Cryptography

Oscar Barrera

AST4930 Project 1 - Desika Narayanan – November 14, 2022

Using MCMC methods, the algorithm decrypted “jack and jill went up the hill to fetch a pail of water” to ‘jack and jill rent um the hill to fetch a mail of rates,’ a phrase that achieved 91% accuracy. This was based on a random substitution cipher that was fed into the system as “zywdynfzmbboanxjrxiaimbbxpgaxwiyrymbpgoyxal.” To decrypt, the MCMC algorithm scored each proposed solution on a bi-gram probability matrix, along with referencing a dictionary to weigh solutions which matched words in the English Language. The solution converged after ~2000 iterations.

INTRODUCTION

The art of deciphering coded messages has become more accessible than ever thanks to the ever-increasing accessibility to computational resources. During World War 2, Allied experts devised a machine (the “bombe”) which would decipher enemy messages by testing every possible position of cylinders in the Enigma machine (which was one of the most common encryption computers of the day) [1]. This has extended into the modern age with the statistical method of Markov Chain Monte Carlo (MCMC) which allows us to iteratively converge to solutions to decipher coded messages with relatively low computational effort. [2]

CODE ARCHITECTURE

The first hurdle in deciphering this cypher was that the coded message did not have any spaces. To reproduce the proper phrase, the entire process was iterated over with a random number of spaces (1,20) broken up into a random assortment of word blocks, which was then scored and compared. However, this did not prove very effective, and the phrase had spaces manually added instead as there were no substitution characters provided for the cipher.

With this new cipher, the phrase was decoded from a simple unigram distribution as shown in section 3.8 from [2]. While this deciphered phrase did not show much promise, it nonetheless served as a valuable starting point for the solution key of the following bigram attack. A valuable benefit of the unigram starting point is that the spaces were not mapped to each other, as would be true of almost every single cipher since spaces would be the most common character. I thus excluded the shuffling of spaces from the jumbling of keys since it was already mapped properly.

It then followed to expand the unigram method to finding the distribution of how often every combination of 2 letters appears in a long text like War and Peace. This was done in a very simple way – looping through the alphabet twice and counting instances with ‘war and peace file’ .count(letter1 + letter2). This bigram probability matrix was saved into a dictionary (e.g., {'AB':53, ’AD':432,'PO':12}; however, the initial solve attempted to save the probabilities into a pandas DataFrame - the results of which are discussed in the next section.

With this arsenal of probability matrices, the following logic was implemented to decipher the text.

1. 'Unjumble the given phrase by iteratively swapping each letter with its corresponding character in the solution key.

2. Turn this unjumbled phrase into a bigram dictionary

3. Score this phrase by comparing to distributions from long text (this process will be explored in depth in the "Scoring Function" section).

4. Two indices are swapped at random in the solution key

5. This new proposed solution key is then scored again

6. If the proposed solution scores higher, then accept the proposed solution. However, if it scores worse, then the probability of accepting the lower solution is taken by testing the difference against a random uniform function. For example, one can take the min on (1, proposed / current) and compare it to an independently drawn random variable U from (0,1). If U < min, then the lower scoring solution is accepted, or else one keeps the current solution. It is obvious that as the score of the proposed solution decreases, the probability of keeping the current solution increases. Nonetheless, it is this randomness that allows the MCMC algorithm to escape local maxima. In the final algorithm, the swapping probability followed min (1 , math .exp (proposed score - (current score))), which is explained in the "Scoring Function" section.

7. This process is iterated over until the score converges. This is our resulting deciphered text.

VARYING PARAMETERS

The bigram probability distribution from War and Peace was first attempted to be generated by utilizing the pandas module. This created a 27x27 DataFrame which allowed data to be accessed in a much simpler way, notably, df[letter1][letter2]. Pandas is typically faster for large scale operations, and so it was my first guess to implement it for this function. However, it resulted in very slow computation times. I believe this is due to the fact that the deciphering code does not require any computations on the bigram distribution, only memory recall. It is for this reason that I switched the datatype to a dictionary.

