Lab06

September 28, 2018

Probability for Data Science
UC Berkeley, Fall 2018
Ani Adhikari and Jim Pitman

```
In [1]: from datascience import *
        from prob140 import *
        import numpy as np
        from scipy import special
        import pickle
```

```
In [2]: # Run this code

        import itertools as it
        from collections import Counter

        def clean_string(string):
            """
            Cleans an input string by replacing all newlines with spaces, and then
            removing all letters not in *allowable_letters*
            """
            string = string.replace("\n"," ")
            return "".join([i for i in string.lower().strip() if i in allowable_letters])

        def load_bigrams(text):
            """
            Takes a string which has already been cleaned, and returns a dictionary
            of conditional bigram probabilities

            cond_bigram[(a,b)] = P(X_{n+1} = b | X_{n} = a)

            Uses Laplace smoothing with size $1$, to remove zero transition probabilities
            """
            bigram_counter = Counter(list(it.product(allowable_letters, repeat=2)))
            gram_counter = Counter(allowable_letters*len(allowable_letters))
            for l1, l2 in zip(text,text[1:]):
                bigram_counter[(l1, l2)] += 1
                gram_counter[l1] += 1
```

```
    cond_bigram = {k:v/gram_counter[k[0]] for k,v in bigram_counter.items()}
    return cond_bigram

def bigram_from_file(filename):
    """
    Given a filename, this reads it, cleans it, and returns the conditional bigram
    """
    file_text = open(filename).read()
    file_text = clean_string(file_text)
    return load_bigrams(file_text)

def reverse_cipher(cipher):
    return {v:k for k, v in cipher.items()}

def print_differences(cipher1, cipher2):
    for k in cipher1:
        if cipher1[k] != cipher2[k]:
            print("%s: %s  %s"%(k,cipher1[k], cipher2[k]))

def num_errors(cipher, encoded_text, original_text):
    decoded = np.array(list(decode_text(encoded_text, cipher)))
    original = np.array(list(original_text))
    num_errors = np.count_nonzero(decoded != original)
    return num_errors

def get_secret_text(student_id):
    with open('Lab06_data/secret_strings.p', 'rb') as f:
        texts = pickle.load(f)
    return texts[student_id % len(texts)]
```

1 Lab 6: Code Breaking by MCMC

Cryptography is the study of algorithms used to encode and decode messages. Markov Chain Monte Carlo (MCMC) methods have been successfully used to decode messages encrypted using substitution codes and also more complex encryption methods. In this lab you will apply MCMC and the Metropolis algorithm to decode English text that has been encrypted by a substitution code.

The lab is based on the paper The Markov Chain Monte Carlo Revolution by Persi Diaconis. It was presented at the 25th anniversary celebrations of MSRI up the hill, and appeared in the Bulletin of the American Mathematical Society in November 2008. The code is based on Simulation and Solving Substitution Codes written by Stephen Connor in 2003.

Recall that in class we worked with a tiny alphabet and a short message that had been encoded using a substitution code. We used a Markov chain model and a scoring system to choose the decoder that had the highest score. That is, we chose the decoder that had the highest likelihood given the message.

Because the alphabet was small, we were able to list all the decoders and their corresponding scores. When the alphabet is large, making an exhaustive list of all possible decoders and scores is not feasible. That's where the Monte Carlo part of the algorithm comes in. The idea is to choose decoders at random using an algorithm that favors good decoders over bad ones.

In this lab you will work with an alphabet consisting of all 26 English letters as well as a space character that will have a special status. Before you begin, please review Sections 11.3 (Code Breaking) and 11.4 (Markov Chain Monte Carlo) of the textbook. The lab follows those sections closely.

Unlike all the other labs, this lab is not divided into parts. It's just one small project. You will learn how to: - Apply a decoder to encoded text - Use transformations for improved numerical accuracy - Implement the Metropolis algorithm - Decode a message encrypted by a substitution code

The programming needed for this lab is more complex than what can reasonably be expected based on just Data 8 background, so much of it has been done for you. Those of you who have more extensive Python knowledge might be interested in looking at the details.

