



ECOM SCHOOL

המכללה למקצועות הדיגיטל וההייטק

NLP - Introduction

Natural Language Processing (NLP) → Natural Language Processing (NLP), is a subfield of artificial intelligence (AI) that focuses on the interaction between computers and humans through natural language. The goal of NLP is to enable computers to understand, interpret, and produce human language in a way that is both meaningful and useful.

Key Components of NLP:

- **Morphology** → The study of the structure of words. For example, understanding that "running" is derived from the root verb "run".
- **Syntax** → The arrangement of words and phrases to create well-formed sentences. This involves parsing sentences to understand their grammatical structure.
- **Semantics** → The meaning of words and how these meanings combine in sentences. This can involve word sense disambiguation and semantic role labeling.
- **Pragmatics** → The study of how context influences the interpretation of meaning. This includes understanding implied meanings and resolving ambiguities based on the conversational context.

NLP - Introduction

Applications of NLP:

- **Virtual Assistants** → Systems like Siri, Google Assistant, and Alexa leverage NLP to understand and respond to voice commands.
- **Customer Service** → Chatbots that provide automated responses to customer inquiries (Chat-GPT, Gemini, etc...).
- **Content Moderation** → Automated systems that detect and remove inappropriate content in social media and forums.
- **Healthcare** → NLP is used for processing and analyzing clinical notes, patient records, and literature to support medical decision-making.
- **Finance** → NLP algorithms analyze financial documents, news, and social media to gauge market sentiment and make predictions.

NLP - Introduction

Common NLP Tasks:

- **Tokenization** → Splitting text into individual words or phrases.
- **Part-of-Speech Tagging** → Identifying parts of speech (nouns, verbs, adjectives, etc.) for each token.
- **Named Entity Recognition (NER)** → Identifying and classifying entities in text such as names of people, organizations, locations, etc.
- **Sentiment Analysis** → Determining the emotional tone behind a body of text.
- **Text Summarization** → Creating a concise summary of a longer text.

Challenges in NLP:

- **Ambiguity** → Words and sentences can have multiple interpretations depending on context.
- **Complexity of Human Language** → Languages have nuances and evolving slang that are difficult to model.
- **Low-Resource Languages** → There is often a lack of annotated data for many of the world's languages.

SpaCy Library - Introduction

SpaCy library → SpaCy is a popular open-source library for Natural Language Processing (NLP) in Python. It is designed to be fast, efficient, and production-ready. SpaCy is known for its high performance, ease of use, and robust architecture, making it a preferred choice for many developers and data scientists working on NLP projects.

SpaCy offers robust tokenization that divides text into meaningful units while preserving context. The library excels in Part-of-Speech (POS) tagging, which labels tokens by their grammatical categories, and dependency parsing, which analyzes sentence structure.

SpaCy also includes lemmatization to convert words to their base forms, and supports customizable text classification for tasks like sentiment analysis. In addition, it provides pre-trained word vectors for semantic tasks, and offers rule-based matching for complex text processing.

SpaCy works seamlessly with TensorFlow, PyTorch, and Keras.

Installing & Importing SpaCy

Installing SpaCy library is a little more complex than installing other Python libraries we used in the course. The reason is that SpaCy also required to download the language model itself and its not coming as part of the installation package.

In order to successfully install and use the SpaCy library we will need to execute the following actions:

1. Create a new Python environment → This environment will be separated from our regular Python environment and will allow us to install any package we need without any conflicts or dependencies in previous libraries.
2. Activate the new Python environment and connect it to Jupiter Notebook
3. Download and install SpaCy
4. Download and install SpaCy english language model - '**en_core_web_sm**'



Installing & Importing SpaCy

Let's now perform the following actions in a new Jupyter Notebook:

Creating a new Python environment called 'spacy_env'

```
!conda create -n spacy_env python=3.11 -y
```

