Spacy

* Spacy is used for feature extraction in text data. Then we can apply ML model for classification problem/sentiment analysis
* Spacy is a free, open-source library for advanced Natural Language Processing (NLP) in Python. It's designed specifically for production use and helps you build applications that process and "understand" large volumes of text.
* spaCy comes with pretrained NLP models that can perform most common NLP tasks, such as tokenization, parts of speech (POS) tagging, [named entity recognition (NER)](https://www.machinelearningplus.com/nlp/training-custom-ner-model-in-spacy/), [lemmatization](https://www.machinelearningplus.com/nlp/lemmatization-examples-python/), transforming to word vectors etc.
* If you are dealing with a particular language, you can load the spacy model specific to the language using spacy.load() function.

|  |
| --- |
| Model Version |
| es\_core\_web\_md |
| fr\_depvec\_web\_lg |
| en\_core\_web\_md |
| en\_depent\_web\_md |
| en\_core\_web\_sm |
| en\_core\_web\_md |
| en\_depent\_web\_md |
| de\_core\_news\_md |
| en\_vectors\_glove\_md |

**English**

Available trained pipelines for English

## [en\_core\_web\_sm](https://spacy.io/models/en#en_core_web_sm)--------13 MB Size(Slow wifi)

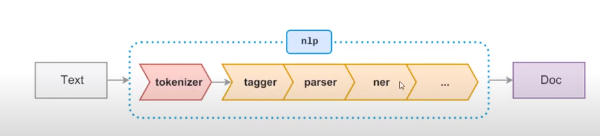
## [en\_core\_web\_md](https://spacy.io/models/en#en_core_web_md)--------44 MB(modrate wifi)

## [en\_core\_web\_lg](https://spacy.io/models/en#en_core_web_lg)----------742 MB

## [en\_core\_web\_trf](https://spacy.io/models/en#en_core_web_trf)----------438 MB

**spaCy’s Processing Pipeline**

The first step for a text string, when working with spaCy, is to pass it to an NLP object. This object is essentially a pipeline of several text pre-processing operations through which the input text string has to go through.

****

As you can see in the figure above, the NLP pipeline has multiple components, such as *tokenizer*, *tagger*, *parser*, *ner*, etc. So, the input text string has to go through all these components before we can work on it.

**If you are dealing with a particular language, you can load the spacy model specific to the language using spacy.load() function.**

**How to Read a String**

You can use spaCy to create a processed [Doc](https://spacy.io/api/doc) object, which is a container for accessing linguistic annotations, for a given input string.

**The Doc object**

First, call the loaded nlp object on the text. It should return a processed Doc object. But, what exactly is a Doc object ?

It is a sequence of tokens that contains not just the original text but all the results produced by the spaCy model after processing the text. Useful information such as the lemma of the text, whether it is a stop word or not, named entities, the word vector of the text and so on are pre-computed and readily stored in the Doc object.

The good thing is that you have complete control on what information needs to be pre-computed and customized.

Also, though the text gets split into tokens, no information of the original text is actually lost.

**Tokenization**

Tokenization is the process of converting a text into smaller sub-texts, based on certain predefined rules. For example, sentences are tokenized to words (and punctuation optionally). And paragraphs into sentences, depending on the context.

Tokenization  is the task of splitting the text in to meaningful segments, called as tokens. The input to the tokenizer is a Unicode text & the o/p is a doc object.

It allows you to identify the basic units in your text. These basic units are called tokens. Tokenization is useful because it breaks a text into meaningful units. These units are used for further analysis, like part of speech tagging.

spaCy preserves the starting index of the tokens. It’s useful for in-place word replacement. spaCy provides [various attributes](https://spacy.io/api/token#attributes) for the Token class

* text\_with\_ws prints token text with trailing space (if present).
* is\_alpha detects if the token consists of alphabetic characters or not.
* is\_punct detects if the token is a punctuation symbol or not.
* is\_space detects if the token is a space or not.
* shape\_ prints out the shape of the word.
* is\_stop detects if the token is a stop word or not.

**Sentence Detection**

Sentence Detection is the process of locating the start and end of sentences in a given text. This allows you to you divide a text into linguistically meaningful units. You’ll use these units when you’re processing your text to perform tasks such as part of speech tagging and entity extraction.

spaCy is correctly able to identify sentences in the English language, using a full stop(.) as the sentence delimiter. You can also customize the sentence detection to detect sentences on custom delimiters.

**Text-Preprocessing with spaCy**

As mentioned in the last section, there is ‘noise’ in the tokens. The words such as ‘the’, ‘was’, ‘it’ etc are very common and are referred as ‘stop words’.

Besides, you have punctuation like commas, brackets, full stop and some extra white spaces too. The process of removing noise from the doc is called **Text Cleaning** or **Preprocessing**.

