GIT Department of Computer Engineering

CSE 654 / 484 Fall 2022

Homework 2 # Report

Okan Torun

1801042662

How I Handled the Problem:

1) In order to better analyze the results in the assignment, I first consider a small dataset.
2) I used a ready-made library to split the text read from the file into syllables.First,Iinstalled pip install **git+https://github.com/ftkurt/python-syllable.git@master.** Then I did the syllable splitting as seen in the code block below.

```
text_file = open("dataset.txt", "r",encoding="utf-8")
line = text_file.read()
encoder = Encoder(lang="tr", limitby="vocabulary", limit=3000)
tokens = encoder.tokenize(line)
```

3) Then I created the n-gram tables respectively as requested. In order to create the n-gram tables, I had to adjust the text I read accordingly.

Bigram:

I created bigram using **nltk** library for syllabic text for bigram.

```
def generate_ngrams(s, n):
    s = s.lower()
    s = re.sub(r'[^a-zA-Z0-9\s]', ' ', s)
    tokens = [token for token in s.split(" ") if token != ""]
    ngrams = zip(*[tokens[i:] for i in range(n)])
    dt = datetime.now()
    return [" ".join(ngram) for ngram in ngrams]
```

```
bigram = list(generate_ngrams(tokens, 1))
```

With this process, it was converted to text bigram format.

```
['o', 'kan', 'o', 'kul', 'da', 'ki', 'tap', 'o', 'ku', 'du']
```

Then, before creating the frequency table in the converted bigram format, I removed the repeating elements from the list.

```
unique_bigram = unique(bigram)
```

```python
def unique(list1):
    unique_list = []
    for x in list1:
        if x not in unique_list:
            unique_list.append(x)
    return unique_list
```

The values representing the rows and columns of this matrix are the same. Because it's a bigram. Then I created bigram matrix and this matrix represents frequency values.

The algorithm I created the frequencies is as follows:

```python
for i in range(0,len(bigram)-1):
    row = unique_bigram.index(bigram[i])
    col = unique_bigram.index(bigram[i+1])
    zeors_array[row][col]+=1
```

Here, I traverse the syllabic form of the text, increasing the index, that is, the frequency, of each consecutive syllable in the matrix.

```
[[0. 1. 1. 0. 0. 0. 1. 0.]
 [1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0.]
 [0. 0. 0. 0. 0. 1. 0. 0.]
 [1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 1.]
 [0. 0. 0. 0. 0. 0. 0. 0.]]
```

```
['o', 'kan', 'kul', 'da', 'ki', 'tap', 'ku', 'du']
```

This list represents the rows and columns in the table because I made the text unique. For example, cells [0,1] and [0,2] of the matrix, respectively "o kan" and "o kul". And the frequency values are 1. This means that 'kan' came once after 'o' and 'kul' came once after 'o'.

Probability with MLE:

Then I moved on to the probability matrix calculation step. The homework was the desired GT smoothing method. But first I tried the matrix creation process with the **MLE** method.

$$P(w_i \mid w_{i-1}) = \frac{count(w_{i-1}, w_i)}{count(w_{i-1})}$$

In the formula here, C('okan')/C('o') is calculated while calculating the probability of 'kan' coming after 'o'. And I filled the MLE table by calculating all the probabilities in the table in this way.

```
[[0.         0.33333333 0.33333333 0.         0.         0.
  0.33333333 0.        ]
 [1.         0.          0.         0.         0.         0.
  0.         0.        ]
 [0.         0.          0.         1.         0.         0.
  0.         0.        ]
 [0.         0.          0.         0.         1.         0.
  0.         0.        ]
 [0.         0.          0.         0.         0.         1.
  0.         0.        ]
 [1.         0.          0.         0.         0.         0.
  0.         0.        ]
 [0.         0.          0.         0.         0.         0.
  0.         1.        ]
 [0.         0.          0.         0.         0.         0.
  0.         0.        ]]
```

The reason for finding 0.333 here is that there are three 'o' and one 'okan'. The same is valid for other transactions.

The algorithm here is as follows:

```
for i in range(0,len(unique_bigram)):
    for j in range(0,len(unique_bigram)):
        row = unique_bigram.index(unique_bigram[i])
        col = unique_bigram.index(unique_bigram[j])
        numerator = zeors_array[row][col]
        denominator = countX(bigram,unique_bigram[i])
        probability_arr[row][col] = numerator/denominator
```

Here, I go through each cell in the matrix and do the necessary operations to obtain the calculation given in the formula. I write the value in the frequency matrix in the numerator part and the total number of that syllable in the corpus in the denominator part.

