Big Data and Data Mining

Text Mining

Fenerbahce University

Instructors

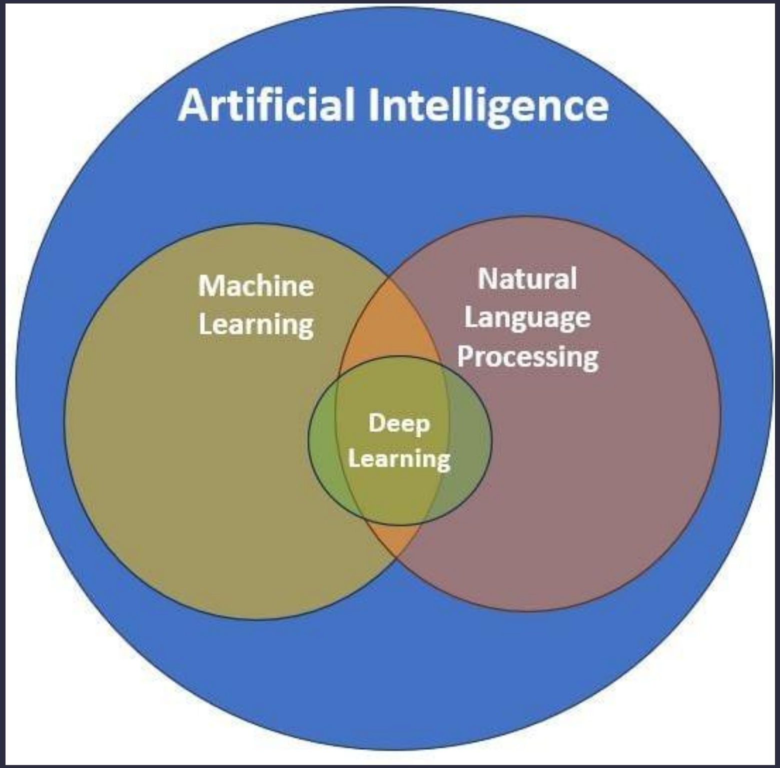
Assist. Prof. Vecdi Emre Levent

Office: 311

Email : emre.levent@fbu.edu.tr

Natural Language Processing (NLP)

- A subfield of **Artificial Intelligence (AI)**
- Helps computers to understand **human language**
- Helps extract insights from unstructured data
- Incorporates **statistics**, **machine learning models** and **deep learning models**



NLP use cases

Sentiment analysis

Use of computers to determine the underlying subjective tone of a piece of writing

NLP use cases

Named entity recognition (NER)

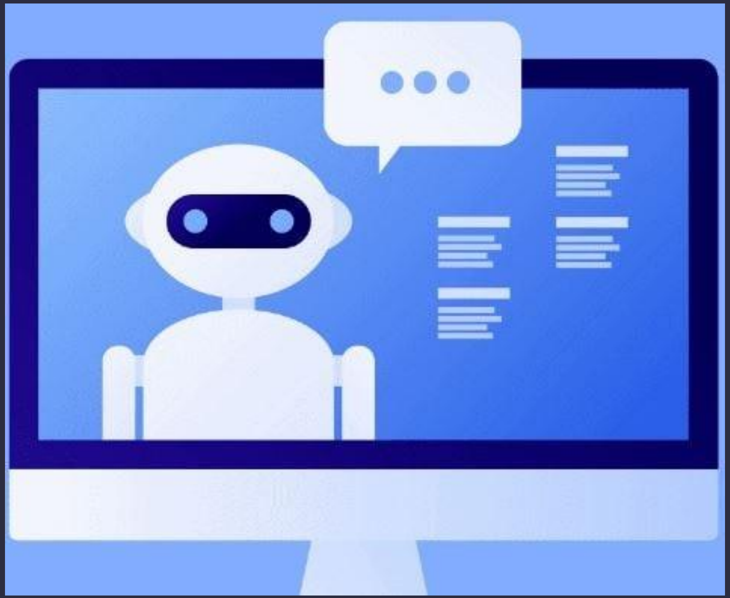
- Locating and classifying named entities mentioned in unstructured text into pre-defined categories
- **Named entities** are

  real-world objects

  such as a person or location

NLP use cases

- Generate human-like responses to text input, such as **ChatGPT**



Introduction to spaCy

spaCy is a **free**, **open-source** library for NLP in **Python** which:

- Is designed to build systems for **information extraction**
- Provides **production-ready** code for NLP use cases
- Supports **64+** languages
  - Included Turkish
- Is **robust** and **fast** and has **visualization libraries**

Install and import spaCy

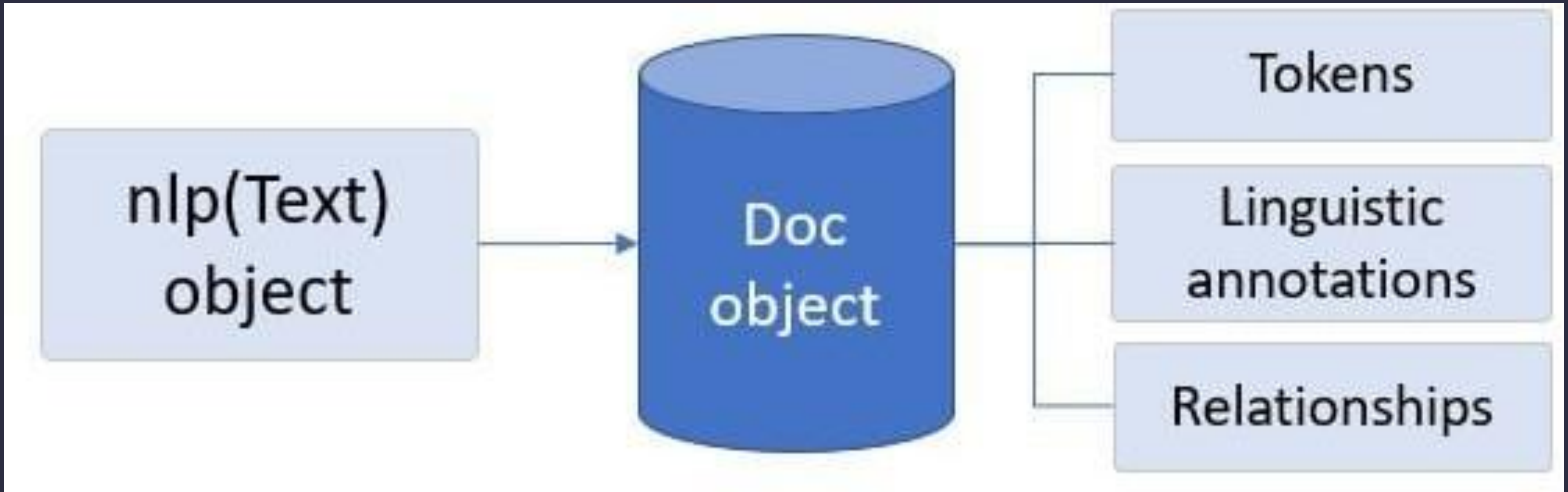
- As the first step, `spaCy` can be installed using the Python package manager pip
- `spaCy` trained models can be downloaded
- Multiple trained models are available for English language at **spacy.io**

```
python -m pip install spacy
```

```
python3 -m spacy download en_core_web_sm
import spacy
nlp = spacy.load("en_core_web_sm")
```

Read and process text with spaCy

- Loaded `spaCy` model `en_core_web_sm` = `nlp` object
- `nlp` object converts text into a `Doc` object (container) to store processed text



spaCy in action

- **Processing** a string using `spaCy`

```python
import spacy
nlp = spacy.load("en_core_web_sm")
text = "A spaCy pipeline object is created."
doc = nlp(text)
```

- **Tokenization**
  - A `Token` is defined as the smallest meaningful part of the text.
  - **Tokenization**: The process of dividing a text into a list of meaningful tokens

```python
print([token.text for token in doc])
```

