Python Knowledge Graph: Understanding Semantic Relationships

NLP tutorial for building a Knowledge Graph with class-subclass relationships using Python, NLTK and SpaCy.

MARIUS BORCAN

8 AUG 2020 • 16 MIN READ

Knowledge Graphs are very powerful NLP tools and advanced studies in the field of Knowledge Graphs have created awesome products that are used by milions of people everyday: think of Google, Youtube, Pinterest, they are all very important companies in this field and their knowledge graphs results are spectacular to analyze and use.

Photo by Kuma Kum / Unsplash

In one of my previous articles I wrote about a naive approach on building a small knowledge graph based on triples. But the thing is, the more spectacular knowledge graphs are, the more difficult they are to build. You actually need more than one way of building a feature like this: think of triples, relationships, integrating with other data sources and so on.

In this article I'm going to talk about a small subset of knowledge graph

relationships: *type-of relationships* or *is-a relationships*, meaning we will try to build a small knowledge graph using Python, SpaCy and NLTK. The knowledge graph will tell us if a certain object is a subclass (a type) of another object.

> *Interested in more? Follow me on Twitter at @b_dmarius and I'll post there every new article.*

As usual on this blog, I will go through a little bit of theory, then code presentation and explanations and in the end results analysis. Feel free to skip to whichever section you feel is relevant for you. Also, all the code for this article is uploaded on Github so you can check it out (please make sure to star the repository as it helps me know the code I write is helpful in any way). We will go through all the code anyways.

Article Overview

- What is Information Extraction
- What is a Knowledge Graph
- Knowledge Graph applications
- Semantic relationships: hypernyms and hyponyms
- Python Knowledge Graph project overview and setup
- Python Knowledge Graph implementation using Python and SpaCy
- Conclusions
- Related articles

- References

What is Information Extraction

Information Extraction is one of the most important fields of Natural Language Processing tasks and it consists of techniques of extracting structured information from unstructured text.

There is a lot of information out there stored in plain text that we as humans are able to understand in a blink, but computers have lots of troubles with this task because they don't understand text, language and context.

So in information extraction tasks we try to process textual information and transform it in a way that computers are able to understand and use. For this we need to use various NLP tasks like:

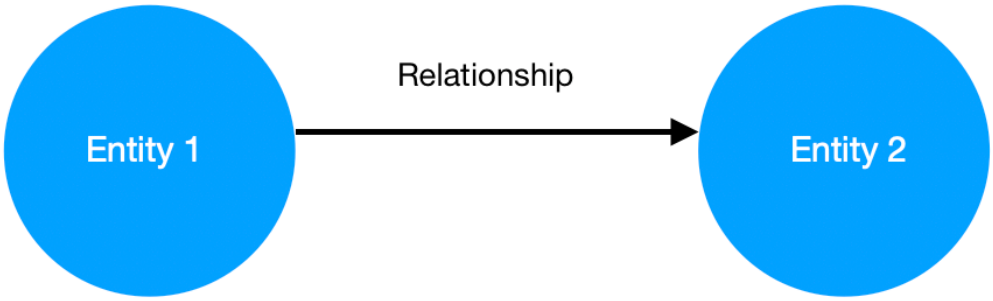
- Named Entity Recognition: the task of understanding where and how entities such as people, organisations, events and so on are mentioned in a text
- Named Entity Linking: understand how 2 or more entities are related to each other
- Keywords extraction: Extracting the most relevant words from a text. This can be done using multiple algorithms
- Relations extraction using triples
- Knowledge Graphs

- Knowledge Graphs

What is a Knowledge Graph

A knowledge graph is a particular representation of data and data relationships which is used to model which entities and concepts are present in a text corpus and how these entities relate to each other.

The concept of Knowledge Graphs borrows from the Graph Theory. In this particular representation we store data as:



Knowledge Graph relationship

Entity 1 and Entity 2 are called nodes and the Relationship is called an edge. Of course, in a real world knowledge graph there are lots of entities and relationships and there is more than one way to arrive at one entity starting from another.

Usually these type of graphs are modeled with **triples**, which are sets of three

items like (subject, verb, object), with the verb being the relationship between the subject and the object - for example (London, is_capital, England).

In this article we are focusing on only one particular type of relationship, the "is-a" relationship. If we replace this in the image above we read it as "Entity 1 is a type of Entity 2", meaning Entity 2 is the broader type and Entity 1 is the narrower type - for example (Londin, is_a, City).

