Natural Language Programming (NLP)

Topics to be covered:

- Tokenizing text using functions word_tokenize and sent_tokenize.
- Computing Frequencies with FreqDist and ConditionalFreqDist.
- Generating Bigrams and collocations with bigrams and collocations.
- Stemming word affixes using PorterStemmer and LancasterStemmer.
- Tagging words to their parts of speech using pos_tag.

Humans communicate in natural languages such as English, German, Japanese and so on.On the other hand, a Computer communicates in Machine Language, which has a defined set of rules.

As a reason, a computer cannot communicate with humans in an effective way.Natural Language Processing helps in increasing computer intelligence to understand human languages as spoken and to respond.

Index

Why NLP?

NLP techniques are capable of processing and extracting meaningful insights, from huge unstructured data available online.

- It can automate translating text from one language to other.
- These techniques can be used for performing sentiment analysis.
- It helps in building applications that interact with humans as humans do.
- Also, NLP can help in automating Text Classification, Spam Filtering, and more.

nltk

nltk is a popular Python framework used for developing Python programs to work with human language data. Key features of nltk:

- It provides access to over 50 text corpora and other lexical resources.
- It is a suite of text processing tools.
- It is free to use and Open source.
- It is available for Windows, Mac OS X, and Linux.

Installing nltk

Having Python installed is the prerequisite for nltk.

Once Python is installed, nltk can be installed, automatically, using the shown command.

```
pip install nltk
```

Once the installation is done, it can be verified by opening the Python terminal and typing below command.

```
>>> import nltk
```

If no error occurs by running the above command, then this indicates successful installation of nltk.

Basic Understanding of nltk

Now let's understand by performing simple tasks in the next couple of slides.

Note: nltk.download('punkt') after the mentioned line below

Splitting a sample text into a list of sentences.

```
>>> import nltk

>>> text = "Python is an interpreted high-level programming language for
general-purpose programming. Created by Guido van Rossum and first releas
ed in 1991."

>>> sentences = nltk.sent_tokenize(text)

>>> len(sentences)

2
```

As seen above, sent_tokenize function generates sentences from the given text.

Splitting a sample text into words using word_tokenize function.

```
>>> words = nltk.word_tokenize(text)

>>> len(words)

22

>>> words[:5]

['Python', 'is', 'an', 'interpreted', 'high-level']
```

The expression words[:5] displays first five words of list words.

Determining the frequency of words present in sample text using FreqDist function.

```
>>> wordfreq = nltk.FreqDist(words)

>>> wordfreq.most_common(2)

[('programming', 2), ('.', 2)]
```

The expression wordfreq.most_common(2) displays two highly frequent words with their respective frequency count.

Nltk book collection

These texts are available in collection book of nltk.

They can be downloaded by running the following command in Python interpreter, after importing nltk successfully.

```
>>> import nltk
```

```
>>> nltk.download('book')
```

```
>>> from nltk.book import *
*** 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
```

The above figure illustrates the output of the command from nltk.book import *.

The command loads nine texts and nine sentences, from the collection book.

Searching Text

There are multiple ways of searching for a pattern in a text.

The example shown below searches for words starting with tri, and ending with r.

```
>>> text1.findall("<tri.*r>")

triangular; triangular; triangular; triangular
```

Basic Tasks with Text

In this topic, you will understand how to perform the following activities, using text1 as input text.

- Total Word Count
- Unique Word Count
- Transforming Words
- Word Coverage
- Filtering Words
- Frequency Distribution

Determining Total Word Count

NLTK Book module consits of 9 text class. Type the name of the text or sentence to view it.

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

The text1, imported from nltk.book is an object of nltk.text.Text class.

```
>>> from nltk.book import *
>>> type(text1)
<class 'nltk.text.Text'>
```

Total number of words in text1 is determined using len.

```
>>> n_words = len(text1)
>>> n_words
260819
```

Determining Unique Word Count

A unique number of words in text1 is determined using set and len methods.

```
>>> n_unique_words = len(set(text1))
>>> n_unique_words
19317
```

set(text1) generates list of unique words from text1.

Transforming Words

It is possible to apply a function to any number of words and transform them.

Now let's transform every word of text1 to lowercase and determine unique words once again.

```
>>> text1_lcw = [ word.lower() for word in set(text1) ]
>>> n_unique_words_lc = len(set(text1_lcw))
>>> n_unique_words_lc
17231
```

A difference of 2086 can be found from n_unique_words.