Then, I calculated the perplexity using the Markov Chain rule algorithm from the probability matrix.

$$p(w_1, w_2, w_3, \cdots w_n) = p(w_1) * p(w_2|w_1) * \cdots * p(w_n|w_{n-1})$$

$$p(w) = \frac{count(w)}{count(vocab)}$$

This is the perplexity formula I calculated for the bigram. Since the homework also asked for the product of the logarithm, I took the logarithm based on log2 before multiplying the probability value. When I search for the 'o' syllable, the perplexity result is as follows.

```
[[0.         0.33333333 0.33333333 0.         0.         0.
  0.33333333 0.        ]
 [1.         0.         0.         0.         0.         0.
  0.         0.        ]
 [0.         0.         0.         1.         0.         0.
  0.         0.        ]
 [0.         0.         0.         0.         1.         0.
  0.         0.        ]
 [0.         0.         0.         0.         0.         1.
  0.         0.        ]
 [1.         0.         0.         0.         0.         0.
  0.         0.        ]
 [0.         0.         0.         0.         0.         0.
  0.         1.        ]
 [0.         0.         0.         0.         0.         0.
  0.         0.        ]]
Bigram perplexity 2.666666666666667
```

The algorithm is as below:

```
for i in range(0,len(sentence_token)-1):
    row = unique_bigram.index(sentence_token[i])
    col = unique_bigram.index(sentence_token[i+1])
    perplexity = perplexity + np.log2(probability_arr[row][col])

exist_count = bigram.count(sentence_token[0])
perplexity = perplexity + np.log2(exist_count/len(unique_bigram))
perplexity = np.power(2, -perplexity)
return perplexity
```

These operations are performed by visiting the probability matrix of the given token.

GT_SMOOTHING

I also performed the calculation of the probability using GT SMOOTHING. But when I try it in the small corpus, the perplexity is 0. Because this is how the analysis is done in the GT smoothing algorithm. Good Turing algoritması ile olasılık hesaplarken 2 formül kullandım.GT 'nin amacı tablodaki 0 değerlerini olabildiğince azaltmaktır.

If the index value in the frequency table is not 0:

$$c^* = (c+1)\frac{N_{c+1}}{N_c}$$

Its value in the index in the frequency table is 0 :

$$p_0 = N_1/N$$

But before I started this algorithm, I created a **sparse matrix** with the np array in the frequency table. My aim was to access the frequency values faster. That is, to increase the efficiency.

```
def generate_gt_bigram_matrix(sparse_mtr,freq_arr,bigram):
    rows = freq_arr.shape[0]
    cols = freq_arr.shape[1]
    gt_freq_arr = np.zeros( (rows, cols) )
    freq_one_result = find_frequency(sparse_mtr,1)

    for i in range(0,rows):
        for j in range(0,cols):
            if freq_arr[i][j] == 0:
                numerator = freq_one_result
                denumerator = len(bigram)
                gt_freq_arr[i][j] = numerator/denumerator
            else :
                count = find_frequency(sparse_mtr,freq_arr[i][j])
                numerator = find_frequency(sparse_mtr,count+1)
                denumerator = count
                gt_freq_arr[i][j] = (((count+1)*numerator)/denumerator)
    return gt_freq_arr
```

This function is the one that converts the matrix where I keep the frequency values to the probability table with GT SMOOTHING. I performed the operations in this function with the formula I gave above.

In the case I applied with gt smoothing, the perplexity result is as follows.

4)

```
[[0. 1. 1. 0. 0. 0. 1. 0.]
 [1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0.]
 [0. 0. 0. 0. 0. 1. 0. 0.]
 [1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 1.]
 [0. 0. 0. 0. 0. 0. 0. 0.]]
Bigram perplexity 2.666666666666667
```

TWO GRAM:

I also needed bigram and trigram data to search in TWOgram. Because it allowed me to increase the frequency by searching for the token that I combined in the twogram table as an algorithm to see if it exists in the trigram table.

```python
def markov_chain_twogram_perp(sentence_token_bigram,twogram,unique_twogram,bigram,trigram,perplexity):
    probability_arr = generate_twogram_matrix(unique_twogram,twogram,bigram,trigram)
    unique_bigram = unique(bigram)
    probability_arr_bigram = generate_bigram_matrix(bigram,unique_bigram)

    #ilk hece
    exist_count = bigram.count(sentence_token_bigram[0])
    perplexity = perplexity + np.log2((exist_count/len(unique_bigram)))
    #ikinci heceden sonra ilk hecenin gelmesi
    row = unique_bigram.index(sentence_token_bigram[0])
    col = unique_bigram.index(sentence_token_bigram[1])
    perplexity = perplexity + np.log2(probability_arr_bigram[row][col])

    sentence_token_twogram = list(generate_ngrams(sentence, 2))

    for i in range(0,len(sentence_token_twogram)-1):
        row = unique_twogram.index(sentence_token_twogram[i])
        col = unique_bigram.index(sentence_token_bigram[i+2])
        perplexity = perplexity + np.log2(probability_arr[row][col])

    perplexity = perplexity/2
    perplexity = np.power(2, -perplexity)
    return perplexity
```

Here, while applying the markov rule, I calculated the first and second syllables specially since there are twograms. After that, the double binary checks started.
Here is the adapted version of 'okan okulda kitap okudu' for twograms:

```
['o kan', 'kan o', 'o kul', 'kul da', 'da ki', 'ki tap', 'tap o', 'o ku', 'ku du']
```

Frequency table:

```
[[1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0.]
 [0. 0. 0. 0. 0. 1. 0. 0.]
 [1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 0. 0. 0. 1.]
 [0. 0. 0. 0. 0. 0. 0. 0.]]
```

The [0,0] element of the Matrix is the frequency of occurrence of 'o' after ‘o kan’ and its number is 1.

GT smoothing matrix output:

```
[[0.  0.8 0.8 0.8 0.8 0.8 0.8 0.8]
 [0.8 0.8 0.  0.8 0.8 0.8 0.8 0.8]
 [0.8 0.8 0.8 0.  0.8 0.8 0.8 0.8]
 [0.8 0.8 0.8 0.8 0.  0.8 0.8 0.8]
 [0.8 0.8 0.8 0.8 0.8 0.  0.8 0.8]
 [0.  0.8 0.8 0.8 0.8 0.8 0.8 0.8]
 [0.8 0.8 0.8 0.8 0.8 0.8 0.  0.8]
 [0.8 0.8 0.8 0.8 0.8 0.8 0.8 0. ]
 [0.8 0.8 0.8 0.8 0.8 0.8 0.8 0.8]]
```

Perplexity result:

```
C:\Users\okant\Desktop\nlp-hw2\hw2.py:125: RuntimeWarning: divide by zero encountered in log2
  perplexity = perplexity + np.log2(probability_arr_bigram[row][col])
twogram perplexity inf
```

As I mentioned at the beginning of the report, when we apply gt smoothing in the small dataset, the number of cells that is 0 in the matrix is more and when we apply the logarithm operation, the result is infinite.

TRI GRAM:

If we apply the small dataset example for TRIgram:
Here is the adapted version of 'okan okulda kitap okudu' for trigrams:

```
['o kan o', 'kan o kul', 'o kul da', 'kul da ki', 'da ki tap', 'ki tap o', 'tap o ku', 'o ku du']
```

Frequency table:

```
[[0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0.]
 [0. 0. 0. 0. 0. 1. 0. 0.]
 [1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 0. 0. 0. 1.]
 [0. 0. 0. 0. 0. 0. 0. 0.]]
```

The [0,2] element of the Matrix is the frequency of occurrence of 'kul' after ‘o kan o’ and its number is 1.

GT smoothing matrix output:

```
[[0.7 0.7 0.  0.7 0.7 0.7 0.7 0.7]
 [0.7 0.7 0.7 0.  0.7 0.7 0.7 0.7]
 [0.7 0.7 0.7 0.7 0.  0.7 0.7 0.7]
 [0.7 0.7 0.7 0.7 0.7 0.  0.7 0.7]
 [0.  0.7 0.7 0.7 0.7 0.7 0.7 0.7]
 [0.7 0.7 0.7 0.7 0.7 0.7 0.  0.7]
 [0.7 0.7 0.7 0.7 0.7 0.7 0.7 0. ]
 [0.7 0.7 0.7 0.7 0.7 0.7 0.7 0.7]]
```

Perplexity result:

```
C:\Users\okant\Desktop\nlp-hw2\hw2.py:155: RuntimeWarning: divide by zero encountered in log2
  perplexity = perplexity+np.log2(probability_arr_bigram[row][col])
C:\Users\okant\Desktop\nlp-hw2\hw2.py:160: RuntimeWarning: divide by zero encountered in log2
  perplexity = perplexity+np.log2(probability_arr_twogram[row][col])
['o', 'kan', 'o', 'kul', 'da', 'ki', 'tap', 'o', 'ku', 'du']
trigram perplexity inf
```

The same issue I mentioned for TWOgram applies here as well.