**What is the need for Text Preprocessing?**

The outcome of the NLP task you perform, be it classification, finding sentiments, [topic modelling](https://www.machinelearningplus.com/nlp/topic-modeling-gensim-python/) etc, the quality of the output depends heavily on the quality of the input text used.

Stop words and punctuation usually (not always) don’t add value to the meaning of the text and can potentially impact the outcome. To avoid this, its might make sense to remove them and clean the text of unwanted characters can reduce the size of the corpus.

**How to identify and remove the stopwords and punctuation?**

The tokens in spacy have attributes which will help you identify if it is a stop word or not.

The token.is\_stop attribute tells you that. Likewise, token.is\_punct and token.is\_space tell you if a token is a punctuation and white space respectively.

**Stop Words**

Stop words are the most common words in a language. In the English language, some examples of stop words are the, are, but, and they. Most sentences need to contain stop words in order to be full sentences that make sense.

Generally, stop words are removed because they aren’t significant and distort the word frequency analysis. spaCy has a list of stop words for the English language

[**Morphology**](https://spacy.io/usage/linguistic-features#morphology)

Inflectional morphology is the process by which a root form of a word is modified by adding prefixes or suffixes that specify its grammatical function but do not change its part-of-speech. We say that a **lemma** (root form) is **inflected** (modified/combined) with one or more **morphological features** to create a surface form

Morphological features are stored in the [MorphAnalysis](https://spacy.io/api/morphology" \l "morphanalysis) under Token.morph, which allows you to access individual morphological features.

**Lemmatization**

Have a look at these words: **“played”, “playing”, “plays”, “play”.**

These words are not entirely unique, as they all basically refer to the root word: “play”. Very often, while trying to interpret the meaning of the text using NLP, you will be concerned about the root meaning and not the tense.

For algorithms that work based on the number of occurrences of the words, having multiple forms of the same word will reduce the number of counts for the root word, which is ‘play’ in this case.

Hence, counting “played” and “playing” as different tokens will not help.

**Lemmatization is the method of converting a token to it’s root/base form.**

Fortunately, spaCy provides a very easy and robust solution for this and is considered as one of the optimal implementations.

After you’ve formed the Document object (by using nlp()), you can access the root form of every token through Token.lemma\_ attribute.

Lemmatization is the process of reducing inflected forms of a word while still ensuring that the reduced form belongs to the language. This reduced form or root word is called a lemma.

For example, organizes, organized and organizing are all forms of organize. Here, organize is the lemma. The inflection of a word allows you to express different grammatical categories like tense (organized vs organize), number (trains vs train), and so on. Lemmatization is necessary because it helps you reduce the inflected forms of a word so that they can be analyzed as a single item. It can also help you normalize the text.

In this example, organizing reduces to its lemma form organize. If you do not lemmatize the text, then organize and organizing will be counted as different tokens, even though they both have a similar meaning. Lemmatization helps you avoid duplicate words that have similar meanings

**Strings to Hashes**

whenever you create a doc , the words of the doc are stored in the Vocab.

Also, consider you have about 1000 text documents each having information about various clothing items of different brands. The chances are, the words “shirt” and “pants” are going to be very common. Each time the word “shirt” occurs, if spaCy were to store the exact string , you’ll end up losing huge memory space.

But this doesn’t happen. Why?

spaCy **hashes** or converts each string to a unique ID that is stored in the StringStore.

But, what is StringStore?

It’s a dictionary **mapping of hash values to strings**, for example 10543432924755684266 –> box

You can print the hash value if you know the string and vice-versa. This is contained in nlp.vocab.strings as shown below.

Interestingly, a word will have the same hash value irrespective of which document it occurs in or which spaCy model is being used. So your results are reproducible even if you run your code in some one else’s machine.

**Lexical attributes of spaCy**

Recall that we used is\_punct and is\_space attributes in Text Preprocessing. They are called as **‘lexical attributes’**.

The spaCy model provides many useful lexical attributes. These are the attributes of Token object, that give you information on the type of token.

For example, you can use like\_num attribute of a token to check if it is a number. Let’s print all the numbers in a text.

* token.is\_alpha : Returns True if the token is an alphabet
* token.is\_ascii : Returns True if the token belongs to ascii characters
* token.is\_digit : Returns True if the token is a number(0-9)
* token.is\_upper : Returns True if the token is upper case alphabet
* token.is\_lower : Returns True if the token is lower case alphabet
* token.is\_space : Returns True if the token is a space ‘ ‘
* token.is\_bracket : Returns True if the token is a bracket
* token.is\_quote : Returns True if the token is a quotation mark
* token.like\_url : Returns True if the token is similar to a URl (link to website)

**Detecting Email Addresses**

Consider you have a text document about details of various employees.

What if you want all the emails of employees to send a common email ?

