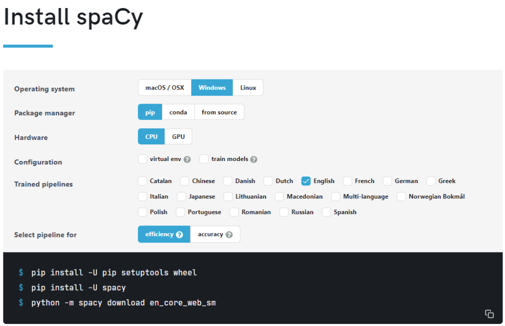
Natural Language Toolkit (Spacy)

Natural Language Processing
Lecture 3

THE GEORGE WASHINGTON UNIVERSITY
WASHINGTON, DC

Finding Words, Phrases, Names and Concepts

- Before going further you should install Spacy.
- Check the spacy Webster for installation guide. https://spacy.io/usage



- The nlp object.

```
from spacy.lang.en import English
nlp = English()
```

  - At the center of spaCy is the object containing the processing pipeline. We usually call this variable ”nlp”.
  - For example, to create an English nlp object, you can import the English language class from spacy.lang.en and instantiate it. You can use the nlp object like a function to analyze text.
  - It contains all the different components in the pipeline.
  - It also includes language-specific rules used for tokenizing the text into words and punctuation.

- The Doc object.
  - The Doc lets you access information about the text in a structured way, and no information is lost.

```python
doc = nlp("Hello world!")
for token in doc:
    print(token.text)
```

- The Token object.
  - To get a token at a specific position, you can index into the doc.

```python
token = doc[1]
print(token.text)
```

- The Span object.
  - A Span object is a slice of the document consisting of one or more tokens.

```python
span = doc[1:3]
print(span.text)
```

More spaCy Objects

- The Lexical object.

```python
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])
```

  - 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.
  - These attributes are also called lexical attributes

Statistical Models Attributes

- What are statistical models?

- It enable spaCy to predict linguistic attributes in contex.
- Part-of-speech tags.
- Syntactic dependencies.
- Named entities.
- It can be trained.
- It can ne be fine tuned.

- Some of the most interesting things, you can analyze are context-specific.
- For example, whether a word is a verb or whether a span of text is a person name.
- Statistical models enable spaCy to make predictions in context.
- This usually includes part-of speech tags, syntactic dependencies and named entities.
- Models are trained on large datasets of labeled example texts.
- They can be updated with more examples to fine-tune their predictions.
- For example, to perform better on your specific data.

Model Packages

- spaCy provides a number of pre-trained model packages you can download using the spacy download command.

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

- 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.

Predicting Part-of-speech Tags

- Lets load and process a text.

```python
import spacy

nlp = spacy.load("en_core_web_sm")
doc = nlp("She ate the pizza")
for token in doc:
    print(token.text, token.pos_)
```

- In this example, we're using spaCy to predict part-of-speech tags, the word types in context.
- For each token in the doc, we can print the text and the .pos_ attribute, the predicted part-of-speech tag.
- In spaCy, attributes that return strings usually end with an **underscore** – attributes without the underscore return an **integer ID value.**
- Here, the model correctly predicted ”ate” as a verb and ”pizza” as a noun.

Predicting Syntactic Dependencies

- Lets find more insight about the text.

```
import spacy

nlp = spacy.load("en_core_web_sm")
doc = nlp("She ate the pizza")
for token in doc:
    print(token.text, token.pos_, token.dep_, token.head.text)
```

- In addition to the part-of-speech tags, we can also predict how the words are related.
- For example, whether a word is the subject of the sentence or an object.
- The **.dep_ attribute** returns the predicted dependency label.
- The **.head** attribute returns the syntactic head token. You can also think of it as the parent token this word is attached to.

Dependency Label Scheme

- To describe syntactic dependencies, spaCy uses a standardized label scheme.
  - The pronoun ”She” is a nominal subject attached to the verb – in this case, to ”ate”.
  - The pronoun ”She” is a nominal subject attached to the verb – in this case, to ”ate”.
  - The pronoun ”She” is a nominal subject attached to the verb – in this case, to ”ate”.

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

- Lets find more insight about the text.

```
import spacy
nlp = spacy.load("en_core_web_sm")
doc = nlp("Apple is looking at buying U.K. startup for $1 billion")

for ent in doc.ents:
    print(ent.text, ent.label_)
```

- Named entities are ”real world objects” that are assigned a name – for example, a person, an organization or a country.
- The **doc.ents** property lets you access the named entities predicted by the model.
- It returns an iterator of Span objects, so we can print the entity text and the entity label using the **.label_ attribute**.
- In this case, the model is correctly predicting ”Apple” as an organization, ”U.K.” as a geopolitical entity and ”$1 billion” as money.

