IST664 : Introduction to spaCy

Edited by Jeff Stanton and Preeti Jagadev

SpaCy is an open source natural language processing library written by Matthew Honnibal and Ines Montani. Most of spaCy is written natively in Python. Unlike NLTK, which was designed for teaching and research, spaCy was created from the start to support production applications - real world activities that require natural language processing. SpaCy uses a "pipeline" metaphor such that input documents and data go through a variety of typical processing stages where each stage feeds into the next one. Examples of these stages include tokenization, part of speech tagging, named entity recognition, and transformation into word vectors.

Try searching for "spaCy" on Kaggle.com. At this writing there were more than 4600 projects that used spaCy. Part of the appeal is that spaCy makes it easy to get started with a project. SpaCy contains support for dozens of different languages and its integration with word- and sentence-embedding approaches provides access to the advantages of pre-trained deep learning models.

In this lab you will get a more comprehensive view of the architecture and capabilities of spaCy.

Sections of this lab:

- Basics: Getting Started
- Lemmatization
- Token Extracting / Removing / Transforming
- Sentence Segmentation
- Part of Speech Tagging
- Named Entity Recognition
- Dependency Parsing
- Word Vectors
- Sentence Similarity
- Customizing pipeline components

Basics: Getting Started

```
# Every spaCy project begins with importing the package and
# instantiating a processing object that is initialized with a particular
# language model. In this case we will start with a small English pipeline
# trained from text harvested from the web.
import spacy
nlp = spacy.load("en_core_web_sm")

# That's equivalent to:
# import en_core_web_sm
# nlp = en_core_web_sm.load()

type(nlp)
```

```
spacy.lang.en.English
def __call__(text: Union[str, Doc], *, disable: Iterable[str]=SimpleFrozenList(), component_cfg:
Optional[Dict[str, Dict[str, Any]]]=None) -> Doc

/usr/local/lib/python3.11/dist-packages/spacy/lang/en/__init__.py
A text-processing pipeline. Usually you'll load this once per process,
and pass the instance around your application.

Defaults (class): Settings, data and factory methods for creating the `nlp`
    object and processing pipeline.
```

```
# There are lots of things this object can do. Let's use
# dir to get a list of them:

[m for m in dir(nlp) if m[0] != "_"]
```

```
['Defaults',
 'add_pipe',
 'analyze_pipes',
 'batch_size',
 'begin_training',
 'component',
```

```
 'component_names',
 'components',
 'config',
 'create_optimizer',
 'create_pipe',
 'create_pipe_from_source',
 'default_config',
 'default_error_handler',
 'disable_pipe',
 'disable_pipes',
 'disabled',
 'enable_pipe',
 'evaluate',
 'factories',
 'factory',
 'factory_names',
 'from_bytes',
 'from_config',
 'from_disk',
 'get_factory_meta',
 'get_factory_name',
 'get_pipe',
 'get_pipe_config',
 'get_pipe_meta',
 'has_factory',
 'has_pipe',
 'initialize',
 'lang',
 'make_doc',
 'max_length',
 'memory_zone',
 'meta',
 'path',
 'pipe',
 'pipe_factories',
 'pipe_labels',
 'pipe_names',
 'pipeline',
 'rehearse',
 'remove_pipe',
 'rename_pipe',
 'replace_listeners',
 'replace_pipe',
 'resume_training',
 'select_pipes',
 'set_error_handler',
 'set_factory_meta',
 'to_bytes',
 'to_disk',
 'tokenizer',
 'update',
 'use_params',
```

Note that all of the methods shown above pertain to pipelines - a modular sequence of processing steps. In general these follow a standard order:

- Tokenizer
- Part of speech tagger
- Dependency parser (organizes each sentence into its constituent parts
- Named entity recognizer
- Lemmatizer
- Additional elements (including document classification)

You know enough of the essential foundations of NLP to know why these pipeline elements appear in this order. For example, you could not apply part of speech tags to words without first tokenizing the raw text.

```
# At the most basic level, and at the beginning of most
# NLP pipelines, we tokenize a document:
doc = nlp("Hello World!") # This is the most basic way to use the instance
type(doc), len(doc) # What is the result?
#Output is 3 - Maybe for Hello, World !?
```

```
(spacy.tokens.doc.Doc, 3)
```

```
# A spaCy "tokens-doc" behaves like a list, such that
# we can use a list comprehension to access the individual
# tokens in the document:
[token.text for token in doc]
```

```
['Hello', 'World', '!']
```

```
# And because it behaves like a list, we can also use
# slicing to get access to the individual tokens.
first_token = doc[0] # Slice the first token
print(type(first_token)) # What is its type?
print(first_token.text) # Show the text of the token
```

```
<class 'spacy.tokens.token.Token'>
Hello
```

```
# In spaCy terminology, a span is any contiguous set of tokens.
# Spans are often used to break up a document into sentences. Here
# we are just using slicing to create a span with the first two
# of our three tokens.
span = doc[0:2]
[token.text for token in span]
```

```
['Hello', 'World']
```

```
# For this first exercise, tokenize a longer text excerpted from Wikipedia.
# Use slicing to show the first five tokens:

longtext = """A neural network is either a biological neural network or an
artificial neural network for solving artificial intelligence (AI) problems.
The connections of the biological neuron are modeled as weights. A positive
weight reflects an excitatory connection, while negative values mean
inhibitory connections."""

# 5.1: Tokenize longtext
doc = nlp(longtext)

# 5.2: Display the *texts* of tokens in a span consisting of the first 5 tokens
print([token.text for token in doc[0:5]])

# 5.2a: (Challenge) Use Python slicing notation to show the *last* 5 tokens
print([token.text for token in doc[-5:]])

```

```
['A', 'neural', 'network', 'is', 'either']
['mean', '\n', 'inhibitory', 'connections', '.']
```

```
# SpaCy uses the language model to make better tokenization decisions. Let's
# compare spaCy tokenization with the primitive use of split(). Remember that
# split() defaults to splitting on spaces.
headline = "Rare Bird’s Detection Highlights Promise of ‘Environmental DNA’"

splitspacy = nlp(headline) # Use spaCy tokenization
splitspace = headline.split() # Use simple splitting on spaces

print("SpaCy tokens:")
print([t.text for t in splitspacy])
print(len(splitspacy), "tokens.")

print("\nSimple splitting:")
print([s for s in splitspace])
print(len(splitspace), "tokens.")
```

```
SpaCy tokens:
['Rare', 'Bird', '’s', 'Detection', 'Highlights', 'Promise', 'of', '‘', 'Environmental', 'DNA', '’']
11 tokens.

Simple splitting:
['Rare', 'Bird’s', 'Detection', 'Highlights', 'Promise', 'of', '‘Environmental', 'DNA’']
8 tokens.
```

```
# Why do you think it might be helpful to tokenize the possessive "Bird's" into
# two tokens? Add a comment that explains your reasoning. Then find or write

#Maybe to show Punctuation or the lack thereof - Wanting to Breakout the Root Words?

# a new sentence that contains a hyphenated noun phrase. How does spaCy treat that?
headline = "I really love Mac-and-Cheese from Chick-Fil-A"

```

```
# 5.2b: Use spaCy to tokenize a sentence that contains a hyphenated phrase.
splitspacy = nlp(headline)

print("SpaCy tokens:")
print([t.text for t in splitspacy])
print(len(splitspacy), "tokens.")

```

```
SpaCy tokens:
['I', 'really', 'love', 'Mac', '-', 'and', '-', 'Cheese', 'from', 'Chick', '-', 'Fil', '-', 'A']
14 tokens.
```

Lemmatization

Lemmatization is the process of reducing inflected forms, sometimes derivationally related forms of a word to a common base form. This reduced form or root word is called a lemma. Lemmas have an advantage over simple stemming: Lemmas are always dictionary words. Lemmatizing can be a valuable data reduction technique because it aggregates various inflective forms of a word down to a single root.

```
# Demonstrate spaCy lemmatization with a verb form
text = "I am, you are, and he is." # All the verbs are variations on the verb "to be"

# Note that the underscore following the attribute name in
# the expression token.lemma_ provides the human readable form of the attribute.
[token.lemma_ for token in nlp(text)]
```

```
['I', 'be', ',', 'you', 'be', ',', 'and', 'he', 'be', '.']
```

```
# Look at the non-text form of the lemma:

# 5.3: use token.lemma instead of token.lemma_
text = "I am, you are, and he is." # All the verbs are variations on the verb "to be"
[token.lemma for token in nlp(text)]

# Write a comment describing what you see. These values are
# ID numbers for spaCy's "StringStore." More information here:
# https://spacy.io/usage/spacy-101#vocab
#Looks like the token.lemma somehow converted the Words to Assigned Numbers? IE you could input these numbers and convert them back to t
```

```
[4690420944186131903,
 10382539506755952630,
 2593208677638477497,
 7624161793554793053,
 10382539506755952630,
 2593208677638477497,
 2283656566040971221,
 1655312771067108281,
 10382539506755952630,
 12646065887601541794]
```

```
# Here's another example
text = "Look! It looks like he looked."
doc = nlp(text)
for token in doc:
    print("token:{} -> lemma:{}".format(token.text,token.lemma_ ))
```

```
token:Look -> lemma:look
token:! -> lemma:!
token:It -> lemma:it
token:looks -> lemma:look
token:like -> lemma:like
token:he -> lemma:he
token:looked -> lemma:look
token:. -> lemma:.
```

```
# Add your own example, this time using different forms of a noun
text = "The Geese and Mice like Mangos"
# 5.4: Lemmatize two or more inflective forms of a noun. What about
# irregular inflections (such as the plural of mouse)?
doc = nlp(text)
for token in doc:
    print("token:{} -> lemma:{}".format(token.text,token.lemma_ ))
```

```
token:The -> lemma:the
token:Geese -> lemma:geese
token:and -> lemma:and
token:Mice -> lemma:mouse
token:like -> lemma:like
token:Mangos -> lemma:mangos
```

Token Extracting / Removing / Transforming

Here's an overview of all of the bound methods and attributes that a token has. When creating an NLP pipeline, it is helpful not to have to write our own code to find out these things.

