

# Introduction to *NLTK*

## Inf2a Lab

Srinivasan C Janarthanam  
Original version by Ivan V. Meza-Ruiz

20 October 2008

*NLTK* is a suite of Python libraries and programs for symbolic and statistical natural language processing. It has extensive documentation, including tutorials that explain the concepts underlying the language processing tasks supported by the toolkit (<http://nltk.sourceforge.net/>.)

- ▷ *Open a terminal window, and go to the folder “MyPython” that you created during the first lab.*
- ▷ ***Launch the python shell using the command “python2.4”.** This ensures you are running the version of python needed by *NLTK*.*

## 1 Starting with *NLTK*

To load the *NLTK* libraries we use `import` statement.

▷ *Type:*

```
>>> from nltk.book import *
>>> text1
>>> text2
```

A description of a class is available using the `help` function.

▷ *Type:*

```
>>> help(text1)
```

Type `q` to finish the help mode. You can check the documentation of the *NLTK* API to see all available modules and classes in:

<http://nltk.org/doc/api/>.

## 2 Tokenization

For most kinds of linguistic processing, we need to identify and classify the words of the text (ie, associate them with their part-of-speech in each particular context). This turns out to be a non-trivial task. Here, we start by showing how texts can be tokenized (ie, divided into a sequence of word tokens).

▷ *Type:*

```
>>> text = 'Hello world! This is a test string.'
>>> text.split()
```

`split` breaks a string at each white space. As you can see, tokenization based on whitespace alone is not sufficient. (What shows this?) The method `nltk.tokenize.regexp_tokenize` uses a regular expression to specify how text should be split up. This regular expression specifies the characters that can be included in a valid word.

▷ *Type:*

```
>>> from nltk import tokenize
>>> text = "Hello. Isn't this fun?"
>>> pattern = r'\w+|[\^\w\s]+'
>>> nltk.tokenize.regexp_tokenize(text, pattern)
```

The statement `r'\w+|[\^\w\s]+'` creates a regular expression which accepts sequences formed by “word” characters, i.e., characters other than whitespace or punctuation (the subexpression `\w+`) or by characters which are neither word characters nor whitespace characters (`[\^\w\s]+'`, the sign `^` means complement). This regular expression will not deliver correct results with strings containing amounts such as `$22.40` and `44.50%`, where we might want to keep the symbol `$` and `%` attached to the number.

▷ *Type:*

```
>>> text = 'That poster costs $22.40.'
>>> nltk.tokenize.regexp_tokenize(text, pattern)
```

The union of the previous regular expression with this one: `\$\d+\.\d`, where `\d` represents a decimal digit, solves the first problem.

▷ *Type:*

```
>>> pattern = r'\w+|\$\d+\.\d+|[\^\w\s]+'
>>> nltk.tokenize.regexp_tokenize(text, pattern)
```

For more information about how to create regular expression check:

<http://www.python.org/doc/2.3/lib/re-syntax.html>.

## Exercise 1

Add to the `pattern` the option to tokenize percentages as a single tokens, for instance

- 100.00%, 10.5%, and 10.234%.

Use the union operator (`|`) to add your option to the start of `pattern`. Test with the following text:

```
>>> text = 'The interest does not exceed 8.25%.'
```

### 3 Tokens, Types and Lexical richness

`nltk.book` contains texts from the Gutenberg corpus. Let's see the contents of `text1`.

▷ *Type:*

```
>>> from nltk.book import *
>>> list(text1)
```

Let's find out the length of the text in terms of words and punctuation marks. This count gives the number of **Tokens**.

▷ *Type:*

```
>>> len(text1)
```

The count includes words (and punctuation marks) that repeat several times in the text. How do we get a list of distinct words and their count? This count is called the **Type** count (because it's the number of word types).

▷ *Type:*

```
>>> set(text1)
>>> len(set(text1))
```

Lexical richness of a given text gives a count of how many times any word in the text appears on average. We get this by dividing the token count by the type count.

