gensim Documentation

Release 0.6.0

Radim Řehůřek

August 10, 2010

CONTENTS

What's new?

Version 0.6 is out!

It contains several minor bug fixes and a **fully online** implementation of Latent Semanting Indexing! You can now *update LSI with new documents* at will, and use the resulting LSI transformation at each step. The training document stream may even be infinite!

For an introduction on what gensim does (or does not do), go to the *introduction*.

To download and install *gensim*, consult the *install* page.

For examples on how to use it, try the *tutorials*.

CHAPTER

ONE

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 twenty latent dimensions)
>>> lsi = models.LsiModel(corpus, numTopics = 200)
>>>
>>> # convert the same corpus to latent space and index it
>>> index = similarities.MatrixSimilarity(lsi[corpus])
>>>
>>> # perform similarity query of another vector in LSI space against the whole corpus
>>> sims = index[query]
```

CHAPTER

TWO

CONTENTS

2.1 Introduction

Gensim is a Python framework designed to help make the conversion of natural language texts to the Vector Space Model as simple and natural as possible.

Gensim contains algorithms for unsupervised learning from raw, unstructured digital texts, such as **Latent Semantic Analysis**, **Latent Dirichlet Allocation** or **Random Projections**. These algorithms discover hidden (*latent*) corpus structure. Once found, documents can be succinctly expressed in terms of this structure, queried for topical similarity and so on.

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

Note: Gensim's target audience is the NLP research community and interested general public; gensim is not meant to be a production tool for commercial environments.

2.1.1 Design objectives

Gensim includes the following features:

- Memory independence – there is no need for the whole text corpus (or any intermediate term-document matrices) to reside fully in RAM at any one time.
- Provides implementations for several popular vector space algorithms, including Tf-Idf, Latent Semantic Analysis (LSA/LSI via incremental SVD), Latent Dirichlet Allocation (LDA) or Random Projection, and makes adding new ones simple.
- Contains I/O wrappers and converters around several popular data formats.
- Allows similarity queries across documents in their latent, topical representation.

Creation of gensim was motivated by a perceived lack of available, scalable software frameworks that realize topic modeling, and/or their overwhelming internal complexity. You can read more about the motivation in our LREC 2010 workshop paper.

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

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

2.1.2 Availability

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

See Also:

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

2.1.3 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, so that the representation of a 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 our 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 sparse representation of a document can be unambiguously resolved to zero, `0.0`.

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) than the initial one. The transformation process is also sometimes called *clustering* in machine learning terminology, or *noise reduction*, from signal processing.

See Also:

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

2.2 Installation

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 and 2.6.
- NumPy >= 1.0.4. Tested with version 1.4.0, 1.3.0rc2 and 1.0.4.
- SciPy >= 0.6. Tested with version 0.8.0b1, 0.7.1 and 0.6.0.

2.2.1 Install Python

Check what version of Python you have with:

```
python --version
```

You can download Python 2.5 from http://python.org/download.

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

2.2.2 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.

2.2.3 Install gensim

You can now install (or upgrade) gensim with:

```
sudo easy_install gensim
```

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

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 svn), 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 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 package when importing.

2.2.4 Testing gensim

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

```
python setup.py test
```

2.2.5 Contact

If you encounter problems or have any questions regarding *gensim*, let me know at:

```
>>> '@'.join(['radimrehurek', '.'.join(['seznam', 'cz'])])
```

2.3 Tutorial

This tutorial is 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:

2.3.1 Corpora and Vector Spaces

Don't forget to set

```
>>> import logging
>>> logging.root.setLevel(logging.INFO) # will suppress DEBUG level events
```

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
>>> allTokens = sum(texts, [])
>>> tokensOnce = set(word for word in set(allTokens) if allTokens.count(word) == 1)
>>> texts = [[word for word in text if word not in tokensOnce]
>>>           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, we only split on whitespace to tokenize, followed by lowercasing each word. In fact, we 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 we 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 we 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 will 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.fromDocuments(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 by calling `Dictionary.fromDocuments()`. This sweeps across the texts, collecting words 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:

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

```
>>> newDoc = "Human computer interaction"
>>> newVec = dictionary.doc2bow(newDoc.lower().split())
>>> print newVec # 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.saveCorpus('/tmp/deerwester.mm', corpus) # store to disk, for later use
>>> print 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)]]
```

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*.

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 storing a Vector Space corpus (~sequence of vectors) to disk. *Gensim* implements them via the *streaming corpus interface* mentioned earlier: documents are read from 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 formats is the Market Matrix format. To save a corpus in the Matrix Market format:

```
>>> from gensim import corpora
>>>
>>> corpus = [[(1, 0.5)], []] # create a toy corpus of 2 documents (one of them empty, for the heck o
>>>
>>> corpora.MmCorpus.saveCorpus('/tmp/corpus.mm', corpus)
```

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

```
>>> corpora.SvmLightCorpus.saveCorpus('/tmp/corpus.svmlight', corpus)
>>> corpora.BleiCorpus.saveCorpus('/tmp/corpus.lda-c', corpus)
>>> corpora.LowCorpus.saveCorpus('/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 corpus in Blei's LDA-C format,

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

In this way, *gensim* can also be used as a simple **I/O format conversion tool**: just load a document stream using one format and immediately save it in another format.

