Python Natural Language Tool Kit

Tokenising Sentences

See Sentence.py. The code for this is shown below:

text = open('theroartext.txt','r').read()

#load PunktSentenceTokeniser from pickle and call its tokenize() method

import nltk.data

tokenizer = nltk.data.load('tokenizers/punkt/english.pickle')

lstSentences = tokenizer.tokenize(text)

print(lstSentences)

Word Tokenizing

There are two tokenizers. Both inherit from the TokenizerI interface. The first is the PunktWordTokenizer and then there is the TreebankWordTokenizer.

from nltk.tokenize import PunktWordTokenizer

tokenizer = PunktWordTokenizer()

strExSentence = "Can't is a contraction."

lstWordPunkt = tokenizer.tokenize(strExSentence)

print(lstWordPunkt)

#OUTPUT is ['Can', "'t", 'is', 'a', 'contraction.']

from nltk.tokenize import WordPunctTokenizer

tokenizer = WordPunctTokenizer()

lstWordPunkt = tokenizer.tokenize(strExSentence)

print(lstWordPunkt)

#OUTPUT is ['Can', "'", 't', 'is', 'a', 'contraction', '.']

The book showed how to use regular expressions to tokenise text. But these have not been included.

Stopwords

The following (stopwords.py) shows how to filter out stopwords:

from nltk.corpus import stopwords

english\_stops = set(stopwords.words('english'))

#if called "stopwords.words()" with no argument...results in stopwords for all languages.

#can get a list of words with "stopwords.fileids()"

words = ["Can't", 'is', 'a', 'contraction']

lstFiltered = [word for word in words if word not in english\_stops]

print(lstFiltered)

#OUTPUT: ["Can't", 'contraction']

print(english\_stops)

Synsets

These are words that express the same concept (synonyms). Many words have only one synset but some have several. The following are a few examples of code:

>>> from nltk.corpus import wordnet #import the module

>>> syn = wordnet.synsets('cookbook')[0] #first element of the list. Each list element is a class #instance. There can be multiple list elements returned

>>> print(type(syn))

<class 'nltk.corpus.reader.wordnet.Synset'> #this is the class instance

>>> syn.name #name is an attribute of the class

'cookbook.n.01'

>>> syn.definition #definition is an attribute of the class

'a book of recipes and cooking directions'

>>> syn = wordnet.synsets('book') #the noun “book” has multiple synonyms

>>> syn

[Synset('book.n.01'), Synset('book.n.02'), Synset('record.n.05'), Synset('script.n.01'), Synset('ledger.n.01'), Synset('book.n.06'), Synset('book.n.07'), Synset('koran.n.01'), Synset('bible.n.01'), Synset('book.n.10'), Synset('book.n.11'), Synset('book.v.01'), Synset('reserve.v.04'), Synset('book.v.03'), Synset('book.v.04')]

>>> syn[0].examples #Print an example for the first element

['I am reading a good book on economics']

>>> syn[0].definition #Print a definition for the first element

'a written work or composition that has been published (printed on pages bound together)'

Hypernyms

Synsets are organised in a kind of inheritance tree. ***More abstract terms are known as hypernyms*.**  **More specific terms are known as hyponyms.** You can calculate similarity based on the distance of two words in the hypernym tree.

#book is syn[0]

>>> syn[0].hypernyms() #get more general terms for “book”

[Synset('publication.n.01')]

>>> syn[0].hypernyms()[0].hyponyms() #get more specific terms for “publication”

[Synset('new\_edition.n.01'), Synset('book.n.01'), Synset('volume.n.04'), Synset('impression.n.06'), Synset('republication.n.01'), Synset('tip\_sheet.n.01'), Synset('magazine.n.01'), Synset('reference.n.08'), Synset('collection.n.02'), Synset('reissue.n.01'), Synset('periodical.n.01'), Synset('read.n.01')]

>>> syn[0].root\_hypernyms() #find the root of the tree for “book”

[Synset('entity.n.01')]

>>> syn[0].hypernym\_paths() #shows the path from most general

[[Synset('entity.n.01'), Synset('physical\_entity.n.01'), Synset('object.n.01'), Synset('whole.n.02'), Synset('artifact.n.01'), Synset('creation.n.02'), Synset('product.n.02'), Synset('work.n.02'), Synset('publication.n.01'), Synset('book.n.01')]]

Part of Speech (POS)

You can look up a simplified part of speech tag using “pos” after a synset. The following results in “n” which means noun.