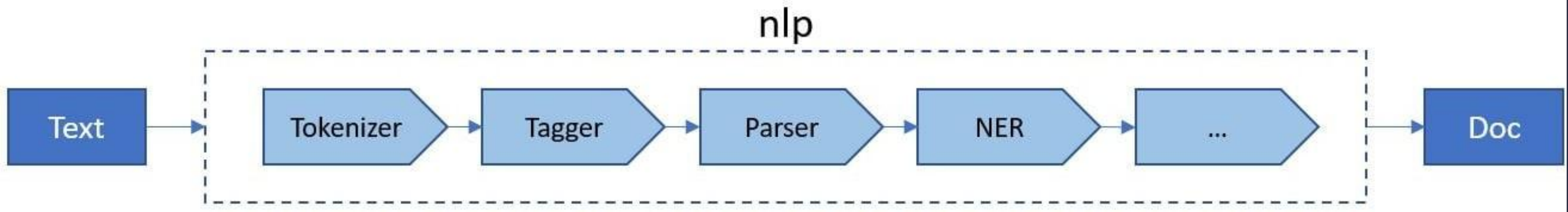
```
['A', 'spaCy', 'pipeline', 'object', 'is', 'created', '.']
```

```python
import spacy
nlp = spacy.load("en_core_web_sm")
doc = nlp("Here's my spaCy pipeline.")
```

- Import `spaCy`
- Use `spacy.load()` to return `nlp`, a `Language` class
  - The `Language` object is the text processing pipeline
- Apply `nlp()` on any text to get a `Doc` container

spaCy NLP pipeline

`spaCy` applies some processing steps using its `Language` class:



Container objects in spaCy

- There are multiple data structures to represent text data in `spaCy` :

Name	Description
`Doc`	A container for accessing linguistic annotations of text
`Span`	A slice from a `Doc` object
`Token`	An individual token, i.e. a word, punctuation, whitespace, etc.

Pipeline components

The `spaCy` language processing pipeline always depends on the loaded model and its capabilities.

Component	Name	Description
Tokenizer	Tokenizer	Segment text into tokens and create `Doc` object
Tagger	Tagger	Assign part-of-speech tags
Lemmatizer	Lemmatizer	Reduce the words to their root forms
EntityRecognizer	NER	Detect and label named entities

Pipeline components

- Each component has unique features to process text
  - **Language**
  - **DependencyParser**
  - **Sentencizer**

Tokenization

- Always the first operation
- All the other operations require tokens

Tokens can be words, numbers and punctuation

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

doc = nlp("Tokenization splits a sentence into its
tokens.")
print([token.text for token in doc])
```

```
['Tokenization', 'splits', 'a', 'sentence', 'into', 'its', 'tokens', '.']
```

Sentence segmentation

- More complex than tokenization
- Is a part of `DependencyParser` component

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

text = "We are learning NLP. This course introduces spaCy."
doc = nlp(text)
for sent in doc.sents:
    print(sent.text)
```

```
We are learning NLP.
This course introduces spaCy.
```

Lemmatization

- A lemma is a the base form of a token The lemma of eats and ate is eat Improves accuracy of language models

```
import spacy
nlp = spacy.load("en_core_web_sm")
doc = nlp("We are seeing her after one year.")
print([(token.text, token.lemma_) for token in doc])
```

```
[('We', 'we'), ('are', 'be'), ('seeing', 'see'), ('her', 'she'),
('after', 'after'), ('one', 'one'), ('year', 'year'), ('.',
'.')]
```

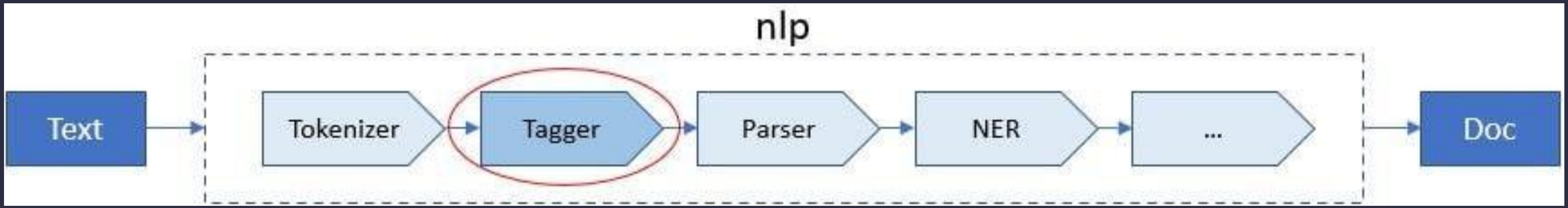
POS tagging

- **Categorizing** words grammatically, based on function and context within a sentence

POS	Description	Example
VERB	Verb	run, eat, ate, take
NOUN	Noun	man, airplane, tree, flower
ADJ	Adjective	big, old, incompatible, conflicting
ADV	Adverb	very, down, there, tomorrow
CONJ	Conjunction	and, or, but

POS tagging with spaCy

- POS tagging confirms the meaning of a word
- Some words such as **watch** can be both noun and verb
- `spaCy` captures POS tags in the `pos_` feature of the nlp pipeline
- `spacy.explain()` explains a given POS tag



POS tagging with spaCy

```
verb_sent = "I watch TV."

print([(token.text, token.pos_,
spacy.explain(token.pos_))
for token in nlp(verb_sent)])
```

```
[('I', 'PRON', 'pronoun'),
('watch', 'VERB', 'verb'),
('TV', 'NOUN', 'noun'),
('.', 'PUNCT', 'punctuation')]
```

```
noun_sent = "I left without my watch."

print([(token.text, token.pos_,
spacy.explain(token.pos_))
for token in nlp(noun_sent)])
```

```
[('I', 'PRON', 'pronoun'),
('left', 'VERB', 'verb'),
('without', 'ADP', 'adposition'),
('my', 'PRON', 'pronoun'),
('watch', 'NOUN', 'noun'),
('.', 'PUNCT', 'punctuation')]
```

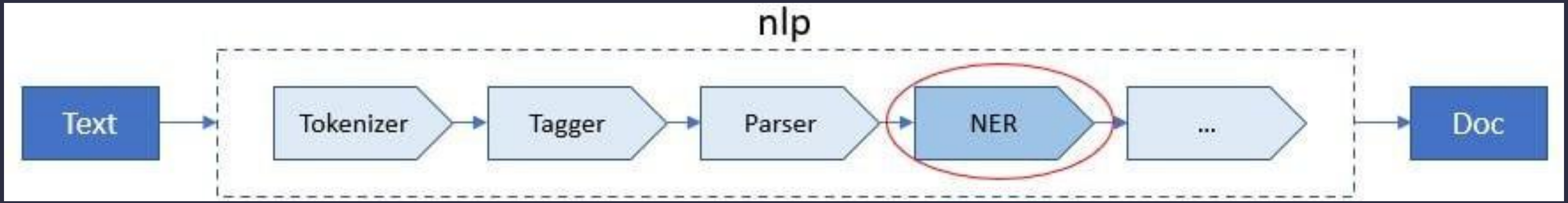
Named entity recognition

- A **named entity** is a word or phrase that refers to a specific entity with a name
- **Named-entity recognition** (NER) classifies named entities into pre-defined categories

Entity type	Description
PERSON	Named person or family
ORG	Companies, institutions, etc.
GPE	Geo-political entity, countries, cities, etc.
LOC	Non-GPE locations, mountain ranges, etc.
DATE	Absolute or relative dates or periods
TIME	Time smaller than a day

NER and spaCy

- `spaCy` models extract named entities using the `NER` pipeline component
- Named entities are available via the `doc.ents` property
- `spaCy` will also tag each entity with its entity label (`.label_`)



