gensim
topic modelling for humans

Get Expert Help
- machine learning, NLP, data mining
- custom SW design, development, optimizations
- corporate trainings & IT consulting

models.doc2vec – Deep learning with paragraph2vec

Deep learning via the distributed memory and distributed bag of words models from [1], using either hierarchical softmax or negative sampling [2] [3]. See [tutorial]

Make sure you have a C compiler before installing gensim, to use optimized (compiled) doc2vec training (70x speedup [blog]).

Initialize a model with e.g.:

```
>>> model = Doc2Vec(documents, size=100, window=8, min_count=5, workers=4)
```

Persist a model to disk with:

```
>>> model.save(fname)
>>> model = Doc2Vec.load(fname)  # you can continue training with the loaded model!
```

If you're finished training a model (=no more updates, only querying), you can do

```
>>> model.delete_temporary_training_data(keep_doctags_vectors=True, keep_inference=True):
```

to trim unneeded model memory = use (much) less RAM.

[1] Quoc Le and Tomas Mikolov. Distributed Representations of Sentences and Documents. http://arxiv.org/pdf/1405.4053v2.pdf
[2] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient Estimation of Word Representations in Vector Space. In Proceedings of Workshop at ICLR, 2013.
[3] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed Representations of Words and Phrases and their Compositionality. In Proceedings of NIPS, 2013.
[blog] Optimizing word2vec in gensim, http://radimrehurek.com/2013/09/word2vec-in-python-part-two-optimizing/
[tutorial] Doc2vec in gensim tutorial, https://github.com/RaRe-Technologies/gensim/blob/develop/docs/notebooks/doc2vec-lee.ipynb

class `gensim.models.doc2vec.Doc2Vec`(*documents=None*, *dm_mean=None*, *dm=1*, *dbow_words=0*, *dm_concat=0*, *dm_tag_count=1*, *docvecs=None*, *docvecs_mapfile=None*, *comment=None*, *trim_rule=None*, ***kwargs*)

Bases: `gensim.models.word2vec.Word2Vec`

Class for training, using and evaluating neural networks described in http://arxiv.org/pdf/1405.4053v2.pdf

Initialize the model from an iterable of *documents*. Each document is a TaggedDocument object that will be used for training.

The *documents* iterable can be simply a list of TaggedDocument elements, but for larger corpora, consider an iterable that streams the documents directly from disk/network.

If you don't supply *documents*, the model is left uninitialized – use if you plan to initialize it in some other way.

dm defines the training algorithm. By default (*dm=1*), 'distributed memory' (PV-DM) is used. Otherwise, *distributed bag of words* (PV-DBOW) is employed.

size is the dimensionality of the feature vectors.

window is the maximum distance between the predicted word and context words used for prediction within a document.

alpha is the initial learning rate (will linearly drop to zero as training progresses).

seed = for the random number generator. Note that for a fully deterministically-reproducible run, you must also limit the model to a single worker thread, to eliminate ordering jitter from OS thread scheduling. (In Python 3, reproducibility between interpreter launches also requires use of the PYTHONHASHSEED environment variable to control hash randomization.)

min_count = ignore all words with total frequency lower than this.

max_vocab_size = limit RAM during vocabulary building; if there are more unique words than this, then prune the infrequent ones. Every 10 million word types need about 1GB of RAM. Set to *None* for no limit (default).

sample = threshold for configuring which higher-frequency words are randomly downsampled;
default is 0 (off), useful value is 1e-5.

workers = use this many worker threads to train the model (=faster training with multicore machines).

iter = number of iterations (epochs) over the corpus. The default inherited from Word2Vec is 5, but values of 10 or 20 are common in published 'Paragraph Vector' experiments.

hs = if 1 (default), hierarchical sampling will be used for model training (else set to 0).

negative = if > 0, negative sampling will be used, the int for negative specifies how many "noise words" should be drawn (usually between 5-20).

