Project: Cipher Breaking using Markov Chain Monte Carlo

Shiyin Wang (shiyinw@mit.edu)

Part I
Basic Solution

Problem 1: Bayesian Framework

(a)

$$P_{y|f}(y|f) = P_{f^{-1}(y_1)}\Pi_{k=2}^{m} M_{f^{-1}(y_k),f^{-1}(y_{k-1})} \tag{1}$$

(b)

$$P_{f|y}(f|y) \propto P_{y|f}(y|f)P_f(f) \propto P_{f^{-1}(y_1)}\Pi_{k=2}^{m} M_{f^{-1}(y_k),f^{-1}(y_{k-1})} \tag{2}$$

According to the problem description, we model the ciphering function f as random and drawn from the uniform distribution over the set of permutations of the symbols in alphabet $\mathcal{A}$.

$$\bar{f}_{MAP}(y) = \underset{f}{\operatorname{argmax}}\, P_{f|y}(f|y) = \underset{f}{\operatorname{argmax}}\, P_{f^{-1}(y_1)}\Pi_{k=2}^{m} M_{f^{-1}(y_k),f^{-1}(y_{k-1})} \tag{3}$$

(c)

The number of permutations is $28!$, which is very large to enumerate all of them.

Problem 2: Markov Chain Monte Carlo Method

(a)

If we fix the ciphering function f_1, there are $28!$ possibilities for f_2. Among them, only $\binom{28}{2}$ differ in exactly two symbol assignments.

$$\frac{\binom{28}{2}}{28!} = 1.239798 \times 10^{-27}$$

(b)

Here I sample symbol pairs uniformly and exchange these two symbols as the ciphering function. According to the Metropolis-Hastings algorithm, the transition function is

$$a(f_1 \to f_2) = max\{1, \frac{P_{f|y}(f_2|y)V(f_1|f_2)}{P_{f|y}(f_1|y)V(f_2|f_1)}\} = max\{1, \frac{P_{f_2^{-1}(y_1)}\Pi_{k=2}^{m} M_{f_2^{-1}(y_k),f_2^{-1}(y_{k-1})}}{P_{f_1^{-1}(y_1)}\Pi_{k=2}^{m} M_{f_1^{-1}(y_k),f_1^{-1}(y_{k-1})}}\} \tag{4}$$

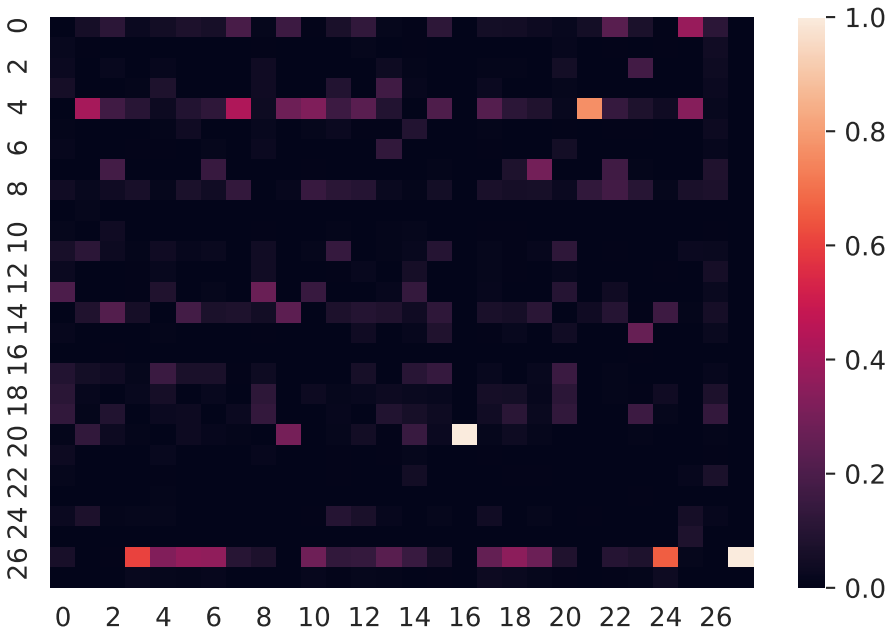

Figure 1: Visualization of Transition Matrix

(c)

