Cipher Breaking using Markov Chain Monte Carlo

Brian Huang

Spring 2022

One very common application of inference in crypto-analysis is in cipher decoding. A cipher is a permutation i.e. one-to-one mapping of the alphabet, often the 26 English letters, used to transform an original plaintext message into an encoded string. Without knowledge of the cipher, an adversary attempting to crack a ciphered message can use inference techniques to deduce the most likely cipher. We'll see that a first-pass Bayesian approach to cipher breaking would be computationally intractable, but fortunately, MCMC techniques and the Metropolis-Hastings algorithm provide a tractable approach to cipher breaking.

1 Bayesian Framework

We first demonstrate how a maximum a posteriori (MAP) estimator would be formulated for our cipher breaking setting. We denote our observed encoded message in terms of its sequence of characters, $\mathbf{y} = \{y_1, ..., y_n\}$. For a given cipher f, an observed encoded message $\mathbf{y} = \{y_1, ..., y_n\}$ fully determines its original plaintext message as $f^{-1}(\mathbf{y}) = \{x_1, ..., x_n\}\{f^{-1}(y_1), ..., f^{-1}(y_n)\}$. The first plaintext character $f^{-1}(y_1)$ appears with probability determined by the alphabet distribution P_i, and every other character depends on the character before it, i.e. has probability determined by the transition probabilities $M_{i,j}$. Then we may write out the likelihood for a given f as:

$$p_{\mathsf{y}|\mathsf{f}}(\mathbf{y}|f) = P_{x_1} \prod_{i=1}^{n-1} M_{x_{i+1},x_i}$$

Next, recall that we assumed a prior distribution for the ciphers as uniform over all permutations of the alphabet. The posterior uses the likelihood and prior, and therefore it obeys the proportionality

$$p_{\mathsf{f}|\mathsf{y}}(f|\mathbf{y}) \propto p_{\mathsf{y}|\mathsf{f}}(\mathbf{y}|f)p_{\mathsf{f}}(f)$$

where the proportionality factor is the inverse of the marginal

$$p_{\mathsf{y}}(\mathbf{y}) = \sum_{f \in \mathcal{A}!} p_{\mathsf{y}|\mathsf{f}}(\mathbf{y}|f)p_{\mathsf{f}}(f)$$

(using A! to denote the set of permutations of our alphabet A). Because all the priors $p_{\mathsf{f}}(f)$ are equal across $f \in \mathcal{A}!$, our proportionality simplifies to

$$p_{\mathsf{f}|\mathsf{y}}(f|\mathbf{y}) \propto p_{\mathsf{y}|\mathsf{f}}(\mathbf{y}|f).$$

However, none of these formulations help for computing the MAP estimate in practice. Such a computation requires computing likelihoods of the observed data conditioned on each of the possible ciphers to find the maximum. However, $|\mathcal{A}!| = |\mathcal{A}|! = 28! \approx 3 \times 10^{29}$, and computing this many likelihoods is intractable.

For this reason, we turn to MCMC and the Metropolis-Hastings algorithm.

2 Metropolis-Hastings Formulation

In Metropolis-Hastings, our goal is to transform a homogeneous Markov chain with a proposal distribution $V(\cdot|\cdot)$ into a chain with the posterior distribution $p_{\mathsf{f}|\mathsf{y}}(\cdot|\mathbf{y})$. We must specify a proper proposal distribution and also choose a computable nonnegative function $p_\circ(\cdot)$ which is proportional to the posterior distribution. We found earlier that the posterior is proportional to the likelihood, which is easily computable for a given f (in linear time, assuming constant-time multiplication).

For the proposal distribution, we prefer it meet some conditions:

- $V(\cdot|f)$ could depend on f, so that we may sample single realizations of ciphers from $\mathcal{A}!$ across our random walk instead of sampling all of $\mathcal{A}!$ beforehand to generate our proposal distribution.
- The proposal could be symmetric, i.e. $V(f|f') = V(f'|f)$, so that the acceptance factor depends only on $p_\circ(\cdot)$ and not $V(\cdot|\cdot)$.

