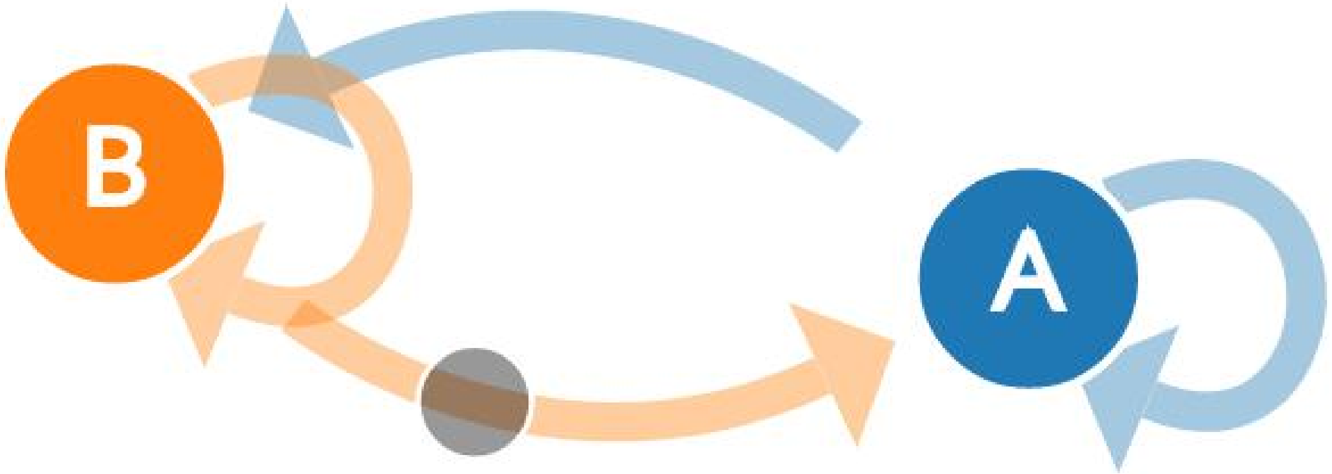


Solving Substitution codes by MCMC

TianShu Fu 2016210898

Substitution codes

- substitution cipher is a method of encrypting by which units of plaintext are replaced with ciphertext, according to a fixed system

-------------- From Wikipedia

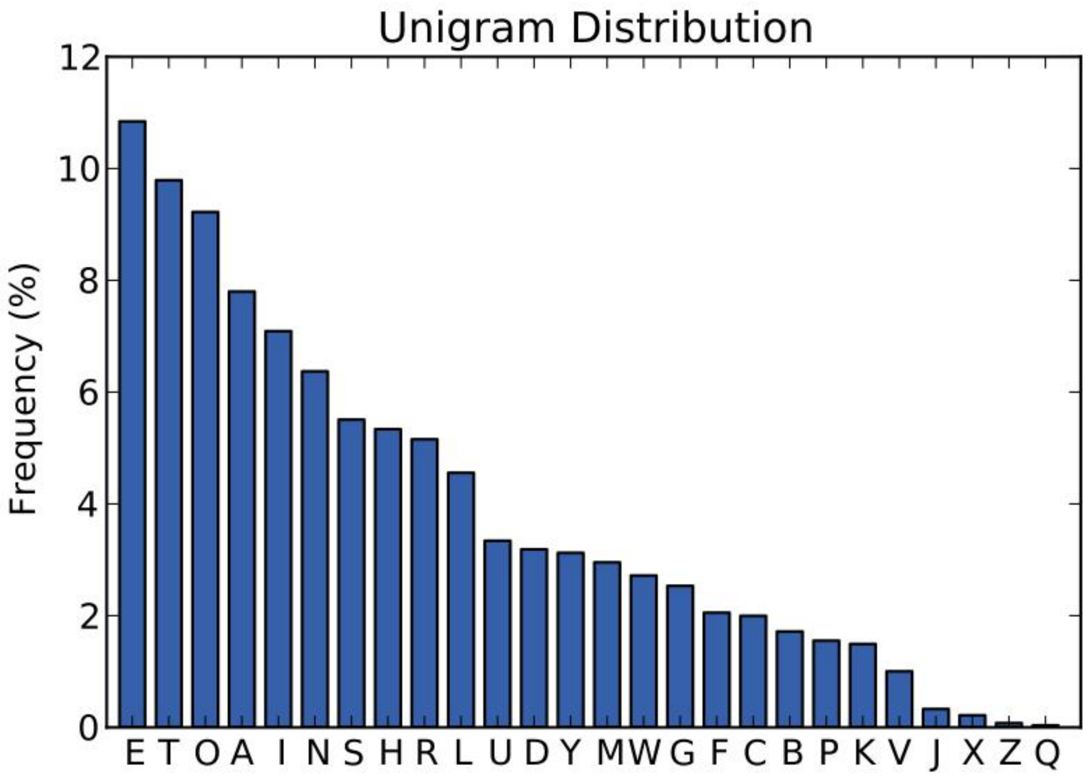
A	B	C	D	E	F	G	H	I	J	K	L	M
D	O	H	J	Q	C	Y	M	W	F	P	V	A

N	O	P	Q	R	S	T	U	V	W	X	Y	Z
T	K	Z	I	E	X	U	G	R	N	B	S	L

26!

$$4.03 * 10^{26}$$

Ways to solve given from class



Frequency analysis

1. Have to be done manually
2. Not accurate enough
3. Lots of restirctions

Introduce a new way !

