Natural Language Processing is Fun!

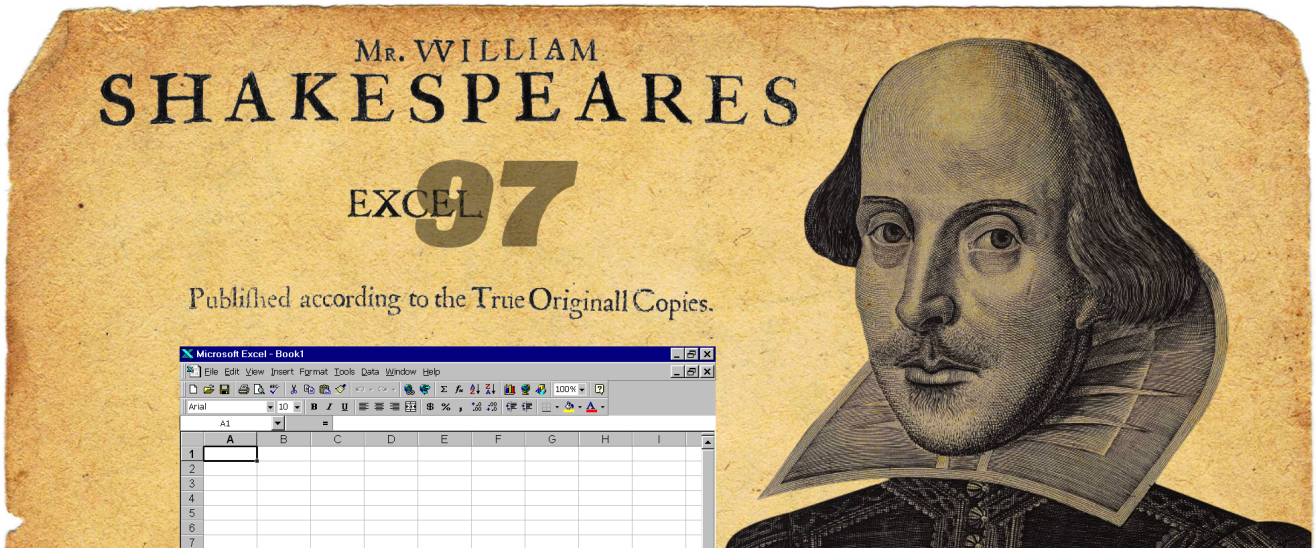
How computers understand Human Language

Adam Geitgey
Jul 18, 2018 · 16 min read

This article is part of an on-going series on NLP: Part 1, Part 2, Part 3. You can also read a reader-translated version of this article in 普通话.

***Giant update:** I've written a new book based on these articles! It not only expands and updates all my articles, but it has tons of brand new content and lots of hands-on coding projects. Check it out now!*

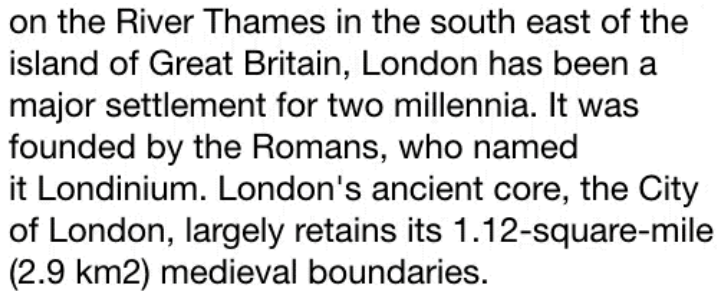
Computers are great at working with structured data like spreadsheets and database tables. But us humans usually communicate in words, not in tables. That's unfortunate for computers.



Unfortunately we don't live in this alternate version of history where all data is structured.

A lot of information in the world is unstructured — raw text in English or another human language. How can we get a computer to understand unstructured text and extract data from it?

London is the capital and most populous city
of England and the United Kingdom. Standing



Natural Language Processing, or *NLP*, is the sub-field of AI that is focused on enabling computers to understand and process human languages. Let's check out how NLP works and learn how to write programs that can extract information out of raw text using Python!

Note: If you don't care how NLP works and just want to cut and paste some code, skip way down to the section called "Coding the NLP Pipeline in Python".

Can Computers Understand Language?

As long as computers have been around, programmers have been trying to write programs that understand languages like English. The reason is pretty obvious — humans have been writing things down for thousands of years and it would be really helpful if a computer could read and understand all that data.

Computers can't yet truly understand English in the way that humans do — but they can already do a lot! In certain limited areas, what you can do with NLP already seems like magic. You might be able to save a lot of time by applying NLP techniques to your own projects.

And even better, the latest advances in NLP are easily accessible through open source Python libraries like spaCy, textacy, and neuralcoref. What you can do with just a few lines of python is amazing.

Extracting Meaning from Text is Hard

The process of reading and understanding English is very complex — and that's not even considering that English doesn't follow logical and consistent rules. For example, what does this news headline mean?

