MCMC Cryptography

Cooper DeVane-Prugh

November 2022

1 Code Architecture

My code is split into files based on functionality. They are named such that the prefix indicates the general category that functions within that file are apart of. Utilities are named with the prefix **util**, MCMC methods have the prefix **mcmc**, etc.

1.1 General Utilities

To generate a random key, encrypt, and decrypt plain text, I built off of the pseudo code you gave us in the project instructions. Keys use a dictionary to hold lowercase letters as the keys, with randomly chosen letters as their values. Encrypt and decrypt iterate through a given string, and return a new string based off of the input encryption/decryption key. They are functionally the same, only named differently for ease of use throughout the code.

The compare string function is used at the end of an experiment to measure accuracy. It compares the final decrypted cipher text to the original plain text, and returns the percentage of matching letters.

Valid words and invalid words take a list of words, and return the number of valid (or invalid) words in that list. They use the dictionary found on Unix operating systems, at **usr/share/dict/words**. These are used in the dictionary style attack that occurs toward the end of an experiment.

1.2 Reference Text Analysis

These functions are used in analyzing sequential letter frequency in War and Peace. In **clean_text**, I use regular expressions to remove capital letters, line continuations, special characters, and multiple spaces from the text. It returns a string with only lowercase letters and single spaces.

I opted to use dictionaries instead of arrays to hold the sequential two letter (bigram) and three letter (trigram) counts from War and Peace. Empty dictionaries are created with every possible two or three letter combination as a key. Then the cleaned text is iterated through, and the dictionary is updated based on the current n-gram in the text. I initialize each value in the dictionary at one, to avoid issues with taking the log while scoring later on.

1.3 Search Space Reduction

Towards the end of an experiment, a search space reduction is done to remove solved letters from the pool that our RNG can choose from when proposing a swap. This accomplishes two things. One, it reduces the number of candidates to choose from, making it easier to randomly pick the remaining unsolved letters correctly. Two, it prevents correctly solved letters from being swapped out.

It takes our decrypted cipher text, and makes lists of valid and invalid words using the functions described in 1.1. It then decomposes those words into valid (solved) and invalid (unsolved) letters. If an unsolved letter appears in two or more valid words, the letter is removed from the list and considered solved. Two lists are then returned. A list of unsolved letters, and a list of remaining candidates for the unsolved letters.

1.4 Plausibility Scoring

I have two primary methods for calculating P, the plausibility score that a proposed letter sequence is real. They both work by iterating through the decrypted text and adding the value in the occurrence dictionary (probability matrix) that corresponds to the proposed text's current n-gram. In the summation methods, the number of occurrences for that n-gram is added to the score. In the Stanford scoring methods (defined in detail in [2] and [1]) the log of the value is taken, then added. Both scoring methods have a bigram and trigram version.

Additionally, there is a word score implemented in the Stanford methods. It uses a dictionary of the top 100 words used in War and Peace, along with the corresponding number of occurrences of those words. If words in the proposed text are in the top 100 dictionary, their score is also added to the n-gram score.

The final scoring method, **score_words**, is used in the dictionary attack. It is similar to the valid_words function found in section 1.1, except it returns the number of valid words found in the decrypted cipher text.

1.5 Experiments and MCMC Methods

There are three MCMC functions I used. They all follow the same basic structure as our travelling salesman code, the difference is the conditional statements used to accept or reject a proposed key. All MCMC functions always accept a better score, as well as return a list of scores, final key, and the decrypted text.

mcmc_fixed accepts a worse n-gram score at a fixed probability, defined when calling the function. When annealing, a worse score is accepted if a random number between 0 and 1, from a uniform distribution, is less than $e^{\frac{\Delta S}{T}}$. Where ΔS is the change in score, and T is the current temperature in the annealing process.

