gensim Documentation

Release 0.4.3

Radim Řehůřek

March 30, 2010

CONTENTS

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 = 20)
>>>
>>> # convert corpus to latent space and index it
>>> index = similarities.SparseMatrixSimilarity(lsi[corpus])
>>>
>>> # perform similarity query of a new 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 and Latent Dirichlet Allocation. 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

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 topic inference algorithms, including Latent Semantic Analysis (LSA/LSI via SVD) and Latent Dirichlet Allocation (LDA), 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 and Mac OS X and should also run on Windows and any platform that supports Python 2.5 and NumPy. Gensim depends on the following software:

- 3.0 > Python >= 2.5. Tested with version 2.5.
- NumPy >= 1.2. Tested with version 1.3.0rc2.
- SciPy >= 0.7. Tested with version 0.7.1.

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!

There are also alternative routes:

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 source and run:

```
python setup.py test
```

2.2.5 Contact

If you encounter problems or have any questions regarding *gensim*, please let us know by emailing <radimrehurek(at)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 the Vector Space Model

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 >>>.

Quick Example

First, let's import some classes 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 (VSM), 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 texts and vectors in the second example a bit later.

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

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

to keep the corpus on disk, and access its documents sequentially, one at a time. All the operations and corpora transformations are implemented in such a way that makes them independent of the size of the corpus, RAM-wise.

Next, let's initialize a transformation:

```
>>> 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).

To index and prepare the whole TfIdf corpus for similarity queries:

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

and to query the similarity of our 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,
```

According to TfIdf and cosine similarity, 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.

A More Complete Example

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.

Firstly, 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) setup to mimick the experiment done in Deerwester et al.'s original LSA article [1].

The ways to process documents are so versatile 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; what follows is just one common scenario.

To convert documents to vectors, we will use a document representation called bag-of-words. In this representation, each vector element is a question-answer pair, in the style of:

> "How many times does the word *system* appear in the document? Once."

There are twelve distinct words in the preprocessed corpus, so each document will be represented by twelve numbers (ie., by a 12-D vector).

The `gensim.corpora.Dictionary` class can be used to convert tokenized texts to vectors:

```
>>> dictionary = corpora.Dictionary()
>>> corpus = [dictionary.doc2bow(text, allowUpdate = True) for text in texts]
```

Here we passed a list of tokens to `Dictionary.doc2bow()`, one list for each document. As a matter of fact, we have arrived at exactly the same corpus of vectors as in the first example, except that we now know what each vector dimension stands for:

```
>>> 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}
```

For example, the vector feature with `id=10` stands for the question "How many times does the word *graph* appear in the document?". The answer is "zero" for the first six documents and "one" for the remaining three.

```
>>> 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)]]
```

The function `doc2bow` simply counts the number of occurences of each distinct word, converts the word to its integer *question id* and returns the result as a sparse vector. With the `allowUpdate` option set, newly introduced words will be assigned a new id; otherwise, they are ignored. Put differently, this option decides whether new questions should be created upon encountering new words, or whether we're only interested in answering a fixed, pre-determined set of questions.

```
>>> newDoc = "Human computer interaction"
>>> newVec = dictionary.doc2bow(newDoc.lower().split(), allowUpdate = False)
>>> print newVec # the word "interaction" is ignored
[(0, 1), (1, 1)]
```

To finish the example, we transform our `"Human computer interaction"` document via Latent Semantic Indexing into a 2-D space:

```
>>> lsi = models.LsiModel(corpus, numTopics = 2)
>>> newVecLsi = lsi[newVec]
>>> print newVecLsi
[(0, -0.461821), (1, 0.0700277)]
```

and print proximity of this query document against our original corpus of nine documents:

```
>>> index = similarities.SparseMatrixSimilarity(lsi[corpus]) # "index" the corpus in LSI space
>>> print list(enumerate(index[newVecLsi])) # perform query against the corpus
[(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)]
```

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 high similarity scores, which corresponds better to our intuition of them sharing a "computer-related" topic with the query. In fact, this is the reason why we apply transformations and do topic modeling in the first place.

Corpus Formats

There exist several file formats for storing a collection 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 most notable formats is the Market Matrix format. To save a corpus in the Matrix Market format:

```
>>> from gensim import corpora
>>> corpora.MmCorpus.saveCorpus('/tmp/corpus.mm', corpus)
```

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

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

```
>>> corpus = corpora.MmCorpus('/tmp/corpus.mm')
>>> print list(corpus) # convert from MmCorpus object (document stream) to plain Python list
[[(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)]]
```

and to save it in Blei's LDA-C format again,

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

In this way, *gensim* can also be used as a simple I/O format conversion tool.

For a complete reference, see the *API documentation*.

2.3.2 Topics and Transformations

2.3.3 Similarity Queries

2.4 API Reference

Modules:

2.4.1 `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.

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, numBest=None*)
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 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).

