COMP 10261: Extracting Linguistic Knowledge

Sam Scott, Mohawk College, 2021.

WHAT IS LINGUISTIC KNOWLEDGE?

Linguistic knowledge refers to knowledge that speakers of a natural language have about the **phonology** (sounds), **syntax** (grammar), **morphology** (construction of words), **semantics** (meaning), and **pragmatics** (non-literal use) of their language. This knowledge is often inaccessible to the speaker. We all know how to put together and take apart sentences in our native language without really being able to explain how we do it. The scientific study of language, **linguistics**, has yet to fully explain the nature of the knowledge that a person has when they know a language. We do understand a lot about how natural languages work, but there is much we do not understand, and this lack of understanding is a major barrier both to building more effective NLP and to creating Strong AI. This handout takes a quick look at what linguistic knowledge we can reliably extract from a text, and what we can do with it.

For the purposes of NLP, linguistic knowledge involves **tokenizing** a text and then **tagging** it to identify **parts of speech** (nouns, verbs, etc.), **key phrases**, **named entities**, **relationships** between words, and word roots (e.g., "drinks", "drinking", and "drunk" are all different forms of the root word "drink"). Linguistic knowledge of this kind can be important for **Information Extraction (IE)**, for **question answering**, and for processing utterances in a deeper way. We will use a **rule-based** approach that is built on top of both **statistical** and rule-based knowledge extraction systems.

USING SPACY FOR LINGUISTIC KNOWLEDGE

SpaCy (https://spacy.io/) is a Python implementation of a **generic NLP pipeline** with an easy-to-use API for extracting linguistic knowledge. It contains pre-trained language models for tagging **Parts of Speech**, **Lemmas** (word roots), **Named Entities**, and syntactic **relationships**. It allows you to create powerful **Pattern Matching** rules based on these tags, and it can be customized and retrained to fit specific text domains and to add components to the default pipeline.[1]

INSTALL THE SPACY MODULE

SpaCy is not included in Anaconda Python, but it can easily be installed from the Python shell using the command below. Be patient, it can take a while!

```
conda install spacy
```

INSTALL A LANGUAGE MODEL

You will also need to download and install pre-trained language models for English (or for any other language you're using). This is done from your operating system command line, and it requires that you first add Anaconda's **\Library\bin** and **\Scripts** folders to your **PATH** environment variable.

[1] We will use the default components. You can read about custom components at https://spacy.io.

To modify the PATH variable on Windows 10…

1. Open the **Control Panel**, go to **System and Security**, then **System**, then **Advanced System Settings** (or use the search bar to find "**Edit the System Environment Variables**")
2. Press the **Environment Variables** button, select the **Path** variable, and press **Edit…**
3. Press **New** and add the Anaconda paths **…\Library\bin** and **…\Scripts**, customizing them to match your installation details. For example, on my machine, the correct paths are F:\Anaconda3\Library\bin and F:\Anaconda3\Scripts.
4. To check that it worked, open a fresh command line window and type **echo %PATH%**. You should see the new paths you added.

Once this is done, open a command line window on your Anaconda folder (the one that contains **python.exe**) and invoke the command shown below.

```
python -m spacy download en_core_web_sm
```

This downloads the "small" version of the core English language model. Once this is finished, you can start using spaCy, but you should get the medium, large, and transformer models as well. They take up more space and require more processing time, but they also contain more linguistic knowledge and can sometimes make more accurate predictions.

