Natural Language Processing: Spacy Tutorial

Dr. Hafiz Khan

mkhan74@Kennesaw.edu

Computer Science, Kennesaw State University

https://ahafizk.github.io/

Outline

- Introduction to Spacy
  - Spacy V2 used in this tutorial
- Rule-based matching
- Large-scale data analysis
- Processing pipeline
- Scaling and performance

Introduction to Spacy

The nlp object

```
# Import the English language class
from spacy.lang.en import English

# Create the nlp object
nlp = English()
```

- contains the processing pipeline
- includes language-specific rules for tokenization etc.

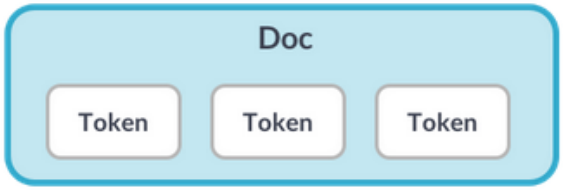
The Doc object

```
# Created by processing a string of text with the nlp object
doc = nlp("Hello world!")

# Iterate over tokens in a Doc
for token in doc:
    print(token.text)
```

```
Hello
world
!
```

- When you process a text with the nlp object, spaCy creates a Doc object – short for "document".
- The Doc lets you access information about the text in a structured way, and no information is lost.
- The Doc behaves like a normal Python sequence by the way and lets you iterate over its tokens, or get a token by its index. But more on that later!

The Token object



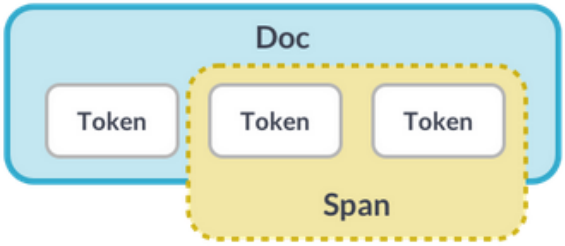

```
doc = nlp("Hello world!")

# Index into the Doc to get a single Token
token = doc[1]

# Get the token text via the .text attribute
print(token.text)
```

```
world
```

- Token objects represent the tokens in a document – for example, a word or a punctuation character.
- To get a token at a specific position, you can index into the doc.

- Token objects also provide various attributes that let you access more information about the tokens. For example, the .text attribute returns the verbatim token text.

The Span object




```
doc = nlp("Hello world!")

# A slice from the Doc is a Span object
span = doc[1:3]

# Get the span text via the .text attribute
print(span.text)
```

```
world!
```

- A Span object is a slice of the document consisting of one or more tokens. It's only a view of the Doc and doesn't contain any data itself.
- To create a span, you can use Python's slice notation. For example, 1:3 will create a slice starting from the token at position 1, up to – but not including! – the token at position 3.

Lexical Attributes

```
doc = nlp("It costs $5.")
```

```
print("Index:    ", [token.i for token in doc])
print("Text:     ", [token.text for token in doc])

print("is_alpha:", [token.is_alpha for token in doc])
print("is_punct:", [token.is_punct for token in doc])
print("like_num:", [token.like_num for token in doc])
```

```
Index:    [0, 1, 2, 3, 4]
Text:     ['It', 'costs', '$', '5', '.']

is_alpha: [True, True, False, False, False]
is_punct: [False, False, False, False, True]
like_num: [False, False, False, True, False]
```

- Here you can see some of the available token attributes:
  - i is the index of the token within the parent document.
  - text returns the token text.
  - is_alpha, is_punct and like_num return boolean values indicating whether the token consists of alphabetic characters, whether it's punctuation or whether it *resembles* a number. For example, a token "10" – one, zero – or the word "ten" – T, E, N.
- These attributes are also called lexical attributes: they refer to the entry in the vocabulary and don't depend on the token's context.

Exercise

```
# Import the English language class
from spacy.lang.____ import ____

# Create the nlp object
nlp = ____

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

# Print the document text
print(____.text)
```

Solution

```
# Import the English language class
from spacy.lang.en import English

# Create the nlp object
nlp = English()

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

# Print the document text
print(doc.text)
```

```
This is a sentence.
```

Exercise: Document, Span and token

- When you call nlp on a string, spaCy first tokenizes the text and creates a document object. In this exercise, you'll learn more about the Doc, as well as its views Token and Span.
- Step 1:
  - Import the English language class and create the nlp object.
  - Process the text and instantiate a Doc object in the variable doc

```
# Import the English language class and create the nlp object
from ____ import ____

nlp = ____

# Process the text
doc = ____("I like tree kangaroos and narwhals.")

# Select the first token
first_token = doc[____]

# Print the first token's text
print(first_token.____)
```

Exercise: Document, Span and token - Solution

```
# Import the English language class and create the nlp object
from spacy.lang.en import English

nlp = English()

# Process the text
doc = nlp("I like tree kangaroos and narwhals.")

# Select the first token
first_token = doc[0]

# Print the first token's text
print(first_token.text)
```

Exercise-2: Document, Span and token

- When you call nlp on a string, spaCy first tokenizes the text and creates a document object. In this exercise, you'll learn more about the Doc, as well as its views Token and Span.
- Step 2:
  - Import the English language class and create the nlp object.
  - Process the text and instantiate a Doc object in the variable doc.

```
# Import the English language class and create the nlp object
from spacy.lang.en import English

nlp = English()

# Process the text
doc = nlp("I like tree kangaroos and narwhals.")

# A slice of the Doc for "tree kangaroos"
tree_kangaroos = doc[2:4]
print(tree_kangaroos.text)

# A slice of the Doc for "tree kangaroos and narwhals" (without the ".")
tree_kangaroos_and_narwhals = doc[2:6]
print(tree_kangaroos_and_narwhals.text)
```

Exercise-2: Document, Span and token -- Solution

- When you call nlp on a string, spaCy first tokenizes the text and creates a document object. In this exercise, you'll learn more about the Doc, as well as its views Token and Span.
- Step 2:
  - Import the English language class and create the nlp object.
  - Process the text and instantiate a Doc object in the variable doc

```
# Import the English language class and create the nlp object
from ____ import ____

nlp = ____

# Process the text
doc = ____("I like tree kangaroos and narwhals.")

# A slice of the Doc for "tree kangaroos"
tree_kangaroos = ____
print(tree_kangaroos.text)

# A slice of the Doc for "tree kangaroos and narwhals" (without the ".")
tree_kangaroos_and_narwhals = ____
print(tree_kangaroos_and_narwhals.text)
```

Lexical Attribute: Example

- In this example, you'll use spaCy's Doc and Token objects, and lexical attributes to find percentages in a text.
- You'll be looking for two subsequent tokens: a number and a percent sign.
- Use the like_num token attribute to check whether a token in the doc resembles a number.
- Get the token *following* the current token in the document.
- The index of the next token in the doc is token.i + 1.
- Check whether the next token's text attribute is a percent sign "%".