▷ *Type:*

```
>>> len(text1) / len(set(text1))
```

How do we find the most frequent words in a text? Using **FreqDist**, we can find the frequency of every word in a given text.

▷ *Type:*

```
>>> from nltk.probability import FreqDist
>>> fd = FreqDist(text1)
```

To find the frequency of different types (e.g. 'the'), use `fd[type]`.

▷ *Type:*

```
>>> fd['the']
>>> fd['with']
```

To get a list of **types** in the frequency distribution, use `fd.keys()`. Alternatively, you can use `fd.sorted()` to get a list of types, sorted on frequency (most frequent type first).

▷ *Type:*

```
>>> fd.keys()
>>> fd.sorted()
>>> fd.sorted()[0:10]
```

## Exercise 2

- Find the token count, type count, lexical richness and top 15 frequent types in texts `text2`, `text3` and `text4`.
- This process counts tokens like `The` and `the` as two different types, while they are really the same. Use `s.lower()` (where `s` is a string) to normalize the words to lower case before counting for types and tokens.

Whereas a frequency distribution gives the distribution of word types across the entire text, a conditional frequency distribution gives the distribution relative to a particular context – e.g., the distribution of words found after the verb “said”. We can use `ConditionalFreqDist` to determine this. First, we pass each token of the corpus together with its previous token to the `ConditionalFreqDist` class. We call this process *training*.

▷ *Type:*

```
>>> from nltk.probability import ConditionalFreqDist
>>> cfdist = ConditionalFreqDist()

>>> prev=None
>>> for token in text1:
...     cfdist[prev].inc(token)
...     prev = token
... 
```

Notice that the first token of the corpus has no “previous word”. For this example, use `None` as the context for the first token.

We can see the distribution of words which follow a given word using the `samples()` method.

▷ *Type:*

```
>>> cfdist['said'].samples()
```

Try this with a different word of your own choice.

We can use the information stored in `ConditionalFreqDist` to create a text generator using the most frequent word following a given word. The following code creates a sentence of word length 20 starting with ‘an’.

▷ *Type:*

```
>>> word = 'an'
>>> for i in range(20):
...     print word,
...     word = cfdist[word].max()
... 
```

See what sentence you get if you start with a different word.

### Exercise 3

1. With 'an', the text generator gets stuck in a cycle within the first 20 words. Modify the program to choose the next word randomly from a list of the words following the given word. (Hint: get the word list `lwords` using `samples()` method, and then randomly choose a word from the list using `random.choice(lwords)`. Don't forget to `import` the `random` module).
2. The above program has been trained on `text1` of the Gutenberg corpus. Now, train your system on other texts (say, `text2`, `text3` or `text4`) of the Gutenberg corpus and get it to generate random text following the initial word `an`.
3. Try using the most frequent word in the text as the start word instead of `an`. (Hint: use `FreqDist`).

## 4 Tagged corpora

*NLTK* is distributed with several corpora. They can be accessed using `nltk.corpus` package. First we import the Brown Corpus, the first million-word, part-of-speech tagged electronic corpus of English. Each of the sections *ca* through *cr* represents a different genre. List of available sections can be accessed using `items`.

▷ *Type:*

```
>>> from nltk.corpus import brown
>>> brown.items
>>> help(brown)
```

With the methods `words` and `sents` we can access the corpus as a list of words and sentences, respectively.

▷ *Type:*

```
>>> brown.words('ca01')
>>> brown.sents('ca01')
>>> brown.sents('ca01')[0]
```

The first argument of `words` and `sents` methods is the item of the corpus. Try the example with `'cb01'`.

With the methods `tagged_words` and `tagged_sents` we can access the tagged text of an already-tagged corpus. Every word in the sentence is tagged with a PoS tag and is presented as a list of binary tuples containing the word and its tag (for e.g. `('The', 'AT')`).

▷ *Type:*

```
>>> print brown.tagged_words('ca01')
>>> print brown.tagged_sents('ca01')
>>> print brown.tagged_sents('ca01')[0]
```

Since the structures accessed through `nltk.corpus` are basic Python structures. We can use them to do some analysis on a corpus. For instance, we can count the number of times each part-of-speech tag occurs in the Brown corpus.