Attribute Name	Type	Description
lemma	int	Base form of the token, with no inflectional suffixes.
lemma_	unicode	Base form of the token, with no inflectional suffixes.
norm	int	The token's norm, i.e. a normalized form of the token text. Usually set in the language's tokenizer exceptions or norm exceptions.
norm_	unicode	The token's norm, i.e. a normalized form of the token text. Usually set in the language's tokenizer exceptions or norm exceptions.
lower	int	Lowercase form of the token.
lower_	unicode	Lowercase form of the token text. Equivalent to Token.text.lower().
shape	int	Transform of the tokens's string, to show orthographic features. For example, "Xxxx" or "dd".
shape_	unicode	Transform of the tokens's string, to show orthographic features. For example, "Xxxx" or "dd".
prefix	int	Hash value of a length-N substring from the start of the token. Defaults to N=1.
prefix_	unicode	A length-N substring from the start of the token. Defaults to N=1.
suffix	int	Hash value of a length-N substring from the end of the token. Defaults to N=3.
suffix_	unicode	Length-N substring from the end of the token. Defaults to N=3.
is_alpha	bool	Does the token consist of alphabetic characters? Equivalent to token.text.isalpha().
is_ascii	bool	Does the token consist of ASCII characters? Equivalent to all(ord(c) < 128 for c in token.text).
is_digit	bool	Does the token consist of digits? Equivalent to token.text.isdigit().
is_lower	bool	Is the token in lowercase? Equivalent to token.text.islower().
is_upper	bool	Is the token in uppercase? Equivalent to token.text.isupper().
is_title	bool	Is the token in titlecase? Equivalent to token.text.istitle().
is_punct	bool	Is the token punctuation?
is_left_punct	bool	Is the token a left punctuation mark, e.g. (?
is_right_punct	bool	Is the token a right punctuation mark, e.g.)?
is_space	bool	Does the token consist of whitespace characters? Equivalent to token.text.isspace().
is_bracket	bool	Is the token a bracket?
is_quote	bool	Is the token a quotation mark?
is_currency V2.0.8	bool	Is the token a currency symbol?
like_url	bool	Does the token resemble a URL?
like_num	bool	Does the token represent a number? e.g. "10.9", "10", "ten", etc.
like_email	bool	Does the token resemble an email address?

Extracting

The list of attributes that spaCy makes available on the token object provide a variety of type tests. The is_ attributes allow testing for alpahnumeric, uppercase, title case, left punctuation mark, right punctuation mark, any punctuation mark, a bracket, a quote mark or a currency symbol. These are all very helpful in navigating within a string of tokens: Later we will show a search capability that allows us to include these in pattern matching.

There are also three "like" attributes that show if a token looks like a web address, a numeric string, or an email address.

Let's run some tests on a long and complex sentence from Wikipedia:

```
text='''An information retrieval technique using latent semantic structure was
patented in 1988 (US Patent 4,839,853, now expired) by Scott Deerwester,
Susan Dumais, George Furnas, Richard Harshman, Thomas Landauer, Karen Lochbaum
and Lynn Streeter. In the context of its application to information retrieval,
it is sometimes called latent semantic indexing (LSI).'''
```

```
my_list=[] # Initialize a blank list
doc = nlp(text) # Tokenize the text string
for token in doc: # Check each token
    if token.is_punct: # Run the bound method
```

```
        my_list.append(token) # Append to the list

for item in my_list: # Review each item in the list
    print(item) # Print the item
```

```
(
,
)
,
,
,
,
,
.
,
(
)
.
```

```
# Now do something similar but use a list comprehension
[tok for tok in doc if tok.is_left_punct]
```

```
[(, (]
```

```
# Add a line of code to display right punctuation
[tok for tok in doc if tok.is_right_punct]
# 5.5: Use the bound method to detect and print right puncutation
bound_method = [tok for tok in doc if tok.is_right_punct]
print(bound_method)
```

```
[), )]
```

```
# Add a line of code to detect tokens that seem like numbers
num_tok = [tok for tok in doc if tok.like_num]
# 5.6: Use the bound method to detect and display numbers
bound_method = [tok for tok in doc if tok.like_num]
print(bound_method)
```

```
[1988, 4,839,853]
```

```
# For diagnostic purposes, it may be useful to examine these attributes
# all together. Here's a code fragment that sets up a pandas df of token
# attributes:

import pandas as pd # Use a pandas DF

# These will be out column names
cols = ("text", "lemma_","is_punct", "is_stop", "is_alpha","is_space","lower_")

rows = [] # A blank list to hold the rows

for t in doc: # Iterate through the tokens - will work for any length document
    # build the next row
    row = [t.text, t.lemma_,  t.is_punct,  t.is_stop,  t.is_alpha,  t.is_space,  t.lower_]
    rows.append(row) # Append the row to the existing rows

# Create the pandas data frame from the column names and the list of rows
attri_pdf = pd.DataFrame(rows, columns=cols)

attri_pdf # Gives a preview, but may not show all rows
```

	text	lemma_	is_punct	is_stop	is_alpha	is_space	lower_
0	An	an	False	True	True	False	an
1	information	information	False	False	True	False	information
2	retrieval	retrieval	False	False	True	False	retrieval
3	technique	technique	False	False	True	False	technique
4	using	use	False	True	True	False	using
...	...	...	...	...	...	...	...
62	indexing	indexing	False	False	True	False	indexing
63	(	(	True	False	False	False	(
64	LSI	LSI	False	False	True	False	lsi
65	)	)	True	False	False	False	)
66	.	.	True	False	False	False	.

67 rows × 7 columns

Next steps: Generate code with attri_pdf | View recommended plots | New interactive sheet

In previous weeks we have considered stop words and why in some cases it makes sense to remove them from the token stream. Let's examine spaCy's stop word list.

```
import spacy
spacy_stopwords = spacy.lang.en.stop_words.STOP_WORDS
len(spacy_stopwords)
```

```
326
```

```
list(spacy_stopwords)[:8]
```

```
['without',
 'being',
 'and',
 'myself',
 'wherever',
 'nevertheless',
 'take',
 'amount']
```

It is good to know what is on a stop list because sometimes these lists contain words that we do not want to discard. Because spaCy tags each token with an attribute showing whether that token is a stop word, but does not discard the stop words, we have the opportunity to do diagnostics on the results.

Let's process another piece of text from Wikipedia to focus on the stop words.

```
text = """In natural language processing, the Latent Dirichlet Allocation (LDA)
is a generative statistical model that allows sets of observations to be
explained by unobserved groups that explain why some parts of the data are
similar. For example, if observations are words collected into documents,
it posits that each document is a mixture of a small number of topics and that
each word's presence is attributable to one of the document's topics. LDA is
an example of a topic model and belongs to the machine learning field and in
a wider sense to the artificial intelligence field."""

doc = nlp(text)
type(doc), len(doc)
```

```
(spacy.tokens.doc.Doc, 113)
```

```
# Display the tokens that are stop words:
print([token for token in doc if token.is_stop])
```

```
[In, the, is, a, that, of, to, be, by, that, why, some, of, the, are, For, if, are, into, it, that, each, is, a, of, a, of, and, that, e
```

Note how spaCy has correctly tagged stop words even when they begin with a capital letter.

```
# Make a list of the tokens that are not stop-words
no_stops = [token for token in doc if not token.is_stop]
type(no_stops), len(no_stops)
```

```
(list, 64)
```

```
# Use slicing to view the first few non-stop words.
no_stops[0:12]
```

```
[natural,
 language,
 processing,
 ,,
 Latent,
 Dirichlet,
 Allocation,
 (,
 LDA,
 ),
 ,
 generative]
```

