

Natural Language Processing Applications

Lecture 4: Corpus Processing and Tokenizing with NLTK

Claire Gardent

CNRS/LORIA
Campus Scientifique,
BP 239,
F-54 506 Vandœuvre-lès-Nancy, France



2007/2008

NLTK

- ▶ Open Source Project:
<http://nltk.sourceforge.net>
- ▶ Lead developers: Steven Bird, Ed Loper, Ewan Klein
- ▶ NLTK is a set of Python modules which you can import into your programs, eg:

```
import nltk, re, pprint
```
- ▶ NLTK modules perform tasks useful for NLP (RE processing, extracting sentences, tagging, chunking, parsing, etc.) standard python code
- ▶ Latest release: NLTK-Lite 0.9b2 [12 September 2007]
- ▶ Detailed tutorials are available
- ▶ You are strongly encouraged to read the NLTK tutorials related to what is covered in the course

NLTK

NLTK corpora and corpus processing

Tokenising

Summary

NLTK Corpora

- ▶ NLTK is distributed with several *corpora*.
- ▶ A *corpus* is a body of text (or other language data, eg speech).
- ▶ Example NLTK corpora:
 - ▶ `gutenberg`: works of literature from Gutenberg project
 - ▶ `brown`: the first million word, PoS-tagged corpus, in 1961
 - ▶ `treebank`: parsed text from (part of) the Penn treebank,

Listing the files making up an NLTK-corpus (eg `gutenberg`):

```
>>> nltk.corpus.gutenberg.items  
( 'austen-emma', 'austen-persuasion', 'austen-sense',  
  'bible-kjv', 'blake-poems', 'blake-songs', 'chesterton-  
  'chesterton-brown', 'chesterton-thursday', 'milton-par  
  'shakespeare-caesar', 'shakespeare-hamlet', 'shakespea  
  'whitman-leaves')
```

What are corpora good for?

In NLP, corpora are used:

- ▶ to create linguistic resources (lexicon, annotated corpora, etc.)
- ▶ to train statistical tools (taggers, parsers, etc.)
- ▶ to evaluate existing tools (taggers, parsers, etc.)

Simple corpus operations

In NLP, a number of corpus operations are recurrently used to create, enrich and exploit corpora such as:

- ▶ *tokenising*: splitting the text into word tokens,
- ▶ *text normalising* e.g., by case,
- ▶ obtaining word *statistics*,
- ▶ *tagging*: associating each word in the corpus with a syntactic category
- ▶ *parsing*: associating each sentence in the corpus with a syntactic tree
- ▶ etc.

Some corpus operations in NLTK

- ▶ Extracting the corpora content
- ▶ Counting words
- ▶ Counting word frequency

Printing the words of a corpus

```
>>> nltk.corpus.brown.words('a')
['The', 'Fulton', 'County', 'Grand', 'Jury', 'said', ...]

#iterate over all word tokens in Macbeth
>>> for w in nltk.corpus.brown.words('a'):
...     print w
...
...

```

Printing the sentences of a corpus

- ▶ The Gutenberg corpus is tokenized as a sequence of words, with no further structure.
- ▶ The Brown corpus has sentences marked, and is stored as a list of sentences, where a sentence is a list of word tokens. We can use the `extract` function to obtain individual sentences

```
>>> nltk.corpus.brown.sents('a')
[['The', 'Fulton', 'County', 'Grand', 'Jury', 'said', 'Friday',
```

Printing the sentences of a parsed corpora

Parsed text from the Penn treebank can also be accessed:

```
>>> for t in nltk.corpus.treebank.parsed_sents():
...     print t
...
```

Printing the words of a tagged corpus

- ▶ The Brown corpus is segmented into sentences and is tagged
- ▶ Part-of-speech tagged text can also be extracted

```
>>> nltk.corpus.brown.tagged_words()
[('The', 'AT'), ('Fulton', 'NP-TL'), ...]
```

Counting the number of words in the Gutenberg corpus

```
>>> for book in nltk.corpus.gutenberg.items:
...     print book + ':', len(nltk.corpus.gutenberg.words(book))
...
austen-emma: 192432
austen-persuasion: 98191
austen-sense: 141586
bible-kjv: 1010735
blake-poems: 8360
blake-songs: 6849
chesterton-ball: 97396
chesterton-brown: 89090
chesterton-thursday: 69443
milton-paradise: 97400
shakespeare-caesar: 26687
shakespeare-hamlet: 38212
shakespeare-macbeth: 23992
whitman-leaves: 154898
```