▷ *Type:*

```
>>> dict={}
>>> for sntc in brown.tagged_sents('ca01'):
...     for word,tag in sntc:
...         if tag in dict:
...             dict[tag]+=1
...         else:
...             dict[tag]=0
...
>>> for tag,count in dict.items():
...     print tag," ",count
...
```

#### Exercise 4

Create a dictionary of the words used in item 'ca01' of the Brown corpus. The key of the dictionary should be the part-of-speech tag of a word. The value of the dictionary should be the list of the words which are tagged with the part-of-speech tag. For instance, `dict['NN']` returns `['director', 'chairman']...`, if *directory* and *chairman* were labelled as 'NN'. Test the dictionary using a similar `for` loop of the previous example.

Text does not always come already PoS-tagged. In fact, it rarely does. In NLTK, there are several ways to build PoS taggers. A simple way to tag is to assign the same tag to all the words in the sentence. Using `DefaultTagger`, we can build such a tagger. The following tagger tags all words with NN tag.

▷ *Type:*

```
>>> from nltk import DefaultTagger
>>> tokens = 'John saw 3 polar bears .'.split()
>>> default_tagger = DefaultTagger('NN')
>>> default_tagger.tag(tokens)
```

But, how good is this tagger? Using `tag.accuracy()`, we can find out how accurate the tagger is with respect to the `brown` corpus.

▷ *Type:*

```
>>> from nltk import tag
>>> tag.accuracy(default_tagger, brown.tagged_sents('ca01'))
```

Can we build a better tagger? We don't want to tag the verb `said` with NN. Using `UnigramTagger`, we can build a tagger that assigns word in the training data its most frequent tag within that corpus. This tagger will assign more

appropriate tags to the words (for e.g. VBD to **said**) than the `DefaultTagger`. But, it will ignore the contextual information around the words. For instance, **frequent** will always be tagged as an adjective (JJ), even if it not an adjective according to the context (for e.g. I **frequent** this **cafe**).

▷ *Type:*

```
>>> from nltk import UnigramTagger
>>> tokens = 'John saw 3 polar bears .'.split()
>>> unigram_tagger = UnigramTagger(brown.tagged_sents('ca01'))
>>> unigram_tagger.tag(tokens)
>>> tag.accuracy(unigram_tagger, brown.tagged_sents('ca01'))
```

To take account of the previous word as context, NLTK provides a bigram tagger. Such a tagger should learn that "frequent" following "I" should be tagged as a verb (eg, "I frequent this cafe"), while following "this", it should be tagged as an adjective (eg, "this frequent annoyance"). You can learn more about a bigram tagger, which can potentially out-perform a unigram tagger, if sufficient data is provided to train it, in the NLTK book chapter on Tagging (<http://nltk.org/doc/en/ch03.html>).

## 5 Grammars

The `nltk.cfg` module defines a set of classes that are used to define context free grammars:

1. The `cfg.Nonterminal` class is used to represent nonterminals.
2. The `cfg.Production` class is used to represent CFG productions.
3. The `cfg.Grammar` class is used to represent CFGs.

`Nonterminal` is a simple class that is used to let *NLTK* distinguish terminals from nonterminals.

▷ *Type:*

```
>>> from nltk.parse import cfg
>>> S = cfg.Nonterminal('S')
>>> S

>>> NP = cfg.Nonterminal('NP')
>>> NP

>>> VP, Adj, V, N = cfg.nonterminals('VP, Adj, V, N')
>>> VP, Adj, V, N
```

Each `Production` specifies that a nonterminal (the left-hand side of a rule) can be expanded to the sequence of terminals and nonterminals given in the right-hand side of the rule.

▷ *Type:*

```

>>> prod1 = cfg.Production(S, [NP, VP])
>>> prod1

>>> prod2 = cfg.Production(NP, ['the', Adj, N])
>>> prod2

```

Context free grammars are encoded by the `Grammar` class. A Grammar consists of a special start nonterminal, and an ordered list of productions.