- Get quick definitions of the most common tags and labels.

```python
import spacy

print(spacy.explain("GPE"))
print(spacy.explain("NNP"))
print(spacy.explain("dobj"))
```

- To get definitions for the most common tags and labels, you can use the **spacy.explain** helper function.
- For example, ”GPE” for geopolitical entity isn’t exactly intuitive – but spacy.explain can tell you that it refers to countries, cities and states.
- The same works for part-of-speech tags and dependency labels.

Rule-based Matching

- Match on Doc objects, not just strings.
- Match on tokens and token attributes
- 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

- 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 are lists of dictionaries. Each dictionary describes one token.
- The keys are the names of token attributes, mapped to their expected values.

- In this example, we're looking for **two tokens** with the text "iPhone" and "X".
- Also we can match two tokens whose lowercase forms equal "iphone" and "x".
- We can even write patterns using attributes predicted by the model.
- For example, find the **word "duck" only if it's a verb, not a noun.**
- Here, we're matching a token with the lemma "buy", plus a noun. The lemma is the base form, so this pattern would match phrases like "buying milk" or "bought flowers".

Matcher 1

- To use a pattern, we first import the matcher from spacy.matcher.

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

nlp = spacy.load("en_core_web_sm")

matcher = Matcher(nlp.vocab)
pattern = [{"TEXT": "iPhone"}, {"TEXT": "X"}]
matcher.add("IPHONE_PATTERN", [pattern])

doc = nlp("Upcoming iPhone X release date leaked")

matches = matcher(doc)
```

- The matcher is initialized with the shared vocabulary, nlp.vocab.
- 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 the pattern.

Matcher 2

- When you call the matcher on a doc, it returns a list of tuples.

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

nlp = spacy.load("en_core_web_sm")
matcher = Matcher(nlp.vocab)
pattern = [{"TEXT": "iPhone"}, {"TEXT": "X"}]
matcher.add("IPHONE_PATTERN", [pattern])

doc = nlp("Upcoming iPhone X release date leaked")
matches = matcher(doc)

for match_id, start, end in matches:
    matched_span = doc[start:end]
    print(matched_span.text)
```

- Each tuple consists of three values: the match ID, the start index and the end index of the matched span.
- The matcher.add method lets you add a pattern.
- This means we can iterate over the matches and create a Span object: a slice of the doc at the start and end index.

Matcher 3 - Lexical Attributes Example

```
import spacy
from spacy.matcher import Matcher
nlp = spacy.load("en_core_web_sm")
matcher = Matcher(nlp.vocab)
pattern = [{"IS_DIGIT": True}, {"LOWER": "fifa"}, {"LOWER": "world"},
           {"LOWER": "cup"},  {"IS_PUNCT": True}]

matcher.add("FIFA", [pattern])
doc = nlp("2018 FIFA World Cup: France won!")
matches = matcher(doc)

for match_id, start, end in matches:
    matched_span = doc[start:end]
    print(matched_span.text)
```

- 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.

Matcher 4 - Other Token Attributes

```
import spacy
from spacy.matcher import Matcher

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

pattern = [
    {"LEMMA": "love", "POS": "VERB"},
    {"POS": "NOUN"}
]
matcher.add("Other", [pattern])

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

for match_id, start, end in matches:
    matched_span = doc[start:end]
    print(matched_span.text)
```

- In this example, we're looking for two tokens:
- A token consisting of only digits.
- A verb with the lemma ”love”, followed by a noun.

Matcher 5 - Operators and Quantifiers

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

pattern = [{"LEMMA": "buy"}, {"POS": "DET", "OP": "?"},  {"POS": "NOUN"}]
matcher.add("Other", [pattern])

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

for match_id, start, end in matches:
    matched_span = doc[start:end]
    print(matched_span.text)
```

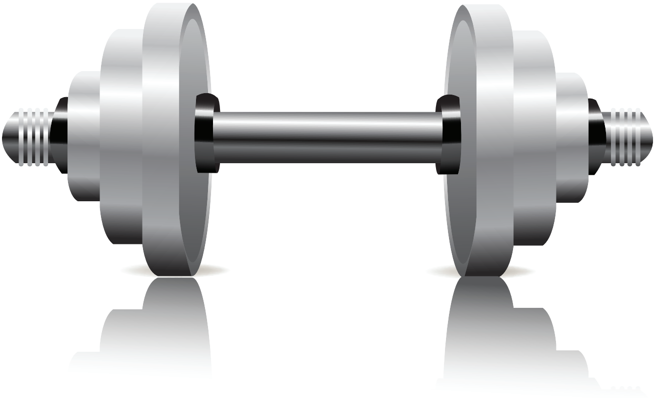
- Operators and quantifiers let you define how often a token should be matched.
- They can be added using the ”OP” key.
- Here, the ”?” operator makes the determiner token optional, so it will match a token with the lemma ”buy”, an optional article and a noun.

Using Operators And Quantifiers

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

- 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

- Exercise 1 to 7
- Class-Ex-Lecture3.py



Data Structures: Vocab, Lexemes and String Store

Shared Vocabulary and String Store

- spaCy stores all shared data in a vocabulary, the Vocab.
- To save memory, all strings are encoded to hash IDs. If a word occurs more than once, we don't need to save it every time.
- The string store is available as nlp.vocab.strings.It's a lookup table that works in both directions.