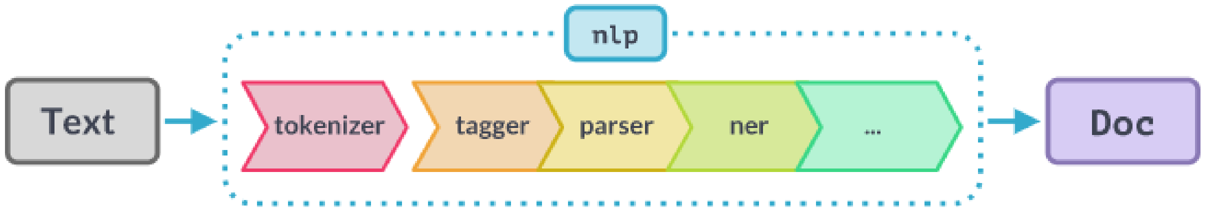
```
python -m spacy download en_core_web_md
python -m spacy download en_core_web_lg
python -m spacy download en_core_web_trf
```

THE SPACY PIPELINE

Once **spaCy** and **en_core_web_sm** are installed, try the following in the Python shell or a script file:

```
import spacy
nlp = spacy.load("en_core_web_sm")
```

This creates an **English** pipeline object called **nlp** based on the **en_core_web_sm** language model. Here is a diagram of the pipeline you get, copied from https://spacy.io.



A **string** containing text goes in one end of the pipeline. The pipeline components separate the string into tokens and then annotate the tokens with various kinds of useful information. Out the other end of the pipeline comes a spaCy **Doc** object containing the original text plus all the information added by the various pipeline components.

Here is an example of how to run a **string** through the spaCy pipeline to get a **Doc** object:

```
doc = nlp("Twitter says the panda umbrella company is trading at
$4.00 per share today? That’s really low!")
```

If you **print** the **doc** object, the output looks just like the original string. But the string has been **tokenized** into separate words and sentences, the tokens have been **tagged** and **parsed**, a **Named Entity Recognition (NER)** module has labelled some important **entities** and a **lemmatizer** (not shown in the pipeline diagram above) has identified the **lemma** (base form) for each word.

TOKENS

WHAT ARE TOKENS?

Tokens are the words, symbols, numerals, punctuation marks, and other important linguistic elements found in a text. A **Doc** object is basically a list of tokens from the original sentence. You can use the **len** function to find out how many tokens it contains, and you can iterate over it just like a list:

```
doc = nlp("Twitter says the panda umbrella company is trading at
$4.00 per share today? That’s really low!")

print("The sixth token is", doc[5])

for token in doc:
     print(token)
```

If you run the above code, you will notice that not all the tokens in the output are words. In the above example, the token list includes non-words like *'s*, *$*, and *?*.

USING TOKENS

Tokens represent the most basic kind of linguistic knowledge – the foundation upon which spaCy extracts more sophisticated knowledge.[2] Once the tokens are identified, spaCy can tag them, perform transformations on them, and look for patterns to extract.

In spaCy, tokens are instances of the **Token** class. Each **Token** instance contains the original text and index number from the document plus annotations that were added as the **Doc** was passed from component to component in the pipeline. The example below demonstrates how to extract the text and index number for each token in a **Doc**.

```
for token in doc:
     print(token.i, token.text)
```

[2] SpaCy tokens include symbols from written language that are not words and don’t really count as “linguistic knowledge” in the strict sense. Apologies to any linguists who might be grinding their teeth while reading this.

You can also get a lowercase version of a token with `token.lower_` or you can find out if it's lowercase with `token.is_lower`, find out if it's purely alphabetic with `token.is_alpha`, and so on. For the full list of Token attributes, see https://spacy.io/api/token#attributes.

Doc objects also support the Python **slice** operator (:). When you slice a **Doc** object you get a spaCy **Span** object that holds the **Token** objects you sliced out from the original **Doc** – example below.

```
>>> span = doc[2:6]
>>> print(span.text)
the panda umbrella company
```

Under the Hood

Tokens are identified in a **rule-based** way, often with the aid of **regular expressions** that specify sequences of **delimiter** (separator) characters.

For example, you could use the **split** function in the **re** or **regex** module to identify tokens, instructing it to split on any sequence of one or more whitespace characters, using **r"\s+"** as the delimiter expression.

```
import regex as re
re.split(r"\s+",doc.text)
```

The result is not quite as nice as what spaCy does for us. For one thing, it leaves punctuation attached to words. Can you make it better?

