Exploring imagery in literary corpora with the Natural Language ToolKit

***Claire Brierley*[1, 2] *and Eric Atwell*[2]**
[1] School of Games Computing and Creative Technologies
University of Bolton, UK
[2] School of Computing
University of Leeds, UK
cb5@bolton.ac.uk
eric@comp.leeds.ac.uk

Abstract

This paper presents a middle way for corpus linguists between use of “off-the-shelf” corpus analysis software and building tools from scratch, which presupposes competence in a general-purpose programming language. The Python Natural Language ToolKit (NLTK) offers a range of sophisticated natural language processing tools which we have applied to literary analysis, through case studies in *Macbeth* and *Hamlet*, with code snippets and experiments that can be replicated for research and research-led teaching with other literary texts.

Introduction

Many literary and linguistic researchers rely on “off-the-shelf” text analysis software packages in their empirical research on texts. For example, corpus linguists have used concordance and frequency-analysis tools such as WordSmith (Scott, 2004) and WMatrix (Rayson, 2003) for the exploration and analysis of English literature. Others have advocated that literary and linguistic scholars should become adept in the use of a general-purpose programming language such as Perl or Java (*e.g.* Mason, 2001), as this allows researchers greater freedom to build tools “from scratch” with functionality matching the specific needs of the research question. This paper explores a middle way. Version 0.9.8 of the Python Natural Language Toolkit (Bird *et al*, 2009) includes a range of sophisticated NLP tools for corpus analysis, in sub-modules such as: `collocations` (new for version 0.9.8); `corpus`; `probability`; `metrics`; `text`; and `tokenize`.

In this paper, we have explored the application of NLTK to literary analysis, in particular to explore imagery or ‘imaginative correspondence’ (Wilson-Knight, 2001: 161) in Shakespeare’s *Macbeth* and *Hamlet*; and we anticipate that code snippets and experiments can be adapted for research and research-led teaching with other literary texts. The initial discussion refers to built-in functionality for NLTK’s `text` module (§1) and provides a rationale and user access code for preserving *form* during verse tokenization, while detaching punctuation tokens as normal; we also look at compounds in *Macbeth* (§2). The same text is used to illustrate and experiment with NLTK’s built-in support for frequency distributions and collocation discovery (§3). Finally, we explore synsets of polysemous words (e.g. *death*) in *Hamlet* and measure their semantic similarity as a correlate of path length between their senses (§4).

1 Analysing web-based versions of literary texts as NLTK Text objects

NLTK’s `text` module supports a variety of methods for text exploration. It is possible to access a web-based version of a literary text from Project Gutenberg[1], for example, transform it into an NLTK Text object, and perform analyses in a few lines of code. Listing 1.1 shows how this is done by (i) reading in present-day English eTexts of *Macbeth* and *Hamlet* as raw strings; (ii) transforming them into a list of word tokens so that they can be instantiated as Text objects; and (iii) calling example methods for obtaining concordance lines and distributionally similar items for given words: *children* in *Macbeth* and *mother* in *Hamlet*. Nine instances of the word *children* are returned, and the five most similar items to the seed word *mother* are: *father*; *lord*; *son*; *heart*; *nothing*.

```
# text_objects.py
# NLTK version 0.9.8
# Copyright: Claire Brierley
# 20.04.2009

import nltk, re, pprint
from urllib import urlopen
from nltk.tokenize import *

# READ IN TEXTS OF MACBETH AND HAMLET AS RAW STRINGS FROM PROJECT GUTENBERG
url1 = urlopen("http://www.gutenberg.org/dirs/etext98/2ws3410.txt").read() # length: 120027
url2 = urlopen("http://www.gutenberg.org/dirs/etext98/2ws2610.txt").read() # length: 193082

# TRANSFORM STRING OBJECTS TO LISTS OF WORD TOKENS VIA NLTK BUILT-IN TOKENIZE FUNCTION
url1 = nltk.wordpunct_tokenize(url1) # transforms raw text into list of tokens, length: 26931
url2 = nltk.wordpunct_tokenize(url2) # transforms raw text into list of tokens, length: 44590

# SEARCH FOR THE BEGINNING OF EACH PLAY
for index, value in enumerate(url1):
       if value == 'Thunder': print index, value, # early stage direction in Macbeth

print
for index, value in enumerate(url2):
       if value == 'Elsinore': print index, value, # early stage direction in Hamlet

print

# INDICES RETURNED INDICATE LIKELY POSITION FOR START OF PLAY
#2628 Thunder 3467 Thunder 16295 Thunder 17219 Thunder 17912 Thunder 18023 Thunder 18143
Thunder
#2558 Elsinore 2566 Elsinore 4434 Elsinore 6269 Elsinore 16062 Elsinore 17121 Elsinore 18967
Elsinore 31875 Elsinore

# CREATE NLTK TEXT OBJECTS FOR EACH PLAY
macbeth = url1[2618:26911] # len = 24293
macbeth2 = nltk.Text(macbeth) # the Text object
hamlet = url2[2556:44569] # len = 42013
hamlet2 = nltk.Text(hamlet) # the Text object
```

```
# EXAMPLE METHOD CALLS
>>> macbeth2.concordance('children')
Building index...
Displaying 9 of 9 matches:
e reason prisoner ? MACBETH . Your children shall be kings . BANQUO . You shal
our pains .-- Do you not hope your children shall be kings , When those that g
ies Are to your throne and state , children and servants ; Which do but what t
uell ? MACBETH . Bring forth men - children only ; For thy undaunted mettle sh
 Why , well . MACDUFF . And all my children ? ROSS . Well too . MACDUFF . The
, and bids it break . MACDUFF . My children too ? ROSS . Wife , children , ser
 . My children too ? ROSS . Wife , children , servants , all That could be fou
deadly grief . MACDUFF . He has no children .-- All my pretty ones ? Did you s
th no stroke of mine , My wife and children ' s ghosts will haunt me still . I

>>> hamlet2.similar('mother', num=5)
Building word-context index...
father lord son heart nothing
```
Listing 1.1: Example method calls on an NLTK Text object created from web-based literary texts

The same syntax can be used to call further methods on the Text object, largely self-evident: `count()`; `dispersion_plot()`; `generate()` which returns random text based on a trigram language model; and `collocations()` which will be discussed in more detail (§3.1).

