

C2_W3_lecture_nb_01_corpus_preprocessing

December 14, 2023

N-grams Corpus preprocessing

The input corpus in this week's assignment is a continuous text that needs some preprocessing so that you can start calculating the n-gram probabilities.

Some common preprocessing steps for the language models include: - lowercasing the text - remove special characters - split text to list of sentences - split sentence into list words

Can you note the similarities and differences among the preprocessing steps shown during the Course 1 of this specialization?

```
[ ]: import nltk          # NLP toolkit
import re                # Library for Regular expression operations

nltk.download('punkt')   # Download the Punkt sentence tokenizer
```

Lowercase

Words at the beginning of a sentence and names start with a capital letter. However, when counting words, you want to treat them the same as if they appeared in the middle of a sentence.

You can do that by converting the text to lowercase using `[str.lowercase]` (<https://docs.python.org/3/library/stdtypes.html?highlight=split#str.lower>).

```
[ ]: # change the corpus to lowercase
corpus = "Learning% makes 'me' happy. I am happy be-cause I am learning! :)"
corpus = corpus.lower()

# note that word "learning" will now be the same regardless of its position in
# → the sentence
print(corpus)
```

Remove special charactes

Some of the characters may need to be removed from the corpus before we start processing the text to find n-grams.

Often, the special characters such as double quotes ‘”’ or dash ‘-’ are removed, and the interpunction such as full stop ‘.’ or question mark ‘?’ are left in the corpus.

```
[ ]: # remove special characters
corpus = "learning% makes 'me' happy. i am happy be-cause i am learning! :)"
```

```
corpus = re.sub(r"[^a-zA-Z0-9.?! ]+", "", corpus)
print(corpus)
```

Note that this process gets rid of the happy face made with punctuations :). Remember that for sentiment analysis, this emoticon was very important. However, we will not consider it here.

Text splitting

In the assignment, the sentences in the corpus are separated by a special delimiter `\n`. You will need to split the corpus into an array of sentences using this delimiter. One way to do that is by using the `str.split` method.

The following examples illustrate how to use this method. The code shows: - how to split a string containing a date into an array of date parts - how to split a string with time into an array containing hours, minutes and seconds

Also, note what happens if there are several back-to-back delimiters like between “May” and “9”.

```
[ ]: # split text by a delimiter to array
input_date="Sat May 9 07:33:35 CEST 2020"

# get the date parts in array
date_parts = input_date.split(" ")
print(f"date parts = {date_parts}")

# get the time parts in array
time_parts = date_parts[4].split(":")
print(f"time parts = {time_parts}")
```

This text splitting is more complicated than the tokenization process used for sentiment analysis.

Sentence tokenizing

Once you have a list of sentences, the next step is to split each sentence into a list of words.

This process could be done in several ways, even using the `str.split` method described above, but we will use the NLTK library `nltk` to help us with that.

In the code assignment, you will use the method `word_tokenize` to split your sentence into a list of words. Let us try the method in an example.

```
[ ]: # tokenize the sentence into an array of words

sentence = 'i am happy because i am learning.'
tokenized_sentence = nltk.word_tokenize(sentence)
print(f'{sentence} -> {tokenized_sentence}')
```

Now that the sentence is tokenized, you can work with each word in the sentence separately. This will be useful later when creating and counting N-grams. In the following code example, you will see how to find the length of each word.

```
[ ]: # find length of each word in the tokenized sentence
sentence = ['i', 'am', 'happy', 'because', 'i', 'am', 'learning', '.']
word_lengths = [(word, len(word)) for word in sentence] # Create a list with
↳ the word lengths using a list comprehension
print(f' Lengths of the words: \n{word_lengths}')
```

The previous result produces a list of pairs. This is not equivalent to a dictionary.

N-grams ### Sentence to n-gram

The next step is to build n-grams from the tokenized sentences.

A sliding window of size n-words can generate the n-grams. The window scans the list of words starting at the sentence beginning, moving by a step of one word until it reaches the end of the sentence.

Here is an example method that prints all trigrams in the given sentence.

```
[ ]: def sentence_to_trigram(tokenized_sentence):
    """
    Prints all trigrams in the given tokenized sentence.

    Args:
        tokenized_sentence: The words list.

    Returns:
        No output
    """
    # note that the last position of i is 3rd to the end
    for i in range(len(tokenized_sentence) - 3 + 1):
        # the sliding window starts at position i and contains 3 words
        trigram = tokenized_sentence[i : i + 3]
        print(trigram)

tokenized_sentence = ['i', 'am', 'happy', 'because', 'i', 'am', 'learning', '.']

print(f'List all trigrams of sentence: {tokenized_sentence}\n')
sentence_to_trigram(tokenized_sentence)
```

Prefix of an n-gram

As you saw in the lecture, the n-gram probability is often calculated based on the (n-1)-gram counts. The prefix is needed in the formula to calculate the probability of an n-gram.

$$P(w_n | w_1^{n-1}) = \frac{C(w_1^n)}{C(w_1^{n-1})}$$

The following code shows how to get an (n-1)-gram prefix from n-gram on an example of getting trigram from a 4-gram.

```
[ ]: # get trigram prefix from a 4-gram
fourgram = ['i', 'am', 'happy', 'because']
trigram = fourgram[0:-1] # Get the elements from 0, included, up to the last
    ↪ element, not included.
print(trigram)
```

Start and end of sentence word $\langle s \rangle$ and $\langle e \rangle$ You could see in the lecture that we must add some special characters at the beginning and the end of each sentence: $\ast \langle s \rangle$ at beginning $\ast \langle e \rangle$ at the end

For n-grams, we must prepend n-1 of characters at the beginning of the sentence.

Let us have a look at how you can implement this in code.

```
[ ]: # when working with trigrams, you need to prepend 2 <s> and append one </s>
n = 3
tokenized_sentence = ['i', 'am', 'happy', 'because', 'i', 'am', 'learning', '.']
tokenized_sentence = ["<s>"] * (n - 1) + tokenized_sentence + ["<e>"]
print(tokenized_sentence)
```

That's all for the lab for "N-gram" lesson of week 3.