PSTAT 194CS Final Project

4PM Section - Group 4 - Justin Chan, Allen Wang, Karen Zhao

Contents

---

1. Title of the project: **Text Decryption using Markov chain Monte Carlo (MCMC) methods**

2. Group Number from google document: **4pm Section Group 4**

3. justinjchan Justin Chan
   xin874 Allen Wang: Group Leader
   karenezhao Karen Zhao

4. The names and netids of the peer review group members, identifying the group leader:
   Jessica Crum, Yixuan Li, Surabhi Sharma
   (netids and the group leader not identified in rough draft)

5. Links to three most relevant sources for your project/methods.

- "Decrypting classical cipher text using Markov chain Monte Carlo"
- "Behold the power of MCMC"
- "Decipher with Tempered MCMC"

Text Decryption using Markov chain Monte Carlo (MCMC) methods

Abstract

We attempt to implement Markov Chain Monte Carlo (MCMC) methods to decrypt ciphers. We focus on decrypting single substitution ciphers using bi-grams with the Metropolis-Hasting Algorithm. We find our algorithms to run fairly slow and produce adequate accuracy. The results suggest that we can potentially improve our algorithm by tuning various parameters and experimenting in a different software.

Introduction

In cryptography, some information (called plain text) is transformed into the unintelligible gibberish (called cipher text) with the information known only by communicating parties (called key). The act of encoding messages to mask their meaning has actually been around for centuries, at least as far back as the Caesar Cipher and as recent as today's chipped credit cards. In this project, we will explore Markov Chain Monte Carlo (MCMC) methods on decoding text.

Cryptography is the study of algorithms to encrypt and decrypt messages between senders and receivers. In cryptography, the original text is called the plain text, and the encrypted text is called the cipher text. The algorithms to perform encryption and decryption are referred to as ciphers. The substitution cipher is one of the most basic forms of encryption. For the purpose of this project, we will focus on a single substitution cipher. This cipher works by replacing each letter with another one. Again, in this project, we only substitute alphabetic letters. (Spaces are left untouched and all other non-alphabetic characters are removed.) The encryption key is just the inverse function of the decryption key. For example, in encryption, all 'I's in the plain text are replaced by letter 'A', 'B's replaced by 'E', etc. For the decryption, all 'A's in the cipher text are replaced by 'I', 'E's replaced by 'B', etc. See below for an example:

plain text	THE PROJECT GUTENBERG EBOOK OF OLOVER TWIST
encryption key	XEBPROHYAUFTIDSJLKZMWVNGQC
cipher text	MYR JKSURBM HWMRDERKH RESSF SO STAVRK MNAZM
decryption key	ICZNBKXGMPRQTWFDYEOLJVUAHS
decrypted text	THE PROJECT GUTENBERG EBOOK OF OLIVER TWIST

The standard approach to breaking substitution ciphers uses frequency analysis: certain letters and their combinations occur more frequently than others in English. Ciphers like substitution ciphers are often broken by comparing the letter frequencies of the cipher text to a reference text (usually a large text such as *War and Peace*). When faced with a simple substitution cipher, the simplest form of frequency analysis is a uni-gram attack, which involves simply replacing the most frequent letter in the cipher text by the most frequent occurred letter in the reference text, and the second-most-frequent letter in the cipher text by the second-most-frequent letter in the reference text, and the third-most by the third-most, and so on. This attack does not succeed very well. (See Reference) This is because some letters have similar frequencies and are thus likely to be interchanged in this process. Because of results like this, more complicated attacks involving pair frequencies have to be employed. This method can be enhanced by considering frequencies of bi-grams (two letters), tri-grams (three letters), or even n-grams (string of n letters). Our project will choose to perform a bi-gram attack, looking at combinations of two letters.

