8.

Using RegEx with spaCy

Dr. W.J.B. Mattingly
Smithsonian Data Science Lab and United States Holocaust Memorial Museum
January 2021

Contents

Print to PDF

8.1. Key Concepts in this Notebook

1. What is RegEx (Regular Expressions)?
2. The Strengths of RegEx
3. The Weaknesses of RegEx
4. How to use RegEx in Python
5. How to use RegEx in spaCy

8.2. What is Regular Expressions (RegEx)?

Regular Expressions, or RegEx for short, is a way of achieving complex string matching based on simple or complex patterns. It can be used to perform finding and retrieving patterns or replacing matching patterns in a string with some other pattern. It was invnted by an Stephen Cole Kleene in the 1950s and is still widely used today for numerous tasks, but particularly string matching in texts. RegEx are fully integrated with most search engines and can allow for more robust searching. Nearly all data scientists, especially those who work with texts, use RegEx at some stage in their workflow, from data searching, to cleaning data, to implementing machine learning models. It is an essential tool for any text-based researcher. For these reasons, it merits a few chapters in this textbook.

In spaCy it can be leveraged in a few different pipes (depending on the task at hand as we shall see), to identify things such as entities or pattern matching.

8.3. The Strengths of RegEx

There are several strengths to RegEx.

1. Due to its complex syntax, it can allow for programmers to write robust rules in short spaces.
2. It can allow the researcher to find all types of variance in strings
3. It can perform remarkably quickly when compared to other methods.
4. It is universally supported

8.4. The Weaknesses of RegEx

Despite these strengths, there are a few weaknesses to RegEx.

1. Its syntax is quite difficult for beginners. (I still find myself looking up how to do certain things).
2. It order to work well, it requires a domain-expert to work alongside the programmer to think of all ways a pattern may vary in texts.

8.5. How to Use RegEx in Python

Python comes prepackaged with a RegEx library. We can import it like so:

```
import re
```

Now that we have it imported, we can begin to write out some RegEx rules. Let's say we want to find an occurrence of a date in a text. As noted in an earlier notebook, there are a finite number of ways this can be represented. Let's try to grab all instances of a day followed by a month first.

```
pattern = r"((\d){1,2}
(January|February|March|April|May|June|July|August|September|October|November|December)
)"

text = "This is a date 2 February. Another date would be 14 August."
matches = re.findall(pattern, text)
print (matches)
```

```
[('2 February', '2', 'February'), ('14 August', '4', 'August')]
```

In this bit of code, we see a real-life RegEx formula at work. While this looks quite complex, its syntax is fairly straight forward. Let's break it down. The first (tells RegEx that I'm looking for something within the ending). In other words, I'm looking for a pattern that's going to match the whole pattern, not just components.

Next, we state (\d){1,2}. This means that we are looking for any digit (0-9) that occurs either once or twice ({1,2}).

Next, we have a space to indicate the space in the string that we would expect with a date.

Next, we have (January|February|March|April|May|June|July|August|September|October|November|December) – this indicates another component of the pattern (because it is parentheses). The | indicates the same concept as "or" in English, so either January, or February, or March, etc.

When we bring it together, this pattern will match anything that functions as a set of one or two numbers followed by a month. What happens when we try and do this with a date that is formed the opposite way?

```
text = "This is a date February 2. Another date would be 14 August."
matches = re.findall(pattern, text)
print (matches)
```

```
[('14 August', '4', 'August')]
```

It fails. But this is no fault of RegEx. Our pattern cannot accommodate that variation. Nevertheless, we can account for it by adding it as a possible variation. Possible variations are accounted for with a *

```
pattern = r"(((\d){1,2}(
(January|February|March|April|May|June|July|August|September|October|November|December)
))|
(((January|February|March|April|May|June|July|August|September|October|November|Decembe
r) )(\d){1,2}))"

text = "This is a date February 2. Another date would be 14 August."
matches = re.findall(pattern, text)
print (matches)
```

```
[('February 2', '', '', '', '', 'February 2', 'February ', 'February', '2'), ('14
August', '14 August', '4', ' August', 'August', '', '', '', '')]
```

There are more concise ways to write the same RegEx formula. I have opted here to be more verbose to make it a bit easier to read. You can see that we've allowed for two main options for our pattern matcher.