Determining Word Coverage

Word Coverage: Word Coverage refers to an average number of times a word is occurring in the text.

$$W_c = \frac{total_{words}}{unique_{words}}$$

The following examples determine Word Coverage of raw and transformed text1.

```
>>> word_coverage1 = n_words / n_unique_words
>>> word_coverage1
13.502044830977896
```

On average, a single word in text1 is repeated 13.5 times.

```
>>> word_coverage2 = n_words / n_unique_words_lc
>>> word_coverage2
15.136614241773549
```

Filtering Words

Now let's see how to filter words based on specific criteria. The following example filters words having characters more than 17.

```
>>> big_words = [word for word in set(text1) if len(word) > 17 ]
>>> big_words
['uninterpenetratingly', 'characteristically']
```

A list of comprehension with a condition is used above.

Now let's see one more example which filters words having the prefix Sun.

```
>>> sun_words = [word for word in set(text1) if word.startswith('Sun') ]
>>> sun_words
['Sunday', 'Sunset', 'Sunda']
```

The above example is case-sensitive. It doesn't filter the words starting with lowercase s and followed by un.

Frequency Distribution

FreqDist functionality of nltk can be used to determine the frequency of all words, present in an input text.

The following example, determines frequency distribution of text1 and further displays the frequency of word Sunday.

```
>>> text1_freq = nltk.FreqDist(text1)
>>> text1_freq['Sunday']
7
```

Example	Description
`fdist = FreqDist(samples)`	Create a frequency distribution containing the given samples
`fdist.inc(sample)`	Increment the count for this sample
`fdist['monstrous']`	Count of the number of times a given sample occurred
`fdist.freq('monstrous')`	Frequency of a given sample
`fdist.N()`	Total number of samples
`fdist.keys()`	The samples sorted in order of decreasing frequency
`for sample in fdist:`	Iterate over the samples, in order of decreasing frequency
`fdist.max()`	Sample with the greatest count
`fdist.tabulate()`	Tabulate the frequency distribution
`fdist.plot()`	Graphical plot of the frequency distribution
`fdist.plot(cumulative=True)`	Cumulative plot of the frequency distribution
`fdist1 < fdist2`	Test if samples in `fdist1` occur less frequently than in `fdist2`

Now let's identify three frequent words from text1_freq distribution using most_common method.

```
>>> top3_text1 = text1_freq.most_common(3)
>>> top3_text1
[(',', 18713), ('the', 13721), ('.', 6862)]
```

The output says the three most frequent words are , , the, and .

It may be weird for few of you.

In general, you would be interested in finding frequent words which are not common in usage and specific to input text. In the next example, you will perform the following:

- Filter words having all characters and of larger length.
- Determine frequency distribution of the filtered words.
- Identify the three most common words.

```
>>> large_uncommon_words = [word for word in text1 if word.isalpha() and len(word) > 7 ]
>>> text1_uncommon_freq = nltk.FreqDist(large_uncommon_words)
>>> text1_uncommon_freq.most_common(3)
[('Queequeg', 252), ('Starbuck', 196), ('something', 119)]
```

Queequeg, Starbuck and something are the top three words, based on the chosen criteria.

Text Corpora

In the previous topic, you have worked with a simple text collection text1.

In this topic, you will work with larger text collections known as Text Corpora or Text Corpus.

The following code snippet downloads more text corpus, which is varied in content.

```
>>> import nltk
>>> nltk.download('book')
```

Two popular Text Corpora available from nltk, which you will be using in this course are:

- Genesis: It is a collection of few words across multiple languages.
- Brown: It is the first electronic corpus of one million English words.

Other Corpus in nltk

- Gutenberg : Collections from Project Gutenberg
- Inaugural : Collection of U.S Presidents inaugural speeches

Popular text copora.

- stopwords : Collection of stop words.
- reuters : Collection of news articles.
- cmudict : Collection of CMU Dictionary words.
- movie_reviews : Collection of Movie Reviews.
- np_chat : Collection of chat text.
- names : Collection of names associated with males and females.
- state_union : Collection of state union address.
- wordnet : Collection of all lexical entries.
- words : Collection of words in Wordlist corpus.

Accessing Text Corpora

Any text corpus has to be imported before you start working with it.