> *"Environmental regulators grill business owner over illegal coal fires."*

Are the regulators questioning a business owner about burning coal illegally? Or are the regulators literally cooking the business owner? As you can see, parsing English with a computer is going to be complicated.

Doing anything complicated in machine learning usually means *building a pipeline*. The idea is to break up your problem into very small pieces and then use machine learning to solve each smaller piece separately. Then by chaining together several machine learning models that feed into each other, you can do very complicated things.

And that's exactly the strategy we are going to use for NLP. We'll break down the process of understanding English into small chunks and see how each one works.

Building an NLP Pipeline, Step-by-Step

Let's look at a piece of text from Wikipedia:

> *London is the capital and most populous city of England and the United Kingdom. Standing on the River Thames in the south east of the island of Great Britain, London has been a major settlement for two millennia. It was founded by the Romans, who named it Londinium.*
>
> *(Source: Wikipedia article "London")*

This paragraph contains several useful facts. It would be great if a computer could read this text and understand that London is a city, London is located in England, London was settled by Romans and so on. But to get there, we have to first teach our computer the most basic concepts of written language and then move up from there.

Step 1: Sentence Segmentation

The first step in the pipeline is to break the text apart into separate sentences. That gives us this:

1. "London is the capital and most populous city of England and the United Kingdom."
2. "Standing on the River Thames in the south east of the island of Great Britain, London has been a major settlement for two millennia."
3. "It was founded by the Romans, who named it Londinium."

We can assume that each sentence in English is a separate thought or idea. It will be a lot easier to write a program to understand a single sentence than to understand a whole paragraph.

Coding a Sentence Segmentation model can be as simple as splitting apart sentences whenever you see a punctuation mark. But modern NLP pipelines often use more complex techniques that work even when a document isn't formatted cleanly.

Step 2: Word Tokenization

Now that we've split our document into sentences, we can process them one at a time. Let's start with the first sentence from our document:

> *"London is the capital and most populous city of England and the United Kingdom."*

The next step in our pipeline is to break this sentence into separate words or *tokens*. This is called *tokenization*. This is the result:

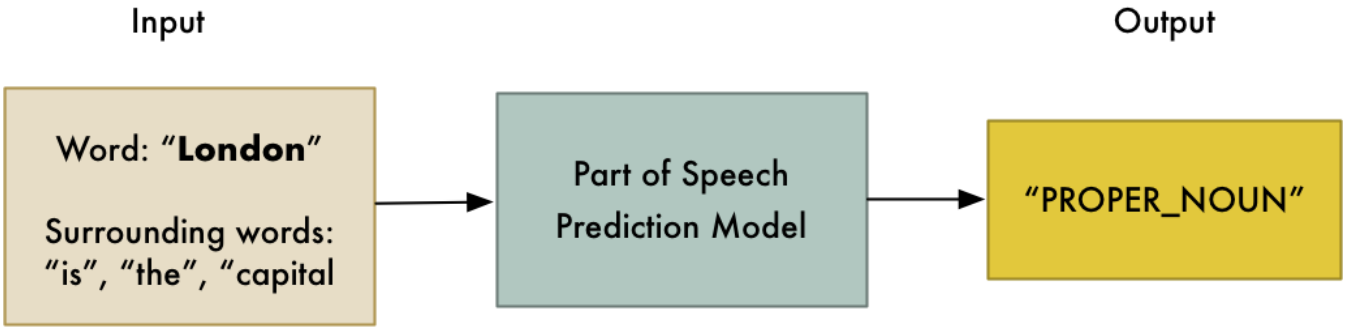
> *"London", "is", "the", "capital", "and", "most", "populous", "city", "of", "England", "and", "the", "United", "Kingdom", "."*

Tokenization is easy to do in English. We'll just split apart words whenever there's a space between them. And we'll also treat punctuation marks as separate tokens since punctuation also has meaning.

Step 3: Predicting Parts of Speech for Each Token

Next, we'll look at each token and try to guess its part of speech — whether it is a noun, a verb, an adjective and so on. Knowing the role of each word in the sentence will help us start to figure out what the sentence is talking about.

We can do this by feeding each word (and some extra words around it for context) into a pre-trained part-of-speech classification model:



The part-of-speech model was originally trained by feeding it millions of English sentences with each word's part of speech already tagged and having it learn to replicate that behavior.

Keep in mind that the model is completely based on statistics — it doesn't actually understand what the words mean in the same way that humans do. It just knows how to guess a part of speech based on similar sentences and words it has seen before.

After processing the whole sentence, we'll have a result like this:

With this information, we can already start to glean some very basic meaning. For example, we can see that the nouns in the sentence include "London" and "capital", so the sentence is probably talking about London.

Step 4: Text Lemmatization

In English (and most languages), words appear in different forms. Look at these two sentences:

I had a **pony**.

I had two **ponies**.

Both sentences talk about the noun **pony,** but they are using different inflections. When working with text in a computer, it is helpful to know the base form of each word so that you know that both sentences are talking about the same concept. Otherwise the strings "pony" and "ponies" look like two totally different words to a computer.

