**Monte Carlo Methods to Decrypt Substitution Ciphers**

**Abstract**

This project will investigate the use of Markov Chain Monte Carlo (MCMC) methods to tackle random substitution ciphers. Python and its relevant libraries will be used to implement the algorithms and visualize the results.

**Motivation**

Upon first encountering the movie “The Imitation Game”, which features Alan Turing’s efforts to decode Nazi Germany’s Enigma Code during WWII, the concept of using computers to implement logic to solve problems was very fascinating. It depicted the ingenuity of Turing and the tenure in solving even the simplest ciphers. However, with the development of computers and mathematics, it became increasingly accessible to tackle such problems. Serving this as an inspiration, this project will aim to decrypt substitution ciphers using MCMC.

**Introduction and Background**

Cryptography is the study of algorithms to encrypt and decrypt messages between senders and receivers. MCMC method is a class of algorithms for sampling from a probability distribution.

In Cryptography, the original text is referred to as the plain text, and the encrypted text is referred to as the cipher text. The algorithms to perform these encryption and decryption are referred to as ciphers. Ciphers are classified in two categories: classical ciphers and modern ciphers. Classical ciphers refer to simpler ciphers such as substitution ciphers and transposition ciphers that operate at the byte level. Modern ciphers are much more complex such as the RSA and the DES algorithm, which are much more secure and used in modern technologies.

A simple substitution cipher which will be investigated in this project works by replacing individual letters with another one. To achieve reasonable computation time and to simplify the probabilities, only uppercase alphabet characters will be substituted. This means the number of possible keys is equal to 26!. The process involves an encryption key which maps onto the individual alphabet characters and replaces them. For example,

|  |  |
| --- | --- |
| Plain Text | APPLE |
| Encryption Key | A -> P  P -> H  L -> Y  E -> S |
| Cipher Text | PHHYS |
| Decryption Key | P-> A  H -> P  Y -> L  S -> E |
| Decrypted Text | APPLE |

**Figure 1.** Table to show an example of the encryption and decryption cipher process.

In the encryption step, all ‘A’ are replaced with ‘P’, ‘P’ with ‘H’, ‘L’ with ‘Y’ and ‘E’ with ‘S’. Likewise in the decryption step, all the ‘P’ are replaced with ‘A’, ‘H’ with ‘P’, ‘Y’ with ‘L’, and ‘S’ with ‘E’.

In ‘The Imitation Game’, one of the approaches the Allied Forces’ mathematicians used to tackle the cipher involved the fact that certain alphabet characters are used more frequently in the English Language than other characters. This is called Frequency Analysis, the study of the frequency of letters or groups of letters in a cipher text.

**Graph to show the frequency of alphabets in common English Language[1]**

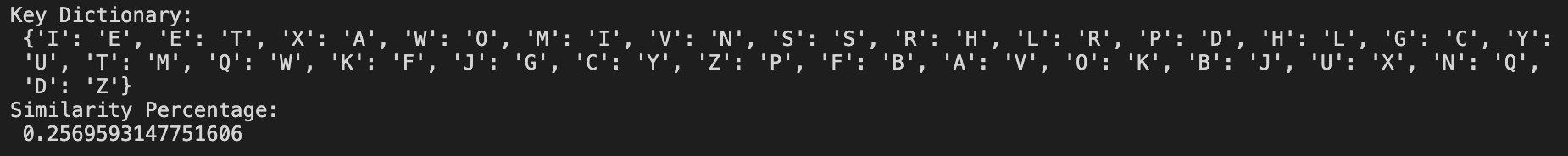


The graph above shows the typical frequency distribution of alphabets in English Language texts. Using this, a first-hand approach the cipher could be implemented.

|  |  |  |  |
| --- | --- | --- | --- |
| Letter | Count | Letter | Frequency |
| E | 21912 | E | 12.02 |
| T | 16587 | T | 9.10 |
| A | 14810 | A | 8.12 |
| O | 14003 | O | 7.68 |
| I | 13318 | I | 7.31 |
| N | 12666 | N | 6.95 |
| S | 11450 | S | 6.28 |
| R | 10977 | R | 6.02 |
| H | 10795 | H | 5.92 |
| D | 7874 | D | 4.32 |
| L | 7253 | L | 3.98 |
| U | 5246 | U | 2.88 |
| C | 4943 | C | 2.71 |
| M | 4761 | M | 2.61 |
| F | 4200 | F | 2.30 |
| Y | 3853 | Y | 2.11 |
| W | 3819 | W | 2.09 |
| G | 3693 | G | 2.03 |
| P | 3316 | P | 1.82 |
| B | 2715 | B | 1.49 |
| V | 2019 | V | 1.11 |
| K | 1257 | K | 0.69 |
| X | 315 | X | 0.17 |
| Q | 205 | Q | 0.11 |
| J | 188 | J | 0.10 |
| Z | 128 | Z | 0.07 |

