Theory: Text normalization

20 minutes 0 / 5 problems solved

Skip this topic Start practicing

128 users solved this topic. Latest completion was about 8 hours ago.

In this topic, we are going to learn more about text normalization, one of the steps of text preprocessing. Let's imagine that we have some text and want to count all instances of the verb *play*. Sounds easy, right? What about word forms like *"played"*, *"plays"*, or *"playing"*? They are all forms of one single verb. We can count them manually if our text is short, but with big data, it is just not possible. This is where **text normalization** (or **word normalization)** steps in. The main idea is to reduce different forms of one word to a single form. With this algorithm all forms like *"plays"*, *"playing"*, or *"played"* will be changed to *"play"*.

There are two approaches to text normalization: **stemming** and **lemmatization**. Both are widely used in information retrieval tasks, search engines, topic modeling, and other NLP applications. In the upcoming sections, we will discuss the differences between the approaches, as well as their implementations in the NLTK library.

Note that before stemming or lemmatization, it is better to tokenize your text and get rid of digits and punctuation marks. Otherwise, most algorithms will recognize *"play!"* not as *"play"*, but rather as an unknown word, so it will not be processed correctly.

§1. Stemming

Stem is the most important part of the word, and other word parts (called ***affixes)*** are added to it. For instance, if we take the stem ***play*** and add the affix ***-ed*** to it, we get the past form of the verb ***play***.

In the process of **stemming**, we remove all affixes to get a stem. Note that the result may not represent a real word and it is okay! For example, for the word ***"ladies",*** we may have ***"ladi"***.

Let's see how to carry it out using NLTK. It has different algorithms for stemming and we will learn how to use them. First, we need to import the library:

```
import nltk
```

For the English language, we normally use the Porter stemmer and the Lancaster stemmer. You can find these and some other stemming algorithms in the `nltk.stem` module.

The **Porter stemmer** is the earliest and the most popular algorithm for this task. To use it, we need to import the `PorterStemmer` class from the `nltk.stem` module and then create an object of this class. It is used only for English. After that we call the `stem()` method and put the word in brackets:

```
from nltk.stem import PorterStemmer


porter = PorterStemmer()
porter.stem('played')   # play
porter.stem('playing')  # play
```

The **Snowball stemmer** can be seen as an improvement over the original Porter stemmer as it gives slightly better results. The `SnowballStemmer` class in NLTK also supports 13 non-English languages such as Spanish, French, Russian, German, Swedish, and others. To use this algorithm, we need to create a new instance of the class and specify the language.

Current topic:

Text normalization

Topic depends on:

✓ Methods Stage 1 3★

✗ POS tagging

Topic is required for:

Overview of SpaCy

Table of contents:

```
from nltk.stem import SnowballStemmer


snowball = SnowballStemmer('english')
snowball.stem('playing')  # play
snowball.stem('played')   # play
```

As we said earlier, the Snowball stemmer works better than Porter. Let's compare the examples below:

```
snowball.stem('generously')   # generous
porter.stem('generously')     # gener

snowball.stem('dangerously')  # danger
porter.stem('dangerously')    # danger
```

The Porter stemmer would remove not only the affix "*-ly*" but also "*-ous*" from the input, as it would do for the word "*dangerously*". In this case, it is unnecessary and incorrect. The Snowball stemmer provides a better result for the word "*generously*".

NLTK also has the implementation of the Lancaster or Paice-Husk stemming algorithm. To use the **Lancaster stemmer**, we need to do the same as before, but now we need to import the `LancasterStemmer` class from the `nltk.stem` package:

```
from nltk.stem import LancasterStemmer


lancaster = LancasterStemmer()
lancaster.stem('played')       # play
lancaster.stem('playing')      # play
lancaster.stem('generously' )  # gen
lancaster.stem('dangerously')  # dang
```

All stemmers are quite similar, but the original Porter stemmer and Snowball stemmer provide better results, while the Lancaster stemmer works faster. So, if you are working with really big text data and need to process it in a short time, use Lancaster Stemmer. If you need more accurate results — choose the Snowball or Porter stemmers.

You can learn more about the non-English stemmers available in NLTK on nltk.org.

§2. Lemmatization

Now let's talk about **lemmatization**. Even though it may seem similar to stemming at first sight, there is a difference in how these algorithms work. Stemmers just remove affixes while **lemmatizers** are like people — they analyze the word, its context, its part of speech, and then give the answer. The result is always a real word in its dictionary form called a **lemma**. In general, lemmatizers rely on dictionaries (or corpora) when looking for lemmas.

