`models.doc2vec` – Doc2vec paragraph embeddings

Introduction

Learn paragraph and document embeddings via the distributed memory and distributed bag of words models from Quoc Le and Tomas Mikolov: "Distributed Representations of Sentences and Documents".

The algorithms use either hierarchical softmax or negative sampling; see Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean: "Efficient Estimation of Word Representations in Vector Space, in Proceedings of Workshop at ICLR, 2013" and 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".

For a usage example, see the Doc2vec tutorial.

Make sure you have a C compiler before installing Gensim, to use the optimized doc2vec routines (70x speedup compared to plain NumPy implementation, https://rare-technologies.com/parallelizing-word2vec-in-python/).

Usage examples

Initialize & train a model:

```
>>> from gensim.test.utils import common_texts
>>> from gensim.models.doc2vec import Doc2Vec, TaggedDocument
>>>
>>> documents = [TaggedDocument(doc, [i]) for i, doc in
enumerate(common_texts)]
>>> model = Doc2Vec(documents, vector_size=5, window=2,
min_count=1, workers=4)
```

Persist a model to disk:

```
>>> from gensim.test.utils import get_tmpfile
>>>
>>> fname = get_tmpfile("my_doc2vec_model")
>>>
>>> model.save(fname)
>>> model = Doc2Vec.load(fname)  # you can continue training
with the loaded model!
```

Infer vector for a new document:

```
>>> vector = model.infer_vector(["system", "response"])
```

class **`gensim.models.doc2vec.Doc2Vec`**(*documents=None, corpus_file=None, vector_size=100, dm_mean=None, dm=1, dbow_words=0, dm_concat=0, dm_tag_count=1, dv=None, dv_mapfile=None, comment=None, trim_rule=None, callbacks=(), window=5, epochs=10, shrink_windows=True, **kwargs*)

Bases: `gensim.models.word2vec.Word2Vec`

Class for training, using and evaluating neural networks described in Distributed Representations of Sentences and Documents.

Parameters:

- **documents** (iterable of list of `TaggedDocument`, optional) – Input corpus, can be simply a list of elements, but for larger corpora,consider an iterable that streams the documents directly from disk/network. If you don't supply *documents* (or *corpus_file*), the model is left uninitialized – use if you plan to initialize it in some other way.
- **corpus_file** (*str, optional*) – Path to a corpus file in `LineSentence` format. You may use this argument instead of *documents* to get performance boost. Only one of *documents* or *corpus_file* arguments need to be passed (or none of them, in that case, the model is left uninitialized). Documents' tags are assigned automatically and are equal to line number, as in `TaggedLineDocument`.
- **dm** (*{1,0}, optional*) – Defines the training algorithm. If *dm=1*, 'distributed memory' (PV-DM) is used. Otherwise, *distributed bag of words* (PV-DBOW) is employed.

- **vector_size** (*int, optional*) – Dimensionality of the feature vectors.
- **window** (*int, optional*) – The maximum distance between the current and predicted word within a sentence.
- **alpha** (*float, optional*) – The initial learning rate.
- **min_alpha** (*float, optional*) – Learning rate will linearly drop to *min_alpha* as training progresses.
- **seed** (*int, optional*) – Seed for the random number generator. Initial vectors for each word are seeded with a hash of the concatenation of word + *str(seed)*. Note that for a fully deterministically-reproducible run, you must also limit the model to a single worker thread (*workers=1*), 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** (*int, optional*) – Ignores all words with total frequency lower than this.
- **max_vocab_size** (*int, optional*) – Limits the 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.
- **sample** (*float, optional*) – The threshold for configuring which higher-frequency words are randomly downsampled, useful range is (0, 1e-5).
- **workers** (*int, optional*) – Use these many worker threads to train the model (=faster training with multicore machines).
- **epochs** (*int, optional*) – Number of iterations (epochs) over the corpus. Defaults to 10 for Doc2Vec.
- **hs** (*{1,0}, optional*) – If 1, hierarchical softmax will be used for model training. If set to 0, and *negative* is

non-zero, negative sampling will be used.

