6.437 Project Part II

Evan Tey

May 2020

1 Introduction

Given a *ciphertext* $\boldsymbol{y}$ the goal of this project is to infer the substitution cipher $f(\cdot)$ and *plaintext* $\boldsymbol{x}$ used to create the ciphertext. We assume all original plaintexts are in English using the 28-character alphabet $A = \{\text{“a”}, \text{“b”}, \dots, \text{“z”}, \text{“␣”}, \text{“.”}\}$, though we'll frequently remap these letters onto $\{0, 1, \dots, 25, 26, 27\}$ for convenience. All substitution ciphers are assumed to be permutation functions.

In Section 2, we set up the inference problem and describe an MCMC algorithm we can use to infer $f(\cdot)$ using statistics of the English language. In Section 3, we discuss a series of enhancements to the algorithm to improve speed and accuracy. In Section 4, we generalize our approach to a scenario where two different ciphers are used during encryption – so $f_1(\cdot)$ is used until an unknown b-th letter and $f_2(\cdot)$ is used afterwards. Finally, in Section 5, we discuss our algorithm's performance on both problems and mention avenues for improvement.

2 Overview

After observing a length-n ciphertext $\boldsymbol{y}$, we want the best $\boldsymbol{x}$ under the English language that could generate $\boldsymbol{y}$. Since our cipher functions are bijections, we can reframe this as an inference problem over $f(\cdot)$'s. So, we define a random variable f assumed to be drawn randomly from the set of all permutations of A, and we would like the posterior distribution

$$p_{\mathsf{f}|\mathbf{Y}}(f|\boldsymbol{y}) = \frac{p_{\mathbf{Y}|\mathsf{f}}(\boldsymbol{y}|f)\ p_{\mathsf{f}}(f)}{p_{\mathbf{Y}}(\boldsymbol{y})} \tag{1}$$

Since $p_{\mathsf{f}}(f) = \frac{1}{|A|!}$, the posterior becomes proportional to the likelihood $p_{\mathsf{f}|\mathbf{Y}}(f|\boldsymbol{y}) \propto p_{\mathbf{Y}|\mathsf{f}}(\boldsymbol{y}|f)$. So, one simple estimator we can use is $\hat{\mathsf{f}}_{MAP}(\boldsymbol{y}) = \hat{\mathsf{f}}_{ML}(\boldsymbol{y}) = \text{argmax}_f\, p_{\mathbf{Y}|\mathsf{f}}(\boldsymbol{y}|f)$.

If we use a bigram model for the English language, we can write down the likelihood of a particular ciphertext as

$$p_{\mathbf{Y}|\mathsf{f}}(\boldsymbol{y}|f) = P_{f^{-1}(y_1)} \prod_{i=2}^{n} M_{f^{-1}(y_i), f^{-1}(y_{i-1})} \tag{2}$$

where P_α is the probability of seeing letter α in English and $M_{\alpha,\beta}$ is the probability that the next letter is an α given the current letter β. We assume these probabilities are known or can be approximated empirically by scanning example English texts.

Since there are 28! possible permutation functions, rather than trying out all possible functions or writing an ascent over functions, we can estimate the posterior using MCMC.

We propose the following transition distribution:

$$V(f'|f) = \begin{cases} \frac{1}{\binom{28}{2}} & \text{if f' and f differ in exactly 2 symbols} \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

Then, the acceptance probability is:

$$a(f \to f') = \min(1, \frac{p(f')V(f|f')}{p(f)V(f'|f)}) = \min(1, \frac{p(f')}{p(f)}) \tag{4}$$

where $p(f)$ is the posterior $p_{\mathsf{f}|\mathbf{Y}}(f|\boldsymbol{y})$. Together, this gives the effective transition distribution:

$$W(f'|f) = V(f'|f)a(f \to f') \tag{5}$$

which defines a Markov chain over permutation functions with stationary distribution $p_{\mathsf{f}|\mathbf{Y}}$.

Our decoding algorithm then works as follows:

Algorithm 1: Substitution Cipher Inference

```
input  : ciphertext
output: plaintext
$f_0 \sim$ Unif(permutations of $A$);
for $n$ in $[1, \ldots, T]$ do
    $f_{\text{proposed}} \sim$ Neighbor($f_{n-1}$);
    $a = \min(1, \frac{p_{\mathsf{f}|\mathbf{Y}}(f'|\boldsymbol{y})}{p_{\mathsf{f}|\mathbf{Y}}(f|\boldsymbol{y})})$;
    $u \sim$ Unif(0, 1);
    if $u < a$ then
        $f_n = f_{\text{proposed}}$;
    else
        $f_n = f_{n-1}$;
    end
end
$f_{MAP} = \text{argmax}_{f_i} \, p_{\mathbf{Y}|\mathsf{f}}(\boldsymbol{y}|f_i)$;
return $f_{MAP}^{-1}(y)$
```

3 Enhancements

3.1 Independent Ensemble of Chains

Depending on the initialization, our chain of f's may find itself stuck in a local maximum or far away from f's with higher likelihoods. This means our performance may be highly dependent on our initialization. One way mitigate this dependence is by running our procedure N times with independent initializations. With an example ciphertext with length 2017, our procedure is only able to find an f with over 99% decode accuracy 76 out of 100 times, where decode accuracy is the percent of correctly decoded symbols when compared to the true plaintext. If we repeat this procedure N times, the probability of failing to find a good f is $.24^N$, so if we pick $N = 7$, we should have a failure probability below 0.01%.