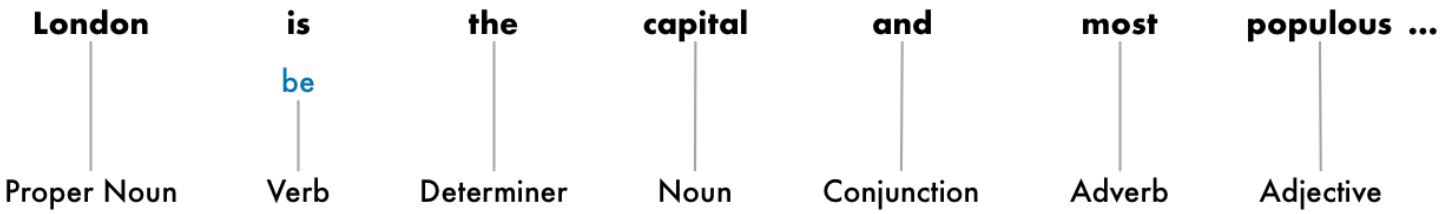
In NLP, we call finding this process *lemmatization* — figuring out the most basic form or *lemma* of each word in the sentence.

The same thing applies to verbs. We can also lemmatize verbs by finding their root, unconjugated form. So "**I had two ponies**" becomes "**I [have] two [pony].**"

Lemmatization is typically done by having a look-up table of the lemma forms of words based on their part of speech and possibly having some custom rules to handle words

that you've never seen before.

Here's what our sentence looks like after lemmatization adds in the root form of our verb:

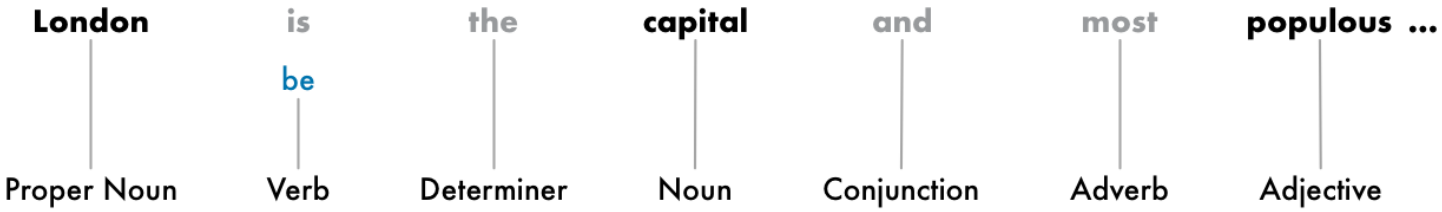


The only change we made was turning "is" into "be".

Step 5: Identifying Stop Words

Next, we want to consider the importance of a each word in the sentence. English has a lot of filler words that appear very frequently like "and", "the", and "a". When doing statistics on text, these words introduce a lot of noise since they appear way more frequently than other words. Some NLP pipelines will flag them as **stop words**—that is, words that you might want to filter out before doing any statistical analysis.

Here's how our sentence looks with the stop words grayed out:



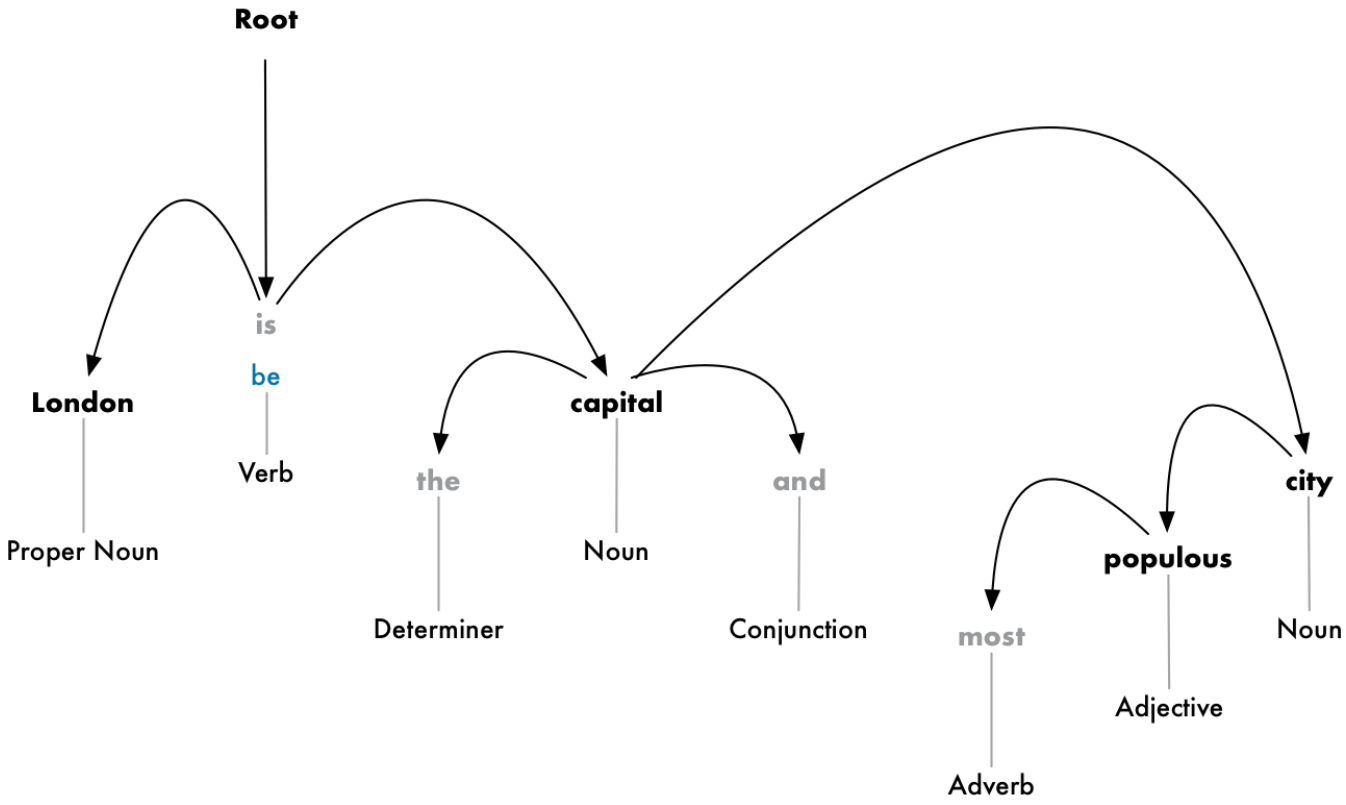
Stop words are usually identified by just by checking a hardcoded list of known stop words. But there's no standard list of stop words that is appropriate for all applications. The list of words to ignore can vary depending on your application.

For example if you are building a rock band search engine, you want to make sure you don't ignore the word "The". Because not only does the word "The" appear in a lot of band names, there's a famous 1980's rock band called *The The*!

