

1/11

easy start.

2. Gensim (<https://radimrehurek.com/gensim/>) is a topic modelling library (https://en.wikipedia.org/wiki/Topic_model) for Python that provides access to Word2Vec and other word embedding algorithms for training, and it also allows pre-trained word embeddings that you can download from the internet to be loaded.

In this post, we examine how to load pre-trained models first, and then provide a tutorial for creating your own word embeddings using Gensim (<https://radimrehurek.com/gensim/>) and the 20_newsgroups (<https://www.kaggle.com/crawford/20-newsgroups>) dataset.

Pre-trained Word Embeddings

Pre-trained models are the simplest way to start working with word embeddings. A pre-trained model is a set of word embeddings that have been created elsewhere that you simply load onto your computer and into memory.

The advantage of these models is that they can leverage massive datasets that you may not have access to, built using billions of different words, with a vast corpus of language that captures word meanings in a statistically robust manner. Example training data sets include the entire corpus of wikipedia text (<https://dumps.wikimedia.org>), the common crawl dataset (<http://commoncrawl.org>), or the Google News Dataset (<https://code.google.com/archive/p/word2vec/>). Using a pre-trained model removes the need for you to spend time obtaining, cleaning, and processing (intensively) such large datasets.

Pre-trained models are also available in languages other than English, opening up multi-lingual opportunities for your applications.

The disadvantage of pre-trained word embeddings is that the words contained within may not capture the peculiarities of language in your specific application domain. For example, Wikipedia may not have great word exposure to particular aspects of legal doctrine or religious text, so if your application is specific to a domain like this, your results may not be optimal due to the generality of the downloaded model's word embeddings.

Pre-trained models in Spacy

Using pre-trained models in Spacy (<https://spacy.io/usage/spacy-101>) is incredible convenient, given that they come **built in**. Simply download the core English model using (<https://spacy.io/usage/models>):

```
1 # run this from a normal command line
2 python -m spacy download en_core_web_md
```

Spacy has a number of different models (<https://spacy.io/usage/models>) of different sizes available for use, with models in 7 different languages (include English, Polish, German, Spanish, Portuguese, French, Italian, and Dutch), and of different sizes to suit your requirements. The code snippet above installs the larger-than-standard en_core_web_md (<https://spacy.io/models/en>) library, which includes 20k unique vectors with 300 dimensions.

```
doc = nlp(u"Apple and banana are similar. Pasta and hippo aren't.")
apple = doc[0]
banana = doc[2]
pasta = doc[6]
hippo = doc[8]
assert apple.similarity(banana) > pasta.similarity(hippo)
assert apple.has_vector
assert banana.has_vector
assert pasta.has_vector
assert hippo.has_vector
```

(<https://shanelynnwebsite-mid9n9g1q9y8tt.netdna-ssl.com/wp-content/uploads/2018/03/example-use-spacy-vectors.png>)

Spacy parses entire blocks of text and seamlessly assigns word vectors from the loaded models.

