Introduction to StanfordNLP: An NLP Library for 53 Languages (with Python code)

Mohd Sanad Zaki Rizvi Follow
Feb 3 · 8 min read

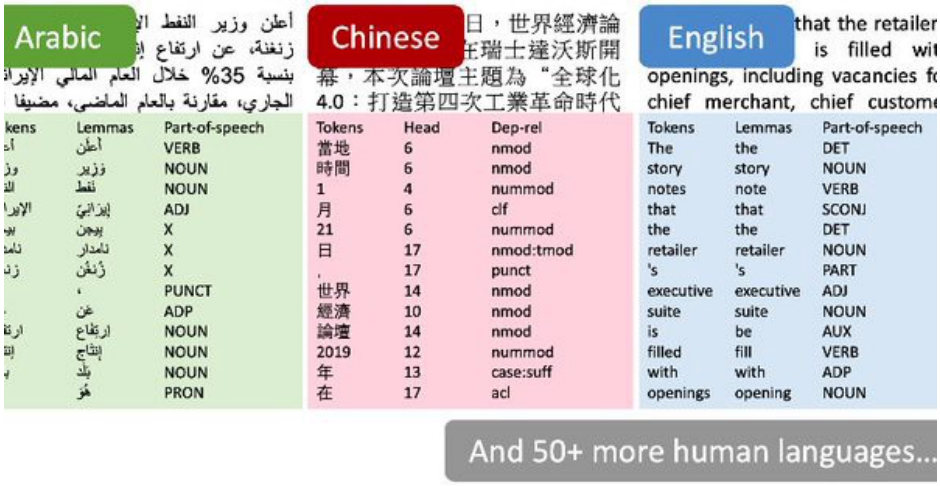
A tutorial on Stanford's latest library—StanfordNLP. I showcase an implementation on basic NLP tasks in Python + an awesome case study!

A common challenge I came across while learning Natural Language Processing (NLP)—can we build models for non-English languages? The answer has been no for quite a long time. Each language has its own grammatical patterns and linguistic nuances. And there just aren't many datasets available in other languages.

That's where Stanford's latest NLP library steps in—StanfordNLP.

I could barely contain my excitement when I read the news last week. The authors claimed StanfordNLP could support more than 53 human languages! Yes, I had to double-check that number.

I decided to check it out myself. There's no official tutorial for the library yet so I got the chance to experiment and play around with it. And I found that it opens up a world of endless possibilities. StanfordNLP contains pre-trained models for rare Asian languages like Hindi, Chinese and Japanese in their original scripts.



What is StanfordNLP and Why Should You Use it?

StanfordNLP is a collection of pre-trained state-of-the-art models. These models were used by the researchers in the CoNLL 2017 and 2018 competitions. All the models are built on PyTorch and can be trained and evaluated on your own annotated data. Awesome!

Additionally, **StanfordNLP also contains an official wrapper to the popular behemoth NLP library—CoreNLP**. This had been somewhat limited to the Java ecosystem until now. You should check out this tutorial to learn more about CoreNLP and how it works in Python.

What more could an NLP enthusiast ask for? Now that we have a handle on what this library does, let's take it for a spin in Python!

Setting up StanfordNLP in Python

There are some peculiar things about the library that had me puzzled initially. For instance, you need **Python 3.6.8/3.7.2** or later to use StanfordNLP. To be safe, I set up a separate environment in Anaconda for **Python 3.7.1**. Here's how you can do it:

1. Open conda prompt and type this:

```
conda create -n stanfordnlp python=3.7.1
```

2. Now activate the environment:

```
source activate stanfordnlp
```

3. Install the StanfordNLP library:

```
pip install stanfordnlp
```

4. We need to download a language's specific model to work with it. Launch a python shell and import StanfordNLP:

```
import stanfordnlp
```

then download the language model for English ("en"):

```
stanfordnlp.download('en')
```

This can take a while depending on your internet connection. These language models are pretty huge (the English one is 1.96GB).

