[Conor McDonald](https://towardsdatascience.com/@conrmcdonald?source=post_header_lockup)

Data Scientist at Deloitte, PhD, Economics (University of Leeds). R, Python and applied social/data science. @conormacd blogging at <https://dataflume.wordpress>.

Mar 17

**A short introduction to NLP in Python with spaCy**

Natural Language Processing (NLP) is one of the most interesting sub-fields of data science, and data scientists are increasingly expected to be able to whip up solutions that involve the exploitation of unstructured text data. Despite this, many applied data scientists (both from STEM and social science backgrounds) lack NLP experience.

In this post I explore some fundamental NLP concepts and show how they can be implemented using the increasingly popular [spaCy](https://spacy.io/) package in Python. This post is for the absolute NLP beginner, but knowledge of Python is assumed.

**spaCy, you say?**

spaCy is a relatively new package for “Industrial strength NLP in Python” developed by Matt Honnibal at [Explosion AI.](https://explosion.ai/) It is designed with the applied data scientist in mind, meaning it does not weigh the user down with decisions over what esoteric algorithms to use for common tasks and it’s fast. Incredibly fast (it’s implemented in Cython). If you are familiar with the Python data science stack, spaCy is your numpy for NLP – it’s reasonably low-level, but very intuitive and performant.

**So, what can it do?**

spacy provides a one-stop-shop for tasks commonly used in any NLP project, including:

* Tokenisation
* Lemmatisation
* Part-of-speech tagging
* Entity recognition
* Dependency parsing
* Sentence recognition
* Word-to-vector transformations
* Many convenience methods for cleaning and normalising text

I’ll provide a high level overview of some of these features and show how to access them using spaCy.

**Let’s get started!**

First, we load spaCy’s pipeline, which by convention is stored in a variable named nlp. declaring this variable will take a couple of seconds as spaCy loads its models and data to it up-front to save time later. In effect, this gets some heavy lifting out of the way early, so that the cost is not incurred upon each application of the nlp parser to your data. Note that here I am using the English language model, but there is also a fully featured German model, with tokenisation (discussed below) implemented across several languages.

实际上，这得到了很大的提升，所以每次将nlp解析器应用到数据上都不会产生成本。 请注意，在这里我使用的是英语语言模式，但也有一个全功能的德语模式，在多种语言中实现了令牌化（下文讨论）。

We invoke nlp on the sample text to create a Doc object. The Doc object is now a vessel for NLP tasks on the text itself, slices of the text (Span objects) and elements (Token objects) of the text. It is worth noting that Token and Span objects actually hold no data. Instead they contain pointers to data contained in the Doc object and are evaluated lazily (i.e. upon request). Much of spaCy’s core functionality is accessed through the methods on Doc(n=33), Span (n=29) and Token (n=78) objects.

我们在示例文本上调用nlp来创建一个Doc对象。 Doc对象现在是文本本身的NLP任务的容器，文本的切片（Span对象）和元素（Token对象）。 值得注意的是，Token和Span对象实际上不包含任何数据。 相反，它们包含指向包含在Doc对象中的数据的指针，并被懒惰地评估（即根据请求）。 通过Doc（n = 33），Span（29）和Token（n = 78）对象的方法访问spaCy的大部分核心功能。

In[1]: import spacy

...: nlp = spacy.load("en")

...: doc = nlp("The big grey dog ate all of the chocolate, but fortunately he wasn't sick!")

**Tokenization**

Tokenisation is a foundational step in many NLP tasks. Tokenising text is the process of splitting a piece of text into words, symbols, punctuation, spaces and other elements, thereby creating “tokens”. A naive way to do this is to simply split the string on white space:

In[2]: doc.text.split()   
...: Out[2]: ['The', 'big', 'grey', 'dog', 'ate', 'all', 'of', 'the', 'chocolate,', 'but', 'fortunately', 'he', "wasn't", 'sick!']

On the surface, this looks fine. But, note that a) it disregards the punctuation and, b) it does not split the verb and adverb (“was”, “n’t”). Put differently, it is naive, it fails to recognise elements of the text that help us (and a machine) to understand its structure and meaning. Let’s see how SpaCy handles this:

In[3]: [token.orth\_ for token in doc]   
...: Out[3]: ['The', 'big', 'grey', 'dog', 'ate', 'all', 'of', 'the', 'chocolate', ',', 'but', 'fortunately', 'he', 'was', "n't", ' ', 'sick', '!']

Here we access the each token’s .orth\_ method, which returns a string representation of the token rather than a SpaCy token object, this might not always be desirable, but worth noting. SpaCy recognises punctuation and is able to split these punctuation tokens from word tokens. Many of SpaCy’s token method offer both string and integer representations of processed text – methods with an underscore suffix return strings, methods without an underscore suffix return integers. For example:

In[4]: [(token, token.orth\_, token.orth) for token in doc]   
...: Out[4]: [  
(The, 'The', 517),   
(big, 'big', 742),   
(grey, 'grey', 4623),   
(dog, 'dog', 1175),   
(ate, 'ate', 3469),   
(all, 'all', 516),   
(of, 'of', 471),   
(the, 'the', 466),   
(chocolate, 'chocolate', 3593),   
(,, ',', 416),   
(but, 'but', 494),   
(fortunately, 'fortunately', 15520),  
 (he, 'he', 514),  
 (was, 'was', 491),  
 (n't, "n't", 479),  
 ( , ' ', 483),   
(sick, 'sick', 1698),   
(!, '!', 495)]

Here, we return the SpaCy token, the string representation of the token and the integer representation of the token in a list of tuples.

If you want to avoid returning tokens that are punctuation or white space, SpaCy provides convienence methods for this (as well as many other common text cleaning tasks — for example, to remove stop words you can call the .is\_stopmethod.