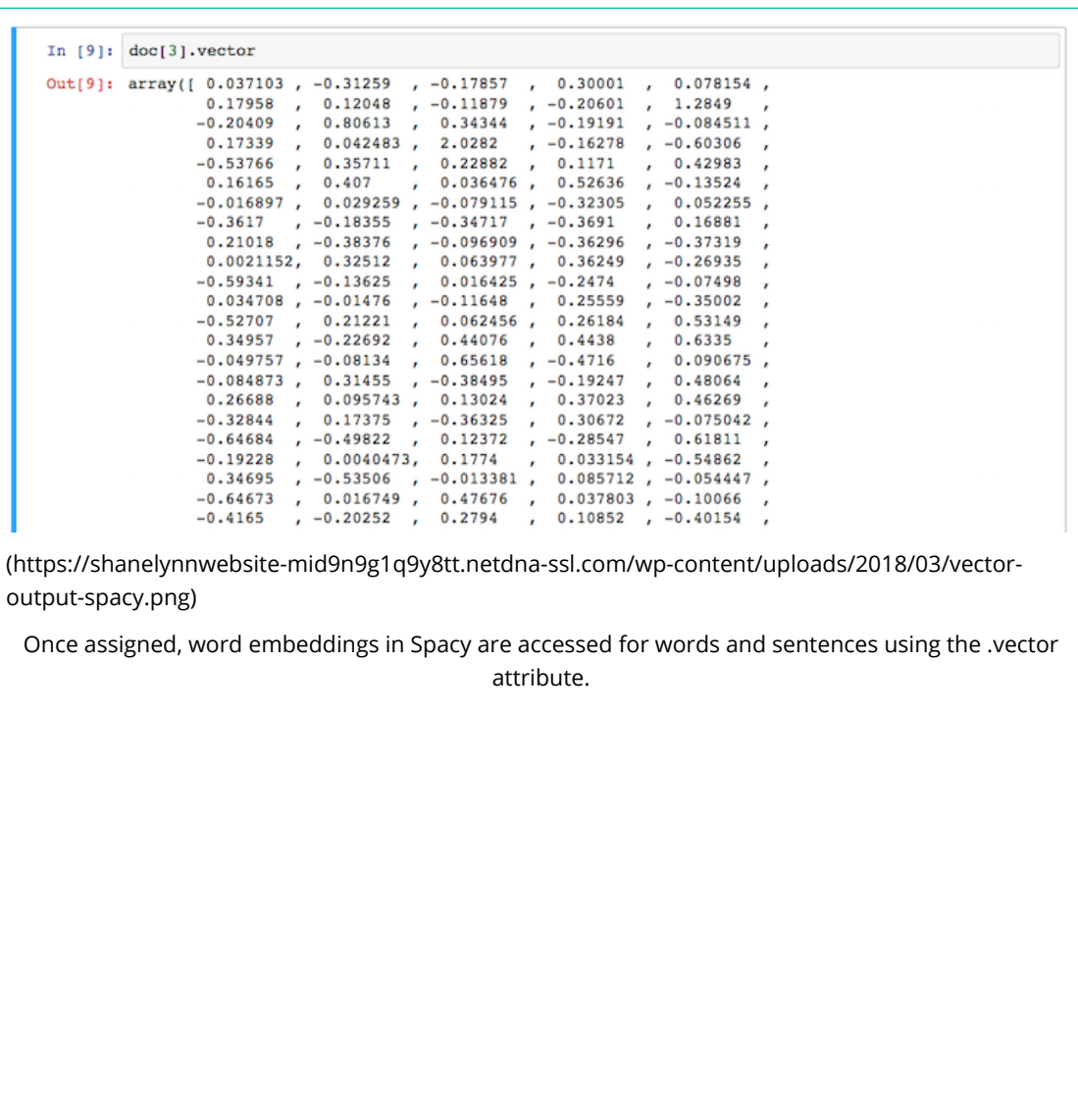
Use the vectors in Spacy by first loading the model (<https://spacy.io/usage/spacy-101>), and then processing text (see below):

```

1 import spacy
2
3 # Load the spacy model that you have installed
4 nlp = spacy.load('en_core_web_md')
5
6 # process a sentence using the model
7 doc = nlp("This is some text that I am processing with Spacy")
8
9 # It's that simple - all of the vectors and words are assigned after this point
10 # Get the vector for 'text':
11 doc[3].vector
12
13 # Get the mean vector for the entire sentence (useful for sentence classification etc.)
14 doc.vector

```

The vectors can be accessed directly using the `.vector` attribute of each processed token (word). The mean vector for the entire sentence is also calculated simply using `.vector`, providing a very convenient input for machine learning models based on sentences.



The screenshot shows a Jupyter Notebook cell with the input `In [9]: doc[3].vector` and the output `Out[9]: array([0.037103, -0.31259, -0.17857, 0.30001, 0.078154, ..., -0.4165, -0.20252, 0.2794, 0.10852, -0.40154])`. The output is a NumPy array of 300 floating-point numbers, representing the word vector for the word 'text'.