A couple of important notes

- **StanfordNLP is built on top of PyTorch 1.0.0.** It might crash if you have an older version. Here's how you can check the version installed on your machine:

```
pip freeze | grep torch
```

which should give an output like `torch==1.0.0`

- I tried using the library without GPU on my Lenovo Thinkpad E470 (8GB RAM, Intel Graphics). I got a memory error in Python pretty quickly. Hence, I switched to a GPU enabled machine and would advise you to do the same as well. You can try Google Colab which comes with free GPU support

That's all! Let's dive into some basic NLP processing right away.

Using StanfordNLP to Perform Basic NLP Tasks

Let's start by creating a text pipeline:

```
nlp = stanfordnlp.Pipeline(processors =
"tokenize,mwt,lemma,pos")


doc = nlp("""The prospects for Britain's orderly withdrawal
from the European Union on March 29 have receded further,
even as MPs rallied to stop a no-deal scenario. An
amendment to the draft bill on the termination of London's
membership of the bloc obliges Prime Minister Theresa May
to renegotiate her withdrawal agreement with Brussels. A
Tory backbencher's proposal calls on the government to come
up with alternatives to the Irish backstop, a central tenet
of the deal Britain agreed with the rest of the EU.""")
```

The **processors = “”** argument is used to specify the task. All five processors are taken by default if no argument is passed. Here is a quick overview of the processors and what they can do:

NAME	ANNOTATOR CLASS NAME	GENERATED ANNOTATION	DESCRIPTION
tokenize	TokenizePro-cessor	Segments a `Document` into `Sentence` s, each containing a list of `Token` s. This processor also predicts which tokens are multi-word tokens, but leaves expanding them to the MWT expander.	Tokenizes the text and performs sentence segmentation.
mwt	MWTProcessor	Expands multi-word tokens into multiple words when they are predicted by the tok-enizer.	Expands multi-word tokens (MWT) predicted by the tok-enizer.
lemma	LemmaProces-sor	Perform lemmatization on a `Word` using the `Word.text` and `Word.upos` value. The result can be accessed in `Word.lemma` .	Generates the word lemmas for all tokens in the corpus.
pos	POSProcessor	UPOS, XPOS, and UFeats annotations accessible through `Word` 's properties `pos` , `xpos` , and `ufeats` .	Labels tokens with their universal POS (UPOS) tags, treebank-specific POS (XPOS) tags, and universal morphological features (UFeats).
depparse	DepparsePro-cessor	Determines the syntactic head of each word in a sentence and the dependency relation between the two words that are accessible through `Word` 's `governor` and `dependency_relation` attributes.	Provides an accurate syntactic dependency parser.

Let’s see each of them in action.

Tokenization

This process happens implicitly once the Token processor is run. It is actually pretty quick. You can have a look at tokens by using

print_tokens():

```
doc.sentences[0].print_tokens()
```

```
<Token index=1;words=[<Word index=1;text=The;lemma=the;upos=DET;xpos=DT;feats=Definite=Def|PronType=Art>]>
<Token index=2;words=[<Word index=2;text=prospects;lemma=prospect;upos=NOUN;xpos=NNS;feats=Number=Plur>]>
<Token index=3;words=[<Word index=3;text=for;lemma=for;upos=ADP;xpos=IN;feats=_>]>
<Token index=4;words=[<Word index=4;text=Britain;lemma=Britain;upos=PROPN;xpos=NNP;feats=Number=Sing>]>
<Token index=5;words=[<Word index=5;text='s;lemma='s;upos=PART;xpos=POS;feats=_>]>
<Token index=6;words=[<Word index=6;text=orderly;lemma=orderly;upos=ADJ;xpos=JJ;feats=Degree=Pos>]>
<Token index=7;words=[<Word index=7;text=withdrawal;lemma=withdrawal;upos=NOUN;xpos=NN;feats=Number=Sing>]>
<Token index=8;words=[<Word index=8;text=from;lemma=from;upos=ADP;xpos=IN;feats=_>]>
<Token index=9;words=[<Word index=9;text=the;lemma=the;upos=DET;xpos=DT;feats=Definite=Def|PronType=Art>]>
<Token index=10;words=[<Word index=10;text=European;lemma=european;upos=PROPN;xpos=NNP;feats=Number=Sing>]>
<Token index=11;words=[<Word index=11;text=Union;lemma=union;upos=PROPN;xpos=NNP;feats=Number=Sing>]>
<Token index=12;words=[<Word index=12;text=on;lemma=on;upos=ADP;xpos=IN;feats=_>]>
<Token index=13;words=[<Word index=13;text=March;lemma=March;upos=PROPN;xpos=NNP;feats=Number=Sing>]>
<Token index=14;words=[<Word index=14;text=29;lemma=29;upos=NUM;xpos=CD;feats=NumType=Card>]>
```

