# gensim Documentation

***Release 0.8.6***

**Radim Řehůřek**

**Nov 14, 2017**

# Contents

# CHAPTER 1

## Quick Reference Example

```
>>> from gensim import corpora, models, similarities
>>>
>>> # Load corpus iterator from a Matrix Market file on disk.
>>> corpus = corpora.MmCorpus('/path/to/corpus.mm')
>>>
>>> # Initialize a transformation (Latent Semantic Indexing with 200 latent␣
↪dimensions).
>>> lsi = models.LsiModel(corpus, num_topics=200)
>>>
>>> # Convert another corpus to the latent space and index it.
>>> index = similarities.MatrixSimilarity(lsi[another_corpus])
>>>
>>> # determine similarity of a query document against each document in the index
>>> sims = index[query]
```

**What's new?**

- 15 Sep 2012: release 0.8.6 : added the hashing trick to allow online changes to the vocabulary; fixed parallel lemmatization + other minor improvements
- 22 Jul 2012: release 0.8.5 : better Wikipedia parsing, faster similarity queries, maintenance fixes
- 30 Apr 2012: William Bert's interview with me
- 9 Mar 2012: release 0.8.4: new model Hierarchical Dirichlet Process (full CHANGELOG)

## 1.1 Introduction

Gensim is a *free* Python framework designed to automatically extract semantic topics from documents, as efficiently (computer-wise) and painlessly (human-wise) as possible.

Gensim aims at processing raw, unstructured digital texts ("*plain text*"). The algorithms in *gensim*, such as **Latent Semantic Analysis**, **Latent Dirichlet Allocation** or **Random Projections**, discover semantic structure of documents, by examining word statistical co-occurrence patterns within a corpus of training documents. These algorithms are unsupervised, which means no human input is necessary – you only need a corpus of plain text documents.

Once these statistical patterns are found, any plain text documents can be succinctly expressed in the new, semantic representation, and queried for topical similarity against other documents.

---

**Note:** If the previous paragraphs left you confused, you can read more about the Vector Space Model and unsupervised document analysis on Wikipedia.

---

### 1.1.1 Features

- **Memory independence** – there is no need for the whole training corpus to reside fully in RAM at any one time (can process large, web-scale corpora).
- Efficient implementations for several popular vector space algorithms, including **Tf-Idf**, distributed incremental **Latent Semantic Analysis**, distributed incremental **Latent Dirichlet Allocation (LDA)** or **Random Projection**; adding new ones is easy (really!).
- I/O wrappers and converters around **several popular data formats**.
- **Similarity queries** for documents in their semantic representation.

Creation of *gensim* was motivated by a perceived lack of available, scalable software frameworks that realize topic modelling, and/or their overwhelming internal complexity (hail java!). You can read more about the motivation in our LREC 2010 workshop paper. If you want to cite *gensim* in your own work, please refer to that article (BibTeX).

You're welcome to share your results on the mailing list, so others can learn from your success :)

The **principal design objectives** behind *gensim* are:

1. Straightforward interfaces and low API learning curve for developers. Good for prototyping.
2. Memory independence with respect to the size of the input corpus; all intermediate steps and algorithms operate in a streaming fashion, accessing one document at a time.

**See also:**

If you're interested in document indexing/similarity retrieval, I also maintain a higher-level package of document similarity server. It uses gensim internally.

### 1.1.2 Availability

Gensim is licensed under the OSI-approved GNU LPGL license and can be downloaded either from its github repository or from the Python Package Index.

**See also:**

See the *install* page for more info on *gensim* deployment.

### 1.1.3 Core concepts

The whole gensim package revolves around the concepts of *corpus*, *vector* and *model*.

**Corpus** A collection of digital documents. This collection is used to automatically infer structure of the documents, their topics etc. For this reason, the collection is also called a *training corpus*. The inferred latent structure can be later used to assign topics to new documents, which did not appear in the training corpus. No human intervention (such as tagging the documents by hand, or creating other metadata) is required.

**Vector** In the Vector Space Model (VSM), each document is represented by an array of features. For example, a single feature may be thought of as a question-answer pair:

1. How many times does the word *splonge* appear in the document? Zero.
2. How many paragraphs does the document consist of? Two.
3. How many fonts does the document use? Five.

The question is usually represented only by its integer id (such as *1*, *2* and *3* here), so that the representation of this document becomes a series of pairs like `(1, 0.0), (2, 2.0), (3, 5.0)`. If we know all the questions in advance, we may leave them implicit and simply write `(0.0, 2.0, 5.0)`. This sequence of answers can be thought of as a high-dimensional (in this case 3-dimensional) *vector*. For practical purposes, only questions to which the answer is (or can be converted to) a single real number are allowed.

The questions are the same for each document, so that looking at two vectors (representing two documents), we will hopefully be able to make conclusions such as "The numbers in these two vectors are very similar, and therefore the original documents must be similar, too". Of course, whether such conclusions correspond to reality depends on how well we picked our questions.

**Sparse vector** Typically, the answer to most questions will be `0.0`. To save space, we omit them from the document's representation, and write only `(2, 2.0), (3, 5.0)` (note the missing `(1, 0.0)`). Since the set of all questions is known in advance, all the missing features in a sparse representation of a document can be unambiguously resolved to zero, `0.0`.

Gensim is specific in that it doesn't prescribe any specific corpus format; a corpus is anything that, when iterated over, successively yields these sparse vectors. For example, *set([(2, 2.0), (3, 5.0)], ([0, -1.0], [3, -1.0]))* is a trivial corpus of two documents, each with two non-zero *feature-answer* pairs.

**Model** For our purposes, a model is a transformation from one document representation to another (or, in other words, from one vector space to another). Both the initial and target representations are still vectors – they only differ in what the questions and answers are. The transformation is automatically learned from the traning *corpus*, without human supervision, and in hopes that the final document representation will be more compact and more useful: with similar documents having similar representations.

**See also:**

For some examples on how this works out in code, go to *tutorials*.

## 1.2 Installation

### 1.2.1 Quick install

Run in your terminal:

```
sudo easy_install -U gensim
```

In case that fails, or you don't know what "terminal" means, read on.

### 1.2.2 Dependencies

Gensim is known to run on Linux, Windows and Mac OS X and should run on any other platform that supports Python 2.5 and NumPy. Gensim depends on the following software:

- 3.0 > Python >= 2.5. Tested with versions 2.5, 2.6 and 2.7.
- NumPy >= 1.3. Tested with version 1.6.1rc2, 1.5.0rc1, 1.4.0, 1.3.0, 1.3.0rc2.
- SciPy >= 0.7. Tested with version 0.9.0, 0.8.0, 0.8.0b1, 0.7.1, 0.7.0.

**Windows users** are well advised to try the Enthought distribution, which conveniently includes Python&NumPy&SciPy in a single bundle, and is free for academic use.

### 1.2.3 Install Python and *easy_install*

Check what version of Python you have with:

```
python --version
```

You can download Python from http://python.org/download.

**Note:** Gensim requires Python 2.5 or greater and will not run under earlier versions.

Next, install the easy_install utility, which will make installing other Python programs easier.

### 1.2.4 Install SciPy & NumPy

These are quite popular Python packages, so chances are there are pre-built binary distributions available for your platform. You can try installing from source using easy_install:

```
sudo easy_install numpy
sudo easy_install scipy
```

If that doesn't work or if you'd rather install using a binary package, consult http://www.scipy.org/Download.

### 1.2.5 Install *gensim*

You can now install (or upgrade) *gensim* with:

```
sudo easy_install --upgrade gensim
```

That's it! Congratulations, you can proceed to the *tutorials*.