2 Customized tokenization for Shakespearian verse

The `wordpunct_tokenize()` built-in function in Listing 1.1 returns text as a list of word tokens but this disrupts the structure of Shakespearian (and other) verse and is not suitable for certain kinds of linguistic analysis – the analysis of prosody, for example (Brierley and Atwell, 2009). Listing 1.2 resumes the previous listing and utilises yet another Text object method to locate an approximate index position for Macbeth's soliloquy in Act 1, Scene 7; a slice of this speech is then returned to demonstrate how line onset and sentence initial tokens are not differentiated and how compositionality has been lost from compounds, a characteristic feature of Macbeth's speaking style (§2.2):

```
>>> macbeth2.index('surcease')
4406
>>> macbeth2[4372:4428]
['MACBETH', '.', 'If', 'it', 'were', 'done', 'when', "'", 'tis', 'done', ',',
'then', "'", 'twere', 'well', 'It', 'were', 'done', 'quickly', '.', 'If', 'the',
'assassination', 'Could', 'trammel', 'up', 'the', 'consequence', ',', 'and',
'catch', ',', 'With', 'his', 'surcease', ',', 'success', ';', 'that', 'but',
'this', 'blow', 'Might', 'be', 'the', 'be', '-', 'all', 'and', 'the', 'end', '-',
'all', '--', 'here', ',']
```

Listing 1.2: Output from NLTK's `wordpunct_tokenize()` method

Listing 2.1 provides a solution for preserving verse *form* during tokenization and for rhythmic fidelity in the treatment of enclitics and speaker fidelity in the treatment of hyphenated words and the prosodic-syntactic phrasing inherent in punctuation. An extract from Hamlet's first soliloquy (*Hamlet* Act 1, Scene 2), accessed via NLTK's Shakespeare corpus, is used in this case and tokenization is a two-step process: (i) an instance of the `LineTokenizer()` class returns lines as substrings; and (ii) patterns (i.e. tokens) are retrieved from each raw substring via the regular expression `<<\w+(?:[-']\w+)*|[-.]+|\S\w*>>` as argument to the `re.findall()` method (§2.1).

```
# tokenizing_verse.py
# NLTK version 0.9.8
# Copyright: Claire Brierley
# 20.04.2009

import nltk, re, pprint
from nltk.corpus import shakespeare
from nltk.tokenize import *

# INSTANTIATE A TOKENIZER TO CAPTURE LINE TOKENS AS RAW STRINGS
tokenizer1 = LineTokenizer(blanklines='keep')

# REMOVE XML TAGS, CAPTURE LINE TOKENS & LOCATE DESIRED SLICE OF TEXT
text = nltk.corpus.shakespeare.raw('hamlet.xml')
text = tokenizer1.tokenize(nltk.clean_html(text))
soliloquy1 = text[721:752]

# INSTANTIATE A CONTAINER
sample = []

# APPEND NESTED LISTS OF TOKENS MATCHING A REGULAR EXPRESSION ADAPTED FROM NLTK 3.7
for line in soliloquy1: sample.append(re.findall(r"\w+(?:[-']\w+)*|[-.]+|\S\w*", line))
for line in sample[17:21]: print line
```

Listing 2.1: Customizing tokenization for Shakespearian and other verse

2.1 A regular expression for separating word-internal versus normal punctuation

NLTK's `WordPunctTokenizer()` from Listing 1.1 uses the regular expression `<<\w+|[^\w\s]+>>` to divide text into sequences of alphanumeric and non-alphanumeric characters, resulting in the following tokenization for this line from our sample:

```
['Let', 'me', 'not', 'think', 'on', "'", 't', '--', 'Frailty', ',', 'thy',
'name', 'is', 'woman', '!--']
```

Example 1: Output from NLTK's WordPunctTokenizer() class

The regular expression adapted from NLTK's online book (Bird *et al.*, 2009: 3.7) and used to tokenize lines in Listing 2.1 is more sophisticated and is explained in Table 1.

`<<\w+(?:[-']\w+)*|[-.]+|\S\w*>>`	
`\w+`	Retrieves all alphanumeric characters.
`(?:[-']\w+)*`	Modifies the first part of this regular expression so that hyphens and apostrophes preceding alphanumerics are also captured *e.g.* {'gainst} {-slaughter} {'t} {-Frailty}.
`[-.]+`	Retrieves double hyphens *i.e.* house style for signifying true pauses or hesitations.
`\S\w*`	Retrieves any non-whitespace character which may then be followed by zero or more alphanumeric characters.

