GET STARTED
Installation
Models & Languages
Facts & Figures
spaCy 101
New in v2.0

GUIDES
Linguistic Features
Processing Pipelines
Vectors & Similarity
Training Models
Adding Languages
Visualizers

IN-DEPTH
Code Examples
Resources

Word Vectors and Semantic Similarity

TRAINING WORD VECTORS

Dense, real valued vectors representing distributional similarity information are now a cornerstone of practical NLP. The most common way to train these vectors is the word2vec family of algorithms. If you need to train a word2vec model, we recommend the implementation in the Python library Gensim.

spaCy is able to compare two objects, and make a prediction of **how similar they are**. Predicting similarity is useful for building recommendation systems or flagging duplicates. For example, you can suggest a user content that's similar to what they're currently looking at, or label a support ticket as a duplicate if it's very similar to an already existing one.

Each `Doc`, `Span` and `Token` comes with a `.similarity()` method that lets you compare it with another object, and determine the similarity. Of course similarity is always subjective – whether "dog" and "cat" are similar really depends on how you're looking at it. spaCy's similarity model usually assumes a pretty general-purpose definition of similarity.

```
tokens = nlp(u'dog cat banana')

for token1 in tokens:
    for token2 in tokens:
        print(token1.similarity(token2))
```

● **similarity:** identical
✓ **similarity:** similar (higher is more similar)
✗ **similarity:** dissimilar (lower is less similar)

SPACY CHAT

DOG	1.00 ●	0.80 ✓	0.24 ✗
CAT	0.80 ✓	1.00 ●	0.28 ✗
BANANA	0.24 ✗	0.28 ✗	1.00 ●

In this case, the model's predictions are pretty on point. A dog is very similar to a cat, whereas a banana is not very similar to either of them. Identical tokens are obviously 100% similar to each other (just not always exactly `1.0`, because of vector math and floating point imprecisions).

Similarity is determined by comparing **word vectors** or "word embeddings", multi-dimensional meaning representations of a word. Word vectors can be generated using an algorithm like word2vec and usually look like this:

BANANA.VECTOR

```
array([2.02280000e-01,  -7.66180009e-02,   3.70319992e-01,
       3.28450017e-02,  -4.19569999e-01,   7.20689967e-02,
      -3.74760002e-01,   5.74599989e-02,  -1.24009997e-02,
       5.29489994e-01,  -5.23800015e-01,  -1.97710007e-01,
      -3.41470003e-01,   5.33169985e-01,  -2.53309999e-02,
       1.73800007e-01,   1.67720005e-01,   8.39839995e-01,
       5.51070012e-02,   1.05470002e-01,   3.78719985e-01,
       2.42750004e-01,   1.47449998e-02,   5.59509993e-01,
       1.25210002e-01,  -6.75960004e-01,   3.58420014e-01,
```

⚠ IMPORTANT NOTE

To make them compact and fast, spaCy's small models (all packages that end in `sm`) **don't ship with word vectors**, and only include context-sensitive **tensors**. This means you can still use the `similarity()` methods to compare documents, spans and tokens – but the result won't be as good, and individual tokens won't have any vectors assigned. So in order to use *real* word vectors, you need to download a larger model:

```
$ python -m spacy download en_core_web_lg
```

Models that come with built-in word vectors make them available as the `Token.vector` attribute. `Doc.vector` and `Span.vector` will default to an average of their token vectors. You can also check if a token has a vector assigned, and get the L2 norm, which can be used to normalise vectors.

```
nlp = spacy.load('en_core_web_lg')
tokens = nlp(u'dog cat banana sasquatch')

for token in tokens:
    print(token.text, token.has_vector, token.vector_norm, token.is_oov)
```

Text: The original token text.
has vector: Does the token have a vector representation?

TEXT	HAS VECTOR	VECTOR NORM	OOV
dog	`True`	`7.033672992262838`	`False`
cat	`True`	`6.68081871208896`	`False`
banana	`True`	`6.700014292148571`	`False`
sasquatch	`False`	`0`	`True`

The words "dog", "cat" and "banana" are all pretty common in English, so they're part of the model's vocabulary, and come with a vector. The word "sasquatch" on the other hand is a lot less common and out-of-vocabulary – so its vector representation consists of 300 dimensions of `0`, which means it's practically nonexistent. If your application will benefit from a **large vocabulary** with more vectors, you should consider using one of the larger models or loading in a full vector package, for example, `en_vectors_web_lg`, which includes over **1 million unique vectors**.

Similarities in context

Aside from spaCy's built-in word vectors, which were trained on a lot of text with a wide vocabulary, the parsing, tagging and NER models also rely on vector representations of the **meanings of words in context**. As the processing pipeline is applied spaCy encodes a document's internal meaning representations as an array of floats, also called a tensor. This allows spaCy to make a reasonable guess at a word's meaning, based on its surrounding words. Even if a word hasn't been seen before, spaCy will know *something* about it. Because spaCy uses a 4-layer convolutional network, the tensors are sensitive to up to **four words on either side** of a word.