Please start by running all the cells above this one.

1.1 Alphabet and Bigrams

The text that you are going to decode was written in English. For data on the frequencies of all the different bigrams in English, we will start by counting all the bigrams in *War and Peace* by Tolstoy. That is one of the longest novels in English (actually in Russian, but we are using an English translation from Project Gutenberg) and is often used as a "corpus" or body of language on which to base analyses of other text.

To keep the calculations manageable, we will restrict ourselves to a 27-character alphabet, consisting of the 26 lower case letters and a space. The cell below places all 27 in a list.

```
In [3]: allowable_letters = list("abcdefghijklmnopqrstuvwxyz ")
```

In the cell below, we find the relative frequencies of all bigrams in War and Peace and place them in the dictionary `wp_bigrams`. You don't need to know what a dictionary is. It's fine to imagine it containing the same information as a Table that has a column with the bigrams and another column with the relative frequencies.

```
In [4]: wp_bigrams = bigram_from_file("Lab06_data/warandpeace.txt")
```

```
In [5]: #wp_bigrams
```

1.1.1 1. Transition Matrix

As in class, the model is that the sequence of characters in the text is a Markov chain. The state space of the chain is the alphabet of 27 characters. Of course the text has other punctuation, but we are stripping all of it. We are also replacing upper case letters by lower case.

Construct the transition matrix for this Markov chain based on the relative frequencies in `wp_bigrams` defined above.