- **negative** (*int, optional*) – If > 0, negative sampling will be used, the int for negative specifies how many "noise words" should be drawn (usually between 5-20). If set to 0, no negative sampling is used.
- **ns_exponent** (*float, optional*) – The exponent used to shape the negative sampling distribution. A value of 1.0 samples exactly in proportion to the frequencies, 0.0 samples all words equally, while a negative value samples low-frequency words more than high-frequency words. The popular default value of 0.75 was chosen by the original Word2Vec paper. More recently, in https://arxiv.org/abs/1804.04212, Caselles-Dupré, Lesaint, & Royo-Letelier suggest that other values may perform better for recommendation applications.
- **dm_mean** (*{1,0}, optional*) – If 0 , 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** (*{1,0}, optional*) – If 1, use concatenation of context vectors rather than sum/average; Note concatenation results in a much-larger model, as the input is no longer the size of one (sampled or arithmetically combined) word vector, but the size of the tag(s) and all words in the context strung together.
- **dm_tag_count** (*int, optional*) – Expected constant number of document tags per document, when using dm_concat mode.
- **dbow_words** (*{1,0}, optional*) – If set to 1 trains word-vectors (in skip-gram fashion) simultaneous with DBOW doc-vector training; If 0, only trains doc-vectors (faster).
- **trim_rule** (*function, optional*) – 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, look to `keep_vocab_item()`), or a callable that accepts parameters (word, count, min_count) and returns either `gensim.utils.RULE_DISCARD`, `gensim.utils.RULE_KEEP` or `gensim.utils.RULE_DEFAULT`. The rule, if given, is only used to prune vocabulary during current method call and is not stored as part of the model.

The input parameters are of the following types:

  - *word* (str) - the word we are examining
  - *count* (int) - the word's frequency count in the corpus
  - *min_count* (int) - the minimum count threshold.

- **callbacks** – List of callbacks that need to be executed/run at specific stages during training.
- **shrink_windows** (*bool, optional*) – New in 4.1. Experimental. If True, the effective window size is uniformly sampled from [1, *window*] for each target word during training, to match the original word2vec algorithm's approximate weighting of context words by distance. Otherwise, the effective window size is always fixed to *window* words to either side.
- **important internal attributes are the following** (*Some*) –

`wv`

This object essentially contains the mapping between words and embeddings. After training, it can be used directly to query those embeddings in various ways. See the module level docstring for examples.

Type: `KeyedVectors`

`dv`

This object contains the paragraph vectors learned from the training data. There will be one such vector for each unique document tag supplied during training. They may be individually accessed using the tag as an indexed-access key. For example, if one of the training documents used a tag of 'doc003':

```
>>> model.dv['doc003']
```

Type: `KeyedVectors`

`add_lifecycle_event`(*event_name, log_level=20, **event*)

Append an event into the *lifecycle_events* attribute of this object, and also optionally log the event at *log_level*.

Events are important moments during the object's life, such as "model created", "model saved", "model loaded", etc.

The *lifecycle_events* attribute is persisted across object's `save()` and `load()` operations. It has no impact on the use of the model, but is useful during debugging and support.

Set *self.lifecycle_events = None* to disable this behaviour. Calls to *add_lifecycle_event()* will not record events into *self.lifecycle_events* then.

Parameters:

- **event_name** (*str*) – Name of the event. Can be any label, e.g. "created", "stored" etc.
- **event** (*dict*) –
  Key-value mapping to append to *self.lifecycle_events*. Should be JSON-serializable, so keep it simple. Can be empty.

  This method will automatically add the following key-values to *event*, so you don't have to specify them:
  - *datetime*: the current date & time
  - *gensim*: the current Gensim version
  - *python*: the current Python version
  - *platform*: the current platform
  - *event*: the name of this event
- **log_level** (*int*) – Also log the complete event dict, at the specified log level. Set to False to not log at all.

`add_null_word()`

`build_vocab`(*corpus_iterable=None*, *corpus_file=None*, *update=False*, *progress_per=10000*, *keep_raw_vocab=False*, *trim_rule=None*, ***kwargs*)

Build vocabulary from a sequence of documents (can be a once-only generator stream).