dm_mean = if 0 (default), use the sum of the context word vectors. If 1, use the mean. Only applies when dm is used in non-concatenative mode.

dm_concat = if 1, use concatenation of context vectors rather than sum/average; default is 0 (off). Note concatenation results in a much-larger model, as the input is no longer the size of one (sampled or arithmatically combined) word vector, but the size of the tag(s) and all words in the context strung together.

dm_tag_count = expected constant number of document tags per document, when using dm_concat mode; default is 1.

dbow_words if set to 1 trains word-vectors (in skip-gram fashion) simultaneous with DBOW doc-vector training; default is 0 (faster training of doc-vectors only).

trim_rule = vocabulary trimming rule, specifies whether certain words should remain in the vocabulary, be trimmed away, or handled using the default (discard if word count < min_count). Can be None (min_count will be used), or a callable that accepts parameters (word, count, min_count) and returns either util.RULE_DISCARD, util.RULE_KEEP or util.RULE_DEFAULT. Note: The rule, if given, is only used prune vocabulary during build_vocab() and is not stored as part of the model.

`accuracy`(*questions*, *restrict_vocab=30000*, *most_similar=None*, *case_insensitive=True*)

`build_vocab`(*sentences*, *keep_raw_vocab=False*, *trim_rule=None*, *progress_per=10000*, *update=False*)

> Build vocabulary from a sequence of sentences (can be a once-only generator stream). Each sentence must be a list of unicode strings.

`clear_sims`()

`create_binary_tree`()

> Create a binary Huffman tree using stored vocabulary word counts. Frequent words will have shorter binary codes. Called internally from *build_vocab()*.

`dbow`

`delete_temporary_training_data`(*keep_doctags_vectors=True*, *keep_inference=True*)

> Discard parameters that are used in training and score. Use if you're sure you're done training a model. Set *keep_doctags_vectors* to False if you don't want to save doctags vectors, in this case you can't to use docvecs's most_similar, similarity etc. methods. Set *keep_inference* to False if you don't want to store parameters that is used for infer_vector method

---

`dm`

`doesnt_match`(*words*)

`estimate_memory`(*vocab_size=None*, *report=None*)

> Estimate required memory for a model using current settings.

`evaluate_word_pairs`(*pairs*, *delimiter='\t'*, *restrict_vocab=300000*, *case_insensitive=True*, *dummy4unknown=False*)

`finalize_vocab`(*update=False*)

> Build tables and model weights based on final vocabulary settings.

`infer_vector`(*doc_words*, *alpha=0.1*, *min_alpha=0.0001*, *steps=5*)

> Infer a vector for given post-bulk training document.
>
> Document should be a list of (word) tokens.

`init_sims`(*replace=False*)

> init_sims() resides in KeyedVectors because it deals with syn0 mainly, but because syn1 is not an attribute of KeyedVectors, it has to be deleted in this class, and the normalizing of syn0 happens inside of KeyedVectors

---

`initialize_word_vectors`()

`intersect_word2vec_format`(*fname*, *lockf=0.0*, *binary=False*, *encoding='utf8'*, *unicode_errors='strict'*)

Merge the input-hidden weight matrix from the original C word2vec-tool format given, where it intersects with the current vocabulary. (No words are added to the existing vocabulary, but intersecting words adopt the file's weights, and non-intersecting words are left alone.)

binary is a boolean indicating whether the data is in binary word2vec format.

lockf is a lock-factor value to be set for any imported word-vectors; the default value of 0.0 prevents further updating of the vector during subsequent training. Use 1.0 to allow further training updates of merged vectors.

`load`(**args*, ***kwargs*)

`load_word2vec_format`(*fname*, *fvocab=None*, *binary=False*, *encoding='utf8'*, *unicode_errors='strict'*, *limit=None*, *datatype=<type 'numpy.float32'>*)

Deprecated. Use gensim.models.KeyedVectors.load_word2vec_format instead.