(<https://shanelynnwebsite-mid9n9g1q9y8tt.netdna-ssl.com/wp-content/uploads/2018/03/vector-output-spacy.png>)

Once assigned, word embeddings in Spacy are accessed for words and sentences using the `.vector` attribute.

Pre-trained models in Gensim

Gensim (<https://spacy.io/usage/spacy-101>) doesn't come with the same in built models as Spacy, so to load a pre-trained model into Gensim, you first need to find and download one. This post on Ahogrammers's blog (<http://ahogrammer.com/2017/01/20/the-list-of-pretrained-word-embeddings/>) provides a list of pertained models that can be downloaded and used.

A popular pre-trained option is the Google News dataset model, containing 300-dimensional embeddings for 3 millions words and phrases. Download the binary file 'GoogleNews-vectors-negative300.bin' (1.3 GB compressed) from <https://code.google.com/archive/p/word2vec/> (<https://code.google.com/archive/p/word2vec/>).

Loading and accessing vectors is then straightforward:

```

1 from gensim.models import KeyedVectors
2
3 # Load vectors directly from the file
4 model = KeyedVectors.load_word2vec_format('data/GoogleGoogleNews-vectors-negative300.bin', binary=True)
5
6 # Access vectors for specific words with a keyed lookup:
7 vector = model['easy']
8 # see the shape of the vector (300,)
9 vector.shape
10
11 # Processing sentences is not as simple as with Spacy:
12 vectors = [model[x] for x in "This is some text I am processing with Spacy".split(' ')]

```

Gensim includes functions to explore the vectors loaded, examine word similarity, and to find synonyms in of words using 'similar' vectors:

```

In [17]: model.similarity('straightforward', 'easy')
Out[17]: 0.5717043285477517

In [18]: model.similarity('simple', 'impossible')
Out[18]: 0.29156160264633707

In [19]: model.most_similar('simple')
Out[19]: [('straightforward', 0.7460169196128845),
          ('Simple', 0.7108174562454224),
          ('uncomplicated', 0.6297484636306763),
          ('simplest', 0.6171397566795349),
          ('easy', 0.5990299582481384),
          ('fairly_straightforward', 0.5893306732177734),
          ('deceptively_simple', 0.5743066072463989),
          ('simpler', 0.5537199378013611),
          ('simplistic', 0.5516539216041565),
          ('disarmingly_simple', 0.5365327000617981)]

```

(<https://shanelynwebsite-mid9n9g1q9y8tt.netdna-ssl.com/wp-content/uploads/2018/03/similarity-vectors-gensim.png>)

Gensim provides a number of helper functions to interact with word vector models. Similarity is determined using the cosine distance between two vectors.

Create Custom Word Embeddings

Training your own word embeddings need not be daunting, and, for specific problem domains, will lead to enhanced performance over pre-trained models. The Gensim library provides a simple API (<https://radimrehurek.com/gensim/models/word2vec.html>) to the Google word2vec (<https://code.google.com/archive/p/word2vec/>) algorithm which is a go-to algorithm for beginners.

To train your own model, the main challenge is getting access to a training data set. Computation is not massively onerous – you'll manage to process a large model on a powerful laptop in hours rather than days.

In this tutorial, we will train a Word2Vec model based on the 20_newsgroups data set (<http://qwone.com/~jason/20Newsgroups/>) which contains approximately 20,000 posts distributed across 20 different topics. The simplicity of the Gensim Word2Vec (<https://radimrehurek.com/gensim/models/word2vec.html>) training process is demonstrated in the code snippets below.

Training the model in Gensim requires the input data in a list of sentences, with each sentence being a list of words, for example:

```

1 input_data = [['This', 'is', 'sentence', 'one'], ['And', 'this', 'is', 'sentence', 'two']]

```

As such, our initial efforts will be in cleansing and formatting the data to suit this form.

