Published in Towards Data Science



Omar Sharaki Follow

Jul 10, 2020 · 12 min read ★ · Listen



# Detecting Document Similarity With Doc2vec

A step-by-step, hands-on introduction in Python

"assorted berries" by William Felker on Unsplash

There is no shortage of ways out there that we can use to analyze and make sense of textual data. Such methods generally deal with an area of artificial intelligence called Natural Language Processing (NLP).

NLP allows us to perform a multitude of tasks where our data consists of text or speech. Sentiment analysis, machine translation, and information retrieval are just a few examples of NLP applications, many of which we use daily. Today, many of these tasks can be solved

In this post, I'd like to illustrate one such method, doc2vec, and hopefully provide some basic insight into how it works and how to implement it.

## The task

Put simply, given a large number of text documents, we want to be able to:

1. Measure how similar the documents are to each other semantically.
2. Use this information to cluster the documents based on their similarities.

## What we'll cover

I realize this is a longer post. So before we get started, here's an outline of everything we'll cover:

1. Introducing the dataset
2. An overview of doc2vec and vector representations
3. Training the doc2vec models
4. Visualizing the generated document vectors
5. Evaluating the models

Feel free to read it at whatever pace feels comfortable to you. I even encourage you to break it up into parts and read it over multiple sessions as you see fit to stay engaged.

## The data

For the purpose of training and testing our models, we're going to be using the 20Newsgroups data set. This data set consists of about 18000 newsgroup posts on 20

| | | |
|---|---|---|
| comp.sys.ibm.pc.hardware<br>comp.sys.mac.hardware<br>comp.windows.x | rec.motorcycles<br>rec.sport.baseball<br>rec.sport.hockey | sci.electronics<br>sci.med<br>sci.space |
| misc.forsale | talk.politics.misc<br>talk.politics.guns<br>talk.politics.mideast | talk.religion.misc<br>alt.atheism<br>soc.religion.christian |

Structure of the 20Newsgroups data set

To speed up training and to make our later evaluation clearer, we limit ourselves to four categories. Also, to ensure that these categories are as distinct as possible, the four categories are chosen such that they don't belong to the same partition.

For example, this would mean that instead of picking *rec.sport.baseball* and *rec.sport.hockey*, we might want to replace one of them with, say, *soc.religion.christian*. Here, I decided to go with the categories *soc.religion.christian*, *sci.space*, *talk.politics.mideast*, and *rec.sport.baseball*.

Having chosen the categories, their documents are split into training and test sets, while keeping track of which documents belong to which category in order to make it easier to judge the models' performance later.

In Python we can use sklearn to get the data:

```
from sklearn import datasets

categories = ["soc.religion.christian", "sci.space", "talk.politics.mideast", "rec.sport
cat_dict = {} # Contains raw training data organized by category
cat_dict_test = {} # Contains raw test data organized by category
for cat in categories:
    cat_dict[cat] = datasets.fetch_20newsgroups(subset='train', remove=('headers', 'foot
    cat_dict_test[cat] = datasets.fetch_20newsgroups(subset='test', remove=('headers', '
```

**20Newsgroupsimport.py** hosted with ❤️ by **GitHub** **view raw**

where one dictionary contains training documents and the other contains test documents. Furthermore, the parameter `remove=('headers', 'footers', 'quotes')` removes metadata such as headers, footers, and quotes from the documents in order to prevent our models from overfitting to them.

## Generating vectors

So now that we have our data, let's revisit our task. Remember that we first want to make sense of our documents and how their contexts relate to each other.