Markov chain Monte Carlo (MCMC) algorithms are popular methods of approximating sampling from complicated probability distributions. MCMC algorithms could be extremely useful in the case where we couldn't exactly calculate the desired distribution due to the high dimensionality. In the past few decades, MCMC algorithms have been used to iteratively search for solutions which allow us to break simple substitution codes. This approach was first introduced by students in the Stanford statistical consulting service when a psychologist from the California state prison system presented a collection of coded messages. Later research

advanced MCMC algorithms and extensively studied the use of MCMC for decryptions of many kinds of ciphers.

In this project, we will start by obtaining a piece of text and form a hypothesis that the data has been scrambled using a single substitution cipher. We don't know the encryption key, and we would like to know the decryption key so that we can decrypt the data and read the text. To create this example, we will get a piece of text from *Oliver Twist*. We scrambled the data using a random encryption key, which we forgot after encrypting and we would like to decrypt this encrypted text using MCMC Chains. At the end, we will compare the resulting decryption key to the **real** decryption key.

Methods

MCMC has been widely used to sample from complicated high-dimensional distribution.

Given observed data $x = x_1, ..., x_n$, and parameters θ, we know x depends on the prior distribution $f_\theta(\theta)$. We use a likelihood function $f(x_1, ..., x_m|\theta)$ to express this dependence. Then, we calculate the joint distribution

$$f_{x,\theta}(x, \theta) = f_{x|\theta}(x_1, ..., x_n|\theta) f_\theta(\theta)$$

By the Bayes Theorem, we have the posterior distribution of θ by

$$f_{\theta|x}(\theta|x) = \frac{f_{x|\theta}(x_1, ..., x_n|\theta) f_\theta(\theta)}{\int f(x|\theta)(x_1, ..., x_n|\theta) f_\theta(\theta) d|\theta}$$

Here, let $\pi'(\cdot)$ be the posterior distribution on a state space X, which is finite in the context of simple substitution cipher. The MCMC algorithm will define an iterative sequence $X_0, X_1, X_2, ...$ which finally converge in distribution to $\pi'(\cdot)$. In the mathematical context,

$$\lim_{n\to\infty} P(X_n \in A) = \int_A \pi(x)dx$$

If our markov chain $\{X_n\}$ is *irreducible and aperiodic* on a finite state space. By the law of large number, we can sample from $\pi'(\cdot)$ and compute its expected values. In our project, we adopt **Metropolis Algorithm** to implement the Monte Carlo Markov Chain.

To attack the simple substitution cipher, we get prior distribution through a plain text, such as *War and Peace*. Then, the likelihood function and posterior distribution can be derived through cipher text and randomly generated encryption key.

Metropolis Algorithm to attack the simple substitution ciphers:
1. Choose an initial state(initial decryption key), and a fixed scaling parameter $p > 0$
2. Repeat the following steps for many iterations
(a) Given the current state x, propose a new state y from some symmetric density $q(x, y)$
(b) Sample $u \sim$ Uniform$[0, 1]$ independently
(c) If $u < (\frac{\pi(y)}{\pi(x)})^p$ then accept the proposal y by replacing x with y, otherwise reject y by leaving x unchanged

To compute the acceptance probability in 2(c), We can introduce a scaling parameter to replace it as $U_n < (\frac{\pi(Y_n)}{\pi(X_{n-1})})^p$. This modification alters probabilities and expected values in $\pi'(x)$ but leave its *mode* unchanged. Therefore, it might help the chain escape from local modes and increase the accuracy. Usually, a higher acceptance rate with a small p lets us try more decryption keys. A smaller acceptance rate with a large p may have higher accuracy but costs much more time to converge. We will test this later.

Also, for a particular decryption key x, we define its score function

$$\pi(x) = \prod_{\beta_1,\beta_2} r(\beta_1, \beta_2)^{f_x(\beta_1,\beta_2)}$$

where $r(\beta_1, \beta_2)$ record the number of times that specific pair appears consecutively in the reference text and $f_x(\beta_1, \beta_2)$ record the number of times that pair appears when the cipher text is decrypted using the decryption key x. Intuitively, the score function is higher when the pair frequencies in the decrypted text most closely match those of the reference text, and the decryption key is thus most likely to be correct.