Parameters:

- **documents** (iterable of list of `TaggedDocument`, optional) – Can be simply a list of `TaggedDocument` elements, but for larger corpora, consider an iterable that streams the documents directly from disk/network. See `TaggedBrownCorpus` or `TaggedLineDocument`
- **corpus_file** (*str, optional*) – Path to a corpus file in `LineSentence` format. You may use this argument instead of *documents* to get performance boost. Only one of *documents* or *corpus_file* arguments need to be passed (not both of them). Documents' tags are assigned automatically and are equal to a line number, as in `TaggedLineDocument`.
- **update** (*bool*) – If true, the new words in *documents* will be added to model's vocab.
- **progress_per** (*int*) – Indicates how many words to process before showing/updating the progress.
- **keep_raw_vocab** (*bool*) – If not true, delete the raw vocabulary after the scaling is done and free up RAM.
- **trim_rule** (*function, optional*) – 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, look to `keep_vocab_item()`), or a callable that accepts parameters (word, count, min_count) and returns either `gensim.utils.RULE_DISCARD`, `gensim.utils.RULE_KEEP` or

`gensim.utils.RULE_DEFAULT`. The rule, if given, is only used to prune vocabulary during current method call and is not stored as part of the model.

The input parameters are of the following types:

  - *word* (str) – the word we are examining
  - *count* (int) – the word's frequency count in the corpus
  - *min_count* (int) – the minimum count threshold.

- ****kwargs** – Additional key word arguments passed to the internal vocabulary construction.

build_vocab_from_freq(*word_freq, keep_raw_vocab=False, corpus_count=None, trim_rule=None, update=False*)

Build vocabulary from a dictionary of word frequencies.

Build model vocabulary from a passed dictionary that contains a (word -> word count) mapping. Words must be of type unicode strings.

Parameters:

- **word_freq** (*dict of (str, int)*) – Word <-> count mapping.
- **keep_raw_vocab** (*bool, optional*) – If not true, delete the raw vocabulary after the scaling is done and free up RAM.
- **corpus_count** (*int, optional*) – Even if no corpus is provided, this argument can set corpus_count explicitly.
- **trim_rule** (*function, optional*) – 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, look to `keep_vocab_item()`), or a callable that accepts parameters (word, count, min_count) and returns either `gensim.utils.RULE_DISCARD`, `gensim.utils.RULE_KEEP` or

`gensim.utils.RULE_DEFAULT`. The rule, if given, is only used to prune vocabulary during `build_vocab()` and is not stored as part of the model.

The input parameters are of the following types:

- *word* (str) - the word we are examining
- *count* (int) - the word's frequency count in the corpus
- *min_count* (int) - the minimum count threshold.

- **update** (*bool, optional*) – If true, the new provided words in *word_freq* dict will be added to model's vocab.

`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()`.

property **`dbow`**

Indicates whether 'distributed bag of words' (PV-DBOW) will be used, else 'distributed memory' (PV-DM) is used.

property **`dm`**

Indicates whether 'distributed memory' (PV-DM) will be used, else 'distributed bag of words' (PV-DBOW) is used.

property **`docvecs`**

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

Estimate required memory for a model using current settings.

Parameters:

- **vocab_size** (*int, optional*) – Number of raw words in the vocabulary.
- **report** (*dict of (str, int), optional*) – A dictionary from string representations of the **specific** model's memory consuming members to their size in bytes.

Returns: A dictionary from string representations of the model’s memory consuming members to their size in bytes. Includes members from the base classes as well as weights and tag lookup memory estimation specific to the class.

Return type: dict of (str, int), optional

`estimated_lookup_memory`()

Get estimated memory for tag lookup, 0 if using pure int tags.

Returns: The estimated RAM required to look up a tag in bytes.

Return type: int

`get_latest_training_loss`()

Get current value of the training loss.

Returns: Current training loss.

Return type: float

`infer_vector`(*doc_words*, *alpha=None*, *min_alpha=None*, *epochs=None*)

Infer a vector for given post-bulk training document.

Notes

Subsequent calls to this function may infer different representations for the same document. For a more stable representation, increase the number of epochs to assert a stricter convergence.