NER and spaCy

```python
import spacy
nlp = spacy.load("en_core_web_sm")
text = "Albert Einstein was genius."
doc = nlp(text)
print([(ent.text, ent.start_char,
ent.end_char, ent.label_) for ent in doc.ents])
```

```
>>> [('Albert Einstein', 0, 15, 'PERSON')]
```

NER and spaCy

- We can also access entity types of each token in a `Doc` container

```
import spacy
nlp = spacy.load("en_core_web_sm")
text = "Albert Einstein was genius."
doc = nlp(text)
print([(token.text, token.ent_type_) for token in doc])
```

```
>>> [('Albert', 'PERSON'), ('Einstein',
'PERSON'), ('was', ''), ('genius', ''), ('.', '')]
```

displaCy

- `spaCy` is equipped with a modern visualizer: `displaCy`
- The `displaCy` entity visualizer highlights named entities and their labels

```python
import spacy
from spacy import displacy

text = "Albert Einstein was genius."
nlp = spacy.load("en_core_web_sm")
doc = nlp(text)
displacy.serve(doc, style="ent")
```

POS tagging

- POS tags depend on the **context**, surrounding words and their tags

```python
import spacy
nlp = spacy.load("en_core_web_sm")
text = "My cat will fish for a fish tomorrrow in a fishy way."
print([(token.text, token.pos_, spacy.explain(token.pos_))
        for token in nlp(text)])
```

```
>>> [('My', 'PRON', 'pronoun'), ('cat', 'NOUN', 'noun'), ('will', 'AUX', 'auxiliary'),
('fish', 'VERB', 'verb'), ('for', 'ADP', 'adposition'), ('a', 'DET', 'determiner'),
('fish', 'NOUN', 'noun'), ('tomorrow', 'NOUN', 'noun'), ('in', 'ADP', 'adposition'),
('a', 'DET', 'determiner'), ('fishy', 'ADJ', 'adjective'), ('way', 'NOUN', 'noun'),
('.', 'PUNCT', 'punctuation')]
```

What is the importance of POS?

- **Word-sense disambiguation** (WSD) is the problem of deciding in which **sense** a word is used in a sentence.
- Determining the sense of the word can be crucial in machine translation, etc.

Word	POS tag	Description
Play	VERB	engage in activity for enjoyment and recreation
Play	NOUN	a dramatic work for the stage or to be broadcast

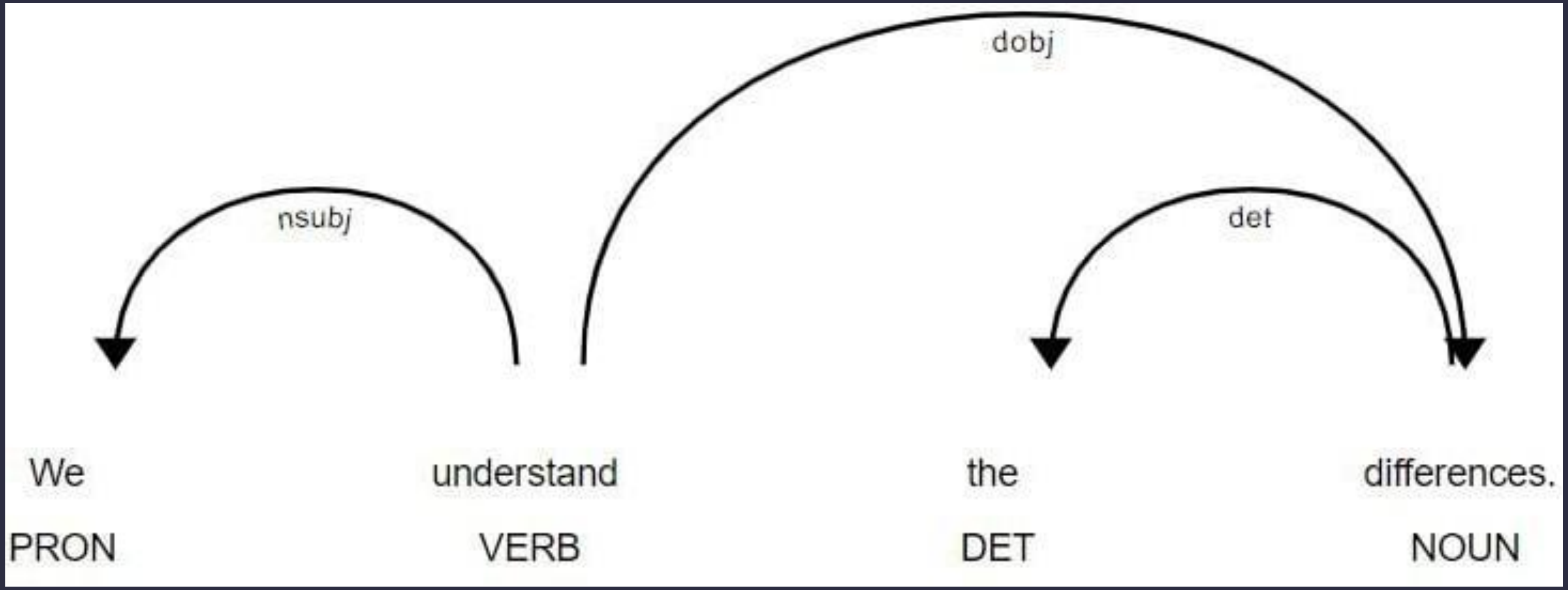
Word-sense disambiguation

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


verb_text = "I will fish tomorrow."
noun_text = "I ate fish."

print([(token.text, token.pos_) for token in nlp(verb_text) if "fish" in token.text], "\n")
print([(token.text, token.pos_) for token in nlp(noun_text) if "fish" in token.text])
```

```
[('fish', 'VERB', 'verb')]
[('fish', 'NOUN', 'noun')]
```

Dependency parsing

Explores a sentence syntax Links between two tokens Results in a tree



Dependency parsing and spaCy

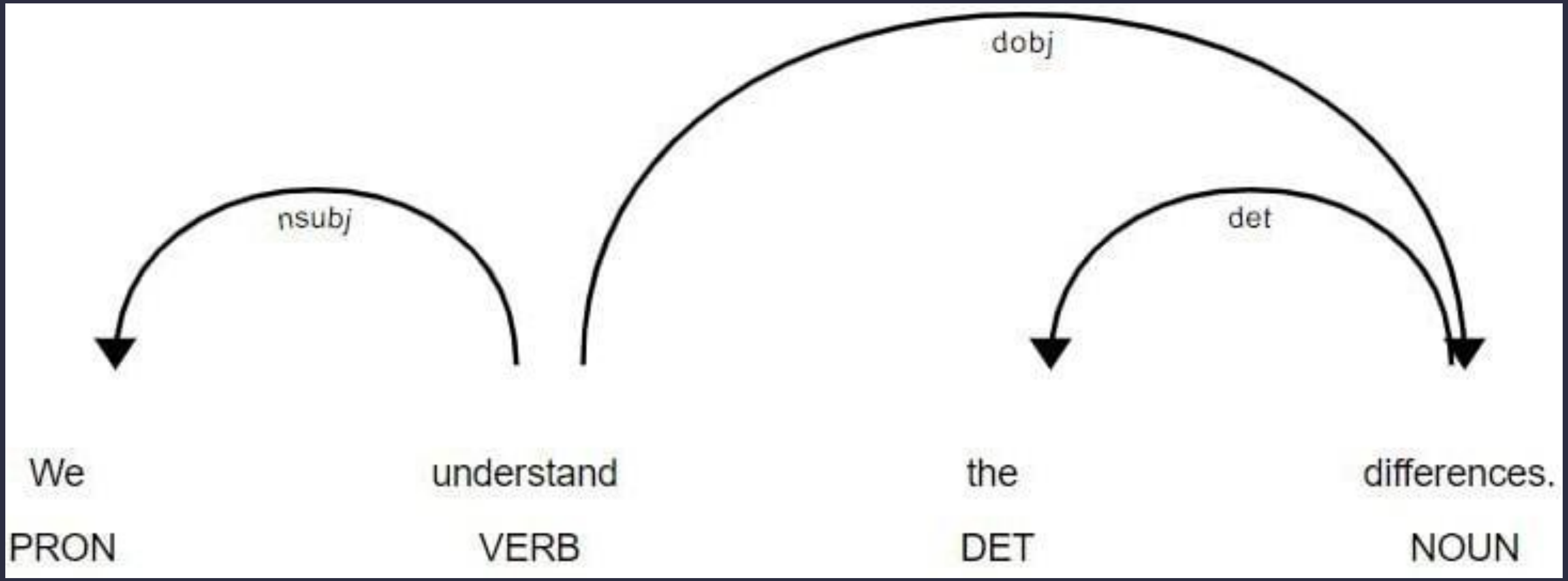
- **Dependency label** describes the type of syntactic relation between two tokens

Dependency label	Description
nsubj	Nominal subject
root	Root
det	Determiner
dobj	Direct object
aux	Auxiliary

Dependency parsing and displaCy

- `displaCy` can draw dependency trees