- The below code imports genesis text corpus.

```
>>> from nltk.corpus import genesis
```

Various text collections available under genesis text corpus are viewed by fileids method.

```
>>> genesis.fileids()
['english-kjv.txt',
 'english-web.txt',
  ....]
```

The output displays eight text collections, present in genesis corpus.

Working with a Text Corpus

Now let's understand how to work with a text corpus.

The following example determines the average word length and average sentence length of each text collection present in genesis corpus.

```
>>> for fileid in genesis.fileids():
...     n_chars = len(genesis.raw(fileid))
...     n_words = len(genesis.words(fileid))
...     n_sents = len(genesis.sents(fileid))
...     print(int(n_chars/n_words), int(n_words/n_sents), fileid)
```

The methods raw, words and sents used in code determine the total number of characters, words, and sentences present in a specific text collection.

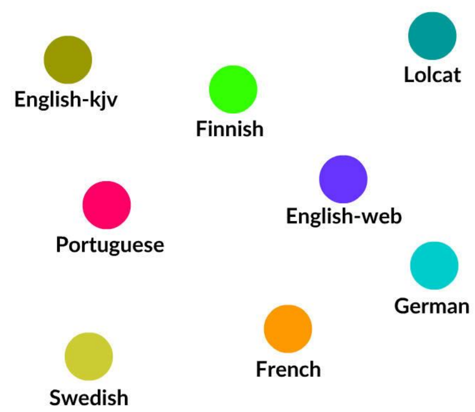
The output of code is shown below. Text collection finnish.txt has different average word length.

```
4 30 english-kjv.txt
4 19 english-web.txt
5 15 finnish.txt
4 23 french.txt
4 23 german.txt
4 20 lolcat.txt
4 27 portuguese.txt
4 30 swedish.txt
```

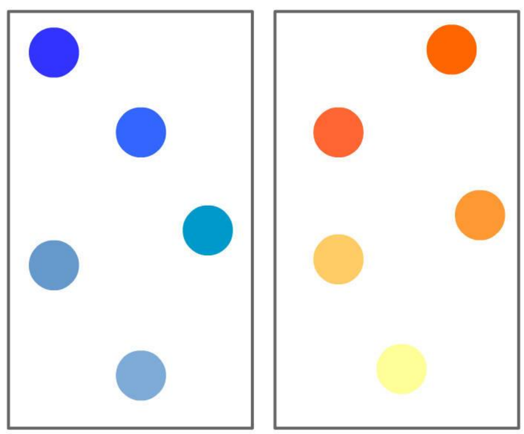
Text Corpus Structure

A text corpus is organized into any of the following four structures.

- Isolated - Holds Individual text collections.
- Categorized - Each text collection tagged to a category.
- Overlapping - Each text collection tagged to one or more categories, and
- Temporal - Each text collection tagged to a period, date, time, etc.



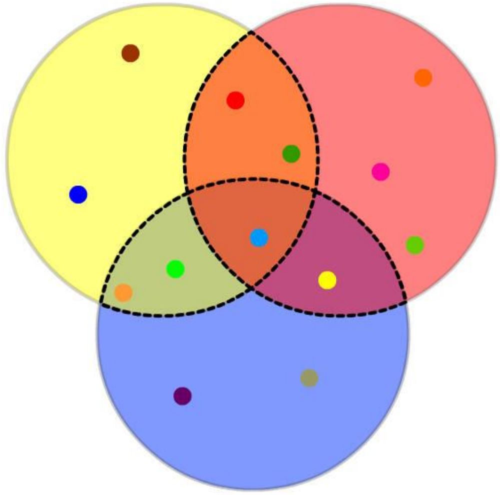
genesis text corpus has eight text collections, which are isolated in structure.



Each text collection is tagged to a specific category or genre.

E.g.: Brown text corpus contains 500 collections, which are categorized into 15 genres.

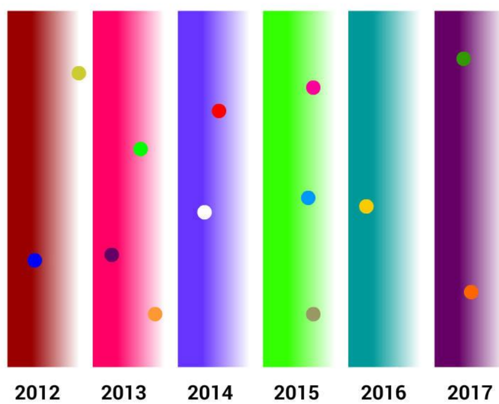
Overlapping Text Corpus



Each collection is categorized into one or more genre.

