1.9 Hangman

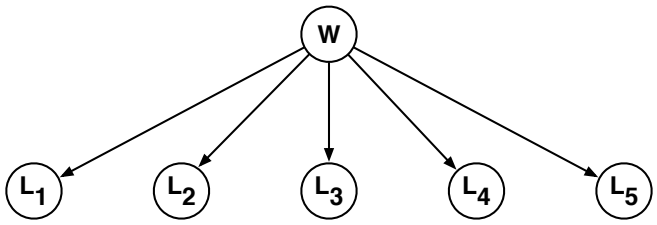
Consider the belief network shown below, where the random variable W stores a five-letter word and the random variable $L_i \in \{\texttt{A}, \texttt{B}, \ldots, \texttt{Z}\}$ reveals only the word's ith letter. Also, suppose that these five-letter words are chosen at random from a large corpus of text according to their frequency:

$$P(W=w) \;=\; \frac{\text{COUNT}(w)}{\sum_{w'} \text{COUNT}(w')},$$

where $\text{COUNT}(w)$ denotes the number of times that w appears in the corpus and where the denominator is a sum over all five-letter words. Note that in this model the conditional probability tables for the random variables L_i are particularly simple:

$$P(L_i=\ell|W=w) \;=\; \begin{cases} 1 & \text{if } \ell \text{ is the } i\text{th letter of } w, \\ 0 & \text{otherwise.} \end{cases}$$

Now imagine a game in which you are asked to guess the word w one letter at a time. The rules of this game are as follows: after each letter (A through Z) that you guess, you'll be told whether the letter appears in the word and also where it appears. Given the *evidence* that you have at any stage in this game, the critical question is what letter to guess next.



Let's work an example. Suppose that after three guesses—the letters D, I, M—you've learned that the letter I does *not* appear, and that the letters D and M appear as follows:

M ___ D ___ M

Now consider your next guess: call it ℓ. In this game the best guess is the letter ℓ that maximizes

$$P\Big(L_2=\ell \textbf{ or } L_4=\ell \;\Big|\; L_1=\texttt{M}, L_3=\texttt{D}, L_5=\texttt{M}, L_2 \notin \{\texttt{D},\texttt{I},\texttt{M}\}, L_4 \notin \{\texttt{D},\texttt{I},\texttt{M}\}\Big).$$

In other works, pick the letter ℓ that is most likely to appear in the blank (unguessed) spaces of the word. For any letter ℓ we can compute this probability as follows:

$$\begin{aligned}
&P\Big(L_2=\ell \textbf{ or } L_4=\ell \;\Big|\; L_1=\texttt{M}, L_3=\texttt{D}, L_5=\texttt{M}, L_2 \notin \{\texttt{D},\texttt{I},\texttt{M}\}, L_4 \notin \{\texttt{D},\texttt{I},\texttt{M}\}\Big) \\
&\quad= \sum_w P\Big(W=w, L_2=\ell \textbf{ or } L_4=\ell \;\Big|\; L_1=\texttt{M}, L_3=\texttt{D}, L_5=\texttt{M}, L_2 \notin \{\texttt{D},\texttt{I},\texttt{M}\}, L_4 \notin \{\texttt{D},\texttt{I},\texttt{M}\}\Big), \qquad \boxed{\textbf{marginalization}} \\
&\quad= \sum_w P(W=w|L_1=\texttt{M}, L_3=\texttt{D}, L_5=\texttt{M}, L_2 \notin \{\texttt{D},\texttt{I},\texttt{M}\}, L_4 \notin \{\texttt{D},\texttt{I},\texttt{M}\}\Big)\; P(L_2=\ell \textbf{ or } L_4=\ell|W=w) \;\boxed{\textbf{product rule \& CI}}
\end{aligned}$$

where in the third line we have exploited the conditional independence (**CI**) of the letters L_i given the word W. Inside this sum there are two terms, and they are both easy to compute. In particular, the second

term is more or less trivial:

$$P(L_2 = \ell \textbf{ or } L_4 = \ell | W = w) \;=\; \begin{cases} 1 & \text{if } \ell \text{ is the second or fourth letter of } w \\ 0 & \text{otherwise.} \end{cases}$$

And the first term we obtain from Bayes rule:

$$P\Big(W = w | L_1 = \texttt{M}, L_3 = \texttt{D}, L_5 = \texttt{M}, L_2 \notin \{\texttt{D}, \texttt{I}, \texttt{M}\}, L_4 \notin \{\texttt{D}, \texttt{I}, \texttt{M}\}\Big)$$

$$= \quad \frac{P\Big(L_1 = \texttt{M}, L_3 = \texttt{D}, L_5 = \texttt{M}, L_2 \notin \{\texttt{D}, \texttt{I}, \texttt{M}\}, L_4 \notin \{\texttt{D}, \texttt{I}, \texttt{M}\} | W = w\Big) P(W = w)}{P(L_1 = \texttt{M}, L_3 = \texttt{D}, L_5 = \texttt{M}, L_2 \notin \{\texttt{D}, \texttt{I}, \texttt{M}\}, L_4 \notin \{\texttt{D}, \texttt{I}, \texttt{M}\})} \quad \boxed{\textbf{Bayes rule}}$$

