Markov Chain Monte Carlo for Decryption

Introduction

So I like to play this one online game, and in it they recently introduced a challenge, whereby a coded journal was presented to players. About 20 coded pages (where each page was only about 2 or 3 sentences) were included, along with one page that was not encoded. The non-encoded page had a riddle, and if you solved the riddle it provided clues on how to decode the cipher on the remaining pages. It was a substitution cipher: every letter in the coded message actually stood for some other real letter, and you basically have to map them to each other.

Now, I hate riddles . . . I like problems, and problem-solving, but I absolutely hate riddles. So I completely ignored the decrypted page, and decided to use math and programming instead. This entry summarizes the approach I used (and it worked!).

Before we get even get into the programming, this is an example of what one of the pages looked like. This is what we're trying to solve:

```
# Let's start with a coded sample, taken from a puzzle in an online game
coded_sample = "Lg amp xffqb bfzam fo tcamp qtip, l ofzgq t bhtii rtypcg spgptam tg fzarcfeelgn fo cfrj
```

Looks like complete nonsense to me. Let's start by importing some R packages we'll need along the way, but the cool thing is that neither of these are actually required – one is for plot formatting and the other is for something called piping. So you can do all of this in base R!

```
require( dplyr )
require( RColorBrewer )
```

Teaching our Algorithm How to Read English

Before we ask a computer to read a coded message and spit it back out in good English for us, we need to teach it good English. We'll do this by importing some English text and using that as a calibration. We'll need a large body of text to do this, so let's use "War and Peace".

```
# import the War and Peace text and convert it all to upper-case characters
calibration_document=
  readLines("war_and_peace.txt")  %>%
  toupper()
```

Now, when I say we need to "teach it good English", that was probably an overstatement. Much more simply, we're just looking for spelling patterns. Specifically, given some starting letter (or space), what is the probability of some other letter (or space) coming after it?

So, we'll go through War and Peace, line by line, and then character by character within those lines, and count the number of times that one letter is followed by another letter. This will form what we'll call a transition matrix.

```
# Initialize a matrix that's 27x27 -- 26 letters plus 1 for spaces
transition_matrix = matrix(0,27,27)

# Label all the rows and columns
rownames(transition_matrix) =
  colnames(transition_matrix)=
  c( " " , toupper(letters))
```

```r
# Initialize our previous_letter variable, which will be used to keep track of the
# last letter -- this is important because we are looking for patterns from one
# variable to the next
previous_letter = " "


# Loop through each line of the War and Peace document
for( line in seq_along( calibration_document ) ){

  # Extract a single line and store it
  tmp_string = calibration_document[ line ]

  # Loop through each character in the line we extracted
  for( char in 1:nchar( tmp_string ) ) {

    # extract the single character and also convert it to upper case
    upper_case_letter =
      tmp_string %>%
      substring( char , char ) %>%
      toupper()

    # Check to see if the character we extract is one of the characters that we care about.
    # Commas, for example, we don't care about -- they will push us to the ELSE clause of
    # this IF statement, where they will not be counted and the previous_letter will
    # be set to a space
    if( upper_case_letter %in% rownames( transition_matrix ) ) {

      if( !( upper_case_letter == " " & previous_letter == " " ) ) {

        transition_matrix[ previous_letter , upper_case_letter ] =
          transition_matrix[ previous_letter , upper_case_letter ] + 1

        previous_letter = upper_case_letter

      }

    } else if( previous_letter != " " ) {

      upper_case_letter = " "

      transition_matrix[ previous_letter , upper_case_letter ] =
        transition_matrix[ previous_letter , upper_case_letter ] + 1

      previous_letter = " "

    }

  }

  # print some status updates since this will take a few minutes
  if( ( line %% 1000 ) == 0 ) {
    cat(   round( line/length(calibration_document) * 100 , 1 )
           , '% complete. \n'    ) }
```

```
}
```

```
## 3.8 % complete.
## 7.6 % complete.
## 11.4 % complete.
## 15.2 % complete.
## 19 % complete.
## 22.9 % complete.
## 26.7 % complete.
## 30.5 % complete.
## 34.3 % complete.
## 38.1 % complete.
## 41.9 % complete.
## 45.7 % complete.
## 49.5 % complete.
## 53.3 % complete.
## 57.1 % complete.
## 60.9 % complete.
## 64.8 % complete.
## 68.6 % complete.
## 72.4 % complete.
## 76.2 % complete.
## 80 % complete.
## 83.8 % complete.
## 87.6 % complete.
## 91.4 % complete.
## 95.2 % complete.
## 99 % complete.
```

And now we have all the letter patterns of War and Peace embedded in this transition matrix. So if you were to start from the "Q" row, then trace over to the "U" row, the number in that grid will tell you how many times the letter "Q" was followed by the letter "U". Let's check:

```
# How many times was the letter "Q" followed by the letter "U" in War and Peace?
transition_matrix[ "Q" , "U" ]
```

```
## [1] 2329
```

Note that this is a raw count – specifically, the number of times this pattern occured in all of Ware and Peace. But what's much more useful are probabilities: given "Q" as a starting letter, what is the probability that the next letter will be a "U"?