Table 1: Decompositon of the regular expression used in Listing 2.1

Thus the customised tokenization process in Listing 2.1, which incorporates this regular expression pattern, preserves verse form and compositionality in enclitics and compounds, while capturing word and punctuation tokens. Example 2 gives output from both soliloquies sampled in this section and suggests how different kinds of tokens could be used for stylometric analysis of character speech – for example, compounding in *Macbeth*; and the exclamations and hesitations habitual to the protagonist in *Hamlet*.

```
['If', 'it', 'were', 'done', 'when', "'tis", 'done', ',', 'then', "'twere",
'well']
['Might', 'be', 'the', 'be-all', 'and', 'the', 'end-all', 'here', ',']
['Let', 'me', 'not', 'think', "on't", '--', 'Frailty', ',', 'thy', 'name',
'is', 'woman', '!', '--']
```

Example 2: Output from the customized tokenizer in Listing 2.1

2.2 Compounding in *Macbeth*

Wilson-Knight asserts that the fast-paced, creative economy of *Macbeth* represents "...one swift act of the poet's mind..." (Wilson-Knight, 2001:161). Such an effect is partly achieved through the prevalence of compounds in the play, especially as used by the protagonist. Nevalainen (2001:240) considers clauses compacted and packaged as compound adjectives based on past participles as an essential element of Shakespeare's style; a typical example from *Macbeth* would be the "...air-drawn dagger..." which *marshals* the way to Duncan's sleeping quarters and his demise.

Another stylistic device characteristic of Shakespeare is conversion (*ibid*: 242); parts of speech are recycled as in Lady Macbeth's:

"...Come, you spirits
That tend on mortal thoughts! *unsex* me here..,"

Macbeth: Act 1, Scene 5

Here affixation simultaneously generates an antonym for the noun *sex* and also sanctions verb-like behaviour as in {do; undo}. Listing 2.2 uses code similar to Listing 2.1 to customise tokenization for *Macbeth* and then to capture, sort and print all compound expressions in the play so that their patterns, including conversions in component parts of speech, can be inspected.

```
# tokenizing_verse.py
# NLTK version 0.9.8
# Copyright: Claire Brierley
# 20.04.2009

import nltk, re, pprint
from nltk.corpus import shakespeare
from nltk.tokenize import *

# INSTANTIATE A TOKENIZER TO CAPTURE LINE TOKENS AS RAW STRINGS
tokenizer1 = LineTokenizer(blanklines='keep')
```

```
# REMOVE XML TAGS & CAPTURE LINE TOKENS
text = nltk.corpus.shakespeare.raw('macbeth.xml')
text = tokenizer1.tokenize(nltk.clean_html(text))

# INSTANTIATE A CONTAINER
macbeth = []

# APPEND NESTED LISTS OF TOKENS MATCHING A REGULAR EXPRESSION ADAPTED FROM NLTK 3.7
for line in text: macbeth.append(re.findall(r"\w+(?:[-']\w+)*|[-.]+|\S\w*", line))

# CAPTURE ALL COMPOUNDS (i.e. HYPHENATED FORMS)
hyphens = []
for line in macbeth:
    for token in line:
        if '-' in token:
            if re.search('-$', token): pass # ignore tokens ending in a hyphen
            elif re.search('^-', token): pass # ignore tokens beginning with a hyphen
            else: hyphens.append(token)

print sorted(set(hyphens))
```
Listing 2.2: A program for capturing compounds or hyphenated forms in *Macbeth*

Perhaps the most interesting outputs here are new nouns which exhibit changes in grammatical function for one or both base words: {*the be-all*; *the end-all*; *his taking-off*; *the all-hail*; *my more-having*; *thy here-approach*; *my here-remain*} and yet are immediately accepted and understood: “...one intriguing aspect of Shakespeare’s new words is that there is something familiar about most of them...” (*ibid*: 237).

3 NLTK’s built-in support for frequency distributions

The experiments using simple statistics in this section are performed on *Macbeth* the play, plus Macbeth’s and Banquo’s speeches as subsets of the same. Tokenization is relatively unsophisticated, as in Listing 1.1, and function words and punctuation are filtered out unless stated otherwise.

The `FreqDist()` class in NLTK’s probability module encodes frequency distributions which map each sample (each word token in this case) to the number of times it occurs as an outcome of an experiment. The commented code in Listing 3.1 prepares the texts as described by discarding XML tags, punctuation and stopwords before passing these stripped-down versions to a function which initialises a frequency distribution object and returns counts and tokens for the twenty most frequent words in a given text in descending order of frequency.

```
# find_frequencies.py
# NLTK version 0.9.8
# Copyright: Claire Brierley
# 11.05.2009

import nltk, re, pprint
from nltk.corpus import shakespeare
from nltk.tokenize import *
from nltk.util import bigrams, trigrams # import statement used in Listing 6
from nltk.collocations import * # new functionality for NLTK 0.9.8 & used in Listing 7

# NOTE: CODE REPRESENTS FILEPATH AS: 'C:\\...\\filename.txt'
mac = nltk.wordpunct_tokenize(nltk.clean_html(open('C:\\...\\just_macbeth.txt', 'rU').read()))
ban = nltk.wordpunct_tokenize(nltk.clean_html(open('C:\\...\\just_banquo.txt', 'rU').read()))
play = nltk.wordpunct_tokenize(nltk.clean_html(nltk.corpus.shakespeare.raw('macbeth.xml')))

# ALL WORD TOKENS IN SAME CASE OTHERWISE WORD COUNTS WOULD BE AFFECTED
mac = [word.lower() for word in mac]
ban = [word.lower() for word in ban]
play = [word.lower() for word in play]

# APPLY FILTERS FOR PUNCTUATION & STOPWORDS
extras = ['.--', ',--', '?--', '!--', ':--', ';--'] # house style punctuation
functionWords = nltk.corpus.stopwords.words('english')

bin1  = []
```

```
bin2  = []
bin3  = []
mac2  = [] # container for filtered version of text
ban2  = []
play2 = []

for word in mac:
    if len(word) <= 2 or word in extras:
        bin1.append(word)

for token in mac:
    if token not in functionWords and token not in bin1:
        mac2.append(token) # filtered version of Macbeth's speeches

for word in ban:
    if len(word) <= 2 or word in extras:
        bin2.append(word)

for token in ban:
    if token not in functionWords and token not in bin2:
        ban2.append(token)

for word in play:
    if len(word) <= 2 or word in extras:
        bin3.append(word)

for token in play:
    if token not in functionWords and token not in bin3:
        play2.append(token)

# DEFINE FUNCTION FOR RETRIEVING FREQUENCY DISTRIBUTION FOR GIVEN TEXT
def find_frequencies(words, num=20): # words is assumed to be a list of tokens
       fdist = nltk.FreqDist(words) # initialise frequency distribution
       for sample in fdist.keys()[:num]: # sorts samples in decreasing order of frequency
              print fdist[sample], sample # prints count & word token
```

