gensim Documentation

Release 0.4.5

Radim Řehůřek

April 03, 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 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 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 >= 1.0.4. Tested with version 1.3.0rc2 and 1.0.4.
- SciPy >= 0.6. Tested with version 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!

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 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*.

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 (optional), run

```
>>> import logging
>>> logging.root.level = logging.INFO # will suppress DEBUG level events
```

The examples are divided into parts on:

2.3.1 Corpora and Vector Spaces

Quick Example

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

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

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

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 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) and scales the resulting vector to unit length.

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.

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; what follows below 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."

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

```
>>> dictionary = corpora.Dictionary.fromDocuments(texts)
>>> print dictionary
Dictionary(12 unique tokens)
```

Here we assigned a unique integer id to all words appearing in the corpus by calling `fromDocuments()`. This sweeps across the texts, collecting words and relevant statistics. In the end, there are twelve distinct words in the preprocessed corpus, so 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:

```
>>> 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.

```
>>> corpus = [dictionary.doc2bow(text) for text in texts]
>>> 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)]]
```

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. As a matter of fact, we have arrived at exactly the same corpus of vectors as in the first example.

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 every one of the nine original documents, in the same 2-D LSI space:

```
>>> index = similarities.MatrixSimilarity(lsi[corpus]) # transform corpus to LSI space and "index" i
>>> sims = index[newVecLsi] # perform similarity query against the corpus
>>> print list(enumerate(sims))
[(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 more 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

2.4.2 `utils` – Various utility functions

2.4.3 `matutils` – Math utils

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

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

2.4.6 `corpora.dmlcorpus` – Corpus in DML-CZ format

2.4.7 `corpora.lowcorpus` – Corpus in List-of-Words format

2.4.8 `corpora.mmcorpus` – Corpus in Matrix Market format

2.4.9 `corpora.svmlightcorpus` – Corpus in SVMlight format

2.4.10 `models.ldamodel` – Latent Dirichlet Allocation

2.4.11 `models.lsimodel` – Latent Semantic Indexing

2.4.12 `models.tfidfmodel` – TF-IDF model

2.4.13 `similarities.docsim` – Document similarity queries

CHAPTER

THREE

INDICES AND TABLES

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

INDEX