Lexical attribute: Exercise

```
from spacy.lang.en import English

nlp = English()

# Process the text
doc = nlp(
    "In 1990, more than 60% of people in East Asia were in extreme poverty. "
    "Now less than 4% are."
)

# Iterate over the tokens in the doc
for token in doc:
    # Check if the token resembles a number
    if ____.____:
        # Get the next token in the document
        next_token = ____[____]
        # Check if the next token's text equals "%"
        if next_token.____ == "%":
            print("Percentage found:", token.text)
```

Lexical attribute: Solution

```
from spacy.lang.en import English

nlp = English()

# Process the text
doc = nlp(
    "In 1990, more than 60% of people in East Asia were in extreme poverty. "
    "Now less than 4% are."
)

# Iterate over the tokens in the doc
for token in doc:
    # Check if the token resembles a number
    if token.like_num:
        # Get the next token in the document
        next_token = doc[token.i + 1]
        # Check if the next token's text equals "%"
        if next_token.text == "%":
            print("Percentage found:", token.text)
```

Statistical Models

What are statistical models?

- Enable spaCy to predict linguistic attributes *in context*
  - Part-of-speech tags
  - Syntactic dependencies
  - Named entities

- Trained on labeled example texts
- Can be updated with more examples to fine-tune predictions

https://spacy.io/usage/models

Model Packages

```
$ python -m spacy download en_core_web_sm
```

```
import spacy

nlp = spacy.load("en_core_web_sm")
```

- Binary weights
- Vocabulary
- Meta information (language, pipeline)

- spaCy provides a number of pre-trained model packages you can download using the spacy download command. For example, the "en_core_web_sm" package is a small English model that supports all core capabilities and is trained on web text.
- The spacy.load method loads a model package by name and returns an nlp object.
- The package provides the binary weights that enable spaCy to make predictions.
- It also includes the vocabulary, and meta information to tell spaCy which language class to use and how to configure the processing pipeline.

Loading Models

- The models we're using in this course are already pre-installed. For more details on spaCy's statistical models and how to install them on your machine, see the documentation.
- Use spacy.load to load the small English model "en_core_web_sm".
- Process the text and print the document text.

- https://spacy.io/usage/models

Loading Models

```
import spacy

# Load the small English model
nlp = spacy.load("en_core_web_sm")

text = "It's official: Apple is the first U.S.\
public company to reach a $1 trillion market value"

# Process the text
doc = nlp(text)

# Print the document text
print(doc.text)
```

Predicting Linguistic Annotations

- spaCy's pre-trained model packages and see its predictions in action.
- To find out what a tag or label means,
  - you can call spacy.explain in the loop.
  - For example: spacy.explain("PROPN") or spacy.explain("GPE").

Exercise: Predicting Linguistic Annotations

```
import spacy

nlp = spacy.load("en_core_web_sm")

text = "It's official: Apple is the first U.S. public company\
to reach a $1 trillion market value"

# Process the text
doc = ____

for token in doc:
    # Get the token text, part-of-speech tag and dependency label
    token_text = ____.____
    token_pos = ____.____
    token_dep = ____.____
    # This is for formatting only
    print(f"{token_text:<12}{token_pos:<10}{token_dep:<10}")
```

Task:

- Process the text with the nlp object and create a doc.
- For each token, print the token text, the token's .pos_ (part-of-speech tag) and the token's .dep_ (dependency label).

Solution: Predicting Linguistic Annotations

```python
import spacy

nlp = spacy.load("en_core_web_sm")

text = "It's official: Apple is the first U.S. public company \
to reach a $1 trillion market value"

# Process the text
doc = nlp(text)

for token in doc:
    # Get the token text, part-of-speech tag and dependency label
    token_text = token.text
    token_pos = token.pos_
    token_dep = token.dep_
    # This is for formatting only
    print(f"{token_text:<12}{token_pos:<10}{token_dep:<10}")
```

```
It          PRON      nsubj
's          VERB      punct
official    NOUN      ROOT
:           PUNCT     punct
Apple       PROPN     nsubj
is          AUX       ROOT
the         DET       det
first       ADJ       amod
U.S.        PROPN     nmod
public      ADJ       amod
company     NOUN      attr
to          PART      aux
reach       VERB      relcl
a           DET       det
$           SYM       quantmod
1           NUM       compound
trillion    NUM       nummod
market      NOUN      compound
value       NOUN      dobj
```

Exercise: Predicting Linguistic Annotations

```
import spacy

nlp = spacy.load("en_core_web_sm")

text = "It's official: Apple is the first U.S. \
public company to reach a $1 trillion market value"

# Process the text
doc = nlp(text)

# Iterate over the predicted entities
for ent in doc.ents:
    # Print the entity text and its label
    print(ent.text, ent.label_)
```

Task 2:

- Process the text and create a doc object.
- Iterate over the doc.ents and print the entity text and label_ attribute.

```
Apple ORG
first ORDINAL
U.S. GPE
$1 trillion MONEY
```

Predicting Name Entities in Context

- Models are statistical and not *always* right. Whether their predictions are correct depends on the training data and the text you're processing. Let's take a look at an example.
  - Process the text with the nlp object.
  - Iterate over the entities and print the entity text and label.
  - Looks like the model didn't predict "iPhone X". Create a span for those tokens manually.

Predicting Name Entities in Context

```
import spacy
nlp = spacy.load("en_core_web_sm")

text = "Upcoming iPhone X release date leaked\
as Apple reveals pre-orders"

# Process the text
doc = nlp(text)

# Iterate over the entities
for ent in doc.ents:
    # Print the entity text and label
    print(ent.text, ent.label_)

# Get the span for "iPhone X"
iphone_x = doc[1:3]

# Print the span text
print("Missing entity:", iphone_x.text)
```

```
release date DATE
Apple ORG
Missing entity: iPhone X
```

Rule-based matching

Why not just regular expressions?

- Compared to regular expressions, the matcher works with Doc and Token objects instead of only strings.
- It's also more flexible: you can search for texts but also other lexical attributes.
- You can even write rules that use the model's predictions.
- For example, find the word "duck" only if it's a verb, not a noun.