```
doc = nlp("We understand the differences.")

spacy.displacy.serve(doc, style="dep")
```



Dependency parsing and spaCy

- `.dep_` attribute to access the dependency label of a token

```
doc = nlp("We understand the differences.")
print([(token.text, token.dep_, spacy.explain(token.dep_))  for token in doc])
```

```
[('We', 'nsubj', 'nominal subject'), ('understand', 'ROOT', 'root'),
('the', 'det', 'determiner'), ('differences', 'dobj', 'direct object'),
('.', 'punct', 'punctuation')]
```

Word vectors (embeddings)

- Numerical representations of words
- Bag of words method: `{"I": 1, "got": 2, ...}`

- Older methods do not allow to understand the **meaning**:

Sentences	I	got	covid	coronavirus
Igot covid	1	2	3	
Igot coronavirus	1	2		4

Word vectors

- A **pre-defined number of dimensions**
- Considers **word frequencies** and the **presence of other words** in **similar contexts**

	living being	feline	human	gender	royalty	verb	plural
cat →	0.6	0.9	0.1	0.4	−0.7	−0.3	−0.2
kitten →	0.5	0.8	−0.1	0.2	−0.6	−0.5	−0.1
dog →	0.7	−0.1	0.4	0.3	−0.4	−0.1	−0.3
houses →	−0.8	−0.4	−0.5	0.1	−0.9	0.3	0.8

Word vectors

- **Multiple approaches** to produce word vectors:
  - word2vec, Glove, fastText and transformer-based architectures

- An example of a word vector:

```
array([ 2.2407    ,  1.0389    ,  1.3092    , -1.7335    , -0.78466   ,
       -0.29269   , -1.8059    , -2.5223    ,  0.78025   ,  2.4899    ,
       -0.091849  ,  0.28755   , -1.5057    ,  2.6337    ,  2.5252    ,
       -0.22432   , -2.2068    , -0.57895   , -0.56551   , -1.9338    ,
        1.4973    ,  0.85889   ,  3.3559    , -3.7527    ,  0.22585   ,
       -0.16969   ,  0.51389   ,  0.46073   , -0.28248   , -2.6048    ,
       -3.5896    , -1.0971    , -1.5517    , -0.12185   ,  2.8633    ,
       -1.2525    , -1.6924    , -2.2917    ,  0.97793   ,  0.46954   ,
       -3.595     , -0.17357   ,  0.9805    , -1.8044    , -0.72183   ,
       -0.40709   , -3.0943    ,  0.13095   , -2.9015    ,  1.4768    ,
       -1.0588    , -2.8123    ,  1.2936    , -0.0075977,  2.9975    ,
       -2.4438    ,  0.12348   ,  1.8322    ,  0.35869   , -0.018335  ,
        1.9534    ,  1.4417    ,  0.99895   , -2.8209    , -0.75846   ,
       -1.8438    , -3.2658    , -0.46574   ,  0.90322   ,  0.79868   ,
       -1.6134    , -0.33082   ,  1.1541    , -4.7334    ,  1.4964    ,
       -2.4014    , -1.3461    , -0.95551   ,  0.29671   , -1.4506    ,
       -0.87128   , -3.0714    ,  1.3597    , -0.038133 ,  1.6414    ,
       -0.90879   ,  2.7406    ,  2.2951    , -3.1423    , -3.7525    ,
        0.74033   ,  1.4921    ,  0.47422   , -1.8337    , -1.8168    ,
        0.66901   , -1.3612    , -2.2729    , -1.7656    , -0.73968   ],
      dtype=float32)
```

spaCy vocabulary

- A part of many spaCy models.
- `en_core_web_md` has **300**-dimensional vectors for **20,000** words.

python -m spacy download en_core_web_md

```
import spacy
nlp = spacy.load("en_core_web_md")
print(nlp.meta["vectors"])
```

```
>>> {'width': 300, 'vectors': 20000, 'keys': 514157,
'name': 'en_vectors', 'mode': 'default'}
```

Word vectors in spaCy

- `nlp.vocab` : to access vocabulary (`Vocab` class)
- `nlp.vocab.strings` : to access word IDs in a vocabulary

```
import spacy
nlp = spacy.load("en_core_web_md")
like_id = nlp.vocab.strings["like"]
print(like_id)
```

```
>>> 18194338103975822726
```

- `.vocab.vectors` : to access words vectors of a model or a word, given its corresponding ID

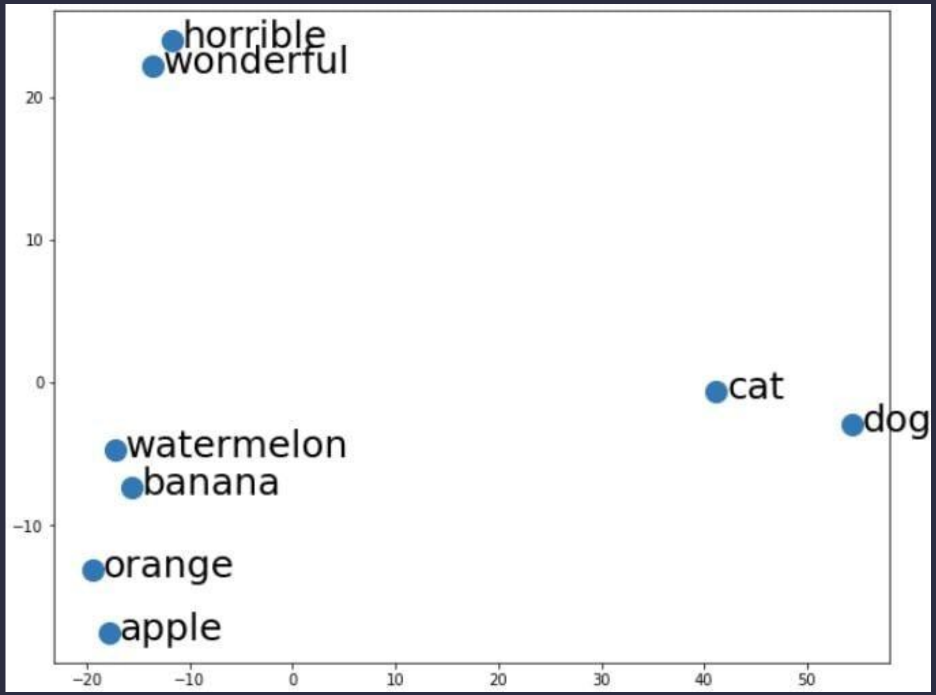
```
print(nlp.vocab.vectors[like_id])
```

