The Natural Language Toolkit (NLTK)

Markus Dickinson

Dept. of Linguistics, Indiana University

Catapult Workshop Series; March 8, 2013

NLTK

Natural Language Toolkit (NLTK) is:

> *Open source Python modules, linguistic data and documentation for research and development in natural language processing and text analytics, with distributions for Windows, Mac OSX and Linux.*

http://www.nltk.org/

Today, we'll look at:

- Some basic functionality for working with text files
  - http://nltk.org/book/ch01.html
  - http://nltk.org/book/ch03.html
- One example of an NLP process, POS tagging
  - http://nltk.org/book/ch05.html

Where we're going

NLTK is a package written in the programming language Python, providing a lot of tools for working with text data

Goals: By the end of today, you should be:

- Familiar enough with Python to work with NLTK
- Familiar with NLTK, so as to be able to:
  - Use their pre-installed data files
  - Import your own text data files
  - Employ basic data manipulation
  - Part-of-speech (POS) tag your data
- Comfortable with NLTK, so as to be able to teach yourself more on other NLP applications, e.g.,
  - Classification
  - Parsing, Chunking, & Grammar Writing
  - Propositional Semantics & Logic

Python

NLTK is based on Python

- We will assume Python 2.7 for now, but Python 3 is the way to go for the future ...
  - Python 3 is Unicode all the way through, allowing for easy handling of various languages
- Python is a full programming language

Python has two modes:

- **Interactive** → our focus today
- File-based

There are a lot of good Python resources out there:

- The main Python tutorial: http://www.python.org/doc/current/tut/
- Code Academy: http://www.codecademy.com/tracks/python
- etc.

Interactive Python

To start, type `python` in a terminal or command prompt

- Better yet might be to use the Interactive DeveLopment Environment (IDLE)

```
> python
Python 2.7.2 (default, Jun 20 2012, 16:23:33)
[GCC 4.2.1 Compatible Apple Clang 4.0 (tags/Apple/clang-418.0.60)]
Type "help", "copyright", "credits" or "license" for more informati
>>>
```

Numbers & Strings

Some uses of numbers:

```
>>> 2+2
4
>>> 3/2.
1.5
```

Some uses of strings:

- single quotes: `'string'`
- double quotes: `"string"`
- There are string characters with special meaning: e.g., `\n` (newline) and `\t` (tab)

String indices & slices

You can use slices to get a part of a string

```
>>> s = "happy"
>>> len(s)  # use the len function
5
>>> s[3]    # indexed from 0, so 4th character
'p'
>>> s[1:3]  # characters 1 and 2
'ap'
>>> s[:3]   # first 3 characters
'hap'
>>> s[3:]   # everything except first 3 characters
'py'
>>> s[-4]   # 4th character from the back
'a'
```

Variables

Definition

A variable is a name that refers to some value (could be a number, a string, a list etc.)

1. Store the value 42 in a variable named *foo*
   ```
   foo = 42
   ```
2. Store the value of foo+10 in a variable named bar
   ```
   bar = foo + 10
   ```

Installing NLTK

Installing NLTK is pretty straightforward:

- http://nltk.org/install.html
- I recommend installing Numpy, but that can sometimes present challenges

Getting Started

Download the materials from the NLTK book:

```
>>> import nltk
>>> nltk.download()
...
Downloader> d book
...
```

This command gives us various texts to work with, which we need to load:

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

Searching Text

We now have texts available:

```
>>> text1
<Text: Moby Dick by Herman Melville 1851>
```

Methods which do some basic analysis:

- `concordance`
  `text1.concordance("monstrous")`
- `similar`
  `text1.similar("monstrous")`
- `common_contexts`
  `text2.common_contexts(["monstrous", "very"])`

Texts as Lists of Words

NLTK treats texts as lists of words (more on lists in a bit):

Here are the first 20 words of *Moby Dick*

```
>>> text1[:20]
['[', 'Moby', 'Dick', 'by', 'Herman', 'Melville',
'1851', ']', 'ETYMOLOGY', '.', '(', 'Supplied',
'by', 'a', 'Late', 'Consumptive', 'Usher', 'to',
'a', 'Grammar']
```

Counting Vocabulary

Because it's Python-based, it's easy to create functions to analyze the texts

```
>>> from __future__ import division
>>> def lexical_diversity(text):
...     return len(text) / len(set(text))
...
>>> lexical_diversity(text1)
13.502044830977896
>>> lexical_diversity(text2)
20.719449729255086
```

Note: `set()` converts a list to a set

- If you're not familiar with sets, check 'em out! (http://docs.python.org/2/tutorial/datastructures.html#sets)

Lists in Python

Let's detour back to Python to talk about lists:

- Lists are containers for more than one element
  - example: `employee = [Markus', Dickinson', 'assistant prof', 'MM317']`
  - empty list: `[]`
- Each element in the sequence is assigned a position number, an **index** (starting from 0)
  - example: `employee[1]`
- **Indexing**: accessing elements in a list
  - `greeting = ['hi', 'there', 'partner']`
    `greeting[2]`
- Adding lists:
  `long_greeting = greeting + ['how', 'are', 'you']`

Slicing

- accessing parts of segments is called **slicing**
  - example:
    `long_greeting[3:6]`
  - the slice starts at the first index and goes up to the second (non-inclusive)!
- going all the way to the end:
  `long_greeting[3:]`
- starting at the beginning:
  `long_greeting[:3]`
- steps are given as optional third number:
  `long_greeting[1:6:2]`

Operations on Lists

- membership:
  `employee = ['Markus, Dickinson', 'assistant prof', 'MM317']`
  `'MM317' in employee`
- check length:
  `len(employee)`
- add at the end: append
  `employee.append('Computational Linguistics')`
- retrieve from the end: pop
  `employee.pop()`
  - This returns a value!
- add at the beginning:
  `employee.insert(0, 'Linguistics')`
- retrieve from the beginning:
  `employee.pop(0)`

Simple Statistics

Frequency Distributions

NLTK has pre-built packages for creating distributions

```
>>> fdist1 = FreqDist(text1)
>>> fdist1
<FreqDist with 19317 samples and 260819 outcomes>
>>> fdist1['whale']
906
```

You could build your own dictionaries, but some capabilities are quickly calculated with `FreqDist()`:

```
>>> vocabulary1 = fdist1.keys()
>>> vocabulary1[:10]
[',', 'the', '.', 'of', 'and', 'a', 'to', ';',
 'in', 'that']
```

Organizing by word length

```
>>> fdist = FreqDist([len(w) for w in text1])
>>> fdist
<FreqDist with 19 samples and 260819 outcomes>
>>> fdist.keys()
[3, 1, 4, 2, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, ... ]
>>> fdist.items()
[(3, 50223), (1, 47933), (4, 42345), (2, 38513), ... ]
>>> fdist.max()
3
>>> fdist[3]
50223
>>> fdist.freq(3)
0.19255882431878046
```

Control structures

In the previous slide, what was happening here:

- `[len(w) for w in text1]`

To answer this, we need to discuss control structures:

- Conditionals: `if`/`elif`/`else`
- Loops: `for` and `while`
  - We'll only look at `for` today

If Statement

- syntax:

```
if <test>:
    do this
```

- full program:

```
known_users = ['Sandra', 'Markus']
name = raw_input('type your name: ')

if name in known_users:
    print 'Hello ' + name
```

Truth Values

- a test (in the if statement) corresponds to a yes/no question and can be either true or false
- the following values count as false:

```
False
None
0
[]  (empty list)
{}  (empty dict)
''  (empty string)
()  (empty tuple)
```

- everything else counts as true!

Else Statements

- In case the program needs to do something when the test is false, use the `else:` statement
- E.g. if a user is not known, add him/her to the list

Example

```
known_users = ['Sandra', 'Markus']
name = raw_input('type␣your␣name:␣')

if name in known_users:
    print 'Hello␣' + name + '.'
    print 'It␣is␣nice␣to␣have␣you␣back.'
else:
    known_users.append(name)
    print 'You␣have␣been␣added␣to␣the␣list.'
```

Elif

- if you want to check the next condition in the else case, there is a shortcut for *else if* called `elif`

Example

```
known_users = ['Sandra', 'Markus']
name = raw_input('type your name: ')

if name in known_users:
    print 'Hello ' + name + '.'
    print 'It is nice to have you back.'
elif len(name) > 20:
   print 'Your name is too long!'
else:
    known_users.append(name)
    print 'You have been added to the list.'
```

Tests

`x == y`	x equals y
`x < y`	x is less than y
`x > y`	x is greater than y
`x >= y`	x is greater than or equal to y
`x <= y`	x is less than or equal to y
`x != y`	x is not equal to y
`x is y`	x is the same object as y
`x is not y`	x is not the same object as y
`x in y`	x is a member of y
`x not in y`	x is not a member of y

Word comparison tests

`s.startswith(t)`	test if `s` starts with `t`
`s.endswith(t)`	test if `s` ends with `t`
`t in s`	test if `t` is contained inside `s`
`s.islower()`	test if all cased characters in `s` are lowercase
`s.isupper()`	test if all cased characters in `s` are uppercase
`s.isalpha()`	test if all characters in `s` are alphabetic
`s.isalnum()`	test if all characters in `s` are alphanumeric
`s.isdigit()`	test if all characters in `s` are digits
`s.istitle()`	test if `s` is titlecased (all words in `s` have have initial capitals)