Match patterns

- Lists of dictionaries, one per token
- Match exact token texts

```
[{"TEXT": "iPhone"}, {"TEXT": "X"}]
```

- Match lexical attributes

```
[{"LOWER": "iphone"}, {"LOWER": "x"}]
```

- Match any token attributes

```
[{"LEMMA": "buy"}, {"POS": "NOUN"}]
```

Match Patterns

```
import spacy

# Import the Matcher
from spacy.matcher import Matcher

# Load a model and create the nlp object
nlp = spacy.load("en_core_web_sm")

# Initialize the matcher with the shared vocab
matcher = Matcher(nlp.vocab)

# Add the pattern to the matcher
pattern = [{"TEXT": "iPhone"}, {"TEXT": "X"}]
matcher.add("IPHONE_PATTERN", None, pattern)

# Process some text
doc = nlp("Upcoming iPhone X release date leaked")

# Call the matcher on the doc
matches = matcher(doc)
```

- To use a pattern, we first import the matcher from spacy.matcher.
- We also load a model and create the nlp object.
- The matcher is initialized with the shared vocabulary, nlp.vocab. You'll learn more about this later – for now, just remember to always pass it in.
- The matcher.add method lets you add a pattern. The first argument is a unique ID to identify which pattern was matched. The second argument is an optional callback. We don't need one here, so we set it to None. The third argument is the pattern.
- To match the pattern on a text, we can call the matcher on any doc.
- This will return the matches.

Match Patterns

```
# Call the matcher on the doc
doc = nlp("Upcoming iPhone X release date leaked")
matches = matcher(doc)

# Iterate over the matches
for match_id, start, end in matches:
    # Get the matched span
    matched_span = doc[start:end]
    print(matched_span.text)
```

```
iPhone X
```

- `match_id` : hash value of the pattern name
- `start` : start index of matched span
- `end` : end index of matched span

- When you call the matcher on a doc, it returns a list of tuples.
- Each tuple consists of three values: the match ID, the start index and the end index of the matched span.
- This means we can iterate over the matches and create a Span object: a slice of the doc at the start and end index.

Matching lexical attributes

```
pattern = [
    {"IS_DIGIT": True},
    {"LOWER": "fifa"},
    {"LOWER": "world"},
    {"LOWER": "cup"},
    {"IS_PUNCT": True}
]
```

```
doc = nlp("2018 FIFA World Cup: France won!")
```

```
2018 FIFA World Cup:
```

- Here's an example of a more complex pattern using lexical attributes.
- We're looking for five tokens:
- A token consisting of only digits.
- Three case-insensitive tokens for "fifa", "world" and "cup".
- And a token that consists of punctuation.
- The pattern matches the tokens "2018 FIFA World Cup:".

Matching other token attributes

```
pattern = [
    {"LEMMA": "love", "POS": "VERB"},
    {"POS": "NOUN"}
]
```

```
doc = nlp("I loved dogs but now I love cats more.")
```

```
loved dogs
love cats
```

- In this example, we're looking for two tokens:
- A verb with the lemma "love", followed by a noun.
- This pattern will match "loved dogs" and "love cats".

Using operators and quantifiers (1)

```
pattern = [
    {"LEMMA": "buy"},
    {"POS": "DET", "OP": "?"},  # optional: match 0 or 1 times
    {"POS": "NOUN"}
]
```

```
doc = nlp("I bought a smartphone. Now I'm buying apps.")
```

```
bought a smartphone
buying apps
```

- Operators and quantifiers let you define how often a token should be matched. They can be added using the "OP" key.
- Here, the "?" operator makes the determiner token optional, so it will match a token with the lemma "buy", an optional article and a noun.

Using operators and quantifiers (2)

Example	Description
`{"OP": "!"}`	Negation: match 0 times
`{"OP": "?"}`	Optional: match 0 or 1 times
`{"OP": "+"}`	Match 1 or more times
`{"OP": "*"}`	Match 0 or more times

- "OP" can have one of four values:
- An "!" negates the token, so it's matched 0 times.
- A "?" makes the token optional, and matches it 0 or 1 times.
- A "+" matches a token 1 or more times.
- And finally, an "*" matches 0 or more times.
- Operators can make your patterns a lot more powerful, but they also add more complexity – so use them wisely.

Exercise: Using the Matcher

```
import spacy

# Import the Matcher
from spacy.____ import ____

nlp = spacy.load("en_core_web_sm")
doc = nlp("Upcoming iPhone X release date leaked as Apple reveals pre-orders")

# Initialize the Matcher with the shared vocabulary
matcher = ____(____.____)

# Create a pattern matching two tokens: "iPhone" and "X"
pattern = [____]

# Add the pattern to the matcher
____.____("IPHONE_X_PATTERN", None, ____)

# Use the matcher on the doc
matches = ____
print("Matches:", [doc[start:end].text for match_id, start, end in matches])
```

You'll be using the example from the previous exercise and write a pattern that can match the phrase "iPhone X" in the text.

Steps:

- Import the Matcher from spacy.matcher.
- Initialize it with the nlp object's shared vocab.
- Create a pattern that matches the "TEXT" values of two tokens: "iPhone" and "X".
- Use the matcher.add method to add the pattern to the matcher.
- Call the matcher on the doc and store the result in the variable matches.
- Iterate over the matches and get the matched span from the start to the end index.

Solution: Using the Matcher

```
# Import the Matcher
from spacy.matcher import Matcher

nlp = spacy.load("en_core_web_sm")
doc = nlp("Upcoming iPhone X release date leaked as Apple reveals pre-orders")

# Initialize the Matcher with the shared vocabulary
matcher = Matcher(nlp.vocab)

# Create a pattern matching two tokens: "iPhone" and "X"
pattern = [{"TEXT": "iPhone"}, {"TEXT": "X"}]

# Add the pattern to the matcher
matcher.add("IPHONE_X_PATTERN", None, pattern)

# Use the matcher on the doc
matches = matcher(doc)
print("Matches:", [doc[start:end].text for match_id, start, end in matches])
```

Writing Match Patterns

```
import spacy
from spacy.matcher import Matcher

nlp = spacy.load("en_core_web_sm")
matcher = Matcher(nlp.vocab)

doc = nlp(
    "After making the iOS update you won't notice a radical system-wide "
    "redesign: nothing like the aesthetic upheaval we got with iOS 7. Most of "
    "iOS 11's furniture remains the same as in iOS 10. But you will discover "
    "some tweaks once you delve a little deeper."
)

# Write a pattern for full iOS versions ("iOS 7", "iOS 11", "iOS 10")
pattern = [{"TEXT": ____}, {"IS_DIGIT": ____}]

# Add the pattern to the matcher and apply the matcher to the doc
matcher.add("IOS_VERSION_PATTERN", None, pattern)
matches = matcher(doc)
print("Total matches found:", len(matches))

# Iterate over the matches and print the span text
for match_id, start, end in matches:
    print("Match found:", doc[start:end].text)
```

Task: Write **one** pattern that only matches mentions of the *full* iOS versions: “iOS 7”, “iOS 11” and “iOS 10”.