You can tokenize the document and check which tokens are emails through like\_email attribute. like\_email returns True if the token is a email

**Word Frequency**

You can now convert a given text into tokens and perform statistical analysis over it. This analysis can give you various insights about word patterns, such as common words or unique words in the text

**Part of Speech Tagging**

In English grammar, the parts of speech tell us what is the function of a word and how it is used in a sentence. Some of the common parts of speech in English are Noun, Pronoun, Adjective, Verb, Adverb, etc.

POS tagging is the task of automatically assigning POS tags to all the words of a sentence. It is helpful in various downstream tasks in NLP, such as feature engineering, language understanding, and information extraction.

Part of speech or POS is a grammatical role that explains how a particular word is used in a sentence. There are eight parts of speech:

1. Noun
2. Pronoun
3. Adjective
4. Verb
5. Adverb
6. Preposition
7. Conjunction
8. Interjection

Part of speech tagging is the process of assigning a POS tag to each token depending on its usage in the sentence. POS tags are useful for assigning a syntactic category like noun or verb to each word.

Consider a sentence, **“Emily likes playing football”**.

Here , Emily is a NOUN , and playing is a VERB. Likewise , each word of a text is either a noun, pronoun, verb, conjection, etc. These tags are called as Part of Speech tags (POS).

How to identify the part of speech of the words in a text document ?

It is present in the pos\_ attribute.

Consider you have a text document of reviews or comments on a post. Apart from genuine words, there will be certain junk like “etc” which do not mean anything. How can you remove them ?

Using spacy’s pos\_ attribute, you can check if a particular token is junk through token.pos\_ == 'X' and remove them.

For better understanding of various POS of a sentence, you can use the visualization function displacy of spacy.

## Visualization: Using displaCy

spaCy comes with a built-in visualizer called displaCy. You can use it to visualize a dependency parse or named entities in a browser or a [Jupyter notebook](https://realpython.com/jupyter-notebook-introduction/).

## Preprocessing Functions

You can create a preprocessing function that takes text as input and applies the following operations:

* Lowercases the text
* Lemmatizes each token
* Removes punctuation symbols
* Removes stop words

A preprocessing function converts text to an analyzable format. It’s necessary for most NLP tasks.

## Rule-Based Matching Using spaCy

Rule-based matching is one of the steps in extracting information from unstructured text. It’s used to identify and extract tokens and phrases according to patterns (such as lowercase) and grammatical features (such as part of speech).

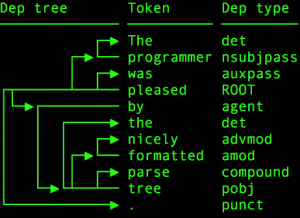
Rule-based matching can use [regular expressions](https://realpython.com/regex-python/) to extract entities (such as phone numbers) from an unstructured text. It’s different from extracting text using regular expressions only in the sense that regular expressions don’t consider the lexical and grammatical attributes of the text.

With rule-based matching, you can extract a first name and a last name, which are always proper nouns

Rule-based matching helps you identify and extract tokens and phrases according to lexical patterns (such as lowercase) and grammatical features (such as part of speech).

## Dependency Parsing Using spaCy

Every sentence has a grammatical structure to it and with the help of dependency parsing, we can extract this structure. It can also be thought of as a directed graph, where nodes correspond to the words in the sentence and the edges between the nodes are the corresponding dependencies between the word.

[](https://cdn.analyticsvidhya.com/wp-content/uploads/2020/03/dep_tree.png)

Dependency parsing is the process of extracting the dependency parse of a sentence to represent its grammatical structure. It defines the dependency relationship between headwords and their dependents. The head of a sentence has no dependency and is called the root of the sentence. The verb is usually the head of the sentence. All other words are linked to the headword.

The dependencies can be mapped in a directed graph representation:

* Words are the nodes.
* The grammatical relationships are the edges.

Dependency parsing helps you know what role a word plays in the text and how different words relate to each other. It’s also used in shallow parsing and named entity recognition.

### [Noun chunks](https://spacy.io/usage/linguistic-features#noun-chunks)

Noun chunks are “base noun phrases” – flat phrases that have a noun as their head. You can think of noun chunks as a noun plus the words describing the noun – for example, “the lavish green grass” or “the world’s largest tech fund”. To get the noun chunks in a document, simply iterate over [Doc.noun\_chunks](https://spacy.io/api/doc" \l "noun_chunks).

### [Navigating the parse tree](https://spacy.io/usage/linguistic-features#navigating)

spaCy uses the terms **head** and **child** to describe the words **connected by a single arc** in the dependency tree. The term **dep** is used for the arc label, which describes the type of syntactic relation that connects the child to the head. As with other attributes, the value of .dep is a hash value. You can get the string value with .dep\_.

