6.437 Final Project: Decoder for a Subsitution Cipher

Zhening Li

May 2024

1 Introduction

A *subsitution cipher* encodes a string of symbols through a permutation of the alphabet. Formally, if $\mathcal{X}$ denotes the alphabet, then a subsitution cipher is a permutation σ on $\mathcal{X}$ defining an encoding function $f_\sigma(x_1x_2 \dots x_n) = \sigma(x_1)\sigma(x_2)\dots\sigma(x_n)$ for $x_1x_2\dots x_n \in \mathcal{X}^n$. When using a *breakpoint*, the input string is broken into two segments at the breakpoint and each segment is encoded using a potentially different substitution cipher. The encoding function $f_{\sigma_l,\sigma_r,b}$ is thus defined by two ciphers σ_l and σ_r as well as the breakpoint b ($0 \leq b \leq n$), so that $f_{\sigma_l,\sigma_r,b}(x_1x_2\dots x_n) = \sigma_l(x_1)\dots\sigma_l(x_b)\sigma_r(x_{b+1})\dots\sigma_r(x_n)$.

In this project, we implement an algorithm for decoding a ciphertext encoded using either a single substitution cipher ("no-breakpoint setting") or two substitution ciphers with a breakpoint ("breakpoint setting"). We use Bayesian inference by assuming the plaintext $\mathbf{x}$ and encoding function f are drawn from known priors, and we wish to recover the maximum a posteriori (MAP) estimate of f and $\mathbf{x}$ given the ciphertext $\mathbf{y} = f(\mathbf{x})$.

Focusing on English texts, we model the plaintext as a homogeneous second-order Markov chain with transition probabilities calculated from a corpus of the 100000 most common English words. Given a ciphertext, our decoding algorithm first applies Metropolis-Hastings (MH) to sample from the posterior distribution over encoding functions. Hundreds of runs are run concurrently, and the result with the highest log probability is kept. Next, we finetune the result by successively changing the encoding function to increase the number of decoded words that are in the top-10000 list of the most common English words. Several runs are run concurrently as well, and we output the result of the run that showed the most improvement in the number of valid decoded words.

Our algorithm achieves an accuracy of 99.83% on the GradeScope Leaderboard (99.98% in the no-breakpoint setting and 99.74% in the breakpoint setting), achieving first place among over 80 participants.

2 Method

Our algorithm has two stages: Metropolis-Hastings (MH) using a trigram model of English text, and a finetuning stage that spell-checks the decoded words.

2.1 Metropolis-Hastings with a trigram model of English

2.1.1 A trigram model target distribution

Using a uniform prior over encoding functions f, the posterior distribution over encoding functions given the ciphertext $\mathbf{y}$ is proportional to the likelihood $p_{\mathsf{x}}(f^{-1}(\mathbf{y}))$, where $\mathbf{x}$ is the random variable representing English plaintext. We thus use $p_\circ(f) = p_{\mathsf{x}}(f^{-1}(\mathbf{y}))$ as our unnormalized target distribution in MH. The distribution $p_{\mathbf{x}}$ over English plaintext is modeled as a trigram model, namely a homogeneous second-order Markov chain.[1] The transition probabilities are calculated from the trigram statistics in the Google Books Common Words dataset, which contains the 100k most common English words and their frequencies. Statistics of letter trigrams give rise to $p_{\mathbf{x}_i|\mathbf{x}_{i-1}\mathbf{x}_{i-2}}(x_i|x_{i-1}x_{i-2})$ whenever x_i, x_{i-1}, x_{i-2} are letters. When any of them are "␣" or ".", then the transition probability is estimated assuming that text is formed from words by independently selecting words from the Google Books empirical distribution of words, joining them together with either "␣" or ".␣". Here, each choice of "␣" vs. ".␣" is selected independently from the Bernoulli distribution with known parameter determined by the character frequency distribution given to us in `data/letter_probabilities.csv`.

