Corpora and Vector Spaces

This tutorial is available as a Jupyter Notebook **here**.

Don't forget to set

```
>>> import logging
>>> logging.basicConfig(format='%(asctime)s : %(levelname)s : %(message)s', level=logging.INFO)
```

if you want to see logging events.

From Strings to Vectors

This time, let's start from documents represented as strings:

```
>>> from gensim import corpora
>>>
>>> documents = ["Human machine interface for lab abc computer applications",
>>>              "A survey of user opinion of computer system response time",
>>>              "The EPS user interface management system",
>>>              "System and human system engineering testing of EPS",
>>>              "Relation of user perceived response time to error measurement",
>>>              "The generation of random binary unordered trees",
>>>              "The intersection graph of paths in trees",
>>>              "Graph minors IV Widths of trees and well quasi ordering",
>>>              "Graph minors A survey"]
```

This is a tiny corpus of nine documents, each consisting of only a single sentence.

First, let's tokenize the documents, remove common words (using a toy stoplist) as well as words that only appear once in the corpus:

```
>>> # remove common words and tokenize
>>> stoplist = set('for a of the and to in'.split())
>>> texts = [[word for word in document.lower().split() if word not in stoplist]
>>>          for document in documents]
>>>
>>> # remove words that appear only once
>>> from collections import defaultdict
>>> frequency = defaultdict(int)
>>> for text in texts:
>>>     for token in text:
>>>         frequency[token] += 1
>>>
>>> texts = [[token for token in text if frequency[token] > 1]
>>>          for text in texts]
>>>
>>> from pprint import pprint  # pretty-printer
>>> pprint(texts)
[['human', 'interface', 'computer'],
 ['survey', 'user', 'computer', 'system', 'response', 'time'],
 ['eps', 'user', 'interface', 'system'],
 ['system', 'human', 'system', 'eps'],
 ['user', 'response', 'time'],
 ['trees'],
 ['graph', 'trees'],
 ['graph', 'minors', 'trees'],
 ['graph', 'minors', 'survey']]
```

Your way of processing the documents will likely vary; here, I only split on whitespace to tokenize, followed by lowercasing each word. In fact, I use this particular (simplistic and inefficient) setup to mimic the experiment done in Deerwester et al.'s original LSA article [1].

The ways to process documents are so varied and application- and language-dependent that I decided to *not* constrain them by any interface. Instead, a document is represented by the features extracted from it, not by its "surface" string form: how you get to the features is up to you. Below I describe one common, general-purpose approach (called *bag-of-words*), but keep in mind that different application domains call for different features, and, as always, it's **garbage in, garbage out**...

To convert documents to vectors, we'll use a document representation called **bag-of-words**. In this representation, each document is represented by one vector where each vector element represents a question-answer pair, in the style of:

“How many times does the word *system* appear in the document? Once.”

It is advantageous to represent the questions only by their (integer) ids. The mapping between the questions and ids is called a dictionary:

```
>>> dictionary = corpora.Dictionary(texts)
>>> dictionary.save('/tmp/deerwester.dict')  # store the dictionary, for future reference
>>> print(dictionary)
Dictionary(12 unique tokens)
```

Here we assigned a unique integer id to all words appearing in the corpus with the `gensim.corpora.dictionary.Dictionary` class. This sweeps across the texts, collecting word counts and relevant statistics. In the end, we see there are twelve distinct words in the processed corpus, which means each document will be represented by twelve numbers (ie., by a 12-D vector). To see the mapping between words and their ids:

```
>>> print(dictionary.token2id)
{'minors': 11, 'graph': 10, 'system': 5, 'trees': 9, 'eps': 8, 'computer': 0,
'survey': 4, 'user': 7, 'human': 1, 'time': 6, 'interface': 2, 'response': 3}
```

To actually convert tokenized documents to vectors:

```
>>> new_doc = "Human computer interaction"
>>> new_vec = dictionary.doc2bow(new_doc.lower().split())
>>> print(new_vec)  # the word "interaction" does not appear in the dictionary and is ignored
[(0, 1), (1, 1)]
```

