EE3731C Programming Assignment

20% of Final Grade

Project Deadline: 11.59pm, Friday, Nov 15, 2019

Submit the following electronic files in one zipped folder onto the "CA2 Student Submission" folder on IVLE.

1. **The pdf file of a well-written, concise project report. The report should NOT be longer than 10 pages (font 12, single space, arial). The report filename MUST be "[name_on_matric_card]_[matric_number]_report.pdf".**
2. **Your source code folder and if necessary, a readme file containing instructions to run your code**

Before you start, take note of the following:

1. You may discuss the assignment with your classmates, but must write the code completely on your own.
2. Questions should be tackled sequentially because later questions depend on earlier ones.
3. In addition to this pdf, there is a matlab data file (**sem1_2019_encrypt.mat**) and six matlab functions. One of the functions (**mcmc_decrypt_text.m**) has already been written for you. The remaining functions have to be filled in by you (instructions below).
4. Please use matlab for this assignment.

Acknowledgement: Yidong Chong (NTU, Singapore) suggested this exercise as an excellent introduction to the power of markov chains and probabilistic modeling. The algorithm used in this exercise is from Chapter 1 of the article "The Markov Chain Monte Carlo Revolution" by Persi Diaconis.

Introduction

In this assignment, we explore the use of Markov chains in cryptography. At the end of this assignment, you will be able to take a seemingly random string of characters like this:

> umigbvquovzbkittitkvjdvkbjveblcvjilbpvdnvoijjitkvzcvfblvoiojblvdtvjfbvzutyvvut
> pvdnvfueitkvtdjfitkvjdvpdvvdtgbvdlvjqigbvofbvfupvabbabpvitjdvjfbvzddyvfblvoio
> jblvquovlbupitkvvz jvijvfupvtdvaigj lbovdlvgdtebloujidtovitvijvvvutpvqfujviovjfb
> v obvdnvuvzddyvvvjfd kfjvumigbvvqijfd jvaigj lbovdlvgdtebloujidto...

and decrypt it to become:

alice was beginning to get very tired of sitting by her sister on the bank and of ha
ving nothing to do once or twice she had peeped into the book her sister was read
ing but it had no pictures or conversations in it   and what is the use of a book   t
hought alice without pictures or conversations...

We shall make the following mild assumptions:

1. Each character in the encrypted text corresponds to a unique character in the original (unencrypted) text. This is known as a substitution cipher. In the above example, all instances of 'u' in the encrypted text corresponds to all instances of 'a' in the original text, all instances of 'm' in the encrypted text corresponds to all instances of 'l' in the original text, etc.

2. There are 27 unique characters corresponding to the 26 alphabets and white space. All punctuations are considered white space and white space is considered the 27-th alphabet. This mild assumption simplifies this assignment, but is not a real limitation of the decryption algorithm we will explore.

Q1. Mapping Between Character and Double Arrays (15%)

Let's explore a convenient mapping between character and double arrays, which will simplify many subsequent operations. The benefits of such a mapping will be apparent in Question 2.

The matlab **char** function converts any numeric array into a character array. Conversely, the **double** function converts any input array into a double precision numeric array. More specifically, if the input to **double** is a character array, **double** inverts the **char** function. For example, **char([65 66])** gives 'AB', while **double('AB')** gives [65 66].

(a) Report the results of the following operations in matlab:

>> NumericArray = double('Mary is good at math.')
>> CharacterArray = char(NumericArray)
>> CharacterArray = char([80 114 111 98 108 101 109 32 105 115 32 115 111 108 118 101 100 46])

(b) While **char** and **double** are very useful, it is much more convenient (in this assignment) to have 'a' corresponds to 1, 'b' corresponds to '2', and so on. Fill in the empty matlab function (**char2double.m**) provided in the zip file. The function should take in a $1 \times N$ character array and outputs a $1 \times N$ double array. The character 'a' or 'A' should map to 1, character 'b' or 'B' should map to 2, ..., character 'z' or 'Z' should map to 26, and all other characters should map to 27. For example, both **char2double('abc')** and **char2double('ABC')** should give [1 2 3]. **char2double('A YZ')** should give [1 27 25 26]. Report the results of the following operation in matlab:

>> char2double('Mary is good at math.')

[Hint: The built-in function **double** is useful for writing **char2double.m**].