The token object contains the index of the token in the sentence and a list of word objects (in case of a multi-word token). **Each word object contains useful information, like the index of the word, the lemma of the text, the pos (parts of speech) tag and the feat (morphological features) tag.**

Lemmatization

This involves using the "lemma" property of the words generated by the lemma processor. Here's the code to get the lemma of all the words:

```
import pandas as pd

def extract_lemma(doc):
    parsed_text = {'word':[], 'lemma':[]}
    for sent in doc.sentences:
        for wrd in sent.words:
            #extract text and lemma
            parsed_text['word'].append(wrd.text)
```

This returns a *pandas* data frame for each word and its respective lemma:

	word	lemma
0	The	the
1	prospects	prospect
2	for	for
3	Britain	Britain
4	's	's
5	orderly	orderly
6	withdrawal	withdrawal
7	from	from
8	the	the
9	European	european
10	Union	union
11	on	on
12	March	March
13	29	29
14	have	have
15	receded	recede

Parts of Speech (PoS) Tagging

The PoS tagger is quite fast and works really well across languages. Just like lemmas, PoS tags are also easy to extract:

```
#dictionary to hold pos tags and their explanations
pos_dict = {
'CC': 'coordinating conjunction',
'CD': 'cardinal digit',
'DT': 'determiner',
'EX': 'existential there (like: \"there is\" ... think
'FW': 'foreign word',
'IN':  'preposition/subordinating conjunction',
'JJ': 'adjective \'big\'',
'JJR': 'adjective, comparative \'bigger\'',
'JJS': 'adjective, superlative \'biggest\'',
'LS': 'list marker 1)',
'MD': 'modal could, will',
'NN': 'noun, singular \'desk\'',
'NNS': 'noun plural \'desks\'',
'NNP': 'proper noun, singular \'Harrison\'',
'NNPS': 'proper noun, plural \'Americans\'',
'PDT': 'predeterminer \'all the kids\'',
'POS': 'possessive ending parent\'s',
'PRP': 'personal pronoun I, he, she',
'PRP$': 'possessive pronoun my, his, hers',
'RB': 'adverb very, silently,',
'RBR': 'adverb, comparative better',
'RBS': 'adverb, superlative best',
'RP': 'particle give up',
'TO': 'to go \'to\' the store.',
'UH': 'interjection errrrrrrrm',
'VB': 'verb, base form take',
'VBD': 'verb, past tense took',
'VBG': 'verb, gerund/present participle taking',
'VBN': 'verb, past participle taken',
'VBP': 'verb, sing. present, non-3d take',
'VBZ': 'verb, 3rd person sing. present takes',
'WDT': 'wh-determiner which',
'WP': 'wh-pronoun who, what',
'WP$': 'possessive wh-pronoun whose',
'WRB': 'wh-abverb where, when'
```

Notice the big dictionary in the above code? It is just a mapping between PoS tags and their meaning. This helps in getting a better understanding of our document's syntactic structure.

The output would be a data frame with three columns—word, pos and exp (explanation). The explanation column gives us the most information about the text (and is hence quite useful).

word	pos	exp
the	DT	determiner
agreed	VBD	verb, past tense took
could	MD	modal could, will
between	IN	preposition/subordinating conjunction
the	DT	determiner
reservations	NNS	noun plural 'desks'
Tuesday	NNP	proper noun, singular 'Harrison'
guarantee	VB	verb, base form take
the	DT	determiner
the	DT	determiner
May	NNP	proper noun, singular 'Harrison'
for	IN	preposition/subordinating conjunction
Ireland	NNP	proper noun, singular 'Harrison'
backstop	NN	noun, singular 'desk'
's	POS	possessive ending parent's
the	DT	determiner
.	.	NA
bloc	NNP	proper noun, singular 'Harrison'

Adding the explanation column makes it much easier to evaluate how accurate our processor is. I like the fact that the tagger is on point for the majority of the words. It even picks up the tense of a word and whether it is in base or plural form.

