Getting Started with LatinCy

Patrick J. Burns

2023-12-13

Table of contents

Preface

"Getting Started with LatinCy" is an always-a-work-in-progress combination of documentation and demo notebooks for working with the LatinCy models on a variety of Latin text analysis and NLP tasks.

Key links

Models: https://huggingface.co/latincy
Universe: https://spacy.io/universe/project/latincy
Preprint: https://arxiv.org/abs/2305.04365

This book has been written using Jupyter notebooks which have then been collated with Quarto. To learn more about Quarto books visit https://quarto.org/docs/books.

Abbreviations

Where possible, I will include references to standard NLP works using the following abbreviations:

SLP Jurafsky, D., and Martin, J.H. 2020. *Speech and Language Processing.* 3rd Edition, Draft. https://web.stanford.edu/~jurafsky/slp3/. (Jurafsky and Martin 2020)

Part I

Model/Pipeline Basics

1 Installing LatinCy models

1.1 Installing spaCy

The LatinCy models are designed to work with the spaCy natural language platform, so you will need to have this package installed before anything else. The following cell has the pip install command for spaCy. At the time of writing, the latest version available for spaCy compatible with LatinCy is v3.7.2.

NB: To run the cells below, uncomment the commands by removing the # at the beginning of the line. The exclamation point at the beginning of the line is shorthand for the `%system` magic command in Jupyter which can be used to run shell commands from within a notebook.

```
# !pip install spacy==3.7.2
```

1.2 Installing the LatinCy models

LatinCy models are currently available in three sizes: 'sm', 'md', and 'lg'. We will use the different models throughout the tutorials, so let's install all three now so that they are available for future chapters.

```
# !pip install https://huggingface.co/latincy/la_core_web_sm/resolve/main/la_core_web_sm-a
# !pip install https://huggingface.co/latincy/la_core_web_md/resolve/main/la_core_web_md-a
# !pip install https://huggingface.co/latincy/la_core_web_lg/resolve/main/la_core_web_lg-a
```

1.3 Installing additional packages

We will also use other packages throughout these tutorials. They are included here for your convenience, as well as in the `requirements.txt` file in the code repository for this Quarto book.

```
# !pip install pandas
# !pip install matplotlib
# !pip install seaborn
# !pip install scikit-learn
# !pip install tqdm
```

2 Loading LatinCy models

2.1 Loading LatinCy models with spaCy

```
# Imports

import spacy
from pprint import pprint
```

```
nlp = spacy.load('la_core_web_lg')
```

2.2 Creating a spaCy doc from a string

```
text = "Haec narrantur a poetis de Perseo."
doc = nlp(text)
print(doc)
```

```
Haec narrantur a poetis de Perseo.
```

```
print(type(doc))
```

```
<class 'spacy.tokens.doc.Doc'>
```

```
print(doc.__repr__())
```

```
Haec narrantur a poetis de Perseo.
```

```
print(doc.text)
```

Haec narrantur a poetis de Perseo.

```
print(type(doc.text))
```

```
<class 'str'>
```

Part II

NLP Tasks

3 Sentence segmentation

3.1 Sentence segmentation with LatinCy

```
# Imports & setup

import spacy
from pprint import pprint
nlp = spacy.load('la_core_web_lg')
text = "Haec narrantur a poetis de Perseo. Perseus filius erat Iovis, maximi deorum. Avus
doc = nlp(text)
print(doc)
```

```
Haec narrantur a poetis de Perseo. Perseus filius erat Iovis, maximi deorum. Avus eius Acris
```

Sentence segmentation is the task of splitting a text into sentences. For the LatinCy models, this is a task that is jointly learned with dependency parsing and has been trained to terminate sentences at both strong and weak stops, following the example of Clayman (1981) (see also, Wake (1957), Janson (1964)), who writes: "If all stops are made equivalent, i.e. if no distinction is made between the strong stop, weak stop and interrogation mark, editorial differences will be kept to a minimum."

Given a spaCy `Doc`, the `sents` attribute will produce a `generator` object with the sentence from that document as determined by the dependency parser. Each sentence is a `Span` object with the start and end token indices from the original `Doc`.

```
sents = doc.sents
print(type(sents))
```

```
<class 'generator'>
```

Like all `Span` objects, the text from each sentence can be retrieved with the `text` attribute. For convenience below, we convert the generator to list so that we can iterate over it multiple times. Here are the three (3) sentences identified in the example text as well as an indication of the sentences' `type`, i.e. `<class 'spacy.tokens.span.Span'>`.

```
sents = list(sents)

for i, sent in enumerate(sents, 1):
    print(f'{i}: {sent.text}')
```

```
1: Haec narrantur a poetis de Perseo.
2: Perseus filius erat Iovis, maximi deorum.
3: Avus eius Acrisius appellabatur.
```

```
sent = sents[0]
print(type(sent))
```

```
<class 'spacy.tokens.span.Span'>
```

