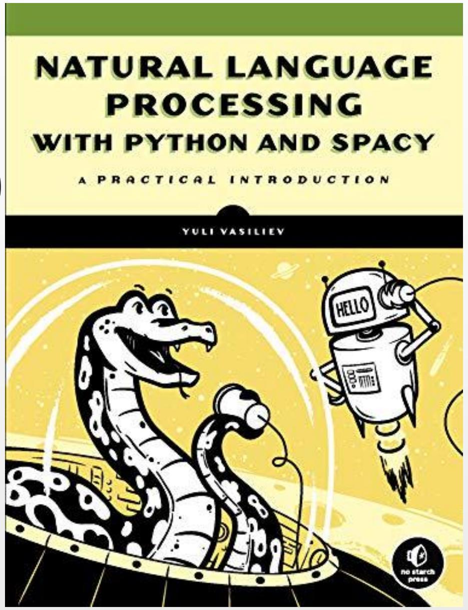
spaCy Lib for NLP

- Gain Meaning from Unstructured Text



Chapter 1: Finding words, phrases, names and concepts

ADVANCED NLP with 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.

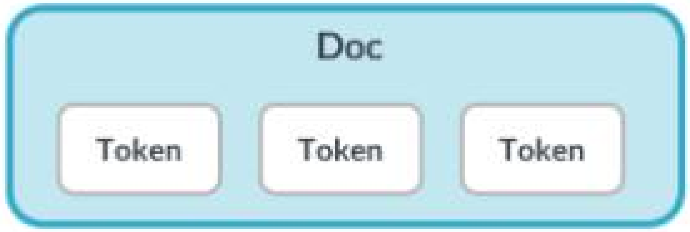
ADVANCED NLP with spaCy

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)
```

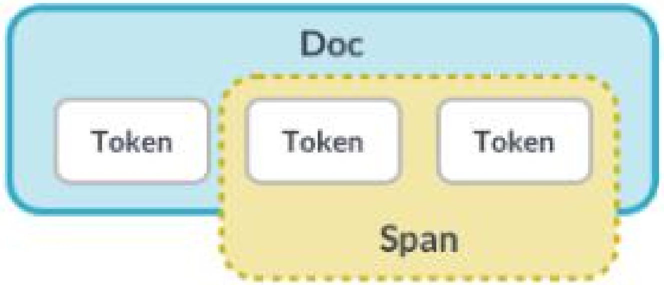
```
Hello
world
!
```

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
```

The Span 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
```

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]
```

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

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)

Predicting Part-of-speech Tags

```
import spacy

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

# Process a text
doc = nlp("She ate the pizza")

# Iterate over the tokens
for token in doc:
    # Print the text and the predicted part-of-speech tag
    print(token.text, token.pos_)
```

```
She PRON
ate VERB
the DET
pizza NOUN
```

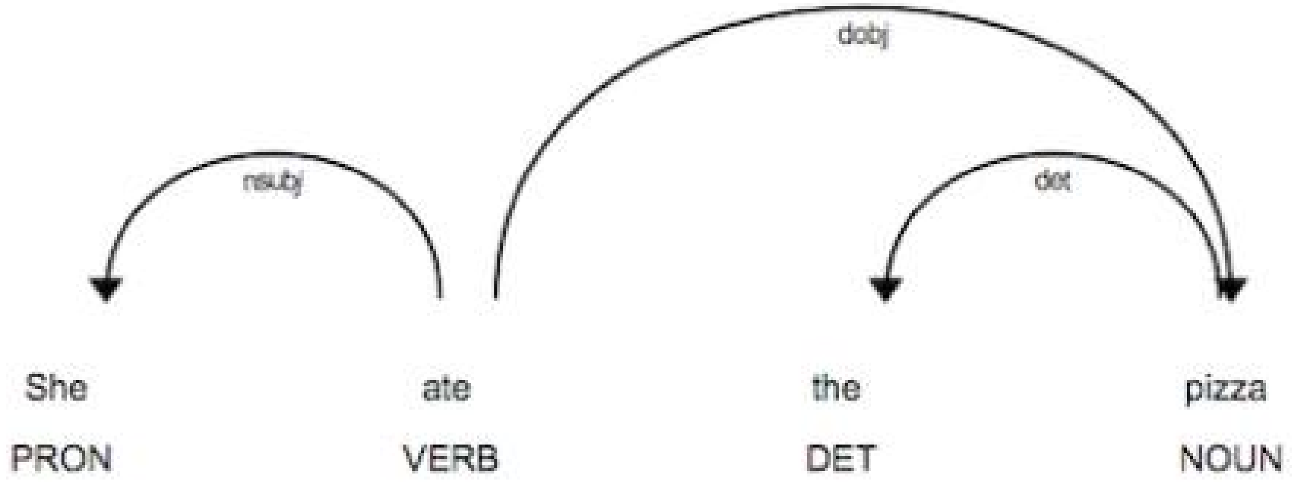
Predicting Syntactic Dependencies

```
for token in doc:
    print(token.text, token.pos_, token.dep_, token.head.text)
```

```
She PRON nsubj ate
ate VERB ROOT ate
the DET det pizza
pizza NOUN dobj ate
```

