Natural Language Processing - spaCy

Raul-Mihai Acu

May 31, 2018

Contents

1 Overview

1.1 Running instructions

You can install spaCy available as source package using:
Pip

```
>pip install -U spacy
```

Conda

```
>conda install -c conda-forge spacy
```

Or from source

```
>git clone https://github.com/explosion/spaCy
>cd spaCy
>export PYTHONPATH=‘pwd‘
>pip install -r requirements.txt
>python setup.py build_ext --inplace
```

1.2 Theoretical aspects

spaCy is a free, open-source library for advanced Natural Language Processing (NLP) in Python.

Natural-language processing (NLP) is an area of computer science and artificial intelligence concerned with the interactions between computers and human (natural) languages, in particular how to program computers to fruitfully process large amounts of natural language data.

If you're working with a lot of text, you'll eventually want to know more about it. For example, what's it about? What do the words mean in context? Who is doing what to whom? What companies and products are mentioned? Which texts are similar to each other?

spaCy is designed specifically for production use and helps you build applications that process and "understand" large volumes of text. It can be used to build information extraction or natural language understanding systems, or to pre-process text for deep learning.

1.2.1 Data representation

The central data structures in spaCy are the Doc and the Vocab.

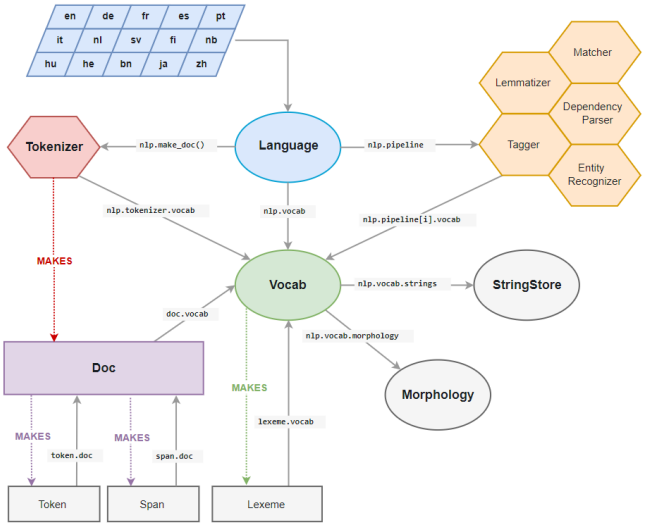
The Doc object owns the sequence of tokens and all their annotations.

The Vocab object owns a set of look-up tables that make common information available across documents.

By centralising strings, word vectors and lexical attributes, we avoid storing multiple copies of this data. This saves memory, and ensures there's a single source of truth.

Text annotations are also designed to allow a single source of truth: the Doc object owns the data, and Span and Token are views that point into it. The Doc object is constructed by the Tokenizer, and then modified in place by the components of the pipeline.

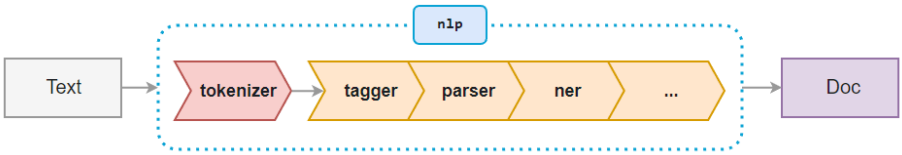
The Language object coordinates these components. It takes raw text and sends it through the pipeline, returning an annotated document. It also orchestrates training and serialization.



1.2.2 Algorithm

When you call nlp on a text, spaCy first tokenizes the text to produce a Doc object. The Doc is then processed in several different steps this is also referred to as the processing pipeline. The pipeline used by the default models consists of a tagger, a parser and an entity recognizer. Each pipeline component returns the processed Doc, which is then passed on to the next component.

The processing pipeline always depends on the statistical model and its capabilities. For example, a pipeline can only include an entity recognizer component if the model includes data to make predictions of entity labels. This is why each model will specify the pipeline to use in its meta data.



tokenizer - Segments text into tokens.
tagger - Assigns part-of-speech tags.
parser - Assigns dependency labels.
ner - Detects and labes named entities.
textcat - Assigns document labels.
... - Assign customs attributes, methods or properties.

