COMP 10261: Linguistic Pattern Matching

Sam Scott, Mohawk College, 2021.

A Recap of spaCy's Linguistic Knowledge

The spaCy NLP pipeline converts a string of text to a **Doc** object. As the text goes through the pipeline it is split into **Token** objects and each Token is tagged with a **part of speech**, a **lemma**, and lots of other useful information (see https://spacy.io/api/token#attributes). The **Doc** object also gets tagged with information, including **Span** objects for **named entities** and **noun phrases**.

Now that you have all the linguistic information from a tokenized and tagged document, you can search, classify, and extract information from it by defining **pattern matching rules** using a **regular grammar**.

Pattern Matching with spaCy

Suppose you wanted to extract noun phrases that indicate locations from a user's utterance, maybe to embed mapping or routing assistance into an email client or discord chat channel. One observation you might make is that the verb "to go" is often followed by a noun phrase that indicates a location. Here are some examples:

1. We **went to *the store***.
2. Brody is going to **go to *a library***.
3. It all depends on whether she **goes to *the stadium***.
4. When are you **going into *that salon***?
5. They **went into *the red building*** and didn't come out.
6. I saw them **go into *the ugly, red building*** and then leave.

You could probably write an ordinary regular expression that would help you capture the locations mentioned above, but knowing, for example, that "go", "going" and "goes" are all different forms of the verb "to go" will simplify things. SpaCy identifies **lemmas** to help with this. Similarly, knowing that "a", "the", and "that" are all determiners will be very useful as well. SpaCy identifies **parts of speech** to help with that.

Here is a first cut at a pattern that will match some of the sentences above. In the pattern below, you're looking for a word that has the lemma "go", followed by the word "to" and then a determiner and a noun:

This pattern matches sentences 1, 2, and 3 above. To capture sentences 4, 5 and 6 you will have to figure out how to match "into" as well as "to" and expand your definition of a noun phrase to include adjectives.

BASIC TOKEN MATCHING

To create a **pattern matcher**, first import spaCy's **Matcher** class and use it to create a **Matcher** instance.

```
import spacy
from spacy.matcher import Matcher
nlp = spacy.load("en_core_web_sm")
matcher = Matcher(nlp.vocab)
```

A spaCy **pattern** is a **list** of **dictionaries** where each dictionary object is used to match a token. Here is a pattern that matches the diagram in the previous section.

```
pattern = [
      {"LEMMA": "go"},        ← any token that has "go" as its lemma
      {"LOWER": "to"},        ← any token with a lowercase form that matches "to"
      {"POS": "DET"},         ← any token with DET as its part of speech
      {"POS": "NOUN"}         ← any token with NOUN as its part of speech
]
```

You can add this pattern to the **Matcher** and name it "location phrase", as shown below. Note that the **add** method requires a list of patterns, so you have to put the single pattern from above inside square brackets.

```
matcher.add("location phrase", [pattern])
```

Now you can use the pattern to search a document for matches. In the example below, the document contains all 6 sentences from above.

```
doc = nlp("We went to the store. Brody is going to go to a
library. It all depends on whether she goes to the stadium. When
are you going into that salon? They went into the red building
and didn't come out. I saw them go into the ugly, red building
and then leave.")

matches = matcher(doc)
```

The **matches** variable above contains a **list** of **tuples** that indicate the id and the span of each match. As expected, this document contains 3 matches:

```
[
 (16835261413858320 30, 1, 5),
 (16835261413858320 30, 10, 14),
 (16835261413858320 30, 21, 25)
]
```

The long number in each tuple is the "match id" and will probably be different when you run the code yourself. This id is linked to the name given to this particular pattern. You can see the name like this:

```
>>> nlp.vocab.strings[16835261413858320 30]
'location phrase'
```

The id is useful because you can load a single matcher with multiple patterns for different kinds of entities. Then the id in each tuple tells you which pattern was matched.