Activate the new environment and install dependencies

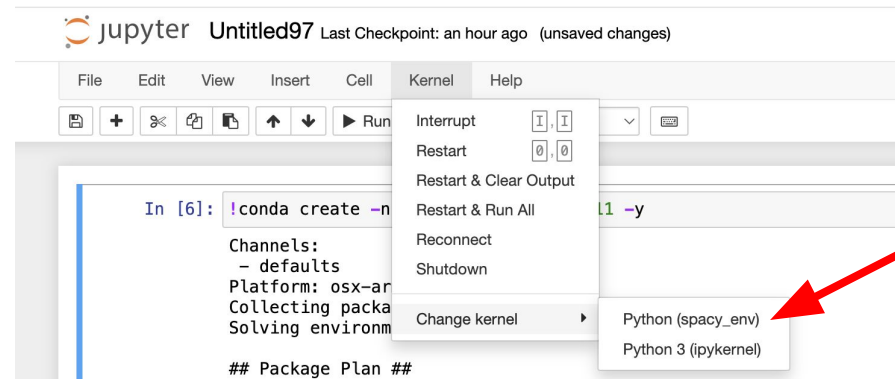
```
!conda install -n spacy_env ipykernel -y
```

```
!python -m ipykernel install --user --name spacy_env  
--display-name "Python (spacy_env)"
```



Installing & Importing SpaCy

Switch to the New Kernel in Jupyter → In the Jupyter Notebook menu go to Kernel → Change kernel → Python (spacy_env)



Choose the new Python environment we created

Now we have our new Python environment connected to Jupyter Notebook.

Next we can install SpaCy and download the english language model - '**en_core_web_sm**'

!pip install spacy

!python -m spacy download en_core_web_sm



ECOM SCHOOL

המכללה למקצועות הדיגיטל וההייטק

Installing & Importing SpaCy

We can verify that the download and installation succeeded by running:

```
In [11]: !python -m spacy validate
```

```
✓ Loaded compatibility table
```

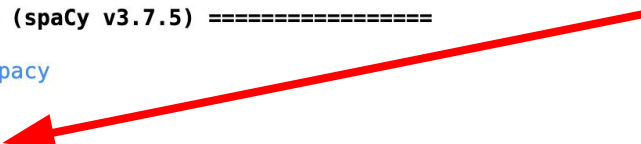
```
===== Installed pipeline packages (spaCy v3.7.5) =====
```

```
i spaCy installation:
```

```
/opt/anaconda3/lib/python3.11/site-packages/spacy
```

NAME	SPACY	VERSION
en_core_web_sm	>=3.7.2,<3.8.0	3.7.1 ✓

The 'en_core_web_sm' model we downloaded is been used by SpaCy



In addition, we can run the following Python code to see the NLP model in action:

```
In [12]: import spacy
```

```
# Load the en_core_web_sm model
```

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

```
text = "Apple is looking at buying U.K. startup for $1 billion"
```

```
doc = nlp(text)
```

```
# Print the named entities
```

```
for ent in doc.ents:
```

```
    print(ent.text, ent.label_)
```

```
Apple ORG
```

```
U.K. GPE
```

```
$1 billion MONEY
```

Jupyter Notebook - Remove Python Environment

In case we want to remove our new Python environment from Jupyter Notebook we can run the following commands:

First we need to get a list of all current Kernel environments that are running on our computer:

```
In [9]: !jupyter kernelspec list

0.00s - Debugger warning: It seems that frozen modules are being used, which may
0.00s - make the debugger miss breakpoints. Please pass -Xfrozen_modules=off
0.00s - to python to disable frozen modules.
0.00s - Note: Debugging will proceed. Set PYDEVD_DISABLE_FILE_VALIDATION=1 to disable this validation.
Available kernels:
  spacy_env    /Users/ben.meir/Library/Jupyter/kernels/spacy_env
  python3     /opt/anaconda3/share/jupyter/kernels/python3
```

Next we need to copy from the previous response the environment path and run the following command:

```
In [7]: !rm -rf /Users/ben.meir/Library/Jupyter/kernels/spacy_env
```



SpaCy - English Language Model

As we saw, we had to download the english language model - '**en_core_web_sm**' in order to use the SpaCy library. Let's now understand what is this model and why we need it.

'**en_core_web_sm**' is one of the pre-trained natural language models specifically designed for processing English text. These models are bundled with spaCy, and each is designed to be used for a variety of natural language processing (NLP) tasks.

Here's a breakdown of the model name:

en → Indicates that this is an English language model.

core → Suggests that this is the core model, designed for general-purpose use.

web → Indicates that the model is trained on web data.

sm → Denotes that this is the "small" version of the model, indicating that it is a lightweight model with a balanced trade-off between performance and accuracy.

SpaCy - English Language Model

Why we need pre-trained NLP model and not training our own?