In **mcmc_dictionary**, only the score_words scoring method is used. When swapping a single letter, it is usually the case that the difference in word score (ΔWS) is -1, 0, or 1. That is, swapping a single letter usually results in the number of valid words staying the same, or changing by one. If $\Delta WS > 0$, the

proposed swap is accepted and the letters in the newly found word are removed from the search space. If $\Delta WS = 0$, the proposed swap is accepted but the search space is not reduced. If $\Delta WS < 0$, the swap is rejected.

The two experiment files call the various MCMC methods. The single experiment file was used for testing functions and plotting. The multi experiment file is used to run multiple trials of an experiment and gather results.

1.6 Data

The data folder contains the n-gram and word dictionaries created in **war_and_peace_analysis**. It also contains the Unix dictionary, as well as plain text files for War and Peace and the Bee Movie script.

2 Scientific Investigation

2.1 Scoring Method Comparison

It has been shown that a cipher text of 2000 characters is long enough to consistently solve a simple substitution cipher with high accuracy [1]. As a "proof of concept" I use the first 2000 characters of the Bee Movie script, to test my scoring methods. For each scoring method, I run 100 MCMC experiments of various lengths. Each experiment encrypts the plain text with a randomly generated key, and accepts only positively scoring proposals. The final decrypted text is checked for accuracy against the plain text.

Tables 2.1 and 2.1 both show that as the predetermined number of iterations in the Markov Chain increases, our accuracy in decrypting the cipher text also increases. All scoring methods show a convergence by 2000 steps in figure 1. The Stanford scoring method is the most effective at decrypting our cipher text, with about 10-20 more successful decryptions than the summation methods. Additionally, our average accuracy and score don't improve beyond 10,000 iterations. Therefore, going forward I will only be using the Stanford scoring method, as well as 10,000 iterations for the fixed probability MCMC experiments.

Iteration	Average Accuracy	Successful Decryptions
500	55.61%	0
1000	76.50%	2
5000	94.44%	82
10000	96.02%	81
20000	95.34%	79

Table 1: Summation Bigram Scoring. Results averaged over 100 experiments.

Iteration	Average Accuracy	Successful Decryptions
500	59.49%	0
1000	76.23%	3
5000	96.78%	92
10000	99.21%	98
20000	97.97%	95

Table 2: Stanford Bigram Scoring Test. Results averaged over 100 experiments.

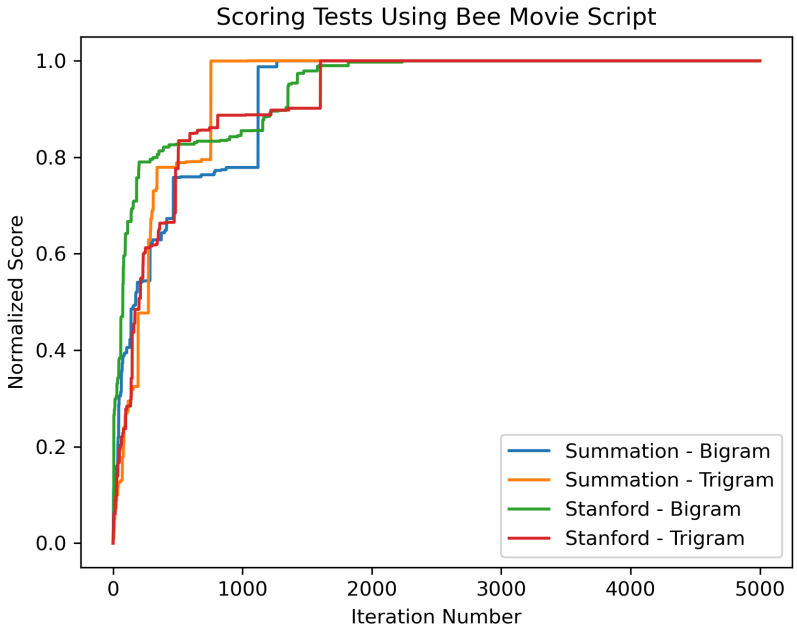

Figure 1: Test of convergence for all scoring methods. Scores are normalized.

