Zürcher Hochschule
für Angewandte Wissenschaften

School of
Engineering

IDP Institut für Datenanalyse
und Prozessdesign

FS 2016

Stochastische Prozesse

Aufgabe Text decryption with Markov Chain Monte Carlo (MCMC)

The goal of this exercise is to explore the Metropolis-Hastings Algorithm applied to text decryption. Given an encrypted target text (such as, for instance, produced by the Enigma machine during World War II) you need to decrypt (decode) it so that it becomes readable. The number of characters in an encrypted word corresponds to the number of characters in the decypted word.

The encrypted text is

```
[1] "TRSGS WQLFRSKSUFS ESKVTRPES OKYQLTYQRAU WRS EYGSU FSU QSDQ SUQWPETLSWWSTQ"
[2] " GRQQS WPERPXSU WRS IRK FRSWSU QSDQ, MYTTW WRS SW OSWPEYMMQ EYGSU"
[3] " FSK SKWQS GSXAIIQ FYUU FSU YLWOSWPEKRSGSUSU BKSRW"
[4] " RPE EAMMS REUSU EYQ FRS JAKTSWLUO WQAPEYWQRWPES BKAVSWWS OSMYTTSU"
[5] " JRSTTSRPEQ WSESU CRK LUW HY RU FSK JAKTSWLUO WQYQRWQRWPESW FYQYIRURUO"
[6] " WA LUF ULU UAPE JRST WBYWW RU FSK MKYPXCAPES LUF JRST SKMATO GSR FSU BKLSMLUOSU"
[7] " REK ATRJSK FLSKK"
```

You may read more on the methodology used in the background information at the end of this exercise.

You are given the following auxiliary functions: - decode.R - log.prob.R - proposed_mapping.R

- ***decode*** is a function which, given a coded characted string and a mapping, applied the mapping to the string. For example, let the mapping be defined as

```r
set.seed(2)
( mapping = sample(LETTERS) )
```

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

Let's apply the mapping above to the encrypted text:

```r
text_mapped = decode( mapping, text_coded )
unlist(strsplit(text_mapped, split = ".", fixed = TRUE))
```

```
[1] "FBYXY LPZWBYMYEWY AYMNFBHAY SMKPZFKPBTE LBY AKXYE WYE PYDP YEPLHAFZYLLYFP"
[2] " XBPPY LHABHQYE LBY IBM WBYLYE PYDP, RKFFL LBY YL SYLHAKRRP AKXYE"
[3] " WYM YMLPY XYQTIIP WKEE WYE KZLSYLHAMBYXYEYE VMYBL"
[4] " BHA ATRRY BAEYE AKP WBY JTMFYLZES LPTHAKLPBLHAY VMTNYLLY SYRKFFYE"
[5] " JBYFFYBHAP LYAYE GBM ZEL UK BE WYM JTMFYLZES LPKPBLPBLHAYL WKPKIBEBES"
[6] " LT ZEW EZE ETHA JBYF LVKLL BE WYM RMKHQGTHAY ZEW JBYF YMRTFS XYB WYE VMZYRZESYE"
[7] " BAM TFBJYM WZYMM"
```

- ***log.prob*** is a function which goes over all characters in a string and computes the loglikelihood of this input string (the "decoded" argument) using the transition matrix *trans.prob.mat*. For example,

Let's compute the loglikelihood for the encrypted text mapped above (text_mapped):

```
( loglike_current = log.prob( mapping, text_mapped ) )
```

```
[1] -2157.394
```

- ***proposed_mapping*** is a function which, given a mapping ("mappoing0" argument), interchanges two random letters in the mapping, which can then be applied to the text. It changes a mapping randomly, and returns the new mapping. For example, let's apply this function to the mapping and the coded text above:

```
( prop.mapping = proposed_mapping( mapping0 = mapping) )
```

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

```
prop.decode = decode(prop.mapping, text_coded)
unlist(strsplit(prop.decode, split = ".", fixed = TRUE))
```

```
[1] "FBYXY LPZWBYMYEWY AYMNFBHAY SMKPZFKPBTE LBY AKXYE WYE PYIP YEPLHAFZYLLYFP"
[2] " XBPPY LHABHQYE LBY DBM WBYLYE PYIP, RKFFL LBY YL SYLHAKRRP AKXYE"
[3] " WYM YMLPY XYQTDDP WKEE WYE KZLSYLHAMBYXYEYE VMYBL"
[4] " BHA ATRRY BAEYE AKP WBY JTMFYLZES LPTHAKLPBLHAY VMTNYLLY SYRKFFYE"
[5] " JBYFFYBHAP LYAYE GBM ZEL UK BE WYM JTMFYLZES LPKPBLPBLHAYL WKPKDBEBES"
[6] " LT ZEW EZE ETHA JBYF LVKLL BE WYM RMKHQGTHAY ZEW JBYF YMRTFS XYB WYE VMZYRZESYE"
[7] " BAM TFBJYM WZYMM"
```

```
( prop.loglike = log.prob(prop.mapping, prop.decode) )
```

```
[1] -2163.112
```

Tasks:

a) Explain the applicability of each auxiliary function and how you plan to apply them in a MCMC code. Provide pseudocode for the Metropolis-Hastings algorithm.

b) Use the auxiliary functions to implement the Metropolis-Hastings algorithm in a *for*-loop with iterations over 4000 proposed mappings. Be sure to set the seed (*set.seed(1)*) beforehand. Then initialize a mapping, and a dataframe with columns "index", "loglike" and "decoded" to be filled within the *for*-loop. The index column would correspond to the mapping number, loglike will keep track of the corresponding loglikelihood of the current mapping, and the decoded variable will contain the currently decoded text. (Note: decoding could take a few minutes)

c) Provide the best solution achieved by the MCMC and its likelihood. The first one wins the prize.

