gensim
topic modelling for humans

Get Expert Help

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

models.wrappers.fasttext – FastText Word Embeddings

Warning

Deprecated since version 3.2.0: Use gensim.models.fasttext.FastText instead of gensim.models.wrappers.fasttext.FastText.

Python wrapper around word representation learning from FastText, a library for efficient learning of word representations and sentence classification [1].

This module allows training a word embedding from a training corpus with the additional ability to obtain word vectors for out-of-vocabulary words, using the fastText C implementation.

The wrapped model can NOT be updated with new documents for online training – use gensim's *Word2Vec* for that.

Example:

```
>>> from gensim.models.wrappers import FastText
>>> model = FastText.train('/Users/kofola/fastText/fasttext', corpus_file='text8')
>>> print model['forests']  # prints vector for given out-of-vocabulary word
```

[1] https://github.com/facebookresearch/fastText#enriching-word-vectors-with-subword-information

class gensim.models.wrappers.fasttext.FastText(*sentences=None*, *size=100*, *alpha=0.025*, *window=5*, *min_count=5*, *max_vocab_size=None*, *sample=0.001*, *seed=1*, *workers=3*, *min_alpha=0.0001*, *sg=0*, *hs=0*, *negative=5*, *cbow_mean=1*, *hashfxn=<built-in function hash>*, *iter=5*, *null_word=0*, *trim_rule=None*, *sorted_vocab=1*, *batch_words=10000*, *compute_loss=False*)

Bases: gensim.models.word2vec.Word2Vec

Class for word vector training using FastText. Communication between FastText and Python takes place by working with data files on disk and calling the FastText binary with subprocess.call(). Implements functionality similar to [fasttext.py](https://github.com/salestock/fastText.py), improving speed and scope of functionality like *most_similar*, *similarity* by extracting vectors into numpy matrix.

Warning

Deprecated since version 3.2.0: Use gensim.models.fasttext.FastText instead of gensim.models.wrappers.fasttext.FastText.

Initialize the model from an iterable of *sentences*. Each sentence is a list of words (unicode strings) that will be used for training.

The *sentences* iterable can be simply a list, but for larger corpora, consider an iterable that streams the sentences directly from disk/network. See BrownCorpus, Text8Corpus or LineSentence in this module for such examples.

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

sg defines the training algorithm. By default (*sg=0*), CBOW is used. Otherwise (*sg=1*), skip-gram is employed.

size is the dimensionality of the feature vectors.

window is the maximum distance between the current and predicted word within a sentence.

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

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, 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 1e-3, useful range is (0, 1e-5).

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

hs = if 1, hierarchical softmax will be used for model training. If set to 0 (default), and *negative* is non-zero, negative sampling will be used.

negative = if > 0, negative sampling will be used, the int for negative specifies how many "noise words" should be drawn (usually between 5-20). Default is 5. If set to 0, no negative samping is used.

cbow_mean = if 0, use the sum of the context word vectors. If 1 (default), use the mean. Only applies when cbow is used.

hashfxn = hash function to use to randomly initialize weights, for increased training reproducibility. Default is Python's rudimentary built in hash function.

iter = number of iterations (epochs) over the corpus. Default is 5.

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 *utils.RULE_DISCARD*, *utils.RULE_KEEP* or *utils.RULE_DEFAULT*. Note: The rule, if given, is only used to prune vocabulary during build_vocab() and is not stored as part of the model.

sorted_vocab = if 1 (default), sort the vocabulary by descending frequency before assigning word indexes.

batch_words = target size (in words) for batches of examples passed to worker threads (and thus cython routines). Default is 10000. (Larger batches will be passed if individual texts are longer than 10000 words, but the standard cython code truncates to that maximum.)

---

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.

---

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 (word,word count). Words must be of type unicode strings.

