CLTK

P. S. Langeslag

March 15, 2024

Contents

Preface

This guide was written as required reading for a humanities course on natural language processing (NLP) to offer more guidance and a lower point of entry than is given in CLTK's official demo notebook, let alone in the official documentation. I'm using the opportunity to explain a few other concepts about Python because my audience is largely new to it. As I'm not myself a CLTK contributor, I cannot vouch for the below instructions being accurate in every respect.

Please note that CLTK 1.0 contains vestiges of the legacy 0.1.x releases that no longer serve as much of a purpose for the user. Some such functionality, including the corpus downloader, is here described in appendices rather than in the document's body.

Lining Up Text

Having discontinued the corpus reader found in 0.1.x releases, CLTK 1.0 expects you to identify one or more documents directly on disk as input for processing. For the purposes of our course, a plaintext corpus of Old English homiletic prose has been made available;

the below code assumes this is available at `echoe/`. As you won't know the contents of the corpus off by heart, you may want to inspect this director with help from the `os` module:

```
>>> import os
>>> os.listdir('echoe/')
```

You can then open and read individual files as follows:

```
>>> v9file = open('echoe/394.11.txt')
>>> v9 = v9file.read()
>>> v9
'men ða leofestan ...'
```

Alternatively, you can use NLTK's corpus reader as explained in the NLTK textbook:

```
>>> from nltk.corpus import PlaintextCorpusReader
>>> corpus_root = 'echoe'
>>> echoe = PlaintextCorpusReader(corpus_root, '.*')
```

You can then list available files as follows:

```
>>> echoe.fileids()
['018.40.txt', '018.42.txt', '021.27.txt', '021.28.txt', ...]
```

and open individual files as follows:

```
>>> v9 = echoe.raw('394.11.txt')
```

But an advantage of the corpus reader is that you can alternatively prepare the entire corpus for analysis (just be forewarned that the processing of a substantial corpus will take some time, and if your pipeline includes a lemmatizer you will run out of memory fast):

```
>>> corpus = echoe.raw()
```

If you want to evaluate a different text or corpus that is not yet in plaintext, you will have to either parse or strip it first. For instance, you can parse HTML or XML using the `BeautifulSoup` module from the `bs4` package as follows:

```
>>> from bs4 import BeautifulSoup
>>> raw = open('somefile.html')
>>> soup = BeautifulSoup(raw, 'html.parser')
>>> text = soup.get_text()
```

While you may alternatively clean up your corpus before loading it in Python, doing all your preprocessing in a Python script is the better approach in terms of both transparency and reproducibility.

Default Pipeline Functionality

The functionality shown in the official demo notebook is available off the shelf, with the important caveat that the demo is centrally focused on Latin because it has the most comprehensive pipeline. The present section similarly confines itself to a selected few languages, but it explains CLTK functionality available directly within Python at greater length.

Old English

The only CLTK module we normally need to import is `cltk.NLP`. We then tie the `NLP` module to a process variable specifying the language with which we'll be working. In response, the interpreter outputs the full list of processes available for that language:

```
>>> from cltk import NLP
>>> pipeline = NLP(language='ang')
 CLTK version '1.0.24'.
Pipeline for language 'Old English (ca. 450-1100)' (ISO: 'ang'):
`MultilingualTokenizationProcess`, `OldEnglishLemmatizationProcess`,
`OldEnglishEmbeddingsProcess`, `StopsProcess`, `OldEnglishNERProcess`.
```

For Old English, CLTK ships with a language-independent tokenizer and language-specific stopword and lemma lists. It draws on an external library (fastText) with Old English support for word embedding (i.e. vector modelling). Although it claims to have a language-specific named-entity-recognition (NER) process, this process appears to be essentially empty: unlike in the Latin model, the Old English directory structure within the `cltk_data` folder lacks a list of proper names, and it employs a library (spaCy) that has no support for Old English. What we can expect CLTK to do off the shelf, then, is tokenize, lemmatize, and vector model.

CLTK has been set up to run the pipeline with all these processes (the default) or a hand-culled selection of them, then inspect the results. Thus for instance:

```
>>> processed = pipeline.analyze(text=v9)
>>> processed.tokens[:12]
['men', 'ða', 'leofestan', 'we', 'geleornodon', 'on',
'godcundum', 'gewritum', 'þæt', 'æghwylces', 'monnes',
'sawul']
>>> processed.lemmata[:12]
['mann', 'þa', 'leofestan', 'we', 'geleornodon', 'on',
'godcundum', 'gewrit', 'þæt', 'æghwylces', 'monnes',
'sawul']
```

Note that in this usage whenever it fails to identify a lemma, it returns the inflected form as found (`leofestan`, `geleornodon`, etc.).