By the *Markov Chain Convergence Theorem*, this markov chain will converge in probability to its stationary distribution with density proportional to $(\pi(x))^p$ with $\pi(\cdot)$ and we can get a decryption key.

```
# Takes as input a text to decrypt and runs a MCMC algorithm for n_iter.
    Returns the state having maximum score and also
# the last few states

MCMC_decrypt <- function(n_iter, cipher_text, scoring_params) {
  current_cipher <- paste(sample(LETTERS, 26, replace = FALSE), collapse = "")
  state_keeper = c()
  best_state <- ""
  score <- 0
  for (i in 1:n_iter) {
    state_keeper[i] = current_cipher
    score_current_cipher <- get_cipher_score(cipher_text, current_cipher,
        scoring_params)
    proposed_cipher <- generate_cipher(current_cipher)
    score_proposed_cipher <- get_cipher_score(cipher_text, proposed_cipher,
        scoring_params)
    # tune the scaling parameter
    acceptance_probability <- min(1,    exp(score_proposed_cipher-score_current
        _cipher))
    if (score_current_cipher > score) {
      best_state <- current_cipher
    }
    if (random_coin(acceptance_probability)) {
      current_cipher <- proposed_cipher
    }
    if (i%%1000 == 0) {
      show <- paste("iter",i,":",substr(apply_cipher_on_text(cipher_text,
          current_cipher),1,100))
      print(show)
    }
  }

  return(best_state) # state_kepper,
}
```

Testing Methodology

To create the prior distribution, we have decided to use one of the commonly used reference texts, *War and Peace*. To ensure uniformity between different texts, we converted all letters to uppercase and converted all non-alphabetic characters into spaces. So, in total we would have 27 characters, including the 26 English alphabet letters and a space character.

When testing our algorithm, we run the encryption and decryption process $n = 50$ separate times. In each separate simulation, a new random key is generated to encrypt our plain text. Then we perform the algorithm on the cipher text. By the end of the attack, we compare the decryption key found to the actual decryption key. The accuracy is calculated by the number of letters correct out of the 26 alphabet letters.

We consider the run to be a "success" if the decryption key found is exactly the same as the actual decryption key. Obviously, the more successful runs out of n, the better our algorithm performed.

We adjust various parameters in our algorithm to compare results and see which tuning allows the algorithm to perform optimally:
One, Number of iterations. With a fixed scaling parameter of 1, we perform the process described above on 5 different numbers of iterations, ranging from 1000 to 20000.
Two, Scaling parameter p. We can increase/decrease the acceptance rate by lowering/raising the scaling parameter. With a fixed number of iterations at 5000, We perform the process described above on 4 different parameters.