- Markov Chain Monte Carlo (MCMC) methods
- Metropolis algorithm

Basic concept :Find the decoding combination which is most closed to natural language

First lets see an simple example

- Now suppose we get the coded message : atdt

and we believe this is an enlgish word

how can we find the original pattern?

1.First we list all the possible conditions(3!)

atdt Decoded
['a' 't' 'd' 't']
['a' 'd' 't' 'd']
['d' 't' 'a' 't']
['d' 'a' 't' 'a']
['t' 'd' 'a' 'd']
['t' 'a' 'd' 'a']

2.Then we list the transition probability among those three letters and calculate the score of each conditions

	a	d	t
a	0.018099	0.219458	0.762443
d	0.570995	0.159772	0.269233
t	0.653477	0.049867	0.296656

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

The score of each decoder!

Decoder	atdt Decoded	Score of Decoder
['d' 't' 'a']	['d' 'a' 't' 'a']	0.284492
['d' 'a' 't']	['d' 't' 'a' 't']	0.134142
['t' 'd' 'a']	['t' 'a' 'd' 'a']	0.0818868
['a' 'd' 't']	['a' 't' 'd' 't']	0.0102363
['t' 'a' 'd']	['t' 'd' 'a' 'd']	0.00624874
['a' 't' 'd']	['a' 'd' 't' 'd']	0.00294638

Here comes a problem

- The 52 lower case and upper case letters, along with a space character and all the punctuations, form an alphabet of around 70 characters

which contains 70! conditions !!!

Thats how much it is :

11978571669969890269925854460558840225267029209529303278944419871214396524861374498691473966836482048

We need an algorithm!

The data shows the transition probabitlity amoung all the letters

	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

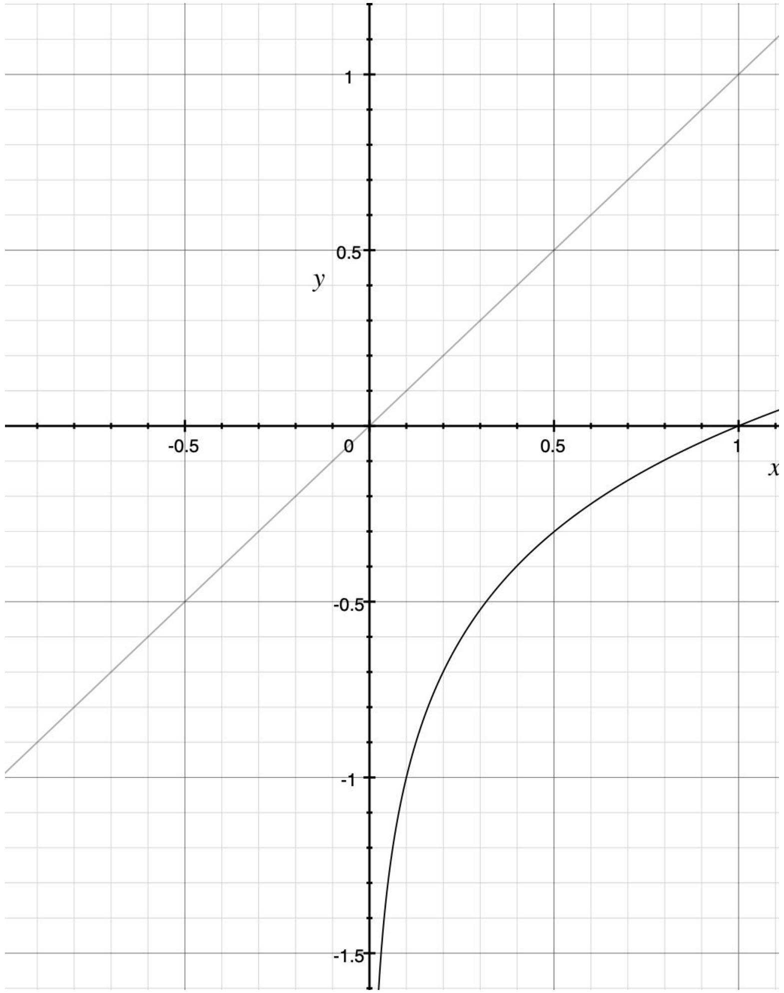
We need a new score system

Too small!