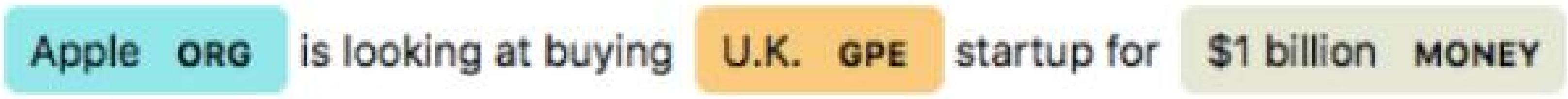
ADVANCED NLP with spaCy

Dependency label scheme



Label	Description	Example
nsubj	nominal subject	She
dobj	direct object	pizza
det	determiner (article)	the

Predicting Named Entities




```
# Process a text
doc = nlp("Apple is looking at buying U.K. startup for $1 billion")

# Iterate over the predicted entities
for ent in doc.ents:
    # Print the entity text and its label
    print(ent.text, ent.label_)
Apple ORG
U.K. GPE
$1 billion MONEY
```

Tip: the spacy.explain method

Get quick definitions of the most common tags and labels.

```
spacy.explain("GPE")
'Countries, cities, states'

spacy.explain("NNP")
'noun, proper singular'

spacy.explain("dobj")
'direct object'
```

Rule-based matching

Why not just regular expressions?

- Match on Doc objects, not just strings
- Match on tokens and token attributes
- Use the model's predictions
- Example: "duck" (verb) vs. "duck" (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"}]

Using the Matcher (1)

```
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)
```

Using the Matcher (2)

```
# 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

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:
```

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
```

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
```

ADVANCED NLP with spaCy

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

```
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)
```

```
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)
```

```
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": ____}, {"POS": ____}, {"POS": ____, "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)
```