▷ *Type:*

```

>>> grammar = cfg.Grammar(S, [prod1, prod2])
>>> grammar

>>> grammar.start()

>>> grammar.productions()

```

Additionally, with `cfg.parse_cfg` function, it is possible to create a grammar from its text description.

▷ *Type:*

```

>>> grammar2 = cfg.parse_cfg('''
...     S -> NP VP
...     NP -> "I" | "John" | "Mary" | "Bob" | Det N
...     VP -> V NP | V NP PP
...     V -> "saw" | "ate"
...     Det -> "a" | "an" | "the" | "my"
...     N -> "dog" | "telescope" | "apple"
...     PP -> P NP
...     P -> "on" | "with"
...     ''')

>>> grammar2

>>> grammar2.start()

>>> grammar2.productions()

```

For a grammar, we can parse a sentence and get its syntactic tree using the `parse.RecursiveDescent()` function.

▷ *Type:*

```

>>> from nltk import parse
>>> sntc1 = str.split('I saw Mary')
>>> sntc2 = str.split('John ate my apple')
>>> rd_parser = parse.RecursiveDescentParser(grammar2)
>>> for p in rd_parser.nbest_parse(sntc1):
...     print p

```



```
>>> for p in rd_parser.nbest_parse(sntc2):
...     print p
```

## Exercise 5

Create a new grammar (**grammar3**) using as a base **grammar2** plus the production **NP -> Det N PP**.

1. Parse the sentence *Mary saw the dog with the telescope*. How many parse trees do you get?
2. For **grammar2**, write down two unambiguous sentences (ie, sentences which have a single syntactic tree structure with respect to the grammar), two ambiguous sentences (each of which yields more than one tree), and one ungrammatical sentences (that yields no tree at all).

## 6 Treebank

*NLTK* also includes a 10% fragment of the Wall Street Journal section of the Penn Treebank. Each sentence of the corpus is available in three forms; (1) as tokenized text, (2) as tokens labelled with part of speech, and (3) as parse trees. These can be accessed using **treebank.raw()** for the raw text, **treebank.tagged\_sents()** for the tagged text, and **treebank.parsed\_sents()** for the parse trees.

▷ *Type:*

```
>>> from nltk.corpus import treebank
>>> help(treebank)
>>> treebank.items

>>> print treebank.raw('wsj_0001.mrg')

>>> print treebank.tagged_sents('wsj_0001.mrg')[0]

>>> print treebank.parsed_sents('wsj_0001.mrg')[0]
```

The argument in the above three functions, **'wsj\_0001.mrg'**, is a section of the treebank. We can see the list of all sections using **treebank.items**. The function **treebank.parsed\_sents()[0]** returns an object of class **nltk.Tree**. We can analyse the tree using different functions available in this class.

▷ *Type:*

```
>>> t = treebank.parsed_sents('wsj_0001')[0]
>>> t.node

>>> t.leaves()
```

```

>>> print t

>>> len(t)

>>> t[0]

>>> t[0].node

>>> len(t[0])

```

`t.node` contains the top-most node in the tree `t`. We can see the list of words that yields the parse tree using `t.leaves()`. Using `len(t)`, we can get the number of child nodes to `t.node`. Each of these child nodes and their subtrees can be accessed using `t[0].node` and `t[0]` respectively.

Using the above utilities, we can find out the subject of any given sentence. The following code prints the subject NP phrase in the first sentence of `'wsj_0001.mrg'`

▷ *Type:*

```

>>> t = treebank.parsed_sents('wsj_0001.mrg')[0]
>>> for ch_tree in t:
...     if (ch_tree.node.startswith('NP-SBJ')):
...         print ch_tree.leaves()

```

## Exercise 6

1. Extend the above program to identify the subject in all the sentences in `'wsj_0003.mrg'`.
2. A subordinate clause in a sentence will have its own subject. So, extend the code (using recursion) from the previous problem to identify all the subjects in every sentence.