Introduction to NLTK: Worksheet 1

Trevor Cohn and Yves Peirsman

Euromasters Summer School 2005

Python and NLTK are installed on the DICE workstations (i.e. the machines in Appleton Tower). Login using your allocated account and password, then launch a Terminal. You can either run Python by either typing `python2.3` or `idle2.3` at the command prompt. The former launches Python within the current terminal, while the latter will open a new window and also allows editing of files and debugging.

You should use either a standard Unix editor (such as emacs, vim or pico) or use idle in order to create and edit source files.

If you wish to install Python and NLTK on your own machine, you will find distributions online. Currently, the latest release of Python is 2.4.1, and is available from `http://www.python.org`. You can also retrieve NLTK from `http://nltk.sourceforge.net` – installation instructions are given at `http://nltk.sourceforge.net/install.html`.

Note: the environment variable `NLTK_CORPORA` must be set to `/usr/share/nltk-data` in order to access the corpora. This should be done automatically in your login scripts. However, if you have problems using the corpora, use `export` or `setenv` in your shell to set the variable before running Python.

Tokenization

NLTK Tokenizers convert a string into a list of `Token`s. For the tokenization exercises you will need to import the following modules:

```
>>> from nltk.token import *
>>> from nltk.tokenizer import *
>>> from nltk.corpus import *
```

1. Try creating some tokens using the built-in whitespace tokenizer:

   ```
   >>> ws = WhitespaceTokenizer(SUBTOKENS='WORDS')
   >>> sentence = Token(TEXT='My dog likes your dog')
   >>> ws.tokenize(sentence, add_locs=True)
   >>> sentence
   <[<My>@[0:2c], <dog>@[3:6c], <likes>@[7:12c], <your>@[13:17c], <dog>@[18:21c]]>
   ```

   Extract the third token's type and location using the `WORDS`, `TEXT` and `LOC` properties.

2. Next, use the corpus module to extract some tokenized data.

   ```
   >>> gutenberg.items()
   ['austen-emma.txt', 'bible-kjv.txt', ..., 'chesterton-ball.txt', ...]
   >>> text = gutenberg.read('chesterton-ball.txt')
   ```

   The text variable contains the entire novel as a list of tokens. Extract tokens 2631-2643.

3. The regular expression tokenizer `RegexpTokenizer()` was discussed in the lecture. Provide a regular expression to match 'words' containing punctuation.

   ```
   >>> t = RegexpTokenizer('your regular expression goes here')
   >>> sentence = Token(TEXT='OK, we\'ll email $20.95 payment to Amazon.com.')
   >>> t.tokenize(sentence)
   ```

4. Try out your tokenizer on the header of a Gutenberg corpus file, which contains a lot of punctuation. How many tokens are there? Is it the same as what other people get?

   ```
   >>> text = Token(TEXT=gutenberg.raw_read('chesterton-ball.txt')[:1020])
   >>> t.tokenize(text)
   >>> len(text['SUBTOKENS'])
   ```

Part-of-speech tagging

Tokens are often "tagged" with additional information, such as their part-of-speech. The `TAG` property is used to store POS tags, while the `TEXT` property stores the word type. For these exercises you will need to additionally import the following module:

```
>>> from nltk.tagger import *
```

1. Try creating a few tagged tokens:

   ```
   >>> chair = Token(TEXT='chair', TAG='NN')
   >>> chair
   <chair/NN>
   >>> chair = Token(TEXT='chair', TAG='NN', LOC=CharSpanLocation(1, 6))
   >>> chair
   <chair/NN>@[1:6c]
   ```

   Extract the token's type and the type's tag using token's `TEXT` and `TAG` properties.

2. Use the `TaggedTokenReader` to tokenize a tagged sentence into a list of tagged tokens. The input is a string of the form:

   ```
   >>> input = 'I/NP saw/VB a/DT man/NN'
   ```

   You will need to use the token reader's `read_tokens` method.