Listing 3.1: A program for pre-processing text prior to capturing its top 20 most frequent content words

Calling this user-defined `find_frequencies()` function on the filtered version of Macbeth's speeches (`mac2`) returns the following top twenty items and their counts: {41 *thou*; 24 *thy*; 21 *thee*; 18 *fear*; 18 *time*; 16 *hath*; 15 *blood*; 14 s*leep*; 14 *till*; 14 *banquo*; 13 *man*; 13 *good*; 13 *mine*; 13 *night*; 12 *life*; 12 *make*; 11 *heart*; 11 *hear*; 10 *love*; 10 *nature*}.

Whilst highlighting words and preoccupations made resonant partly through the immersive effect of repetition {*fear*; *time*; *blood*; *sleep*}, it appears we have allowed some archaic pronouns to slip through the net, simply because they are not in the stoplist. Furthermore, the unusually high frequency count for *thou* in comparison to the other items merits further investigation. If at this point we transform the *un*filtered and tokenized version of Macbeth's speeches into an NLTK Text object (§1), we can examine patterns of use for *thou* from concordance lines. Since there are 41 occurrences of *thou*, we will need to change the number of lines returned by the `concordance()` method which defaults to 25.

```
mac3 = nltk.Text(mac)
mac3.concordance('thou', lines=45)
```

```
not , and yet i see thee still . art thou not , fatal vision , sensible to fee
ible to feeling as to sight ? or art thou but a dagger of the mind , a false c
 palpable as this which now i draw . thou marshall ' st me the way that i was
here ? which of you have done this ? thou canst not say i did it : never shake
how say you ? why , what care i ? if thou canst nod , speak too . if charnel -
are marrowless , thy blood is cold ; thou hast no speculation in those eyes wh
t no speculation in those eyes which thou dost glare with . what man dare , i
 . what man dare , i dare : approach thou like the rugged russian bear , the a
l ' em : let me see ' em . tell me , thou unknown power ,— whate ' er thou art
 cauldron ? and what noise is this ? thou art too like the spirit of banquo ;
 sear mine eyeballs : and thy hair , thou other gold - bound brow , is like th
t came by ? fled to england ! time , thou anticipat ' st my dread exploits ; t
 s macbeth . no , nor more fearful . thou wast born of woman : but swords i sm
rg ' d with blood of thine already . thou losest labour : as easy mayst thou t
 . thou losest labour : as easy mayst thou the intrenchant air with thy keen sw
```

```
rnam wood be come to dunsinane , and thou oppos ' d , being of no woman born ,
```
Listing 3.2: Sample concordances for *thou* in *Macbeth* returned by an NLTK built-in method for Text objects

Sixteen lines have been selected for comment. These exhibit an association of *thou* with heightened psychological states, with hallucinations and fatal apparitions: air-drawn daggers; blood-bolter'd murderees; armed heads; bloody children crowned; and moving groves. In the semantics of power (Brown and Gilman, 2003: 158), *thou* as a familiar form of address erects an asymmetric relation between speaker and addressee. Macbeth's use of *thou* in these contexts evokes both the intimacy of a private state and a desperate self-assertiveness in the battle for control of events and his sanity. The final concordance line in Listing 3.2 has a different timbre, however:

> **"...Though Birnam wood be come to Dunsinane,**
> **And *thou* oppos'd, being of no woman born,**
> **Yet I will try the last..."**
>
> *Macbeth*: Act 5, Scene 7

In the semantics of solidarity (ibid: 160), *thou* sets up a symmetrical relation between equals ("...address between twin brothers or ... a man's soliloquizing address to himself..."); the undeceived, self-knowing and ultimately sane Macbeth acknowledges and faces his nemesis.

3.1 Collocation discovery

NLTK's `util` module has a built-in function `bigrams()` which operates on a sequence of items to produce a new list of pairs. Thus, resuming code Listing 3.1, we can obtain the top twenty most frequent bigrams for the filtered version of Macbeth's speeches, based simply on a raw count.

```
bigramsMac = bigrams(mac2)
fdMac2 = nltk.FreqDist(bigramsMac)
for sample in fdMac2.keys()[:20]: print fdMac2[sample], sample
```
```
6 ('thane', 'cawdor')
4 ('born', 'woman')
4 ('thou', 'art')
3 ('thy', 'face')
3 ('woman', 'born')
3 ('mine', 'eyes')
3 ('thee', 'thy')
3 ('thee', 'thee')
3 ('birnam', 'wood')
3 ('wood', 'dunsinane')
3 ('till', 'birnam')
2 ('thou', 'dost') # again, archaic forms have evaded the filter
2 ('morrow', 'morrow')
2 ('didst', 'thou')
2 ('thy', 'blade')
2 ('till', 'famine')
2 ('murder', 'sleep')
2 ('weird', 'sisters')
2 ('fears', 'banquo')
2 ('mine', 'armour')
```
Listing 3.3: Obtaining the top 20 most frequent bigrams in *Macbeth*