We can formulate the state transition of our Markov chain as swapping any two distinct symbol assignments in our input cipher to generate the output cipher. Note that, for a given input cipher, there are $\binom{28}{2}$ possible output ciphers for this rule out of 28! alphabet ciphers. (Similarly, the probability that any two independently drawn ciphers from $\mathcal{A}!$ differ in exactly two symbol assignments is $\frac{\binom{28}{2}}{28!}$.) Then we can make the transition probabilities uniform for each swap, giving rise to the proposal distribution:

$$\forall f \in \mathcal{A}!,\ \ V(f'|f) = \begin{cases} \frac{1}{\binom{28}{2}} & \text{if } f' \text{ and } f \text{ differ by exactly two symbol assignments} \\ 0 & \text{otherwise} \end{cases}$$

The following Metropolis-Hastings algorithm generates a Markov chain with steady-state distribution equal to $p_{\mathsf{f}|\mathsf{y}}(\cdot|\mathbf{y})$.

1. Initialize f_0 to a random permutation uniformly chosen from $\mathcal{A}!$.
2. At time n, suppose a sample cipher f_n has been generated. Sample a proposal cipher f' by uniformly selecting two characters in the English alphabet and swapping their assignments in f_n. This implements the proposal distribution $V(\cdot|f_n)$ described earlier.
3. Compute the acceptance factor $a(f_n \to f') = \min\left\{1, \frac{p_{\mathsf{y}|\mathsf{f}}(\mathbf{y}|f')}{p_{\mathsf{y}|\mathsf{f}}(\mathbf{y}|f_n)}\right\}$.
4. With probability $a(f_n \to f')$, accept the proposal cipher as $f_{n+1} = f'$, and otherwise reject the proposal cipher and set $f_{n+1} = f_n$.
5. Iterate from step 2.

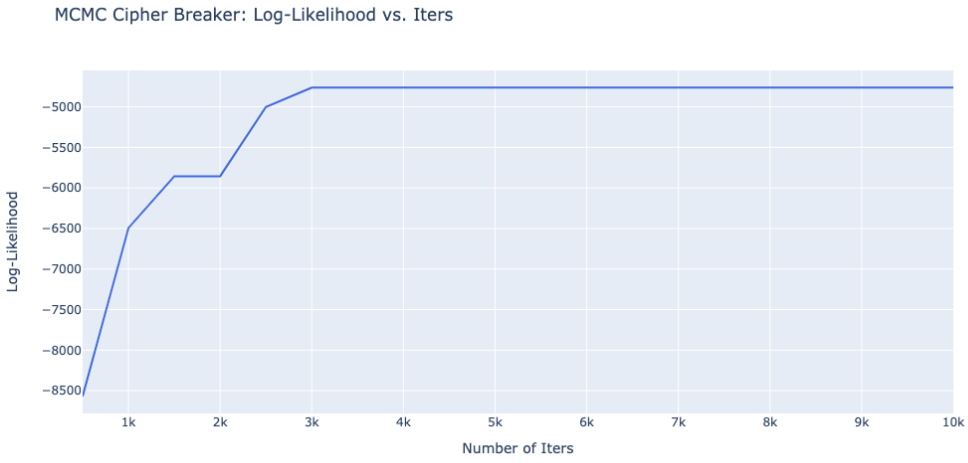
Finally, we can use this formulation of Metropolis-Hastings to build an MCMC-based decoder, outlined with the following pseudocode:

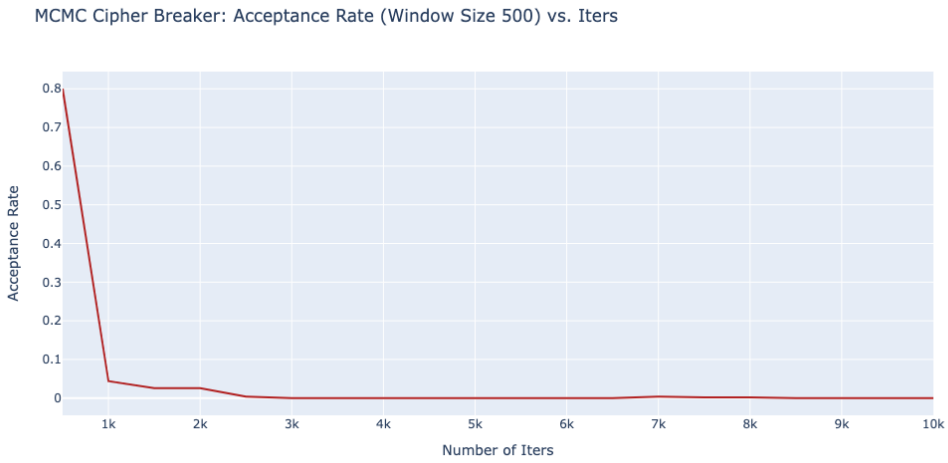
```
P = [list of English character distribution values]
M = [[2D array of English character transition probabilities]]
y = observed encoded string

def decode(y, f):
    return [list of f^-1(y_i) for all characters y_i in y]

def likelihood(decoded):
    return P[decoded[0]]*product(M[decoded[i+1]][decoded[i]]
    for i from 1 to length of y)

epochs = 10000
f0 = identity
random_walk = [list containing f0]
for epoch in epochs
    fn = last elt of random_walk
    sample f' by swapping two permutation assignments in fn
    acceptance_factor = min(1, likelihood(decode(y,f'))/likelihood(decode(y,fn)))
    ber_flip = random from uniform(0,1)
    if ber_flip < acceptance_factor
        append f' to random_walk
    else
        append fn to random_walk

map_cipher = (most frequently-appearing cipher in random_walk)

final_decode = map_cipher^-1(y)
```