`log_accuracy`(*section*)

`log_evaluate_word_pairs`(*pearson*, *spearman*, *oov*, *pairs*)

`make_cum_table`(*power=0.75*, *domain=2147483647*)

Create a cumulative-distribution table using stored vocabulary word counts for drawing random words in the negative-sampling training routines.

To draw a word index, choose a random integer up to the maximum value in the table (cum_table[-1]), then finding that integer's sorted insertion point (as if by bisect_left or ndarray.searchsorted()). That insertion point is the drawn index, coming up in proportion equal to the increment at that slot.

Called internally from 'build_vocab()'.

`most_similar`(*positive=[]*, *negative=[]*, *topn=10*, *restrict_vocab=None*, *indexer=None*)

`most_similar_cosmul`(*positive=[]*, *negative=[]*, *topn=10*)

`n_similarity`(*ws1*, *ws2*)

`reset_from`(*other_model*)

Reuse shareable structures from other_model.

`reset_weights`()

`save`(**args*, ***kwargs*)

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

fname_or_handle is either a string specifying the file name to save to, or an open file-like object which can be written to. If the object is a file handle, no special array handling will be performed; all attributes will be saved to the same file.

If *separately* is None, automatically detect large numpy/scipy.sparse arrays in the object being stored, and store them into separate files. This avoids pickle memory errors and allows mmap'ing large arrays back on load efficiently.

You can also set *separately* manually, in which case it must be a list of attribute names to be stored in separate files. The automatic check is not performed in this case.

ignore is a set of attribute names to *not* serialize (file handles, caches etc). On subsequent load() these attributes will be set to None.

pickle_protocol defaults to 2 so the pickled object can be imported in both Python 2 and 3.

`save_word2vec_format`(*fname*, *fvocab=None*, *binary=False*)

Deprecated. Use model.wv.save_word2vec_format instead.

`scale_vocab`(*min_count=None*, *sample=None*, *dry_run=False*, *keep_raw_vocab=False*, *trim_rule=None*, *update=False*)

Apply vocabulary settings for *min_count* (discarding less-frequent words) and *sample* (controlling the downsampling of more-frequent words).

Calling with *dry_run=True* will only simulate the provided settings and report the size of the retained vocabulary, effective corpus length, and estimated memory requirements. Results are both printed via logging and returned as a dict.

Delete the raw vocabulary after the scaling is done to free up RAM, unless *keep_raw_vocab* is set.

`scan_vocab`(*documents*, *progress_per=10000*, *trim_rule=None*, *update=False*)

`score`(*sentences*, *total_sentences=1000000*, *chunksize=100*, *queue_factor=2*, *report_delay=1*)

Score the log probability for a sequence of sentences (can be a once-only generator stream). Each sentence must be a list of unicode strings. This does not change the fitted model in any way (see Word2Vec.train() for that).

We have currently only implemented score for the hierarchical softmax scheme, so you need to have run word2vec with hs=1 and negative=0 for this to work.

Note that you should specify total_sentences; we'll run into problems if you ask to score more than this number of sentences but it is inefficient to set the value too high.

See the article by [taddy] and the gensim demo at [deepir] for examples of how to use such scores in document classification.

[taddy] Taddy, Matt. Document Classification by Inversion of Distributed Language Representations, in Proceedings of the 2015 Conference of the Association of Computational Linguistics.
[deepir] https://github.com/piskvorky/gensim/blob/develop/docs/notebooks/deepir.ipynb

`seeded_vector`(*seed_string*)

Create one 'random' vector (but deterministic by seed_string)

`similar_by_vector`(*vector*, *topn=10*, *restrict_vocab=None*)

`similar_by_word`(*word*, *topn=10*, *restrict_vocab=None*)

`similarity`(*w1*, *w2*)

`sort_vocab`()

Sort the vocabulary so the most frequent words have the lowest indexes.