If, however, we are interested in bigrams or collocations that occur more often than expected given the frequency of each of their component words, then we can use one of the scoring functions for various association measures available in NLTK's `association` module. This is in fact what happens with the `collocations()` method for Text objects (§1), where source code implements: (i) filters similar to those in Listing 3.1; and (ii) the likelihood ratio test for independence recommended for sparse data (*cf.* Manning and Schutze, 1999: 172). By way of illustration, the following code adapted from NLTK's Collocations HOWTO[2] returns the top ten bigram collocations specific to our filtered version of *Macbeth* after applying the likelihood statistic for candidates which occur at least twice.

```
find_collocations = BigramCollocationFinder.from_words(play2)
find_collocations.apply_freq_filter(2)
find_collocations.nbest(bigram_measures.likelihood_ratio, 10)
```

```
[('lady', 'macbeth'), ('exeunt', 'scene'), ('thane', 'cawdor'), ('lady', 'macduff'), ('malcolm', 'donalbain'), ('young', 'siward'), ('weird', 'sisters'), ('knock', 'knock'), ('drum', 'colours'), ('birnam', 'wood')]
```

Listing 3.4: Obtaining the top 10 collocations in *Macbeth* as scored via likelihood ratios

Finally, if we compare the twenty most frequent bigrams from Listing 3.3 to the output from the `collocations()` method when called on Macbeth's speeches, we find there is roughly 50% correspondence. Common bigrams are: {`('thou', 'art')`; `('thy', 'face')`; `('woman', 'born')`; `('mine', 'eyes')`; `('birnam', 'wood')`; `('till', 'birnam')`; `('thou', 'dost')`; `('mine', 'armour')`; `('till', 'famine')`; `('thy', 'blade')`}.

3.2 Conditional frequency distributions

Suppose we wished to quickly compare different characters' usage of particular words: the number of times Macbeth and Banquo express fear, for example? We could set up a conditional frequency distribution with two conditions, where each event – the appearance of target words – was mapped to (*i.e.* conditioned by) the name of a character. NLTK's online book (Bird *et al.*, 2009: 2.2) suggests creating a `ConditionalFreqDist` data type, with built-in methods for tabulation and plotting, from a list of pairs; in our case, pairs at the beginning of the list have the form `('macbeth', event)` and those at the end `('banquo', event)`. The whole exercise could be scaled up to compare word usage over several plays or plays by different dramatists, for example; number of dramas/authors would then dictate the number of conditions. Listing 3.5 presupposes Listing 3.1 and represents a small scale experiment to compare word usage from a mini lexicon for two important concepts – *doing* and *seeing* – in *un*filtered versions of Macbeth's and Banquo's speeches.

```
# INSTANTIATE MINI LEXICONS FOR DOING & SEEING
doing = ['do', 'does', 'dost', 'did', 'doing', 'done', 'undone', 'deed', 'undo', 'dost']
seeing = ['seem', 'seems', 'see', 'sees', 'saw', 'seen', "look'd", 'look', 'looks', "seem'd"]

# CREATE LIST OF PAIRS & CONDITIONAL FREQUENCY DISTRIBUTION OBJECT
macPairs = [('Macbeth', index) for index in mac]
banPairs = [('Banquo', index) for index in ban]
pairs = list((macPairs + banPairs))
cfdPairs = nltk.ConditionalFreqDist(pairs)

# TABULATE COMPARATIVE OUTPUT FOR MACBETH & BANQUO
cfdPairs.tabulate(conditions=['Macbeth', 'Banquo'], samples=doing)
print
cfdPairs.tabulate(conditions=['Macbeth', 'Banquo'], samples=seeing)
print
```

```
          do does dost  did doing done undone deed undo dost
Macbeth   21    6    2   11     2   13      1    9    0    2
 Banquo    3    2    0    0     0    0      0    0    0    0

        seem seems  see sees  saw seen look'd look looks seem'd
Macbeth    1     1   13    0    2    2      0    6     0      0
 Banquo    2     1    0    0    0    0      0    4     0      0
```

Listing 3.5: Using a conditional frequency distribution to compare word usage in 2 datasets

Even simple statistics such as these focus manual interpretation of results via concordance lines – an essential part of the process. Displaying concordances for thirteen instances of *done* using our NLTK Text object (§3; §1), begins to uncover an association between doing and seeing ("*...the eye fears, when it is done, to see...*") which can be further investigated; self-realisation for Macbeth works through dreams and hallucinations, a kind of seeing which cannot be repressed:

> **“...Strange things I have in head that will to hand,**
> **Which must be *acted* ere they may be *scann'd*...”**
>
> *Macbeth*: Act 3, Scene 4

```
>>> mac3.concordance('done')
Displaying 13 of 13 matches:
 be which the eye fears , when it is done , to see . my dearest love , duncan
. we will speak further . if it were done when ' tis done , then ' twere well
further . if it were done when ' tis done , then ' twere well it were done qui
tis done , then ' twere well it were done quickly ; if the assassination could
 their very daggers , that they have done ' t ? i am settled , and bend up eac
cold breath gives . i go , and it is done ; the bell invites me . hear it not
. who ' s there ? what , ho ! i have done the deed . didst thou not hear a noi
e : i am afraid to think what i have done ; look on ' t again i dare not . whe
 the moment on ' t ; for ' t must be done to - night , and something from the
l fever he sleeps well ; treason has done his worst : nor steel , nor poison ,
ht ' s yawning peal , there shall be done a deed of dreadful note . be innocen
' s full . where ? which of you have done this ? thou canst not say i did it :
oughts with acts , be it thought and done : the castle of macduff i will surpr
```
Listing 3.6: Sample concordances for *done* in *Macbeth* returned by an NLTK built-in method for Text objects

4 Exploring imagery in *Hamlet* via NLTK’s WordNet interface

NLTK includes a version of the WordNet lexical database for English (Fellbaum, 1998) plus a `wordnet` module which implements methods for displaying chains of relations between lemma and synset objects (*e.g.* the following relations for nouns: hypernym/hyponym; synonym/antonym; meronym/holonym) and functions for quantifying lexical semantic relatedness via the different metrics (*e.g.* edge-counting to determine conceptual distance between terms) evaluated in Budanitsky and Hirst (2006).

