Stat 650: Statistical Computing
Fall 2024

Miniproject **Due: 11:59 pm, Thursday, December 12, 2024**

In this miniproject, you will implement a statistical program to decrypt encrypted text. So far this semester, we have practiced specific computing and programming skills each week. The goal of this project is to integrate and synthesize those component skills by implementing and analyzing a slightly larger statistical program. Completing this assignment successfully is worth 15% of the final course grade (see the course syllabus for a detailed description of the computation of the final grades). The general guidelines for this project, given below, are the same as for a regular lab.

General guidelines:

You should hand in your code by pushing it to the repository associated with this assignment and then submitting the repository using Gradescope's GitHub integration. You will also need to produce a written project report outlining your work. The project report should give a *brief* description of the work you've done in this project and answer any specific questions given below. The project report needs to be submitted using Gradescope. There are no specific format or length requirements for the report, but it must look professional and address any questions given below. *Your grade is based on both the submitted code repository and the project report.* Only Gradescope submissions made before the above submission deadline will count toward your grade. **Remember to submit on Gradescope both your project report and your repository.**

We expect you to follow professional programming practices as outlined during lectures. Specifically:

1. We expect you to use Git for version control of your code. This means that as you work on this assignment, you should make multiple commits documenting your progress on the code. Using branching is optional, unless otherwise specified.

2. We expect you to follow good coding style. This means that we expect your code to pass `flake8` without warnings or errors. Your Gradescope submission will be automatically checked for this and the results of this check will be taken into account when grading your submission.

3. We expect you to document your code. This means that we expect you to use comments and docstrings throughout your code where appropriate. The expected input/output types should be documented using either docstrings or type hints. Using Sphinx is not expected.

4. Unless otherwise specified, we expect that you implement *reasonable* unit tests for your code. When you submit your repository on Gradescope, we will automatically run `pytest -v` on your code. We expect that the unit tests pass for your submitted code.

5. We expect that your code will raise exceptions when called with erroneous inputs or if other error cases occur. Your goal should be that in all (reasonable) cases your code either produces a correct output or raises an exception.

A reminder about collaboration and use of external resources in this course: You are allowed to discuss this assignment and work together with other students in this course unless otherwise noted; however, *you must produce your own code and write up your project report on your own.* You may not copy solutions, code, text or other materials from other students. You are allowed to use external resources (such as Google, Stack Overflow or ChatGPT) to help with your work, but they must *assist* your own thinking, not replace it. Specifically, you are not allowed to use AI to complete entire exercises (including subparts of exercises) or to write your project report. Any use of external resources must be clearly

indicated in your project report. Please see the course syllabus on Canvas for more details about these policies. Please ask the instructor first if you are unsure whether a particular form of collaboration or use of external resources is allowed.

Notice that you may need to look up resources and documentation online on specific Python commands and packages as your work on this assignment.

Miniproject assignment:

Start by accepting and cloning this repository: `https://classroom.github.com/a/ilH-qGWN`

Markov chain Monte Carlo (MCMC) samplers are algorithms that enable you to sample values from a probability distribution that is only known up to a normalization constant. In this assignment, you will implement an MCMC sampler for decrypting encrypted text as described in this publication: J. Chen and J. S. Rosenthal, Decrypting classical cipher text using Markov chain Monte Carlo, *Statistics and Computing* 22, 397–413, 2012. (You can download a copy of the manuscript through the CMU Library if you would like to have more background on the problem or more implementation hints in addition to what is given below.)

Encryption and decryption are important problems in computer science and mathematics. In this assignment, we will use simulations from a Markov chain to decrypt text. The basic strategy is to create a Markov chain (i.e., a stochastic process x_t whose distrbution at time t only depends on the previous time, $p(x_t|x_{t-1}, x_{t-2}, \ldots, x_0) = p(x_t|x_{t-1})$) that will asymptotically sample from a distribution that is maximized at a decoding that best "fits" the encrypted text. We run the chain for a while and collect the best-fitting decoding.

Let $c_1, c_2, \ldots, c_n$ be a sequence of n encrypted characters, and let $b(c, c')$ be the frequency of *character pair* (c, c') in a reference text.

We assume that we can decrypt our text using a *substitution decryption* d which maps a given character to another character bijectively. Then, $d(c_1), \ldots, d(c_n)$ is our decoded text under d. We do not know the mapping d so our task is to statistically estimate the most likely d.

We assign each decryption d a score

$$L(d) = \prod_{i=2}^{n} b(d(c_{i-1}), d(c_i)).$$

Large L is preferred, so we want to find the d that maximizes $L(d)$, at least approximately.

The Markov chain we construct will sample the space of decryptions d. In other words, we will construct a stochastic process that moves from one decryption to another according to certain rules. The chain is constructed so that it asymptotically samples d from the distribution $\pi(d) = \frac{L(d)^p}{\sum_{d'} L(d')^p}$, where $p > 0$ is a parameter. The algorithm (called the *Metropolis algorithm*) that produces a Markov chain with the required properties is as follows:

1. Choose an initial decryption key d (e.g., the identity mapping).
2. Repeat the following steps for many iterations (e.g., 10000):
   - Given the current decryption d, randomly sample a *candidate* decryption from a pre-specified *proposal distribution* $q(d'|d)$.
   - Sample an independent Uniform$(0, 1)$ random variable U.
   - If $U < \frac{\pi(d')}{\pi(d)} = \left(\frac{L(d')}{L(d)}\right)^p$, move to decryption d', else remain in decryption d.

