Decrypting ciphers using Markov Chain Monte Carlo

Course Project for Statistical Computing

Cosmin Borsa[*], *Yichu Li*[†]

11/22/2018

Abstract

We will implement a Markov Chain Monte Carlo algorithm that decodes classical cipher texts.

1 Introduction

We are going to use primarily the work of Jian Chen and Jeffrey S. Rosenthal to implement a Markov Chain Monte Carlo algorithm that decrypts coded messages. We will decode a simple substitution codes, in which each symbol of the coded message stands for a letter of the English alphabet.

2 Reference Text

To solve this problem, we will use the statistics of written English, in particular the frequency analysis of pairs of letters to crack the coded message. This approach works since certain pairs of letters occur more frequently than others. For instance, 'E' is the most used single letter in the English language while 'Z' is the least used single letter. Thus, the pairs 'TH' and 'ER' are more likely to arise in a text than 'ZG'.

The Markov chain has a large but finite state space $\mathcal{X}$ which consists of all the possible decryption keys. Thus, we will use a reference text such as the novel *War and Peace* by Leo Tolstoy to construct a matrix $M(a, b)$ with the transition frequencies of consecutive letters. To do that, we will record in the matrix `transition.mat` the number of times that a certain pair of letters (β_1, β_2) appears in the reference text. We are going to add one to each entry in `transition.mat` to avoid possible computational problems generated by having a frequency with value zero. To simplify the problem, we will disregard any special characters such as a commas, semicolons, or periods, but instead we will track a 27^{th} character, which corresponds to any symbol that is not a letter. After computing the frequency of each pair, we will normalize the matrix `transition.mat` by dividing each entry by its row total. Consequently, we obtain $M(a, b)$. We will now implement this algorithm into R.

```
reference <- readLines("warandpeace.txt")
reference <- toupper(reference)
transition.mat <- matrix(0,27,27)
rownames(transition.mat) <- colnames(transition.mat) <- c(toupper(letters),"")
last.letter <- ""
for (ln in 1:length(reference)) {
```

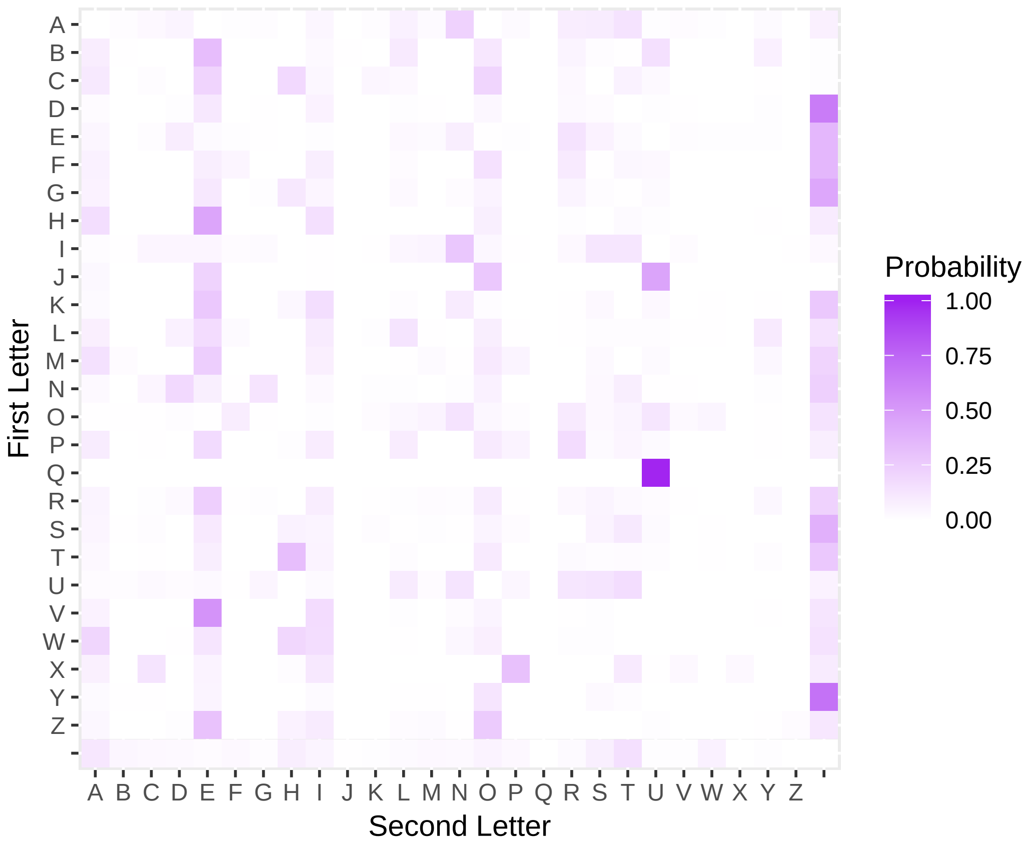
[*]cosmin.borsa@uconn.edu; M.S. in Applied Financial Mathematics, Department of Mathematics, University of Connecticut.

