Natural Language Processing

24

LEARNING OBJECTIVES

After studying this chapter, you will be able to:

LO 1	Understand the layout of the NLTK library
LO 2	Learn how to load various types of resources
LO 3	Understand and use NLTK for tokenizing and also examine its source code for deeper understanding
LO 4	Understand and use "stop words"
LO 5	Understand and use the various algorithms on stemming and lemmatizing
LO 6	Understand POS tagging

24.1 Natural Language Processing

24.1.1 Introduction

The NLTK is a complex library. It is useful to have a look at some of its directories and files to get a greater understanding of how this library works. But discussion on the directory structure and source code of the library may potentially break the flow of the ongoing discussion and confuse a beginner. Therefore, this chapter has shaded boxes which contain advanced discussion on the "internals" of the library. A beginner can easily skip all these discussions, without loss of continuity.

NLTK can be a confusing library for a beginner. So it is important that to give a brief outline of the chapter. The topics are covered in the following order:

- Downloading and installing the NLTK library and the NLTK corpora. The important difference between the two is that the NLTK library are Python scripts while the NLTK corpora are basically data files (largely text files).

- Understanding that the term corpora/corpus is a folder at two different places:
    - The nltk_corpora/corpora folder stores the data files while
    - the /Lib/site-packages/nltk/corpus folder stores the scripts used in the NLTK library. So the scripts stored under the /Lib/site-packages/nltk/corpus folder are used to load data (Mostly text) files in the library.
- Understanding and using the nltk.data.load(path_to_file) function. This load() function can be used to load different types of data since this function returns different types of objects depending upon the type of resource loaded. Some common types of resources which may be loaded are: tokenizers, taggers, chunkers and text data.

    Note that if text is loaded, then the load() function returns the loaded text as a string. But in case of tokenizers, taggers and chunkers, the function load() loads an object. The point to be noted is that the load() function may return objects of different types.
- Understanding how tokenizing works in NLTK. Note that there are many different types of tokenizers in NLTK (like RegexpTokenizer, WhitespaceTokenizer, etc.) but they all inherit from one abstract base class called `TokenizerI`. Further all the different types of Tokenizers (Inherited from `TokenizerI`) implement one method tokenize().
- Understanding that there is a wrapper function sent_tokenize(). And another wrapper function word_tokenize(). So you may use these wrapper functions directly as nltk.sent_tokenizer(some_text) or nltk.word_tokenize(some_text). Having a look at the different types of Tokenizers provided by NLTK library. Having a look at the internals of TweetTokenizer class, TreebankWordTokenizer.
- Having a look at the file regexp.py to understand RegexpTokenizer() class. Also understanding that there is also a module level wrapper function regexp_tokenize() which does all the work of tokenizing done by the RegexpTokenizer() class.
- Stopwords: The nltk.corpora has a module stopwords. This module in turn has a function words("language_name"). So you can get the stop words of a particular language in the form of a list by using stopwords.words(language_name). The return of this function is a list of stop words for that particular language.
- Understanding stemmers. There are a large number of stemmers available in NLTK such as Lancaster stemmer, Porter stemmer, Regular expression stemmer and Wordnet stemmer
- Understanding Wordnet. Wordnet has a typical vocabulary and uses concepts like morphology, morpheme etc.
- Understanding lemmatizing, taking the WordNetLemmatizer class as an example. The way this lemmatizer works is (1) Create an object of WordNetLemmatizer class (2) This class provides a method lemmatize(): So use this method to lemmatize.
- Parts of speech (POS) tagging: Here POS tagging is discussed. Now NLTK library provides a library level function called pos_tag(). (Note that library level function means that this function can be directly imported using a statement like: `from nltk import pos_tag`

24.2 Basic Concept of NLTK

This section deals with subjects like installation of NLTK and installation of NLTK data. It also discusses the internals of NLTK corpora.

24.2.1 Installing NLTK

On windows you can install NLTK by the following command on command prompt. (you can also do it on Jupyter by starting the command with exclamation, that is, (!):

```
pip install nltk  # On Command prompt
!pip install nltk  # On Jupyter
```

You can check the location/path of your NLTK installation by using nltk.__path__ on Jupyter as shown as follows:

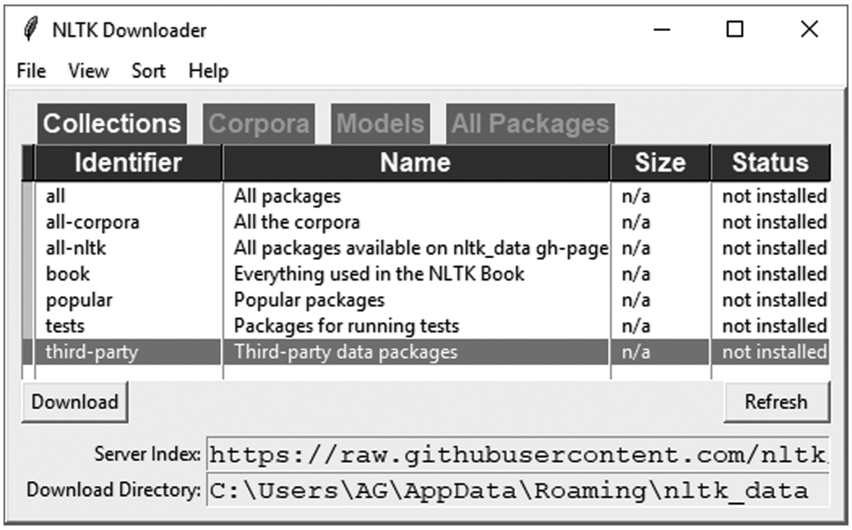
```
import nltk
print(nltk.__path__)
# OUTPUT
['C:\\Users\\AG\\Anaconda3\\lib\\site-packages\\nltk']
```

24.2.2 Installing NLTK Data

After installing NLTK, you need to install some NLTK "copra". Corpus (plural corpora) basically means a body. So NLTK corpora are basically bodies of text. To download NLTK corpora, use the following commands on Jupyter:

```
import nltk
nltk.download()
# OUTPUT
showing info https://raw.githubusercontent.com/nltk/nltk_data/gh-pages/index.xml
```

A new window, as shown in Figure 24.1, should open.



Identifier	Name	Size	Status
all	All packages	n/a	not installed
all-corpora	All the corpora	n/a	not installed
all-nltk	All packages available on nltk_data gh-page	n/a	not installed
book	Everything used in the NLTK Book	n/a	not installed
popular	Popular packages	n/a	not installed
tests	Packages for running tests	n/a	not installed
third-party	Third-party data packages	n/a	not installed

Server Index: https://raw.githubusercontent.com/nltk
Download Directory: C:\Users\AG\AppData\Roaming\nltk_data

FIGURE 24.1 Screen shot of the window that opens when you give command `nltk.download()` to download NLTK data.

Once you download the NLTK data, you also get some books. You can see the downloaded books by the following command:

```
from nltk.book import *
# Output
*** Introductory Examples for the NLTK Book ***
Loading text1, ..., text9 and sent1, ..., sent9
Type the name of the text or sentence to view it.
Type: 'texts()' or 'sents()' to list the materials.
text1: Moby Dick by Herman Melville 1851
text2: Sense and Sensibility by Jane Austen 1811
text3: The Book of Genesis
text4: Inaugural Address Corpus
text5: Chat Corpus
text6: Monty Python and the Holy Grail
text7: Wall Street Journal
text8: Personals Corpus
text9: The Man Who Was Thursday by G . K . Chesterton 1908
```

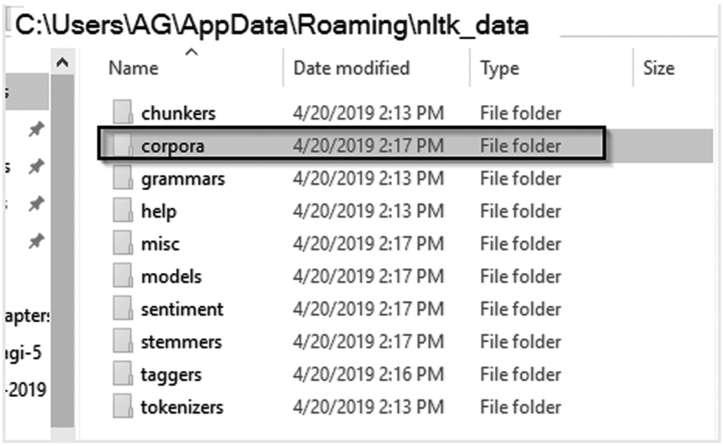
As explained earlier, NLTK corpora is data and not the actual library. It is important to always remember the distinction.

24.3 NLTK Corpora

There are two things relating to a corpus in NLTK. One is the actual "corpora" and the other is the "corpus reader". The "corpora" is "text" while the corpus reader is a Python function which can read a corpora. The corpus reader can read not only the corpora provided by NLTK, but even other corpora. This section deals with some aspects of corpus and corpus readers of the NLTK.

24.3.1 Folder nltk_data

When you use the command nltk.download (), it creates a new folder on your computer with a default name of nltk_data. The path to this folder may vary and on author's system it looks like as shown in Figure 24.2.


C:\Users\AG\AppData\Roaming\nltk_data

Name	Date modified	Type	Size
chunkers	4/20/2019 2:13 PM	File folder	
corpora	4/20/2019 2:17 PM	File folder	
grammars	4/20/2019 2:13 PM	File folder	
help	4/20/2019 2:13 PM	File folder	
misc	4/20/2019 2:17 PM	File folder	
models	4/20/2019 2:17 PM	File folder	
sentiment	4/20/2019 2:17 PM	File folder	
stemmers	4/20/2019 2:17 PM	File folder	
taggers	4/20/2019 2:16 PM	File folder	
tokenizers	4/20/2019 2:13 PM	File folder	

FIGURE 24.2 Screen shot of location of nltk_data folder with various folders like chunkers, corpora, taggers, etc.

24.3.2 NLTK Corpora Usage

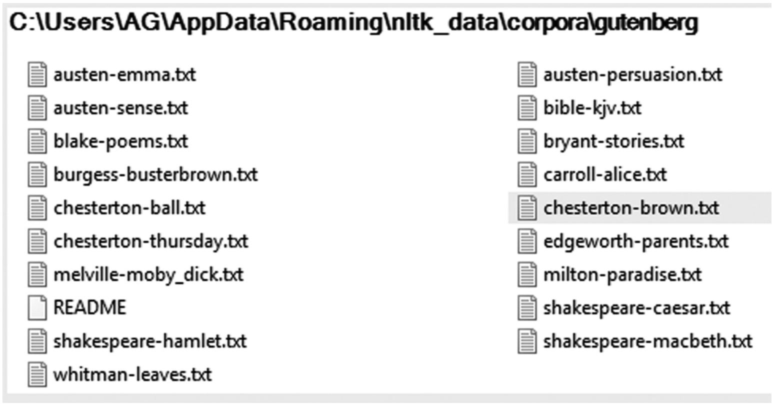
Now when you want to use the corpora package in a NLTK application, you do an import of corpus module available in NLTK library as:

```
from nltk.corpus import corpus_name_to_import
```

Where corpus_name_to_import is one of the corpora available.

Remember the corpus_name_to_import is a directory and each such directory consists of individual files.

The corpora folder in turn has a number of corpus. One such corpus is "gutenberg". The screen shot of the contents of this folder is shown in Figure 24.3.


C:\Users\AG\AppData\Roaming\nltk_data\corpora\gutenberg

austen-emma.txt
austen-sense.txt
blake-poems.txt
burgess-busterbrown.txt
chesterton-ball.txt
chesterton-thursday.txt
melville-moby_dick.txt
README
shakespeare-hamlet.txt
whitman-leaves.txt
austen-persuasion.txt
bible-kjv.txt
bryant-stories.txt
carroll-alice.txt
chesterton-brown.txt
edgeworth-parents.txt
milton-paradise.txt
shakespeare-caesar.txt
shakespeare-macbeth.txt

FIGURE 24.3 Screen shot of contents of the "gutenberg" folder in "corpora" folder of "nltk_data"

24.3.3 Reading the Corpus in NLTK

The NLTK library provides a number of "readers" to read data from a corpus.