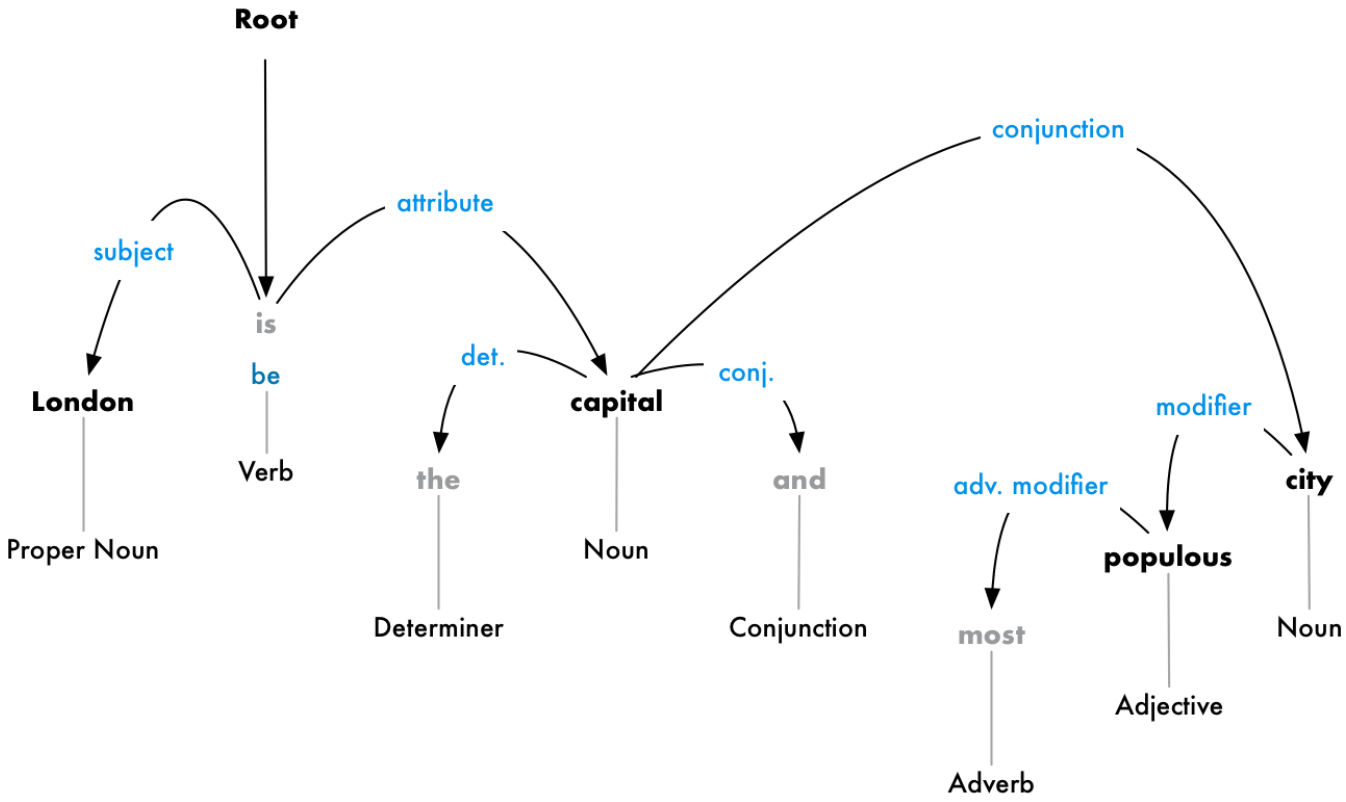
Step 6: Dependency Parsing

The next step is to figure out how all the words in our sentence relate to each other. This is called *dependency parsing*.

The goal is to build a tree that assigns a single **parent** word to each word in the sentence. The root of the tree will be the main verb in the sentence. Here's what the beginning of the parse tree will look like for our sentence:



But we can go one step further. In addition to identifying the parent word of each word, we can also predict the type of relationship that exists between those two words:



This parse tree shows us that the subject of the sentence is the noun "*London*" and it has a "*be*" relationship with "*capital*". We finally know something useful — *London* is a *capital*! And if we followed the complete parse tree for the sentence (beyond what is shown), we would even found out that London is the capital of the *United Kingdom*.

Just like how we predicted parts of speech earlier using a machine learning model, dependency parsing also works by feeding words into a machine learning model and outputting a result. But parsing word dependencies is particularly complex task and would require an entire article to explain in any detail. If you are curious how it works, a great place to start reading is Matthew Honnibal's excellent article *"Parsing English in 500 Lines of Python"*.

But despite a note from the author in 2015 saying that this approach is now standard, it's actually out of date and not even used by the author anymore. In 2016, Google released a new dependency parser called *Parsey McParseface* which outperformed previous benchmarks using a new deep learning approach which quickly spread throughout the industry. Then a year later, they released an even newer model called *ParseySaurus* which improved things further. In other words, parsing techniques are still an active area of research and constantly changing and improving.

It's also important to remember that many English sentences are ambiguous and just really hard to parse. In those cases, the model will make a guess based on what parsed version of the sentence seems most likely but it's not perfect and sometimes the model will be embarrassingly wrong. But over time our NLP models will continue to get better at parsing text in a sensible way.

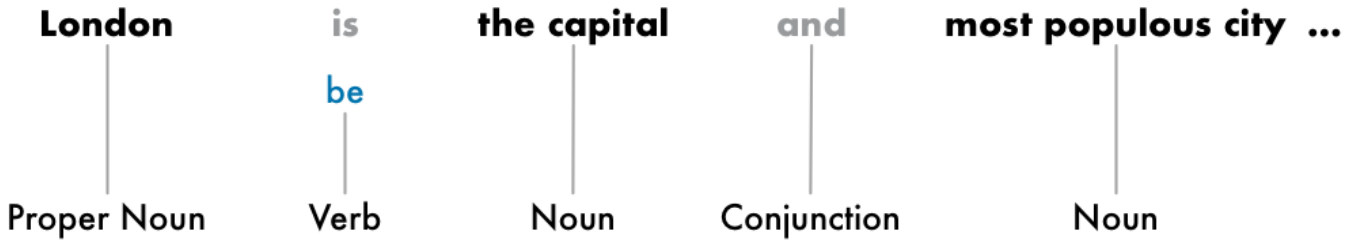
Want to try out dependency parsing on your own sentence? There's a great interactive demo from the spaCy team here.

Step 6b: Finding Noun Phrases

So far, we've treated every word in our sentence as a separate entity. But sometimes it makes more sense to group together the words that represent a single idea or thing. We can use the information from the dependency parse tree to automatically group together words that are all talking about the same thing.

For example, instead of this:

We can group the noun phrases to generate this:



Whether or not we do this step depends on our end goal. But it's often a quick and easy way to simplify the sentence if we don't need extra detail about which words are adjectives and instead care more about extracting complete ideas.

Step 7: Named Entity Recognition (NER)

Now that we've done all that hard work, we can finally move beyond grade-school grammar and start actually extracting ideas.

In our sentence, we have the following nouns:

London is the **capital** and most populous **city** of **England** and the **United Kingdom**.

Some of these nouns present real things in the world. For example, *"London"*, *"England"* and *"United Kingdom"* represent physical places on a map. It would be nice to be able to detect that! With that information, we could automatically extract a list of real-world places mentioned in a document using NLP.

The goal of *Named Entity Recognition*, or *NER*, is to detect and label these nouns with the real-world concepts that they represent. Here's what our sentence looks like after running each token through our NER tagging model:

But NER systems aren't just doing a simple dictionary lookup. Instead, they are using the context of how a word appears in the sentence and a statistical model to guess which type of noun a word represents. A good NER system can tell the difference between "*Brooklyn Decker*" the person and the place "*Brooklyn*" using context clues.