Parameters:

- **doc_words** (*list of str*) – A document for which the vector representation will be inferred.
- **alpha** (*float, optional*) – The initial learning rate. If unspecified, value from model initialization will be reused.
- **min_alpha** (*float, optional*) – Learning rate will linearly drop to *min_alpha* over all inference epochs. If unspecified, value from model initialization will be reused.
- **epochs** (*int, optional*) – Number of times to train the new document. Larger values take more time, but may improve quality and run-to-

run stability of inferred vectors. If unspecified, the *epochs* value from model initialization will be reused.

Returns: The inferred paragraph vector for the new document.

Return type: np.ndarray

`init_sims`(*replace=False*)

Precompute L2-normalized vectors. Obsoleted.

If you need a single unit-normalized vector for some key, call `get_vector()` instead: `doc2vec_model.dv.get_vector(key, norm=True)`.

To refresh norms after you performed some atypical out-of-band vector tampering, call *:meth:`~gensim.models.keyedvectors.KeyedVectors.fill_norms()* instead.

Parameters: **replace** (*bool*) – If True, forget the original trained vectors and only keep the normalized ones. You lose information if you do this.

`init_weights`()

Reset all projection weights to an initial (untrained) state, but keep the existing vocabulary.

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

Load a previously saved `Doc2Vec` model.

Parameters:
- **fname** (*str*) – Path to the saved file.
- ***args** (*object*) – Additional arguments, see *~gensim.models.word2vec.Word2Vec.load*.
- ****kwargs** (*object*) – Additional arguments, see *~gensim.models.word2vec.Word2Vec.load*.

See also

`save()`

Save `Doc2Vec` model.

Returns: Loaded model.

Return type: `Doc2Vec`

`make_cum_table(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.

`predict_output_word(context_words_list, topn=10)`

Get the probability distribution of the center word given context words.

Note this performs a CBOW-style propagation, even in SG models, and doesn't quite weight the surrounding words the same as in training – so it's just one crude way of using a trained model as a predictor.

Parameters:
- **context_words_list** (*list of (str and/or int)*) – List of context words, which may be words themselves (str) or their index in *self.wv.vectors* (int).
- **topn** (*int, optional*) – Return *topn* words and their probabilities.

Returns: *topn* length list of tuples of (word, probability).

Return type: list of (str, float)

`prepare_vocab(update=False, keep_raw_vocab=False, trim_rule=None, min_count=None, sample=None, dry_run=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.

`prepare_weights(update=False)`

Build tables and model weights based on final vocabulary settings.

`reset_from`(*other_model*)

Copy shareable data structures from another (possibly pre-trained) model.

This specifically causes some structures to be shared, so is limited to structures (like those rleated to the known word/tag vocabularies) that won't change during training or thereafter. Beware vocabulary edits/updates to either model afterwards: the partial sharing and out-of-band modification may leave the other model in a broken state.

Parameters: **other_model** (`Doc2Vec`) – Other model whose internal data structures will be copied over to the current object.

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

Save the model. This saved model can be loaded again using `load()`, which supports online training and getting vectors for vocabulary words.

Parameters: **fname** (*str*) – Path to the file.

`save_word2vec_format`(*fname, doctag_vec=False, word_vec=True, prefix='*dt_', fvocab=None, binary=False*)

Store the input-hidden weight matrix in the same format used by the original C word2vec-tool.

Parameters:

- **fname** (*str*) – The file path used to save the vectors in.
- **doctag_vec** (*bool, optional*) – Indicates whether to store document vectors.
- **word_vec** (*bool, optional*) – Indicates whether to store word vectors.
- **prefix** (*str, optional*) – Uniquely identifies doctags from word vocab, and avoids collision in case of repeated string in doctag and word vocab.
- **fvocab** (*str, optional*) – Optional file path used to save the vocabulary.
- **binary** (*bool, optional*) – If True, the data will be saved in binary word2vec format, otherwise - will

be saved in plain text.

`scan_vocab`(*corpus_iterable=None*, *corpus_file=None*, *progress_per=10000*, *trim_rule=None*)

Create the models Vocabulary: A mapping from unique words in the corpus to their frequency count.

