example:

```
scp -r 6437project kerberos@athena.dialup.mit.edu:~/6437project
```

(b) **Test code on Athena**:

- Login to your Athena account. For remote login, you can visit `https://athena.dialup.mit.edu/` in your web browser.
- In the terminal, go to the folder that you copied to Athena in step (b). For example, if this folder `6437project` is put directly under your home directory, then you can execute the command below

  ```
  cd ~/6437project
  ```

- Execute the command below in the terminal

  ```
  python test.py
  ```

  Wait for your code to finish. If it creates a file `upload.zip`, then your code has passed this basic compatibility test. If there are error messages at the end (starting from a line with "`!!! ERROR !!!`"), then there are errors in your `Python` scripts that cause `Python` to crash or required files are missing from the path. **As a minimal check for compatibility, be sure that your code passes this test before submission!**

Submission Guidelines

There are two items you must submit for (this) Part II.

(a) Submit, via Stellar, the file `upload.zip` created in part (b) of the testing guidelines.

The file `decode` must 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 breakpoints (i.e., the same cipher function is used for the whole text).

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

Make sure that your submission contains all the files needed by the decoder, including any files from Part I that might be required by your decoder. And remember to use *relative* pathnames so that your `decode` function can find any other functions that it needs.

Additionally, be sure to *remove all functions related to figures* in your submitted code, such as `figure` and `plot`. These functions are not supported in the automated evaluation platform and thus will lead to a crash and prevent your code from being evaluated.

(b) Submit a short (approximately 2-3 page) description of your methodology and algorithm in the form of a PDF named `report.pdf` uploaded to Stellar. This description should include details of what refinements and enhancements you implemented to improve performance of the basic decoder, and a discussion of how you analyzed, evaluated, optimized, and tested it. This must be turned in by the submission deadline.

Code Evaluation

Each submitted decoder will be run on `Python 2.7.15` on Athena and applied to a collection of different ciphertexts we create from different plaintexts and ciphers.

For reference, a typical, well-engineered decoder you might submit should converge accurately within at most a few minutes. Nevertheless, we will allow each student submission up to 1 hour of run time, to avoid unduly penalizing inefficient but otherwise good implementations. This is the time allotted per student to start the program and call the function for *all* of the ciphertexts, one at a time. Hence, if your decoder doesn't have a built in stopping criterion, it will terminate after an hour, and the deciphered text in the output file (or files, if some ciphertexts have finished) at that point will be used.

NOTE: please do *not* estimate the run time of your code in the evaluation from its execution time on the Athena dialup server (`https://athena.dialup.mit.edu/`) used in code testing, since a different and more powerful server will be used for evaluation.

The criteria for the evaluation emphasize the accuracy of the deciphered results. We will measure how accurately your coded algorithm can decode each of the different ciphertexts we provide as input. More specifically, we will measure the accuracy rate for each ciphertext, defined as the ratio between the number of correctly deciphered symbols and the overall number of symbols in a plaintext. In addition to accuracy,

our evaluation will also consider the computational running time of your solution. In particular, the time your method requires to process each ciphertext.

Finally...

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