Knowledge Graph applications

Knowledge Graphs have broad applications, out of which some have not even been succesfully built yet. If you need to better understand your data and the relationships between your data points, a knowledge graph is the way to go. But there are some particulary famous examples of uses of knowledge graphs used in real world use cases:

- The Google Knowledge Graph is particulary because they have made this term so popular in the latest years. They use knowledge graphs to enhance search results using some additional information that has been gathered from lots of sources on the internet. They also use it for the Instant Answers feature, where you get an info box answering a question you put in the Google search bar (try for example "Who is the godfather of Harry Potter?"

- Pinterest is using knowledge graphs to model interests of users on the platform. The Pinterest engineering blog is one of my favourites, feel free to read some of their posts.
- Youtube is also using Knowledge Graph to understand what's behind a video and to recommend the videos to users (thus helping them solve the cold-start problem of their collaborative filtering algorithm).

Semantic relationships: hypernyms and hyponyms

So we said we are going to use Python and SpaCy to build a knowledge graph containing "is-a" relationships. But before that (and I promise this is the last introductory section) we need to look into some theoretical aspects.

In more fancy linguistics terms, "is-a" relationships are named **Hypernymy** and **Hyponymy** relationships. So for example, if we say "Harry Potter is a book character", then "Harry Potter" is the hyponym (the narrow entity) of the relationship, while "book character" is the hypernym (the broad entity) of the relationship.

And because we are using only plain text to extract such information, we need to look at the structure of the sentences, take a look at what Part Of Speech each word represents and try to figure out relationships from there. That's why we say that we are analyzing **semantic relationships**

that we are analyzing **semantic relationships**.

Now, there are many techniques we can use to extract relationships from text: supervised, unsupervised, semi-supervised techniques are rule-based techniques. And in this article we are going to take advantage of the fact that English is a well-structured language, so we can go with the rule-based techniques.

There has been a lot of research in this area but a popular piece of research is done by Marti Hearst [1] the results from this research are popularly known as the Hearst Patterns. She has identified a few patterns that can be used in English to extract hypernyms and hyponyms. In the following table hyponyms are represented by *h* and hypernyms by *H*.

Pattern	Example
h and other H	London, Paris and other cities
H such as h	wars such as WWI and WWII
h or other H	London, Paris or other cities
H including h	European countries, including France and Spain
H especially h	European countries, especially France and Spain

Hearst Patterns

We are going to use these patterns to try and figure out is-a relationships from plain text extracted from Wikipedia. It's now time to switch to the real action.

Python Knowledge Graph project overview and setup

We are going to extract the text from 4 Wikipedia articles about 2 different subjects: London, Paris, WWI and WWII. We are going to use the Hearst Patterns to extract relationships from these 4 articles and add them to a graph. Then are going to display the graph and analyze of results. First let's install some dependencies.

```
pip3 install spacy
pip3 install wikipedia
pip3 install nltk
pip3 install networkx
pip3 install matplotlib
```

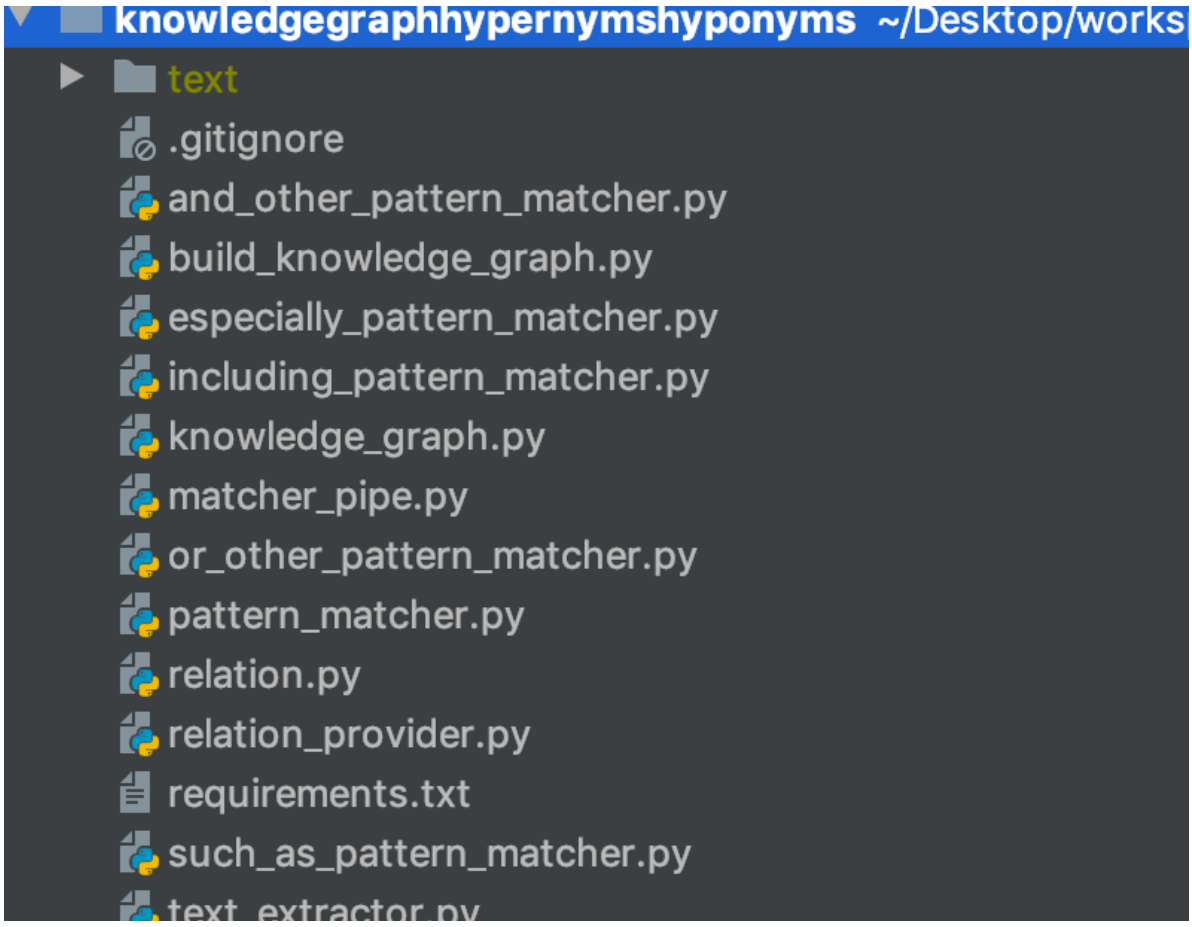
SpaCy is used for text processing, wikipedia is used for extracting the data. We are using NLTK just for a visualization of the relationships between words in a sentence. Networkx is used for building the graph and matplotlib is used for visualization.

