NLTK's YCOE Module

P. S. Langeslag

March 15, 2024

Introduction

Although NLTK lacks a pipeline for Old English, it nevertheless ships with a corpus reader for YCOE, the York-Toronto-Helsinki Parsed Corpus of Old English Prose. YCOE is a subset of DOEC (the *Dictionary of Old English* Corpus) that has been annotated for parts of speech (POS) and syntax. While the creators of YCOE assumed users would rely on the rather involved CorpusSearch program to search for syntactic constructions, the present guide demonstrates how the POS- and syntactically-tagged instances of the corpus, as well as its tokenized plaintext, may be accessed in Python with NLTK.

Importing the Corpus Reader and Loading the Corpus

NLTK's `ycoe` module is found at `nltk.corpus.reader.ycoe`. You'll want to import at least the `YCOECorpusReader` and `documents` functions. The latter will tell you what texts are included:

```
>>> from nltk.corpus.reader.ycoe import YCOECorpusReader
>>> from nltk.corpus.reader.ycoe import documents
>>> documents
{'coadrian.o34': 'Adrian and Ritheus', 'coaelhom.o3':
 'Ælfric, Supplemental Homilies', 'coaelive.o3': ...}
```

However, as YCOE is not itself included with NLTK, you have no access to these documents until you have told the corpus reader where the corpus is located. Hence we'll want to load the corpus next.

A copy of YCOE is available at the Oxford Text Archive. Once downloaded (we'll assume into a folder `ycoe/`), it may be loaded into the corpus reader as follows:

```
>>> root = 'ycoe/'
>>> ycoe = YCOECorpusReader(root)
```

As part of its corpus loading routine, the corpus reader checks that the files match its own `documents` function. If you install YCOE locally and the corpus reader throws an error on loading to do with the contents of the `pos/` and `psd/` directories, your copy of YCOE differs from the one assumed by NLTK.

Querying the Corpus

Although our new object also has a function `ycoe.documents()`, it in fact returns an incomplete output of `nltk.corpus.reader.ycoe.documents`, so if you now run `ycoe.documents()` you will only see the filenames stripped of their `.pos` and `.psd` extensions, not their descriptions as with the original function. This is why we have imported `nltk.corpus.ycoe.documents` separately; we can now simply enter

```
>>> documents
```

whenever we need an overview.

As `help(ycoe)` will tell you, you now have several commands available to you. The more basic among these are the following:

- `documents()` (in the above example accessed under `ycoe.documents()`; this prefix will be taken for granted in the remainder of this index): offers a list of document codes, but without their descriptions.
- `fileids()`: this gives the same information again, now as full filenames. You will see that the corpus contains every text twice: once as a `.pos` file recording parts of speech and once as a `.psd` file describing syntactic structure. The corpus root accordingly has two subfolders `pos/` and `psd/`, each of which contains an instance of each document.
- `words()`: Returns (part of) the corpus as a Python list of words. This is the level you're familiar with as the basic unit of NLP analysis. Please note, however, that YCOE is not a unicode corpus: instead of the special characters <æ, þ, ð>, it prints `+a`, `+t`, and `+d`. Instead of the tironian note (⁊ for Old English "and," or Latin "et") it prints an ampersand; it has inherited that last convention from DOEC.
- `sents()`: same as above, but with a list of sentences as the top-level container. This level is of particular relevance for syntactic analysis.
- `paras()`: same as above, but with a list of paragraphs as the top-level container. This level may be of some use if you need a slightly larger chunk of text for analysis.

These commands allow you to query the corpus much as you would any other: you can conduct word counts (e.g. `len(ycoe.words('covinsal'))`, calculate relative word frequency and lexical diversity as explained in the NLTK textbook, and so on. For any serious usage you would be well advised to convert the document content to unicode by replacing each instance of `+t` with þ etc. as part of any routine you write, for instance as follows:

```
>>> def multireplace(input):
...     substitutions = {'+a': 'æ', '+t': 'þ', '+d': 'ð', '+A': 'Æ', \
...     '+T': 'Þ', '+D': 'Ð'}
...     utflist = []
...     for w in input:
...         for k, v in substitutions.items():
...             w = w.replace(k, v)
...         utflist.append(w)
...     return utflist
...
```

```
>>> multireplace(ycoe.sents('coverhom')[5])
['Wæs', 'þæt', 'se', 'ilca', 'Caifas', 'þe', 'ær', ...]
```

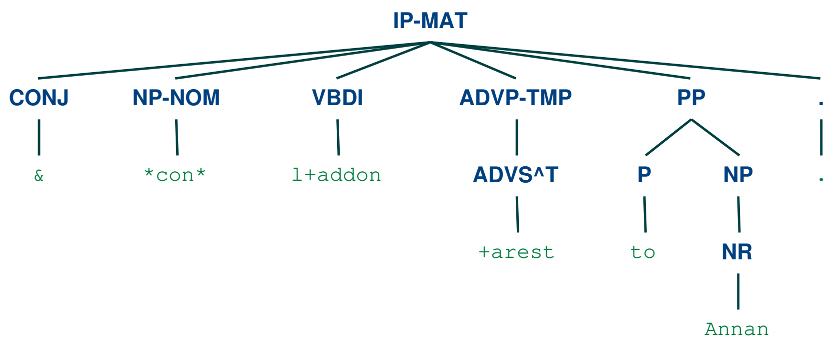
But the added value of YCOE lies in the fact that it is a tagged and parsed corpus as opposed to plaintext. To take advantage of these metadata, NLTK provides the following functions:

- `tagged_words()`: returns each word in a document as a tuple comprised of the word form and its part of speech. Consult the documentation distributed with the corpus (and available online) for explanations on how to read the POS tags. This function reads the `.pos` file only.
- `tagged_sents()`: same as above, but with a list of sentences as the top-level container.
- `tagged_paras()`: same as above, but with a list of paragraphs as the top-level container.
- `parsed_sents()`: returns each sentence syntactically parsed into a nesting structure of lists and tuples, each constituent syntactically labelled. Consult the documentation for explanations on how to read the syntactic tags. This function reads the `.psd` file only.