>>> syn[0]

Synset('book.n.01')

>>> syn[0].pos

'n'

The word “great” can be used as a noun or an adjective. The following shows how to obtain synsets for the different parts of speech:

>>> len(wordnet.synsets('great'))

7

>>> len(wordnet.synsets('great', pos='n'))

1

>>> len(wordnet.synsets('great', pos='a'))

6

Lemma

A ***lemma is the base form.*** From “produced” the lemma is “produce”. A **lexeme** is a unit of meaning and can be more than one word. A lemma is the canonical form. A **synset represents** a group of lemmas that have the same meaning. So, I guess a synset is a unique meaning. A lemma represents a distinct word form. So, a synset can have more than one lemma.

>>> from nltk.corpus import wordnet

>>> syn = wordnet.synsets('cookbook')[0]

>>> lemmas = syn.lemmas

>>> len(lemmas)

2

>>> lemmas

[Lemma('cookbook.n.01.cookbook'), Lemma('cookbook.n.01.cookery\_book')]

>>> syn

Synset('cookbook.n.01')

>>>

>>> lemmas[0].name

'cookbook'

>>> lemmas[1].name

'cookery\_book'

>>> lemmas[0].synset

Synset('cookbook.n.01')

>>> lemmas[1].synset == lemmas[0].synset

***Getting all the lemmas for a synset***

>>> syn = wordnet.synsets('cookbook')[0]

>>> syn

Synset('cookbook.n.01')

>>> syn.lemmas

[Lemma('cookbook.n.01.cookbook'), Lemma('cookbook.n.01.cookery\_book')]

>>> [lemma.name for lemma in syn.lemmas]

['cookbook', 'cookery\_book']

***What is a synset?***

A synset (or synonym set) is a collection of synonymous words (or lemmas). So, the synset of a motorcar is car. The “car” is the base word. The following shows that the word “book” can have many different meanings. A book can mean “a record of commercial accounts” Or, “a written work or composition”. The following shows an example:

>>> for syn in wordnet.synsets('book'):

print(syn.name + '\t' + syn.definition)

book.n.01 a written work or composition that has been published (printed on pages bound together)

book.n.02 physical objects consisting of a number of pages bound together

record.n.05 a compilation of the known facts regarding something or someone

script.n.01 a written version of a play or other dramatic composition; used in preparing for a performance

ledger.n.01 a record in which commercial accounts are recorded

book.n.06 a collection of playing cards satisfying the rules of a card game

book.n.07 a collection of rules or prescribed standards on the basis of which decisions are made

koran.n.01 the sacred writings of Islam revealed by God to the prophet Muhammad during his life at Mecca and Medina

bible.n.01 the sacred writings of the Christian religions

book.n.10 a major division of a long written composition

book.n.11 a number of sheets (ticket or stamps etc.) bound together on one edge

book.v.01 engage for a performance

reserve.v.04 arrange for and reserve (something for someone else) in advance

book.v.03 record a charge in a police register

book.v.04 register in a hotel booker

So, while a synset is a unique meaning. A lemma is a particular word form that represents that meaning. For example, in regards to the koran (koran.n.o1), relevant lemmas are “Koran” “Quran” “al-Quran”).

So, while “book” see above can have many different meanings depending on context, if you don’t care about the context, you can get all related words for the word “book” I don’t think that they synonyms because the meaning can be very different depending on context.

>>> synonyms = []

>>> for syn in wordnet.synsets('book'):

for lemma in syn.lemmas:

synonyms.append(lemma.name)

len(synonyms)

38 #some words are repeated. If we take the set we get 25

***Antonyms***

>>> synGood = wordnet.synset('good.n.02') #get a synset

>>> synGood

Synset('good.n.02')

>>> synGood.lemmas #the lemmas for a synset

[Lemma('good.n.02.good'), Lemma('good.n.02.goodness')]

>>> synGood.lemmas[0].antonyms() #antonyms() returns lemmas

[Lemma('evil.n.03.evil')]

>>> synGood.lemmas[1].antonyms() #return a different antonym

[Lemma('evil.n.03.evilness')]

>>> evil = synGood.lemmas[0].antonyms()[0]

>>> type(evil)

<class 'nltk.corpus.reader.wordnet.Lemma'>

>>> evil.synset #get the synset of a lemma

Synset('evil.n.03')

>>> evil.synset.lemmas #get the lemmas of a synset

[Lemma('evil.n.03.evil'), Lemma('evil.n.03.evilness')]

>>> evil2 = synGood.lemmas[1].antonyms()[0]

>>> evil2.synset

Synset('evil.n.03')

***Calculating WordNet Synset Similarity***

A hypernym has more general meaning. (hyper = beyond). A hyponym has more specific meaning. A “reference book” is the hypernym of “cookbook”. The following shows some code to calculate similarity using Wu-Palmer Similarity:

>>> from nltk.corpus import wordnet

>>> cb = wordnet.synset('cookbook.n.01')

>>> ib = wordnet.synset('instruction\_book.n.01')

>>> cb.wup\_similarity(ib)

0.9166666666666666

This metric calculates the shortest path distance between the two synsets and their common hypernym.