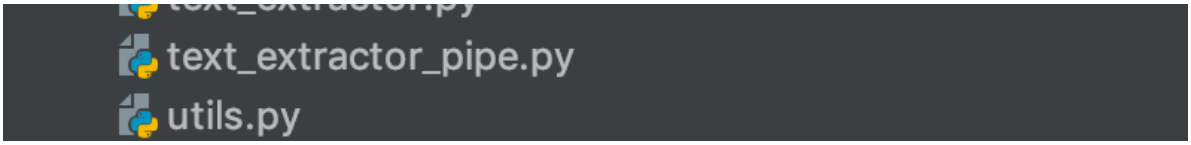
Another command you should run in your terminal (especially if it's the first time you are using spaCy or if you are using a virtual environment is

```
python3 -m spacy download en_core_web_sm
```

This is used to download the spaCy pre-trained model for English that we are going to use in this project. This the the small model and another, larger one is available (en_core_web_lg) but that is not necessary for this project.

Let's take a quick peek at our project file structure.





Project file structure

There are quite a lot of file, but we are going to go through each other one by one and I'll provide simple explanations.

Python Knowledge Graph implementation using Python and SpaCy

First let's get this out of our way: the *utils.py* file contains a small utility function that I've added to visualize the structure of a sentence. It uses the NLTK Tree and it is inspired by this StackOverflow answer.

```
from nltk import Tree

def buildTree(token):
    if token.n_lefts + token.n_rights > 0:
        return Tree(token, [buildTree(child) for chil
    else:
        return buildTree(token)
```

We are going to store relations in a *Relation* object and the code for this class is self-explanatory and located in *relation.py*.

```
class Relation:

    __hypernym: str
    __hyponym: str

    def __init__(self, hypernym, hyponym):
        self.__hypernym = hypernym
        self.__hyponym = hyponym

    def getHypernym(self):
        return self.__hypernym

    def getHyponym(self):
        return self.__hyponym
```

I've also written another class to store all relations. The class is stored in *relation_provider.py* and, again, it is fairly simple.

```
from relation import Relation

class RelationProvider:


    __relations: [Relation]

    def __init__(self, relations=[Relation]):
        self.__relations = relations

    def getRelations(self):
        return self.__relations
```

The first step is to extract the text from Wikipedia. We are using the *wikipedia* package to get that, and this functionality is found in *text_extractor.py*. We are first downloading the data and storing it in a local file. To get the text, we are reading that file and returing the entire text.

```
import wikipedia


class TextExtractor:

    __pageTitle: str
```

```
    __pageId: str

    def __init__(self, pageTitle, pageId):
        self.__pageTitle = pageTitle
        self.__pageId = pageId


    def extract(self):
        page = wikipedia.page(title=self.__pageTitle,
        f = open("./text/" + self.__pageTitle + ".txt
        f.write(page.content)
        f.close()

    def getText(self):
        f = open("./text/" + self.__pageTitle + ".txt
        return f.read()
```

The package that we are using today usually requires only the text for English pages. But, sometimes it gets confused, so that's why I've included the pageId field of the article. To get the pageId of a Wikipedia article, you need to go to Wikidata and search for the article there. The page id will be found in brackets after the title of the result.