We want to be able to measure how similar the documents are to each other semantically

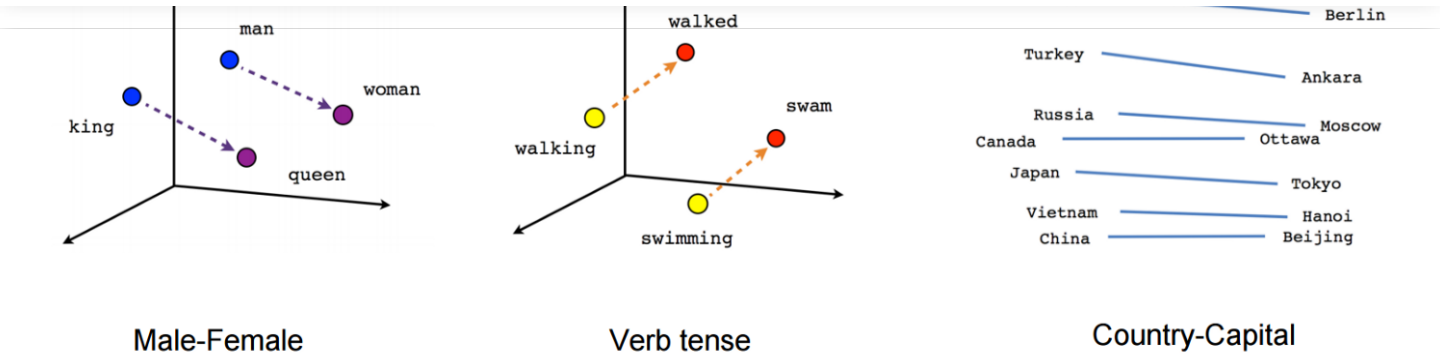
In other words, what we want to do is transform our text documents into a numerical, vectorized form, which can later be used by the clustering algorithm to group similar documents together.

### Doc2vec

One algorithm for generating such vectors is doc2vec [1]. A great introduction to the concept can be found in Gidi Shperber's article. Essentially, doc2vec uses a neural network approach to create vector representations of variable-length pieces of text, such as sentences, paragraphs, or documents. These vector representations have the advantage that they capture the semantics, i.e. the meaning, of the input texts. This means that texts which are similar in meaning or context will be closer to each other in vector space than texts which aren't necessarily related.

Doc2vec builds upon another algorithm called word2vec [2]. As you might have guessed from the name, word2vec functions in a very similar way to doc2vec, except that instead of giving us document vectors we get word vectors. This means that words such as "fast" and "quick" will be closer in vector space to each other than to "London", for example. Not only that, but these vector representations can be used to perform simple vector operations. For example, `vector("King") – vector("Man") + vector("Woman")` results in a vector that is most similar to the vector representation of "Queen".

For a slightly more in-depth introduction to word2vec, I recommend taking a look at this


Showing relationships between words in vector space — TensorFlow

## Training

So now that we have an idea of what doc2vec does, let's take a look at how we can train a doc2vec model on our data. Much of the following implementation is inspired by this tutorial, which I highly recommend checking out.

First, we need to tweak our raw data just a little to prepare it for training.

```
import gensim

def tokenize(text, stopwords, max_len = 20):
    return [token for token in gensim.utils.simple_preprocess(text, max_len=max_len) if

cat_dict_tagged_train = {} # Contains clean tagged training data organized by category.
cat_dict_test_clean = {} # Contains clean un-tagged training data organized by category

offset = 0 # Used for managing IDs of tagged documents
for k, v in cat_dict.items():
    cat_dict_tagged_train[k] = [gensim.models.doc2vec.TaggedDocument(tokenize(text, [],
    offset += len(v)

offset = 0
for k, v in cat_dict_test.items():
    cat_dict_test_clean[k] = [tokenize(text, [], max_len=200) for i, text in enumerate(
    offset += len(v)

# Eventually contains final versions of the training data to actually train the model
train_corpus = [taggeddoc for taggeddoc_list in list(cat_dict_tagged_train.values()) fo
```

Again, we are using dictionaries to keep track of which documents belong to which categories. In order to train a doc2vec model, the training documents need to be in the form *TaggedDocument*, which basically means each document receives a unique id, provided by the variable `offset`.

Furthermore, the function `tokenize()` transforms the document from a string into a list of strings consisting of the document's words. It also allows for a selection of **stopwords** as well as words exceeding a certain length to be removed.

Stopwords are usually common words that add no contextual meaning to a piece of text and are thus removed. You'll notice that here, I've opted not to remove any stopwords as performance seemed to be slightly better this way.