To understand why we need to download and use models like ``en_core_web_sm``, consider the following points:

- **Complexity of NLP Tasks** → Natural language processing is inherently complex, involving a range of tasks. Pre-trained models encapsulate this complexity, making it simpler for developers to perform advanced NLP without deep expertise.
- **Training Costs** → Training an NLP model from scratch requires significant computational resources and high-quality training data. Pre-trained models like ``en_core_web_sm`` save you this cost and effort by providing an already trained model that you can use directly.
- **Quick Setup** → By using a pre-trained model, you can quickly set up your NLP pipeline. This is especially useful for prototyping and rapidly iterating on your projects.
- **Consistency** → Pre-trained models ensure consistency in performance across different systems and applications.

SpaCy - Basic Functionality

The main task of SpaCy model is to perform tokenization which mean splitting text into individual words or phrases. By loading the **'en_core_web_sm'** model we can pass a string sentence to it and SpaCy will perform the tokenization of this sentence.

For example →

```
In [15]: import spacy

nlp = spacy.load('en_core_web_sm')

doc = nlp(u'Tesla is looking at buying U.S. startup for $6 million')

print(type(doc))
print()
for token in doc:
    print(token.text, token.pos, token.pos_, token.dep_)

<class 'spacy.tokens.doc.Doc'>

Tesla 96 PROPN nsubj
is 87 AUX aux
looking 100 VERB ROOT
at 85 ADP prep
buying 100 VERB pcomp
U.S. 96 PROPN compound
startup 92 NOUN dobj
for 85 ADP prep
$ 99 SYM quantmod
6 93 NUM compound
million 93 NUM pobj
```

We load the NLP model we want to use

We provide a string in common english to the model

The model response is a SpaCy Doc object

The model split the words in the sentence into separated tokens

SpaCy - Basic Functionality

Once we have the Doc object we can see what are the different tokens that the model decided to split from the provided sentence. Each token has additional properties that can be useful in understanding the token meaning in the sentence.

- **text** → Returns the original text content of the token.
- **pos** → Returns the part-of-speech (POS) tag of the token as an integer. The POS tag describes the grammatical category of the word, such as whether it is a noun, verb, adjective, etc.
- **pos_** → Returns the string representation of the POS tag. This is usually more human-readable and easier to interpret (For example → `NOUN` for nouns, `VERB` for verbs, `ADJ` for adjectives, `ADV` for adverbs).
- **dep_** → Returns the dependency label of the token as a string. Dependency labels describe the grammatical relationships between words in a sentence. They provide information about the token's function in a sentence and its relation to other tokens (For example → `nsubj` for nominal subject, `dobj` for direct object, `ROOT` for the root verb in a sentence, etc...)

SpaCy - Basic Functionality

In our example, we can see that the model recognized 'Tesla' and 'U.S.' as a 'PROPN' meaning **"Proper Noun"** In linguistic terms, a proper noun is a specific type of noun that names a particular person, place, organization, or sometimes a thing.

We can also see that in the 'U.S.' token the model been able to identify that the '.' are part of the token and not split them out. In addition the model recognized '\$' as a symbol (SYM) and the word 'million' as a number (NUM).

Let's take a look at another tokenization example:

```
In [20]: doc2 = nlp("Tesla isn't looking into startups anymore.")
for token in doc2:
    print(token.text, token.pos, token.pos_, token.dep_)

Tesla 96 PROPN nsubj
is 87 AUX aux
n't 94 PART neg
looking 100 VERB ROOT
into 85 ADP prep
startups 92 NOUN pobj
anymore 86 ADV advmod
. 97 PUNCT punct
```

We can see that the model splitted the 'isn't' word into 2 separated tokens. First for the 'is' (AUX - auxiliary verbs) Second for 'n't' (neg - negativity word)

SpaCy - Basic Functionality

We can use basic indexing (like with regular Python list) to get a specific token value

For example → If we want to see the first token 'pos_' property we can execute:

```
In [21]: doc2[0].pos_
```

```
Out[21]: 'PROPN'
```

We can also see SpaCy explanation of each part-of-speech value given:

```
In [23]: print(spacy.explain('PROPN'))  
         print(spacy.explain('AUX'))
```

```
proper noun  
auxiliary
```

Span → In spaCy, a 'Span' is a slice from a 'Doc' object, representing a contiguous sequence of tokens within the document. Spans are useful for manipulating and analyzing segments of the original text, and they can range from single words to entire phrases or sentences.

SpaCy - Basic Functionality

Example for a Span →

```
In [26]: doc3 = nlp(u'Although commonly attributed to John Lennon from his song "Beautiful Boy", \
the phrase "Life is what happens to us while we are making other plans" was written by \
cartoonist Allen Saunders and published in Reader\'s Digest in 1957, when Lennon was 17.')
```

```
life_quote = doc3[16:30]
```

```
print(life_quote)
print()
print(type(life_quote))
```

"Life is what happens to us while we are making other plans"

<class 'spacy.tokens.span.Span'>

We can see that SpaCy recognize the slice sentence as a span that came from the original Doc.