You can see the details of the corpus readers on Jupyter as follows (The output is very long and therefore has been truncated to show only relevant parts):

```
import nltk
?nltk.corpus
```

```
NLTK corpus readers.  The modules in this package provide functions that can be
used to read corpus fileids in a variety of formats.  These functions can be used
to read both the corpus file ids that are distributed in the NLTK corpus package,
and corpus fileids that are part of external corpora.

Corpus Reader Functions
=======================
Each corpus module defines one or more "corpus reader functions",
which can be used to read documents from that corpus.  These functions
take an argument, ``item``, which is used to indicate which document
should be read from the corpus:

```

```
15 - If ``item`` is one of the unique identifiers listed in the corpus
16   module's ``items`` variable, then the corresponding document will
17   be loaded from the NLTK corpus package.
18 - If ``item`` is a file id, then that file will be read.
19
20 Additionally, corpus reader functions can be given lists of item
21 names; in which case, they will return a concatenation of the
22 corresponding documents.
23
24 Corpus reader functions are named based on the type of information
25 they return.  Some common examples, and their return types, are:
26
27 - words(): list of str
28 - sents(): list of (list of str)
29 - paras(): list of (list of (list of str))
30 - tagged_words(): list of (str,str) tuple
31 - tagged_sents(): list of (list of (str,str))
32 - tagged_paras(): list of (list of (list of (str,str)))
33 - chunked_sents(): list of (Tree w/ (str,str) leaves)
34 - parsed_sents(): list of (Tree with str leaves)
35 - parsed_paras(): list of (list of (Tree with str leaves))
36 - xml(): A single xml ElementTree
37 - raw(): unprocessed corpus contents
```

Explanation:

Lines 27-37: These are the various functions available for "reading" from a corpora. For example, you have the method words(item_name) which can read "words as a list" from the file "item_name" of a corpora.

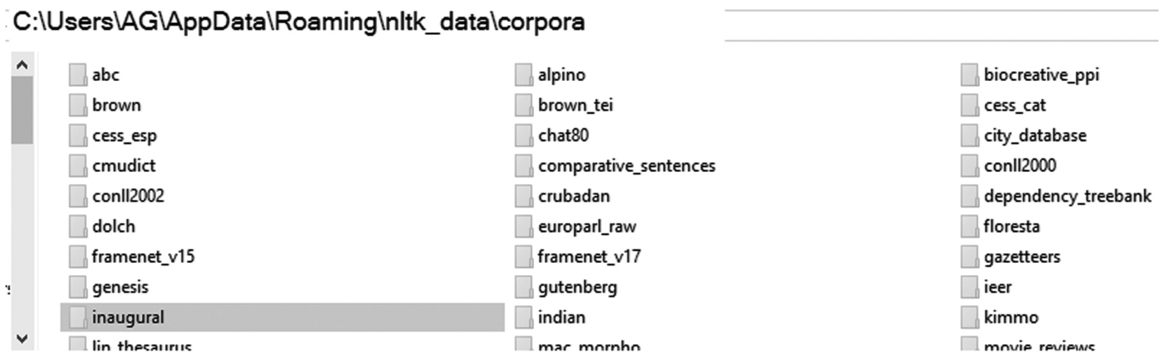
Lines 15-18: These lines specify that you can give two possible attributes to the reader functions like word(item_name).

(1) item_name can be one files of the inbuilt corpora, i.e., corpora provided by NLTK. It can also be a "list of files" in that particular corpora. It can even be left blank ie no attribute. If no attribute is given then the entire corpora is read. If 1 file name is given then only that file from the corpora is read. If a "list of files" in corpora is given, then only those files in the list are read.

(2) You can also give a file path to some text file and in such case the "reader function" like word(file_path) or sent(file_path) will read from the file name given.

So there are two things that you need to know in case you want to use an in-built corpora of NLTK. They are:

- The particular reader function that you want to use like words(), sents(), paras(). Note that you can use this reader function with or without a parameter. If you don't give any parameter, then the entire corpora will be read, but if you give a file name then only that particular file in the corpora will be read. You can even give a "list of files" to the reader functions.
- The name of a particular corpora that you want to use, can be seen in the folder (path_to_nltk_data\nltk_data\corpora). A screen shot of this folder is as shown in Figure 24.4.



C:\Users\AG\AppData\Roaming\nltk_data\corpora

abc	alpino	biocreative_ppi
brown	brown_tei	cess_cat
cess_esp	chat80	city_database
cmudict	comparative_sentences	conll2000
conll2002	crubadan	dependency_treebank
dolch	europarl_raw	floresta
framenet_v15	framenet_v17	gazetteers
genesis	gutenberg	ieer
inaugural	indian	kimmo
lin_thesaurus	mac_morpho	movie_reviews

FIGURE 24.4 Screen shot of various corpora under the corpora directory of nltk_data folder. For example abc, alpino, brown, etc. are all corpora.

Some example scripts are given to clarify the concepts:

```
1  import nltk
2  # --1-- Method words()
3  # Use corpus abc and function words() without any attributes
4  print(nltk.corpus.abc.words())
5  # The folder abc has 2 files in it rural.txt and science.txt
6  # file name science.txt is given as attribute to the method word(‘science.txt’)
7  print(nltk.corpus.abc.words(‘science.txt’))
8  # --2-- Method sents()
9  # Use corpora brown and you can give some of its files as a list
10 # Take 3 files from brown corpora ie ca01, ca02 and ca03
11 some_files_of_brown = [‘ca01’, ‘ca02’, ‘ca03’]
12 print(nltk.corpus.brown.sents(some_files_of_brown))
13 #--3-- Method paras() on corpora genesis giving 1 file of corpora as attribute
14 print(nltk.corpus.genesis.paras(‘english-web.txt’))
15 # OUTPUT - - -(Truncated to save space)
16 [‘PM’, ‘denies’, ‘knowledge’, ‘of’, ‘AWB’, ‘kickbacks’, ...]
17 [‘Cystic’, ‘fibrosis’, ‘affects’, ‘30’, ‘,’, ‘000’, ...]
18 [[‘The’, ‘Fulton’, ‘County’, ‘Grand’, ‘Jury’, ‘said’, ‘Friday’, ‘an’,
19 ‘investigation’, ‘of’,- - - (REST TRUNCATED)
20 [[[‘In’, ‘the’, ‘beginning’, ‘God’, ‘created’, ‘the’, ‘heavens’, ‘and’, ‘the’,
21 ‘earth’, ‘.’], [‘Now’, ‘the’, ‘earth’, (REST TRUNCATED)
```

Explanation:
Line 4: Here the method nltk.corpus.abc.words() is used without giving it any argument. So it will read all the words in all the files in the abc folder.

Line 7: Note that the abc folder has 2 text files. Here one of 2 text files in the abc folder, namely science. txt is given as argument to the words() method, so only the words from this file are read.

Line 11-12: Here some of the files of the folder brown are given as attributes to the method sents(). So the sentences from these specified files only are read.

You can import the corpora from NLTK and then use them also. Following are some more examples of use of NLTK corpora:

```
1  #Import 3 corpora ie brown, movie_review and opinion_lexicon
2  from nltk.corpus import brown, movie_reviews, opinion_lexicon
3  print('brown->', brown.paras('ca01'))
4  #movie_reviews corpus has 2 folders pos and neg
5  #So use parameter /pos/cv000_29590.txt
6  print('movie_reviews->', movie_reviews.sents('pos/cv000_29590.txt'))
7  #Use function without parameter, entire corpora is used
8  print('opinion_lexicon->', opinion_lexicon.words())
9  # Output (Output has been reduced to save space)
10 brown-> [[['The', 'Fulton', 'County', ...
11 movie_reviews-> [['films', 'adapted', ...
12 opinion_lexicon-> ['2-faced', '2-faces', 'abnormal', 'abolish', ...]
```

Explanation:

Line 2: Here you import three copra, i.e., (1) brown, (2) movie_review, and (3) opinion_lexicon.

Line 3: Note that corpora have paras() method and you may give the method a parameter, i.e., ca01. This method will enable the script to "read" paras of the file.

Line 6: The sents() method gives the sentences in the file. Note that the parameter to the method is "pos/cv000_29590.txt". This is because the movie_review corpora has two folders in it, i.e., pos and neg. So you have to specify the relative path name of the file you want to access.

Line 8: Here you use the word() method without giving it any parameter. This is allowed. What NLTK will do is that it will concatenate (i.e., join) the text of all the files in the corpora and read the words in the entire corpora.

24.3.4 Annotated/Tagged Corpora

The text you used in above example was "plain text". In addition NLTK also has samples of what is called "annotated text". Annotation can be considered as "meta data", i.e., data which provides some information about the data. One common "annotation" is "Parts of Speech", i.e., POS tagging.

In the previous section, certain "reader functions" like words(), cents() and paras() were discussed. All these functions are for plaintext. In this section, certain functions for "annotated text" are discussed. However, NLTK provides following three reader functions for "tagged corpora":

```
- tagged_words(): list of (str,str) tuple
- tagged_sents(): list of (list of (str,str))
- tagged_paras(): list of (list of (list of (str,str)))
```

Note that these reader functions cannot be used on all corpora but only on "tagged corpora". Even the tagged corpora may not provide all the three functions but only some of them.

Following example code clarifies the concept:

```
import nltk
print(nltk.corpus.brown.tagged_words())
print(nltk.corpus.brown.tagged_sents())
print(nltk.corpus.brown.tagged_paras())
```

```
# Output (Output has been reduced to save space)
[('The', 'AT'), ('Fulton', 'NP-TL'), ...]
[[('The', 'AT'), ('Fulton', 'NP-TL'), ('County', 'NN-TL'), . . .]]
[[[('The', 'AT'), ('Fulton', 'NP-TL'), ('County', 'NN-TL'). . ]]]
```

The difference in the output of the three functions is obvious. Function tagged_words() gives list of (str,str) tuple, while tagged_sents() gives list of (list of (str,str)) and tagged_paras() gives list of (list of (list of (list of ((str,str))).

24.4 Loading NLTK Resources

In the previous section "reader functions" were discussed. But a reader function can load basically textual data. To load data in other formats, NLTK provides a load() function. This section discusses how to load various types of resources in NLTK so that you may do some processing on it.

24.4.1 nltk.data.load()

NLTK has a module data which has a load function. An edited version of this function on Jupyter is follows:

```
Signature: nltk.data.load(resource_url, format='auto', cache=True, verbose=False,
logic_parser=None, fstruct_reader=None, encoding=None)

While this function has many attributes, only 2 are discussed and the rest can be
ignored. They are:
 -  (1) resource_url: It is of data type string ie str. A URL specifying where
    the resource should be loaded from.  The default protocol is "nltk:", which
    searches for the file in the NLTK data package.
 -  (2) format='auto'. This attribute can accept following data formats
        -  ``pickle``
        -  ``json``
        -  ``yaml``
        -  ``cfg`` (context free grammars)
        -  ``pcfg`` (probabilistic CFGs)
        -  ``fcfg`` (feature-based CFGs)
        -  ``fol`` (formulas of First Order Logic)
        -  ``logic`` (Logical formulas to be parsed by the given logic_parser)
        -  ``val`` (valuation of First Order Logic model)
        -  ``text`` (the file contents as a unicode string)
        -  ``raw`` (the raw file contents as a byte string).
        -  NOTE 1:- If no format is specified, ``load()`` will attempt to
           determine a format based on the resource name's file extension.  If
           that fails, ``load()`` will raise a ``ValueError`` exception.
        -  NOTE 2:- For all text formats (everything except ``pickle``,
           ``json``, ``yaml`` and ``raw``), it tries to decode the raw contents
           using UTF-8, and if that doesn't work, it tries with ISO-8859-1
           (Latin-1), unless the ``encoding`` is specified.
```

Certain things need to be noted:

- You can specify the url from where you want to load a resource.
- You may specify the format of the resource. If you don't specify any format, NLTK will "try" to guess the type of resource you are trying to load. However, if the guess fails, an error will be thrown.
- By default the load() function will load the resource first using UTF-8. If it doesn't succeed, it will try ISO-8859-1

24.4.2 Using load()

Before you can load an in-built resource (i.e., provided by NLTK library), you need to know where it resides. For this, NLTK provides an attribute nltk.data.path. It provides the path to the nltk_data folder on your machine.