2.2 Accepting Only Positive Proposals

To further explore the effect of cipher text length on decryption ability, I try texts of different lengths. The first is the shortened version of Jack and Jill provided in the handout. The second is the full nursery rhyme. Similar to 2.1, I ran 100 experiments on both texts.

In table 3, we can see that the average accuracy of our final decrypted cipher text increases as the length of the cipher text increases. The 35.55% accuracy achieved in solving the short nursery rhyme shows that only accepting positive proposals is completely insufficient for codes of this length. Figure 2.2 shows the percentage of correct letters as a function of iteration number for each of these texts, with results from 1 added for comparison.

Table 3: Accepting only positive proposals in MCMC loops of 10,000 iterations.

Plain Text	Text Length	Average Accuracy	Successful Decryptions
Short Rhyme	55	35.55%	0
Full Rhyme	119	57.84%	0
Bee Movie	2000	99.21%	98

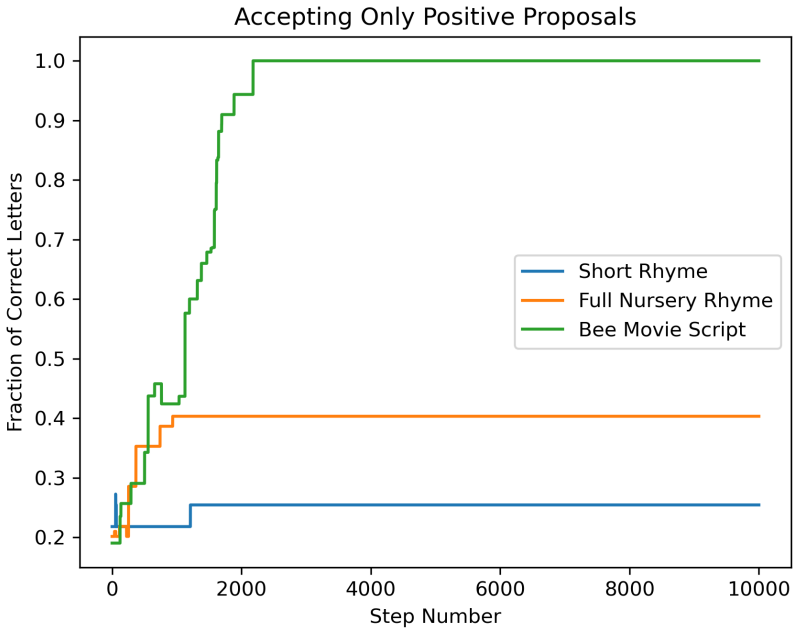

Figure 2: Texts of different lengths, using only positive proposals.

2.3 Accepting Negative Proposals at a Fixed Probability

Next, I tested how adding a fixed probability to accept negative proposals affects accuracy. I tried probabilities of 25%, 10%, 5%, and 1%. Once again I ran 100 experiments, this time only on the short Jack and Jill rhyme. Going forward, I only use the short Jack and Jill rhyme unless otherwise noted. Each experiment used the Stanford bigram scoring method, and a loop of 10,000 iterations. All fixed probability trials did worse than only accepting positive proposals.

Table 4: Accepting negative proposals at various fixed probabilities.

Acceptance Probability	Average Accuracy	Successful Decryptions
25%	27.05%	0
10%	28.55%	0
5%	29.80%	0
1%	34.47%	0