```
>>> array([-2.3334e+00, -1.3695e+00, -1.1330e+00, -6.8461e-01, ...])
```

Word vectors visualization

- Word vectors allow to understand how words are grouped
- Principal Component Analysis projects word vectors into a two-dimensional space

```
array([ 2.2407    ,  1.0389    ,  1.3092    , -1.7335    , -0.78466   ,
       -0.29269   , -1.8059    , -2.5223    ,  0.78025   ,  2.4899    ,
       -0.091849  ,  0.28755   , -1.5057    ,  2.6337    ,  2.5252    ,
       -0.22432   , -2.2068    , -0.57895   , -0.56551   , -1.9338    ,
        1.4973    ,  0.85889   ,  3.3559    , -3.7527    ,  0.22585   ,
       -0.16969   ,  0.51389   ,  0.46073   , -0.28248   , -2.6048    ,
       -3.5896    , -1.0971    , -1.5517    , -0.12185   ,  2.8633    ,
       -1.2525    , -1.6924    , -2.2917    ,  0.97793   ,  0.46954   ,
       -3.595     , -0.17357   ,  0.9805    , -1.8044    , -0.72183   ,
       -0.40709   , -3.0943    ,  0.13095   , -2.9015    ,  1.4768    ,
       -1.0588    , -2.8123    ,  1.2936    , -0.0075977,  2.9975    ,
       -2.4438    ,  0.12348   ,  1.8322    ,  0.35869   , -0.018335  ,
        1.9534    ,  1.4417    ,  0.99895   , -2.8209    , -0.75846   ,
       -1.8438    , -3.2658    , -0.46574   ,  0.90322   ,  0.79868   ,
       -1.6134    , -0.33082   ,  1.1541    , -4.7334    ,  1.4964    ,
       -2.4014    , -1.3461    , -0.95551   ,  0.29671   , -1.4506    ,
       -0.87128   , -3.0714    ,  1.3597    , -0.038133  ,  1.6414    ,
       -0.90879   ,  2.7406    ,  2.2951    , -3.1423    , -3.7525    ,
        0.74033   ,  1.4921    ,  0.47422   , -1.8337    , -1.8168    ,
        0.66901   , -1.3612    , -2.2729    , -1.7656    , -0.73968   ],
      dtype=float32)
```



Word vectors visualization

- Import required libraries and a `spaCy` model.

```
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
import numpy as np
nlp = spacy.load("en_core_web_md")
```

- Extract word vectors for a given list of words and stack them vertically.

```
words = ["wonderful", "horrible",
           "apple", "banana", "orange", "watermelon",
           "dog", "cat"]
word_vectors = np.vstack([nlp.vocab.vectors[nlp.vocab.strings[w]] for w in words])
```

Word vectors visualizations

- Extract two principal components using PCA.

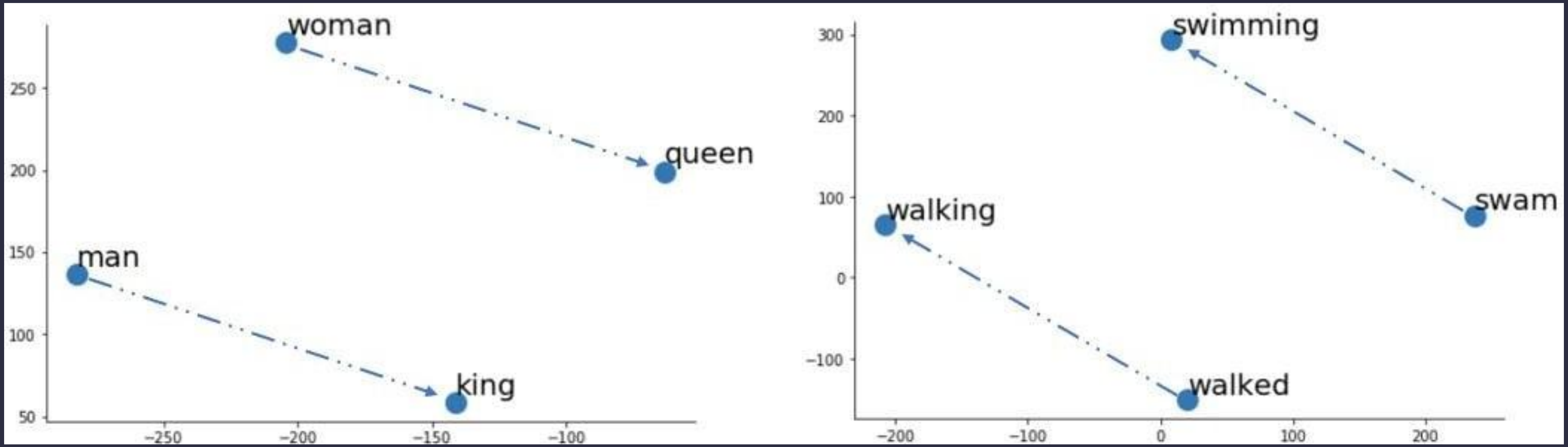
```
pca = PCA(n_components=2)
word_vectors_transformed = pca.fit_transform(word_vectors)
```

- Visualize the scatter plot of transformed vectors.

```
plt.figure(figsize=(10, 8))
plt.scatter(word_vectors_transformed[:, 0], word_vectors_transformed[:, 1])
for word, coord in zip(words, word_vectors_transformed):
    x, y = coord
    plt.text(x, y, word, size=10)
plt.show()
```

Analogies and vector operations

- A semantic relationship between a pair of words.
- **Word embeddings** generate analogies such as gender and tense:
  - queen - woman + man = king



Similar words in a vocabulary

- `spaCy` find **semantically similar terms** to a given term

```
import numpy as np
import spacy
nlp = spacy.load("en_core_web_md")

word = "covid"
most_similar_words = nlp.vocab.vectors.most_similar(
    np.asarray([nlp.vocab.vectors[nlp.vocab.strings[word]]]), n=5)

words = [nlp.vocab.strings[w] for w in most_similar_words[0][0]]
print(words)
```

```
>>> ['Covi', 'CoVid', 'Covici', 'COVID-19', 'corona']
```

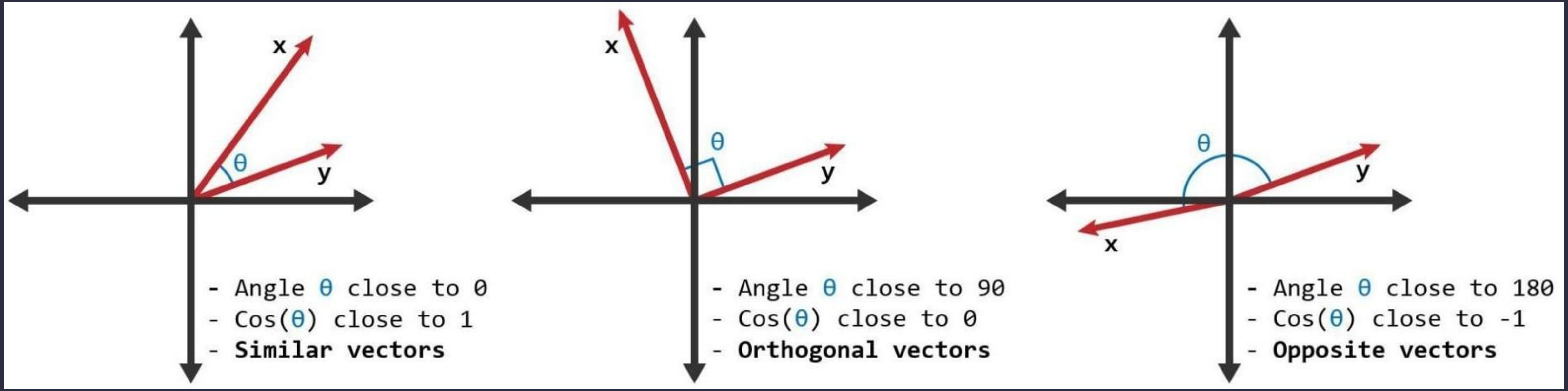
The semantic similarity method

- Process of **analyzing texts** to **identify similarities**
- Categorizes texts into **predefined categories** or detect **relevant texts**
- **Similarity score** measures how similar two pieces of text are