Visualization and Processing

YCOE's `.pos` and `.psd` files are not meant for human consultation, but for natural language processing. Passing them through NLTK's `YCOECorpusReader` allows us to process these data at the word, sentence, or paragraph level. For instance, the syntactic trees encoded in the `.psd` files may be visualized using the `draw()` command, which uses the `tkinter` visualization interface. Thus we may produce a tree visualization of the fourth sentence in the Vercelli homilies as follows:

```
>>> ycoe.parsed_sents('coverhom')[3].draw()
```



If you are interested in locating specific sorts of syntactic constructions in the corpus, you will either have to learn to use CorpusSearch and install it and YCOE locally, or you will have to write some rather clever regular expressions queries of your own to navigate the Python object structure made available through the corpus reader.

POS metadata are easier to integrate into a Python routine. Consider the simple word frequency queries you learned in chapter 3 of the NLTK textbook. We can apply the same routines to YCOE as follows (here analyzing the first series of Ælfric's *Catholic Homilies*):

```
>>> from nltk import FreqDist
>>> ch1FreqDist = FreqDist(ycoe.words('cocathom1.o3'))
>>> ch1FreqDist.most_common(20)
[('.', 11160), ('&', 4853), (':', 2958), ('on', 2371), ('he', 2175),
('+t+at', 2064), ('to', 1858), ('his', 1660), ('mid', 1572), ('+ta',
1550), ('+te', 1545), ('se', 1429), ('swa', 1312), ('hi', 1248),
('+tam', 1182), ('is', 1106), ('ne', 993), ('+de', 936), ('for',
931), ('him', 796)]
```

Of course we have several strategies at our disposal to improve this output: we can strip out punctuation, exclude stopwords (CLTK provides a stopword list for Old English, or you can do up your own), and we can run our unicode conversion function to make the elements easier to read. But with YCOE, we are now additionally able to filter by part of speech. The first step in this approach is as simple as docking the `FreqDist()` function to `tagged_words` instead of `words`:

```
>>> ch1FreqDist = FreqDist(ycoe.tagged_words('cocathom1.o3'))
>>> ch1FreqDist.most_common(20)
```

```
[(('.', '.'), 6081), (('.', ','), 5079), (('&', 'CONJ'), 4853),
(('on', 'P'), 2368), (('he', 'PRO^N'), 2175), ((':', ','), 1801),
(('to', 'P'), 1676), (('his', 'PRO$'), 1629), (('mid', 'P'), 1570),
(('se', 'D^N'), 1427), (('+t+at', 'C'), 1407), (('+te', 'C'), 1353),
(('+tam', 'D^D'), 1182), ((':', '.'), 1157), (('is', 'BEPI'), 1106),
(('hi', 'PRO^N'), 934), (('for', 'P'), 931), (('+de', 'C'), 841),
(('him', 'PRO^D'), 796), (('ne', 'NEG'), 791)]
```

All we have to do then is filter by the second element in the inner tuple. For instance, we may list the most frequent nouns as follows:

```
>>> import re
>>> freqNouns = []
>>> for w in ch1FreqDist.most_common():
...     if re.search('^N(\^.*)*$', w[0][1]):
...             freqNouns.append((w[0][0], w[1]))
>>> freqNouns[:20]
[('manna', 176), ('h+alend', 166), ('mannum', 155), ('man', 154),
('life', 144), ('f+ader', 123), ('d+age', 122), ('sunu', 121),
('dea+de', 109), ('apostol', 107), ('gast', 94), ('+ting', 82),
('rice', 82), ('folce', 78), ('wordum', 77), ('deofles', 76),
('folc', 70), ('geleafan', 69), ('mannes', 68), ('f+ader', 67)]
```

A few notes to help you understand this code:

- Because YCOE's part-of-speech labels consist of extended as well as top-level labels, which are separated by a circumflex (^), it's convenient to use regular expressions to match the desired labels. The pattern used above looks for a string beginning with `N` (^ indicating the start of the string) and either ending there ($) or following up with a sequence consisting of ^ (escaped by backslash because ^ normally means "start of string" in a regular expressions environment) and any number (`*`, meaning zero or more) of any subsequent characters (`.`).
- The bracketed numbers identify elements in a list or tuple, counting from 0. In this example, we are dealing with nested tuples where `w[1]` is the number of times the form occurs in the corpus, `w[0][1]` the POS label, and `w[0][0]` the word form.
- Note that if this syntax throws a type error, as it will on the Vercelli homilies (`coverhom`), this means one of the objects in the iteration structure is not a string, suggesting an error in the data (an empty field perhaps?).

To search for parts of speech other than nouns, study YCOE's part-of-speech labels and draw up a regular expressions query to match the desired parts of speech. Remember that you can match multiple parts of speech as follows:

```
...     if re.search('^(N|ADJ)(\^.*)*$', w[0][1]):
```

Now that you are able to single out specific parts of speech, you can apply all of the bag-of-words techniques learned so far, such as lexical diversity, with greater precision, should you have a use case for doing so.