`train`(*sentences*, *total_words=None*, *word_count=0*, *total_examples=None*, *queue_factor=2*, *report_delay=1.0*)

Update the model's neural weights from a sequence of sentences (can be a once-only generator stream). For Word2Vec, each sentence must be a list of unicode strings. (Subclasses may accept other examples.)

To support linear learning-rate decay from (initial) alpha to min_alpha, either total_examples (count of sentences) or total_words (count of raw words in sentences) should be provided, unless the sentences are the same as those that were used to initially build the vocabulary.

`update_weights`()

Copy all the existing weights, and reset the weights for the newly added vocabulary.

`wmdistance`(*document1*, *document2*)

class `gensim.models.doc2vec.Doctag`

Bases: `gensim.models.doc2vec.Doctag`

A string document tag discovered during the initial vocabulary scan. (The document-vector equivalent of a Vocab object.)

Will not be used if all presented document tags are ints.

The offset is only the true index into the doctags_syn0/doctags_syn0_lockf if-and-only-if no raw-int tags were used. If any raw-int tags were used, string Doctag vectors begin at index (max_rawint + 1), so the true index is (rawint_index + 1 + offset). See also DocvecsArray.index_to_doctag().

Create new instance of Doctag(offset, word_count, doc_count)

`count`(*value*) → integer -- return number of occurrences of value

`doc_count`

Alias for field number 2

`index`(*value*[, *start*[, *stop*]]) → integer -- return first index of value.

Raises ValueError if the value is not present.

`offset`

Alias for field number 0

`repeat`(*word_count*)

`word_count`

Alias for field number 1

class `gensim.models.doc2vec.DocvecsArray`(*mapfile_path=None*)

Bases: `gensim.utils.SaveLoad`

Default storage of doc vectors during/after training, in a numpy array.

As the 'docvecs' property of a Doc2Vec model, allows access and comparison of document vectors.

```
>>> docvec = d2v_model.docvecs[99]
>>> docvec = d2v_model.docvecs['SENT_99']  # if string tag used in training
>>> sims = d2v_model.docvecs.most_similar(99)
>>> sims = d2v_model.docvecs.most_similar('SENT_99')
>>> sims = d2v_model.docvecs.most_similar(docvec)
```

If only plain int tags are presented during training, the dict (of string tag -> index) and list (of index -> string tag) stay empty, saving memory.

Supplying a mapfile_path (as by initializing a Doc2Vec model with a 'docvecs_mapfile' value) will use a pair of memory-mapped files as the array backing for doctag_syn0/doctag_syn0_lockf values.

The Doc2Vec model automatically uses this class, but a future alternative implementation, based on another persistence mechanism like LMDB, LevelDB, or SQLite, should also be possible.

`borrow_from`(*other_docvecs*)

`clear_sims`()

`doesnt_match`(*docs*)

Which doc from the given list doesn't go with the others?

(TODO: Accept vectors of out-of-training-set docs, as if from inference.)

`estimated_lookup_memory`()

Estimated memory for tag lookup; 0 if using pure int tags.

`index_to_doctag`(*i_index*)

Return string key for given i_index, if available. Otherwise return raw int doctag (same int).

`indexed_doctags`(*doctag_tokens*)

Return indexes and backing-arrays used in training examples.

`init_sims`(*replace=False*)

Precompute L2-normalized vectors.

If *replace* is set, forget the original vectors and only keep the normalized ones = saves lots of memory!

Note that you **cannot continue training or inference** after doing a replace. The model becomes effectively read-only = you can call *most_similar*, *similarity* etc., but not *train* or *infer_vector*.

`load`(*fname*, *mmap=None*)

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

If the object was saved with large arrays stored separately, you can load these arrays via mmap (shared memory) using *mmap='r'*. Default: don't use mmap, load large arrays as normal objects.