Preparing 20 Newsgroups Data

Once the newsgroups archive is extracted into a folder, there are some cleaning and extraction steps taken to get data into the input form and then training the model:

```
1 # Import libraries to build Word2Vec model, and load Newsgroups data
2 import os
3 import sys
4 import re
5 from gensim.models import Word2Vec
6 from gensim.models.phrases import Phraser, Phrases
7 TEXT_DATA_DIR = './data/20_newsgroup/'

1 # Newsgroups data is split between many files and folders.
2 # Directory structure 20_newsgroup/<newsgroup label>/<post ID>;
3
4 texts = [] # list of text samples
5 labels_index = {} # dictionary mapping label name to numeric id
6 labels = [] # list of label ids
7 label_text = [] # list of label texts
8
9 # Go through each directory
10 for name in sorted(os.listdir(TEXT_DATA_DIR)):
11     path = os.path.join(TEXT_DATA_DIR, name)
12     if os.path.isdir(path):
13         label_id = len(labels_index)
14         labels_index[name] = label_id
15         for fname in sorted(os.listdir(path)):
16             # News groups posts are named as numbers, with no extensions.
17             if fname.isdigit():
18                 fpath = os.path.join(path, fname)
19                 f = open(fpath, encoding='latin-1')
20                 t = f.read()
21                 i = t.find('\n\n') # skip header in file (starts with two newlines.)
22                 if 0 < i:
23                     t = t[i:]
24                 texts.append(t)
25                 f.close()
26                 labels.append(label_id)
27                 label_text.append(name)
28
29 print('Found %s texts.' % len(texts))
30 # >>> Found 1997 texts.
```

The data is loaded into memory (a single list 'texts') at this point; for preprocessing, remove all punctuation, and excess information.

```
1 # Cleaning data - remove punctuation from every newsgroup text
2 sentences = []
3 # Go through each text in turn
4 for ii in range(len(texts)):
5     sentences = [re.sub(pattern=r'[\!"#%&*\+,-./:;&=;>?@^_`()|~]=',
6                       repl='',
7                       string=x
8                       ).strip().split(' ') for x in texts[ii].split('\n')]
9     if not x.endswith('writes:')]
10    sentences = [x for x in sentences if x != ['']]
11    texts[ii] = sentences
```

Each original document is now represented in the list, 'texts', as a list of sentences, and each sentence is a list of words.

```
In [159]: # Example output
# Each entry in the newsgroups data is a list of sentences, each sentence i
print(texts[6])

[['The', 'motto', 'originated', 'in', 'the', 'StarSpangled', 'Banner', ' ',
  'Tell', 'me', 'that', 'this', 'has'], ['something', 'to', 'do', 'with',
  'atheists'], ['The', 'motto', 'oncoins', 'originated', 'as', 'a', 'McCart
  hyite', 'smear', 'which', 'equated', 'atheism'], ['with', 'Communism', 'a
  nd', 'called', 'both', 'unamerican'], ['No', 'it', 'didn't', ' ', 'The', '
  motto', 'has', 'been', 'on', 'various', 'coins', 'since', 'the', 'Civil',
  'War'], ['It', 'was', 'just', 'required', 'to', 'be', 'on', 'all', 'curre
  ncy', 'in', 'the', '50's'], ['keith']]
```

(<https://shanelynnwebsite-mid9n9g1q9y8tt.netdna-ssl.com/wp-content/uploads/2018/03/example-data-format-gensim-word2vec-training.png>)

For training word embedding models, a list of sentences, where each sentence is a list of words is created. The source data here is the 20_newsgroups (<http://qwone.com/~jason/20Newsgroups/>) data set.

Finally, combine all of the sentences from every document into a single list of sentences.


```

1 # concatenate all sentences from all texts into a single list of sentences
2 all_sentences = []
3 for text in texts:
4     all_sentences += text

```

Phrase Detection using Gensim Phraser

Commonly occurring multiword expressions (bigrams / trigrams) in text carry different meaning to the words occurring singularly. For example, the words 'new' and 'York' expressed singularly are inherently different to the utterance 'New York'. Detecting frequently co-occurring words and combining them can enhance word vector accuracy.