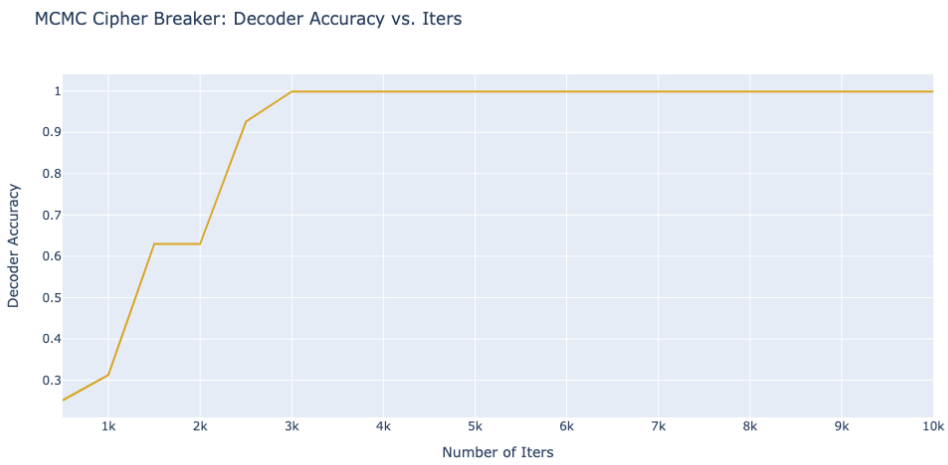
3 Metropolis-Hastings Implementation

We implement an MCMC cipher breaker based on the above pseudocode. In practice, we initialize Metropolis-Hastings with a random permutation sampled uniformly from $\mathcal{A}!$, rather than the identity permutation as shown in the pseudocode. For epochs, we've found that 10,000 is more than enough to reach convergence in the majority of our runs of the MCMC algorithm. We have yet to compute the mixing time for our proposal Markov chain, but this computation would be one of the first future modifications of the algorithm, in order to estimate the burn-in time and truncate the accepted states before the burn-in time.

We performed a sample run of the MCMC algorithm with 10,000 epochs and plotted log-likelihood, acceptance rate (with window size 500), and decoding accuracy for each 500 iterations.







One important line of analysis is to consider how our algorithm performs under different text lengths. Generally, we expect the algorithm to converge more slowly with smaller text segments. Consider the Fisher information associated with a text segment. Because of the Markovianity of our English language texts (each current character depends only on the one previous character), our likelihood was found earlier a product of transition probabilities

$$p_{\mathbf{y}|\mathbf{f}}(\mathbf{y}|f) = P_{x_1} \prod_{i=1}^{n-1} M_{x_{i+1},x_i}$$

so the Fisher information becomes

$$\begin{aligned} J_f(y) = -E[\partial_{ff} \ln(p_{\mathbf{y}|\mathbf{f}}(\mathbf{y}|f))] &= -E[\partial_{ff} \ln(P_{x_1} \prod_{i=1}^{n-1} M_{x_{i+1},x_i})] \\ &= -E[\partial_{ff} \ln(P_{x_1})] + \sum_{i=1}^{n-1} -E[\partial_{ff} \ln(M_{x_{i+1},x_i})]. \end{aligned}$$

