6.437 FINAL PROJECT — WRITE UP I

JUAN M ORTIZ

1. Problem Formulation. In this section, we will define a bayesian framing to the probelm of decoding a ciphertext that has been encoded using two permutation functions with a breakpoint.

In our approach, we model sentences as sequences of symbols taken from the alphabet $\mathcal{A} = \mathcal{E} \cup \{\text{"."}, \text{" "}\}$. Just as in **Part I**, let $f_1, f_2 \in \mathcal{F}$ be permutations on $\mathcal{A}$, X be the original plaintext of size N, Y the ciphertext obtained by

$$Y = f_1(X[1:b]) \oplus f_2(X[b+1:N])$$

Let p_x and $M_{a,b}$ be defined as follows:

$$p_x(x) = \mathbb{P}\left(X_k = x\right) \;\; \forall k \in [1, N] \tag{1.1}$$

$$M_{a,b} = \mathbb{P}\left(x_k = a | x_{k-1} = b\right) \;\; \forall k \in [1, N] \tag{1.2}$$

One possible way to frame the task of decoding the ciphertext Y is then to find the $f_1, f_2 \in F; b \in [1, N]$ that maximize the likelihood of the ciphertext Y given access to p_x and $M_{a,b}$.

Concretelly, we try to solve

$$\arg\max_{y, f_1, f2} p_{y|f_1,f_2,b}(y, f_1, f_2) =$$

$$p_x(f_1^{-1}(y_1)) \prod_{i=2}^{b} M_{f_1^{-1}(y_i), f_1^{-1}(y_{i-1})} \cdot p_x(f_2^{-1}(y_{b+1})) \prod_{i=b+2}^{N} M_{f_2^{-1}(y_i), f_2^{-1}(y_{i-1})}$$

2. Algorithm. To solve this problem, we follow a very simmilar approach to that of **Part I** in that we utilize MCMC sampling to find a MAP estimator of the above objective with the caviat that we now must search over the space $\mathcal{F} \times \mathcal{F} \times \mathbb{Z}_N$. Where $\mathcal{F}$ refers to the functions that are permutations over $\mathbb{Z}_N$.

After initialization of the above quantities, we continously sample f_1, f_2, and b until we satisfy some terminating condition while keepting track of the combination that leads to the largest log-likelihood. Note here that since f_1 and f_2 are chosen separatelly, we can sample them independently in the same manner.

Each Metropolis-Hastings sampling step consists of sampling f_i using the same proposal distribution as in the algorithm of **part I**. Namely, permutation functions are sampled from

$$V(f'|f) = \begin{cases} \binom{|\mathcal{A}|}{2}^{-1} \text{ for f and f' differ in exactly two symbol assignments} \\ 0 \text{ otherwise} \end{cases}$$

Then, we accept the sampled f' with probability $\alpha(f \to f')$ defined as

$$\alpha(f \to f') = \min(1, \frac{p_{y|f}(y_*|f')}{p_{y|f}(y_*|f)})$$

Where y_* reffers to the portion of the ciphertext y corresponding to the sampled function. In other words, y_* refers to $y[1:b]$ if sampling f_1 and $y[b+1:N]$ if sampling

f_2. Note here that $p_{y|f}$ refers to the posterior on y given f for the ciphertext without a breakpoint used in **part I**.

Simmilarly, we sample b according to some distribution and decide whether to accept it based on the acceptance factor

$$\alpha(b \to b') = \min(1, \frac{p_{y|f_1,f_2,b}(y|f_1, f_2, b')}{p_{y|f_1,f_2,b}(y|f_1, f_2, b)})$$

The final step to specify how to do sampling is then to show a proposal distribution for b. In my implementation, I used the intuition that as we continue to iterate, the true breakpoint should be in the visinity of our current breakpoint and chose to propose b' using a normal distribution centered around b clipped around on the tails so that all the probability mass outside of the range $[1, N]$ goes to whichever endpoint is closer.