**Figure 2.** Table to show the distribution of alphabet letters in a sample text of 40,000 English words.[1]

An initial tackle using the frequency analysis had been conducted on a sample test. Only individual alphabets had been used without using any information on frequent bigrams (e.g. ‘HE’, ‘AT’) and trigrams (‘THE’, ‘SHE’). The results were bad:



Only ~26% of the decrypted characters matched the original text and this has no place for any practical application. This is expected, because a naïve approach of using frequency analysis of single letters is open to a lot of errors in the process, because for example obviously the most frequently appearing letter does not always equal to the letter ‘e’- the text may be using a different convention or a different writing style for that matter. Furthermore, the sample text is too small, with only 40,000 words-although the distribution of the frequency of letters may be constant, a larger pool allows for a more diverse analysis of bigrams and trigrams (but this is irrelevant, because only unigrams were used). In addition, a more dynamic process which takes into account more information during computation is favorable. Therefore, a more systematic method, such as MCMC should be used, along with a wider database on the frequency of the English Language including bigrams and trigrams from a large sample. The code for the above frequency analysis is shown below:

#Frequency\_Analysis.py

letter\_frequency = {'E': 12.70, 'T': 9.06, 'A': 8.17, 'O': 7.51, 'I': 6.97, 'N': 6.75, 'S': 6.33, 'H': 6.09, 'R': 5.99, 'D': 4.25, 'L': 4.03, 'C': 2.78, 'U': 2.76, 'M': 2.41, 'W': 2.36, 'F': 2.23, 'G': 2.02, 'Y': 1.97, 'P': 1.93, 'B': 1.29, 'V': 0.98, 'K': 0.77, 'J': 0.15, 'X': 0.15, 'Q': 0.10, 'Z': 0.07}

ETAOIN = 'ETAOINSHRDLCUMWFGYPBVKJXQZ'

ETAOIN\_list = []

for i in range(len(ETAOIN)):

ETAOIN\_list.append(ETAOIN[i])

# print(ETAOIN\_list)

LETTERS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'

text = "LIVITCSWPIYVEWHEVSRIQMXLEYVEOIEWHRXEXIPFEMVEWHKVSTYLXZIXLIKIIXPIJVSZEYPERRGERIMWQLMGLMXQERIWGPSRIHMXQEREKIETXMJTPRGEVEKEITREWHEXXLEXXMZITWAWSQWXSWEXTVEPMRXRSJGSTVRIEYVIEXCVMUIMWERGMIWXMJMGCSMWXSJOMIQXLIVIQIVIXQSVSTWHKPEGARCSXRWIEVSWIIBXVIZMXFSJXLIKEGAEWHEPSWYSWIWIEVXLISXLIVXLIRGEPIRQIVIIBGIIHMWYPFLEVHEWHYPSRRFQMXLEPPXLIECCIEVEWGISJKTVWMRLIHYSPHXLIQIMYLXSJXLIMWRIGXQEROIVFVIZEVAEKPIEWHXEAMWYEPPXLMWYRMWXSGSWRMHIVEXMSWMGSTPHLEVHPFKPEZINTCMXIVJSVLMRSCMWMSWVIRCIGXMWYMX"

def Frequency\_Analysis(text):

letter\_count = {'A': 0, 'B': 0, 'C': 0, 'D': 0, 'E': 0, 'F': 0, 'G': 0, 'H': 0, 'I': 0, 'J': 0, 'K': 0, 'L': 0, 'M': 0, 'N': 0, 'O': 0, 'P': 0, 'Q': 0, 'R': 0, 'S': 0, 'T': 0, 'U': 0, 'V': 0, 'W': 0, 'X': 0, 'Y': 0, 'Z': 0}

for letter in text.upper():

if letter in LETTERS:

letter\_count[letter]+=1

#sort the dictionary from smallest to largest

letter\_count = sorted((value,key) for (key,value) in letter\_count.items())

#reverse the list to start from largest value

letter\_count.reverse()

return letter\_count

# print(Frequency\_Analysis(text))

# output = Frequency\_Analysis(text)

# Map the decrpytion key according to the frequency

def map\_frequency(frequency\_letters):

decryption\_key = []

for letters in range(len(frequency\_letters)):

decryption\_key.append(frequency\_letters[letters][1])

return decryption\_key

# print((map\_frequency(Frequency\_Analysis(text))))

def decrypt(text):

decryption\_key = map\_frequency(Frequency\_Analysis(text))

# Convert the ETAOIN list and decryption key into a dictionary

dictionary = dict(zip(decryption\_key,ETAOIN\_list))

print('Key Dictionary:','\n',dictionary)

# print("Text before", text)

translated\_text = [dictionary[letter] for letter in text]