To use lemmatizer from the NLTK library, you need to make sure that you have access to WordNet — you can do it by typing `nltk.download('wordnet')`.

WordNet is a large lexical database of the English language, which is used for lemmas in the NLTK lemmatizer. There is only one algorithm for lemmatization in NLTK.

We need to import the `WordNetLemmatizer` class from the `nltk.stem` module and create an instance of the class. We use the method `lemmatize()` that takes a word we want to lemmatize as an argument.

```
from nltk.stem import WordNetLemmatizer


lemmatizer = WordNetLemmatizer()
lemmatizer.lemmatize('playing')  # playing
```

As you can see, the word remained unchanged after lemmatization. As far as we know, lemmatizers need to know the context or the part of speech of the word. The default part of speech here is a noun and, as a noun, the word *'playing'* is its own lemma. In the small chart below you can find the most common part-of-speech tags in WordNet:

Part of speech	Tag
Noun	n
Verb	v
Adjective	a

So, we just need to assign the tag that corresponds to the part of speech for our word.

```
lemmatizer.lemmatize('playing', pos='v')  # play
lemmatizer.lemmatize('plays')             # play
```

When we lemmatize a text we cannot manually tag all words, so you need to define a function that will assign a part-of-speech tag to each word.

Note that part-of-speech tags must be the same as in the WordNet! If the tags you got in the result of POS-tagging do not correspond to the ones in the WordNet, you will need to convert them.

§3. Stemming vs. Lemmatization

What should you choose? The answer to this question mainly depends on the task and the language you are dealing with. There is no universal stemmer or lemmatizer for all languages — each language is unique and has specific rules. So you need to use different algorithms with different languages.

For some languages, both stemming and lemmatization give good results, but for others, it is better to opt for lemmatization. For instance, languages like Russian, Latin, Finnish, or Turkish have grammatical cases, meaning that words have different affixes depending on their role in a sentence. Here is an example from Latin: "*rēx respondit*" can be translated as "*the king replied*", and *rēgis fīlia* — as "*the daughter of the king*". Both these words, "*rēx*" and "*rēgis*", are forms of the noun "*rēx*", which stands for "*king*". Cutting off the affixes will give us two different forms, so it is better to apply lemmatization here. Of course, it is possible to use stemming for such languages, but the list of rules what affixes in which cases should be removed is going to be pretty long and complex.

Also, if you need to get valid words after text normalization, go for lemmatization. Sometimes, different forms of one word look completely different, and we just cannot write rules for them. For instance, in English, there are irregular verbs (*be — am*, *is*, *are*), plural forms of nouns (*goose — geese*, *mouse — mice*), and comparative and superlative adjectives (*bad — worse — the worst*). Lemmatizers will detect such cases and give the correct word as a result, while stemmers will not. You can see the example below:

```
#stemming
snowball.stem('worst')                  # worst
snowball.stem('bought')                 # bought
snowball.stem('mice')                   # mice

#lemmatization
lemmatizer.lemmatize('worst', pos='a')   # bad
lemmatizer.lemmatize('bought', pos='v')  # buy
lemmatizer.lemmatize('mice')             # mouse
```

Finally, do not forget about the resources. Lemmatizers usually scan a big dictionary or rely on corpora to find lemmas. It can take a lot of time. If you need to normalize text faster, stemming is the right choice.

Let's sum up the main points of using stemming and lemmatization.

	Stemming	Lemmatization
Pros	• works fast (good for big data) • gives good results for some languages (English) • does not require much memory	• gives a valid word as a result • recognizes cases of suppletion
Cons	• gives as result a stem that may not be a real word	• takes longer to process

§4. Other implementations

NLTK is not the only library that has implementations of text normalization algorithms. Below is a list of libraries where you can find implementations of text normalization for English:

- Hunspell (stemming);
- Gensim (stemming);
- SpaCy (lemmatization);
- TextBlob (lemmatization);
- Pattern (lemmatization).

§5. Summary

In this topic, we have learned about text preprocessing and the role of text normalization in it, the difference between two main approaches (stemming and lemmatization), and how to implement some algorithms using NLTK. Let's recap:

- **Text normalization** is an important step of text preprocessing. It reduces various word forms to one single form.
- There are two approaches to text normalization: **stemming** removes affixes according to some rules and keeps the **stem**, while **lemmatization** analyzes the word and returns its **lemma** with the help of a dictionary.
- Both stemming and lemmatization have their advantages and disadvantages. Stemming works faster than lemmatization but the latter is usually more precise and always returns a real word.