CSE654

NATURAL LANGUAGE PROGRAMMING

HW2

REPORT

1801042631

PART 1

Turkish Wikipedia Dump is downloaded.

[Turkish Wikipedia Dump | Kaggle]

PART 2

First, it was checked whether the letters read from the text file are Turkish characters.

```
def turkish_to_english(string):
    choices = {"İ": "I", "ş": "s", "Ş": "S", "ı": "i", "ö": "o", "ü": "u", "ç": "c", "ğ": "g", "Ç": "C", "Ö": "O", "Ü": "U", "Ğ"
               "ê": "e", "ô": "o", "Â": "A", "Î": "I", "Û": "U", "Ê": "E", "Ô": "O", "â": "a", "î": "i", "û": "u", "ê": "e", "ô"
    for i in range(len(string)):
        string = string.replace(
            string[i:i+1], choices.get(string[i], string[i]))
    return string
```

Secondly, the read string is divided into syllables. A ready-made library was used for this process. Link of the library: https://github.com/ftkurt/python-syllable/blob/master/syllable/syllable.py

You can set up the library:

“pip install git+https://github.com/ftkurt/python-syllable.git@master”

```
# params chosen for demonstration purposes
encoder = Encoder(lang="tr", limitby="vocabulary", limit=3000)


# parse string into syllables


def parse_syllable(string):
    string = turkish_to_english(string)
    return encoder.tokenize(string)
```

PART 3

Tables of 1 gram, 2 gram and 3 gram of the considered string were extracted.

For example:

1-gram of “samet sakat salata sakız sakal” is:

['sa', 'met', 'sa', 'kat', 'sa', 'la', 'ta', 'sa', 'kiz', 'sa', 'kal']

2-gram of “sakallı adam” is:

['sa met', 'met sa', 'sa kat', 'kat sa', 'sa la', 'la ta', 'ta sa', 'sa kiz', 'kiz sa', 'sa kal']

3-gram of “sakallı adam” is:

['sa met sa', 'met sa kat', 'sa kat sa', 'kat sa la', 'sa la ta', 'la ta sa', 'ta sa kiz', 'sa kiz sa', 'kiz sa kal']

```python
def generate_ngrams(s, n):
    # Convert to lowercases
    s = s.lower()

    # Replace all none alphanumeric characters with spaces
    s = re.sub(r'[^a-zA-Z0-9\s]', ' ', s)

    # Break sentence in the token, remove empty tokens
    tokens = [token for token in s.split(" ") if token != ""]

    # Use the zip function to help us generate n-grams
    # Concatentate the tokens into ngrams and return
    ngrams = zip(*[tokens[i:] for i in range(n)])
    return [" ".join(ngram) for ngram in ngrams]
```

```
# collect bigrams
n = 1
bigrams = generate_ngrams(parsed_words_file, n)
# collect two grams
n = 2
towgrams = generate_ngrams(parsed_words_file, n)
# collect three grams
n = 3
threegrams = generate_ngrams(parsed_words_file, n)
```

The unique ones were kept to make the table.

For example:

1-gram of “samet sakat salata sakız sakal” is:

['sa', 'met', 'kat', 'la', 'ta', 'kiz', 'kal']

2-gram of “sakallı adam” is:

['sa met', 'met sa', 'sa kat', 'kat sa', 'sa la', 'la ta', 'ta sa', 'sa kiz', 'kiz sa', 'sa kal']

3-gram of “sakallı adam” is:

['sa met sa', 'met sa kat', 'sa kat sa', 'kat sa la', 'sa la ta', 'la ta sa', 'ta sa kiz', 'sa kiz sa', 'kiz sa kal']

```
# collect unique bigrams in a list
unique_bigrams = []
for grams in bigrams:
    if grams not in unique_bigrams:
        unique_bigrams.append(grams)
# collect unique two grams in a list
unique_towgrams = []
for grams in towgrams:
    if grams not in unique_towgrams:
        unique_towgrams.append(grams)
# collect unique three grams in a list
unique_threegrams = []
for grams in threegrams:
    if grams not in unique_threegrams:
        unique_threegrams.append(grams)
```