If the file being loaded is compressed (either '.gz' or '.bz2'), then *mmap=None* must be set. Load will raise an *IOError* if this condition is encountered.

`most_similar`(*positive=[]*, *negative=[]*, *topn=10*, *clip_start=0*, *clip_end=None*, *indexer=None*)

Find the top-N most similar docvecs known from training. Positive docs contribute positively towards the similarity, negative docs negatively.

This method computes cosine similarity between a simple mean of the projection weight vectors of the given docs. Docs may be specified as vectors, integer indexes of trained docvecs, or if the documents were originally presented with string tags, by the corresponding tags.

The 'clip_start' and 'clip_end' allow limiting results to a particular contiguous range of the underlying doctag_syn0norm vectors. (This may be useful if the ordering there was chosen to be significant, such as more popular tag IDs in lower indexes.)

`n_similarity`(*ds1*, *ds2*)

Compute cosine similarity between two sets of docvecs from the trained set, specified by int index or string tag. (TODO: Accept vectors of out-of-training-set docs, as if from inference.)

`note_doctag`(*key*, *document_no*, *document_length*)

Note a document tag during initial corpus scan, for structure sizing.

`reset_weights`(*model*)

`save`(**args*, ***kwargs*)

`similarity`(*d1*, *d2*)

Compute cosine similarity between two docvecs in the trained set, specified by int index or string tag. (TODO: Accept vectors of out-of-training-set docs, as if from inference.)

`similarity_unseen_docs`(*model*, *doc_words1*, *doc_words2*, *alpha=0.1*, *min_alpha=0.0001*, *steps=5*)

Compute cosine similarity between two post-bulk out of training documents.

Document should be a list of (word) tokens.

`trained_item`(*indexed_tuple*)

Persist any changes made to the given indexes (matching tuple previously returned by indexed_doctags()); a no-op for this implementation

class `gensim.models.doc2vec.LabeledSentence`(**args*, ***kwargs*)

Bases: `gensim.models.doc2vec.TaggedDocument`

Create new instance of TaggedDocument(words, tags)

`count`(*value*) → integer -- return number of occurrences of value

`index`(*value*[, *start*[, *stop*]]) → integer -- return first index of value.

Raises ValueError if the value is not present.

`tags`

Alias for field number 1

`words`

Alias for field number 0

class `gensim.models.doc2vec.TaggedBrownCorpus`(*dirname*)

Bases: `object`

Iterate over documents from the Brown corpus (part of NLTK data), yielding each document out as a TaggedDocument object.

class `gensim.models.doc2vec.TaggedDocument`

Bases: `gensim.models.doc2vec.TaggedDocument`

A single document, made up of *words* (a list of unicode string tokens) and *tags* (a list of tokens). Tags may be one or more unicode string tokens, but typical practice (which will also be most memory-efficient) is for the tags list to include a unique integer id as the only tag.

Replaces "sentence as a list of words" from Word2Vec.

Create new instance of TaggedDocument(words, tags)

`count`(*value*) → integer -- return number of occurrences of value

`index`(*value*[, *start*[, *stop*]]) → integer -- return first index of value.

Raises ValueError if the value is not present.

`tags`

Alias for field number 1

`words`

Alias for field number 0

class `gensim.models.doc2vec.TaggedLineDocument`(*source*)

Bases: `object`

Simple format: one document = one line = one TaggedDocument object.

Words are expected to be already preprocessed and separated by whitespace, tags are constructed automatically from the document line number.

source can be either a string (filename) or a file object.

Example:

```
documents = TaggedLineDocument('myfile.txt')
```

Or for compressed files:

```
documents = TaggedLineDocument('compressed_text.txt.bz2')
documents = TaggedLineDocument('compressed_text.txt.gz')
```

gensim

Home | Tutorials | Install | Support | API | About

Support:

Stay informed via gensim mailing list:

your@email.com
Subscribe

0

Tweet @RadimRehurek