Chapter 2: Large-scale data analysis with spaCy

Data Structures (1): Vocab, Lexemes and StringStore

ADVANCED NLP with spaCy

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

ADVANCED NLP with spaCy

Lexemes: entries in the vocabulary

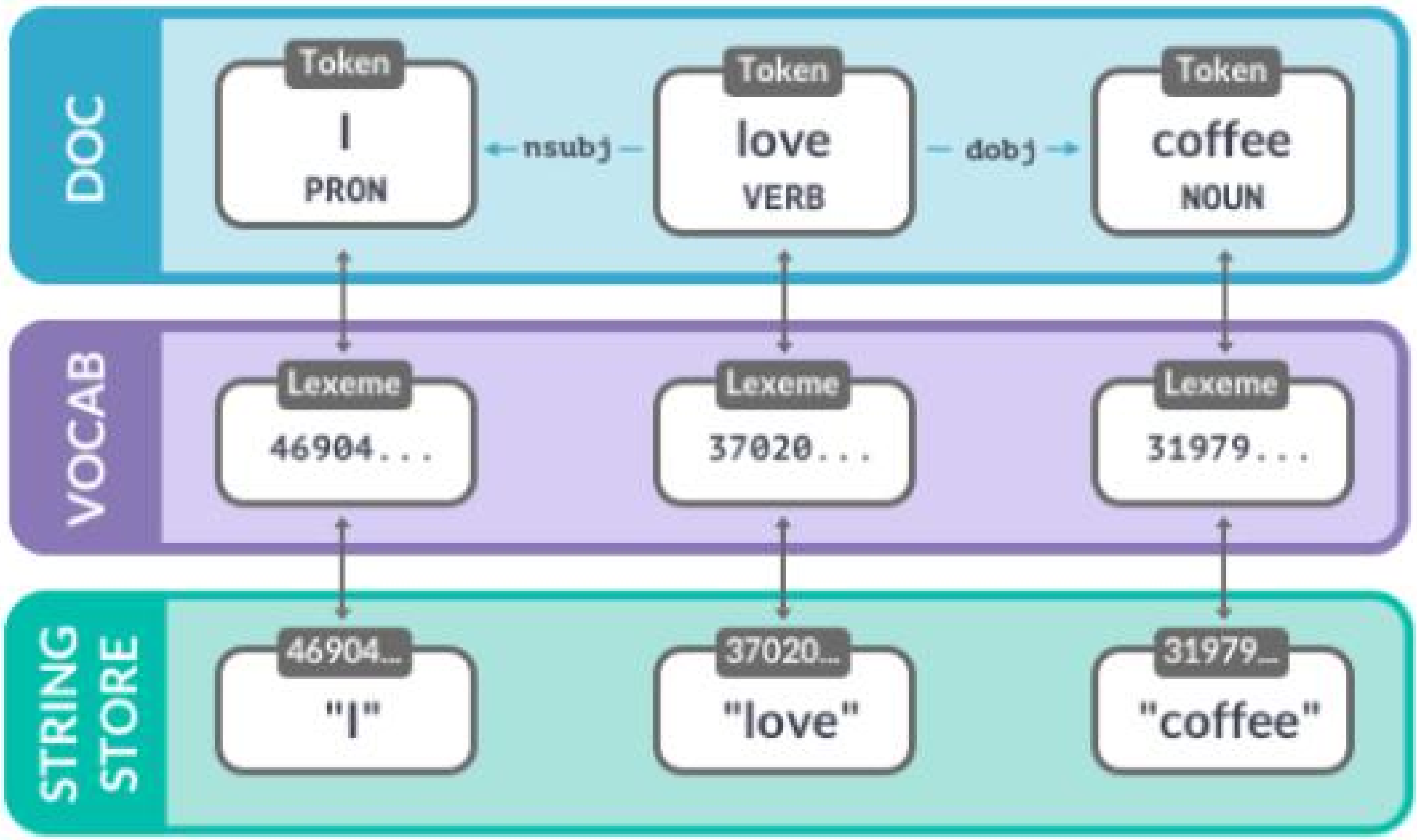
- A Lexeme object is an entry in the vocabulary

```
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



Vocab, hashes and lexemes