For the scoring function, the initial attempt took advantage of the derivations from [2] by utilizing Equation 2.

$$\pi(x) = \prod_{\beta_1,\beta_2} r(\beta_1,\beta_2)^{f_x(\beta_1,\beta_2)} .$$

This process was achieved by finding the number of times a given combination of letters occurs in the reference text and multiplying it by the number of times it appears in the cipher text after a proposed solution key 'unjumbles' it. A '1' is added to every value as well to avoid issues from 0 as all these values are multiplied together. However, this scoring function did to perform too well, and it was then swapped for the method described in 'Scoring Function.'

The way that the swapping indices function operated was also varied. I tried swapping the letters of the characters which the unigram decoding function was unsure of first. Since not every letter is equally as probable, some stand out more than others; thus, the program should try to swap the letter J and C more than A and Q. This results in a weighted distribution from which the swapping function could choose from. However, with such a small phrase, the probability distribution was not useful, and this method was thrown out.

I also modified the scaling parameter p, which changed and flattens the probability density ($0 < p < 1$). This leaves the mode of the probability density unchanged while changing corresponding probabilities. This allows the MCMC escape local maximas. However, the best value I found was simply 1, which agrees with Sec. 4.2 of [2].

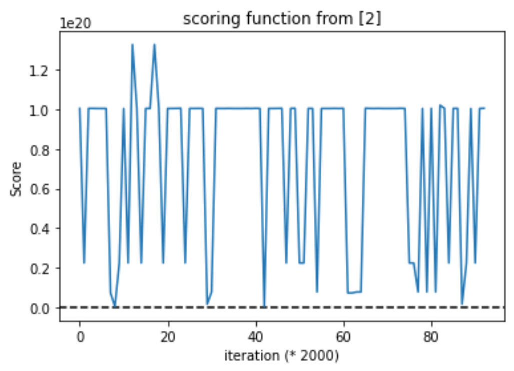
THE SCORING FUNCTION

The scoring function is taken as

```
for key, value in unjumbled_dict.items():
    score += value * math.log(bigram_dict[key])
```

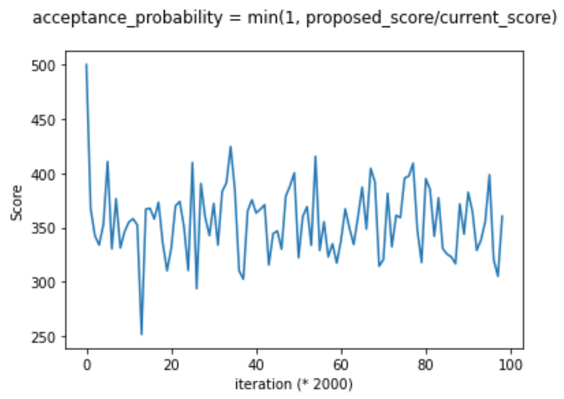
That is, for every combination of two letters in the unjumbled phrase, the score is equal to the amount of times that particular combination of letters appears in the reference text multiplied by the amount of times it appears in the unjumbled phrase. This is done in log space as well. This differs from the equation in [2] as the value in the reference text is multiplied rather than used as an exponent. When using the equation from [2], the score has trouble escaping minima for thousands of iterations (as seen on Figure. 1) Further, it has a wide range of values that it cycles through until it finds another resistance point. It is for this reason that the scoring function was changed to the log space equation.

Figure 1



The scoring function and subsequent acceptance probability has the biggest effect on the outcome of the MCMC decryption. When the swapping function followed the min on (1, proposed / current), it did not converge very well

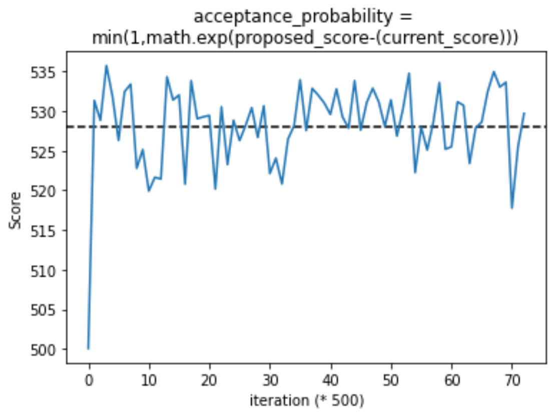
Figure 2



Resulting in the best phrase of "xacu ads xorr yedt wz the horr tj vetch a zaor jv yatek" with 0.56 %. The phrase is not recognizable yet.

