Similarity Queries

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.

Similarity interface

In the previous tutorials on **Corpora and Vector Spaces** and **Topics and Transformations**, we covered what it means to create a corpus in the Vector Space Model and how to transform it between different vector spaces. A common reason for such a charade is that we want to determine **similarity between pairs of documents**, or the **similarity between a specific document and a set of other documents** (such as a user query vs. indexed documents).

To show how this can be done in gensim, let us consider the same corpus as in the previous examples (which really originally comes from Deerwester et al.'s **"Indexing by Latent Semantic Analysis"** seminal 1990 article):

```
>>> from gensim import corpora
>>> dictionary = corpora.Dictionary.load('/tmp/deerwester.dict')
>>> corpus = corpora.MmCorpus('/tmp/deerwester.mm')  # comes from the first tutorial, "From strings to vectors"
>>> print(corpus)
MmCorpus(9 documents, 12 features, 28 non-zero entries)
```

To follow Deerwester's example, we first use this tiny corpus to define a 2-dimensional LSI space:

```
>>> from gensim import models
>>> lsi = models.LsiModel(corpus, id2word=dictionary, num_topics=2)
```

Now suppose a user typed in the query *"Human computer interaction"*. We would like to sort our nine corpus documents in decreasing order of relevance to this query. Unlike modern search engines, here we only concentrate on a single aspect of possible similarities—on apparent semantic relatedness of their texts (words). No hyperlinks, no random-walk static ranks, just a semantic extension over the boolean keyword match:

```
>>> doc = "Human computer interaction"
>>> vec_bow = dictionary.doc2bow(doc.lower().split())
>>> vec_lsi = lsi[vec_bow]  # convert the query to LSI space
>>> print(vec_lsi)
[(0, -0.461821), (1, 0.070028)]
```

In addition, we will be considering **cosine similarity** to determine the similarity of two vectors. Cosine similarity is a standard measure in Vector Space Modeling, but wherever the vectors represent probability distributions, **different similarity measures** may be more appropriate.

Initializing query structures

To prepare for similarity queries, we need to enter all documents which we want to compare against subsequent queries. In our case, they are the same nine documents used for training LSI, converted to 2-D LSA space. But that's only incidental, we might also be indexing a different corpus altogether.

```
>>> index = similarities.MatrixSimilarity(lsi[corpus])  # transform corpus to LSI space and index it
```

Warning

The class `similarities.MatrixSimilarity` is only appropriate when the whole set of vectors fits into memory. For example, a corpus of one million documents would require 2GB of RAM in a 256-dimensional LSI space, when used with this class.

Without 2GB of free RAM, you would need to use the `similarities.Similarity` class. This class operates in fixed memory, by splitting the index across multiple files on disk, called shards. It uses `similarities.MatrixSimilarity` and `similarities.SparseMatrixSimilarity` internally, so it is still fast, although slightly more complex.

Index persistency is handled via the standard `save()` and `load()` functions:

```
>>> index.save('/tmp/deerwester.index')
>>> index = similarities.MatrixSimilarity.load('/tmp/deerwester.index')
```

This is true for all similarity indexing classes (`similarities.Similarity`, `similarities.MatrixSimilarity` and `similarities.SparseMatrixSimilarity`). Also in the following, *index* can be an object of any of these. When in doubt, use `similarities.Similarity`, as it is the most scalable version, and it also supports adding more documents to the index later.

Performing queries

To obtain similarities of our query document against the nine indexed documents:

```
>>> sims = index[vec_lsi]  # perform a similarity query against the corpus
>>> print(list(enumerate(sims)))  # print (document_number, document_similarity) 2-tuples
[(0, 0.99809301), (1, 0.93748635), (2, 0.99844527), (3, 0.9865886), (4, 0.90755945),
(5, -0.12416792), (6, -0.1063926), (7, -0.098794639), (8, 0.05004178)]
```

Cosine measure returns similarities in the range *<-1, 1>* (the greater, the more similar), so that the first document has a score of 0.99809301 etc.

With some standard Python magic we sort these similarities into descending order, and obtain the final answer to the query *"Human computer interaction"*:

```
>>> sims = sorted(enumerate(sims), key=lambda item: -item[1])
>>> print(sims)  # print sorted (document number, similarity score) 2-tuples
[(2, 0.99844527), # The EPS user interface management system
(0, 0.99809301), # Human machine interface for lab abc computer applications
(3, 0.9865886), # System and human system engineering testing of EPS
(1, 0.93748635), # A survey of user opinion of computer system response time
(4, 0.90755945), # Relation of user perceived response time to error measurement
(8, 0.050041795), # Graph minors A survey
(7, -0.098794639), # Graph minors IV Widths of trees and well quasi ordering
(6, -0.1063926), # The intersection graph of paths in trees
(5, -0.12416792)] # The generation of random binary unordered trees
```

(I added the original documents in their "string form" to the output comments, to improve clarity.)

The thing to note here is that documents no. 2 (`"The EPS user interface management system"`) and 4 (`"Relation of user perceived response time to error measurement"`) would never be returned by a standard boolean fulltext search, because they do not share any common words with `"Human computer interaction"`. However, after applying LSI, we can observe that both of them received quite high similarity scores (no. 2 is actually the most similar!), which corresponds better to our intuition of them sharing a "computer-human" related topic with the query. In fact, this semantic generalization is the reason why we apply transformations and do topic modelling in the first place.

Where next?

Congratulations, you have finished the tutorials – now you know how gensim works :-) To delve into more details, you can browse through the **API documentation**, see the **Wikipedia experiments** or perhaps check out **distributed computing** in *gensim*.

Gensim is a fairly mature package that has been used successfully by many individuals and companies, both for rapid prototyping and in production. That doesn't mean it's perfect though:

- there are parts that could be implemented more efficiently (in C, for example), or make better use of parallelism (multiple machines cores)
- new algorithms are published all the time; help gensim keep up by **discussing them** and **contributing code**
- your **feedback is most welcome** and appreciated (and it's not just the code!): **idea contributions**, **bug reports** or just consider contributing **user stories and general questions**.

Gensim has no ambition to become an all-encompassing framework, across all NLP (or even Machine Learning) subfields. Its mission is to help NLP practitioners try out popular topic modelling algorithms on large datasets easily, and to facilitate prototyping of new algorithms for researchers.