In the numerator of Bayes rule are two terms; the left term is equal to zero or one (depending on whether the evidence is compatible with the word w), and the right term is the prior probability $P(W = w)$, as determined by the empirical word frequencies. The denominator of Bayes rule is given by:

$$\begin{aligned} & P(L_1 = \texttt{M}, L_3 = \texttt{D}, L_5 = \texttt{M}, L_2 \notin \{\texttt{D}, \texttt{I}, \texttt{M}\}, L_4 \notin \{\texttt{D}, \texttt{I}, \texttt{M}\}) \\ &= \quad \sum_w P(W = w, L_1 = \texttt{M}, L_3 = \texttt{D}, L_5 = \texttt{M}, L_2 \notin \{\texttt{D}, \texttt{I}, \texttt{M}\}, L_4 \notin \{\texttt{D}, \texttt{I}, \texttt{M}\}), \quad \boxed{\textbf{marginalization}} \\ &= \quad \sum_w P(W = w) P(L_1 = \texttt{M}, L_3 = \texttt{D}, L_5 = \texttt{M}, L_2 \notin \{\texttt{D}, \texttt{I}, \texttt{M}\}, L_4 \notin \{\texttt{D}, \texttt{I}, \texttt{M}\} | W = w), \quad \boxed{\textbf{product rule}} \end{aligned}$$

where again all the right terms inside the sum are equal to zero or one. Note that the denominator merely sums the empirical frequencies of words that are compatible with the observed evidence.

Now let's consider the general problem. Let E denote the evidence at some intermediate round of the game: in general, some letters will have been guessed correctly and their places revealed in the word, while other letters will have been guessed incorrectly and thus revealed to be absent. There are two essential computations. The first is the *posterior* probability, obtained from Bayes rule:

$$P(W = w | E) \;=\; \frac{P(E | W = w)\, P(W = w)}{\sum_{w'} P(E | W = w')\, P(W = w')}.$$

The second key computation is the *predictive* probability, based on the evidence, that the letter ℓ appears somewhere in the word:

$$P\Big(L_i = \ell \text{ for some } i \in \{1, 2, 3, 4, 5\} \Big| E\Big) \;=\; \sum_w P\Big(L_i = \ell \text{ for some } i \in \{1, 2, 3, 4, 5\} \Big| W = w\Big) P\Big(W = w \Big| E\Big).$$

Note in particular how the first computation feeds into the second. Your assignment in this problem is implement both of these calculations. **You may program in the language of your choice.**

(a) Download the file *hw1_word_counts_05.txt* that appears with the homework assignment. The file contains a list of 5-letter words (including names and proper nouns) and their counts from a large corpus of Wall Street Journal articles (roughly three million sentences). From the counts in this file compute the prior probability $P(w) = \text{COUNT}(w) / \sum_{w'} \text{COUNT}(w')$. **As a sanity check, print out the fifteen most frequent 5-letter words, as well as the fourteen least frequent 5-letter words. Do your results make sense?**

(b) Consider the following stages of the game. For each of the following, indicate the best next guess—namely, the letter ℓ that is most likely (probable) to be among the missing letters. Also report the probability $P(L_i = \ell \text{ for some } i \in \{1,2,3,4,5\}|E)$ for your guess ℓ. Your answers should fill in the last two columns of this table. (Some answers are shown so that you can check your work.)

correctly guessed	incorrectly guessed	best next guess ℓ	$P(L_i = \ell \text{ for some } i \in \{1,2,3,4,5\}\|E)$
_ _ _ _ _	{}		
_ _ _ _ _	{E, A}		
A _ _ _ S	{}		
A _ _ _ S	{I}		
_ _ O _ _	{A, E, M, N, T}		
_ _ _ _ _	{E, O}	I	0.6366
D _ _ I _	{}	A	0.8207
D _ _ I _	{A}	E	0.7521
_ U _ _ _	{A, E, I, O, S}	Y	0.6270

(c) Turn in a scanned **printout** of your source code. **Do not forget the source code**. It is worth many points on this assignment.

Just to be perfectly clear, you are **not** required in this problem to implement a user interface or any general functionality for the game of hangman. You will only be graded on your word lists in (a), the completed table for (b), and your source code in (c).