Algorithm 1 Metropolis-Hastings

Uniformly sample the initial f_0 from all the permutations
for t from 1 to N **do**
  Generate f' by randomly flip two symbol assignments from f_{n-1}
  Compute acceptance factor $a(f_1 \to f_2) = max\{1, \frac{P_{f_2^{-1}(y_1)} \Pi_{k=2}^{m} M_{f_2^{-1}(y_k), f_2^{-1}(y_{k-1})}}{P_{f_1^{-1}(y_1)} \Pi_{k=2}^{m} M_{f_1^{-1}(y_k), f_1^{-1}(y_{k-1})}}$
  Generate a sample u from a Bernoulli random variable to determine whether to accept this new state with probability $a(f_1 \to f_2)$
  if Accept **then**
    $f_t = f'$
  else
    $f_t = f_{t-1}$
  end if
end for

Problem 3: Implementation

(a)

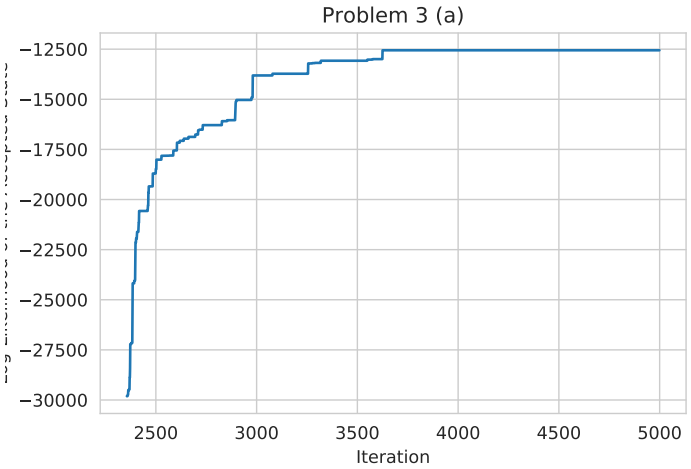

Figure 2: The log-likelihood of the accepted state in MCMC algorithm as a function of the iteration count

(b)

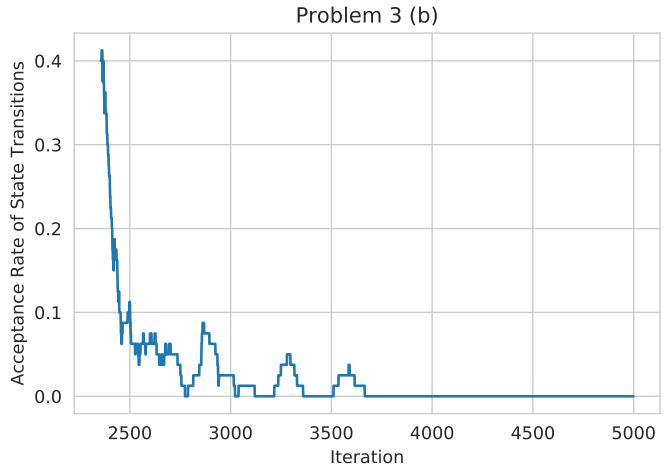

Figure 3: The acceptance rate of state transitions in the MCMC algorithm as a function of the iteration count. Set T=40.