We can also ask SpaCy to return a list of all the different sentences in the string.

```
In [28]: doc4 = nlp('This is the first sentence. This is another sentence. This is the last sentence.')
```

```
for sentence in doc4.sents:
    print(sentence)
```

This is the first sentence.

This is another sentence.

This is the last sentence.

SpaCy recognize each sentence on the original string by the '.' in the end of it and the context.

```
In [35]: print(f'word: {doc4[6]}, start a sentence: {doc4[6].is_sent_start}')
```

```
print(f'word: {doc4[8]}, start a sentence: {doc4[8].is_sent_start}')
```

word: This, start a sentence: True

word: another, start a sentence: False

Also SpaCy can recognize which words start a new sentence and which are not

SpaCy - Tokenization

As we learned one of main tasks of NLP model is tokenization which is splitting the string into separated tokens. Let's understand how the NLP model is performing this task.

SpaCy uses language-specific tokenization rules that are carefully designed and optimized for performance. These rules are preloaded in spaCy's language models.

When you load a spaCy language model (for example ``en_core_web_sm`` for English), it comes with predefined tokenization rules applicable to that language.

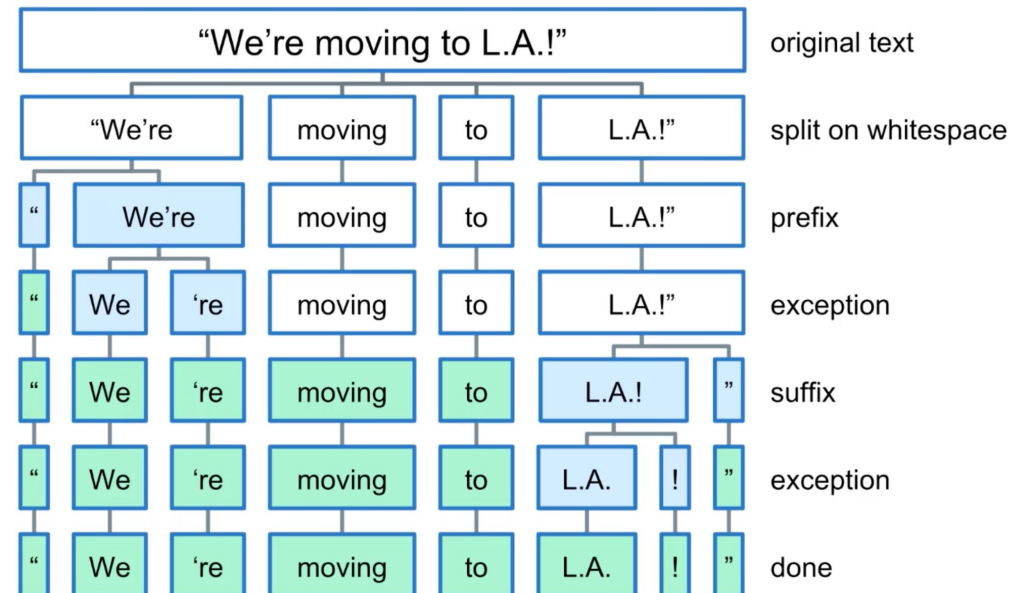
spaCy uses prefix, suffix, and infix handling to define how tokens should be split:

- **Prefixes** → Patterns that should be removed from the start of a token (for example, \$, (, etc...).
- **Suffixes** → Patterns to be removed from the end of a token (for example, ., ?, !, etc...).
- **Infixes** → Patterns to split within tokens (for example, hyphens in 'e-mail').

SpaCy - Tokenization

In addition, SpaCy use matching rules that are language-specific. These rules determine how text should be split into tokens based on whitespace, punctuation, and more complex linguistic rules.

- **Whitespace** → Splits tokens at spaces, tabs, and new lines.
- **Punctuation** → Handles common punctuation marks to split tokens accordingly (for example '!').
- **Complex linguistic rules** → Deals with contractions (for example "don't" → "do" + "n't"), abbreviations (for example "U.S."), and special cases.



SpaCy - Tokenization Example

Let's now see SpaCy tokenization in action with simple Python examples.

```
In [38]: mystring = '"We\'re moving to L.A.!"'

print(mystring)
print()

doc = nlp(mystring)

for token in doc:
    print(token.text)
```

"We're moving to L.A.!"

