✨ Member-only story

In-Depth spaCy Tutorial For Beginners in NLP

Learn the Scikit-learn of Natural Language Processing

Bex T. · Follow

Published in Towards Data Science · 9 min read · Jun 9, 2022

Photo by Anni Roenkae

Introduction

No, we won't be building language models with billions of parameters today. We will start smaller and learn about the basics of NLP with spaCy. We will closely look at how the library works and how you can use it to solve beginner/intermediate NLP problems with ease.

The post is already long, so I'll cut it here and jump to the meat of the article.

standard with vast features to solve many NLP tasks with state-of-the-art speed, accuracy, and performance.

At its core are pipelines, which you can think of as language-specific models already trained on millions of text instances.

It is also at the head of the spaCy ecosystem that includes dozens of libraries and tools such as Prodigy, Forte, displaCy, explacy, ADAM, Coreferee, etc.

spaCy can also shake hands with custom models from TensorFlow, PyTorch, and other frameworks.

Right now, spaCy supports 66 languages as separate pipelines, and new languages are being added slowly.

Basics of spaCy

Before seeing how spaCy works, let's install it:

```
pip install -U spacy
```

spaCy has three pipelines for English, with varying sizes and functionality for complex tasks. In this tutorial, we will only need to install the small and medium pipelines, but I included the large one as well for completeness:

```
python -m spacy download en_core_web_sm  # 12 MB
python -m spacy download en_core_web_md  # 31 MB
python -m spacy download en_core_web_lg  # 382 MB
```

11801.sh hosted with ❤ by GitHub view raw

After importing spaCy, we need to load one of the pipelines we just installed. For now, we will load the small one and store it to `nlp`:

```
import spacy

txt = "The tallest living man is 37-year-old Sultan Kosen, from Turkey,"
      " who is 8 feet, 2.8 inches, who set the record in 2009."

# Create the Language object
nlp = spacy.load("en_core_web_sm")
nlp
<spacy.lang.en.English at 0x1f33e3af550>
```

11802.py hosted with ❤ by GitHub view raw

It is a convention to name any loaded language models `nlp` in the spaCy ecosystem. This object can now be called on any text to start information extraction:

```
# Create the Doc object
doc = nlp(txt)
```

The `doc` object is also a convention, and now it is already filled with extra information about the given text's sentences and words.

In general, the `doc` object is just an iterator:

```
for token in doc[:5]:
    print(token)

[OUT]:

The
tallest
living
man
is
```

11803.py hosted with ❤ by GitHub view raw

You can use slicing or indexing notations to extract individual tokens:

```
>>> type(token)
spacy.tokens.token.Token
>>> len(doc)
31
```

> *Tokenization is splitting sentences into words and punctuation. A single token can be a word, a punctuation or a noun chunk, etc.*

If you extract more than one token, then you have a span object:

```
>>> span = doc[:5]
>>> type(span)

spacy.tokens.span.Span

>>> span.text

'The tallest living man is'
```

spaCy is also built for memory efficiency. That's why both `token` and `span` objects are just views of the `doc` object. There is no duplication.

The pre-trained English pipeline and many other pipelines have language-specific rules for tokenization and extracting their lexical attributes. Here are 6 of such attributes:

```
print("Index:    ", [token.i for token in doc[3:10]])
print("Text:     ", [token.text for token in doc[3:10]])
print("is_alpha: ", [token.is_alpha for token in doc[3:10]])
print("is_punct: ", [token.is_punct for token in doc[3:10]])
print("like_num: ", [token.like_num for token in doc[3:10]])
print("Base word:", [token.lemma_ for token in doc[3:10]])

[OUT]:

Index:     [3, 4, 5, 6, 7, 8, 9]
Text:      ['man', 'is', '37', '-', 'year', '-', 'old']
is_alpha:  [True, True, False, False, True, False, True]
is_punct:  [False, False, False, True, False, True, False]
like_num:  [False, False, True, False, False, False, False]
Base word: ['man', 'be', '37', '-', 'year', '-', 'old']
```

11804.py hosted with ❤ by GitHub view raw

Some interesting attributes are `lemma_`, which returns the base word stripped from any suffixes, prefixes, tense, or any other grammatical attributes, and the `like_num` which recognizes both literal and lettered numbers.

You will be spending most of your time on these four objects — `nlp`, `doc`, `token` and `span`. Let's take a closer look at how they are related.