Results

The following is the result of one single run of the MCMC decryption with 10, 000 iterations

```
## [1] "iter 1000 : AS OLIVEN GAVE THIS FINST MNOOF OF THE FNEE ARD MNOMEN
    ACTIOR OF HIS LURGS  THE MATCHWONB COVENLET W"
## [1] "iter 2000 : AS OLIVER GAVE THIS FIRST PROOF OF THE FREE AND PROPER
    ACTION OF HIS LUNGS  THE PATCHWORK COVERLET W"
## [1] "iter 3000 : AS OLIVER GAVE THIS FIRST PROOF OF THE FREE AND PROPER
    ACTION OF HIS LUNGS  THE PATCHWORK COVERLET W"
## [1] "iter 4000 : AS OLIVER GAVE THIS FIRST PROOF OF THE FREE AND PROPER
    ACTION OF HIS LUNGS  THE PATCHWORK COVERLET W"
## [1] "iter 5000 : AS OLIVER GAVE THIS FIRST PROOF OF THE FREE AND PROPER
    ACTION OF HIS LUNGS  THE PATCHWORK COVERLET W"
## [1] "iter 6000 : AS OLIVER GAVE THIS FIRST PROOF OF THE FREE AND PROPER
    ACTION OF HIS LUNGS  THE PATCHWORK COVERLET W"
## [1] "iter 7000 : AS OLIVER GAVE THIS FIRST PROOF OF THE FREE AND PROPER
    ACTION OF HIS LUNGS  THE PATCHWORK COVERLET W"
## [1] "iter 8000 : AS OLIVER GAVE THIS FIRST PROOF OF THE FREE AND PROPER
    ACTION OF HIS LUNGS  THE PATCHWORK COVERLET W"
## [1] "iter 9000 : AS OLIVER GAVE THIS FIRST PROOF OF THE FREE AND PROPER
    ACTION OF HIS LUNGS  THE PATCHWORK COVERLET W"
## [1] "iter 10000 : AS OLIVER GAVE THIS FIRST PROOF OF THE FREE AND PROPER
    ACTION OF HIS LUNGS  THE PATCHWORK COVERLET W"

## Text To Decode: XZ STAVRK HXVR MYAZ OAKZM JKSSO SO MYR OKRR XDP JKSJRK
    XBMASD SO YAZ TWDHZ  MYR JXMBYNSKF BSVRKTRM NYABY NXZ BXKRTRZZTQ OTWDH
    SVRK MYR AKSD ERPZMRXP  KWZMTRP  MYR JXTR OXBR SO X QSWDH NSIXD NXZ KXAZRP
     ORRETQ OKSI MYR JATTSN  XDP X OXADM VSABR AIJRKORBMTQ XKMABWTXMRP MYR
    NSKPZ  TRM IR ZRR MYR BYATP  XDP PAR  MYR ZWKHRSD YXP ERRD ZAMMADH NAMY
    YAZ OXBR MWKDRP MSNXKPZ MYR OAKR  HAVADH MYR JXTIZ SO YAZ YXDPZ X NXKI XDP
     X KWE XTMRKDXMRTQ  XZ MYR QSWDH NSIXD ZJSFR  YR KSZR  XDP XPVXDBADH MS
    MYR ERP Z YRXP  ZXAP  NAMY ISKR FADPDRZZ MYXD IAHYM YXVR ERRD RGJRBMRP SO
    YAI

## Decoded 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
```

MCMC KEY FOUND: ICZNBKXGMPRQTWFDYEOLJVUAHS

ACTUAL DECRYPTION KEY: ICZNBKXGMPRQTWFDYEOLJVUAHS

Out of 26 letters 26 in the MCMC KEY FOUND is correct

In a single run, we discover that the markov chain converges to the desired stationary distribution in the first 5000 iterations since the following decoded texts remain the same and are correct. However, it raises the probability that the excessive MCMC iterations can jump out of the correct stationary distribution. Moreover, since we randomly selected the encryption key and initial key for each MCMC run, it is possible that the markov chain stays in a local mode and never reaches the correct decryption key. In some runs, the decoded text can show totally no sense. To address these two concerns, we tune the number of MCMC iterations and the scaling parameters to optimize our algorithm.

The following table is the results of bi-gram attacks for substitution ciphers, with different numbers of MCMC iterations, and a fixed scaling parameter $p = 1$

Iterations	Accuracy	No. of successful runs out of 50
1,000	0.5192	0
2,000	0.7446	4
5,000	0.7800	7
10,000	0.8215	3
20,000	0.7869	3

The following table is the results of bi-gram attacks for substitution ciphers after 5,000 iterations, for different choices of the scaling parameter p

Scaling parameter p	Accuracy	No. of successful runs out of 50
0.1	0.3277	0
1	0.7800	7
10	0.8431	7
20	0.7369	5
50	0.8008	4