Notice, however, that we have a lot of superfluous information for each match. These are the components of each match. There are several ways we can remove them. One way is to use the command finditer, rather than findall in RegEx.

```
text = "This is a date February 2. Another date would be 14 August."
iter_matches = re.finditer(pattern, text)
print (iter_matches)
```

```
<callable_iterator object at 0x00000217A415BC10>
```

This is an iterator object, we can loop over it, however, and get our results.

```
text = "This is a date February 2. Another date would be 14 August."
iter_matches = re.finditer(pattern, text)
print (iter_matches)
for hit in iter_matches:
    print (hit)
```

```
<callable_iterator object at 0x00000217A4256670>
<re.Match object; span=(15, 25), match='February 2'>
<re.Match object; span=(49, 58), match='14 August'>
```

Within each of these is some very salient information, such as the start and end location (inside the span) and the text itself (match). We can use the start and end location to grab the text within the string.

```
text = "This is a date February 2. Another date would be 14 August."
iter_matches = re.finditer(pattern, text)
for hit in iter_matches:
    start = hit.start()
    end = hit.end()
    print (text[start:end])
```

```
February 2
14 August
```

8.6. How to Use RegEx in spaCy

Things like dates, times, IP Addresses, etc. that have either consistent or fairly consistent structures are excellent candidates for RegEx. Fortunately, spaCy has easy ways to implement RegEx in three pipes: Matcher, PhraseMatcher, and EntityRuler. One of the major drawbacks to the Matcher and PhraseMatcher, is that they do not align the matches as doc.ents. Because this textbook is about NER and our goal is to store the entities in the doc.ents, we will focus on using RegEx with the EntityRuler. In the next notebook, we will examine other methods.

In the previous notebook, we saw how the code below allowed for us to capture the phone number in the string. I have modified it a bit here for reasons that will become a bit more clear below.

```
#Import the requisite library
import spacy

#Sample text
text = "This is a sample number 555-5555."

#Build upon the spaCy Small Model
nlp = spacy.blank("en")

#Create the Ruler and Add it
ruler = nlp.add_pipe("entity_ruler")

#List of Entities and Patterns (source: https://spacy.io/usage/rule-based-matching)
patterns = [
                {"label": "PHONE_NUMBER", "pattern": [{"SHAPE": "ddd"},
                {"ORTH": "-", "OP": "?"}, {"SHAPE": "dddd"}]}
            ]
#add patterns to ruler
ruler.add_patterns(patterns)

#create the doc
doc = nlp(text)

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

```
INFO:tensorflow:Enabling eager execution
INFO:tensorflow:Enabling v2 tensorshape
INFO:tensorflow:Enabling resource variables
INFO:tensorflow:Enabling tensor equality
INFO:tensorflow:Enabling control flow v2
555-5555 PHONE_NUMBER
```

This method worked well for grabbing the phone number. But what if we wanted to use RegEx as opposed to linguistic features, such as shape? First, let's write some RegEx to capturee 555-5555.

```
pattern = r"((\d){3}-(\d){4})"
text = "This is a sample number 555-5555."
matches = re.findall(pattern, text)
print (matches)
```

```
[('555-5555', '5', '5')]
```

Okay. So, now we know that we have a RegEx pattern that works. Let's try and implement it in the spaCy EntityRuler. We can do that with the code below. When we execute the code below, we have no output.

```
#Import the requisite library
import spacy

#Sample text
text = "This is a sample number (555) 555-5555."