With Old English, there isn't much else you can do but lemmatize, but you can ask CLTK whether a given token is registered as a stopword, which may be of some limited use in further processing:

```
>>> processed.words[0].stop
False
>>> processed.words[1].stop
True
```

Another way of querying data is through the `words` accessor, which organizes what has been learned on a per-token basis. For Old English, this means you can access word form and inferred lemma from a single object.

```
>>> processed.words[0]
Word(index_char_start=0, index_char_stop=3, index_token=0,
index_sentence=None, string='men', pos=None, lemma='mann',
```

```
stem=None, scansion=None, xpos=None, upos=None,
dependency_relation=None, governor=None, features={},
category={}, stop=False, named_entity=False, syllables=None,
phonetic_transcription=None, definition=None)
```

```
>>> for token in processed.words[:3]:
...     print(token.string, 'is a form of', token.lemma)
men is a form of mann
ða is a form of þa
leofestan is a form of leofestan
```

The final functionality available for Old English is word embeddings. This may not look like much when queried directly (try querying `processed.embeddings[0]`), but these "vectors" or lists of logarithms store valuable data on each form's relationships to other word forms in the corpus. This means that if we process a large enough corpus, we have access to a great deal of implicit semantic information, as we'll learn in week 12.

Middle English

Although CLTK claims a Middle English pipeline, it contains no more than a stopword list, in addition to the universal tokenizer:

```
>>> pipeline = NLP(language='enm')
✗ CLTK version '1.0.24'.
Pipeline for language 'Middle English' (ISO: 'enm'):
`MiddleEnglishTokenizationProcess`, `StopsProcess`.
```

Latin

To demonstrate the capabilities of CLTK more fully, we will have to resort to Latin. That pipeline is demonstrated in the official demonstration notebook, so there is no need to rehearse it here. For completeness' sake, here is the pipeline prompt for Latin:

```
>>> pipeline = NLP(language='lat')
✗ CLTK version '1.0.24'.
Pipeline for language 'Latin' (ISO: 'lat'):
`LatinNormalizeProcess`, `LatinStanzaProcess`,
`LatinEmbeddingsProcess`, `StopsProcess`,
`LatinNERProcess`, `LatinLexiconProcess`.
```

Additional Functionality

Detailed Output

As we saw above, the Old English lemmatizer as contained in the default pipeline returns either a headword or, if none is found, the input string. However, the underlying function has two optional arguments with which we may tweak this return: `best_guess=False`, which causes it to return all hits rather than just the one it considers the most likely, and `return_frequencies=True`, which returns the logarithm of the returned headword's relative frequency in the underlying word list. To access this functionality, we can call the function directly, bypassing the pipeline:

```
>>> from cltk.lemmatize.ang import OldEnglishDictionaryLemmatizer as lem
>>> lem.lemmatize_token('man')
'man'
>>> lem.lemmatize_token('man', return_frequencies=True, best_guess=False)
[('mann', -6.829400539827225), ('man', -4.832846657953158)]
```

In the above example, the form "man" may represent either the impersonal pronoun *man* "one," identical in meaning and origin to German *man*, or it may be the noun *mann* "person; man," similar in meaning to German *Mann*. If we run the lemmatizer with default options, all it returns is the likeliest headword, i.e. the pronoun. With the two optional arguments, it returns a list of tuples, each of which consists of a headword and the logarithm of its relative frequency in the word list. As these are all negative logarithms, whichever is closest to zero is evaluated as the likeliest headword, while absolute zero means only one match has been found. To evaluate a document as opposed to a single word form, you'll have to tokenize it first:

```
>>> sample = 'MEN ða leofestan manað us and myngaþ þeos halige boc þæt we \
sien gemyndige ymb ure sawle þearfe'
>>> from nltk.tokenize.punkt import PunktLanguageVars as punkt
>>> tokens = punkt.word_tokenize(sample.lower())
>>> lem.lemmatize(tokens, return_frequencies=True, best_guess=False)
[[('mann', -6.829400539827225)], [('þe', -3.109198614584248),
('ða', -3.1405210857132895), ('þa', -2.344858341627749),
('se', -2.9011463394704973)], [], [], [('we', -5.037641070599171),
('us', -5.826098430963441)], [('and', -2.8869365088978443)], [], ... ]
```

Note that with these settings, an input with no hits returns an empty value (`[]`) with no frequency data.