E.g.: Reuters corpus contains 10788 collections, which are tagged to 90 genre.

Temporal Text Corpus.



Each text collection is tagged to a period of time.

E.g.: inaugural corpus contain text collections corresponding to U.S inaugural presidential speeches, gathered over a period of time.

Loading User Specific Corpus

Now let's see how to convert your collection of text files into a text corpus. Suppose, you have three files c1.txt, c2.txt and c3.txt in /usr/home/dict path. Creation of corpus wordlists corpus is shown in the following example.

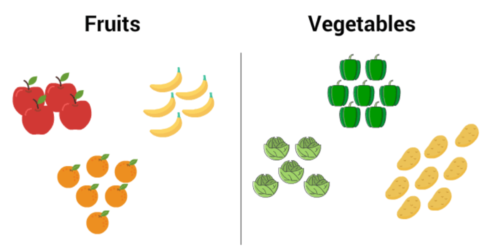
```
>>> from nltk.corpus import PlaintextCorpusReader
>>> corpus_root = '/usr/share/dict'
>>> wordlists = PlaintextCorpusReader(corpus_root, '.*')
```

```
>>> wordlists.fileids()
['c1.txt',
 'c2.txt',
 'c3.txt']
```

Conditional Frequency Distributions

In the previous topic, you have studied about Frequency Distributions. FreqDist function computes the frequency of each item in a list. While computing a frequency distribution, you observe occurrence count of an event.

```
>>> items = ['apple', 'apple', 'kiwi', 'cabbage', 'cabbage', 'potato']
>>> nltk.FreqDist(items)
FreqDist({'apple': 2, 'cabbage': 2, 'kiwi': 1, 'potato': 1})
```



A **Conditional Frequency** is a collection of frequency distributions, computed based on a condition.

For computing a conditional frequency, you have to attach a condition to every occurrence of an event. Let's consider the following list for computing Conditional Frequency.

```
>>> c_items = [('F','apple'), ('F','apple'), ('F','kiwi'), ('V','cabbage'
), ('V','cabbage'), ('V','potato') ]
```

Each item is grouped either as a fruit F or a vegetable V.

Computing conditional Frequency

ConditionalFreqDist function of nltk is used to compute Conditional Frequency Distribution (CDF).

The same can be viewed in the following example.

```
>>> cfd = nltk.ConditionalFreqDist(c_items)
>>> cfd.conditions()
['V', 'F']
>>> cfd['V']
FreqDist({'cabbage': 2, 'potato': 1})
>>> cfd['F']
FreqDist({'apple': 2, 'kiwi': 1})
```

Example	Description
`cfdist = ConditionalFreqDist(pairs)`	Create a conditional frequency distribution from a list of pairs
`cfdist.conditions()`	Alphabetically sorted list of conditions
`cfdist[condition]`	The frequency distribution for this condition
`cfdist[condition][sample]`	Frequency for the given sample for this condition
`cfdist.tabulate()`	Tabulate the conditional frequency distribution
`cfdist.tabulate(samples, conditions)`	Tabulation limited to the specified samples and conditions
`cfdist.plot()`	Graphical plot of the conditional frequency distribution
`cfdist.plot(samples, conditions)`	Graphical plot limited to the specified samples and conditions
`cfdist1 < cfdist2`	Test if samples in `cfdist1` occur less frequently than in `cfdist2`

Now let's determine the frequency of words, of a particular genre, in brown corpus.

```
>>> cfd = nltk.ConditionalFreqDist([
(genre, word)
for genre in brown.categories()
for word in brown.words(categories=genre) ])
```

The conditions applied can be viewed as shown below.

```
>>> cfd.conditions()
['adventure',
 'hobbies',
 ...]
```

Viewing Word Count

Once after computing conditional frequency distribution, tabulate method is used for viewing the count along with arguments conditions and samples.

```
>>> cfd.tabulate(conditions=['government', 'humor', 'reviews'], samples=[
'leadership', 'worship', 'hardship'])
```

```
            leadership  worship  hardship
government          12        3         2
     humor           1        0         0
   reviews          14        1         2
```

The cumulative count for different conditions is found by setting cumulative argument value to True.

```
>>> cfd.tabulate(conditions=['government', 'humor', 'reviews'], samples=[
'leadership', 'worship', 'hardship'], cumulative = True)


            leadership   worship  hardship
government          12        15        17
     humor           1         1         1
   reviews          14        15        17
```

Accessing Individual Frequency Distributions

From the obtained conditional frequency distribution, you can access individual frequency distributions.