A synset is a lexicalised concept represented and expressed by its list of synonymous words or lemmas, where the latter constitute pairings of synset and word and thus a specific sense of a given word form. So for example, the synset `rest.n.05`, described via its gloss as *euphemisms for death*, contains five lemmas accessed in NLTK as follows:

```
>>> import nltk, re, pprint
>>> import nltk.corpus
>>> from nltk.corpus import wordnet as wn

>>> for lemma in wn.synset('rest.n.05').lemmas: print lemma

Lemma('rest.n.05.rest')
Lemma('rest.n.05.eternal_rest')
Lemma('rest.n.05.sleep')
Lemma('rest.n.05.eternal_sleep')
Lemma('rest.n.05.quietus')
```
Listing 4.1: Accessing synonyms for *rest* as a euphemism for *death* via NLTK’s WordNet interface

One of the lemmas returned is `rest.n.05.quietus`; another is `rest.n.05.sleep`. Navigating WordNet’s concept hierarchy via an additional line of code in Listing 4.2, we find that the immediate hypernym or superordinate for `rest.n.05` is the synset `death.n.03`.

```
>>> for synset in wn.synset('rest.n.05').hypernyms(): print synset, synset.definition

Synset('death.n.03') the absence of life or state of being dead
```
Listing 4.2: Obtaining the immediate hypernym and its gloss for a given concept in NLTK’s WordNet interface

Hence we begin to uncover a semantic association between *quietus* – a misleading word, as it seems also to be associated with violence in Roget’s Thesaurus (Lloyd, 1982: 208) and *quittance for a debt* in The Arden Shakespeare (Jenkins, 2003: 279) – and *death-as-eternal-sleep* directly reminiscent of Hamlet’s famous soliloquy:

“...When he himself might his *quietus* make
With a bare bodkin?..”

Hamlet: Act 3, Scene 1

4.1 Resolving jarring opposites in *Hamlet*: sleep⇔death⇔flux⇔action

All the experiments in this section have been inspired by the essay *Hamlet Reconsidered* in Wilson-Knight’s recently re-published classic of literary criticism: *The Wheel of Fire* (Wilson-Knight, 2001: 338-366). Here, Hamlet’s central speech is described as: “...a sequence of abnormal thinking holding in solution, as it were, the jarring opposites of [the] play...” (ibid: 347). His choice of the word *solution* is interesting; it suggests several meanings: *fluid*; *mixture*; *answer-to-a-problem*. And it reflects the language of the play, as we will shortly demonstrate.

We have touched on the *sleep-death* relation in *Hamlet* and will revisit this theme, together with the seemingly irreconcilable concept (or “jarring opposite”) of *death-as-purposeful-action*, by exploring the hypernym-hyponym relation between certain sysnets in NLTK’s WordNet interface. However, we begin with a discussion of the word *resolve* as used by Hamlet early in the play, when he first contemplates suicide as a way out:

“...O that this too too solid flesh would melt
Thaw and *resolve* itself into a dew,
Or that the Everlasting had not fix’d
His canon ’gainst self-slaughter...”

Hamlet: Act 1, Scene 2

4.1.1 Resolve, disintegration and flux

Physically wasting away, dying and decaying are virtually indistinguishable processes in Hamlet’s expression of the death-wish; and images of dirty (sullied) solids melting and thawing and liquefying into a dew-like substance primarily equate *resolve* with *dissolve* in this context. Constraining part-of-speech (POS) in NLTK’s `wn.synsets()` method returns `dissolve.v.02` as one of seven distinct meanings, given in the glosses or definitions, for the verb *resolve* in Listing 4.3:

```
>>> for synset in wn.synsets('resolve', pos=wn.VERB): print synset, synset.definition

Synset('decide.v.02') bring to an end; settle conclusively
Synset('conclude.v.03') reach a conclusion after a discussion or deliberation
Synset('purpose.v.02') reach a decision
Synset('answer.v.04') understand the meaning of
Synset('resolve.v.05') make clearly visible
Synset('resolve.v.06') find the solution
Synset('dissolve.v.02') cause to go into a solution
```

Listing 4.3: Accessing hyponyms and their glosses for a given concept in NLTK’s WordNet interface

However, the glosses also invoke parallel meanings for *resolve* to do with reaching a decision and seeing clearly and these cannot be dismissed, particularly because words used poetically hold different meanings *in solution* (to borrow Wilson-Knight’s phrase) and because modern pronunciation does not differentiate between *resolve-dissolve* and *resolve-decide*: we never say `/’ri’s0lv/`.

If we now inspect synsets for the word *dissolve*, we unearth new glosses and new meanings which may also be admissible in this context, for example: {*to stop functioning or cohering as a unit*; *to cause to fade away*; *to lose control emotionally*} – maybe the *dew* also signifies *tears*? We also find *melt* and *thaw* as synonyms for *dissolve*; and we find evidence of *resolve* as an obsolete form of *dissolve* – *cf*. Collins English Dictionary (Sinclair, 1994) – in the lemma `dissolve.v.02.resolve`.