Then, tables were created based on their frequencies.

```
def generate_bigram_matrix(unique_bigrams, bigrams):
    bigram_matrix = np.zeros((len(unique_bigrams), len(unique_bigrams)))
    for i in range(len(bigrams)-1):
        bigram_matrix[unique_bigrams.index(
            bigrams[i])][unique_bigrams.index(bigrams[i+1])] += 1
    gt_smooth = good_turing_smooting(bigram_matrix, bigrams, unique_bigrams)

    # edit bigram_matrix to make it more accurate
    # for i in range(len(gt_smooth)):
    #     if(np.sum(gt_smooth[i]) != 0):
    #         gt_smooth[i] = gt_smooth[i] / np.sum(gt_smooth[i])
    return gt_smooth
```

```
def generate_towgram_matrix(unique_towgrams, towgrams, unique_bigrams):
    towgram_matrix = np.zeros((len(unique_towgrams), len(unique_bigrams)))
    for i in range(len(towgrams)-1):
        towgram_matrix[unique_towgrams.index(
            towgrams[i])][unique_bigrams.index(parse_string_two(towgrams[i+1]))] += 1

    gt_smooth = good_turing_smooting(towgram_matrix, towgrams, unique_towgrams)
    # edit towgram_matrix to make it more accurate
    # for i in range(len(towgram_matrix)):
    #     if(np.sum(towgram_matrix[i]) != 0):
    #         towgram_matrix[i] = towgram_matrix[i] / np.sum(towgram_matrix[i])
    return gt_smooth
```

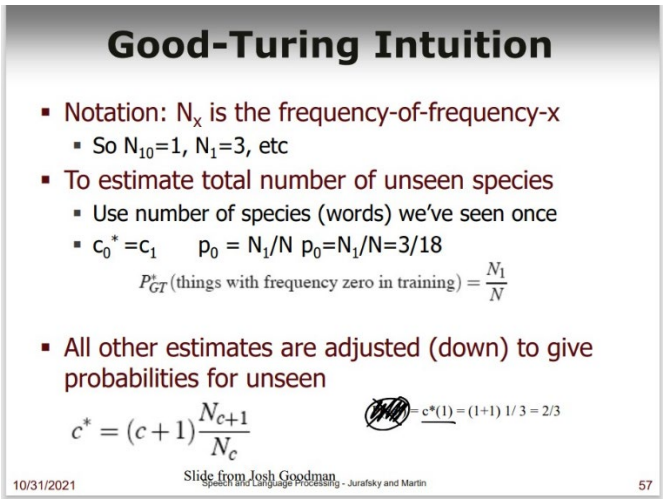
```
def generate_threegram_matrix(unique_threegrams, threegrams, unique_bigrams):
    threegram_matrix = np.zeros((len(unique_threegrams), len(unique_bigrams)))
    for i in range(len(threegrams)-1):
        threegram_matrix[unique_threegrams.index(
            threegrams[i])][unique_bigrams.index(parse_string_three(threegrams[i+1]))] += 1

    gt_smoothing = good_turing_smooting(
        threegram_matrix, threegrams, unique_threegrams)
    # edit threegram_matrix to make it more accurate
    # for i in range(len(threegram_matrix)):
    #     if(np.sum(threegram_matrix[i]) != 0):
    #         threegram_matrix[i] = threegram_matrix[i] / \
    #             np.sum(threegram_matrix[i])
    return gt_smoothing
```

While filling the tables, **Good Turing Smooting** method was used against the probability of getting 0.

```
def good_turing_smooting(ngram_matrix, ngrams, unique_ngrams):
    gt_smooth = np.zeros((len(unique_ngrams), len(unique_ngrams)))
    sparse_matrix = csc_matrix(ngram_matrix)
    count_one = count_element_matrix(sparse_matrix, 1)
    # calculate good turing smoothing
    for i in range(len(ngram_matrix)):
        for j in range(len(ngram_matrix[i])):
            if(ngram_matrix[i][j] == 0):
                gt_smooth[i][j] = count_one / len(ngrams)
            else:
                gt_smooth[i][j] = (ngram_matrix[i][j]+1) * \
                    count_element_matrix(sparse_matrix, ngram_matrix[i][j]+1) / \
                    count_element_matrix(sparse_matrix, ngram_matrix[i][j])
    return gt_smooth
```

The Good Turing Smooting method was used as follows:



While using the Good Turing Method, the **Sparse Matrix** method was used to find non-0 indexes more easily.

```
def count_element_matrix(sparse_matrix, element):
    count = 0
    for i in sparse_matrix.data:
        if(i == element):
            count += 1
    return count
```

PART 4

Some sentences were selected from the text file and the probability of the sentence being found in the text file was calculated. Chain rule and markov assumption methods were used in the calculation.