Solution. *(a) We print out the fifteen most frequent 5-letter words, as well as the fourteen least frequent 5-letter words.*

hw1_question 1.9a

October 3, 2022

1 Source Code and output for part (a)_Problem 1.9_Hw1_CSE 250A_Fall 2022

[4]:
```
# SOLUTION of problem 1.9 part a, HW1 : CSE 250 A, Fall 2022
rawwordcountdict ={}
with open("hw1_word_counts_05.txt")as file:
    for line in  file:
        (key,value)=line.split()
        rawwordcountdict[key]=int(value)    # this imports our file as a␣
 ↪dictionary with words and their number of occurrences as values

# print(rawwordcountdict)

# converting the dictionary to a sorted list and then again a dictionary


marklist=sorted(rawwordcountdict.items(), key=lambda x:x[1]) # creates a sorted␣
 ↪list based on values (occurences) of the dictionary above
sortedwordcountdict = dict(marklist)   # creates a dictionary out of the sorted␣
 ↪list again

#print(sortedwordcountdict) # prints the sorted dictionary with values in␣
 ↪ascending order of occurrences

# creating probability dictionary
totwordoc=0

for k in sortedwordcountdict.values():
    totwordoc=totwordoc+k

#print(totwordoc)    # total number of occurence for all the words: used for␣
 ↪calculating the probability of occurence of a word

probabilitieswords = {}
for word in sortedwordcountdict.keys():
    probabilitieswords[word] = sortedwordcountdict[word] /totwordoc #␣
 ↪calculates the probabilities
```

```
l=len(sortedwordcountdict) # gives the length of the dictionary or rather no.␣
 ↪of words (5 letter) in the text file

# printing the fifteen most frequent 5 letter words

print('the most frequent fifteen 5 letter words (in the ascending order of␣
 ↪their number of occurrences) are ')

print((marklist[l-15:l]))

# print(list(probabilitieswords.items())[l-15:l]) # sanity check that the list␣
 ↪of words with probabilities of occurrence gives the same output

print("\n \n")

# printing the fourteen least frequent 5 letter words
print('the least frequent fourteen 5 letter words (in the ascending order of␣
 ↪their number of occurrences) are ')

print(marklist[:14])
#print(list(probabilitieswords.items())[:14]) # sanity check that the list of␣
 ↪words with probabilities of occurrence gives the same output
```

```
the most frequent fifteen 5 letter words (in the ascending order of their number
of occurrences) are
[('SIXTY', 73086), ('THERE', 86502), ('YEARS', 88900), ('FORTY', 94951),
('OTHER', 106052), ('FIFTY', 106869), ('FIRST', 109957), ('AFTER', 110102),
('WHICH', 142146), ('THEIR', 145434), ('ABOUT', 157448), ('WOULD', 159875),
('EIGHT', 165764), ('SEVEN', 178842), ('THREE', 273077)]
```

```
the least frequent fourteen 5 letter words (in the ascending order of their
number of occurrences) are
[('BOSAK', 6), ('CAIXA', 6), ('MAPCO', 6), ('OTTIS', 6), ('TROUP', 6), ('CCAIR',
7), ('CLEFT', 7), ('FABRI', 7), ('FOAMY', 7), ('NIAID', 7), ('PAXON', 7),
('SERNA', 7), ('TOCOR', 7), ('YALOM', 7)]
```

These outputs do make sense as we can see the first set of outputs are very frequent words in english and second set of outputs are very rare (some of them may not even be english words perhaps?).

(b) Based on the source code of problem 1.9 c below, we complete (and verify) the given table of predicting the next letters with the probabilities of occurrence of those letters.

correctly guessed	*incorrectly guessed*	*best next guess* ℓ	$P(L_i = \ell \text{ for some } i \in \{1,2,3,4,5\}\|E)$
– – – – –	{}	***E***	*0.5394172389647974*
– – – – –	{E, A}	***O***	*0.5340315651557659*
A – – – S	{}	***E***	*0.7715371621621623*
A – – – S	{I}	***E***	*0.7127008416220353*
– – O – –	{A, E, M, N, T}	***R***	*0.7453866259829711*
– – – – –	{E, O}	I	*0.6365554141009612*
D – – I –	{}	A	*0.8206845238095238*
D – – I –	{A}	E	*0.7520746887966805*
– U – – –	{A, E, I, O, S}	Y	*0.6269651101630529*

(c) In the following we have published the source code with which we do the part (b) of the problem. The code is written in Python and is run using 'JupyterLab'.