Solution: Writing Match Patterns

```
import spacy
from spacy.matcher import Matcher

nlp = spacy.load("en_core_web_sm")
matcher = Matcher(nlp.vocab)

doc = nlp(
    "After making the iOS update you won't notice a radical system-wide "
    "redesign: nothing like the aesthetic upheaval we got with iOS 7. Most of "
    "iOS 11's furniture remains the same as in iOS 10. But you will discover "
    "some tweaks once you delve a little deeper."
)

# Write a pattern for full iOS versions ("iOS 7", "iOS 11", "iOS 10")
pattern = [{"TEXT": "iOS"}, {"IS_DIGIT": True}]

# Add the pattern to the matcher and apply the matcher to the doc
matcher.add("IOS_VERSION_PATTERN", None, pattern)
matches = matcher(doc)
print("Total matches found:", len(matches))

# Iterate over the matches and print the span text
for match_id, start, end in matches:
    print("Match found:", doc[start:end].text)
```

```
Total matches found: 3
Match found: iOS 7
Match found: iOS 11
Match found: iOS 10
```

Exercise 2: Writing Match Patterns

```
import spacy
from spacy.matcher import Matcher

nlp = spacy.load("en_core_web_sm")
matcher = Matcher(nlp.vocab)

doc = nlp(
    "i downloaded Fortnite on my laptop and can't open the game at all. Help? "
    "so when I was downloading Minecraft, I got the Windows version where it "
    "is the '.zip' folder and I used the default program to unpack it... do "
    "I also need to download Winzip?"
)

# Write a pattern that matches a form of "download" plus proper noun
pattern = [{"LEMMA": "download"}, {"POS": "PROPN"}]

# Add the pattern to the matcher and apply the matcher to the doc
matcher.add("DOWNLOAD_THINGS_PATTERN", None, pattern)
matches = matcher(doc)
print("Total matches found:", len(matches))

# Iterate over the matches and print the span text
for match_id, start, end in matches:
    print("Match found:", doc[start:end].text)
```

- Write **one** pattern that only matches forms of “download” (tokens with the lemma “download”), followed by a token with the part-of-speech tag "PROPN" (proper noun).

```
Total matches found: 2
Match found: downloaded Fortnite
Match found: downloading Minecraft
```

Exercise 3: Writing Match Patterns

```python
import spacy
from spacy.matcher import Matcher

nlp = spacy.load("en_core_web_sm")
matcher = Matcher(nlp.vocab)

doc = nlp(
    "Features of the app include a beautiful design, smart search, automatic "
    "labels and optional voice responses."
)

# Write a pattern for adjective plus one or two nouns
pattern = [{"POS": "ADJ"}, {"POS": "NOUN"}, {"POS": "NOUN", "OP": "?"}]

# Add the pattern to the matcher and apply the matcher to the doc
matcher.add("ADJ_NOUN_PATTERN", None, pattern)
matches = matcher(doc)
print("Total matches found:", len(matches))

# Iterate over the matches and print the span text
for match_id, start, end in matches:
    print("Match found:", doc[start:end].text)
```

```
Total matches found: 5
Match found: beautiful design
Match found: smart search
Match found: automatic labels
Match found: optional voice
Match found: optional voice responses
```

Large-scale data analysis with spaCy

Shared vocab and string store (1)

```
nlp.vocab.strings.add("coffee")
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]
```

- 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

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

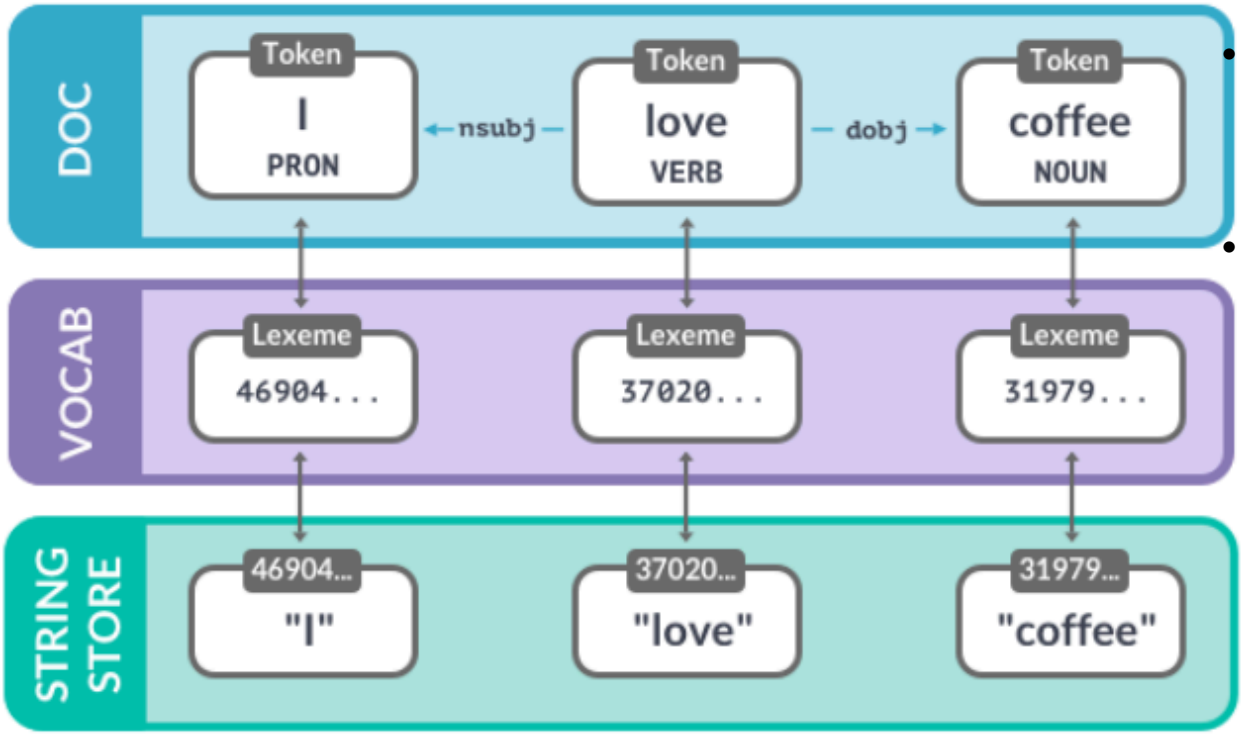
- A `Lexeme` object is an entry in the vocabulary

```
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



- The Doc contains words in context – in this case, the tokens "I", "love" and "coffee" with their part-of-speech tags and dependencies.
- Each token refers to a lexeme, which knows the word's hash ID. To get the string representation of the word, spaCy looks up the hash in the string store.

String to hashes