#list to string

translated\_text = ''.join(translated\_text)

translated\_text = translated\_text.lower()

# print("Text after", text)

return translated\_text

decrypted\_text = decrypt(text)

original\_text = "hereuponlegrandarosewithagraveandstatelyairandbroughtmethebeetlefromaglasscaseinwhichitwasencloseditwasabeautifulscarabaeusandatthattimeunknowntonaturalistsofcourseagreatprizeinascientificpointofviewthereweretworoundblackspotsnearoneextremityofthebackandalongoneneartheotherthescaleswereexceedinglyhardandglossywithalltheappearanceofburnishedgoldtheweightoftheinsectwasveryremarkableandtakingallthingsintoconsiderationicouldhardlyblamejupiterforhisopinionrespectingit"

length\_of\_text = len(original\_text)

similarity\_score = 0

for i in range(length\_of\_text):

if decrypted\_text[i] == original\_text[i]:

similarity\_score += 1

percentage\_similarity = similarity\_score/length\_of\_text

print('Similarity Percentage:','\n',percentage\_similarity)

**Use of MCMC Method**

After using a brute frequency analysis approach and learning about its limitations, an improved method can be used- the MCMC method. MCMC method is good for sampling from a complex probability distribution, but without a metric to check the score of each computation, it will not be any useful. Therefore, learning from the previous part, a much more larger sample text should be mined to get a table of the most common unigrams, and bigrams, and possibly trigrams if necessary. Including trigrams should increase the accuracy of the output, but at a cost of computation time. Therefore, if accurate results can be achieved without the use of trigram data, unigram and bigram data should suffice.

**Methodology**

Learning from the previous section, a larger text is needed to mine the frequency of the unigrams and bigrams (and possibly trigrams) which appear in the text. Project Gutenberg[2] offered free to use and distribute English classics in .txt format, which was precisely what was needed in this method. Initially, War and Peace, and Oliver Twist had been used.

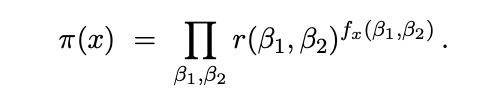
Multiple functions were needed to execute the program:

1. The encrypt\_text function applies cipher onto the input text to output the encrypted text.
2. The text\_frequency\_analysis function mines text data from a sample long text (English Classics in txt format) and creates a dictionary to map unigrams and bigrams and their frequencies.
3. The text\_frequency\_analysis\_oncipher is used to mirror the text\_frequency\_analysis function but is customized for shorter texts of string datatype. Initially, the same function was going to be used, but the different datatypes and methodologies to read them were different, and therefore a new function as created.
4. The cipher\_score function returns the score of the trial cipher.
5. The generate\_random\_cipher function generates random ciphers to test and score.
6. The MCMC\_cipher is the key part where all the helper functions come together to implement the MCMC method.
7. The text\_similarity function compared the similarity of the decrypted text to the original text, and returned the percentage similarity.

In the code, there are comments above the individual functions explaining the logic and reason behind the choice of the algorithm. It was obtained primarily on a Guess, Check, Try It Again basis, which turned out to work very well.

The scoring function was referenced from 2010, Rosenthal:

For each pair of characters β1 and β2 (e.g. β1 =T and β2 =H), we let r(β1, β2) record 5 the number of times that specific pair (e.g. “TH”) appears consecutively in the reference text. Similarly, for a putative decryption key x ∈ X , we let fx(β1, β2) record the number of times that pair appears when the cipher text is decrypted using the decryption key x. To avoid problems from zeroes, we also add one to each of r(β1, β2) and fx(β1, β2). For a particular decryption key x, we then define its score function as follows:[3]



