«گزارش کار تمرین شماره ۲»

عنوان تمرین	گزارش تکلیف شماره ۲
عنوان درس	مباحث ویژه در تجارت الکترونیک (پردازش زبان طبیعی – NLP)
نام و نام خانوادگی	امیرحسین خانیکی
شماره دانشجویی	۴۰۱۳۶۱۵۰۰۶
راه ارتباطی	۰۹۱۵۵۱۷۶۹۷۶ و t.me/khaniki_ah

۱) پیش‌پردازش

در این مرحله به منظور پاک‌سازی (پیش‌پردازش) محتوا و جداسازی علائم نگارشی میتوان هم از regex و متد re.sub() استفاده کرد و هم از قابلیت string.punctuation به کمک تابع string.translate() تا هر کاراکتری غیر از حروف و اعداد حذف شوند؛ گفتنی است که translate(str.maketrans('', '', string.punctuation)) این مجموعه از نمادها [!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~]: را حذف می کند. همچنین به منظور ارتقای کیفیت فضاهای اضافی در متن نیز اصلاح شوند.

```
# content preprocessing
def pre_process(content):
    processed_content = content.lower() #=> convert text to lowercase
    processed_content = processed_content.translate(str.maketrans('', '',
string.punctuation)) #=> remove punctuation
    #processed_content = re.sub(r'[^\w\s]', '', processed_content) #=> remove
anything base on regex
    processed_content = ' '.join(processed_content.strip().split()) #=> remove
whitespaces
    #=> above defined functions can also be used such as remove_stopwords,
stem_words, lemmatize_word, and convert_number if needed

    return processed_content
```

همچنین می‌توان از توابع تعریف شده remove_stopwords، stem_words، lemmatize_word و convert_number نیز برای ارتقای فرآیند پیش‌پردازش استفاده کرد.

۲) ساخت n-gramها

برای ایجاد و استخراج n-gram ها از متن مد نظر ابتدا به صورت عادی گام‌های مورد نیاز شامل ساخت توکن (word_tokenize) و بهره‌گیری از nltk.ngrams انجام شد. سپس به منظور بهینه نمودن عملیات به شکل پویا یک تابع با استفاده گام‌های شناسایی شده نوشته و تعریف شد که ورودی‌های آن متن (پاک‌سازی شده) و مقدار یا نوع gram است که به طور مثال 1 نمایانگر uni-gram و 2 معادل bi-gram.

```
# extract n-grams from text/content
def ngram(content, n):
    tokens = word_tokenize(content)
    ngram_result = list(ngrams(tokens, n))
    return ngram_result
```

همچنین برای شمارش n-gram های تکراری تابع count_ngrams تعریف شده است که از متد collections.Counter برای شمارش تکرارها استفاده می‌کند. ورودی آن یک نوع از n-gram ها و خروجی آن یک دیکشنری با کلید عبارت/کلمه و مقدار تعداد تکرار آن است.

```
# count n-grams repeat
def count_ngrams(ngram):
    return ngram_counter(ngram)
```

از سوی دیگر برای دست‌یابی به پر تکرارترین‌ها (top-n) در مجموعه n-gram تابع take_top تعریف شده که از متد ()most_common در دیکشنری برای یافتن n تکرار بالای مجموعه داده استفاده شده است.

```
# take top n of n-gram set
def take_top(ngram_set, n):
    return ngram_set.most_common(n)
```

در انتها از توابع تعریف شده فوق، برای رسیدن به پاسخ استفاده می‌کنیم که به عنوان نمونه برای bi-gram به شرح ذیل است.

```
#tokens = word_tokenize(preprocessed_content)
#bigram = list(ngrams(tokens, 2)) #=> list(bigrams(tokens))
#bigram_count = ngram_counter(bigram)
#bigram_top = bigram_count.most_common(5)
bigram = ngram(preprocessed_content, 2)
bigram_count = count_ngrams(bigram)
bigram_top = take_top(bigram_count, 5)
```

۳) هموارسازی جملات

در بخش سوم، طبق دستورالعمل اعلام شده تابعی برای هموارسازی جملات (smoothing) طبق دو روش هموارسازی Laplace smoothing و Good-Turing smoothing به شرح ذیل تعریف شده است.

```
# smoothing n-gram set
def smooth(ngram, n):
    freqDist = FreqDist(ngram)
    if n == 1:
        laplaceDist = LaplaceProbDist(freqDist, bins = freqDist.N())
        return laplaceDist
    else:
        turingDist = SimpleGoodTuringProbDist(freqDist, bins = freqDist.N())
        return turingDist
```