For a complete reference (want to prune the dictionary to a smaller size? convert between sparse vectors and numpy/scipy arrays?), see the *API documentation*. Or continue to the next tutorial on *Topics and Transformations*.

2.3.2 Topics and Transformations

Don't forget to set

```
>>> import logging
>>> logging.root.setLevel(logging.INFO) # will suppress DEBUG level events
```

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, we 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 the original features and use them to describe the documents in a new and (hopefully) more realistic 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, so that transformations can be thought of as 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 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 are initialized to 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
```

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 correct 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. This is because conversion at the time of calling `corpus2 = model[corpus]` would mean storing the result in main memory, which contradicts gensim's objective of memory-indepedence. If you will be iterating over the transformed `corpus2` 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.id2token, numTopics = 2) # initialize a
>>> corpus_lsi = lsi[corpus_tfidf] # create a double wrapper over the original corpus: bow->tfidf->f
```

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

```
>>> for topicNo in range(lsi.numTopics):
>>>     print 'topic %i: %s' % (topicNo, lsi.printTopic(topicNo))
topic 0: -0.703 * "trees" + -0.538 * "graph" + -0.402 * "minors" + -0.187 * "survey" + -0.061 * "syst
topic 1: 0.460 * "system" + 0.373 * "user" + 0.332 * "eps" + 0.328 * "interface" + 0.320 * "time" + (
```

It appears that according to LSI, "trees", "graphs" 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 he
>>>     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" [2].

  ```
  >>> model = lsimodel.LsiModel(tfidf_corpus, id2word = dictionary.id2token, numTopics = 300)
  ```

  LSI training is unique in that it only inspects each input document once. This allows us to 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.addDocuments(another_tfidf_corpus) # now LSI has been trained on tfidf_corpus + anothe
  >>> lsi_vec = model[tfidf_vec] # convert a new document into the LSI space, without affecting th
  >>> ...
  >>> model.addDocuments(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 and how to tweak parameters affecting speed vs. memory footprint vs. numerical precision of the algorithm.

- 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, numTopics = 500)
  ```

- Latent Dirichlet Allocation, LDA is yet another transformation from bag-of-words counts into a topic space of lower dimensionality. LDA is **much** slower than the other algorithms, so we are currently looking into ways of making it faster (see eg. [3], [4]). If you could help, let us know!

  ```
  >>> model = ldamodel.LdaModel(bow_corpus, id2word = dictionary.id2token, numTopics = 200)
  ```

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, we are now implementing *Distributed Computing*, to improve CPU efficiency, too. If you feel you could contribute, please let us know!

---

[2] Bradford, R.B., 2008. An empirical study of required dimensionality for large-scale latent semantic indexing applications.
[3] Asuncion, A., 2009. On Smoothing and Inference for Topic Models.
[4] Yao, Mimno, McCallum, 2009. Efficient Methods for Topic Model Inference on Streaming Document Collections

2.3.3 Similarity Queries

Don't forget to set

```
>>> import logging
>>> logging.root.setLevel(logging.INFO) # will suppress DEBUG level events
```

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 t
>>> 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.id2token, numTopics = 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 an extension over a 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 # result is already scaled by singular values
[(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 will 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 1GB of RAM in a 256-dimensional LSI space, when used with this class. Without 1GB of free RAM, you would need to use the `similarities.Similarity` class. This class operates in constant memory, in a streaming (and more gensim-like) fashion, but is also much slower than `similarities.MatrixSimilarity`, which uses fast level-2 BLAS routines to determine similarities.

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

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

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
```

(We 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, 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 modeling 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*.

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.

Gensim has no ambition to become a 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.

2.3.4 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)
```

2.3.5 Quick Example

First, let's import gensim and create a small corpus of nine documents [5]:

```
>>> 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*:

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

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

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

```
>>> 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,
```

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, all 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.

2.4 Distributed Computing

2.4.1 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 8 times!** So before you start buying those extra computers, consider installing a fast, threaded BLAS first. 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 your NumPy's using, type into your shell:
>
> ```
> python -c 'import numpy; numpy.show_config()'
> ```

2.4.2 Distributed computing in *gensim*

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

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 automatically try to look for and enslave all available worker nodes. If at least one worker is found, things will run in the distributed mode; if not, in serial node.

To remove a node from your cluster, simply kill its worker script process.

For communication between nodes, *gensim* uses Pyro (PYthon Remote Objects), version >= 4.1. This is a library for low-level socket communication and remote procedure calls (RPC) in Python. *Pyro* is a pure-Python library, so installation is quite painless and only involves copying its **.py* files somewhere onto your Python's import path:

```
sudo easy_install Pyro
```

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). Currently, network broadcasting is used to discover and connect all communicating nodes, so the nodes must lie within the same broadcast domain.

2.4.3 Available distributed algorithms

Currently, there is only distributed *Latent Semantic Analysis (LSA, LSI)*. Distributed *Latent Dirichlet Allocation* is on its way.

Distributed LSA

We will show how to run distributed Latent Semantic Analysis on an example. Let's say we have 5 computers at our disposal, all in the same broadcast domain. To start with, install *gensim* and *Pyro* on each one of them with:

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

Let's say our example cluster consists of five 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 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 (but 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, start Pyro's name server and an *LSA dispatcher*:

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

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 will be in charge of negotiating all computations, queueing and distributing ("dispatching") individual jobs to the workers (not doing much with CPU most of the time), so pick a computer with ample memory. Computations never "talk" to worker nodes directly, only through the dispatcher.

