Python Natural Language Tool Kit

Chapter 2 – Replacing and Correcting Words

Stemming Words

This is the method of removing affixes from a word ending up with the stem. From cooking we get cook. A search engine stores only the stem. One of the best known algorithms is the Porter Stemming Alorithm (Martin Porter). The resulting stem is not always a valid word. From cookery we get “cookeri” The following implements this algorithm by instanciating the PorterStemmer() class.

>>> from nltk.stem import PorterStemmer #import the class

>>> stemmer = PorterStemmer() #instanciate the class

>>> stemmer.stem('cooking') #call the stem() function

'cook'

>>> stemmer.stem('cookery')

'cookeri

There are a couple of other stemmers all of which inherit from the StemmerI inferface which defines the stem() method. The Porter stemmer is the default choice. But the following implements the **Lancaster Stemmer.**

>>> stemmer = LancasterStemmer()

>>> stemmer.stem('cooking')

'cook'

>>> stemmer.stem('cookery')

'cookery'

>>> stemmer.stem('running')

The regular Expression stemmer should be used only in specific cases, it excludes the regular expression passed to it.

>>> from nltk.stem import RegexpStemmer

>>> stemmer = RegexpStemmer('ing')

>>> stemmer.stem('cooking')

'cook'

>>> stemmer.stem('cookery')

'cookery'

>>> stemmer.stem('ingleside')

'leside'

Lemmatizing Words With Wordnet (05.07.13)

A lemma is a root word not a root stem. So, I think a lemma has the same morpheme.

>>> from nltk.stem import WordNetLemmatizer

>>> lemmatizer = WordNetLemmatizer()

>>> lemmatizer.lemmatize('cooking')

'cooking'

>>> lemmatizer.lemmatize('cooking', pos='v') #part of speech is important. Default is noun.

'cook'

>>> lemmatizer.lemmatize('cookbooks')

'cookbook'

The word ‘cooking’ *I like ‘cooking’* in this context is a noun. There is no word stem from this. But when cooking is a verb then ‘cook’ is the stem.

Difference between Stemming and Lemmatization

>>> from nltk.stem import PorterStemmer

>>> stemmer = PorterStemmer()

>>> stemmer.stem('believes') #stemmer just looks at affixes

'believ'

>>> lemmatizer.lemmatize('believes') #lemmatizer looks at the meaning of the word

'belief'

By returning a lemma, you always get a valid word.

Translating text with Babelfish *(not done)*

Replacing words matching regular expressions

Word replacement can be thought of as *error correction* or text *normalisation.* In the following we will try to expand contractions. Such as replacing “can’t” with “cannot”. Firstly, the following code is a module. This has been saved as “replacers.py”

import re

#define a list of tuples

# 'w' this is a word character such as: letters, digits and underscores.

replacement\_patterns = [

(r'won\'t', 'will not'),

(r'can\'t', 'cannot'),

(r'i\'m', 'i am'),

(r'ain\'t', 'is not'),

(r'(\w+)\'ll', '\g<1> will'),

(r'(\w+)n\'t', '\g<1> not'),

(r'(\w+)\'ve', '\g<1> have'),

(r'(\w+)\'s', '\g<1> is'),

(r'(\w+)\'re', '\g<1> are'),

(r'(\w+)\'d', '\g<1> would') ]

class RegexpReplacer(object):

#\_\_init\_\_ is run when class is instanciated

def \_\_init\_\_(self, patterns=replacement\_patterns):

#this just compiles the replacement\_patterns above

self.patterns = [(re.compile(regex), repl) for (regex, repl) in patterns]

def replace(self, text):

s = text

#iterate throught the list 'replacement\_patterns'

for (pattern, repl) in self.patterns:

#look in 's' for 'pattern' and replace with 'repl'

#rs.subn is like re.sub but returns a tuple with count for the number of matches obtained.

(s, count) = re.subn(pattern, repl, s)

return s

The following is the calling code.

>>> import sys #import sys

>>> sys.path.append('/Users/zurich/Google Drive/NLTK-2.0') #set path to where module is stored

>>> from replacers import RegexpReplacer # RegexpReplacer is the class name

>>> replacer = RegexpReplacer() #instanciate class

>>> replacer.replace("can't is a contraction") #replaces with list element 2

'cannot is a contraction'

>>> replacer.replace("I should've done that thing I didn't do") #’ve is replaced with “have”

'I should have done that thing I did not do'

>>>

This basic process can work with replacing “&” with “and”. And eliminate all occurances of “-“ and replace this with the empty string.

Replacement before tokenization

>>> from nltk.tokenize import word\_tokenize

>>> from replacers import RegexpReplacer

>>> replacer = RegexpReplacer()

>>> word\_tokenize("can't is a contraction")

['ca', "n't", 'is', 'a', 'contraction']

>>> word\_tokenize(replacer.replace("can't is a contraction"))

['can', 'not', 'is', 'a', 'contraction']

Removing repeating characters

In regular expressions, a back reference is a way to refer to a previously matched group.