```
from spacy.lang.en import English

nlp = English()
doc = nlp("I have a cat")

# Look up the hash for the word "cat"
cat_hash = nlp.vocab.strings["cat"]
print(cat_hash)

# Look up the cat_hash to get the string
cat_string = nlp.vocab.strings[cat_hash]
print(cat_string)
```

- Look up the string “cat” in nlp.vocab.strings to get the hash.
- Look up the hash to get back the string.

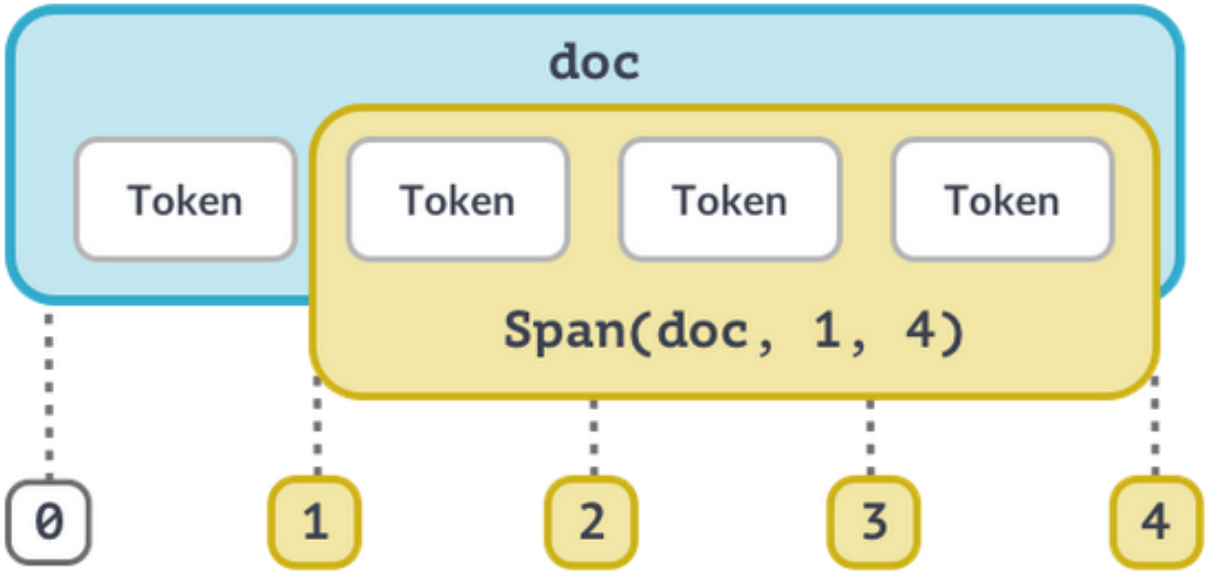
```
5439657043933447811
cat
```

Data Structure (2):Doc, Span, and Token

```
# 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 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.
- After creating the nlp object, we can import the Doc class from spacy.tokens.
- Here we're creating a doc from three words. 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 create a Span manually, we can also import the class from spacy.tokens. We can then instantiate it with the doc and the span's start and end index, and an optional label argument.
- The doc.ents are writable, so we can add entities manually by overwriting 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**
  - **Use token attributes if available** – for example, token.i for the token index
- Don't forget to pass in the shared vocab

Let's create some Doc objects from scratch!

```
from spacy.lang.en import English

nlp = English()

# Import the Doc class
from spacy.tokens import Doc

# Desired text: "spaCy is cool!"
words = ["spaCy", "is", "cool", "!"]
spaces = [True, True, False, False]

# Create a Doc from the words and spaces
doc = Doc(nlp.vocab, words=words, spaces=spaces)
print(doc.text)
```

- Import the Doc from spacy.tokens.
- Create a Doc from the words and spaces. Don't forget to pass in the vocab!

Docs, Spans and entities from scratch

```
from spacy.lang.en import English

nlp = English()

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

words = ["I", "like", "David", "Bowie"]
spaces = [True, True, True, False]

# Create a doc from the words and spaces
doc = Doc(nlp.vocab, words=words, spaces=spaces)
print(doc.text)

# Create a span for "David Bowie" from the doc and assign it the label "PERSON"
span = Span(doc, 2, 4, label="PERSON")
print(span.text, span.label_)

# Add the span to the doc's entities
doc.ents = [span]

# Print entities' text and labels
print([(ent.text, ent.label_) for ent in doc.ents])
```

- In this exercise, you'll create the Doc and Span objects manually, and update the named entities – just like spaCy does behind the scenes. A shared nlp object has already been created.
- Import the Doc and Span classes from spacy.tokens.
- Use the Doc class directly to create a doc from the words and spaces.
- Create a Span for "David Bowie" from the doc and assign it the label "PERSON".
- Overwrite the doc.ents with a list of one entity, the "David Bowie" span.

Word vectors and semantic similarity

- **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:
    - ✅ `en_core_web_md` (medium model)
    - ✅ `en_core_web_lg` (large model)
    - 🚫 NOT `en_core_web_sm` (small model)

Similarity examples (1)

```
# 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
```

- Here's an example. Let's say we want to find out whether two documents are similar.
- First, we load the medium English model, "en_core_web_md".
- We can then create two doc objects and use the first doc's similarity method to compare it to the second.
- Here, a fairly high similarity score of 0.86 is predicted for "I like fast food" and "I like pizza".
- The same works for tokens.
- According to the word vectors, the tokens "pizza" and "pasta" are kind of similar, and receive a score of 0.7.

Similarity examples (2)

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

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

```
0.32531983166759537
```

```
# 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
```

- You can also use the similarity methods to compare different types of objects.
- For example, a document and a token.
- Here, the similarity score is pretty low and the two objects are considered fairly dissimilar.
- Here's another example comparing a span – "pizza and pasta" – to a document about McDonalds.
- The score returned here is 0.61, so it's determined to be kind of similar.

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

Word vectors in spaCy

```
# 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)
```

```
[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,
 ...
```

- To give you an idea of what those vectors look like, here's an example.
- First, we load the medium model again, which ships with word vectors.
- Next, we can process a text and look up a token's vector using the .vector attribute.
- The result is a 300-dimensional vector of the word "banana".

