https://campus.datacamp.com/courses/advanced-nlp-with-spacy/large-scale-data-analysis-with-spacy?ex=1

02.01

Data Structures: Vocab, Lexemes and StringStore

ADVANCED NLP WITH SPACY

Ines Montani
spaCy core developer

Shared vocab and string store (1)

- `Vocab` : stores data shared across multiple documents
- To save memory, spaCy encodes all strings to **hash values**
- Strings are only stored once in the `StringStore` via `nlp.vocab.strings`
- String store: **lookup table** in both directions

```
coffee_hash = nlp.vocab.strings['coffee']
coffee_string = nlp.vocab.strings[coffee_hash]
```

  - Hashes can't be reversed – that's why we need to provide the shared vocab

```
# Raises an error if we haven't seen the string before
string = nlp.vocab.strings[3197928453018144401]
```

Shared vocab and string store (2)

- Look up the string and hash in `nlp.vocab.strings`

```
doc = nlp("I love coffee")
print('hash value:', nlp.vocab.strings['coffee'])
print('string value:', nlp.vocab.strings[3197928453018144401])
```

```
hash value: 3197928453018144401
string value: coffee
```

- The `doc` also exposes the vocab and strings

```
doc = nlp("I love coffee")
print('hash value:', doc.vocab.strings['coffee'])
```

```
hash value: 3197928453018144401
```

Lexemes: entries in the vocabulary

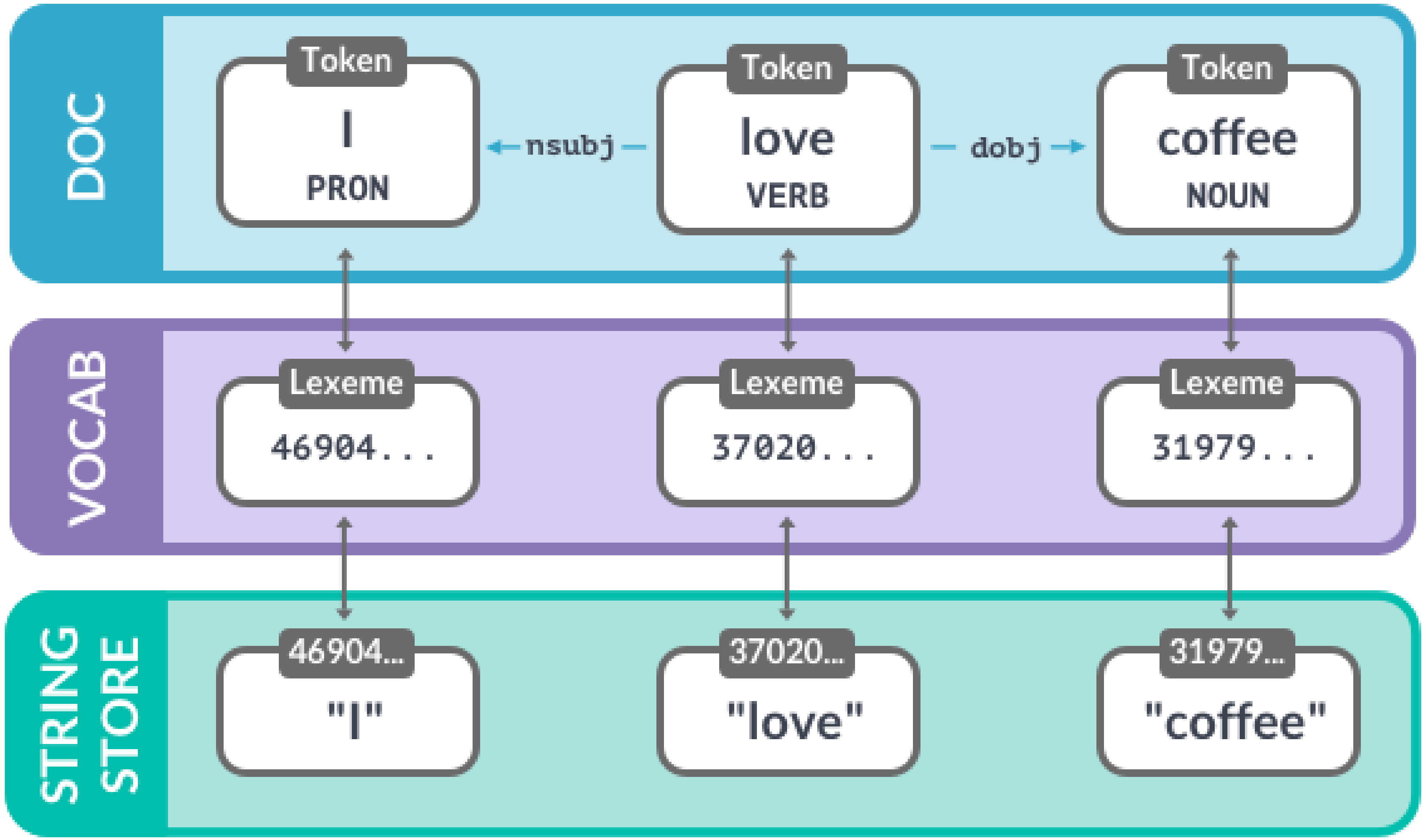
- A `Lexeme` object is an entry in the vocabulary

```
doc = nlp("I love coffee")
lexeme = nlp.vocab['coffee']

# print the lexical attributes
print(lexeme.text, lexeme.orth, lexeme.is_alpha)
```

```
coffee 3197928453018144401 True
```