['t' 'd' 'a' 'd']	0.00624874
['a' 'd' 't' 'd']	0.00294638

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

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



First lets define a random decoder generation function

```
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)}
```

Creates the shuffle

```
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
```

Creates the decoded text

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

Out[10]: 'rcsrcsr'
```

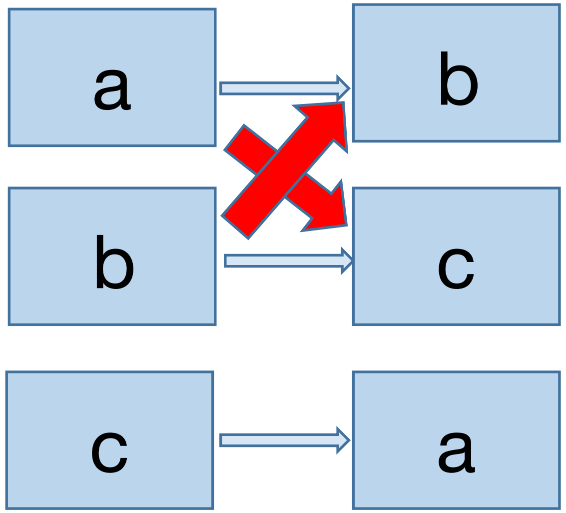
Metropolis Algorithm: Proposal Chain

- 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

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

Making a Proposal to Move

randomly picks two different letters of the alphabet and swaps the decoding for those letters




```
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'}
```

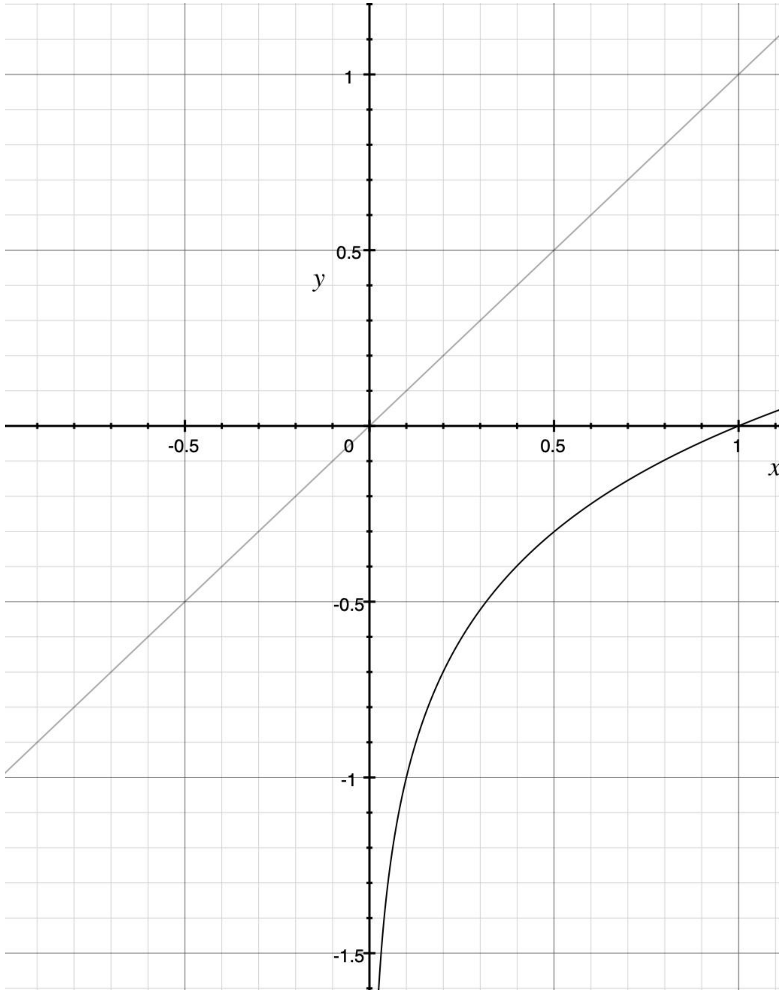
We need a new score system

Too small!

['t' 'd' 'a' 'd']	0.00624874
['a' 'd' 't' 'd']	0.00294638

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

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



Acceptance Ratios

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

As we had used the log function during last page

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

When r(i,j)<1:

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

Main function of MCMC

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`

```
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
```

```
lis(string_to_decode, bigrams, reps):
 = random_decoder() # Starting decoder

         :oder
         score(
         ode,




         _score

         ge(reps):

         rint out our progress
         s == 0: # Repeat every 10%
         p(.01)
         ext = 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
```

Running the function

```
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 ourselv
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
what we are going to try to do. That is the resolve of His Majestys Government-every
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

'w hysr zntrbm mqbb lodmwgrdlr xhyx wm ybb go xhrwu gqxn wm doxhwdi wt dribrlxrg  ydg
```

Original Text

Encoded Text

Decoding process and resault

```
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
```

Limitations:not applied for non English text

```
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
```

Not correct!

Comes from:UCB Prob140 lab6

PROB 140

Probability for Data Science

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.