---

If you also want to run the algorithms over a cluster of computers, in *Distributed Computing*, you should install with:

```
sudo easy_install gensim[distributed]
```

The optional *distributed* feature installs Pyro (PYthon Remote Objects). If you don't know what distributed computing means, you can ignore it: *gensim* will work fine for you anyway. This optional extension can also be installed separately later with:

```
sudo easy_install Pyro4
```

There are also alternative routes to install:

1. If you have downloaded and unzipped the tar.gz source for *gensim* (or you're installing *gensim* from github), you can run:

   ```
   sudo python setup.py install
   ```

   to install *gensim* into your `site-packages` folder.

2. If you wish to make local changes to the *gensim* code (*gensim* is, after all, a package which targets research prototyping and modifications), a preferred way may be installing with:

   ```
   sudo python setup.py develop
   ```

   This will only place a symlink into your `site-packages` directory. The actual files will stay wherever you unpacked them.

3. If you don't have root priviledges (or just don't want to put the package into your `site-packages`), simply unpack the source package somewhere and that's it! No compilation or installation needed. Just don't forget to set your PYTHONPATH (or modify `sys.path`), so that Python can find the unpacked package when importing.

### 1.2.6 Testing *gensim*

To test the package, unzip the tar.gz source and run:

```
python setup.py test
```

### 1.2.7 Contact

Use the gensim discussion group for any questions and troubleshooting. For private enquiries, you can also send me an email to the address at the bottom of this page.

## 1.3 Tutorials

The tutorials are organized as a series of examples that highlight various features of *gensim*. It is assumed that the reader is familiar with the Python language and has read the *Introduction*.

The examples are divided into parts on:

### 1.3.1 Corpora and Vector Spaces

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, models, similarities
>>>
>>> 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
>>> all_tokens = sum(texts, [])
>>> tokens_once = set(word for word in set(all_tokens) if all_tokens.count(word) == 1)
>>> texts = [[word for word in text if word not in tokens_once]
>>>          for text in texts]
>>>
>>> print 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 mimick 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."

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

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 occurences 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:

```
>>> # 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 dictionary.dfs.iteritems() 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:

```
>>> from gensim import corpora
>>> # 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.

---

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

### 1.3.2 Topics and Transformations

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.

#### Transformation interface

In the previous tutorial on *Corpora and Vector Spaces*, we created a corpus of documents represented as a stream of vectors. To continue, let's fire up gensim and use that corpus:

```
>>> from gensim import corpora, models, similarities
>>> dictionary = corpora.Dictionary.load('/tmp/deerwester.dict')
>>> corpus = corpora.MmCorpus('/tmp/deerwester.mm')
>>> print corpus
MmCorpus(9 documents, 12 features, 28 non-zero entries)
```

In this tutorial, I will show how to transform documents from one vector representation into another. This process serves two goals:

1. To bring out hidden structure in the corpus, discover relationships between words and use them to describe the documents in a new and (hopefully) more semantic way.
2. To make the document representation more compact. This both improves efficiency (new representation consumes less resources) and efficacy (marginal data trends are ignored, noise-reduction).

#### Creating a transformation

The transformations are standard Python objects, typically initialized by means of a *training corpus*:

```
>>> tfidf = models.TfidfModel(corpus) # step 1 -- initialize a model
```

We used our old corpus from tutorial 1 to initialize (train) the transformation model. Different transformations may require different initialization parameters; in case of TfIdf, the "training" consists simply of going through the supplied corpus once and computing document frequencies of all its features. Training other models, such as Latent Semantic Analysis or Latent Dirichlet Allocation, is much more involved and, consequently, takes much more time.

---

**Note:** Transformations always convert between two specific vector spaces. The same vector space (= the same set of feature ids) must be used for training as well as for subsequent vector transformations. Failure to use the same input feature space, such as applying a different string preprocessing, using different feature ids, or using bag-of-words input vectors where TfIdf vectors are expected, will result in feature mismatch during transformation calls and consequently in either garbage output and/or runtime exceptions.

---

#### Transforming vectors

From now on, `tfidf` is treated as a read-only object that can be used to convert any vector from the old representation (bag-of-words integer counts) to the new representation (TfIdf real-valued weights):

```
>>> doc_bow = [(0, 1), (1, 1)]
>>> print tfidf[doc_bow] # step 2 -- use the model to transform vectors
[(0, 0.70710678), (1, 0.70710678)]
```

Or to apply a transformation to a whole corpus:

```
>>> corpus_tfidf = tfidf[corpus]
>>> for doc in corpus_tfidf:
>>>     print doc
[(0, 0.57735026918962573), (1, 0.57735026918962573), (2, 0.57735026918962573)]
[(0, 0.44424552527467476), (3, 0.44424552527467476), (4, 0.44424552527467476), (5, 0.
↪32448702061385548), (6, 0.44424552527467476), (7, 0.32448702061385548)]
[(2, 0.5710059809418182), (5, 0.41707573620227772), (7, 0.41707573620227772), (8, 0.
↪5710059809418182)]
[(1, 0.49182558987264147), (5, 0.71848116070837686), (8, 0.49182558987264147)]
[(3, 0.62825804686700459), (6, 0.62825804686700459), (7, 0.45889394536615247)]
[(9, 1.0)]
[(9, 0.70710678118654746), (10, 0.70710678118654746)]
[(9, 0.50804290089167492), (10, 0.50804290089167492), (11, 0.69554641952003704)]
[(4, 0.62825804686700459), (10, 0.45889394536615247), (11, 0.62825804686700459)]
```

In this particular case, we are transforming the same corpus that we used for training, but this is only incidental. Once the transformation model has been initialized, it can be used on any vectors (provided they come from the same vector space, of course), even if they were not used in the training corpus at all. This is achieved by a process called folding-in for LSA, by topic inference for LDA etc.

---

**Note:** Calling `model[corpus]` only creates a wrapper around the old `corpus` document stream – actual conversions are done on-the-fly, during document iteration. We cannot convert the entire corpus at the time of calling `corpus_transformed = model[corpus]`, because that would mean storing the result in main memory, and that contradicts gensim's objective of memory-indepedence. If you will be iterating over the transformed `corpus_transformed` multiple times, and the transformation is costly, *serialize the resulting corpus to disk first* and continue using that.

---

Transformations can also be serialized, one on top of another, in a sort of chain:

```
>>> lsi = models.LsiModel(corpus_tfidf, id2word=dictionary, num_topics=2) #␣
↪initialize an LSI transformation
>>> corpus_lsi = lsi[corpus_tfidf] # create a double wrapper over the original␣
↪corpus: bow->tfidf->fold-in-lsi
```

Here we transformed our Tf-Idf corpus via Latent Semantic Indexing into a latent 2-D space (2-D because we set `num_topics=2`). Now you're probably wondering: what do these two latent dimensions stand for? Let's inspect with `models.LsiModel.print_topics()`:

```
>>> lsi.print_topics(2)
topic #0(1.594): -0.703*"trees" + -0.538*"graph" + -0.402*"minors" + -0.187*"survey"␣
↪+ -0.061*"system" + -0.060*"response" + -0.060*"time" + -0.058*"user" + -0.049*
↪"computer" + -0.035*"interface"
topic #1(1.476): -0.460*"system" + -0.373*"user" + -0.332*"eps" + -0.328*"interface"␣
↪+ -0.320*"response" + -0.320*"time" + -0.293*"computer" + -0.280*"human" + -0.171*
↪"survey" + 0.161*"trees"
```

(the topics are printed to log – see the note at the top of this page about activating logging)