The other pair of numbers defines the span of the phrase. To get the set of tokens from the third match above, you can create a span object like this:

```
>>> phrase1 = doc[21:25]
>>> print(phrase1.text)
goes to the stadium
```

Since you know the first two tokens in each phrase are "go" and "to" you can extract the locations like this:

```
spans = []
for match_id, start, end in matches:    ← assigns each tuple to 3 variables
    spans.append(doc[start+2,end])
print(spans)
```

This prints the following:

```
[the store, a library, the stadium]
```

TOKEN RANGES

To match sentence 4, you need to match "into" as well as "to". You could use the **ADP** (preposition) part of speech here, but that might be too broad (e.g. it would match non-location phrases "I saw him **go for *the cookie***" and "They **went about *their business***"). So maybe you just want to match a few specific prepositions, like "to", "into", and "toward" – a **range** of tokens. This can be done with the **IN** operator.

The syntax for this is a little clunky, requiring the use of nested lists and dictionaries. The rule below says the second token's lowercase form can match any one several possibilities.

```
pattern = [
      {"LEMMA": "go"},
      {"LOWER": {"IN": ["to", "into", "toward"]}},
      {"POS": "DET"},
      {"POS": "NOUN"}
]
```

Making this change to the code gets you the location from the 4th sentence as well. Re-running the matching and extracting code yields the following:

```
[the store, a library, the stadium, that salon]
```

Note that the **IN** operator will work with **LEMMA** and **POS** as well.

REPETITION

To match sentences 4 and 5 and other sentences like it, you need to be able to specify that there could be adjectives between the determiner and the noun. The problem is you do not know in advance exactly how many adjectives to expect, so you need to be able to specify zero or more matches. Repetition of this kind is done with the *****, **+**, and **?** operators you know and love from regular expressions.

```
pattern = [
      {"LEMMA": "go"},
      {"LOWER": {"IN": ["to", "into", "toward"]}},
      {"POS": "DET"},
      {"POS": "ADJ", "OP": "*"},        ← match 0 or more adjectives
      {"POS": "NOUN"}
]
```

This is enough to match “the red building” but it doesn’t match “the ugly, red building” because of the comma token. But you can fix that by allowing the comma as well, which has a POS label of PUNCT.

```
pattern = [
      {"LEMMA": "go"},
      {"LOWER": {"IN": ["to", "into", "toward"]}},
      {"POS": "DET"},
      {"POS": {"IN": ["ADJ", "PUNCT"]}, "OP": "*"},
      {"POS": "NOUN"}
]
```

Now the pattern is extracting all 6 locations:

```
[the store, a library, the stadium, that salon, the red building,
the ugly, red building]
```

WILD CARDS

There might be some instances where you want to be able to match any token or token sequence (like using **.** or **.*** in a regular expression). You can do this by simply not specifying "LEMMA", "POS", "LOWER", or any other attribute for matching.

```
{}          ← matches any token
{OP="?"}    ← matches any token 0 or 1 times
{OP="*"}    ← matches any token 0 or more times
{OP="+"}    ← matches any token 1 or more times
```

You should use wild cards sparingly, because they can often end up matching too much, but it is good to know you can do it if you need to.

ADDING MORE PATTERNS

A single **Matcher** object can contain as many patterns as you like. For example, the code below will add a new pattern with a different name to the **Matcher** object from before.

```
matcher.add("determiner", [[{"POS":"DET"}]])
```

Now when you run the matcher, you will get matches of type “location phrase” and “determiner” in the result.