- Contains the **context-independent** information about a word
  - Word text: `lexeme.text` and `lexeme.orth` (the hash)
  - Lexical attributes like `lexeme.is_alpha`
  - **Not** context-dependent part-of-speech tags, dependencies or entity labels

Vocab, hashes and lexemes



Let's practice!

ADVANCED NLP WITH SPACY

https://campus.datacamp.com/courses/advanced-nlp-with-spacy/large-scale-data-analysis-with-spacy?ex=4

02.04

Data Structures: Doc, Span and Token

ADVANCED NLP WITH SPACY

Ines Montani
spaCy core developer

The Doc object

The doc is one of the central data structures in spaCy.
It's created automatically when you process a text with the nlp object.
But you can also instantiate the class manually.

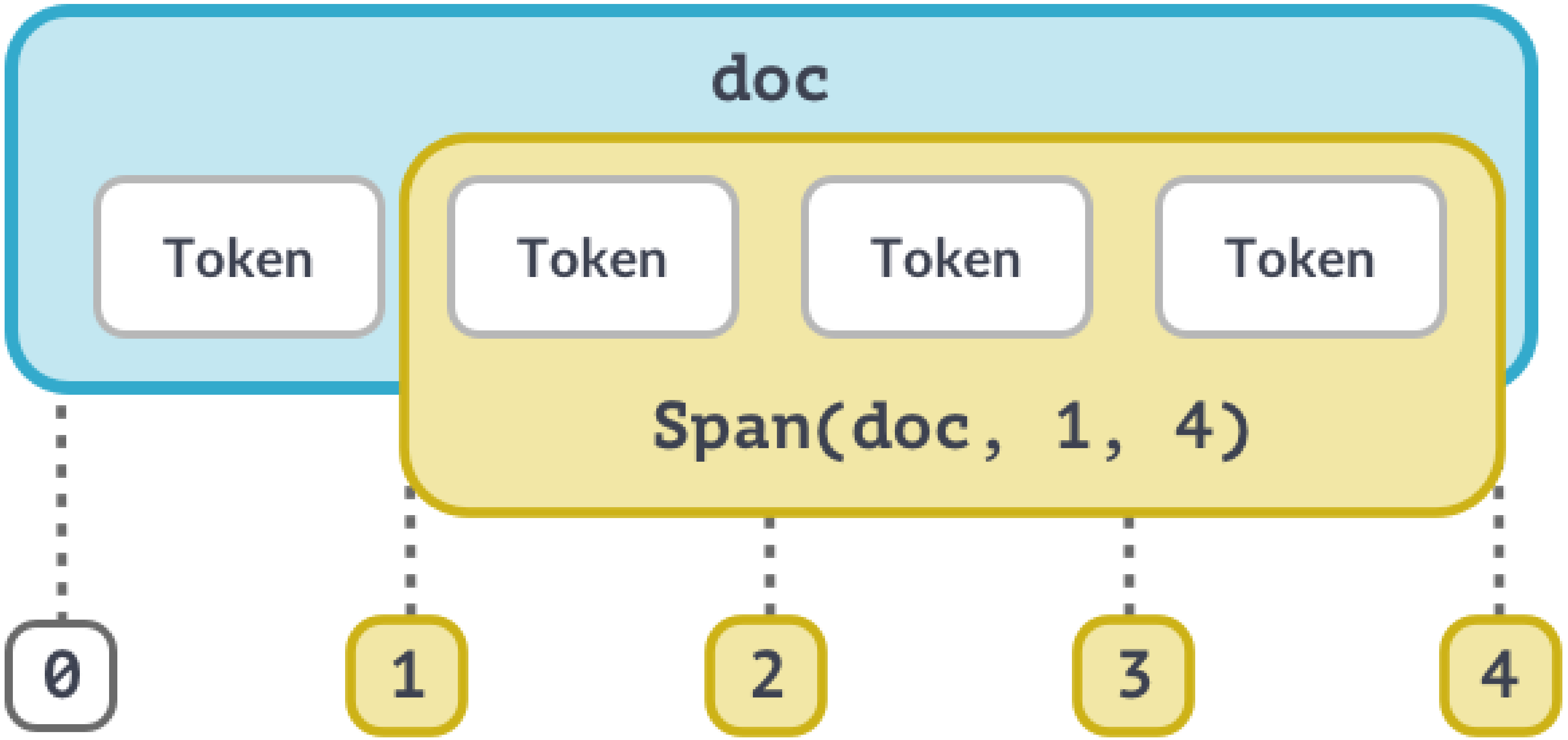
```
# Create an nlp object
from spacy.lang.en import English
nlp = English()

# Import the Doc class
from spacy.tokens import Doc

# The words and spaces to create the doc from
words = ['Hello', 'world', '!']
spaces = [True, False, False]

# Create a doc manually
doc = Doc(nlp.vocab, words=words, spaces=spaces)
```