For Loops

Iteration

`for` loops allow us to iterate over each element of a set or sequence

Syntax:

```
for <var> in <set>:
    do ...
    do ...
```

Example

```
words = ['a', 'rose', 'is', 'a', 'rose', 'is',
                    'a', 'rose']
for w in words:
    print w
```

List comprehensions

Python has a cool shorthand called **list comprehensions** for creating new lists from old ones:

- `a = [1,2,3,4,5]`
  `b = [x**2 for x in a]`
  b is set to `[1, 4, 9, 16, 25]`

So: `[len(w) for w in text1]` gives a list of word lengths

- What does this do?

```
sorted([w for w in set(text1)
          if w.endswith('ableness')])
```

Functions

Returning to NLTK functions ...

- Get `bigrams` from a text (or list):

```
>>> bigrams(text1[:10])
[('[', 'Moby'), ('Moby', 'Dick'), ('Dick', 'by'),
('by', 'Herman'), ('Herman', 'Melville'),
('Melville', '1851'), ('1851', ']'),
(']', 'ETYMOLOGY'), ('ETYMOLOGY', '.')]
```

- Get the most frequent `collocations`:

```
>>> text1.collocations()
Building collocations list
Sperm Whale; Moby Dick; White Whale; old man;
Captain Ahab; sperm whale; Right Whale;
Captain Peleg; New Bedford; Cape Horn; cried Ahab;
years ago; lower jaw; never mind; Father Mapple; ...
```

Using your own data

Using `.read()`, you can read a text file as a string in Python

- With a string representation, you can use NLTK's utilities

`raw` is *Crime and Punishment*, from Project Gutenberg

```
>>> raw = open('crime.txt').read()
>>> tokens = nltk.word_tokenize(raw)
>>> tokens[:10]
['The', 'Project', 'Gutenberg', 'EBook', 'of',
 'Crime', 'and', 'Punishment', ',', 'by']
```

- `open()` opens a file & `read()` converts it to a string

Creating an NLTK text

`nltk.Text()` creates a NLTK text, with all its internal methods available:

```
>>> text = nltk.Text(tokens)
>>> type(text)
<class 'nltk.text.Text'>
>>> text[:10]
['The', 'Project', 'Gutenberg', 'EBook', 'of',
'Crime', 'and', 'Punishment', ',', 'by']
>>> text.collocations()
Building collocations list
Katerina Ivanovna; Pyotr Petrovitch;
Pulcheria Alexandrovna; Avdotya Romanovna;
Marfa Petrovna; Rodion Romanovitch;
Sofya Semyonovna; old woman; Project Gutenberg-tm;
Porfiry Petrovitch; Amalia Ivanovna; great deal; ...
```

Managing corpora in NLTK

There is much more you can do to use your own corpus data in NLTK

- Some of this involves using Corpus Readers
- See: http://nltk.googlecode.com/svn/trunk/doc/howto/corpus.html

Stemming

There are options for normalizing words, as well

```
>>> porter = nltk.PorterStemmer()
>>> lancaster = nltk.LancasterStemmer()
>>> [porter.stem(t) for t in tokens]
['DENNI', ':', 'Listen', ',', 'strang',
 'women', 'lie', ...]
>>> [lancaster.stem(t) for t in tokens]
['den', ':', 'list', ',', 'strange',
 'wom', 'lying', ...]
```

POS Tagging

We can use NLTK to perform a variety of NLP tasks

- We will quickly cover the utilities for POS tagging
- Other modules include:
  - Classification
  - Parsing, Chunking, & Grammar Writing
  - Propositional Semantics & Logic

Segmentation & Tokenization

As we saw, you can use `nltk.word_tokenize()` to break a sentence into tokens

- `nltk.sent_tokenize` breaks a text into sentences

```
>>> nltk.sent_tokenize("Hello, you fool.  I love\
... you.  Come join the joyride.")
['Hello, you fool.', 'I love you.',
  'Come join the joyride.']
```

Basic NLTK tagging

A very basic way to tag:

```
>>> import nltk
text = nltk.word_tokenize("They refuse to permit us
                          to obtain the refuse permit")
>>> nltk.pos_tag(text)
    [('They', 'PRP'), ('refuse', 'VBP'),
    ('to', 'TO'), ('permit', 'VB'), ('us', 'PRP'),
    ('to', 'TO'), ('obtain', 'VB'), ('the', 'DT'),
    ('refuse', 'NN'), ('permit', 'NN')]
```