Discussion

When only tuning the MCMC iterations, we discover that more iterations does not necessarily increase the decryption accuracy and the number of successful runs. From looking at **Table 1** we can see our accuracy is greatest with 10,000 iterations at 0.8215385. However, when comparing the the number of successful runs, we can see that 5,000 iterations has the highest runs with 7 successful runs out of 50. This data makes sense to the reader when we take a look at the outputed decrypted text. At around 5,000 iterations, we can see that our once encrypted text is almost fully decrypted. The accuracy keeps increasing as iterations range from 1,000 to 10,000, but 20,000 iterations oddly output a smaller accuracy. This result reflects the concern we mentioned previously. More iterations in our text does not make any significant difference which would explain why 5,000 iterations produces the highest number of successful runs. This was very surprising to us because it is common to think that that with more iterations, the more accurate our model will be. But data shows that too many iterations can actually harm our model and be less efficient.

Reviewing the algorithm, the acceptance rate is determined by $(\frac{\pi(y)}{\pi(x)})^p$, where p is the scaling parameter. Tuning the scaling parameters directly inversely changes the acceptance rate for the MCMC process. If the acceptance rate is high, the markov chain will move very often and will not converge well. We also need a reasonable acceptance rate to let the markov chain have enough number of steps. On the other hand, larger scaling parameters give lower acceptance rates. But when the acceptance rate is too low, the chain moves

slowly and the decryption process can be very expensive. From our results, the scaling parameter should be at least 1, and 10 can be a good option. Usually, a larger value of scaling parameters leads to more successful runs, but the accuracy does not show an increasing trend when we increase the scaling parameter. This is because a large p forces the markov chain to stay at a local mode, while smaller p gives the algorithm flexibility to run through the distribution. In addition, higher scaling parameters give lower acceptance rates which ultimately creates longer runtime for our code. We believe that a scaling parameter of 1 is the most ideal because it still produces a high accuracy score and also has the same number of successful runs as a parameter of 10, but also runs more efficiently.

We also care about how long it takes for our code to run, since a good algorithm should finish within a reasonable time. During our testing, the algorithm took an extensively long time to run. This is mainly due to limitation of R and the overwhelming number of for loops. Since R is an interpreted language (not compiled), loops can be slow compared to languages like Python and C, especially for a large number of iterations. Therefore, with larger simulations for testing and greater number of iterations, we decided to test our algorithm in Python. Python was able to run the code much more efficiently and provided us data for these larger iterations.

MCMC algorithms can be very efficient in decoding text. We are able to iterate our text thousands and thousands of times to produce an estimated key used for decrypting encoded messages. When implementing our MCMC algorithm in R, we discovered: For the limited options we experimented with, having 5,000 MCMC iterations with a scaling parameter of 1 will produce the most efficient algorithm.

Further Extensions

We find that our basic MCMC algorithm doesn't give a perfect result, though the decrypted text is comprehensible to some degree. It might be pretty close but it is not exactly the key we are searching for. The number of successful runs out of 50 is quite lower than what we would expect. There are several things we can experiment to improve our work:

1. Instead of simply seeing how many letters are in the correct place in the decryption key found, we can compute accuracy by weighing the frequencies of the letters in order to see a more precise scoring of the effectiveness of the algorithm.

2. With the knowledge that larger score functions usually indicate better solutions, therefore, we can derive the decryption key with the largest log scoring function instead of the last key in the MCMC process as our best state. The new algorithm should have better accuracy overall, according to our reference.

3. Instead of bi-gram attacks, we can use different ways of attacking substitution ciphers. As mentioned previously, after doing some research, we have discovered that uni-gram attack does not work well for English texts. In the references we found, tri-gram also do not necessarily increase the accuracy. A better solution would be to use a combination of unigram and bi-gram. This approach would require some modification of our current algorithm.

4. To increase the efficiency of our code, vectorization and paralleling processing in R can help. Another option would be to perform our algorithm in a more commonly used language in text mining, for example, Python or C++.

