CMPE 409 Machine Translation

Replacing and Correcting Words

Murat ORHUN

Istanbul Bilgi University

April 6, 2022

Replacing & Correcting Words

- Spell correction
- Text normalization
- Word Replacing
- Word Correcting

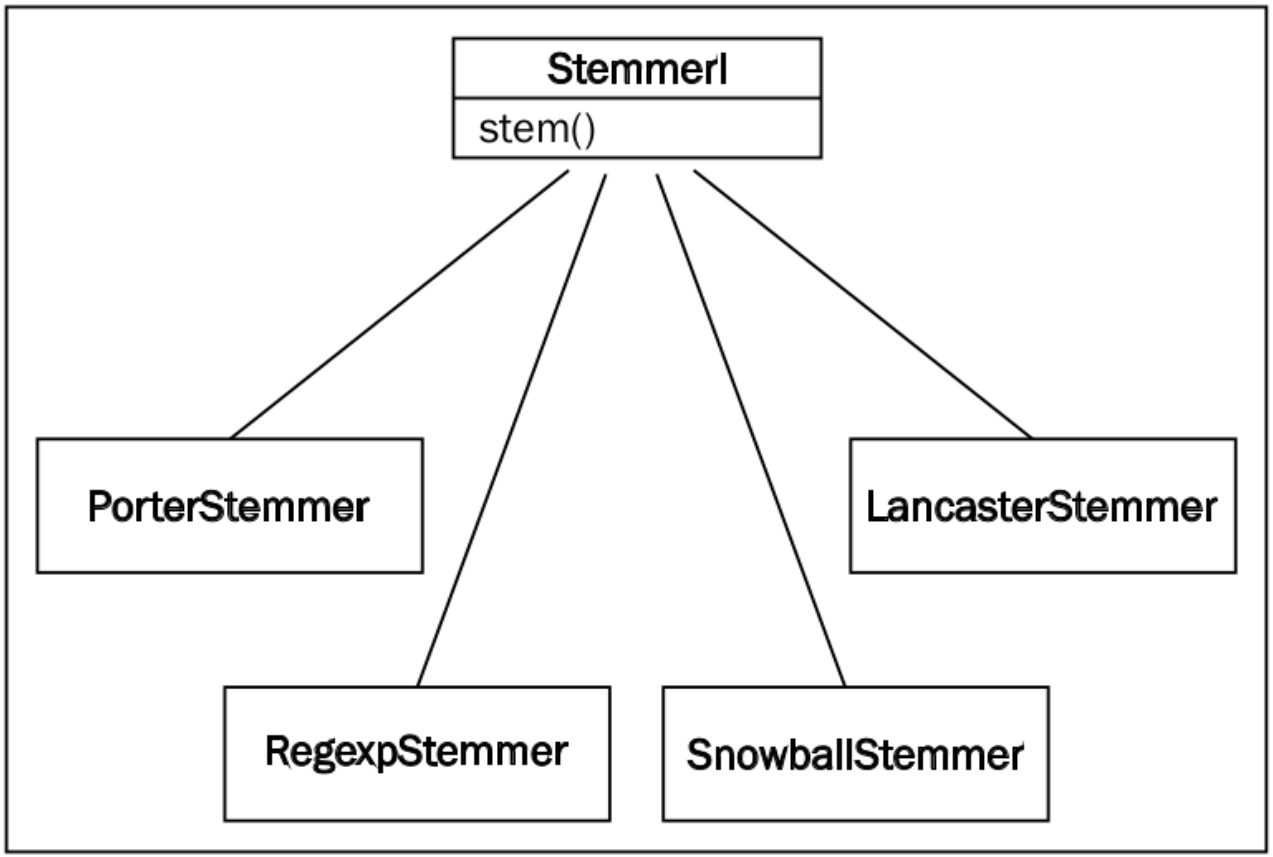
Stemming

- **Stemming** is a technique to remove affixes from a word, ending up with the stem
- Example:

```
cooking    :cook
played     :play
books      :book
cookery    :cookeri (not valid word)
```

- Stemming is most commonly used by search engines for indexing words
- NLTK includes modules