Let's assume the sentence is: "Ali okula yürüyerek gitti"

P("Ali okula yürüyerek gitti")

For 1-gram:

= P("Ali") * P("okula"|" Ali")* P("yürüyerek"|" okula")* P("gitti"|" yürüyerek")

For 2-gram:

= P("Ali") * P("okula"|" Ali")* P("yürüyerek"|" Ali okula")* P("gitti"|"okula yürüyerek")

For 3-gram:

= P("Ali") * P("okula"|" Ali")* P("yürüyerek"|" Ali okula")* P("gitti"|"Ali okula yürüyerek")

Calculations were made using the tables we created. bigram_matrix,twogram_matrix,threegram_matrix

```
def chain_rule_bigram(bigram_matrix, search, bigrams, unique_bigrams, index, prob):
    if index == len(search)-1:
        return prob * probab(search[0], bigrams, unique_bigrams)
    else:
        prob = prob * \
            bigram_matrix[unique_bigrams.index(
                search[index])][unique_bigrams.index(search[index+1])]
        return chain_rule_bigram(bigram_matrix, search, bigrams, unique_bigrams, index+1, prob)
```

```
def chain_rule_towgram(towgram_matrix, bigram_matrix, search, towgrams, bigrams, unique_towgrams, unique_bigrams, index, prob):
    if index == len(search)-1:
        return prob * probab(search[0].split(" ")[0], bigrams, unique_bigrams) * bigram_matrix[unique_bigrams.index(
            search[0].split(" ")[0])][unique_bigrams.index(search[0].split(" ")[1])]
    else:
        prob = prob * \
            towgram_matrix[unique_towgrams.index(
                search[index])][unique_bigrams.index(parse_string_two(search[index+1]))]
        return chain_rule_towgram(towgram_matrix, bigram_matrix, search, towgrams, bigrams, unique_towgrams, unique_bigrams, ind
```

```
def chain_rule_threegram(threegram_matrix, towgram_matrix, bigram_matrix, search, threegrams, towgrams, bigrams, unique_threegra
    if index == len(search)-1:
        return prob * probab(search[0].split(" ")[0], bigrams, unique_bigrams) * bigram_matrix[unique_bigrams.index(
            search[0].split(" ")[0])][unique_bigrams.index(search[0].split(" ")[1])] * towgram_matrix[unique_towgrams.index(
                search[0].split(" ")[0]+" "+search[0].split(" ")[1])][unique_bigrams.index(search[0].split(" ")[2])]
    else:
        prob = prob * \
            threegram_matrix[unique_threegrams.index(
                search[index])][unique_bigrams.index(parse_string_three(search[index+1]))]
        return chain_rule_threegram(threegram_matrix, towgram_matrix, bigram_matrix, search, threegrams, towgrams, bigrams, uniq
```

And then the **perplexity** of these sentences was calculated using the logarithmic formula.

```
def perplexity_bigram(bigram_matrix, search, bigrams, unique_bigrams, index, perp):
    if index == len(search)-1:
        perp = perp + math.log2(probab(search[0], bigrams, unique_bigrams))
        return pow(2, -perp)
    else:
        perp = perp + \
            math.log2(bigram_matrix[unique_bigrams.index(
                search[index])][unique_bigrams.index(search[index+1])])
        return perplexity_bigram(bigram_matrix, search, bigrams, unique_bigrams, index+1, perp)
```

```
def perplexity_towgram(towgram_matrix, bigram_matrix, search, towgrams, bigrams, unique_towgrams, unique_bigrams, index, perp):
    if index == len(search)-1:
        perp = perp + math.log2(probab(search[0].split(" ")[0], bigrams, unique_bigrams)) + math.log2(bigram_matrix[unique_bigra
            search[0].split(" ")[0])][unique_bigrams.index(search[0].split(" ")[1])])
        return pow(2, -perp/2)
    else:
        perp = perp + \
            math.log2(towgram_matrix[unique_towgrams.index(
                search[index])][unique_bigrams.index(parse_string_two(search[index+1]))])
        return perplexity_towgram(towgram_matrix, bigram_matrix, search, towgrams, bigrams, unique_towgrams, unique_bigrams, ind
```

```
def perplexity_threegram(threegram_matrix, towgram_matrix, bigram_matrix, search, threegrams, towgrams, bigrams, unique_threegra
    if index == len(search)-1:
        perp = perp + math.log2(probab(search[0].split(" ")[0], bigrams, unique_bigrams)) + math.log2(bigram_matrix[unique_bigra
            search[0].split(" ")[0])][unique_bigrams.index(search[0].split(" ")[1])]) + math.log2(towgram_matrix[unique_towgrams
                search[0].split(" ")[0]+" "+search[0].split(" ")[1])][unique_bigrams.index(search[0].split(" ")[2])])
        return pow(2, -perp/3)
    else:
        perp = perp + \
            math.log2(threegram_matrix[unique_threegrams.index(
                search[index])][unique_bigrams.index(parse_string_three(search[index+1]))])
        return perplexity_threegram(threegram_matrix, towgram_matrix, bigram_matrix, search, threegrams, towgrams, bigrams, uniq
```