```
What is the cheapest flight from Boston to Seattle?
Which airline serves Denver, Pittsburgh and Atlanta?
What kinds of planes are used by American Airlines?
```

Similarity score

- A **metric** defined over texts
- To measure similarity use **Cosine similarity** and **word vectors**
- **Cosine similarity** is any number between 0 and 1



Token similarity

- spaCy calculates similarity scores between Token objects

```
nlp = spacy.load("en_core_web_md")
doc1 = nlp("We eat pizza")
doc2 = nlp("We like to eat pasta")
token1 = doc1[2]
token2 = doc2[4]
print(f"Similarity between {token1} and {token2} = ", round(token1.similarity(token2), 3))
```

```
>>> Similarity between pizza and pasta =   0.685
```

Span similarity

- `spaCy` calculates semantic similarity of two given `Span` objects

```
doc1 = nlp("We eat pizza")
doc2 = nlp("We like to eat pasta")

span1 = doc1[1:]
span2 = doc2[1:]
print(f"Similarity between \"{span1}\" and \"{span2}\" = ",
        round(span1.similarity(span2), 3))
```

```
>>> Similarity between "eat pizza" and "like to eat pasta" =  0.588
```

```
print(f"Similarity between \"{doc1[1:]}\" and \"{doc2[3:]}\" = ",
        round(doc1[1:].similarity(doc2[3:]), 3))
```

```
>>> Similarity between "eat pizza" and "eat pasta" =  0.936
```

Doc similarity

- `spaCy` calculates the similarity scores between two documents

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

doc1 = nlp("I like to play basketball")
doc2 = nlp("I love to play basketball")
print("Similarity score :", round(doc1.similarity(doc2), 3))
```

```
>>> Similarity score : 0.975
```

- High cosine similarity shows highly semantically similar contents
- `Doc` vectors default to an average of word vectors

Sentence similarity

- `spaCy` finds relevant content to a given keyword
- Finding similar customer questions to the word **price**:

```
sentences = nlp("What is the cheapest flight from Boston to Seattle?
                 Which airline serves Denver, Pittsburgh and Atlanta?
                 What kinds of planes are used by American Airlines?")


keyword = nlp("price")
for i, sentence in enumerate(sentences.sents):
    print(f"Similarity score with sentence {i+1}: ", round(sentence.similarity(keyword), 5))
```

```
>>> Similarity score with sentence 1:   0.26136
Similarity score with sentence 2:   0.14021
Similarity score with sentence 3:   0.13885
```

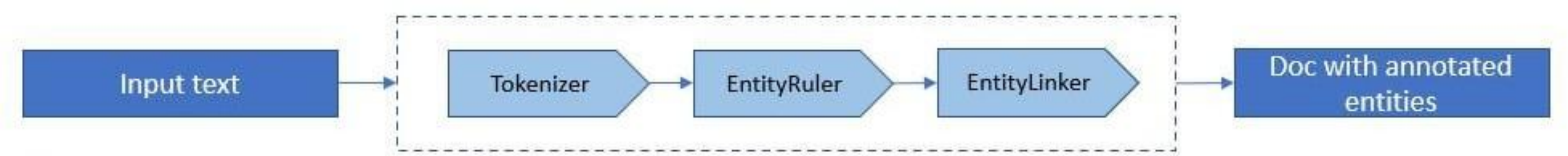
spaCy pipelines

- `spaCy` first tokenizes the text to produce a `Doc` object
- The `Doc` is processed in several different steps of **processing pipeline**

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


doc = nlp(example_text)
```

spaCy pipelines

- A **pipeline** is a **sequence of pipes**, or **actors on data**
- A `spaCy` **NER** pipeline:
  - Tokenization
  - Named entity identification
  - Named entity classification




```
print([ent.text for ent in doc.ents])
```

Adding pipes

- `sentencizer` : `spaCy` pipeline component for sentence segmentation.

```
text = " ".join(["This is a test sentence."]*10000)
en_core_sm_nlp = spacy.load("en_core_web_sm")
start_time = time.time()
doc = en_core_sm_nlp(text)
print(f"Finished processing with en_core_web_sm model in
        {round((time.time() - start_time)/60.0 , 5)} minutes")
```

```
>>> Finished processing with en_core_web_sm model in 0.09332 minutes
```

Adding pipes

- Create a blank model and add a `sentencizer` pipe:

```
blank_nlp = spacy.blank("en")
blank_nlp.add_pipe("sentencizer")
start_time = time.time()
doc = blank_nlp(text)
print(f"Finished processing with blank model in
       {round((time.time() - start_time)/60.0 , 5)} minutes")
```

```
>>> Finished processing with blank model in 0.00091 minutes
```

Analyzing pipeline components

- `nlp.analyze_pipes()` analyzes a `spaCy` pipeline to determine:
  - Attributes that pipeline components set
  - Scores a component produces during training
  - Presence of all required attributes

- Setting `pretty` to `True` will print a table instead of only returning the structured data.

```
import spacy


nlp = spacy.load("en_core_web_sm")
analysis = nlp.analyze_pipes(pretty=True)
```

Analyzing pipeline components

```
============================= Pipeline Overview =============================

#   Component         Assigns               Requires         Scores              Retokenizes
-   ---------------   -------------------   --------------   -----------------   -----------
0   tok2vec           doc.tensor                                                 False

1   tagger            token.tag                              tag_acc             False

2   parser            token.dep                              dep_uas             False
                      token.head                             dep_las
                      token.is_sent_start                    dep_las_per_type
                      doc.sents                              sents_p
                                                             sents_r
                                                             sents_f

3   attribute_ruler                                                              False

4   lemmatizer        token.lemma                            lemma_acc           False

5   ner               doc.ents                               ents_f              False
                      token.ent_iob                          ents_p
                      token.ent_type                         ents_r
                                                             ents_per_type

6   entity_linker     token.ent_kb_id       doc.ents         nel_micro_f         False
                                            doc.sents        nel_micro_r
                                            token.ent_iob    nel_micro_p
                                            token.ent_type

✓ No problems found.
```

spaCy EntityRuler

- `EntityRuler` adds named-entities to a `Doc` container
- It can be used on its own or combined with `EntityRecognizer`
- **Phrase entity patterns** for exact string matches (string):

```
{"label": "ORG", "pattern": "Microsoft"}
```

- **Token entity patterns** with one dictionary describing one token (list):

```
{"label": "GPE", "pattern": [{"LOWER": "san"}, {"LOWER": "francisco"}]}
```

Adding EntityRuler to spaCy pipeline

- Using `.add_pipe()` method
- List of patterns can be added using `.add_patterns()` method

```
nlp = spacy.blank("en")
entity_ruler = nlp.add_pipe("entity_ruler")
patterns = [{"label": "ORG", "pattern": "Microsoft"},
            {"label": "GPE", "pattern": [{"LOWER": "san"}, {"LOWER": "francisco"}]}]
entity_ruler.add_patterns(patterns)
```

Adding EntityRuler to spaCy pipeline

- `.ents` store the results of an `EntityLinker` component

```
doc = nlp("Microsoft is hiring software developer in San
Francisco.") print([(ent.text, ent.label_) for ent in doc.ents])
```

```
[('Microsoft', 'ORG'), ('San Francisco', 'GPE')]
```

EntityRuler in action

- Integrates with `spaCy` pipeline components
- Enhances the named-entity recognizer
- `spaCy` model without `EntityRuler` :

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

doc = nlp("Manhattan associates is a company in the U.S.")
print([(ent.text, ent.label_) for ent in doc.ents])
```

```
>>> [('Manhattan', 'GPE'), ('U.S.', 'GPE')]
```

