Theory: Tokenization

22 minutes 0 / 5 problems solved

NLP includes a great variety of procedures. **Tokenization** is one of them. The main task is to split a given sequence of characters into units, called **tokens**. Tokens are usually represented by words, numbers, or punctuation marks. Sometimes, they can be represented by sentences or morphemes (word parts). Tokenization is the first step in text preprocessing and it is a very important procedure; before carrying on with more sophisticated NLP procedures, we need to identify *words* that can help us to interpret the meaning of texts.

Current topic:

Tokenization

Topic depends on:

✓ Indexes Stage 3 7

✕ Regexp quantifiers

✕ Overview of NLTK

Topic is required for:

POS tagging

§1. Some specific issues of tokenization

The major issue of tokenization is choosing the right token. Let's analyze the example below.

This example is rather trivial; we use whitespaces to split the sentence into tokens and get rid of that dot at the end. But sometimes the English language features less obvious cases. What should we do with the apostrophes or a string represented by a combination of numbers and letters?

What is the most suitable token for the example given above? Intuitively, we can say that the first option is what we should go for. Of course, the second option also makes sense as *we're* is the contraction for *we are*. All other options are also theoretically possible.

What if we speak about a city with a complex name, for example: `"New York"` or `["New", "York"]`?

As you can see, choosing the right token may be a tough task. There is no "right answer", so it makes sense to choose a tokenizer that, in your opinion, suits your purposes the best and carry on with it. We will focus on the basics of tokenization in NLTK further.

§2. Tokenization in NLTK

NLTK has a module `tokenize` that consists of different *sub-modules*, the most significant of which we will overview in this section. The chart below contains sub-modules and the results they return. The first column contains the names of tokenizers. To import a particular one, use the construction `from nltk.tokenize import <tokenizer>`. There are some examples of importing.

```
from nltk.tokenize import word_tokenize
from nltk.tokenize import sent_tokenize
```

Syntax	Description
`word_tokenize()`	Return word and punctuation tokens.

`WordPunctTokenizer()`	Return tokens from a string of alphabetic or non-alphabetic characters (like integers, *$*, @...).
`regexp_tokenize()`	Return tokens using standard regular expressions.
`TreebankWordTokenizer()`	Return the tokens as in the Penn Treebank using regular expressions.
`sent_tokenize()`	Return tokenized sentences.

§3. Word tokenization

First, let's create an example for further analysis. Imagine we have a string consisting of three sentences.

```
text = "I have got a cat. My cat's name is C-3PO. He's golden."
```

Now let's have a look at each tokenization way from the table. Don't forget to import all of them in advance.

In the example below, we pass the `text` variable to the `word_tokenize()` method.

```
print(word_tokenize(text))

# ['I', 'have', 'got', 'a', 'cat', '.', 'My', 'cat', "'s", 'name', 'is', 'C-
3PO', '.', 'He', "'s", 'golden', '.']
```

The result is a list of strings (tokens), the function splits the string into words and punctuation marks. Mind the possessives and the contractions in the sentences. The tokenizer transformed all `'s` into separate words, though, as far as we know, `"cat's"` could also be recognized as one token.

The next code snippet introduces the `WordPunctTokenizer()`. This tokenizer is similar to the first one, but the result is a little bit different. All the punctuation marks including dashes and apostrophes are recognized as separate tokens. Now the name of the cat is split into three tokens. In this case, this behavior is not optimal.

```
wpt = WordPunctTokenizer()
print(wpt.tokenize(text))

# ['I', 'have', 'got', 'a', 'cat', '.', 'My', 'cat', "'", 's', 'name', 'is', 'C',
'-', '3PO', '.', 'He', "'", 's', 'golden', '.']
```

The next example shows the results of the `TreebankWordTokenizer()`.

```
tbw = TreebankWordTokenizer()
print(tbw.tokenize(text))
# ['I', 'have', 'got', 'a', 'cat.', 'My', 'cat', "'s", 'name', 'is', 'C-
3PO.', 'He', "'s", 'golden', '.']
```

The `TreebankWordTokenizer()` works almost the same way as the `word_tokenize()` but mind full stops — they form a token with the previous word, the last full stop is a separate token; the `word_tokenize()`, on the contrary, recognizes full stops as separate tokens in all cases. Moreover, the *apostrophes* and the *s* are not separated as in the `WordPunctTokenizer()`.