To get this, we take each row of the transition matrix, and divide it by the total number of counts in that same row:

```
# normalize each row of the matrix in order to convert to probabilities
transition_probs = sweep( transition_matrix + 1
                          , MARGIN = 1
                          , STATS = rowSums(transition_matrix+1)
                          , FUN="/")

cat( round( transition_probs[ "Q" , "U" ] * 100 , 1 ) , '%' )
```

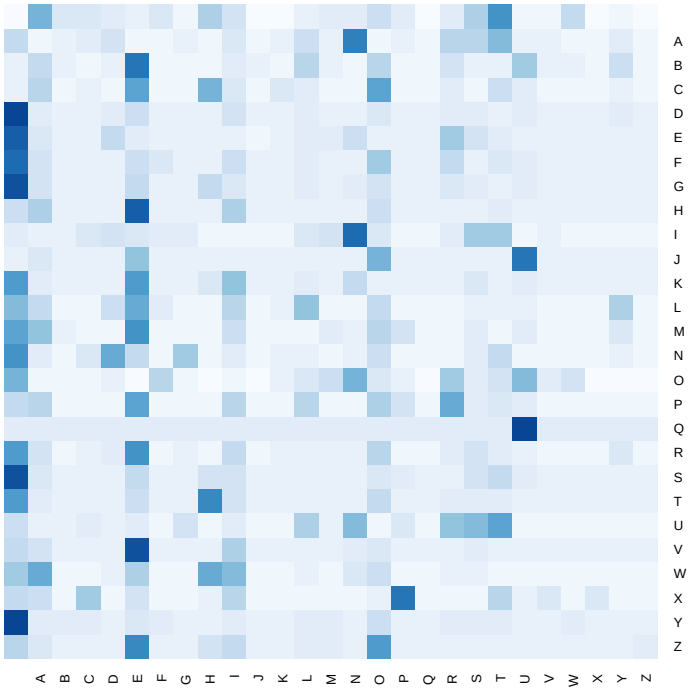
```
## 98.8 %
```

Ah! That's much more useful! The 2,329 above doesn't really tell me much, but the 98.8% is so much more intuitive! By an overwhelming majority, the letter Q is followed by the letter U . . . that makes sense! Queen,

quiet, quaint . . .

Let's also plot the transition probability matrix as a heatmap, where the darker the blue in the grid, the more likely that letter combination is to occur.

```
heatmap_color = colorRampPalette(brewer.pal(8, "Blues"))(25)
heatmap( transition_probs
         , Colv = NA, Rowv = NA
         , cexRow = 0.5 , cexCol = 0.5
         , revC = TRUE
         #, scale = "row"
         , col = heatmap_color
         , xlab=NA, ylab=NA
         , main= NA
)
```



And again I would just highlight as an example the very dark blue grid cell at the intersection of row Q and column U.

Create a Decoding Function

We're going to be trying many, many combinations of possible cipher keys and trying to decode the message each time, so let's create a function that does that for us, so that we can just keep calling it:

```
DECODED_MESSAGE <- function(
  legend = legend
  , coded_message = coded_sample
) {
  decoded_message = coded_message

  for(let in 1:nchar(coded_message) ) {
    tmp_letter = coded_message %>%
```

```
      substring(let,let) %>%
      toupper()

    if( tmp_letter %in% legend ) {
      substring(decoded_message,let,let) =
        names( which( legend == tmp_letter ) )
    }
  }
  return( decoded_message )
}
```

All that's happening here is that we basically pass it some coded message as well as whatever we think the legend or key or cipher is, and it passes back to us the decoded message. Let's try this was some initial random guess of a legend:

```
# initialize some starting guess for the decoded message
legend = sample( toupper(letters) )
names(legend) = toupper(letters)

legend
```

```
##   A   B   C   D   E   F   G   H   I   J   K   L   M   N   O   P   Q   R
## "Q" "N" "O" "Y" "P" "B" "W" "R" "K" "Z" "T" "S" "M" "A" "G" "L" "F" "E"
##   S   T   U   V   W   X   Y   Z
## "J" "U" "V" "C" "D" "I" "X" "H"
```

```
DECODED_MESSAGE( legend , coded_sample )
```

```
## [1] "PO NME YQQAF FQJNM QC KVNME AKXE, P CQJOA K FZKXX HKDEVO LEOEKNM KO QJNHVQRRPOB QC VQHSF.  APBBI
```

Evaluating the Accuracy of a Decoded Message

We've randomly generated some initial guess at what letters in the coded message map to which "real" letters. But how do we know how right we were? This is where our War and Peace calibration comes in.

We can measure the patterns in our decoded message (which letters follow which letters), and then compare that to the patterns from the War and Peace calibration. The better our legend, the better our decoded message should match the patterns in War and Peace.