[†]yichu.li@uconn.edu; M.S. in Applied Financial Mathematics, Department of Mathematics, University of Connecticut.

```
  for (pos in 1:nchar(reference[ln])) {
    current.letter <- substring(reference[ln], pos, pos)
    if (current.letter %in% toupper(letters)) {
      transition.mat[rownames(transition.mat) == last.letter,
                colnames(transition.mat) == current.letter] =
        transition.mat[rownames(transition.mat) == last.letter,
                  colnames(transition.mat) == current.letter] + 1
      last.letter <- current.letter
    }
    else {
      if (last.letter != "") {
        transition.mat[rownames(transition.mat) == last.letter,27] <-
          transition.mat[rownames(transition.mat) == last.letter,27] + 1
        last.letter <- ""
      }
    }
  }
  current.letter <- ""
  if (last.letter != "") {
    transition.mat[rownames(transition.mat) == last.letter,27] =
      transition.mat[rownames(transition.mat) == last.letter,27] + 1
  }
  last.letter <- ""
}
transition.prob.mat <- sweep(transition.mat + 1, 1, rowSums(transition.mat + 1),
                             FUN="/")
```

After obtaining $M(a, b)$, we will plot the transition matrix `transition.prob.mat`. It is worth mentioning that the blank space corresponds to the 27^{th} character which stands for any non-letter symbol.

```
ggplot(melt(transition.prob.mat), aes(Var2, Var1)) + geom_tile(aes(fill = value))+
  scale_fill_gradient2(low = "white", high = "purple", limits = c(0,1)) +
  labs(x = "Second Letter", y = "First Letter", fill = "Probability") +
  scale_y_discrete(limits = rev(levels(melt(transition.prob.mat)$Var1))) +
  coord_fixed()
```



3 The Decoding Function

We attempt to decode the cipher by first designing the function `decode` which uses a decryption key `decr.key` from the state space $\mathcal{X}$ to decrypt a message. We will now implement the function in R.

```r
decode <- function(decr.key, coded) {
  coded <- toupper(coded)
  decoded <- coded
  for (i in 1:nchar(coded)) {
    if (substring(coded, i, i) %in% toupper(letters)) {
      substring(decoded, i, i) <- toupper(letters[decr.key ==
                                        substring(coded, i, i)])
    }
  }
  decoded
}
```

4 The Log-Likelihood Function

To find the best decryption key $x \in \mathcal{X}$ for the cipher we will use a score function π_x. The score function π_x assess the goodness of the decryption key by taking the sequence of coded letters in the cipher and yielding the product of frequencies with which the pairs of decrypted symbols appear in the reference text. High values of π_x indicate that the decryption key x is good in assigning pairs of letters while low values point to the fact that the key x is assigning consecutive symbols in the cipher to pairs of letters which are unlikely to occur in standard written English.

$$\pi_x(s_1 s_2 s_3 \ldots) = \prod_i M(s_i, s_{i+1})$$

To avoid computational errors, we will code the log-likelihood function $l(x|s_1 s_2 s_3 \ldots)$ instead of the score function π_x. We thus have

$$l(x|s_1 s_2 s_3 \ldots) = \log\left(\pi_x(s_1 s_2 s_3 \ldots)\right) = \sum_i \log\left(M(s_i, s_{i+1})\right)$$

We are now going to implement the log-likelihood function into R.

```r
loglikelihood <- function(decr.key, decoded) {
  log.sum <- 0
  last.letter <- ""
  for (i in 1:nchar(decoded)) {
    current.letter <- substring(decoded, i, i)
    if (current.letter %in% toupper(letters)) {
      log.sum <- log.sum + log(transition.prob.mat[rownames(transition.mat) ==
                 last.letter, colnames(transition.mat) == current.letter])
      last.letter <- current.letter
    }
    else {
      if (last.letter != "") {
        log.sum <- log.sum + log(transition.prob.mat[rownames(transition.mat) ==
                   last.letter, 27])
        last.letter <- ""
      }
    }
  }

  if (last.letter != "") {
    log.sum <- log.sum + log(transition.prob.mat[rownames(transition.mat) ==
               last.letter, 27])
    last.letter <- ""
  }
  log.sum
}
```

5 Metropolis Algorithm

We will now outline the Metropolis algorithm that we're going to use in decrypting the coded message. However, before we do that we want to mention that in a sense we have solved this problem backwards. We were given a short text which was saved in the variable `correct.text`. Using that text, we are going to use a random encryption key to code the message in the variable `coded`. By following this approach it will be easier to check our work since we know the desired output of the algorithm.