We have started the code off for you by defining the transition function `wp_transition`. Use `allowable_letters` in your definition.

```
In [6]: def wp_transition(a, b):
            if (a, b) in wp_bigrams:
                return wp_bigrams[(a, b)]
```

```
    return 0

wp_bigram_mc =MarkovChain.from_transition_function(allowable_letters, wp_transition)
wp_bigram_mc
```

```
Out[6]:        a         b         c         d         e         f         g  \
a  0.000123  0.016969  0.034439  0.055522  0.000809  0.008217  0.016857
b  0.092429  0.006631  0.000144  0.000490  0.327913  0.000029  0.000029
c  0.109112  0.000016  0.017047  0.000162  0.213245  0.000016  0.000016
d  0.021188  0.000659  0.000110  0.011046  0.116413  0.000963  0.003702
e  0.042608  0.000884  0.017399  0.091233  0.025779  0.010148  0.006876
f  0.070639  0.000564  0.000692  0.000419  0.083948  0.051705  0.000036
g  0.066342  0.000058  0.000117  0.001013  0.114711  0.000234  0.008724
h  0.164932  0.000412  0.000388  0.000382  0.448388  0.000424  0.000024
i  0.016801  0.007746  0.049227  0.050081  0.048754  0.021611  0.024772
j  0.038432  0.000384  0.000384  0.000384  0.217525  0.000384  0.000384
k  0.027715  0.000635  0.002493  0.000196  0.277055  0.000880  0.000831
l  0.079685  0.000860  0.001119  0.071669  0.175475  0.023976  0.000963
m  0.151252  0.017642  0.000616  0.000081  0.245638  0.002254  0.000032
n  0.030173  0.000858  0.050129  0.187753  0.080584  0.004978  0.135691
o  0.007304  0.004705  0.007650  0.016043  0.002656  0.088668  0.004055
p  0.100020  0.000285  0.002809  0.000066  0.180110  0.002217  0.000241
q  0.000424  0.000424  0.000424  0.000424  0.000424  0.000424  0.000424
r  0.055996  0.001495  0.009956  0.029470  0.240263  0.003058  0.010030
s  0.051883  0.002719  0.013724  0.000546  0.107847  0.001921  0.000307
t  0.034031  0.000150  0.002893  0.000022  0.084535  0.000839  0.000066
u  0.020441  0.014008  0.031733  0.022107  0.029450  0.005122  0.050631
v  0.066239  0.000074  0.000037  0.000406  0.535332  0.000037  0.000148
w  0.205470  0.000203  0.000405  0.006229  0.130987  0.000844  0.000135
x  0.076627  0.000227  0.136930  0.000227  0.071639  0.000453  0.000227
y  0.028471  0.003413  0.003737  0.000410  0.056121  0.003305  0.000281
z  0.057167  0.002071  0.000829  0.011599  0.304060  0.000414  0.000414
   0.119834  0.043355  0.036531  0.029719  0.020259  0.036824  0.016908

          h         i         j    ...           r         s         t  \
a  0.001598  0.042852  0.000829    ...    0.087455  0.097825  0.138167
b  0.000086  0.029493  0.005074    ...    0.057228  0.013925  0.007496
c  0.188266  0.041799  0.000016    ...    0.033608  0.001622  0.067070
d  0.000676  0.066075  0.002003    ...    0.030282  0.020191  0.000625
e  0.002347  0.011143  0.000273    ...    0.140364  0.065150  0.023902
f  0.000182  0.085168  0.000036    ...    0.103246  0.004060  0.039234
g  0.116172  0.048798  0.000019    ...    0.053451  0.017486  0.004595
h  0.000084  0.155101  0.000006    ...    0.007752  0.001135  0.025150
i  0.000040  0.002659  0.000006    ...    0.034191  0.125124  0.122580
j  0.000384  0.004612  0.000384    ...    0.000384  0.000384  0.000384
k  0.040375  0.166390  0.000049    ...    0.004839  0.037785  0.000440
l  0.000166  0.098503  0.000010    ...    0.004878  0.016964  0.017234
m  0.000292  0.080117  0.000016    ...    0.003211  0.031359  0.003421
```

```
n  0.000847  0.033772  0.000657    ...     0.000429  0.036715  0.093331
o  0.001918  0.011763  0.000639    ...     0.103606  0.035482  0.052531
p  0.009196  0.092294  0.000022    ...     0.172428  0.023770  0.053686
q  0.000424  0.000424  0.000424    ...     0.000424  0.000424  0.000424
r  0.001536  0.089210  0.000027    ...     0.033242  0.060522  0.031234
s  0.066675  0.055946  0.000031    ...     0.000184  0.059598  0.109596
t  0.325798  0.061567  0.000018    ...     0.025040  0.026859  0.019590
u  0.000740  0.027151  0.000031    ...     0.125189  0.137731  0.169911
v  0.000074  0.176772  0.000037    ...     0.006712  0.022719  0.002877
w  0.201351  0.169258  0.000017    ...     0.008947  0.014603  0.000405
x  0.013829  0.116527  0.000227    ...     0.000227  0.001587  0.107005
y  0.000648  0.024453  0.000043    ...     0.002463  0.033893  0.018361
z  0.064209  0.101906  0.000414    ...     0.000414  0.001243  0.000829
   0.085827  0.053078  0.002858    ...     0.026106  0.070295  0.150823

          u         v         w         x         y         z
a  0.012351  0.021411  0.010605  0.000255  0.025883  0.001893  0.074040
b  0.152223  0.001355  0.000605  0.000029  0.073546  0.000029  0.005824
c  0.027444  0.000016  0.000162  0.000016  0.005044  0.000097  0.009878
d  0.009432  0.002721  0.000549  0.000008  0.010066  0.000025  0.641109
e  0.001018  0.017494  0.010275  0.009970  0.012008  0.000604  0.353170
f  0.033918  0.000018  0.000601  0.000018  0.001566  0.000018  0.353724
g  0.026112  0.000019  0.000526  0.000019  0.001908  0.000136  0.431058
h  0.009562  0.000036  0.000257  0.000006  0.006134  0.000006  0.092114
i  0.000756  0.020112  0.000046  0.001874  0.000012  0.003426  0.031365
j  0.455419  0.000384  0.000384  0.000384  0.000384  0.000384  0.000384
k  0.033092  0.000684  0.004399  0.000049  0.006648  0.000049  0.247287
l  0.015773  0.004578  0.004961  0.000010  0.103671  0.000269  0.133903
m  0.027127  0.000114  0.000259  0.000016  0.039856  0.000016  0.202474
n  0.005266  0.004093  0.000727  0.000451  0.011362  0.000125  0.220043
o  0.123395  0.030939  0.052007  0.000744  0.003762  0.000629  0.140942
p  0.025636  0.003995  0.000527  0.000022  0.007287  0.000022  0.069138
q  0.988545  0.000424  0.000424  0.000424  0.000424  0.000424  0.000849
r  0.020437  0.004756  0.002054  0.000007  0.038738  0.000559  0.212013
s  0.024281  0.000491  0.004389  0.000006  0.002320  0.000043  0.385687
t  0.013664  0.004346  0.005251  0.000009  0.013898  0.002946  0.257271
u  0.000046  0.001913  0.000046  0.000679  0.000339  0.001296  0.059255
v  0.000848  0.000111  0.000295  0.000037  0.004057  0.000037  0.098473
w  0.000523  0.000017  0.000456  0.000017  0.000709  0.000017  0.139512
x  0.004081  0.035593  0.000227  0.036046  0.001587  0.000227  0.084335
y  0.001318  0.000562  0.002139  0.000022  0.000238  0.000238  0.678180
z  0.012842  0.001657  0.000829  0.000414  0.006214  0.019884  0.054681
   0.009706  0.006878  0.070007  0.000777  0.011618  0.000255  0.025147

[27 rows x 27 columns]
```