As I said we are going to extract text from more than one article so I've written a small pipe class that takes a collection of text extractors, runs them to get the text

and concatenates the results. This is found in *text_extractor_pipe.py*.

```
from text_extractor import TextExtractor


class TextExtractorPipe:

    __textExtractors: [TextExtractor]

    def __init__(self):
        self.__textExtractors = []

    def addTextExtractor(self, textExtractor: TextExt
        self.__textExtractors.append(textExtractor)

    def extract(self) -> str:
        result = ''
        for textExtractor in self.__textExtractors:
            result = result + textExtractor.getText()
        return result
```

Now we need to write our pattern matchers. We are going to use the *Matcher* class from spaCy and add some other functionality of our own. Because I want to pipe multiple matchers and pass the text through all of them at once, I've written a base class for all the matchers which contains an abstract

Once, I've written a base class for all the matchers which contains an abstract method that will be implemented by all the matchers. This is the *pattern_matcher.py* file.

```
from spacy.matcher import Matcher
from abc import abstractmethod
from spacy.tokens import Doc
from relation import Relation


class PatternMatcher:


    def __init__(self, pattern, nlp, matcherId):
        self._nlp = nlp
        self._matcher = Matcher(nlp.vocab)
        self._matcher.add(matcherId, None, pattern)

    @abstractmethod
    def getRelations(self, doc: Doc) -> [Relation]:
        ...
```

Let's take a closer look at the constructor. The *pattern* parameter contains the actual pattern that each matcher will use to extract the nodes for our knowledge graph. Then we have the *nlp* argument, which is the spaCy pre-trained NLP

model. Finally, the matcherId is just a string that helps us identify from which matcher each match comes. Now let's take a look at each matcher class to see the logic behind them.

We are starting with a simple pattern, the "***h** and other **H***" one. The class is found in *and_other_pattern_matcher.py* file.

```
from pattern_matcher import PatternMatcher
from spacy.tokens import Doc
from relation import Relation


class AndOtherPatternMatcher(PatternMatcher):


    def __init__(self, nlp):
        pattern = [{'POS': 'NOUN'},
                   {'LOWER': 'and'},
                   {'LOWER': 'other'},
                   {'POS': 'NOUN'}]
        PatternMatcher.__init__(self, pattern, nlp, "

    def getRelations(self, doc: Doc) -> [Relation]:
        relations = []
        matches = self._matcher(doc)
        for match_id, start, end in matches:
```

```
            span = doc[start:end]
            firstToken = span.root.head
            results = [firstToken]
            while firstToken and firstToken.head.pos_
                results.append(firstToken.head)
                firstToken = firstToken.head
            hypernym = span.text.split()[-1]
            relations.append(Relation(hypernym, span.

            if len(results) > 0:
                for result in results:
                    relations.append(Relation(hyperny
        return relations
```

In the constructor you can observe the pattern we are using for this matcher. We are telling the matcher: "look for structures containing 4 words: the first word is a NOUN (POS stands for Part-Of-Speech), second word is <<and>>, third is <<other>> and the last word is also a Noun". Then we override the abstract method defined in the PatternMatcher class.

SpaCy is doing the hard work for us here. Remember the Matcher class imported in the base class of this matcher? That class takes a document, runs is through the patterns we've defined and returns a list of matches. The list of matches is

actually a list of spaCy Span objects, which is a container for one or more words.