The first step in the Metropolis algorithm is to come up with an initial decryption key $x \in \mathcal{X}$, and to compute the value of the log-likelihood function of that decrytion key. Next, we have to repeat the following steps for many iterations (e.g. $2,000$) by using a while loop. Thus, for each looping we will propose a new decrytion key $y \in \mathcal{X}$ which is generated by randomly switching 2 letters in the current decrytion key x. For instance, if x maps the symbol s_i to A and s_j to H, the proposed key y will map s_j to A and s_i to H. After computing the value of the log-likelihood function of the proposed key, we are going to sample a random variable from a uniform distribution $u \sim U[0,1]$. Next, we are going to check whether the following inequality holds.

$$u < \frac{\pi_y}{\pi_x}$$

If it holds, then we will accept the proposed key y; therefore, y becomes now the current key x. If the inequality doen't hold, we will reject the proposed key. Since we've already coded the log-likelihood function, the algorithm is going to compute $e^{l(y)-l(x)}$ instead of the fraction since $e^{l(y)-l(x)} = \frac{\pi_y}{\pi_x}$. It is also worth noting that the we only count successful attempts to modify the decrytion key as valid iterations.

By the Markov chain convergence theorem, a Markov chain will converge in probability to its stationary distribution π. Therefore, after many iterations, the algorithm is likely to come up with a decryption key which matches the pair frequencies of letters in the decryption text with those of the reference text; thus, coming close to the correct solution.

```r
correct.text <- "AS WHALES GO THROUGH THEIR ANNUAL CYCLES OF SUMMER BINGE
EATING AND WINTER MIGRATIONS THE WAX IN THEIR EARS CHANGES FROM LIGHT TO
DARK THESE CHANGES MANIFEST AS ALTERNATING BANDS WHICH YOU CAN SEE IF YOU
SLICE THROUGH THE PLUGS"
coded <- decode(sample(toupper(letters)), correct.text)

decr.key <- sample(toupper(letters))
i <- max <- 1
iters <- 2000
current.decoded <- decode(decr.key,coded)
current.loglike <- loglikelihood(decr.key,current.decoded)
max.loglike <- current.loglike
max.decoded <- current.decoded
Metropolis.iter <- c()
while (i <= iters) {
  proposal <- sample(1:26, 2)
  propose.decr.key <- decr.key
  propose.decr.key[proposal[1]] <- decr.key[proposal[2]]
  propose.decr.key[proposal[2]] <- decr.key[proposal[1]]
  propose.decode <- decode(propose.decr.key, coded)
  propose.loglike <- loglikelihood(propose.decr.key, propose.decode)
  u <- runif(1)
  if (u < exp(propose.loglike - current.loglike)) {
    decr.key <- propose.decr.key
    current.decoded <- propose.decode
    current.loglike <- propose.loglike
    if (current.loglike > max.loglike) {
      max.loglike <- current.loglike
      max.decoded <- current.decoded
      max <- i
    }
    if (i %% 400 == 0 || i == 1) {
      Metropolis.iter <- rbind(Metropolis.iter, c(i, current.decoded))
    }
    i <- i + 1
  }
}
colnames(Metropolis.iter) <- c("Iteration", "Text")
knitr::kable(Metropolis.iter, booktabs = TRUE, format = "latex", longtable = TRUE,
             caption = 'Iterations of the Metropolis Algorithm') %>%
  kable_styling(bootstrap_options = "bordered", full_width = TRUE) %>%
  column_spec(1) %>%
  column_spec(2, width = "36em")
```