(c) Fill in the empty matlab function (**double2char.m**) provided in the zip file. The function should take in a $1 \times N$ double array and outputs a $1 \times N$ character array. The double 1 should map to 'a', double 2 should map to 'b', ..., double 26 should map to 'z', and double 27 should map to white space (which corresponds to **char(32)** in matlab). For example, **double2char([1 2 3])** should give 'abc', and **double2char([1 27 26 1])** should give 'a za'. Report the results of the following operation in matlab:

>> double2char([20 15 15 27 13 1 14 25 27 2 21 7 19])

[Hint: The built-in function **char** is useful for writing **double2char.m**].

Q2. Encrypting/Decrypting A Message (10%)

The functions **char2double.m** and **double2char.m** allows us to encrypt or decrypt a message (character array) in a few lines of code. **sem1_2019_encrypt.mat** contains two character arrays (**frank_original_txt** and **frank_encrypted_txt**) and two double arrays (**frank_decrypt_key** and **frank_encrypt_key**).

(a) **frank_original_txt** is a character array corresponding to a text snippet from the book "Frankenstein". **frank_encrypted_txt** was generated by encrypting **frank_original_txt**:

>> frank_original_double = char2double(frank_original_txt);
>> frank_encrypted_double = frank_encrypt_key(frank_original_double);
>> frank_encrypted_txt = double2char(frank_encrypted_double);

frank_encrypt_key is a 1×27 double array where the i-th element specifies which character the i-th alphabet in the original text is mapped to. For example, **frank_encrypt_key(2)** is equal to 9, which means that the 2-nd alphabet ('b') in the original text is mapped to the 9-th alphabet ('i') in the encrypted text. Encrypt the character array 'Mary is good at math.' with **frank_encrypt_key** and report the resulting encrypted text.

(b) Similarly, **frank_encrypted_txt** can be decrypted as follows:

>> frank_encrypted_double = char2double(frank_encrypted_txt);
>> frank_decrypted_double = frank_decrypt_key(frank_encrypted_double);
>> frank_decrypted_txt = double2char(frank_decrypted_double);

frank_decrypt_key is a 1×27 double array where the i-th element specifies which character the i-th alphabet in the encrypted text is mapped to. For example, **frank_decrypt_key(9)** is equal to 2, which means that the 9-th alphabet ('i') in the encrypted text is mapped to the second alphabet ('b') in the decrypted text. Decrypt 'gvwdukdnagdvaza' with **frank_decrypt_key** and report the resulting decrypted text.

Q3. Probability of Consecutive Characters (35%)

The trick to the decryption algorithm in this assignment is to exploit the statistical regularity of normal written English. For example, there is a very high chance for the letter 'u' to immediately follow the letter 'q' in English. Therefore we can quantify the likelihood of any sequence of characters using precomputed statistics of normal written English.

(a) Fill in the empty function **compute_transition_probability.m** provided in the zip file. The function should convert an input $1 \times N$ character array into **pr_trans** (a 27×27 double precision matrix). **pr_trans**(i, j) is the probability the j-th alphabet occurs immediately after the **i**-th alphabet. For example pr_trans$(1, 27)$ is the probability that a white space occurs immediately after the character 'a'. We will estimate pr_trans(i, j) from the input character array as follows:

$$\text{pr\_trans}(i,j) = \frac{1 + \#\text{times } j\text{-th alphabet appears after } i\text{-th alphabet in input text}}{27 + \#\text{times } i\text{-th alphabet appears in input text except at the last position}}$$

The '1' in the numerator and '27' in the denominator ensure none of the probability estimates is equal to zero. For example, if the input character array is 'aaabababa', then $\textbf{pr\_trans}(1,1) = (1+2)/(27+5) = 0.09375$, $\textbf{pr\_trans}(1,2) = (1+3)/(27+5) = 0.125$, and $\textbf{pr\_trans}(1,3) = \cdots = \textbf{pr\_trans}(1,27) = 1/(27+5) = 0.03125$. Perform the following operation in matlab:

>> pr_trans = compute_transition_probability(training_txt);

Report $\textbf{pr\_trans}(1,1)$ and $\textbf{pr\_trans}(2,3)$. What is the highest probability in **pr_trans**? Which alphabetical transition does the highest probability correspond to?