Parameters:

- **documents** (iterable of `TaggedDocument`, optional) – The tagged documents used to create the vocabulary. Their tags can be either str tokens or ints (faster).
- **corpus_file** (*str, optional*) – Path to a corpus file in `LineSentence` format. You may use this argument instead of *documents* to get performance boost. Only one of *documents* or *corpus_file* arguments need to be passed (not both of them).
- **progress_per** (*int*) – Progress will be logged every *progress_per* documents.
- **trim_rule** (*function, optional*) – 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, look to `keep_vocab_item()`), or a callable that accepts parameters (word, count, min_count) and returns either `gensim.utils.RULE_DISCARD`, `gensim.utils.RULE_KEEP` or `gensim.utils.RULE_DEFAULT`. The rule, if given, is only used to prune vocabulary during `build_vocab()` and is not stored as part of the model.

  The input parameters are of the following types:

  - *word* (str) - the word we are examining
  - *count* (int) - the word's frequency count in the corpus
  - *min_count* (int) - the minimum count threshold.

Returns: Tuple of (Total words in the corpus, number of documents)

Return type: (int, int)

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

Score the log probability for a sequence of sentences. This does not change the fitted model in any way (see `train()` for that).

Gensim has 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*; you'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 Matt Taddy: "Document Classification by Inversion of Distributed Language Representations" and the gensim demo for examples of how to use such scores in document classification.

Parameters:

- **sentences** (*iterable of list of str*) – The *sentences* iterable can be simply a list of lists of tokens, but for larger corpora, consider an iterable that streams the sentences directly from disk/network. See `BrownCorpus`, `Text8Corpus` or `LineSentence` in `word2vec` module for such examples.
- **total_sentences** (*int, optional*) – Count of sentences.
- **chunksize** (*int, optional*) – Chunksize of jobs
- **queue_factor** (*int, optional*) – Multiplier for size of queue (number of workers * queue_factor).
- **report_delay** (*float, optional*) – Seconds to wait before reporting progress.

`seeded_vector`(*seed_string, vector_size*)

`similarity_unseen_docs`(*doc_words1, doc_words2, alpha=None, min_alpha=None, epochs=None*)

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

Parameters:

- **model** (`Doc2Vec`) – An instance of a trained *Doc2Vec* model.
- **doc_words1** (*list of str*) – Input document.
- **doc_words2** (*list of str*) – Input document.
- **alpha** (*float, optional*) – The initial learning rate.
- **min_alpha** (*float, optional*) – Learning rate will linearly drop to *min_alpha* as training progresses.
- **epochs** (*int, optional*) – Number of epoch to train the new document.

Returns: The cosine similarity between *doc_words1* and *doc_words2*.

Return type: float

train(*corpus_iterable=None, corpus_file=None, total_examples=None, total_words=None, epochs=None, start_alpha=None, end_alpha=None, word_count=0, queue_factor=2, report_delay=1.0, callbacks=(), **kwargs*)

Update the model's neural weights.

To support linear learning-rate decay from (initial) *alpha* to *min_alpha*, and accurate progress-percentage logging, either *total_examples* (count of documents) or *total_words* (count of raw words in documents) **MUST** be provided. If *documents* is the same corpus that was provided to `build_vocab()` earlier, you can simply use *total_examples=self.corpus_count*.

To avoid common mistakes around the model's ability to do multiple training passes itself, an explicit *epochs* argument **MUST** be provided. In the common and recommended case where `train()` is only called once, you can set *epochs=self.iter*.

Parameters:

- **corpus_iterable** (iterable of list of `TaggedDocument`, optional) – Can be simply a list of elements, but for larger corpora,consider an iterable that streams the documents directly from disk/network. If you don't supply *documents* (or *corpus_file*), the model is left uninitialized – use if you plan to initialize it in some other way.