Additional Utilities

CLTK offers a few utilities additional to those integrated into the pipeline. For Old English, we have access to a syllabifier, an IPA transcription utility, and a transliterator of Old English runes. These may be used as follows:

```
>>> from cltk.phonology.ang.transcription import Transcriber
>>> Transcriber.transcribe('habbað æfre ānrædne ġelēafan')
'hab:að æ:fre a:nræ:dne jelæ:avan'
```

```
>>> from cltk.phonology.ang.phonology import OldEnglishSyllabifier
>>> syll = OldEnglishSyllabifier()
>>> syll('monan')
['mo', 'nan']
```

```
>>> from cltk.phonology.ang.transliteration import Transliterate
>>> Transliterate.transliterate('ᚱᛗᛗᚹᚪᛚᚢᛋ ᚠᚾᛞ ᚱᛖᚢᛗᚹᚪᛚᚢᛋ ᛏᚹᛟᛖᚷᛖᚾ ᚷᛁᛒᚱᚩᚦᚩᚱ')
'romwalus and reumwalus twoegen gibrothær'
```

Please be advised that these utilities are not to be taken as authoritative for the purposes they were designed to serve. Also please note that the syllabifier only takes a single word at a time as input.

Appendix 1: Installation

This appendix may be of interest if you're looking to install CLTK on your local machine, and it may help solve or at least explain some issues if you're relying on JupyterCloud or JupyterWeb.

CLTK 1.0 only supports Python 3.7, 3.8, and 3.9 on POSIX-compliant operating systems (e.g. macOS or Linux). As these are somewhat dated versions of Python now, you'll most likely want to work with a Python version manager such as `pyenv`, which allows you to switch between multiple versions of Python as needed. Real Python has a great introduction to `pyenv`; as the process of installing it can be a little involved, I won't repeat it here. Once you've installed it, you'll want to use it to install a compatible version of Python as follows:

```
pyenv install -v 3.9.10
```

You'll ideally want to set up a virtual environment where you can run CLTK not just on retro Python but also with the specific dependencies it needs, in isolation from your system-wide Python setup. This is because CLTK has highly specific version requirements and may not always work with the latest version of a required package. To install CLTK using `pip` in a virtual environment, first set up the environment and give it a title (here "nlp") as follows:

```
pyenv virtualenv 3.9.10 nlp
```

Then if your project directory is to be located at `~/python/nlp/`, change to that working directory and enter:

```
pyenv local nlp
```

Test whether the correct version of Python is running as follows:

```
pyenv which python
python -V
```

If the first response points to a folder containing your project name, and the second reports the desired version of Python, you may proceed:

```
pip install cltk
```

`pip` will automatically pull in compatible versions of all dependencies and install them below your project folder rather than in your system-wide `sites-packages/` folder. It is important to activate your virtual environment whenever you start working with it, but if you have `eval "$(pyenv virtualenv-init -)"` set up in your environment this should be done automatically, and you can set up your shell prompt to reflect this.

CLTK will create a folder `cltk_data/` in which it stores your downloaded corpora and language models. By default, this folder is created directly within your home folder, even when working within a virtual environment, though (supposedly!) setting the environment variable `$CLTK_DATA` (e.g. by adding `CLTK_DATA="/home/username/python/nlp/ cltk_data"` to your `~/.bashrc`) should give you control over where it goes.

Appendix 2: Downloading Corpora

CLTK has a function `FetchCorpus()` that links to repositories for a range of premodern corpora. For Old English, it offers the complete Old English poetic corpus, which it downloads in HTML format, then converts to JSON so it may be accessed in Python (though it fails to strip out `<strong>` tags meant to indicate emendations in the print edition). The other Old English "corpus" on offer is in fact CLTK's Old English model, which allows us to explore the package's processing strategies in isolation from the general pipeline; but that model is downloaded the first time we run the NLP on an Old English text, so there is no need to download it separately. There is surely more value in downloading the Latin and Greek corpora.