Appendix

Data Dictionary

scoring_params: dictionary storing all choices of bi-gram in the reference text and their counts
plain_text: the string written in English which is used to encrypt and decrypt
en_key, encryotion_key: the string that indicates the key used to encrypt the plain text
cipher_text: the encrypted string applied by the en_key
de_key, decryption_key: the string that indicates the key used to decrypt the plain text
best_state: the final string that indicates the final decryption key the MCMC chooses to output
n: number of encryption and decryption we want to do
num_success: store the time the best_state is exactly equal to en_key in n iterations
num_error: number of string difference between best_state and de_key
accuracy: vector storing each accuracy ratio between best_state and de_key in each decryption process
decoded_text: the string that indicates text decrypted from cipher_text by the best_state key

Code

```
#options(width = 20) #scipen = 1, digits = 4,
library(knitr)
opts_chunk$set(echo=FALSE, #eval=FALSE, #
                cache=TRUE, autodep=TRUE, cache.comments=FALSE,
                message=FALSE, warning=FALSE)
library(stringr)
library(kableExtra)
# Example of encryption key in Introduction
tab1 <- rbind(c("plain text", "THE PROJECT GUTENBERG EBOOK OF OLOVER TWIST"),
              c("encryption key", "XEBPROHYAUFTIDSJLKZMWVNGQC"),
              c("cipher text", "MYR JKSURBM HWMRDERKH RESSF SO STAVRK MNAZM"),
              c("decryption key", "ICZNBKXGMPRQTWFDYEOLJVUAHS"),
              c("decrypted text", "THE PROJECT GUTENBERG EBOOK OF OLIVER TWIST
                "))
kable(tab1) %>% kable_styling(position = "center", latex_options = "HOLD_
    position")
# This function takes as input a decryption key and creates a dict for key
    where each letter in the decryption key
# maps to a alphabet For example if the decryption key is "DGHJKL...." this
    function will create a dict like {D:A,G:B,H:C....}

create_cipher_dict <- function(cipher){
  cipher_dict <- str_split(cipher, "")[[1]]
  alphabet_list <- LETTERS #list(string.ascii_uppercase)
  names(cipher_dict) <- alphabet_list
  return(cipher_dict)
}

# This function takes a text and applies the cipher/key on the text and
    returns text.
apply_cipher_on_text <- function(text, cipher) {
  cipher_dict <- create_cipher_dict(cipher)
  text <- str_split(str_replace_all(text, "[\n]" , ""),
                    ""
                    )[[1]]
```

```
  newtext <- ""
  for (elem in text){
    if (toupper(elem) %in% cipher_dict) {
      newtext <- paste(newtext, cipher_dict[[toupper(elem)]], sep ="")
      } else {
        newtext <- paste(newtext, "␣", sep = "")
    }
  }
  return(newtext)
}
`%!in%` <- Negate(`%in%`)

create_scoring_params_dict = function(longtext_path){
  scoring_param <- c()
  alphabet_list <- LETTERS
  fp <- readLines(longtext_path)
  for (line in fp){
    newline = trimws(line)
    #newline = str_replace_all(line, " ", "")
    # delete space in the begining and end
    newline = str_split(newline, "")[[1]]
    #print(length(newline))

    for(i in 1:(length(newline)) - 1) {
      alpha_i = toupper(newline[i])
      if(identical(alpha_i, character(0)))
        next
      #print(alpha_i)

      alpha_j = toupper(newline[i+1])
      if(identical(alpha_i, character(0)))
        next
      #print(alpha_j)

      if(c(alpha_i) %!in% alphabet_list & (alpha_i != "␣")){