2.1.2 The proposal distribution

In the no-breakpoint setting, the proposal distribution used in MH is a uniformly random transposition of the permutation defining the subsitution cipher. In the breakpoint setting:

- With probability $p_{\text{bp}} = 0.1$, we change the location of the breakpoint. Given the old breakpoint b, the new breakpoint b' is sampled from the binomial distribution $\mathcal{B}(n, p)$ where p is b/N clipped to the range $[1, n-1]$. To increase the acceptance rate, we resample b' whenever it is equal to b.
- With probability $1 - p_{\text{bp}} = 0.9$, we change one of the two subsitution ciphers chosen uniformly at random. The chosen cipher is changed by a uniformly random transposition of the permutation defining the cipher.

The choice of using a binomial distribution $\mathcal{B}(n, \text{clip}(b/N, 1, n-1))$ for the transition of the breakpoint is motivated as follows. Suppose that, whenever we change b, we shift b by a random amount from the range $[-O(\Delta b), +O(\Delta b)]$. As a rough model of the evolution of b thorughout MH, it first takes $O(n/\Delta b)$ iterations for b to evolve into the vicinity of the true breakpoint location. Then, it takes $O(\Delta b)$ attempts to have b land precisely in the right location. The total number of transitions involving b is thus estimated to be $O(n/\Delta b + \Delta b)$. This is minimized when $\Delta b = O(\sqrt{n})$, and the binomial distribution $\mathcal{B}(n, \text{clip}(b/N, 1, n-1))$ is just a simple proposal distribution with this Δb that has support over $\{0, 1, \ldots, n\}$.

2.1.3 Multiple attempts

The result of a single MH run has high variance due to the local maxima of the target distribution. To mitigate this issue, we run $A_{\text{MH}} = 512$ MH runs in parallel each for $N_{\text{MH}} = 5000$ steps.[2] The output of each run is the result of the last iteration as opposed to the best iteration as we did not observe better performance with the latter. Among all A_{MH} outputs, we keep the one with the highest log likelihood, which enters the finetuning stage.

2.2 Finetuning with spell-check

Given the decoded plaintext from the MH stage, we finetune the result using stochastic descent on the number of bad words. Here, a *bad word* is a word that doesn't exist in the `google-10000-english`

[1]The alphabet is taken to be $\mathcal{X} = \{\text{a}, \text{b}, \ldots, \text{z}, \text{␣}, .\}$, namely, we do not distinguish between lower- vs. uppercase letters, nor do we distinguish punctuation marks that mark the end of a sentence (all are denoted by the full stop); all other punctuation marks are simply removed from the plaintext.

[2]Since the grading environment has 4 vCPUs, our implementation splits the A_{MH} runs across 4 concurrent processes, each running $A_{\text{MH}}/4 = 128$ runs concurrently by batching the ciphers.

Benchmark	Overall		No breakpoint		Breakpoint	
	Accuracy	Time/test	Accuracy	Time/test	Accuracy	Time/test
Leaderboard	0.9983	∼14 s	0.9998	—	0.9974	—
3 Passages	0.9975	32.7 s	1.0000	28.3 s	0.9950	37.1 s
3 Passages (short)	0.8799	21.4 s	0.9685	18.8 s	0.7913	23.9 s

Table 1: The results of our substitution cipher decoding algorithm. The decoding accuracy is defined as the fraction of all characters over all test cases that are decoded correctly. We use $N_{\text{MH}} = 5000$ for Leaderboard and $N_{\text{MH}} = 10000$ for 3 Passages and 3 Passages (short).

dataset of the top-10000 most common English words.

In the no-breakpoint setting, two random non-space symbols in the alphabet are chosen and swapped, similar to the MH proposal transition. We accept the resultant plaintext if it has fewer bad words than the previous plaintext. Otherwise, it is rejected.

In the breakpoint setting, we choose the segment of text before the breakpoint with probability b/n, and we choose the segment after the breakpoint with probability $1 - b/n$. Then, as in the no-breakpoint setting, we choose two random non-space symbols of the alphabet to swap within that segment, and accept the resultant plaintext if it has fewer bad words. Note that we do not finetune the location of the breakpoint.

We output the result after $N_{\text{ft}} = 2000$ iterations.

The finetuning stage is repeated $A_{\text{ft}} = 4$ times in parallel using multiprocessing. We output the plaintext resultant from the run with the best improvement in the number of bad words.