For example, here are three sentences containing the out-of-vocabulary word "labrador" in different contexts.

```
doc1 = nlp(u"The labrador barked.")
doc2 = nlp(u"The labrador swam.")
doc3 = nlp(u"the labrador people live in canada.")

for doc in [doc1, doc2, doc3]:
    labrador = doc[1]
    dog = nlp(u"dog")
    print(labrador.similarity(dog))
```

Even though the model has never seen the word "labrador", it can make a fairly accurate prediction of its similarity to "dog" in different contexts.

The **labrador** barked.	0.56 ✓
The **labrador** swam.	0.48 ✗
the **labrador** people live in canada.	0.39 ✗

The same also works for whole documents. Here, the variance of the similarities is lower, as all words and their order are taken into account. However, the context-specific similarity is often still reflected pretty accurately.

```
doc1 = nlp(u"Paris is the largest city in France.")
doc2 = nlp(u"Vilnius is the capital of Lithuania.")
doc3 = nlp(u"An emu is a large bird.")

for doc in [doc1, doc2, doc3]:
    for other_doc in [doc1, doc2, doc3]:
        print(doc.similarity(other_doc))
```

Even though the sentences about Paris and Vilnius consist of different words and entities, they both describe the same concept and are seen as more similar than the sentence about emus. In this case, even a misspelled version of "Vilnius" would still produce very similar results.

	Paris is the largest city in France.	Vilnius is the capital of Lithuania.	An emu is a large bird.
Paris is the largest city in France.	1.00 ●	0.85 ✓	0.65 ✗
Vilnius is the capital of Lithuania.	0.85 ✓	1.00 ●	0.55 ✗
An emu is a large bird.	0.65 ✗	0.55 ✗	1.00 ●

Sentences that consist of the same words in different order will likely be seen as very similar – but never identical.

```
docs = [nlp(u"dog bites man"), nlp(u"man bites dog"),
        nlp(u"man dog bites"), nlp(u"dog man bites")]

for doc in docs:
    for other_doc in docs:
        print(doc.similarity(other_doc))
```

Interestingly, "man bites dog" and "man dog bites" are seen as slightly more similar than "man bites dog" and "dog bites man". This may be a coincidence – or the result of "man" being interpreted as both sentence's subject.

dog bites man	1.00 ●	0.90 ✓	0.89 ✓	0.92 ✓
man bites dog	0.90 ✓	1.00 ●	0.93 ✓	0.90 ✓
man dog bites	0.89 ✓	0.93 ✓	1.00 ●	0.92 ✓
dog man bites	0.92 ✓	0.90 ✓	0.92 ✓	1.00 ●

Customising word vectors

Word vectors let you import knowledge from raw text into your model. The knowledge is represented as a table of numbers, with one row per term in your vocabulary. If two terms are used in similar contexts, the algorithm that learns the vectors should assign them **rows that are quite similar**, while words that are used in different contexts will have quite different values. This lets you use the row-values assigned to the words as a kind of dictionary, to tell you some things about what the words in your text mean.

Word vectors are particularly useful for terms which **aren't well represented in your labelled training data**. For instance, if you're doing named entity recognition, there will always be lots of names that you don't have examples of. For instance, imagine your training data happens to contain some examples of the term "Microsoft", but it doesn't contain any examples of the term "Symantec". In your raw text sample, there are plenty of examples of both terms, and they're used in similar contexts. The word vectors make that fact available to the entity recognition model. It still won't see examples of "Symantec" labelled as a company. However, it'll see that "Symantec" has a word vector that usually corresponds to company terms, so it can **make the inference**.

In order to make best use of the word vectors, you want the word vectors table to cover a **very large vocabulary**. However, most words are rare, so most of the rows in a large word vectors table will be accessed very rarely, or never at all. You can usually cover more than **95% of the tokens** in your corpus with just **a few thousand rows** in the vector table. However, it's those **5% of rare terms** where the word vectors are **most useful**. The problem is that increasing the size of the vector table produces rapidly diminishing returns in coverage over these rare terms.

Optimising vector coverage V2.0

To help you strike a good balance between coverage and memory usage, spaCy's `Vectors` class lets you map **multiple keys** to the **same row** of the table. If you're using the `spacy vocab` command to create a vocabulary, pruning the vectors will be taken care of automatically. You can also do it manually in the following steps:

1. Start with a **word vectors model** that covers a huge vocabulary. For instance, the `en_vectors_web_lg` model provides 300-dimensional GloVe vectors for over 1 million terms of English.
2. If your vocabulary has values set for the `Lexeme.prob` attribute, the lexemes will be sorted by descending probability to determine which vectors to prune. Otherwise, lexemes will be sorted by their order in the `Vocab`.
3. Call `Vocab.prune_vectors` with the number of vectors you want to keep.

```
nlp = spacy.load('en_vectors_web_lg')
n_vectors = 105000  # number of vectors to keep
```

```
assert nlp.vocab.vectors.n_keys > n_vectors  # but not the total entries
```

`Vocab.prune_vectors` reduces the current vector table to a given number of unique entries, and returns a dictionary containing the removed words, mapped to `(string, score)` tuples, where `string` is the entry the removed word was mapped to, and `score` the similarity score between the two words.