It was mentioned earlier that the nltk.data.load() function returns an object whose type depends upon the type of resource loaded. You may write a script which loads different types of resources and then check the "type" of resource loaded. Note that the following script loads a number of different resources like the "punket" tokenizer, the "treebank POS" tagger and the "maxent NE" chunkers. The terms tokenizer, tagger and chunker are discussed later. For the moment, just accept the fact that the load() method can load a number of resources including tokenizers, taggers and chunkers.

```
import nltk.data
# Put your file paths here
p1='file:C:/nltk_data/tokenizers/punkt/english.pickle'
p2='file:C:/nltk_data/taggers/maxent_treebank_pos_tagger/english.pickle'
p3='file:C://nltk_data/chunkers/maxent_ne_chunker/english_ace_binary.pickle'
p4='file:C:/nltk_data/corpora/gutenberg/carroll-alice.txt'
my_tokenizer = nltk.data.load(p1)
my_tagger = nltk.data.load(p2)
my_chunker = nltk.data.load(p3)
my_text = nltk.data.load(p4)
print('tokenizer type->',type(my_tokenizer))
print('tagger type->', type(my_tagger))
print('chunker type->', type(my_chunker))
print('corpora type->', type(my_text))
print(my_text[0:50])#Since my_text is string type so can use it as array
print(dir(my_tokenizer)) #get all methods and attributes of my_tokenizer
```

```
# OUTPUT - - -
tokenizer type-><class 'nltk.tokenize.punkt.PunktSentenceTokenizer'>
tagger type-><class 'nltk.tag.sequential.ClassifierBasedPOSTagger'>
chunker type-><class'nltk.chunk.named_entity.NEChunkParser'>
corpora type-><class'str'>
[Alice's Adventures in Wonderland by Lewis Carroll
['PUNCTUATION', '_Token', '__abstractmethods__', '__class__', '__delattr__',
'__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__',
'__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', '__weakref__', '_abc_cache', '_abc_negative_cache',
'_abc_negative_cache_version', '_abc_registry', '_annotate_first_pass',
'_annotate_second_pass', '_annotate_tokens', '_build_sentence_list',
'_first_pass_annotation', '_lang_vars', '_ortho_heuristic', '_params',
'_realign_boundaries', '_second_pass_annotation', '_slices_from_text',
'_tokenize_words', 'debug_decisions', 'dump', 'sentences_from_text',
'sentences_from_text_legacy', 'sentences_from_tokens', 'span_tokenize',
'span_tokenize_sents', 'text_contains_sentbreak', 'tokenize', 'tokenize_sents',
'train']
```

Lines 7-10: Here you load different types of resources. In lines 7-9, the resources loaded are all files of type .pickle whereas in line 10 it is of type txt. But note that even though three pickle resources are loaded, they create different objects. This is why my_tokenizer is of type PunktSentenceTokenizer, whereas my_tagger is of type ClassifierBasedPOSTagger and so on.

24.5 Tokenizing

Tokenizing is a process whereby a piece of text may be broken up into pieces as per some rule. So breaking up a para into sentences and then breaking up a sentence into words are both tokenizing but with different rules.

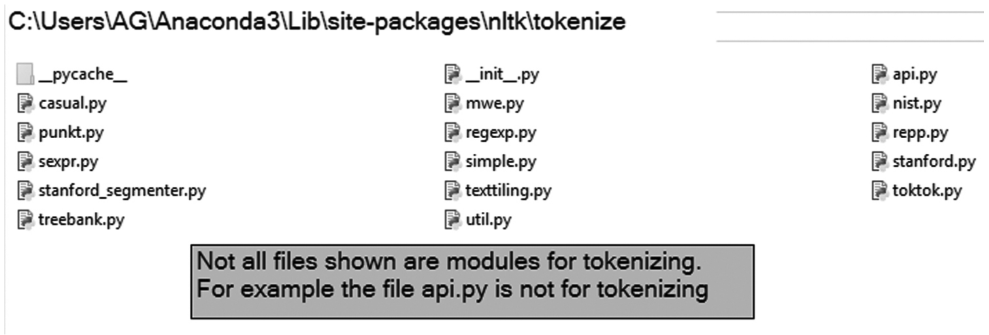
In NLTK, you can tokenize by two different methods:

- By creating a tokenizer object and then using its tokenize method. (You have the option of creating a number of tokenizing object using various tokenizers available like regexp, Treebank, etc.)
- By using some helper functions available. (But you don't have the option of using any tokenizer you want to use. This is because as shown above the sent_tokenize() helper function uses punkt tokenizer and the word_tokenizer() helper function uses the TreebankWordTokenizer() class).

This section discusses how tokenizing works in NLTK.

24.5.1 How the Various Tokenizers are Organized

NLTK offers a large number of tokenizers. Most of them are available as modules, i.e., .py files in the tokenize folder. A screen shot of the contents of the tokenize folder is shown in Figure 24.5.



C:\Users\AG\Anaconda3\Lib\site-packages\nltk\tokenize

__pycache__
casual.py
punkt.py
sexpr.py
stanford_segmenter.py
treebank.py
__init__.py
mwe.py
regexp.py
simple.py
texttiling.py
util.py
api.py
nist.py
repp.py
stanford.py
toktok.py

Not all files shown are modules for tokenizing.
For example the file api.py is not for tokenizing

Figure 24.5 Screen shot of contents of ..\nltk\tokenize folder showing the various tokenizing modules available in NLTK

Table 24.1 gives brief summary of some important tokenizing modules.

TABLE 24.1 Some important tokenizing modules.

Sl. No.	Name	Function
1	casual.py	This tokenizer is for Twitter.
2	moses.py	This is the Moses Tokenizer. It has perhaps been removed from NLTK because of licencing issues. But if you want to use it, it is available on github as part of module sacremoses. You will first have to install sacremoses using pip.
3	mwe.py	This is the MWE, i.e., Multi-Word Expression Tokenizer. (not discussed.)
4	punkt.py	This tokenizer breaks up text into sentences. Note that the PunktSentenceTokenizer is an unsupervised trainable model. What this means is that you can also train it on unlabelled data, i.e., text which has not been split into sentences. So this tokenizer uses algorithm for "sentence boundary detection". You can use pre-trained models or even train your own models. The pre-trained model for English is available at: nltk_data/tokenizers/punkt/english.pickle. Pre-trained models for other languages are also available.
5	regexp.py	This tokenizer uses regular expressions for tokenizing.
6	repp.py	Regular Expression Pre-Processor (REPP). Details not discussed here.
7	sexpr.py	S-Expression tokenizer. (not discussed)
8	simple.py	It has the simple tokenizers which uses the split() method to divide strings into substrings.
9	stanford.py	Stanford tokenizer
10	stanford _segmenter.py	Interface to the Stanford segmenter for Chinese and Arabic. Not discussed
11	texttiling.py	Not discussed
12	toktok.py	The tok-tok tokenizer is used when the input is a single sentence. So only final period is tokenized.
13	treebank.py	This tokenizer uses regular expressions to tokenize text.

24.5.2 Tokenizers of the regexp Module

The regexp.py module has a number of classes and functions for tokenizing. As pointed out earlier, you have the following two ways of tokenizing:

- Create an object of appropriate tokenizing class and then use it for tokenizing.
- Use a helper method.

First, the use of creating a tokenizing class is shown. However, to use a class inside a module, you need to know what classes exist in a module. To find out, there are two methods as follows:

- Use the dir() built-in Python function.
- Examine the source code of file regexp.py

Using the dir() function on regexp, the output on Jupyter is as follows:

```
import nltk
print(dir(nltk.tokenize.regexp))
```

```
# OUTPUT
['BlanklineTokenizer', 'RegexpTokenizer', 'TokenizerI', 'WhitespaceTokenizer',
'WordPunctTokenizer', '__builtins__', '__cached__', '__doc__', '__file__',
'__loader__', '__name__', '__package__', '__spec__', 'blankline_tokenize',
'python_2_unicode_compatible', 're', 'regexp_span_tokenize', 'regexp_tokenize',
'unicode_literals', 'wordpunct_tokenize']
```

From above you can see that:

- Some classes of tokenizers available are: (1) BlanklineTokenizer, (2) RegexpTokenizer, (3) WhiteSpaceTokenizer, and (4) WordPunctTokenizer. Note that class Tokenizer is an abstract base class for other tokenizers and hence cannot be used for tokenizing. (This can be confirmed by having a look at the source code of this class). If you see the source code you will find that BlanklineTokenizer, WhiteSpaceTokenizer and WordPunctTokenizer are all derived from the class RegexpTokenizer. Further this class has a method tokenize() which is most often used. Further since the other three classes have all derived from RegexpTokenizer, hence they all have the tokenize() method available.
- Moreover, you can also see that some functions namely:
  (1) regexp_tokenize(). Note that the function nltk.regexp_tokenize() is similar to re.findall() function of regular expression module.
  (2) blankline_tokenize() and
  (3) wordpunct_tokenize().
- If you see the source code for these three functions, you will find that they are creating objects of the respective classes. For example, the regexp_tokenize() function in turn creates an object of class RegexpTokenizer.

You may see the signature of the RegexpTokenizer class on Jupyter as follows (output has been modified to show only those attributes used for this exercise):

```
import nltk
?nltk.tokenize.regexp.RegexpTokenizer
```

```
Init signature: nltk.tokenize.regexp.RegexpTokenizer(pattern, gaps=False, discard_
empty=True, flags=<RegexFlag.UNICODE|DOTALL|MULTILINE: 56>)

Docstring:
A tokenizer that splits a string using a regular expression, which matches either
the tokens or the separators between tokens.

Parameters (Only 2 parameters ie "pattern" and "gaps" are discussed.)
 -   (1) pattern. It is of type string ie str. It is the pattern used to build
     this tokenizer. This pattern must not contain capturing parentheses;     Use
     non-capturing parentheses, e.g. (?:...), instead)
 -   (2) gaps. It is a bool with default of False. If False then the pattern
     parameter is presumed to be the pattern for finding the tokens. However if
     the gaps parameter is True, then the pattern parameter is used not for
     finding the tokens but for specifying the separators between the tokens
```

Note that you should use non-capturing groups for building this regular expression.

Note on capturing/non-capturing groups: This topic has been discussed in the chapter on regular expressions (re). As a re-cap, recollect that there are three types of brackets used in re. They are:

- Square brackets [] for a character class.
- Curly brackets, i.e., {m, n} for indicating the number of times a character is to be repeated and
- Round brackets, i.e., () for indicating a group. The group may be "capturing" or "non-capturing". To create a non-capturing group you need to have syntax of type (?:   ). A non-capturing group will match the text but ignore it in the final result.

The following script shows how RegexpTokenizer objects can be created and then the tokenize() method used to split some text:

```
from nltk.tokenize.regexp import RegexpTokenizer
pat_for_toks = r'\w+'  # \w+ matches one or more word characters
pat_for_gaps = r'\s+'  # \s+ matches whitespace blank, tab \t, newline \r or \n
text =  "That's one small step for man, one giant leap for mankind."
# gaps is False. So match will take place on tokens
re_toks = RegexpTokenizer(pattern = pat_for_toks, gaps = False)
# gaps is True. So match will take place on seperators
re_gaps = RegexpTokenizer(pattern = pat_for_gaps, gaps = True)
print('with token match->', re_toks.tokenize(text))
print('with gap match->', re_gaps.tokenize(text))
```
```
# OUTPUT
with token match-> ['That', 's', 'one', 'small', 'step', 'for', 'man',
'one', 'giant', 'leap', 'for', 'mankind']
with gap match-> ["That's", 'one', 'small', 'step', 'for', 'man,', 'one',
'giant', 'leap', 'for', 'mankind.']
```

If you want to see the signature of the tokenize() method of the RegexpTokenizer class, you can do so on Jupyter as follows. (Note in order to get to the docstring of the tokenize() method you need to specify the full path, i.e., `nltk.tokenize.regexp.RegexpTokenizer.tokenize`):

```
import nltk
?nltk.tokenize.regexp.RegexpTokenizer.tokenize
```
```
# OUTPUT
Signature: nltk.tokenize.regexp.RegexpTokenizer.tokenize(self, text)
Docstring:
Return a tokenized copy of *s*.
Return type: list of str
```

The following script shows how to use the regexp module classes and methods:

```
1  # Import class RegexpTokenizer
2  from nltk.tokenize import RegexpTokenizer
3  #Create tokenizer objects of type RegexpTokenizer
4  # Give reg exp as parameter to class constructor
5  #\w is [a-zA-Z0-9_] ie lower, upper, digits and _
6  tokObj = RegexpTokenizer("[\w']+")#
7  # [A-Z]\w+ is for those beginning with upper case
8  capTokObj = RegexpTokenizer('[A-Z]\w+')
9  some_text = "Tom uses #smile, while Kate uses @girl"
10 print(tokObj.tokenize(some_text))
11 print(capTokObj.tokenize(some_text))
12 # Output
13 ['Tom', 'uses', 'smile', 'while', 'Kate', 'uses', 'girl']
14 ['Tom', 'Kate']
```

Explanation:

Line 2: Import the RegexpTokenizer class

Line 6: Create a RegexpTokenizer object giving it a re of [\w']+. This is equivalent to [a-zA-Z0-9_], so it will match: a-z (all lowercase letters), A-Z (all uppercase letters), 0-9 (all digits) and _ (an underscore).