"

We

're

moving

to

L.A.

!

"

The opening quotation mark is treated as a separate token. spaCy recognizes quotation marks as distinct punctuation characters.

spaCy separates the contraction "'re" from "We", treating them as distinct tokens. This allows for better analysis of verbs.

spaCy recognizes "L.A." as an abbreviation and correctly handles the periods. It does not split "L.A." into multiple tokens.

The exclamation mark is treated as a separate token. spaCy splits punctuation marks to isolate them



ECOM SCHOOL

המכללה למקצועות הדיגיטל וההייטק

SpaCy - Tokenization Example

Let's take a look at another SpaCy tokenization example.

```
In [40]: doc8 = nlp('Apple to build a Hong Kong factory for $6 million')  
  
for token in doc8:  
    print(token.text, end=' | ')
```

Apple | to | build | a | Hong | Kong | factory | for | \$ | 6 | million |

Now, we can request from SpaCy to only print the 'entity' tokens from this doc, In addition we can also print the entity category and label.

```
In [41]: for entity in doc8.ents:  
    print(entity)  
    print(entity.label_)  
    print(str(spacy.explain(entity.label_)))  
    print('\n')
```

Apple
ORG
Companies, agencies, institutions, etc.

Hong Kong
GPE
Countries, cities, states

\$6 million
MONEY
Monetary values, including unit

SpaCy recognize Apple as an ORG
(organization entity)

SpaCy recognize Honk Kong as an GPE
(Geopolitical entity)

SpaCy recognize \$6 million as MONEY
entity 21

SpaCy - Noun Chunks

Noun Chunks → In Natural Language Processing (NLP) with spaCy, a "noun chunk" refers to a noun phrase, which is a continuous sequence of words that functions as the subject, object, or prepositional object in a sentence. A noun chunk typically consists of a noun (or pronoun) and its modifiers, such as determiners, adjectives, and prepositional phrases.

Why Noun Chunks are Important?

Noun chunks are semantically meaningful units that are often more informative and useful than individual words for certain tasks, such as information retrieval, text summarization, and syntactic parsing. By identifying noun chunks, you can easily extract and analyze the primary elements of the text.

For example → If we will take the sentence "The quick brown fox jumps over the lazy dog."

The noun chunks in this sentence are:

- "The quick brown fox"
- "the lazy dog"

SpaCy - Noun Chunks

noun_chunks → By using the '**noun_chunks**' property we can get a list of all the noun chunks SpaCy has identified in the Doc.

```
In [44]: doc9 = nlp("The quick brown fox jumps over the lazy dog.")  
  
for chunk in doc9.noun_chunks:  
    print(chunk.text)
```

```
The quick brown fox  
the lazy dog
```

```
In [45]: doc10 = nlp("Autonomous cars shift insurance liability toward manufacturers.")  
  
for chunk in doc10.noun_chunks:  
    print(chunk.text)
```

```
Autonomous cars  
insurance liability  
manufacturers
```



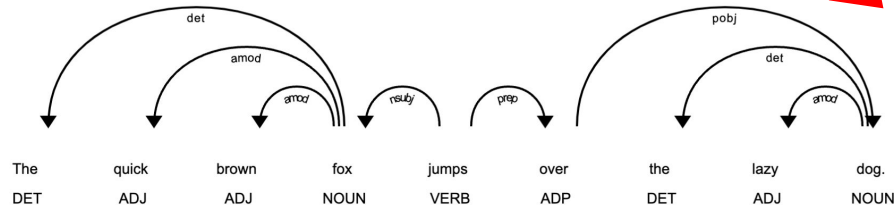
SpaCy - Display

SpaCy provides a number of visualization tools that can help you display and better understand the linguistic features of your text. One of the most commonly used visualization tools in spaCy is `displaCy`, a built-in visualizer for visualizing dependency parses and named entities.

For example →

```
In [49]: import spacy
         from spacy import displacy

         nlp = spacy.load("en_core_web_sm")
         text = "The quick brown fox jumps over the lazy dog."
         doc = nlp(text)
         displacy.render(doc, style="dep", jupyter=True, options={"distance": 100})
```



Visualizing Dependency Parsing

```
In [51]: from spacy import displacy

         doc = nlp('Over the last quarter Apple sold nearly 20 thousand iPods for a profit of $6 million.')
         displacy.render(doc, style='ent', jupyter=True)
```

Over the last quarter DATE Apple ORG sold nearly 20 thousand CARDINAL iPods PRODUCT for a profit of \$6 million MONEY .