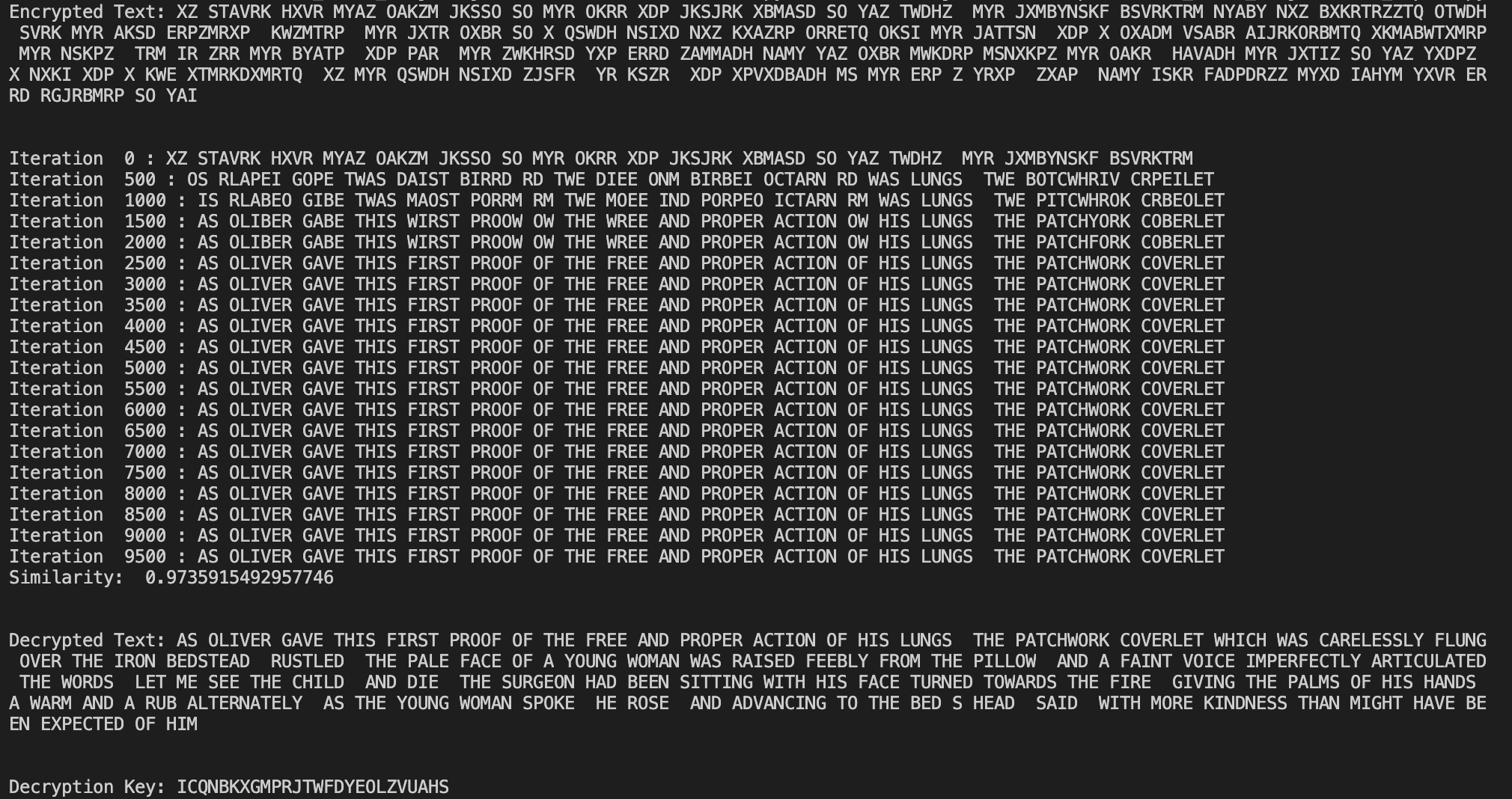
For the acceptance ratio, multiple values had been tried, but none of it converged. For example had been tested for multiple p values ranging from small numbers from 0, mathematical constants (pi, e, etc…) to larger numbers up to 100, none of the results converged. Therefore, upon more research, I referenced the Lecture Notes 11, pg 5/16 acceptance ratio[4], and it converged perfectly.

# referencing Lecture notes 11, pg 5/16, the acceptance ratio,

acceptance\_probability = min(1, math.exp(next\_cipher\_score-current\_cipher\_score))