Count the frequency of each word in a corpora

```
count = {}                                # initialize dictionary

for word in nltk.corpus.gutenberg.words('shakespeare-macbeth'):
    word = word.lower()                   # normalize case
    if word not in count:                 # previously unseen word?
        count[word] = 0                   # if so set count to 0
    count[word] += 1                       # increment word count
```

Count the frequency of each word in a corpora

```
count = {}                                # initialize dictionary

for word in nltk.corpus.gutenberg.words('shakespeare-macbeth'):
    word = word.lower()                   # normalize case
    if word not in count:                 # previously unseen word?
        count[word] = 0                   # if so set count to 0
    count[word] += 1                       # increment word count
```

We can inspect the dictionary:

```
print count['scotland']                   # 12
print count['thane']                       # 25
print count['blood']                       # 24
print count['duncan']                     # 10
```

Sorting by frequency

We would like to sort the dictionary by frequency, but:

- ▶ If we just sort the values (`count.values()`) we lose the link to the keys
- ▶ `count.items()` returns a list of the pairs, but naively sorting that list doesn't do what we want:

```
wordfreq = count.items()
wordfreq.sort()           # WRONG! - sorted by word!
```

We will show three different ways of doing this right in Python...

Sorting by word frequency (1)

One way to do it:

```
wordfreq = count.items()
res = []
for wf in wordfreq:
    res.append((wf[1], wf[0]))

res.sort()
res.reverse()
print res[:10]
```

Sorting by word frequency (2)

slightly less clunky

```
wordfreq = count.items()
res = []

#for wf in wordfreq:
#    res.append((wf[1], wf[0]))

# explicitly assign to elements of a tuple
for (w, f) in wordfreq:
    res.append((f, w))

res.sort()
res.reverse()
```

Sorting by word frequency (3)

Could even use a list comprehension:

```
wordfreq = count.items()

#res = []
#for (w, f) in wordfreq:
#    res.append((f, w))

# use a list comprehension instead
res = [(f, w) for (w, f) in wordfreq]

res.sort()
res.reverse()
```

Sorting by word frequency

In fact, we can write the same code more conveniently using NLTK's frequency distribution class `FreqDist()`.

```
>>> sec_a = nltk.corpus.brown.words('a')
>>> fd = nltk.FreqDist(sec_a)
>>> for token in sorted(fd)[:5]:
...     print fd[token], token
```

Sorting by word frequency

To get the 20 most frequent tokens:

```
>>> from operator import itemgetter
>>> swc = sorted(fd.items(), key=itemgetter(1), reverse=True)
>>> [token for (token, freq) in swc[:20]]
['the', ',', '.', 'of', 'and', 'to', 'a', 'in', 'for',
 'The', 'that', "'", 'is', 'was', '"', 'on', 'at', 'with']
```

Even simpler:

```
>>> print_freq(nltk.corpus.brown.words('a'), 20)
```

- ▶ A text is often encoded into a single string
- ▶ Tokenising consists in splitting this string into words or tokens.
- ▶ This is needed so that the text can undergo further processing, e.g., parsing into grammatical structure.

What is a Word (1)?

Notion of 'word' is not straightforward

Orthographic word: string of characters with 'whitespace' at each end; e.g. *they will want to leave*

Phonological word: 'words' which are pronounced as a phonological unit; e.g. *they'll wanna leave*

Word form: *have, has, had, having* are word forms of the lemma **have**

Lemma/Citation Form: Grammatical form that is chosen to represent a lexeme. In English, usually the **base** form (i.e., with no grammatical marking)

What is a Word (2)?

Multi-part/Discontinuous Words: Sequences which are multiple orthographic words but exhibit the semantic coherence of words; e.g. *Kim rang her up*

Short Forms: abbreviations (*Dept.*), logograms (£), contractions (*we'll*), acronyms (*BBC*)

- ▶ What counts as word token in English often arbitrary:
e.g. *ice cream*, *ice-cream*, *icecream*
- ▶ Tokenization may also depend on requirements of downstream processing; e.g. maybe treat *we've* as two word tokens and try to find *'ve* as a contracted form in lexicon.
- ▶ Tokenization decisions can effect part-of-speech tagging