```
from spacy.lang.en import English

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

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

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

Vocab, hashes and lexemes

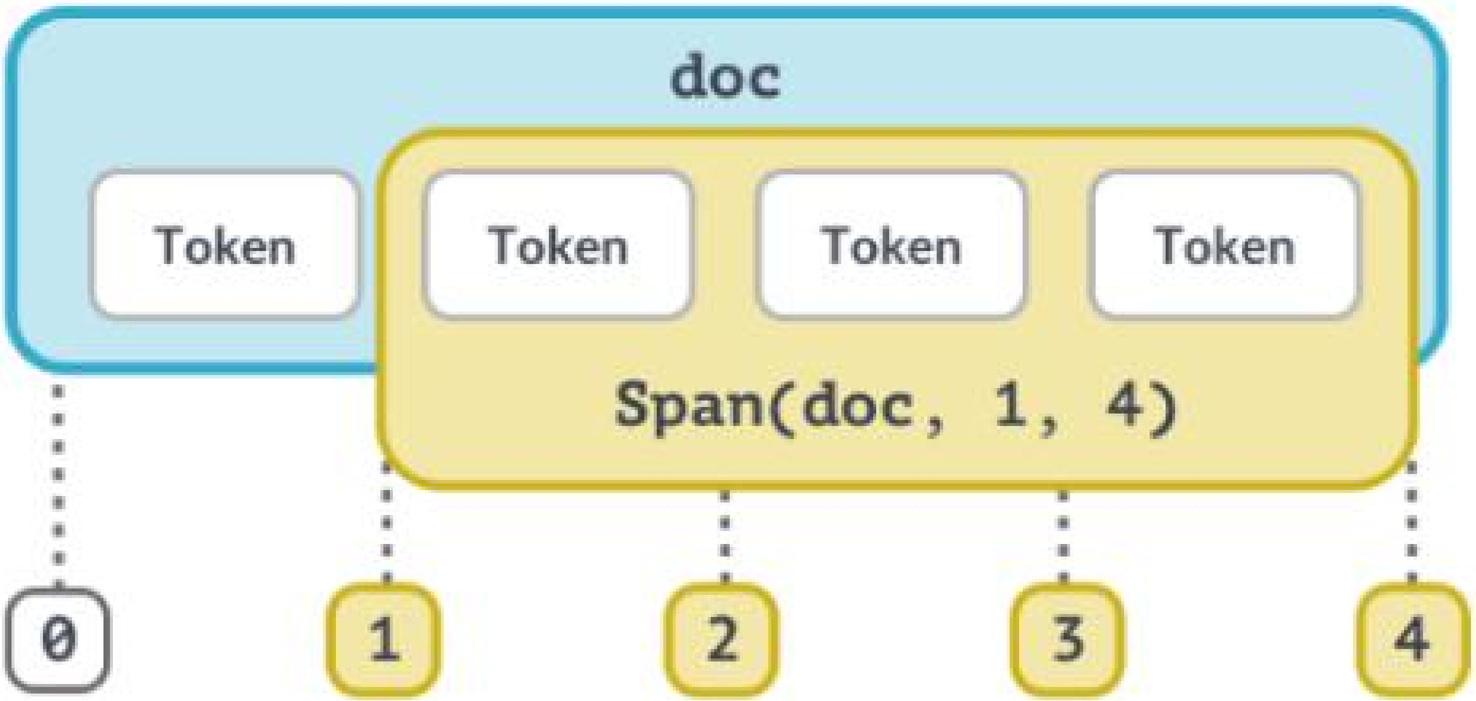
```
import spacy

nlp = spacy.load("en_core_web_sm")
doc = nlp("David Bowie is a PERSON")

# Look up the hash for the string label "PERSON"
person_hash = nlp.vocab.strings["PERSON"]
print(person_hash)

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

Data Structures (2): Doc, Span and Token

The Doc object

```
# 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 Span object (1)



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]
```

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

Best practices

```
import spacy

nlp = spacy.load("en_core_web_sm")
doc = nlp("Berlin is a nice city")

# Iterate over the tokens
for token in doc:
    # Check if the current token is a proper noun
    if token.pos_ == "PROPN":
        # Check if the next token is a verb
        if doc[token.i + 1].pos_ == "VERB":
            print("Found proper noun before a verb:", token.text)
```

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
```

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

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

ADVANCED NLP with spaCy

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,
   ...
```

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)
```

Combining models and rules

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`

Recap: Rule-based Matching

```
# 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"}]
matcher.add("VERY_HAPPY", None, pattern)

# 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_)
```

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

Efficient phrase matching (1)

- 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

ADVANCED NLP with spaCy

Efficient phrase matching (2)