In our example, we will use the fifth computer to act as the dispatcher and run the *lsi_dispatcher* and *Pyro.naming* scripts from there.

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). If you terminate *lsi_dispatcher*, you won't be able to run computations until you run it again on some node (surviving workers can be re-used though).

So let's test our setup and run one computation of distributed LSA. Open a Python shell on a worker node (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 Tuto
>>> id2word = corpora.Dictionary.load('/tmp/deerwester.dict').id2token
>>>
>>> lsi = models.LsiModel(corpus, id2word, numTopics = 200, chunks = 1) # run distributed LSA on nine
```

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:

```
>>> for i in xrange(2): lsi.printTopic(i, topN = 5)
0.644 * "survey" + 0.404 * "response" + 0.301 * "user" + 0.265 * "time" + 0.265 * "system"
0.623 * "graph" + 0.490 * "trees" + 0.451 * "minors" + 0.274 * "eps" + -0.167 * "survey"
```

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

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

```
>>> corpus1m = utils.RepeatCorpus(corpus, 1000000) # inflate the corpus to 1M documents, by repeatin
>>> lsi1m = models.LsiModel(corpus1m, id2word, numTopics = 200, serial_only = False) # run distribute


>>> for i in xrange(2): lsi1m.printTopic(i, topN = 5)
-0.644 * "survey" + -0.404 * "response" + -0.301 * "user" + -0.265 * "time" + -0.265 * "system
0.623 * "graph" + 0.490 * "trees" + 0.451 * "minors" + 0.274 * "eps" + -0.167 * "survey"
```

The *serial_only* parameter instructs *gensim* whether to run in serial or distributed mode. Setting it to *True* will result in LSA running inside the active Python shell, without any inter-node communication whatsoever, even if there are worker nodes available. Setting *serial_only=False* forces distributed mode (raising an exception in case of failure). And finally, leaving *serial_only* unspecified tells *gensim* to try running in distributed mode, or, failing that, run in serial mode.

On my Macbook (all 8 "distributed" workers operating on a single physical machine), the log from 1M LSA looks 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 fini
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, Wikipedia FIXME TODO.

2.5 API Reference

Modules:

2.5.1 `interfaces` – Core gensim interfaces

This module contains basic interfaces used throughout the whole gensim package.

The interfaces are realized as abstract base classes (ie., some optional functionality is provided in the interface itself, so that the interfaces can be subclassed).

class `CorpusABC`()

Interface for corpora. A *corpus* is simply an iterable, where each iteration step yields one document. A document is a list of (fieldId, fieldValue) 2-tuples.

See the *corpora* package for some example corpus implementations.

Note that although a default `len()` method is provided, it is very inefficient (performs a linear scan through the corpus to determine its length). Wherever the corpus size is needed and known in advance (or at least doesn't change so that it can be cached), the `len()` method should be overridden.

See the `gensim.corpora.mmcorpus` module for an example of a corpus.

class **`load`**(*fname*)

Load a previously saved object from file (also see *save*).

`save`(*fname*)

Save the object to file via pickling (also see *load*).

class `SimilarityABC`(*corpus*)

Abstract interface for similarity searches over a corpus.

In all instances, there is a corpus against which we want to perform the similarity search.

For each similarity search, the input is a document and the output are its similarities to individual corpus documents.

Similarity queries are realized by calling `self[query_document]`.

There is also a convenience wrapper, where iterating over *self* yields similarities of each document in the corpus against the whole corpus (ie., the query is each corpus document in turn).

`getSimilarities`(*doc*)

Return similarity of a sparse vector *doc* to all documents in the corpus.

The document is assumed to be either of unit length or empty.

class **load**(*fname*)
Load a previously saved object from file (also see *save*).

save(*fname*)
Save the object to file via pickling (also see *load*).

class TransformationABC()
Interface for transformations. A 'transformation' is any object which accepts a sparse document via the dictionary notation *[]* and returns another sparse document in its stead.

See the gensim.models.tfidfmodel module for an example of a transformation.

class **load**(*fname*)
Load a previously saved object from file (also see *save*).

save(*fname*)
Save the object to file via pickling (also see *load*).

2.5.2 utils – Various utility functions

This module contains various general utility functions.

class FakeDict(*numTerms*)
Objects of this class act as dictionaries that map integer->str(integer), for a specified range of integers <0, numTerms).

This is meant to avoid allocating real dictionaries when numTerms is huge, which is a waste of memory.

keys()
Override the dict.keys() function, which is used to determine the maximum internal id of a corpus = the vocabulary dimensionality.

HACK: To avoid materializing the whole range(0, self.numTerms), we return [self.numTerms - 1] only.

class RepeatCorpus(*corpus, reps*)
Used in the tutorial on distributed computing and likely not useful anywhere else.

Wrap a *corpus* as another corpus of length *reps*. This is achieved by repeating documents from *corpus* over and over again, until requested length is reached. Repetition is done on-the-fly=efficiently, via itertools.