The following formula was used while making perplexity calculations:

$$PP(s) = 2^{log_2^{PP(s)}} = 2^{-\frac{1}{n}log(p(s))}$$

let $l = \frac{1}{n}log(p(s))$

For unigram $l = \frac{1}{n}(logp(w_1) + \cdots + logp(w_n))$

For bigram $l = \frac{1}{n}(logp(w_1) + logp(w_2|w_1) + \cdots + logp(w_n|w_{n-1}))$

TEST CASE OF PART 4

Important!

Since my computer's capacity is not enough, it has been tried with a smaller text than normal text.

Searhed Text is: “gelen onlu”

1-gram Frequency Table of corpora:

```
bigram_matrix:
[[0.23260145 0.         0.23260145 ... 0.23260145 0.23260145 0.23260145]
 [0.23260145 0.23260145 0.         ... 0.23260145 0.23260145 0.23260145]
 [0.23260145 0.23260145 0.23260145 ... 0.23260145 0.5280236  0.23260145]
 ...
 [0.23260145 0.23260145 0.23260145 ... 0.23260145 0.23260145 0.23260145]
 [0.23260145 0.23260145 0.23260145 ... 0.23260145 0.23260145 0.23260145]
 [0.23260145 0.23260145 0.23260145 ... 0.23260145 0.23260145 0.23260145]]
```

2-gram Frequency Table of corpora:

```
towgram_matrix:
[[0.59344214 0.59344214 0.         ... 0.         0.         0.        ]
 [0.59344214 0.59344214 0.59344214 ... 0.         0.         0.        ]
 [0.59344214 0.59344214 0.59344214 ... 0.         0.         0.        ]
 ...
 [0.59344214 0.59344214 0.59344214 ... 0.         0.         0.        ]
 [0.59344214 0.59344214 0.59344214 ... 0.         0.         0.        ]
 [0.59344214 0.59344214 0.59344214 ... 0.         0.         0.        ]]
---------------------------------
```

3-gram Frequency Table of corpora:

```
---------------------------------
threegram_matrix:
[[0.80379375 0.80379375 0.80379375 ... 0.         0.         0.        ]
 [0.80379375 0.80379375 0.80379375 ... 0.         0.         0.        ]
 [0.80379375 0.80379375 0.80379375 ... 0.         0.         0.        ]
 ...
 [0.80379375 0.80379375 0.80379375 ... 0.         0.         0.        ]
 [0.80379375 0.80379375 0.80379375 ... 0.         0.         0.        ]
 [0.80379375 0.80379375 0.80379375 ... 0.         0.         0.        ]]
```

This searhed text probability and perplexity results for 1-gram,2-gram and 3-gram:

```
 [0.80379375 0.80379375 0.80379375 ..
---------------------------------
string_bigrams:
['ge', 'len', 'on', 'lu']
prob_bigram:
0.30524534840683576
perp_bigram:
3.276053198580391
---------------------------------
string_towgrams:
['ge len', 'len on', 'on lu']
prob_twogram:
0.06315189235262995
perp_twogram:
3.979301225181597
---------------------------------
string_threegrams:
['ge len on', 'len on lu']
prob_threegram:
0.03161942176621316
perp_threegram:
3.1623894959272056
```

Searhed Text is: "çeşitli konferanslarda"

1-Gram Frequency Table of Corpora:

```
bigram_matrix:
[[0.57313869 0.         0.17678788 ... 0.17678788 0.17678788 0.17678788]
 [0.17678788 0.17678788 0.         ... 0.17678788 0.17678788 0.17678788]
 [1.3158431  0.17678788 2.58304297 ... 0.17678788 0.17678788 0.17678788]
 ...
 [0.17678788 0.17678788 0.17678788 ... 0.17678788 0.17678788 0.17678788]
 [0.17678788 0.17678788 0.17678788 ... 0.17678788 0.17678788 0.17678788]
 [0.17678788 0.17678788 0.17678788 ... 0.17678788 0.17678788 0.17678788]]
-----------------------------------
```