Parameters:

- **word_freq** (*dict*) – Word,Word_Count dictionary.
- **keep_raw_vocab** (*bool*) – If not true, delete the raw vocabulary after the scaling is done and free up RAM.
- **corpus_count** (*int*) – Even if no corpus is provided, this argument can set corpus_count explicitly.
- **= vocabulary trimming rule, specifies whether certain words should remain** (*trim_rule*) –
- **the vocabulary, be trimmed away, or handled using the default (discard if word count < min_count)** (*in*) –
- **be None (min_count will be used), or a callable that accepts parameters (word, count, min_count) and** (*Can*) –
- **either utils.RULE_DISCARD, utils.RULE_KEEP or utils.RULE_DEFAULT.** (*returns*) –
- **update** (*bool*) – If true, the new provided words in *word_freq* dict will be added to model's vocab.

Returns:

Return type: None

Examples

```
>>> from gensim.models.word2vec import Word2Vec
>>> model= Word2Vec()
>>> model.build_vocab_from_freq({"Word1": 15, "Word2": 20})
```

clear_sims()

Removes all L2-normalized vectors for words from the model. You will have to recompute them using init_sims method.

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()*.

delete_temporary_training_data(*replace_word_vectors_with_normalized=False*)

Discard parameters that are used in training and score. Use if you're sure you're done training a model. If *replace_word_vectors_with_normalized* is set, forget the original vectors and only keep the normalized ones = saves lots of memory!

classmethod delete_training_files(*model_file*)

Deletes the files created by FastText training doesnt_match(**args*, ***kwargs*)

Deprecated. Use self.wv.doesnt_match() instead. Refer to the documentation for *gensim.models.KeyedVectors.doesnt_match*

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

Estimate required memory for a model using current settings and provided vocabulary size.

evaluate_word_pairs(**args*, ***kwargs*)

Deprecated. Use self.wv.evaluate_word_pairs() instead. Refer to the documentation for *gensim.models.KeyedVectors.evaluate_word_pairs*

finalize_vocab(*update=False*)

Build tables and model weights based on final vocabulary settings.

get_latest_training_loss()

init_ngrams()

Computes ngrams of all words present in vocabulary and stores vectors for only those ngrams. Vectors for other ngrams are initialized with a random uniform distribution in FastText. These vectors are discarded here to save space.

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.

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

---

load_binary_data(*encoding='utf8'*)

Loads data from the output binary file created by FastText training

---

load_dict(*file_handle*, *encoding='utf8'*)

classmethod load_fasttext_format(*model_file*, *encoding='utf8'*)

Load the input-hidden weight matrix from the fast text output files.

Note that due to limitations in the FastText API, you cannot continue training with a model loaded this way, though you can query for word similarity etc.

model_file is the path to the FastText output files. FastText outputs two model files - */path/to/model.vec* and */path/to/model.bin*

Expected value for this example: */path/to/model* or */path/to/model.bin*, as gensim requires only *.bin* file to load entire fastText model.

---

load_model_params(*file_handle*)

---

load_vectors(*file_handle*)

---

load_word2vec_format(**args*, ***kwargs*)

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

---

log_accuracy(*section*)

---

log_evaluate_word_pairs(**args*, ***kwargs*)

Deprecated. Use self.wv.log_evaluate_word_pairs() instead. Refer to the documentation for *gensim.models.KeyedVectors.log_evaluate_word_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(**args*, ***kwargs*)

Deprecated. Use self.wv.most_similar() instead. Refer to the documentation for *gensim.models.KeyedVectors.most_similar*

---

most_similar_cosmul(**args*, ***kwargs*)

Deprecated. Use self.wv.most_similar_cosmul() instead. Refer to the documentation for *gensim.models.KeyedVectors.most_similar_cosmul*

---

n_similarity(**args*, ***kwargs*)

Deprecated. Use self.wv.n_similarity() instead. Refer to the documentation for *gensim.models.KeyedVectors.n_similarity*

---

predict_output_word(*context_words_list*, *topn=10*)

Report the probability distribution of the center word given the context words as input to the trained model.