### [Visualizing dependencies](https://spacy.io/usage/linguistic-features#displacy)

The best way to understand spaCy’s dependency parser is interactively. To make this easier, spaCy comes with a visualization module. You can pass a Doc or a list of Doc objects to displaCy and run [displacy.serve](https://spacy.io/api/top-level" \l "displacy.serve) to run the web server, or [displacy.render](https://spacy.io/api/top-level" \l "displacy.render) to generate the raw markup. If you want to know how to write rules that hook into some type of syntactic construction, just plug the sentence into the visualizer and see how spaCy annotates it.

[Disabling the parser](https://spacy.io/usage/linguistic-features#disabling)

In the [trained pipelines](https://spacy.io/models) provided by spaCy, the parser is loaded and enabled by default as part of the [standard processing pipeline](https://spacy.io/usage/processing-pipelines). If you don’t need any of the syntactic information, you should disable the parser. Disabling the parser will make spaCy load and run much faster. If you want to load the parser, but need to disable it for specific documents, you can also control its use on the nlp object. For more details, see the usage guide on [disabling pipeline components](https://spacy.io/usage/processing-pipelines/#disabling).

## Navigating the Tree and Sub tree

The dependency parse tree has all the properties of a [tree](https://en.wikipedia.org/wiki/Tree_(data_structure)). This tree contains information about sentence structure and grammar and can be traversed in different ways to extract relationships.spaCy provides attributes like children, lefts, rights, and subtree to navigate the parse tree

## Shallow Parsing

Shallow parsing, or chunking, is the process of extracting phrases from unstructured text. Chunking groups adjacent tokens into phrases on the basis of their POS tags. There are some standard well-known chunks such as noun phrases, verb phrases, and prepositional phrases.

## Noun Phrase Detection

A noun phrase is a phrase that has a noun as its head. It could also include other kinds of words, such as adjectives, ordinals, determiners. Noun phrases are useful for explaining the context of the sentence. They help you infer what is being talked about in the sentence.

spaCy has the property noun\_chunks on Doc object. You can use it to extract noun phrases

## Verb Phrase Detection

A verb phrase is a syntactic unit composed of at least one verb. This verb can be followed by other chunks, such as noun phrases. Verb phrases are useful for understanding the actions that nouns are involved in. spaCy has no built-in functionality to extract verb phrases, so you’ll need a library called [textacy](https://chartbeat-labs.github.io/textacy/)

## Named Entity Recognition

A named entity is a “real-world object” that’s assigned a name – for example, a person, a country, a product or a book title. spaCy can **recognize various types of named entities in a document, by asking the model for a prediction**. Because models are statistical and strongly depend on the examples they were trained on, this doesn’t always work *perfectly* and might need some tuning later, depending on your use case.

Named entities are available as the ents property of a Doc:

As you can see in the figure above, the NLP pipeline has multiple components, such as *tokenizer*, *tagger*, *parser*, *ner*, etc. So, the input text string has to go through all these components before we can work on it.

Named Entity Recognition (NER) is the process of locating named entities in unstructured text and then classifying them into pre-defined categories, such as person names, organizations, locations, monetary values, percentages, time expressions, and so on.

You can use NER to know more about the meaning of your text. For example, you could use it to populate tags for a set of documents in order to improve the keyword search. You could also use it to categorize customer support tickets into relevant categories. spaCy has the property ents on Doc objects. You can use it to extract named entities

spacy.explain gives descriptive details about an entity label. The spaCy model has a pre-trained [list of entity classes](https://spacy.io/api/annotation#named-entities). You can use displaCy to visualize these entities

Have a look at this text “John works at Google1″. In this, ” John ” and ” Google ” are names of a person and a company. These words are referred as **named-entities**. They are real-world objects like name of a company , place,etc..

How can find all the named-entities in a text ?

Using spaCy’s ents attribute on a document, you can access all the named-entities present in the text.

Each named entity belongs to a category, like name of a person, or an organization, or a city, etc. The common Named Entity categories supported by spacy are :

PERSON : Denotes names of people

GPE : Denotes places like counties, cities, states.

ORG : Denotes organizations or companies

WORK\_OF\_ART : Denotes titles of books, fimls,songs and other arts

PRODUCT : Denotes products such as vehicles, food items ,furniture and so on.

EVENT : Denotes historical events like wars, disasters ,etc…

LANGUAGE : All the recognized languages across the globe.

You can access the same through .label\_ attribute of spacy. It prints the label of named entities

spaCy also provides special visualization for NER through displacy. Using displacy.render() function, you can set the style=ent to visualize.

## Rule based Matching

Consider the sentence “Windows 8.0 has become outdated and slow. It’s better to update to Windows 10”. What if you want to extracts all versions of Windows mentioned in the text ?