Stemmers


StemmerI
stem()
PorterStemmer
LancasterStemmer
RegexpStemmer
SnowballStemmer

Porter stemming algorithm

It is designed to remove and replace well-known suffixes of English words

```
>>> from nltk.stem import PorterStemmer
>>> stemmer = PorterStemmer()

>>> stemmer.stem('cooking')
>>> stemmer.stem('cookery')
>>> stemmer.stem('books')
>>> stemmer.stem('played')
```

The LancasterStemmer class

It produces slightly different results than Porter.

```
>>> from nltk.stem import LancasterStemmer
>>> stemmer = LancasterStemmer()

>>> stemmer.stem('cooking')
>>> stemmer.stem('cookery')
>>> stemmer.stem('books')
>>> stemmer.stem('played')
```

The RegexpStemmer class

- You can also construct your own stemmer using the RegexpStemmer class.
- It takes a single regular expression iled or as a string) and removes any prefix or suffix that matches the expression.

```
>>> from nltk.stem import RegexpStemmer
>>> stemmer = RegexpStemmer('ing')

>>> stemmer.stem('cooking')
>>> stemmer.stem('cookery')
>>> stemmer.stem('books')
>>> stemmer.stem('played')
>>> stemmer.stem('ingplayed')
```

The SnowballStemmer class

- The SnowballStemmer class supports 13 non-English languages
- It also provides two English stemmers: the original porter algorithm as well as the new English stemming algorithm

```
>>> from nltk.stem import SnowballStemmer
>>> stemmer = SnowballStemmer('english')

>>> stemmer.stem('cooking')
>>> stemmer.stem('cookery')
>>> stemmer.stem('books')
>>> stemmer.stem('played')
>>> stemmer.stem('ingplayed')
```

Lemmatization

- Lemmatization is very similar to stemming, but is more akin to synonym replacement.
- A lemma is a root word, as opposed to the root stem. So unlike stemming, you are always left with a valid word that means the same thing. However, the word you end up with can be completely different.
- NLTK has this module under the *wordnet* package.

Lemmatization

```
>>> from nltk.stem import WordNetLemmatizer
>>> lemmatizer = WordNetLemmatizer()

>>> lemmatizer.lemmatize('cooking')
>>> lemmatizer.lemmatize('cooking', pos='v')
>>> lemmatizer.lemmatize('cookbooks')
>>> lemmatizer.lemmatize('students')
>>> lemmatizer.lemmatize('slowly')
```

Different Between Stem and Lemma

Stemming algorithms work by cutting off the end or the beginning of the word, taking into account a list of common prefixes and suffixes that can be found in an inflected word.

Form	Suffix	Stem
studies	-es	studi
studying	-ing	study
niñas	-as	niñ
niñez	-ez	niñ

https://blog.bitext.com/what-is-the-difference-between-stemming-and-lemmatization/

Different Between Stem and Lemma

- **Lemmatization**, takes into consideration the morphological analysis of the words.
- it is necessary to have detailed dictionaries which the algorithm can look through to link the form back to its lemma

Form	Morphological information	Lemma
studies	Third person, singular number, present tense of the verb study	study
studying	Gerund of the verb study	study
niñas	Feminine gender, plural number of the noun niño	niño
niñez	Singular number of the noun niñez	niñez

https://blog.bitext.com/what-is-the-difference-between-stemming-and-lemmatization/

Different Between Stem and Lemma

```
>>> from nltk.stem import WordNetLemmatizer
>>> lemmatizer = WordNetLemmatizer()

>>> lemmatizer.lemmatize('believes')
output is: belief

>>> from nltk.stem import PorterStemmer
>>> stemmer = PorterStemmer()

>>> stemmer.stem('believes')
output is: believ
```

Combining stemming with Lemmatization

Stemming and lemmatization can be combined to compress words

```
>>> stemmer.stem('buses')
'buse'
>>> lemmatizer.lemmatize('buses')
'bus'
>>> stemmer.stem('bus')
'bu'
```

Replacement

- Problem with contradictions when tokenizing
- Expand contradictions

```
can't      : can not
would've   : would have
won't      : will not
i'm        : i am
```

Use the: re.sub("regex","replacement", "sourcetxt")