(c)

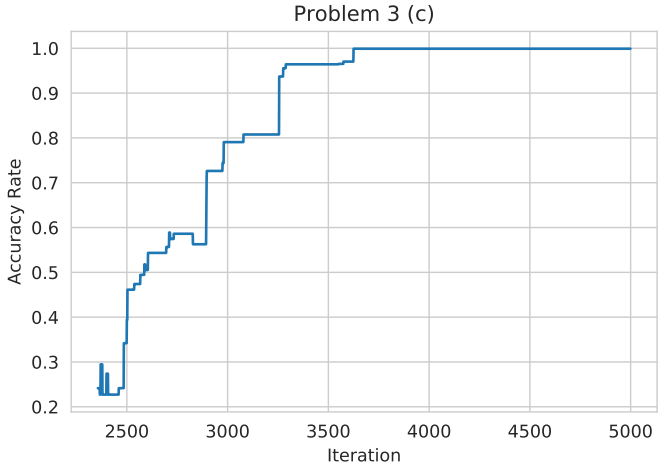

Figure 4: The accuracy rate as a function of the iteration count

(d)

The accuracy is not stable for small segments. For example, the ciphertext[1500:2000] shows an accuracy of 27.4% after 10000 iterations. According to the central limit theory, the empirical transition probability is close to its mean with smaller variance as the size of segments becomes larger. In conclude, adequate sample sequence is essential for this frequency based approach.

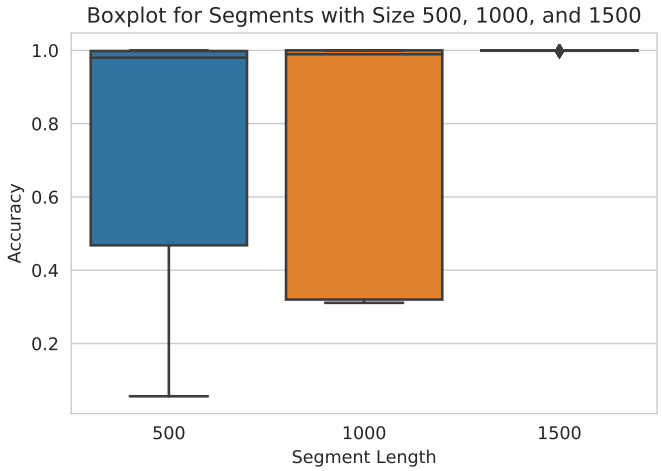

Figure 5: Experiments with partitioning the ciphertext into segments and running your algorithm independently on different segments.

(e)

The log-likelihood per symbol, in bits, fluctuates increasing. The steady-state value for the log-likelihood per symbol is the total log-likelihood for this sequence divided by the string length, which is about 2.374859945006275.

$$H(A) = E_{i,j \in A}[P_i M_{j,i} log(P_i M_{j,i})] \tag{5}$$

My estimated entropy for the English language is 5.176798462822324.

Part II
Improvements on Accuracy and Efficiency with Breakpoints

1 Method

Similar with the basic case, I use MCMC algorithm to randomly sample variables and accept with probability in each iteration. However, the existence of breakpoint makes it a little bit harder. I have to define a new variable b which indicate the breaking point position. Here I assume the breakpoint distributs uniformly on its possible position set in Section. 3.1