If, then, we swap $\pi(x)$ by a power, $(\pi(x))^p$ then the acceptance probability can be replaced by $U_n < ((\pi(Y_n)/\pi(X_{n-1}))^p$. Using this (with a p value of 1), we begin to see higher rates of convergence and improved scores.

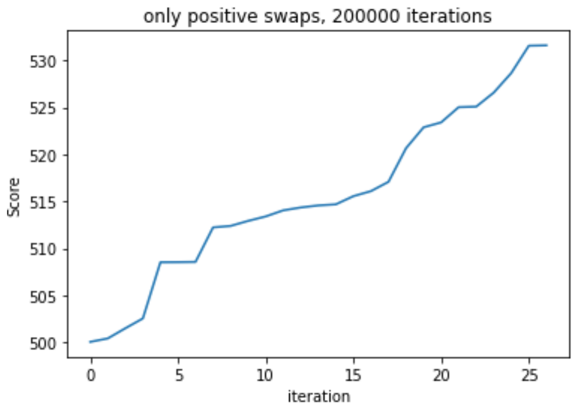
Figure 3



We see that the MCMC starts to converge around a score of 528. The highest score this achieved was 85% with the phrase "back and bill went rm the hill to setch a mail os watep." With this, a keen eye could pick out the phrase.

If we then decide to take only positive increases in score (rather than sometimes accepting negative swaps given a condition), then the whole processes ceases after just 25 iterations as it cannot find any single substitution to increase its score. It converges to "caum anw ciee lont fr tso siee td hotus a raie dh latog" at 0.49 % accuracy.

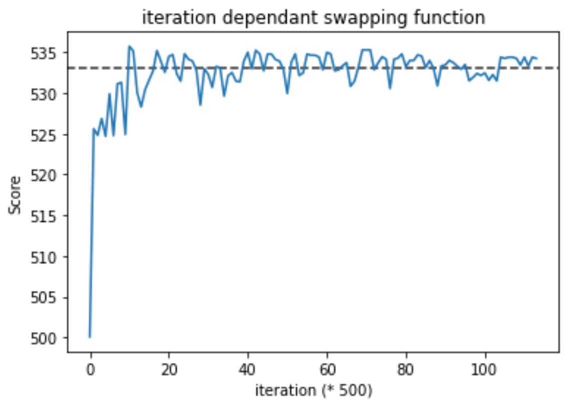
Figure 4



It is necessary to have negative swaps so that the MCMC can escape local maxima and converge at higher accuracy.

The swapping function was then varied by iteration. That is, it calculates the ratio q = current iteration / total iterations (which will be 0 < q < 1), and calculates the acceptance percentage by adding q/4 to the usual score. The 1/4th term was chosen arbitrarily by trial and error to see which fraction works best. When using this iteration-dependant method,

Figure 5.



The score converged much more nicely, resulting in the phrase "hacu ang hiss went bl tre riss to fetcr a lais of wated" with 71% accuracy. While it performed worse than without the iterative dependent method, it can be further analyzed in future decryptions to tailor the convergence rate.

The deciphered phrase was also compared to MIT dictionary of 10,000 words [3]. This was then scored by multiplying the previous score by the number of matches the phrase has with words in the English language and dividing by the length of the phrase. This denominator term can also be varied, but if it is too large then the MCMC can get stuck on an iteration where any swap reduces the number of words significantly.

RESULTS

Finally, all these methods were combined to create a much more powerful code breaker with 3 strong parameters to vary. Since the MCMC is dependent on iteration weight, word match weight, and scaling parameter weight, varying them with machine learning techniques will change how the MCMC escapes local minima and breaks the code. Since that may be outside of the scope of this class (we are focusing on MCMC methods) the code was then run with varying iterations from 100,000 to 400,000 in 10 steps. (~6 mins computation time). This resulted in the phrase: jack and jill rent um the hill to fetch a mail of rates' which achieved 91% accuracy.