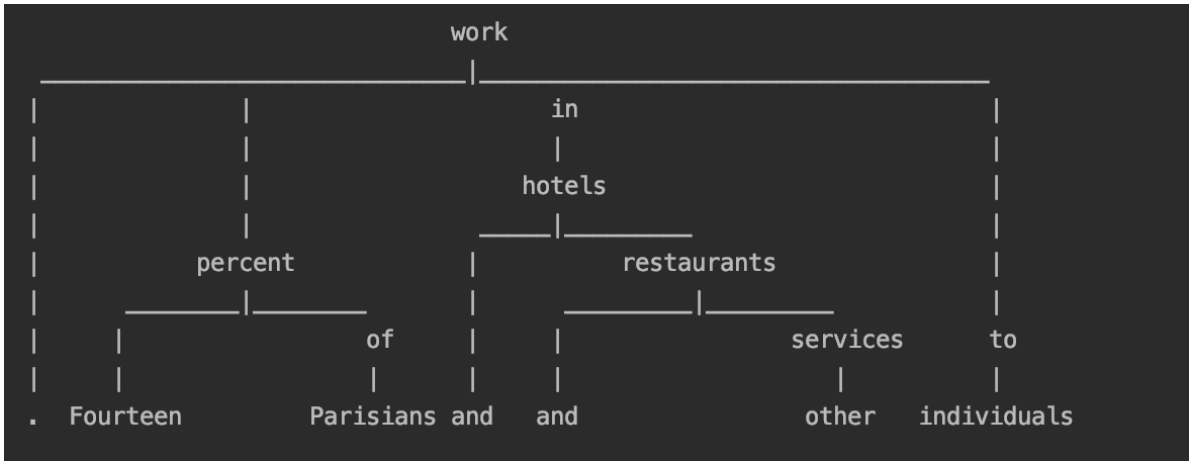
The *match_id* is unique for each match and the *start* and *end* values are positions of each match in the sentence.

Now a basic scenario would be: "Ok, I've found my match, I take the first word as a hyponym, the last word hypernym and that's it, I have my relation". But it's not that simple, because we might have more than one hyponym in the same relation and we want to capture as much information as we can.

For example, let's take this sentence from the article about Paris:

"*Fourteen percent of Parisians work in hotels and restaurants and other services to individuals.*"

Ideally, we should be able to capture that both hotels and restaurants are types of services. Let's take a look at the sentence structure:



Sentence structure

So we know where our "services" is located - at the end of our matched Span.

```
hypernym = span.text.split()[-1]
```

We also know that our first hyponym is at the beginning of our matched Span. So we can already build our first Relation.

```
relations.append(Relation(hypernym, span.text.split()
```

We can also see that the second hyponym as the parent of our first hyponym. Naturally, a third hyponym, if it existed, would have been the parent of our second hyponym. So the only solution is to go to the top of the sentence, until we find the first word that is not a NOUN. That's what the code for this class does.

The next pattern is "***h** or other **H***" and yes, your intuition is right, this is the same logic. The code for this is located in *or_other_pattern_matcher.py*.

```
from pattern_matcher import PatternMatcher
from spacy.tokens import Doc
from relation import Relation


class OrOtherPatternMatcher(PatternMatcher):

    def __init__(self, nlp):
        pattern = [{'POS': 'NOUN'},
                   {'LOWER': 'or'},
                   {'LOWER': 'other'},
                   {'POS': 'NOUN'}]
        PatternMatcher.__init__(self, pattern, nlp, "

    def getRelations(self, doc: Doc) -> [Relation]:
        relations = []
        matches = self._matcher(doc)
        for match_id, start, end in matches:
            span = doc[start:end]
            firstToken = span.root.head
            results = [firstToken]
            while firstToken and firstToken.head.pos_
                results.append(firstToken.head)
                firstToken = firstToken.head
            hypernym = span.text.split()[-1]
            relations.append(Relation(hypernym, span.
            if len(results) > 0:
                for result in results:
```

```
                    relations.append(Relation(hyperny
        return relations
```

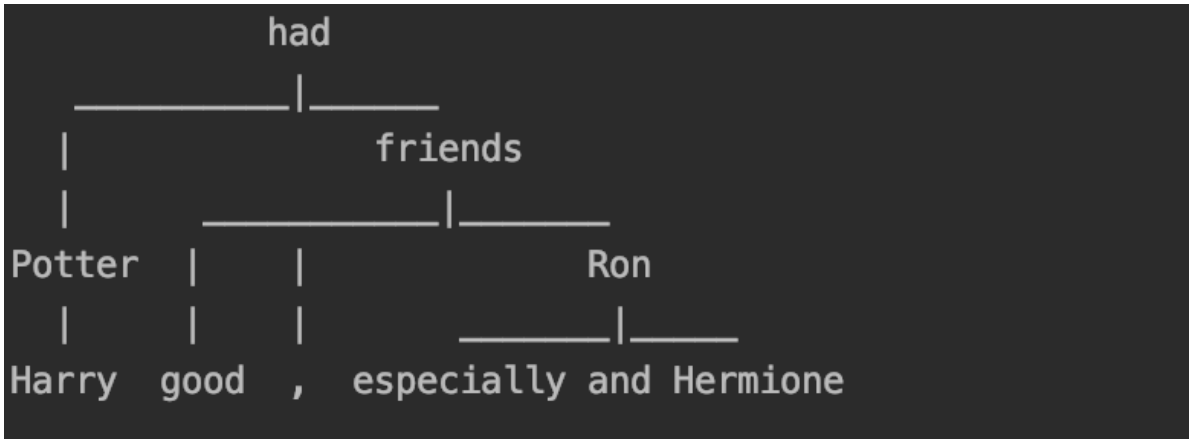
It's time now for our "***H** especially **h***" pattern. This one is matched in the *especially_pattern_matcher.py* file.

```
from pattern_matcher import PatternMatcher
from spacy.tokens import Doc
from relation import Relation


class EspeciallyPatternMatcher(PatternMatcher):

    def __init__(self, nlp):
        pattern = [{'POS': 'NOUN'},
                   {'IS_PUNCT': True, 'OP': '?'},
                   {'LOWER': 'especially'},
                   {'POS': 'NOUN'}]
        PatternMatcher.__init__(self, pattern, nlp, "

    def getRelations(self, doc: Doc) -> [Relation]:
        relations = []
        matches = self._matcher(doc)
        for match_id, start, end in matches:
```

```
        for match_id, start, end in matches:
            span = doc[start:end]
            candidates = set()
            for sent in doc.sents:
                for token in sent:
                    # Find relation
                    if token.i == span.root.i:
                        for token2 in sent:
                            # First hyponym
                            if token2.head.i == token
                                for token3 in sent:
                                    startToken = toke
                                while startToken
                                    if startToken
                                        candidate
                                    startToken =
            if len(candidates) > 0:
                hypernym = span.text.split()[0].repla
                for candidate in candidates:
                    relations.append(Relation(hyperny

        return relations
```