Visualizing Named Entities

SpaCy - Stemming

Stemming → Stemming is a preprocessing technique used in NLP and information retrieval to reduce words to their base or root form. The primary goal of stemming is to ensure that words with the same base form or root are treated as equivalent, even if they appear in different forms. For example, "running," "runner," and "ran" could all be reduced to the stem "run".

Popular Stemming Algorithms:

- **Porter's Stemmer** → One of the most widely used stemming algorithms, particularly in English. It applies a series of rules to iteratively reduce words to their stems.
- **Snowball Stemmer** → An improvement over Porter's stemmer, this is also a rule-based algorithm but supports multiple languages.

Note: In SpaCy we don't have a build in stemmer, in case we want to use one we need to import it from a different NLP library called **NLTK**.

SpaCy - Stemming Python Example

Let's now see a Python example of using stemming with the NLTK library:

First let's install the library by running `'pip install nltk'`.

```
In [5]: pip install nltk

Collecting nltk
  Downloading nltk-3.8.1-py3-none-any.whl.metadata (2.8 kB)
Requirement already satisfied: click in /opt/anaconda3/envs/myenv/lib/python3.8/site-packages (from nltk) (8.1.7)
Collecting joblib (from nltk)
  Downloading joblib-1.4.2-py3-none-any.whl.metadata (5.4 kB)
Collecting regex<=2021.8.3 (from nltk)
  Downloading regex-2024.5.15-cp38-cp38-macosx_11_0_arm64.whl.metadata (40 kB)
  40.9/40.9 kB 486.7 kB/s eta 0:00:00 0:00:01
```

Now, let's create an **PorterStemmer** instance and use it on an array of strings that we want to stem.

```
In [4]: import nltk

from nltk.stem.porter import PorterStemmer

p_stemmer = PorterStemmer()

words = ['run', 'runner', 'running', 'ran', 'runs', 'easily', 'fairly', 'fairness']

for word in words:
    print(word + ' --> ' + p_stemmer.stem(word))

run --> run
runner --> runner
running --> run
ran --> ran
runs --> run
easily --> easili
fairly --> fairli
fairness --> fair
```

We can see that the words run, running and runs got the same stem word 'run'

We can see that the word fairness got the stem 'fair'

SpaCy - Stemming Python Example

We can also use the second, more advanced porter algorithm called '**Snowball**', when using 'snowball' stemmer we also need to provide the language because the algorithm support multiple languages.

```
In [5]: from nltk.stem.snowball import SnowballStemmer
s_stemmer = SnowballStemmer(language='english')
words = ['run', 'runner', 'running', 'ran', 'runs', 'easily', 'fairly', 'fairness']
for word in words:
    print(word+' --> '+s_stemmer.stem(word))

run --> run
runner --> runner
running --> run
ran --> ran
runs --> run
easily --> easili
fairly --> fair
fairness --> fair
```

We provide the language to the stemmer as parameter

We can see that the 'snowball' algorithm decided to stem the word 'fairly' to 'fair' and not to 'fairli' like the 'porter' algorithm

Lemmatization → Lemmatization in NLP transforms words into their base or dictionary form, known as the lemma. Unlike stemming, which simply trims word suffixes indiscriminately to reduce words to their root forms, lemmatization takes into account the context and morphological structure of the word to identify its correct lemma.



SpaCy - Lemmatization

Key features of Lemmatization:

- **Context-Aware** → Lemmatization uses the context around the word to ensure that the correct base form is identified. For example, it distinguishes that "better" would be lemmatized to "good" and "running" would be lemmatized to "run".
- **Language-Specific Rules** → It employs language-specific dictionaries and morphological analysis to ensure accuracy. This is why lemmatization tends to be more precise but also more computationally intensive compared to stemming.
- **Morphological Analysis** → Lemmatization considers the part of speech (POS) of a word, which is crucial for determining its correct lemma. For example, "running" can be a noun ("the running was difficult") or a verb ("he is running fast"), and lemmatization would correctly identify "running" as "run" if it's a verb, but it might consider additional POS-specific rules to treat it appropriately if it's a noun. This context-driven analysis ensures that the transformation preserves the original meaning and grammatical structure of the text.

Note: SpaCy library is using a built-in Lemmatization algorithm.