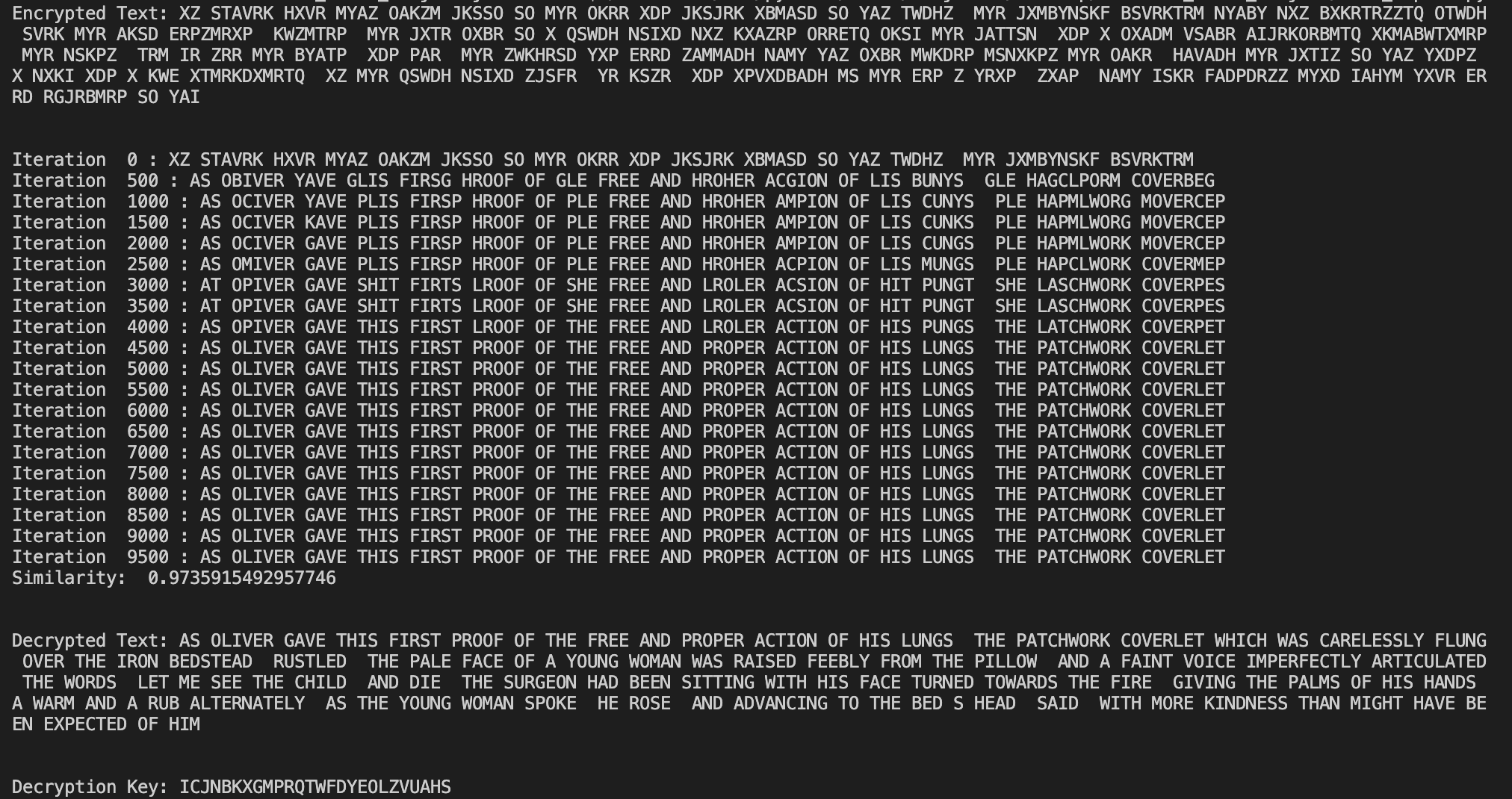
**Results**

Upon executing the code, the results converged well and the similarity was very high for Oliver Twist’s sample text with War and Peace as the text for frequency analysis. However, the result did not always converge (although it did most of the time). If the cipher was misdirected into a bad random direction with a high score, the final cipher was entirely wrong, no matter how long the iteration. This may be improved by implementing trigram frequency analysis in the code. (Refer to mcmc\_run\_code.gif for the animation)



**Figure 3.** War and Peace and Oliver Twist. Output of execution using unigrams and bigrams only.

Trigrams in addition to unigrams and bigrams were used to test if it helped with the accuracy:



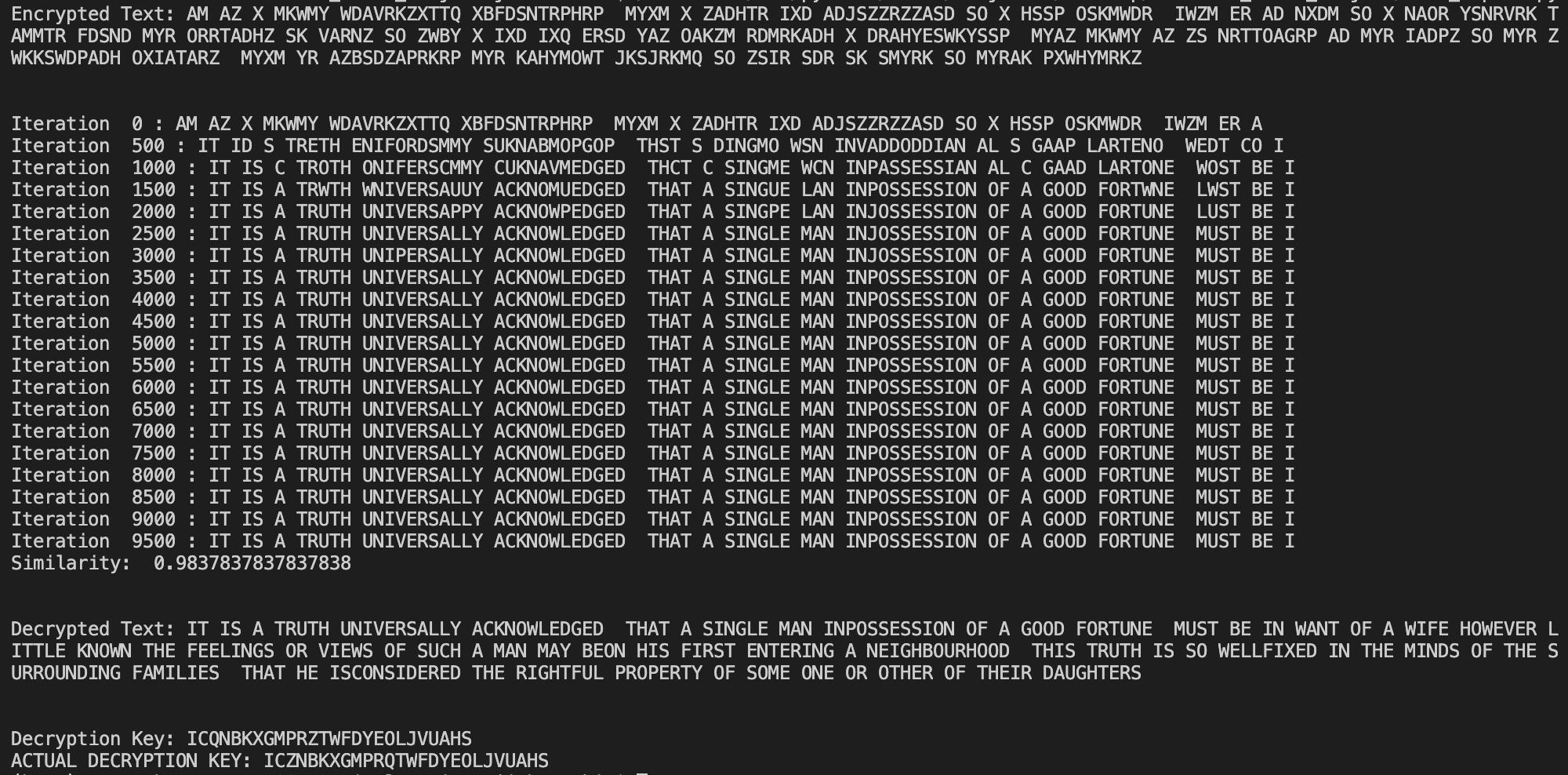
**Figure 4.** War and Peace and Oliver Twist. Output of execution using unigrams and bigrams and trigrams.

The similarity percentage did not significantly change, so this may be suggesting using unigrams and bigrams alone are sufficient. However, to test more on this, new texts were used: Alice in Wonderland and Pride and Prejudice.

When Alice in Wonderland was used as a reference text and sample from Pride and Prejudice was used to test the code, the inclusion of trigrams showed significant improvement in accuracy. Without the frequency analysis of trigrams included as a reference, the similarity was only 70%, but with the trigrams included, without sacrificing computation time (the computation time was negligible, because both were quick), the similarity was 98%.



**Figure 5.** Alice in Wonderland and Pride and Prejudice using unigrams and bigrams only.



**Figure 6.** Alice in Wonderland and Pride and Prejudice including trigrams.

Therefore, it can be concluded that trigrams will generally increase the accuracy of the output, but this is dependent on the sample text and reference text. If the sample and references were relatively compatible and already accurate, using trigrams would not show much change in accuracy, but if the sample and reference text are not very compatible, trigrams improve the accuracy significantly better. This can be generalized to n-grams, but it must be balanced between the computation time and accuracy. Using very high number of n would cost a lot of computation time, but it may not always be needed because as the results show, unigrams and bigrams (and trigrams if needed) as enough to suffice for general English classic texts.

**What I’ve learned**

Overall, before I started the project, I had no understanding of the difficulties involved in the process of deciphering messages. In the process, I learned a lot about cryptography, and how difficult it is to crack even the simplest forms of cryptography. The use of simple frequency analysis to attack simple ciphers was less effective than I had thought, and I began to understand the application of mathematics, and the Monte Carlo methods to very practical uses. In addition, this project further sparked my interest in data science, because after researching on how to collect, clean and analyse data, and trying it out, the whole process was very intuitive and fun. Overall, It really helped solidify my understanding of the Monte Carlo method, and admire how it seems to work.