(b) Given **pr_trans** computed using **training_txt**, we can now compute the probability of any sequence (array) of characters. More specifically, given a character array '$c_1c_2\cdots c_N$', we can compute the following probability

$$\text{probability}(\text{'}c_1c_2\cdots c_N\text{'}) = \prod_{n=1}^{N-1} \text{probability of observing } c_{n+1} \text{ immediately after } c_n,$$

where we have assumed the observation of c_{n+1} is dependent only on the immediately preceding letter c_n (i.e., Markov assumption), rather than c_{n-1}, c_{n-2}, etc. Because the above equation involves the multiplication of many numbers less than one, it is numerically more stable to compute the logarithm of the probability:

$$\ln \text{pr}(\text{'}c_1c_2\cdots c_N\text{'}) = \sum_{n=1}^{N-1} \ln\big(\text{probability of observing } c_{n+1} \text{ immediately after } c_n\big), \tag{1}$$

where ln is the natural logarithm. For example, the natural logarithm of the probability of observing 'abca' is equal to $\ln(\text{pr\_trans}(1,2)) + \ln(\text{pr\_trans}(2,3)) + \ln(\text{pr\_trans}(3,1))$.

Fill in the empty function **logn_pr_txt.m** provided in the zip folder using the formula defined in Eq. (1). The function should take as inputs a $1 \times N$ character array and a 27×27 transition probability matrix. The output is a scalar corresponding to the natural logarithm of the probability of observing the character array. Use the **pr_trans** computed from **training_txt**, and perform the following operations in matlab:

>> logn_pr = logn_pr_txt(frank_encrypted_txt, pr_trans)
>> logn_pr = logn_pr_txt(frank_original_txt, pr_trans)

Report the values of logn_pr in the above operations. [Hint: logn_pr for **frank_encrypted_txt** is -8685.4692.]

(c) Given an encrypted message, we can compute the logarithm of $p(\text{encrypted message} \mid \text{decrypt\_key})$ for any decrypt_key by unscrambling the encrypted message using the decrypt_key (see Q2) and then apply **logn_pr_txt.m** to the resulting decrypted text.

Compute and report natural logarithm of $p(\textbf{frank\_encrypted\_txt} \mid \textbf{frank\_decrypt\_key})$.
Compute and report natural logarithm of $p(\textbf{frank\_encrypted\_txt} \mid \textbf{mystery\_decrypt\_key})$.

Q4. Metropolis Algorithm (40%)

From Q3c, we know how to compute the likelihood $p(\text{encrypted message} \mid \text{decrypt\_key})$. Assuming all decryption keys are equally likely a priori, the posterior $p(\text{decrypt\_key} \mid \text{encrypted message}) \propto p(\text{encrypted message} \mid \text{decrypt\_key})$ by Bayes' law. Therefore the Metropolis algorithm can be used to sample from the posterior by using $p(\text{encrypted message} \mid \text{decrypt\_key})$ as a surrogate (the proportionality constants cancel out as explained in class). The algorithm is summarised in the Appendix and is mostly implemented for you except for a critical portion of steps (iii) and (iv), which you should implement in metropolis.m:

(a) Fill in the empty function **metropolis.m** provided in the zip file. **metropolis.m** has four inputs: (1) current decrypt_key (1×27 double array), (2) new decrypt_key (1×27 double array), (3) transition probability matrix (27×27) and (4) encrypted text ($1 \times N$ character array). **metropolis.m** has two outputs. The first output is equal to 1 if the new decrypt_key is accepted and is equal to 0 otherwise. The second output is the probability that the new key is accepted based on the Metropolis criteria. To reiterate the criteria, let logn_pr_curr be the logarithm of $p(\text{encrypted message} \mid \text{current\_decrypt\_key})$ and logn_pr_new be the logarithm of the $p(\text{encrypted message} \mid \text{new\_decrypt\_key})$

- (i) If $\text{logn\_pr\_new} \geq \text{logn\_pr\_curr}$, **metropolis.m** should definitely accept the new key. In other words, both outputs of **metropolis.m** are equal to 1.
- (ii) If $\text{logn\_pr\_new} < \text{logn\_pr\_curr}$, then **metropolis.m** should accept the new key with probability $p = \exp(\text{logn\_pr\_new} - \text{logn\_pr\_curr})$. In other words, the second output of **metropolis.m** should be equal to p. The first output of **metropolis.m** depends on a "coin toss" inside **metropolis.m**. If $\text{rand}(1) < p$, then the first output is 1, otherwise the first output is 0.