---

reset_from(*other_model*)

Borrow shareable pre-built structures (like vocab) from the other_model. Useful if testing multiple models in parallel on the same corpus.

---

reset_weights()

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

---

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

---

save_word2vec_format(**args*, ***kwargs*)

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(*sentences*, *progress_per=10000*, *trim_rule=None*)

Do an initial scan of all words appearing in sentences.

---

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 [2] and the gensim demo at [3] for examples of how to use such scores in document classification.

[2] Taddy, Matt. Document Classification by Inversion of Distributed Language Representations, in Proceedings of the 2015 Conference of the Association of Computational Linguistics.
[3] 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(**args*, ***kwargs*)

Deprecated. Use self.wv.similar_by_vector() instead. Refer to the documentation for *gensim.models.KeyedVectors.similar_by_vector*

---

similar_by_word(**args*, ***kwargs*)

Deprecated. Use self.wv.similar_by_word() instead. Refer to the documentation for *gensim.models.KeyedVectors.similar_by_word*

---

similarity(**args*, ***kwargs*)

Deprecated. Use self.wv.similarity() instead. Refer to the documentation for *gensim.models.KeyedVectors.similarity*

---

sort_vocab()

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

---

struct_unpack(*file_handle*, *fmt*)

classmethod train(*ft_path*, *corpus_file*, *output_file=None*, *model='cbow'*, *size=100*, *alpha=0.025*, *window=5*, *min_count=5*, *word_ngrams=1*, *loss='ns'*, *sample=0.001*, *negative=5*, *iter=5*, *min_n=3*, *max_n=6*, *sorted_vocab=1*, *threads=12*)

ft_path is the path to the FastText executable, e.g. */home/kofola/fastText/fasttext*.

corpus_file is the filename of the text file to be used for training the FastText model. Expects file to contain utf-8 encoded text.

model defines the training algorithm. By default, cbow is used. Accepted values are 'cbow', 'skipgram'.

size is the dimensionality of the feature vectors.

window is the maximum distance between the current and predicted word within a sentence.

alpha is the initial learning rate.

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

word_ngram = max length of word ngram

loss = defines training objective. Allowed values are *hs* (hierarchical softmax), *ns* (negative sampling) and *softmax*. Defaults to *ns*

sample = threshold for configuring which higher-frequency words are randomly downsampled;
default is 1e-3, useful range is (0, 1e-5).

negative = the value for negative specifies how many "noise words" should be drawn (usually between 5-20). Default is 5. If set to 0, no negative samping is used. Only relevant when *loss* is set to *ns*

iter = number of iterations (epochs) over the corpus. Default is 5.

min_n = min length of char ngrams to be used for training word representations. Default is 3.

max_n = max length of char ngrams to be used for training word representations. Set *max_n* to be lesser than *min_n* to avoid char ngrams being used. Default is 6.

sorted_vocab = if 1 (default), sort the vocabulary by descending frequency before assigning word indexes.

threads = number of threads to use. Default is 12.

---

update_weights()

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

---

wmdistance(**args*, ***kwargs*)

Deprecated. Use self.wv.wmdistance() instead. Refer to the documentation for *gensim.models.KeyedVectors.wmdistance*

---

class gensim.models.wrappers.fasttext.FastTextKeyedVectors

Bases: gensim.models.keyedvectors.EuclideanKeyedVectors

Class to contain vectors, vocab and ngrams for the FastText training class and other methods not directly involved in training such as most_similar(). Subclasses KeyedVectors to implement oov lookups, storing ngrams and other FastText specific methods

---

accuracy(*questions*, *restrict_vocab=30000*, *most_similar=<function most_similar>*, *case_insensitive=True*)

Compute accuracy of the model. *questions* is a filename where lines are 4-tuples of words, split into sections by ": SECTION NAME" lines. See questions-words.txt in **https://storage.googleapis.com/google-code-archive-source/v2/code.google.com/word2vec/source-archive.zip** for an example.