SpaCy - Lemmatization Python Example

Let's use the Lemmatization functionality built in SpaCy library, for this example we will provide SpaCy a sentence with different prons of the word 'run' and see what is the lemma of each word.

```
In [6]: doc1 = nlp(u"I am a runner running in a race because I love to run since I ran today")
```

```
for token in doc1:  
    print(token.text, '\t', token.pos_, '\t', token.lemma, '\t', token.lemma_)
```

I	PRON	4690420944186131903	I
am	AUX	10382539506755952630	be
a	DET	11901859001352538922	a
runner	NOUN	12640964157389618806	runner
running	VERB	12767647472892411841	run
in	ADP	3002984154512732771	in
a	DET	11901859001352538922	a
race	NOUN	8048469955494714898	race
because	SCONJ	16950148841647037698	because
I	PRON	4690420944186131903	I
love	VERB	3702023516439754181	love
to	PART	3791531372978436496	to
run	VERB	12767647472892411841	run
since	SCONJ	10066841407251338481	since
I	PRON	4690420944186131903	I
ran	VERB	12767647472892411841	run
today	NOUN	11042482332948150395	today

We can see that running, run and ran got the same lemma 'run'

A lemma hash value → each different lemma has it's own hash value. Lemmas that have the same hash are the same lemma



SpaCy - Stop Words

Stop words → Stop words refer to common words that are often removed from text during preprocessing because they are considered to carry less meaningful information for tasks like text analysis, search, and machine learning. These words typically include articles, prepositions, conjunctions, and some pronouns, such as "the," "and," "is," "in," "to," etc.

Importance of Stop Words:

- **Dimensionality Reduction** → Removing stop words reduces the number of tokens in the text, which can help improve the efficiency of various NLP algorithms.
- **Focus on Meaningful Content** → By filtering out these common words, the focus shifts to words that are more content-bearing and potentially more relevant for tasks like topic modeling, sentiment analysis, and information retrieval.

SpaCy - Stop Words

We can print all SpaCy stop words by calling the `Defaults.stop_words` property:

```
In [9]: # Print the set of spaCy's default stop words (remember that sets are unordered):
print(nlp.Defaults.stop_words)

{'whenever', 'along', 'for', 'get', 'mine', 'only', 'whereby', 'had', 'hundred', 'whatever', 'everyone', 'hereafter', 'besides', 'twenty', 'and', 'up', 'with', 'bottom', 'hereby', 'am', 'does', 'eleven', 'rather', 'did', 'since', 'name', 'can', 'then', 'over', 'me', 'four', 'yourself', 'under', 're', 'from', 'among', 'all', 'wherever', 'done', 'could', 'within', 'herself', 'other', 'nothing', 're', 'alone', 'not', 'he', 're', 'really', 'part', 'put', 'when', 'these', 'enough', 'd', 'itself', 'still', 'those', 'therefore', 'less', 'should', 'every', 'against', 'wh
```

We can also check if a specific word consider a stop word by calling the `'vocab'` functionally:

```
In [13]: print(nlp.vocab['myself'].is_stop)
print(nlp.vocab['mystery'].is_stop)

True
False
```

We can also customize the original stop words default list and add additional stop words in case we need to:

```
In [20]: nlp.Defaults.stop_words.add('btw')
nlp.vocab['btw'].is_stop = True
nlp.vocab['btw'].is_stop
```

```
Out[20]: True
```

We add the new word to the `stop_words` dictionary, make sure to use only lowercase letters

We mark the new word as stop word by passing `True` to the word `is_stop` property

SpaCy - Stop Words

Same way we can remove existing stop words from the list:

```
In [21]: nlp.Defaults.stop_words.remove('beyond')
         nlp.vocab['beyond'].is_stop = False
         nlp.vocab['beyond'].is_stop

Out[21]: False
```

Phrase Matching → Phrase matching refers to the process of identifying and locating specific sequences of tokens (words or phrases) within a larger body of text. Phrase matching can be an essential step for various NLP tasks such as information retrieval, entity recognition, text classification, and more. This technique allows you to detect predefined lists of phrases or terms that are of interest in the text data.

SpaCy provides a **PhraseMatcher** class which is designed to match sequences of tokens efficiently. This is especially useful for matching large sets of phrases and can leverage spaCy's tokenization and linguistic features for accurate matching.