It appears that according to LSI, "trees", "graph" and "minors" are all related words (and contribute the most to the direction of the first topic), while the second topic practically concerns itself with all the other words. As expected,

the first five documents are more strongly related to the second topic while the remaining four documents to the first topic:

```
>>> for doc in corpus_lsi: # both bow->tfidf and tfidf->lsi transformations are␣
↪actually executed here, on the fly
>>>     print doc
[(0, -0.066), (1, 0.520)] # "Human machine interface for lab abc computer applications
↪"
[(0, -0.197), (1, 0.761)] # "A survey of user opinion of computer system response time
↪"
[(0, -0.090), (1, 0.724)] # "The EPS user interface management system"
[(0, -0.076), (1, 0.632)] # "System and human system engineering testing of EPS"
[(0, -0.102), (1, 0.574)] # "Relation of user perceived response time to error␣
↪measurement"
[(0, -0.703), (1, -0.161)] # "The generation of random binary unordered trees"
[(0, -0.877), (1, -0.168)] # "The intersection graph of paths in trees"
[(0, -0.910), (1, -0.141)] # "Graph minors IV Widths of trees and well quasi ordering"
[(0, -0.617), (1, 0.054)] # "Graph minors A survey"
```

Model persistency is achieved with the `save()` and `load()` functions:

```
>>> lsi.save('/tmp/model.lsi') # same for tfidf, lda, ...
>>> lsi = models.LsiModel.load('/tmp/model.lsi')
```

The next question might be: just how exactly similar are those documents to each other? Is there a way to formalize the similarity, so that for a given input document, we can order some other set of documents according to their similarity? Similarity queries are covered in the *next tutorial*.

### Available transformations

Gensim implements several popular Vector Space Model algorithms:

- Term Frequency * Inverse Document Frequency, Tf-Idf expects a bag-of-words (integer values) training corpus during initialization. During transformation, it will take a vector and return another vector of the same dimensionality, except that features which were rare in the training corpus will have their value increased. It therefore converts integer-valued vectors into real-valued ones, while leaving the number of dimensions intact. It can also optionally normalize the resulting vectors to (Euclidean) unit length.

  ```
  >>> model = tfidfmodel.TfidfModel(bow_corpus, normalize=True)
  ```

- Latent Semantic Indexing, LSI (or sometimes LSA) transforms documents from either bag-of-words or (preferrably) TfIdf-weighted space into a latent space of a lower dimensionality. For the toy corpus above we used only 2 latent dimensions, but on real corpora, target dimensionality of 200–500 is recommended as a "golden standard"[1].

  ```
  >>> model = lsimodel.LsiModel(tfidf_corpus, id2word=dictionary, num_topics=300)
  ```

  LSI training is unique in that we can continue "training" at any point, simply by providing more training documents. This is done by incremental updates to the underlying model, in a process called *online training*. Because of this feature, the input document stream may even be infinite – just keep feeding LSI new documents as they arrive, while using the computed transformation model as read-only in the meanwhile!

  ```
  >>> model.add_documents(another_tfidf_corpus) # now LSI has been trained on tfidf_
  ↪corpus + another_tfidf_corpus
  >>> lsi_vec = model[tfidf_vec] # convert some new document into the LSI space,␣
  ↪without affecting the model
  ```

[1] Bradford. 2008. An empirical study of required dimensionality for large-scale latent semantic indexing applications.

```
>>> ...
>>> model.add_documents(more_documents) # tfidf_corpus + another_tfidf_corpus +␣
↪more_documents
>>> lsi_vec = model[tfidf_vec]
>>> ...
```

See the `gensim.models.lsimodel` documentation for details on how to make LSI gradually "forget" old observations in infinite streams. If you want to get dirty, there are also parameters you can tweak that affect speed vs. memory footprint vs. numerical precision of the LSI algorithm.

*gensim* uses a novel online incremental streamed distributed training algorithm (quite a mouthful!), which I published in[5]. *gensim* also executes a stochastic multi-pass algorithm from Halko et al.[4] internally, to accelerate in-core part of the computations. See also *Experiments on the English Wikipedia* for further speed-ups by distributing the computation across a cluster of computers.

- Random Projections, RP aim to reduce vector space dimensionality. This is a very efficient (both memory- and CPU-friendly) approach to approximating TfIdf distances between documents, by throwing in a little randomness. Recommended target dimensionality is again in the hundreds/thousands, depending on your dataset.

```
>>> model = rpmodel.RpModel(tfidf_corpus, num_topics=500)
```

- Latent Dirichlet Allocation, LDA is yet another transformation from bag-of-words counts into a topic space of lower dimensionality. LDA is a probabilistic extension of LSA (also called multinomial PCA), so LDA's topics can be interpreted as probability distributions over words. These distributions are, just like with LSA, inferred automatically from a training corpus. Documents are in turn interpreted as a (soft) mixture of these topics (again, just like with LSA).

```
>>> model = ldamodel.LdaModel(bow_corpus, id2word=dictionary, num_topics=100)
```

*gensim* uses a fast implementation of online LDA parameter estimation based on[2], modified to run in *distributed mode* on a cluster of computers.

- Hierarchical Dirichlet Process, HDP is a non-parametric bayesian method (note the missing number of requested topics):

```
>>> model = hdpmodel.HdpModel(bow_corpus, id2word=dictionary)
```

*gensim* uses a fast, online implementation based on[3]. The HDP model is a new addition to *gensim*, and still rough around its academic edges – use with care.

Adding new VSM (Vector Space Model) transformations (such as different weighting schemes) is rather trivial; see the *API reference* or directly the Python code for more info and examples.

It is worth repeating that these are all unique, **incremental** implementations, which do not require the whole training corpus to be present in main memory all at once. With memory taken care of, I am now improving *Distributed Computing*, to improve CPU efficiency, too. If you feel you could contribute (by testing, providing use-cases or code), please let me know.

Continue on to the next tutorial on *Similarity Queries*.

---

[5] Řehůřek. 2011. Subspace tracking for Latent Semantic Analysis.
[4] Halko, Martinsson, Tropp. 2009. Finding structure with randomness.
[2] Hoffman, Blei, Bach. 2010. Online learning for Latent Dirichlet Allocation.
[3] Wang, Paisley, Blei. 2011. Online variational inference for the hierarchical Dirichlet process.

### 1.3.3 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, models, similarities
>>> 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:

```
>>> 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. 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*.

Please remember that gensim is an experimental package, aimed at the NLP research community. This means that:

- there certainly are parts that could be implemented more efficiently (in C, for example), and there may also be bugs in the code
- your **feedback is most welcome** and appreciated, be it in code and idea contributions, bug reports or just user stories and general questions.

Gensim has no ambition to become an all-encompassing production level tool, with robust failure handling and error recoveries. Its main goal is to help NLP newcomers try out popular algorithms and to facilitate prototyping of new algorithms for NLP researchers.

## 1.3.4 Experiments on the English Wikipedia

To test *gensim* performance, we run it against the English version of Wikipedia.

This page describes the process of obtaining and processing Wikipedia, so that anyone can reproduce the results. It is assumed you have *gensim* properly *installed*.

### Preparing the corpus

1. First, download the dump of all Wikipedia articles from http://download.wikimedia.org/enwiki/ (you want a file like *enwiki-latest-pages-articles.xml.bz2*). This file is about 8GB in size and contains (a compressed version of) all articles from the English Wikipedia.
2. Convert the articles to plain text (process Wiki markup) and store the result as sparse TF-IDF vectors. In Python, this is easy to do on-the-fly and we don't even need to uncompress the whole archive to disk. There is a script included in *gensim* that does just that, run:

```
$ python -m gensim.scripts.make_wiki
```

---

**Note:** This pre-processing step makes two passes over the 8.2GB compressed wiki dump (one to extract the dictionary, one to create and store the sparse vectors) and takes about 9 hours on my laptop, so you may want to go have a coffee or two.

Also, you will need about 35GB of free disk space to store the sparse output vectors. I recommend compressing these files immediately, e.g. with bzip2 (down to ~13GB). Gensim can work with compressed files directly, so this lets you save disk space.

---

### Latent Sematic Analysis

First let's load the corpus iterator and dictionary, created in the second step above:

```
>>> import logging, gensim, bz2
>>> logging.basicConfig(format='%(asctime)s : %(levelname)s : %(message)s',␣
↪level=logging.INFO)

>>> # load id->word mapping (the dictionary), one of the results of step 2 above
```

```
>>> id2word = gensim.corpora.Dictionary.load_from_text('wiki_en_wordids.txt')
>>> # load corpus iterator
>>> mm = gensim.corpora.MmCorpus('wiki_en_tfidf.mm')
>>> # mm = gensim.corpora.MmCorpus(bz2.BZ2File('wiki_en_tfidf.mm.bz2')) # use this if
↪you compressed the TFIDF output (recommended)

>>> print mm
MmCorpus(3931787 documents, 100000 features, 756379027 non-zero entries)
```

We see that our corpus contains 3.9M documents, 100K features (distinct tokens) and 0.76G non-zero entries in the sparse TF-IDF matrix. The Wikipedia corpus contains about 2.24 billion tokens in total.

Now we're ready to compute LSA of the English Wikipedia:

```
>>> # extract 400 LSI topics; use the default one-pass algorithm
>>> lsi = gensim.models.lsimodel.LsiModel(corpus=mm, id2word=id2word, num_topics=400)

>>> # print the most contributing words (both positively and negatively) for each of
↪the first ten topics
>>> lsi.print_topics(10)
topic #0(332.762): 0.425*"utc" + 0.299*"talk" + 0.293*"page" + 0.226*"article" + 0.
↪224*"delete" + 0.216*"discussion" + 0.205*"deletion" + 0.198*"should" + 0.146*
↪"debate" + 0.132*"be"
topic #1(201.852): 0.282*"link" + 0.209*"he" + 0.145*"com" + 0.139*"his" + -0.137*
↪"page" + -0.118*"delete" + 0.114*"blacklist" + -0.108*"deletion" + -0.105*
↪"discussion" + 0.100*"diff"
topic #2(191.991): -0.565*"link" + -0.241*"com" + -0.238*"blacklist" + -0.202*"diff"
↪+ -0.193*"additions" + -0.182*"users" + -0.158*"coibot" + -0.136*"user" + 0.133*"he
↪" + -0.130*"resolves"
topic #3(141.284): -0.476*"image" + -0.255*"copyright" + -0.245*"fair" + -0.225*"use"
↪+ -0.173*"album" + -0.163*"cover" + -0.155*"resolution" + -0.141*"licensing" + 0.
↪137*"he" + -0.121*"copies"
topic #4(130.909): 0.264*"population" + 0.246*"age" + 0.243*"median" + 0.213*"income"
↪+ 0.195*"census" + -0.189*"he" + 0.184*"households" + 0.175*"were" + 0.167*"females
↪" + 0.166*"males"
topic #5(120.397): 0.304*"diff" + 0.278*"utc" + 0.213*"you" + -0.171*"additions" + 0.
↪165*"talk" + -0.159*"image" + 0.159*"undo" + 0.155*"www" + -0.152*"page" + 0.148*
↪"contribs"
topic #6(115.414): -0.362*"diff" + -0.203*"www" + 0.197*"you" + -0.180*"undo" + -0.
↪180*"kategori" + 0.164*"users" + 0.157*"additions" + -0.150*"contribs" + -0.139*"he
↪" + -0.136*"image"
topic #7(111.440): 0.429*"kategori" + 0.276*"categoria" + 0.251*"category" + 0.207*
↪"kategorija" + 0.198*"kategorie" + -0.188*"diff" + 0.163*"" + 0.153*"categoría" + 0.
↪139*"kategoria" + 0.133*"categorie"
topic #8(109.907): 0.385*"album" + 0.224*"song" + 0.209*"chart" + 0.204*"band" + 0.
↪169*"released" + 0.151*"music" + 0.142*"diff" + 0.141*"vocals" + 0.138*"she" + 0.
↪132*"guitar"
topic #9(102.599): -0.237*"league" + -0.214*"he" + -0.180*"season" + -0.174*"football
↪" + -0.166*"team" + 0.159*"station" + -0.137*"played" + -0.131*"cup" + 0.131*"she"
↪+ -0.128*"utc"
```

Creating the LSI model of Wikipedia takes about 4 hours and 9 minutes on my laptop[1]. That's about **16,000 documents per minute, including all I/O**.

**Note:** If you need your results even faster, see the tutorial on *Distributed Computing*. Note that the BLAS libraries inside *gensim* make use of multiple cores transparently, so the same data will be processed faster on a multicore

[1] My laptop = MacBook Pro, Intel Core i7 2.3GHz, 16GB DDR3 RAM, OS X with *libVec*.

machine "for free", without any distributed setup.

We see that the total processing time is dominated by the preprocessing step of preparing the TF-IDF corpus from a raw Wikipedia XML dump, which took 9h.[2]

The algorithm used in *gensim* only needs to see each input document once, so it is suitable for environments where the documents come as a non-repeatable stream, or where the cost of storing/iterating over the corpus multiple times is too high.

### Latent Dirichlet Allocation

As with Latent Semantic Analysis above, first load the corpus iterator and dictionary:

```
>>> import logging, gensim, bz2
>>> logging.basicConfig(format='%(asctime)s : %(levelname)s : %(message)s',␣
↪level=logging.INFO)

>>> # load id->word mapping (the dictionary), one of the results of step 2 above
>>> id2word = gensim.corpora.Dictionary.load_from_text('wiki_en_wordids.txt')
>>> # load corpus iterator
>>> mm = gensim.corpora.MmCorpus('wiki_en_tfidf.mm')
>>> # mm = gensim.corpora.MmCorpus(bz2.BZ2File('wiki_en_tfidf.mm.bz2')) # use this if␣
↪you compressed the TFIDF output

>>> print mm
MmCorpus(3931787 documents, 100000 features, 756379027 non-zero entries)
```

We will run online LDA (see Hoffman et al.[3]), which is an algorithm that takes a chunk of documents, updates the LDA model, takes another chunk, updates the model etc. Online LDA can be contrasted with batch LDA, which processes the whole corpus (one full pass), then updates the model, then another pass, another update... The difference is that given a reasonably stationary document stream (not much topic drift), the online updates over the smaller chunks (subcorpora) are pretty good in themselves, so that the model estimation converges faster. As a result, we will perhaps only need a single full pass over the corpus: if the corpus has 3 million articles, and we update once after every 10,000 articles, this means we will have done 300 updates in one pass, quite likely enough to have a very accurate topics estimate:

```
>>> # extract 100 LDA topics, using 1 pass and updating once every 1 chunk (10,000␣
↪documents)
>>> lda = gensim.models.ldamodel.LdaModel(corpus=mm, id2word=id2word, num_topics=100,␣
↪update_every=1, chunksize=10000, passes=1)
```

[2] Here we're mostly interested in performance, but it is interesting to look at the retrieved LSA concepts, too. I am no Wikipedia expert and don't see into Wiki's bowels, but Brian Mingus had this to say about the result:

```
There appears to be a lot of noise in your dataset. The first three topics
in your list appear to be meta topics, concerning the administration and
cleanup of Wikipedia. These show up because you didn't exclude templates
such as these, some of which are included in most articles for quality
control: http://en.wikipedia.org/wiki/Wikipedia:Template_messages/Cleanup

The fourth and fifth topics clearly shows the influence of bots that import
massive databases of cities, countries, etc. and their statistics such as
population, capita, etc.