EntityRuler in action

- `EntityRuler` added after existing `ner` component:

```
nlp = spacy.load("en_core_web_sm")
ruler = nlp.add_pipe("entity_ruler", after='ner')
patterns = [{"label": "ORG", "pattern": [{"lower": "manhattan"}, {"lower": "associates"}]}]
ruler.add_patterns(patterns)

doc = nlp("Manhattan associates is a company in the U.S.")
print([(ent.text, ent.label_) for ent in doc.ents])
```

```
>>> [('Manhattan', 'GPE'), ('U.S.', 'GPE')]
```

EntityRuler in action

- `EntityRuler` added before existing `ner` component:

```
nlp = spacy.load("en_core_web_sm")
ruler = nlp.add_pipe("entity_ruler", before='ner')
patterns = [{"label": "ORG", "pattern": [{"lower": "manhattan"}, {"lower": "associates"}]}]
ruler.add_patterns(patterns)

doc = nlp("Manhattan associates is a company in the U.S.")
print([(ent.text, ent.label_) for ent in doc.ents])
```

```
>>> [('Manhattan associates', 'ORG'), ('U.S.', 'GPE')]
```

What is RegEx?

- **Rule-based information extraction** (IR) is useful for many NLP tasks
- **Regular expression** (**RegEx**) is used with complex string matching patterns
- RegEx **finds** and **retrieves** patterns or replace matching patterns

RegEx strengths and weaknesses

Pros:

- Enables writing robust rules to retrieve information
- Can allow us to find many types of variance in strings
- Runs fast
- Supported by programming languages

Cons:

- Syntax is challenging for beginners
- Requires knowledge of all the ways a pattern may be mentioned in texts

RegEx in Python

- Python comes prepackaged with a RegEx library, `re`.
- The first step in using `re` package is to define a `pattern`.
- The resulting pattern is used to find matching content.

```
import re

pattern = r"((\d){3}-(\d){3}-(\d){4})"
text = "Our phone number is 832-123-5555 and their phone number is 425-123-4567."
```

RegEx in Python

- We use `.finditer()` method from `re` package

```
iter_matches = re.finditer(pattern, text)
for match in iter_matches:
    start_char = match.start()
    end_char = match.end()
    print ("Start character: ", start_char, "| End character: ", end_char,
           "| Matching text: ", text[start_char:end_char])
```

```
>>> Start character:  20 | End character:  32 | Matching text:  832-123-5555
Start character:  59 | End character:  71 | Matching text:  425-123-4567
```

RegEx in spaCy

- **RegEx** in three pipeline components: `Matcher`, `PhraseMatcher` and `EntityRuler`.

```
text = "Our phone number is 832-123-5555 and their phone number is 425-123-4567."
nlp = spacy.blank("en")
patterns = [{"label": "PHONE_NUMBER", "pattern": [{"SHAPE": "ddd"},
            {"ORTH": "-"}, {"SHAPE": "ddd"},
            {"ORTH": "-"}, {"SHAPE": "dddd"}]}]
ruler = nlp.add_pipe("entity_ruler")
ruler.add_patterns(patterns)
doc = nlp(text)
print ([(ent.text, ent.label_) for ent in doc.ents])
```

```
>>> [('832-123-5555', 'PHONE_NUMBER'), ('425-123-4567', 'PHONE_NUMBER')]
```

Matcher in spaCy

- **RegEx** patterns can be complex, difficult to read and debug.
- `spaCy` provides a readable and production-level alternative, the `Matcher` class.

```python
import spacy
from spacy.matcher import Matcher
nlp = spacy.load("en_core_web_sm")
doc = nlp("Good morning, this is our first day on campus.")
matcher = Matcher(nlp.vocab)
```

Matcher in spaCy

- Matching output include **start** and **end** token indices of the matched pattern.

```
pattern = [{"LOWER": "good"}, {"LOWER": "morning"}]
matcher.add("morning_greeting", [pattern])
matches = matcher(doc)
for match_id, start, end in matches:
    print("Start token: ", start, " | End token: ", end,
          "| Matched text: ", doc[start:end].text)
```

```
>>> Start token:   0   | End token:   2 | Matched text:   Good morning
```

Matcher extended syntax support

- Allows operators in defining the matching patterns.
- Similar operators to Python's `in`, `not in` and comparison operators

Attribute	Value type	Description
`IN`	any type	Attribute value is a member of a list
`NOT_IN`	any type	Attribute value is ***not*** a member of a list
`==`, `>=`, `<=`, `>`, `<`	int, float	Comparison operators for equality or inequality checks

Matcher extended syntax support

- Using `IN` operator to match both `good morning` and `good evening`

```
doc = nlp("Good morning and good evening.")
matcher = Matcher(nlp.vocab)
pattern = [{"LOWER": "good"}, {"LOWER": {"IN": ["morning", "evening"]}}]
matcher.add("morning_greeting", [pattern])
matches = matcher(doc)
```

- The output of matching using `IN` operator

```
for match_id, start, end in matches:
    print("Start token: ", start, " | End token: ", end,
          "| Matched text: ", doc[start:end].text)
```

```
>>> Start token:  0  | End token:  2 | Matched text:  Good morning
Start token:  3  | End token:  5 | Matched text:  good evening
```

PhraseMatcher in spaCy

- `PhraseMatcher` class matches a long list of phrases in a given text.

```
from spacy.matcher import PhraseMatcher
nlp = spacy.load("en_core_web_sm")
matcher = PhraseMatcher(nlp.vocab)
terms = ["Bill Gates", "John Smith"]
```

PhraseMatcher in spaCy

- PhraseMatcher outputs include **start** and **end** token indices of the matched pattern

```
patterns = [nlp.make_doc(term)  for term in terms]
matcher.add("PeopleOfInterest", patterns)
doc = nlp("Bill Gates met John Smith for an important discussion regarding
           importance of AI.")
matches = matcher(doc)
for match_id, start, end in matches:
    print("Start token: ", start, " | End token: ", end,
          "| Matched text: ", doc[start:end].text)
```

```
>>> Start token:  0  | End token:  2 | Matched text:  Bill Gates
Start token:  3  | End token:  5 | Matched text:  John Smith
```

PhraseMatcher in spaCy

- We can use `attr` argument of the `PhraseMatcher` class

```
matcher = PhraseMatcher(nlp.vocab, attr = "LOWER")
terms = ["Government", "Investment"]
patterns = [nlp.make_doc(term) for term in terms]
matcher.add("InvestmentTerms", patterns)
doc = nlp("It was interesting to the investment division of the government.")
```

```
matcher = PhraseMatcher(nlp.vocab, attr = "SHAPE")
terms = ["110.0.0.0", "101.243.0.0"]
patterns = [nlp.make_doc(term) for term in terms]
matcher.add("IPAddresses", patterns)
doc = nlp("The tracked IP address was 234.135.0.0.")
```

Why train spaCy models?

- Go a long way for general NLP use cases
- But may **not** have seen **specific domains** data during their training, e.g.
  - **Twitter** data
  - **Medical** data

PAST MEDICAL HISTORY: Significant for history of pulmonary fibrosis DISEASE and atrial fibrillation DISEASE .He is status post bilateral lung transplant back in 2004 because of the pulmonary fibrosis DISEASE .

ALLERGIES: There are no known allergies.

MEDICATIONS: Include multiple medications that are significant for his lung transplant including Prograf, CellCept CHEMICAL , prednisone CHEMICAL , omeprazole CHEMICAL , Bactrim CHEMICAL which he is on chronically, folic acid CHEMICAL , vitamin D CHEMICAL , Mag-Ox, Toprol-XL, calcium CHEMICAL 500 mg DOSAGE , vitamin B1, Centrum Silver, verapamil CHEMICAL , and digoxin CHEMICAL .

