6.437 Project R and D Report

Allen Wang
awang23@mit.edu

May 8, 2020

1 Introduction

The purpose of the project was to create a substitution cryptogram solver via Markov Chain Monte Carlo (MCMC). In the first part, each key consisted of some permutation of the 28 characters: the lowercase English letters, the space character, and the period character. In the second part, the two keys consisted of some permutation of the 28 characters along with a *breakpoint* where a switch of key was made.

The given datasets included frequencies of the 28 letters and transition probabilities of letters in English: i.e. a matrix M where $M[a_2][a_1]$ was the probability that a_2 appears given that a_1 appeared immediately before it. Using these datasets, when given ciphertext and key, decrypting the text and multiplying transition probabilities yields the likelihood of a key. Assuming a uniform prior, the MAP estimate for the key equals the ML estimate for the key. And indeed finding the key that maximizes this likelihood gives results that are quite promising. This motived the backbone of the MCMC approach for decryption without a breakpoint:

1. Randomly initialize key
2. At step n, propose the next key k by swapping any two letters in the current key
3. Compute the likelihood ratio r of the next key to current key
4. With probability $\max(r, 1)$, set the current key as k
5. Repeat until convergence and select key with greatest likelihood ratio

2 Additions to Part I

There were a few revisions made to the above backbone algorithm to facilitate convergence including better initialization, better transitions, and repeated trials. The first observation during testing was that in most texts, the space character is easily decrypted even by the human eye as it is the most common character by quite a margin. This then motivated initializing the key in terms of frequencies of characters: the most frequent key in the cipher text is mapped to the most frequent key in English, and the second most frequent key is mapped similarly, and so on.

Further improvements that were added include conducting the above initialization followed by reinitializing the key based on double letters, single letter words, and possible periods. This "reinitialization" is simply done by swapping the letters into the correct place. For example if k is the most common double letter in the ciphertext, we swap letters in the key so that k maps to l, which is the most common double letter in English. An identical procedure is done with single letter words. Lastly, it is known that all periods in the text must be followed by a space, and especially when texts are longer, we can take advantage of this. These

possible periods are hunted down in the ciphertext, and the character that is the most probable period is the one whose frequency in the ciphertext is closest to the frequency of a period in English, 0.0092.

Since statistics regarding double letters have greater variance than those regarding single letter words and periods, we first do the double letter swaps followed by those for single letter words and periods. So if such swaps were to overwrite one another, the ones with greater significance occur last.

A second key observation was that oftentimes the MCMC algorithm would achieve 100% accuracy at some point in the algorithm, but return a suboptimal answer. This arose because our notion of "likelihood" was only based on 2-grams in English. For example the work "just" was often decrypted to "qust", since following a "q", having a "u" was highly probable. This then motivated the concept of score, which rewarded the MCMC when English words were decrypted. In particular, the score function searches the decrypted string and adds 1 for each English letter part of the 10000 most frequently used English words. This score statistic was included in the MCMC algorithm by biasing log-likelihood. After numerous trials, empirically, the best results come when the score statistic is doubled and added to log-likelihood (base being the natural log). Thus, the ML estimate is found by taking the key that has the best log-likelihood plus score sum.

However, the above fix still fails to prove useful at times, since the disparity in likelihood may be great enough so that the acceptance probability of the proposal is too small. Hence, a biased form of the accept function: instead of only basing acceptance off probability, we automatically accept the proposal when there is a significant increase in score and the number of iterations elapsed is over 2000. This iteration condition ensures good convergence properties early on and corrections to the key using score in later stages of the algorithm.

The third observation was that no matter how robust the decryption algorithm was, there were inputs on which it performed suboptimally due to luck. This demanded multiple trials under different initializations. This motivated the final approach: decrypting the ciphertext in three different ways: using smart initialization and score-biased transitions, using initialization based only off frequencies and unbiased transitions, and using random initialization and unbiased transitions.

3 Finding the Breakpoint

The main challenge in part II of the project was having a good mechanism to identify the breakpoint. At first, the breakpoint was found by studying the identity of the space character in the first half and second half of the ciphertext, but eventually evolved into looking at the divergence bewteen the distributions of the characters in the first and second parts of the ciphertext. Specifically, for each index i in the ciphertext, the distribution up to i from the beginning and the end of the ciphertext is computed. This distribution is computed using a Jeffery's prior of the Dirichlet distribution over 28 parameters, which worked well empirically. Finally, the KL divergence is computed for these pairs of distributions for all possible i. The index with the maximum divergence is designated the most probable breakpoint.

Two comments are needed regarding the above approach. The first is that KL divergence is not symmetric. In fact, oftentimes the algorithm above was favoring extremely small indices for this reason (or extremely large indices if the KL divergence is computed the other way). It then seemed natural to compute KL divergence in both ways and sum them as our criteria for comparison: i.e. $D(p||q) + D(q||p)$ was used to compare breakpoints.

The second comment is the motivation behind Jeffery's prior. A Dirichlet prior seemed appropriate, but the question of the parameters remained. Many parameterizations were tested including the method of diminishing the importance of the prior as more observations are made. In the end Jeffery's prior, which corresponds exactly to observing 0.5 of each letter prior to any real observations, led to the best results.

4 Method for Part II

Extensive tests were done on various approaches for part II. The steps below yielded on average the most promising results:

1. Identify the 15 indices with the greatest divergence in the distribution of letters before and and after the index.
2. The most probable breakpoint b_1 is given a fast pass to the final stage.
3. Out of the remaining 14 breakpoints and the "no breakpoint" possibility (breakpoint index 0), a quick and rough decryption is done via random initialization and unbiased transitions, and the breakpoint b_2 with the greatest sum between log-likelihood and score advances to the final stage.
4. In the final stage, we have two possible breakpoints from above. The most probably breakpoint b_1 is decrypted using two methods: once via better initialization and score-biased transitions and once via random initialization and unbiased transitions. The breakpoint that advanced without a fast pass b_2 is only decrypted once via better initialization and score-biased transitions.
5. Finally, the three plaintexts from above along with the rough decryption of the b_2 from before are all compared via the sum of log-likelihood and score. The plaintext with the highest measure is then spellchecked via pyspellchecker and returned

Pyspellchecker is used in part II since there is a higher chance of misspelling at the breakpoint. In part I, however, accuracy is high enough that spellchecking often made no significant difference.

5 Number of Iterations

The number iterations used was often capped by the time limit of five minutes. Especially in part II, this restricted the number of such trials and breakpoints tested. In part I, the number of such trials to reach extremely good accuracy is listed:

len(ciphertext)	250	500	1000	2000
interations	15000	8000	6000	3000

Via runtime analysis, we see that each iteration of MCMC runs in $O(n)$ where n is the length of the ciphertext. Thus, to keep timing consistent. In part II, the number of iterations used in the rough decryption is inversely proportional to the size of the ciphertext.

6 Extensions and Conclusions

Some possible extensions that I would have liked to incorporate had time allowed:

- Better use of pyspellchecker to recommend switches in decryption
- Explore some sort of binary search method to find breakpoint
- Explore other priors in finding the breakpoint and keep better notes and data
- Use multithreading to boost computation time

Conclusion: All in all, the project was highly enjoyable and one of my favorite parts of the course. Thank you to all the 6.437 staff!