**Code for MCMC Method**

import string

import math

import random

import numpy as np

# initialize the alphabet and convert it into a list

LETTERS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'

alphabet\_list = list(LETTERS)

# print(alphabet\_list)

# take an input cipher and create a dictionary which maps the alphabet onto its cipher dict.

# e.g {D: I, X:J, Y: P, ...}

def cipher\_dictionary(cipher):

cipher\_dict = {}

for letter in range(len(cipher)):

cipher\_dict[alphabet\_list[letter]] = cipher[letter]

return cipher\_dict

# apply cipher onto the input text to output the encrypted text

def encrypt\_text(text,cipher):

#create the cipher dictionary which maps the alphabet to its substitution letters

cipher\_dict = cipher\_dictionary(cipher)

#convert string format text into list format

text = list(text)

#initialize the encrpyted text string to append the new text

encrypted\_text = []

for letter in text:

if letter.upper() in cipher\_dict:

encrypted\_text.append(cipher\_dict[letter.upper()])

else:

encrypted\_text.append(' ')

# the output will be in list, so to make it easier to view, convert it into a string

encrypted\_text = ''.join(encrypted\_text)

return encrypted\_text

# Mine text data from a sample text, and create a dictionary to map unigrams and bigrams and their frequencies.

# Will be used as a score in the MCMC method.

def text\_frequency\_analysis(text):

letters\_frequency = {}

# to open a file and read it line by line, use the 'with open' block

with open(text) as textfile:

for line in textfile:

# returns a list of individual characters of a single line. Using .strip() clears the '\n' so it should be used to clean the data

letter\_data = list(line.strip())

# print(letter\_data)

for i in range(len(letter\_data)-2):

unigram = letter\_data[i].upper()

bigram = letter\_data[i+1].upper()

# trigram = letter\_data[i+2].upper()

# if its not an empty space and not found in alphabet, it must be garbage words such as '\ufeff' which can be found in text formatting and introduction, but not in the actual text

if unigram != ' ' and unigram not in alphabet\_list:

# replace these garbage words with ' '

unigram = ' '

#bigram referring to letter after the current letter

if bigram != ' ' and bigram not in alphabet\_list:

bigram = ' '

#trigram referring to letter two letters after the current letter

# if trigram != ' ' and trigram not in alphabet\_list:

# trigram = ' '

# combined\_letter = unigram+bigram+trigram

combined\_letter = unigram+bigram

# if letter found in dictionary, add 1 to its count

if combined\_letter in letters\_frequency:

letters\_frequency[combined\_letter] += 1

# if first time encountering this letter, add it to the dictionary.

else:

letters\_frequency[combined\_letter] = 1

return letters\_frequency

# Comment/uncomment these sections to change reference texts

# this will serve as a score metric to test the scores of individual trials.

# score\_metric = text\_frequency\_analysis('war\_and\_peace.txt')

score\_metric = text\_frequency\_analysis('alice\_in\_wonderland.txt')

# Initially the above text\_frequency\_analysis function was going to be used to analyse the cipher

# But, the input datatype is different so a mirror function specific for the input cipher text datatype had been created.

def text\_frequency\_analysis\_oncipher(text):

letters\_frequency = {}

# to open a file and read it line by line, use the 'with open' block

# returns a list of individual characters of a single line. Using .strip() clears the '\n' so it should be used to clean the data

letter\_data = list(text)

# print(letter\_data)

for i in range(len(letter\_data)-2):

unigram = letter\_data[i].upper()

bigram = letter\_data[i+1].upper()

# trigram = letter\_data[i+2].upper()

# if its not an empty space and not found in alphabet, it must be garbage words such as '\ufeff' which can be found in text formatting and introduction, but not in the actual text

if unigram != ' ' and unigram not in alphabet\_list:

# replace these garbage words with ' '

unigram = ' '

#bigram referring to letter after the current letter

if bigram != ' ' and bigram not in alphabet\_list:

bigram = ' '

#trigram referring to letter two letters after the current letter

# if trigram != ' ' and trigram not in alphabet\_list:

# trigram = ' '

# combined\_letter = unigram+bigram+trigram

combined\_letter = unigram+bigram

# if letter found in dictionary, add 1 to its count

if combined\_letter in letters\_frequency:

letters\_frequency[combined\_letter] += 1

# if first time encountering this letter, add it to the dictionary.

else:

letters\_frequency[combined\_letter] = 1

return letters\_frequency

# Instead of using the final decryption key, the best scoring key should be used. Because for example,

# 'OLIVER' may become 'OBIVER' as iterations increase, which is not accurate. Therefore, a score for each

# cipher is needed, and the best scoring cipher should be returned.

def cipher\_score(text, cipher):

working\_text = encrypt\_text(text,cipher)

cipher\_frequency = text\_frequency\_analysis\_oncipher(working\_text)

trial\_score = 0

# loop through the dictionary of the cipher\_frequency dict

# if the key in cipehr\_frequency dict appears in score\_metric, add score.

# after testing simple summation tries for the score, it does not converge. Therefore, new method is neeeded.

# referencing Rosenthal, "This function can be thought of as multiplying, for each consecutive pair of letters in the decrypted

# text, the number of times that pair occurred in the reference text."

for key, value in cipher\_frequency.items():

if key in score\_metric:

trial\_score += math.log(score\_metric[key]\*\*value)

return trial\_score

# current\_cipher = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'

# print(generate\_cipher(current\_cipher))

def generate\_random\_cipher(cipher):