Some terminology

- ▶ Grammatical markings: used to differentiate different forms of a lexeme; e.g., *bake*, *bakes*, *baker*
 - ▶ *bake* is the **root** or **stem** form
 - ▶ *-s* and *-r* are **morphological affixes** that attach to the root
 - ▶ *bakes* is a (grammatically) **inflected** form (i.e., 3rd person singular present verb)
 - ▶ *baker* is a **derived** form
- ▶ **Stemming** is an operation that strips off grammatical markings to leave the stem; e.g. *foxes* ⇒ *fox*, *flies* ⇒ *fly*
- ▶ **Lemmatization** is an operation that specifies the lemma corresponding to a word form together with corresponding morpho-syntactic and possibly morphoderivational information

- ▶ ... a previously described *FK506-binding protein-associated protein*
- ▶ two possible tokenizations, depending on whether we tokenize the first hyphen in its own right:
 - ▶ FK506 - binding protein-associated protein
 - ▶ FK506-binding protein-associated protein
- ▶ This leads to two different part-of-speech, using an existing tagger:
FK506_SYM -_ : binding_VBG protein-associated_JJ protein_NN
FK506-binding_JJ protein-associated_JJ protein_NN

Why Tokenize?

- ▶ The simplest way to represent a text is with a single string.

```
>>> open("hello.txt").read()
'Hello world!\tThis is a \ntest file.\n'
```
- ▶ We need to identify parts of the string that should undergo further processing, e.g., parsing into grammatical structure.
- ▶ We call the parts **tokens**.
- ▶ In NLTK, it's convenient to work with a **list** of tokens, typically corresponding to orthographic words:

Simple Word Tokenization

The simple 'space' tokenizer in NLTK:

```
>>> s = "This is a string."  
>>> words = s.split()  
>>> print words  
['This', 'is', 'a', 'string.']
```

`split()` just splits the string at spaces. N.B.
`nltk.WhitespaceTokenizer(text)` is another possibility (does the same thing).

NLTK Word Tokenization

The function `nltk.tokenize.regexp_tokenize()` takes a text string and a regular expression, and returns the list of substrings that match the regular expression.

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

Problems: breaks \$22.50 or U.S.A. into several tokens.

NLTK Word Tokenization

Refining the RE:

```
>>> text = 'That poster costs $22.40.'  
>>> pattern = r'''(?x)      # strip out embedded whitespace :  
...   \w+                # sequences of 'word' characters  
...   | \${?}\d+(\.\d+)?  # currency amounts, e.g. $12.50  
...   | ([A-Z]\.)+        # abbreviations, e.g. U.S.A.  
...   | [\^\w\s]+         # sequences of punctuation  
... '''  
>>> nltk.tokenize.regexp_tokenize(text, pattern)  
['That', 'poster', 'costs', '$22.40', '.']
```

NLTK Word Tokenization

The `nltk.tokenize.regexp_tokenize()` function permits an optional boolean parameter `gaps`; when set to `True` the pattern is matched against the gaps.

```
>>> nltk.tokenize.regexp_tokenize(text, pattern=r'\s+',  
gaps=True)  
['That', 'poster', 'costs', '$22.40.']
```

Tokens can be of various different types:

- ▶ words (most usual)
- ▶ lines
- ▶ sentences (also useful, but tricky!)
- ▶ paragraphs

Stemming in NLTK

```
>>> stemmer = nltk.PorterStemmer()
>>> verbs = ['appears', 'appear', 'appeared', 'calling',
'called']
>>> stems = []
>>> for verb in verbs:
...     stemmed_verb = stemmer.stem(verb)
...     stems.append(stemmed_verb)
>>> sorted(set(stems))
['appear', 'call']
```

Lemmatization and stemming are special cases of normalization. They identify a canonical representative for a set of related word forms.

Lemmatisation is a process that map word forms to their lemma (*appeared* → *appear*). It uses rules for the regular word patterns, and table look-up for the irregular patterns.

Stemming is a simpler process that map word forms to a stem. It is deterministic (one solution only) and does not verify that the produced stem is a correct lemma of the language.

Summary

- ▶ Accessing (raw / tagged / parsed) corpora from NLTK
- ▶ Sorting words by corpus frequency
- ▶ Tokenising
 - ▶ Most text processing makes assumptions about linguistic units; good to be aware of the major distinctions in notion of 'word'.
 - ▶ Tokenization into words is important for subsequent processing
 - ▶ Tokenization into sentences also important
 - ▶ But not always easy to tokenize in a consistent and sensible manner, and no Right Answer in general.
- ▶ Stemming and lemmatising for normalisation

Reading: Chapter 3 of the NLTK Book *Words: The Building Blocks of Language*