        alpha_i = "␣"
      }
      if(c(alpha_j) %!in% alphabet_list & (alpha_j != "␣")){
        alpha_j = "␣"
      }

      key = paste(alpha_i, alpha_j, sep = "")

      if(key %in% names(scoring_param)) {
        scoring_param[key] = scoring_param[key] + 1
      } else {
        scoring_param[length(scoring_param) + 1] = 1
        names(scoring_param)[length(scoring_param)] = key
      }
    }

  }
  return(scoring_param)
}
```

```
# This function takes as input a text and creates scoring_params dict which
    contains the
# number of time each pair of alphabet appears together
# Ex. {'AB':234, 'TH':2343, 'CD':23 ..}

score_params_on_cipher <- function(text) {
  all <- c()
  alphabet_list <- LETTERS  #list(string.ascii_uppercase)
  data <- str_split(trimws(text), "")[[1]]
  for (i in 1:(length(data)-1)){
    alpha_i = toupper(data[i])
    alpha_j = toupper(data[i+1])
    if (!(alpha_i %in% alphabet_list) & (alpha_i != "␣")){
      alpha_i = "␣"
    }
    if (!(alpha_j %in% alphabet_list) & (alpha_j != "␣")){
      alpha_j = "␣"
    }
    key <- paste(alpha_i, alpha_j, sep="")

    if(key %in% names(all)) {
        all[key] = all[key] + 1
      } else {
        all[length(all) + 1] = 1
        names(all)[length(all)] = key
      }
  }

  return(all)
}
# This function takes the text to be decrypted and a cipher to score the
    cipher.
# This function returns the log(score) metric

get_cipher_score <- function(text, cipher, scoring_params){
  cipher_dict <- create_cipher_dict(cipher)

  decrypted_text <- apply_cipher_on_text(text,cipher)

  scored_f <- score_params_on_cipher(decrypted_text)

  cipher_score <- 0
  for (k in names(scored_f)) {
    v = unname(scored_f[k])

    if (k %in% names(scoring_params)){
      cipher_score = cipher_score + v*log(unname(scoring_params[k]))
    }
  }

  return(cipher_score)
}
# Generate a proposal cipher by swapping letters at two random location
```

```
generate_cipher <- function(cipher) {
  pos1 <- sample(1:(str_length(cipher)), size=1)  # random.randint(0, len(list
      (cipher))-1)
  pos2 <- sample(1:(str_length(cipher)), size=1)
  if (pos1 == pos2) {
    return(generate_cipher(cipher))
  } else {
    cipher = str_split(cipher, "")[[1]]
    pos1_alpha = cipher[pos1]
    pos2_alpha = cipher[pos2]
    cipher[pos1] = pos2_alpha
    cipher[pos2] = pos1_alpha
    joined <- paste(cipher, collapse = '')
    return(joined)
  }
}

# Toss a random coin with robability of head p. If coin comes head return true
    else false.
random_coin <- function(p) {
  unif <- runif(1)  #if unif>=p:  return False
  return(unif < p)
}
# Takes as input a text to decrypt and runs a MCMC algorithm for n_iter.
   Returns the state having maximum score and also
# the last few states

MCMC_decrypt <- function(n_iter,cipher_text,scoring_params) {
  current_cipher <- paste(sample(LETTERS, 26, replace = FALSE), collapse = "")
  state_keeper = c()
  best_state <- ""
  score <- 0
  for (i in 1:n_iter) {
    state_keeper[i] = current_cipher
    score_current_cipher <- get_cipher_score(cipher_text, current_cipher,
        scoring_params)
    proposed_cipher <- generate_cipher(current_cipher)
    score_proposed_cipher <- get_cipher_score(cipher_text, proposed_cipher,
        scoring_params)
    # tune the scaling parameter
    acceptance_probability <- min(1,   exp(score_proposed_cipher-score_current
        _cipher))