***Finding the shortest path***

>>> cb.hypernyms()[0] #get the hypernym for “cookbook”

Synset('reference\_book.n.01')

>>> ref = cb.hypernyms()[0] #assign the hypernym to a reference

>>> cb.shortest\_path\_distance(ref) #shortestPath(“cookbook”, hypernym(cookbook))

1

>>> ib.shortest\_path\_distance(ref) #shortestPath(“instructionBook”, hypernym(cookbook))

1

>>> cb.shortest\_path\_distance(ib) #shotestPath(“cookbook”, “instructionBook”)

2

***Comparing Verbs (Also showing common hypernyms)***

>>> cook = wordnet.synset('cook.v.01')

>>> bake = wordnet.synset('bake.v.02')

>>> cook.wup\_similarity(bake)

0.75

>>> cook.common\_hypernyms(bake)

[Synset('create\_from\_raw\_material.v.01'), Synset('make.v.03')]

>>>

***Paths and parts of speech***

Most nouns can be traced up to “object” which provides a basis for similarity. But many verbs do not share common hypernyms therefore, Wordnet cannot calcuate simility. For this reason, you cannot calculate similarity between words with different parts of speech.

***Two other methods of similarity: Leacock Chodorow (LCH) & Path Similarity***

>>> cb.path\_similarity(ib) #cookbook compared to instruction book

0.3333333333333333

>>> cb.path\_similarity(dog) #cookbook compared to dog

0.07142857142857142

>>> cb.lch\_similarity(ib) #cookbook compared to instruction book

2.538973871058276

>>> cb.lch\_similarity(dog) #cookbook compared to dog

0.9985288301111273

***Discovering word collocations (ngrams)***

The following does not work very well as it includes punctuation:

>>> from nltk.corpus import webtext

>>> from nltk.collocations import BigramCollocationFinder

>>> from nltk.metrics import BigramAssocMeasures

>>> words = [w.lower() for w in webtext.words('grail.txt')]

>>> bcf = BigramCollocationFinder.from\_words(words)

>>> bcf = BigramCollocationFinder.from\_words(words)

>>> bcf.nbest(BigramAssocMeasures.likelihood\_ratio, 4)

[("'", 's'), ('arthur', ':'), ('#', '1'), ("'", 't')]

***Short Digression (lambda functions)***

filter\_stops = lambda w: len(w) < 3 or w in stopset #define a function

>>> filter\_stops('with')

True #”with” is contained in stop words

>>> filter\_stops('hello') #”hello” is not contained in stop words

False

***Now for the collocation***

>>> from nltk.corpus import stopwords

>>> stopset = set(stopwords.words('english'))

>>> filter\_stops = lambda w: len(w) < 3 or w in stopset

>>> bcf.apply\_word\_filter(filter\_stops)

>>> bcf.nbest(BigramAssocMeasures.likelihood\_ratio, 4)

[('black', 'knight'), ('clop', 'clop'), ('head', 'knight'), ('mumble', 'mumble')]

The BigramCollocationFinder constructs two frequency distributions: one for each word and one for the bigrams. Any filtering functions that are applied, reduce the size of these two Frequency Distributions by eliminating words that do not pass the filter. By using a filtering function to eliminate all words that don’t pass the filter (eliminate words that are one or two characters or words not in the stoplist), we can get a cleaner result.

***Trigrams***

>>> from nltk.collocations import TrigramCollocationFinder

>>> from nltk.metrics import TrigramAssocMeasures

>>> words = [w.lower() for w in webtext.words('singles.txt')]

>>> tcf = TrigramCollocationFinder.from\_words(words)

>>> tcf.apply\_word\_filter(filter\_stops)

>>> tcf.apply\_freq\_filter(3)

>>> tcf.**nbest**(TrigramAssocMeasures.**likelihood\_ratio**, 4) [('long', 'term', 'relationship')]

In Addition to the stopword filter previously applied, we also applied a frequency filter which removed any trigrams that occurred less than three times, this is why we only got one trigram when we asked for four. The **likelihood\_ratio** is ascoring function. There are other such as: **raw\_freq()** Consult the NgramAssocMeasures. In addtion to the **nbest** there are other ways to get ngrams.

***The above is not so clear. Need to do more work understanding this aspect of the library.***