Architecture and core data structures

Let's start again with the `nlp`, which is a Language object under the hood:

```
import spacy

nlp = spacy.load("en_core_web_md")

>>> type(nlp)
spacy.lang.en.English
```

11805.py hosted with ❤ by GitHub view raw

Language objects are pre-trained on millions of text instances and labels and loaded into spaCy with their binary weights. These weights allow you to perform various tasks on new datasets without worrying about the hairy details.

As I mentioned earlier, spaCy has fully-trained pipelines for 22 languages, some of which you can see below:

```
nlp = spacy.load("es_core_news_sm")  # Spanish
nlp = spacy.load("ru_core_news_sm")  # Russian
nlp = spacy.load("zh_core_web_sm")  # Chinese
nlp = spacy.load("de_core_news_sm")  # German
```

11806.py hosted with ❤ by GitHub view raw

For other +40 languages, spaCy only offers basic tokenization rules, and other functionality is being slowly integrated with community effort.

It is also possible to load the language models directly from the `lang` sub-module:

```
from spacy.lang.en import English
from spacy.lang.es import Spanish

nlp = English()

>>> type(nlp)
spacy.lang.en.English

txt = """The original name for the search engine Google was Backrub.
         It was renamed Google after the googol,
         which is the number one followed by 100 zeros."""

doc = nlp(txt)
```

11807.py hosted with ❤ by GitHub view raw

After processing a text, words and punctuation are stored in the vocabulary object of `nlp`:

```
>>> type(nlp.vocab)
spacy.vocab.Vocab
```

This `Vocab` is shared between documents, meaning it stores all new words from all docs. In contrast, the `doc` object's vocabulary only contains the words from the `txt`:

```
>>> type(doc.vocab)
spacy.vocab.Vocab
```

Internally, spaCy communicates in hashes to save memory and has a two-way lookup table called `StringStore`. You can get the hash of a string or get the string if you have the hash:

```
>>> type(nlp.vocab.strings)
spacy.strings.StringStore
>>> nlp.vocab.strings["google"]
19886227373981203 58
>>> nlp.vocab.strings[19886227373981203 58]
'google'
```

When tokens go into the `Vocab`, they lose all their context-specific information. So, when you look up words from the vocab, you are looking up `lexeme`s:

```
>>> lexeme = nlp.vocab["google"]
>>> type(lexeme)
spacy.lexeme.Lexeme
```

Lexemes don't contain context-specific information like part-of-speech tags, morphological dependencies, etc. But they still offer many lexical attributes of the word:

```
>>> print(lexeme.text, lexeme.orth, lexeme.is_digit)
google 19886227373981203 58 False
```

> `orth` *attribute is for the hash of the lexeme.*

So, if you are looking at a word through the `doc` object, it is a token. If it is from a `Vocab`, it is a lexeme.

Now, let's talk more about the `doc` object.

```
txt = """Mosquitoes are the deadliest animal in the world:
         They kill more people than any other creature,
         due to the diseases they carry."""

doc = nlp(txt)

>>> type(doc)
spacy.tokens.doc.Doc
```

11808.py hosted with ❤ by GitHub view raw

Calling `nlp` object on text generates the doc along with its special attributes.

```
>>> doc.text
'Mosquitoes are the deadliest animal in the world: \n'
'         They kill more people than any other creature, '
'\n         due to the diseases they carry.'

>>> len(doc)
27
```

11809.py hosted with ❤ by GitHub view raw

You can create docs manually by importing the `Doc` class from `tokens` module:

```python
from spacy.lang.en import English
from spacy.tokens import Doc

nlp = English()

words = ["I", "love", "Barcelona", "!"]
spaces = [True, True, False, False]

# Create the doc object manually
doc = Doc(nlp.vocab, words=words, spaces=spaces)
```

11810.py hosted with ❤ by GitHub view raw

`Doc` requires three arguments - the vocabulary from `nlp`, a list of words and another list specifying if the words are followed by a space (including the last one). All tokes in `doc` have this information.

```
>>> len(doc)
4
>>> doc.text
'I love Barcelona!'
```

Spans are also a class of their own and expose a range of attributes, even though they are just a view of the `doc` object:

```
txt = """The hardest working muscle in your body is your heart:
         It pumps more than 2,000 gallons of blood a day
         and beats more than 2.5 billion times in a 70-year life span."""

doc = nlp(txt)

span = doc[:10]

>>> type(span)
spacy.tokens.span.Span

>>> print(span.text)
The hardest working muscle in your body is your heart

>>> print(span.start, span.end)
0 10
```

11811.py hosted with ❤ by GitHub view raw

To create the span objects manually, pass the doc object and start/end indices of the tokens to the `Span` class:

```
from spacy.tokens import Span

span = Span(doc, 0, 10)

>>> span.text
'The hardest working muscle in your body is your heart'
```

11812.py hosted with ❤ by GitHub view raw

Named Entity Recognition (NER)

One of the most common tasks in NLP is predicting named entities, like people, locations, countries, brands, etc.