Algorithm 2 Metropolis-Hastings

Uniformly sample the initial $f_1^{(0)}$, $f_2^{(0)}$ from all the permutations
Compute possible region $[l : r]$ for breakpoint
Sample $b \sim Uniform([l : r])$
for t from 1 to N **do**
  Generate f_1' by randomly flip two symbol assignments from $f_1^{(n-1)}$
  if Valid according to the "." and " " support sets **then**
    Compute likelihood ratio $a_1 = \frac{P_{f_1'^{-1}(y_1)} \Pi_{k=2}^{b^{(t-1)}} M_{f_1'^{-1}(y_k), f_1'^{-1}(y_{k-1})}}{P_{f_1^{(n-1)-1}(y_1)} \Pi_{k=2}^{b^{(t-1)}} M_{f_1^{(n-1)-1}(y_k), f_1^{(n-1)-1}(y_{k-1})}}$
    Accept this new state $f_1^{(n)} = f_1'$ with probability $\min\{1, a_1\}$; Otherwise, $f_1^{(n)} = f_1^{(n-1)}$
  end if
  Generate f_2' by randomly flip two symbol assignments from $f_2^{(n-1)}$
  if Valid according to the "." and " " support sets **then**
    Compute acceptance factor $a_2 = \frac{\Pi_{k=b^{(t-1)}}^{m} M_{f_2'^{-1}(y_k), f_2'^{-1}(y_{k-1})}}{\Pi_{k=b^{(t-1)}}^{m} M_{f_2^{(n-1)-1}(y_k), f_2^{(n-1)-1}(y_{k-1})}}$
    Accept this new state $f_2^{(n)} = f_2'$ with probability $\min\{1, a_2\}$; Otherwise, $f_2^{(n)} = f_2^{(n-1)}$
  end if
  Generate b' from Gaussion distribution $N(b^{(n-1)}, \beta(r-l)^2)$ and check whether it is in the region $[l : r]$.
  Compute acceptance factor $a_3 = \frac{P_{f_1^{(n-1)-1}(y_1)} \Pi_{k=2}^{b'} M_{f_1^{(n-1)-1}(y_k), f_1^{(n-1)-1}(y_{k-1})} \Pi_{k=b'}^{m} M_{f_2^{(n-1)-1}(y_k), f_2^{(n-1)-1}(y_{k-1})}}{P_{f_1^{(n-1)-1}(y_1)} \Pi_{k=2}^{b^{(t-1)}} M_{f_1^{(n-1)-1}(y_k), f_1^{(n-1)-1}(y_{k-1})} \Pi_{k=b^{(t-1)}}^{m} M_{f_2^{(n-1)-1}(y_k), f_2^{(n-1)-1}(y_{k-1})}}$
  Accept this new state $b^{(n)} = b'$ with probability $\min\{1, a_3\}$; Otherwise, $b^{(n)} = b^{(n-1)}$
  if Recent log-Likelihood scores are stable **then**
    Break
  end if
end for
Iterate all $b \in [l : r]$ and find the one with maximum number of word matchings
Iterate all the permutation of letters $\pi(k, x, j, q, z)$ and find the one with maximum number of word matchings

2 Accuracy Improvements

2.1 Dealing with Low-Frequency Letters

Our basic model in the part I is based on our assumption that the bi-gram frequency for general English articles has a stable approximating distribution. Then we can use the distance of our decoded text and the empirical transition frequency matrix as our loss function. This approach is intuitive but not effective enough. For example, in *test_ciphertext.txt*, the answer is not equal to the maximum likelihood estimator. This is because our cipher text is not long enough. In the plain text, there are only 5 j's and 0 q. Moreover, j and q has similar empirical probabiliy $p_y(y)$, which is 0.00101834620924219 and 0.0009056323797782092. Frequency-based approaches are not good at dealing with low-frequent cipher characters. We need to introduce semantic information into our algorithm.

Cipher	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z		.
Ans.	w	p	e	i	a	.	m	g	h	r	o	j	c	f	v	d	b	k	q	t	n	l	x		z	s	u	y
ML Est.	w	p	e	i	a	.	m	g	h	r	o	q	c	f	v	d	b	k	j	t	n	l	x		z	s	u	y

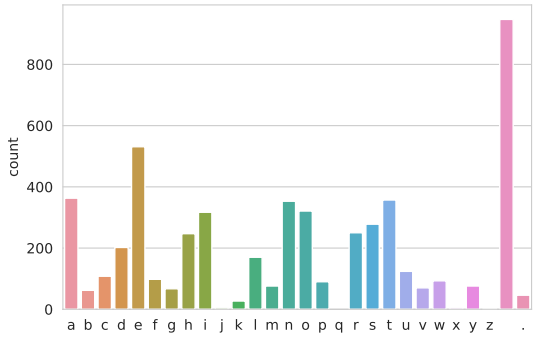

Figure 6: Letter frequency counts for *test_ciphertext.txt*

k, *x*, *j*, *q*, and *z* are the 5 least frequent letters, each of which has frequency lower than 1%[1]. I collect all the frequent words containing *k*, *x*, *j*, *q*, and *z* with length between 3 and 5[2]. I define the following new loss. The W represents the crawled list of words containing *k*, *x*, *j*, *q*, or *z*.