SpaCy - Phrase Matching

Key Aspects of Phrase Matching:

- **Pattern Specification** → Phrases are typically specified as patterns or sequences of tokens you want to match in the text. These patterns can include simple words, or more complex structures that account for variations in syntax or punctuation.
- **Efficiency** → Phrase matching is often optimized for speed and efficiency, especially when dealing with large text or real-time text processing applications.
- **Customization** → It allows for customization to handle different cases, punctuation, varying word forms, or specific linguistic constructs according to the requirements of the application.
- **Applications** → Allowing Information extraction (identifying names, dates, or specific terminology), text categorization (detecting the presence of specific phrases that categorize text into predefined categories) and enhancing search engines by recognizing specific phrases to return more relevant results.



SpaCy - Phrase Matching Python Example

Let's now create a simple phrase matching in SpaCy. For this example we would like to match all different phrases of 'Solar Power' phrase. For example → If our text will contain 'solarpower' or 'solar-power' or 'Solar power' we would like SpaCy to recognized all as single phrase.

```
In [74]: from spacy.matcher import Matcher
matcher = Matcher(nlp.vocab)

pattern1 = [{'LOWER': 'solarpower'}] ## - SolarPower
pattern2 = [{'LOWER': 'solar'}, {'LOWER': 'power'}] ## Solar power
pattern3 = [{'LOWER': 'solar'}, {'IS_PUNCT': True}, {'LOWER': 'power'}] ## - Solar-power

matcher.add('SolarPower', [pattern1, pattern2, pattern3])
```

Matcher name and
list of phrases

- **Pattern1** → Designed to match a single token that is the word "solarpower" in any case ("solarpower", "SolarPower", "SOLARPOWER", etc...)
- **Pattern2** → Designed to match two consecutive tokens: "solar" followed by "power", again in any case ("solar power", "Solar Power", "SOLAR POWER", etc...)
- **Pattern3** → Designed to match the phrase "solar-power" where there is a punctuation mark (such as a hyphen) between "solar" and "power" ("solar-power", "Solar-Power", "SOLAR-POWER", etc...).

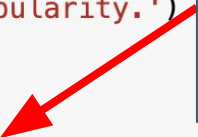
SpaCy - Phrase Matching Python Example

Now, let's provide the matcher a sentence to SpaCy with the phrases we search in the phrase matcher and see how it recognize them.

```
In [75]: doc = nlp('The Solar Power industry continues to grow as demand \
for solarpower increases. Solar-power cars are gaining popularity.')
found_matches = matcher(doc)
print(found_matches)

[(8656102463236116519, 1, 3), (8656102463236116519, 10, 11), (8656102463236116519, 13, 16)]
```

The matcher provide a list of all the matches. Each match contain the matcher id, the first phrase index and the last phrase index (not included)



We can also use the following code to make the matcher response be more readable:

```
In [76]: for match_id, start, end in found_matches:
          string_id = nlp.vocab.strings[match_id]
          span = doc[start:end]
          print(match_id, string_id, start, end, span.text)

8656102463236116519 SolarPower 1 3 Solar Power
8656102463236116519 SolarPower 10 11 solarpower
8656102463236116519 SolarPower 13 16 Solar-power
```

We can increase or decrease the start / end indexes to get more or less text context for each phrase match. This can help us understand the context of the phrase match in the text and not just the phrase itself.

SpaCy - Phrase Matching Python Example

Let's see another example with 'SolarPower' matcher, this time we will use different match phrases.

```
In [83]: pattern1 = [{'LOWER': 'solarpower'}]  
pattern2 = [{'LOWER': 'solar'}, {'IS_PUNCT': True, 'OP': '*'}, {'LEMMA': 'power'}]  
  
matcher.remove('SolarPower')  
matcher.add('SolarPower', [pattern1, pattern2])
```

Because we are using the same matcher name we first need to remove the previous matcher before add a new one.

- **Pattern1** → Designed to match a single token that is the word "solarpower" in any case ("solarpower", "SolarPower", "SOLARPOWER", etc...)
- **Pattern2** → Designed to match sequences where "solar" is followed by zero or more punctuation marks and then a token with the lemma "power" ("solar-power", "solar--power", "solar.power", "solar_ powered", etc...)

```
In [90]: doc2 = nlp('Solarpower energy runs solar-powered cars.')  
found_matches = matcher(doc2)  
  
for match_id, start, end in found_matches:  
    string_id = nlp.vocab.strings[match_id]  
    span = doc2[start:end]  
    print(match_id, string_id, start, end, span.text)  
  
8656102463236116519 SolarPower 0 1 Solarpower  
8656102463236116519 SolarPower 3 6 solar-powered
```

The matcher successfully identified 'Solarpower' and 'solar-powered'