The sixth shows the influence of sports bots, and the seventh of music bots.
```

So the top ten concepts are apparently dominated by Wikipedia robots and expanded templates; this is a good reminder that LSA is a powerful tool for data analysis, but no silver bullet. As always, it's garbage in, garbage out... By the way, improvements to the Wiki markup parsing code are welcome :-)

[3] Hoffman, Blei, Bach. 2010. Online learning for Latent Dirichlet Allocation [pdf] [code]

```
using serial LDA version on this node
running online LDA training, 100 topics, 1 passes over the supplied corpus of 3931787␣
↪documents, updating model once every 10000 documents
...
```

Unlike LSA, the topics coming from LDA are easier to interpret:

```
>>> # print the most contributing words for 20 randomly selected topics
>>> lda.print_topics(20)
topic #0: 0.009*river + 0.008*lake + 0.006*island + 0.005*mountain + 0.004*area + 0.
↪004*park + 0.004*antarctic + 0.004*south + 0.004*mountains + 0.004*dam
topic #1: 0.026*relay + 0.026*athletics + 0.025*metres + 0.023*freestyle + 0.
↪022*hurdles + 0.020*ret + 0.017*divisão + 0.017*athletes + 0.016*bundesliga + 0.
↪014*medals
topic #2: 0.002*were + 0.002*he + 0.002*court + 0.002*his + 0.002*had + 0.002*law + 0.
↪002*government + 0.002*police + 0.002*patrolling + 0.002*their
topic #3: 0.040*courcelles + 0.035*centimeters + 0.023*mattythewhite + 0.021*wine + 0.
↪019*stamps + 0.018*oko + 0.017*perennial + 0.014*stubs + 0.012*ovate + 0.011*greyish
topic #4: 0.039*al + 0.029*sysop + 0.019*iran + 0.015*pakistan + 0.014*ali + 0.
↪013*arab + 0.010*islamic + 0.010*arabic + 0.010*saudi + 0.010*muhammad
topic #5: 0.020*copyrighted + 0.020*northamerica + 0.014*uncopyrighted + 0.
↪007*rihanna + 0.005*cloudz + 0.005*knowles + 0.004*gaga + 0.004*zombie + 0.
↪004*wigan + 0.003*maccabi
topic #6: 0.061*israel + 0.056*israeli + 0.030*sockpuppet + 0.025*jerusalem + 0.
↪025*tel + 0.023*aviv + 0.022*palestinian + 0.019*ifk + 0.016*palestine + 0.
↪014*hebrew
topic #7: 0.015*melbourne + 0.014*rovers + 0.013*vfl + 0.012*australian + 0.
↪012*wanderers + 0.011*afl + 0.008*dinamo + 0.008*queensland + 0.008*tracklist + 0.
↪008*brisbane
topic #8: 0.011*film + 0.007*her + 0.007*she + 0.004*he + 0.004*series + 0.004*his +␣
↪0.004*episode + 0.003*films + 0.003*television + 0.003*best
topic #9: 0.019*wrestling + 0.013*château + 0.013*ligue + 0.012*discus + 0.
↪012*estonian + 0.009*uci + 0.008*hockeyarchives + 0.008*wwe + 0.008*estonia + 0.
↪007*reign
topic #10: 0.078*edits + 0.059*notability + 0.035*archived + 0.025*clearer + 0.
↪022*speedy + 0.021*deleted + 0.016*hook + 0.015*checkuser + 0.014*ron + 0.
↪011*nominator
topic #11: 0.013*admins + 0.009*acid + 0.009*molniya + 0.009*chemical + 0.007*ch + 0.
↪007*chemistry + 0.007*compound + 0.007*anemone + 0.006*mg + 0.006*reaction
topic #12: 0.018*india + 0.013*indian + 0.010*tamil + 0.009*singh + 0.008*film + 0.
↪008*temple + 0.006*kumar + 0.006*hindi + 0.006*delhi + 0.005*bengal
topic #13: 0.047*bwebs + 0.024*malta + 0.020*hobart + 0.019*basa + 0.019*columella +␣
↪0.019*huon + 0.018*tasmania + 0.016*popups + 0.014*tasmanian + 0.014*modèle
topic #14: 0.014*jewish + 0.011*rabbi + 0.008*bgwhite + 0.008*lebanese + 0.
↪007*lebanon + 0.006*homs + 0.005*beirut + 0.004*jews + 0.004*hebrew + 0.004*caligari
topic #15: 0.025*german + 0.020*der + 0.017*von + 0.015*und + 0.014*berlin + 0.
↪012*germany + 0.012*die + 0.010*des + 0.008*kategorie + 0.007*cross
topic #16: 0.003*can + 0.003*system + 0.003*power + 0.003*are + 0.003*energy + 0.
↪002*data + 0.002*be + 0.002*used + 0.002*or + 0.002*using
topic #17: 0.049*indonesia + 0.042*indonesian + 0.031*malaysia + 0.024*singapore + 0.
↪022*greek + 0.021*jakarta + 0.016*greece + 0.015*dord + 0.014*athens + 0.
↪011*malaysian
topic #18: 0.031*stakes + 0.029*webs + 0.018*futsal + 0.014*whitish + 0.013*hyun + 0.
↪012*thoroughbred + 0.012*dnf + 0.012*jockey + 0.011*medalists + 0.011*racehorse
topic #19: 0.119*oblast + 0.034*uploaded + 0.034*uploads + 0.033*nordland + 0.
↪025*selsoviet + 0.023*raion + 0.022*krai + 0.018*okrug + 0.015*hålogaland + 0.
↪015*russiae + 0.020*manga + 0.017*dragon + 0.012*theme + 0.011*dvd + 0.011*super +␣
↪0.011*hunter + 0.009*ash + 0.009*dream + 0.009*angel
```

Creating this LDA model of Wikipedia takes about 6 hours and 20 minutes on my laptop[1]. If you need your results faster, consider running *Distributed Latent Dirichlet Allocation* on a cluster of computers.

Note two differences between the LDA and LSA runs: we asked LSA to extract 400 topics, LDA only 100 topics (so the difference in speed is in fact even greater). Secondly, the LSA implementation in *gensim* is truly online: if the nature of the input stream changes in time, LSA will re-orient itself to reflect these changes, in a reasonably small amount of updates. In contrast, LDA is not truly online (the name of the[3] article notwithstanding), as the impact of later updates on the model gradually diminishes. If there is topic drift in the input document stream, LDA will get confused and be increasingly slower at adjusting itself to the new state of affairs.

In short, be careful if using LDA to incrementally add new documents to the model over time. **Batch usage of LDA**, where the entire training corpus is either known beforehand or does not exhibit topic drift, **is ok and not affected**.

To run batch LDA (not online), train *LdaModel* with:

```
>>> # extract 100 LDA topics, using 20 full passes, no online updates
>>> lda = gensim.models.ldamodel.LdaModel(corpus=mm, id2word=id2word, num_topics=100,␣
↪update_every=0, passes=20)
```

As usual, a trained model can used be to transform new, unseen documents (plain bag-of-words count vectors) into LDA topic distributions:

```
>>> doc_lda = lda[doc_bow]
```

---

## 1.3.5 Distributed Computing

### Why distributed computing?

Need to build semantic representation of a corpus that is millions of documents large and it's taking forever? Have several idle machines at your disposal that you could use? Distributed computing tries to accelerate computations by splitting a given task into several smaller subtasks, passing them on to several computing nodes in parallel.

In the context of *gensim*, computing nodes are computers identified by their IP address/port, and communication happens over TCP/IP. The whole collection of available machines is called a *cluster*. The distribution is very coarse grained (not much communication going on), so the network is allowed to be of relatively high latency.

**Warning:** The primary reason for using distributed computing is making things run faster. In *gensim*, most of the time consuming stuff is done inside low-level routines for linear algebra, inside NumPy, independent of any *gensim* code. **Installing a fast** BLAS (Basic Linear Algebra) **library for NumPy can improve performance up to 15 times!** So before you start buying those extra computers, consider installing a fast, threaded BLAS that is optimized for your particular machine (as opposed to a generic, binary-distributed library). Options include your vendor's BLAS library (Intel's MKL, AMD's ACML, OS X's vecLib, Sun's Sunperf, ...) or some open-source alternative (GotoBLAS, ALTAS).

To see what BLAS and LAPACK you are using, type into your shell:

```
python -c 'import numpy; numpy.show_config()'
```

### Prerequisites

For communication between nodes, *gensim* uses Pyro (PYthon Remote Objects), version >= 4.8. This is a library for low-level socket communication and remote procedure calls (RPC) in Python. *Pyro* is a pure-Python library, so its installation is quite painless and only involves copying its **.py* files somewhere onto your Python's import path:

```
sudo easy_install Pyro4
```

You don't have to install *Pyro* to run *gensim*, but if you don't, you won't be able to access the distributed features (i.e., everything will always run in serial mode, the examples on this page don't apply).

### Core concepts

As always, *gensim* strives for a clear and straightforward API (see *Features*). To this end, *you do not need to make any changes in your code at all* in order to run it over a cluster of computers!

What you need to do is run a *worker* script (see below) on each of your cluster nodes prior to starting your computation. Running this script tells *gensim* that it may use the node as a slave to delegate some work to it. During initialization, the algorithms inside *gensim* will try to look for and enslave all available worker nodes.

**Node** A logical working unit. Can correspond to a single physical machine, but you can also run multiple workers on one machine, resulting in multiple logical nodes.

**Cluster** Several nodes which communicate over TCP/IP. Currently, network broadcasting is used to discover and connect all communicating nodes, so the nodes must lie within the same broadcast domain.

**Worker** A process which is created on each node. To remove a node from your cluster, simply kill its worker process.

**Dispatcher** The dispatcher will be in charge of negotiating all computations, queueing and distributing ("dispatching") individual jobs to the workers. Computations never "talk" to worker nodes directly, only through this dispatcher. Unlike workers, there can only be one active dispatcher at a time in the cluster.

### Available distributed algorithms

### Distributed Latent Semantic Analysis

---

**Note:** See *Distributed Computing* for an introduction to distributed computing in *gensim*.

---

### Setting up the cluster

We will show how to run distributed Latent Semantic Analysis by means of an example. Let's say we have 5 computers at our disposal, all on the same network segment (=reachable by network broadcast). To start with, install *gensim* and *Pyro* on each computer with:

```
$ sudo easy_install gensim[distributed]
```

and run Pyro's name server on exactly one of the machines (doesn't matter which one):

```
$ python -m Pyro4.naming -n 0.0.0.0 &
```

Let's say our example cluster consists of dual-core computers with loads of memory. We will therefore run **two** worker scripts on four of the physical machines, creating **eight** logical worker nodes:

```
$ python -m gensim.models.lsi_worker &
```

This will execute *gensim*'s *lsi_worker.py* script (to be run twice on each of the four computer). This lets *gensim* know that it can run two jobs on each of the four computers in parallel, so that the computation will be done faster, while also taking up twice as much memory on each machine.

Next, pick one computer that will be a job scheduler in charge of worker synchronization, and on it, run *LSA dispatcher*. In our example, we will use the fifth computer to act as the dispatcher and from there run:

```
$ python -m gensim.models.lsi_dispatcher &
```

In general, the dispatcher can be run on the same machine as one of the worker nodes, or it can be another, distinct computer (within the same broadcast domain). The dispatcher won't be doing much with CPU most of the time, but pick a computer with ample memory.

And that's it! The cluster is set up and running, ready to accept jobs. To remove a worker later on, simply terminate its *lsi_worker* process. To add another worker, run another *lsi_worker* (this will not affect a computation that is already running, the additions/deletions are not dynamic). If you terminate *lsi_dispatcher*, you won't be able to run computations until you run it again (surviving worker processes can be re-used though).

### Running LSA

So let's test our setup and run one computation of distributed LSA. Open a Python shell on one of the five machines (again, this can be done on any computer in the same broadcast domain, our choice is incidental) and try:

```
>>> from gensim import corpora, models, utils
>>> import logging
>>> logging.basicConfig(format='%(asctime)s : %(levelname)s : %(message)s',␣
↪level=logging.INFO)

>>> corpus = corpora.MmCorpus('/tmp/deerwester.mm') # load a corpus of nine documents,
↪ from the Tutorials
>>> id2word = corpora.Dictionary.load('/tmp/deerwester.dict')

>>> lsi = models.LsiModel(corpus, id2word=id2word, num_topics=200, chunksize=1,␣
↪distributed=True) # run distributed LSA on nine documents
```

This uses the corpus and feature-token mapping created in the *Corpora and Vector Spaces* tutorial. If you look at the log in your Python session, you should see a line similar to:

```
2010-08-09 23:44:25,746 : INFO : using distributed version with 8 workers
```

which means all went well. You can also check the logs coming from your worker and dispatcher processes — this is especially helpful in case of problems. To check the LSA results, let's print the first two latent topics:

```
>>> lsi.print_topics(num_topics=2, num_words=5)
topic #0(3.341): 0.644*"system" + 0.404*"user" + 0.301*"eps" + 0.265*"time" + 0.265*
↪"response"
topic #1(2.542): 0.623*"graph" + 0.490*"trees" + 0.451*"minors" + 0.274*"survey" + -0.
↪167*"system"
```

Success! But a corpus of nine documents is no challenge for our powerful cluster... In fact, we had to lower the job size (*chunksize* parameter above) to a single document at a time, otherwise all documents would be processed by a single worker all at once.

So let's run LSA on **one million documents** instead:

```
>>> # inflate the corpus to 1M documents, by repeating its documents over&over
>>> corpus1m = utils.RepeatCorpus(corpus, 1000000)
>>> # run distributed LSA on 1 million documents
>>> lsi1m = models.LsiModel(corpus1m, id2word=id2word, num_topics=200,␣
↪chunksize=10000, distributed=True)
```

```
>>> lsi1m.print_topics(num_topics=2, num_words=5)
topic #0(1113.628): 0.644*"system" + 0.404*"user" + 0.301*"eps" + 0.265*"time" + 0.
↪265*"response"
topic #1(847.233): 0.623*"graph" + 0.490*"trees" + 0.451*"minors" + 0.274*"survey" + -
↪0.167*"system"
```

The log from 1M LSA should look like:

```
2010-08-10 02:46:35,087 : INFO : using distributed version with 8 workers
2010-08-10 02:46:35,087 : INFO : updating SVD with new documents
2010-08-10 02:46:35,202 : INFO : dispatched documents up to #10000
2010-08-10 02:46:35,296 : INFO : dispatched documents up to #20000
...
2010-08-10 02:46:46,524 : INFO : dispatched documents up to #990000
2010-08-10 02:46:46,694 : INFO : dispatched documents up to #1000000
2010-08-10 02:46:46,694 : INFO : reached the end of input; now waiting for all␣
↪remaining jobs to finish
2010-08-10 02:46:47,195 : INFO : all jobs finished, downloading final projection
2010-08-10 02:46:47,200 : INFO : decomposition complete
```

Due to the small vocabulary size and trivial structure of our "one-million corpus", the computation of LSA still takes only 12 seconds. To really stress-test our cluster, let's do Latent Semantic Analysis on the English Wikipedia.

### Distributed LSA on Wikipedia

First, download and prepare the Wikipedia corpus as per *Experiments on the English Wikipedia*, then load the corpus iterator with:

```
>>> import logging, gensim, bz2
>>> logging.basicConfig(format='%(asctime)s : %(levelname)s : %(message)s',␣
↪level=logging.INFO)

>>> # load id->word mapping (the dictionary)
>>> id2word = gensim.corpora.Dictionary.load_from_text('wiki_en_wordids.txt')
>>> # load corpus iterator
>>> mm = gensim.corpora.MmCorpus('wiki_en_tfidf.mm')
>>> # mm = gensim.corpora.MmCorpus(bz2.BZ2File('wiki_en_tfidf.mm.bz2')) # use this if␣
↪you compressed the TFIDF output

>>> print mm
MmCorpus(3199665 documents, 100000 features, 495547400 non-zero entries)
```

Now we're ready to run distributed LSA on the English Wikipedia:

```
>>> # extract 400 LSI topics, using a cluster of nodes
>>> lsi = gensim.models.lsimodel.LsiModel(corpus=mm, id2word=id2word, num_topics=400,␣
↪chunksize=20000, distributed=True)

>>> # print the most contributing words (both positively and negatively) for each of␣
↪the first ten topics
>>> lsi.print_topics(10)
2010-11-03 16:08:27,602 : INFO : topic #0(200.990): -0.475*"delete" + -0.383*"deletion
↪" + -0.275*"debate" + -0.223*"comments" + -0.220*"edits" + -0.213*"modify" + -0.208*
↪"appropriate" + -0.194*"subsequent" + -0.155*"wp" + -0.117*"notability"
2010-11-03 16:08:27,626 : INFO : topic #1(143.129): -0.320*"diff" + -0.305*"link" + -
↪0.199*"image" + -0.171*"www" + -0.162*"user" + 0.149*"delete" + -0.147*"undo" + -0.
↪144*"contribs" + -0.122*"album" + 0.113*"deletion"
```

```
2010-11-03 16:08:27,651 : INFO : topic #2(135.665): -0.437*"diff" + -0.400*"link" + -
↪0.202*"undo" + -0.192*"user" + -0.182*"www" + -0.176*"contribs" + 0.168*"image" + -
↪0.109*"added" + 0.106*"album" + 0.097*"copyright"
2010-11-03 16:08:27,677 : INFO : topic #3(125.027): -0.354*"image" + 0.239*"age" + 0.
↪218*"median" + -0.213*"copyright" + 0.204*"population" + -0.195*"fair" + 0.195*
↪"income" + 0.167*"census" + 0.165*"km" + 0.162*"households"
2010-11-03 16:08:27,701 : INFO : topic #4(116.927): -0.307*"image" + 0.195*"players"␣
↪+ -0.184*"median" + -0.184*"copyright" + -0.181*"age" + -0.167*"fair" + -0.162*
↪"income" + -0.151*"population" + -0.136*"households" + -0.134*"census"
2010-11-03 16:08:27,728 : INFO : topic #5(100.326): 0.501*"players" + 0.318*"football
↪" + 0.284*"league" + 0.193*"footballers" + 0.142*"image" + 0.133*"season" + 0.119*
↪"cup" + 0.113*"club" + 0.110*"baseball" + 0.103*"f"
2010-11-03 16:08:27,754 : INFO : topic #6(92.298): -0.411*"album" + -0.275*"albums" +␣
↪-0.217*"band" + -0.214*"song" + -0.184*"chart" + -0.163*"songs" + -0.160*"singles"␣
↪+ -0.149*"vocals" + -0.139*"guitar" + -0.129*"track"
2010-11-03 16:08:27,780 : INFO : topic #7(83.811): -0.248*"wikipedia" + -0.182*"keep"␣
↪+ 0.180*"delete" + -0.167*"articles" + -0.152*"your" + -0.150*"my" + 0.144*"film" +␣
↪-0.130*"we" + -0.123*"think" + -0.120*"user"
2010-11-03 16:08:27,807 : INFO : topic #8(78.981): 0.588*"film" + 0.460*"films" + -0.
↪130*"album" + -0.127*"station" + 0.121*"television" + 0.115*"poster" + 0.112*
↪"directed" + 0.110*"actors" + -0.096*"railway" + 0.086*"movie"
2010-11-03 16:08:27,834 : INFO : topic #9(78.620): 0.502*"kategori" + 0.282*"categoria
↪" + 0.248*"kategorija" + 0.234*"kategorie" + 0.172*"" + 0.165*"categoría" + 0.161*
↪"kategoria" + 0.148*"categorie" + 0.126*"kategória" + 0.121*"catégorie"
```

In serial mode, creating the LSI model of Wikipedia with this **one-pass algorithm** takes about 5.25h on my laptop (OS X, C2D 2.53GHz, 4GB RAM with *libVec*). In distributed mode with four workers (Linux, dual-core Xeons of 2Ghz, 4GB RAM with *ATLAS*), the wallclock time taken drops to 1 hour and 41 minutes. You can read more about various internal settings and experiments in my research paper.

### Distributed Latent Dirichlet Allocation

**Note:** See *Distributed Computing* for an introduction to distributed computing in *gensim*.

### Setting up the cluster

See the tutorial on *Distributed Latent Semantic Analysis*; setting up a cluster for LDA is completely analogous, except you want to run *lda_worker* and *lda_dispatcher* scripts instead of *lsi_worker* and *lsi_dispatcher*.

### Running LDA

Run LDA like you normally would, but turn on the *distributed=True* constructor parameter:

```
>>> # extract 100 LDA topics, using default parameters
>>> lda = LdaModel(corpus=mm, id2word=id2word, num_topics=100, distributed=True)
using distributed version with 4 workers
running online LDA training, 100 topics, 1 passes over the supplied corpus of 3199665␣
↪documets, updating model once every 40000 documents
..
```

In serial mode (no distribution), creating this online LDA *model of Wikipedia* takes 10h56m on my laptop (OS X, C2D 2.53GHz, 4GB RAM with *libVec*). In distributed mode with four workers (Linux, Xeons of 2Ghz, 4GB RAM with ATLAS), the wallclock time taken drops to 3h20m.

To run standard batch LDA (no online updates of mini-batches) instead, you would similarly call:

```
>>> lda = LdaModel(corpus=mm, id2word=id2token, num_topics=100, update_every=0,␣
↪passes=20, distributed=True)
using distributed version with 4 workers
running batch LDA training, 100 topics, 20 passes over the supplied corpus of 3199665␣
↪documets, updating model once every 3199665 documents
initializing workers
iteration 0, dispatching documents up to #10000/3199665
iteration 0, dispatching documents up to #20000/3199665
...
```

and then, some two days later:

```
iteration 19, dispatching documents up to #3190000/3199665
iteration 19, dispatching documents up to #3199665/3199665
reached the end of input; now waiting for all remaining jobs to finish
```

