Analyzing and Processing Text With `spaCy`

`spaCy` is an open-source natural language processing library for Python. It is designed particularly for production use, and it can help us to build applications that process massive volumes of text efficiently. First, let's take a look at some of the basic analytical tasks `spaCy` can handle.

Installing `spaCy`

We can also use `spaCy` in a Juypter Notebook. It's not one of the pre-installed libraries that Jupyter includes by default, though, so we'll need to run these commands from the notebook to get `spaCy` installed in the correct Anaconda directory. Note that we use `!` in front of each command to let the Jupyter notebook know that it should be read as a command line command.

```
!pip install spacy

!python -m spacy download en
```

Tokenizing the Text

Tokenization is the process of breaking text into pieces, called tokens, and ignoring characters like punctuation marks (,. " ') and spaces. `spaCy`'s tokenizer takes input in form of unicode text and outputs a sequence of token objects.

Let's take a look at a simple example. Imagine we have the following text, and we'd like to tokenize it:

```
When learning data science, you shouldn't get discouraged.

Challenges and setbacks aren't failures, they're just part of the
journey.
```

There are a couple of different ways we can appoach this. The first is called **word tokenization**, which means breaking up the text into individual words. This is a critical step for many language processing applications, as they often require input in the form of individual words rather than longer strings of text.

In the code below, we'll import `spaCy` and its English-language model, and tell it that we'll be doing our natural language processing using that model. Then we'll assign our text string to `text`. Using `nlp(text)`, we'll process that text in `spaCy` and assign the result to a variable called `my_doc`.

At this point, our text has already been tokenized, but `spaCy` stores tokenized text as a doc, and we'd like to look at it in list form, so we'll create a `for` loop that iterates through our doc, adding each word token it finds in our text string to a list called `token_list` so that we can take a better look at how words are tokenized.

```
# Word tokenization
from spacy.lang.en import English

# Load English tokenizer, tagger, parser, NER and word vectors
nlp = English()

text = """When learning data science, you shouldn't get discouraged!
Challenges and setbacks aren't failures, they're just part of the journey.
You've got this!"""

#  "nlp" Object is used to create documents with linguistic annotations.
my_doc = nlp(text)

# Create list of word tokens
token_list = []
for token in my_doc:
    token_list.append(token.text)
print(token_list)
['When', 'learning', 'data', 'science', ',', 'you', 'should', "n't", 'get',
'discouraged', '!', '\n', 'Challenges', 'and', 'setbacks', 'are', "n't",
'failures', ',', 'they', "'re", 'just', 'part', 'of', 'the', 'journey', '.',
'You', "'ve", 'got', 'this', '!']
```

As we can see, `spaCy` produces a list that contains each token as a separate item. Notice that it has recognized that contractions such as *shouldn't* actually represent two distinct words, and it has thus broken them down into two distinct tokens.

Fist we need to load language dictionaries, Here in abve example, we are loading english dictionary using English() class and creating nlp nlp object. "nlp" object is used to create documents with linguistic annotations and various nlp properties. After creating document, we are creating a token list.

If we want, we can also break the text into sentences rather than words. This is called **sentence tokenization**. When performing sentence tokenization, the tokenizer looks for specific characters that fall between sentences, like periods, exclaimation points, and newline characters. For sentence tokenization, we will use a preprocessing pipeline because sentence preprocessing using `spaCy` includes a tokenizer, a tagger, a parser and an entity recognizer that we need to access to correctly identify what's a sentence and what isn't.

In the code below,`spaCy` tokenizes the text and creates a Doc object. This Doc object uses our preprocessing pipeline's components tagger,parser and entity recognizer to break the text down into components. From this pipeline we can extract any component, but here we're going to access sentence tokens using the `sentencizer` component.

```
# sentence tokenization

# Load English tokenizer, tagger, parser, NER and word vectors
nlp = English()

# Create the pipeline 'sentencizer' component
sbd = nlp.create_pipe('sentencizer')
```

```
# Add the component to the pipeline
nlp.add_pipe(sbd)

text = """When learning data science, you shouldn't get discouraged!
Challenges and setbacks aren't failures, they're just part of the journey.
You've got this!"""

#  "nlp" Object is used to create documents with linguistic annotations.
doc = nlp(text)

# create list of sentence tokens
sents_list = []
for sent in doc.sents:
    sents_list.append(sent.text)
print(sents_list)
["When learning data science, you shouldn't get discouraged!", "\nChallenges
and setbacks aren't failures, they're just part of the journey.", "You've got
this!"]
```

Again, `spaCy` has correctly parsed the text into the format we want, this time outputting a list of sentences found in our source text.

Cleaning Text Data: Removing Stopwords

Most text data that we work with is going to contain a lot of words that aren't actually useful to us. These words, called **stopwords**, are useful in human speech, but they don't have much to contribute to data analysis. Removing stopwords helps us eliminate noise and distraction from our text data, and also speeds up the time analysis takes (since there are fewer words to process).