You can also add more patterns with the same name. For example, you might develop a new pattern for location phrases based on a different leading verb (“drive to” or “walk to”) or some other type of pattern. You can add the new pattern like this:

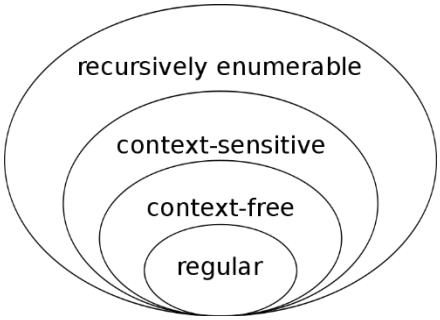
```
matcher.add("location phrase",[new_pattern])
```

Now if a phrase matches either “location phrase” pattern it will get marked as a “location phrase”.

A REMINDER: THE LIMITATIONS OF REGULAR GRAMMARS

You might recall from the first lesson on NLP that regular grammars can only be used to model the simplest types of formal languages. As powerful and useful as spaCy’s pattern matcher is, it could not be used to match even the simple, context-free grammar of a programming language, let alone model the full grammar of a natural language.

Because of this limitation, you should not expect your patterns to be 100% accurate. Instead, an NLP engineer is generally happy when their pattern matching rules reveal *some* useful information about a document or utterance at least *some* of the time.



Chomsky’s hierarchy of Formal Languages. From bsodtutorials.wordpress.com.

FURTHER HELP

We have covered most of the pattern matching basics here, but for more help, you can go to the spaCy documentation.

https://spacy.io/usage/rule-based-matching
https://spacy.io/api/matcher

EXERCISES

1. The pattern developed for locations above may be over productive. For example, it will match sentences like "This place is going to the dogs" or "I went to the mat for you", neither of which contain location phrases. Try loading a large file of English text from https://gutenberg.org, run it through the matcher, and look at the phrases it identifies. How many matches did it find? How many of the identified phrases name locations? Can you improve the pattern based on your results?

2. Create and test a pattern matcher to identify proper names (first and last name, first and last name with initial). Test it on manifesto.txt or another large file of English text from https://gutenberg.org.

3. Write a simple bot that uses spaCy's Matcher to identify and then ask questions about prepositional phrases. Here is an example of a dialog with the bot:

   ```
   USER: The cat in that big, black box was placed in the car
   BOT: What else was in a big, black box? What else was in a car?
   ```

   Hint: Match prepositional phrases using a rule similar to the one developed above. Then echo back the preposition that was matched, followed by "a", followed by the noun phrase.

4. Bruce, the automated assistant, can respond to voice commands to play any Spotify artist the user wants. Write a simple bot that responds to such requests with the template "Now playing ___ on Spotify", where "___" is the artist requested. If the user is not asking the bot to play anything, it should not respond.

   Some examples of transcribed commands are given below – you should match and respond to the ✓ examples and not the ✗ examples.

   ```
   ✓ Please play some St. Vincent
   ✓ Please start playing The Beatles
   ✓ I would love it if you played some Neil Young
   ✗ Please tell me the weather, Bruce.
   ✗ That was a terrible play dude
   ```

5. Bruce also does simple arithmetic in response to voice commands. Some examples of transcribed commands are shown below. You might have to research the spaCy matcher and Python to figure out how to identify numbers and convert strings to numbers. How flexible can you make this functionality?

   ```
   USER: What's 5 plus 7 please
   BOT: 5 plus 7 is 12

   USER: Tell me what 56 times 2 is
   BOT: 56 times 2 is 112

   USER: Compute 5 point 5 minus 5
   BOT: 5.5 minus 5 is 0.5
   ```

6. Bruce the assistant can also send text messages using the contacts on your phone. When it detects a voice command to send a message, it always responds with the recipient and the message, like this "Messaging Pat: You up".

   Here is a list of transcribed voice commands. You should match and respond to the ✓ examples and not the ✗ examples.

   - ✓ `Tell Prewal that I am on my way`
   - ✓ `Message Sam Scott I am going to the store.`
   - ✗ `Tell me I didn't just do that`
   - ✗ `Tell you what I want, what I really really want`
   - ✓ `Please message my boyfriend that I love him`
   - ✓ `Tell dad I'll meet him later`