The below example extracts frequency distribution of words present in news genre of brown corpus.

```
>>> news_fd = cfd['news']
>>> news_fd.most_common(3)
[('the', 5580), (',', 5188), ('.', 4030)]
```

You can further access count of any sample as shown below.

```
>>> news_fd['the']
5580
```

Comparing Frequency Distributions

Now let's see another example, which computes the frequency of last character appearing in all names associated with males and females respectively and compares them.

The text corpus names contain two files male.txt and female.txt.

```
>>> from nltk.corpus import names
```

```
>>> nt = [(fid.split('.')[0], name[-1])     for fid in names.fileids()
  for name in names.words(fid) ]
>>> cfd2 = nltk.ConditionalFreqDist(nt)
>>> sum([cfd2['female'][x] for x in cfd2['female']]) > sum([cfd2['male'][
x] for x in cfd2['male']])
True
```

The expression sum([cfd2['female'][x] for x in cfd2['female']]) > sum([cfd2['male'][x] for x in cfd2['male']]) checks if the last characters in females occur more frequently than the last characters in males.

The following code snippet displays frequency count of characters a and e in females and males, respectively.

```
>>> cfd2.tabulate(samples=['a', 'e'])
          a    e
female 1773 1432
  male   29  468
```

You can observe a significant difference in frequencies of a and e.

Raw Text Processing

For most of the NLTK studies that you carry out, data is not readily available in the form of a text corpus.

Also, raw text data from a different source can be obtained, processed and used for doing NLTK studies.

Some of the processing steps that you perform are:

- Tokenization
- Stemming

Reading a Text File

In this topic, you will understand how data is read from different external sources. The following example reads content from a text file, available at Project Gutenberg site.

```
>>> from urllib import request
>>> url = "http://www.gutenberg.org/files/2554/2554-0.txt"
>>> content1 = request.urlopen(url).read()
```

Reading a HTML file

The following example reads content from a news article available over the web. Beautifulsoup module is used for scrapping the required text from the webpage.

```
>>> from urllib import request

>>> url = "http://www.bbc.com/news/health-42802191"
>>> html_content = request.urlopen(url).read()

>>> from bs4 import BeautifulSoup
>>> soup = BeautifulSoup(html_content, 'html.parser')
```

```
inner_body = soup.find_all('div', attrs={'class':'story-body__inner'})

inner_text = [elm.text for elm in inner_body[0].find_all(['h1', 'h2', 'p'
, 'li']) ]

text_content2 = '\n'.join(inner_text)
```

find_all method returns all inner elements of div element, having class attribute value as story-body__inner.

Reading from Other Sources

You can also read text from some other text resources such as RSS feeds, FTP repositories, local text files, etc.

It is also possible to read a text in binary format, from sources like Microsoft Word and PDF.

Third party libraries such as pywin32, pypdf are required for accessing Microsoft Word or PDF documents.

Tokenization

Natural Language Processing in Python

Tokenization is a step in which a text is broken down into words and punctuation.

The simplest way of tokenizing is by using word_tokenize method. The below example tokenizes text read from Project Gutenberg.

```
>>> text_content1 = content1.decode('unicode_escape')  # Converts bytes t
o unicode

>>> tokens1 = nltk.word_tokenize(text_content1)

>>> tokens1[3:8]

['EBook', 'of', 'Crime', 'and', 'Punishment']
```

The following example tokenizes text scrapped from the HTML page.

```
>>> tokens2 = nltk.word_tokenize(text_content2)

>>> tokens2[:5]

['Smokers', 'need', 'to', 'quit', 'cigarettes']

>>> len(tokens2)

751
```

Regular Expressions for Tokenization

Regular expressions can also be utilized to split the text into tokens.