```python
import spacy

nlp = spacy.load("en_core_web_sm")

print(nlp.vocab.strings.add("text"))
coffee_hash = nlp.vocab.strings["text"]
coffee_string = nlp.vocab.strings[coffee_hash]
print(coffee_string)
```

Lexemes

- A Lexeme object is an entry in the vocabulary

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

doc = nlp("I love natural language processing.")
lexeme = nlp.vocab["natural"]
print(lexeme.text, lexeme.orth, lexeme.is_alpha)
```

- You can get a lexeme by looking up a string or a hash ID in the vocab.
- Lexemes expose attributes, just like tokens.
- They hold context-independent information about a word, like the text, or whether the word consists of alphabetic characters.
- Lexemes don't have part-of-speech tags, dependencies or entity labels.

- The Doc is one of the central data structures in spaCy.

```python
from spacy.lang.en import English
nlp = English()

from spacy.tokens import Doc

words = ["Hello", "world", "!"]
spaces = [True, False, False]

doc = Doc(nlp.vocab, words=words, spaces=spaces)
```

- It's created automatically when you process a text with the nlp object. But you can also instantiate the class manually.
- 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.

- A Span is a slice of a doc consisting of one or more tokens.

```
from spacy.tokens import Doc, Span
from spacy.lang.en import English
nlp = English()

words = ["Hello", "world", "!"]
spaces = [True, False, False]

doc = Doc(nlp.vocab, words=words, spaces=spaces)
span = Span(doc, 0, 2)
span_with_label = Span(doc, 0, 2, label="GREETING")
doc.ents = [span_with_label]; print(doc.ents)
```

- 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.

Word Vectors And Semantic Similarity

- spaCy can compare two objects and predict similarity
- **Doc.similarity(), Span.similarity() and Token.similarity()** are the syntaxes.
- It is suggested to use either en_core_web_md (medium model) or en_core_web_lg (large model).
- The Doc, Token and Span objects have a .similarity method that takes another object and returns a floating point number between 0 and 1, indicating how similar they are.
- In order to use similarity, you need a larger spaCy model that has word vectors included.

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

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


doc1 = nlp("I like fast food")
doc2 = nlp("I like pizza")
print(doc1.similarity(doc2))

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

- 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.
- The same works for tokens.

Similarity examples 2

- You can also use the similarity methods to compare different types of objects.

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

doc = nlp("I like pizza")
token = nlp("soap")[0]

print(doc.similarity(token))
span = nlp("I like pizza and pasta")[2:5]
doc = nlp("McDonalds sells burgers")

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

- 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.

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
- Cosine similarity, but can be adjusted.
- Vectors can be added to spaCy's statistical models.

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

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

doc = nlp("I have a banana")
print(doc[3].vector)
```

- 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”.

Combining Statistical Models and Rules

- Combining statistical models with rule-based systems is one of the most powerful tricks you should have in your NLP toolbox.
- For instance, detecting product or person names usually benefits from a statistical model.
- Instead of providing a list of all person names ever, your application will be able to predict whether a span of tokens is a person name.
- Similarly, you can predict dependency labels to find subject/object relationships.
- To do this, you would use spaCy's entity recognizer, dependency parser or part-of-speech tagger.

Adding Statistical Predictions

- Here's an example of a matcher rule for ”golden car”.

```python
from spacy.matcher import Matcher
import spacy
nlp = spacy.load("en_core_web_sm")
matcher = Matcher(nlp.vocab)

matcher.add("CAR", [[{"LOWER": "golden"}, {"LOWER": "car"}]])
doc = nlp("I have a Golden Car")
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_)
```

- If we iterate over the matches returned by the matcher, we can get the match ID and the start and end index of the matched span.

- Span objects give us access to the original document and all other token attributes and linguistic features predicted by the model.

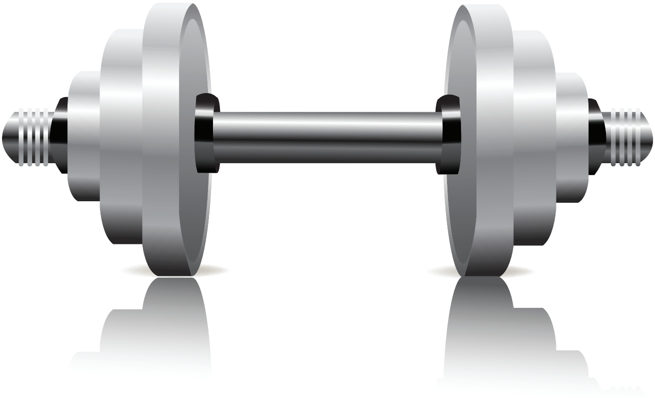
- The phrase matcher can be imported from spacy.matcher and follows the same API as the regular matcher.