1.2 Decoders

As you know, when a substitution code has been used for encryption, a decoder is a permutation of the alphabet. We apply this permutation to our encoded text in order to generate the decoded text.

We will be representing the decoder as a Python dictionary. Here is an example of how a dictionary works, using a three-letter alphabet 'a', 'b', 'c'.

If the decoder is ['b', 'c', 'a'] then the substitutions for decoding the encrypted message are

$$a \to b \qquad b \to c \qquad c \to a$$

This decoder written as a dictionary dictionary looks like this:

```
simple_decoder = {'a':'b','b':'c','c':a'}
```

To access a value in the decoder, use the method `.get`. For example, to access the translation of a, use

```
simple_decoder.get('a')
```

```
In [7]: simple_decoder = {'a':'b', 'b':'c', 'c':'a'}
        simple_decoder.get('a')
```

```
Out[7]: 'b'
```

1.2.1 Keeping Spaces Intact

To simplify calculations, we are assuming that our substitution code keeps spaces unchanged. Letters can precede and follow spaces, but spaces will be fixed points in the encoding permutation.

So our decoder must keep spaces unchanged as well. The following cell defines a function called `random_decoder` which will randomly generate a decoder. It will operate on the 26 letters, omitting the space.

Run the cell below. The function `random_decoder` constructs a random permutation of the alphabet and places it in a dictionary to make it a decoder.

```
In [8]: decoder_letters = np.array(list("abcdefghijklmnopqrstuvwxyz"))
        # We don't operate on spaces

        identity_decoder = \
            {letter:letter for letter in decoder_letters}

        # Random starting decoder
        def random_decoder():
            """ Random decoder """
            new_letters = decoder_letters.copy()
            np.random.shuffle(new_letters)
            return {orig:new for orig ,new in zip(decoder_letters, new_letters)}
```

```
In [ ]:
```

2 newpage

2.0.1 2. Applying a Decoder

Define a function `decode_text` that takes as its arguments a string to be decoded and the decoder. It should return the decoded string.

Remember that we are only decoding alphabetical characters. If a character is not in the decoder (e.g. a space), leave the character alone.

```
In [9]: def decode_text(string, decoder):
            new_string = ""

            for char in string:
                if char in decoder:
                    new_letter = decoder.get(char)
                else:
                    new_letter =char

                # Now we append the letter to the back of the new string
                new_string = new_string + new_letter

            return new_string
```

```
In [10]: decode_text('afdafda',random_decoder())
```

```
Out[10]: 'rcsrcsr'
```

2.1 Metropolis Algorithm: Proposal Chain

The Metropolis algorithm we will use is essentially the same as the one defined in class and proved to converge to the correct distribution. There is one tweak to increase computatational accuracy, and another to deal with spaces. You shouldn't worry about either of them.