There will be situations like these, where you’ll need extract specific pattern type phrases from the text. This is called **Rule-based matching**.

Rule-based matching in spacy allows you write your own rules to find or extract words and phrases in a text. spacy supports three kinds of matching methods :

1. Token Matcher
2. Phrase Matcher
3. Entity Ruler

## Token Matcher

spaCy features a rule-matching engine, the [Matcher](https://spacy.io/api/matcher), that operates over tokens, similar to regular expressions. The rules can refer to token annotations (e.g. the token text or tag\_, and flags (e.g. IS\_PUNCT). The rule matcher also lets you pass in a custom callback to act on matches – for example, to merge entities and apply custom labels. You can also associate patterns with entity IDs, to allow some basic entity linking or disambiguation. To match large terminology lists, you can use the [PhraseMatcher](https://spacy.io/api/phrasematcher), which accepts Doc objects as match patterns.

### [Adding patterns](https://spacy.io/usage/rule-based-matching#adding-patterns)

Let’s say we want to enable spaCy to find a combination of three tokens:

1. A token whose **lowercase form matches “hello”**, e.g. “Hello” or “HELLO”.
2. A token whose is\_punct**flag is set to**True, i.e. any punctuation.
3. A token whose **lowercase form matches “world”**, e.g. “World” or “WORLD”.

spaCy supports a rule based matching engine Matcher, which operates over individual tokens to find desired phrases.

The procedure to implement a token matcher is:

1. Initialize a Matcher object
2. Define the pattern you want to match
3. Add the pattern to the matcher
4. Pass the text to the matcher to extract the matching positions.

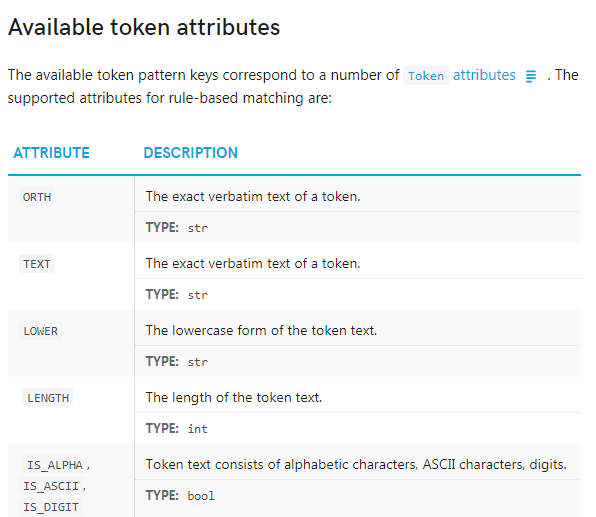
Now , you can add the pattern to your Matcher through matcher.add() function

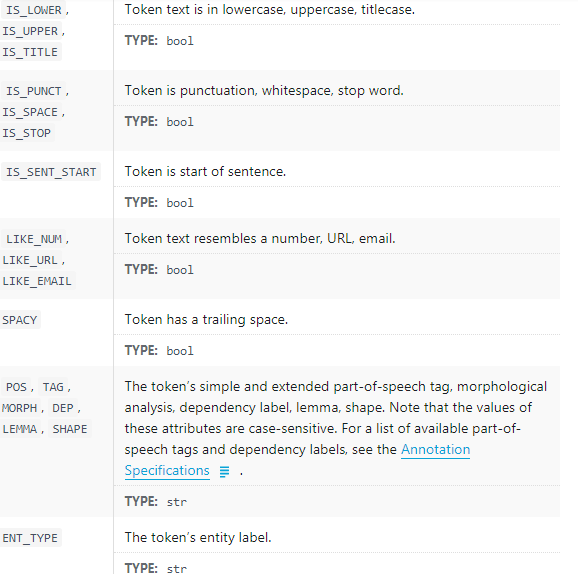
* match\_id – a custom id for your matcher . In this case I use ” Versionfinder”
* match\_on– It is an optional parameter, where you can call functions when a match is found. Otherwise, use None
* \*patterns – You need to pass your pattern (list of dicts describing tokens)
* Passing the Doc to matcher() returns a list of tuples as shown above. Each tuple has the structure –(match\_id, start, end).
* match\_id denotes the hash value of the matching string.You can find the string corresponding to the ID in nlp.vocab.strings. The start and end denote the starting and ending token numbers of the document, which is a match.

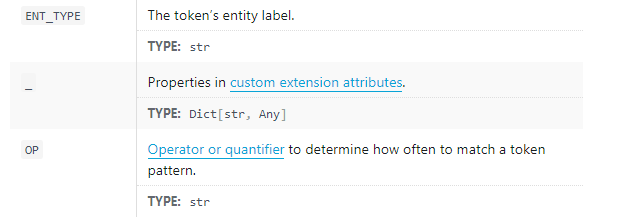
How to extract the phrases that matches from this list of tuples ?