Normally, stopword removal is highly dependent on the task at hand, and finding out which stopwords to remove, if at all, is not always straightforward.

What we end up with are the following variables:

- `train_corpus` : a list of training ready documents
- `cat_dict_test_clean` : contains tokenized test documents organized by category

*Note that only the documents that will actually be used for training need to be tagged.*

With our training documents ready, we can now actually start training our model.

```
model = gensim.models.doc2vec.Doc2Vec(vector_size=30, min_count=2, epochs=40, window=2)
model.build_vocab(train_corpus)
model.train(train_corpus, total_examples=model.corpus_count, epochs=model.epochs)
```

**train_doc2vec_model.py** hosted with ❤ by **GitHub** **view raw**

Training the model

We first create a *doc2vec* object, which will serve as our model, and initialize it with different hyperparameter values.


Without going into too much detail, the `window` is used during training to determine how many words to include as a given word's context while inspecting it. More in section 2.2. of [1].

When training such models from scratch, the optimal values for these parameters are found through a process called *hyperparameter tuning*.

*The values I'm using here are by no means optimal nor are they set in stone. Feel free to experiment with training different models using different sets of hyperparameters.*

We then build the vocabulary, which is basically a dictionary that contains the occurrence counts of all unique words in the training corpus. Finally, the model is trained.

We can now infer new vectors for unseen documents from the test set and use them to evaluate our model. And this is where keeping track of which categories our documents belong to comes in handy.

```
1  metadata = {}
2  inferred_vectors_test = {} # Contains, category-wise, inferred doc vecs for each documen
3  for cat, docs in cat_dict_test_clean.items():
4      inferred_vectors_test[cat] = [model.infer_vector(doc) for doc in list(docs)]
5      metadata[cat] = len(inferred_vectors_test[cat])
```

**inferring_vectors.py** hosted with ❤️ by **GitHub** **view raw**

Inferring document vectors for test documents and organizing them to save to file later

We save these vectors category-wise in `inferred_vectors_test` . At the same time, we initialize another dictionary, `metadata` , that maps each category to an integer corresponding to the number of inferred vectors for that category. If this seems strange it will make sense in just a minute. These two variables can now be used to create two files like so:

```
    with open(output_file, "w") as f:
        writer = csv.writer(f, delimiter=delimiter)
        writer.writerows(input)

veclist_metadata = []
veclist = []
for cat in cat_dict.keys():
    for tag in [cat]*metadata[cat]:
        veclist_metadata.append([tag])
    for vec in inferred_vectors_test[cat]:
        veclist.append(list(vec))
write_to_csv(veclist, "doc2vec_20Newsgroups_vectors.csv")
write_to_csv(veclist_metadata, "doc2vec_20Newsgroups_vectors_metadata.csv")
```

**vectors_to_file.py** hosted with ❤️ by **GitHub** **view raw**

Writing vectors and metadata to file

The first of the two files, `doc2vec_20Newsgroups_vectors.csv`, contains one inferred document vector per line represented as tab-separated values, where the vectors are ordered by category.

The second file, `doc2vec_20Newsgroups_vectors_metadata.csv`, contains on each line the category of the corresponding vector in the first file. This might look something like this:

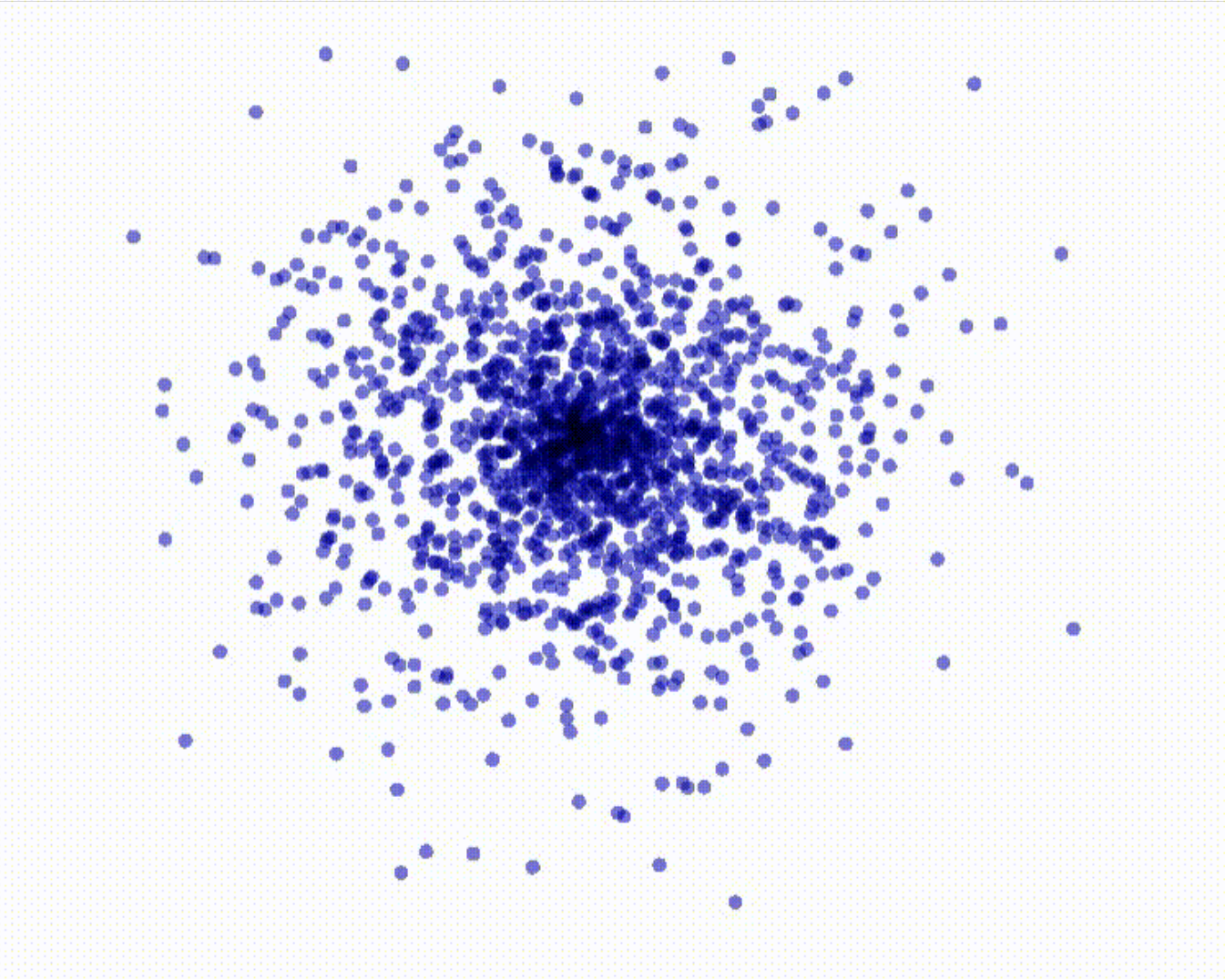
```
talk.politics.mideast
talk.politics.mideast
.
.
.
talk.politics.mideast
rec.sport.baseball
rec.sport.baseball
.
.
.
rec.sport.baseball
sci.space
sci.space
```


```
.
.
.
soc.religion.christian
```

## Visualization

So why are we saving our vectors and their metadata to files anyways? Well, we can now use these two files to visualize the similarities between our documents using TensorFlow's projector tool. In the projector tool, you can choose between different dimensionality reduction methods, namely t-SNE, PCA, and custom axis labeling, to represent the vectors in either 2D or 3D space.

The projector tries to cluster the data points so that similar points are closer to each other. Each method will represent the relationships between data points, that is, distribute the data points differently. For example, one method will focus more on representing local similarities between individual points while another might focus on preserving the overall structure of the data set.

Visualizing the document vectors with a 2D t-SNE plot

Aside from looking extremely cool, we can use these visualizations as a way of judging the quality of our vectors. By searching for the category names given to each vector in the metadata file we can see which category each point on the plot belongs to.

