Lemmatization and Stemming

Alec Mccabe Oct 14, 2020 · 5 min read

A Brief Article on the History, Differences and Use-Cases of each Rooting Approach

Stemming and Lemmatization are text preprocessing methods within the field of NLP that are used to standardize text, words, and documents for further analysis. Both in stemming and in lemmatization, we attempt to reduce a given word to its "root". The root word is called a "stem" in the stemming process, and it is called a "lemma" in the lemmatization process.

You may, for instance, want your program to acknowledge that the words "shoot" and "shot" are just different tenses of the same verb. This can be accomplished with either stemming or lemmatization.

However each approach functions a little differently. One approach is not necessarily better than the other, and like most things in the field of Data Science, choosing one method is entirely contextual.

How is Lemmatization different from Stemming

In stemming, a part of the word is just chopped off at the tail end to arrive at the stem of the word. There are definitely different algorithms used to find out how many characters have to be chopped off, but the algorithms don't actually know the meaning of the word in the language it belongs to. In lemmatization, on the other hand, the algorithms have this knowledge. In fact, you can even say that these algorithms refer a dictionary to understand the meaning of the word before reducing it to its root word, or lemma.

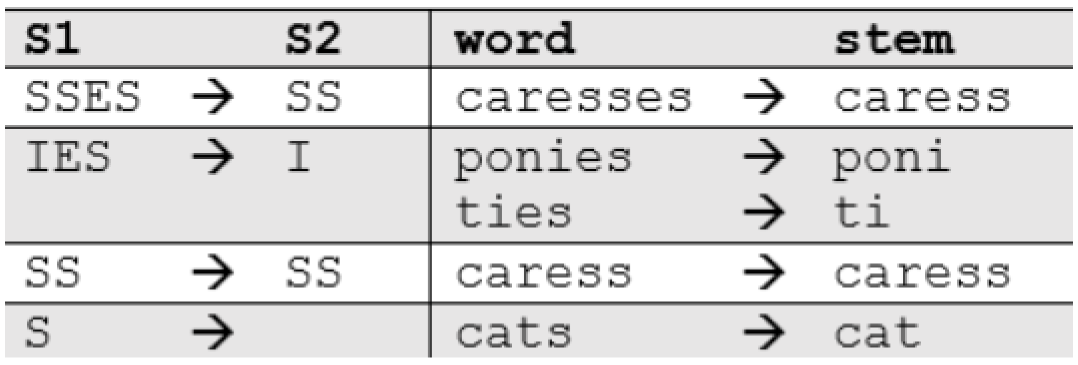
So, a lemmatization algorithm would know that the word *better* is derived from the word *good*, and hence, the lemme is *good*. But a stemming algorithm wouldn't be able to do the same. There could be over-stemming or under-stemming, and the word *better* could be reduced to either *bet*, or *bett*, or just retained as *better*. But there is no way in stemming that it could be reduced to its root word *good*. This, basically is the difference between stemming and lemmatization.

More on Stemming

Stemming is the process of reducing morphological variants of a root/base word to its root. Stemming programs are commonly referred to as stemming algorithms or stemmers.

Often when searching text for a certain keyword, it helps if the search returns variations of the word. For instance, searching for “boat” might also return “boats” and “boating”. Here, “boat” would be the stem for [boat, boater, boating, boats].

Stemming is a somewhat crude method for cataloging related words; it essentially chops off letters from the end until the stem is reached. This works fairly well in most cases, but unfortunately English has many exceptions where a more sophisticated process is required. In fact, spaCy doesn’t include a stemmer, opting instead to rely entirely on lemmatization.



