Decoding a Substitution Cipher with MCMC
6.437: Inference and Information Final Project

Brice Huang

May 11, 2018

1 Introduction

Let f be a permutation cipher on an alphabet $\mathcal{E}$. For the purposes of this report, $\mathcal{E}$ will be the set `{‘a’-‘z’,‘ ’,‘.’}`. Let $\boldsymbol{y} = y_1 y_2 \cdots y_N$ be the ciphertext of some English plaintext, encrypted by the cipher f. We will decode this ciphertext using statistics of the English language.

Let P_i denote the statistical frequency of $i \in \mathcal{E}$, and $M_{i,j}$ be the probability that $i \in \mathcal{E}$ is followed by $j \in \mathcal{E}$. Define the frequency vector $\boldsymbol{P} = (P_i)_{i \in \mathcal{E}}$ and the transition-probability matrix $M = (M_{i,j})_{i,j \in \mathcal{E}}$.

We model English as a Markov chain, in which each character depends only on the previous. In this model, the likelihood of a ciphertext $\boldsymbol{y}$, conditioned on cipher f, is

$$p_{\mathsf{y}|\mathsf{f}}(\boldsymbol{y}|f) = P_{f^{-1}(y_1)} \prod_{i=1}^{N-1} M_{f^{-1}(y_i) f^{-1}(y_{i+1})}. \qquad (*)$$

Assuming a uniform prior on f, by Bayes' Rule $p_{\mathsf{f}|\mathsf{y}}(f|\boldsymbol{y}) \propto p_{\mathsf{y}|\mathsf{f}}(\boldsymbol{y}|f)$.

Let $q(\cdot|f)$ be uniform over all permutations one transposition away from f. Then, a Metropolis-Hastings algorithm to sample from $p_{\mathsf{f}|\mathsf{y}}(\cdot|\boldsymbol{y})$ with proposal distribution $q(\cdot|f)$ is:

```
def mh_step(ciphertext, f):
    f’ = sample from q(.|f)
    with probability a(f -> f’):
        f = f’

def decode(ciphertext):
    f = uniformly random permutation
    for i in 1,...,ITERATIONS:
        mh_step(ciphertext, f)
    return f
```

where

$$a(f \to f') = \min\left[\frac{p_{\mathsf{y}|\mathsf{f}}(\boldsymbol{y}|f')}{p_{\mathsf{y}|\mathsf{f}}(\boldsymbol{y}|f)}, 1\right].$$

Moreover, because we expect $p_{\mathsf{y}|\mathsf{f}}(\boldsymbol{y}|f)$ to be exponentially larger at the true f than at any other f, sampling from $p_{\mathsf{f}|\mathsf{y}}(\cdot|\boldsymbol{y})$ should give, with high probability, the true f.

In the rest of this report, we will discuss optimizations to this basic implementation.

2 Stopping Rule

We choose the following stopping rule: stop when the last 1,000 transitions are all rejected, or after the 10,000th iteration, whichever comes first.

By analyzing the convergence behavior of this algorithm over many runs, we find that once the algorithm rejects 1,000 consecutive transitions, it has essentially converged – at least in the sense that once this happens, it has never been observed to accept one of the following 10,000 transitions.

We cut off our algorithm at the 10,000th transition to guaranteee that it terminates. 10,000 is a conservative upper bound for the convergence time, as the data shows the algorithm typically converges by the 3,000th iteration.

3 Accuracy Optimizations

A naïve implementation of Metropolis-Hastings, run with the above stopping rule, finds the correct answer with probability about 40%. This section will explore ways to improve this accuracy.

3.1 Removing Probability-0 Events

A failure mode of Metropolis-Hastings is that it can get stuck in a suboptimal local maximum, if all the transitions out of that local maximum are sufficiently unfavorable. When this happens, unless we run the algorithm for long enough that the it eventually jumps out of the local maximum, Metropolis-Hastings will output a wrong answer.

We can reduce the incidence of this failure mode by replacing probabilities of zero in the transition-probability matrix with a small value, e^{-20}. This change smooths out the infinitely deep holes that exist in our Markov chain in log-likelihood space; this reduces the number of local maxima that are difficult to transition out of.

With this optimization, individual runs of Metropolis-Hastings succeed with probability about 70%.

3.2 Independent Repeated Trials

To further amplify the algorithm's probability of success, we run our implementation of Metropolis-Hastings 20 times independently. Each run of Metropolis-Hastings outputs a guess $\hat{f}$ of the cipher permutation f, which has some likelihood $p_{\mathsf{y}|\mathsf{f}}(\boldsymbol{y}|\hat{f})$. We accept as our final answer the guess $\hat{f}$ with the largest likelihood.

