6.437 Project: Cipher Breaking using Markov Chain Monte Carlo Method

Rick Huang

1 Basic Algorithm Description

This algorithm relies on a general Markov Chain Monte Carlo method to generate the correct substitution cipher for a given ciphertext with given probabilities for each letter's occurrence in any location and transition probabilities between two letters in the English language. Here, I describe the general algorithm, and I will describe the specific optimizations for performance in Section 3.

We accomplish this by considering a starting cipher function, initialized as `curfunc`. Label our ciphertext as `ciphertext`, and our optimal function as the maximum likelihood function currently considered given the ciphertext as `optfunc`. At the current time, `optfunc` = `curfunc`.

Now, for each iteration, we generate a proposed new function `newfunc` as per Metropolis-Hastings to be a uniformly random function that satisfies the criteria of only having two assignments different from `curfunc`. The acceptance factor a is set to be the ratio of the likelihoods of `newfunc` over `curfunc` based on `ciphertext`. Then, we choose to reassign `curfunc` based on the sampled Bernoulli variable of a.

Within each iteration, we also consider the likelihood of `newfunc` compared with `optfunc`, and if the likelihood of `newfunc` is greater than `optfunc`, we set `optfunc` = `newfunc`. We run this basic algorithm until we get to our maximum number of iteration steps or our stopping condition triggers, at which point we output `optfunc`.

We run this algorithm several times up to `bagnum`, at which point out of all the proposed `optfunc`, we compute the maximum likelihood of all `optfunc` over a different subsequence of `ciphertext` and return the best `optfunc`.

2 Analysis and Testing

To analyze the performance of the overall algorithm, we can consider both runtime complexity and the accuracy over some sample set.

2.1 Runtime Complexity

To evaluate our overall runtime complexity, we must consider two major points of consideration within our algorithm. First, when calculating the maximum likelihood, the overall maximum likelihood computation spans each character of the ciphertext. Hence, though we want to scale our overall maximum likelihood based on the length of the overall ciphertext, we can consider a subsequence of the ciphertext and also cap the total length of the subsequences considered.

The second point of consideration comes from the stopping condition. We cap the total number of iterations so that there are at most 4000 iterations of training for the Metropolis-Hastings portion of the algorithm, and so if our subsequence is at most length 5000, we can expect around the order of 10^8 operations, which ensures that our algorithm likely falls under the 1 hour time constraint

consideration. Of course, our constraints on the subsequence and total iterations should hopefully require less operations, which will be described in the next section.

2.2 Testing Accuracy

For the accuracy, we run multiple iterations of the algorithm over the plaintexts given in both project part I and project part II, and evaluate whether or not the overall accuracy rate is 100% or not. In all of the course-provided texts, all iterations of the current algorithm run converged to the correct cipher 100% of the time.

We also consider the accuracy of our algorithm over certain "borderline" cases of valid English ciphertexts. This can be early modern English in the form of Shakespeare or a text that has a repetition of the same set of sentences or words, with a lack of diversity in the overall transition probabilities. Such texts still provide a 100% rate as measured previously, and we run as many automated iterations as possible over the period of a few hours for each such text.

3 Optimizations

We now describe some of the optimizations made and clarify certain aspects of the basic algorithm described in Section 1.

3.1 Function Initialization

When initializing `curfunc`, we want the ability to have a convergence within a reasonable number of iterations given some known prior about the text. Hence, we generate a subsequence of `ciphertext` and consider the frequency distribution of each letter here. We then match this to the given probabilities of each English letter appearing in each text, rank each English letter in terms of their frequency, and match this with the frequency ranks of the `ciphertext` subsequence. We then build our initial `curfunc` based on this quick prior on which characters are likely to match up.

3.2 Likelihood Computation

Because each probability is below 1, note that our maximum likelihood will have thousands of terms below 1 multiplied together. Hence, we replace each instance of likelihood with the log-likelihood, and adjust the acceptance factor a accordingly, so that our algorithm more accurately represents the computed values for the likelihood.

3.3 Transition Probabilities in Likelihood Computation

Note that some given transition probabilities are zero. However, in our overall log-likelihood computation, the zero factor would nullify the computation, so we reseed transition probabilities that are zero to some minimal value such as 10^{-4}.

3.4 Using Subsequences in Likelihood Computation

As previously stated, we care about the computation time of our likelihood given the length of the subsequence of the ciphertext considered. Hence, for each run of the algorithm, we consider a

random subsequence of the given ciphertext, specifically $\frac{1}{10}$ of the overall length to run the algorithm. If the ciphertext is too long, we use a maximum sequence length of 5000, which represents, in the worst case scenario, approximately a $(\frac{999}{1000})^{5000}$ 10^{-3} probability of not including a specific transition probability for a reasonable English text.

3.5 Boosting

we can also boost the accuracy of our algorithm by running the basic algorithm multiple times and taking the maximum log-likelihood `optfunc`. This helps because now we can consider a unique random subsequence in each iteration of our algorithm, and pick the best with the overall likelihood of a longer subsequence of the text. To ensure that our runtime is still equivalent, we can multithread our program so that each iteration of our basic algorithm also runs in a different thread, and combine our results in the end.

3.6 Stopping Condition

To ensure that our program terminates, we preprogram a reasonable number of iterations, approximately 4000. We also include a good stopping condition to ensure that our algorithm converges properly and compare our set number of iterations to the expected time it took our algorithm to converge with the stopping condition upon practice testing trials. Our stopping condition relies on the acceptance of a new `optfunc` given a better maximum likelihood, and if there is no change in this overall sampling after around 450 iterations, then we are confident in outputting `optfunc`. We choose 450 as there are $\binom{26}{2=325}$ overall possible functions. If $\frac{324}{325}$ is the probability of picking worse functions than the current function but one function is potentially better, then we have only a 25% chance of missing a better function. Coupled with boosting, our possibility of missing the correct function becomes exponentially lower.