Line 8: Here create another RegexpTokenizer object and give it a RE of [A-Z]\w+ which will take only those words which begin with a capital letter.

Line 9: A sample text which has some upper case words and also some beginning with # and @ which are left out by the tokenizer.

24.5.3 regexp_tokenize() Wrapper Function

Many Python libraries and modules provide what are called "wrapper functions". This is done to make the library easy to use. If you go by the traditional long route (i.e., you don't use a wrapper function), then you first create an object of a class in the library and then use the methods of that object.

However it is possible to directly create a class and also use its method in a single step using wrapper functions.

The concept will be clear if you see the source code for the wrapper function regexp_tokenize() in file regexp.py. The relevant portions of source code (along with line numbers of the original file) are as follows:

```
196 # source code function regexp_tokenize()
197 def regexp_tokenize(text, pattern, gaps=False, discard_empty=True,
198                     flags=re.UNICODE | re.MULTILINE | re.DOTALL):
199     """
200     Return a tokenized copy of *text*.  See :class:`.RegexpTokenizer`
201     for descriptions of the arguments.
202     """
203     tokenizer = RegexpTokenizer(pattern, gaps, discard_empty, flags)
204     return tokenizer.tokenize(text)
```

Line 203: You can see that the function is creating an object of type RegexpTokenizer. So even though the user of this function may not be aware, but behind the scene, the wrapper function regexp_tokenize() creates an object of class RegexpTokenizer.

Following is an example of use of both the techniques (ie Method1: create a RegexpTokenizer class and Method2: Use the wrapper function) of using regexp module:

```
1  import nltk
2  from nltk.tokenize import RegexpTokenizer, regexp_tokenize
3  my_string = "New Delhi is capital of India"
4  #Method1 Use RegexpTokenizer() class
5  cap_tokenizer = RegexpTokenizer('[A-Z]\w+') #Uppercase
6  my_tok1 = cap_tokenizer.tokenize(my_string)
7  print(my_tok1)
8  #Method2 Using regexp_tokenize() method
9  my_tok2 = regexp_tokenize(my_string, pattern = '[A-Z]\w+' )
10 print(my_tok2)
11 #Output
12 ['New', 'Delhi', 'India']
13 ['New', 'Delhi', 'India']
```

Line 5: Pattern '[A-Z]\w+' is for selecting words beginning with uppercase letters. Here you create a tokenizer which works on the pattern given to it.

Line 6: Here you use the tokenize() method of the RegexpTokenizer object. Notice that cap_tokenizer is an "object of the RegexpTokenizer class". So you can use the RegexpTokenizer.tokenize() method on it.

Line 9: This line shows the use of "helper function" regexp_tokenize(). Notice that you imported this function also from the module nltk.tokenize. Further note that this function begins with a small letter which as per style guide indicates that it is function and not a class. This helper function takes two attributes, so its signature is: regexp_tokenize(string_to_tokenize, pattern_to_be_used). Notice that if you use this helper function, then you don't need to create an object of class RegexpTokenizer.

24.5.4 NLTK Internals (Source code of regexp.py)

The file regexp.py can general be found under <Path_to_anaconda>\Anaconda3\Lib\site-packages\nltk\tokenize folder. It might be at different locations depending upon your settings or if you are not using Anaconda but some other form of Python.

It is also available on line at https://www.nltk.org/_modules/nltk/tokenize/regexp.html. However, the online source file does not have line numbers and so may be difficult to follow. It is therefore better to either open the local file in a text editor like notepad++ which gives line numbers, or copy and paste from the online source file into a text editor.

The following code follows the line numbers of the source file in the local machine.

The definition of class RegexpTokenizer is in line 78 as shown and it is clear that this class sub-classes, i.e., derives from class TokenizerI

```
78  class RegexpTokenizer(TokenizerI):
```

PP-PSPL_24.indd 132 8/19/2019 3:20:29 PM

The __init__() method of this class is as follows (From line 105 to 114):

```
105     def __init__(self, pattern, gaps=False, discard_empty=True,
106                  flags=re.UNICODE | re.MULTILINE | re.DOTALL):
107         # If they gave us a regexp object, extract the pattern.
108         pattern = getattr(pattern, 'pattern', pattern)
109 
110         self._pattern = pattern
111         self._gaps = gaps
112         self._discard_empty = discard_empty
113         self._flags = flags
114         self._regexp = None
```

The __init__() shows that the method takes four parameters (apart from self) and all except the parameter 'pattern' have default values.

The source code for the tokenize() method of the RegexpTokenizer class is as follows:

```
120     def tokenize(self, text):
121         self._check_regexp()
122         # If our regexp matches gaps, use re.split:
123         if self._gaps:
124             if self._discard_empty:
125                 return [tok for tok in self._regexp.split(text) if tok]
126             else:
127                 return self._regexp.split(text)
128 
129         # If our regexp matches tokens, use re.findall:
130         else:
131            return self._regexp.findall(text)
```

The source code shows that this method works with two options, i.e., whether to tokenize on the tokens or the separators.

The definition of the WhitespaceTokenizer class and its __init__() method are as follows:

```
150 class WhitespaceTokenizer(RegexpTokenizer):
162     def __init__(self):
163         RegexpTokenizer.__init__(self, r'\s+', gaps=True)
```

The __init__() method of the WhitespaceTokenizer class shows that the regexp used is r'\s' which is a well-known regexp for white space.

The definition of the BlanklineTokenizer along with its __init__() method are shown as follows:

```
166 class BlanklineTokenizer(RegexpTokenizer):
167     """
168     Tokenize a string, treating any sequence of blank lines as a delimiter.
169     Blank lines are defined as lines containing no characters, except for
```

```
    space or tab characters.
    """

    def __init__(self):
        RegexpTokenizer.__init__(self, r'\s*\n\s*\n\s*', gaps=True)
```

The __init__() method shows two things: (1) the regexp used is r'\s*\n\s*\n\s*' and (2) the attribute gaps is True. This regexp will find any sequence of blank line as delimiter. Also since gaps has been made true, it will tokenize on the seperators.

The following code shows the definition and __init__() method of the WordPunctTokenizer:

```
class WordPunctTokenizer(RegexpTokenizer):
    """
    Tokenize a text into a sequence of alphabetic and
    non-alphabetic characters, using the regexp ``\w+|[^\w\s]+``.

        >>> from nltk.tokenize import WordPunctTokenizer
          >>> s = "Good muffins cost $3.88\\nin New York.  Please buy me\\ntwo of
them.\\n\\nThanks."
        >>> WordPunctTokenizer().tokenize(s)
        ['Good', 'muffins', 'cost', '$', '3', '.', '88', 'in', 'New', 'York',
        '.', 'Please', 'buy', 'me', 'two', 'of', 'them', '.', 'Thanks', '.']
    """

    def __init__(self):
        RegexpTokenizer.__init__(self, r'\w+|[^\w\s]+')
```

The docstring of the class and also the parameter of the __init__() method show that this class uses the regexp:- r'\w+|[^\w\s]+').

The source code for the three functions namely regexp_tokenize(), blankline_tokenize and wordpunct_tokenize are as follows:

```
def regexp_tokenize(text, pattern, gaps=False, discard_empty=True,
                    flags=re.UNICODE | re.MULTILINE | re.DOTALL):
    """
    Return a tokenized copy of *text*.  See :class:`.RegexpTokenizer`
    for descriptions of the arguments.
    """
    tokenizer = RegexpTokenizer(pattern, gaps, discard_empty, flags)
    return tokenizer.tokenize(text)


blankline_tokenize = BlanklineTokenizer().tokenize
wordpunct_tokenize = WordPunctTokenizer().tokenize
```

From the definition of these three functions, it is quite clear that they in turn are simply creating classes of the respective types. So they are a kind of 'wrapper' functions.

24.5.5 TweetTokenizer of NLTK

NLTK has a module available in a file casual.py for tokenizing tweets.

You can see the docstring of this module on Jupyter as follows:

```
from nltk.tokenize import casual
?casual
```

But suppose you want to see the global variables, classes and functions of this module, you may do so using the dir() function as follows:

```
from nltk.tokenize import casual
print(dir(casual))
```

```
# OUTPUT
['EMOTICONS', 'EMOTICON_RE', 'ENT_RE', 'HANG_RE', 'REGEXPS', 'TweetTokenizer',
'URLS', 'WORD_RE', '__builtins__', '__cached__', '__doc__', '__file__',
'__loader__', '__name__', '__package__', '__spec__', '_replace_html_entities',
'_str_to_unicode', 'casual_tokenize', 'html_entities', 'int2byte', 're',
'reduce_lengthening', 'remove_handles', 'unichr', 'unicode_literals']
```

The above output can be easily understood if you understand and follow the "naming conventions" used in Python. You can easily see that TweetTokenizer is the only class in this module. (It is a class because it follows the CamelCase.)

You can see the docstring of TweetTokenizer class but the docstring is not much useful.

```
from nltk.tokenize import casual
print(casual.TweetTokenizer.__doc__)
```

```
    Tokenizer for tweets.
        >>> from nltk.tokenize import TweetTokenizer
        >>> tknzr = TweetTokenizer()
        >>> s0 = "This is a cooool #dummysmiley: :-) :-P <3 and some arrows < > -
> <--"
        >>> tknzr.tokenize(s0)
        ['This', 'is', 'a', 'cooool', '#dummysmiley', ':', ':-)', ':-P', '<3',
'and', 'some', 'arrows', '<', '>', '->', '<--']
    Examples using `strip_handles` and `reduce_len parameters`:

        >>> tknzr = TweetTokenizer(strip_handles=True, reduce_len=True)
        >>> s1 = '@remy: This is waaaaayyyy too much for you!!!!!!'
        >>> tknzr.tokenize(s1)
        [':', 'This', 'is', 'waaayyy', 'too', 'much', 'for', 'you', '!', '!',
'!']
```

However, if you study the docstring carefully and also examine the source code of the file casual.py, following informations can be drawn:

- The signature of the __init__() method of the TweetTokenizer class is (The __init__() method in source code file starts from around line number 282:

```python
# The __init__() method of TweetTokenizer class
def __init__(self, preserve_case=True, reduce_len=False, strip_handles=False):
    self.preserve_case = preserve_case
    self.reduce_len = reduce_len
    self.strip_handles = strip_handles
```

From above it is clear that the class can accept three optional parameters.

- preserve_case: This parameter has a default value of True. If it is set to False, it will lowercase the tweets except the emoticons. (You can confirm this by seeing the source code of function tokenize(). If you are having difficulty in locating the relevant part of source code, look for the lambda function.)
- strip_handles: This parameter has a default of False, so it does not strip handles. If set to True, it will remove twitter handles.
- reduce_len: This is by default set to False. If True, Replace repeated character sequences of length 3 or greater with sequences of length 3. (You can confirm this by seeing the docstring of function reduce_lengthening() in the source code in file casual.py)

Some example code will clarify the concepts:

```python
from nltk.tokenize.casual import TweetTokenizer
tweet1 = "#ThePSF and @pycon are cooooolest for #Hi ##Guy"
# Create 3 tokenizer objects with different options
tok1 = TweetTokenizer(preserve_case=True, reduce_len=False, strip_handles=False)
tok2 = TweetTokenizer(preserve_case=False, reduce_len=False, strip_handles=True)
tok3 = TweetTokenizer(preserve_case=True, reduce_len= True, strip_handles= True)
print('Using tok1->', tok1.tokenize(tweet1))
print('Using tok2->', tok2.tokenize(tweet1))
print('Using tok3->', tok3.tokenize(tweet1))
```

```
# OUTPUT
Using tok1-> ['#ThePSF', 'and', '@pycon', 'are', 'cooooolest', 'for', '#Hi',
'##Guy']
Using tok2-> ['#thepsf', 'and', 'are', 'cooooolest', 'for', '#hi', '##guy']
Using tok3-> ['#ThePSF', 'and', 'are', 'cooolest', 'for', '#Hi', '##Guy']
```

You can see:

- Setting strip_handles to True removes handles.
- If you set reduce_length to True, then cooooolest becomes coolest.