1.3 Existing Example

A simple example of extracting relations between phrases and entities using spaCy's named entity recognizer and the dependency parse. Here, we extract money and currency values (entities labelled as MONEY) and then check the dependency tree to find the noun phrase they are referring to for example: "$9.4 million" "Net income".

Code:

```
from __future__ import unicode_literals, print_function
import plac
import spacy

TEXTS = ['Net income was $9.4 million compared to the prior year of $2.7 million.',
    'Revenue exceeded twelve billion dollars, with a loss of $1b.']

@plac.annotations(
    model=("Model to load (needs parser and NER)", "positional", None, str))
def main(model='en_core_web_sm'):
    nlp = spacy.load(model)
    print("Loaded model '%s'" % model)
    print("Processing %d texts" % len(TEXTS))

    for text in TEXTS:
        doc = nlp(text)
        relations = extract_currency_relations(doc)
```

```
        for r1, r2 in relations:
            print('{:<10}\t{}\t{}'.format(r1.text, r2.ent_type_, r2.text))

def extract_currency_relations(doc):
    # merge entities and noun chunks into one token
    spans = list(doc.ents) + list(doc.noun_chunks)
    for span in spans:
        span.merge()

    relations = []
    for money in filter(lambda w: w.ent_type_ == 'MONEY', doc):
        if money.dep_ in ('attr', 'dobj'):
            subject = [w for w in money.head.lefts if w.dep_ == 'nsubj']
            if subject:
                subject = subject[0]
                relations.append((subject, money))
        elif money.dep_ == 'pobj' and money.head.dep_ == 'prep':
            relations.append((money.head.head, money))
    return relations


if __name__ == '__main__':
    plac.call(main)
```

Output:

```
Loaded model 'en_core_web_sm'
Processing 2 texts
Net income      MONEY   $9.4 million
the prior year  MONEY   $2.7 million
Revenue         MONEY   twelve billion dollars
a loss          MONEY   1b
```

1.4 Small Personal Example

Code:

```
import spacy

nlp = spacy.load('en')
docs = nlp(u'This is some weird sentence. Raul, check it!')
docs2 = nlp(u'Apple is looking at buying U.K. startup for $1 billion')

for token in docs:
    print(token.text, " --> ", token.pos_)

for ent in docs2.ents:
    print(ent.text, ent.start_char, ent.end_char, ent.label_)
```

Output:

```
This  -->  DET
is  -->  VERB
some  -->  DET
weird  -->  ADJ
sentence  -->  NOUN
.  -->  PUNCT
Raul  -->  PROPN
,  -->  PUNCT
check  -->  VERB
it  -->  PRON
!  -->  PUNCT

Apple 0 5 ORG
U.K. 27 31 GPE
$1 billion 44 54 MONEY
```

2 Proposed problem

2.1 Specification

The target of this project is to identify the genre(s) of a movie, deduced from it‘s overview by using the spaCy library for Natural Language Processing in Python. This can be obtained by extracting linguistic features like part-of-speech tags, dependency labels and named entities, customising the tokenizer and working with the rule-based matcher. Processing raw text intelligently is difficult: most words are rare, and it's common for words that look completely different to mean almost the same thing. The same words in a different order can mean something completely different. We can avoid or test that using Inflectional morphology, 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 changes its part-of-speech. For identifying the genre of a movie, I will make use of spaCy‘s linguistic features, parsing, tokenization and similarity functiions in order to process raw text from a movie script and classifying the relevant parts of the plot and the involved entities. This can also be considered to be similar to sentiment analysis as well, caused by inferrence from some rule-based matches.