Here are just some of the kinds of objects that a typical NER system can tag:

- People's names
- Company names
- Geographic locations (Both physical and political)
- Product names
- Dates and times
- Amounts of money
- Names of events

NER has tons of uses since it makes it so easy to grab structured data out of text. It's one of the easiest ways to quickly get value out of an NLP pipeline.

Want to try out Named Entity Recognition yourself? There's another great interactive demo from spaCy here.

Step 8: Coreference Resolution

At this point, we already have a useful representation of our sentence. We know the parts of speech for each word, how the words relate to each other and which words are talking about named entities.

However, we still have one big problem. English is full of pronouns — words like *he*, *she*, and *it*. These are shortcuts that we use instead of writing out names over and over in each sentence. Humans can keep track of what these words represent based on context. But our NLP model doesn't know what pronouns mean because it only examines one sentence at a time.

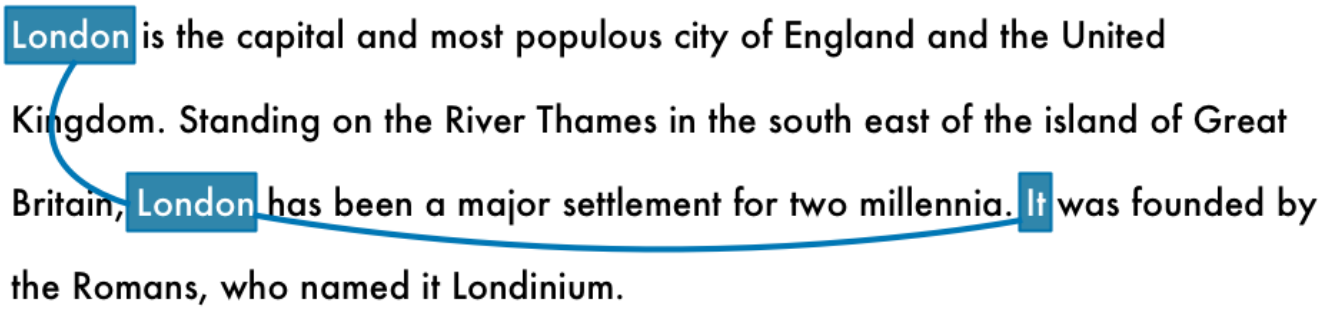
Let's look at the third sentence in our document:

> *"It was founded by the Romans, who named it Londinium."*

If we parse this with our NLP pipeline, we'll know that "it" was founded by Romans. But it's a lot more useful to know that "London" was founded by Romans.

As a human reading this sentence, you can easily figure out that "*it*" means "*London*". The goal of coreference resolution is to figure out this same mapping by tracking pronouns across sentences. We want to figure out all the words that are referring to the same entity.

Here's the result of running coreference resolution on our document for the word "London":



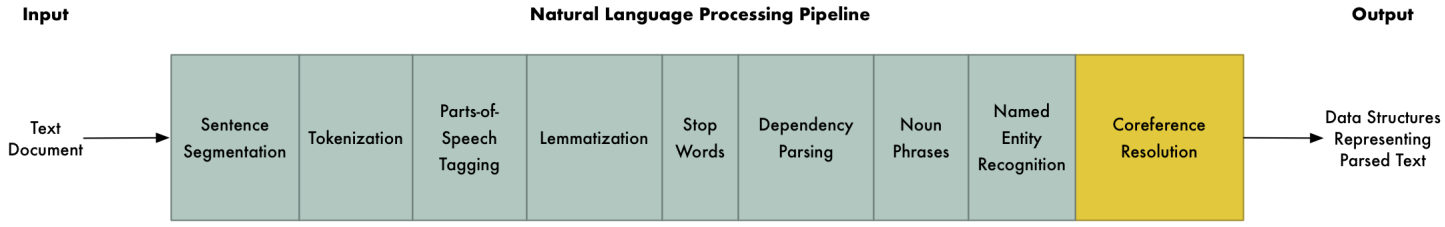
With coreference information combined with the parse tree and named entity information, we should be able to extract a lot of information out of this document!

Coreference resolution is one of the most difficult steps in our pipeline to implement. It's even more difficult than sentence parsing. Recent advances in deep learning have resulted in new approaches that are more accurate, but it isn't perfect yet. If you want to learn more about how it works, start here.

Want to play with co-reference resolution? Check out this great co-reference resolution demo from Hugging Face.

Coding the NLP Pipeline in Python

Here's an overview of our complete NLP pipeline:



Coreference resolution is an optional step that isn't always done.

Whew, that's a lot of steps!

Note: Before we continue, it's worth mentioning that these are the steps in a typical NLP pipeline, but you will skip steps or re-order steps depending on what you want to do and how your NLP library is implemented. For example, some libraries like spaCy do sentence segmentation much later in the pipeline using the results of the dependency parse.

