CSE654

NATURAL LANGUAGE PROGRAMMING

HW3

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']

```
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']

```python
# 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.

```python
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
```

```python
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
```

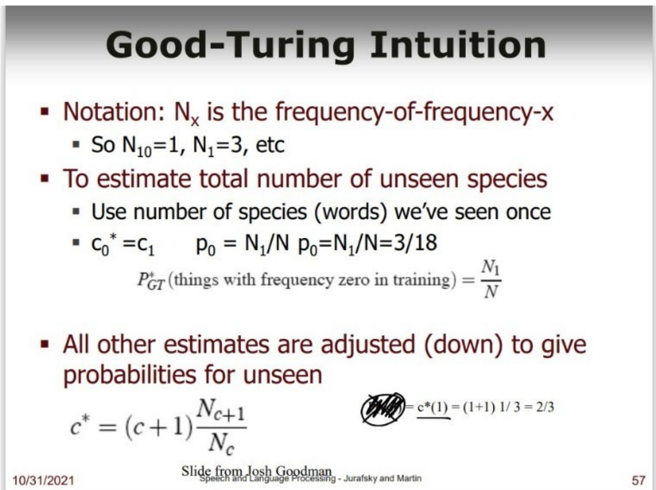
```
def generate_threegram_matrix(unique_threegrams, threegrams, unique_bigrams):
    threegram_matrix = np.zeros((len(unique_threegrams), len(unique_bigrams)))
    for i in range(len(threegrams)-1):
        threegram_matrix[unique_threegrams.index(
            threegrams[i])][unique_bigrams.index(parse_string_three(threegrams[i+1]))]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

Vectors are assigned for 1-gram, 2-gram and 3-gram models.

Gensim library is used to assign vectors.

First, the text read from the file is divided into sentences. The sent_tokenize function of the nltk library was used to separate them into sentences. Then the sentences are divided into words. The word_tokenize function of the nltk library was used to separate it into words. Afterwards, words were divided into syllables and an n-gram model was created using syllables. The text file was trained using the n-gram models created. The created vectors were saved in separate text files for each n-gram model.

For 1-gram: “model1.txt” was created

For 2-gram: “model2.txt” was created

For 3-gram: “model3.txt” was created

For 1-gram:

```
uni_sentences = []
cores = multiprocessing.cpu_count()

# iterate through each sentence in the file
for i in sent_tokenize(f):
    temp = []

    # tokenize the sentence into words
    for j in word_tokenize(i):
        parsed_words = parse_syllable(j.lower())
        unigram = generate_ngrams(parsed_words, 1)
        for i in unigram:
            temp.append(i)

    uni_sentences.append(temp)


unigram_w2v_model = Word2Vec(min_count=20,
                     window=2,
                     sample=6e-5,
                     alpha=0.03,
                     min_alpha=0.0007,
                     negative=20,
                     workers=cores-1)
unigram_w2v_model.build_vocab(uni_sentences, progress_per=10000)

unigram_w2v_model.train(uni_sentences, total_examples=unigram_w2v_model.corpus_count, epochs=30, report_delay=1)

unigram_w2v_model.init_sims(replace=True)
unigram_w2v_model.wv.save_word2vec_format('model1.txt', binary=False)
```

For 2-gram:

```
two_sentences = []
cores = multiprocessing.cpu_count()

# iterate through each sentence in the file
for i in sent_tokenize(f):
    temp = []

    # tokenize the sentence into words
    for j in word_tokenize(i):
        parsed_words = parse_syllable(j.lower())
        twogram = generate_ngrams(parsed_words, 2)
        for i in twogram:
            temp.append(i)

    two_sentences.append(temp)

twogram_w2v_model = Word2Vec(min_count=20,
                     window=2,
                     sample=6e-5,
                     alpha=0.03,
                     min_alpha=0.0007,
                     negative=20,
                     workers=cores-1)
twogram_w2v_model.build_vocab(two_sentences, progress_per=10000)

twogram_w2v_model.train(two_sentences, total_examples=twogram_w2v_model.corpus_count, epochs=30, report_delay=1)

twogram_w2v_model.init_sims(replace=True)
twogram_w2v_model.wv.save_word2vec_format('model2.txt', binary=False)
```

For 3-gram:

```
three_sentences = []
cores = multiprocessing.cpu_count()