Optional:

d) What is the difference between the number of iterations and the number of proposed mapping used? How many iterations did the the loop do? How many proposed mappings were rejected?

e) Plot the loglikelihood vs the index number to explore its evolution. What pattern do you observe? what can be said about the spped of convergence to the best solution?

f) Now try setting different starting seeds in b) and rerunning the loop again with 4000 proposed mappings. Is a good(recognisable) solution always achieved? Then try changing the number of proposed mappings to explore the speed of convergence vs. different starting seeds. Which pattern do you contemplate? It may take a while to complete all iterations.

Background Information:

The decryption we will be attempting is called substitution cipher, where each letter of the alphabet corresponds to another letter (possibly the same one). Our sample space is composed of $26! = 4.032915 \times 10^{26}$ possible mappings of the Latin alphabet letters to one another. Clearly, trying out each possible mapping is not feasible, so we need to sample the space of all mappings wisely, and MCMC provides one way to do it[1].

First, one needs a reference text large enough so that a transition matrix from one letter to another could be estimated relatively precisely. In other words, for every letter in the alphabet we are trying to estimate the probability that one letter follows another one. Then a Markov chain is constructed based on this transition matrix. It is a 27 by 27 matrix, where the i^{th} row and j^{th} column is the probability of the j^{th} letter, given the i^{th} letter preceded it.

To create a transition matrix, a Thomas Mann novel "Joseph und seine Brüder"(1492 pages) was used, looping through each letter and counting the number of times each letter followed the previous. A 27th character was also introduced, which was anything that was not a letter - ? ! : ;

[1]You can read more on this in the paper www.probability.ca/jeff/ftpdir/decipherart.pdf

etc. Commas and full stops were left as unchanged and ignored. It should be noted that we are only considering the mapping of letters and it is assumed that the special characters are known. Spaces and periods are assumed to be the same.

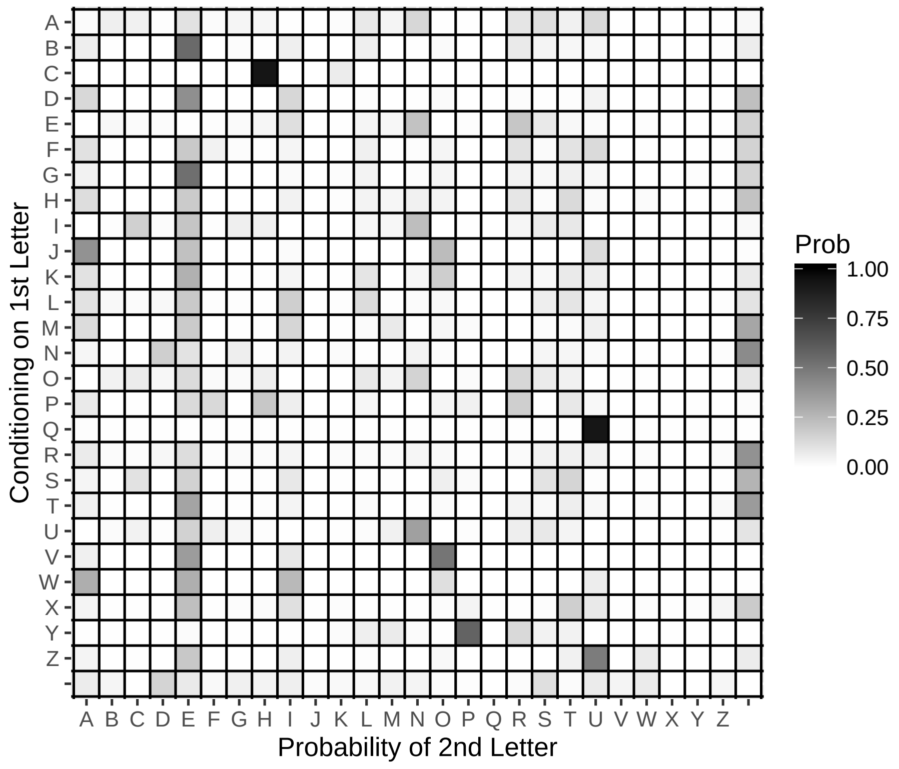
This lets one know which letters are more likely to start a word or end a word. Consecutive entries of these special characters are not considered. Moreover, the special German characters like umlauts were manually substituted with their Latin analogs in the reference text and in the target text which needs to be decrypted:

- ü → ue
- ä → ae
- ö → oe
- Eszett → ss

In the end all characters were capitalized.

After the character counts are made, each cell in the matrix was normalized by dividing by the row total. Before normalizing, 1 was added to each cell so that there are no probabilities of 0. This also corresponds to prior information of each transition being equally likely.

The resulting transition matrix is provided in *trans.prob.mat.Rdata* and is visualized below.


Conditioning on 1st Letter
Probability of 2nd Letter
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
Prob