Let's take a look at the stopwords `spaCy` includes by default. We'll import `spaCy` and assign the stopwords in its English-language model to a variable called `spacy_stopwords` so that we can take a look.

```
#Stop words
#importing stop words from English language.
import spacy
spacy_stopwords = spacy.lang.en.stop_words.STOP_WORDS

#Printing the total number of stop words:
print('Number of stop words: %d' % len(spacy_stopwords))

#Printing first ten stop words:
print('First ten stop words: %s' % list(spacy_stopwords)[:20])
Number of stop words: 312
First ten stop words: ['was', 'various', 'fifty', "'s", 'used', 'once',
'because', 'himself', 'can', 'name', 'many', 'seems', 'others', 'something',
'anyhow', 'nowhere', 'serious', 'forty', 'he', 'now']
```

As we can see, `spaCy`'s default list of stopwords includes 312 total entries, and each entry is a single word. We can also see why many of these words wouldn't be useful for data analysis. Transition words like *nevertheless*, for example, aren't necessary for understanding the basic

meaning of a sentence. And other words like *somebody* are too vague to be of much use for NLP tasks.

If we wanted to, we could also create our own customized list of stopwords. But for our purposes in this tutorial, the default list that `spaCy` provides will be fine.

Removing Stopwords from Our Data

Now that we've got our list of stopwords, let's use it to remove the stopwords from the text string we were working on in the previous section. Our text is already stored in the variable `text`, so we don't need to define that again.

Instead, we'll create an empty list called `filtered_sent` and then iterate through our `doc` variable to look at each tokenized word from our source text. `spaCy` includes a bunch of helpful token attributes, and we'll use one of them called `is_stop` to identify words that *aren't* in the stopword list and then append them to our `filtered_sent` list.

```
from spacy.lang.en.stop_words import STOP_WORDS

#Implementation of stop words:
filtered_sent=[]

#  "nlp" Object is used to create documents with linguistic annotations.
doc = nlp(text)

# filtering stop words
for word in doc:
    if word.is_stop==False:
        filtered_sent.append(word)
print("Filtered Sentence:",filtered_sent)
Filtered Sentence: [learning, data, science, ,, discouraged, !,
, Challenges, setbacks, failures, ,, journey, ., got, !]
```

It's not too difficult to see why stopwords can be helpful. Removing them has boiled our original text down to just a few words that give us a good idea of what the sentences are discussing: learning data science, and discouraging challenges and setbacks along that journey.

Lexicon Normalization

Lexicon normalization is another step in the text data cleaning process. In the big picture, normalization converts high dimensional features into low dimensional features which are appropriate for any machine learning model. For our purposes here, we're only going to look at **lemmatization**, a way of processing words that reduces them to their roots.

Lemmatization

Lemmatization is a way of dealing with the fact that while words like *connect*, *connection*, *connecting*, *connected*, etc. aren't exactly the same, they all have the same essential meaning:

connect. The differences in spelling have grammatical functions in spoken language, but for machine processing, those differences can be confusing, so we need a way to change all the words that are *forms* of the word *connect* into the word *connect* itself.

One method for doing this is called **stemming**. Stemming involves simply lopping off easily-identified prefixes and suffixes to produce what's often the simplest version of a word. *Connection*, for example, would have the *-ion* suffix removed and be correctly reduced to *connect*. This kind of simple stemming is often all that's needed, but lemmatization—which actually looks at words and their roots (called *lemma*) as described in the dictionary—is more precise (as long as the words exist in the dictionary).

Since `spaCy` includes a build-in way to break a word down into its *lemma*, we can simply use that for lemmatization. In the following very simple example, we'll use `.lemma_` to produce the lemma for each word we're analyzing.

```
# Implementing lemmatization
lem = nlp("run runs running runner")
# finding lemma for each word
for word in lem:
    print(word.text,word.lemma_)
run run
runs run
running run
runner runner
```

Part of Speech (POS) Tagging

A word's **part of speech** defines its function within a sentence. A noun, for example, identifies an object. An adjective describes an object. A verb describes action. Identifying and tagging each word's part of speech in the context of a sentence is called Part-of-Speech Tagging, or POS Tagging.

Let's try some POS tagging with `spaCy`! We'll need to import its `en_core_web_sm` model, because that contains the dictionary and grammatical information required to do this analysis. Then all we need to do is load this model with `.load()` and loop through our new `docs` variable, identifying the part of speech for each word using `.pos_`.