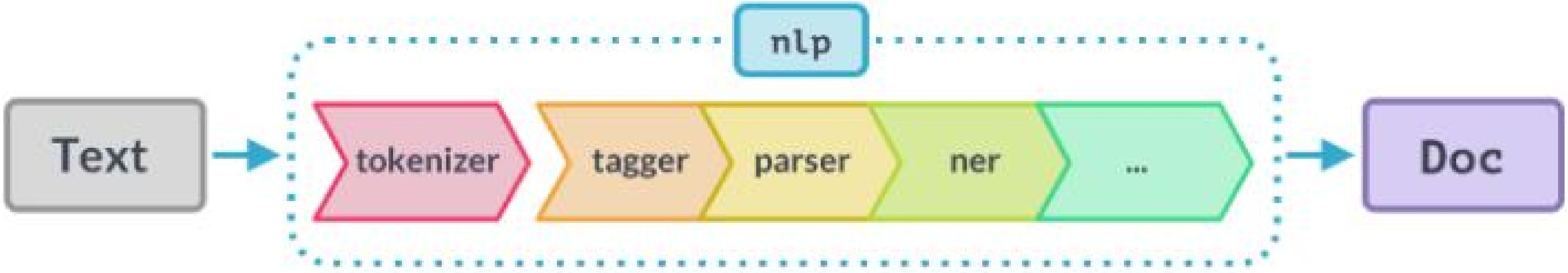
```
from spacy.matcher import PhraseMatcher

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
```

```
import json
from spacy.lang.en import English
with open("exercises/en/countries.json") as f:
    COUNTRIES = json.loads(f.read())

nlp = English()
doc = nlp("Czech Republic may help Slovakia protect its airspace")

# Import the PhraseMatcher and initialize it
from spacy.matcher import PhraseMatcher
matcher = PhraseMatcher(nlp.vocab)

# Create pattern Doc objects and add them to the matcher
# This is the faster version of: [nlp(country) for country in COUNTRIES]
patterns = list(nlp.pipe(COUNTRIES))
matcher.add("COUNTRY", None, *patterns)

# Call the matcher on the test document and print the result
matches = matcher(doc)
print([doc[start:end] for match_id, start, end in matches])
```

Chapter 3: Processing Pipelines

Processing pipelines

What happens when you call nlp?




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

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

Pipeline attributes

- nlp.pipe_names: list of pipeline component names

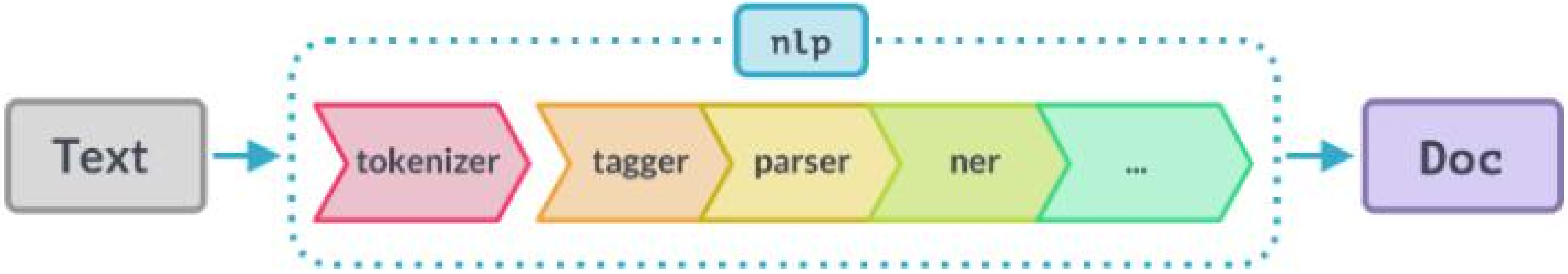
```
print(nlp.pipe_names)
['tagger', 'parser', 'ner']
```

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

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

Custom pipeline components

Why custom components?



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

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)
```

ADVANCED NLP with spaCy

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 (1)

```
# 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']

Example: a simple component (2)

```
# 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)

# Process a text
doc = nlp("Hello world!")
Doc length: 3
```

Extension attributes

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)
```

Extension attribute types

1. Attribute extensions
2. Property extensions
3. Method extensions

Attribute extensions

- Set a default value that can be overwritten

```
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
```

Property extensions (1)

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

```
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)
```

True - blue

Property extensions (2)

- Span extensions should almost always use a getter

```
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

Method extensions

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

```
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