Please have Section 11.4 of the textbook at hand as you go through the exercises below.

- The state space S of the proposal chain is the set of all possible decoders.
- Define a matrix Q as follows: given that the chain is at decoder i, pick another decoder by randomly swapping two elements of i. For any two decoders i and j, define

$$Q(i,j) = \begin{cases} \frac{1}{\binom{26}{2}} & \text{if } i \text{ and } j \text{ differ in exactly two places} \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

You will show below that Q satisfies the conditions to be the transition matrix of a proposal chain.

3 newpage

3.0.1 3a) Properties of Q

The state space of the proposal chain is large. How many elements does it contain? For reference, `special.comb(n, k)` evaluates to $\binom{n}{k}$ and `np.math.factorial(n)` evaluates to $n!$. The modules were imported at the start of the lab.

```
In [11]: np.math.factorial(26)

Out[11]: 403291461126605635584000000
```

Define the three permutations below:

- decoder_1: alphbetical order, that is, `abcdef ... xyz`
- decoder_2: `bacdef ... xyz`, that is, `ba` followed by the other letters in alphabetical order
- decoder_3: `badcef ... xyz`, that is, `badc` followed by the other letters in alphabetical order

For each of (i)-(iv) below, type 1/(26_choose_2) if you think the value is $1/\binom{26}{2}$ and 0 otherwise, and enter the reason for your choice.

(i) Q(decoder_1, decoder_2) = $1/\binom{26}{2}$ because decoder_1 and decoder_2's difference is the change of 'ab' to 'ba' which is in exactly two places

(ii) Q(decoder_2, decoder_1) = $1/\binom{26}{2}$ because the two states are in exactly two places

(iii) Q(decoder_1, decoder_3) = 0 because the two states are in four places , not 2 ,thus the answer is 0

(iv) Q(decoder_2, decoder_3) = $1/\binom{26}{2}$ because the two states are in exactly two places

Symmetric Transition Matrix: Explain why Q is a symmetric transition matrix.

[Note: You have to establish that Q is symmetric and that it is a transition matrix. For the latter, it is a good idea to focus on the row corresponding to decoder_1 above. What criteria do the entries of each row have to satisfy? Why are the criteria satisfied?]

Your answer here. As for matrix Q, every row has figure apart from 0 ,and adds up to 1,therefore ,each state can reach to other staes,therefore,its a transition matrix .If i,j are in exactly two states, Q(i,j)=Q(j,i)=$1/\binom{26}{2}$, when they are not in two states , Q(i,j)=Q(j,i)=0, therefore, Q(i,j)=Q(j,i), Q=Q', therefore, Q is a symmetric transition matrix.

Irreducible: It is a fact that Q is the transition matrix of an irreducible chain, but formally establishing that takes a bit of writing. To avoid complicated writing, let's reduce the problem to an alphabet of just three letters `a`, `b`, and `c` and transition behavior defined analogously on the six decoders, with positive transition probabilities each equal 1/(3_choose_2) = 1/3 if the two permutations differ in exactly two places. Why does this create an irreducible chain?

Your answer here. As for each reach transition equals to 1/3 for every state you are in , it can reach to other states,thus it creates an irreducible chain.

For any size of alphabet, there is a famous algorithm for creating all possible permutations by successively swapping two *adjacent* elements. It was known to bell-ringers in 17th-century England. Adjacent swaps are among the transitions allowed by the matrix Q, so the chain is irreducible.

Proposal Chain: Now explain why Q satisfies all the criteria for the transition matrix of a proposal chain. You have done all the work already; carefully read the Metropolis Algorithm section of Section 11.4 to see what is needed.

Your answer here.

3.0.2 3b) Making a Proposal to Move

Define the function `generate_proposed_decoder` which takes an existing decoder and returns a new decoder as follows: it randomly picks two different letters of the alphabet and swaps the decoding for those letters. Make sure to use `decoder_letters` rather than `allowable_letters` because we want to keep the spaces as is.