Again, we'll be evaluate lots and lots of these guesses, so let's write a function that will do this for us:

```
LOG_PROB <- function(
  legend = legend
  , decoded_message = DECODED_MESSAGE( legend = legend , coded_message = coded_sample )
  , transition_probs = transition_probs
) {
  log_prob = 0

  previous_letter = " "
  for( i in 1:nchar(decoded_message) ) {

    tmp_letter = substring(decoded_message,i,i)

    if( tmp_letter %in% c( " " , names(legend) ) ) {
```

```
      if( !( tmp_letter == " " & previous_letter == " " ) ) {

        log_prob = log_prob +
          log( transition_probs[ previous_letter , tmp_letter ] )

        previous_letter = tmp_letter

      }

    } else {

      tmp_letter = " "

      log_prob = log_prob +
        log( transition_probs[ previous_letter , tmp_letter ] )

      previous_letter = " "

    }
  }
  return( log_prob )
}
```

And now we're pretty much ready for the actual decoding to happen.

Decoding the Message

Everything we start with at this point should be familiar from the above. Without going line-by-line, the procedure or algorithm can be summarized as follows:

1. Start with some initial guess for the legend. Decode the message according to this guess and evaluate its accuracy.
2. Now start the algorithm loop:

- Randomly select two letters in your legend to swap. Let's say A was mapped to F and G was mapped to J, and these are the two randomly selected . . . our new legend will be identiccal, except that A will now be mapped to J and that G will now be mapped to F.
- Decode the message according to this new guess and evaluate its accuracy.
- The difference between the accuracy of the new guess and the old guess is compared to a random number. The larger the difference, the more likely it is that this new guess will be accepted.
- If the new guess is the best seen so far in terms of accuracy, it is saved and the process repeats.

The code looks like this:

```
# set some control paramters
i = 1
iterations = 2000

# initialize some starting guess for the decoded message
# (we already did this above, but I do it again here for ease of reference and following along)
legend = sample( toupper(letters) )
names(legend) = toupper(letters)

# What text does our initial guess give us as a decoded message?
```

```
decoded_guess = DECODED_MESSAGE( legend , coded_sample )

# How does our initial guess score against our calibrated transition probabilities?
guess_log_likelihood = LOG_PROB( legend , decoded_guess , transition_probs )

# Initialize some of the tracking metrics
# These are our first guesses -- so they're our best so far
best_guess_likelihood = guess_log_likelihood
best_guess_decoded_message = decoded_guess

k = kmax = lik_thru_time = 0 # initialize a counter

while( i <= iterations ) {

  k = k + 1
  if( k > kmax ){ kmax = k}

  # randomly select two letters to switch, with uniform probability
  switch_sample = sample( 1:26 , 2 )

  # make a copy of the legend and then perform the switch
  new_legend = legend
  new_legend[ switch_sample[1] ] = legend[ switch_sample[2] ]
  new_legend[ switch_sample[2] ] = legend[ switch_sample[1] ]

  # with this new legend that we've randomly created, try to decode the message again
  new_decoded_guess = DECODED_MESSAGE( new_legend , coded_sample )
  # then calculate the probability "score" of this new guess
  new_guess_log_likelihood = LOG_PROB( new_legend , new_decoded_guess , transition_probs )

  random_numb = runif(1)

  if( random_numb < ( new_guess_log_likelihood - guess_log_likelihood ) ) {
    legend = new_legend
    decoded_guess = new_decoded_guess
    guess_log_likelihood = new_guess_log_likelihood

    if ( guess_log_likelihood > best_guess_likelihood ) {
      best_guess_likelihood = guess_log_likelihood
      best_guess_decoded_message = decoded_guess
      lik_thru_time = append( lik_thru_time , best_guess_likelihood )
    }

    cat(i,"/",k,best_guess_decoded_message,"\n")
    i=i+1
    k = 0

  }

}
```

You can see in the code that there is an "i" loop counter – at around i=80 I got the following answer:

- IN THE WOODS SOUTH OF ARTHE DALE, I FOUND A SMALL CAVERN BENEATH AN

> OUTCROPPING OF ROCKS. DIGGING SOON REVEALED A FURTHER TUNNEL AGLOW WITH GEMS. I EXPLORED AS DEEP AS I WAS ABLE BY THEIR LIGHT, BUT FURTHER IN IT WAS TOO DARK TO CONTINUE, AT WHICH POINT A MASSIVE SCALED CREATURE SAVAGED ME FROM BEHIND. IT WAS NO MERELEW. I WAS ABLE TO ESCAPE AND TEND MY WOUNDS, AND WHEN I RETURNED THE FOLLOWING ANDU, THE GEMS WERE GONE

Looks good to me!

Conclusion

Like I said, this was for a game, so obviously not the best code. For example, I did note several instances of the algorithm getting "stuck" in a local optimum that clearly made no sense in terms of actual language. So you kind of have to watch it.

I also didn't introduce any sort of automatic stopping mechanism – I would just sort of watch the outputs and wait for them to look like something that made sense.

Both of these could probably be implemented to make it more automatic!