Notebook

February 22, 2019

Local date & time is : 02/22/2019 22:59:20 PST

```
In [5]: 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[5]:          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
        ... Omitting 81 lines ...
           0.009706  0.006878  0.070007  0.000777  0.011618  0.000255  0.025147

        [27 rows x 27 columns]
```

1 newpage

1.0.1 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 [8]: 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
```

2 newpage

2.0.1 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 [16]: np.math.factorial(26)
```

```
Out[16]: 403291461126605635584000000
```

(i) Q(decoder_1, decoder_2) = 1/(26_choose_2)

because only ab chanages, so only 1st and 2nd place differ

(ii) Q(decoder_2, decoder_1) = 1/(26_choose_2)

because same as (i), only 1st and 2nd place differ

(iii) Q(decoder_1, decoder_3) = 0

because 1,2,3,4th place differ. hence its zero

(iv) Q(decoder_2, decoder_3) = 1/(26_choose_2)

because only 3,4th place differ.

If Q(decoder 1, decoder 2) = 1 / 26_choose_2, that they differ by two letters, hence Q(decoder 2, decoder 1) also equals 1 / 26_choose_2. Same logic applies for Q value equal to zero. For each row, for a fixed decoder i, there will be 26 choose 2 number of decoders that Q value would be 1 / 26_choose_2. Hence, adding all the values in the row, it would sum up to 1. Hence the matrix is symmetric and transition Matrics.

In the 6 states of three letter permutation, there is no state or a group of state is isolated from other states. Hence it is evident that three letter permutation transition chain is irreducible.

We require Q to be a symmetric, irreducible transition matrix, which our current Q all satifies as evident from above.

```
In [9]: def generate_proposed_decoder(decoder):
            new_decoder = decoder.copy()

            letters = np.random.choice(
                decoder_letters, 2, replace=False)

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

            new_value_of_letter1 = decoder[letter2]
            new_value_of_letter2 = 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
```

3 newpage

3.0.1 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 [25]: def log_prob_path(string, bigram_mc):
             start = string[0]

             path = list(string[1:])

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

4 newpage

4.0.1 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 [14]: a = ''
         a + 'b'
```

```
Out[14]: 'b'
```

```
In [15]: def log_score(string, decoder, bigrams):
             decodstr = decode_text(string, decoder)
             return log_prob_path(decodstr, bigrams)
```

```
In [19]: np.log(.5)
```

```
Out[19]: -0.6931471805599453
```

for log(x) where x < 1, log(x) < 0. Hence log(s(j) / (s(i)) < 0 for r(i,j) < 1. Hence this is equivalent to (log(s(j)) - log(s(i))) < 0

5 newpage

5.0.1 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 [20]: student_id = 24605319
         secret_text = get_secret_text(student_id)
         secret_text
```

```
Out[20]: 'svegrd ysgbjn uso ajgv jv kjigwrrvwp livr rmhpwrrv kmkwd omt mv giooms pr swwrvere frwrgoaigh
```

6 newpage

6.0.1 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 distributionπwhich 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 = last_score
                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(np.e**(log_s_new - log_s_orig)):

                    last_score = log_s_new
                    best_decoder = proposed_decoder

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

```
In [36]: decode_text(string_to_decode, new_decoder)

Out[36]: 'i have myself full confidence that if all do their duty if nothing is neglected  and if the b

In [38]: secret_decoder = metropolis(secret_text, wp_bigram_mc, 10000)
         decode_text(secret_text, secret_decoder)

Score: -6335          Guess: hfezvt qhzogx mhs dgzf gf ugaznvvfnw paf
Score: -3439          Guess: ardnoy wanvif bas uinr ir cilntoorth jlr
Score: -2996          Guess: andrey parkof was born on mourteenth jun
... Omitting 4 lines ...
Score: -2947          Guess: andrey markov was born on fourteenth jun
Score: -2947          Guess: andrey markov was born on fourteenth jun


Out[38]: 'andrey markov was born on fourteenth june eighteen fifty six in russia he attended petersburg
```