For example, if we have the decoder `{'a':'b','b':'c','c':a'}` and decide to swap `a` and `b`, our new decoder should be `{'a':'c','b':'b','c':a'}`.

There are several different ways you can use Python to draw random samples. In this instance we recommend `np.random.choice`. It takes two arguments: - a list from which to sample uniformly at random with replacement - the number of times to sample

The optional argument `replace=False` leads to simple random sampling.

It returns an array consisting of the sampled elements.

```
In [12]: def generate_proposed_decoder(decoder):
             new_decoder = decoder.copy()

             letters = np.random.choice(
                 list(new_decoder.keys()), 2, replace=False)

             letter1 =letters[0]
             letter2 =letters[1]

             new_value_of_letter1 =new_decoder[letter2]
             new_value_of_letter2 =new_decoder[letter1]

             # This code replaces the value of letter1 and letter2
             # with new_value_of_letter1 and new_value_of_letter2
             new_decoder[letter1] = new_value_of_letter1
             new_decoder[letter2] = new_value_of_letter2
             #return new_decoder[letter1]
             return new_decoder
```

```
In [13]: #p=list({'a':'b','b':'c','c':'a'}.keys())
         #type(p)
         l={'a':'b','b':'c','c':'a'}
         generate_proposed_decoder(l)

Out[13]: {'a': 'c', 'b': 'b', 'c': 'a'}
```

```
In [14]: decoder={'a':'b','b':'c','c':'a'}
         t=np.random.choice(list(decoder.keys()),2,replace=False)
         t

Out[14]: array(['b', 'a'], dtype='<U1')
```

3.1 The Target Distribution π

For letters x_1 and x_2, let $B(x_1, x_2)$ be the (x_1, x_2)th entry in the bigram transition matrix. Suppose the decoded text consists of the characters $x_0, x_1, \ldots, x_n$. Recall from class that we defined the *score* of the corresponding decoder i to be

$$s(i) = \prod_{k=0}^{n-1} B_i(x_k, x_{k+1})$$

Here's a tweak to what was done in class. When the string is long, the score of any decoder is going to be very small. This causes problems with numerical accuracy. So we will work with the log-score instead, that is, with the log of the score.

$$\log(s(i)) = \sum_{k=0}^{n-1} \log(B_i(x_k, x_{k+1}))$$

4 newpage

4.0.1 4. Log Probability of Path

Define a function `log_prob_path` that takes two arguments: - a string `message` that is a sequence of letters - the bigram transition matrix of a `MarkovChain` `mc`

and returns the log of the probability of the path `message` taken by `mc`, given that the first character of `message` is the initial state of the path.

Though `mc.prob_of_path` returns the probability of the path, don't use `np.log(mc.prob_of_path)`. Each path has very small probability, so the numbers will round too soon and your computations might be inaccurate.

Instead, you can use the function `mc.log_prob_of_path`. This takes as its first argument the starting character of the string, and as its next argument a list or array containing the characters in the rest of the string. It returns the log of the probability of the path, conditional on the starting character.

Some string methods will be helpful:

- string[0] returns the first letter of a string. For example:

```
>>> x = 'HELLO'
>>> x[0]
'H'
```

- string[1:] returns everything except the first letter of a string. For example:

```
>>> x[1:]
'ELLO'
```

- list(string) splits the string into a list of its characters. This was used above to generate `allowable_letters`.

```
In [15]: def log_prob_path(string, bigram_mc):
             start = string[0]
```

```
        path = list(string[1:])

        return bigram_mc.log_prob_of_path(start, path)
```

5 newpage

5.0.1 5. Log Score of Decoder

Define a function `log_score` that takes the following arguments:

- an encrypted string
- a decoder
- a bigram transition matrix

The function should decode the encrypted string using the decoder, and return the log score of the decoder.