4.1.2 Death as state and event

Words like *sleep* and *death* are polysemous and therefore deemed to be ambiguous: the noun *sleep* has six synsets or distinct senses in WordNet and *death* has eight. Adapting code from Listing 4.3, we find that concepts pertinent to *Hamlet*, namely *death-as-event* (*the event of dying or departure from life*) and *death-as-state* (*the absence of life or state of being dead*), are given by the synsets `death.n.01` and `death.n.03` respectively. By looking up the depth of these concepts and by navigating superordinate concepts up to root synsets or unique beginners in Listing 4.4, we also find that *death-as-state* is higher in the taxonomy – and is thus more general than, and subsumes *death-as-event*. The method `wn.synset(synset).hypernym_paths()` returns a nested list object where each inner list constitutes one path along the hypernym-hyponym spine; here there is only one path: hence `index [0]`. The method is called within a list comprehension so that outcomes are returned as a list.

```
>>> wn.synset('death.n.03').min_depth() # death-as-state
4
>>> [synset.name for synset in wn.synset('death.n.03').hypernym_paths()[0]]

['entity.n.01', 'abstraction.n.06', 'attribute.n.02', 'state.n.02', 'death.n.03']
>>> wn.synset('death.n.01').min_depth() # death-as-event
6
>>> [synset.name for synset in wn.synset('death.n.01').hypernym_paths()[0]]

['entity.n.01',  'abstraction.n.06',  'psychological_feature.n.01',  'event.n.01',
'happening.n.01', 'change.n.01', 'death.n.01']
```

Listing 4.4: Situating a given concept in the WordNet taxonomy via the `hypernym_paths()`method

4.1.3 Imaginative correspondences: death, flux, and action

There are seventy coordinate concepts for *death-as-state* (`death.n.03`) given by the hyponyms of `state.n.02` (*cf.* Listing 4.5).

```
>>> coordinates = wn.synset('state.n.02').hyponyms()
>>> len(coordinates)
71
>>> coordinates = [lemma.name for synset in coordinates for lemma in synset.lemmas]
```

Listing 4.5: Reassigning the variable `coordinates` to the compiled list of lemma names (i.e. words) for lemmas which appear in the list of hyponyms for `state.n.02`

On inspection, these coordinates contain some Hamlet-like incompatibles: {action-inaction; being-nonbeing; flux-stillness} and triggers: {death; end; eternal damnation; existence; integrity} – and also *readiness*, as in '*...the readiness is all...*' The final experiment in this series looks at the degree of semantic relatedness between three of these concepts {death; flux; action} which are said to be *held in solution* in the '*To be or not to be...*' speech. For example, action opposes flux in *taking arms against a sea of troubles*, while *shuffling off this mortal coil* is at once action and death; finally, the *currents* of *great enterprises turn*[*ing*] *away* is action stymied by flux. Wilson-Knight (2001: 348) writes eloquently of this relatedness: '...Hamlet's mind...in his reverie...through enigmatic phrases and suicide thoughts, [is] half-creating the synthesis of his agonising incompatibles. For once these extremes intershade, they are fluid and run into each other, like dreams. This is...a creative state, like poetry...'

Listing 4.6 instantiates two list objects `state` and `event` each holding values denoting hyponym concepts for {death; flux; action} for the synsets `state.n.02` and `event.n.01` respectively. The function `concept_hierarchy()`then returns (1) taxonomic depth and gloss for each aforesaid value when called on `state` and `event`; and (2) the taxonomic path from each value to a common root synset: `entity.n.01` (*cf.* Fig. 1).

```
# using_wordnet.py
# NLTK version 0.9.8
# Copyright: Claire Brierley
# 09.03.2009

import nltk, re, pprint
import nltk.corpus
from nltk.corpus import wordnet as wn

# DEATH, FLUX AND ACTION AS HYPONYMS OF STATE
d3 = wn.synset('death.n.03')
f5 = wn.synset('flux.n.05')
a2 = wn.synset('action.n.02')

# DEATH, FLUX AND ACTION AS HYPONYMS OF EVENT
d1 = wn.synset('death.n.01')
f8 = wn.synset('flux.n.08')
a1 = wn.synset('action.n.01')

state = [d3, f5, a2]
event = [d1, f8, a1]

# USER-DEFINED FUNCTION
def concept_hierarchy(synset):
    for synset in synset:
        print synset.name, ':', 'depth =', synset.min_depth(), ':', synset.definition
        print 'taxonomic chain:', synset.hypernym_paths()
        print

concept_hierarchy(state)
print
concept_hierarchy(event)
```
Listing 4.6: A program for measuring lexical semantic relatedness between given concepts in terms of taxonomic depth

Table 2 displays results from step (1). From the glosses for hyponyms of the list object state, we find which senses of {death; flux; action} are coordinate, that is at the same depth in the taxonomy. Perhaps the most interesting concept retrieved here in respect of the play is given by the gloss for `flux.n.05` which seems to describe the protagonist's deliberations for much of the play, and suggests that this state is neither mad nor cowardly but *natural* "following some important event".

HYPERNYM	**CONCEPT**	**DEPTH**	**GLOSS**
state	death.n.03	4	the absence of life or state of being dead
state	flux.n.05	4	**a state of uncertainty about what should be done (usually following some important event) preceding the establishment of a new direction of action**
state	action.n.02	4	the state of being active
event	death.n.01	6	the event of dying or departure from life
event	flux.n.08	7	in constant change
event	action.n.01	5	something done (usually as opposed to something said)

Table 2: Tabulated output from Listing 4.6

Figure 1 is a diagrammatic representation of lexical relations from step (2). Hamlet wishes for *death-as-state*, a subsumer for senses of sleep, quietus and eternal rest, and "...*a consummation devoutly to be wish'd*..." Yet it is the concept of *death-as-event* that informs his thinking: "...[the] central thought is suicide. Suicide is the one obvious fusion – the best Hamlet can reach at this stage – of the opposing principles of fine action and death-shadowed passivity..." (*ibid.* 2001: 347).