A 'Phraser' (<https://radimrehurek.com/gensim/models/phrases.html>) from Gensim (<https://radimrehurek.com/gensim/>) can detect frequently occurring bigrams easily, and apply a transform to data to create pairs, i.e. 'New York' -> 'New_York'. Pre-processing text input to account for such bigrams can improve the accuracy and usefulness of the resulting word vectors. Ultimately, instead of training vectors for 'new' and 'york' separately, a new vector for 'New_York' is created.

The gensim.models.phrases (<https://radimrehurek.com/gensim/models/phrases.html>) module provides everything required in a simple form:

```

1 # Phrase Detection
2 # Give some common terms that can be ignored in phrase detection
3 # For example, 'state_of_affairs' will be detected because 'of' is provided here:
4 common_terms = ["of", "with", "without", "and", "or", "the", "a"]
5 # Create the relevant phrases from the list of sentences:
6 phrases = Phrases(all_sentences, common_terms=common_terms)
7 # The Phraser object is used from now on to transform sentences
8 bigram = Phraser(phrases)
9
10 # Applying the Phraser to transform our sentences is simply
11 all_sentences = list(bigram[all_sentences])

```

```
In [7]: len(all_sentences)
```

```
Out[7]: 564196
```

```
In [8]: print(all_sentences[5676])
```

```
['guilty', 'in', 'a', 'court', 'of', 'law', '', 'As', 'his', 'guilt', 'has',
 'not', 'been', 'established', 'it', 'is']
```

```
In [9]: # Phrase Detection
common_terms = ["of", "with", "without", "and", "or", "the", "a"]
phrases = Phrases(all_sentences, common_terms=common_terms)
bigram = Phraser(phrases)
```

```
In [10]: print(bigram[all_sentences[5676]])
```

```
['guilty', 'in', 'a', 'court_of_law', 'As', 'his', 'guilt', 'has', 'not',
 'been_established', 'it', 'is']
```

```
In [11]: all_sentences = list(bigram[all_sentences])
```

(<https://shanelynnwebsite-mid9n9g1q9y8tt.netdna-ssl.com/wp-content/uploads/2018/03/Phraser-application-example-gensim.png>)

A Phraser detects frequently co-occurring words in sentences and combines them. Training and applying is simple using the Gensim library.

The Gensim Phraser process can be repeated to detect trigrams (groups of three words that co-occur) and more by training a second Phraser object on the already processed data. (see gensim docs (<https://radimrehurek.com/gensim/models/phrases.html>)). The parameters are tuneable to include or exclude terms based on their frequency, and should be fine tuned. In the example above, 'court_of_law' is a good example phrase, whereas 'been_established' may indicate an overly greedy application of the phrase detection algorithm.

Creating the Word Embeddings using Word2Vec

The final step, once data has been preprocessed and cleaned is creating the word vectors.

```

1 model = Word2Vec(all_sentences,
2                 min_count=3, # Ignore words that appear less than this
3                 size=200,    # Dimensionality of word embeddings
4                 workers=2,    # Number of processors (parallelisation)
5                 window=5,     # Context window for words during training
6                 iter=30)      # Number of epochs training over corpus

```

This example, with only 564k sentences, is a toy example, and the resulting word embeddings would not be expected to be as useful as those trained by Google / Facebook on larger corpus' of training data.

In total, the 20_newsgroups dataset provided 80,167 different words for our model, and, even with the smaller data set, relationships between words can be observed.

```
In [184]: model
Out[184]: <gensim.models.word2vec.Word2Vec at 0x2496d2400>
```

```
In [185]: # Word embedding dimensions
          model.vector_size
```

```
Out[185]: 200
```

```
In [197]: # Total number of words in model
          len(model.wv.vocab)
```

```
Out[197]: 80167
```

```
In [186]: model.most_similar('New_York')
```

```
Out[186]: [('Los_Angeles', 0.4942966401576996),
            ('City', 0.46237313747406006),
            ('Virginia', 0.4425775408744812),
            ('Florida', 0.4137933552265167),
            ('anytime_soon', 0.40825462341308594),
            ('Texas', 0.4043610990047455),
            ('Boston', 0.40270715951919556),
            ('Colorado', 0.3968525528907776),
            ('Washington', 0.39651554822921753),
            ('County', 0.3962520956993103)]
```

```
In [187]: model.most_similar('engine')
```

```
Out[187]: [('car', 0.5038468837738037),
            ('fuel', 0.4788314402103424),
            ('suspension', 0.4744623303413391),
            ('motor', 0.4727802276611328),
            ('bike', 0.4694923758506775),
            ('clutch', 0.45287370681762695),
            ('battery', 0.4508274793624878),
            ('valve', 0.44455477595329285),
            ('tires', 0.43563735485076904),
            ('drive', 0.4341405928134918)]
```