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
```

Inspecting Word Vectors

```
import spacy

# Load the en_core_web_md model
nlp = spacy.load("en_core_web_md")

# Process a text
doc = nlp("Two bananas in pyjamas")

# Get the vector for the token "bananas"
bananas_vector = doc[1].vector
print(bananas_vector)
```

- In this exercise, you'll use a larger English model, which includes around 20.000 word vectors. The model is already pre-installed.
- Load the medium "en_core_web_md" model with word vectors.
- Print the vector for "bananas" using the token.vector attribute.

Comparing Similarity

```
import spacy

nlp = spacy.load("en_core_web_md")

doc1 = nlp("It's a warm summer day")
doc2 = nlp("It's sunny outside")

# Get the similarity of doc1 and doc2
similarity = doc1.similarity(doc2)
print(similarity)
```

- Use the doc.similarity method to compare doc1 to doc2 and print the result.
- Use the token.similarity method to compare token1 to token2 and print the result.
- Create spans for “great restaurant”/“really nice bar”.
- Use span.similarity to compare them and print the result.

Combining Models and rules

Statistical predictions vs. rules

	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 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`

Efficient phrase matching (2)

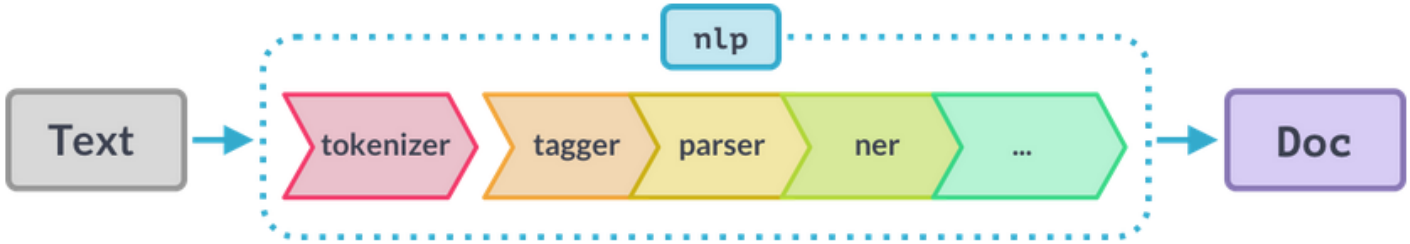
```
from spacy.matcher import PhraseMatcher

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)
```

```
Matched span: Golden Retriever
```

- The phrase matcher can be imported from spacy.matcher and follows the same API as the regular matcher.
- Instead of a list of dictionaries, we pass in a Doc object as the pattern.
- We can then iterate over the matches in the text, which gives us the match ID, and the start and end of the match. This lets us create a Span object for the matched tokens "Golden Retriever" to analyze it in context.

Processing Pipelines

What happens when you call nlp?




```
doc = nlp("This is a sentence.")
```

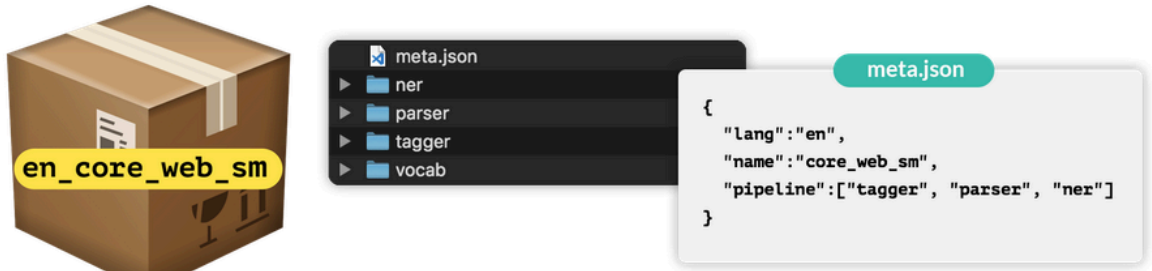
- First, the tokenizer is applied to turn the string of text into a Doc object.
- Next, a series of pipeline components is applied to the doc in order. In this case, the tagger, then the parser, then the entity recognizer.
- Finally, the processed doc is returned, so you can work with it.

Built-in pipeline component

Built-in pipeline components

Name	Description	Creates
tagger	Part-of-speech tagger	`Token.tag`, `Token.pos`
parser	Dependency parser	`Token.dep`, `Token.head`, `Doc.sents`, `Doc.noun_chunks`
ner	Named entity recognizer	`Doc.ents`, `Token.ent_iob`, `Token.ent_type`
textcat	Text classifier	`Doc.cats`

Under the hood



- Pipeline defined in model's `meta.json` in order
- Built-in components need binary data to make predictions

- All models you can load into spaCy include several files and a meta.json.
- The meta defines things like the language and pipeline. This tells spaCy which components to instantiate.
- The built-in components that make predictions also need binary data. The data is included in the model package and loaded into the component when you load the model.

Pipeline attribute

- `nlp.pipe_names` : list of pipeline component names

```
print(nlp.pipe_names)
```

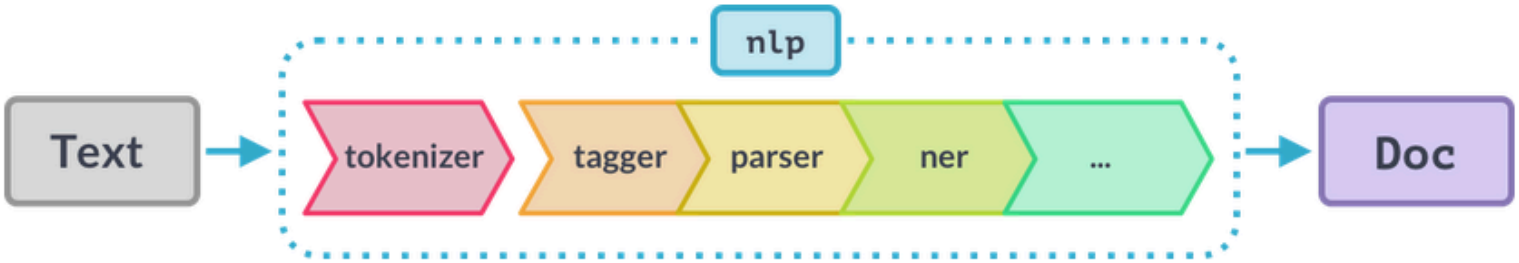
```
['tagger', 'parser', 'ner']
```

- `nlp.pipeline` : list of `(name, component)` tuples