(Note the *u* in `u"All is well that ends well."` signifies that the string is a Unicode string.)

```
# POS tagging

# importing the model en_core_web_sm of English for vocabluary, syntax &
entities
import en_core_web_sm

# load en_core_web_sm of English for vocabluary, syntax & entities
nlp = en_core_web_sm.load()

#  "nlp" Objectis used to create documents with linguistic annotations.
```

```
docs = nlp(u"All is well that ends well.")

for word in docs:
    print(word.text,word.pos_)
All DET
is VERB
well ADV
that DET
ends VERB
well ADV
. PUNCT
```

Hooray! `spaCy` has correctly identified the part of speech for each word in this sentence. Being able to identify parts of speech is useful in a variety of NLP-related contexts, because it helps more accurately understand input sentences and more accurately construct output responses.

Entity Detection

Entity detection, also called entity recognition, is a more advanced form of language processing that identifies important elements like places, people, organizations, and languages within an input string of text. This is really helpful for quickly extracting information from text, since you can quickly pick out important topics or indentify key sections of text.

Let's try out some entity detection using a few paragraphs from this recent article in the Washington Post. We'll use `.label` to grab a label for each entity that's detected in the text, and then we'll take a look at these entities in a more visual format using `spaCy`'s `displaCy` visualizer.

```
#for visualization of Entity detection importing displacy from spacy:

from spacy import displacy

nytimes= nlp(u"""New York City on Tuesday declared a public health emergency
and ordered mandatory measles vaccinations amid an outbreak, becoming the
latest national flash point over refusals to inoculate against dangerous
diseases.

At least 285 people have contracted measles in the city since September,
mostly in Brooklyn's Williamsburg neighborhood. The order covers four Zip
codes there, Mayor Bill de Blasio (D) said Tuesday.

The mandate orders all unvaccinated people in the area, including a
concentration of Orthodox Jews, to receive inoculations, including for
children as young as 6 months old. Anyone who resists could be fined up to
$1,000.""")

entities=[(i, i.label_, i.label) for i in nytimes.ents]
entities
[(New York City, 'GPE', 384),
 (Tuesday, 'DATE', 391),
 (At least 285, 'CARDINAL', 397),
 (September, 'DATE', 391),
```

```
(Brooklyn, 'GPE', 384),
(Williamsburg, 'GPE', 384),
(four, 'CARDINAL', 397),
(Bill de Blasio, 'PERSON', 380),
(Tuesday, 'DATE', 391),
(Orthodox Jews, 'NORP', 381),
(6 months old, 'DATE', 391),
(up to $1,000, 'MONEY', 394)]
```

Using this technique, we can identify a variety of entities within the text. The `spaCy` documentation provides a full list of supported entity types, and we can see from the short example above that it's able to identify a variety of different entity types, including specific locations (`GPE`), date-related words (`DATE`), important numbers (`CARDINAL`), specific individuals (`PERSON`), etc.

Using `displaCy` we can also visualize our input text, with each identified entity highlighted by color and labeled. We'll use `style = "ent"` to tell `displaCy` that we want to visualize entities here.

```
displacy.render(nytimes, style = "ent",jupyter = True)
```

New York City GPE on Tuesday DATE declared a public health emergency and ordered mandatory measles vaccinations amid an outbreak, becoming the latest national flash point over refusals to inoculate against dangerous diseases. At least 285 CARDINAL people have contracted measles in the city since September DATE , mostly in Brooklyn GPE 's Williamsburg GPE neighborhood. The order covers four CARDINAL Zip codes there, Mayor Bill de Blasio PERSON (D) said Tuesday DATE .The mandate orders all unvaccinated people in the area, including a concentration of Orthodox Jews NORP , to receive inoculations, including for children as young as 6 months old DATE . Anyone who resists could be fined up to $1,000 MONEY .

Dependency Parsing

Depenency parsing is a language processing technique that allows us to better determine the meaning of a sentence by analyzing how it's constructed to determine how the individual words relate to each other.

Consider, for example, the sentence "Bill throws the ball." We have two nouns (Bill and ball) and one verb (throws). But we can't just look at these words individually, or we may end up

thinking that the ball is throwing Bill! To understand the sentence correctly, we need to look at the word order and sentence structure, not just the words and their parts of speech.

Doing this is quite complicated, but thankfully `spaCy` will take care of the work for us! Below, let's give `spaCy` another short sentence pulled from the news headlines. Then we'll use another `spaCy` called `noun_chunks`, which breaks the input down into nouns and the words describing them, and iterate through each chunk in our source text, identifying the word, its root, its dependency identification, and which chunk it belongs to.

```
docp = nlp (" In pursuit of a wall, President Trump ran into one.")

for chunk in docp.noun_chunks:
   print(chunk.text, chunk.root.text, chunk.root.dep_,
         chunk.root.head.text)
pursuit pursuit pobj In
a wall wall pobj of
President Trump Trump nsubj ran
```

This output can be a little bit difficult to follow, but since we've already imported the `displaCy` visualizer, we can use that to view a dependency diagraram using `style = "dep"` that's much easier to understand:

```
displacy.render(docp, style="dep", jupyter= True)
```