The spaces are a list of boolean values indicating whether the word is followed by a space. Every token includes that information, even the last one!.

The DOC class takes three arguments: the shared vocab, the words and the spaces.

The Span object (1)

A Span is a slice of a Doc consisting of one or more tokens.
The SPAN takes at least three arguments: the doc it refers to, and the start and end index of the span. Remember that the end index is exclusive!



The Span object (2)

```
# Import the Doc and Span classes
from spacy.tokens import Doc, Span

# The words and spaces to create the doc from
words = ['Hello', 'world', '!']
spaces = [True, False, False]

# Create a doc manually
doc = Doc(nlp.vocab, words=words, spaces=spaces)
# Create a span manually
span = Span(doc, 0, 2)
# Create a span with a label
span_with_label = Span(doc, 0, 2, label="GREETING")
# Add span to the doc.ents
doc.ents = [span_with_label]
```

To add an entity label to the span, we can pass in the label name as the label argument. For consisting, we usually write label names in capital letters.
The doc.ents are writable, so we can add entities manually by overwritting it with a list of spans.

Best practices

- `Doc` and `Span` are very powerful and hold references and relationships of words and sentences
  - **Convert result to strings as late as possible** They are optimized for performance.
  - **Use token attributes if available** – for example, `token.i` for the token index
    To keep things consistent, try to use built-in token attributes wherever possible.
- Don't forget to pass in the shared `vocab`

Let's practice!

ADVANCED NLP WITH SPACY

https://campus.datacamp.com/courses/advanced-nlp-with-spacy/large-scale-data-analysis-with-spacy?ex=8

02.08

Word vectors and semantic similarity

ADVANCED NLP WITH SPACY

Ines Montani
spaCy core developer

Comparing semantic similarity

- `spaCy` can compare two objects and predict similarity
- `Doc.similarity()`, `Span.similarity()` and `Token.similarity()`
- Take another object and return a similarity score (`0` to `1`)
- **Important:** needs a model that has word vectors included, for example:
  - YES: `en_core_web_md` (medium model)
  - YES: `en_core_web_lg` (large model)
  - NO: `en_core_web_sm` (small model)

In this video, you'll learn how to use spaCy to predict how similar documents, spans or tokens are to each other.
You'll also learn about how to use word vectors and how to take advantage of them in your NLP application.