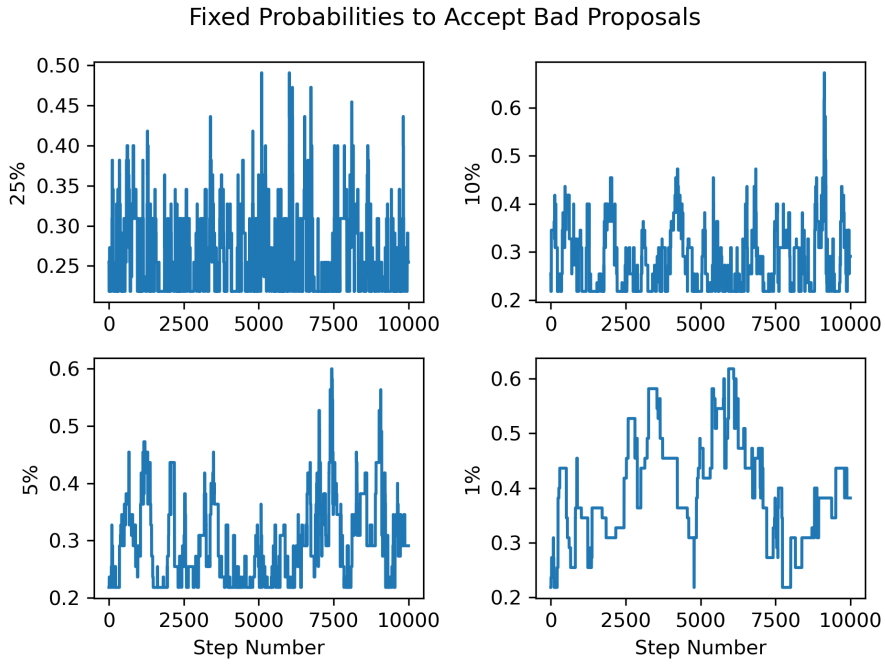

Figure 3: Plots of correct letter as a function of step number for each of the fixed probabilities. Fixed probability is labelled on the y axis of each subplot.

2.4 Annealing Parameters

Following the textbook's guidelines, I started with $T_{max} = 10$, $T_{min} = 10^{-3}$, and $\tau = 10^3$. As τ increases, and T_{min} decreases, the cooling time slows and the loop runs longer, leading to a higher score. I found a τ value of 10^3 cooled too quickly, not giving the code enough time to work itself out of local minimums. While a T_{min} of 10^{-4} usually increased my accuracy, it was not a sufficient or consistent enough increase to warrant the extra computation time. I settled on $T_{min} = 10^{-3}$ and $\tau = 10^4$ as my annealing parameters. This gave me an average accuracy of 53.10%, almost 20% higher than accepting at a fixed probability.

Table 5: Annealing

τ	T_{min}	Average Accuracy	Successful Decryptions
10^3	10^{-3}	41.25%	0
10^3	10^{-4}	43.31%	0
10^4	10^{-3}	52.76%	0
10^4	10^{-4}	53.10%	0

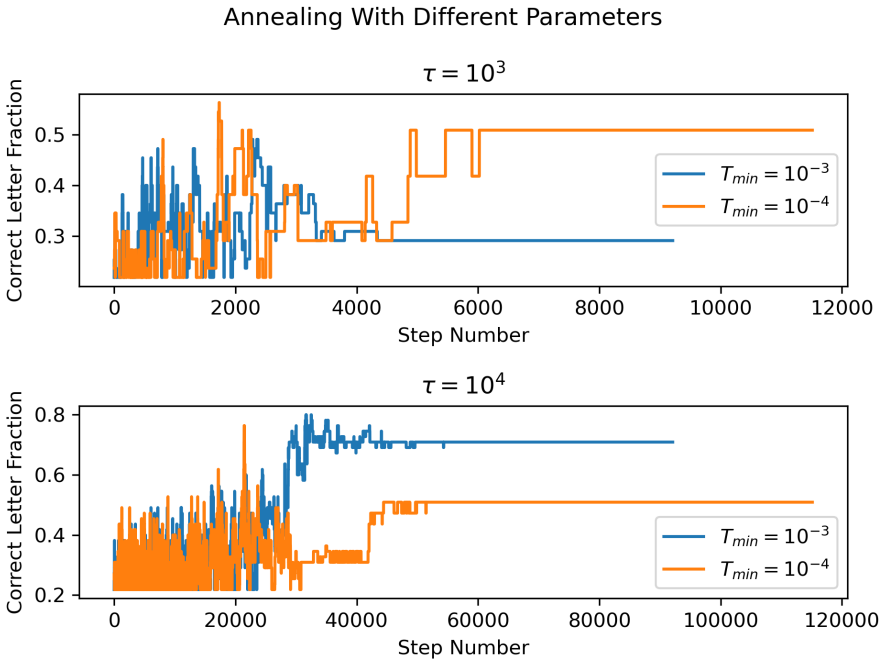

Figure 4: Each subplot is for a specific tau value, with the two Tmin values tried for that tau.