- **corpus_file** (*str, optional*) – Path to a corpus file in `LineSentence` format. You may use this argument instead of *documents* to get performance boost. Only one of *documents* or *corpus_file* arguments need to be passed (not both of them). Documents' tags are assigned automatically and are equal to line number, as in `TaggedLineDocument`.
- **total_examples** (*int, optional*) – Count of documents.
- **total_words** (*int, optional*) – Count of raw words in documents.
- **epochs** (*int, optional*) – Number of iterations (epochs) over the corpus.
- **start_alpha** (*float, optional*) – Initial learning rate. If supplied, replaces the starting *alpha* from the constructor, for this one call to *train*. Use only if making multiple calls to *train*, when you want to manage the alpha learning-rate yourself (not recommended).
- **end_alpha** (*float, optional*) – Final learning rate. Drops linearly from *start_alpha*. If supplied, this replaces the final *min_alpha* from the constructor, for this one call to `train()`. Use only if making multiple calls to `train()`, when you want to manage the alpha learning-rate yourself (not recommended).
- **word_count** (*int, optional*) – Count of words already trained. Set this to 0 for the usual case of training on all words in documents.
- **queue_factor** (*int, optional*) – Multiplier for size of queue (number of workers * queue_factor).
- **report_delay** (*float, optional*) – Seconds to wait before reporting progress.
- **callbacks** – List of callbacks that need to be executed/run at specific stages during training.

`update_weights`()

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

class **`gensim.models.doc2vec.Doc2VecTrainables`**

Bases: `gensim.utils.SaveLoad`

Obsolete class retained for now as load-compatibility state capture

`add_lifecycle_event`(*event_name*, *log_level=20*, ***event*)

Append an event into the *lifecycle_events* attribute of this object, and also optionally log the event at *log_level*.

Events are important moments during the object's life, such as "model created", "model saved", "model loaded", etc.

The *lifecycle_events* attribute is persisted across object's `save()` and `load()` operations. It has no impact on the use of the model, but is useful during debugging and support.

Set *self.lifecycle_events = None* to disable this behaviour. Calls to *add_lifecycle_event()* will not record events into *self.lifecycle_events* then.

Parameters:

- **event_name** (*str*) – Name of the event. Can be any label, e.g. "created", "stored" etc.
- **event** (*dict*) –
  Key-value mapping to append to *self.lifecycle_events*. Should be JSON-serializable, so keep it simple. Can be empty.

  This method will automatically add the following key-values to *event*, so you don't have to specify them:
  - *datetime*: the current date & time
  - *gensim*: the current Gensim version
  - *python*: the current Python version
  - *platform*: the current platform
  - *event*: the name of this event
- **log_level** (*int*) – Also log the complete event dict, at the specified log level. Set to False to not log at all.

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

Load an object previously saved using `save()` from a file.

Parameters:

- **fname** (*str*) – Path to file that contains needed object.
- **mmap** (*str, optional*) – Memory-map option. If the object was saved with large arrays stored separately, you can load these arrays via mmap (shared memory) using *mmap='r'. If the file being loaded is compressed (either '.gz' or '.bz2'), then `mmap=None* **must be** set.

See also

`save()`

Save object to file.

Returns: Object loaded from *fname*.

Return type: object

Raises: **AttributeError** – When called on an object instance instead of class (this is a class method).

`save`(*fname_or_handle*, *separately=None*, *sep_limit=10485760*, *ignore=frozenset({})*, *pickle_protocol=4*)

Save the object to a file.

Parameters:

- **fname_or_handle** (*str or file-like*) – Path to output file or already opened file-like object. If the object is a file handle, no special array handling will be performed, all attributes will be saved to the same file.
- **separately** (*list of str or None, optional*) –
If None, automatically detect large numpy/scipy.sparse arrays in the object being stored, and store them into separate files. This prevent memory errors for large objects, and also allows memory-mapping the large arrays for

efficient loading and sharing the large arrays in RAM between multiple processes.

If list of str: store these attributes into separate files. The automated size check is not performed in this case.

- **sep_limit** (*int, optional*) – Don't store arrays smaller than this separately. In bytes.
- **ignore** (*frozenset of str, optional*) – Attributes that shouldn't be stored at all.
- **pickle_protocol** (*int, optional*) – Protocol number for pickle.

See also

`load()`

Load object from file.

class `gensim.models.doc2vec.Doc2VecVocab`

Bases: `gensim.utils.SaveLoad`

Obsolete class retained for now as load-compatibility state capture

`add_lifecycle_event`(*event_name*, *log_level=20*, ***event*)

Append an event into the *lifecycle_events* attribute of this object, and also optionally log the event at *log_level*.

Events are important moments during the object's life, such as "model created", "model saved", "model loaded", etc.