(<https://shanelynnwebsite-mid9n9g1q9y8tt.netdna-ssl.com/wp-content/uploads/2018/03/word-vector-training-results.png>)

Even with the relatively small (80k unique words) dataset, some informative relations are seen in trained word embeddings.

There are a range of tuneable parameters (<https://radimrehurek.com/gensim/models/word2vec.html>) for the Word2Vec algorithm provided by Gensim to assist in achieving the desired result.

For larger data sets, training time will be much longer, and memory can be an issue if all of the training data is loaded as in our example above. The Rare Technologies blog (<https://rare-technologies.com/word2vec-tutorial/>) provides some useful information for formatting input data as an iterable, reducing memory footprint during the training process, and also in methods for evaluating word vector and performance after training.

Once trained, you can access the newly encoded word vectors in the same way as for pretrained models, and use the outputs in any of your text classification or visualisation tasks.

In addition to Word2Vec, Gensim also includes algorithms for fasttext (<https://radimrehurek.com/gensim/models/fasttext.html#module-gensim.models.fasttext>), VarEmbed (<https://github.com/rguthrie3/MorphologicalPriorsForWordEmbeddings>), and WordRank (<https://radimrehurek.com/gensim/models/wrappers/wordrank.html>) (original (<https://bitbucket.org/shihaoji/wordrank/>)) also.

Conclusion

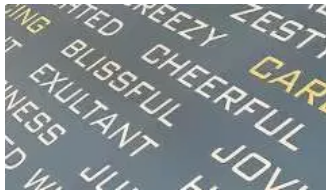
Ideally, this post will have given enough information to start working in Python with Word embeddings, whether you intend to use off-the-shelf models or models based on your own data sets.

A third option exists, which is to take an off-the-shelf model, and then 'continue' the training using Gensim, but with your own application-specific data and vocabulary, also mentioned on the Rare Technologies blog (<https://rare-technologies.com/word2vec-tutorial/>).

For further, and useful reading on these topics, please see:

- My introductory post on Word Embeddings. (<https://www.shanelynn.ie/get-busy-with-word-embeddings-introduction/>)
- Rare Technologies blog post (<https://rare-technologies.com/word2vec-tutorial/>) on training without heavy RAM usage, and evaluation of model.
- Word Vector Tutorial from Machine Learning Mastery (<https://machinelearningmastery.com/develop-word-embeddings-python-gensim/>) that includes some useful visualisation techniques for the resulting model.
- Tutorial from Kavita Ganesan (<http://kavita-ganesan.com/gensim-word2vec-tutorial-starter-code/#.WrfjMMgo924>) training word embeddings on review data.

Related



(<https://www.shanelynn.ie/get-busy-with-word-embeddings-introduction/>)
Get Busy with Word Embeddings - An Introduction (<https://www.shanelynn.ie/get-busy-with-word-embeddings-introduction/>)
February 8, 2018
In "blog"



(<https://www.shanelynn.ie/online-learning-curriculum-for-data-scientists/>)
Online Learning Curriculum for Data Scientists (<https://www.shanelynn.ie/online-learning-curriculum-for-data-scientists/>)
December 18, 2013
In "blog"