Sentences have the same atrributes/methods available to them as any span (listed in the next cell). Following are some attibutes/methods that may be particularly relevant to working with sentences.

```
sent_methods = [item for item in dir(sent) if '_' not in item]
pprint(sent_methods)
```

```
['conjuncts',
 'doc',
 'end',
 'ents',
 'id',
 'label',
 'lefts',
 'rights',
 'root',
 'sent',
 'sentiment',
 'sents',
 'similarity',
 'start',
 'subtree',
 'tensor',
 'text',
 'vector',
 'vocab']
```

You can identify the `root` of the sentence as determined by the dependency parser. Assuming the parsing in correct, this will be the main verb of the sentence.

```
print(sent.root)
```

```
narrantur
```

Each word in the sentence has an associated vector. Sentence (any `Span` in fact) has an associated vector as well that is the mean of the vectors of the words in the sentence. As this example uses the `lg` model, the vector has a length of 300.

```
print(sent.vector.shape)
```

```
(300,)
```

This vector then can be used to compute the similarity between two sentences. Here we see our example sentence compared to two related sentence: 1. a sentence where the character referred to is changed from Perseus to Ulysses; and 2. the active-verb version of the sentence.

```
sent.similarity(nlp('Haec narrantur a poetis de Ulixe.'))
```

```
0.9814934441998264
```

```
sent.similarity(nlp('Haec narrant poetae de Perseo.'))
```

```
0.7961655556379285
```

We can retrieve the start and end indices from the original document for each sentence.

```
sent_2 = sents[1]
start = sent_2.start
end = sent_2.end
print(start)
print(end)
print(sent_2.text)
print(doc[start:end].text)
```

```
7
15
Perseus filius erat Iovis, maximi deorum.
Perseus filius erat Iovis, maximi deorum.
```

References

SLP Chapter 2, Section 2.4.5 “Sentence Segmentation”, pp. 24 link

4 Word Tokenization

4.1 Word tokenization with LatinCy

```
# Imports & setup

import spacy
from pprint import pprint
nlp = spacy.load('la_core_web_md')
text = "Haec narrantur a poetis de Perseo. Perseus filius erat Iovis, maximi deorum. Avus
doc = nlp(text)
print(doc)
```

```
Haec narrantur a poetis de Perseo. Perseus filius erat Iovis, maximi deorum. Avus eius Acris
```

Word tokenization is the task of splitting a text into words (and wordlike units like punctuation, numbers, etc.). For the LatinCy models, tokenization is the fundamental pipeline component on which all other components depend. SpaCy uses non-destructive, "canonical" tokenization, i.e. non-destructive, in that the original text can be untokenized, so to speak, based on `Token` annotations and canonical in that indices are assigned to each token during this process and these indices are used to refer to the tokens in other annotations. (Tokens *can* be separated or merged, but this requires the user to actively undo and redefine the tokenization output.) LatinCy uses a modified version of the default spaCy tokenizer that recognizes a splits enlitic *-que* using a rules-based process. (NB: It is in the LatinCy development plan to move enclitic splitting to a separate post-tokenization component.)

The spaCy `Doc` object is an iterable and tokens are the iteration unit.

```
tokens = [item for item in doc]
print(tokens)
```

```
[Haec, narrantur, a, poetis, de, Perseo, ., Perseus, filius, erat, Iovis, ,, maximi, deorum,
```

```
token = tokens[0]
print(type(token))
```

```
<class 'spacy.tokens.token.Token'>
```

The text content of a `Token` object can be retrieved with the `text` attribute.

```
for i, token in enumerate(tokens, 1):
    print(f'{i}: {token.text}')
```

```
1: Haec
2: narrantur
3: a
4: poetis
5: de
6: Perseo
7: .
8: Perseus
9: filius
10: erat
11: Iovis
12: ,
13: maximi
14: deorum
15: .
16: Avus
17: eius
18: Acrisius
19: appellabatur
20: .
```

Note again that the token itself is a spaCy `Token` object and that the `text` attribute returns a Python string even though their representations in the Jupyter Notebook look the same.

```
token = tokens[0]
print(f'{type(token)} -> {token}')
print(f'{type(token.text)} -> {token.text}')
```

```
<class 'spacy.tokens.token.Token'> -> Haec
<class 'str'> -> Haec
```

4.1.1 Token attributes and methods related to tokenization

Here are some atrributes/methods available for spaCy `Token` objects that are relevant to word tokenization.