You'll notice, however, that you might have a lot of points lying somewhere in the middle not really belonging to any cluster. A reason for this could be your choice of dimensionality reduction method; specifically for t-SNE, the parameter values used have a huge influence on the distribution of the data points.

Furthermore, the fact that some documents may simply be vague in their context cannot be


Same distribution of data points shown multiple times highlighting each category's points

You can try out the projector yourself using the vectors and metadata used in the above example by following this link.

To inspect relationships between documents a bit more numerically, we can calculate the cosine distances between their inferred vectors by using the `similarity_unseen_docs()` function.

This is extremely useful if we want to compare individual documents, but if we need to evaluate our model's performance we'll have to expand this to include not just individual documents, but rather some portion of our data set.

## Evaluating our model

Intuitively, one would expect documents belonging to the same category to be more similar to each other than to documents belonging to other categories. And that's exactly the metric we're going to judge our model by; a good model should give higher similarity values for documents of the same category than for cross-category documents. So before getting into the nitty-gritty of how our code will look, let's first examine how we're going to structure our comparisons.

The first thing we'll do is create sets of document pairs for all categories. More specifically, given our four categories, which we'll denote by $C_1$,..,$C_4$, where each category is a set of documents, we get the following category pairs:

- ($C_1$, $C_1$), ($C_1$, $C_2$), ($C_1$, $C_3$), ($C_1$, $C_4$)
- ($C_2$, $C_2$), ($C_2$, $C_3$), ($C_2$, $C_4$)
- ($C_3$, $C_3$), ($C_3$, $C_4$)
- ($C_4$, $C_4$)

*Note that pairs such as (C3, C4) and (C4, C3) are equivalent when measuring cosine similarity and thus to avoid redundancy only one of the two combinations is considered.*

A pair (Ca, Cb) corresponds to the Cartesian product of the set containing all documents in category a and the set containing all documents in category b. More formally:

$$(C_a, C_b) = C_a \times C_b = \{(d_a, d_b) \mid d_a \in C_a, d_b \in C_b\}$$

Now, for each document pair we calculate the documents' similarity. This results, for each


$$C_a \times C_b = \begin{bmatrix} \cdot & \cdot & \cdot \\ (d_{an}, d_{b1}) & \cdots & (d_{an}, d_{bm}) \end{bmatrix} = \begin{bmatrix} \cdot & \cdot & \cdot \\ 0.56 & \cdots & 0.2 \end{bmatrix}$$

Possible similarity matrix assuming two categories **a** and **b** containing **n** and **m** documents respectively.

Since we're using these matrices to judge how similar the documents in two categories are overall, it would certainly make our job a whole lot easier to condense all the values in a matrix into a single value that acts as a measure of similarity. So what we'll do for each matrix is add up all values inside it to get a single value, which we'll call *similarity total* then divide this value by the total number of elements in the matrix to get an *average similarity* value.

Remember that we're judging our model not by how similar it tells us documents from the same category are, but rather how much more similar these same-category documents are to each other than to documents from other categories. So the value we're really after is not only how high the *average similarity* of, say, ($C_3$, $C_3$) is, but rather how high it is relative to the average similarities of ($C_1$, $C_3$), ($C_2$, $C_3$), and ($C_3$, $C_4$).

So given our four categories, this leaves us with four *average similarities* per category; one for same-category documents and three for cross-category documents.

Here's what we're going to do for each category. Using each category's four *average similarity* values, we'll calculate the mean similarity differences between the cross-category *average similarities* and the same-category *average similarity*. More formally:

$$Mean\ difference = \frac{\sum_{j=1, j\neq 3}^{4} (S(C_3, C_3) - S(C_3, C_j))}{3}$$

Calculating category 3's mean similarity difference. S() denotes the cosine similarity of the two categories. Note how j=3 is being skipped as the resulting subtraction would be redundant.


This might be more recognizable in categories such as *comp.os.ms-windows.misc* and *comp.windows.x* than in *comp.os.ms-windows.misc* and *soc.religion.christian*, for example.