This one is a little bit longer, but is actually simple. Let's look at an example.

"Harry Potter had good friends, especially Ron and Hermione"



Sentence structure

The hypernym is simple to locate, it's the first word in our match. So what we do in our matcher class is locate the token that contains this word. Then we navigate

the depdendency tree down, getting the first NOUN child of the hypernym - that's our first hyponym. From there on, we get other NOUN children of the first hyponym and that's it.

What about the "***H**, including **h***"? You're right, it is the same logic like for the previous pattern.

```
from pattern_matcher import PatternMatcher
from spacy.tokens import Doc
from relation import Relation


class IncludingPatternMatcher(PatternMatcher):
```

```
    def __init__(self, nlp):
        pattern = [{'POS': 'NOUN'},
                   {'IS_PUNCT': True, 'OP': '?'},
                   {'LOWER': 'including'},
                   {'POS': 'NOUN'}]
        PatternMatcher.__init__(self, pattern, nlp, "

    def getRelations(self, doc: Doc) -> [Relation]:
        relations = []
        matches = self._matcher(doc)
        for match_id, start, end in matches:
            span = doc[start:end]
            for sent in doc.sents:

                for token in sent:
                    # Find the relation
                    if token.text == "including" and
                        for token2 in sent:
                            # First hyponym
                            if token2.head.i == token
                                results = set()
                                results.add(span.text
                                # Other hyponyms
                                for token3 in sent:
                                    startToken = toke
                                    while startToken
                                        if startToken
                                            results.a
                                        startToken =
                                if len(results) > 0:
                                    hyponym = span.t
```

```
                                        hypernym = span.t
                                        for result in res
                                            relations.app
        return relations
```

The last pattern we have is the "***H such as h***". This one is very simple too.

```
from pattern_matcher import PatternMatcher
from spacy.tokens import Doc
from relation import Relation


class SuchAsPatternMatcher(PatternMatcher):


    def __init__(self, nlp):
        pattern = [{'POS': 'NOUN'},
                   {'IS_PUNCT': True, 'OP': '?'},
                   {'LOWER': 'such'},
                   {'LOWER': 'as'},
                   {'POS': 'NOUN'}]
        PatternMatcher.__init__(self, pattern, nlp, "

    def getRelations(self, doc: Doc) -> [Relation]:
```

```
    def get_relations(self, doc: Doc) -> [Relation]:
        relations = []
        matches = self._matcher(doc)
        for match_id, start, end in matches:
            span = doc[start:end]
            hypernym = span.root.text
            hyponym = span.text.split()[-1]
            relations.append(Relation(hypernym, hypon
            for right in span.rights:
                if right.pos_ == "NOUN":
                    relations.append(Relation(hyperny
        return relations
```

Here we only get the root of the span as the hypernym, then the last word of the span as the first hyponym, and then we navigate the siblings of the first hyponym to the right to find other hyponyms.

