Decoding a Substitution Cipher using MCMC

Dylan Hu

February 23, 2023

Contents

1 Introduction

Although it seems counterintuitive to many that human language, which we take such effort to generate in novel, creative ways, can be reduced to a mathematical model, we demonstrate in this project that in a certain sense this is the case.

Specifically, we leverage Shannon's probabilistic model of text and Bayesian inference to mathematically formulate the problem of decoding a substitution cipher.

However, given the size of the state space of possible substitution permutations (27!), we formulate a tractable method for solving this problem by turning to Markov Chain Monte Carlo methods, namely the Metropolis-Hastings algorithm.

2 Method

In this work, we model sequences of human language probabilistically using Shannon's bigram model of text. That is, given a set of 27 characters (the lowercase English alphabet and the space character), we model the probability of a sequence of characters as a stationary Markov chain.

Consequently, we can evolve the state of this stochastic process with the forward equation $\pi_{t+1} = \pi_t Q$, where π_t is the probability distribution of the characters at time t, and Q is the transition matrix of the stationary Markov chain.

By mining a large corpus of text, we can estimate the transition matrix Q (and a per-character probability vector P). We can then construct a maximum likelihood estimator for any sequence of characters, and by maximizing this likelihood we determine the substitution permutation applied to generate the ciphertext.

2.1 Generating a prior

To generate an estimate of the prior distribution of human language, we mine the corpus War and Peace by Leo Tolstoy.

2.1.1 Processing the data

We filter the corpus to only include lowercase English characters and spaces:

```
corpus_text = ''
    with open(args.corpus, 'r') as f:
        for line in f:
            line = line.strip()
            if line:
                line = line.lower()
                line = ''.join([c for c in line if
                    c in ' abcdefghijklmnopqrstuvwxyz'])
                corpus_text += line + ' '
    corpus_text = corpus_text.strip()
```

We then convert the text into an encoded Tensor format, where each character is mapped to an index 0-26, with 0-25 corresponding to a-z and 26 being the space character:

```
def text_to_tensor(text: str) -> Tensor:
    idx = lambda c: ord(c) - ord('a') if c != ' ' else 26
    return torch.tensor([idx(c) for c in text])
...
corpus = text_to_tensor(corpus_text)
```

2.1.2 Estimating the transition matrix

We estimate both the transition matrix Q and per-character probability vector P by counting the number of occurrences of each character and storing the count in a 27×27 matrix, where the row index represents the current character and the column index represents the next character.

To compute Q from the counts matrix, we simply normalize over each row. To compute P, we sum over and collapse each row and divide by the total number of characters in the corpus.

```
def mine(corpus: Tensor) -> tuple[Tensor, Tensor]:
    counts = torch.ones((27, 27))
    for i in range(len(corpus) - 1):
        counts[corpus[i], corpus[i + 1]] += 1
    P = counts.sum(dim=0) / counts.sum()
    Q = counts / counts.sum(dim=0)
    return P, Q
```

Note that we initialize the counts as 1 rather than 0, as this corpus is not large enough to have non-zero counts for certain rare bigrams. We avoid 0 in Q to avoid taking the log of 0.

2.2 Sampling with the Metropolis scheme

In order to construct our Metropolis scheme for sampling from the Gibbs distribution corresponding to our Markov chain, we must develop a maximum likelihood estimator and corresponding energy function.

Then, by biasing this easily generated Markov chain, we are able to traverse our state space and converge at a global minimum corresponding to the target permutation.

2.2.1 Computing energy

We compute the likelihood of a permutation σ scrambling a sequence $a_1, a_2, \ldots, a_n$ into $b_1, b_2, \ldots, b_n$ as:

$$L(\sigma) = \mathbb{P}_{\text{true}}(\sigma^{-1}(b_1)) \prod_{j=1}^{n-1} Q^{(2)}(\sigma^{-1}(b_j), \sigma^{-1}(b_{j+1}))$$

Consequently, in order to facilitate better numerical properties, energy is formulated as the negative log of likelihood:

$$E(\sigma) = -\ln \mathbb{P}_{\text{true}}(\sigma^{-1}(b_1)) - \sum_{j=1}^{n-1} \ln Q^{(2)}(\sigma^{-1}(b_j), \sigma^{-1}(b_{j+1}))$$

```
def energy_func(encoded: Tensor, P: Tensor, Q: Tensor)
        -> Callable[[Tensor], Tensor]:
    def energy(permutation: Tensor) -> Tensor:
        unpermuted = permutation[encoded]
        return -torch.log(P[unpermuted[0]]) -
               torch.sum(torch.log(Q[unpermuted[:-1],
                                      unpermuted[1:]]))
    return energy
```

2.2.2 A cheap source of randomness

In order to enable the Metropolis scheme, we must have an easily generated Markov chain evolving over the state space.

In our case, the state space consists of the 27! permutations, and a simple rule for moving between states is to switch two positions in the permutation. In other words, if the current permutation substitutes a with z and b with y, then we can move to a new permutation by substituting a with y and b with z.

```
def step_permutation(permutation: Tensor) -> Tensor:
    i, j = torch.randint(0, len(permutation), (2,))
    permutation_copy = permutation.clone()
    permutation_copy[i], permutation_copy[j] =
        permutation[j], permutation[i]
    return permutation_copy
```

2.3 Traversing the state space

To traverse the state space, we use the Metropolis scheme to inform our decision when moving between permutation states.

For a proposed move $\sigma \mapsto \tau$, we can compute the energy difference $\Delta E = E(\tau) - E(\sigma)$. If $\Delta E < 0$, then we accept the move. Otherwise, we accept the move with probability $\exp(-\beta \Delta E)$ where β is an inverse temperature > 0.

```python
def metropolis(encoded: Tensor, P: Tensor, Q: Tensor) -> Tensor:
    energy = energy_func(encoded, P, Q)
    permutation = torch.arange(27)
    count = 0
    prev_energy = energy(permutation)
    prev_energies = deque([prev_energy], maxlen=100)
    while not stop(prev_energies):
        count += 1
        new_permutation = step_permutation(permutation)
        delta_E = (new_energy := energy(new_permutation)) - prev_energy
        if delta_E < 0 or torch.rand(1) < torch.exp(-delta_E):
            decode(new_permutation, encoded)
            prev_energy = new_energy
            prev_energies.append(new_energy.clone())
            permutation = new_permutation
    return permutation
```

2.3.1 Stopping condition

The stopping condition used is a heuristic that performs well empirically. We keep track of the previous 100 energy values traversed to and compute the sum of the changes between moves. If the this sum is 0, then there were an equal number of moves between the (hopefully) global optimum and any adjacent permutation with greater energy.

3 Analysis

With the use of PyTorch tensors to perform most operations, I believe my implementation is fairly quick to converge without the presence of the stopping condition. It generally converges within 5 seconds on my machine. However, since the stopping condition is not vectorized and must iterate through the deque with every iteration, it adds a bit of overhead (about 1-2 seconds).

4 Further exploration

I did attempt to use a trigram model, but the optimization was far slower and did not converge.

I would like to investigate further more principled stopping heuristics.