```
>>> corpus = [[(1, 0.5)], []] # 2 documents
>>> list(RepeatCorpus(corpus, 5)) # repeat 2.5 times to get 5 documents
>>> [[(1, 0.5)], [], [(1, 0.5)], [], [(1, 0.5)]]
```

class **load**(*fname*)
Load a previously saved object from file (also see *save*).

save(*fname*)
Save the object to file via pickling (also see *load*).

class SaveLoad()
Objects which inherit from this class have save/load functions, which un/pickle them to disk.

This uses cPickle for de/serializing, so objects must not contains unpicklable attributes, such as lambda functions etc.

class **load**(*fname*)
Load a previously saved object from file (also see *save*).

save (*fname*)
Save the object to file via pickling (also see *load*).

deaccent (*text*)
Remove accentuation from the given string.

Input text is either a unicode string or utf8 encoded bytestring. Return input string with accents removed, as unicode.

```
>>> deaccent("Šéf chomutovských komunistů dostal poštou bílý prášek")
u'Sef chomutovskych komunistu dostal postou bily prasek'
```

dictFromCorpus (*corpus*)
Scan corpus for all word ids that appear in it, then construct and return a mapping which maps each `wordId -> str(wordId)`.

This function is used whenever *words* need to be displayed (as opposed to just their ids) but no wordId->word mapping was provided. The resulting mapping only covers words actually used in the corpus, up to the highest wordId found.

getMaxId (*corpus*)
Return highest feature id that appears in the corpus.

For empty corpora (no features at all), return -1.

get_my_ip ()
Try to obtain our external ip (from the pyro nameserver's point of view)

This tries to sidestep the issue of bogus */etc/hosts* entries and other local misconfigurations, which often mess up hostname resolution.

If all else fails, fall back to simple *socket.gethostbyname()* lookup.

isCorpus (*obj*)
Check whether *obj* is a corpus.

NOTE: When called on an empty corpus (no documents), will return False.

synchronous (*tlockname*)
A decorator to place an instance based lock around a method.

Adapted from http://code.activestate.com/recipes/577105-synchronization-decorator-for-class-methods/

tokenize (*text, lowercase=False, deacc=False, errors='strict', toLower=False, lower=False*)
Iteratively yield tokens as unicode strings, optionally also lowercasing them and removing accent marks.

Input text may be either unicode or utf8-encoded byte string.

The tokens on output are maximal contiguous sequences of alphabetic characters (no digits!).

```
>>> list(tokenize('Nic nemůže letět rychlostí vyšší, než 300 tisíc kilometrů za sekundu!', deacc
[u'Nic', u'nemuze', u'letet', u'rychlosti', u'vyssi', u'nez', u'tisic', u'kilometru', u'za', u's
```

2.5.3 `matutils` – Math utils

This module contains math helper functions.

class MmReader (*fname*)
Wrap a term-document matrix on disk (in matrix-market format), and present it as an object which supports iteration over the rows (~documents).

Note that the file is read into memory one document at a time, not the whole matrix at once (unlike scipy.io.mmread). This allows for representing corpora which are larger than the available RAM.

Initialize the matrix reader.

The *fname* is a path to a file on local filesystem, which is expected to be in sparse (coordinate) Matrix Market format. Documents are assumed to be rows of the matrix (and document features are columns).

class **MmWriter** (*fname*)
Store corpus in Matrix Market format.

static **writeCorpus** (*fname, corpus*)
Save the vector space representation of an entire corpus to disk.

Note that the documents are processed one at a time, so the whole corpus is allowed to be larger than the available RAM.

writeVector (*docNo, vector*)
Write a single sparse vector to the file.

Sparse vector is any iterable yielding (field id, field value) pairs.

corpus2csc (*m, corpus, dtype=<type 'numpy.float64'>*)
Convert corpus into a sparse matrix, in scipy.sparse.csc_matrix format.

The corpus must not be empty (at least one document).

pad (*mat, padRow, padCol*)
Add additional rows/columns to a numpy.matrix *mat*. The new rows/columns will be initialized with zeros.

sparse2full (*doc, length*)
Convert document in sparse format (sequence of 2-tuples) into a full numpy array (of size *length*).

unitVec (*vec*)
Scale a vector to unit length. The only exception is the zero vector, which is returned back unchanged.

If the input is sparse (list of 2-tuples), output will also be sparse. Otherwise, output will be a numpy array.

2.5.4 corpora.bleicorpus – Corpus in Blei's LDA-C format

Blei's LDA-C format.

class **BleiCorpus** (*fname, fnameVocab=None*)
Corpus in Blei's LDA-C format.

The corpus is represented as two files: one describing the documents, and another describing the mapping between words and their ids.

Each document is one line:

```
N fieldId1:fieldValue1 fieldId2:fieldValue2 ... fieldIdN:fieldValueN
```

The vocabulary is a file with words, one word per line; word at line K has an implicit `id=K`.

Initialize the corpus from a file.

fnameVocab is the file with vocabulary; if not specified, it defaults to *fname.vocab*.

class **load** (*fname*)
Load a previously saved object from file (also see *save*).

save (*fname*)
Save the object to file via pickling (also see *load*).

static **saveCorpus**(*fname, corpus, id2word=None*)
Save a corpus in the Matrix Market format.

There are actually two files saved: *fname* and *fname.vocab*, where *fname.vocab* is the vocabulary file.

2.5.5 corpora.dictionary – Construct word<->id mappings

This module implements the concept of Dictionary – a mapping between words and their integer ids.

Dictionaries can be created from a corpus and can later be pruned according to document frequency (removing (un)common words via the `Dictionary.filterExtremes()` method), save/loaded from disk via `Dictionary.save()` and `Dictionary.load()` methods etc.

class Dictionary()
Dictionary encapsulates mappings between normalized words and their integer ids.

The main function is *doc2bow*, which converts a collection of words to its bag-of-words representation, optionally also updating the dictionary mapping with new words and their ids.

doc2bow(*document, allowUpdate=False*)
Convert *document* (a list of words) into the bag-of-words format = list of *(tokenId, tokenCount)* 2-tuples. Each word is assumed to be a **tokenized and normalized** utf-8 encoded string.

If *allowUpdate* is set, then also update of dictionary in the process: create ids for new words. At the same time, update document frequencies – for each word appearing in this document, increase its self.docFreq by one.

If *allowUpdate* is **not** set, this function is *const*, ie. read-only.

filterExtremes(*noBelow=5, noAbove=0.5*)
Filter out tokens that appear in

1. less than *noBelow* documents (absolute number) or
2. more than *noAbove* documents (fraction of total corpus size, *not* absolute number).

After the pruning, shrink resulting gaps in word ids.

Note: The same word may have a different word id before and after the call to this function!

filterTokens(*badIds*)
Remove the selected tokens from all dictionary mappings.

badIds is a collection of word ids to be removed.

static **fromDocuments**(*documents*)
Build dictionary from a collection of documents. Each document is a list of tokens (ie. **tokenized and normalized** utf-8 encoded strings).

This is only a convenience wrapper for calling *doc2bow* on each document with *allowUpdate=True*.

```
>>> print Dictionary.fromDocuments(["máma mele maso".split(), "ema má máma".split()])
Dictionary(5 unique tokens)
```

class **load**(*fname*)
Load a previously saved object from file (also see *save*).

rebuildDictionary()
Assign new word ids to all words.

This is done to make the ids more compact, ie. after some tokens have been removed via `filterTokens()` and there are gaps in the id series. Calling this method will remove the gaps.

save (*fname*)
Save the object to file via pickling (also see *load*).

2.5.6 corpora.dmlcorpus – Corpus in DML-CZ format

Corpus for the DML-CZ project.

class DmlConfig (*configId, resultDir, acceptLangs=None*)
DmlConfig contains parameters necessary for the abstraction of a 'corpus of articles' (see the *DmlCorpus* class).

Articles may come from different sources (=different locations on disk/netword, different file formats etc.), so the main purpose of DmlConfig is to keep all sources in one place.

Apart from glueing sources together, DmlConfig also decides where to store output files and which articles to accept for the corpus (= an additional filter over the sources).

class DmlCorpus ()
DmlCorpus implements a collection of articles. It is initialized via a DmlConfig object, which holds information about where to look for the articles and how to process them.

Apart from being a regular corpus (bag-of-words iterable with a *len()* method), DmlCorpus has methods for building a dictionary (mapping between words and their ids).

articleDir (*docNo*)
Return absolute normalized path on filesystem to article no. *docNo*.

buildDictionary ()
Populate dictionary mapping and statistics.

This is done by sequentially retrieving the article fulltexts, splitting them into tokens and converting tokens to their ids (creating new ids as necessary).

getMeta (*docNo*)
Return metadata for article no. *docNo*.

class **load** (*fname*)
Load a previously saved object from file (also see *save*).

processConfig (*config, shuffle=False*)
Parse the directories specified in the config, looking for suitable articles.

This updates the self.documents var, which keeps a list of (source id, article uri) 2-tuples. Each tuple is a unique identifier of one article.

Note that some articles are ignored based on config settings (for example if the article's language doesn't match any language specified in the config etc.).

save (*fname*)
Save the object to file via pickling (also see *load*).

saveAsText ()
Store the corpus to disk, in a human-readable text format.

This actually saves multiple files:

1. Pure document-term co-occurence frequency counts, as a Matrix Market file.
2. Token to integer mapping, as a text file.
3. Document to document URI mapping, as a text file.

The exact filesystem paths and filenames are determined from the config.

2.5.7 corpora.lowcorpus – Corpus in List-of-Words format

Corpus in GibbsLda++ format of List-Of-Words.

class LowCorpus (*fname, id2word=None, line2words=<function splitOnSpace at 0x18c1730>*)
List_Of_Words corpus handles input in GibbsLda++ format.

Quoting http://gibbslda.sourceforge.net/#3.2_Input_Data_Format:

```
Both data for training/estimating the model and new data (i.e., previously
unseen data) have the same format as follows:

[M]
[document1]
[document2]
...
[documentM]

in which the first line is the total number for documents [M]. Each line
after that is one document. [documenti] is the ith document of the dataset
that consists of a list of Ni words/terms.