```
import spacy
from spacy.tokens import Span
nlp = spacy.load("en_core_web_sm")

def get_wikipedia_url(span):
    # Get a Wikipedia URL if the span has one of the labels
    if span.label_ in ("PERSON", "ORG", "GPE", "LOCATION"):
        entity_text = span.text.replace(" ", "_")
        return "https://en.wikipedia.org/w/index.php?search=" + entity_text

# Set the Span extension wikipedia_url using get getter get_wikipedia_url
Span.set_extension("wikipedia_url", getter=get_wikipedia_url)

doc = nlp(
    "In over fifty years from his very first recordings right through to his "
    "last album, David Bowie was at the vanguard of contemporary culture."
)
for ent in doc.ents:
    # Print the text and Wikipedia URL of the entity
    print(ent.text, ent._.wikipedia_url)
```

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))
```

ADVANCED NLP with spaCy

Passing in context (1)

- 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"])
```

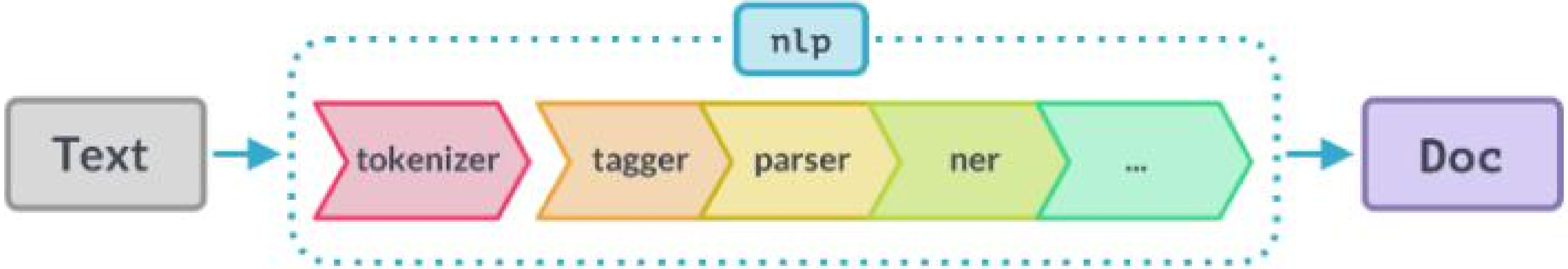
This is a text 15

And another text 16

Passing in context (2)

```
from spacy.tokens import Doc

Doc.set_extension("id", default=None)
Doc.set_extension("page_number", default=None)