    if (score_current_cipher > score) {
      best_state <- current_cipher
    }
    if (random_coin(acceptance_probability)) {
      current_cipher <- proposed_cipher
    }
    if (i%%1000 == 0) {
      show <- paste("iter",i,":",substr(apply_cipher_on_text(cipher_text,
          current_cipher),1,100))
      print(show)
    }
  }
```

```
  return(best_state) # state_kepper,
}
scoring_params = create_scoring_params_dict('war_and_peace.txt')

plain_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: "

encryption_key = "XEBPROHYAUFTIDSJLKZMWVNGQC"
cipher_text = apply_cipher_on_text(plain_text,encryption_key)
decryption_key = "ICZNBKXGMPRQTWFDYEOLJVUAHS"

best_state <- MCMC_decrypt(10000,cipher_text,scoring_params) #states,

cat("Text To Decode:", cipher_text)
cat("\n")
cat("Decoded Text:", apply_cipher_on_text(cipher_text,best_state))
cat("\n")
cat("MCMC KEY FOUND:", best_state)
cat("\n")
cat("ACTUAL DECRYPTION KEY:",decryption_key)
compare <- str_split(best_state, "")[[1]]==str_split(decryption_key, "")[[1]]
cat("Out of 26 letters", sum(compare), "in the MCMC KEY FOUND is correct")
# take in the encryption key and return the decryption key
decyption_key_solver <- function(en_key){
  a = str_split(en_key,"")[[1]]
  names(a) = LETTERS
  b = LETTERS
  names(b) = LETTERS
  for(i in LETTERS){
    b[a[i]] = i
  }
  return(paste(b, collapse = ""))
}
# Experiment example code
# change number of iterations
n = 100
num_success = 0
accuracy = c()
for(i in 1:n){
  #generate a random key
  en_key = paste(sample(LETTERS, 26), collapse = "")
  actual_de_key = decyption_key_solver(en_key)
  ci_text = apply_cipher_on_text(plain_text,en_key)
```

```
  # change number of MCMC steps
  best_state <- MCMC_decrypt(5000, ci_text, scoring_params)
  decoded_text = apply_cipher_on_text(ci_text,best_state)

  if(actual_de_key == best_state){
    num_success = num_success + 1
  }

  num_error = sum(str_split(actual_de_key,"")[[1]] != str_split(best_state,"")
      [[1]])
  accuracy[i] = (26 - num_error) / 26

  print(i)
}

cat("Accuracy = ", mean(accuracy))
cat("\n")
cat("Number of success out of 20:", num_success)
# Output result manually due to time constraints and efficiency in R
is <- c(1, 2, 5, 10, 20)*1000
acc <- c(0.519230769, 0.744615, 0.780000,  0.82153846, 0.786923)
numI <- c(0,4,7,3,3)
kable(cbind(format(is, big.mark = ","), sprintf("%.4f", acc), numI),
      col.names = c("Iterations", "Accuracy", "No. of<br/>successful runs out
          of 50"),
      align = "lcl", escape = F) %>%
  kable_styling(full_width = F, position = "center", latex_options = "HOLD_
      position") %>%
  row_spec(0 ,  bold = F, extra_css = 'vertical-align: top !important;')
p <- c(0.1, 1, 10, 20, 50)
accI <- c(0.327692, 0.78000, 0.8430769, 0.736923, 0.800769)
numP <- c(0,7,7,5,4)
kable(cbind(prettyNum(p), sprintf("%.4f", accI), numP),
      col.names = c("Scaling parameter $p$", "Accuracy", "No. of<br/>
          successful runs out of 50"),
      align = "lcl", escape = F) %>%
  kable_styling(full_width = F, position = "center", latex_options = "HOLD_
      position") %>%
  row_spec(0 ,  bold = F, extra_css = 'vertical-align: top !important;')
```

References

1. "Decrypting classical cipher text using Markov chain Monte Carlo"
2. "Behold the power of MCMC"
3. "Decipher with Tempered MCMC"
4. Aufgabe Text decryption with Markov Chain Monte Carlo (MCMC)
5. Text Decryption Using MCMC