Like with the text extractor class, we also have a pipe for our matchers, so that we can run all of them at the same time.

```
from pattern_matcher import PatternMatcher
from relation import Relation
from spacy.tokens import Doc
```

```
class MatcherPipe:

    __matchers: [PatternMatcher]

    def __init__(self):
        self.__matchers = []

    def addMatcher(self, matcher: PatternMatcher):
        self.__matchers.append(matcher)

    def extract(self, doc: Doc) -> [Relation]:
        results = []
        for matcher in self.__matchers:


            results.extend(matcher.getRelations(doc))
        return results
```

Ok, we went through every matcher and now it's time to build the graph. The class that contains the graph is located in *knowledge_graph.py*

```
from relation import Relation
import networkx as nx
import matplotlib.pyplot as plt
```

```
class KnowledgeGraph:

    __relations: [Relation]
    __graph: nx.Graph
    __colors: {}

    def __init__(self, relations):
        self.__relations = relations
        self.__graph = nx.Graph()
        self.__colors = {}

    def build(self):
        for relation in self.__relations:
            self.__graph.add_node(relation.getHyperny
            self.__colors[relation.getHypernym()] = '
            self.__graph.add_node(relation.getHyponym
            self.__colors[relation.getHyponym()] = '#

            self.__graph.add_edge(relation.getHyperny

    def show(self):
        pos = nx.spring_layout(self.__graph)
        plt.figure()
        colorMap = []
        for node in self.__graph.nodes:
            colorMap.append(self.__colors[node])
        nx.draw(self.__graph, pos, edge_color='black'
                node_size=500, node_color=colorMap, a
                labels={node: node for node in self._
        plt.axis('off')
```

```
        plt.show()
```

The logic is simple. We go through each relation, add the hypernym and hyponym as a node and add an edge between the 2. We also assign different colors for hypernym and hyponym nodes, so that we can easily visualize them.

The last file in our project is the one that puts everything together, the *build_knowledge_graph.py* file.

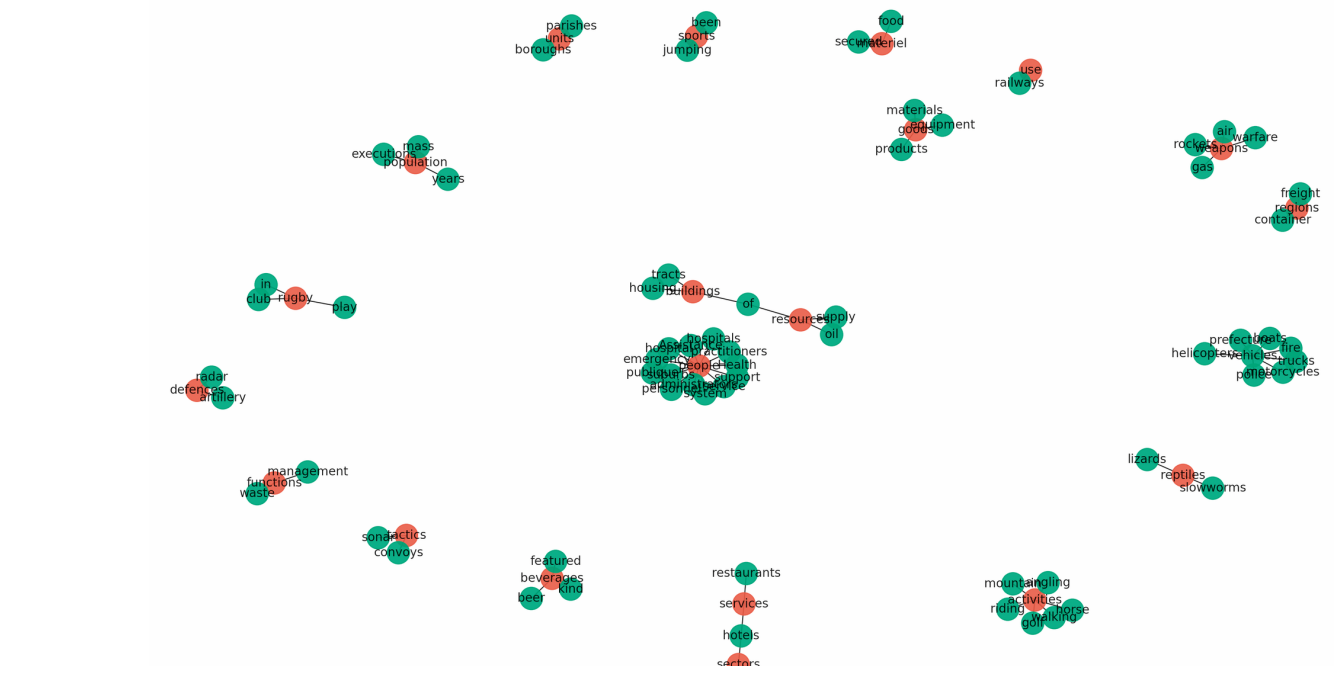
```
from text_extractor import TextExtractor
from and_other_pattern_matcher import AndOtherPattern
from such_as_pattern_matcher import SuchAsPatternMatc
from or_other_pattern_matcher import OrOtherPatternMa
from including_pattern_matcher import IncludingPatter
from especially_pattern_matcher import EspeciallyPatt
from text_extractor_pipe import TextExtractorPipe
from knowledge_graph import KnowledgeGraph
from matcher_pipe import MatcherPipe
import spacy

textExtractor1 = TextExtractor("WWII", "Q362")
textExtractor1.extract()
textExtractor2 = TextExtractor("London", "Q84")
textExtractor2.extract()
```

```
textExtractor3 = TextExtractor("Paris", "Q90")
textExtractor3.extract()
textExtractor4 = TextExtractor("World War I", "Q361")
textExtractor4.extract()
textExtractorPipe = TextExtractorPipe()
textExtractorPipe.addTextExtractor(textExtractor1)
textExtractorPipe.addTextExtractor(textExtractor2)
textExtractorPipe.addTextExtractor(textExtractor3)
textExtractorPipe.addTextExtractor(textExtractor4)



nlp = spacy.load('en_core_web_sm')
nlp.add_pipe(nlp.create_pipe('sentencizer')) # update
doc = nlp(textExtractorPipe.extract())

andOtherPatternMatcher = AndOtherPatternMatcher(nlp)
suchAsMatcher = SuchAsPatternMatcher(nlp)
orOtherMatcher = OrOtherPatternMatcher(nlp)

includingPatternMatcher = IncludingPatternMatcher(nlp
especiallyPatternMatcher = EspeciallyPatternMatcher(n
matcherPipe = MatcherPipe()
matcherPipe.addMatcher(andOtherPatternMatcher)
matcherPipe.addMatcher(suchAsMatcher)
matcherPipe.addMatcher(orOtherMatcher)
matcherPipe.addMatcher(includingPatternMatcher)
matcherPipe.addMatcher(especiallyPatternMatcher)
relations = matcherPipe.extract(doc)

for relation in relations:
```

```
    print (relation.getHypernym(), relation.getHypony

knowledgeGraph = KnowledgeGraph(relations)
knowledgeGraph.build()
knowledgeGraph.show()
```