SpaCy keeps track of both the token indices and the character offsets within a doc using either the `i` or `idx` attributes, respectively…

```
print(token.doc)
```

```
Haec narrantur a poetis de Perseo. Perseus filius erat Iovis, maximi deorum. Avus eius Acrisi
```

```
# token indices

for token in doc:
    print(f'{token.i}: {token.text}')
```

```
0: Haec
1: narrantur
2: a
3: poetis
4: de
5: Perseo
6: .
7: Perseus
8: filius
9: erat
10: Iovis
11: ,
12: maximi
13: deorum
14: .
15: Avus
16: eius
17: Acrisius
18: appellabatur
19: .
```

This is functionally equivalent to using `enumerate`…

```
# token indices, with enumerate

for i, token in enumerate(doc):
    print(f'{i}: {token.text}')
```

```
0: Haec
1: narrantur
2: a
3: poetis
4: de
5: Perseo
6: .
7: Perseus
8: filius
9: erat
10: Iovis
11: ,
12: maximi
13: deorum
14: .
15: Avus
16: eius
17: Acrisius
18: appellabatur
19: .
```

Another indexing option is the `idx` attribute which is the character offset of the token in the original `Doc` object.

```
# character offsets,
for token in doc:
    print(f'{token.idx}: {token.text}')
```

```
0: Haec
5: narrantur
15: a
17: poetis
24: de
27: Perseo
33: .
35: Perseus
```

```
43: filius
50: erat
55: Iovis
60: ,
62: maximi
69: deorum
75: .
77: Avus
82: eius
87: Acrisius
96: appellabatur
108: .
```

Observe these `idx` attributes relate to the character offsets from the original `Doc`. To illustrate the point, we will replace spaces with an underscore in the output. We can see from the output above that *narrantur* begins at `idx` 5 and that the next word *a* begins at `idx` 15. Yet *narrantur* is only 9 characters long and the difference between these two numbers is 10! This is because we need to account for whitespace in the original `Doc`. This is handled by the attribute `text_with_ws`.

```
print(doc.text[5:15].replace(' ', '_'))
```

```
narrantur_
```

```
print(f'text -> {doc[1].text} (length {len(doc[1].text)})')
print()
print(f'text_with_ws -> {doc[1].text_with_ws} (length {len(doc[1].text_with_ws)})')
```

```
text -> narrantur (length 9)

text_with_ws -> narrantur  (length 10)
```

Accordingly, using the `text_with_ws` attribute (as opposed to simply the `text` attribute) we can reconstruct the original text. This is what was meant above by "non-destructive" tokenization. Look at the difference between a text joined using the `text` attribute and one joined using the `text_with_ws` attribute.

```
joined_tokens = ' '.join([token.text for token in doc])
print(joined_tokens)
```

```
print(joined_tokens == doc.text)

print()

reconstructed_text = ''.join([token.text_with_ws for token in doc])
print(reconstructed_text)
print(reconstructed_text == doc.text)
```

```
Haec narrantur a poetis de Perseo . Perseus filius erat Iovis , maximi deorum . Avus eius Ac
False

Haec narrantur a poetis de Perseo. Perseus filius erat Iovis, maximi deorum. Avus eius Acris
True
```

Because spaCy tokenization is set from the outset, you can traverse the tokens in a `Doc` objects from the tokens themselves using the `nbor` method. This method takes an integer argument that specifies the number of tokens to traverse. A positive integer traverses the tokens to the right, a negative integer traverses the tokens to the left.

```
print(doc[:6])
print('-----')
print(f'{doc[3]}, i.e. i = 3')
print(f'{doc[3].nbor(-1)}, i.e. i - 1 = 2')
print(f'{doc[3].nbor(-2)}, i.e. i - 2 = 1')
print(f'{doc[3].nbor(1)}, i.e. i + 1 = 4')
print(f'{doc[3].nbor(2)}, i.e. i + 2 = 5')
```

```
Haec narrantur a poetis de Perseo
-----
poetis, i.e. i = 3
a, i.e. i - 1 = 2
narrantur, i.e. i - 2 = 1
de, i.e. i + 1 = 4
Perseo, i.e. i + 2 = 5
```

4.1.2 Customization of the spaCy tokenizer in LatinCy

LatinCy aims to tokenize the *que* enclitic in Latin texts. As noted above this is currently done through a rule-based approach. Here is the custom tokenizer code (beginning at this line in

the code) followed by a description of the process. Note that this process is based on the following recommendations in the spaCy documentation: https://spacy.io/usage/training#custom-tokenizer.

```
from spacy.util import registry, compile_suffix_regex

@registry.callbacks("customize_tokenizer")
def make_customize_tokenizer():
    def customize_tokenizer(nlp):
        suffixes = nlp.Defaults.suffixes + [
            "que",
            "qve",
        ]
        suffix_regex = compile_suffix_regex(suffixes)
        nlp.tokenizer.suffix_search = suffix_regex.search

        for item in que_exceptions:
            nlp.tokenizer.add_special_case(item, [{"ORTH": item}])
            nlp.tokenizer.add_special_case(item.lower(), [{"ORTH": item.lower()}])
            nlp.tokenizer.add_special_case(item.title(), [{"ORTH": item.title()}])
            nlp.tokenizer.add_special_case(item.upper(), [{"ORTH": item.upper()}])

    return customize_tokenizer
```