The below example splits the entire text text_content2 with regular expression \w+

```
>>> tokens2_2 = re.findall(r'\w+', text_content2)

>>> len(tokens2_2)
```

```
668
```

nltk contains the function regexp_tokenize, which can be used similarly to re.findall and produce the tokens.

```
>>> pattern = r'\w+'
>>> tokens2_3 = nltk.regexp_tokenize(text_content2, pattern)
>>> len(tokens2_3)
668
```

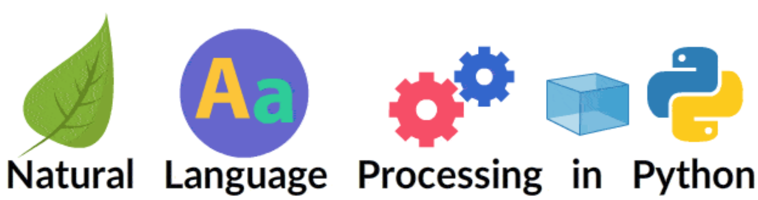
Creation of NLTK text

Using the obtained list of tokens, an object of NLTK text can be created as shown below.

```
>>> input_text2 = nltk.Text(tokens2)
>>> type(input_text2)
nltk.text.Text
```

Thus obtained text can be used for further linguistic processing.

Bigrams, Ngrams and Collocations



Bigrams represent a set of two consecutive words appearing in a text.

bigrams function is called on tokenized words, as shown in the following example, to obtain bigrams.

```
>>> import nltk
>>> s = 'Python is an awesome language.'
>>> tokens = nltk.word_tokenize(s)
>>> list(nltk.bigrams(tokens))
[('Python', 'is'),
 ('is', 'an'),
 ('an', 'awesome'),
 ('awesome', 'language'),
 ('language', '.')]
```

Computing Frequent Bigrams

Now let's find out three frequently occurring bigrams, present in english-kjv collection of genesis corpus.

Let's consider only those bigrams, whose words are having a length greater than 5.

```
>>> eng_tokens = genesis.words('english-kjv.txt')
>>> eng_bigrams = nltk.bigrams(eng_tokens)
>>> filtered_bigrams = [ (w1, w2) for w1, w2 in eng_bigrams if len(w1) >=
5 and len(w2) >= 5 ]
```

After computing bi-grams, the following code computes frequency distribution and displays three most frequent bigrams.

```
>>> eng_bifreq = nltk.FreqDist(filtered_bigrams)
>>> eng_bifreq.most_common(3)
[(('their', 'father'), 19),
 (('lived', 'after'), 16),
 (('seven', 'years'), 15)]
```

Now let's see an example which determines the two most frequent words occurring after living are determined.

```
>>> from nltk.corpus import genesis
>>> eng_tokens = genesis.words('english-kjv.txt')
>>> eng_bigrams = nltk.bigrams(eng_tokens)
```

```
>>> eng_cfd = nltk.ConditionalFreqDist(eng_bigrams)
>>> eng_cfd['living'].most_common(2)
[('creature', 7), ('thing', 4)]
```

Now let's define a function named generate, which returns words occurring frequently after a given word.

```
>>> def generate(cfd, word, n=5):
...     n_words = []
...     for i in range(n):
...       n_words.append(word)
...       word = cfd[word].max()
...     return n_words
```

Generating Most Frequent Next Word

After defining the function generate, it is called with eng_cfd and living parameters.

```
>>> generate(eng_cfd, 'living')
['living', 'creature', 'that', 'he', 'said']
```

The output shows a word which occurs most frequently next to living is creature. Similarly that occurs more frequently after creature and so on.

Trigrams

Similar to Bigrams, Trigrams refers to set of all three consecutive words appearing in text.

```
>>> s = 'Python is an awesome language.'
>>> tokens = nltk.word_tokenize(s)
>>> list(nltk.trigrams(tokens))
[('Python', 'is', 'an'),
 ('is', 'an', 'awesome'),
 ('an', 'awesome', 'language'),
 ('awesome', 'language', '.')]
```

nltk also provides the function ngrams. It can be used to determine a set of all possible n consecutive words appearing in a text.

The following example displays a list of four consecutive words appearing in the text s.

```
>>> list(nltk.ngrams(tokens, 4))
[('Python', 'is', 'an', 'awesome'),
 ('is', 'an', 'awesome', 'language'),
 ('an', 'awesome', 'language', '.')]
```

Collocations

A collocation is a pair of words that occur together, very often. For example, red wine is a collocation.