```
print(nlp.pipeline)
```

```
[('tagger', <spacy.pipeline.Tagger>),
 ('parser', <spacy.pipeline.DependencyParser>),
 ('ner', <spacy.pipeline.EntityRecognizer>)]
```

Custom Pipeline Component



- Make a function execute automatically when you call `nlp`
- Add your own metadata to documents and tokens
- Updating built-in attributes like `doc.ents`

- After the text is tokenized and a Doc object has been created, pipeline components are applied in order. spaCy supports a range of built-in components, but also lets you define your own.
- Custom components are executed automatically when you call the nlp object on a text.
- They're especially useful for adding your own custom metadata to documents and tokens.
- You can also use them to update built-in attributes, like the named entity spans.

Anatomy of a component (1)

- Function that takes a `doc`, modifies it and returns it
- Can be added using the `nlp.add_pipe` method

```
def custom_component(doc):
    # Do something to the doc here
    return doc

nlp.add_pipe(custom_component)
```

- Fundamentally, a pipeline component is a function or callable that takes a doc, modifies it and returns it, so it can be processed by the next component in the pipeline.
- Components can be added to the pipeline using the nlp.add_pipe method. The method takes at least one argument: the component function.

Anatomy of a component (2)

```
def custom_component(doc):
    # Do something to the doc here
    return doc

nlp.add_pipe(custom_component)
```

Argument	Description	Example
`last`	If `True`, add last	`nlp.add_pipe(component, last=True)`
`first`	If `True`, add first	`nlp.add_pipe(component, first=True)`
`before`	Add before component	`nlp.add_pipe(component, before="ner")`
`after`	Add after component	`nlp.add_pipe(component, after="tagger")`

Example: A Simple Component

```
# Create the nlp object
nlp = spacy.load("en_core_web_sm")

# Define a custom component
def custom_component(doc):
    # Print the doc's length
    print("Doc length:", len(doc))
    # Return the doc object
    return doc

# Add the component first in the pipeline
nlp.add_pipe(custom_component, first=True)

# Print the pipeline component names
print("Pipeline:", nlp.pipe_names)
```

```
Pipeline: ['custom_component', 'tagger', 'parser', 'ner']
```

Setting custom attributes

- Add custom metadata to documents, tokens and spans
- Accessible via the `._` property

```
doc._.title = "My document"
token._.is_color = True
span._.has_color = False
```

- Registered on the global `Doc`, `Token` or `Span` using the `set_extension` method

```
# Import global classes
from spacy.tokens import Doc, Token, Span

# Set extensions on the Doc, Token and Span
Doc.set_extension("title", default=None)
Token.set_extension("is_color", default=False)
Span.set_extension("has_color", default=False)
```

- Custom attributes let you add any metadata to docs, tokens and spans. The data can be added once, or it can be computed dynamically.
- Custom attributes are available via the ._ (dot underscore) property. This makes it clear that they were added by the user, and not built into spaCy, like token.text.
- Attributes need to be registered on the global Doc, Token and Span classes you can import from spacy.tokens. You've already worked with those in the previous chapters. To register a custom attribute on the Doc, Token and Span, you can use the set_extension method.
- The first argument is the attribute name. Keyword arguments let you define how the value should be computed. In this case, it has a default value and can be overwritten.

Extension attribute types

- Attribute extensions
- Property extensions
- Method extensions

Attribute extensions

```
from spacy.tokens import Token

# Set extension on the Token with default value
Token.set_extension("is_color", default=False)

doc = nlp("The sky is blue.")

# Overwrite extension attribute value
doc[3]._.is_color = True
```

Set a default value that can be overwritten

Property extensions (1)

```
from spacy.tokens import Token

# Define getter function
def get_is_color(token):
    colors = ["red", "yellow", "blue"]
    return token.text in colors

# Set extension on the Token with getter
Token.set_extension("is_color", getter=get_is_color)

doc = nlp("The sky is blue.")
print(doc[3]._.is_color, "-", doc[3].text)
```

- Define a getter and an optional setter function
- Getter only called when you *retrieve* the attribute value

Property extensions

```
from spacy.tokens import Span

# Define getter function
def get_has_color(span):
    colors = ["red", "yellow", "blue"]
    return any(token.text in colors for token in span)

# Set extension on the Span with getter
Span.set_extension("has_color", getter=get_has_color)

doc = nlp("The sky is blue.")
print(doc[1:4]._.has_color, "-", doc[1:4].text)
print(doc[0:2]._.has_color, "-", doc[0:2].text)
```

```
True - sky is blue
False - The sky
```

Span extensions should almost always use a getter

Method extensions

```
from spacy.tokens import Doc

# Define method with arguments
def has_token(doc, token_text):
    in_doc = token_text in [token.text for token in doc]
    return in_doc

# Set extension on the Doc with method
Doc.set_extension("has_token", method=has_token)

doc = nlp("The sky is blue.")
print(doc._.has_token("blue"), "- blue")
print(doc._.has_token("cloud"), "- cloud")
```

```
True - blue
False - cloud
```

- Assign a **function** that becomes available as an object method
- Lets you pass **arguments** to the extension function

Scaling and performance

Processing large volumes of text

- Use `nlp.pipe` method
- Processes texts as a stream, yields `Doc` objects
- Much faster than calling `nlp` on each text

BAD:

```
docs = [nlp(text) for text in LOTS_OF_TEXTS]
```

GOOD:

```
docs = list(nlp.pipe(LOTS_OF_TEXTS))
```

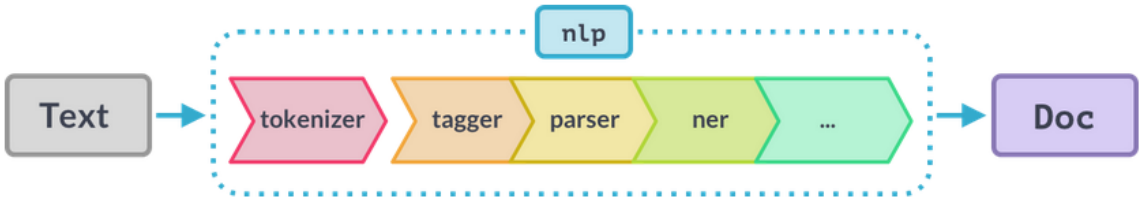
Passing in context

- Setting `as_tuples=True` on `nlp.pipe` lets you pass in `(text, context)` tuples
- Yields `(doc, context)` tuples
- Useful for associating metadata with the `doc`

```
data = [
    ("This is a text", {"id": 1, "page_number": 15}),
    ("And another text", {"id": 2, "page_number": 16}),
]

for doc, context in nlp.pipe(data, as_tuples=True):
    print(doc.text, context["page_number"])
```