IMPORTANT NOTE: To ensure everyone's code behaves the same way, (1) do NOT set the random number generator seed inside **metropolis.m** and (2) use the "$\text{rand}(1) < p$" check from the previous paragraph to decide whether to reject or accept. For example, "$(1-\text{rand}(1)) < p$" is technically also correct, but your code will behave differently from everyone else.

- (i) Apply **metropolis.m** using **frank_decrypt_key** as the current key, **mystery_decrypt_key** as the new key, **pr_trans** computed from Q3a and **frank_encrypted_txt**. Report the probability of accepting the new key as returned by **metropolis.m**.
- (ii) Create a new key from **frank_decrypt_key**, by swapping the 12-th and 13-th elements of the array. Apply **metropolis.m** using **frank_decrypt_key** as the current key, the newly generated key, **pr_trans** computed from Q3a and **frank_encrypted_txt**. Report the probability of accepting the new key as returned by **metropolis.m**. [Hint: Probability should be small but not zero.]

(b) **mcmc_decrypt_text.m** implements the algorithm in the Appendix. The function has already been written for you. The functions written in questions 1, 3 and 4a are utilized by **mcmc_decrypt_text.m**. You should study the code in **mcmc_decrypt_text.m** to understand the logic and flow of the algorithm. Be careful not to accidentally modify the code and break it!

The inputs of **mcmc_decrypt_text.m** are an $1 \times N$ character array of encrypted text and a 27×27 matrix of transition probability (**pr_trans**). The outputs are an $1 \times N$ character array of decrypted text (**decrypt_txt**) and an 1×27 double array (**decrypt_key**) used to decrypt the input array (see Q2b). The current iteration, the current decrypted text and the log probability of the current decrypted text is printed every 100 iterations until 2000 iterations, after which progress is printed every 1000 iterations. The algorithm terminates after 15000 iterations.

Apply **mcmc_decrypt_text.m** to **frank_encrypted_txt** using **pr_trans** computed from Q3a. Report the final decrypted text (**decrypt_txt**) and its log probability.

The true decrypt key (**frank_decrypt_key**) for **frank_encrypted_txt** is provided in **sem1_2019_encrypt.mat**. Verify that **frank_decrypt_key** differs from **decrypt_-key** estimated by **mcmc_decrypt_text.m**.

How are the keys different? How does these differences show up in the final decrypted text?

Explain why the algorithm does not give exactly the correct answer?

(c) **sem1_2019_encrypt.mat** contains **mystery_encrypted_txt** (another encrypted text). Apply **mcmc_decrypt_text.m** to **mystery_encrypted_txt** using **pr_trans** computed from Q3a. Report the final decrypted text (**decrypt_txt**) and its log probability.

The true decrypt key (**mystery_decrypt_key**) for **mystery_encrypted_txt** is provided in **sem1_2019_encrypt.mat**. Verify that **mystery_decrypt_key** differs from **decrypt_key** estimated by **mcmc_decrypt_text.m**.

How are the keys different? How does these differences show up in the final decrypted text?

Why did the algorithm not give exactly the correct answer?

Q5. Extra Credits (5%)

This question is optional and for extra credits. Given that there are still errors in the estimated decryption keys (Q4), suggest improvements and possibly implement them. This is an open ended question. There is no single right answer.

Appendix: Summary of Metropolis Decryption Algorithm

(i) Start with preliminary guess of decrypt_key.

(ii) Perturb the current decrypt_key by randomly swapping two elements in the array. Let's denote the perturbation of the current decrypt_key as new_decrypt_key.

(iii) Compute $\ln P_{current} = \ln p(\text{encrypted message} \mid \text{current\_decrypt\_key})$

(iv) Compute $\ln P_{new} = \ln p(\text{encrypted message} \mid \text{new\_decrypt\_key})$

- If $\ln P_{new} \geq \ln P_{current}$, accept the new decrypt_key as the current decrypt_key
- If $\ln P_{new} < \ln P_{current}$, accept the new decrypt_key as the current decrypt_key with probability $\exp(\ln P_{new} - \ln P_{current})$. Observe how we compute the differences in the ln probability before exponentiating because this is computationally more stable.

(v) Repeat steps (ii) to (iv) "forever" (15000 iterations in our case).

mcmc_decrypt_text.m implements the above algorithm, except for critical portions of steps (iii) and (iv), which you have to write in metropolis.m (Q4a).