One characteristic of a collocation is that the words in it cannot be substituted with words having similar senses.

For example, the combination maroon wine sounds odd.

Generating Collocations

Now let's see how to generate collocations from text with the following example.

```
>>> from nltk.corpus import genesis
>>> tokens = genesis.words('english-kjv.txt')
>>> gen_text = nltk.Text(tokens)
>>> gen_text.collocations()
said unto; pray thee; thou shalt;
....
```

Stemming

Stemming is a process of stripping affixes from words.

More often, you normalize text by converting all the words into lowercase. This will treat both words The and the as same.

With stemming, the words playing, played and play will be treated as single word, i.e. play.

Stemmers in nltk

nltk comes with few stemmers. The two widely used stemmers are Porter and Lancaster stemmers. These stemmers have their own rules for string affixes. The following example demonstrates stemming of word builders using PorterStemmer.

```
>>> from nltk import PorterStemmer
>>> porter = nltk.PorterStemmer()
>>> porter.stem('builders')
builder
```

Now let's see how to use LancasterStemmer and stem the word builders.

```
>>> from nltk import LancasterStemmer
>>> lancaster = LancasterStemmer()
>>> lancaster.stem('builders')
build
```

Lancaster Stemmer returns build whereas Porter Stemmer returns builder.

Normalizing with Stemming

Let's consider the text collection, text1.

Let's first determine the number of unique words present in original text1.

Then normalize the text by converting all the words into lower case and again determine the number of unique words.

```
>>> from nltk.book import *
>>> len(set(text1))
19317
>>> lc_words = [ word.lower() for word in text1]
>>> len(set(lc_words))
17231
```

Now let's further normalize text1 with Porter Stemmer.

```
>>> from nltk import PorterStemmer
>>> porter = PorterStemmer()
>>> p_stem_words = [porter.stem(word) for word in set(lc_words) ]
>>> len(set(p_stem_words))
10927
```

The above output shows that, after normalising with Porter Stemmer, the text1 collection has 10927 unique words.

Now let's normalise with Lancaster stemmer and determine the unique words of text1.

```
>>> from nltk import LancasterStemmer
>>> lancaster = LancasterStemmer()
>>> l_stem_words = [lancaster.stem(word) for word in set(lc_words) ]
>>> len(set(l_stem_words))
9036
```

Applying Lancaster Stemmer to text1 collection resulted in 9036 words.

Understanding Lemma

Lemma is a lexical entry in a lexical resource such as word dictionary.

You can find multiple Lemma's with the same spelling. These are known as homonyms.

For example, consider the two Lemma's listed below, which are homonyms.

```
1. saw [verb] - Past tense of see
2. saw [noun] - Cutting instrument
```

Lemmatization

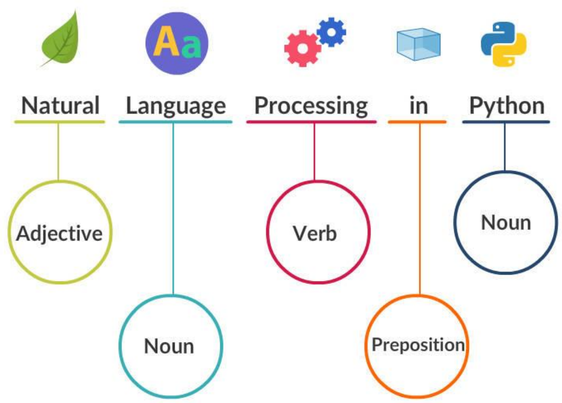
nltk comes with WordNetLemmatizer. This lemmatizer removes affixes only if the resulting word is found in lexical resource, Wordnet.

```
>>> wnl = nltk.WordNetLemmatizer()
>>> wnl_stem_words = [wnl.lemmatize(word) for word in set(lc_words) ]
>>> len(set(wnl_stem_words))
```

```
15168
```

WordNetLemmatizer is majorly used to build a vocabulary of words, which are valid Lemmas.

POS Tagging



The method of categorizing words into their parts of speech and then labeling them respectively is called POS Tagging.

A POS Tagger processes a sequence of words and tags a part of speech to each word.

pos_tag is the simplest tagger available in nltk.