2.2 Documentation of your solution

First of all, I will create an empty model using spacy.blank, and adding the parser to the pipeline in order to train it using an existing dataset and excluding other pipeline stages, to train only the parser.

```
def main(model=None, output_dir=None, n_iter=100, n_texts=10):
    if model is not None:
        nlp = spacy.load(model)  # load existing spaCy model
        print("Loaded model '%s'" % model)
    else:
        nlp = spacy.blank('en')  # create blank Language class
        print("Created blank 'en' model")

    # add the text classifier to the pipeline if it doesn't exist
    # nlp.create_pipe works for built-ins that are registered with spaCy
    if 'textcat' not in nlp.pipe_names:
        textcat = nlp.create_pipe('textcat')
        nlp.add_pipe(textcat, last=True)
    # otherwise, get it, so we can add labels to it
    else:
        textcat = nlp.get_pipe('textcat')
```

After creating the model and selecting only the text categorizer feature, I will add the dependency labels to the parser.

```
# add label to text classifier
textcat.add_label('Horror')
textcat.add_label('Drama')
textcat.add_label('Comedy')
textcat.add_label('Romance')
textcat.add_label('Science')
textcat.add_label('Crime')
textcat.add_label('Documentary')
textcat.add_label('Film')
textcat.add_label('Animation')
```

After getting the model ready to train, all we need is a dataset, and a way of including it to the train data. For that, loading and pre-processing the dataset is required. We shuffle the data and split off a part of it to hold back for evaluation. This way, we'll be able to see results on each training iteration.

The next step of the training process is shuffle and loop over the examples. For each example,the model is updated, which steps through the words of the input. At each word, it makes a prediction. It then consults the annotations to see whether it was right. If it was wrong, it adjusts its weights so that the correct action will score higher next time. We Loop over the training examples and partition them into batches using spaCy's minibatch and compounding helpers. Update the model by , which steps through the examples and makes a prediction. It then consults the annotations to see whether it was right. If it was wrong, it adjusts its weights so that the correct prediction will score higher next time. Optionally, we can also evaluate the text classifier on each iteration, by checking how it performs on the development data held back from the dataset. This lets us print the precision, recall and F-score.

```
print("Loading movie data...")
(train_texts, train_cats), (dev_texts, dev_cats) = load_data(limit=15)
print("Using {} examples ({} training, {} evaluation)"
      .format(n_texts, len(train_texts), len(dev_texts)))
train_data = list(zip(train_texts,
                      [{'cats': cats} for cats in train_cats]))

# get names of other pipes to disable them during training
other_pipes = [pipe for pipe in nlp.pipe_names if pipe != 'textcat']
with nlp.disable_pipes(*other_pipes):  # only train textcat
    optimizer = nlp.begin_training()
    print("Training the model...")
    print('{:^5}\t{:^5}\t{:^5}\t{:^5}'.format('LOSS', 'P', 'R', 'F'))
    for i in range(n_iter):
        losses = {}
        # batch up the examples using spaCy's minibatch
        batches = minibatch(train_data, size=compounding(4., 32., 1.001))
        for batch in batches:
            texts, annotations = zip(*batch)
            nlp.update(texts, annotations, sgd=optimizer, drop=0.2,
                       losses=losses)
        with textcat.model.use_params(optimizer.averages):
            # evaluate on the dev data split off in load_data()
            scores = evaluate(nlp.tokenizer, textcat, dev_texts, dev_cats)
```

After sucessfully training the model, we will save the trained model, and as well as testing the model to make sure the text classifier works as expected.

```
# test the trained model
test_text = "Seth Gecko and his younger brother Richard are on the run after a bloody bank robbery in Texas." \
            " They escape across the border into Mexico and will be home-free the next morning," \
            " when they pay off the local kingpin. They just have to survive 'from dusk till dawn'" \
            " at the rendezvous point, which turns out to be a Hell of a strip joint.\n"
doc = nlp(test_text)
print(test_text, doc.cats)

if output_dir is not None:
    output_dir = Path(output_dir)
    if not output_dir.exists():
        output_dir.mkdir()
    nlp.to_disk(output_dir)
    print("Saved model to", output_dir)

    # test the saved model
    print("Loading from", output_dir)
    nlp2 = spacy.load(output_dir)
    doc2 = nlp2(test_text)
    print(test_text, doc2.cats)
```