The result of the evaluation can then be summarized as follows:

| Category | Mean Difference | Avg. Similarity |
|---|---|---|
| $C_1$ | 0.2 | 0.42 |
| $C_2$ | 0.17 | 0.33 |
| $C_3$ | 0.13 | 0.51 |
| $C_4$ | 0.15 | 0.4 |

The average similarity shown is the average similarity of same-category documents. A good model would be one that gives high **mean difference** and **average similarity** values.

Representing the results in such a compact form makes it more efficient to train multiple models with different hyperparameters and comparing their performance.

Now let's take a look at how we can code this. First, we create a dictionary, mapping lists of document pairs to the category pairs they belong to. Given the large number of resulting pairs, we need to limit that number somewhat in order to perform our evaluation in a reasonable amount of time. To do that we randomly sample 500 document pairs from each dictionary entry and calculate the cosine similarity for each of the document pairs.

*Note that 500 is an arbitrary choice. Ideally the larger the sample the more accurate the representation.*

This results in similarity matrices such as the one we looked at earlier. We finally save those matrices as lists in a new dictionary where each list is mapped, again, to the category pair it represents.


```
cat_id = {id:cat for id, cat in enumerate(categories)} # Give each category a numerical
test_doc_pairs = {tuple(sorted([id,id2])):[] for id in cat_id for id2 in cat_id}
for pair_id in test_doc_pairs:
    # Create same-category doc pairs, e.g. (C3, C3)
    if pair_id[0] == pair_id[1]:
        test_doc_pairs[pair_id] = [(doc, cat_dict_test_clean[cat_id[pair_id[0]]][i]) fo
    #Create cross-category doc pairs, e.g. (C3, C4)
    else:
        test_doc_pairs[pair_id] = [(doc, doc2) for doc in list(cat_dict_test_clean[cat_
similarities_test = {pair_id:[] for pair_id in test_doc_pairs}
for id in cat_id:
    for id2 in cat_id:
        similarities_test[tuple(sorted([id, id2]))] = [model.docvecs.similarity_unseen_
```

**category_pair_dictionary.py** hosted with ❤️ by **GitHub** **view raw**

Generating the similarity matrices

The next step is to use those similari 401 5 te the compact representations we discussed above. We first go through all category pairs we have. If we find a same-category pair we save its *average similarity* to be used later when calculating the *mean difference*. For cross-category pairs, we simply save their *average similarities* in a list.

The final step is to calculate the *mean difference* using the list of *average similarities* and the previously saved, same-category *average similarity*. This number, along with the same-

```
    avg_vec_sims = []
    for pair_id, pair_sim_list in similarities_test.items():
        if id in pair_id:
            if pair_id[0] == pair_id[1]:
                main_avg_vec_sim = sum(pair_sim_list)/len(pair_sim_list)
            else:
                avg_vec_sims.append(sum(pair_sim_list)/len(pair_sim_list))
    mean_diff = sum([main_avg_vec_sim - x for x in avg_vec_sims]) / (len(categories)-1)
    print("Category: {}".format(cat_id[id]))
    print("\tMean difference: {:.2}, Same-category average similarity: {:.2}".format(me
```

**calculating_avgs.py** hosted with ❤️ by **GitHub** **view raw**

## Summary

So that may have been a lot to digest. Here's a summary of everything we talked about in this article:

- We took a look at *doc2vec* a method commonly used to generate vector representations of t documents.
- We saw how to prepare our data and train a doc2vec model in Python using Gensim.
- We saw how useful it can be to visualize our vectors using TensorFlow's projector.
- Finally, we discussed one possible way of evaluating a doc2vec model that allows for efficie comparisons of multiple models.

*Thanks for sticking around! Would love to hear your feedback and answer any questions you may have.*

## References

[1] Q. V. Le and T. Mikolov, Distributed Representations of Sentences and Documents (2014)
[2] T. Mikolov, K. Chen, G. Corrado, and J. Dean, Efficient Estimation of Word Representations