Each $-E[\partial_{ff} \ln(M_{x_{i+1},x_i})]$ term is nonnegative (to satisfy the requirement that the Fisher information is nonnegative) so longer text sequences add more of these $-E[\partial_{ff} \ln(M_{x_{i+1},x_i})]$ terms, therefore increasing the Fisher information. A higher Fisher information directly means that the observed ciphertext provides more information about the underlying cipher. To see how this affects our MCMC algorithm behavior, we can consider the KL-divergence landscape for $D(f_{true}||\cdot)$, i.e. the many-dimensional plot of f vs. $D(f_{true}||f)$ for the unknown true cipher f_{true}. The divergence landscape has its global minimum at $f = f_{true}$ with corresponding divergence 0, and it can also have any number of local minima. Metropolis-Hastings can be interpreted as a sort of stochastic gradient descent on this divergence landscape. The second derivative of KL-divergence increases with Fisher information, so that (a longer text segment leads to) a higher Fisher information leads to a steeper divergence landscape on which Metropolis-Hastings can converge more quickly.

We test our theoretical analysis by running our MCMC algorithm on different text segment lengths. The original runs of MCMC used a 2017-character text segment, so we also attempted 1000-, 500-,

250-, and 125-character text segments, all with 10,000 epochs. We experienced that our MCMC was still able to converge for lengths of 1000, 500, and 250 within this allotment of epochs (with decoding accuracies around 97-99% each). However, there was no convergence for the 125-length segment in 10,000 epochs, and the decoding accuracy here was only 50%. This agrees with our theoretical prediction that smaller text segments result in slower convergence of MCMC. Although we did not plot the evolution of decoding accuracy across iterations for each MCMC run, we can make an educated guess that accuracy improved more slowly as we decreased segment length from 1000 to 500 to 250, although all of these cases were able to get accuracy to the convergent value within 10,000 iters. We can also make an educated guess that the 125-length segment, and even smaller length segments, would have converged given even more epochs. We argue this because the divergence landscape becomes more flat from low Fisher information, but its minima stay the same; then the gradient descent-like traversal of this landscape is slower for the same hyperparameters, but will still converge to some minimum from the same set of minima in the horizon of very many epochs.

Finally, we analyze our log-likelihood convergence behavior with respect to the empirical entropy of English. It is a well-known result of Shannon that the bigram entropy of English is around 3.343 bits. On the other hand, our log-likelihood for the entire string that we ran MCMC on was, in our example trial used for the above graphs, -4763.5 at the convergent value. Dividing by the number of symbols (2017 characters in the string) and converting from natural log to log base 2 (for bits), we obtain a log-likelihood per English symbol of -3.407 bits given our estimated cipher. This log-likelihood per symbol is very close to the negative bigram entropy of English as expected. Furthermore, we know that KL-divergence between our estimated cipher and the true cipher accounts for the difference in log-likelihood and entropy, so our cipher is very close divergence-wise to the true cipher.

4 MCMC Optimizations

In this section, we discuss various strategies and experiments we tried for optimizing the baseline MCMC cipher decoder. There are several aspects of our MCMC that constitute performance deficits:

- The divergence landscape that MCMC optimizes along is not convex and can have any number of local minima. MCMC is expected to approximate the true cipher distribution with arbitrary closeness after an asymptotically large amount of steps, but in this practical setting of very finite epochs, MCMC is somewhat likely to get stuck in a local minimum for much of its allotted epochs, so the predicted cipher it produces can be very different from the true cipher. As such, it could be effective to build a countermeasure into our MCMC cipher decoder against getting stuck in local minima.
- MCMC takes some amount of steps before its distribution adequately approximates the target distribution, i.e. converges to the steady-state. As such, in ideal MCMC settings this "burn-in time" would be calculated and all the initial samples up to the burn-in time would be discarded. We want to discard an appropriate number of the initial samples corresponding to the burn-in time, even if the burn-in time is intractable to explicitly calculate for our transition matrix.
- Our current number of epochs is currently a magic number, but we've see that the actual amount of epochs needed for MCMC to converge is highly dependent on factors such as segment length, acceptance rate across epochs, decoding accuracy across epchs, etc. As such, using these factors to construct a dynamical way to terminate the random walk could save time and prevent unnecessary steps.
- The code itself can be optimized in two different ways. First, the code may contain some heavy calculations that bottleneck the runtime, but across all the epochs of Metropolis-Hastings, there may be a lot of redundancies across these calculations that can be eliminated; we can reuse previous calculations or even precompute some values to be memoized for quick access across the Metropolis-Hastings random walk. Second, our implementation might have some relatively slow code that can be optimized for faster runtimes.