```
>>> lda.print_topics(20)
topic #0: 0.007*disease + 0.006*medical + 0.005*treatment + 0.005*cells + 0.005*cell␣
↪+ 0.005*cancer + 0.005*health + 0.005*blood + 0.004*patients + 0.004*drug
topic #1: 0.024*king + 0.013*ii + 0.013*prince + 0.013*emperor + 0.008*duke + 0.
↪008*empire + 0.007*son + 0.007*china + 0.007*dynasty + 0.007*iii
topic #2: 0.031*film + 0.017*films + 0.005*movie + 0.005*directed + 0.004*man + 0.
↪004*episode + 0.003*character + 0.003*cast + 0.003*father + 0.003*mother
topic #3: 0.022*user + 0.012*edit + 0.009*wikipedia + 0.007*block + 0.007*my + 0.
↪007*here + 0.007*edits + 0.007*blocked + 0.006*revert + 0.006*me
topic #4: 0.045*air + 0.026*aircraft + 0.021*force + 0.018*airport + 0.011*squadron +␣
↪0.010*flight + 0.010*military + 0.008*wing + 0.007*aviation + 0.007*f
topic #5: 0.025*sun + 0.022*star + 0.018*moon + 0.015*light + 0.013*stars + 0.
↪012*planet + 0.011*camera + 0.010*mm + 0.009*earth + 0.008*lens
topic #6: 0.037*radio + 0.026*station + 0.022*fm + 0.014*news + 0.014*stations + 0.
↪014*channel + 0.013*am + 0.013*racing + 0.011*tv + 0.010*broadcasting
topic #7: 0.122*image + 0.099*jpg + 0.046*file + 0.038*uploaded + 0.024*png + 0.
↪014*contribs + 0.013*notify + 0.013*logs + 0.013*picture + 0.013*flag
topic #8: 0.036*russian + 0.030*soviet + 0.028*polish + 0.024*poland + 0.022*russia +␣
↪0.013*union + 0.012*czech + 0.011*republic + 0.011*moscow + 0.010*finland
topic #9: 0.031*language + 0.014*word + 0.013*languages + 0.009*term + 0.009*words +␣
↪0.008*example + 0.007*names + 0.007*meaning + 0.006*latin + 0.006*form
topic #10: 0.029*w + 0.029*toronto + 0.023*l + 0.020*hockey + 0.019*nhl + 0.
↪014*ontario + 0.012*calgary + 0.011*edmonton + 0.011*hamilton + 0.010*season
topic #11: 0.110*wikipedia + 0.110*articles + 0.030*library + 0.029*wikiproject + 0.
↪028*project + 0.019*data + 0.016*archives + 0.012*needing + 0.009*reference + 0.
↪009*statements
topic #12: 0.032*http + 0.030*your + 0.022*request + 0.017*sources + 0.016*archived +␣
↪0.016*modify + 0.015*changes + 0.015*creation + 0.014*www + 0.013*try
topic #13: 0.011*your + 0.010*my + 0.009*we + 0.008*don + 0.008*get + 0.008*know + 0.
↪007*me + 0.006*think + 0.006*question + 0.005*find
topic #14: 0.073*r + 0.066*japanese + 0.062*japan + 0.018*tokyo + 0.008*prefecture +␣
↪0.005*osaka + 0.004*j + 0.004*sf + 0.003*kyoto + 0.003*manga
topic #15: 0.045*da + 0.045*fr + 0.027*kategori + 0.026*pl + 0.024*nl + 0.021*pt + 0.
↪017*en + 0.015*categoria + 0.014*es + 0.012*kategorie
topic #16: 0.010*death + 0.005*died + 0.005*father + 0.004*said + 0.004*himself + 0.
↪004*took + 0.004*son + 0.004*killed + 0.003*murder + 0.003*wife
```

```
topic #17: 0.027*book + 0.021*published + 0.020*books + 0.014*isbn + 0.010*author + 0.
↪010*magazine + 0.009*press + 0.009*novel + 0.009*writers + 0.008*story
topic #18: 0.027*football + 0.024*players + 0.023*cup + 0.019*club + 0.017*fc + 0.
↪017*footballers + 0.017*league + 0.011*season + 0.007*teams + 0.007*goals
topic #19: 0.032*band + 0.024*album + 0.014*albums + 0.013*guitar + 0.013*rock + 0.
↪011*records + 0.011*vocals + 0.009*live + 0.008*bass + 0.008*track
```

If you used the distributed LDA implementation in *gensim*, please let me know (my email is at the bottom of this page). I would like to hear about your application and the possible (inevitable?) issues that you encountered, to improve *gensim* in the future.

### 1.3.6 Preliminaries

All the examples can be directly copied to your Python interpreter shell (assuming you have *gensim installed*, of course). IPython's `cpaste` command is especially handy for copypasting code fragments which include superfluous characters, such as the leading >>>.

Gensim uses Python's standard `logging` module to log various stuff at various priority levels; to activate logging (this is optional), run

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

### 1.3.7 Quick Example

First, let's import gensim and create a small corpus of nine documents[1]:

```
>>> from gensim import corpora, models, similarities
>>>
>>> corpus = [[(0, 1.0), (1, 1.0), (2, 1.0)],
>>>           [(2, 1.0), (3, 1.0), (4, 1.0), (5, 1.0), (6, 1.0), (8, 1.0)],
>>>           [(1, 1.0), (3, 1.0), (4, 1.0), (7, 1.0)],
>>>           [(0, 1.0), (4, 2.0), (7, 1.0)],
>>>           [(3, 1.0), (5, 1.0), (6, 1.0)],
>>>           [(9, 1.0)],
>>>           [(9, 1.0), (10, 1.0)],
>>>           [(9, 1.0), (10, 1.0), (11, 1.0)],
>>>           [(8, 1.0), (10, 1.0), (11, 1.0)]]
```

*Corpus* is simply an object which, when iterated over, returns its documents represented as sparse vectors.

If you're familiar with the Vector Space Model, you'll probably know that the way you parse your documents and convert them to vectors has major impact on the quality of any subsequent applications. If you're not familiar with VSM, we'll bridge the gap between **raw strings** and **sparse vectors** in the next tutorial on *Corpora and Vector Spaces*.

---

**Note:** In this example, the whole corpus is stored in memory, as a Python list. However, the corpus interface only dictates that a corpus must support iteration over its constituent documents. For very large corpora, it is advantageous to keep the corpus on disk, and access its documents sequentially, one at a time. All the operations and transformations are implemented in such a way that makes them independent of the size of the corpus, memory-wise.

---

Next, let's initialize a *transformation*:

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

```
>>> tfidf = models.TfidfModel(corpus)
```

A transformation is used to convert documents from one vector representation into another:

```
>>> vec = [(0, 1), (4, 1)]
>>> print tfidf[vec]
[(0, 0.8075244), (4, 0.5898342)]
```

Here, we used Tf-Idf, a simple transformation which takes documents represented as bag-of-words counts and applies a weighting which discounts common terms (or, equivalently, promotes rare terms). It also scales the resulting vector to unit length (in the Euclidean norm).

Transformations are covered in detail in the tutorial on *Topics and Transformations*.

To transform the whole corpus via TfIdf and index it, in preparation for similarity queries:

```
>>> index = similarities.SparseMatrixSimilarity(tfidf[corpus])
```

and to query the similarity of our query vector `vec` against every document in the corpus:

```
>>> sims = index[tfidf[vec]]
>>> print list(enumerate(sims))
[(0, 0.4662244), (1, 0.19139354), (2, 0.24600551), (3, 0.82094586), (4, 0.0), (5, 0.
↪0), (6, 0.0), (7, 0.0), (8, 0.0)]
```

How to read this output? Document number zero (the first document) has a similarity score of 0.466=46.6%, the second document has a similarity score of 19.1% etc.

Thus, according to TfIdf document representation and cosine similarity measure, the most similar to our query document *vec* is document no. 3, with a similarity score of 82.1%. Note that in the TfIdf representation, any documents which do not share any common features with `vec` at all (documents no. 4–8) get a similarity score of 0.0. See the *Similarity Queries* tutorial for more detail.

---

## 1.4 API Reference

Modules:

# Index