S1		S2	word		stem
SSES	→	SS	caresses	→	caress
IES	→	I	ponies ties	→ →	poni ti
SS	→	SS	caress	→	caress
S	→		cats	→	cat

Porter Stemmer

One of the most common — and effective — stemming tools is Porter’s Algorithm developed by Martin Porter in 1980. The algorithm employs five phases of word reduction, each with its own set of mapping rules. In the first phase, simple suffix mapping rules are defined, such as:

```
# Import the toolkit and the full Porter Stemmer library
import nltk

from nltk.stem.porter import *


p_stemmer = PorterStemmer()


words = ['run','runner','running','ran','runs','easily','fairly']


for word in words:
    print(word+' --> '+p_stemmer.stem(word))
```

Output

```
run → run
runner → runner
running → run
ran → ran
```

```
runs → run
easily → easili
fairly → fairli
```

Snowball Stemmer

This is somewhat of a misnomer, as Snowball is the name of a stemming language developed by Martin Porter. The algorithm used here is more accurately called the “English Stemmer” or “Porter2 Stemmer”. It offers a slight improvement over the original Porter stemmer, both in logic and speed. Since nltk uses the name SnowballStemmer, we’ll use it here.

```
from nltk.stem.snowball import SnowballStemmer

# The Snowball Stemmer requires that you pass a language parameter

s_stemmer = SnowballStemmer(language='english')

words = ['run','runner','running','ran','runs','easily','fairly'

for word in words:
    print(word+' --> '+s_stemmer.stem(word))
```

Output

```
run --> run
runner --> runner
running --> run
ran --> ran
runs --> run
easily --> easili
fairly --> fair
```

In this case, the stemmer performed the same as the Porter Stemmer, with the exception that it handled the stem of “fairly” more appropriately with “fair”

Stemming has its drawbacks. If given the token saw, stemming might always return saw, whereas lemmatization would likely return either see or saw depending on whether the use of the token was as a verb or a noun.

More on Lemmatization

In contrast to stemming, lemmatization looks beyond word reduction and considers a language’s full vocabulary to apply a morphological analysis to words. The lemma of ‘was’ is ‘be’ and the lemma of ‘mice’ is ‘mouse’.

Lemmatization is typically seen as much more informative than simple stemming, which is why Spacy has opted to only have Lemmatization available instead of Stemming.

Lemmatization looks at surrounding text to determine a given word's part of speech, it does not categorize phrases.

```
# Perform standard imports:
import spacy


nlp = spacy.load('en_core_web_sm')def show_lemmas(text):
    for token in text:
        print(f'{token.text:{12}} {token.pos_:{6}} {token.lemma:
<{22}} {token.lemma_}')
```

Here we're using an f-string to format the printed text by setting minimum field widths and adding a left-align to the lemma hash value.

```
doc = nlp(u"I saw eighteen mice today!")

show_lemmas(doc)
```

Output

```
I            PRON   561228191312463089     -PRON-
saw          VERB   11925638236994514241   see
eighteen     NUM    9609336664675087640    eighteen
mice         NOUN   1384165645700560590    mouse
today        NOUN   11042482332948150395   today
!            PUNCT  17494803046312582752   !
```

Notice that the lemma of `saw` is `see`, `mice` is the plural form of `mouse`, and yet `eighteen` is its own number, *not* an expanded form of `eight`.

Because lemmatization is more nuanced than stemming, it requires a little more to actually make work. For lemmatization to resolve a word to its lemma, it needs to know its part of speech. That requires extra computational linguistics power such as a part of speech tagger. This allows it to do better resolutions (like resolving is and are to "be").

Another thing to note about lemmatization is that it's often times harder to create a lemmatizer in a new language than it is a stemming algorithm. Because lemmatizers require a lot more knowledge about the structure of a language, it's a much more intensive process than just trying to set up a heuristic stemming algorithm.

Luckily, if you're working in English, you can quickly use lemmatization through NLTK just like you do with stemming. To get the best results, however, you'll have to feed the part of speech tags to the lemmatizer, otherwise it might not reduce all the words to the lemmas you desire. More than that, it's based off of the WordNet database (which is kind of like a web of synonyms or a thesaurus) so if there isn't a good link there, then you won't get the right lemma anyways.

Advantages and Disadvantages of Lemmatization

As you could probably tell by now, the obvious advantage of lemmatization is that it is more accurate. So if you're dealing with an NLP application such as a chat bot or a virtual assistant where understanding the meaning of the dialogue is crucial, lemmatization would be useful. But this accuracy comes at a cost.

Because lemmatization involves deriving the meaning of a word from something like a dictionary, it's very time consuming. So most lemmatization algorithms are slower compared to their stemming counterparts. There is also a computation overhead for lemmatization, however, in an ML problem, computational resources are rarely a cause of concern.