# random shuffle on the cipher to propose a new cipher did not yield good results and most importantly

# it doesn't converge. Therefore a new method is needed.

# cipher = list(cipher)

# random.shuffle(cipher)

# return ''.join(cipher)

# cipher = ''.join(random.sample(cipher,len(cipher)))

# return cipher

# thinking on the opposite end of the extreme, if shuffling the whole list doesnt converge,

# tried shuffling only 2 letters, and it converged. Very surprising.

# upon thinking about the reason, it may be because shuffling the entire list into a new form

# discards the previous cipher despite how high its score may be. It is same as nearly finishing

# the process, and starting over again. Of course this won't converge. Therefore next generated cipher

# should be based on the previous cipher, which is the best when the shuffle is at its minimum.

# the lack of randomness can be compensated by the number of trials. It is expected to be guranteed to

# converge in this way.

random\_number1 = random.randint(0,len(list(cipher))-1)

random\_number2 = random.randint(0,len(list(cipher))-1)

cipher = list(cipher)

temp = cipher[random\_number1]

cipher[random\_number1] = cipher[random\_number2]

cipher[random\_number2] = temp

cipher = ''.join(cipher)

return cipher

def MCMC\_cipher(iterations, text):

current\_cipher = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'

best\_state = ''

trial\_score = 0

for i in range(iterations):

new\_cipher = generate\_random\_cipher(current\_cipher)

current\_cipher\_score = cipher\_score(text, current\_cipher)

next\_cipher\_score = cipher\_score(text,new\_cipher)

# referencing Lecture notes 11, pg 5/16, the acceptance ratio,

acceptance\_probability = min(1, math.exp(next\_cipher\_score-current\_cipher\_score))

if current\_cipher\_score > trial\_score:

best\_state = current\_cipher

# choosing whether to accept the new cipher. Referenced from Lecture 10 example codes. FULL\_MCMC.py

if acceptance\_probability >= random.uniform(0,1):

current\_cipher = new\_cipher

if i%500 == 0:

print('Iteration ',i,':', encrypt\_text(text,current\_cipher)[0:99])

final\_text = encrypt\_text(text,current\_cipher)

return final\_text,best\_state

# Function to test similarity of the decrpyted and original text

def text\_similarity(original\_text, decrypted\_text):

original\_text = original\_text.upper()

decrypted\_text = decrypted\_text.upper()

length\_of\_text = len(original\_text)

similarity\_score = 0

for i in range(length\_of\_text):

if decrypted\_text[i] == original\_text[i]:

similarity\_score += 1

return similarity\_score/length\_of\_text

# Change the sample text by commenting/uncommenting these sections.

# Sample text from Oliver Twist

# testing\_text = "As Oliver gave this first proof of the free and proper action of his lungs, \

# the patchwork coverlet which was carelessly flung over the iron bedstead, rustled; \

# the pale face of a young woman was raised feebly from the pillow; and a faint voice imperfectly \

# articulated the words, Let me see the child, and die. \

# The surgeon had been sitting with his face turned towards the fire: giving the palms of his hands a warm \

# and a rub alternately. As the young woman spoke, he rose, and advancing to the bed's head, said, with more kindness \

# than might have been expected of him: "

# Sample text from Pride and Prejudice

testing\_text = "It is a truth universally acknowledged, that a single man in\

possession of a good fortune, must be in want of a wife.\

However little known the feelings or views of such a man may be\

on his first entering a neighbourhood, this truth is so well\

fixed in the minds of the surrounding families, that he is\

considered the rightful property of some one or other of their daughters."

encryption\_key = "XEBPROHYAUFTIDSJLKZMWVNGQC"

cipher\_text = encrypt\_text(testing\_text,encryption\_key)

decryption\_key = "ICZNBKXGMPRQTWFDYEOLJVUAHS"

print("Encrypted Text:", cipher\_text)

print("\n")

final\_state, best\_state = MCMC\_cipher(10000,cipher\_text)

print('Similarity: ',text\_similarity(testing\_text,final\_state))

print("\n")

print("Decrypted Text:",encrypt\_text(cipher\_text,best\_state))

print("\n")

print("Decryption Key:",best\_state)

print("ACTUAL DECRYPTION KEY:",decryption\_key)

**References**

[1] Retrieved from: http://pi.math.cornell.edu/~mec/20032004/cryptography/subs/frequencies.html

[2] <http://www.gutenberg.org/wiki/Main_Page>

[3] Chen, Jian, and Jeffrey S. Rosenthal. 2011. “Decrypting Classical Cipher Text Using Markov Chain Monte Carlo.” *Statistics and Computing* 22 (2): 397–413. doi:10.1007/s11222-011-9232-5.

[4] Junwei Liu, 2020. PHYS3142 Lecutre Notes 11.