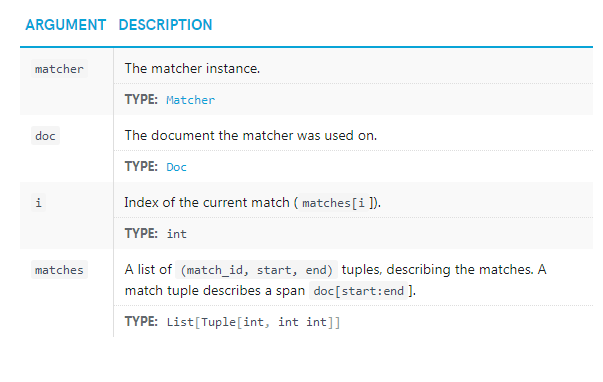
A slice of a Doc object is referred as Span. If you have your spacy doc , and start and end indices, you extract a slice / span of the text through :Span=doc[start:end].

Below code makes use of this to extract matching phrases with the help of list of tuples desired\_matches.









## Phrase Matcher

Using Matcher of spacy you can identify token patterns as seen above. But when you have a phrase to be matched, using Matcher will take a lot of time and is not efficient.

spaCy provides PhraseMatcher which can be used when you have a large number of terms(single or multi-tokens) to be matched in a text document. Writing patterns for Matcher is very difficult in this case. PhraseMatcher solves this problem, as you can pass Doc patterns rather than Token patterns.

The procedure to use PhraseMatcher is very similar to Matcher.

1. Initialize a PhraseMatcher object with a vocab.
2. Define the terms you want to match
3. Add the pattern to the matcher
4. Run the text through the matcher to extract the matching positions.

Another useful feature of PhraseMatcher is that while intializing the matcher, you have an option to use the parameter attr, using which you can set rules for how the matching has to happen.

How to use attr?

Setting a attr to match on will change the token attributes that will be compared to determine a match. For example, if you use attr='LOWER', then case-insensitive matching will happen.

## Entity Ruler

Entity patterns are dictionaries with two keys: "label", specifying the label to assign to the entity if the pattern is matched, and "pattern", the match pattern. The entity ruler accepts two types of patterns:

1. Phrase patterns for exact string matches (string).
2. {"label": "ORG", "pattern": "Apple"}
3. Token patterns with one dictionary describing one token (list).

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

While trying to detect entities, some times certain names or organizations are not recognized by default. It might be because they are small scale or rare. Wouldn’t it be better to improve accuracy of our doc.ents\_ method ?

spaCy provides a more advanced component EntityRuler that let’s you match named entities based on pattern dictionaries. Overall, it makes Named Entity Recognition more efficient.

It is a pipeline supported component

What type of patterns do you pass to the EntityRuler ?

Basically, you need to pass a list of dictionaries, where each dictionary represents a pattern to be matched.

Each dictionary has two keys "label" and "pattern".

* label : Holds the entity type as values eg: PERSON, GPE, etc
* pattern: Holds the the matcher pattern as values eg: John, Calcutta, etc

For example, let us consider a situation where you want to add certain book names under the entity label WORK\_OF\_ART.

What will be your pattern ?

My label will be WORK\_OF\_ART and pattern will contain the book names I wish to add. Below code demonstrates the same.

You can add pattern to the ruler through add\_patterns() function

How can you apply the EntityRuler to your text ?

You can add it to the nlp model through add\_pipe() function. It Adds the ruler component to the processing pipeline

## Word Vectors and similarity

Word Vectors are numerical vector representations of words and documents. The numeric form helps understand the semantics about the word and can be used for NLP tasks such as classification.

Because, vector representation of words that are similar in meaning and context appear closer together.

spaCy models support inbuilt vectors that can be accessed through directly through the attributes of Token and Doc. How can you check if the model supports tokens with vectors ?

First, load a spaCy model of your choice. Here, I am using the medium model for english en\_core\_web\_md. Next, tokenize your text document with nlp boject of spacy model.

You can check if a token has in-buit vector through Token.has\_vector attribute.

How to access the vector of the tokens?

You can access through token.vector method. Also ,token.vector\_norm attribute stores L2 norm of the token’s vector representation.

## How to find similarity of two tokens?

Identifying similarity of two words or tokens is very crucial . It is the base to many everyday NLP tasks like text classification , recommendation systems, etc.. It is necessary to know how similar two sentences are , so they can be grouped in same or opposite category.

Every Doc or Token object has the function similarity(), using which you can compare it with another doc or token.

Know about [cosine similarity](https://www.machinelearningplus.com/nlp/cosine-similarity/).

It returns a float value. Higher the value is, more similar are the two tokens or documents.