Why train spaCy models?

- Better results on your **specific domain**
- Essential for **domain specific text classification**

Before start training, ask the following questions:

- Do `spaCy` models perform well enough on our data?
- Does our domain include many labels that are absent in `spaCy` models?

Models performance on our data

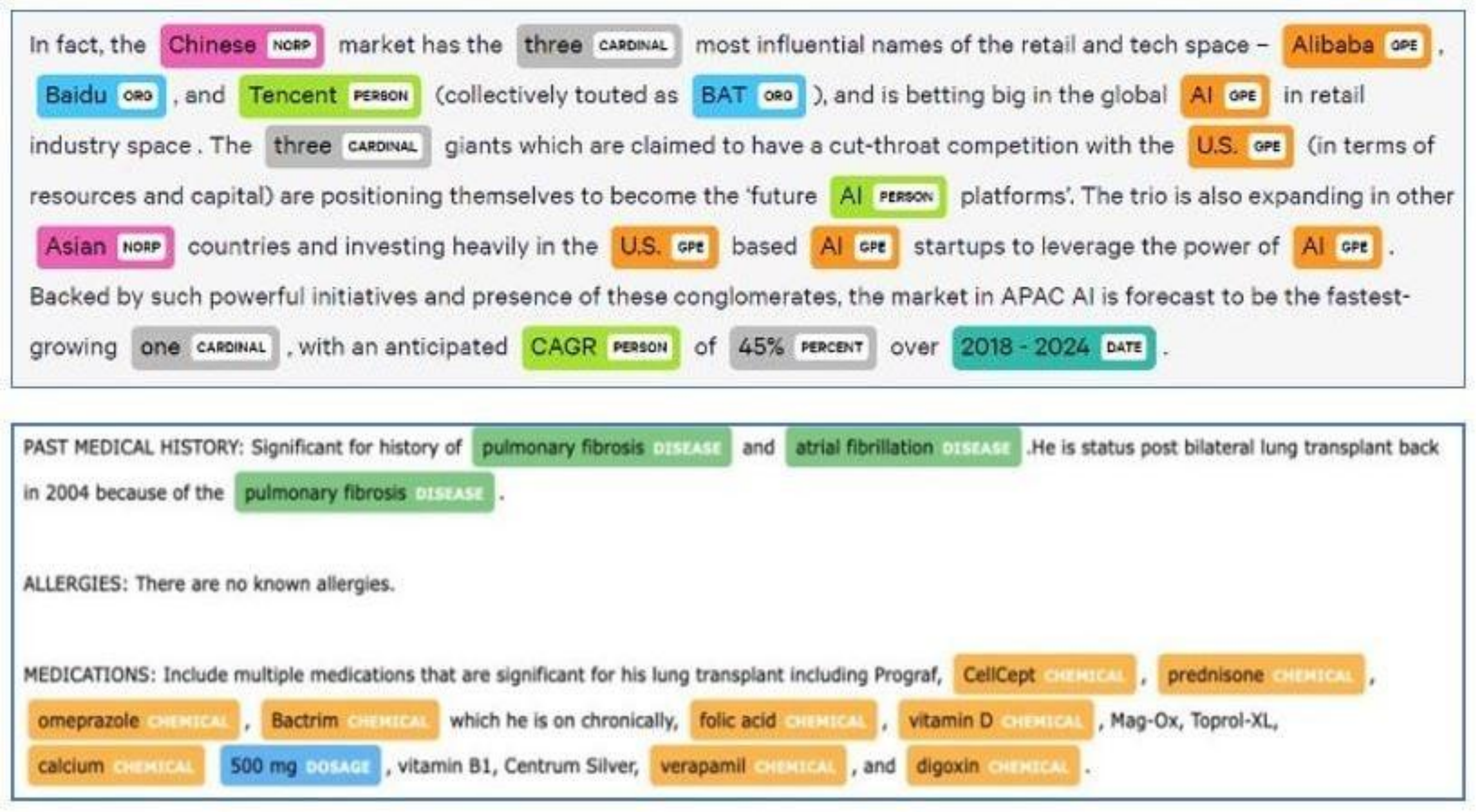
- Do `spaCy` models perform well enough on our data?
- `Oxford Street` is not correctly classified with a `GPE` label:

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

text = "The car was navigating to the Oxford
Street." doc = nlp(text)
print([(ent.text, ent.label_) for ent in doc.ents])
```

```
[('the Oxford Street', 'ORG')]
```

Output labels in spaCy models

- Does our domain include many labels that are absent in `spaCy` models?



Output labels in spaCy models

If we need custom model training, we follow these steps:

- Collect our domain specific data
- Annotate our data
- Determine to update an existing model or train a model from scratch

Training steps

1. Annotate and prepare input data
2. Initialize the model weight
3. Predict a few examples with the current weights
4. Compare prediction with correct answers
5. Use optimizer to calculate weights that improve model performance
6. Update weights slightly
7. Go back to step 3.

Annotating and preparing data

- First step is to prepare training data in required format
- After collecting data, we **annotate** it
- **Annotation** means labeling the intent, entities, etc.
- This is an example of annotated data:

```
annotated_data = {
"sentence": "An antiviral drugs used against influenza is neuraminidase inhibitors.",
"entities": {
              "label": "Medicine",
              "value": "neuraminidase inhibitors",
    }
}
```

Annotating and preparing data

- Here's another example of annotated data:

```
annotated_data = {
"sentence": "Bill Gates visited the SFO Airport.",
"entities": [{"label": "PERSON", "value": "Bill Gates"},
              {"label": "LOC", "value": "SFO Airport"}]
}
```

spaCy training data format

- Data annotation prepares training data for what we want the model to learn Training dataset has to be stored as a dictionary:

```
training_data = [
("I will visit you in Austin.", {"entities": [(20, 26, "GPE")]}),
("I'm going to Sam's house.", {"entities": [(13,18, "PERSON"), (19, 24, "GPE")]}),
("I will go.", {"entities": []})
]
```

Three example pairs:

- Each example pair includes a sentence as the first element
- Pair's second element is list of annotated entities and start and end characters

Example object data for training

- We cannot feed the raw text directly to spaCy
- We need to create an `Example` object for each training example

```
import spacy
from spacy.training import Example

nlp = spacy.load("en_core_web_sm")

doc = nlp("I will visit you in Austin.")
annotations = {"entities": [(20, 26, "GPE")]}

example_sentence = Example.from_dict(doc, annotations)
print(example_sentence.to_dict())
```

Training steps

1. Annotate and prepare input data
2. Disable other pipeline components
3. Train a model for a few epochs
4. Evaluate model performance

Disabling other pipeline components

- **Disable** all pipeline components except **NER**:

```
other_pipes = [pipe for pipe in nlp.pipe_names if pipe != 'ner']

nlp.disable_pipes(*other_pipes)
```

Model training procedure

- Go over the training set several times; one iteration is called an `epoch`.
- In each epoch, update the weights of the model with a small number.
- **Optimizers** update the model weights.

```
optimizer = nlp.create_optimizer()
```

```
losses = {}
for i in range(epochs):
  random.shuffle(training_data)
  for text, annotation in training_data:
    doc = nlp.make_doc(text)
    example = Example.from_dict(doc, annotation)
    nlp.update([example], sgd = optimizer, losses=losses)
```

Save and load a trained model

- Save a trained NER model:

```
ner = nlp.get_pipe("ner")
ner.to_disk("<ner model name>")
```

- Load the saved model:

```
ner = nlp.create_pipe("ner")
ner.from_disk("<ner model name>")
nlp.add_pipe(ner, "<ner model name>")
```

Model for inference

- Use a saved model at inference.

- Apply NER model and store tuples of (entity **text**, entity **label**):

```
doc = nlp(text)
entities = [(ent.text, ent.label_) for ent in doc.ents]
```