So how do we code this pipeline? Thanks to amazing python libraries like spaCy, it's already done! The steps are all coded and ready for you to use.

First, assuming you have Python 3 installed already, you can install spaCy like this:

```
# Install spaCy
pip3 install -U spacy

# Download the large English model for spaCy
python3 -m spacy download en_core_web_lg

# Install textacy which will also be useful
pip3 install -U textacy
```

Install spacy.txt hosted with ♡ by GitHub view raw

Then the code to run an NLP pipeline on a piece of text looks like this:

```
import spacy

# Load the large English NLP model
nlp = spacy.load('en_core_web_lg')

# The text we want to examine
text = """London is the capital and most populous city of England and
the United Kingdom.  Standing on the River Thames in the south east
of the island of Great Britain, London has been a major settlement
for two millennia. It was founded by the Romans, who named it Londinium.
"""

# Parse the text with spaCy. This runs the entire pipeline.
doc = nlp(text)

# 'doc' now contains a parsed version of text. We can use it to do anything we want!
# For example, this will print out all the named entities that were detected:
```

```
for entity in doc.ents:
    print(f"{entity.text} ({entity.label_})")
```

demo.py hosted with ♡ by GitHub view raw

If you run that, you'll get a list of named entities and entity types detected in our document:

```
London (GPE)
England (GPE)
the United Kingdom (GPE)
the River Thames (FAC)
Great Britain (GPE)
London (GPE)
two millennia (DATE)
Romans (NORP)
Londinium (PERSON)
```

You can look up what each of those entity codes means here.

Notice that it makes a mistake on "Londinium" and thinks it is the name of a person instead of a place. This is probably because there was nothing in the training data set similar to that and it made a best guess. Named Entity Detection often requires a little bit of model fine tuning if you are parsing text that has unique or specialized terms like this.

Let's take the idea of detecting entities and twist it around to build a data scrubber. Let's say you are trying to comply with the new GDPR privacy regulations and you've discovered that you have thousands of documents with personally identifiable information in them like people's names. You've been given the task of removing any and all names from your documents.

Going through thousands of documents and trying to redact all the names by hand could take years. But with NLP, it's a breeze. Here's a simple scrubber that removes all the names it detects:

```
import spacy

# Load the large English NLP model
nlp = spacy.load('en_core_web_lg')

# Replace a token with "REDACTED" if it is a name
def replace_name_with_placeholder(token):
```

```
    if token.ent_iob != 0 and token.ent_type_ == "PERSON":
        return "[REDACTED] "
    else:
        return token.string

# Loop through all the entities in a document and check if they are names
def scrub(text):
    doc = nlp(text)
    for ent in doc.ents:
        ent.merge()
    tokens = map(replace_name_with_placeholder, doc)
    return "".join(tokens)

s = """
In 1950, Alan Turing published his famous article "Computing Machinery and Intelligence". In 19
Syntactic Structures revolutionized Linguistics with 'universal grammar', a rule based system c
"""

print(scrub(s))
```

And if you run that, you'll see that it works as expected:

```
In 1950, [REDACTED] published his famous article "Computing
Machinery and Intelligence". In 1957, [REDACTED]
Syntactic Structures revolutionized Linguistics with 'universal
grammar', a rule based system of syntactic structures.
```

Extracting Facts

What you can do with spaCy right out of the box is pretty amazing. But you can also use the parsed output from spaCy as the input to more complex data extraction algorithms. There's a python library called textacy that implements several common data extraction algorithms on top of spaCy. It's a great starting point.

One of the algorithms it implements is called Semi-structured Statement Extraction. We can use it to search the parse tree for simple statements where the subject is "London" and the verb is a form of "be". That should help us find facts about London.

Here's how that looks in code:

```
import spacy
import textacy.extract
```

```
import textacy.extract

# Load the large English NLP model
nlp = spacy.load('en_core_web_lg')

# The text we want to examine
text = """London is the capital and most populous city of England and  the United Kingdom.
Standing on the River Thames in the south east of the island of Great Britain,
London has been a major settlement  for two millennia.  It was founded by the Romans,
who named it Londinium.
"""

# Parse the document with spaCy
doc = nlp(text)

# Extract semi-structured statements
statements = textacy.extract.semistructured_statements(doc, "London")

# Print the results
print("Here are the things I know about London:")

for statement in statements:
    subject, verb, fact = statement
    print(f" - {fact}")
```

fact-extraction.py hosted with ♡ by GitHub view raw

And here's what it prints:

```
Here are the things I know about London:

 - the capital and most populous city of England and the United
Kingdom.
- a major settlement for two millennia.
```