The function `doc2bow()` simply counts the number of occurrences of each distinct word, converts the word to its integer word id and returns the result as a sparse vector. The sparse vector `[(0, 1), (1, 1)]` therefore reads: in the document *“Human computer interaction”*, the words*computer* (id 0) and *human* (id 1) appear once; the other ten dictionary words appear (implicitly) zero times.

```
>>> corpus = [dictionary.doc2bow(text) for text in texts]
>>> corpora.MmCorpus.serialize('/tmp/deerwester.mm', corpus)  # store to disk, for later use
>>> print(corpus)
[(0, 1), (1, 1), (2, 1)]
[(0, 1), (3, 1), (4, 1), (5, 1), (6, 1), (7, 1)]
[(2, 1), (5, 1), (7, 1), (8, 1)]
[(1, 1), (5, 2), (8, 1)]
[(3, 1), (6, 1), (7, 1)]
[(9, 1)]
[(9, 1), (10, 1)]
[(9, 1), (10, 1), (11, 1)]
[(4, 1), (10, 1), (11, 1)]
```

By now it should be clear that the vector feature with `id=10` stands for the question “How many times does the word *graph* appear in the document?” and that the answer is “zero” for the first six documents and “one” for the remaining three. As a matter of fact, we have arrived at exactly the same corpus of vectors as in the **Quick Example**.

Corpus Streaming – One Document at a Time

Note that *corpus* above resides fully in memory, as a plain Python list. In this simple example, it doesn’t matter much, but just to make things clear, let’s assume there are millions of documents in the corpus. Storing all of them in RAM won’t do. Instead, let’s assume the documents are stored in a file on disk, one document per line. Gensim only requires that a corpus must be able to return one document vector at a time:

```
>>> class MyCorpus(object):
>>>     def __iter__(self):
>>>         for line in open('mycorpus.txt'):
>>>             # assume there's one document per line, tokens separated by whitespace
>>>             yield dictionary.doc2bow(line.lower().split())
```

Download the sample **mycorpus.txt file here**. The assumption that each document occupies one line in a single file is not important; you can mold the *__iter__* function to fit your input format, whatever it is. Walking directories, parsing XML, accessing network... Just parse your input to retrieve a clean list of tokens in each document, then convert the tokens via a dictionary to their ids and yield the resulting sparse vector inside*__iter__*.

```
>>> corpus_memory_friendly = MyCorpus()  # doesn't load the corpus into memory!
>>> print(corpus_memory_friendly)
<__main__.MyCorpus object at 0x10d5690>
```

Corpus is now an object. We didn’t define any way to print it, so *print* just outputs address of the object in memory. Not very useful. To see the constituent vectors, let’s iterate over the corpus and print each document vector (one at a time):

```
>>> for vector in corpus_memory_friendly:  # load one vector into memory at a time
```

```
...     print(vector)
[(0, 1), (1, 1), (2, 1)]
[(0, 1), (3, 1), (4, 1), (5, 1), (6, 1), (7, 1)]
[(2, 1), (5, 1), (7, 1), (8, 1)]
[(1, 1), (5, 2), (8, 1)]
[(3, 1), (6, 1), (7, 1)]
[(9, 1)]
[(9, 1), (10, 1)]
[(9, 1), (10, 1), (11, 1)]
[(4, 1), (10, 1), (11, 1)]
```

Although the output is the same as for the plain Python list, the corpus is now much more memory friendly, because at most one vector resides in RAM at a time. Your corpus can now be as large as you want.