The accuracy is reported (=printed to log and returned as a list) for each section separately, plus there's one aggregate summary at the end.

Use *restrict_vocab* to ignore all questions containing a word not in the first *restrict_vocab* words (default 30,000). This may be meaningful if you've sorted the vocabulary by descending frequency. In case *case_insensitive* is True, the first *restrict_vocab* words are taken first, and then case normalization is performed.

Use *case_insensitive* to convert all words in questions and vocab to their uppercase form before evaluating the accuracy (default True). Useful in case of case-mismatch between training tokens and question words. In case of multiple case variants of a single word, the vector for the first occurrence (also the most frequent if vocabulary is sorted) is taken.

This method corresponds to the *compute-accuracy* script of the original C word2vec.

---

cosine_similarities(*vector_1*, *vectors_all*)

Return cosine similarities between one vector and a set of other vectors.

Parameters:
- **vector_1** (*numpy.array*) – vector from which similarities are to be computed. expected shape (dim,)
- **vectors_all** (*numpy.array*) – for each row in vectors_all, distance from vector_1 is computed. expected shape (num_vectors, dim)

Returns: Contains cosine distance between vector_1 and each row in vectors_all. shape (num_vectors,)

Return type: numpy.array

---

distance(*w1*, *w2*)

Compute cosine distance between two words.

Example:

```
>>> trained_model.distance('woman', 'man')
0.34

>>> trained_model.distance('woman', 'woman')
0.0
```

---

distances(*word_or_vector*, *other_words=()*)

Compute cosine distances from given word or vector to all words in *other_words*. If *other_words* is empty, return distance between *word_or_vectors* and all words in vocab.

Parameters:
- **word_or_vector** (*str or numpy.array*) – Word or vector from which distances are to be computed.
- **other_words** (*iterable(str) or None*) – For each word in *other_words* distance from *word_or_vector* is computed. If None or empty, distance of *word_or_vector* from all words in vocab is computed (including itself).

Returns: Array containing distances to all words in *other_words* from input *word_or_vector*, in the same order as *other_words*.

Return type: numpy.array

Notes

Raises KeyError if either *word_or_vector* or any word in *other_words* is absent from vocab.

---

doesnt_match(*words*)

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

Example:

```
>>> trained_model.doesnt_match("breakfast cereal dinner lunch".split())
'cereal'
```

---

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

Compute correlation of the model with human similarity judgments. *pairs* is a filename of a dataset where lines are 3-tuples, each consisting of a word pair and a similarity value, separated by *delimiter*. An example dataset is included in Gensim (test/test_data/wordsim353.tsv). More datasets can be found at **http://technion.ac.il/~ira.leviant/MultilingualVSMdata.html** or **https://www.cl.cam.ac.uk/~fh295/simlex.html**.

The model is evaluated using Pearson correlation coefficient and Spearman rank-order correlation coefficient between the similarities from the dataset and the similarities produced by the model itself. The results are printed to log and returned as a triple (pearson, spearman, ratio of pairs with unknown words).

Use *restrict_vocab* to ignore all word pairs containing a word not in the first *restrict_vocab* words (default 300,000). This may be meaningful if you've sorted the vocabulary by descending frequency. If *case_insensitive* is True, the first *restrict_vocab* words are taken, and then case normalization is performed.

Use *case_insensitive* to convert all words in the pairs and vocab to their uppercase form before evaluating the model (default True). Useful when you expect case-mismatch between training tokens and words pairs in the dataset. If there are multiple case variants of a single word, the vector for the first occurrence (also the most frequent if vocabulary is sorted) is taken.

Use *dummy4unknown=True* to produce zero-valued similarities for pairs with out-of-vocabulary words. Otherwise (default False), these pairs are skipped entirely.

---

get_keras_embedding(*train_embeddings=False*)

Return a Keras 'Embedding' layer with weights set as the Word2Vec model's learned word embeddings

---

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** after doing a replace. The model becomes effectively read-only = you can only call *most_similar*, *similarity* etc.