As we let the algorithm run, we store the decryption key with the largest value of L. This will produce our best fitting decryption.

Notice that if d' has a larger value of L than d, then the algorithm will always move to d'. But if $L(d') < L(d)$, the algorithm will still occasionally move to d' with probability depending on the ratio $L(d')/L(d)$. The parameter p is a tuning parameter. It should be set such that the algorithm occasionally moves to d' even if $L(d') < L(d)$. A good first guess is to set $p = 1$ but you may need to adjust this value to obtain good performance.

You can use as the proposal distribution $q(d'|d)$ a distribution that randomly swaps the mapping of two characters in the decryption d to obtain d'.

Your task is to implement and investigate the performance of the above algorithm. Do the following:

1. Explain why a brute-force maximization of $L(d)$ is not a feasible strategy for solving the problem.
2. Take a moment to plan your software implementation. What modules, functions, methods or classes will you implement and how will they interact with each other? What data structures and algorithms will you use? Give a description of your plan in your project report.
3. Now write your implementation following your plan. As you write your code, use the built-in Python debugger in VS Code to help you with the task. Explain how you used the debugger and what benefit you got from it in your project report. Remember to document your code and to write sufficient unit tests to test your implementation. You may use AI to assist you in producing the code, but you are not required to do so. If you use AI, remember the guidelines given above. If you used AI, describe in your project report how you used it and whether you think it was helpful or not in this assignment.
4. Now try out your implementation using text samples from the Project Gutenberg (`https://www.gutenberg.org/`) collection of free books. Download a book of your choice in plain text format and encrypt a small section of it using a randomly generated substitution encryption. Then decrypt the encrypted text with your algorithm using the same book as the reference text. How well are you able to decrypt the text?
5. Investigate the properties of this algorithm: Does it still work if the encrypted text and the reference text come from two different books? Does the length of the encrypted text matter? How does p affect performance? How many iterations do you need to take? What other factors might affect the behavior of the algorithm? Report your results in your project report.
6. Once you are confident in your implementation of the algorithm and its properties, use your program to decrypt the file `some_text_encrypted.txt` provided in the GitHub repository. Save your result as `some_text_decrypted.txt` and push it to GitHub. Do you obtain a sensible result? Most English language books in the Project Gutenberg collection should be sufficient reference texts for decrypting this file.
7. Profile your code using both function profiling (`cProfile`) and line profiling (`line_profiler`). Which parts of your code take most time? Can you think of ways to optimize those parts of your code?
8. Once you are happy with your code and its performance, create a Python distribution package for your decryption algorithm. Make sure to include package metadata in TOML format. Build a wheel file for your package and push it into the GitHub repository.
9. Push your final files to GitHub, submit the repository on Gradescope and make sure that the automated checks in Gradescope pass for your code.

Here is some further guidance for your implementation:

- It can be a good idea to create a custom conda environment for this miniproject, especially if you have any trouble installing the profilers and build tools within your existing environment.

- Preprocess the text files by converting upper case letters to lower case and by removing all special characters, except for spaces. Convert line breaks to spaces and remove any consecutive spaces. After preprocessing, your text files should only contain lower-case characters a, b, c, d, e, f, ..., w, x, y, z and spaces.

- Consider only encryption and decryption keys that affect the letters, not the spaces. In other words, the spaces will not be encrypted / decrypted. Do include the spaces when you compute the score $L(d)$ based on the character pair frequencies $b(c, c')$.

- There are many ways you could potentially represent the decryption d in your code. A particularly effective representation is to view d as a permutation of our character set.

- Products of large numbers tend to be numerically unstable to compute. A numerically stable way to compute the product $L(d) = \prod_{i=2}^{n} b(d(c_{i-1}), d(c_i))$ is to compute on the log scale: $L(d) = \exp(\log(\prod_{i=2}^{n} b(d(c_{i-1}), d(c_i)))) = \exp(\sum_{i=2}^{n} \log(b(d(c_{i-1}), d(c_i))))$.

- `string.ascii_lowercase` in the `string` module provides a built-in collection of the 26 lower-case letters in the English alphabet.

- It can be a good idea to plot the score L as a function of the iteration number to get an idea of how the algorithm is exploring the space of decryption keys. The algorithm is working as intended if L first increases every few steps before stabilizing near a maximum value with small random up and down fluctuations.

- When computing the score L, it can happen that $b(d(c_{i-1}), d(c_i)) = 0$ for some i. This will yield $L = 0$ which will make it difficult to explore the space of decryption keys, especially at the beginning of the MCMC iteration. To get around this and to explore the space more efficiently, you can replace any zero values of $b(d(c_{i-1}), d(c_i))$ with ones. This is equivalent to computing the score as a product of $\max\{1, b(d(c_{i-1}), d(c_i))\}$, i.e., capping the factors from below by one.