2.5 Adding Word and Trigram Scoring

At this point, I wanted to add a word score feature to the annealing code. It took several iterations to get working correctly. I started by adding a score multiplier that would add various percentages of the total score based on the length of valid words, but my implementation of this was buggy. I tried three different versions before completely scrapping it and switching to an occurrence dictionary similar to the n-grams. I made a dictionary of the most common 100 words, and their number of occurrences in War and Peace. I then integrated this into the Stanford bigram scoring method by having it add the bigram scores, then add the word scores.

At this point I realized trigrams may be more effective to use for scoring than

bigrams. So I made another dictionary for the trigram frequencies in War and Peace, as well as modified the Stanford score function to work with three letter sequences. Combining the word scoring with the trigram dictionary yielded excellent results. In 100 experiments, I achieved an average accuracy of 85% compared to the 75% achieved with bigrams and word scoring.

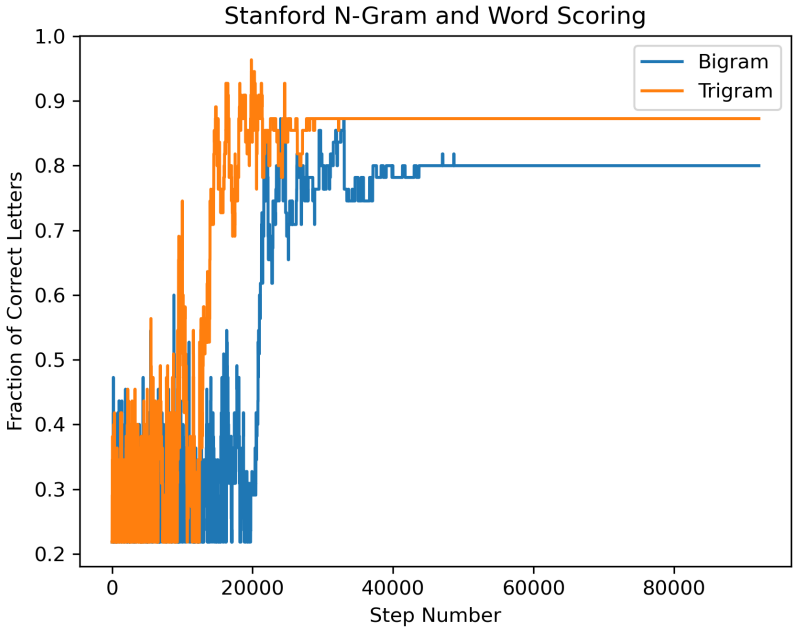

Figure 5: Comparing the effectiveness of word score on the Stanford bigram and trigram scoring methods. Both used the short Jack and Jill rhyme for testing.

2.6 Search Space Reduction and Dictionary Attack

A method to reduce the search space was needed so that there is no chance a solved letter can be randomly swapped out. The goal became to reduce the search space down to a sufficiently small size, so that a combination of luck and brute force would push the key toward the correct solution. This was necessary because the "ja" and "ji" in our Jack and Jill phrase, make it weirdly robust to just attacking with n-grams. I noticed my code constantly forcing the solution into "mack and mill" or "lack and liss." This makes sense considering "ja" only occurs about 100 times in War and Peace. Whereas "ma," "la," and "li" all occur close to 10,000 times.

After running the annealing MCMC loop, the search space is reduced, and a final pass is done in a for loop with a fixed number of iterations. I opt to use the number of valid words in the decrypted text as my score. As explained in 1.1, the Unix dictionary contains the valid words I check against. This portion of the attack was thrown together fairly last minute, as a result I didn't keep detailed experiment logs as I tweaked the function's parameters. I ended up using loops of 5,000 and 10,000 iterations in the final experiment setup because they were long enough to regularly generate correct solutions.