In some applications we may have uses for the punctuation tokens, but it is also good to know how to remove them. Conveniently, spaCy has also tagged every token with an indicator of whether it is punctuation.

```
# Also remove punctuation tokens
no_stops_or_punct = [token for token in no_stops if not token.is_punct]
type(no_stops_or_punct[0]), len(no_stops_or_punct)
```

```
(spacy.tokens.token.Token, 56)
```

```
# Use slicing to view the first few non-stop, non-punct words.
no_stops_or_punct[0:10]
```

```
[natural,
 language,
 processing,
 Latent,
 Dirichlet,
 Allocation,
 LDA,
 ,
 generative,
 statistical]
```

```
# Hmm, no_stops_or_punct[7] seems to be blank. Write a few lines of
# code to find out what it is/was. Hint: Try creating a pandas
# data frame of token attributes like the one we made in an earlier
# section of this lab.

# 5.7: Investigate the mysterious token.
no_stops_or_punct[7] #Confirmed Blank
text = """In natural language processing, the Latent Dirichlet Allocation (LDA)
is a generative statistical model that allows sets of observations to be
explained by unobserved groups that explain why some parts of the data are
similar. For example, if observations are words collected into documents,
it posits that each document is a mixture of a small number of topics and that
each word's presence is attributable to one of the document's topics. LDA is
an example of a topic model and belongs to the machine learning field and in
a wider sense to the artificial intelligence field."""

doc = nlp(text)
no_stops = [token for token in doc if not token.is_stop]
no_stops_or_punct = [token for token in no_stops if not token.is_punct]

df = pd.DataFrame(
    [(i, t.text, t.orth_, repr(t.text), t.is_space, t.is_alpha, t.pos_) for i, t in enumerate(no_stops_or_punct)],
    columns=["Index", "Text", "Orth_ID", "Repr", "Is_Space", "Is_Alpha", "POS"]
)

print(df)
#Used Source - https://www.phind.com/

#I guess 7 is just a Space?
```

```
    Index          Text       Orth_ID            Repr  Is_Space  Is_Alpha  \
0       0       natural       natural       'natural'     False      True
1       1      language      language      'language'     False      True
2       2    processing    processing    'processing'     False      True
3       3        Latent        Latent        'Latent'     False      True
4       4     Dirichlet     Dirichlet     'Dirichlet'     False      True
5       5    Allocation    Allocation    'Allocation'     False      True
6       6           LDA           LDA           'LDA'     False      True
7       7            \n            \n            '\n'      True     False
8       8    generative    generative    'generative'     False      True
9       9   statistical   statistical   'statistical'     False      True
10     10         model         model         'model'     False      True
11     11        allows        allows        'allows'     False      True
12     12          sets          sets          'sets'     False      True
13     13  observations  observations  'observations'     False      True
14     14            \n            \n            '\n'      True     False
15     15     explained     explained     'explained'     False      True
16     16    unobserved    unobserved    'unobserved'     False      True
17     17        groups        groups        'groups'     False      True
18     18       explain       explain       'explain'     False      True
19     19         parts         parts         'parts'     False      True
20     20          data          data          'data'     False      True
21     21            \n            \n            '\n'      True     False
22     22       similar       similar       'similar'     False      True
23     23       example       example       'example'     False      True
24     24  observations  observations  'observations'     False      True
25     25         words         words         'words'     False      True
26     26     collected     collected     'collected'     False      True
27     27     documents     documents     'documents'     False      True
28     28            \n            \n            '\n'      True     False
29     29        posits        posits        'posits'     False      True
30     30      document      document      'document'     False      True
31     31       mixture       mixture       'mixture'     False      True
32     32         small         small         'small'     False      True
33     33        number        number        'number'     False      True
34     34        topics        topics        'topics'     False      True
35     35            \n            \n            '\n'      True     False
36     36          word          word          'word'     False      True
37     37      presence      presence      'presence'     False      True
38     38  attributable  attributable  'attributable'     False      True
39     39      document      document      'document'     False      True
40     40        topics        topics        'topics'     False      True
41     41           LDA           LDA           'LDA'     False      True
42     42            \n            \n            '\n'      True     False
43     43       example       example       'example'     False      True
44     44         topic         topic         'topic'     False      True
45     45         model         model         'model'     False      True
46     46       belongs       belongs       'belongs'     False      True
47     47       machine       machine       'machine'     False      True
48     48      learning      learning      'learning'     False      True
49     49         field         field         'field'     False      True
50     50            \n            \n            '\n'      True     False
51     51         wider         wider         'wider'     False      True
52     52         sense         sense         'sense'     False      True
53     53    artificial    artificial    'artificial'     False      True
54     54  intelligence  intelligence  'intelligence'     False      True
55     55         field         field         'field'     False      True
```

Discuss and Collaborate

Task 5.7, just above, may require some ingenuity. Check in with someone else in the lab to get ideas on how to tackle this. When you have uncovered the result, discuss the implications with your partner.

```
# Another attribute on a token contains the lowercase version
# of the token. Why does this attribute end with an underscore?
lowercased = [ token.lower_ for token in no_stops_or_punct]
lowercased[0:9]
```

```
['natural',
 'language',
 'processing',
 'latent',
 'dirichlet',
 'allocation',
 'lda',
 '\n',
 'generative']
```

When we called nlp() on the text object and created the tokens, spaCy also automatically guessed at the lemma for each token and stuck that in as an attribute. Knowing what you know about lemmatization, what does this imply about other processing that spaCy may have done to this text?

```
# Make an additional list of lemma tokens
lemma_list = [token.lemma_ for token in no_stops_or_punct]
type(lemma_list[0]), len(set(lemma_list))
```

```
(str, 39)
```

```
# The output above suggests that some lemmas appear in the token
# list more than one time. Use Counter from the collections package
# to count instances of lemmas.
from collections import Counter

# 5.8: Instantiate a counter object with Counter(lemma_list). Assign this
# to a new variable such as wc_lemmas
wc_lemmas = Counter(lemma_list)
print(wc_lemmas)

```

```
Counter({'\n': 7, 'document': 3, 'topic': 3, 'LDA': 2, 'model': 2, 'observation': 2, 'explain': 2, 'example': 2, 'word': 2, 'field': 2,
```

```
# 5.9: Display the frequency counts of the five most common lemmas. Hint:
# a Counter has a bound method called most_common() that takes one
# argument called "n"
most_common = wc_lemmas.most_common(5)
print(most_common)
```

```
[('\n', 7), ('document', 3), ('topic', 3), ('LDA', 2), ('model', 2)]
```

```
# 5.10: Grab a new long string from Wikipedia or another source
#      and remove stop words and punctuation, then lemmatize and
#      count the frequencies of the top five lemmas. Write new code
#      below, based on what you did above.

text = "Software packages on Android, which use the APK format, are generally distributed through a proprietary application store; non-Go
doc = nlp(text)
filter_token = [token for token in doc if not token.is_stop and not token.is_punct]
lemma_list = [token.lemma_ for token in filter_token]
wc_lemmas = Counter(lemma_list)
most_common = wc_lemmas.most_common(5)
print(most_common)
#Used Source - https://www.phind.com/

```

```
[('operating', 3), ('system', 3), ('Android', 2), ('software', 1), ('package', 1)]
```

Sentence Segmentation

The spaCy doc object contains an element called "sents" that records the beginning and ending position (counting by tokens) of each sentence in the document. Finding sentence boundaries requires a substantial amount of algorithmic complexity, because the ending punctuation in strings such as U.S. or etc. may or may not indicate a sentence boundary. There are four strategies for sentence boundary detection in spaCy: dependency parser (default), statistical segmenter, rule-based segmenter, or custom function. Let's tokenize a fragment of Wikipedia text using the default (dependency parser) and then examine the resulting sentences.

```
text = """Sentence boundary disambiguation (SBD), also known as sentence breaking, sentence boundary detection, and sentence segmentatic
text[-26:] # Show the end of the string

```

```
' computer code, and slang.'
```

```
doc = nlp(text)
type(doc.sents)
```

```
generator
```

In Python, a generator is a special kind of iterator function that creates the requested elements on demand and "on the fly." This is a helpful approach when working with large sets of data elements that it would be challenging to represent in memory all at once. It is as easy to create a generator as it is to create a list comprehension:

```
(n ** 2 for n in range(50)) # First try it without assigning it to an object
```

```
<generator object <genexpr> at 0x78c3a4fffac0>
```

```
sq_gen = (n ** 2 for n in range(20)) # This time, save the generator and check its type
type(sq_gen)
```

```
generator
```

```
# Try running this cell twice. What happens the second time?
# After the second run, what happens if you run the previous cell again?
for num in sq_gen:
    if num < 100:
      print(num)
#First Time Run
#0
#1
#4
#9
#16
#25
#36
#49
#64
#81

#Second Time it Returns Nothing or Blank on OutPut
```

Looking back a couple of code blocks, doc.sents is a generator object, which means we can iterate through it's elements to find what we need.

```
for sent in doc.sents:
    print("start_pos={}, end_pos={}, text:{}".format(sent.start, sent.end, sent.text))
```

```
start_pos=0, end_pos=36, text:Sentence boundary disambiguation (SBD), also known as sentence breaking, sentence boundary detection, and
start_pos=36, end_pos=67, text:Natural language processing tools often require their input to be divided into sentences; however, senten
start_pos=67, end_pos=103, text:In written English, a period may indicate the end of a sentence, or may denote an abbreviation, a decima
start_pos=103, end_pos=139, text:About 47% of the periods in the Wall Street Journal corpus denote abbreviations.[1] Question marks and
```

```
# Use the doc.sent generator object to iterate through the sentences. Create
# a pandas data frame containing the starting and ending token numbers of each
# sentence. Then use the data frame to fetch the first token from each sentence.
import pandas as pd # Use a pandas DF

# These will be our column names
cols = ("start", "end")
rows = [] # A blank list to hold the rows

# 5.11: Iterate through the sentences, appending a row of start and end
#        positions for each sentence.
text = """Sentence boundary disambiguation (SBD), also known as sentence breaking, sentence boundary detection, and sentence segmentatic
doc = nlp(text)
for sent in doc.sents:
    rows.append([sent.start, sent.end])

sent_pdf = pd.DataFrame(rows, columns=cols)

sent_pdf