Use the functions `decode_text` and `log_prob_path` that you wrote earlier.

```
In [16]: def log_score(string, decoder, bigrams):

             t=decode_text(string,decoder)
             return log_prob_path(t,bigrams)
```

5.0.2 Acceptance Ratios

We would like Decoder i to appear among our randomly selected decoders with target probability $\pi(i)$, where $\pi(i)$ is the likelihood of i given the data. As we saw in class, the likelihood $\pi(i)$ is proportional to $s(i)$ but we don't know the constant of proportionality. However, for any two decoders i and j, we can calculate the ratio

$$r(i,j) = \frac{\pi(j)}{\pi(i)} = \frac{s(j)}{s(i)}$$

The Metropolis algorithm creates a new chain that has π as its limit distribution. Recall that if the new chain is at i, then the the decision about whether or not to accept a transition to a new state j depends on the ratio $r(i,j)$.

Because we are working with log scores and not the scores themselves, every statement that we made in class about scores has to be rewritten in terms of log scores. For example, the ratio can be written as

$$\frac{s(j)}{s(i)} = e^{\log(\frac{s(j)}{s(i)})} = e^{\log(s(j))-\log(s(i))}$$

You will have to calculate the ratios, so keep in mind that `np.e**(x)` evaluates to e^x.

Write the condition $r(i,j) < 1$ in terms of $\log(s(i))$ and $\log(s(j)$. Justify your answer.

Your answer here.

$$e^{\log((j))-\log((i))+\log(p(j,i))}$$

5.1 Implementing the Metropolis Algorithm

6 newpage

6.0.1 6. An Encoded Message

Enter your student id in the cell below to access your secret encoded message. By the end of the lab, you should be able to decode your string.

```
In [17]: student_id =3034431768
         secret_text = get_secret_text(student_id)
         secret_text
```

```
Out[17]: 'yrrjots syjx rf zhc feh xyn jx byrvhb nft svfr yr rje oyes je rvh pyod nybx ptr j de
```

7 newpage

7.0.1 7. Implementation

Define the function `metropolis` which will take as its arguments:

- the encrypted text, as a string
- the transition matrix of bigrams
- the number of repetitions for running the Metropolis algorithm

The function should return the decoder that it arrives at after performing the Metropolis algorithm the specified number of times.

If the number of repetitions is large, you know from class that the distribution of this random decoder is close to the desired distribution π which is the likelihood distribution of the decoder given the data. Therefore you expect to end up with a decoder that has a high likelihood compared to the other decoders.

Between iterations, you should keep track of the following variables:

1. `best_decoder`: the decoder that has the highest log score
2. `best_score`: the log score associated with the best_decoder
3. `decoder`: the decoder that you are currently working with
4. `last_score`: the log score of the current decoder

In each iteration, you should do the following:

1. Generate a new decoder based on the current decoder; the "new" decoder might be the same as the current one.
2. Calculate the log score of your new decoder.
3. **Important:** Follow the Metropolis algorithm to decide whether or not to move to a new decoder.
4. If the new decoder's log score is the best seen so far, update `best_score` and `best_decoder`

```
In [34]: import time


         def p_coin(p):
```

```
    """
    Flips a coin that comes up heads with probability p
    and returns 1 if Heads, 0 if Tails
    """
    return np.random.random() < p


def metropolis(string_to_decode, bigrams, reps):
    decoder = random_decoder() # Starting decoder

    best_decoder = decoder
    best_score = log_score(
        string_to_decode,
        best_decoder,
        bigrams
    )

    last_score = best_score

    for rep in np.arange(reps):

        # This will print out our progress
        if rep*10%reps == 0: # Repeat every 10%
            time.sleep(.01)
            decoded_text = decode_text(
                string_to_decode,
                best_decoder
            )[:40]
            print('Score: %.00f \t Guess: %s'%(best_score, decoded_text))

        #########################
        # Your code starts here #
        #########################
        proposed_decoder = generate_proposed_decoder(best_decoder)
        log_s_orig =log_score(
        string_to_decode,
        best_decoder,
        bigrams)
        log_s_new =log_score(
        string_to_decode,
       proposed_decoder,
        bigrams)

        # If better than before or p-coin flip works
        if log_s_new > log_s_orig or p_coin(log_s_orig/log_s_new):

            decoder=proposed_decoder
            last_score=log_s_new
```

```
                if log_s_new > best_score:
                    best_decoder=proposed_decoder
                    best_score=log_s_new
        return best_decoder
```