```python
import spacy
from spacy.matcher import PhraseMatcher

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

pattern = nlp("Golden Car")
matcher.add("CAR", [pattern])
doc = nlp("I have a Golden Car")

for match_id, start, end in matcher(doc):
    span = doc[start:end]
    print("Matched span:", span.text)
```

- 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.

- Exercise 8 to 11
- Class-Ex-Lecture3.py



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

- The part-of-speech tagger sets the token.tag and token.pos attributes.
- token.dep and token.head attributes and is also responsible for detecting sentences and base noun phrases.
- The named entity recognizer adds the detected entities to the doc.ents property.
- Finally, the text classifier sets category labels that apply to the whole text.

- To see the names of the pipeline components present in the current nlp object, you can use the nlp.pipe_names attribute.

```
import spacy
nlp = spacy.load("en_core_web_sm")
print(nlp.pipe_names)
print(nlp.pipeline)
```

- For a list of component name and component function tuples, you can use the nlp.pipeline attribute.
- The component functions are the functions applied to the doc to process it and set attributes – for example, part-of-speech tags or named entities.

Custom Pipeline Components

- Custom pipeline components let you add your own function to the spaCy pipeline that is executed when you call the nlp object on a text
- 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.

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")

- Setting last to True will add the component last in the pipeline. This is the default behavior.
- Setting first to True will add the component first in the pipeline, right after the tokenizer.
- The before and after arguments let you define the name of an existing component to add the new component before or after. For example, before=”ner” will add it before the named entity recognizer.

Example - Simple Component

- Here's an example of a simple pipeline component.

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

@Language.component("component1")
def custom_component1(doc):
    print("Doc length:", len(doc))
    return doc
nlp.add_pipe('component1', name="component-info-1", first=True)
print("Pipeline:", nlp.pipe_names)

@Language.component("component2")
def custom_component2(doc):
    print("Doc length:", len(doc))
    return doc

nlp.add_pipe('component2', name="component-info-2", first=True)
doc = nlp("Hello world!")
print(doc)
```

Setting Custom Attributes

- 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.
- The first argument is the attribute name.
- Keyword arguments let you define how the value should be computed.

```python
from spacy.tokens import Doc, Token, Span
Doc.set_extension("title", default=None)
Token.set_extension("is_color", default=False)
Span.set_extension("has_color", default=False)
```

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

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

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

Token.set_extension("is_color", getter=get_is_color)

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

- The getter function is only called when you retrieve the attribute.
- This lets you compute the value dynamically, and even take other custom attributes into account.

Method Extensions

- Method extensions make the extension attribute a callable method.

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

def has_token(doc, token_text):
    in_doc = token_text in [token.text for token in doc]
    return in_doc

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")
```

- You can then pass one or more arguments to it, and compute attribute values dynamically
- In this example, the method function checks whether the doc contains a token with a given text. The first argument of the method is always the object itself – in this case, the doc.

Scaling and Performance

- Use nlp.pipe method

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

# docs = [nlp(text) for text in LOTS_OF_TEXTS]---Slow
# docs = list(nlp.pipe(LOTS_OF_TEXTS))---Fast

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

text = 'I love performance'
with nlp.disable_pipes("tagger", "parser"):
    doc = nlp(text)
    print(doc.text)
```

- Using only the tokenizer.
- Disabling pipeline components

Training and Updating Models

- Statistical models make predictions based on the examples they were trained on.
- You can usually make the model more accurate by showing it examples from your domain.
- You often also want to predict categories specific to your problem, so the model needs to learn about them.
- This is essential for text classification, very useful for entity recognition and a little less critical for tagging and parsing.

The Training Data

- The training data tells the model what we want it to predict.
- This could be texts and named entities we want to recognize, or tokens and their correct part-of-speech tags.
- To update an existing model, we can start with a few hundred to a few thousand examples.
- To train a new category we may need up to a million.
- spaCy's pre-trained English models for instance were trained on 2 million words labelled with part-of-speech tags, dependencies and named entities.
- Training data is usually created by humans who assign labels to texts.

- Exercise 12 to 14
- Class-Ex-Lecture3.py