```

	start	end
0	0	36
1	36	67
2	67	103
3	103	139

Next steps: Generate code with sent_pdf View recommended plots New interactive sheet

```
# 5.12: Create a pandas data frame from the rows and column names
df = pd.DataFrame(rows, columns=cols)
```

```
   start  end
0      0   36
1     36   67
2     67  103
3    103  139
```

```
# 5.13: Display the pandas data frame
print(df)
```

```
   start  end
0      0   36
1     36   67
2     67  103
3    103  139
```

```
# 5.14: Iterate through the data frame and print the first token
#         from each sentence.

for index, row in df.iterrows():
    print(doc[row['start']])
```

```
Sentence
Natural
In
About
```

Part of Speech Tagging

POS tagging is a critical processing step in most pipelines, and much more difficult in some languages (e.g., Chinese) than in others. The standard spaCy pipeline always includes a "tagger" that assigns POS tags based on a statistical analysis of a training corpus.

```
doc = nlp("I was reading an article about Berkeley Avenue in Reading, which was closed due to a police investigation.")
type(doc), len(doc)
```

```
(spacy.tokens.doc.Doc, 20)
```

```
from tabulate import tabulate # To make a neat table

tabdata = [ (token, token.tag_, token.pos_, spacy.explain(token.tag_)) for token in doc]

print(tabulate(tabdata,  headers=["Token", "Token Tag", "POS", "Explanation"]))

```

```
Token          Token Tag    POS    Explanation
-------------  -----------  -----  -----------------------------------------
I              PRP          PRON   pronoun, personal
was            VBD          AUX    verb, past tense
reading        VBG          VERB   verb, gerund or present participle
an             DT           DET    determiner
article        NN           NOUN   noun, singular or mass
about          IN           ADP    conjunction, subordinating or preposition
Berkeley       NNP          PROPN  noun, proper singular
Avenue         NNP          PROPN  noun, proper singular
in             IN           ADP    conjunction, subordinating or preposition
Reading        NNP          PROPN  noun, proper singular
,              ,            PUNCT  punctuation mark, comma
which          WDT          PRON   wh-determiner
was            VBD          AUX    verb, past tense
closed         VBN          VERB   verb, past participle
```

```
due            IN           ADP    conjunction, subordinating or preposition
to             IN           ADP    conjunction, subordinating or preposition
a              DT           DET    determiner
police         NN           NOUN   noun, singular or mass
investigation  NN           NOUN   noun, singular or mass
.              .            PUNCT  punctuation mark, sentence closer
```

```
# Make a list of tokens for all of the proper nouns
propnlist = [token for token in doc if token.pos_ == "PROPN"]

[ (token, token.is_ascii, token.is_title) for token in propnlist]
```

```
[(Berkeley, True, True), (Avenue, True, True), (Reading, True, True)]
```

```
# Write another sentence that includes a place name. Tokenize it,
# display the POS tags, and excerpt the proper noun(s).

sent = "I like to go to the Movies on Elm Street Called Cinemark"
# 5.15: Create a new text object for tokenizing.
doc = nlp(sent)

for token in doc:
    print(f"{token.text}: {token.pos_}")

proper_nouns = [token.text for token in doc if token.pos_ == "PROPN"]
print(proper_nouns)

#Used Source - https://www.phind.com/

```

```
I: PRON
like: VERB
to: PART
go: VERB
to: ADP
the: DET
Movies: PROPN
on: ADP
Elm: PROPN
Street: PROPN
Called: VERB
Cinemark: PROPN
['Movies', 'Elm', 'Street', 'Cinemark']
```

```
# 5.16: Tokenize the text object.
doc = nlp(sent)
for token in doc:
    print(token.text)
```

```
I
like
to
go
to
the
Movies
on
Elm
Street
Called
Cinemark
```

```
# 5.17: Display the POS tags for all tokens.
pos_tags = [token.pos_ for token in doc]
print(pos_tags)
```

```
['PRON', 'VERB', 'PART', 'VERB', 'ADP', 'DET', 'PROPN', 'ADP', 'PROPN', 'PROPN', 'VERB', 'PROPN']
```

```
# 5.18: Extract the proper nouns and display them.
prop_noun = [token.text for token in doc if token.pos_ == "PROPN"]
print(prop_noun)
```

```
['Movies', 'Elm', 'Street', 'Cinemark']
```

Dependency Parsing

```
# Let's take a closer look at the dependency structure:
from tabulate import tabulate # To make a neat table

tabdata = [ (token.text, token.tag_, token.dep_, token.head.text, token.head.tag_) for token in doc]

print(tabulate(tabdata,  headers=["Token", "Token POS", "Dependency", "Head Token", "Head POS"]))

```

```
Token     Token POS    Dependency    Head Token    Head POS
--------  -----------  ------------  ------------  ----------
I         PRP          nsubj         like          VBP
like      VBP          ROOT          like          VBP
to        TO           aux           go            VB
go        VB           xcomp         like          VBP
to        IN           prep          go            VB
the       DT           det           Movies        NNPS
Movies    NNPS         pobj          to            IN
on        IN           prep          Movies        NNPS
Elm       NNP          compound      Street        NNP
Street    NNP          pobj          on            IN
Called    VBN          acl           Movies        NNPS
Cinemark  NNP          dobj          like          VBP
```

Take a close look at the output just above. For each token in the sentence, the text of the token is shown along with its part of speech. Then the dependency relation is shown. For example, the first token, "I", is the noun/subject of the sentence and is therefore dependent on the main verb, "reading", which is the gerund form of the verb to read.

Take the time to examine each row of the output and make sure you understand the dependency relation that is being documented. And remember that spaCy's ability to diagram the relations in this way works because of the language model we originally loaded: "en_core_web_sm". Also important: The default sentence segmentation that we examined in a previous block works because spaCy's dependency parser accounts for all of the elements in a sentence, and therefore "knows" when the period character is closing a sentence.

```
# Grab another sentence from the web, but this time, cut off the sentence
# before the end so that some key grammatical element is missing. Do paste
# a period on the end, though, just to see if you can confuse spaCy.

# 5.19: Cut and paste part of a sentence from the web into a text variable.
sent = " After having built a prototype internally known as the Fadden demo predominantly by purchasing licensing agreements for most of
```

```
# 5.20: Tokenize the sentence.
doc = nlp(sent)
doc
```

```
 After having built a prototype internally known as the Fadden demo predominantly by purchasing licensing agreements for most of the
software components built around a custom JavaScript front-end, the company failed to convince investors, and so in April 2004 they
pivoted to building an Operating System for Phones at the.
```

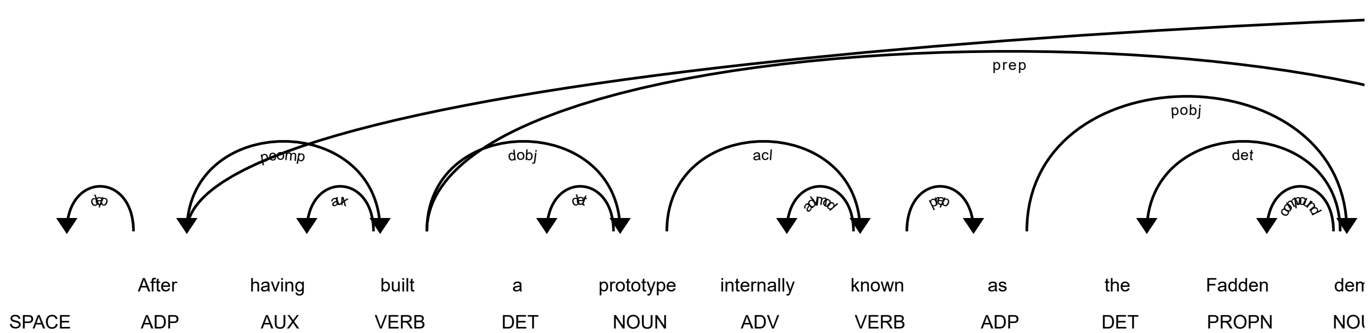
```
# 5.21: Generate a table showing the dependency relations in the sentence.
from tabulate import tabulate

tabdata = [ (token.text, token.tag_, token.dep_, token.head.text, token.head.tag_) for token in doc]

print(tabulate(tabdata,  headers=["Token", "Token POS", "Dependency", "Head Token", "Head POS"]))
```

```
Token          Token POS    Dependency    Head Token    Head POS
-------------  -----------  ------------  ------------  ----------
               _SP          dep           After         IN
After          IN           prep          failed        VBD
having         VBG          aux           built         VBN
built          VBN          pcomp         After         IN
a              DT           det           prototype     NN
prototype      NN           dobj          built         VBN
internally     RB           advmod        known         VBN
known          VBN          acl           prototype     NN
as             IN           prep          known         VBN
the            DT           det           demo          NN
Fadden         NNP          compound      demo          NN
demo           NN           pobj          as            IN
predominantly  RB           advmod        by            IN
by             IN           prep          built         VBN
purchasing     VBG          pcomp         by            IN
```

```
licensing      NN         compound    agreements    NNS
agreements     NNS        dobj        purchasing    VBG
for            IN         prep        purchasing    VBG
most           JJS        pobj        for           IN
of             IN         prep        most          JJS
the            DT         det         components    NNS
software       NN         compound    components    NNS
components     NNS        pobj        of            IN
built          VBN        acl         components    NNS
around         IN         prep        built         VBN
a              DT         det         custom        NN
custom         NN         pobj        around        IN
JavaScript     NNP        nmod        end           NN
front          JJ         compound    end           NN
-              HYPH       punct       end           NN
end            NN         appos       custom        NN
,              ,          punct       failed        VBD
the            DT         det         company       NN
company        NN         nsubj       failed        VBD
failed         VBD        ROOT        failed        VBD
to             TO         aux         convince      VB
convince       VB         xcomp       failed        VBD
investors      NNS        dobj        convince      VB
,              ,          punct       failed        VBD
and            CC         cc          failed        VBD
so             RB         advmod      pivoted       VBD
in             IN         prep        pivoted       VBD
April          NNP        pobj        in            IN
2004           CD         nummod      April         NNP
they           PRP        nsubj       pivoted       VBD
pivoted        VBD        conj        failed        VBD
to             IN         prep        pivoted       VBD
building       VBG        pcomp       to            IN
an             DT         det         System        NNP
Operating      VBG        compound    System        NNP
System         NNP        dobj        building      VBG
for            IN         prep        System        NNP
Phones         NNPS       pobj        for           IN
at             IN         prep        building      VBG
the            DT         pobj        at            IN
.              .          punct       pivoted       VBD
```

```
# 5.22: Add a comment to document any mistakes that spaCy made.
#Documentation of potetntial Mistakes
#-              HYPH        punct        end          NN
#,              ,           punct        failed       VBD
#.              .           punct        pivoted      VBD
```

```
# As we saw in a previous lab, there is a graphical display module for
# spaCy that supports drawing a figure of the dependency relations.
from spacy import displacy
displacy.render(doc, style='dep', jupyter=True, options={'distance': 90})
```



