# Topics and Transformations

**Note**

Click here to download the full example code

Introduces transformations and demonstrates their use on a toy corpus.

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

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

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

## Creating the Corpus

First, we need to create a corpus to work with. This step is the same as in the previous tutorial; if you completed it, feel free to skip to the next section.

```
from collections import defaultdict
from gensim import corpora

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",
]

# 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
frequency = defaultdict(int)
for text in texts:
    for token in text:
        frequency[token] += 1

texts = [
    [token for token in text if frequency[token] > 1]
    for text in texts
]

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

### Creating a transformation

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

```
from gensim import models
```

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

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

**Note**

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

### Transforming vectors

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

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

Out:
```
[(0, 0.7071067811865476), (1, 0.7071067811865476)]
```

Or to apply a transformation to a whole corpus:

```
corpus_tfidf = tfidf[corpus]
for doc in corpus_tfidf:
    print(doc)
```

Out:
```
[(0, 0.5773502691896257), (1, 0.5773502691896257), (2, 0.5773502691896257)]
[(0, 0.44424552527467476), (3, 0.44424552527467476), (4, 0.44424552527467476), (5, 0.3244870206138555), (6, 0.4442455
[(2, 0.5710059809418182), (5, 0.4170757362022777), (7, 0.4170757362022777), (8, 0.5710059809418182)]
[(1, 0.49182558987264147), (5, 0.7184811607083769), (8, 0.49182558987264147)]
[(3, 0.6282580468670046), (6, 0.6282580468670046), (7, 0.45889394536615247)]
[(9, 1.0)]
[(9, 0.7071067811865475), (10, 0.7071067811865475)]
[(9, 0.5080429008916749), (10, 0.5080429008916749), (11, 0.695546419520037)]
[(4, 0.6282580468670046), (10, 0.45889394536615247), (11, 0.6282580468670046)]
```

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

**Note**

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

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

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

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

```
lsi_model.print_topics(2)
```

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

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

```
# both bow->tfidf and tfidf->lsi transformations are actually executed here, on the fly
for doc, as_text in zip(corpus_lsi, documents):
    print(doc, as_text)
```

Out:
```
[(0, 0.06600783396090627), (1, -0.520070330636184)] Human machine interface for lab abc computer applications
[(0, 0.1966759285914279), (1, -0.760956316770005)] A survey of user opinion of computer system response time
```

```
[(0, 0.08992639972446735), (1, -0.7241860626752503)] The EPS user interface management system
[(0, 0.07585847652178428), (1, -0.6320551586003422)] System and human system engineering testing of EPS
[(0, 0.10150299184980327), (1, -0.5737308483002963)] Relation of user perceived response time to error measurement
[(0, 0.7032108939378309), (1, 0.16115180214026148)] The generation of random binary unordered trees
[(0, 0.8774787673119828), (1, 0.16758906864659825)] The intersection graph of paths in trees
[(0, 0.9098624686818573), (1, 0.14086553628719417)] Graph minors IV Widths of trees and well quasi ordering
[(0, 0.6165825350569281), (1, -0.05392907566389154)] Graph minors A survey
```

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

```
import os
import tempfile

with tempfile.NamedTemporaryFile(prefix='model-', suffix='.lsi', delete=False) as tmp:
    lsi_model.save(tmp.name)  # same for tfidf, lda, ...

loaded_lsi_model = models.LsiModel.load(tmp.name)

os.unlink(tmp.name)
```

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 (**Similarity Queries**).

## 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 = models.TfidfModel(corpus, normalize=True)
  ```

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

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

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

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

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

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

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

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

  ```
  model = models.RpModel(tfidf_corpus, num_topics=500)
  ```

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

executes a stochastic multi-pass algorithm from Halko et al. **4** internally, to accelerate in-core part of the computations. See also **Experiments on the English Wikipedia** for further speed-ups by distributing the computation across a cluster of computers.

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

```
model = models.RpModel(tfidf_corpus, num_topics=500)
```

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

```
model = models.LdaModel(tfidf_corpus, num_topics=100)
```

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

## What Next?

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

## References


**1**

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

**2**

Hoffman, Blei, Bach. 2010. Online learning for Latent Dirichlet Allocation.

**3**

Wang, Paisley, Blei. 2011. Online variational inference for the hierarchical Dirichlet process.

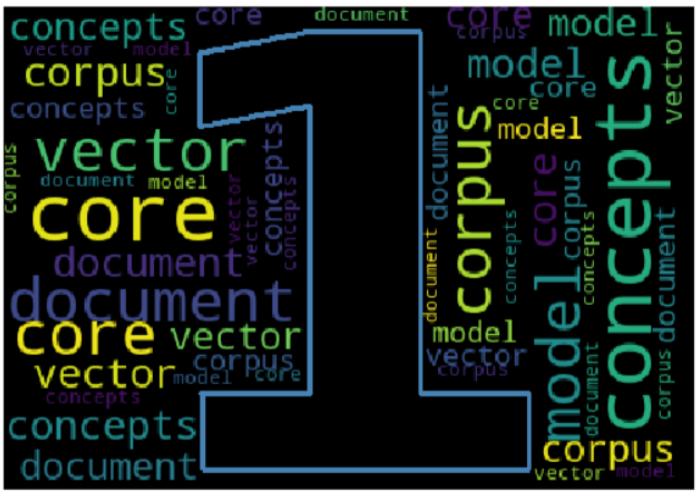
**4**

Halko, Martinsson, Tropp. 2009. Finding structure with randomness.

**5**

Řehůřek. 2011. Subspace tracking for Latent Semantic Analysis.


```
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
img = mpimg.imread('run_topics_and_transformations.png')
imgplot = plt.imshow(img)
plt.axis('off')
plt.show()
```




Out:

```
/home/misha/git/gensim/docs/src/gallery/core/run_topics_and_transformations.py:293: UserWarning: Matplotlib is curren
  plt.show()
```

**Total running time of the script:** ( 0 minutes 0.844 seconds)

**Estimated memory usage:** 44 MB

Download Python source code: run_topics_and_transformations.py

Download Jupyter notebook: run_topics_and_transformations.ipynb

Gallery generated by Sphinx-Gallery



## Support:

Stay informed via gensim mailing list:

your@email.com
Subscribe

0



Tweet @Gensim_py