[documenti] = [wordi1] [wordi2] ... [wordiNi]

in which all [wordij] (i=1..M, j=1..Ni) are text strings and they are separated
by the blank character.
```

Initialize the corpus from a file.

id2word and *line2words* are optional parameters.

If provided, *id2word* is a dictionary mapping between wordIds (integers) and words (strings). If not provided, the mapping is constructed from the documents.

line2words is a function which converts lines into tokens. Defaults to simple splitting on spaces.

class **load** (*fname*)
Load a previously saved object from file (also see *save*).

save (*fname*)
Save the object to file via pickling (also see *load*).

static **saveCorpus** (*fname, corpus, id2word=None*)
Save a corpus in the List-of-words format.

2.5.8 corpora.mmcorpus – Corpus in Matrix Market format

Corpus in the Matrix Market format.

class MmCorpus (*fname*)
Corpus in the Matrix Market format.

Initialize the matrix reader.

The *fname* is a path to a file on local filesystem, which is expected to be in sparse (coordinate) Matrix Market format. Documents are assumed to be rows of the matrix (and document features are columns).

class **load** (*fname*)
Load a previously saved object from file (also see *save*).

save (*fname*)
Save the object to file via pickling (also see *load*).

static **saveCorpus** (*fname, corpus, id2word=None*)
Save a corpus in the Matrix Market format to disk.

2.5.9 corpora.svmlightcorpus – Corpus in SVMlight format

Corpus in SVMlight format.

class SvmLightCorpus (*fname*)
Corpus in SVMlight format.

Quoting http://svmlight.joachims.org/: The input file example_file contains the training examples. The first lines may contain comments and are ignored if they start with #. Each of the following lines represents one training example and is of the following format:

```
<line> .=. <target> <feature>:<value> <feature>:<value> ... <feature>:<value> # <info>
<target> .=. +1 | -1 | 0 | <float>
<feature> .=. <integer> | "qid"
<value> .=. <float>
<info> .=. <string>
```

The "qid" feature (used for SVMlight ranking), if present, is ignored.

Although not mentioned in the specification above, SVMlight also expect its feature ids to be 1-based (counting starts at 1). We convert features to 0-base internally by decrementing all ids when loading a SVMlight input file, and increment them again when saving as SVMlight.

Initialize the corpus from a file.

class **load** (*fname*)
Load a previously saved object from file (also see *save*).

save (*fname*)
Save the object to file via pickling (also see *load*).

static **saveCorpus** (*fname, corpus, id2word=None*)
Save a corpus in the SVMlight format.

The SVMlight <*target*> class tag is set to 0 for all documents.

2.5.10 models.ldamodel – Latent Dirichlet Allocation

This module encapsulates functionality for the Latent Dirichlet Allocation algorithm.

It allows both model estimation from a training corpus and inference on new, unseen documents.

The implementation is based on Blei et al., Latent Dirichlet Allocation, 2003, and on Blei's LDA-C software in particular. This means it uses variational EM inference rather than Gibbs sampling to estimate model parameters.

class LdaModel (*corpus, id2word=None, numTopics=200, alpha=None, initMode='random'*)
Objects of this class allow building and maintaining a model of Latent Dirichlet Allocation.

The code is based on Blei's C implementation, see http://www.cs.princeton.edu/~blei/lda-c/ .

This Python code uses numpy heavily, and is about 4-5x slower than the original C version. The up side is that it is much more straightforward and concise, using vector operations ala MATLAB, easily pluggable/extensible etc.

The constructor estimates model parameters based on a training corpus:

```
>>> lda = LdaModel(corpus, numTopics = 10)
```

You can then infer topic distributions on new, unseen documents:

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

Model persistency is achieved via its load/save methods.

Initialize the model based on corpus.

id2word is a mapping from word ids (integers) to words (strings). It is used to determine the vocabulary size, as well as for debugging and topic printing.

numTopics is the number of requested topics.

alpha is either None (to be estimated during training) or a number between (0.0, 1.0).

computeLikelihood(*doc, phi, gamma*)
Compute the document likelihood, given all model parameters.

countsFromCorpus(*corpus, numInitDocs=1*)
Initialize the model word counts from the corpus. Each topic will be initialized from *numInitDocs* random documents.

docEStep(*doc*)
Find optimizing parameters for phi and gamma, and update sufficient statistics.

getTopicsMatrix()
Transform topic-word distribution via a tf-idf score and return it instead of the simple self.logProbW word-topic probabilities.

The transformation is a sort of TF-IDF score, where the word gets higher score if it's probable in this topic (the TF part) and lower score if it's probable across the whole corpus (the IDF part).

The exact formula is taken from Blei&Laffery: "Topic Models", 2009

The returned matrix is of the same shape as logProbW.

infer(*corpus*)
Convenience wrapper for making inference over a corpus of documents.

This means that a standard `inference()` step is taken for each document from the corpus and the results are saved into file *corpus.fname.lda_inferred*.

The output format of this file is one document per line:

```
doc_likelihood[TAB]topic1:prob topic2:prob ... topicK:prob[TAB]word1:topic word2:topic ... w
```

Topics are sorted by probability, words are in the same order as in the input.

inference(*doc*)
Perform inference on a single document.

Return 3-tuple of *(likelihood of this document, word-topic distribution phi, expected word counts gamma (~topic distribution))*.

A document is simply a bag-of-words collection which supports len() and iteration over (wordIndex, wordCount) 2-tuples.

The model itself is not affected in any way (this function is read-only aka const).

initialize(*corpus, initMode='random'*)
Run LDA parameter estimation from a training corpus, using the EM algorithm.

After the model has been initialized, you can infer topic distribution over other, different corpora, using this estimated model.

initMode can be either 'random', for a fast random initialization of the model parameters, or 'seeded', for an initialization based on a handful of real documents. The 'seeded' mode requires a sweep over the entire corpus, and is thus much slower.

class **load**(*fname*)
Load a previously saved object from file (also see *save*).

mle(*estimateAlpha*)
Maximum likelihood estimate.

This maximizes the lower bound on log likelihood wrt. to the alpha and beta parameters.

optAlpha(*MAX_ALPHA_ITER=1000, NEWTON_THRESH=1.0000000000000001e-05*)
Estimate new Dirichlet priors (actually just one scalar shared across all topics).

printTopics(*numWords=10*)
Print the top *numTerms* words for each topic, along with the log of their probability.

Uses getTopicsMatrix() method to determine the 'top words'.

save(*fname*)
Save the object to file via pickling (also see *load*).

2.5.11 models.lsimodel – Latent Semantic Indexing

Module for Latent Semantic Indexing.

class LsiModel(*corpus=None, id2word=None, numTopics=200, chunks=10000, decay=1.0, serial_only=None*)
Objects of this class allow building and maintaining a model for Latent Semantic Indexing (also known as Latent Semantic Analysis).

The main methods are:

1.constructor, which initializes the projection into latent topics space,

2.the `[]` method, which returns representation of any input document in the latent space,

3.the *addDocuments()* method, which allows for incrementally updating the model with new documents.

Model persistency is achieved via its load/save methods.

numTopics is the number of requested factors (latent dimensions).

After the model has been trained, you can estimate topics for an arbitrary, unseen document, using the `topics = self[document]` dictionary notation. You can also add new training documents, with `self.addDocuments`, so that training can be stopped and resumed at any time, and the LSI transformation is available at any point.

If you specify a *corpus*, it will be used to train the model. See the method *addDocuments* for a description of the *chunks* and *decay* parameters.

The algorithm will automatically try to find active nodes on other computers and run in a distributed manner; if this fails, it falls back to serial mode (single core). To suppress distributed computing, set the *serial_only* constructor parameter to True.

Example:

```
>>> lsi = LsiModel(corpus, numTopics = 10)
>>> print lsi[doc_tfidf]
>>> lsi.addDocuments(corpus2) # update LSI on additional documents
>>> print lsi[doc_tfidf]
```

addDocuments(*corpus, chunks=None, decay=None*)

Update singular value decomposition factors to take into account a new corpus of documents.

Training proceeds in chunks of *chunks* documents at a time. If the distributed mode is on, each chunk is sent to a different worker/computer. Size of *chunks* is a tradeoff between increased speed (bigger *chunks*) vs. lower memory footprint (smaller *chunks*). Default is processing 10,000 documents at a time.

Setting *decay* < 1.0 causes re-orientation towards new data trends in the input document stream, by giving less emphasis to old observations. This allows SVD to gradually "forget" old observations and give more preference to new ones. The decay is applied once after every *chunks* documents.

class **load**(*fname*)

Load a previously saved object from file (also see *save*).

printTopic(*topicNo, topN=10*)

Return a specified topic (0 <= *topicNo* < *self.numTopics*) as string in human readable format.

```
>>> lsimodel.printTopic(10, topN = 5)
'-0.340 * "category" + 0.298 * "$M$" + 0.183 * "algebra" + -0.174 * "functor" + -0.168 * "op
```

save(*fname*)

Save the object to file via pickling (also see *load*).

clipSpectrum(*s, k*)

Given singular values *s*, return how many factors should be kept to avoid storing spurious values. The returned value is at most *k*.

iterSvd(*corpus, numTerms, numFactors, numIter=200, initRate=None, convergence=0.0001*)

Perform iterative Singular Value Decomposition on a streaming corpus, returning *numFactors* greatest factors U,S,V^T (ie., not necessarily the full spectrum).

The parameters *numIter* (maximum number of iterations) and *initRate* (gradient descent step size) guide convergency of the algorithm.

The algorithm performs at most *numFactors*numIters* passes over the corpus.

See **Genevieve Gorrell: Generalized Hebbian Algorithm for Incremental Singular Value Decomposition in Natural Language Processing. EACL 2006.**

Use of this function is deprecated; although it works, it is several orders of magnitude slower than our own, direct (non-stochastic) version (which operates in a single pass, too, and can be distributed).

I keep this function here purely for backup reasons.

svdUpdate(*U, S, V, a, b*)

Update SVD of an (m x n) matrix *X = U * S * V^T* so that *[X + a * b^T] = U' * S' * V'^T* and return *U'*, *S'*, *V'*.

The original matrix X is not needed at all, so this function implements flexible *online* updates to an existing decomposition.

a and *b* are (m, 1) and (n, 1) matrices.

You can set V to None if you're not interested in the right singular vectors. In that case, the returned V' will also be None (saves memory).

This is the rank-1 update as described in *Brand, 2006: Fast low-rank modifications of the thin singular value decomposition*

2.5.12 models.tfidfmodel – TF-IDF model

class TfidfModel (*corpus, id2word=None, normalize=True*)

Objects of this class realize the transformation between word-document co-occurence matrix (integers) into a locally/globally weighted matrix (positive floats).

This is done by combining the term frequency counts (the TF part) with inverse document frequency counts (the IDF part), optionally normalizing the resulting documents to unit length.

The main methods are:

1. constructor, which calculates IDF weights for all terms in the training corpus.
2. the [] method, which transforms a simple count representation into the TfIdf space.