$$L_{pattern}(f|y) = -\sum_{a\in[k,x,j,q,z]}\sum_{p:a\in w} 1[w \in W] \tag{6}$$

In practice, the existence of low frequent letters will not influence the estimation of high frequent letters. This error occurs only if low frequent letters are reversed. This will not influence our minimal loss estimation in the basic model. So we can leave this problem after the MCMC algorithm. In other words, I maximize the number of accurate words containing k, x, j, q, and z when fixing the cipher function on other letters. The code is in Appendix. B.1.

2.2 Fine-tune Breakpoint Position

Having improved the low-frequency words decoding problem, we are thinking of using this same technique to fine-tune the position of breakpoint. The frequency-based loss function can not deal with the precision of several letters. To achieve 100% accuracy, we have to fine-tune it after the MCMC algorithm.

Because the estimation of ciphering functions do not depend on small scale letters, so we can fix the ciphering functions. I have crawled 20115 words from the Internet[3] and use the number of mathced words as a score to decide the position of breakpoint. Several speed-up tricks are applied as well. The code of this part is at the Appendix. B.3.

2.3 Parallelised Monte Carlo

In part I, I have found that the rate of convergence differs significantly among experiments. On the other hand, I hope to avoid this Monte-Carlo process being sucked in a local maximal. The code is in Appendix. B.4

Algorithm 3 Parallelised Monte Carlo

Run MCMC with k different random seeds independently and parallely for n_1 rounds / t_1 time
Choose the result with maximum likelihood
Run the chosen instance for n_2 rounds / t_2 time

I apply 10 threads to run on different random seeds, the total time used will be 2x.

[1] http://norvig.com/mayzner.html
[2] http://scrabble.merriam.com/words/with/z
[3] https://www.thefreedictionary.com/words-containing-z

2.4 Dealing with short articles

For short articles, those with less than 1000 letters, the loglikelihood metrics is not proper. I change the loss function in MCMC with regard to the number of matched words in decoded text. An additional MCMC using this acceptance rate Equ. 8 is run. We then compare the percentage of matched words p_1, p_2 and choose the larger one.

$$a = \min\{1, exp(\frac{p(C, f')}{p(C, f)})\} \tag{7}$$

$$f^* = \underset{f \in \{f_{M_1}, f_{M_2}\}}{\text{argmax}} \; p(C, f) \tag{8}$$

3 Efficiency Improvements

3.1 Precompute Breaking Region

As we all know, there are two facts for every English article. I define a string to be **"valid"** if these two facts hold.

Fact 1. *Both "." and " " can't appear twice in succession.*

Fact 2. *The next letter after "." must be " ".*

Obviously, we have the following finding Lemma. 1

Lemma 1. *If a string holds Fact. 1 and Fact. 2, then its substrings hold as well.*

Lemma 2. *If the support of breakpoint is region $[l : r]$, then substrings $[l :]$ and $[: r]$ hold Fact. 1 and Fact. 2, but substrings $[l-1 :]$ and $[: r+1]$ don't.*

By computing the possible region of breakpoint in advance, we can filter many cases to save time. In practice, I use two binary search to decide the possible region. The code is in Appendix. B.2.

3.2 Precompute Transition Counts

After determinating the region, we can conclude that substring $[: l]$ must be processed for the first ciphering function and substring $[r :]$ must be processed for another. Then we can count the number of transitions in advance.

$$P_{y|f}(y|f) = P_{f^{-1}(y_1)} \Pi_{k=2}^{m} M_{f^{-1}(y_k), f^{-1}(y_{k-1})} = P_{f^{-1}(y_1)} \Pi_{b \in \mathcal{A}} \Pi_{c \in \mathcal{A}} (M_{f^{-1}(b), f^{-1}(c)})^{N(b,c)} \tag{9}$$

4 Experiments: How to Evaluate, and Test

4.1 Estimating Convergence without Ground-Truth

As long as our transition probability matrix can approximate the sequence of letters well. The likelihood and accuracy are positively correlated. I ran my basic model 1000 times with different random seeds. The following line plot shows the positive correlation significantly. Moreover, they tends to have linear relationship when the accuracy is larger than 40%. This finding allow us to use log-likelihood as a metrics when the accuracy can not be directly calculated.