Representing tagged tokens

NLTK uses tuples to represent word, tag pairs:

```
>>> tagged_token = nltk.tag.str2tuple('fly/NN')
>>> tagged_token
('fly', 'NN')
>>>
>>> sent = 'They/PRP refuse/VBP to/TO permit/VB
           us/PRP to/TO obtain/VB the/DT
           refuse/NN permit/NN'
>>> [nltk.tag.str2tuple(t) for t in sent.split()]
[('They', 'PRP'), ('refuse', 'VBP'), ('to', 'TO'),
('permit', 'VB'), ('us', 'PRP'), ('to', 'TO'),
('obtain', 'VB'), ('the', 'DT'), ('refuse', 'NN'),
('permit', 'NN')]
```

Reading tagged corpora

NLTK has a variety of corpora to work with (see http://nltk.org/book/ch02.html)

```
>>> nltk.corpus.brown.tagged_words()
[('The', 'AT'), ('Fulton', 'NP-TL'), ...]
>>> nltk.corpus.brown.tagged_words(simplify_tags=True)
[('The', 'DET'), ('Fulton', 'NP'), ('County', 'N'), ...]
```

Corpus reading options

Ways to access information for tagged corpora:

- `.words()`
  [list of words]
- `.tagged_words()`
  [list of (word,tag) pairs]
- `.sents()`
  [list of list of words]
- `.tagged_sents()`
  [list of list of (word,tag) pairs]
- `.paras()`
  [list of list of list of words]
- `.tagged_paras()`
  [[list of list of list of (word,tag) pairs]

Automatic POS tagging

Most Frequent Tag Tagger

`nltk.DefaultTagger()`:

```
>>> raw = 'I do not like green eggs and ham, I \
          do not like them Sam I am!'
>>> tokens = nltk.word_tokenize(raw)
>>> default_tagger = nltk.DefaultTagger('NN')
>>> default_tagger.tag(tokens)
[('I', 'NN'), ('do', 'NN'), ('not', 'NN'), ...]
```

For stored data (lists of lists of word/tag pairs), you can use `.evaluate()`

```
>>> brown_tagged_sents =
    brown.tagged_sents(categories='news')
>>> default_tagger.evaluate(brown_tagged_sents)
0.13089484257215028
```

Automatic POS tagging

Regular Expression Tagger

Regular expressions capture patterns compactly

```
patterns = [
...     (r'.*ing$', 'VBG'),       # gerunds
...     (r'.*ed$', 'VBD'),        # simple past
...     (r'.*es$', 'VBZ'),        # 3rd sg. pres.
...     (r'.*ould$', 'MD'),       # modals
...     (r'.*\'s$', 'NN$'),       # possessive nouns
...     (r'.*s$', 'NNS'),         # plural nouns
...     (r'^-?[0-9]+(.[0-9]+)?$', 'CD'),  # cardinal #s
...     (r'.*', 'NN')             # nouns (default)
... ]
>>> regexp_tagger = nltk.RegexpTagger(patterns)
```

Note that the patterns are applied *in order*

Automatic POS tagging

Regular Expression Tagger (2)

```
>>> brown_sents = brown.sents(categories='news')
>>> regexp_tagger.tag(brown_sents[3])
[('``', 'NN'), ... ('such', 'NN'),
 ('reports', 'NNS'), ... ('considering', 'VBG'),
 ('the', 'NN'), ...]
>>>
>>> regexp_tagger.evaluate(brown_tagged_sents)
0.20326391789486245
```

N-gram tagging

Unigram tagging

`nltk.UnigramTagger` learns the most frequent tag for every word:

```
>>> size = int(len(brown_tagged_sents) * 0.9)
>>> size
4160
>>> train_sents = brown_tagged_sents[:size]
>>> test_sents = brown_tagged_sents[size:]
>>> unigram_tagger = nltk.UnigramTagger(train_sents)
>>> unigram_tagger.evaluate(test_sents)
0.811023622047244l
```

N-gram tagging

Bigram tagging

`nltk.BigramTagger` learns the most frequent tag for every bigram:

```
>>> bigram_tagger = nltk.BigramTagger(train_sents)
>>> bigram_tagger.tag(brown_sents[2007])
[('Various', 'JJ'), ('of', 'IN'), ('the', 'AT'), ...]
>>>
>>> bigram_tagger.evaluate(test_sents)
0.10216286255357321
```

Note that bigrams which are unseen are assigned nothing

N-gram tagging

Combining taggers

Use the best information if you have it:

```
>>> t0 = nltk.DefaultTagger('NN')
>>> t1 = nltk.UnigramTagger(train_sents,backoff=t0)
>>> t2 = nltk.BigramTagger(train_sents,backoff=t1)
>>> t2.evaluate(test_sents)
0.8447124489185687
```

Unknown words can (also) be handled via regular expressions and be better integrated into contextual information