Let's now move on to the next method. The `regexp_tokenize()` function makes use of regular expressions and accepts two arguments, a string and a pattern for tokens.

```
# 1
print(regexp_tokenize((text), "[A-z]+"))

# ['I', 'have', 'got', 'a', 'cat', 'My', 'cat', 's', 'name', 'is', 'C', 'PO', 'He', 's', 'golden']

# 2
print(regexp_tokenize((text), "[0-9A-z]+"))

# ['I', 'have', 'got', 'a', 'cat', 'My', 'cat', 's', 'name', 'is', 'C', '3PO', 'He', 's', 'golden']

# 3

print(regexp_tokenize((text), "[0-9A-z']+"))


# ['I', 'have', 'got', 'a', 'cat', 'My', "cat's", 'name', 'is', 'C', '3PO', "He's", 'golden']



# 4

print(regexp_tokenize((text), "[0-9A-z'\-]+"))

# ['I', 'have', 'got', 'a', 'cat', 'My', "cat's", 'name', 'is', 'C-3PO', "He's", 'golden']
```

The pattern `"[A-z]+"` in the first example above allows us to find all the words or letters, but it leaves aside integers and the punctuation. Because of that, all the possessive forms and the cat's name are split. The next pattern improves the search for tokens as the integers are added. It improves the search for the name of the cat, but it is not recognized the way it is optimal. The third pattern with an apostrophe also allows the tokenizer to find possessive forms. The last pattern includes the hyphen, so the name of the cat is recognized without mistakes.

You can see that obtaining tokens with the help of regular expressions can be flexible. We change the pattern in each case, and this allows us to get more precise results.

§4. Sentence tokenization

Finally, let's look at the `sent_tokenize()` module. It splits the given string into sentences.

```
print(sent_tokenize(text))
# ['I have got a cat.', "My cat's name is C-3PO.", "He's golden."]
```

However, the sentence tokenization is also a difficult task as a dot, for example, can be used to mark abbreviations or contractions, not exclusively the end of a sentence. Moreover, some dots can indicate both an abbreviation and the end of a sentence. Let's have a look at the examples.

```

text_2 = "Mrs. Beam lives in the U.S.A., it is her motherland. She lost about 9 kilos (20 lbs.) last year."
print(sent_tokenize(text_2))

# ['Mrs. Beam lives in the U.S.A., it is her motherland.', 'She lost about 9 kilos (20 lbs.)', 'last year.']
```

The `sent_tokenize()` includes a list of some instances of typical abbreviations and contractions with dots so they are not recognized as the end of the sentence. But sometimes it still provides confusing results. For example, after tokenizing the `text_2` above, `.)` was recognized as the end of the sentence. It is a mistake.

If you deal with informal texts such as comments, splitting them into sentences may be particularly problematic. For example, in the `text_3`, there are lots of periods and no spaces, so two sentences were recognized as one.

```
text_3 = "The plot of the film is cool!!!!!!! but the characters leave much to be desired....i don't like them."
print(sent_tokenize(text_3))

# ['The plot of the film is cool!!!!!!!', "but the characters leave much to be desired....i don't like them."]
```

§5. Conclusion

To sum up, tokenization is a vitally important procedure for text preprocessing in NLP. In this topic we have learned:

- the main terms and difficulties of tokenization;
- how to split a text into words using different NLTK modules;
- how to split a text into sentences using the `sent_tokenize()` module.

Of course, besides NLTK, there are a lot of other tools for tokenization: spaCy, keras, gensim, etc. You can get acquainted with them by reading their corresponding documentation.

Now it is your time to carry out your tokenization experiments!

Report a typo

21 users liked this theory. **0** didn't like it. **What about you?**

Start practicing

Comments (0) Hints (0) Useful links (0) Show discussion