(<https://www.shanelynn.ie/using-python-threading-for-multiple-results-queue/>)
Using Python Threading and Python's Multiple Results Queue (Tutorial) (<https://www.shanelynn.ie/using-python-threading-for-multiple-results-queue/>)
I recently had an issue with a long running web process that I needed to substantially speed up due to timeouts. The delay arose because the system needed to fetch data from a number of URLs. The total number of December 18, 2014
In "blog"

■ [blog](https://www.shanelynn.ie/category/blog/) (<https://www.shanelynn.ie/category/blog/>), [data science](https://www.shanelynn.ie/category/blog/data-science/) (<https://www.shanelynn.ie/category/blog/data-science/>), [Natural Language Processing](https://www.shanelynn.ie/category/blog/natural-language-processing/) (<https://www.shanelynn.ie/category/blog/natural-language-processing/>), [python](https://www.shanelynn.ie/category/blog/python/) (<https://www.shanelynn.ie/category/blog/python/>), [Tutorials](https://www.shanelynn.ie/category/tutorials/) (<https://www.shanelynn.ie/category/tutorials/>)

🔍 [20_newsgroups](https://www.shanelynn.ie/tag/20_newsgroups/) (https://www.shanelynn.ie/tag/20_newsgroups/), [data science](https://www.shanelynn.ie/tag/data-science/) (<https://www.shanelynn.ie/tag/data-science/>), [gensim](https://www.shanelynn.ie/tag/gensim/) (<https://www.shanelynn.ie/tag/gensim/>), [modelling](https://www.shanelynn.ie/tag/modelling/) (<https://www.shanelynn.ie/tag/modelling/>), [natural language processing](https://www.shanelynn.ie/tag/natural-language-processing/) (<https://www.shanelynn.ie/tag/natural-language-processing/>), [nlp](https://www.shanelynn.ie/tag/nlp/) (<https://www.shanelynn.ie/tag/nlp/>), [python](https://www.shanelynn.ie/tag/python/) (<https://www.shanelynn.ie/tag/python/>), [space](https://www.shanelynn.ie/tag/space/) (<https://www.shanelynn.ie/tag/space/>), [text processing](https://www.shanelynn.ie/tag/text-processing/) (<https://www.shanelynn.ie/tag/text-processing/>), [tutorial](https://www.shanelynn.ie/tag/tutorial/) (<https://www.shanelynn.ie/tag/tutorial/>), [unsupervised training](https://www.shanelynn.ie/tag/unsupervised-training/) (<https://www.shanelynn.ie/tag/unsupervised-training/>), [word embeddings](https://www.shanelynn.ie/tag/word-embeddings/) (<https://www.shanelynn.ie/tag/word-embeddings/>), [word vectors](https://www.shanelynn.ie/tag/word-vectors/) (<https://www.shanelynn.ie/tag/word-vectors/>)

← [Get Busy with Word Embeddings - An Introduction](https://www.shanelynn.ie/get-busy-with-word-embeddings-introduction/) (<https://www.shanelynn.ie/get-busy-with-word-embeddings-introduction/>)

2 thoughts on “Word Embeddings in Python with Spacy and Gensim”

Franco Arda (https://www.facebook.com/app_scoped_user_id/713176574/)
Reply (<https://www.shanelynn.ie/word-embeddings-in-python-with-spacy-and-gensim/?replytocom=718#respond>)
April 20, 2018 at 10:26 am (<https://www.shanelynn.ie/word-embeddings-in-python-with-spacy-and-gensim/#comment-718>)

@Shane, great post on a hot topic!

You guys are so lucky working with an English corpus!

If you change the language, it all gets exponentially more complicated:

- To my knowledge, you can't train GloVe with your own corpus.
- Word2Vec is easily trained with Gensim, but everything else is not easy 😊
- Word2Vec needs a huge corpus. My guess is >500MB

My personal conclusion for non-English word modeling:

- stick to BoW (bi-gram). Unless you're a monster tech firm, BoW (bi-gram) works surprisingly well.

PS: for those into Deep Learning + Natural Language Processing, check out prodi.gy (from the makers of SpaCy). Prodi.gy enables you to manually label sentences efficiently.