Since the true cipher permutation is the f with maximal likelihood, this algorithm succeeds if at least one run of Metropolis-Hastings finds the true f. With the previous optimization, this occurs with probability

$$1 - 0.30^{20} \approx 99.999999997\%.$$

4 Performance Improvements

4.1 Storing Log Likelihoods

Instead of storing the marginal probabilities $\boldsymbol{P}_i$ and transition matrix $M_{i,j}$, we store their logarithms. The formula $(*)$ now takes the form

$$\begin{aligned}\log p_{\mathsf{y}|\mathsf{f}}(\boldsymbol{y}|\hat{f}) &= \log P_{f^{-1}(y_1)} \\ &\quad + \sum_{i=1}^{N-1} \log M_{f^{-1}(y_i)f^{-1}(y_{i+1})}.\end{aligned}$$

This transformation has the advantage that we now work with sums instead of products. The transition acceptance probabilities now take the form

$$\begin{aligned}&a(f \to f') = \\ &\quad \exp\left[\min\left(\log p_{\mathsf{y}|\mathsf{f}}(\boldsymbol{y}|f') - \log p_{\mathsf{y}|\mathsf{f}}(\boldsymbol{y}|f), 0\right)\right].\end{aligned}$$

4.2 Dropping the Marginal Probability

Because the input text is large, the marginal probability of the first character contributes very little to the overall log likelihood. Therefore, we can take the approximation

$$\log p_{\mathsf{y}|\mathsf{f}}(\boldsymbol{y}|f) \approx \sum_{i=1}^{N-1} \log M_{f^{-1}(y_i)f^{-1}(y_{i+1})}.$$

While this optimization, by itself, doesn't achieve a significant performance improvement, it enables future optimizations, as we will see.

For sake of clarity, for the rest of this report we write

$$\log p_{\mathsf{y}|\mathsf{f}}(\boldsymbol{y}|f) = \sum_{i=1}^{N-1} \log M_{f^{-1}(y_i)f^{-1}(y_{i+1})}.$$

4.3 Counting Ciphertext Transitions

Note that

$$\log p_{\mathsf{y}|\mathsf{f}}(\boldsymbol{y}|f) = \sum_{c_1,c_2\in\mathcal{E}} C_{c_1,c_2} \log M_{f^{-1}(c_1)f^{-1}(c_2)}$$

where C_{c_1,c_2} is the number of times c_1c_2 appears as a substring of the ciphertext. Therefore, from the ciphertext we can precompute the counts C_{c_1,c_2}. Then, computing $\log p_{\mathsf{y}|\mathsf{f}}(\boldsymbol{y}|\hat{f})$ takes $O(|\mathcal{E}|^2)$ operations instead of $O(N)$.

Since $|\mathcal{E}| = 28$ and N is is typically in the thousands, this optimization yields a substantial performance boost. It also makes our algorithm runtime, aside from preecomutation time, independent of input length, which is an attractive feature.

With this optimization, and all computations done in Python (no Numpy), our algorithm runs about 8,000 iterations of Metropolis-Hastings a second.

4.4 Maximizing a Dot Product

To enable the following optimizations, we reinterpret the problem of maximizing $\log p_{\mathsf{y}|\mathsf{f}}(\boldsymbol{y}|f)$ as follows. Given matrices $C = (C_{i,j})$ of ciphertext transition counts and $M' = (\log M_{i,j})$ of English transition frequencies, find the permutation $g = f^{-1}$ such that the dot product

$$h(g) := \sum_{i,j} C_{i,j} M'_{g(i),g(j)}$$

is maximized.

The Metropolis-Hastings algorithm, in terms of this reparametrized problem, is as follows. Start at a uniformly random permutation g; the proposal distribution is uniform on $q(\cdot|g)$, and the acceptance probability is given by

$$a(g \to g') = \exp\left[\min(h(g') - h(g), 0)\right].$$

4.5 Directly Computing Acceptance Probabilities

In the basic implementation, we compute the acceptance probability $a(g \to g')$ by computing $h(g')$ and $h(g)$ and subtracting the two. Because g' differs from g only by a transposition, many terms from the two expansions are the same. So, we can save computation by computing $h(g')-h(g)$ directly, without computing $h(g')$ and $h(g)$.

Specifically, suppose g' differs from g by the transposition (u, v). Then, the following expression equals $h(g') - h(g)$:

$$\begin{aligned}
&+\sum_{i\neq u,v} C_{u,i}M'_{g(v),g(i)} + \sum_{i\neq u,v} C_{v,i}M'_{g(u),g(i)}\\
&+\sum_{i\neq u,v} C_{i,u}M'_{g(i),g(v)} + \sum_{i\neq u,v} C_{i,v}M'_{g(i),g(u)}\\
&+C_{u,u}M'_{g(v),g(v)} + C_{u,v}M'_{g(v),g(u)}\\
&+C_{v,u}M'_{g(u),g(v)} + C_{v,v}M'_{g(u),g(u)}\\
&-\sum_{i\neq u,v} C_{u,i}M'_{g(u),g(i)} - \sum_{i\neq u,v} C_{v,i}M'_{g(v),g(i)}\\
&-\sum_{i\neq u,v} C_{i,u}M'_{g(i),g(u)} - \sum_{i\neq u,v} C_{i,v}M'_{g(i),g(v)}\\
&-C_{u,u}M'_{g(u),g(u)} - C_{u,v}M'_{g(u),g(v)}\\
&-C_{v,u}M'_{g(v),g(u)} - C_{v,v}M'_{g(v),g(v)}.
\end{aligned}$$

With this optimization, and all computations done in Python, our algorithm runs about 20,000 iterations a second.