Similarly, to construct the dictionary without loading all texts into memory:

```
>>> from six import iteritems
>>> # collect statistics about all tokens
>>> dictionary = corpora.Dictionary(line.lower().split() for line in open('mycorpus.txt'))
>>> # remove stop words and words that appear only once
>>> stop_ids = [dictionary.token2id[stopword] for stopword in stoplist
>>>             if stopword in dictionary.token2id]
>>> once_ids = [tokenid for tokenid, docfreq in iteritems(dictionary.dfs) if docfreq == 1]
>>> dictionary.filter_tokens(stop_ids + once_ids)  # remove stop words and words that appear only once
>>> dictionary.compactify()  # remove gaps in id sequence after words that were removed
>>> print(dictionary)
Dictionary(12 unique tokens)
```

And that is all there is to it! At least as far as bag-of-words representation is concerned. Of course, what we do with such corpus is another question; it is not at all clear how counting the frequency of distinct words could be useful. As it turns out, it isn't, and we will need to apply a transformation on this simple representation first, before we can use it to compute any meaningful document vs. document similarities. Transformations are covered in the ***next tutorial***, but before that, let's briefly turn our attention to *corpus persistency*.

Corpus Formats

There exist several file formats for serializing a Vector Space corpus (~sequence of vectors) to disk. *Gensim* implements them via the *streaming corpus interface* mentioned earlier: documents are read from (resp. stored to) disk in a lazy fashion, one document at a time, without the whole corpus being read into main memory at once.

One of the more notable file formats is the **Market Matrix format**. To save a corpus in the Matrix Market format:

```
>>> # create a toy corpus of 2 documents, as a plain Python list
>>> corpus = [[(1, 0.5)], []]  # make one document empty, for the heck of it
>>>
>>> corpora.MmCorpus.serialize('/tmp/corpus.mm', corpus)
```

Other formats include **Joachim's SVMlight format**, **Blei's LDA-C format** and **GibbsLDA++ format**.

```
>>> corpora.SvmLightCorpus.serialize('/tmp/corpus.svmlight', corpus)
>>> corpora.BleiCorpus.serialize('/tmp/corpus.lda-c', corpus)
>>> corpora.LowCorpus.serialize('/tmp/corpus.low', corpus)
```

Conversely, to load a corpus iterator from a Matrix Market file:

```
>>> corpus = corpora.MmCorpus('/tmp/corpus.mm')
```

Corpus objects are streams, so typically you won't be able to print them directly:

```
>>> print(corpus)
MmCorpus(2 documents, 2 features, 1 non-zero entries)
```

Instead, to view the contents of a corpus:

```
>>> # one way of printing a corpus: load it entirely into memory
>>> print(list(corpus))  # calling list() will convert any sequence to a plain Python list
[[(1, 0.5)], []]
```

or

```
>>> # another way of doing it: print one document at a time, making use of the streaming interface
>>> for doc in corpus:
...     print(doc)
[(1, 0.5)]
```

```
[]
```

The second way is obviously more memory-friendly, but for testing and development purposes, nothing beats the simplicity of calling`list(corpus)`.

To save the same Matrix Market document stream in Blei's LDA-C format,

```
>>> corpora.BleiCorpus.serialize('/tmp/corpus.lda-c', corpus)
```

In this way, *gensim* can also be used as a memory-efficient **I/O format conversion tool**: just load a document stream using one format and immediately save it in another format. Adding new formats is dead easy, check out the **code for the SVMlight corpus** for an example.

Compatibility with NumPy and SciPy

Gensim also contains **efficient utility functions** to help converting from/to numpy matrices:

```
>>> import gensim
>>> import numpy as np
>>> numpy_matrix = np.random.randint(10, size=[5,2])  # random matrix as an example
>>> corpus = gensim.matutils.Dense2Corpus(numpy_matrix)
>>> numpy_matrix = gensim.matutils.corpus2dense(corpus, num_terms=number_of_corpus_features)
```

and from/to *scipy.sparse* matrices:

```
>>> import scipy.sparse
>>> scipy_sparse_matrix = scipy.sparse.random(5,2)  # random sparse matrix as example
>>> corpus = gensim.matutils.Sparse2Corpus(scipy_sparse_matrix)
>>> scipy_csc_matrix = gensim.matutils.corpus2csc(corpus)
```

For a complete reference (Want to prune the dictionary to a smaller size? Optimize converting between corpora and NumPy/SciPy arrays?), see the ***API documentation***. Or continue to the next tutorial on ***Topics and Transformations***.

[1] This is the same corpus as used in **Deerwester et al. (1990): Indexing by Latent Semantic Analysis**, Table 2.