## Merging and Splitting Tokens with retokenize

When nlp object is called on a text document, spaCy first tokenizes the text to produce a Docobject. The Tokenizer is the pipeline component responsible for segmenting the text into tokens.

Sometime tokenization splits a combined word into two tokens instead of keeping it as one unit.

How to combine the tokens?

spaCy provides Doc.retokenize , a context manager that allows you to merge and split tokens. For merging two or more tokens , you can make use of the retokenizer.merge() function.

How to use the retokenizer.merge () ?

The input arguments shall be:

* span : You can pass a span, which contains the slice of doc you wanted to be treated as a single token. In this case, John wick is stored in a span and passed as input. span=doc[0:2]
* attrs : You can use it to set attributes to set on the merged token. Here, I want to set the POS (part of speech tag) for “John Wick” as PROPN.(proper noun). You can use attrs={"POS" : "PROPN"} to achieve it.

spaCy provides retokenzer.split() method to serve this purpose.

The input parameters are :

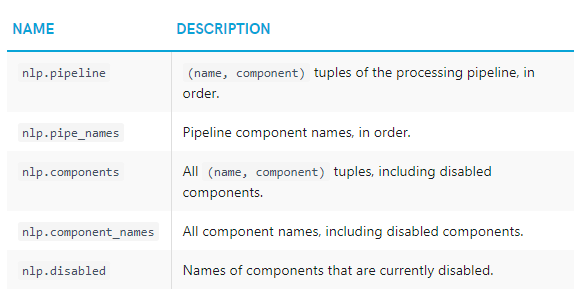
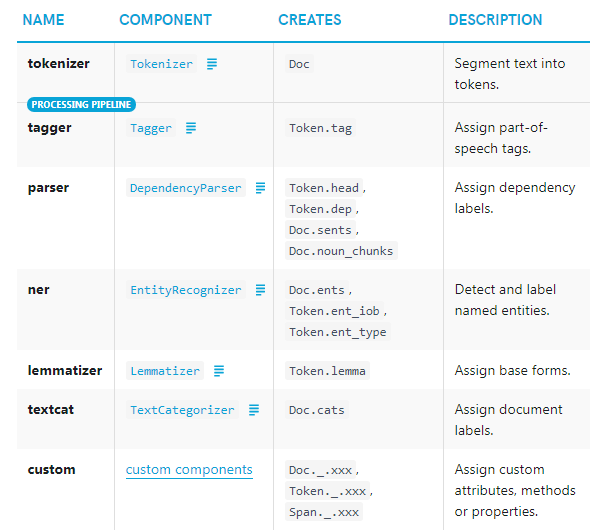
* token : The token of the doc which has to be split
* orths : A list of texts, matching the original token. This is to tell the retokinzer how to split the token
* heads : List of token or (token, subtoken) tuples specifying the tokens to attach the newly split subtokens to.
* attrs : You can pass a dictionary to set attributes on all split tokens. Attribute names mapped to list of per-token attribute values.

## spaCy pipelines

You have used tokens and docs in many ways till now. In this section, let’s dive deeper and understand the basic pipeline behind this.

When you call the nlp object on spaCy, the text is segmented into tokens to create a Doc object. Following this, various process are carried out on the Doc to add the attributes like POS tags, Lemma tags, dependency tags,etc..

This is referred as the **Processing Pipeline**

****

## What are pipeline components ?

The processing pipeline consists of components, where each component performs it’s task and passes the Processed Doc to the next component. These are called as **pipeline components**.

spaCy provides certain in-built pipeline components. Let’s look at them.

The built-in pipeline components of spacy are :

* Tokenizer : It is responsible for segmenting the text into tokens are turning a Doc object. This the first and compulsory step in a pipeline.
* Tagger : It is responsible for assigning Part-of-speech tags. It takes a Doc as input and createsDoc[i].tag
* DependencyParser : It is known as **parser**. It is responsible for assigning the dependency tags to each token. It takes a Doc as input and returns the processed Doc
* EntityRecognizer : This component is referred as **ner**. It is responsible for identifying named entities and assigning labels to them.
* TextCategorizer : This component is called **textcat**. It will assign categories to Docs.
* EntityRuler : This component is called *\* entity\_ruler*\*.It is responsible for assigning named entitile based on pattern rules. Revisit Rule Based Matching to know more.
* Sentencizer : This component is called \*\*sentencizer\*\* and can perform rule based sentence segmentation.
* merge\_noun\_chunks : It is called **merge***noun***chunks**. This component is responsible for merging all noun chunks into a single token. It has to be add in the pipeline after tagger and parser.
* merge\_entities : It is called **merge\_entities** .This component can merge all entities into a single token. It has to added after the ner.
* merge\_subtokens : It is called **merge\_subtokens**. This component can merge the subtokens into a single token.