Following is another example script:

```python
from nltk.tokenize.casual import TweetTokenizer
tTokObj1 = TweetTokenizer(reduce_len = False)
tTokObj2 = TweetTokenizer(preserve_case = False,
                          strip_handles = True, reduce_len = True)
sample_tweet= '@smile @sad are Some Commmmmmmon handles available at abc@xyz.com'

print(tTokObj1.tokenize(sample_tweet))
print(tTokObj2.tokenize(sample_tweet))
```

```
# Output
['@smile', '@sad', 'are', 'Some', 'Commmmmmmon', 'handles', 'available', 'at',
'abc@xyz.com']
['are', 'some', 'common', 'handles', 'available', 'at', 'abc@xyz.com']
```

Line 2: Create a simple TweetTokenizer object using `reduce_len = False and` all other default parameters. Since reduce_len is False, so Commmmmmmon is not reduced in output in line 9

Lines 3-4: Create a TweetTokenizer object giving `preserve_case = False, strip_handles = True, reduce_len = True`

Line 9: The output preserves all inputs and does not reduce the number of 'm' in Commmmmmmon.

Line 10: Tweet handles, upper case are reduced to lower case and the number of m in Commmmmmmon are reduced to 3.

24.5.6 TreebankWordTokenizer

The TreebankWordTokenizer is available as a class in file treebank.py. Its docstring is as follows:

```
from nltk.tokenize.treebank import TreebankWordTokenizer
print(TreebankWordTokenizer.__doc__)
```

```
# Output

    The Treebank tokenizer uses regular expressions to tokenize text as in Penn
Treebank.
    This is the method that is invoked by ``word_tokenize()``.  It assumes that
the
    text has already been segmented into sentences, e.g. using
``sent_tokenize()``.

    This tokenizer performs the following steps:

    - split standard contractions, e.g. ``don't`` -> ``do n't`` and ``they'll`` -
> ``they 'll``
    - treat most punctuation characters as separate tokens
    - split off commas and single quotes, when followed by whitespace
    - separate periods that appear at the end of line

        >>> from nltk.tokenize import TreebankWordTokenizer
        >>> s = '''Good muffins cost $3.88\nin New York.  Please buy me\ntwo of
them.\nThanks.'''
        >>> TreebankWordTokenizer().tokenize(s)
        ['Good', 'muffins', 'cost', '$', '3.88', 'in', 'New', 'York.', 'Please',
'buy', 'me', 'two', 'of', 'them.', 'Thanks', '.']
        >>> s = "They'll save and invest more."
        >>> TreebankWordTokenizer().tokenize(s)
        ['They', "'ll", 'save', 'and', 'invest', 'more', '.']
```

```
        >>> s = "hi, my name can't hello,"
        >>> TreebankWordTokenizer().tokenize(s)
        ['hi', ',', 'my', 'name', 'ca', "n't", 'hello', ',']
```

The docstring of the TreebankWordTokenizer class is not very helpful. But if you examine the source code of the file treebank.py, you can find out some important details as follows:

- The TreebankWordTokenizer does not have an __init__() method, so this means it doesn't take any parameters during construction.

PP-PSPL_24.indd 137 8/19/2019 3:20:29 PM

- The TreebankWordTokenizer has a method tokenize(), whose signature is as follows (Line number in source code file is around 110):

```
110 def tokenize(self, text, convert_parentheses=False, return_str=False):
```

- So the tokenize() method takes three parameters, out of which two have default values. The parameter convert_parantheses, converts round and square brackets to their PTB (Penn Treebank) symbols. For example the PTB symbol for "(" is RRB, i.e., Right Round Bracket. The parameter return_str returns tokens if False and a string if True.

```
from nltk.tokenize.treebank import TreebankWordTokenizer
text1 = 'Hi (Hello) Mr. X [Y] (Z)'
t1 = TreebankWordTokenizer()
print(t1.tokenize(text = text1))
print(t1.tokenize(text = text1, convert_parentheses = True))
print(t1.tokenize(text = text1, convert_parentheses = True, return_str = True))
```

```
# Output
['Hi', '(', 'Hello', ')', 'Mr.', 'X', '[', 'Y', ']', '(', 'Z', ')']
['Hi', '-LRB-', 'Hello', '-RRB-', 'Mr.', 'X', '-LSB-', 'Y', '-RSB-', '-LRB-',
'Z', '-RRB-']
 Hi  -LRB- Hello -RRB-  Mr. X  -LSB- Y -RSB-   -LRB- Z -RRB-
```

24.5.7 TreebankWordDeTokenizer

The treebank module has another class called TreebankWordDeTokenizer which does the reverse of tokenizing, i.e., it joins tokens into a string.

This class also has no __init__() method. However, it has a method detokenize() which does de-tokenizing. You can see the signature of this method on Jupyter as follows (you can also see its __docstring__ attribute):

```
from nltk.tokenize.treebank import TreebankWordDetokenizer
?TreebankWordDetokenizer.detokenize
```

```
# Output Has been modified and truncated

Signature: TreebankWordDetokenizer.detokenize(self, tokens, convert_parentheses =
False)

Parameters
   (1) tokens: This parameter takes a list of strings, i.e. tokenized text in
       form of list of strings ie list(str)
   (2) convert_parantheses. By default it is False. But if it is taken as True,
       then it converts the PTB symbols to parantheses and brackets
Return value:
The method returns a string ie str
```

The following example script will clarify the concept of tokenizing and detokenizing:

```
from nltk.tokenize.treebank import TreebankWordTokenizer, TreebankWordDetokenizer
text1 = 'Hi (Hello) Mr. X [Y] (Z)'
t1 = TreebankWordTokenizer()
tok1 = t1.tokenize(text = text1)
print(tok1)
tok2 = t1.tokenize(text = text1, convert_parentheses = True)
print(tok2)

d1 = TreebankWordDetokenizer()
dtok1 = d1.tokenize(tokens = tok1, convert_parentheses=False)
print(dtok1)
dtok2 = d1.tokenize(tokens = tok2, convert_parentheses= True)
print(dtok2)
# However if during tokenizing convert_parantheses was True, but False in
# detokenizing, then you will get the PTB symbols for brackets and parantheses
dtok3 = d1.tokenize(tokens = tok2, convert_parentheses= False)
print(dtok3)
```

```
# Output
['Hi', '(', 'Hello', ')', 'Mr.', 'X', '[', 'Y', ']', '(', 'Z', ')']
['Hi', '-LRB-', 'Hello', '-RRB-', 'Mr.', 'X', '-LSB-', 'Y', '-RSB-', '-LRB-',
'Z', '-RRB-']
Hi (Hello) Mr. X [Y] (Z )
Hi (Hello) Mr. X [Y] (Z )
Hi -LRB- Hello -RRB- Mr. X -LSB- Y -RSB- -LRB- Z -RRB-
```

Following script is another example of use of TreebankWordTokenizer and TreebankWordDeTokenizer (here a file which is in nltk_data/corpora/gutenberg/carroll-alice.txt has been used):

```
import nltk.data
from nltk.tokenize.treebank import TreebankWordTokenizer, TreebankWordDetokenizer
tbTokObj = TreebankWordTokenizer()
# Use your file path here
p1 = r'file:C:/Users/AG/nltk_data/corpora/gutenberg/carroll-alice.txt'
sample_text = nltk.data.load(p1)
out_text = tbTokObj.tokenize(sample_text[0:50])
print('Tokens->', out_text)
#Use join() to detokenize
new_text = ' '.join(out_text)
print('Detokenize using join()->', new_text)
#Use class TreebankWordDetokenizer(TokenizerI) to detokenize
detokObj = TreebankWordDetokenizer()
detok_text = detokObj.detokenize(out_text)
print('Detokenize using TreebankWordDetokenizer->', detok_text)
```

```
# Output
Tokens-> ['[', 'Alice', "'s", 'Adventures', 'in', 'Wonderland', 'by', 'Lewis',
'Carroll']
Detokenize using join()-> [ Alice 's Adventures in Wonderland by Lewis Carroll
Detokenize using TreebankWordDetokenizer-> [ Alice's Adventures in Wonderland by
Lewis Carroll
```

Explanation:
Line 3: Here you create a TreebankTokenizer object and call it tbTokObj.

Lines 5-6: Here you load some text from the lewis carroll file of the Gutenberg corpora is in string format in a variable sample_text.

Line 7: Here you use the tokenize method of the TreebankTokenizer object and pass the string sample_text to it as a parameter. Note only the first 50 characters of the string are taken.

Line 10: Here you join back the tokens (i.e., detokenize) using the .join() method.

Line 13: Here you create an object of class TreebankWordTokenizer() and call it detokObj.

Line 14: Here you use the detokenize method of the detokObj to rejoin the tokens.

24.5.8 The Wrapper Functions sent_tokenize()

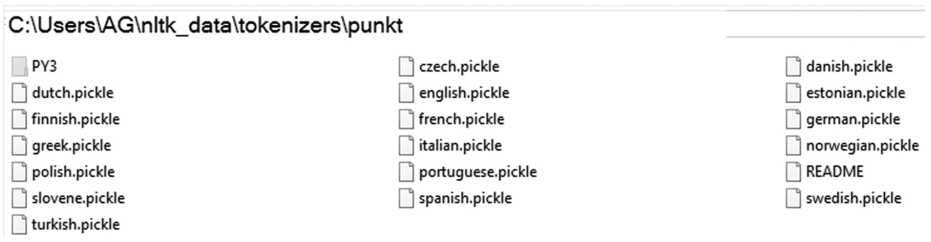
It was mentioned earlier that the regexp module has a wrapper function. There are two other wrapper functions available. They are sent_tokenize() and word_tokenize().

You can see the source code file __init__.py available under the tokenize folder. Note the location of these wrapper functions has also been kept under the tokenize folder so that they become available at the module level.

The source code for the function sent_tokenize() is as follows:

```
83 # Lines 84-95 of __init__.py of tokenize module
84 def sent_tokenize(text, language='english'):
85     """
86     Return a sentence-tokenized copy of *text*,
87     using NLTK's recommended sentence tokenizer
88     (currently :class:`.PunktSentenceTokenizer`
89     for the specified language).
90 
91     :param text: text to split into sentences
92     :param language: the model name in the Punkt corpus
93     """
94     tokenizer = load('tokenizers/punkt/{0}.pickle'.format(language))
95     return tokenizer.tokenize(text)
```

Line 94: You can see that the wrapper function is using the load() function described earlier. Also note that the load() function loads a file of type ".pickle". The file is loaded from folder path <path_to nltk_data>\nltk_data\tokenizers\punkt. The default value for language is "English", so if you don't specify a language, "English" will be used by default. You can load any of the language specific tokenizers given under this folder. The screen shot of this folder is shown in Figure 24.6.



C:\Users\AG\nltk_data\tokenizers\punkt

```
PY3               czech.pickle       danish.pickle
dutch.pickle      english.pickle     estonian.pickle
finnish.pickle    french.pickle      german.pickle
greek.pickle      italian.pickle     norwegian.pickle
polish.pickle     portuguese.pickle  README
slovene.pickle    spanish.pickle     swedish.pickle
turkish.pickle
```

FIGURE 24.6 Screen shot of files under ..\nltk_data\tokenizer\punkt. You can see that tokenizers in many languages are available

PP-PSPL_24.indd 140 8/19/2019 3:20:30 PM

Note that if you use the wrapper function sent_tokenize(), then your script will automatically use the Punkt tokenizer. This is a shortcoming of using this wrapper function. You cant use this wrapper to use other tokenizers.

The following script shows use of the wrapper function sent_tokenizer() using first the default language, i.e., English and then using French.

```
from nltk.tokenize import sent_tokenize
# Opening lines of French National Anthem
fr_anthem = “““Allons enfants de la Patrie,
            Le jour de gloire est arrivé !
            Contre nous de la tyrannie
            L’étendard sanglant est levé, (bis)
            Entendez-vous dans les campagnes
            Mugir ces féroces soldats ?
            Ils viennent jusque dans vos bras
            Égorger vos fils, vos compagnes !”””

# Use sent_tokenizer without specifying language
sent1 =sent_tokenize(text = fr_anthem)
print(sent1)
# Use french for the language attribute
sent2 =sent_tokenize(text = fr_anthem, language = ‘french’)
print(sent2)
```

You can try out other languages to see how this works.

24.5.9 Wrapper Function word_tokenize()

Just like the sent_tokenize() wrapper function, there is also a word_tokenize() wrapper function. This function is also in the __init__.py file under <path to nltk>\nltk\tokenize folder.