Parts of Speech

What are Parts of Speech?

Parts of speech (POS) are categories of words like noun, verb, adjective, and adverb. A word's POS tells you something about the role it is playing in the sentence. Here is a rough guide to some of the most important parts of speech for English.

Part of Speech	Explanation	Examples
Noun	A word for a "thing" or a category of "things".	**dog**, **number**, **love**, **unicorn**, **Sam**
Verb	A word for an "action" or a "state of being".	**run**, **ran**, **read**, **stops**, **went**
Auxiliary	A "helper verb" that expresses the tense or mood of another verb.	I **will** run, I **am** driving, I **have** voted
Adjective	A word that modifies a noun.	**green** eggs, **silly** rabbit, **huge** teeth, **invisible** band aid
Adverb	A word that modifies a verb, adjective, or adverb.	run **slowly**, speak **softly**, **very** big shoes, walk **too quickly**
Determiner	A word that goes in front of a noun to specify quantity or level of abstractness.	**the** dog, **a** number, **that** guy, **those** birds
Preposition	A word that goes in front of a noun or noun phrase to show direction, time, place, location, etc.	**at** noon, **on** Thursday, **in** September, **under** the bridge, **over** my objections, **beyond** a shadow **of** doubt

Here is an example sentence showing the parts of speech as determined by spaCy. Note that **ADP** stands for "**adposition**" which is what spaCy calls prepositions.

```
The cat  on  the rug  quickly ate  a   big bowl of  cheerios .
DET NOUN ADP DET NOUN ADV     VERB DET ADJ NOUN ADP NOUN     PUNCT
```

Sometimes it is useful to identify **phrases** based on parts of speech. For example, a **noun phrase** is a noun with some other words around it that qualify or provide more information about it in some way. Here are some phrases from the sentence above.

```
the cat on the rug                     ← noun phrase
on the rug                             ← prepositional phrase
quickly ate a big bowl of cheerios     ← verb phrase
a big bowl of cheerios                 ← noun phrase
```

Every **Token** object in spaCy is labeled with a **pos_** tag that comes from the **Tagger** pipeline component. Here is an example of how to get access to those tags. Before you run the code, see if you can predict the **pos_** tag for each word. How many did you get right?

```
doc = nlp("Twitter says the panda umbrella company is trading at
$4.00 per share today? That's really low!")

for token in doc:
     print(token.text, token.pos_)
```

If you are not sure what a tag means, you can use the **explain** function to get a bit more info.

```
>>> spacy.explain("PROPN")
'proper noun'
```

Using Parts of Speech

Once you have identified the parts of speech in a document, you can look for patterns that might indicate a named entity or important phrase of some kind. You can also use the parts of speech to identify **syntactic dependencies** between words as well as phrases involving multiple words. This is what the spaCy **Parser** does.

The Parser provides a rich level of syntactic information that is mostly beyond the scope of what we are doing here. However, one result of parsing is to tag the **Doc** object with simple **noun phrases** from the text. These noun phrases are useful because they often correspond to important **entities**. **SpaCy** calls them **Noun Chunks**. They are available as a **generator** field in the **Doc** object called **noun_chunks**. **Generators** can be used in **for** loops or converted to a list using the **list** function.

```
doc = nlp("Twitter says the panda umbrella company is trading at
$4.00 per share today? That's really low!")

for np in doc.noun_chunks:
     print(np.text, np.start, np.end, np.label_)

print( list(doc.noun_chunks) )
```

Each object in **noun_chunks** is a spaCy **Span** object. It has a **text** field, a **start** and **end** field that specifies the location of this **Span's Tokens** in the original **Doc** object, and a **label_** field that is usually set to "NP" (Noun Phrase).