Now that you can see the dependency relations as a graph, do you notice any problems with the parsing? Did spaCy make any mistakes in connecting the various elements of the sentence?

Named Entity Recognition

Whenever spaCy finds a token that looks like a proper noun, it tags it as a predicted named entity. "Predicted," because each spaCy language model has a trained classifier that makes predictions of whether or not a token might be a named entity and also what type of entity it is (e.g., an organization, a country, or something else.

The IOB tagging method is a a straightforward way of notating the status of tokens. Tokens can have one of the following four statuses:

TAG	ID	DESCRIPTION
I	1	Token is inside an entity.
O	2	Token is outside an entity.
B	3	Token begins an entity.
	0	No entity tag is set (missing value).

```
text='''We’re bringing the celebration of Syracuse University’s 150 years of impact to Chicago'''
doc = nlp(text)
type(doc.ents)
```

```
tuple
```

```
# So the list of entities is a tuple, which means we should be able to slice it.
doc.ents[0]
```

```
Syracuse University
```

```
# We can iterate through all of the entities in the document.
for ent in doc.ents:
    print("{}, [{},{}), {}".format(ent.text, ent.start_char, ent.end_char, ent.label_))

    # For each entity, we can also access each entity as a span and
    # iterate through its tokens
    for token in ent.as_doc():
        print("    {} {} {}".format(token, token.ent_iob_, token.ent_type_))
```

```
Syracuse University, [34,53), ORG
    Syracuse B ORG
    University I ORG
150 years, [56,65), DATE
    150 B DATE
    years I DATE
Chicago, [79,86), GPE
    Chicago B GPE
```

Notice that for each of the three entities above, there's a numeric expression that shows the start and end of the character span where the entities can be found in the original string.

In the output shown above, ORG and DATE are pretty clear, but what does GPE stand for? Do a web search on "spacy GPE" to find out.

```
# The displacy module can also provide a graphical view of the named entities:
from spacy import displacy
displacy.render(doc, style="ent", jupyter=True)
```

We’re bringing the celebration of Syracuse University **ORG** ’s 150 years **DATE** of impact to Chicago **GPE**

```
# Now write or find another sentence that contains some named entities.
text = "Duke University has had 25 Good Years of Basketball to the Durham Community"
# 5.23: Assign a new sentence containing some entities into a text variable.

```

```
# 5.24: Tokenize the sentence.
doc = nlp(text)
doc
```

```
Duke University has had 25 Good Years of Basketball to the Durham Community
```

```
# 5.24: Generate a displacy graphic with the named entities.
from spacy import displacy
displacy.render(doc, style="ent", jupyter=True)
```

Duke University **ORG** has had 25 Good Years **MONEY** of Basketball to the Durham Community **ORG**

Word Vectors

Word vectors are numeric representations of words in multidimensional space. Notable implementations of word vectors include GloVe, a technique that obtains vectors from a large word co-occurence matrix developed from a corpus, and Word2vec, an predictive neural network training process that does repetitive, incremental training over strings of tokens in a corpus. Word vectors have the interesting property that semantically similar words appear close to each other in multidimensional space. Given any two word vectors, we can calculate the similarity (or distance) between them.

SpaCy language models come in several sizes. Small models do not contain word vectors - a space saving strategy. So if we want to use word vectors, we need to load the medium or large version of a language model as shown in the next cell. Note that because of the size of a large language model file, the next code cell may take about three minutes to complete. That's a lot of time and therefore a lot of data. Imagine how this might impact the operation of a deployed system.

```
import spacy.cli # Use the command line interface
spacy.cli.download("en_core_web_lg") # This imports the large model onto your virtual machines
import en_core_web_lg # Now that it is downloaded, we can import it
nlp_lg = en_core_web_lg.load() # Create an instance for further use
type(nlp_lg)
```

```
✔ Download and installation successful
You can now load the package via spacy.load('en_core_web_lg')
⚠ Restart to reload dependencies
If you are in a Jupyter or Colab notebook, you may need to restart Python in
order to load all the package's dependencies. You can do this by selecting the
'Restart kernel' or 'Restart runtime' option.
```

```
spacy.lang.en.English
def __call__(text: Union[str, Doc], *, disable: Iterable[str]=SimpleFrozenList(), component_cfg:
Optional[Dict[str, Dict[str, Any]]]=None) -> Doc
```

```
/usr/local/lib/python3.11/dist-packages/spacy/lang/en/__init__.py
A text-processing pipeline. Usually you'll load this once per process,
and pass the instance around your application.

Defaults (class): Settings, data and factory methods for creating the `nlp`
    object and processing pipeline.
```

```
# Let's get the vectors for three words to examine similarities
apple = nlp_lg.vocab["apple"]
banana = nlp_lg.vocab["banana"]
car = nlp_lg.vocab["car"]

print(apple.vector) # Take a look at one of them
```

```
[-3.6391e-01  4.3771e-01 -2.0447e-01 -2.2889e-01 -1.4227e-01  2.7396e-01
 -1.1435e-02 -1.8578e-01  3.7361e-01  7.5339e-01 -3.0591e-01  2.3741e-02
 -7.7876e-01 -1.3802e-01  6.6992e-02 -6.4303e-02 -4.0024e-01  1.5309e+00
 -1.3897e-02 -1.5657e-01  2.5366e-01  2.1610e-01 -3.2720e-01  3.4974e-01
 -6.4845e-02 -2.9501e-01 -6.3923e-01 -6.2017e-02  2.4559e-01 -6.9334e-02
 -3.9967e-01  3.0925e-02  4.9033e-01  6.7524e-01  1.9481e-01  5.1488e-01
 -3.1149e-01 -7.9939e-02 -6.2096e-01 -5.3277e-03 -1.1264e-01  8.3528e-02
 -7.6947e-03 -1.0788e-01  1.6628e-01  4.2273e-01 -1.9009e-01 -2.9035e-01
  4.5630e-02  1.0120e-01 -4.0855e-01 -3.5000e-01 -3.6175e-01 -4.1396e-01
  5.9485e-01 -1.1524e+00  3.2424e-02  3.4364e-01 -1.9209e-01  4.3255e-02
  4.9227e-02 -5.4258e-01  9.1275e-01  2.9576e-01  2.3658e-02 -6.8737e-01
 -1.9503e-01 -1.1059e-01 -2.2567e-01  2.4180e-01 -3.1230e-01  4.2700e-01
  8.3952e-02  2.2703e-01  3.0581e-01 -1.7276e-01  3.2536e-01  5.4696e-03
 -3.2745e-01  1.9439e-01  2.2616e-01  7.4742e-02  2.2033e-01 -4.0301e-01
 -3.1594e-01 -2.8910e-02  9.7858e-01  7.1860e-01  1.4995e-01  6.3421e-02
  2.8332e-01 -1.5231e-01  3.9330e-04  1.8076e-01 -4.0199e-01  6.0187e-02
 -2.7543e-02  1.6590e-01 -2.5774e-01  1.6150e-01  3.7247e-01 -3.8273e-01
  2.4012e-01 -4.2617e-02 -6.6785e-01 -9.4437e-01  2.7916e-01  1.0476e-01
  1.3952e+00 -1.4296e-01 -5.5049e-01  5.3982e-02 -7.7524e-01 -2.8255e-01
 -2.3323e-02  2.4801e-01  2.2855e-01 -3.7408e-01  7.6012e-02  2.4031e-01
  1.0746e-01  1.2411e-01 -2.0676e-01 -2.5804e-01 -1.6791e-01  4.3499e-01
  6.1762e-01 -2.9955e-02  1.6196e-01 -2.9001e-01 -3.1159e-01 -8.7262e-01
  4.3167e-01 -1.5071e-01 -4.1420e-01 -5.3730e-01 -1.9910e-01  1.3270e-01
 -1.5018e-01 -4.9335e-01 -2.5127e+00  3.1660e-01  3.6396e-01 -5.9248e-02
  3.1120e-02  4.1071e-02  1.6917e-02  5.8410e-01 -2.0201e-01  7.0238e-02
  8.7547e-01 -2.0114e-01  5.1920e-01  2.6786e-01 -5.5643e-01 -3.1247e-01
 -3.7992e-01  4.2857e-01  4.1780e-01  3.0608e-01 -2.1657e-01  7.2464e-01
  6.1734e-01  5.8085e-02 -6.2708e-01  5.2895e-02 -2.5628e-01 -3.2688e-01
```

```
 -6.1280e-01  6.2609e-01 -1.7965e-01  8.8925e-01  2.1963e-01 -3.4052e-03
 -7.8663e-02  3.4799e-01 -2.6062e-01  8.0410e-03  1.1721e-01 -4.5147e-01
 -1.2178e-01 -5.7030e-01  4.6602e-01  2.5059e-02  5.3986e-02 -7.6693e-01
  1.3173e-01 -2.8776e-02 -4.1915e-01 -2.4415e-01 -4.0295e-01 -4.1520e-01
  3.7643e-02 -1.4843e-01  2.6094e-02  1.5315e-01  3.8310e-01 -5.5825e-01
 -3.3433e-01 -2.7939e-02 -4.3712e-01 -3.1802e-01 -3.1731e-01  9.2891e-02
 -9.9397e-02 -1.8846e-01  5.2270e-02  2.9061e-01  1.0639e+00  9.9584e-02
 -5.6775e-01  2.9446e-01  3.7797e-01 -2.1905e-01 -5.2616e-01 -4.1744e-01
 -6.5951e-01 -4.0820e-01 -6.0945e-01  1.1759e-02 -2.9122e-01 -3.1457e-01
  5.7076e-02  4.1503e-01  3.7345e-01 -4.7119e-02 -7.1996e-02  1.4587e-01
 -3.0763e-01  1.0759e-01 -5.9447e-01 -4.0205e-01  3.0677e-01 -1.9891e-01
 -7.0775e-01 -1.1513e-01  3.0866e-01 -6.9235e-01  2.1219e-01  1.0554e-01
  2.2617e-01 -2.6145e-01 -3.9298e-01 -2.3585e-01  3.0795e-02 -1.0193e-01
  3.2070e-01  3.0505e-01 -5.3470e-01 -7.9272e-02 -1.6817e-01 -2.2115e-01
 -3.5143e-01 -9.2376e-02  1.4686e-01 -1.9859e-01  2.0460e-01  2.0276e-01
  3.6144e-01 -3.5867e-01  4.0095e-01  6.3686e-02 -1.2763e-01 -1.6226e-01
 -3.1763e-01 -5.8732e-01 -5.4009e-01 -4.9035e-01 -4.6035e-01 -1.9794e-01
 -2.5209e-01  2.5706e-01  4.0110e-01  5.2830e-02 -3.2079e-01  3.9563e-01
 -4.4512e-01 -9.1862e-02 -1.9243e-01  1.5397e-01 -2.8923e-01  6.0561e-01
  5.8133e-01  3.2268e-01  6.3892e-02  8.5438e-02  1.4956e-01  3.8134e-01
 -1.1820e-01 -2.3951e-01 -6.7731e-01  2.8090e-01 -5.1770e-01 -4.1098e-01
 -4.1292e-01 -6.7856e-02 -3.3721e-02 -7.2958e-01 -4.7891e-01  7.2956e-01]
```

```python
# A spaCy lexeme, with a variety of attribute tests available
type(apple), [m for m in dir(apple) if m[0:3] == "is_"]
```

```
(spacy.lexeme.Lexeme,
 ['is_alpha',
  'is_ascii',
  'is_bracket',
  'is_currency',
  'is_digit',
  'is_left_punct',
  'is_lower',
  'is_oov',
  'is_punct',
  'is_quote',
  'is_right_punct',
  'is_space',
  'is_stop',
  'is_title',
  'is_upper'])
```

```python
# We can also confirm that a lexeme has a vector representation available
banana.has_vector
```

```
True
```

```python
# The .similarity() bound method allows us to generate a cosine similarity
# score for two vectors.
apple.similarity(banana)
```

```
0.5831844806671143
```

```python
# The similarity is transitive, and by the calculation spaCy uses, seems
# to result in values in the interval 0 to 1. Higher values indicate
# closer similarity.
banana.similarity(apple)
```