```
>>> tfidf = TfidfModel(corpus)
>>> print = tfidf[some_doc]
>>> tfidf.save('/tmp/foo.tfidf_model')
```

Model persistency is achieved via its load/save methods.

normalize dictates whether the resulting vectors will be set to unit length.

initialize (*corpus*)

Compute inverse document weights, which will be used to modify term frequencies for documents.

class **load** (*fname*)

Load a previously saved object from file (also see *save*).

save (*fname*)

Save the object to file via pickling (also see *load*).

2.5.13 models.rpmodel – Random Projections

class RpModel (*corpus, id2word=None, numTopics=300*)

Objects of this class allow building and maintaining a model for Random Projections (also known as Random Indexing). For theoretical background on RP, see:

Kanerva et al.: "Random indexing of text samples for Latent Semantic Analysis."

The main methods are:

1. constructor, which creates the random projection matrix
2. the [] method, which transforms a simple count representation into the TfIdf space.

```
>>> rp = RpModel(corpus)
>>> print rp[some_doc]
>>> rp.save('/tmp/foo.rp_model')
```

Model persistency is achieved via its load/save methods.

id2word is a mapping from word ids (integers) to words (strings). It is used to determine the vocabulary size, as well as for debugging and topic printing. If not set, it will be determined from the corpus.

initialize (*corpus*)

Initialize the random projection matrix.

class **load**(*fname*)
Load a previously saved object from file (also see *save*).

save(*fname*)
Save the object to file via pickling (also see *load*).

2.5.14 similarities.docsim – Document similarity queries

This module contains functions and classes for computing similarities across a collection of vectors=documents in the Vector Space Model.

The main classes are :

1. *Similarity* – answers similarity queries by linearly scanning over the corpus. This is slow but memory independent.
2. *MatrixSimilarity* – stores the whole corpus **in memory**, computes similarity by in-memory matrix-vector multiplication. This is much faster than the general *Similarity*, so use this when dealing with smaller corpora (must fit in RAM).
3. *SparseMatrixSimilarity* – same as *MatrixSimilarity*, but uses less memory if the vectors are sparse.

Once the similarity object has been initialized, you can query for document similarity simply by

```
>>> similarities = similarity_object[query_vector]
```

or iterate over within-corpus similarities with

```
>>> for similarities in similarity_object:
>>>     ...
```

---

class MatrixSimilarity(*corpus, numBest=None, dtype=<type 'numpy.float32'>, numFeatures=None*)
Compute similarity against a corpus of documents by storing its term-document (or concept-document) matrix in memory. The similarity measure used is cosine between two vectors.

This allows fast similarity searches (simple sparse matrix-vector multiplication), but loses the memory-independence of an iterative corpus.

The matrix is internally stored as a numpy array.

If *numBest* is left unspecified, similarity queries return a full list (one float for every document in the corpus, including the query document):

If *numBest* is set, queries return *numBest* most similar documents, as a sorted list:

```
>>> sms = MatrixSimilarity(corpus, numBest = 3)
>>> sms[vec12]
[(12, 1.0), (30, 0.95), (5, 0.45)]
```

getSimilarities(*doc*)
Return similarity of sparse vector *doc* to all documents in the corpus.

doc may be either a bag-of-words iterable (standard corpus document), or a numpy array, or a *scipy.sparse* matrix.

class **load**(*fname*)
Load a previously saved object from file (also see *save*).

save (*fname*)
Save the object to file via pickling (also see *load*).

class Similarity (*corpus, numBest=None*)
Compute cosine similarity against a corpus of documents. This is done by a full sequential scan of the corpus.

If your corpus is reasonably small (fits in RAM), consider using *MatrixSimilarity* or *SparseMatrixSimilarity* instead, for (much) faster similarity searches.

If *numBest* is left unspecified, similarity queries return a full list (one float for every document in the corpus, including the query document):

If *numBest* is set, queries return *numBest* most similar documents, as a sorted list:

```
>>> sms = MatrixSimilarity(corpus, numBest = 3)
>>> sms[vec12]
[(12, 1.0), (30, 0.95), (5, 0.45)]
```

class **load** (*fname*)
Load a previously saved object from file (also see *save*).

save (*fname*)
Save the object to file via pickling (also see *load*).

class SparseMatrixSimilarity (*corpus, numBest=None, dtype=<type 'numpy.float32'>*)
Compute similarity against a corpus of documents by storing its sparse term-document (or concept-document) matrix in memory. The similarity measure used is cosine between two vectors.

This allows fast similarity searches (simple sparse matrix-vector multiplication), but loses the memory-independence of an iterative corpus.

The matrix is internally stored as a *scipy.sparse.csr* matrix.

If *numBest* is left unspecified, similarity queries return a full list (one float for every document in the corpus, including the query document):

If *numBest* is set, queries return *numBest* most similar documents, as a sorted list:

```
>>> sms = SparseMatrixSimilarity(corpus, numBest = 3)
>>> sms[vec12]
[(12, 1.0), (30, 0.95), (5, 0.45)]
```

getSimilarities (*doc*)
Return similarity of sparse vector *doc* to all documents in the corpus.

doc may be either a bag-of-words iterable (standard corpus document), or a numpy array, or a *scipy.sparse* matrix.

class **load** (*fname*)
Load a previously saved object from file (also see *save*).

save (*fname*)
Save the object to file via pickling (also see *load*).

MODULE INDEX

G

INDEX

T

U

V

W