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):
    doc._.id = context["id"]
    doc._.page_number = context["page_number"]
```

Using only the tokenizer (1)



- don't run the whole pipeline!

Using only the tokenizer (2)

Use nlp.make_doc to turn a text into a Doc object

BAD:

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

GOOD:

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

ADVANCED NLP with spaCy

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

Chapter 4: Training a neural network model

ADVANCED NLP with spaCy

Training and updating models

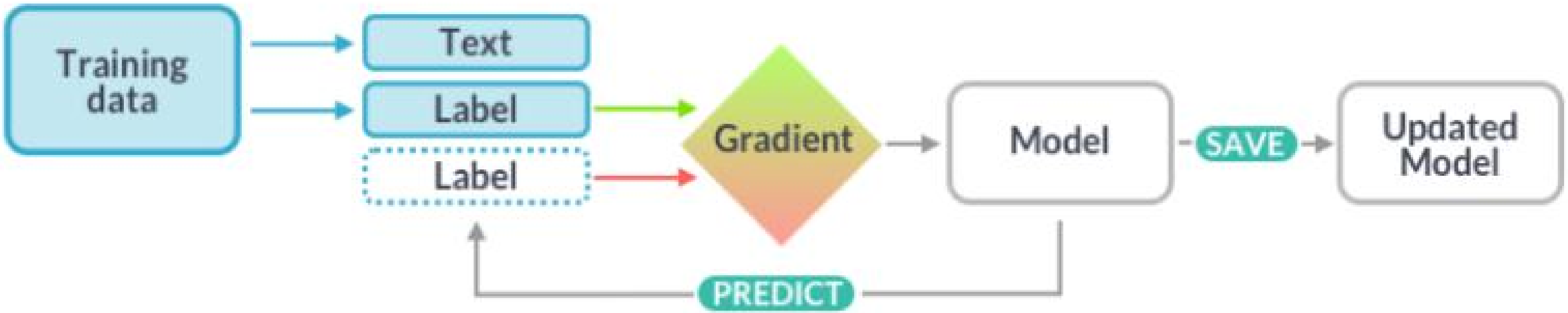
Why updating the model?

- 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)

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 (2)



- 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!

ADVANCED NLP with spaCy

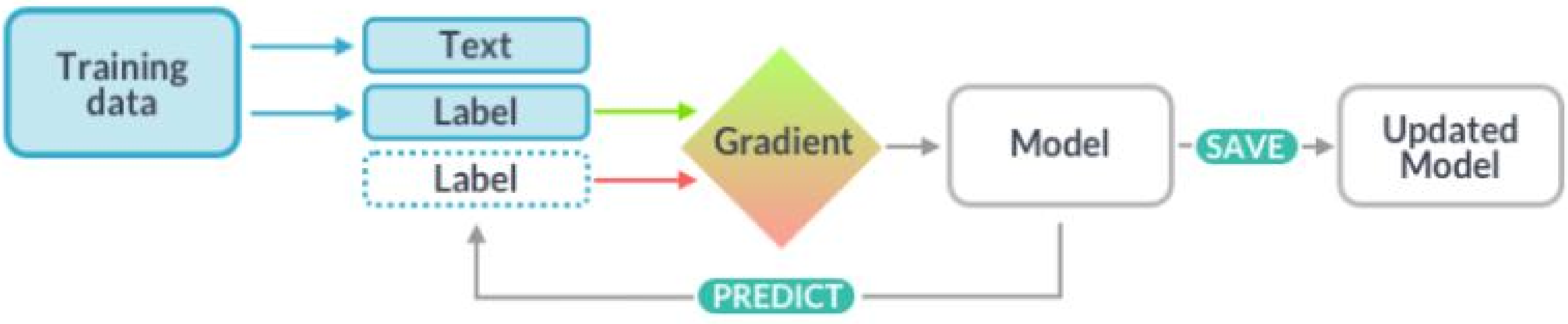
The training loop

ADVANCED NLP with spaCy

The steps of a 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.

Recap: 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 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

ADVANCED NLP with spaCy

Setting up a new pipeline from scratch

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

# 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 practices for training spaCy models

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")]})
```

Problem 2: Models can't learn everything

- spaCy's models make predictions based on local context
- Model can struggle to learn if decision is difficult to make based on context
- Label scheme needs to be consistent and not too specific
  - For example: "CLOTHING" is better than "ADULT_CLOTHING" and "CHILDRENS_CLOTHING"

Solution 2: Plan your label scheme carefully

- Pick categories that are reflected in local context
- More generic is better than too specific
- Use rules to go from generic labels to specific categories

BAD:

LABELS = ["ADULT_SHOES", "CHILDRENS_SHOES", "BANDS_I_LIKE"]

GOOD:

LABELS = ["CLOTHING", "BAND"]

Wrapping up

Your new spaCy skills

- Extract linguistic features: part-of-speech tags, dependencies, named entities
- Work with pre-trained statistical models
- Find words and phrases using Matcher and PhraseMatcher match rules
- Best practices for working with data structures Doc, Token Span, Vocab, Lexeme
- Find semantic similarities using word vectors
- Write custom pipeline components with extension attributes
- Scale up your spaCy pipelines and make them fast
- Create training data for spaCy' statistical models
- Train and update spaCy's neural network models with new data

More things to do with spaCy (1)

- Training and updating other pipeline components
  - Part-of-speech tagger
  - Dependency parser
  - Text classifier

ADVANCED NLP with spaCy

More things to do with spaCy (2)

- Customizing the tokenizer
    - Adding rules and exceptions to split text differently
- Adding or improving support for other languages
    - 55+ languages currently
    - Lots of room for improvement and more languages
    - Allows training models for other languages