The *lifecycle_events* attribute is persisted across object's `save()` and `load()` operations. It has no impact on the use of the model, but is useful during debugging and support.

Set *self.lifecycle_events = None* to disable this behaviour. Calls to *add_lifecycle_event()* will not record events into *self.lifecycle_events* then.

Parameters:

- **event_name** (*str*) – Name of the event. Can be any label, e.g. "created", "stored" etc.
- **event** (*dict*) –

Key-value mapping to append to *self.lifecycle_events*. Should be JSON-serializable, so keep it simple. Can be empty.

This method will automatically add the following key-values to *event*, so you don't have to specify them:

- *datetime*: the current date & time
- *gensim*: the current Gensim version
- *python*: the current Python version
- *platform*: the current platform
- *event*: the name of this event

- **log_level** (*int*) – Also log the complete event dict, at the specified log level. Set to False to not log at all.

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

Load an object previously saved using `save()` from a file.

Parameters:

- **fname** (*str*) – Path to file that contains needed object.
- **mmap** (*str, optional*) – Memory-map option. If the object was saved with large arrays stored separately, you can load these arrays via mmap (shared memory) using *mmap='r'. If the file being loaded is compressed (either '.gz' or '.bz2'), then `mmap=None* **must be** set.

See also

`save()`

Save object to file.

Returns: Object loaded from *fname*.

Return type: object

Raises: **AttributeError** – When called on an object instance instead of class (this is a class method).

save(*fname_or_handle, separately=None, sep_limit=10485760, ignore=frozenset({}), pickle_protocol=4*)

Save the object to a file.

Parameters:

- **fname_or_handle** (*str or file-like*) – Path to output file or already opened file-like object. If the object is a file handle, no special array handling will be performed, all attributes will be saved to the same file.
- **separately** (*list of str or None, optional*) –
  If None, automatically detect large numpy/scipy.sparse arrays in the object being stored, and store them into separate files. This prevent memory errors for large objects, and also allows memory-mapping the large arrays for efficient loading and sharing the large arrays in RAM between multiple processes.

  If list of str: store these attributes into separate files. The automated size check is not performed in this case.
- **sep_limit** (*int, optional*) – Don't store arrays smaller than this separately. In bytes.
- **ignore** (*frozenset of str, optional*) – Attributes that shouldn't be stored at all.
- **pickle_protocol** (*int, optional*) – Protocol number for pickle.

See also

`load()`

Load object from file.

class `gensim.models.doc2vec.Doctag`(*doc_count: int, index: int, word_count: int*)

Bases: `object`

A dataclass shape-compatible with keyedvectors.SimpleVocab, extended to record details of string document tags discovered during the initial vocabulary scan.

Will not be used if all presented document tags are ints. No longer used in a completed model: just used during initial scan, and for backward compatibility.

property **`count`**

`doc_count`

`index`

`word_count`

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

Bases: `object`

Reader for the Brown corpus (part of NLTK data).

Parameters: **dirname** (*str*) – Path to folder with Brown corpus.

class **`gensim.models.doc2vec.TaggedDocument`**(*words, tags*)

Bases: `gensim.models.doc2vec.TaggedDocument`

Represents a document along with a tag, input document format for `Doc2Vec`.

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 the 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 `gensim.models.word2vec.Word2Vec`.

Create new instance of TaggedDocument(words, tags)

`count`(*value*) → integer–returnnumberofoccurrencesofvalue

`index`(*value* [, *start* [, *stop*]]) → integer–returnfirstindexofvalue.

Raises ValueError if the value is not present.

property **`tags`**

Alias for field number 1

property **`words`**

Alias for field number 0

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

Bases: `object`

Iterate over a file that contains documents: one line = `TaggedDocument` object.

Words are expected to be already preprocessed and separated by whitespace. Document tags are constructed automatically from the document line number (each document gets a unique integer tag).

Parameters: **source** (*string or a file-like object*) – Path to the file on disk, or an already-open file object (must support *seek(0)*).

Examples

```
>>> from gensim.test.utils import datapath
>>> from gensim.models.doc2vec import TaggedLineDocument
>>>
>>> for document in
TaggedLineDocument(datapath("head500.noblanks.cor")):
...     pass
```