Maybe that's not too impressive. But if you run that same code on the entire London wikipedia article text instead of just three sentences, you'll get this more impressive result:

```
Here are the things I know about London:

 - the capital and most populous city of England and the United
Kingdom
 - a major settlement for two millennia
 - the world's most populous city from around 1831 to 1925
 - beyond all comparison the largest town in England
```

```
 - still very compact
 - the world's largest city from about 1831 to 1925
 - the seat of the Government of the United Kingdom
 - vulnerable to flooding
 - "one of the World's Greenest Cities" with more than 40 percent
green space or open water
 - the most populous city and metropolitan area of the European
Union and the second most populous in Europe
 - the 19th largest city and the 18th largest metropolitan region in
the world
 - Christian, and has a large number of churches, particularly in
the City of London
 - also home to sizeable Muslim, Hindu, Sikh, and Jewish communities
 - also home to 42 Hindu temples
 - the world's most expensive office market for the last three years
according to world property journal (2015) report
 - one of the pre-eminent financial centres of the world as the most
important location for international finance
 - the world top city destination as ranked by TripAdvisor users
 - a major international air transport hub with the busiest city
airspace in the world
 - the centre of the National Rail network, with 70 percent of rail
journeys starting or ending in London
 - a major global centre of higher education teaching and research
and has the largest concentration of higher education institutes in
Europe
 - home to designers Vivienne Westwood, Galliano, Stella McCartney,
Manolo Blahnik, and Jimmy Choo, among others
 - the setting for many works of literature
 - a major centre for television production, with studios including
BBC Television Centre, The Fountain Studios and The London Studios
 - also a centre for urban music
 - the "greenest city" in Europe with 35,000 acres of public parks,
woodlands and gardens
 - not the capital of England, as England does not have its own
government
```

Now things are getting interesting! That's a pretty impressive amount of information we've collected automatically.

For extra credit, try installing the neuralcoref library and adding Coreference Resolution to your pipeline. That will get you a few more facts since it will catch sentences that talk about "it" instead of mentioning "London" directly.

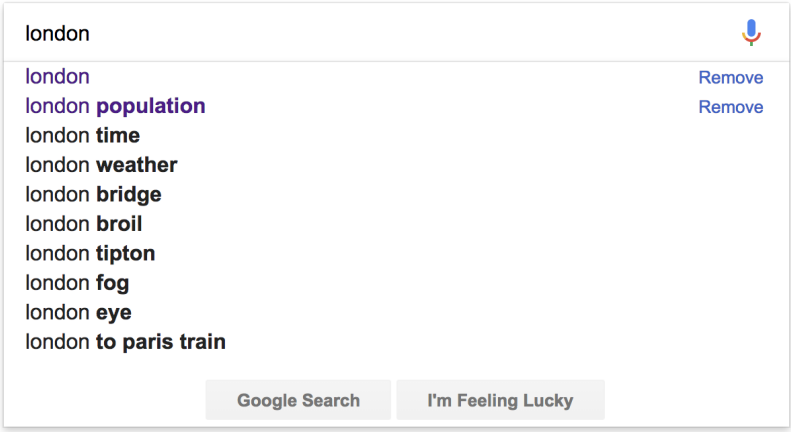
What else can we do?

By looking through the spaCy docs and textacy docs, you'll see lots of examples of the ways you can work with parsed text. What we've seen so far is just a tiny sample.

Here's another practical example: Imagine that you were building a website that let's the user view information for every city in the world using the information we

extracted in the last example.

If you had a search feature on the website, it might be nice to autocomplete common search queries like Google does:



Google's autocomplete suggestions for "London"

But to do this, we need a list of possible completions to suggest to the user. We can use NLP to quickly generate this data.

Here's one way to extract frequently-mentioned noun chunks from a document:

```
import spacy
import textacy.extract

# Load the large English NLP model
nlp = spacy.load('en_core_web_lg')

# The text we want to examine
text = """London is [.. shortened for space ..]"""

# Parse the document with spaCy
doc = nlp(text)

# Extract noun chunks that appear
noun_chunks = textacy.extract.noun_chunks(doc, min_freq=3)

# Convert noun chunks to lowercase strings
noun_chunks = map(str, noun_chunks)
noun_chunks = map(str.lower, noun_chunks)

# Print out any nouns that are at least 2 words long
for noun_chunk in set(noun_chunks):
    if len(noun_chunk.split(" ")) > 1:
```

```
23          print(noun_chunk)
```

key_terms.py hosted with ♡ by GitHub view raw

If you run that on the London Wikipedia article, you'll get output like this:

```
westminster abbey
natural history museum
west end
east end
st paul's cathedral
royal albert hall
london underground
great fire
british museum
london eye

.... etc ....
```

Going Deeper

This is just a tiny taste of what you can do with NLP. In future posts, we'll talk about other applications of NLP like Text Classification and how systems like Amazon Alexa parse questions.

But until then, install spaCy and start playing around! Or if you aren't a Python user and end up using a different NLP library, the ideas should all work roughly the same way.

This article is part of an on-going series on NLP. You can continue on to Part 2.

. . .

If you liked this article, consider signing up for my Machine Learning is Fun! newsletter:

You can also follow me on Twitter at @ageitgey, email me directly or find me on linkedin. I'd love to hear from you if I can help you or your team with machine learning.

Machine Learning NLP Naturallanguageprocessing