2-Gram Frequency Table of Corpora:

```
towgram_matrix:
[[0.52423476 0.52423476 0.         ... 0.         0.         0.        ]
 [0.28269004 0.52423476 0.52423476 ... 0.         0.         0.        ]
 [0.52423476 0.99373041 0.52423476 ... 0.         0.         0.        ]
 ...
 [0.52423476 0.52423476 0.52423476 ... 0.         0.         0.        ]
 [0.52423476 0.52423476 0.52423476 ... 0.         0.         0.        ]
 [0.52423476 0.52423476 0.52423476 ... 0.         0.         0.        ]]
-----------------------------------
```

3-Gram Frequency Table of Corpora:

```
threegram_matrix:
[[0.14681848 0.75485869 0.75485869 ... 0.         0.         0.        ]
 [0.75485869 0.14681848 0.75485869 ... 0.         0.         0.        ]
 [0.75485869 0.75485869 0.78528179 ... 0.         0.         0.        ]
 ...
 [0.75485869 0.75485869 0.75485869 ... 0.         0.         0.        ]
 [0.75485869 0.75485869 0.75485869 ... 0.         0.         0.        ]
 [0.75485869 0.75485869 0.75485869 ... 0.         0.         0.        ]]
```

This searhed text probability and perplexity results for 1-gram,2-gram and 3-gram:

```
---------------------------------
string_bigrams:
['ce', 'sit', 'li', 'kon', 'fe', 'rans', 'lar', 'da']
prob_bigram:
8.252982937348886
perp_bigram:
0.12116831060857992
---------------------------------
string_towgrams:
['ce sit', 'sit li', 'li kon', 'kon fe', 'fe rans', 'rans lar', 'lar da']
prob_twogram:
0.00808698290576088
perp_twogram:
11.120049934573366
---------------------------------
string_threegrams:
['ce sit li', 'sit li kon', 'li kon fe', 'kon fe rans', 'fe rans lar', 'rans lar da']
prob_threegram:
0.00030558891646582237
perp_threegram:
14.846387799176572
```

PART 5

By entering a word, the syllables that can come after that word were calculated by looking at the 1-gram, 2-gram and 3-gram tables. 5 syllables are added after the word.

The word 'Yıllarda' has been studied:

```
---------------------------------
Make sentence 'yıllarda':
For bigram:
yıllarda buyukbirlikte
For towgram:
yıllarda ceningizila
For threegram:
yıllarda cenacenacen
```

```python
def find_max_probable_word_bigram(bigram_matrix, unique_bigrams, word):
    max_prob = 0
    max_word = ""
    count = 0
    index = 0
    for j in range(5):
        for i in range(len(bigram_matrix[unique_bigrams.index(word[len(word)-1])])):
            if bigram_matrix[unique_bigrams.index(word[len(word)-1])][i] > max_prob:
                max_prob = bigram_matrix[unique_bigrams.index(
                    word[len(word)-1])][i]
                index = i

        max_word += unique_bigrams[index]
        word.append(unique_bigrams[index])
        max_prob = 0
    return max_word
```

```python
def find_max_probable_word_twogram(twogram_matrix, unique_twograms, unique_bigrams, word):
    max_prob = 0
    max_word = ""
    count = 0
    index = 0
    for j in range(5):
        for i in range(len(twogram_matrix[unique_twograms.index(word[len(word)-1])])):
            if twogram_matrix[unique_twograms.index(word[len(word)-1])][i] > max_prob:
                max_prob = twogram_matrix[unique_twograms.index(
                    word[len(word)-1])][i]
                index = i

        max_word += unique_bigrams[index]
        word.append(unique_twograms[index])
        max_prob = 0
    return max_word
```

```
def find_max_probable_word_threegram(threegram_matrix, unique_threegrams, unique_bigrams, word):
    max_prob = 0
    max_word = ""
    count = 0
    index = 0
    for j in range(5):
        for i in range(len(threegram_matrix[unique_threegrams.index(word[len(word)-1])])):
            if threegram_matrix[unique_threegrams.index(word[len(word)-1])][i] > max_prob:
                max_prob = threegram_matrix[unique_threegrams.index(
                    word[len(word)-1])][i]
                index = i

        max_word += unique_bigrams[index]
        word.append(unique_threegrams[index])
        max_prob = 0
    return max_word
```

In the algorithm, the syllable with the highest probability that can come after each syllable was found.

NOTES

The program does not work on very small texts. Error while importing logarithm.

I added the text file I was working on into the zip file.

It is “text.txt”