7.1 Running It

The text you are going to first encrypt and then decode is taken from Winston Churchill's speech to the House of Commons on 4 June 1940. Naturally, it contains upper case letters and punctuation that we have not included in our alphabet. In the cell below, the function `clean_string` strips out all the punctuation and replaces upper case letters by the corresponding lower case letters.

```
In [35]: ogtext = """
         I have, myself, full confidence that if all do their duty, if nothing is neglected,
         and if the best arrangements are made, as they are being made, we shall prove ourselve
         once again able to defend our Island home, to ride out the storm of war, and to outli
         the menace of tyranny, if necessary for years, if necessary alone. At any rate, that i
         what we are going to try to do. That is the resolve of His Majestys Government-every m
         of them. That is the will of Parliament and the nation.
         """
         original_string = clean_string(ogtext)
         encoding_cipher = random_decoder()
         string_to_decode = decode_text(original_string,encoding_cipher)
         string_to_decode

Out[35]: 'w hysr zntrbm mqbb lodmwgrdlr xhyx wm ybb go xhrwu gqxn wm doxhwdi wt dribrlxrg  ydg
```

7.1.1 8a) Finding the Decoder

Run the following cell to generate your decoder. You may have to run the cell a couple of times to get the correct decoder.

```
In [36]: new_decoder = metropolis(string_to_decode, wp_bigram_mc, 10000)

Score: -2544          Guess: r bdkc zwhcap puaa qilprtclqc xbdx rp da
Score: -1155          Guess: i have wuselm mfll conmidence that im al
Score: -1104          Guess: i have pyself full confidence that if al
Score: -1088          Guess: i have myself full confidence that if al
Score: -1088          Guess: i have myself full confidence that if al
Score: -1088          Guess: i have myself full confidence that if al
Score: -1088          Guess: i have myself full confidence that if al
Score: -1088          Guess: i have myself full confidence that if al
Score: -1088          Guess: i have myself full confidence that if al
Score: -1088          Guess: i have myself full confidence that if al
```

Run the cell below to decode the entire string.

```
In [37]: decode_text(string_to_decode, new_decoder)
```

```
Out[37]: 'i have myself full confidence that if all do their duty if nothing is neglected  and
```

7.1.2 8b) Decoding Your Secret Message

`secret_text` contains your customized, secret message. Use MCMC to decode it.

```
In [30]: secret_decoder = metropolis("adawd",wp_bigram_mc,100)
         decode_text(secret_text, secret_decoder)
```

```
Score: -23          Guess: nfnaf
Score: -17          Guess: nonao
Score: -17          Guess: nonao
Score: -17          Guess: nonao
Score: -17          Guess: nonao
Score: -11          Guess: nonyo
Score: -11          Guess: nonyo
Score: -11          Guess: nonyo
Score: -9          Guess: nonto
Score: -9          Guess: nonto
```

```
Out[30]: 'fzzkgrp pfkc zs xwv sbw cfh kc mfziwm hsr pisz fz zkb gfbp kb ziw lfgo hfmc lrz k obs
```

7.2 Conclusion

What you have learned: - How to implement the Metropolis algorithm to decode text encrypted by a substitution code, with simplifying assumptions about spaces, capitalization, and punctuation - MCMC works - Abstract concepts such as balance and convergence of n-step transition matrices are remarkable in their own right and also have powerful applications

```
In [38]: import gsExport
         gsExport.generateSubmission("Lab06.ipynb")
```

```
Processing Lab06.ipynb
Found a cell that has a little too much written in it; try to bring it down
Here's a preview of that cell:


 import time

def p_coin(p):
    """
    Flips a coin that comes up heads with probability p
    and returns 1 if Heads, 0 if Tails
    """
    return np.random.random() < p
```

```
def metropolis(string_to_
Generated notebook and autograded
Attempting to compile LaTeX
Finished generating PDF
```

```
<IPython.core.display.HTML object>
```

```
In [ ]:
```

```
In [ ]:
```