The source code of this wrapper function is as follows:

```
# Lines 113- 131 of file __init__.py
def word_tokenize(text, language='english', preserve_line=False):
    “””
    Return a tokenized copy of *text*,
    using NLTK's recommended word tokenizer
    (currently an improved :class:`.TreebankWordTokenizer`
    along with :class:`.PunktSentenceTokenizer`
    for the specified language).

    :param text: text to split into words
    :type text: str
    :param language: the model name in the Punkt corpus
    :type language: str
     :param preserve_line: An option to keep the preserve the sentence and not sentence
tokenize it.
    :type preserver_line: bool
    “””
```

```
129     sentences = [text] if preserve_line else sent_tokenize(text, language)
130     return [token for sent in sentences
131             for token in _treebank_word_tokenizer.tokenize(sent)]
```

Following are relevant in the above code:

- This wrapper function uses the TreebankWordTokenizer. You can see that in line 131 an object of type _treebank_word_tokenizer is created. Further if you see around line 98 of the source code file, you will see the following line which clearly shows that treebankWordTokenizer is being used:

  ```
  97  # Standard word tokenizer.
  98  _treebank_word_tokenizer = TreebankWordTokenizer()
  ```

- This wrapper function internally calls the sent_tokenize() wrapper function before it does word tokenizing. But it does so only if the attribute preserve_lines is True.
- The language parameter passed to this wrapper function is meant only for sentence tokenization and not for word tokenization.
- So this wrapper function does not have capability to do word tokenization based on language. The language parameter given as one of the parameters might make you think that this wrapper function can do word tokenization based on language, but this is not the case.

The following script shows how the wrapper function word_tokenize() is imported from nltk.tokenize and used:

```
# Since word_tokenize() function is available in __init__.py file
# of tokenize module, you can import it from nltk.tokenize
from nltk.tokenize import word_tokenize
some_text = '''The old order changeth yielding place to new
            And God fulfills himself in many ways
            Lest one good custom should corrupt the world.'''
my_toks = word_tokenize(text = some_text)
print(my_toks)
```

```
# OUTPUT
[‘The’, ‘old’, ‘order’, ‘changeth’, ‘yielding’, ‘place’, ‘to’, ‘new’, ‘And’,
‘God’, ‘fulfills’, ‘himself’, ‘in’, ‘many’, ‘ways’, ‘Lest’, ‘one’, ‘good’,
‘custom’, ‘should’, ‘corrupt’, ‘the’, ‘world’, ‘.’]
```

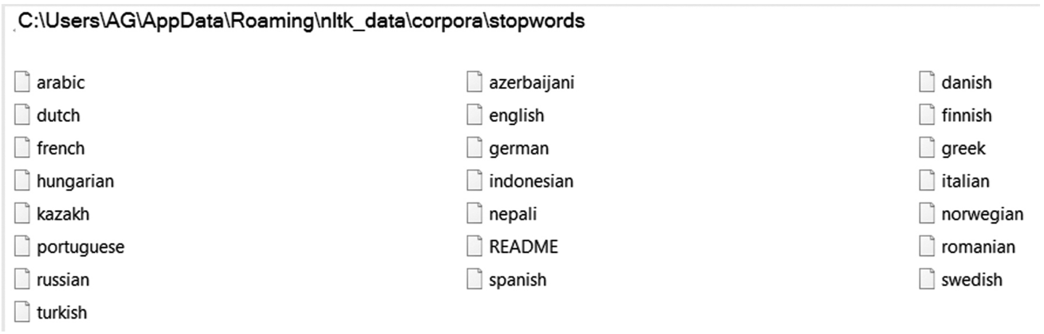
Following script gives another example of how the wrapper functions sent_tokenize() and word_tokenize() are used:

```
import nltk
s = “A quick brown fox jumped over a lazy dog. So we shall see. ”
sen_tok = nltk.sent_tokenize(s)
word_tok = nltk.word_tokenize(s)
print(‘Sentences->’, sen_tok)
print(‘Words->’, word_tok)
```

```
# Output
Sentences-> [‘A quick brown fox jumped over a lazy dog.’, ‘So we shall see.’]
Words-> [‘A’, ‘quick’, ‘brown’, ‘fox’, ‘jumped’, ‘over’, ‘a’, ‘lazy’, ‘dog’, ‘.’, ‘So’,
‘we’, ‘shall’, ‘see’, ‘.’]
```

24.6 Stop Words in NLTK

Stop words are just "filler words" and carry no own meaning. They serve the purpose of connecting other words together to create grammatical sentences (Figure 24.7).



C:\Users\AG\AppData\Roaming\nltk_data\corpora\stopwords

```
arabic        azerbaijani   danish
dutch         english       finnish
french        german        greek
hungarian     indonesian    italian
kazakh        nepali        norwegian
portuguese    README        romanian
russian       spanish       swedish
turkish
```

Figure 24.7 Screen shot of various files with stop words from different languages. Note that these files are data files and not scripts and are present under folder nltk_data\corpora

24.6.1 Using Function stopwords()

Throughout this book, the way to introduce a function/method/class has been to first examine its source code and then use it. For the stop words, however the source code is a bit complicated and hence discussed later (in a box).

For the present you can do with following:

- You can import stop words from nltk.corpus.
- This object stopwords has a method words(fileids = None). You can give name of one of the languages available in nltk_data\corpora\stopwords folder (whose screen shot was shown above). If you don't give any language_name parameter to the fields field, then it will load the stop words from all the languages.
- The words(fileids = None) method returns a "list of stop words".

The following script prints out the stop words in English language.

```
1  from nltk.corpus import stopwords
2  stop_words= stopwords.words("english")
3  print('Stop words length->', len(stop_words))
4  print('Stop words->', stop_words)
```

```
5  # Output (Output truncated to save space)
6  Stop words length-> 179
7  Stop words-> ['i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves', 'you',
8  "you're", "you've", "you'll", "you'd", 'your', 'yours', 'yourself', 'yourselves',
9  'he', 'him', 'his', 'himself', 'she', "she's", 'her', 'hers', 'herself', 'it',
10 "it's", 'its', 'itself', 'they', 'them', 'their', 'theirs', .....]
```

Line 1: As pointed out in the discussion earlier, stopwords is available as a corpora from nltk.corpus.

Line 2: Also as pointed out in discussion above, the .words() method is available. As a result of this line of code, the variable loads all the stop words in the file English. You can see this file in a file editor like notepad++ and in the author's copy of English stopwords, there are 179 stopwords.

Line 3: The output confirms that there are actually 179 stop words.

Line 4: This print statement shows that the corpora is stored as a “list of strings”.

You could choose a different language. The following example takes stop words from Nepali. The script and output on Jupyter notebook is as follows: (Note if the output is not encoded properly, you might get an error of type: UnicodeEncodeError)

```
from nltk.corpus import stopwords
stop_words= stopwords.words(“nepali”)
print(‘Stop words length->’, len(stop_words))
print(‘Stop words->’, stop_words)
# Output (Output truncated to save space)
Stop words length-> 255
Stop words-> [‘छ’, ‘र’, ‘पनि’, ‘छन्’, ‘लागि’, ‘भएको’, ‘गरेको’, ‘भने’, ‘गर्न’, ‘गर्ने’,
‘हो’, ‘तथा’, ‘यो’, ‘रहेको’,….]
```

Suppose you don’t give any value to the fileids attribute, then your list of stop words will have stop words of all the languages. The following script confirms this:

```
from nltk.corpus import stopwords
# If you dont give any value for fileids, you get stop words for all languages
stop_words= stopwords.words()
print(‘Stop words length->’, len(stop_words))
print(‘Stop words->’, stop_words)
# Output (Output truncated to save space)
Stop words length-> 4850
Stop words-> ['إذ', 'إذا', 'إذما', 'إذن', 'أف', 'أقل', 'أكثر', 'ألا', 'إلا', 'التي',
'الذي', 'الذين', 'اللاتي', . . .]
```

From the output you can see that a total of 4850 stop words are present in all the languages combined.

In any text processing task, what you can do is to load a file containing stop words and then use it to remove stop words. The following script takes a sample text and then tokenizes it and removes stop words from it (It gives list of those stop words which have been removed and also the non-stop words):

```
from nltk.book import *
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
stop_words= set(stopwords.words(“english”))
print(‘Stop words->’, stop_words)

sample_text = “””Scepticism is as much the result of knowledge, as knowledge is
of scepticism. To be content with what we at present know, is, for the most
part, to shut our ears against conviction;”””

sample_tok = word_tokenize(sample_text)
print(‘Tokens->’,sample_tok)
nonstop_words = []
for tok in sample_tok:
    if tok not in stop_words:
        nonstop_words.append(tok)

print('Non-stop words->',nonstop_words)
```

```
# Output (Truncated to save space)
Stop words-> {'then', 'he', 'do', 'what', "shan't", 'mustn', 'does', 'its',
'hers', 'are', 'didn', 'their', 'i', 'doing', 'of', 'yourselves', 'have', '
Tokens-> ['Scepticism', 'is', 'as', 'much', ...';']
Non-stop words-> ['Scepticism', 'much', 'result', 'knowledge', ',', 'knowledge',
'scepticism', '.', 'To', 'content', 'present', 'know', ',', ',', 'part', ',',
'shut', 'ears', 'conviction', ';']
```

Studying source code file to know how stop words work in NLTK

The steps in studying the source code area:

- You can import the attribute stopwords from nltk.corpus (not corpora since corpora is under nltk_data). The stopwords variable is present in the __init__.py file of nltk\corpus folder. The relevant portion of the file is as follows:

```
    # Lines 194-195 of file __init__.py in \nltk\corpus
194 stopwords = LazyCorpusLoader(
195     'stopwords', WordListCorpusReader, r'(?!README|\.).*', encoding='utf8')
```

- So what the attribute stopwords does is to create an object of class WordListCorpusReader. You can check this by checking the type() of variable stopwords. Further in Python you can always use the mro() method of an object to know the corresponding class in the source code file. This is shown in the following code:

```
from nltk.corpus import stopwords
print(type(stopwords))
# Using an object's mro() you can also know its location in source code
print(type(stopwords).mro())
```

```
# OUTPUT
<class 'nltk.corpus.reader.wordlist.WordListCorpusReader'>
[<class 'nltk.corpus.reader.wordlist.WordListCorpusReader'>, <class
'nltk.corpus.reader.api.CorpusReader'>, <class 'object'>]
```

- From the above output, two things are very clear: (1) The class WordListCorpusReader is defined in file nltk\corpus\reader\wordlist.py and (2) the WordListCorpusReader class in turn inherits from class CorpusReader which is defined in file nltk\corpus\reader\api.py
- Now to know the methods of the WordListCorpusReader class, you can either see its source code or use the dir() function on it.

```
# Use dir() function to get list of all attributes and methods
# of class WordListCorpusReader
import nltk
print(dir(nltk.corpus.reader.wordlist.WordListCorpusReader))
```

```
# OUTPUT
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
'__str__', '__subclasshook__', '__unicode__', '__weakref__', '_get_root',
'abspath', 'abspaths', 'citation', 'encoding', 'ensure_loaded', 'fileids',
'license', 'open', 'raw', 'readme', 'root', 'unicode_repr', 'words']
```

- From the above output, you can see that the names towards the end like abspath represent methods.
- Here the focus is only on one method, i.e., words(). So open the wordlist.py file to have a look at the source code of the class WordListCorpusReader and its method words(). Relevant portion of source code along with original line numbers is shown as follows:

```
# Definition of WordListCorpusReader class and its method words()
# Taken from file nltk\corpus\reader\wordlist.py
17 class WordListCorpusReader(CorpusReader):
18     """
19     List of words, one per line.  Blank lines are ignored.
20     """
21     def words(self, fileids=None, ignore_lines_startswith='\n'):
22         return [line for line in line_tokenize(self.raw(fileids))
23                 if not line.startswith(ignore_lines_startswith)]
```

- From the signature of the words() method you can see that an important parameter to this is fileids.
- This WordListCorpusReader class in turn inherits from a class called CorpusReader. But the source code of this class is not taken up here. Those interested may see the relevant file, i.e., \nltk\corpus\reader\api.py

24.7 Stemming and Lemmatizing

Stemming is the process whereby a word is reduced to its root form. On other hand, lemmatization is that process whereby the different "inflected forms[1]" of a word are treated as same.

Stemming and lemmatizing are closely linked. The difference is that in stemming, the context of the word does not matter whereas in lemmatization, the context in which a word appears matters. Stemming algorithms generally work by "cutting off" the suffixes or prefixes. For example, take the two forms of the word "fly", i.e., "flying" and "flies". Stemming algorithm will reduce "flying" → "fly", but it will reduce "flies" to "fli" which is incorrect. On the other hand, lemmatizing does "morphological analysis" of the words. A lemma is the "base form" of all inflected words. So a dictionary of words will contain the lemmas and not the stems of a word. For example, the lemma for "dying" would be "die" not "dyi", where as the stem for "dying" would be "dyi".

The original Porter stemmer (by Martin Porter) and its revised version Porter2 (aka Snowball stemmer) are the most popular NLP stemmers. Nowadays, the Porter2, stemmer is called Snowball

[1] In order to express the various grammatical forms of words (nouns, verbs, adjectives. Etc.), extra letter(s) may be added to it. This is called inflection. In English language, a noun may be inflected to get its pulural form like table→ tables. Similarly, verbs are inflected to get the various tenses. Adjectives may be inflected to get their comparatives and superlatives.