3 Results

We test our algorithm on 3 benchmarks:

- "Leaderboard": the GradeScope Leaderboard consisting of around 20 test cases each of length 210–2100 characters;
- "3 Passages": 89 test cases of length 40–400 words generated from the 3 passages given to us in `data/texts/`;
- "3 Passages (short)": 909 test cases of length 4–39 words generated from the same 3 passages.

The results are in Table 1. While we are able to achieve near-perfect accuracy on inputs that are at least about 40 words or 200 characters long, accuracy drops when the input is short as in 3 Passages (short). This is the main reason the accuracy is lower in the breakpoint setting for inputs of the same length, since introducing a breakpoint splits the passage up into two segments of shorter length, one of which can be so short that decoding becomes very difficult on that segment.

Appendix A: R&D process

Based on my Part I code for the no-breakpoint setting (single MH run with a bigram model), I began by modifying MH to incorporate the breakpoint. Various transition kernels for the breakpoint resulted in high variance in the performance on the GradeScope Leaderboard due to the randomness, with the accuracy varying between 0.5 and 0.75. I then suddenly realized I can simply conduct several MH runs and take the best result, which both increases the performance and reduces the variance. I decided to use a binomial distribution transition kernel for the breakpoint based on my argument from Section 2.1.2.[3]

By this point, my overall score was 0.9326. I started optimizing my code so that I can run MH for more iterations or for more attempts to further improve accuracy. First, I noticed from the project specification that the grading environment provides 4 vCPUs, thus opening up the option of multiprocessing. This allowed me to increase the number of MH attempts from 8 to 32, increasing my overall score to 0.9729.

When talking to Antti Asikainen about an inefficiency in my code (a Python list comprehension in computing the log likelihood of the data), he suggested using NumPy operations, which led me to discover the option of using NumPy indexing and summation. This made my code run 5–10 times faster, allowing me to further increase the number of MH attempts to 200. My overall score was now 0.9782. Through hyperparameter search over p_{bp}, I was able to improve this to 0.9870.

I started investigating the error cases in the default test cases we are given for the no-breakpoint setting. With Adithya Balachandran's help, I realized that the bigram model results in the wrong ML estimate for some uncommon letters (e.g., "␣qust" has a higher probability than "␣just" under this model). This motivated using either a better model of English such as a trigram model, or to finetune with spell-check. Antti had told me he was going to try spell-check, and it's easier to implement as well, so I decided to try that first. Even with a preliminary implementation that ignores the breakpoint, spell-check brought my no-breakpoint score to a perfect score and increased my breakpoint score as well, resulting in an overall score of 0.9919. After supporting breakpoints and some small tweaks, my score was now 0.9960.

Afterwards, I implemented the trigram model. This took a while as I also needed to cleverly compute the trigram transition table from a word frequency list, which does not contain spaces or periods. The trigram model increased my score to 0.9979.[4] Multiprocessing for 4 concurrent runs of spell-check finetuning then slightly increased that to 0.9981.

0.998 was basically the maximum score I was able to obtain. My later endeavors to raise that to a perfect score all failed, but I will briefly describe them anyways so that the reader can understand what does *not* work.

I made my own test cases (3 Passages and 3 Passages (short)) to study error cases, as well as tried various techniques to improve the decoding accuracy further, but none were successful. For the MH stage, I tried changing it to simulated annealing. I also tried taking the best iteration instead of the last iteration. I realized these approaches don't improve accuracy because the likelihood function has very sharp local maxima, so no annealing is required to converge to the maximum. For the finetuning stage, I tried the following.

- Compile a list of words that are one letter swap away from a real word, so that during finetuning I can incorporate improvement from a bad word that becomes within Hamming distance 1 of a real word.
- Allow changing the breakpoint location.
- Allow swapping a symbol with the space character.

[3] I did eventually conduct ablation analysis to justify this choice, although this was done much later on my final algorithm: shifting b by ± 1 results in an accuracy of 0.9136; uniformly choosing b from $[0, n]$ results in 0.5498; a normal distribution with fixed $\sigma = 20$ results in 0.9972. The best transition kernel indeed has $\Delta b \sim 20 \sim \sqrt{n}$ for the Gradescope test cases ($n \in [210, 2100]$), and allowing Δb to depend on n by using $\mathcal{B}(n, b/n)$ results in an even better accuracy of 0.9983.