These are the various in-built pipeline components. It is not necessary for every spaCy model to have each of the above components.

After loading a spaCy model , you check or inspect what pipeline components are present.

## How to inspect the pipeline ?

After loading the spacy model and creating a Language object nlp, you view the list of pipeline components present by default using nlp.pipe\_names attribute

You can also check if a particular component is present in the pipline through nlp.has\_pipe. You have to pass the name of the component like tagger , ner ,textcat as input.

## How to add a component to the pipeline ?

You can add a component to the processing pipeline through nlp.add\_pipe() method. You have to pass the component to be added as input.

The component can also be written by you, i.e, custom made pipeline component. (We will come to this later). In case you want to add an in-built component like textcat, how to do it ?

You can use nlp.create\_pipe() and pass the component name to get any in-built pipeline component.

## How to specify where you want to add the new component?

The nlp.add\_pipe() method provides various arguments for this. You can set one among before, after, first or last to True.

By default, last=True is used.

If you want textcat before ner, you can set before=ner. If you want it to be at first you can set first=True. Just remeber that you should not pass more than one of these arguments as it will lead to contradiction.

## How to remove, replace and rename pipepline components ?

It is always advisable to have only the necessary components in the processing pipeline. Otherwise, the component will create and store attributes which are not going to be used . This causes waste of memory and also takes more time to process.

To avoid this , you can remove unnecessary pipeline components, using nlp.remove\_pipe() method .

You can rename a pipeline component giving your own custom name through nlp.rename\_pipe() method.

spaCy also allows you to create your own custom pipelines. We shall discuss more on this later. When you have to use different component in place of an existing component, you can use nlp.replace\_pipe() method.

## Methods for Efficient processing

While dealing with huge amount of text data , the process of converting the text into processed Doc ( passing through pipeline components) is often time consuming.

In this section , you’ll learn various methods for different situations to help you reduce computational expense.

Let’s say you have a list of text data , and you want to process them into Doc onject. The traditional method is to call nlp object on each of the text data .

Another efficient method of creating the doc is using nlp.pipe() method. You can pass the list as input to this. This method takes less time , as it **processes the texts as a stream** rather than individually.

From above output , you can observe that time taken is less using nlp.pipe() method. When the amount of data will be very large, the time difference will be very important.

Another way to keep the process efficient is using only the pipeline components you need. For example , if your problem does not use POS tags , then tagger is not necessary.

The unnecessary pipeline components can be disabled to improve loading speed and efficiency.

## How to disable pipeline components in spaCy?

There are two common cases where you will need to disable pipeline components.

First case is when you don’t need the component throughout your project. In this case, you can disable the component while loading the spacy model itself. This will save you a great deal of time. It can be done through the disable argument of spacy.load() function.

The second case is when you need the component during specific times of your task, but not throughout. So, here you’ll have to load the components and their weights.

At some point, if you need a Doc object with only part-of speech tags, there is no need for ner and parser . You can use the disable keyword argument on nlp.pipe() method to temporarily disable the components during processing.

An extension of this method is to disable pipeline components for a whole block.

The context manager nlp.disable\_pipes() can be used for disabling components for a whole block. You can write the code which doesn’t require the component inside the block. For any code written outside the block , the pipeline components are available.

## Creating custom pipeline components

First, write a function that takes a Doc as input, performs neccessary tasks and returns a new Doc. Then, add this function to the spacy pipeline through nlp.add\_pipe() method.

The parameters of add\_pipe you have to provide :

* component : You have to pass the function\_name as input . This serves as our component
* name : You can assign a name to the component. The component can be called using this name. If you don’t provide any ,the function\_name will be taken as name of the component
* first,last : If you want the new component to be added first or last ,you can setfirst=True or last=True accordingly.
* before , after : If you want to add the component specifically before or after another component , you can use these arguments.

Note that you can set only one among first, last, before, after arguments, otherwise it will lead to error.

Let’s discuss a set of examples to understand the implementation.

Say you want to add a pipeline component that will print the length of the doc, and also the various types of named entities present in the doc.

First step – Write a function my\_custom\_component() to perform the tasks on the input doc and return it.

Second step – Add the component to the pipeline using nlp.add\_pipe(my\_custom\_component). Also , you need to insert this component after ner so that entities will bw stored in doc.ents

#### Pipeline component example

Let’s level up and try implementing more complex case.

Consider you have a doc and you want to add a pipeline component that can find some book names present and add add them to doc.ents.

To make this possible , you can create a custom pipeline component that uses PhraseMatcherto find book names in the doc and add the to the doc.ents attribute.

I suggest you to scroll up and have another read through Rule based matching with PhraseMatcher . Let’s first import and initialize the matcher with vocab . Next, write the pattern with names of books you want to be matched. Add the pattern to the matcher using matcher.add() by passing the pattern.