Performing NER is ridiculously easy in spaCy. After processing a text, just extract the `ents` attribute of the `doc` object:

```
txt = """Cleopatra wasn't actually Egyptian!
         As far as historians can tell, Egypt's
         famous femme fatal was actually Greek!.
         She was a descendant of Alexander the Great's
         Macedonian general Ptolemy"""

nlp = spacy.load("en_core_web_md")

doc = nlp(txt)

for ent in doc.ents:
    print(f"{ent.text:<20}{ent.label_:<20}")

[OUT]:

Cleopatra           PERSON
Egyptian            NORP
Egypt               GPE
Greek               NORP
Macedonian          NORP
Ptolemy             PERSON
```

11813.py hosted with ❤ by GitHub view raw

Cleopatra is recognized as a PERSON, while Egypt is a Geo-political entity (GPE). To know the meaning of other labels, you can use the `explain` function:

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

>>> spacy.explain("NORP")
'Nationalities or religious or political groups'
```

11814.py hosted with ❤ by GitHub view raw

Instead of printing text, you can use spaCy's visual entity tagger, available via `displacy`:

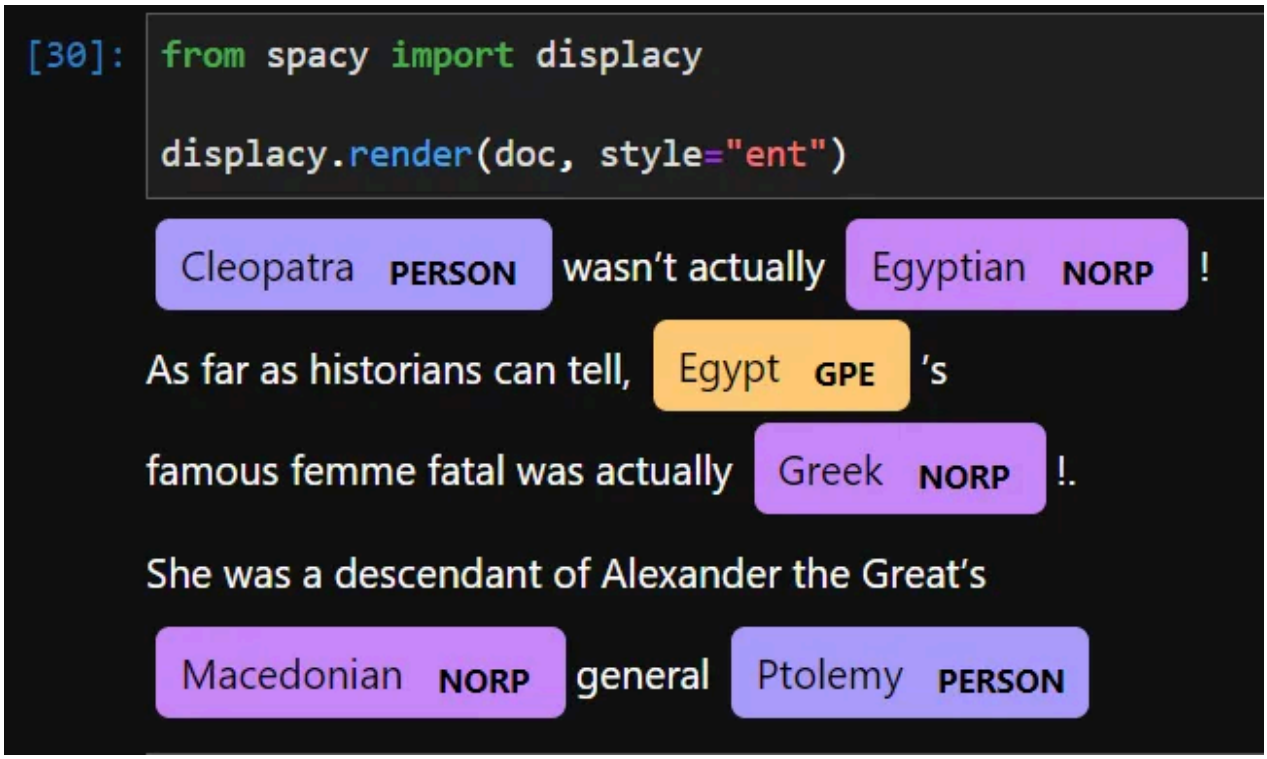


Image by author

The image shows that Alexander the Great isn't recognized as a PERSON because it is not a common name. But no matter, we can label Alexander as a PERSON manually.

First, extract the full name as a span by giving it a label (PERSON):