The below example shows usage of pos_tag.

```
>>> import nltk
>>> text = 'Python is awesome.'
>>> words = nltk.word_tokenize(text)
>>> nltk.pos_tag(words)
[('Python', 'NNP'),
 ('is', 'VBZ'),
 ('awesome', 'JJ'),
 ('.', '.')]
```

The words Python, is and awesome are tagged to Proper Noun (NNP), Present Tense Verb (VB), and adjective (JJ) respectively.

You can read more about the pos tags with the below help command

```
>>> nltk.help.upenn_tagset()
```

To know about a specific tag like JJ, use the below-shown expression

```
>>> nltk.help.upenn_tagset('JJ')
JJ: adjective or numeral, ordinal
```

Tagging Text

Constructing a list of tagged words from a string is possible.

A tagged word or token is represented in a tuple, having the word and the tag.

In the input text, each word and tag are separated by /.

```
>>> text = 'Python/NN is/VB awesome/JJ ./.'
>>> [ nltk.tag.str2tuple(word) for word in text.split() ]
[('Python', 'NN'),
 ('is', 'VB'),
 ('awesome', 'JJ'),
 ('.', '.')]
```

Tagged Corpora

Many of the text corpus available in nltk, are already tagged to their respective parts of speech.

tagged_words method can be used to obtain tagged words of a corpus.

The following example fetches tagged words of brown corpus and displays few.

```
>>> from nltk.corpus import brown
>>> brown_tagged = brown.tagged_words()
>>> brown_tagged[:3]
[('The', 'AT'),
 ('Fulton', 'NP-TL'),
 ('County', 'NN-TL')]
```

DefaultTagger

DefaultTagger assigns a specified tag to every word or token of given text.

An example of tagging NN tag to all words of a sentence, is shown below.

```
>>> import nltk
>>> text = 'Python is awesome.'
>>> words = nltk.word_tokenize(text)
>>> default_tagger = nltk.DefaultTagger('NN')
>>> default_tagger.tag(words)
[('Python', 'NN'),
 ('is', 'NN'),
 ('awesome', 'NN'),
 ('.', 'NN')]
```

Lookup Tagger

You can define a custom tagger and use it to tag words present in any text.

The below-shown example defines a dictionary defined_tags, with three words and their respective tags.

```
>>> import nltk
>>> text = 'Python is awesome.'
>>> words = nltk.word_tokenize(text)
>>> defined_tags = {'is':'BEZ', 'over':'IN', 'who': 'WPS'}
```

Lookup Tagger

The example further defines a UnigramTagger with the defined dictionary and uses it to predict tags of words in text.

```
>>> baseline_tagger = nltk.UnigramTagger(model=defined_tags)
>>> baseline_tagger.tag(words)
[('Python', None),
 ('is', 'BEZ'),
 ('awesome', None),
 ('.', None)]
```

Since the words Python and awesome are not found in defined_tags dictionary, they are tagged to None.

Unigram Tagger

UnigramTagger provides you the flexibility to create your taggers. Unigram taggers are built based on statistical information. i.e., they tag each word or token to most likely tag for that particular word. You can build a unigram tagger through a process known as training. Then use the tagger to tag words in a test set and evaluate the performance.

Let's consider the tagged sentences of brown corpus collections, associated with government genre.

Let's also compute the training set size, i.e., 80%.

```
>>> from nltk.corpus import brown
>>> brown_tagged_sents = brown.tagged_sents(categories='government')
>>> brown_sents = brown.sents(categories='government')
>>> len(brown_sents)
3032
>>> train_size = int(len(brown_sents)*0.8)
>>> train_size
2425
```

```
>>> train_sents = brown_tagged_sents[:train_size]
>>> test_sents = brown_tagged_sents[train_size:]
>>> unigram_tagger = nltk.UnigramTagger(train_sents)
>>> unigram_tagger.evaluate(test_sents)
0.7804399607678296
```

unigram_tagger is built by passing trained tagged sentences as argument to UnigramTagger.

The built unigram_tagger is further evaluated with test sentences.

The following code snippet shows tagging words of a sentence, taken from the test set.

```
>>> unigram_tagger.tag(brown_sents[3000])
[('The', 'AT'),
 ('first', 'OD'),
 ('step', 'NN'),
 ('is', 'BEZ'),
 ('a', 'AT'),
 ('comprehensive', 'JJ'),
```

```
('self', None),
('study', 'NN'),
....
('.', '.')]
```