3. Optimizations. In order to allocate the time I spent optimizing my above algorithm, I chose to take some inspiration from Bruce Lee's quote "I fear not the man who has practiced 10,000 kicks once, but I fear the man who has practiced one kick 10,000 times". My approach here was to create than on its own worked reasobly well a good percentage of the time and make it fast enough so it would be computationally feasable to run it longer and for more iterations.

3.1. Time Optimizations. Non-surpricingly, after profiling my original implementation of the above algorithm, I found that over 98% of my computing time was spent calculating log-likelihood of sequences of the ciphertext given a permutation function. As such, I decided to concentrate my efforts on making that function as efficient as possible.

3.1.1. Efficient Access and Vectorization. Computing the log-likelihood of y under the permutation function f, $\log(p_{y|f})$ requires computing a sum over the log-likelihood of (1) the first symbol on y and (2) a sequence of $N-1$ transitions where $|y| = N$. Note that this is equivalent to taking the product of their respective likelihoods, but is much less prone to underflow when working with small numbers.

Since we can work entirelly with log-likelihoods, we take the step of pre-computing $\log P$ and $\log M$.

At this point, two approahces seemed reasobale to me depending on whether the time-consumption of this step was dominated my the computation of the sum of the log-likelihoods or the access to the array. In eihter case, I wanted to make sure the operations could take advantage of Numpy's vectorization capabilities that allow the execution of the same instruction on multiple pieces of data.

If the first was true, I would pre-compute the counts of all transitions present in the ciphertext so that the overall log-likelihood of the transitions could be computed over a weighted sum over the log-likelihood of transitions between sybmol pairs. Note this can be easily implemented as the L1 norm of the element-wise product of matrices M and (hopefully sparce) T where T_{ij} referes to the number of transitions between symbol i and symbol j.

If the second was true, then I would make sure to minimize the number of times that I needed to access the arrays that store log-likelihoods and get the entire sequence at once. Once they were fetched, I took advantage of the already highly optimized implementations of sum from numpy to calculate the sum of the sequence. Note this is preferable to somehting like a sequential sum as it can be parallelized and reduce the number of required opertaions logarithmically.

Empirically, I found the second to be true. Although this was only two lines of code, it was by far my largest performance improvement. On its own, this was able to speed up each run 150-fold.

3.1.2. Terminating Condition. A good terminating condition is one that identifies a point at which continueing to iteratevily sample more permutations and breakpoints is unlikely to find new values for f_1, f_2 and b that yield a greater negative log-likelihood.

Origianlly, I followed the approach of keeping track of how often I was able to sample a set of values that resulted in a greater NLL from the ciphertext. After I failed to find bettwer combination of values for a given number, N, of consecutive iterations, I would give up trying to find new values. This approach requires one to waste at least N trials in order to realize that finding a better set of values is unlikely. Since we must choose N to fit most cases and the exploration of the log-likelihood landscape can be quite variable, one is forced to pick a large N and thus be very wasteful.

A better approach to take advantage of the shape taken by current exploration of landscape is to focus on the relative improvement in the recent past. The intuition being that if the landscape has been mostly flat, then it is likely that we are stuck in a point that is either locally or globally optimal and continueing to sample is futile. To see the landscape I keep track of a running window the last W best negative log-likelihoods and calculate the average relative improvement on that window. If at any point the average relative improvement of the last window falls below a certain threshold, I stop iterating.

In practice, I use a running window of size 500 and a threshold of 0.3%

3.2. Initial Conditions. For the non-breakpoint case, I expect the frequency distribution of the symbols in the decoded ciphertext to match that of english. As such, rather than initilizing my permutation function randomly, I initialize it to the permutation that makes the previous statement true. On average, this decreased the number of iterations to convergance needed by approximatelly 1000 on ciphertexts obtained by encoding random exerpts of the novel Don Juan by Lord Byron of lengths distributed normally around $10,000$ and with a standard deviation of 2000.

Since the frequency of symbols would be greatly affected by the placement of the breakpoint, I only utilize this initialization techique for the case where there are no breakpoints present.

3.3. Independent Trials. To improve accuracy, rather than reliying on the output of a single decoding run, I run several independent trials and choose to output the one that maximizes the negative log likelihood. This was quite useful as each trial is quite prone to get stuck at a local minima.