```
This is a text 15
And another text 16
```

Using only tokenizer



- don't run the whole pipeline!
  - Use `nlp.make_doc` to turn a text into a `Doc` object

BAD:

```
doc = nlp("Hello world")
```

GOOD:

```
doc = nlp.make_doc("Hello world!")
```

- This is also how spaCy does it behind the scenes: nlp.make_doc turns the text into a doc before the pipeline components are called.

Disabling pipeline components

- Use `nlp.disable_pipes` to temporarily disable one or more pipes

```
# Disable tagger and parser
with nlp.disable_pipes("tagger", "parser"):
    # Process the text and print the entities
    doc = nlp(text)
    print(doc.ents)
```

- Restores them after the `with` block
- Only runs the remaining components

Training a neural network model

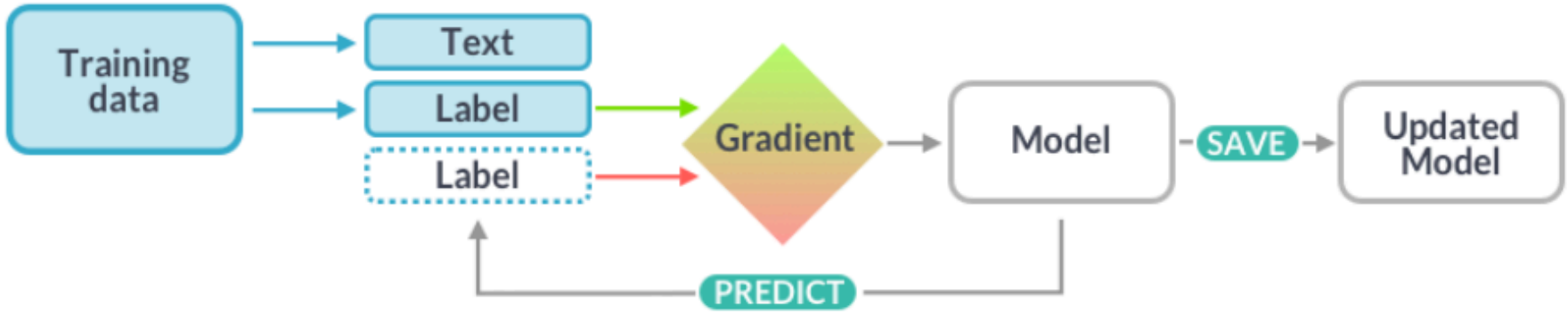
Training and updating models

- **Why updating the model?**
  - Better results on your specific domain
  - Learn classification schemes specifically for your problem
  - Essential for text classification
  - Very useful for named entity recognition
  - Less critical for part-of-speech tagging and dependency parsing

How training works

1. **Initialize** the model weights randomly with nlp.begin_training
2. **Predict** a few examples with the current weights by calling nlp.update
3. **Compare** prediction with true labels
4. **Calculate** how to change weights to improve predictions
5. **Update** weights slightly
6. Go back to 2.

How training works



- **Training data:** Examples and their annotations.
- **Text:** The input text the model should predict a label for.
- **Label:** The label the model should predict.
- **Gradient:** How to change the weights.

Example: Training the entity recognizer

- The entity recognizer tags words and phrases in context
- Each token can only be part of one entity
- Examples need to come with context

```
("iPhone X is coming", {"entities": [(0, 8, "GADGET")]})
```

- Texts with no entities are also important

```
("I need a new phone! Any tips?", {"entities": []})
```

- **Goal:** teach the model to generalize

The training data

- Examples of what we want the model to predict in context
- Update an **existing model**: a few hundred to a few thousand examples
- Train a **new category**: a few thousand to a million examples
  - spaCy's English models: 2 million words
- Usually created manually by human annotators
- Can be semi-automated – for example, using spaCy's Matcher!

The Training Loop

1. **Loop** for a number of times.
2. **Shuffle** the training data.
3. **Divide** the data into batches.
4. **Update** the model for each batch.
5. **Save** the updated model.

Example loop

```
TRAINING_DATA = [
    ("How to preorder the iPhone X", {"entities": [(20, 28, "GADGET")]})
    # And many more examples...
]
```

```
# Loop for 10 iterations
for i in range(10):
    # Shuffle the training data
    random.shuffle(TRAINING_DATA)
    # Create batches and iterate over them
    for batch in spacy.util.minibatch(TRAINING_DATA):
        # Split the batch in texts and annotations
        texts = [text for text, annotation in batch]
        annotations = [annotation for text, annotation in batch]
        # Update the model
        nlp.update(texts, annotations)

# Save the model
nlp.to_disk(path_to_model)
```

Updating an existing model

- Improve the predictions on new data
- Especially useful to improve existing categories, like "PERSON"
- Also possible to add new categories
- Be careful and make sure the model doesn't "forget" the old ones

Setting up a new pipeline from scratch

```
# Start with blank English model
nlp = spacy.blank("en")
# Create blank entity recognizer and add it to the pipeline
ner = nlp.create_pipe("ner")
nlp.add_pipe(ner)
# Add a new label
ner.add_label("GADGET")

# Start the training
nlp.begin_training()
# Train for 10 iterations
for itn in range(10):
    random.shuffle(examples)
    # Divide examples into batches
    for batch in spacy.util.minibatch(examples, size=2):
        texts = [text for text, annotation in batch]
        annotations = [annotation for text, annotation in batch]
        # Update the model
        nlp.update(texts, annotations)
```

Best Practice – Model training

- **Problem 1: Models can "forget" things**
  - Existing model can overfit on new data
    - e.g.: if you only update it with "WEBSITE", it can "unlearn" what a "PERSON" is
  - Also known as "catastrophic forgetting" problem
- **Solution 1: Mix in previously correct predictions**
  - For example, if you're training "WEBSITE", also include examples of "PERSON"
  - Run existing spaCy model over data and extract all other relevant entities

BAD:

```
TRAINING_DATA = [
    ("Reddit is a website", {"entities": [(0, 6, "WEBSITE")]})
]
```

GOOD:

```
TRAINING_DATA = [
    ("Reddit is a website", {"entities": [(0, 6, "WEBSITE")]}),
    ("Obama is a person", {"entities": [(0, 5, "PERSON")]})
]
```

Model training -- more

- More about model training
  - https://spacy.io/usage/training

Text Classification

- https://www.machinelearningplus.com/nlp/custom-text-classification-spacy/
- https://www.dataquest.io/blog/tutorial-text-classification-in-python-using-spacy/
- https://www.kaggle.com/poonaml/text-classification-using-spacy

- Readings:
- https://course.spacy.io/en/