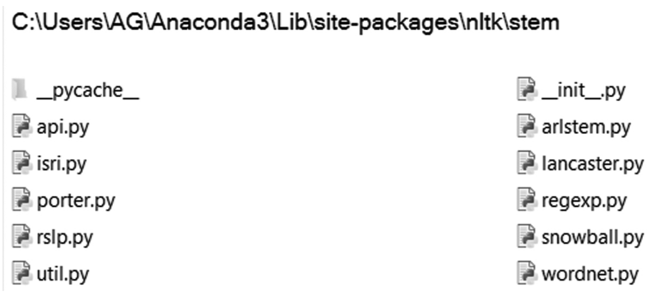
stemmer (Snowball is a language that Martin Porter developed later to support other languages than English; so, people sometimes call the Porter2 stemmer as "Snowball English stemmer").

The following table compares and contrasts stemming and lemmatizing:

Issue	Stem	Lemma
Goal	To reduce inflected form to a base form	Same goal
Method	Crude heuristic process	Here the "base" form or the dictionary form of a word is created by removing the inflectional endings.
Building	Easier	More difficult since it requires deep linguistic knowledge.
The way the algorithm works	♦ A stemmer does not take into account the "context" of a word. So it is unable to differentiate between those words whose meanings change depending upon their context. ♦ Stemmer tends to be more crude. For example "happiness" may stem to "happi".	♦ Some lemmatizers may do a broad "fuzzy" matching. For example they might match "hot" to "warm". ♦ Tends to be more accurate. "happiness" is likely to be lemmatized to "happy" not "happi".

24.7.1 Understanding the Stemmers Made Available in NLTK

To understand the stemmers available in NLTK, you need to see the file structure of stem directory in NLTK directory. On author's system, the screen shot of stem directory is shown in Figure 24.8.



```
C:\Users\AG\Anaconda3\Lib\site-packages\nltk\stem

__pycache__        __init__.py
api.py             arlstem.py
isri.py            lancaster.py
porter.py          regexp.py
rslp.py            snowball.py
util.py            wordnet.py
```

FIGURE 24.8 Screen shot of folder nltk\stem. It shows the different stemmers available in NLTK. Note that Wordnet is a lemmatizer. (Also note that not all are for English language.)

24.7.2 Using SnowballStemmer

It is not possible to discuss all the stemmers here. So the popular stemmer SnowballStemer is taken up here. Some important points regarding using this stemmer are:

- This stemmer can be used by instantiating an object of class SnowballStemmer.
- To do so you need to import SnowballStemmer from nltk.stem module.
- Its constructor (ie the __init__()) method takes only 2 optional parameter namely: (1) "language" and (2) "ignore_stopwords=False". The language parameter is mandatory and must be one of the languages specified by the SnowballStemmer.languages id. (Note that the SnoballStemmer class

has an attribute languages, which specifies the languages for which this stemmer works). However, the "ignore_stopwords" parameter is optional and has a default value of False.

- So if you want to create a stemmer of this class, you may provide at least one parameter, i.e., "language" and you may provide an optional parameter ignore_stopwords. For the parameter ignore_stopwords, you can only give a bool value of True or False.
- Note that the SnowballStemmer is based on the algorithm by Porter and is called SnowballStemmer because "Porter created a programming language with this name for creating new stemming algorithms."
- Further the SonwballStemmer has a method stem(token). This method takes a single parameter token which is a "list of words".
- Also note that the module stem, i.e., the file stem.py also has definition of a class called Porter-Stemmer which implements the original Porter stemming algorithm. This is not discussed here.

The following script shows all the languages available in the SnowballStemmer module:

```
from nltk.stem import SnowballStemmer
print(SnowballStemmer.languages)
```

```
#Output
('arabic', 'danish', 'dutch', 'english', 'finnish', 'french', 'german',
'hungarian', 'italian', 'norwegian', 'porter', 'portuguese', 'romanian', '
russian', 'spanish', 'swedish')
```

Note: The output shows an entry 'porter'. Note 'porter' is not a language but an "alternative English stemmer".

Following is a script which uses two different stemmers, i.e., 'English' stemmer and 'porter' stemmer.

```
from nltk.stem.snowball import SnowballStemmer
from nltk.tokenize import word_tokenize
words = 'It is a truth universally acknowledged, that a single man in possession
of a good fortune, must be in want of a wife.'
stemmer1 = SnowballStemmer('english', ignore_stopwords = True)
stemmer2 = SnowballStemmer('porter')
toks = word_tokenize(words)
stemwords1 = []
stemwords2 = []
for a_tok in toks:
    s1 = stemmer1.stem(a_tok)
    stemwords1.append(s1)
    s2 = stemmer2.stem(a_tok)
    stemwords2.append(s2)
print("SnowballStemmer english->", stemwords1)
print("SnowballStemmer porter ->", stemwords2)
```

```
#Output
SnowballStemmer english-> ['it', 'is', 'a', 'truth', 'univers', 'acknowledg',
',', 'that', 'a', 'singl', 'man', 'in', 'possess', 'of', 'a', 'good', 'fortun',
',', 'must', 'be', 'in', 'want', 'of', 'a', 'wife', '.']
SnowballStemmer porter -> ['It', 'is', 'a', 'truth', 'univers', 'acknowledg',
',', 'that', 'a', 'singl', 'man', 'in', 'possess', 'of', 'a', 'good', 'fortun',
',', 'must', 'be', 'in', 'want', 'of', 'a', 'wife', '.']
```

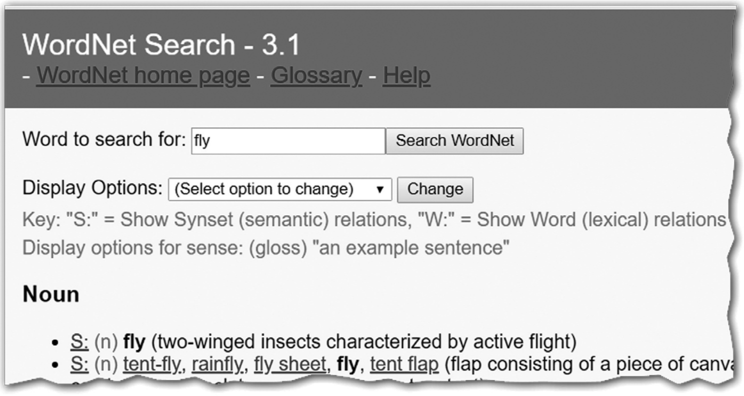
The following script is another example of use of the SnowballStemmer:

```
import nltk
from nltk.stem.snowball import SnowballStemmer
from nltk.tokenize import word_tokenize

stemmer = SnowballStemmer(language="english")

sample_text = """Scepticism is as much the result of knowledge, as knowledge is
ofscepticism. To be content with what we at present know, is, for the most
part, to shut our ears against conviction;"""
sample_stem = []
sample_tok = word_tokenize(sample_text)
for tok in sample_tok:
    another_stem = stemmer.stem(tok)
    sample_stem.append(another_stem)
print('Stemmed text->', sample_stem)
```

```
# Output
Stemmed text-> ['sceptic', 'is', 'as', 'much', 'the', 'result', 'of', 'knowledg',
',', 'as', 'knowledg', 'is', 'of', 'sceptic', '.', 'to', 'be', 'content', 'with',
'what', 'we', 'at', 'present', 'know', ',', 'is', ',', 'for', 'the', 'most',
'part', ',', 'to', 'shut', 'our', 'ear', 'against', 'convict', ';']
```

24.7.3 A Brief Note on Wordnet

Wordnet is available at: https://wordnet.princeton.edu/. Screen shot is shown in Figure 24.9.



WordNet Search - 3.1
- WordNet home page - Glossary - Help

Word to search for: fly [Search WordNet]

Display Options: (Select option to change) [Change]

Key: "S:" = Show Synset (semantic) relations, "W:" = Show Word (lexical) relations

Display options for sense: (gloss) "an example sentence"

Noun

- S: (n) **fly** (two-winged insects characterized by active flight)
- S: (n) tent-fly, rainfly, fly sheet, **fly**, tent flap (flap consisting of a piece of canva

FIGURE 24.9 Screenshot of Wordnet online. Here the word being searched is "fly".

Wordnet has a vocabulary of its own. Discussing it here would interrupt the flow of the chapter. So details about Wordnet are given in a box at the end of the chapter.

24.7.4 Lemmatizing in NLTK

The following regarding Wordnet lemmatizing in NLTK are relevant:

- NLTK implements the Wordnet algorithm also in its module stem as a class WordNetLemmatizer.
- So you have to import the class WordNetLemmatizer from the module stem.

- The __init__() method of this class only takes self as a parameter. This means that an object of class WordNetLemmatizer is constructed without giving any parameter.
- Further this class has a method lemmatize(word, pos=NOUN), which takes two parameters. The first mandatory parameter "word" is the word to be given to the lemmatizer. The second parameter is "pos" which has a default value of NOUN. <pos> is one of the module attributes ADJ, ADJ_SAT, ADV, NOUN or VERB
- It is useful to have a look at the source code of the lemmatize(), method of WordNetLemmatizer class, which is as follows:

```
-  # Source code lines 39-41 of file wordnet.py
-  # This file is at \nltk\stem ie in stem module of nltk
-  # Here definition of lemmatize() method of class WordNetLemmatizer is given
39     def lemmatize(self, word, pos=NOUN):
40         lemmas = wordnet._morphy(word, pos)
41         return min(lemmas, key=len) if lemmas else word
```

- The source code of the lemmatize() method shows that it in turn calls the _morphy method of Wordnet module. Those interested can see the file wordnet.py available at nltk\corpus\reader.

24.8 Using WordNetLemmatizer

Following code shows how stemming and lemmatizing are used in NLTK:

```
from nltk import pos_tag
from nltk.tokenize import word_tokenize
from nltk.stem import PorterStemmer, WordNetLemmatizer
try:
    wordInp = input("give a word ")
    word_stemmer = PorterStemmer()
    word_lem = WordNetLemmatizer()
    wordStem = word_stemmer.stem(wordInp)
    wordLem = word_lem.lemmatize(wordInp)
    print("word->", wordInp, "stem->", wordStem, "Lemma->", wordLem)
except:
    print("Something wrong")
```

```
# Output
give a word swimming
word-> swimming stem-> swim Lemma-> swimming
```

The above script takes a string as input from user and gives its stem and lemma.

It was shown in the signature of the lemmatize() method of the WordNetlemmatizer, that it can take a second optional parameter "pos = NOUN". But you can always give it some other value from: ADJ, ADJ_SAT, ADV, NOUN or VERB. If you do so, the lemmatize() method will behave differently. However, note that the options ADJ, ADV, etc. are defined in the Wordnet module. So you have to import it also.

The following script shows how the lemmatize() method behaves differently when different values for pos are given:

```python
# You need to import wordnet because NOUN, VERB etc are defined in it
from nltk.corpus.reader import wordnet as wn
from nltk.stem import WordNetLemmatizer
a_word = 'howling'
wnl = WordNetLemmatizer()
print(wnl.lemmatize(word = a_word, pos = wn.NOUN))
print(wnl.lemmatize(word = a_word, pos = wn.VERB))
```

```
# Output
howling
howl
```

24.9 Parts of Speech Tagging

A Parts Of Speech (POS) tagger is a program/script which takes some string (para, sentence, etc.) and assigns a "part of speech" POS (POS is referred to by different names like "word classes", "grammatical category", "lexical categories" etc.). Traditional parts of speech are nouns, verbs, adverbs, conjunctions, etc. A collection of tags is known as a tagset. The alphabetical list of POS tags as described in Penn Treebank Project are as follows[2]: (This tag set uses uppercase letters)

Number	Tag	Description	Number	Tag	Description
1.	CC	Coordinating conjunction	19.	PRP$	Possessive pronoun
2.	CD	Cardinal number	20.	RB	Adverb
3.	DT	Determiner	21.	RBR	Adverb, comparative
4.	EX	Existential *there*	22.	RBS	Adverb, superlative
5.	FW	Foreign word	23.	RP	Particle
6.	IN	Preposition or subordinating conjunction	24.	SYM	Symbol
7.	JJ	Adjective	25.	TO	*to*
8.	JJR	Adjective, comparative	26.	UH	Interjection
9.	JJS	Adjective, superlative	27.	VB	Verb, base form
10.	LS	List item marker	28.	VBD	Verb, past tense
11.	MD	Modal	29.	VBG	Verb, gerund or present participle
12.	NN	Noun, singular or mass	30.	VBN	Verb, past participle
13.	NNS	Noun, plural	31.	VBP	Verb, non-3rd person singular present
14.	NNP	Proper noun, singular	32.	VBZ	Verb, 3rd person singular present
15.	NNPS	Proper noun, plural	33.	WDT	Wh-determiner
16.	PDT	Predeterminer	34.	WP	Wh-pronoun
17.	POS	Possessive ending	35.	WP$	Possessive wh-pronoun
18.	PRP	Personal pronoun	36.	WRB	Wh-adverb

[2] For complete list and discussion see: http://groups.inf.ed.ac.uk/switchboard/POS-Treebank.pdf

(Above list taken from Part-of-Speech Tagging Guidelines for the Penn Treebank Project) (3rd Revision, 2nd printing) available at: http://groups.inf.ed.ac.uk/switchboard/POS-Treebank.pdf)

The NLTK library has a module tag which provides a function pos_tag() for pos tagging. This function is available in nltk\tag\__init__.py

The signature for this function is as follows:

```
def pos_tag(tokens, tagset=None, lang='eng'):
Parameters:-
(1) tokens: It is the Sequence of tokens to be tagged. Must be of type list(str)
(2) tagset: the tagset to be used, e.g. universal, wsj, brown. Must be of type
string ie str.
(3) lang: the ISO 639 code of the language, e.g. 'eng' for English, 'rus' for
Russian. Must be of type string ie str
Return type:-
The method returns tagged tokens in form list(tuple(str, str))
```

Note that this method gives a list of tuples.