4.1 Avoiding Local Minima

In avoiding local minima, we use the insight that our cipher decoder is intended to output the single cipher that it finds with maximum likelihood, i.e. the cipher that appears with highest frequency during its random walk. As such, we are permitted to deviate from "pure" Metropolis-Hastings and add features that make the decoder are less accurate at approximating the entire distribution, if these same features increase the accuracy of converging to the true cipher.

Here, we borrow some ideas from gradient descent optimizations in the machine learning community. In machine learning, pure gradient descent is sometimes augmented using "momentum," in which any gradient descent step is performed as a weighted average of the current gradient and the k most recent steps; with this augmentation, previous gradients can cause the current iteration to move even if the algorithm is currently stuck in a local minimum (i.e. the current gradient is zero), which helps prevent getting stuck in local minima. Also, in machine learning, stochastic gradient descent is sometimes performed, where the gradient is calculated from a random subset of the data instead

of the entire dataset; this augmentation can be interpreted as adding random perturbations to the gradient that help it move away from local minima.

We implement various analogies of these machine learning gradient descent augmentations for our MCMC decoder; specifically, we modify the acceptance rate calculation as well as the Bernoulli event of accepting or rejecting the proposal distribution. The most coarse-grained augmentation we add is a small chance to accept the new proposal distribution, regardless of the acceptance rate or Bernoulli realization. The small chance is controlled by a "perturb" parameter, which we set somewhere in the range of 0.1-1%, based on the observation that, after convergence, our decoder exhibits an acceptance rate in this same ballpark of 0.1-1%. We expect this bypassing perturbation to have very little impact when Metropolis-Hastings has a large acceptance rate and has not converged yet, but when Metropolis-Hastings is stuck in a local minimum, we have an additional chance to move away from the local minimum in spite of the very high rejection rate at that local minimum.

We also implement an analogue of momentum gradient descent: if some of the k most recent steps were acceptances, we add an additional chance to accept in the current step, with the additional chance proportional to the amount of acceptances in the last k steps. Finally, we implemented an analogue of stochastic gradient descent: the log-likelihood is calculated on a random substring of the ciphertext rather than the entire ciphertext. In practice, we use the previous two augmentations instead of this SGD analogue, since some of our forthcoming optimizations are incompatible with the SGD analogue.

4.2 Burn-In Time

We again keep in mind that our decoder aims to output a single maximum-likelihood prediction rather than approximate the target distribution as in "pure" Metropolis-Hastings. As such, for our needs it is not as necessary to discard the initial steps before the burn-in time. Still, the discarding is easy to implement and can reap some benefits in decoding accuracy, as discarding initial steps that don't reflect the target distribution makes it more likely that the most frequently appearing cipher does reflect the maximum likelihood cipher of our target distribution. Because burn-in time is intractable to directly calculate for our transition matrix, we prescribe a magic number for it based on educated guessing from our log-likelihood, acceptance rate, and decoder accuracy behavior. From our earlier graphs, metrics such as acceptance rate and decoder accuracy begin to stabilize around 1000-2000 iters, so we set the burn-in time in the same ballpark.

4.3 Metropolis-Hastings Termination

We seek a dynamical method of terminating Metropolis-Hastings that allows us to terminate as soon as we are reasonably certain we have found the maximum likelihood cipher. Fortunately, we can bring in domain knowledge about English language information theory to help solve this problem. We previously mentioned that the bigram entropy of the English language is around 3.343 bits. When (or if) we have closely approximated the true cipher, the divergence between our predicted cipher and true cipher is close to 0. Recall that normalized log-likelihood (average log-likelihood per unit text length) can be reformulated as the negative sum of divergence and entropy; then, after a successful decoder run, the log-likelihood for the predicted cipher should be very close to the bigram entropy times the text length. We indeed saw this behavior for the log-likelihood after convergence in our baseline MCMC decoder.

As such, the log-likelihood can be a terminating condition for our MCMC decoder. If the normalized log-likelihood is close to the English bigram entropy (in nats), then we are reasonably certain we have closely approximated the true cipher and we can stop the decoder. After each interval of iterations (say, window size 500 like in our previous graphs), we extract the maximum likelihood cipher from the current history of the random walk, measure whether its normalized log-likelihood is close to -2.317 nats within some threshold/tolerance level (say, -2.4 nats), and terminate and output the cipher if this threshold is reached.