Under the Hood

Part of Speech tagging is not as simple as you might think, because words can belong to different parts of speech depending on context. For example, you can go for a drive (noun) or you can drive (verb) a car. POS taggers must use the context of each word to figure out the correct part of speech.

The first taggers used a **rule-based** approach to assigning parts of speech based on surrounding context. Modern taggers are more likely to use a **statistical** ("stochastic") approach that requires a large **training corpus** (set of documents) that has already been correctly tagged by human beings. These taggers use Artificial Neural Networks, Machine Learning, probabilities, and other statistical techniques to form a model that can predict the tags of words in sentences they have not seen before. The spaCy tagger is a Deep Learning (Artificial Neural Network) system based on the THiNC library (https://thinc.ai/)

Both approaches to tagging can work well, but no matter which approach is used, no tagger is 100% accurate.

Named Entities

What are Named Entities?

Named entities are noun phrases that signify something of a predefined type: a person, an organization, a geopolitical entity (like a country, province, or city), a time or date, a monetary value, etc.

The **Named Entity Recognizer (NER)** module in the spaCy pipeline adds any named entities it finds to the **ents** field of the **Doc** object (a Python **tuple**). As with **noun_chunks**, each item in the **ents** tuple is a **Span** object with a **label_**. In this case, the **label_** fields indicate the type of each Named Entity.

```
doc = nlp("Twitter says the panda umbrella company is trading at
$4.00 per share today? That's really low!")

for ent in doc.ents:
     print(ent.text, ent.label_)
```

If you are using the **en_core_web_sm** language model in version 3.0 of spaCy, the above code will show two entities with different labels:[3]

```
4.00 MONEY
today DATE
```

It's a shame that this language model fails to identify "Twitter" which is quite clearly an important named entity in this utterance. (Perhaps the difficulty is the way we're using the word "Twitter"? Try

[3] Results may differ for other versions. Use **help(spacy)** to find out what version you're running. Try this code on your version – what do the results look like for you?

using Twitter in different sentences to see when it gets flagged as a named entity and when it does not.) If you use other language models, the results will change. For example, **en_core_web_trf** finds the following entities:

```
Twitter ORG
4.00 MONEY
today DATE
```

Remember that you can use the **spacy.explain** function in the Python shell to get more information on what any label means.

```
>>> spacy.explain("ORG")
'Companies, agencies, institutions, etc.'
```
[4]

USING NAMED ENTITIES

News organizations and other content providers can use Named Entity Recognition to tag documents and link together documents that mention the same entities. These tags can then be used for filtering document lists or for recommending similar documents to a user. Named Entity Recognition can also help categorize customer feedback that comes in the form of product reviews or social media posts, and it can be used by a chat bot to help figure out how to respond to an utterance. For example, if the user says, “I need to find an LCBO near the Fennel Campus”, the chat bot could recognize that LCBO and Fennel Campus are named entities that are both locations and could send a query to a map system to help answer the user’s question.

UNDER THE HOOD

The simplest way to identify named entities is to maintain a **gazetteer** – a list of names stored in an efficient data structure like a hash table. But this won’t work for named entities that are not known in advance. For unknown entities and for more difficult or variably named entities like times, dates, money, etc., both **rule-based** and **statistical Machine Learning** approaches achieve good results – almost as good as human beings. The spaCy **NER** module uses a kind of **Artificial Neural Network** called a **Convolution Network** using **word embeddings** – we will look at word embeddings later in the course. The NER component was trained using a large corpus of text with the named entities pre-labelled by human volunteers, but it is also able to identify likely named entities that it has never seen before.

Even though the spaCy **NER** component is highly accurate, we still should not expect 100% accuracy from it. It will occasionally make mistakes or miss something, even when using the best language model. In the example from the last section, “the panda umbrella company” should probably have been identified as a named entity, since the author of the text clearly believes it is a real company. But since this company did not exist in the training data, it’s hard to recognize.