```
1  from spacy.tokens import Span
2
3  alexander = Span(doc, 31, 34, label="PERSON")
4  alexander
5  Alexander the Great
```

11815.py hosted with ❤ by GitHub view raw

Then, update the `ents` list with the span:

```
doc.ents = list(doc.ents) + [alexander]
```

Now, `displacy` tags it as well:

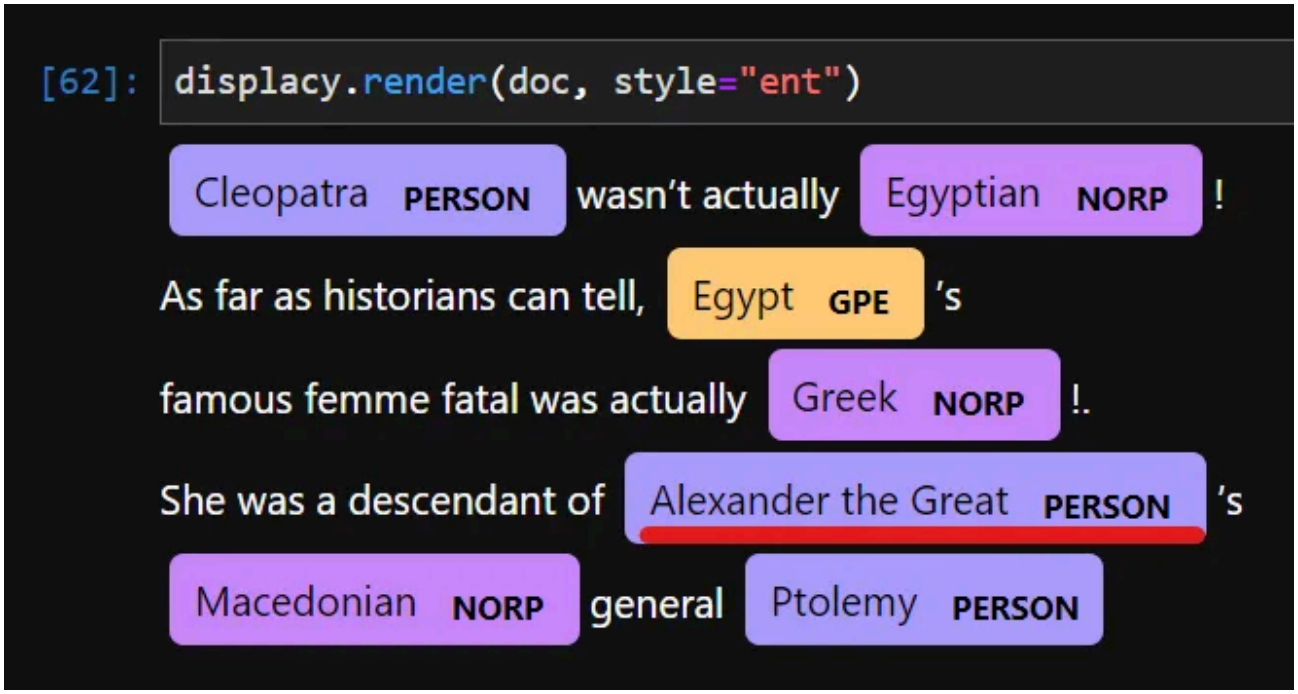


Image by author

You could've set the new entity with `set_ents` function as well:

```
# Leaves the rest of ents untouched
doc.set_ents([alexander], default="unmodified")
```

Predicting part-of-speech (POS) tags and syntactic dependencies

spaCy also offers a rich selection of tools for grammar analysis of a document. Lexical and grammatical attributes of tokens are given as attributes.

For example, let's take a look at each token's part-of-speech tag and its syntactic dependency:

```
txt = "The first footprints on the moon will remain there for a million years"

doc = nlp(txt)

print(
    f"{'Text':<20} {'Part-of-speech':<20} "
    f"{'Dependency':<20} {'Dependency text':<20}\n"
)
for token in doc:
    print(f"{token.text:<20} {token.pos_:<20} "
          f"{token.dep_:<20} {token.head.text:<20}")