One thing that's very important: In order to use similarity, you need a larger spaCy model that has word vectors included.
So if you want to use vectors, always go with a model that ends in "md" or "lg". You can find more details on this in the models documentation.

Similarity examples (1)

Here's an example.
Let's say we want to find out whether two documents are similar.

```
# Load a larger model with vectors
nlp = spacy.load('en_core_web_md')

# Compare two documents
doc1 = nlp("I like fast food")
doc2 = nlp("I like pizza")
print(doc1.similarity(doc2))
```

```
0.8627204117787385
```

```
# Compare two tokens
doc = nlp("I like pizza and pasta")
token1 = doc[2]
token2 = doc[4]
print(token1.similarity(token2))
```

```
0.7369546
```

Similarity examples (2)

You can also use the similarity methods to compare different types of objects.
For example: a document and a token.

```
# Compare a document with a token
doc = nlp("I like pizza")
token = nlp("soap")[0]

print(doc.similarity(token))
```

```
0.32531983166759537
```

Here's another example comparing a span ("pizza and pasta") to a document.

```
# Compare a span with a document
span = nlp("I like pizza and pasta")[2:5]
doc = nlp("McDonalds sells burgers")

print(span.similarity(doc))
```

```
0.619909235817623
```

How does spaCy predict similarity?

- Similarity is determined using **word vectors**
- Multi-dimensional meaning representations of words
- Generated using an algorithm like **Word2Vec** and lots of text
- Can be added to spaCy's statistical models
- Default: cosine similarity, but can be adjusted
- `Doc` and `Span` vectors default to average of token vectors
- Short phrases are better than long documents with many irrelevant words

But how does spaCy do this under the hood?

You might have heard of Word2Vec, which is an algorithm that's often used to train word vectors from raw text.
Vectors can be added to spaCy's statistical models.

By default, the similarity returned by spaCy is the cosine similarity between two vectors, but this can be adjusted if necessary.

Vectors for objects consisting of several tokens, like the Doc and Span, default to the average of their token vectors.

That's also why you usually get more value out of shorter phrases with fewer irrelevant words.

To give you an idea of what those vectors look like, here's an example.

Word vectors in spaCy

```python
# Load a larger model with vectors
nlp = spacy.load('en_core_web_md')
doc = nlp("I have a banana")
# Access the vector via the token.vector attribute
print(doc[3].vector)
```

We look up a token's vector using the .vector attribute. The result is a 300-dimensional vector of the word "banana".

```
[2.02280000e-01,  -7.66180009e-02,   3.70319992e-01,
 3.28450017e-02,  -4.19569999e-01,   7.20689967e-02,
-3.74760002e-01,   5.74599989e-02,  -1.24009997e-02,
 5.29489994e-01,  -5.23800015e-01,  -1.97710007e-01,
-3.41470003e-01,   5.33169985e-01,  -2.53309999e-02,
 1.73800007e-01,   1.67720005e-01,   8.39839995e-01,
 5.51070012e-02,   1.05470002e-01,   3.78719985e-01,
 2.42750004e-01,   1.47449998e-02,   5.59509993e-01,
 1.25210002e-01,  -6.75960004e-01,   3.58420014e-01,
-4.00279984e-02,   9.59490016e-02,  -5.06900012e-01,
-8.53179991e-02,   1.79800004e-01,   3.38669986e-01,
 ...
```

Similarity depends on the application context

- Useful for many applications: recommendation systems, flagging duplicates etc.
- There's no objective definition of "similarity"
- Depends on the context and what application needs to do

```
doc1 = nlp("I like cats")
doc2 = nlp("I hate cats")

print(doc1.similarity(doc2))
```

```
0.9501447503553421
```

Predicting similarity can be useful for many types of applications.

For example, to recommend a user similar texts based on the ones they have read.

It can also be helpful to flag duplicate content, like posts on an online platform.
However, it's important to keep in mind that there's no objective definition of what's similar and what isn't. It always depends on the contex and what your application needs to do.
Here's an example: spaCy's default word vectors assign a very high similarity score to "I like cats" and "I hate cats". This makes sense, because both texts express sentiment about cats. But in a different application context, you might want to consider the phrases as very dissimilar, because they talk about opposite sentiments.

Let's practice!