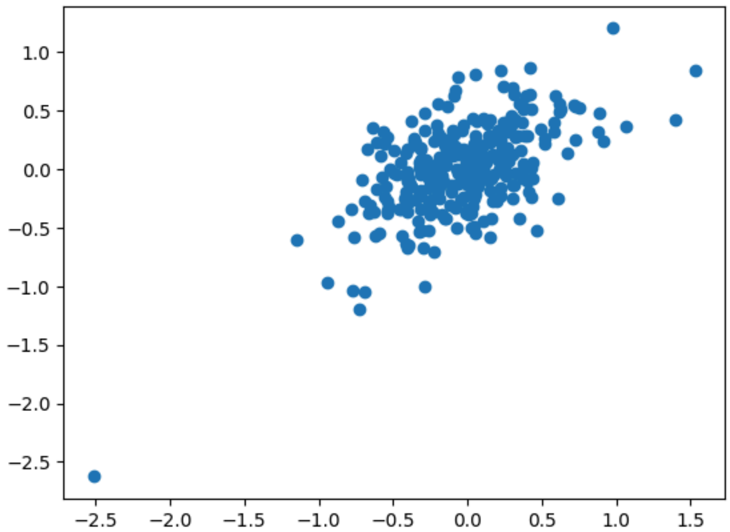
```
0.5831844806671143
```

```python
apple.similarity(car)
```

```
0.2174709141254425
```