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):

#self.repeat\_regexp is the regular expression defined in the constructor

#sub(r'\1\2\3', 'loooove')

repl\_word = self.repeat\_regexp.sub(self.repl, word)

#if the return value of the regular expression is not the word then....

if repl\_word != word:

return self.replace(repl\_word)

else:

return repl\_word

Now, this is how this gets called:

>>> from replacers import RepeatReplacer

>>> replacer = RepeatReplacer()

>>> replacer.replace('looooove') 'love'

>>> replacer.replace('oooooh') 'oh'

>>> replacer.replace('goose') 'gose'

Now, this is how the following works: (r'(\w\*)(\w)\2(\w\*)')

* The \w\* part means zero or more starting characters
* Then (\w)\2 means a single character followed by another instance of that character
* (\w\*) means zero or more ending characters.

re.sub (pattern, repl)

Return the string by replacing *pattern* with *repl*. The replacement string is then used to keep all the matched groups, while discarding the backreference to the second group. So the word "looooove" gets split into (l)(o)o(ooove) and then recombined as "loooove", discarding the second "o". This continues until only one "o" remains, when repeat\_regexp no longer matches the string, and no more characters are removed.

***Augmenting this with Word Net (insert this)***

def replace(self, word):

if wordnet.synsets(word):

return word

Return ***leaves*** a particular function

***Need to do more work on this. More work on regular expressions.***

Spelling correction with Enchant (08.07)

The following class was created:

class SpellingReplacer(object):

def \_\_init\_\_(self, dict\_name='en', max\_dist=2):

self.spell\_dict = enchant.Dict(dict\_name)

self.max\_dist=2

def replace(self, word):

if self.spell\_dict.check(word):

return(word)

suggestions = self.spell\_dict.suggest(word)

if suggestions and edit\_distance(word, suggestions[0]) <= self.max\_dist:

return suggestions[0]

else:

return word

And then run with the following code:

>>> from replacers import SpellingReplacer

>>> replacer = SpellingReplacer()

>>> replacer.replace('cookbok')

'cookbook'

The max\_distance variable in the code above is set at 2 which means that the number of character changes that need to be made to correct the word.

***The enchant languages:***

>>> enchant.list\_languages()

['en\_AU', 'en\_GB', 'en\_US', 'en\_ZA', 'en\_CA', 'en']

***Using personal word lists***

The following assumes that you have a file called mywords.txt that had ‘nltk’ on one line. You can then create a dictionary augmented with your own lists as follows:

>>> d = enchant.Dict('en\_US')

>>> d.check('nltk')

False

>>> d = enchant.DictWithPWL('en\_US', 'mywords.txt')

>>> d.check('nltk')

True

Replacing Synonyms (08.07)

This can be done to reduce the vocabularly without losing meaning. Vocabularly reduction can increase collocations (or ngrams).

The python ***dictionary.get(index, default)*** returns a dictionary entry associated with the index. If an index value cannot be found, then the default is returned.

class wordReplacer(object):

def \_\_init\_\_(self, word\_map):

self.word\_map = word\_map

def replace(self, word):

return self.word\_map.get(word, word)

This works as follows:

>>> from replacers import wordReplacer

>>> replacer = WordReplacer({'bday': 'birthday'})

>>> replacer.replace('bday')

'birthday'

>>> replacer.replace('happy') 'happy'

***The following creates a subclass CsvWordReplacer [for a CSV file]***

import csv

class CsvWordReplacer(wordReplacer): #extend ‘wordReplacer’

def \_\_init\_\_(self, fname): #constructor take a file name

word\_map = {} #constructors purpoase is to create a word\_map object

for line in csv.reader(open(fname)):

word, syn = line

word\_map[word] = syn

super(CsvWordReplacer, self).\_\_init\_\_(word\_map) #call the parent

This works as follows:

from replacers import CsvWordReplacer

replacer = CsvWordReplacer('/Users/zurich/Google Drive/NLTK-2.0/synonyms.csv')

replacer.replace('bday')

'birthday'

An example was displayed using YAML (YAML aren’t a markup language) but this is not noted.

Replacing Negations with Antonyms (08.07)

If we have a sentence “Let **not uglify** our code” We can replace this with: “Lets beautify our code” Here is the code below:

from nltk.corpus import wordnet

class AntonymReplacer(object):

def replace(self, word, pos=None):

antonyms = set()

for syn in wordnet.synsets(word, pos=pos):

for lemma in syn.lemmas:

for antonym in lemma.antonyms():

antonyms.add(antonym.name)

if len(antonyms) == 1: #only return antonym if unambiguous

return antonyms.pop() #returns and removes element

else:

return None #if no antonyms or many antonyms return null

def replace\_negations(self, sent):

i, l = 0, len(sent) #l is length of list

words = []

while i < l: #iterate through sent list

word = sent[i] # word is sent list element

if word == 'not' and i+1 < l: #"not","nextword" exists("netword") true

ant = self.replace(sent[i+1]) # get antonym(nextword)

if ant: # check if we could actually geta an antonym

words.append(ant) # append antonym to words list

i += 2 # skip to next potential "not"

continue # returns to beggining of while

words.append(word) # append to word list

i += 1 #increment i

return words #return list of words

*Now, to call the code*

>>> import sys

>>> sys.path.append('/Users/zurich/Google Drive/NLTK-2.0')

>>> replacer.replace('uglify')

>>> from replacers import AntonymReplacer

>>> replacer = AntonymReplacer()

>>> replacer.replace('uglify')

'beautify'

>>> replacer.replace('good')

>>>

>>> sent = ["let's", 'not', 'uglify', 'our', 'code']

>>> replacer.replace\_negations(sent)

["let's", 'beautify', 'our', 'code']

***Multiple inheritence example not included.***