There is another method of this module called pos_tag_sents(). It is similar to pos_tag() except that it takes a list of sentences and gives a list of list of tuples.

```
def pos_tag_sents(sentences, tagset=None, lang='eng'):
Parameters:-
(1) tokens: It is a list of sentences to be tagged. Each sentence forms an inner
list. So the parameter is of type list(list(str))
(2) tagset: It is the ISO 639 code of the language, e.g. 'eng' for English, 'rus'
for Russian. Its type is string ie str.
Return Type:-
The method returns the list of tagged sentences in format list(list(tuple(str,
str)))
```

The following script shows how NLTK can be used to POS tag text:

```
from nltk import pos_tag, word_tokenize
text_sent = " The goods that men do are buried with their bones."
tagged_sent = word_tokenize(text_sent)
sent_with_pos = pos_tag(tagged_sent)
#parameter tagset = 'universal' gives a different set of tags
sent_with_pos2 = pos_tag(tagged_sent, tagset = 'universal')
print('Tagged with default tags->', sent_with_pos)
print('Tagged with universal tags->',sent_with_pos2)
```

```
# Output
Tagged with default tags-> [('The', 'DT'), ('goods', 'NNS'), ('that', 'WDT'),
('men', 'NNS'), ('do', 'VBP'), ('are', 'VBP'), ('buried', 'VBN'), ('with', 'IN'),
('their', 'PRP$'), ('bones', 'NNS'), ('.', '.')]
Tagged with universal tags-> [('The', 'DET'), ('goods', 'NOUN'), ('that', 'DET'),
('men', 'NOUN'), ('do', 'VERB'), ('are', 'VERB'), ('buried', 'VERB'), ('with',
'ADP'), ('their', 'PRON'), ('bones', 'NOUN'), ('.', '.')]
```

24.10 Some Additional Details about Wordnet

Even though the NLTK implementation of Wordnet algorithm was used in the chapter, many of the concepts of Wordnet were not discussed or explained. Some of the important terms/concepts related to Wordnet are given in the following box:

Wordnet

Those who are familiar with graph theory can think of WordNet as a graph. A graph has "nodes" and "edges". In Wordnet semantically similar words (called "synsets") are represented by "Nodes" and the semantic relationships are shown as edges from one node to another.

In this graph the "edges" or "relationships" can be of the following five types:

- **Synonym or synset:** A synonym or synset is a group of words with the same meaning and so can be interchanged in some context without changing the meaning of the proposition. For instance: roof and ceiling.
- **Hypernym:** A hypernym is a word that is more general than some other word. So a hypernym includes the meaning of the "other word". So a "bird" is a type of animal and so is a "fish". So you can say that "animal" is hypernym of "bird" and "fish". In technical terms, you call a hypernym a "Is-a relationship". So a bird is an animal.
- **Hyponym:** A word that is more specific than some other word is said to be its hyponym. So a "fish" is a hyponym of "bird". You can think of hypernym as "parent" and hyponym as "child".
- **Meronym:** A word that is a part of some other word is called a meronym. So a "thumb" is a meronym of "hand". (Meronymy can be thought of as the part-whole relation holds between synsets.)
- **Holonym:** If X contains Y, then X is a holonym of Y. So a "hand" is a holonym of "thumb".

Certain concepts used in Wordnet

Certain concepts in Wordnet need to be understood before it can be used. These are:

1. Morphology and Morphemes

In a language, a morpheme is the smallest grammatical unit. The characteristics of a morpheme are:

- It is the smallest word or part of a word which has meaning.
- A morpheme cannot be divided into smaller parts.
- A morpheme will have the same meaning in different contexts.

The basic difference between a "word" and a "morpheme" is that a word by definition is "freestanding", while a morpheme may or may not stand alone.

Morphemes are basically of two types, i.e., stems and affixes. Affixes can be further divided into prefixes and suffixes. A prefix is an affix which is placed before the stem of a word. This can be shown as follows:

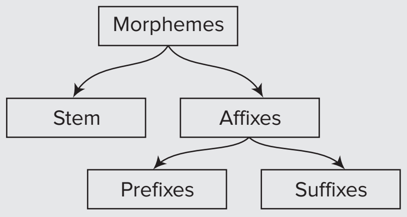


Figure Shows relationship between (1) morpheme, (2) stem, (3) affixes, (4) prefixes, and (5) suffixes

2. Stem

Stemming algorithms work by cutting off the end or the beginning of the inflected word using list of common prefixes or suffixes. This indiscriminate cutting can be successful in some occasions, but not always.

3. Affixes

Affixes are of four types namely prefixes, suffixes, infixes and circumfixes. A "prefix" precedes the "stem". A "suffix" follows a stem. An infix is 'inserted into' the stem. Finally a "circumfix" both precedes and "follows" a stem.

4. Word forms

Word forms are the different inflexions, i.e., the modifications of a word to express different grammatical categories such as number, gender, case, etc. For example, fly, flew, flying, flown, etc. are different word forms of the word 'fly'.

5. Lexeme, inflections

A lexeme is the fundamental unit or the minimal unit of a lexicon.

On the other hand, an inflection or inflexion is the modification of a lexeme to denote various grammatical categories of a lexeme. These categories can be the various forms like tense, gender, number, mood, case, etc. The inflection of a lexeme is generally (But not always) done by a prefix, postfix or infix.

For example, the word "dog" is a lexeme. In English, one common rule to have the "plural inflected" form of a noun is to add an "s" ie a postfix. So the word "dog" in its (plural) inflected form becomes "dogs". Similarly for the lexeme "write", the inflection "wrote" is the "past tense" inflection.

In corpus linguistics, lexemes are commonly referred to as lemmas.

6. Lemma

Lemma is the 'base' form of a word. It is generally that form of a word under which all other forms of the word are found in a dictionary entry. For example flying, flew etc will all be found under the 'head' fly. So you can say that fly is the lemma for all these word forms.

In corpus linguistics, lexemes are commonly referred to as lemmas.

7. Lemmatising

In lemmatization one converts a "word" into its "dictionary form". Lemmatisation is similar to stemming. The difference is that a stemmer does not take in account the "context" of a word. Hence, a stemmer is unable to "differentiate" which may have different meanings depending on part of speech.

Conceptual Questions

1. Examine the source code for TokenizerI abstract base class available at: https://www.nltk.org/_modules/nltk/tokenize/api.html#TokenizerI. Why is it called an Abstract Base Class (ABC)? Can you create an object of type TokenizerI? Why not? Which are the abstract methods of this ABC?
2. The RefexpTokenizer() class for tokenizing may be used in two different ways. First by creating an object of type RegexpTokenizer() and second by using the wrapper function regexp_tokenize(). What are the advantages/disadvantages of the two methods?
3. What are stop words?
4. What are "stemming" and "lemmatizing"? What are the differences between them?

5. What are "synonyms or synsets"?
6. What do the terms: "hypernym", "hyponym", "meronym", "holonym" mean?
7. What is a "morpheme"?
8. What are "affixes", "prefixes", "suffixes" and "circumfixes"?
9. What are "lexemes" and "inflections"?
10. What is "Part Of Speech, i.e., POS" tagging?

Exercise

1. What will be the output of the following script?

```
from nltk.tokenize import RegexpTokenizer
s1 = '''
Some random text.
11 o12., dad, 56rsde
abcdef 9ii10odafsr. \n  abc \d mmm \n /m /n
  '''
tokenizer = RegexpTokenizer('\w+|\$[\d\.]+|\S+')
tokenizer.tokenize(s1)
print(s1)
```

2. Can you programmatically find out which are the languages for which NLTK library provides for stop words?
3. Write a script which prints the first five stop words of each of the stop word languages in NLTK.

Beyond Textbook

1. Study source code of RegexpTokenizer
The NLTK library has its own website where all details about the library including the source code is available. The source code is also available on github at this link[3]. If you study the source code for the RegexpTokenizer, you will find that it sub-classes (i.e., inherits from) the TokenizerI class. Further, the TokenizerI is an abstract base class and has an abstract method tokenize().
The task here is to see the source code of the RegexpTokenizer() class available here and to confirm the following:
That this class ie RegexpTokenizer inherits from TokenizerI class.
That the RegexpTokenize() class implements the tokenize() method.
To further see from source code that there are three other tokenizers in this file regexp.py namely WhitespaceTokenizer, BlanklineTokenizer, and WordPunctTokenizer. To see that all these three tokenizer classes inherit from RegexpTokenizer and therefore don't implement the tokenize() method since the tokenize() method is already implemented in the RegexpTokenizer class.

[3] See: https://github.com/nltk/nltk/tree/develop/nltk

To study the init method of the RegexpTokenizer class. The __init__() method of RegexpTokenizer class is as follows. By looking at this method you can clearly make out that you can create a RegexpTokenizer object by giving it your own pattern also.

```
def __init__(
        self,
        pattern,
        gaps=False,
        discard_empty=True,
        flags=re.UNICODE | re.MULTILINE | re.DOTALL,
    ):
```

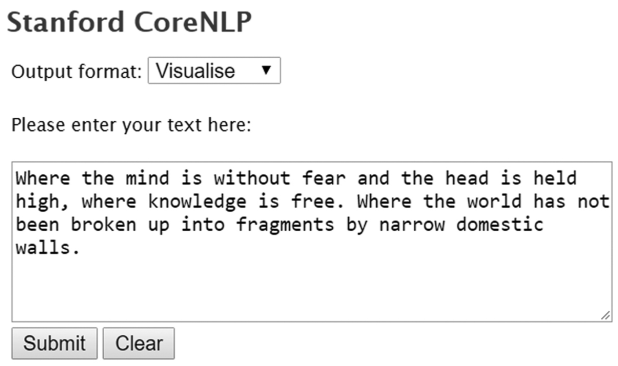
2. Again look at the source code file regexp.py file at the link shown above. By looking at the source code, can you tell the regexp pattern used by the class WhitespaceTokenizer and the class BlanklineTokenizer?
   Hint: See the source code of the __init__() method of the WhitespaceTokenizer class. It is as follows:

```
def __init__(self):
    RegexpTokenizer.__init__(self, r'\s+', gaps=True)
```

   From the above code for the __init__() method of WhitespacTokenizer, it is clear that the regexp pattern for whitespace is `r'\s+'`.
   Similarly the __init__() method for the class BlanklineTokenizer is:

```
def __init__(self):
    RegexpTokenizer.__init__(self, r'\s*\n\s*\n\s*', gaps=True)
```

   From above, it is clear that the regexp pattern for the BlanklineTokenizer is:
   `r'\s*\n\s*\n\s*'` .
3. Online demo of Stanford CoreNLP
   There is an online tool "Stanford CoreNLP" available at: http://nlp.stanford.edu:8080/corenlp/ . If you give it a sentence, it will tag it with POS (Parts of Speech). Sample is shown in Figure 24.10.



Where the mind is without fear and the head is held high, where knowledge is free. Where the world has not been broken up into fragments by narrow domestic walls.

FIGURE 24.10 Screen shot of Stanford online tool for NLP