Basically, we treat *que* (and its case and u/v norm variants) as punctuation. These are added to the `Defaults.suffixes`. If no other intervention were made, then any word ending in *que* or a variant would be split into a before-*que* part and *que*. Since there are large number of relatively predictable words that end in *que*, these are maintained in a list called `que_exceptions`. All of the words in the `que_exceptions` list are added as a "special case" using the tokenizer's `add_special_case` method and so will not be split. The `que_exceptions` lists is as follows:

```
que_exceptions = ['quisque', 'quidque', 'quicque', 'quodque', 'cuiusque', 'cuique', 'quemq
```

You can see these words in the `rules` attribute of the tokenizer.

```
# Sample of 10 que rules from the custom tokenizer

tokenizer_rules = nlp.tokenizer.rules
print(sorted(list(set([rule.lower() for rule in tokenizer_rules if 'que' in rule])))[:10])
```

```
['absque', 'abusque', 'adaeque', 'adusque', 'aeque', 'antique', 'atque', 'circumundique', 'c
```

With the exception of basic enclitic splitting, the LatinCy tokenizer is the same as the default spaCy tokenizer. The default spaCy tokenizer is described in detail in the spaCy documentation. Here are some useful attributes/methods for working with LatinCy.

Tokenize a string without any other pipeline annotations with a call.

```
tokens = nlp.tokenizer(text)
print(tokens)
print(tokens[0].text)
print(tokens[0].lemma_) # Note that there is no annotation here because since the tokenize
```

```
Haec narrantur a poetis de Perseo. Perseus filius erat Iovis, maximi deorum. Avus eius Acris
Haec
```

A list of texts can be tokenized in one pass with the `pipe` method. This yields a generator object where each item is `Doc` object of tokenized-only texts

```
texts = ["Haec narrantur a poetis de Perseo. Perseus filius erat Iovis, maximi deorum; avu

tokens = list(nlp.tokenizer.pipe(texts))

print(len(tokens)) # number of documents
print(len(tokens[0])) # number of tokens in first document
```

```
3
76
```

You can get an explanation of the tokenization "decisions" using the `explain` method. In the example below, we see how the *que* in *virumque* is treated as a suffix (as discussed above) and so is split during tokenization.

```
tok_exp = nlp.tokenizer.explain('arma virumque cano')
print(tok_exp)
```

```
[('TOKEN', 'arma'), ('TOKEN', 'virum'), ('SUFFIX', 'que'), ('TOKEN', 'cano')]
```

```
tokens = nlp.tokenizer('arma uirumque cano')
for i, token in enumerate(tokens):
    print(f'{i}: {token.text}')
```

```
0: arma
1: uirum
2: que
3: cano
```

References

SLP Chapter 2, Section 2.4.2 “Word Tokenization”, pp. 18-20 link

Open LatinCy Projects

Listed below are some open projects that I would like to see implemented in the LatinCy pipelines. If you are interested in contributing to the development of these projects or have recommendations for additional projects that could be added, please contact me through the LatinCy GitHub or HuggingFace repositories.

Model-based enclitic splitting

As discussed in the Word Tokenization chapter, enclitic splitting is currently limited to *que* (and variants) and is a rules-based process that is part of the LatinCy custom tokenizer. It would be preferable to make enclitic splitting a separate pipeline component and moreover one that is model-based. This component could be placed immediately after the tokenizer and use the `Retokenizer.split` method to reconstruct token sequences where valid enclitics are identified. This would have the added advantage of allowing enclitic splitting to be "turned off", so to speak, by removing the component from the pipeline.

References

Clayman, Dee. 1981. "Sentence Length in Greek Hexameter Poetry." *Quantitative Linguistics* 11: 107–36. https://papers.ssrn.com/abstract=1627358.

Janson, Tore. 1964. "The Problems of Measuring Sentence-Length in Classical Texts." *Studia Linguistica* 18 (1): 26–36. https://doi.org/10.1111/j.1467-9582.1964.tb00443.x.

Jurafsky, Daniel, and James H. Martin. 2020. "Speech and Language Processing (3rd Edition, Draft)." https://web.stanford.edu/~jurafsky/slp3/.

Wake, William C. 1957. "Sentence-Length Distributions of Greek Authors." *Journal of the Royal Statistical Society. Series A (General)* 120 (3): 331–46. https://www.jstor.org/stable/2343104.