In[5]: [token.orth\_ for token in doc if not token.is\_punct | token.is\_space]   
...: Out[5]: ['The', 'big', 'grey', 'dog', 'ate', 'all', 'of', 'the', 'chocolate', 'but', 'fortunately', 'he', 'was', "n't", 'sick']

In[5]: [token.orth\_ for token in doc if not token.is\_punct | token.is\_space]   
...: Out[5]: ['The', 'big', 'grey', 'dog', 'ate', 'all', 'of', 'the', 'chocolate', 'but', 'fortunately', 'he', 'was', "n't", 'sick']

Cool, right?

**Lemmatization**

A related task to tokenisation is lemmatisation. Lemmatisation is the process of reducing a word to its base form, its mother word if you like. Different uses of a word often have the same root meaning. For example, practice, practised and practising all essentially refer to the same thing. It is often desirable to standardise words with similar meaning to their base form. With SpaCy we can access each word’s base form with a token’s .lemma\_ method:

In[6]: practice = "practice practiced practicing"   
...: nlp\_practice = nlp(practice)   
...: [word.lemma\_ for word in nlp\_practice]   
...: Out[6]: ['practice', 'practice', 'practice']

Why is this useful? An immediate use case is in machine learning, specifically text classification. Lemmatising the text prior to, for example, creating a “bag-of-words” avoids word duplication and, therefore, allows for the model to build a clearer picture of patterns of word usage across multiple documents.

**POS Tagging**

Part-of-speech tagging is the process of assigning grammatical properties (e.g. noun, verb, adverb, adjective etc.) to words. Words that share the same POS tag tend to follow a similar syntactic structure and are useful in rule-based processes.

For example, in a given description of an event we may wish to determine who owns what. By exploiting possessives, we can do this (providing the text is grammatically sound!). SpaCy uses the popular Penn Treebank POS tags, see <https://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html>. With SpaCy you can access coarse and fine-grained POS tags with the .pos\_ and .tag\_ methods, respectively. Here, I access the fine grained POS tag:

In[7]: doc2 = nlp("Conor's dog's toy was hidden under the man's sofa in the woman's house")   
...:   
pos\_tags = [(i, i.tag\_) for i in doc2] ...:   
pos\_tags   
...: Out[7]: [(Conor, 'NNP'), ('s, 'POS'), (dog, 'NN'), ('s, 'POS'), (toy, 'NN'), (was, 'VBD'), (hidden, 'VBN'), (under, 'IN'), (the, 'DT'), (man, 'NN'), ('s, 'POS'), (sofa, 'NN'), (in, 'IN'), (the, 'DT'), (woman, 'NN'), ('s, 'POS'), (house, 'NN')]

We can see that the “ ’s ” tokens are labelled as POS. We can exploit this tag to extract the owner and the thing that they own:

In[8]: owners\_possessions = []   
...: for i in pos\_tags:   
 ...: if i[1] == "POS": ...: owner = i[0].nbor(-1)   
...: possession = i[0].nbor(1)   
...: owners\_possessions.append((owner, possession)) ...: ...: owners\_possessions   
...: Out[8]: [(Conor, dog), (dog, toy), (man, sofa), (woman, house)]

This returns a list of owner-possession tuples. If you want to be super Pythonic about it, you can do this in a list comprehenion (which, I think is preferable!):

In[9]: [(i[0].nbor(-1), i[0].nbor(+1)) for i in pos\_tags if i[1] == "POS"]   
...: Out[9]: [(Conor, dog), (dog, toy), (man, sofa), (woman, house)]

Here we are using each token’s .nbor method which returns a token’s neighbouring tokens.

**Entity recognition**

Entity recognition is the process of classifying named entities found in a text into pre-defined categories, such as persons, places, organizations, dates, etc. spaCy uses a statistical model to classify a broad range of entities, including persons, events, works-of-art and nationalities / religions (see the documentation for the full list [https://spacy.io/docs/usage/entity-recognition).](https://spacy.io/docs/usage/entity-recognition%29.)

For example, let’s take the first two sentences from Barack Obama’s wikipedia entry. We will parse this text, then access the identified entities using the Doc object’s .ents method. With this method called on the Docwe can access additional Token methods, specifically .label\_ and .label:

In[10]: wiki\_obama = """Barack Obama is an American politician who served as ...: the 44th President of the United States from 2009 to 2017. He is the first ...: African American to have served as president, ...: as well as the first born outside the contiguous United States."""   
...:   
...: nlp\_obama = nlp(wiki\_obama) ...: [(i, i.label\_, i.label) for i in nlp\_obama.ents]   
...: Out[10]: [(Barack Obama, 'PERSON', 346), (American, 'NORP', 347), (the United States, 'GPE', 350), (2009 to 2017, 'DATE', 356), (first, 'ORDINAL', 361), (African, 'NORP', 347), (American, 'NORP', 347), (first, 'ORDINAL', 361), (United States, 'GPE', 350)]

You can see the entities that the model has identified and how accurate they are (in this instance). PERSON is self explanatory, NORP is natianalities or religuos groups, GPE identifies locations (cities, countries, etc.), DATE recognises a specific date or date-range and ORDINAL identifies a word or number representing some type of order.

While we are on the topic of Doc methods, it is worth mentioning spaCy’s sentence identifier. It is not uncommon in NLP tasks to want to split a document into sentences. It is simple to do this with SpaCy by accessing a Doc's .sents method:

In[11]: for ix, sent in enumerate(nlp\_obama.sents, 1):   
...: print("Sentence number {}: {}".format(ix, sent))   
...:   
Sentence number 1: Barack Obama is an American politician who served as the 44th President of the United States from 2009 to 2017. Sentence number 2: He is the first African American to have served as president, as well as the first born outside the contiguous United States.