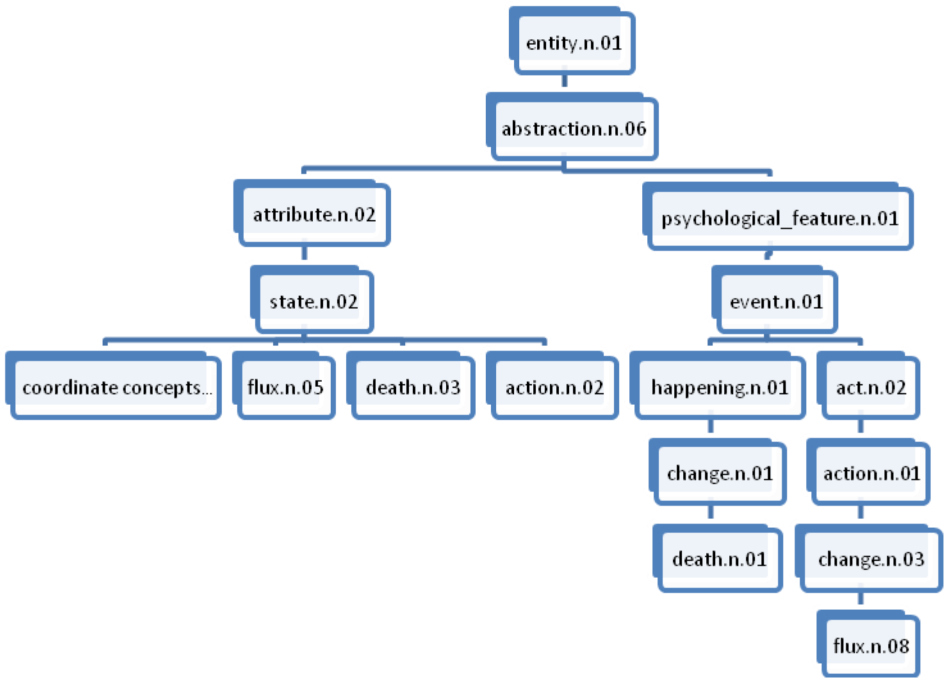

Figure 1: Visualising lexical relations from Listing 4.6

Using WordNet has helped to separate out different senses of a highly charged (*i.e.* polysemous) word which is also being used poetically, thus accruing imaginative and contextual correspondences (*e.g.* action). We have gained distinguishing insights such as *death-as-state* versus *death-as-event* which resolve the association of apparently incompatible ideas within the play.

Conclusions

We have demonstrated that NLTK offers a middle way between the off-the-shelf packages, which offer a friendly user interface but constrain the researcher to think in terms of the functionality on offer; and developing software from scratch in a general-purpose programming language like Java, which offers more flexibility but a steep learning curve for the literary scholar. If the researcher can spare the time to learn the basics of the Python programming language, this opens an Aladdin's cave of possibilities, including a wide range of language-analysis tools within NLTK, such as Text object methods, customised tokenisation, compounding analysis, frequency distributions and conditional frequency distributions, collocation discovery, WorldNet for exploring lexical semantics and imagery. We urge readers to try NLTK for themselves!

Endnotes

[1] **Project Gutenberg:** http://www.gutenberg.org/wiki/Main_Page

[2] **Collocations HOWTO:** http://nltk.googlecode.com/svn/trunk/doc/howto/collocations.html

References:

Bird, S., E. Klein and E. Loper. (2009). *The Natural Language ToolKit*. http://www.nltk.org/

Brierley, C. and Atwell, E. (2009). "Exploring Imagery and Prosody in Literary Corpora with NLP Tools". Submission for: Willie van Peer, Sonia Zyngier, Vander Viana (editors) *Literary Education and Digital Learning*. IGI Global

Brown, R. and Gilman, A. (2003). “The Pronouns of Power & Solidarity”. In: Christina Bratt Paulston, G. Richard Tucker (editors) *Sociolinguistics: the essential readings*. Wiley-Blackwell

Budanitsky, A. and Hirst, G. (2006). “Evaluating WordNet-based Measures of Lexical Semantic Relatedness”. In *Computational Linguistics*, 32(1), March 2006, 13-47.

Fellbaum, C. (Ed.) (1998). *WordNet*: *An Electronic Lexical Database*. The MIT Press

Jenkins, H. (Ed.) (2003). *The Arden Shakespeare: Hamlet*. London: Thomson Learning

Lloyd, S.M. (1982). *Roget’s Thesaurus*. Harlow: Longman Group Ltd

Manning, C.D. and Schutze, H. (1999). *Foundations of Statistical Natural Language Processing*. London: MIT Press

Mason, O. (2001). *Programming For Corpus Linguistics: How To Do Text Analysis With Java*. Edinburgh University Press.

Nevalainen, T. (2001). *Reading Shakespeare's Dramatic Language: A Guide*. London: Thomson

Rayson, P. (2003). “Matrix: A statistical method and software tool for linguistic analysis through corpus comparison”. *Ph.D. thesis*. Lancaster University.

Scott, M. (2004). *WordSmith Tools* version 4. Oxford: Oxford University Press.

Sinclair, J.M. (Ed.) (1994). *Collins English Dictionary* (3rd. edition). Aylesbury: Harper-Collins

Wilson-Knight, G. (2001). *The Wheel of Fire.* London: Routledge Classics