A full list of all linked corpora may be obtained by issuing

```
>>> import cltk
>>> cltk.data.fetch.LANGUAGE_CORPORA.values()
```

or, to sort them by language:

```
>>> cltk.data.fetch.LANGUAGE_CORPORA.items()
```

Both these lists are rather hard to read, however. It may help to print the values one per line, like so:

```
>>> print(*cltk.data.fetch.LANGUAGE_CORPORA.values(), sep="\n")
```

or you can narrow down by language. To this end, you'll need the relevant language codes. These are contained in `cltk.languages.glottolog.LANGUAGES`, though to print them in an acceptably clean list would require a few lines of code. Instead, it may help to know that the languages codes used follow the ISO 639-3 standard, with values like the following:

Table 1: Selection of medieval northern European language codes in the ISO 639-3 standard

Code	Language
`ang`	Old English
`enm`	Middle English
`gmh`	Middle High German
`goh`	Old High German
`grc`	Acient Greek
`lat`	Latin
`non`	Old Norse
`osx`	Old Saxon/Old Low German

Alternatively, you may search CLTK's index of languages using part of a language's Modern English name as a keyword string:

```
>>> cltk.languages.utils.find_iso_name('english')
['enm', 'ang']
>>> cltk.languages.utils.get_lang('ang')
Language(name='Old English (ca. 450-1100)', glottolog_id='olde1238',
latitude=51.06, longitude=-1.31, dates=[], family_id='indo1319',
parent_id='angl1265', level='language', iso_639_3_code='ang', type='h')
```

Returning to our list of corpora, we can now limit it to languages we're interested in:

```
>>> from cltk.data.fetch import FetchCorpus
>>> FetchCorpus('ang').list_corpora
['old_english_text_sacred_texts', 'ang_models_cltk']
```

As note above, there are in fact only two exclusively Old English corpora available, one of which is the Anglo-Saxon Poetic Records (here called "Sacred Texts" because that is the title of the somewhat iffy website now hosting this HTML corpus), the other is CLTK's own model tree for Old English. But what this return doesn't tell you is that CLTK is also aware of multilingual corpora, one of which includes several Old English prose texts. Use the "language code" `multilingual` to discover these:

```
>>> FetchCorpus('multilingual').list_corpora
['multilingual_treebank_proiel', 'multilingual_treebank_iswoc',
'multilingual_treebank_torot']
```

The ISWOC treebank is the corpus we want; it contains a modest subset of the *Dictionary of Old English* Corpus (DOEC). We may download corpora as follows:

```
>>> FetchCorpus('multilingual').import_corpus('multilingual_treebank_iswoc')
>>> for i in FetchCorpus('ang').list_corpora:
...     FetchCorpus('ang').import_corpus(i)
```

The latter command tells `FetchCorpus()` to download each of its Old English corpora in turn. Here please note the following:

1. Where other programming languages use braces to set off loops and conditionals, Python relies on spacing (tabbing). The combination of `for` and the colon in the first line tells the interpreter that the command is not yet complete, hence the interpreter prints ellipses rather than the usual prompt, but it is up to us to then indent the next line of code using the Tab key. When we're done, we hit return twice so the interpreter knows the loop is complete.
2. `i` serves as the variable and can in fact be given any non-reserved name. When we say `for i`, we are telling the interpreter to carry out the indented action for each member of a list; the syntax surrounding `i` in the next line then tells it what to do with each value. Note that `i` in this case lacks quotation marks, unlike the ISWOC identifier in the first line, because text in quotation marks is interpreted as literal whereas leaving them out makes clear that it refers to a defined entity (a variable in this case).
3. Upon completion of the download, the interpreter may keep saying `loaded 100%`. It is not stuck; you can simply type your next command.
4. If you run the same `for`-loop code for a language like Latin, which has a larger set of corpora linked in CLTK, you may hit an error because one of the repositories is no longer available at the listed URL. If this is the case, you can try downloading the corpora one by one to see which one failed.

You will find your downloaded corpora in your `cltk_data/` folder, in subfolders named for their language keys. Whereas CLTK 0.1.x supplied a corpus reader, this feature seems to have been abandoned with release 1.0. This is unfortunate inasmuch as it results in a lack of continuity between the experience of downloading and accessing a corpus; the latter will have to make reference to the location of the downloaded files on disk. You can, however, point NLTK's general-purpose corpus reader (not its YCOE reader) to that location and

use some of the functionality of NLTK. For instructions, see the NLTK textbook under §2.1.9 Loading Your Own Corpus.