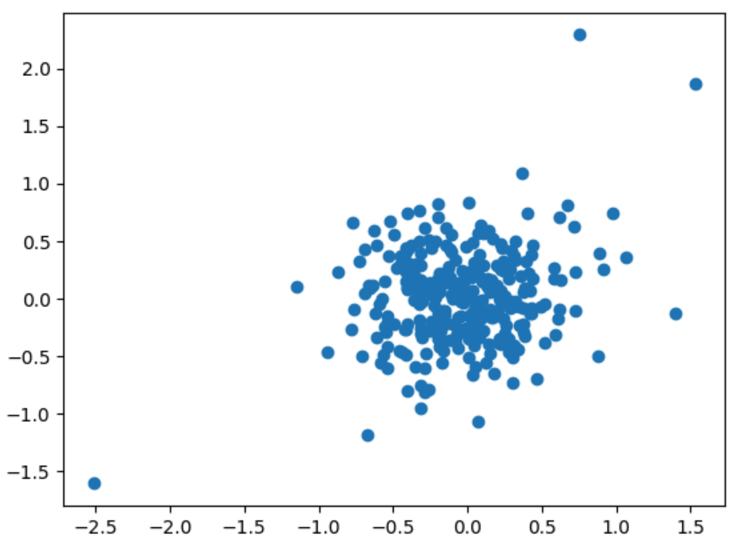
```python
# A scatterplot of two similar vectors should indicate linearity
import matplotlib.pyplot as plt # Make a simple plot
plt.scatter(apple.vector, banana.vector) # Show the result
```

```
<matplotlib.collections.PathCollection at 0x78c4981197d0>
```




```
# A scatterplot of two different vectors should be a circular cloud
plt.scatter(apple.vector, car.vector)
```

```
<matplotlib.collections.PathCollection at 0x78c39df13fd0>
```




```
# There are many options for a cosine similarity calculation,
# including this one from scipy that we are not using here.

#from scipy.spatial.distance import cosine
```

```
# With some help from numpy, we can calculate our own cosine scores
# using a vector dot product, computed with np.dot()
import numpy as np
def cosine(x,y):
    return np.dot(x,y) / (np.sqrt(np.dot(x,x)) * np.sqrt(np.dot(y,y)))
```

```
# One of the notable aspects of word vectors is the possibility of
# doing simple math to create analogies.
man = nlp_lg.vocab["man"].vector
woman = nlp_lg.vocab["woman"].vector
king = nlp_lg.vocab["king"].vector
queen = nlp_lg.vocab["queen"].vector

analogy = king - man + woman # Calculate the analogy with vector math
cosine(analogy, queen)
```

```
np.float32(0.78808445)
```

```
# Let's compare the analogy to another vector
cosine(analogy, car.vector) # Note how we needed to extract the vector for car
```

```
np.float32(0.13843127)
```

```
# Now you compute an anology and see what kind of results you get

# 5.25: Obtain the vectors for Paris, France, Berlin, and Germany
paris = nlp_lg.vocab["paris"]
berlin = nlp_lg.vocab["Berlin"]
germany = nlp_lg.vocab["Germany"]
france = nlp_lg.vocab["france"]

print(germany.vector) # Take a look at one of them
```

```
[-3.5532e-01  5.9025e-01  1.7082e-01 -2.9313e-03  7.3128e-01 -5.3945e-02
 -4.7846e-02  2.4606e-01 -1.8597e-01  8.0269e-01 -1.2702e+00 -1.2990e-01
 -1.7094e-01 -1.5757e-01  8.6082e-01  4.9421e-03 -2.7626e-01  1.3027e+00
  6.9247e-01  1.3457e-01  7.2636e-01  2.7259e-01  3.3717e-01 -2.1307e-01
 -4.1304e-01 -3.2571e-01 -3.9285e-01  9.8547e-02 -4.2460e-02  7.9824e-01
  2.2305e-01 -5.6391e-02 -1.9127e-03  4.4519e-01  3.8532e-01  2.1436e-01
  3.3141e-01  6.0297e-02  2.6125e-02  1.5253e-03  3.9830e-01 -2.2976e-01
 -1.9558e-01  3.7927e-01 -2.0930e-01 -1.5060e-01  8.1397e-02 -1.0093e+00
 -2.7606e-02  4.2331e-02 -5.1963e-01  1.5031e-01 -3.1286e-01 -8.1624e-01
  7.6164e-02  9.4180e-02  9.1425e-02 -5.0202e-01  3.7440e-01 -3.6910e-01
 -4.2274e-01  2.0528e-02  3.5944e-01 -5.9502e-01 -4.3034e-01  6.5409e-01
 -1.7446e-01  5.4362e-01  1.1969e-01 -2.2330e-01 -1.7697e-01  3.1174e-01
 -6.6029e-02 -5.2093e-01  5.4626e-01 -3.9850e-01 -3.9863e-01  2.2108e-01
  1.3207e-01  6.8904e-01 -3.0168e-01  6.0699e-01  5.7867e-01  3.3999e-01
 -1.9682e-01 -5.4443e-01  1.0600e+00 -4.6676e-01 -5.7628e-02 -1.1523e-01
 -4.0842e-01 -3.0253e-01  1.6459e-01 -2.9100e-01  2.5571e-01 -2.0017e-01
 -1.6753e-01 -1.9953e-01 -1.2312e-01  2.2200e-02  1.8670e-01 -1.5847e-02
  2.0861e-01  5.6133e-01  3.8892e-01 -1.3207e+00  8.6137e-02 -5.7136e-02
 -1.1727e-01 -7.7089e-01  2.8236e-01 -3.2194e-01 -1.4631e-01 -3.8873e-01
  2.4520e-01 -3.8108e-02  9.0389e-01 -7.2336e-02 -8.8831e-01 -2.7025e-01
  9.4136e-02  4.5059e-01  3.1964e-01 -2.5763e-01 -1.7744e-01  5.8418e-01
  4.3581e-01  2.8566e-02  5.5962e-01  2.8713e-01 -1.0694e-03 -5.8154e-01
 -2.2003e-01 -3.0540e-01  3.9144e-01 -6.6091e-01 -2.2653e-01  9.5001e-02
 -1.7457e-01 -3.6791e-02 -1.5287e+00 -2.7696e-01  3.0186e-01 -3.1090e-01
 -1.9702e-01  5.8790e-01  9.8363e-02  5.0177e-01 -1.4607e-02 -1.2669e-01
 -1.8857e-02 -1.9668e-01 -4.9711e-01 -4.1141e-01  1.7557e-01  2.2816e-01
 -1.8279e-01  6.2052e-01  3.8148e-01  8.7166e-01  6.1474e-01 -1.4620e-02
  1.3988e-01 -1.7228e-01  1.6204e-01 -1.5001e-01 -2.4463e-01  2.1547e-01
 -5.0795e-01  2.4293e-01 -5.7434e-01 -2.3912e-01 -5.1797e-01  6.5299e-01
  1.8023e-01  2.3604e-01 -8.8867e-02  4.2646e-01  8.1626e-01  2.2606e-01
  2.3301e-01 -2.3897e-01 -2.3303e-01  2.1172e-01  3.4523e-01 -2.8430e-01
 -6.1216e-03 -1.1932e-01 -3.1599e-01  1.7572e-01 -3.2610e-02  1.5303e-01
  2.4861e-01 -3.8134e-01 -4.8207e-01  6.9897e-01  9.0488e-01 -3.6598e-01
  6.9226e-01 -6.1707e-01  1.8368e-01 -2.8152e-01  6.8885e-01 -1.6421e-01
  1.4285e-01 -2.5463e-01  6.4269e-01 -2.0218e-01  3.7522e-01  3.0399e-02
 -1.2285e-01 -1.2358e-01  4.3093e-01 -4.4876e-01 -2.4861e-01  5.6688e-02
  8.9564e-02  1.2192e-01 -5.2339e-01  6.1985e-01  2.6750e-01 -3.3201e-01
  4.9707e-01 -2.8828e-01 -6.8352e-02  6.5845e-02  3.0167e-01  2.0211e-01
  1.5745e-01  2.7303e-01 -3.4166e-01  4.5752e-01 -6.3572e-01 -4.9968e-01
 -1.9339e-01  1.8362e-01  1.6941e-01 -6.2114e-02 -1.3672e-02 -9.6475e-04
 -3.3685e-01  2.8164e-01  6.8517e-02 -4.5437e-02  3.8221e-01 -7.3306e-02
  1.5074e-01  1.3741e-01  2.4259e-01 -4.2146e-01  7.5622e-01  4.2130e-01
  8.0590e-01 -6.4212e-01  2.9125e-01 -2.9884e-01 -2.5247e-01 -1.8504e-01
 -2.7947e-01 -1.2723e-01  4.0416e-01  1.2838e-01 -4.7400e-01  3.8117e-01
 -4.3376e-01 -1.9804e-01  4.7486e-01 -2.1127e-01 -5.5112e-01  3.2791e-01
 -7.0879e-01 -1.2255e-01 -8.5436e-03  3.0136e-01  4.2968e-01  6.6084e-01
  4.2748e-01 -1.7740e-01  2.3307e-01 -1.9780e-01 -1.3235e-01 -2.5483e-01
  1.1043e-01  8.4103e-02  3.6823e-02  5.7533e-01 -1.3012e-01 -5.6691e-02
 -9.3585e-02  5.4244e-01 -1.4315e-01  3.4506e-01  1.2744e-01  2.1567e-01
  1.5834e-01 -2.1950e-01  5.3764e-01 -2.7061e-01 -2.4100e-01  8.9329e-01]
```

```
# 5.26: Compute an analogy using three of these vectors

paris = nlp_lg.vocab["paris"].vector
germany = nlp_lg.vocab["Germany"].vector
france = nlp_lg.vocab["france"].vector

analogy = france - paris + germany
cosine(analogy, germany)
```

```
np.float32(0.8795712)
```

```
# 5.27: Find the cosine similarity of the analogy with the fourth vector

paris = nlp_lg.vocab["paris"].vector
```

```
germany = nlp_lg.vocab["Germany"].vector
france = nlp_lg.vocab["france"].vector
berlin = nlp_lg.vocab["berlin"].vector

analogy = france - paris + germany
cosine(analogy, berlin)
```

```
np.float32(0.52002084)
```

```
# 5.28: Find the cosine similarity of the analogy with banana
apple = nlp_lg.vocab["apple"].vector
banana = nlp_lg.vocab["banana"].vector
car = nlp_lg.vocab["car"].vector

analogy = apple - banana + car
cosine(analogy, car)
```

```
np.float32(0.7700715)
```

Sentence Similarity

As you have seen above, word vectors for similar words have similar patterns and though we don't know what each position in a vector signifies, these patterns support simple vector math for analogical reasoning. This suggests the possibility - which research seems to back up - that averaging two or more vectors provides a semantic summary of the vectors. Document and span objects in spaCy have attached vectors that repesents the average of all of the component word vectors.

```
doc1 = nlp_lg("The quick brown fox jumps over the lazy dog.")
doc2 = nlp_lg("The lazy dog jumps over the quick brown fox.")
doc1.similarity(doc1)
```

```
1.0
```

```
# We can also manually calculate the cosine similarity. It should be transitive
# such that cosine(A,B) == cosine(B,A)
doc1_vec = doc1.vector
doc2_vec = doc2.vector
cosine(doc2_vec, doc1_vec), cosine(doc1_vec, doc2_vec)
```

```
(np.float32(1.0), np.float32(1.0))
```

```
# Let's get the similarity of the first sentence (doc1) with a new sentence.
docdiff = nlp_lg("Four score and seven years ago, our fathers brought forth upon this continent a new nation.")
cosine(docdiff.vector, doc1_vec)
```

```
np.float32(0.72331554)
```

```
# The semantics can be quite muddled by the averaging process. These two
# sentences express the opposite sentiment
doc3 = nlp_lg("I like snow")
doc4 = nlp_lg("I hate snow")
cosine(doc3.vector, doc4.vector)
```

```
np.float32(0.94878143)
```