[4] Although this increase may seem insignificant, ablation analysis shows that the improvement due to the trigram model is more significant when we don't use spell-check finetuning and have fewer MH attempts.

- Penalize the wrong average length of words while allowing swapping the space charactor with other symbols.
- Use a longer list of the most common English words (100k instead of 10k).
- Use the sum of Hamming distances from the word list as the objective function, and optimize it using MH (with negative log probability equal to the objective function) or simulated annealing.

I concluded that the reason all these attempts failed is that the previous MH stage already brings us to the vicinity to the correct answer most of the time, so that we can simply use descent on the number of bad words without needing to worry about local maxima. If MH wasn't able to bring us to the vicinity of the correct answer, then we don't have enough compute to use spell-check to find the correct answer anyways. (One iteration of spell-check finetuning is a lot more costly than one iteration of MH.)

At the end, I made one more optimization, i.e., switching from running 200 processes for the MH runs to running 4 processes each with size-64 batches. Since the ciphers are represented as NumPy arrays, they can easily be batched to make the code almost twice as efficient.

Finally, I conducted hyperparameter tuning to arrive at my chosen hyperparameters: $p_{\mathrm{bp}} = 0.1$, $N_{\mathrm{MH}} = 5000$, $A_{\mathrm{MH}} = 512$, $N_{\mathrm{ft}} = 2000$, $A_{\mathrm{ft}} = 4$. See the next section for more details . Hyperparameter tuning slightly improved my score from 0.9981 to 0.9983.

Appendix B: Hyperparameter tuning

While I did a bit of hyperparameter tuning here and there throughout my R&D process, I also conducted more systematic partial grid searches at the end, which is presented below. It was found that:

- Performance is most sensitive to p_{bp}, the fraction of MH transitions that move the breakpoint. It should be 0.1 for optimal performance.
- Performance plateaus after the number of iterations N_{MH} of MH is high enough (5000–10000).
- Performance plateaus after the number of iterations N_{ft} of finetuning is high enough (around 2000).
- It is important for the number of attempts A_{MH} of MH to be high, but performance also plateaus beyond a certain point which is around 500.
- It suffices for A_{ft} to be a small integer greater than 1, such as 4. Increasing the number of attempts of fine-tuning to something larger does not improve performance.

The following is the series of partial grid searches conducted for hyperparameter tuning.

Fixing $p_{\mathrm{bp}} = 0.1$, $N_{\mathrm{MH}} = 5000$, and $N_{\mathrm{ft}} = 2000$, a grid search was conducted for:

- $A_{\mathrm{ft}} \in \{1, 4, 8\}$
- $A_{\mathrm{MH}} \in \{32, 64, 128, 256, 512, 768, 1024\}$

We found $A_{\mathrm{ft}} = 4, A_{\mathrm{MH}} = 512$ to be optimal. Fixing these parameters, as well as $p_{\mathrm{bp}} = 0.1$, $N_{\mathrm{MH}} = 5000$, we searched over N_{ft}:

- $N_{\mathrm{ft}} \in \{1000, 2000, 4000\}$

The optimal N_{ft} was found to be 2000. Finally, we fixed all the optimal parameters we've found and conducted grid search for:

- $p_{\mathrm{bp}} \in \{0.05, 0.1, 0.2\}$
- $N_{\mathrm{MH}} \in \{2500, 5000, 10000\}$

We found $p_{\mathrm{bp}} = 0.1$ and $N_{\mathrm{MH}} = 10000$ to be optimal. However, the improvement in accuracy from $N_{\mathrm{MH}} = 5000$ to $N_{\mathrm{MH}} = 10000$ is marginal (0.0001) on Leaderboard, while the running time is twice as long, so we decided to use $N_{\mathrm{MH}} = 5000$.

Acknowledgments

The author thanks Antti Asikainen and Adithya Balachandran for helpful discussions.