5) As requested in this article, I tried to produce sentences through n-gram models. I tried to continue this pattern over n-grams by taking a pattern from the small dataset I sent and used in the homework file. While performing this operation, I got help from the **gt_smoothing table.**

```
def generate_sentece(sentence_token,unique_ngram,unique_bigram,gt_array,n):
    row = unique_ngram.index(sentence_token[len(sentence_token)-1])
    cols = gt_array.shape[1]
    random_sentence = ''
    max_prob = gt_array[row][0]
    max_col = 0
    print(gt_array)
    for i in range(0,cols):
        if gt_array[row][i] > max_prob:
            max_prob = gt_array[row][i]
            max_col = i
    if n==1:
        random_sentence = random_sentence + unique_bigram[max_col]
```

After filling the GT_smoothing table, I examined this table with the above function. If I am working on bigram, I went to the row of the last syllable and took the syllable in the column with the highest rate and added to the string. Then I continued this process based on the syllable I just added. I repeated once.

Test Cases:

BiGram:

gt-smoothing table in 1-grams when I use the "mücadele edecek" token:

```
[[ 0.43586359 19.          0.43586359 ...  0.43586359  0.43586359
   0.43586359]
 [ 0.43586359  0.43586359 19.          ...  0.43586359  0.43586359
   0.43586359]
 [ 0.43586359  0.43586359  0.43586359 ...  0.43586359  0.43586359
   0.43586359]
 ...
 [ 0.43586359  0.43586359  0.43586359 ...  0.43586359  0.43586359
   0.43586359]
 [ 0.43586359  0.43586359  0.43586359 ...  0.43586359  0.43586359
   0.43586359]
 [ 0.43586359  0.43586359  0.43586359 ...  0.43586359  0.43586359
   0.43586359]]
```

The result of the generated sentence:

```
cümle bigram:::: ['mu', 'ca', 'de', 'le', 'e', 'de', 'cek'] cengizhaninka
```

Perplexity result:

```
Bigram perplexity 35.7151316733074
```

TwoGram:

gt-smoothing table in 2-grams when I use the "mücadele edecek" token:

```
[[0.74719472 0.74719472 0.         ... 0.74719472 0.74719472 0.74719472]
 [0.74719472 0.74719472 0.74719472 ... 0.74719472 0.74719472 0.74719472]
 [0.74719472 0.74719472 0.74719472 ... 0.74719472 0.74719472 0.74719472]
 ...
 [0.74719472 0.74719472 0.74719472 ... 0.74719472 0.74719472 0.74719472]
 [0.74719472 0.74719472 0.74719472 ... 0.74719472 0.74719472 0.74719472]
 [0.74719472 0.74719472 0.74719472 ... 0.74719472 0.74719472 0.74719472]]
```

The result of the generated sentence:

```
cümle twogram:::: ['mu ca', 'ca de', 'de le', 'le e', 'e de', 'de cek'] yirsoydumkamdek
```

Perplexity result:

```
twogram perplexity 3.460008567369807
```

TriGram:

gt-smoothing table in 3-grams when I use the "mücadele edecek" token:

```
[[0.88426843 0.88426843 0.88426843 ... 0.88426843 0.88426843 0.88426843]
 [0.88426843 0.88426843 0.88426843 ... 0.88426843 0.88426843 0.88426843]
 [0.88426843 0.88426843 0.88426843 ... 0.88426843 0.88426843 0.88426843]
 ...
 [0.88426843 0.88426843 0.88426843 ... 0.88426843 0.88426843 0.88426843]
 [0.88426843 0.88426843 0.88426843 ... 0.88426843 0.88426843 0.88426843]
 [0.88426843 0.88426843 0.88426843 ... 0.88426843 0.88426843 0.88426843]]
```

The result of the generated sentence:

```
cümle trigram:::: ['mu ca de', 'ca de le', 'de le e', 'le e de', 'e de cek'] dekpabudalso
```

Perplexity result:

```
trigram perplexity 134.5374467448458
```

Since I am using a small dataset, I think that it has a high perplexity rate and is not very meaningful.

****IMPORTANT****

It takes a long time to produce results in the given dataset due to computer requirements. Therefore, I produced the results with a smaller dataset that I created from the same dataset.

Note: While opening the file for testing, I gave the name of the dataset I sent myself. I also assigned a sentence as Sentence. These can be changed to get different results.