2.7 Final Experiment Setup

Based on the investigations above, the best way to attack this substitution cipher is:

1. Generate a random key and encrypt the plain text.
2. Use the Stanford scoring method, with trigrams and word score.
3. Run the annealing MCMC function with the following parameters:
   - $T_{max} = 10$
   - $T_{min} = 10^{-3}$
   - $\tau = 10^4$
4. Reduce the search space by removing solved letters from the RNG's swap pool.
5. Run the dictionary MCMC function and continue to reduce the search space as more words are found.
6. The final key is used to decrypt the cipher text.
7. The decrypted text is checked for accuracy against the plain text.

Table 6: Final experiment setup using the short and full Jack and Jill nursery rhymes. The dictionary attack was tested at 5000 and 10000 iterations for each.

Iterations	Text	Average Accuracy	Successful Decryptions
5000	Short Rhyme	91.27%	6
5000	Full Rhyme	99.46%	94
10000	Short Rhyme	92.45%	10
10000	Full Rhyme	98.00%	92

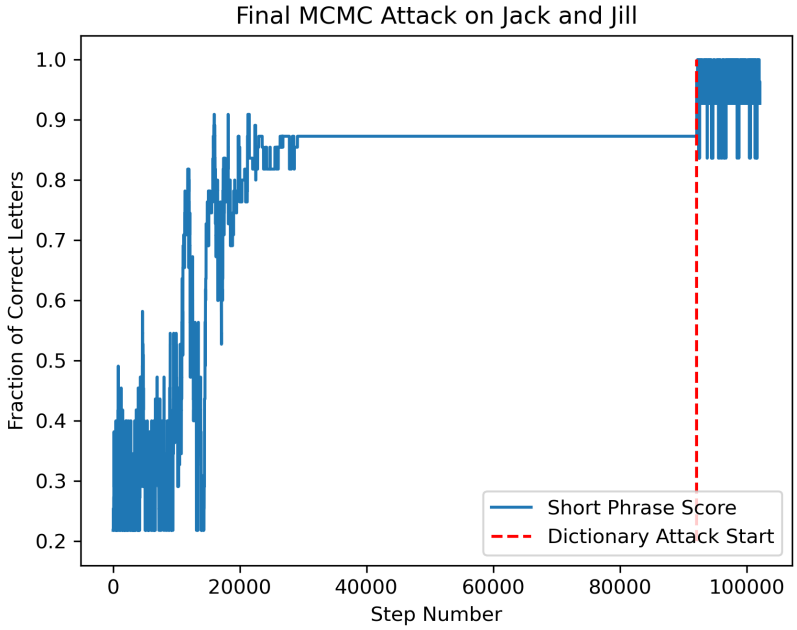

Figure 6: Final experiment setup on the short Jack and Jill nursery rhyme.

3 Conclusion

In this project, I have reproduced the results found in [1]. Showing that a cipher text of 2000 characters is sufficiently long to crack substitution ciphers with high accuracy. I have shown that the longer a cipher text is, the more successful n-gram frequency analysis is. Fixed probability alone is not enough to break these types of ciphers, annealing with sufficiently slow cooling is required. Additionally, using trigrams instead of bigrams is more effective, as is checking for valid words in the decrypted text. Finally, the Jack and Jill nursery rhyme (at least the shortened version) is surprisingly resistant to frequency analysis due to the letter j. Using the full nursery rhyme or switching to a phrase with a

different distribution of n-gram frequency may enable more future students to successfully decode the cipher text.

References

[1] Jian Chen and Jeffrey Rosenthal. "Decrypting classical cipher text using Markov chain Monte Carlo". In: *Statistics and Computing* 22 (Mar. 2012), pp. 397–413. DOI: 10.1007/s11222-011-9232-5.

[2] Persi Diaconis. "The Markov chain Monte Carlo revolution". In: *Bulletin of the American Mathematical Society* 46 (2008), pp. 179–205.