[OUT]:

Text                Part-of-speech      Dependency          Dependency text

The                 DET                 det                 footprints
first               ADJ                 amod                footprints
footprints          NOUN                nsubj               remain
on                  ADP                 prep                footprints
the                 DET                 det                 moon
moon                NOUN                pobj                on
will                AUX                 aux                 remain
remain              VERB                ROOT                remain
there               ADV                 advmod              remain
for                 ADP                 prep                remain
a                   DET                 quantmod            million
million             NUM                 nummod              years
years               NOUN                pobj                for
```

11816.py hosted with ❤ by GitHub view raw

The output contains some confusing labels, but we can infer some of them like verbs, adverbs, adjectives, etc. Let's see the explanation for a few others:

```
pos_tags = ["DET", "AUX", "ADP"]
dep_tags = ["amod", "nsubj", "nummod"]

for pos in pos_tags:
    print(pos, "-->", spacy.explain(pos))

for dep in dep_tags:
    print(dep, "-->", spacy.explain(dep))

[OUT]:

DET --> determiner
AUX --> auxiliary
ADP --> adposition
amod --> adjectival modifier
nsubj --> nominal subject
nummod --> numeric modifier
```

11817.py hosted with ❤ by GitHub view raw

The last column in the previous table represents word relations like "the first", "first footprints", "remain there", etc.

spaCy contains many more powerful features for linguistic analysis. As the last example, here is how you extract noun chunks:

```
txt = """The teddy bear is named after President Theodore Roosevelt.
          After he refused to shoot a captured black bear on a hunt,
          a stuffed-animal maker decided to create
          a bear and name it after the president."""

doc = nlp(txt)

for chunk in doc.noun_chunks:
    print(chunk.text)

[OUT]:

The teddy bear
President Theodore Roosevelt
he
a captured black bear
a hunt
a stuffed-animal maker
a bear
it
the president
```

1818.py hosted with ❤ by GitHub view raw

Learn more about linguistic features from this page of the spaCy User Guide.

Custom rule-based tokenization

Until now, spaCy had complete control over tokenization rules. But a language can have many culture-specific idiosyncracies and edge-cases that don't fit spaCy's rules. For example, in an earlier example, we saw that "Alexander the Great" was missed as an entity.

```
txt = """Cleopatra wasn't actually Egyptian!
          As far as historians can tell, Egypt's
          famous femme fatal was actually Greek!.
          She was a descendant of Alexander the Great's
          Macedonian general Ptolemy"""
```

11819.py hosted with ❤ by GitHub view raw

If we process the same text again, spaCy regards the entity as three tokens. We need to tell it that titled names like Alexander the Great or Bran the Broken should be considered a single token rather than three because splitting them makes no sense.

Let's see how to do that by creating custom tokenization rules.

We will start by creating a pattern as a dictionary:

```
# Create a pattern
pattern = [
    {"IS_ALPHA": True, "IS_TITLE": True},
    {"IS_STOP": True},
    {"IS_ALPHA": True, "IS_TITLE": True},
]
```

11820.py hosted with ❤ by GitHub view raw

In spaCy, there is a large set of keywords you can use in combination to parse virtually any type of token pattern. For example, the above pattern is for a three-token pattern, with the first and last tokens being an alphanumeric text and the middle one being a stop word (like *the, and, or, etc.)*. In other words, we are matching "Alexander the Great" without explicitly telling it to spaCy.

Now, we will create a `Matcher` object with this pattern:

```
from spacy.matcher import Matcher

# Init the matcher with the shared vocab
matcher = Matcher(nlp.vocab)

# Add the pattern to the matcher
matcher.add("TITLED_PERSON", [pattern])
```

11822.py hosted with ❤ by GitHub view raw

After processing the text, we call this matcher object on the `doc` object, which returns a list of matches. Each match is a tuple with three elements - match ID, start, and end:

```
# Process the text
doc = nlp(txt)

# Find all matches
matches = matcher(doc)

# Iterate over matches
for match_id, start, end in matches:
    # Get the span
    span = doc[start:end]
    print(span.text)


[OUT]: Alexander the Great
```

11823.py hosted with ❤ by GitHub view raw

You can tweak the pattern in any you want. For example, you can use quantifiers like `OP` with reGex keywords or use reGex itself:

```
pattern = [
    {"TEXT": {"REGEX": "[a-zA-Z]+"}},
    {"IS_DIGIT": True, "OP": "?"},  # Match one or more times
    {"DEP": "quantmod"},  # Match based on dependency
    # etc.
]
```

11824.py hosted with ❤ by GitHub view raw

You can learn more about custom rule-based matching from here.

Word vectors and semantic similarity

Another everyday use case of NLP is predicting semantic similarity. Similarity scores can be used in recommender systems, plagiarism, duplicate content, etc.

spaCy calculates semantic similarity using word vectors (explained below), which are available in the medium-sized model:

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

doc1 = nlp("What a lukeworm sentiment.")
doc2 = nlp("What a short sentence.")

>>> doc1.similarity(doc2)
0.9200780919749721
```

11825.py hosted with ❤ by GitHub view raw

All `doc`, `token` and `span` objects have this `similarity` method:

```
token1 = doc1[-1]
token2 = doc2[2]

>>> token1.similarity(token2)
0.5222234129905701
```

11826.py hosted with ❤ by GitHub view raw

The three classes can be compared to each other as well, like a token to a span:

```
>>> doc1[0:2].similarity(doc[3])
0.8700238466262817
```

The similarity is calculated using word vectors, which are multi-dimensional mathematical representations of words. For example, here is the vector of the first token in the document:

```
array = doc1[0].vector

>>> array.shape
(300,)

>>> array[:10]
array([-0.73351 ,  0.41392 , -0.4425  , -0.29127 , -0.096179,  0.097562,
        0.13151 , -0.51825 ,  0.10671 ,  2.4144  ], dtype=float32)
```

11827.py hosted with ❤ by GitHub view raw

All about pipelines

Under the hood, language models aren't one pipeline but a collection:

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

>>> nlp.pipe_names
['tok2vec', 'tagger', 'parser', 'attribute_ruler', 'lemmatizer', 'ner']
```

11828.py hosted with ❤ by GitHub view raw

When a text is processed with `nlp`, it is first tokenized and passed down to each pipeline from the above list, which, in turn, modifies and returns the Doc object with new information. Here is how it is illustrated in the Spacy docs:

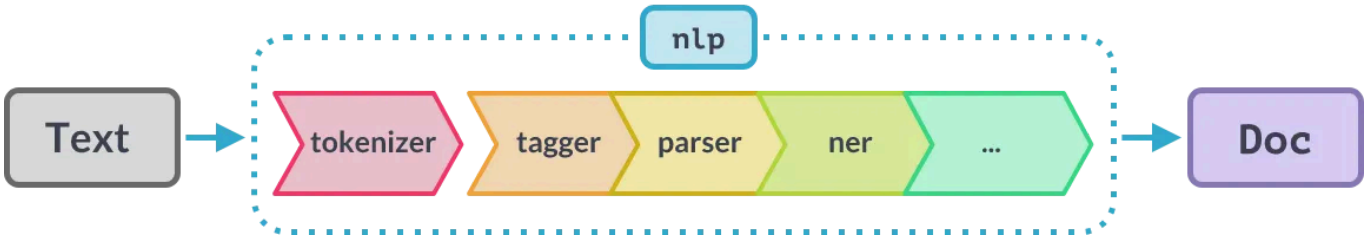


Image from the spaCy docs

But, spaCy doesn't have a custom pipeline for any NLP problem you might face in the real world. For example, you may want to perform your preprocessing steps before the text is passed to other pipelines or write callbacks to extract different types of information between pipelines.

For such cases, you should learn how to write custom pipeline functions and add them to the `nlp` object, so they are automatically run when you call `nlp` on text.

Here is a basic overview of how to do this:

```
from spacy.language import Language

@Language.component("your_component")
def your_component(doc):
    # Do something on the doc
    print(f"There are {len(doc)} tokens in this text.")

    return doc
```

11829.py hosted with ❤ by GitHub view raw

You need the general `Language` class and decorate its `component` method over your function. The custom function must accept and return a single argument for the doc object. Above, we are defining a simple pipeline that prints out the length of the doc object.

Let's add it to the `nlp` object:

```
>>> nlp.add_pipe("your_component")
<function __main__.your_component(doc)>

>>> nlp.pipe_names
['tok2vec',
 'tagger',
 'parser',
 'attribute_ruler',
 'lemmatizer',
 'ner',
 'your_component']
```

11820.py hosted with ❤ by GitHub view raw

As you can see, the custom pipeline is added to the end. Now, we will call `nlp` on a sample text:

```
>>> doc = nlp("Bird dies, but you remember the flight.")
```

```
There are 9 tokens in this text.
```

Working as expected.

Now, let's do a more serious example. We will go back to "Alexander the Great" example and write a pipeline that adds the conqueror's name as an entity. And we want all this happen automatically without finding the entity outside the pipeline.

Here is the complete code:

```
from spacy.language import Language
from spacy.matcher import Matcher
from spacy.util import filter_spans

nlp = spacy.load("en_core_web_sm")

@Language.component("titled_person")
def titled_person(doc):
    pattern = [
        {"IS_ALPHA": True, "IS_TITLE": True},
        {"IS_STOP": True},
        {"IS_ALPHA": True, "IS_TITLE": True},
    ]
    # Create the matcher
    matcher = Matcher(nlp.vocab)
    # Add the pattern
    matcher.add("TITLED_PERSON", [pattern])

    matches = matcher(doc)
    matched_spans = [Span(doc, start, end, label="PERSON") for _, start, end in matches]

    # Filter the entities for potential overlap
    filtered_matches = filter_spans(list(doc.ents) + matched_spans)
    # Add the matched spans to doc's entities
    doc.ents = filtered_matches

    return doc

>>> nlp.add_pipe("titled_person")
<function __main__.titled_person(doc)>
```

11821.py hosted with ❤ by GitHub view raw

We define the pattern, create a matcher object, add the matches to the entities, and return the doc. Let's test it:

```
txt = """Cleopatra wasn't actually Egyptian!
         As far as historians can tell, Egypt's
         famous femme fatal was actually Greek!.
         She was a descendant of Alexander the Great's
         Macedonian general Ptolemy"""

doc = nlp(txt)

>>> nlp.pipe_names
['tok2vec',
 'tagger',
 'parser',
 'attribute_ruler',
 'lemmatizer',
 'ner',
 'titled_person']

>>> doc.ents
(Egyptian, Egypt, Greek, Alexander the Great, Ptolemy)
```

11833.py hosted with ❤ by GitHub view raw

It is working as expected.

So far, we have been appending custom pipelines to the end, but we can control this behavior. The `add_pipe` function has arguments to specify precisely where you want to insert the function:

```
nlp.add_pipe("titled_person", first=True)  # Beginning
nlp.add_pipe("titled_person", after="parser")  # After parser
nlp.add_pipe("titled_person", before="tagger") # Before POS tagger
```

11834.py hosted with ❤ by GitHub view raw

Conclusion

Today, you have taken bold steps towards mastering the Scikit-learn of natural language processing. Armed with the article's knowledge, you can now roam freely across spaCy's user guide, which is just as large and information-rich as Scikit-learn's.

Thank you for reading!

Loved this article and, let's face it, its bizarre writing style? Imagine having access to dozens more just like it, all written by a brilliant, charming, witty author (that's me, by the way :).

For only 4.99$ membership, you will get access to not just my stories, but a treasure trove of knowledge from the best and brightest minds on Medium. And if you use my referral link, you will earn my supernova of gratitude and a virtual high-five for supporting my work.

Join Medium with my referral link — Bex T.

Get exclusive access to all my ⚡premium⚡ content and all over Medium without limits. Support my work by buying me a...

ibexorigin.medium.com

Artificial Intelligence | Machine Learning | Data Science | Programming

Editors Pick

Written by Bex T.

41K Followers · Writer for Towards Data Science

Follow

BEXGBoost | DataCamp Instructor | 🏅Top 10 AI/ML Writer on Medium | Kaggle Master | https://www.linkedin.com/in/bextuychiev/

More from Bex T. and Towards Data Science

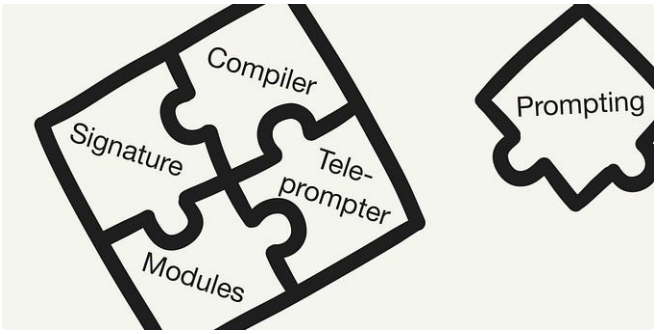
Bex T. in Towards AI

6 Pandas Mistakes That Silently Tell You Are a Rookie

No error messages—that's what makes them subtle

✦ · 7 min read · Sep 4, 2023

2.1K 18



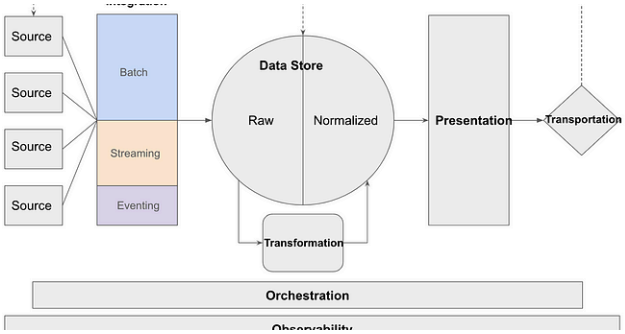
Leonie Monigatti in Towards Data Science

Intro to DSPy: Goodbye Prompting, Hello Programming!

How the DSPy framework solves the fragility problem in LLM-based applications by...

✦ · 13 min read · Feb 27, 2024

3.4K 10



Dave Melillo in Towards Data Science

Building a Data Platform in 2024

How to build a modern, scalable data platform to power your analytics and data science...

Bex T. in Towards AI

How to Boost Pandas Speed And Process 10M-row Datasets in...

Use Pandas the way it was intended to

9 min read · Feb 5, 2024

2.4K 33

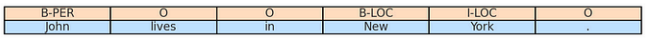
✨ · 6 min read · Sep 7, 2023

1.1K 5

See all from Bex T.

See all from Towards Data Science

Recommended from Medium



Ahmet Münir Kocaman

Mastering Named Entity Recognition with BERT: A...

Introduction

11 min read · Oct 5, 2023

18

Sebastian

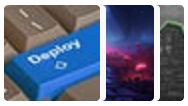
NLP with Spacy: Custom Text Classification Pipeline

Spacy is a powerful NLP library in which many NLP tasks like tokenization, stemming, part-...

10 min read · Dec 3, 2023

126

Lists



Predictive Modeling w/ Python

20 stories · 1013 saves

Natural Language Processing

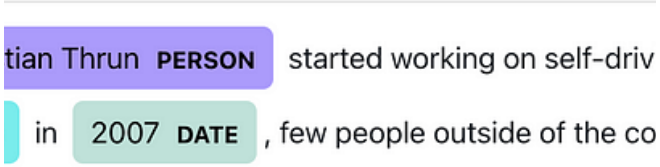
1305 stories · 793 saves

Practical Guides to Machine Learning

10 stories · 1219 saves

ChatGPT prompts

47 stories · 1291 saves



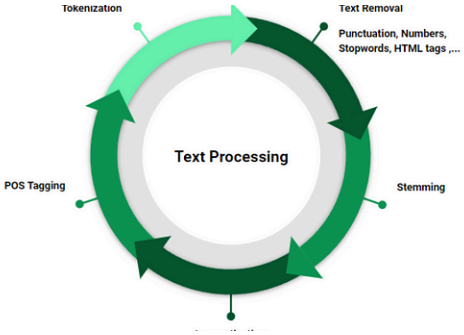
Isidoro Grisanti

Named Entity Recognition with LLMs—Extract Conversation...

This article aims to provide a comprehensive overview how to solve NER tasks based on...

5 min read · Oct 20, 2023

47



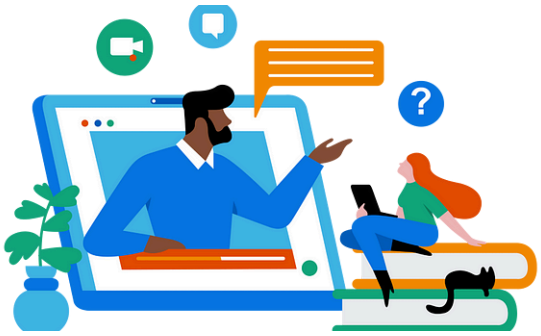
Aysel Aydin

1—Text Preprocessing Techniques for NLP

In this article, we will cover the following topics:

4 min read · Oct 4, 2023

259 2



Johni Douglas Marangon

Yuan An, PhD

Building a custom Named Entity Recognition model using spaCy ...

In today's post, we will learn how to train a NER. In the previous post we saw the...

4 min read · Nov 21, 2023

60 1

Use bert-base-NER in Hugging Face for Named Entity Recognition

Named Entity Recognition (NER) is a subtask of information extraction that classifies...

✦ · 3 min read · Sep 23, 2023

17

See more recommendations