Nonetheless, there is an aspect of randomness here. While it is true that the convergence depends on iterations, we see that the max score achieved when varying the number of iterations appeared random

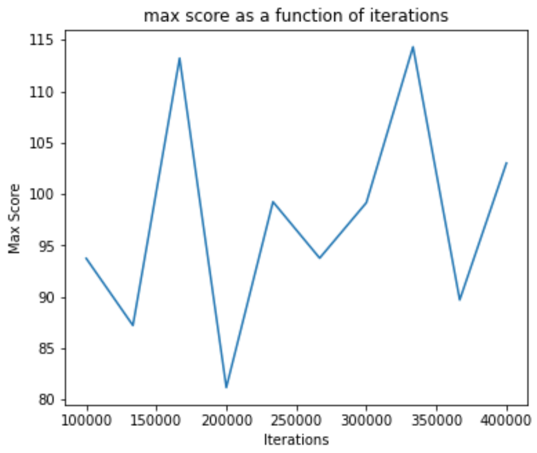
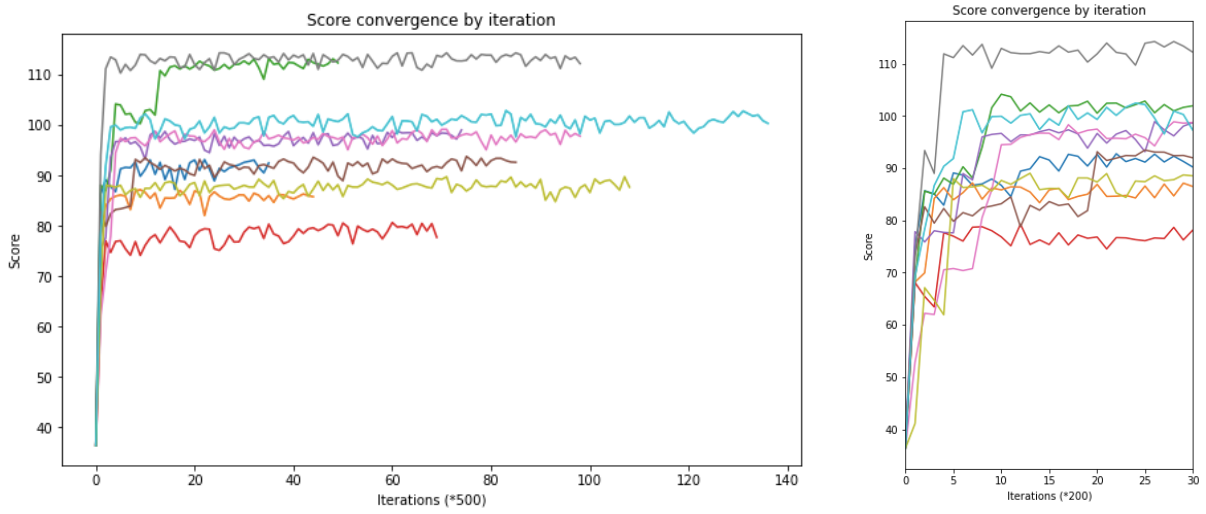


Figure 6

We could increase the number of steps for the iteration to get a better look at how the iteration affects the max score. This, however, would be in a larger scope as it requires extensive computational power.

If this phrase were to be guessed by randomly drawing a letter for each position, then the percentage of letters that were in the right position would be ~0.04%

Figure 7



The Final MCMC algorithm's convergence rates as a function of total iterations chosen. Regardless of the iteration, the MCMC converges after around 2000 iterations.

References

[1] Carter, Frank (2010), "The Turing Bombe", The Rutherford Journal, 3, ISSN 1177-1380

[2] Jian Chen and Jeffrey S. Rosenthal. 2012. Decrypting classical cipher text using Markov chain Monte Carlo. Statistics and Computing 22, 2 (March 2012), 397–413. https://doi.org/10.1007/s11222-011-9232-5

[3] https://www.mit.edu/~ecprice/wordlist.10000