# iterate through each sentence in the file
for i in sent_tokenize(f):
    temp = []

    # tokenize the sentence into words
    for j in word_tokenize(i):
        parsed_words = parse_syllable(j.lower())
        threegram = generate_ngrams(parsed_words, 3)
        for i in threegram:
            temp.append(i)

    three_sentences.append(temp)

threegram_w2v_model = Word2Vec(min_count=20,
                     window=2,
                     sample=6e-5,
                     alpha=0.03,
                     min_alpha=0.0007,
                     negative=20,
                     workers=cores-1)
threegram_w2v_model.build_vocab(three_sentences, progress_per=10000)

threegram_w2v_model.train(three_sentences, total_examples=threegram_w2v_model.corpus_count, epochs=30, report_delay=1)

threegram_w2v_model.init_sims(replace=True)
threegram_w2v_model.wv.save_word2vec_format('model3.txt', binary=False)
```

The parameters of Word2Vec:

min_count = int - Ignores all words with total absolute frequency lower than this - (2, 100)

window = int - The maximum distance between the current and predicted word within a sentence. E.g. window words on the left and window words on the left of our target - (2, 10)

size = int - Dimensionality of the feature vectors. - (50, 300)

sample = float - The threshold for configuring which higher-frequency words are randomly downsampled. Highly influencial. - (0, 1e-5)

alpha = float - The initial learning rate - (0.01, 0.05)

min_alpha = float - Learning rate will linearly drop to min_alpha as training progresses. To set it: alpha - (min_alpha * epochs) ~ 0.00

negative = int - If > 0, negative sampling will be used, the int for negative specifies how many "noise words" should be drown. If set to 0, no negative sampling is used. - (5, 20)

workers = int - Use these many worker threads to train the model (=faster training with multicore machines)

PART 5

Tests were performed for the trained 1-gram, 2-gram and 3-gram models. The most similar words have been identified. For this process, the wv.most_similar function of the Gensim library is used.

Results:

```
Most similar 1-gram words of 'ri':
[('rin', 0.5653178691864014), ('riy', 0.5601409077644348), ('ve', 0.5072356462478638), ('nin', 0.46980857849121094), ('ler', 0.4193567633628845), ('le', 0.3885158896446228), ('ni', 0.37572386860847473), ('ye', 0.36647459864616394), ('tum', 0.3591952919960022), ('ci', 0.35564112663269043)]
-------------------
Most similar 2-gram words of 'le ri':
[('le rin', 0.7904993295669556), ('le re', 0.7496670484542847), ('ri ni', 0.723483681678772), ('ri ne', 0.6991896629333496),
('ri nin', 0.6886616945266724), ('le riy', 0.6594251394271851), ('rin den', 0.6292649507522583), ('rin de', 0.577063262462616),
('riy le', 0.5619356632232666), ('di ger', 0.47960644960403444)]
-------------------
Most similar 3-gram words of 'le ri ni':
[('le ri nin', 0.6961844563484192), ('le ri ne', 0.6721827983856201), ('le ri dir', 0.5485888719558716), ('le riy le', 0.465102881193161), ('nim le ri', 0.45228278636932373), ('la ri ni', 0.44923579692840576), ('et me le', 0.4411030411720276), ('on la ri', 0.43123990297317505), ('le dik le', 0.42211806774139404), ('me dik le', 0.4211006462574005)]
-------------------
```

PART 6

Morphology analogy tests between the words were runned. For this process, the wv.similarity function of the Gensim library is used.

```
Similarity between 'ri' and 'rin' for 1-gram words:
0.4380952
Similarity between 'ni' and 'nin' for 1-gram words:
0.55822575
-------------------
Similarity between 'le ri' and 'le rin' for 2-gram words:
0.7904993
Similarity between 'la ri' and 'la rin' for 2-gram words:
0.80741656
-------------------
Similarity between 'le ri ni' and 'le ri nin' for 3-gram words:
0.69618446
Similarity between 'la ri ni' and 'la ri nin' for 3-gram words:
0.7011584
-------------------
```

As can be seen, the morphology analogy analysis of similar words turns out to be similar.

RESOURCES

https://www.kaggle.com/code/pierremegret/gensim-word2vec-tutorial

https://www.geeksforgeeks.org/python-word-embedding-using-word2vec/

https://en.wikipedia.org/wiki/Word_embedding

https://en.wikipedia.org/wiki/Word2vec

https://github.com/ftkurt/python-syllable/blob/master/syllable/syllable.py

https://albertauyeung.github.io/2018/06/03/generating-ngrams.html/