hw1_question 1.9bc

October 3, 2022

1 Source Code and outputs for part (b and c)_Problem 1.9_Hw1_CSE 250A_Fall 2022

[1]:
```
# SOLUTION of problem 1.9 part b (and c), HW1 : CSE 250 A, Fall 2022
# Author : Soumya Ganguly, PID: A53274333
import string

rawwordcountdict ={}
with open("hw1_word_counts_05.txt")as file:
    for line in  file:
        (key,value)=line.split()
        rawwordcountdict[key]=int(value)    # this imports our file as a␣
 ↪dictionary with words and their number of occurrences as values

        # print(rawwordcountdict)

# converting the dicytionary to a list


marklist=sorted(rawwordcountdict.items(), key=lambda x:x[1]) # creates a sorted␣
 ↪list based on values (occurences) of the dictionary above
sortedwordcountdict = dict(marklist)   # creates a dictionary out of the sorted␣
 ↪list again

#print(sortedwordcountdict) # prints the sorted dictionary with values in␣
 ↪ascending order of occurrences



# creating probability dictionary
totwordoc=0 # total occurrence of all the words: to calculate probabilities :

for k in sortedwordcountdict.values():
    totwordoc=totwordoc+k

#print(totwordoc)    # total number of occurence for all the words: needed for␣
 ↪later probability calculation
```

```
probabilitieswords = {}    # this becomes the probability dictionary that we
 ↪use to call probability of occurrence of a word
for word in sortedwordcountdict.keys():
    probabilitieswords[word] = sortedwordcountdict[word] /totwordoc


# predicting next letter in a word full of blanks based on input

englishalpha=list(string.ascii_uppercase) # all the english alphabets with
 ↪uppercase to check from : which letter will be next


#print(englishalpha)



def probEgivenu(u,correctlyguessed,incorrectguesses):   # calculates
 ↪probability of E being satisfied given a word u
    for l in incorrectguesses:
        for i in range(5):
            if u[i]==l:
                return 0
    listcorrect=dict(correctlyguessed)
    for l in listcorrect.keys():
        for i in listcorrect[l]:
            if u[i] != l:
                return 0
        for i in range(5):
            if u[i] == l:
                if not i in listcorrect[l]:
                    return 0
    return 1



def probEandu(u,correctlyguessed,incorrectguesses):              # calculates
 ↪probability of a word being u and E being satisfied
    p= probabilitieswords[u]* probEgivenu(u,correctlyguessed,incorrectguesses)
    return p


def posteriorprobcalc(w,correctlyguessed,incorrectguesses):     # posterior
 ↪probability of E being satisfied given word being w
    s=0
    for u in sortedwordcountdict.keys():
        s=s+ probEandu(u,correctlyguessed,incorrectguesses)
```

```
    return probEandu(w,correctlyguessed,incorrectguesses)/s


def koccursornot(k,w):    # finds whether a letter k is in a word w or not
    for i in w:
        if k==i:
            return 1
    return 0


def predictiveprobcalc(k,correctlyguessed,incorrectguesses):  # predicts the␣
 ↪probability of letter k coming next, based on info
    if k in incorrectguesses:
        return 0
    lettersofcorrectlyguessed=[x[0] for x in correctlyguessed]
    if k in lettersofcorrectlyguessed:
        return 0
    s=0
    for w in sortedwordcountdict.keys():
        s=s+koccursornot(k,w)*posteriorprobcalc(w,␣
 ↪correctlyguessed,incorrectguesses)
    return s


#predicts the probability of which letter next is highest along with that␣
 ↪letter
def highestletprobpredictor(correctlyguessed,incorrectguesses):
    (highestprob, highestprobletter)=(-1,-1)

    for k in englishalpha:
        if predictiveprobcalc(k,correctlyguessed,incorrectguesses) >␣
 ↪highestprob:
            ␣
 ↪(highestprob,highestprobletter)=(predictiveprobcalc(k,correctlyguessed,incorrectguesses),k)

    print((highestprob,highestprobletter))


# while inputting we assume that indexing of a word starts from zero so that we␣
 ↪put the position of the correctly guessed letters accordingly

# calculates highestletprobpredictor(correctlyguessed,incorrectguesses)

highestletprobpredictor([] ,[])

highestletprobpredictor([] ,['E','A'])
```

```
highestletprobpredictor([('A',[0]),('S',[4])] ,[])

highestletprobpredictor([('A',[0]),('S',[4])] ,['I'])

highestletprobpredictor([('O',[2])] ,['A','E','M','N','T'])

highestletprobpredictor([] ,['E','O'])

highestletprobpredictor([('D',[0]),('I',[3])] ,[])

highestletprobpredictor([('D',[0]),('I',[3])] ,['A'])

highestletprobpredictor([('U',[1])] ,['A','E','I','O','S'])
```

```
(0.5394172389647974, 'E')
(0.5340315651557659, 'O')
(0.7715371621621623, 'E')
(0.7127008416220353, 'E')
(0.7453866259829711, 'R')
(0.6365554141009612, 'I')
(0.8206845238095238, 'A')
(0.7520746887966805, 'E')
(0.6269651101630529, 'Y')
```