4.6 Optimizing with Numpy

Finally, we optimize our algorithm with Numpy, taking advantage of Numpy's ability to compute dot products efficiently.

First, some notation: for a matrix A, let $A[u,:]$ and $A[:,u]$ represent the uth row and column of A. Let $A[u,v]$ be the (u,v) entry of A. For a vector v and permutation p of $[0,1,\dots,\text{len}(v)-1]$, let $v[p]$ be the vector whose ith entry is $v[p[i]]$.

We can compute the following expression for $h(g') - h(g)$:

$$\begin{aligned}
&+C[u,:]\cdot M'[g(v),:][g] + C[v,:]\cdot M'[g(u),:][g]\\
&+C[:,u]\cdot M'[:,g(v)][g] + C[:,v]\cdot M'[:,g(u)][g]\\
&-C[u,:]\cdot M'[g(u),:][g] - C[v,:]\cdot M'[g(v),:][g]\\
&-C[u:,]\cdot M'[:,g(u)][g] - C[:,v]\cdot M'[:,g(v)][g]\\
&+C[u,u]M'[g(u),g(u)] + C[u,v]M'[g(u),g(v)]\\
&+C[v,u]M'[g(v),g(u)] + C[v,v]M'[g(v),g(v)]\\
&+C[u,u]M'[g(v),g(v)] + C[u,v]M'[g(v),g(u)]\\
&+C[v,u]M'[g(u),g(v)] + C[v,v]M'[g(u),g(u)]\\
&-C[u,u]M'[g(u),g(v)] - C[u,v]M'[g(u),g(u)]\\
&-C[v,u]M'[g(v),g(v)] - C[v,v]M'[g(v),g(u)]\\
&-C[u,u]M'[g(v),g(u)] - C[u,v]M'[g(v),g(v)]\\
&-C[v,u]M'[g(u),g(u)] - C[v,v]M'[g(u),g(v)],
\end{aligned}$$

Computing $h(g') - h(g)$ in the above form, using Numpy for the dot products, does not achieve a performance boost. This is for two reasons: each Numpy call requires a context switch from Python to C, and the cost of these switches exceeds the benefit of using Numpy; and, the cost of computing the sixteen additional terms in Python is prohibitive.

We improve upon this by writing the above expression as a single dot product. Let $LEFT(u,v)$ be the concatenation of:

$$\begin{aligned}
&+C[u,:], +C[v,:], +C[:,u], +C[:,v],\\
&-C[u,:], +C[v,:], -C[u:,], -C[:,v],\\
&+C[u,u], +C[u,v], +C[v,u], +C[v,v],\\
&+C[u,u], +C[u,v], +C[v,u], +C[v,v],\\
&-C[u,u], -C[u,v], -C[v,u], -C[v,v],\\
&-C[u,u], -C[u,v], -C[v,u], -C[v,v]
\end{aligned}$$

and $RIGHT(u,v)$ be the concatenation of:

$$\begin{aligned}
&M'[v,:], M'[u,:], M'[:,v], M'[:,u],\\
&M'[v,:], M'[u,:], M'[:,v], M'[:,u],\\
&M'[u,u], M'[u,v], M'[v,u], M'[v,v],\\
&M'[v,v], M'[v,u], M'[u,v], M'[u,u],\\
&M'[u,v], M'[u,u], M'[v,v], M'[v,u],\\
&M'[v,u], M'[v,v], M'[u,u], M'[u,v].
\end{aligned}$$

Then, the above expression for $h(g') - h(g)$ equals

$$LEFT(u,v)\cdot RIGHT(g(u),g(v))[G(g)],$$

where $G(g)$ is the concatenation of

$$g, g + |\mathcal{E}|, \ldots, g + 7|\mathcal{E}|,$$
$$8|\mathcal{E}|, 8|\mathcal{E}| + 1, \ldots, 8|\mathcal{E}| + 15.$$

Computing $h(g') - h(g)$ in this way is still prohibitive, because computing $LEFT(u, v)$, $RIGHT(g(u), g(v))$, and $G(g)$ live in each iteration requires three expensive concatenations.

We improve further by precomputing the values of $LEFT(u, v)$ and $RIGHT(u, v)$ for all (u, v). There are only $O(|\mathcal{E}|^2)$ possible values of $LEFT(u, v)$ and $RIGHT(u, v)$, so this precomputation is possible. We also maintain a single copy of $G(g)$, which we update as we update g; this allows us to avoid constructing $G(g)$ by a concatenation every iteration.

With these optimizations, each iteration of Metropolis-Hastings requires one call to `numpy.dot`, and no other expensive operations.

The fully optimized algorithm runs at 75,000 iterations a second.