3.2 Faster Likelihood Calculation

The accuracy improvements above come at the cost of runtime, so we'd like to speed up our calculations as much as possible. (Another way to say this is that a 2x speedup causes an exponential improvement in accuracy using our ensemble method above.) Using a profiler, we can see that most of the runtime is spend in likelihood calculations. Originally, this calculation required stepping through the entire ciphertext and multiplying in the relevant element of $\boldsymbol{M}$.

The first thing we can do to speed this up is to calculate log likelihoods rather than likelihoods. Our computation becomes a large sum, which is faster to compute:

$$\log p_{\mathbf{Y}|\mathsf{f}}(\boldsymbol{y}|f) = \log P_{f^{-1}(y_1)} + \sum_{i=2}^{n} \log M_{f^{-1}(y_i), f^{-1}(y_{i-1})} \tag{6}$$

Next, we can recognize that the number of $\log M_{\alpha\beta}$'s included in this sum is simply the number of times the bigram "$\beta\alpha$" appears in $f^{-1}(y)$ given some f. So, for a given f, we just need to count the frequency of each bigram before a final sum. We can define the following bigram counts matrix, and rewrite the likelihood as follows:

$$T_{\alpha,\beta}(f) = \sum_{i=1}^{n} \mathbb{1}_{(f^{-1}(y_{i-1}),f^{-1}(y_i))=(\beta,\alpha)} \tag{7}$$

$$\log p_{\mathsf{Y}|\mathsf{f}}(\boldsymbol{y}|f) = \log P_{f^{-1}(y_1)} + \sum_{i=0}^{28}\sum_{j=0}^{28} T_{i,j}(f) \log M_{i,j} \tag{8}$$

Finally, we can speed up computation of the $\boldsymbol{T}$ matrix by realizing for an arbitrary f, $\boldsymbol{T}(f)$ is simply a reindexing of the original transition count matrix $\boldsymbol{T}(\mathbb{1})$ where $\mathbb{1}$ is the identity permutation. So

$$T_{i,j}(f) = T(\mathbb{1})_{f^{-1}(i),f^{-1}(j)} \tag{9}$$

If we precompute $\boldsymbol{T}(\mathbb{1})$, our likelihood simply becomes

$$\log p_{\mathsf{Y}|\mathsf{f}}(\boldsymbol{y}|f) = \log P_{f^{-1}(y_1)} + \sum_{i=0}^{28}\sum_{j=0}^{28} T_{f^{-1}(i),f^{-1}(j)}(\mathbb{1}) \log M_{i,j} \tag{10}$$

Where the original computation required passing through the entire ciphertext for every proposed f, we now only need to make one pre-sampling pass and take an inner product between two length-28^2 vectors for every proposed f – one fixed vector and one that needs reindexing.

In practice, we use NumPy to speed up computations. $\log \boldsymbol{M}$ and $T(\mathbb{1})$ are precomputed. Decryption and reshuffling $T(\mathbb{1})$ can by found by indexing through length-28 permutation arrays. The final inner product is a dot product after unravelling the two matrices. After these speedups, the generation of one chain on my 4 core i7-6560U CPU's @ 2.20 GHz take under 1 second, making the ensemble-related accuracy boost fairly significant.

3.3 Stop Condition

If the likelihood hasn't changed after some number of iterations W, the chain has likely stalled by settling into a local maximum. With this stopping condition, we can forgo the time spent on the rest of the iterations. This adds a small speedup that again allows for a larger ensemble size.

4 Breakpoint Handling

Now, we consider the problem where y is encoded with two ciphers f_1 and f_2. We'll assume f_1 is used until some breakpoint index b, and f_2 is used afterwards. Now, all our setup in Section 2 proceeds similarly, but replacing f with a tuple $\boldsymbol{f} = (f_1, f_2, b)$. We still put uniform priors over the random variables f_1, f_2, b. Our likelihood function becomes

$$\log p_{\mathsf{Y}|\mathbf{f}}(\boldsymbol{y}|\boldsymbol{f}) = \log P_{f_1^{-1}(y_1)} + \sum_{i=2}^{b} \log M_{f_1^{-1}(y_i),f_1^{-1}(y_{i-1})} + \sum_{i=b+1}^{n} \log M_{f_2^{-1}(y_i),f_2^{-1}(y_{i-1})}$$

(Technically, there should be a middle term $M_{f_2^{-1}(y_b),f_1^{-1}(y_{b-1})}$ which is dropped to simplify calculations.)