Dependency Extraction

Dependency extraction is another out-of-the-box feature of StanfordNLP. You can simply call **print_dependencies()** on a sentence to get the dependency relations for all of its words:

```
doc.sentences[0].print_dependencies()
```

```
('The', '2', 'det')
('prospects', '16', 'nsubj')
('for', '7', 'case')
('Britain', '7', 'nmod:poss')
('’s', '4', 'case')
('orderly', '7', 'amod')
('withdrawal', '2', 'nmod')
('from', '11', 'case')
('the', '11', 'det')
('European', '11', 'compound')
('Union', '7', 'nmod')
('on', '13', 'case')
('March', '11', 'nmod')
('29', '13', 'nummod')
('have', '16', 'aux')
('receded', '0', 'root')
('further', '16', 'advmod')
(',', '16', 'punct')
('even', '22', 'advmod')
('as', '22', 'mark')
('MPs', '22', 'nsubj')
('rallied', '16', 'advcl')
('to', '24', 'mark')
('stop', '22', 'xcomp')
('a', '27', 'det')
('no-deal', '27', 'amod')
('scenario', '24', 'obj')
('.', '16', 'punct')
```

The library computes all of the above during a single run of the pipeline. This will hardly take you a few minutes on a GPU enabled machine.

We have now figured out a way to perform basic text processing with StanfordNLP. It’s time to take advantage of the fact that we can do the same for 51 other languages!

Implementing StanfordNLP on the Hindi Language

StanfordNLP really stands out in its performance and multilingual text parsing support. Let's dive deeper into the latter aspect.

Processing text in Hindi (Devanagari Script)

First, we have to download the Hindi language model (comparatively smaller!):

```
stanfordnlp.download('hi')
```

Now, take a piece of text in Hindi as our text document:

```
hindi_doc = nlp("""केंद्र की मोदी सरकार ने शुक्रवार को अपना अंतरिम बजट पेश किया. कार्यवाहक वित्त मंत्री पीयूष गोयल ने अपने बजट में किसान, मजदूर, करदाता, महिला वर्ग समेत हर किसी के लिए बंपर ऐलान किए. हालांकि, बजट के बाद भी टैक्स को लेकर काफी कन्फ्यूजन बना रहा. केंद्र सरकार के इस अंतरिम बजट क्या खास रहा और किसको क्या मिला, आसान भाषा में यहां समझें""")
```

This should be enough to generate all the tags. Let's check the tags for Hindi:

```
extract_pos(hindi_doc)
```

word	pos	exp
केंद्र	NNP	proper noun, singular 'Harrison'
की	PSP	postposition, common in indian langs
मोदी	NNP	proper noun, singular 'Harrison'
सरकार	NN	noun, singular 'desk'
ने	PSP	postposition, common in indian langs
शुक्रवार	NNP	proper noun, singular 'Harrison'
को	PSP	postposition, common in indian langs
अपना	PRP	personal pronoun I, he, she
अंतरिम	JJ	adjective 'big'
बजट	NN	noun, singular 'desk'
पेश	JJ	adjective 'big'
किया	VM	main verb

The PoS tagger works surprisingly well on the Hindi text as well. Look at "अपना" for example. The PoS tagger tags it as a pronoun—I, he, she—which is accurate.

Using CoreNLP's API for Text Analytics

CoreNLP is a time tested, industry grade NLP tool-kit that is known for its performance and accuracy. StanfordNLP takes three lines of code to start utilizing CoreNLP's sophisticated API. Literally, just three lines of code to set it up!