REMOVED WORDS

```
{
    'Shore': ('coast', 0.732257),
    'Precautionary': ('caution', 0.490973),
    'hopelessness': ('sadness', 0.742366),
    'Continous': ('continuous', 0.732549),
    'Disemboweled': ('corpse', 0.499432),
    'biostatistician': ('scientist', 0.339724),
    'somewheres': ('somewheres', 0.402736),
    'observing': ('observe', 0.823096),
    'Leaving': ('leaving', 1.0)
}
```

In the example above, the vector for "Shore" was removed and remapped to the vector of "coast", which is deemed about 73% similar. "Leaving" was remapped to the vector of "leaving", which is identical.

Adding vectors V2.0

spaCy's new `Vectors` class greatly improves the way word vectors are stored, accessed and used. The data is stored in two structures:

- An array, which can be either on CPU or GPU.
- A dictionary mapping string-hashes to rows in the table.

Keep in mind that the `Vectors` class itself has no `StringStore`, so you have to store the hash-to-string mapping separately. If you need to manage the strings, you should use the `Vectors` via the `Vocab` class, e.g. `vocab.vectors`. To add vectors to the vocabulary, you can use the `Vocab.set_vector` method.

ADDING VECTORS

```
from spacy.vocab import Vocab

vector_data = {u'dog': numpy.random.uniform(-1, 1, (300,)),
               u'cat': numpy.random.uniform(-1, 1, (300,)),
               u'orange': numpy.random.uniform(-1, 1, (300,))}

vocab = Vocab()
for word, vector in vector_data.items():
    vocab.set_vector(word, vector)
```

Loading GloVe vectors V2.0

spaCy comes with built-in support for loading GloVe vectors from a directory. The `Vectors.from_glove` method assumes a binary format, the vocab provided in a `vocab.txt`, and the naming scheme of `vectors.{size}.[fd].bin`. For example:

DIRECTORY STRUCTURE

```
└── vectors
    ├── vectors.128.f.bin  # vectors file
    └── vocab.txt          # vocabulary
```

FILE NAME	DIMENSIONS	DATA TYPE
`vectors.128.f.bin`	128	float32
`vectors.300.d.bin`	300	float64 (double)

```
nlp = spacy.load('en')
nlp.vocab.vectors.from_glove('/path/to/vectors')
```

If your instance of `Language` already contains vectors, they will be overwritten. To create your own GloVe vectors model package like spaCy's `en_vectors_web_lg`, you can call `nlp.to_disk`, and then package the model using the `package` command.

Loading other vectors V2.0

You can also choose to load in vectors from other sources, like the fastText vectors for 294 languages, trained on Wikipedia. After reading in the file, the vectors are added to the `Vocab` using the `set_vector` method.

```
#!/usr/bin/env python
# coding: utf8
"""Load vectors for a language trained using fastText
https://github.com/facebookresearch/fastText/blob/master/pretrained-vectors.md
Compatible with: spaCy v2.0.0+
"""
from __future__ import unicode_literals
import plac
import numpy

```

spacy/examples/vectors_fast_text.py VIEW ON GITHUB

Using custom similarity methods

By default, `Token.vector` returns the vector for its underlying `Lexeme`, while `Doc.vector` and `Span.vector` return an average of the vectors of their tokens. You can customise these behaviours by modifying the `doc.user_hooks`, `doc.user_span_hooks` and `doc.user_token_hooks` dictionaries.

For more details on **adding hooks** and **overwriting** the built-in `Doc`, `Span` and `Token` methods, see the usage guide on user hooks.

Storing vectors on a GPU

If you're using a GPU, it's much more efficient to keep the word vectors on the device. You can do that by setting the `Vectors.data` attribute to a `cupy.ndarray` object if you're using spaCy or Chainer, or a `torch.Tensor` object if you're using PyTorch. The `data` object just needs to support `__iter__` and `__getitem__`, so if you're using another library such as TensorFlow, you could also create a wrapper for your vectors data.

SPACY, THINC OR CHAINER

```
import cupy.cuda
from spacy.vectors import Vectors

vector_table = numpy.zeros((3, 300), dtype='f')
vectors = Vectors([u'dog', u'cat', u'orange'], vector_table)
with cupy.cuda.Device(0):
    vectors.data = cupy.asarray(vectors.data)
```

PYTORCH

```
import torch
from spacy.vectors import Vectors

vector_table = numpy.zeros((3, 300), dtype='f')
vectors = Vectors([u'dog', u'cat', u'orange'], vector_table)
vectors.data = torch.Tensor(vectors.data).cuda(0)
```

SUGGEST EDITS </>