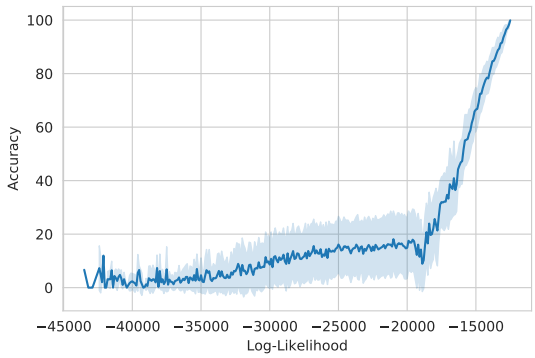

Figure 7: Log-Likelihood and Accuracy have positive correlation

Also, as what is shown on the part I, the acceptance rate decay during the convergence process. If the nearest 500 iterations do not accept any alternative samples, the we can conclude the Monte Carlo Chain converge with high probability.

4.2 Balance between Efficiency and Accuracy

The first trade-off between efficiency and accuracy occurs in deciding the number of iterations for MCMC algorithm. I designed two stopping rules.

- Running time
- The nearest 500 log-likelihood scores are identical

The second trade-off appears on the selection of metrics during MCMC algorithm. Frequency-based method is fast but may not be proper for short text. Word-matching method is proper but requires a lot of computation resources. Therefore, I use frequency-based method for long text and use word-matching method for short text.

4.3 Test

I test my algorithm on the provided data and other data. This algorithm can achieve 100% accuracy in a few minutes.

File	# Letters	Accuracy without Breakpoint	Accuracy with Breakpoint	Elapsed Time
default test	5287	5287/5287	5287/5287	75.4s
feynman	14860	14860/14860	14860/14860	87.1s
paradiselost	5000	5000/5000	5000/5000	87.4s
warandpeace	85491	85491/85491	85491/85491	131.1s

Table 1: Performance

A Part I Basic MCMC Implementation

A.1 Process Data

```
import numpy as np
import pandas as pd
import math, random, time, collections

M = np.zeros(shape=([28, 28]))
with open("data/letter_transition_matrix.csv", "r") as f:
    lines = f.readlines()
    assert len(lines)==28, "The size of the alphabet is 28"
    for i in range(28):
        M[i, :] = [float(x) for x in lines[i].split(",")]
logM = np.log(M) # 0 exists

P = np.zeros(shape=([28]))
with open("data/letter_probabilities.csv", "r") as f:
    line = f.readline()
    assert len(lines)==28, "The size of the alphabet is 28"
    P = [float(x) for x in line[:-1].split(",")]
logP = np.log(P)

with open("data/alphabet.csv", "r") as f:
    line = f.readline()
    assert len(lines)==28, "The size of the alphabet is 28"
    Alphabet = line[:-1].split(",")
    content2idx = {}
    idx2content = {}
    for i in range(28):
        content2idx[Alphabet[i]] = i
        idx2content[i] = Alphabet[i]

with open("test_ciphertext.txt", "r") as f:
    ciphertext = f.read()[:-1] # replace \n at the end of the file
with open("test_plaintext.txt", "r") as f:
    plaintext = f.read()[:-1] # replace \n at the end of the file
```