بر اساس تابع تعریف شده فوق، پیکره و محتوای مدنظر را به طریق زیر هموارسازی می‌کنیم.

```
print(f"Unigram -> smoothing:: {smooth(unigram, 1)}")
print(f"Bigram -> smoothing:: {smooth(bigram, 2)}")
print(f"Trigram -> smoothing:: {smooth(trigram, 3)}")
print(f"Quadgram -> smoothing:: {smooth(quadgram, 4)}")
```

در پاسخ به اینکه چرا Laplace smoothing برای سایر n-gram ها مناسب نیست باید گفت، اصل روش هموارسازی آن بر اساس تخمین احتمالات P است با این فرض که هر نوع کلمه دیده نشده در واقع یک بار رخ داده است که به همین دلیل دارای معایب زیر است:

۱) احتمال n-gram مکرر دست کم گرفته می‌شود.
۲) احتمال n-gram نادر یا دیده نشده بیش از حد برآورد می‌شود.
۳) تمام n-gram های دیده نشده به همین ترتیب هموارسازی می‌شوند.
٤) حجم احتمالی بیش از حد به سمت n-gram نادیده منتقل می‌شود.

۴) پیش‌بینی کلمات

تابع next_word_freq فراوانی کلمه (i+1) را در کل مجموعه و پیکره در جایی که i شاخص جمله یا کلمه است محاسبه می‌کند از سوی دیگر هنگامی عبارت ورودی با جمله در محدوده (i, i+x) منطبق می‌شود و طول آن کمتر از طول بدنه است، کلمه را به فهرست word_list اضافه می‌کند.

```
# calculates the frequency of the (i+1)th word in the whole corpus/set
def next_word_freq(array, sentence):
    sen_len, word_list = len(sentence.split()), []

    for i in range(len(array)):
        if ' '.join(array[i : i + sen_len]).lower() == sentence.lower():
            if i + sen_len < len(array) - 1:
                word_list.append(array[i + sen_len])

    return dict(ngram_counter(word_list)) #=> count of each word in word_list
```

۵) تولید جملات با طول معین

تابع برای تولید عبارت با طول دلخواه تعریف شده است که ورودی آن کلمه است یا می‌تواند متن ورودی دلخواه باشد که پس از طی مراحل پردازشی عبارت تولید شده را برمی‌گرداند. در تولید جملات تابع next_word_freq برای محاسبه فراوانی هر کلمه ورودی در کل مجموعه کلمه یا پیکره فراخوانی می‌شود و از تابع cdf برای محاسبه توزیع تجمعی (CDF) هر کلمه استفاده شده است.

اگر نتیجه توزیع تجمعی معتبر نباشد (یعنی صفر) بدین معنی است که کلمه/جمله وارد شده در مجموعه وجود ندارد و اگر کلمه‌ای که توزیع تجمعی آن بزرگ‌تر یا مساوی با عدد تصادفی مدنظر و کوچک‌تر یا مساوی ۱ است یعنی در مجموعه است. این چرخه تکرار می‌شود تا به محدوده مورد نظر برسیم.

```
# predict next word sentence/word by n-word
def predict_words(entered_phrase, phrase_len, words_number, corpus = ''):
    corpus = default_corpus if corpus == '' else corpus
    l = corpus.split()
    temp_out = ''
    out = entered_phrase + ' '
    for i in range(words_number - phrase_len):
        func_out = next_word_freq(l, entered_phrase)
        cdf_dict = cdf(func_out)
        rand = random.uniform(0, 1)
        try: key, val = zip(*cdf_dict.items())
        except: break
        for j in range(len(val)):
            if rand <= val[j]:
                pos = j
                break
        temp_out = key[pos]
        out = out + temp_out + ' '
        entered_phrase = temp_out
    return out
```

۶) Perplexity

```
train_sentences = ['the best','assume']
tokenized_text = [list(map(str.lower, word_tokenize(sent))) for sent in
train_sentences]
n = 1
train_data, padded_vocab = padded_everygram_pipeline(n, tokenized_text)
model = lm.MLE(n)
model.fit(train_data, padded_vocab)

test_sentences = ['the best', 'between']
tokenized_text = [list(map(str.lower, word_tokenize(sent))) for sent in
test_sentences]

test_data, _ = padded_everygram_pipeline(n, tokenized_text)
for test in test_data:
    print ("MLE estimates:", [((ngram[-1], ngram[:-1]),model.score(ngram[-1],
ngram[:-1])) for ngram in test])
```

```
test_data, _ = padded_everygram_pipeline(n, tokenized_text)

for i, test in enumerate(test_data):
  print(f"Perplexity ({test_sentences[i]}): {model.perplexity(test)}")
```