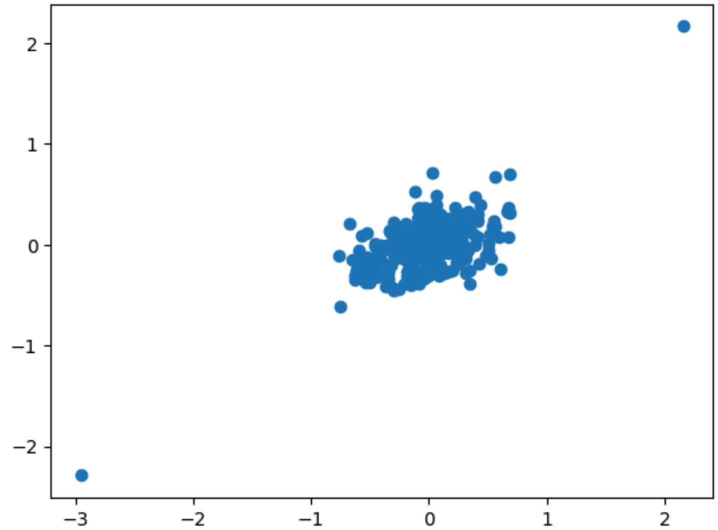
Think through the implication of the result shown above. By creating an averaged word vector, we have given every word in a sentence the same weight in creating the new summary vector. Another exercise to try would be to take out the word snow and see how that changes the results.

```
# 5.29: Recalculate the previous cell after removing the word snow from both sentences.
# Feel free to try any other experiments of interest.
doc3 = nlp_lg("I like")
doc4 = nlp_lg("I hate")
cosine(doc3.vector, doc4.vector)
```

```
np.float32(0.9049743)
```

```
# And here we see part of why this is so.
plt.scatter(nlp_lg.vocab["hate"].vector, nlp_lg.vocab["like"].vector)
```

```
<matplotlib.collections.PathCollection at 0x78c39dca3590>
```



```
# Now you create vector summaries of three sentences and look at the
# cosine similarities for each pair.

# 5.29: Create nlp_lg() objects for three sentences that you write yourself or
# copy and paste from the web.

sent_0 = "I like the Red Balloon on the fence"
sent_1 = "I hate that Cat on the Ledge"
sent_2 = "I wish I could fly in a Spaceship"

sent_0_doc = nlp_lg(sent_0)
sent_1_doc = nlp_lg(sent_1)
sent_2_doc = nlp_lg(sent_2)


```

```
# 5.30: Compute pairwise similarities for their vector summaries
sent_0_vec = sent_0_doc.vector
sent_1_vec = sent_1_doc.vector
sent_2_vec = sent_2_doc.vector

cosine(sent_0_vec, sent_1_vec)
```

```
np.float32(0.8767786)
```

```
# 5.31: Add a comment explaining what you see
#0.87 Correlation or Similar
```

Putting it All Together

Put everything you have learned from the exercises above to use in processing two Wikipedia articles that discuss similar topics (for example, the Super Bowl and the Rose Bowl). Extract the text from two Wikipedia articles. Then process both texts with spaCy. Extract all of the named entities and see which named entities the two articles have in common. Calculate a ratio of matching named entities to all detected entities.

```
# 5.32: Extract Named Entities from Two Wikipedia Articles and Find Matches
sent_0 = "As the NCAA does not organize or award an official national championship for FBS football (instead merely recognizing the deci
sent_1 = "College Football Playoff officials commissioned the trophy for the new playoff system, preferring a new award that was unconne

doc0 = nlp(sent_0)
doc1 = nlp(sent_1)

ents_0 = set(ent.text for ent in doc0.ents)
ents_1 = set(ent.text for ent in doc1.ents)
```

```

common_ents = ents_0 & ents_1
total_unique_ents = ents_0 | ents_1

ratio = len(common_ents) / len(total_unique_ents)
print(common_ents)
print(total_unique_ents)
print(ratio)

```

```
{'FBS'}
{'2014', 'FBS', 'the 2013 season', 'NCAA', 'first', 'the BCS National Championship Game', 'Bowl Championship Series', 'CFP', 'AFCA'}
0.1111111111111111
```

Optional Advanced Topic: Processing pipelines

Throughout this lab, we have been calling a function that we often referred to as nlp(). After loading a spaCy language model such as en_core_web_sm, we instantiate a pipeline object to conduct all of the steps that we will routinely want to accomplish with a document.

The spaCy pipeline can be modified to change the default components or to add new components. Here's a list of the default components from the spaCy documentation:

NAME	COMPONENT	CREATES	DESCRIPTION
tokenizer	Tokenizer	Doc	Segment text into tokens.
tagger	Tagger	Doc[i].tag	Assign part-of-speech tags.
parser	DependencyParser	Doc[i].head, Doc[i].dep, Doc.sents, Doc.noun_chunks	Assign dependency labels.
ner	EntityRecognizer	Doc.ents, Doc[i].ent_iob, Doc[i].ent_type	Detect and label named entities.
textcat	TextCategorizer	Doc.cats	Assign document labels.
…	custom components	`Doc._.xxx`, `Token._.xxx`, `Span._.xxx`	Assign custom attributes, methods or properties.

```
# We can also examine the pipeline for an instantiated object like this:
import spacy
nlp = spacy.load("en_core_web_sm")

nlp.pipeline
```

```
[('tok2vec', <spacy.pipeline.tok2vec.Tok2Vec at 0x78c4d493bb30>),
 ('tagger', <spacy.pipeline.tagger.Tagger at 0x78c3b3bba930>),
 ('parser', <spacy.pipeline.dep_parser.DependencyParser at 0x78c4993699a0>),
 ('attribute_ruler',
  <spacy.pipeline.attributeruler.AttributeRuler at 0x78c3b3c5e290>),
 ('lemmatizer',
  <spacy.lang.en.lemmatizer.EnglishLemmatizer at 0x78c3b3c240d0>),
 ('ner', <spacy.pipeline.ner.EntityRecognizer at 0x78c3b3bd4c80>)]
```

You may notice in the list above that there is no tokenizer. In the spaCy pipeline model, it is assumed that tokenization was accomplished before pipeline processing begins. All pipeline elements receive a doc object, work on it and return a doc object. The tokenizer has a different kind of job becuase it receives a raw character string and returns a list of tokens. Thus, the language object has a different slot where the tokenizer is listed:

```
nlp.tokenizer
```

```
<spacy.tokenizer.Tokenizer at 0x78c3fb163e20>
```

The spaCy pipeline was designed to balance simplicity and computational effort. Simplicity is important for getting started quickly with a language processing tasks, so the default pipeline contains all the stuff that most people need to address a realistic task. But if a component is not needed, it can save a lot of compute time to take a task out of the pipeline. Take a look at this example:

```
print(spacy.__version__) # Some version dependent stuff below
```

```
3.8.5
```

```
# Let's skip the entity recognition and the dependency parsing
nlp_simple = spacy.load("en_core_web_sm", exclude=["parser","ner"])
# Version 3 also adds facilities for enabling
# and disabling pipeline elements on the fly.

print(nlp_simple.pipeline) # SHow the pipeline
```

```
[('tok2vec', <spacy.pipeline.tok2vec.Tok2Vec object at 0x78c3b3124170>), ('tagger', <spacy.pipeline.tagger.Tagger object at 0x78c37de22d
```

```
# Groucho Marx is an entity, but this pipeline doesn't detect it, because the
# ner is disabled.
simple_doc = nlp_simple("Groucho Marx shot an elephant in his underpants.")

[ent for ent in simple_doc.ents]
```

```
[]
```

```
# And the pipeline did not do dependency parsing
[token.dep_ for token in simple_doc]
```

```
['', '', '', '', '', '', '', '', '']
```

```
# 5.32: Create a pipeline that contains a dependency parser, but no
#       entity recognition. Process a sentence and show that the entity
#       recognition is disabled.

nlp_simple = spacy.load("en_core_web_sm", exclude=["ner"])
simple_doc = nlp_simple("Ellie dropped her bone down a rabbit hole into the darkness.")
[ent for ent in simple_doc.ents]

```

```
[]
```

Adding Custom Pipeline Components

A component receives a Doc object and can modify it. By adding a component to the pipeline, you'll get access to the Doc at any point during processing – instead of only being able to modify it afterwards. You can control the position of the new component in the pipeline with the last, first, before, and after arguments.

ARGUMENT	TYPE	DESCRIPTION
doc	Doc	The Doc object processed by the previous component.
RETURNS	Doc	The Doc object processed by this pipeline component.

ARGUMENT	TYPE	DESCRIPTION
last	bool	If set to True, component is added last in the pipeline (default).
first	bool	If set to True, component is added first in the pipeline.
before	unicode	String name of component to add the new component before.
after	unicode	String name of component to add the new component after.

```
import spacy
from spacy.tokens import Doc, Span, Token
import json
from spacy.language import Language

# Here's a custom function for removing stopwords, starting with a decorato
@Language.component("remstop_component")
def remove_stopwords(doc):
    # A pipeline element would not normally contain this kind of diagnostic
    # but we just want to show what the method received before processing.
    print("Before stopwords_removal, this doc is: {}".format(doc))
    space_list = [t.whitespace_  for t in doc if not t.is_stop]
    new_doc = Doc(doc.vocab,
              words=[t.orth_ for t in doc if not t.is_stop],
              spaces=space_list
              )
    return new_doc

# Instantiate a default pipeline
nlp = spacy.load("en_core_web_sm")
```

```

# Add our stopword finder/remover
nlp.add_pipe("remstop_component", name="stopwords_removal", first=True)

# Show the pipeline
print(nlp.pipe_names)  # ['stopwords_removal', 'tagger', 'parser', 'ner']

# Process a sentence
doc = nlp("This is a sentence.")

# See the result
```

```
['stopwords_removal', 'tok2vec', 'tagger', 'parser', 'attribute_ruler', 'lemmatizer', 'ner']
Before stopwords_removal, this doc is: This is a sentence.
After stopwords_removal, this doc is: sentence.
```

Attribute and Method Extensions

For advanced users, it is possible to add new attributes and methods to spaCy objects. In a complex language processing system, additional attributes and methods could be used to annotate, control, and modify specialized features of a document or other spaCy object.

```
# Here we set a new attribute, called "classified" to indicate whether
# a document's contents should be kept secret.
Doc.set_extension("classified", default=True)
assert doc._.classified

# Note that this block will throw an error if it is run more than once
```

```
# Now we can set or retrieve the attribute
doc._.classified = False
```