A.2 MCMC

```
class MCMC:
    def __init__(self, alphabet, ciphertext, answer):
        self.answer = answer
        self.idx2alpha = dict(zip(range(28), alphabet))
        self.alphabet = alphabet.copy()
        random.shuffle(self.alphabet)
        self.cur_f = dict(zip(alphabet, range(28)))
        self.ciphertext_transition = collections.Counter()
        for a in alphabet:
            self.ciphertext_transition[a] = collections.Counter()
        for i in range(1, len(ciphertext), 1): #including /n
            self.ciphertext_transition[ciphertext[i]][ciphertext[i-1]] += 1
        self.ciphertext = ciphertext

    def Pf(self, code2idx):
        logPf = logP[code2idx[self.ciphertext[0]]]
        for a in self.alphabet:
            for b in self.alphabet:
                if self.ciphertext_transition[a][b]!=0:
                    if logM[code2idx[a], code2idx[b]] == -math.inf:
                        return "not exist"
```

```
                    else:
                        logPf += self.ciphertext_transition[a][b] * logM[code2idx[a], code2idx[b]]

        return logPf

    def generate_f(self):
        a, b = random.sample(self.alphabet, 2)
        f2 = self.cur_f.copy()
        f2[a] = self.cur_f[b]
        f2[b] = self.cur_f[a]
        return f2


    def decode(self):
        s = ""
        for c in self.ciphertext:
            s += self.idx2alpha[self.cur_f[c]]
        return s

    def accuracy(self, answer):
        s = 0
        assert len(self.ciphertext)==len(answer), "answer length not align"
        for c in range(len(self.ciphertext)):
            if (self.idx2alpha[self.cur_f[self.ciphertext[c]]]==answer[c]):
                s += 1
#         print("Accuracy {}/{}".format(s, len(self.ciphertext)))
        return s


    def run(self, n=5000):
        start_time = time.time()
        loglikelihood = []
        times = []
        acc = []
        accepted = []
        for t in range(n):
            f2 = self.generate_f()
            pf2 = self.Pf(f2)
            pf1 = self.Pf(self.cur_f)
            rand = random.random()
            if pf1=="not exist" and pf2=="not exist":
                if random.random()<0.5:
                    accepted.append(True)
                    self.cur_f= f2
                else:
                    accepted.append(False)
            elif pf1=="not exist":
                accepted.append(True)
                self.cur_f = f2
            elif pf2=="not exist":
                accepted.append(False)
                pass

            elif rand<min(1, np.exp(pf2-pf1)):
                self.cur_f = f2
                accepted.append(True)

            else:
                accepted.append(False)


            if self.Pf(self.cur_f) != "not exist":
                times.append(t)
                loglikelihood.append(self.Pf(self.cur_f))
```

```
                acc.append(self.accuracy(self.answer)/len(self.ciphertext))
            else:
                times.append(math.nan)
                loglikelihood.append(math.nan)
                acc.append(math.nan)

            if (t%500==0):
                print("Accuracy {}/{}, Log-Liklihood {}, time {} ({}/{})".format(
                self.accuracy(self.answer), len(self.answer), self.Pf(self.cur_f),
                time.time()-start_time, t, n))
        return times, loglikelihood, acc, accepted
```

A.3 Run

```
random.seed(21)
mcmc = MCMC(alphabet=Alphabet, ciphertext=ciphertext, answer=plaintext)
times, loglikelihood, acc, accepted = mcmc.run()
```

B Part II Improvements

B.1 Low-Frequency Letters

```
def refine(content, words_dict):
    words = content.replace("\n", "").split(" ")
    low_freq_words = {}
    for a in ["k", "j", "z", "q", "x"]:
        low_freq_words[a] = [x for x in words if a in x]

    best_per = dict(zip(["k", "j", "z", "q", "x"], ["k", "j", "z", "q", "x"]))
    best_acc = 0

    for per in list(itertools.permutations(["k", "j", "z", "q", "x"])):
        pi = dict(zip(["k", "j", "z", "q", "x"], per))
        score = 0
        for a in ["k", "j", "z", "q", "x"]:
            score += len([x for x in low_freq_words[a] if x.replace(a, pi[a]) in words_dict[a]])
        if score >= best_acc:
            best_acc = score
            best_per = pi

    content = content.replace("k", "1")
    content = content.replace("j", "2")
    content = content.replace("z", "3")
    content = content.replace("q", "4")
    content = content.replace("x", "5")

    content = content.replace("1", best_per["k"])
    content = content.replace("2", best_per["j"])
    content = content.replace("3", best_per["z"])
    content = content.replace("4", best_per["q"])
    content = content.replace("5", best_per["x"])
    return content
```