One theoretical risk of this approach is that, if the MCMC decoder gets stuck in a local minima, it can potentially take a prohibitively long time before the decoder finds a close approximation to the true cipher. Here, we have a tradeoff between continuing the current decoder run when we have gone a while without reaching the log-likelihood threshold, or restarting the decoder from a new random initialization; the preferable choice here is the one which is more likely to reach a close approximation of the true cipher in fewer steps.

Here, we have a few coarse-grained arguments for continuing the current decoder. First consider the case where we restart the run with a new random initialization if the log-likelihood threshold is not reached, and let q be the probability that any given run will end in a restart. Because the random initializations are i.i.d., restarting the decoder will lead to probability q that the new run will again get stuck in a local minimum, and we again must restart. As such, we expect to have to perform $\frac{1}{1-q}$ runs, i.e. $\frac{1}{1-q} - 1$ restarts, in order to achieve a successful run, and furthermore, the variance is high enough that we have a non-negligible chance of needing significantly more restarts than $\frac{1}{1-q} - 1$. On the other hand, continuing the run is a good option if we have strong enough mechanisms for avoiding local minima that don't correspond closely to the true cipher, so that we have a high chance of reaching the global minimum despite traversing local minima, obviating the need to restart. Considering that we do have such mechanisms, directly inspired by tried-and-tested gradient descent augmentations from the machine learning community, we opt to continue the run for extended iterations until the global minimum is found, instead of constructing any dynamical threshold for restarting the run.

This choice constitutes an aggressive decoding strategy: our algorithm will not stop until it finds the "correct" cipher. If the algorithm still gets stuck in local minima, it will simply take a prohibitively long time to terminate. As such, any future iterations on this decoder must pay special attention to the mechanisms for avoiding local minima.

4.4 Code and Calculation Optimizations

In this section, we highlight various optimizations we did to save computational time/power, avoid unnecessary computations, reuse redundant computations, etc. in our practical implementation.

- We memoize data that may be reused across the Metropolis-Hastings run; in particular, we store any log-likelihoods we compute for their corresponding ciphers during the Metropolis-Hastings run, so that if the same cipher reappears, we can retrieve the log-likelihood instead of recalculating.
- For any given ciphertext, we compute the type of the bigrams in the ciphertext, and compute the log-likelihood for a given cipher from this type multiplied element-wise with the cipher's transition probabilities. Then the log-likelihood computation itself is dramatically sped up compared to our baseline implementation, as the most expensive operations are just the element-wise sum and the mappings across the transition probability matrix for a given cipher.

5 Breakpoint Decoding

Aside from the earlier improvements to the baseline cipher decoder, we must build in an additional feature where we can successfully decode text with a breakpoint. This type of text will be arbitrarily divided into two contiguous segments (with the divider as the single breakpoint) and each segment will be encoded with its own independent cipher. To successfully decode breakpointed text, our algorithm must optimally find the breakpoint and the two ciphers corresponding to either side of the breakpoint. Our algorithm achieves this by performing a binary search to find the breakpoint, using the decoding accuracy of the predicted cipher on either end of the binary search in order to determine the next direction of the binary search.

In our practical implementation, we modify the binary search to leverage the "guaranteed" correctness of at least one of our left/right ciphers to perform as few Metropolis-Hastings runs as possible. Because the breakpoint must be in either the left half or right half of the total text, the other half not containing the breakpoint must be fully decodeable by a single cipher. As such, we perform two initial Metropolis-Hastings runs on each half of the total ciphertext, checkpointing regularly to verify the quality of the two current predicted ciphers. Our threshold for high-enough cipher quality is, similarly to the termination condition in the previous section, the event that the cipher's normalized log-likelihood is close to Shannon's English bigram entropy within a small tolerance level.

When a high-quality cipher is found, whether it is the left or right one, this cipher is then used to fully binary search for the breakpoint's approximate location. A traditional binary search continues until the target's exact location is found, but our condition(s) for branching the condition of the search, namely the normalized log-likelihood of the ciphertext substring relative to bigram entropy, is too coarse-grained to find the exact location. As such, we opt to terminate the binary search relatively earlier, when the search's left and right pointers are a somewhat small distance apart (but greater than 1), corresponding to the predicted breakpoint being within a small distance of the true breakpoint. Once we terminate the binary search, a third Metropolis-Hastings run is performed on the left or right substring, whichever one has not been decoded yet. The two output plaintexts of the final left and right decodings are concatenated into the final answer.