Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science
6.437 Inference and Information
Spring 2020

Project: Cipher Breaking using Markov Chain Monte Carlo

Issued: Tuesday, April 21, 2020 **Part I Due:** Tuesday, April 28, 2020
Part II Due: Friday, May 8, 2020

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 Part I is to be submitted via Gradescope. The written portion of part II should be submitted via Gradescope and the code portion of part II should be submitted via Stellar. Be sure to follow the submission guidelines in Part II *carefully*, since some key parts of the evaluation are automated. You are expected to verify that 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 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 $\mathcal{A} = \mathcal{E} \cup \{\texttt{" "}, \texttt{"."}\}$, where $\mathcal{E} = \{\texttt{"a"}, \texttt{"b"}, \ldots, \texttt{"z"}\}$ 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

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

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

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

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 estimate $\hat{\mathsf{f}}_{\mathrm{MAP}}(\mathbf{y})$ of the ciphering function f.

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

Problem 2: Markov chain Monte Carlo method

Given the difficulty of directly evaluating the MAP estimate 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 $\mathcal{A}$, find 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 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.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/test/plaintext.txt`: an example plaintext.
- `data/test/ciphertext.txt`: a ciphertext obtained by applying a cipher to `data/text/plaintext.txt`.

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

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

In the questions below, please make 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 the *acceptance rate* of state transitions as a function of the iteration count. The *acceptance rate* 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 the window length T is appropriately chosen.

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

[1] More specifically Python 3.

(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

> 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.

Evaluation

There are two components to your grade for Part II: the performance of your decoder and your written report.

Your decoder will be evaluated on a collection of ciphertexts created from different plaintexts and ciphers. The evaluation ciphertexts range in length from 256 to 2048 characters. Some of the ciphertexts will have a breakpoint and some will not.

For each input ciphertext, we will allot your solution 5 minutes to finish executing.[2] If your solution takes longer than 5 minutes, the *decode accuracy* for that input will be zero. The *decode accuracy* for a ciphertext is defined as the ratio between the number of correctly deciphered symbols and the overall number of symbols in the plaintext.

[2]Be careful if you estimate the run time of your code from the execution time on the Athena dialup servers (the ones at `https://athena.dialup.mit.edu/`). These servers are under variable load and are less powerful than our evaluation server.

Technical details of evaluation

Your submitted decoder will be run on our evaluation server which will have a Python 3.6.9 runtime.[3] Your decoder will also have access to the NumPy library.[4] The evaluation server has 4 physical 3.16 GHz cores (all with hyper-threading enabled).

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 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.[5] 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`:

- `data/test/ciphertext_breakpoint.txt`: ciphertext obtained by applying a cipher with a breakpoint to `data/text/plaintext.txt`.
- `data/test/short_*`: shorter tests, used in `test.py`.
- `data/texts/*`: some sample English texts.

[3]Since Python 2 has been sunsetted, the staff strongly encourages you to write your decoder in Python 3. However, to ease this transition, we will still provide a Python 2.7.17 runtime on the evaluation server. See the `decode-cli` file for instructions on how to have your solution evaluated using Python 2.

[4]Reach out to the course staff if you would like access to additional libraries.

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

- `encode.py`: a file for cleaning text and generating ciphertext with and without breakpoints.

- `decode-cli`: the script our evaluator calls.

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 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 the following file transfer tools:

- `SecureFX` on Windows: `https://ist.mit.edu/securecrt-fx/win64/recommended`
- `Fetch` on MacOS: `https://ist.mit.edu/fetch/5x/mac`

These tools from the IST website have been preconfigured 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 code folder into your home directory on Athena.

An alternative method for copying that does not require any downloads is to use the terminal command `scp`. You would run a command like the following:

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

(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

  ```
  python3 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 Part II.

(a) Submit, via Stellar, the file `upload.zip` created in part (b) of the testing guidelines. The evaluation code automatically unzips whatever zip file you submit, so it is important that you submit this folder directly, i.e., not inside of some other folder.

The file `decode-cli` 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 breakpoint (i.e., the same cipher function is used for the whole text).

`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 algorithm 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 `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 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) 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...

We will highlight some of the best performing and most interesting solutions in class! We hope you have some fun with this project!