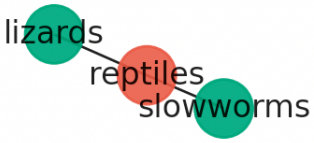
This finally builds our Knowledge Graph. The flow is simple: initialize text extractors, then initialize the pipe, initialize every matcher and the matcher pipe, run the pipe, print the results, build the knowledge graph, show the knowledge graph.

And here is our knowledge graph:

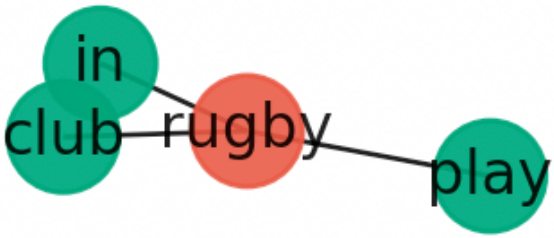


Our Knowledge Graph

There are quite a few clusters here, let's see some of our good results.



Hypernyms are in red, hyponyms are in green. We see they are correct and I quite happy with these results. But let's see some of our bad results also.



These are total failures, I'll need to take a look into this and see what's happening. It's clear though that the biggest defect of rule-based approaches is that they are limited, and there will always be exceptions that break your rule.

Conclusions

This was a long one! 😀 Thank you for reading until here, it was really fun for me to work on the project and I've learned a lot.

To summarize, we took a short look at what is Information Extraction, what a Knowledge Graph is, does and is used for, and then we saw how to use python and spaCy to build a knowledge graph. All the code for this article is uploaded on Github so you can check it out (please make sure to star the repository as it helps me know the code I write is helpful in any way).

Related articles

Throughout this article I've made some references to other articles on this blog, I'll also add them here for ease of reference, if you want to check them out.

- Python NLP Tutorial: Building A Knowledge Graph using Python and SpaCy
- Python Keywords Extraction - Machine Learning Project Series: Part 2
- Automated Python Keywords Extraction: TextRank vs Rake
- Python Named Entity Recognition - Machine Learning Project Series: Part 1

References

[1] Hearst, M., *Automatic Acquisition of Hyponyms From Large Text Corpora*. Link: https://www.aclweb.org/anthology/C92-2082.pdf

Corpora. Link: https://www.aclweb.org/anthology/C92-2082.pdf

Interested in more? Follow me on Twitter at @b_dmarius and I'll post there every new article.