Initialize the similarity search.

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[vec] # result in order of decreasing similarity
[(12, 1.0), (30, 0.95), (5, 0.45)]
```

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*).

2.4.2 `utils` – Various utility functions

This module contains various general utility functions.

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 contruct 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.

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

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

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
```

2.4.3 `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.

doc2vec(*doc, length*)
Convert document in sparse format (sequence of 2-tuples) into a full numpy array (of size *length*).

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

unitVec(*vec*)
Scale a sparse vector to another sparse vector of unit length.

2.4.4 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.

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

This module implements the concept of Dictionary – a mapping between words and their internal 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 coverts a collection of words to its bow 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).

```
>>> print Dictionary.fromDocuments(["máma mele maso".split(), "ema má mama".split()])
Dictionary(6 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*).

class Token (*token, intId*)
Object representing a single token.

2.4.6 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).

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).

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(*normalizeTfidf=False*)
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.

2.4.7 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 0x15b5f70>*)
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.

2.4.8 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.

2.4.9 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.

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.

2.4.10 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*)

Perform inference on 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 doc per line:: doc_likelihood[TAB]topic1:prob ... topicK:prob[TAB]word1:topic ... wordN:topic

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*).

2.4.11 models.lsimodel – Latent Semantic Indexing

Module for Latent Semantic Indexing.

class LsiModel (*corpus, id2word=None, numTopics=200*)
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 calculates the latent topics space, effectively initializing the model,

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

Model persistency is achieved via its load/save methods.

Find latent space based on the corpus provided.

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

After the model has been initialized, you can estimate topics for an arbitrary, unseen document, using the `topics = self[document]` dictionary notation.

Example:

```
>>> lsi = LsiModel(corpus, numTopics = 10)
>>> doc_lsi = lsi[doc_tfidf]
```

initialize (*corpus, chunks=100, keepDecomposition=False*)
Run SVD decomposition on the corpus. This will define the latent space into which terms and documents will be mapped.

The SVD is created incrementally, in blocks of *chunks* documents. In the end, a *self.projection* matrix is constructed that can be used to transform documents into the latent space. The *U, S, V* decomposition itself is discarded, unless *keepDecomposition* is True, in which case it is stored in *self.u*, *self.s* and *self.v*.

The algorithm is adapted from: **M. Brand. 2006. Fast low-rank modifications of the thin singular value decomposition**

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

printTopic (*topicNo, topN=10*)
Print a specified topic (0 <= *topicNo* < *self.numTopics*) in human readable format.

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

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

svdAddCols (*docs, decay=1.0, reorth=False*)
Update singular value decomposition factors to take into account new documents *docs*.

This function corresponds to the general update of Brand (section 2), specialized for *A* = *docs.T* and *B* trivial (no update to matrix rows).

The documents are assumed to be a list of full vectors (ie. not sparse 2-tuples).

Compute new decomposition *u'*, *s'*, *v'* so that if the current matrix *X* decomposes to *u * s * v^T ~= X*, then *u' * s' * v'^T ~= [X docs^T]*

u, *s*, *v* and their new values *u'*, *s'*, *v'* are stored within *self* (ie. as *self.u*, *self.v* etc.).

self.v can be set to *None*, in which case it is completely ignored. This saves a bit of speed and a lot of memory, especially for huge corpora (size of *v* is linear in the number of added documents).

iterSvd (*corpus, numTerms, numFactors, numIter=200, initRate=None, convergence=0.0001*)
Perform iterative Singular Value Decomposition on a streaming matrix (corpus), returning *numFactors* greatest factors (ie., not necessarily full spectrum).

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

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 the direct (non-stochastic) version based on Brand. Use svdAddCols/svdUpdate to compute SVD iteratively. 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'*.

a and *b* are (m, 1) and (n, 1) rank-1 matrices, so that svdUpdate can simulate incremental addition of one new document and/or term to an already existing decomposition.

2.4.12 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)
>>> doc_tfidf = tfidf[doc_tf]
```

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.

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*).

2.4.13 similarities.docsim – Pairwise similarity queries

This module contains functions and classes for similarities across a corpus of documents in the Vector Space Model.

The documents are sparse vectors coming from the TF-IDF model, LSI model, LDA model etc.

The two main classes are :

1. *Similarity* – computes similarity by linearly scanning over the corpus (slower,

   memory independent)

2. *SparseMatrixSimilarity* – 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, that fit in RAM.

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 Similarity (*corpus, numBest=None*)

Compute cosine similary 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 *SparseMatrixSimilarity* instead of *Similarity*, 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, eg. [(docIndex1, 1.0), (docIndex2, 0.95), ..., (docIndexnumBest, 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 for faster 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. It is assumed to be of unit length.

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*).

CHAPTER

THREE

INDICES AND TABLES

- *Index*
- *Module Index*
- *Search Page*

MODULE INDEX

G

INDEX