ADVANCED NLP WITH SPACY

https://campus.datacamp.com/courses/advanced-nlp-with-spacy/large-scale-data-analysis-with-spacy?ex=11

02.11

Combining models and rules

ADVANCED NLP WITH SPACY

Ines Montani
spaCy core developer

Statistical predictions vs. rules

Combining statistical models with rule-based systems is one of the most powerful tricks you should have in your NLP toolbox.
In this video, we'll take a look at how to do it with spaCy.

	Statistical models	Rule-based systems
Use cases	application needs to *generalize* based on examples	
Real-world examples	product names, person names, subject/object relationships	
spaCy features	entity recognizer, dependency parser, part-of-speech tagger	

Statistical models are useful if your application needs to be able to generalize based on a few examples.

For instance, detecting product or person names usually benefits from a statistical model.
Instead of providing a list of all person names ever, your application will be able to predict whether a span of tokens is a person name.

Similarly, you can predict dependency labels to find subject/object relationships.
To do this, you would use spaCy's entity recognizer, dependency parser or part-of speech tagger.

Statistical predictions vs. rules

	Statistical models	Rule-based systems
Use cases	application needs to *generalize* based on examples	dictionary with finite number of examples
Real-world examples	product names, person names, subject/object relationships	countries of the world, cities, drug names, dog breeds
spaCy features	entity recognizer, dependency parser, part-of-speech tagger	tokenizer, `Matcher` , `PhraseMatcher`

Rule-based approaches on the other hand come in handy if there's a more or less finite number of instances you want to find.

For example, all countries or cities of the world, drug names or even dog breeds.

In spaCy, you can achieve this with custom tokenization rules, as well as the matcher and phrase matcher.

Recap: Rule-based Matching

In the last chapter, you learned how to use spaCyy's rule-based matcher to find complex patterns in your texts.

```
# Initialize with the shared vocab
from spacy.matcher import Matcher
matcher = Matcher(nlp.vocab)

# Patterns are lists of dictionaries describing the tokens
pattern = [{'LEMMA': 'love', 'POS': 'VERB'}, {'LOWER': 'cats'}]
matcher.add('LOVE_CATS', None, pattern)

# Operators can specify how often a token should be matched
pattern = [{'TEXT': 'very', 'OP': '+'}, {'TEXT': 'happy'}]

# Calling matcher on doc returns list of (match_id, start, end) tuples
doc = nlp("I love cats and I'm very very happy")
matches = matcher(doc)
```

Adding statistical predictions

```
matcher = Matcher(nlp.vocab)
matcher.add('DOG', None, [{'LOWER': 'golden'}, {'LOWER': 'retriever'}])
doc = nlp("I have a Golden Retriever")
for match_id, start, end in matcher(doc):
    span = doc[start:end]
    print('Matched span:', span.text)
    # Get the span's root token and root head token
    print('Root token:', span.root.text)
    print('Root head token:', span.root.head.text)
    # Get the previous token and its POS tag
    print('Previous token:', doc[start - 1].text, doc[start - 1].pos_)
```

We can get the span's root token. If the span consists of more than one token, this will be the token that decides the category of the phrase.

We can also find the head token of the root. This is the syntatic parent that governs the phrase - in this case, the verb "have".

```
Matched span: Golden Retriever
Root token: Retriever
Root head token: have
Previous token: a DET
```

Efficient phrase matching (1)

The phrase matcher is another helpful tool to find sequencees of words in your data. It performs a keyword search on the document, but instead of only finding strings, it gives you direct access to the tokens in context.

- `PhraseMatcher` like regular expressions or keyword search – but with access to the tokens!
- Takes `Doc` object as patterns
- More efficient and faster than the `Matcher`
- Great for matching large word lists

Efficient phrase matching (2)

```
from spacy.matcher import PhraseMatcher

matcher = PhraseMatcher(nlp.vocab)

pattern = nlp("Golden Retriever")
matcher.add('DOG', None, pattern)

doc = nlp("I have a Golden Retriever")

# iterate over the matches
for match_id, start, end in matcher(doc):
    # get the matched span
    span = doc[start:end]
    print('Matched span:', span.text)
```

Instead of a list of dictionaries, we pass in a Doc object as the pattern.

This lets us create a Span object for the matched tokens "Golden Retriever"
to analyse it in context.

```
Matched span: Golden Retriever
```

Let's practice!

ADVANCED NLP WITH SPACY