We can then define our transition distribution independently for each part of $\mathbf{f}$. Each f_1 and f_2 follow the distribution proposed in (3). $V(b'|b)$ is taken to be a Gaussian centered at b with standard deviation $\sigma = 21$. Again, this describes a Markov chain over $\boldsymbol{f}$'s with stationary distribution $p_{\mathbf{f}|\mathsf{Y}}$.

At a high level, the algorithm stays the same as in Section 2. Now, though, our sampling step requires sampling f_1, f_2, and b. The enhancement made in Section 3.2 also requires adjustment. For a permutation function f, we'd like the count matrices from 1 to b and from $b+1$ to n – we'll denote these $\boldsymbol{T}^{1\ldots b}(f)$ and $\boldsymbol{T}^{(b+1)\ldots n}(f)$.

We can no longer reindex $\boldsymbol{T}(\mathbb{1})$ because the matrices we'd like to reindex $\boldsymbol{T}^{1\cdots b}(\mathbb{1})$ and $\boldsymbol{T}^{(b+1)\cdots n}(\mathbb{1})$ are dependent on b. Instead, we'll precompute $\boldsymbol{T}^{1\cdots i}(\mathbb{1})$ for every $i \in [1, \ldots, n]$. Now, for any b, we have $\boldsymbol{T}^{1\cdots b}(f)$ since it's been precomputed and we have $\boldsymbol{T}^{(b+1)\cdots n}(f) = \boldsymbol{T}^{1\cdots n}(f) - \boldsymbol{T}^{1\cdots b}(f)$. With this, an $O(n)$ memory cost allows us to use our likelihood calculation enhancements.

5 Conclusion

5.1 Evaluation

For evaluation, we generate 1000 test plaintexts and ciphertexts from the provided Feynman, Milton, and Tolstoy texts. For a particular test, one of the three texts is chosen randomly and a random substring is chosen with a length between 256 and 2048.

With the model described, there are still a few hyperparameters to set: T – the maximum chain length, W – the stop condition threshold, and σ – the width of the conditional distribution on b's.

For a given set of hyperparameters, we can measure the decode accuracy and runtime for the best f (or $\boldsymbol{f}$) in each chain. We record the probability of success as the number of chains that achieved over 90 % decode accuracy (over the 1000 chains). Due to time constraints, we don't do a full grid search of the parameter space, but for the measurements we did make, we notice: (1) W's greater than 1000 do little to increase the success probability (only increase by a few percentage points) and (2) after around $T = 5000$ success probability only increases slightly.

If a given set of hyperparameters has probability p and average runtime t, we can calculate its ensemble success probability and runtime as $(1-p)^N$ and Nt (for an ensemble of size N). Since we'd like to constrain our runtime to 300 seconds, we can choose $N \approx \frac{300}{t}$. Then, we can compare success probabilities among different hyperparameter values with the 300 second constraint.

After experimentation with a few different hyperparameters, we choose $T = 7500$, $W = 1000$, $\sigma = 21$ and $N = 500$.

5.2 Future Work

First, with more time and computing power, a proper grid search should be done to optimize the hyperparameters, as the work here was informal.

Second, we note that, particularly for shorter strings, even if the MCMC achieves a maximum likelihood f, the decode accuracy is not guaranteed to be high. Since the bigram model is just an approximation for English, sometimes its likelihood is maximized when a true or ideal English likelihood is not. For example, in our model the string "qust" would have a high likelihood because each of the bigrams "qu," "us," and "st" has a high $M_{i,j}$ value, but in a true model for English, this should have a low likelihood because it is not a real word.

Possible solutions to this include using Markov models with more dependencies (eg. a trigram model or generally a k-gram model). Alternatively, more sophisticated language models have been created in NLP using structures like RNNs.

Another potential place for improvement is initialization. If a chain starts with a poor initialization, it may get stuck in a local maximum or it may just take a long time to reach parts of the distribution with higher likelihoods. One way to fix this is to control the initialization. Since we know $P(\text{“.”} \leftarrow \text{“␣”}) = 1$ (ignoring the character at the end of the string), we know our f should reindex $\boldsymbol{T}$ so the empirical conditional probability (T where each column is divided by the letter counts) at that transition is 1.

Unfortunately, because we can't guarantee uniqueness (multiple empirical conditional probabilities could equal 1), we did not include initialization control in the final algorithm. Limited testing suggested that the wrong initialization choice frequently led to chains getting stuck in local maxima.