#Build upon the spaCy Small Model
nlp = spacy.blank("en")

#Create the Ruler and Add it
ruler = nlp.add_pipe("entity_ruler")

#List of Entities and Patterns (source: https://spacy.io/usage/rule-based-matching)
patterns = [
                {
                    "label": "PHONE_NUMBER", "pattern": [{"TEXT": {"REGEX": "((\d){3}-
(\d){4})"}}
                                                        ]
                }
            ]
#add patterns to ruler
ruler.add_patterns(patterns)


#create the doc
doc = nlp(text)

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

This is for one very important reason. SpaCy's EntityRuler cannot use RegEx to pattern match across tokens. The dash in the phone number throws off the EntityRuler. So, what are we to do in this scenario? Well, we have a few different options that we will explore in the next notebook. But before we get to that, let's try and use RegEx to capture the phone number with no hyphen.

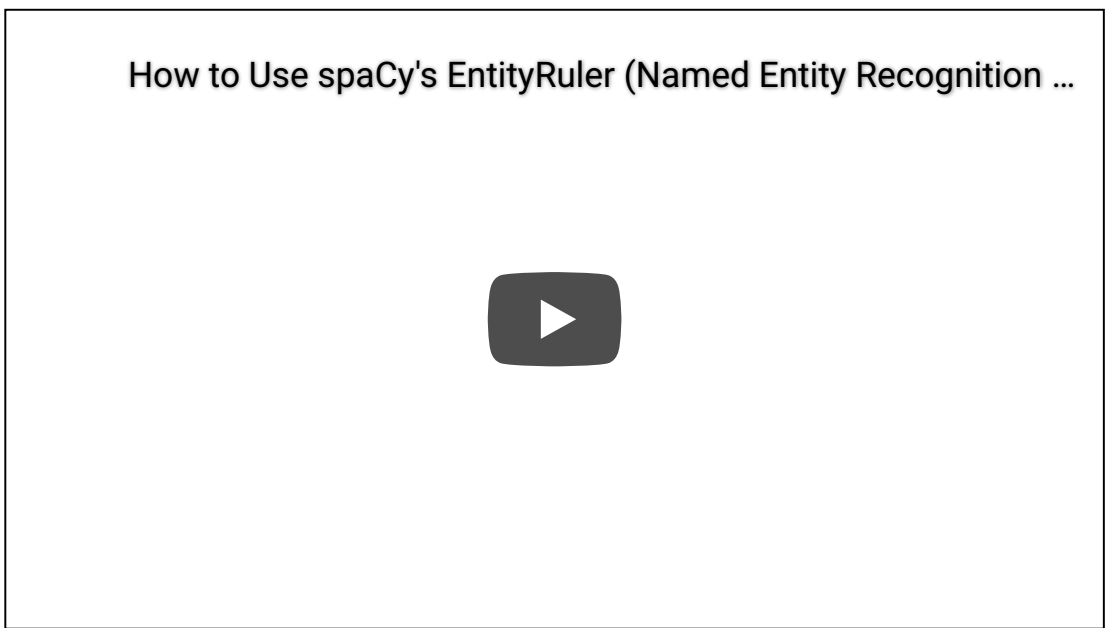
```
#Import the requisite library
import spacy

#Sample text
text = "This is a sample number 5555555."
#Build upon the spaCy Small Model
nlp = spacy.blank("en")

#Create the Ruler and Add it
ruler = nlp.add_pipe("entity_ruler")

#List of Entities and Patterns (source: https://spacy.io/usage/rule-based-matching)
patterns = [
                {
                    "label": "PHONE_NUMBER", "pattern": [{"TEXT": {"REGEX": "((\d)
{5})"}}
                                                        ]
                }
            ]
#add patterns to ruler
ruler.add_patterns(patterns)


#create the doc
doc = nlp(text)

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

```
5555555 PHONE_NUMBER
```

Notice that without the dash and a few modifications to our RegEx, we were able to capture 5555555 because this is a single token in the spaCy doc object. Let's explore how to solve the problem in the next notebook!

8.7. Video

```
%%html
<div align="center">
<iframe width="560" height="315" src="https://www.youtube.com/embed/wpyCzodvO3A"
frameborder="0" allow="accelerometer; autoplay; clipboard-write; encrypted-media;
gyroscope; picture-in-picture" allowfullscreen></iframe>
</div>
```



By William Mattingly