Table 1: Iterations of the Metropolis Algorithm

Iteration	Text
1	JL QCJKVL MX SCNXPMC SCVWN JFFPJK RBRKVL XU LPEEVN TWFMV VJSWFM JFA QWFSVN EWMNJSWXFL SCV QJH WF SCVWN VJNL RCJFMVL UNXE KWMCS SX AJNG SCVLV RCJFMVL EJFWUVLS JL JKSVNFJSWFM TJFAL QCWRC BXP RJF LVV WU BXP LKWRV SCNXPMC SCV IKPML
400	AD HRALED GO TRSOUGR TREIS ANNUAL BYBLED OV DUFFES WINGE EATING ANC HINTES FIGSATIOND TRE HAM IN TREIS EASD BRANGED VSOF LIGRT TO CASK TREDE BRANGED FANIVEDT AD ALTESNATING WANCD HRIBR YOU BAN DEE IV YOU DLIBE TRSOUGR TRE PLUGD
800	AD PHALED GO THSOUGH THEIS ANNUAL CYCLED OM DUFFES RINGE EATING ANW PINTES FIGSATIOND THE PAV IN THEIS EASD CHANGED MSOF LIGHT TO WASK THEDE CHANGED FANIMEDT AD ALTESNATING RANWD PHICH YOU CAN DEE IM YOU DLICE THSOUGH THE BLUGD
1200	AS WHALES GO THROUGH THEIR ANNUAL CYCLES OF SUMMER PINGE EATING AND WINTER MIGRATIONS THE WAV IN THEIR EARS CHANGES FROM LIGHT TO DARK THESE CHANGES MANIFEST AS ALTERNATING PANDS WHICH YOU CAN SEE IF YOU SLICE THROUGH THE BLUGS
1600	AS WHALES GO THROUGH THEIR ANNUAL CYCLES OF SUMMER BINGE EATING AND WINTER MIGRATIONS THE WAV IN THEIR EARS CHANGES FROM LIGHT TO DARK THESE CHANGES MANIFEST AS ALTERNATING BANDS WHICH YOU CAN SEE IF YOU SLICE THROUGH THE PLUGS
2000	AS WHALES GO THROUGH THEIR ANNUAL CYCLES OF SUPPER MINGE EATING AND WINTER PIGRATIONS THE WAV IN THEIR EARS CHANGES FROP LIGHT TO DARK THESE CHANGES PANIFEST AS ALTERNATING MANDS WHICH YOU CAN SEE IF YOU SLICE THROUGH THE BLUGS

6 Conclusion

We can see from Table 1 that after iterating the Metropolis Algorithm $2,000$ times, we obtain decoded messages which are not only readable, but are also surprisingly accurate. In Table 2 we have displayed the decrypted message using the decryption key which maximizes the log-likelihood function. We can observe that even though a few letters might be off, the decoded message can be understood fairly easily.

Table 2: Maximizing the Log-Likelihood Function

Iteration	Text
856	AS WHALES GO THROUGH THEIR ANNUAL CYCLES OF SUMMER PINGE EATING AND WINTER MIGRATIONS THE WAV IN THEIR EARS CHANGES FROM LIGHT TO DARK THESE CHANGES MANIFEST AS ALTERNATING PANDS WHICH YOU CAN SEE IF YOU SLICE THROUGH THE BLUGS

It is also worth mentioning that there might be instances in which the Metropolis algorithm does not generate a readable message in its first run. This might be the case since the initial decryption key could be too far off, and $2,000$ iterations might not suffice in achieving convergence. Since we have limited computing power, we would recommend the reader to run the code a couple of times if the decoded message is not properly displayed.

References

[1] Jian Chen, Jeffrey S. Rosenthal. *Decrypting classical cipher text using Markov chain Monte Carlo*. Statistics and Computing, Volume 22, Issue 2, Pages 397 - 413, 2012.
`http://probability.ca/jeff/ftpdir/decipherart.pdf`

[2] Persi Diaconis. *The Markov Chain Monte Carlo Revolution*. Bulletin of the American Mathematical Society, Volume 6, Number 2, Pages 179 - 205, 2009.
`http://math.uchicago.edu/~shmuel/Network-course-readings/MCMCRev.pdf`

[3] Sam W. Hasinoff. *Solving substitution ciphers* Unpublished report
`http://www.gutenberg.org/files/2600/2600-0.txt`

[4] Stephen Connor. *Simulation and Solving Substitution Codes* Master's thesis, Department of Statistics, University of Warwick.
`http://www-users.york.ac.uk/~sbc502/decode.pdf`

[5] Andrew Landgraf *Text Decryption Using MCMC* Unpublished report
`http://alandgraf.blogspot.com/2013/01/text-decryption-using-mcmc.html`

[6] Jon W. Carr. *MCMC Cipher Solver*
`https://github.com/jwcarr/MCMC-Cipher-Solver`

[7] The Project Gutenberg: *War and Peace* by Leo Tolstoy
`http://www.gutenberg.org/files/2600/2600-0.txt`