6.437 Project II

Tugsbayasgalan Manlaibaatar

May 14, 2018

1 Overview

In this project, we aim to develop an MCMC algorithm that can decode any given cipher text. We assume that the cipher text is computed as $y = f(x)$ where y is the cipher text and x is the plain text. Then f is a 1-to-1 mapping function between letters (e.g permutation of letters). Therefore this problem can be done by maximizing the likelihood $L(y, f)$ that given cipher text y is encoded by the certain function f. Our initial approach was to initialize a random permutation function and start swapping two random letters of the function until it convergences. As it was quite slow, we came up with different optimizations that we will discuss in the next section. We were able to improve the efficiency by almost 30x factor.

2 Improvements

2.1 Underflow

As the length of the cipher text increases, we noticed that the likelihood was being rounded to 0 due to the product of too many small numbers. Therefore, we resolved this issue by changing to *log* domain. Since there were certain transitions that have 0 probability, we replaced them with a really small number such that *log* can be taken.

2.2 Computing Log Likelihood

The most important optimization comes from this part. For the sake of clarity, our likelihood function looks like this:

$$L(p, y) = p_y(f^{-1}(y_0) \prod_{n=2}^{N} p_{y_n|y_{n-1}}(f^{-1}(y_n)|f^{-1}(y_{n-1}))$$

As we can see, this computation takes $O(n)$ time where n is the length of the ciphertext. But this computation can be done in $O(1)$ time using the matrix trick. Essentially, let M be the state transition matrix and N be the observed transition matrix under current f. Then likelihood is just the product of all elements of the matrix that is calculated by element wise multiplication of M and N. Since we are using log-likelihood, we take the sum of elements instead of product of elements. Since both matrices have $28X28$ sizes, this multiplication can be done in $O(1)$ time. Now we need to figure out how to efficiently compute N after the letter swap. This can easily be done by swapping corresponding rows and columns of N. Therefore this computation takes $O(1)$ time per iteration. We also used numpy library to efficiently compute multiplications.

2.3 Local Maximum

We noticed that our code sometimes got stuck at one function due to a bad initialization function. We thought that it was because our code found the local maximum. We avoided this problem by using following tricks:

- **Smaller Search Space**: We know that whitespace always follows period. Therefore we can find corresponding cipher letters for these two from the ciphertext before running MCMC. This helped us to make our swap function better. Essentially we only swap between letters that are not whitespace or period.

- **Different Trials**: We run our algorithm for 10 times, and outputs the most occurred cipher function amonng those 10 results. Even tough there might be incorrect answers due to local maximum, we realized that they could be ignored if we just run enough trials. Correct answer occurred more likely than incorrect ones.

2.4 Termination

We tried to stop when the program decodes the text with sufficient accuracy (e.g. Enough english words), but we realized it was bit too inaccurate due to the fact that English dictionaries did not include conjugated words. This would make our accuracy calculator less accurate. Therefore we did not take this approach. But since our code runs under one minute in most cases, we decided it is okay to run it for enough iterations (in our case, the number of iterations was 20000) and use it as our termination condition. But just to be careful, we also added a condition that if the candidate functions keep getting rejected for more than the threshold value, we also terminated the program. This helped us to avoid from particular bad initialization.

3 Conclusion

By implementing above optimizations, we were able to decode test texts under one minute on average. This was much better compared to the naive implementation as the naive implementation took around 20 30 mins to decode. Finally, we really enjoyed this project since it gave us a chance to gain hands-on experience on all cool theoretical applications we saw in class.