1. Download the CoreNLP package. Open your Linux terminal and type the following command:

```
wget http://nlp.stanford.edu/software/stanford-corenlp-
full-2018-10-05.zip
```

2. Unzip the downloaded package:

```
unzip stanford-corenlp-full-2018-10-05.zip
```

3. Start the CoreNLP server:

```
java -mx4g -cp "*"
edu.stanford.nlp.pipeline.StanfordCoreNLPServer -port 9000
-timeout 15000
```

Note: *CoreNLP requires Java8 to run. Please make sure you have JDK and JRE 1.8.x installed.p*

Now, make sure that StanfordNLP knows where CoreNLP is present. For that, you have to export $CORENLP_HOME as the location of your folder. In my case, this folder was in **the home** itself so my path would be like

```
export CORENLP_HOME=stanford-corenlp-full-2018-10-05/
```

After the above steps have been taken, you can start up the server and make requests in Python code. Below is a comprehensive example of starting a server, making requests, and accessing data from the returned object.

a. Setting up the CoreNLPClient

```
from stanfordnlp.server import CoreNLPClient
# example text
print('---')
print('input text')
print('')
text = "Chris Manning is a nice person. Chris wrote a
print(text)
# set up the client
print('---')
print('starting up Java Stanford CoreNLP Server...')
# set up the client
```

b. Dependency Parsing and POS

```
    #get the dependency parse of the first sentence
    print('---')
    print('dependency parse of first sentence')
    dependency_parse = sentence.basicDependencies
    print(dependency_parse)
    # get the first token of the first sentence
    print('---')
    print('first token of first sentence')
    token = sentence.token[0]
    print(token)
```

c. Named Entity Recognition and Co-Reference Chains

```
# get the named entity tag
    print('---')
    print('named entity tag of token')
    print(token.ner)
    # get an entity mention from the first sentence
    print('---')
    print('first entity mention in sentence')
    print(sentence.mentions[0])
    # access the coref chain
    print('---')
    print('coref chains for the example')
    print(ann.corefChain)
    # Use tokensregex patterns to find who wrote a sen
    pattern = '([ner: PERSON]+) /wrote/ /an?/ []{0,3}
    matches = client.tokensregex(text, pattern)
    # sentences contains a list with matches for each
    assert len(matches["sentences"]) == 3
    # length tells you whether or not there are any ma
    assert matches["sentences"][1]["length"] == 1
    # You can access matches like most regex groups.
    matches["sentences"][1]["0"]["text"] == "Chris wro
    matches["sentences"][1]["0"]["1"]["text"] == "Chri
    # Use semgrex patterns to directly find who wrote
    pattern = '{word:wrote} >nsubj {}=subject >dobj {}
    matches = client.semgrex(text, pattern)
    # sentences contains a list with matches for each
    assert len(matches["sentences"]) == 3
    # length tells you whether or not there are any ma
    assert matches["sentences"][1]["length"] == 1
    # You can access matches like most regex groups.
    matches["sentences"][1]["0"]["text"] == "wrote"
    matches["sentences"][1]["0"]["$subject"]["text"] =
    matches["sentences"][1]["0"]["$object"]["text"] ==
```

corenlp_ner.py hosted with ❤ by GitHub view raw

What I like the most here is the ease of use and increased accessibility this brings when it comes to using CoreNLP in python.

My Thoughts on using StanfordNLP — Pros and Cons

A few things that excite me regarding the future of StanfordNLP:

1. Its out-of-the-box support for multiple languages
2. The fact that it is going to be an official Python interface for CoreNLP. This means it will only improve in functionality and ease of use going forward
3. It is fairly fast (barring the huge memory footprint)
4. Straightforward set up in Python

There are, however, a few chinks to iron out. Below are my thoughts on where StanfordNLP could improve:

1. The size of the language models is too large (English is 1.9 GB, Chinese ~ 1.8 GB)
2. The library requires a lot of code to churn out features. Compare that to NLTK where you can quickly script a prototype—this might not be possible for StanfordNLP
3. Currently missing visualization features. It is useful to have for functions like dependency parsing. StanfordNLP falls short here when compared with libraries like SpaCy

Make sure you check out StanfordNLP's official documentation.

End Notes

Clearly, StanfordNLP is very much in the beta stage. It will only get better from here so this is a really good time to start using it—get a head start over everyone else.

For now, the fact that such amazing toolkits (CoreNLP) are coming to the Python ecosystem and research giants like Stanford are making an effort to open source their software, I am optimistic about the future.

. . .

Originally published at www.analyticsvidhya.com on February 3, 2019.