[4] Hmm… It’s good that this language model identified Twitter as a named entity, but does the original utterance really refer to Twitter the company?

LEMMAS

WHAT ARE LEMMAS?

A **lemma** is a root form that links together several words. For example, “drive”, “drove”, “driving”, “drives”, and “driven” are all different forms of the lemma “drive” and have closely related meanings. Linguists call this phenomenon **morphology** and would say that each form of the word “drive” has been **inflected** in a different way. Inflections in English can show verb tense or plurality.

The spaCy lemmatizer adds a **lemma_** tag to each token that shows the root form of the word:

```
for token in doc:
     print(token.text, token.lemma_)
```

USING LEMMAS

Lemmas allow you to get to a word’s root meaning. For example, if you are looking to extract a list of different kinds of vehicles that are mentioned in a document, it might be helpful to look for a rule-based pattern involving the lemma “drive” followed by a noun phrase (e.g., “Sam **drives a car**.”, “My uncle **drove trains**.”, “She had **driven ATVs** her entire life.”). Lemmas can also be helpful in representing text for statistical NLP tasks such as document classification – we will look at this later in the course.

The usefulness of lemmas may depend on the language you’re processing. English and the other Germanic languages are moderately inflected – they mostly inflect nouns and verbs to show number and tense information. Romance languages like French also inflect adjectives and inflect for gender along with number and tense. Lemmatization has often proved useful for languages such as these.

The most inflected languages are known as “polysynthetic” languages. Iroquoian languages such as Mohawk fall into this category. These languages are so highly inflected that a single word can carry enough meaning to express something that would require an entire sentence in English. Lemmatization might help here, but polysynthetic languages probably require a much more sophisticated approach to morphology and tokenization (including the use of what linguists call **derivational morphology** in addition to inflectional morphology). On the other end of the morphology spectrum, Mainland Southeast Asian languages like Chinese, Vietnamese, and Thai have almost no inflection on individual words, so we should not expect lemmas to help us much at all for these languages.[5]

UNDER THE HOOD: IDENTIFYING LEMMAS

The best approaches for identifying lemmas are **rule-based**. The spaCy lemmatizer uses a simple (very large) hash table for lookup alongside rules for words that are not in the table. For example, if a word is tagged as a verb and ends in “ed” but it does not appear in the hash table, then the lemmatizer can apply the general English rule for making a verb past tense and remove the “ed” or the “d” to take a reasonable guess at the correct Lemma.

[5] I have no direct knowledge of the languages I’m discussing here. If you speak one of these languages (or any other that seems interesting for this discussion) please feel free to offer corrections, context, or examples. I’ll incorporate your feedback into a future version of the document and credit you for the contribution.

EXERCISES

In these exercises, you will use a large text document to explore the different English language models.

1. Find a large plain text document (or use **manifesto.txt** from Canvas). You can read it into a single string and run it through the spaCy pipeline with the following:

```
import spacy
nlp = spacy.load("en_core_web_sm")
print("language model loaded")

text = ""
with open("manifesto.txt", encoding="utf-8") as file:
    for line in file:
        text += line;
print("text loaded")

doc = nlp(text)
print("pipeline finished")
```

2. Python contains a **set** data type that implements a simple hash set. A set is like a dictionary that only stores keys. If you add the same key twice, you only get one copy of the key in the set. Here is a simple example showing how a set works:

```
s = set()
s.add("hello")
s.add("world")
s.add("hello")

if "hello" in s:
    print("found 'hello'")
print( s )
print( len(s) )
```

   Use a set to count and report the total number of unique tokens in the document as well as the total number of different lemmas.

3. Use a Python dictionary to count and report the number of instances of each part of speech tag in the document.

4. Identify and report the longest noun chunk and named entity in the document. You should also report the entity type.

5. Now change the language model. Try the small, medium, large, and transformer models. Do you notice any differences in the results of exercises 1 through 4?