3. Use the corpus module to extract some tagged data. Both the Brown and Penn Treebank corpora contain tagged text. These corpora can be accessed using `brown` and `treebank`. Each corpus is divided into groups and items. Items are the logical units, usually files, into which the corpus has been split. Groups are logical groupings or subdivisions of the corpus corresponding to different sources, genre or markup for instance. The items may be listed exhaustively, or limited to only those beloning to a given group:

   ```
   >>> brown
   <Corpus: brown (contains 500 items; 15 groups)>
   >>> brown.groups()
   ['skill and hobbies', 'humor', 'popular lore', 'fiction: mystery',
   'belles-lettres', ...
   >>> brown.items('popular lore')
   ('cf01', 'cf02', 'cf03', 'cf04', 'cf05', 'cf06', 'cf07', 'cf08',
   'cf09', 'cf10', ...
   >>> brown.raw_read('cf04')
   '{\bs}n{\bs}n{\bs}t``/`` The/at food/nn is/bez wonderful/jj and/cc
   it/pps ...
   >>> brown.read('cf04')['WORDS'][:10]
   ['``'/'``'@[0w], 'The'/'AT'@[1w], 'food'/'NN'@[2w], 'is'/'BEZ'@[3w],
   ...
   ```

   Try extracting some tagged text from other items and groups of the Brown corpus and the Penn Treebank. You will need to use the `'tagged'` group of the Treebank. The tag sets used differs between the two corpora. See `http://www.scs.leeds.ac.uk/amalgam/tagsets/brown.html` and `http://www.scs.leeds.ac.uk/amalgam/tagsets/upenn.html` for descriptions of the tag sets.

4. Use the `DefaultTagger` to tag a sequence of tokens. First extract some tagged text, remove all tags using the `exclude()` method then apply the tagger. Tagging accuracy can be measured using the `accuracy()` method:

   ```
   >>> tagged_tokens = brown.read('cf04')
   >>> retagged_tokens = tagged_tokens.exclude('TAG')
   >>> default_tagger = DefaultTagger('nn', SUBTOKENS='WORDS')
   >>> default_tagger.tag(retagged_tokens)
   >>> tagged_tokens['WORDS'][205:209]
   [<home/nn>, <to/in>, <60/cd>, <children/nns>]
   >>> retagged_tokens['WORDS'][205:209]
   [<home/nn>, <to/nn>, <60/nn>, <children/nn>]
   >>> accuracy(tagged_tokens['WORDS'], retagged_tokens['WORDS'])
   0.16961913197519929
   ```

   Inspect the output (`retagged_tokens`) by hand, comparing it to the original in order to see what kind of errors were made.

5. Use the `UnigramTagger` and `NthOrderTagger` with varying order (1 or more) on the same data. These taggers need to be trained in order to initialise their probability estimates. It is best to train on different data to that tested, hence we'll use a different item:

   ```
   >>> training_tokens = brown.read('cf01')
   >>> unigram = UnigramTagger(SUBTOKENS='WORDS')
   >>> unigram.train(training_tokens)
   >>> unigram.tag(retagged_tokens)
   ```

   This tagger doesn't perform very well as it hasn't seen much training data and thus its probability estimates are quite biased. See if and how much the accuracy can be improved by increasing the amount of training data (while ensuring that you're not using the training data for testing).

6. You may have noticed that the `NthOrderTagger` failed miserably for high orders, where the `UnigramTagger` was quite robust. Why do you think this happens?

   Use a `BackoffTagger` with a unigram and nn cd tagger as shown below. Add some higher order taggers (eg. second, third order) to the start of the list of taggers. Does performance improve?

   ```
   >>> backoff = BackoffTagger([unigram, default_tagger], SUBTOKENS='WORDS')
   >>> backoff.tag(retagged_tokens)
   ```

7. Find the 10 most common tags in a group of items of the Brown corpus. Use the `nltk.probability.FreqDist` class to count the number of instances of each tag, using the `inc()` method for the tag of each token as they are processed. You may need to refer to the lecture slides and NLTK documentation on the `FreqDist` class.

Note: NLTK is installed in the subdirectories `/usr/lib/python2.3/site-packages/nltk` and `/usr/share/nltk-data`. The first contains the source code to NLTK, and the latter contains the corpus data.