Replacement

Use the: re.sub("regex","replacement", "sourcetxt")

```
import re

re.sub("can\'t", "can not","can't")
re.sub("would\'ve", "would have","would've")
re.sub("won\'t", "will not","won't")
re.sub("i\'m", "i am","i'm")
## using gourping....
re.sub("(\w+)\'ve'", '\g<1> have','should've")
replacement is: "should have"
```

Searching

Use the: re.sub("regex" "sourcetxt")

```
import re
name= ["come","coming","book","booking","playing"]
for i in name:
if re.search("ing$",i):
print (i)
output: All names end with "ed"
```

Searching

Use the: re.sub("regex" "sourcetxt")

```
import re
name= [abjectlya', 'adjuster', 'dejected', 'dejectly',
      'injector', 'majestics']
for i in name:
if re.search("..j..t..$",i):
print (i)
output: (test it)
```

Searching

Use the: re.sub("regex" "sourcetxt")

```
import re
name= ['gold', 'golf', 'hold', 'hole']
for i in name:
if re.search('^[ghi][mno][jlk][def]$',i):
print (i)
output: (test it)
```

Basic regular expression

Operator	Behavior
.	Wildcard, matches any character
^abc	Matches some pattern *abc* at the start of a string
abc$	Matches some pattern *abc* at the end of a string
[abc]	Matches one of a set of characters
[A-Z0-9]	Matches one of a range of characters
ed|ing|s	Matches one of the specified strings (disjunction)
*	Zero or more of previous item, e.g., a*, [a-z]* (also known as *Kleene Closure*)
+	One or more of previous item, e.g., a+, [a-z]+
?	Zero or one of the previous item (i.e., optional), e.g., a?, [a-z]?
{n}	Exactly *n* repeats where *n* is a non-negative integer
{n,}	At least *n* repeats
{,n}	No more than *n* repeats
{m,n}	At least *m* and no more than *n* repeats

Regular expression symbools

Symbol	Function
\b	Word boundary (zero width)
\d	Any decimal digit (equivalent to [0-9])
\D	Any non-digit character (equivalent to [^0-9])
\s	Any whitespace character (equivalent to [\t\n\r\f\v]
\S	Any non-whitespace character (equivalent to [^ \t\n\r\f\v])

Regular expression symbools

Symbol	Function
\w	Any alphanumeric character (equivalent to [a-zA-Z0-9_])
\W	Any non-alphanumeric character (equivalent to [^a-zA-Z0-9_])
\t	The tab character
\n	The newline character

Regular expression methods

import re

- re.search()
- re.sub()
- re.split()
- re.findAll()
- match()
- ...

https://docs.python.org/3/library/re.html

Repeating characters

- gooood!
- fine!!!!!!!!!!!!!!
- looooooooooooooveeee
- ok??????????????????
- cooooooooooool

Backreference is a way to refer to a previously matched group in a regular expression. This will allow us to match and remove repeating characters.

Removing repeating characters

```
import re

class RepeatReplacer(object):
  def __init__(self):
    self.repeat_regexp = re.compile(r'(\w*)(\w)\2(\w*)')
    self.repl = r'\1\2\3'

  def replace(self, word):
    repl_word = self.repeat_regexp.sub(self.repl, word)

    if repl_word != word:
      return self.replace(repl_word)
    else:
      return repl_word
```

Removing repeating characters

- 0 or more starting characters (`\w*`)
- A single character (`\w`) that is followed by another instance of that characte
- 0 or more ending characters (`\w*`)

Removing repeating characters

```
>>> replacer = RepeatReplacer()

>>> replacer.replace('looooove')
outp put is: 'love'

>>> replacer.replace('oooooh')
out put is: 'oh'

>>> replacer.replace('goose')
output is: 'gose'
```

Discussion: goose, look, cook, sweet

- Enchant and dictionary
- AbiWord
- http://www.abisource.com/projects/enchant/
- http://aspell.net/
- http://pythonhosted.org/pyenchant/

Read chapter 2

First Assignment

Announce first assignment

Recourse

- Jacob Perkins, **Python 3 Text Processing with NLTK 3 Cookbook, Packt Publishing**,ISBN: 9781782167853
- Steven Bird, Ewan Klein & Edward Loper, **Natural Language Processing with Python**, O'Reily, June, 2009