B.2 Precompute Breakpoint Region

```
l = 0
r = len(self.ciphertext) - 1
```

```
 ciphertext[:x] valid
while (l <= r):
    m = int(l + (r - l) / 2)
    a, b = self.checkvalid(self.ciphertext[m:])
    if (len(a) > 0 and len(b) > 1):
        r = m - 1
    else:
        l = m + 1
self.minbs = l

l = 0
r = len(self.ciphertext) - 1
 ciphertext[x:] valid
while (l <= r):
    m = int(l + (r - l) / 2)
    a, b = self.checkvalid(self.ciphertext[:m])
    if (len(a) > 0 and len(b) > 1):
        l = m + 1
    else:
        r = m - 1
self.maxbs = r
```

B.3 Fine-Tune Breakpoint

```
def refine_breakpoint(self):
    best_breakpoint = self.breakpoint
    best_score = -1e10
    content = self.ciphertext[self.minbs:self.maxbs]
    for b in range(0, self.maxbs-self.minbs, 1):
        decoded = decode_content(content[:b], self.cur_f1) + decode_content(content[b:],
                    self.cur_f2)
        if "." in decoded[:-1] and ". " not in decoded:
            pass
        else:
            score = 0
            ws = decoded.replace(".", "").split(" ")
            for w in ws:
                if len(w)>0 and w in words_dict[w[0]]:
                    score += 1
            if score > best_score:
                best_breakpoint = b
                best_score = score
    self.breakpoint = best_breakpoint+self.minbs
```

B.4 Multiprocessing

```
def run(args):
    ciphertext, seed, runningtime = args
    random.seed(seed)
    mcmc = MCMC(ciphertext=ciphertext)
    loglikelihood, cur_f = mcmc.run(runningtime=runningtime)
    return loglikelihood, cur_f

def multi_merge(ciphertext, runningtime1=-1, np=10, runningtime2=-1):

    p = Pool(processes=np)
    data = p.map(run, zip([ciphertext]*np, [time.time()*random.random() for i in range(np)],
                [runningtime1]*np))
    p.close()
```

```
    best_loglikelihood = -float('inf')
    best_f = None
    for i in data:
        if(i[0]==i[0] and i[0]>best_loglikelihood):
            best_loglikelihood = i[0]
            best_f = i[1]

    final_mcmc = MCMC(ciphertext=ciphertext)
    final_mcmc.cur_f = best_f
    final_mcmc.run(runningtime=runningtime2)
    return final_mcmc.decode(), final_mcmc.cur_f
```

B.5 Word-matching Metrics for Short Text

```
def match_words(content):
    ws = content.replace(".", "").split(" ")
    score = 0
    for w in ws:
        if len(w) > 0 and w in words_dict[w[0]]:
            score += 1
    score = float(score)/len(ws)
    return score

class MCMC_short(MCMC):
    def Pf(self, code2idx):
        decoded_text = decode_content(self.ciphertext, code2idx)
        score = match_words(decoded_text)
        return score

    def generate_f(self, oldf, set1, set2):
        # set1: "."
        # set2 : " "
        newalphabet = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm',
                       'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z']
        a, b = random.sample(newalphabet, 2)
        f2 = oldf.copy()
        f2[a] = oldf[b]
        f2[b] = oldf[a]
        for i, v in f2.items():
            if (v == 27 and i not in set1):
                candidate = random.sample(set1, k=1)[0]
                tmp = f2[i]
                f2[i] = f2[candidate]
                f2[candidate] = tmp
            if (v == 26 and i not in set2):
                candidate = random.sample(set2, k=1)[0]
                tmp = f2[i]
                f2[i] = f2[candidate]
                f2[candidate] = tmp
        return f2
```