---

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.

classmethod load_word2vec_format(**args*, ***kwargs*)

Not suppported. Use gensim.models.KeyedVectors.load_word2vec_format instead.

---

log_accuracy(*section*)

---

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

---

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

Find the top-N most similar words. Positive words contribute positively towards the similarity, negative words negatively.

This method computes cosine similarity between a simple mean of the projection weight vectors of the given words and the vectors for each word in the model. The method corresponds to the *word-analogy* and *distance* scripts in the original word2vec implementation.

If topn is False, most_similar returns the vector of similarity scores.

restrict_vocab is an optional integer which limits the range of vectors which are searched for most-similar values. For example, restrict_vocab=10000 would only check the first 10000 word vectors in the vocabulary order. (This may be meaningful if you've sorted the vocabulary by descending frequency.)

Example:

```
>>> trained_model.most_similar(positive=['woman', 'king'], negative=['man'])
[('queen', 0.50882536), ...]
```

---

most_similar_cosmul(*positive=None*, *negative=None*, *topn=10*)

Find the top-N most similar words, using the multiplicative combination objective proposed by Omer Levy and Yoav Goldberg in [4]. Positive words still contribute positively towards the similarity, negative words negatively, but with less susceptibility to one large distance dominating the calculation.

In the common analogy-solving case, of two positive and one negative examples, this method is equivalent to the "3CosMul" objective (equation (4)) of Levy and Goldberg.

Additional positive or negative examples contribute to the numerator or denominator, respectively – a potentially sensible but untested extension of the method. (With a single positive example, rankings will be the same as in the default most_similar.)

Example:

```
>>> trained_model.most_similar_cosmul(positive=['baghdad', 'england'], negative=['london'])
[(u'iraq', 0.8488819003105164), ...]
```

[4] Omer Levy and Yoav Goldberg. Linguistic Regularities in Sparse and Explicit Word Representations, 2014.

---

most_similar_to_given(*w1*, *word_list*)

Return the word from word_list most similar to w1.

Parameters:
- **w1** (*str*) – a word
- **word_list** (*list*) – list of words containing a word most similar to w1

Returns: the word in word_list with the highest similarity to w1

Raises: KeyError – If w1 or any word in word_list is not in the vocabulary

Example:

```
>>> trained_model.most_similar_to_given('music', ['water', 'sound', 'backpack', 'mouse'])
'sound'

>>> trained_model.most_similar_to_given('snake', ['food', 'pencil', 'animal', 'phone'])
'animal'
```

n_similarity(*ws1*, *ws2*)

Compute cosine similarity between two sets of words.

Example:

```
>>> trained_model.n_similarity(['sushi', 'shop'], ['japanese', 'restaurant'])
0.61540466561049689

>>> trained_model.n_similarity(['restaurant', 'japanese'], ['japanese', 'restaurant'])
1.0000000000000004

>>> trained_model.n_similarity(['sushi'], ['restaurant']) == trained_model.similarity('sushi', 'restaurant')
True
```

rank(*w1*, *w2*)

Rank of the distance of *w2* from *w1*, in relation to distances of all words from *w1*.

Parameters:
- **w1** (*str*) – Input word.
- **w2** (*str*) – Input word.

Returns: Rank of *w2* from *w1* in relation to all other nodes.

Return type: int

Examples

```
>>> model.rank('mammal.n.01', 'carnivore.n.01')
3
```

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

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

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

> *fname* is the file used to save the vectors in *fvocab* is an optional file used to save the vocabulary *binary* is an optional boolean indicating whether the data is to be saved in binary word2vec format (default: False) *total_vec* is an optional parameter to explicitly specify total no. of vectors (in case word vectors are appended with document vectors afterwards)

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

Find the top-N most similar words by vector.

If topn is False, similar_by_vector returns the vector of similarity scores.

restrict_vocab is an optional integer which limits the range of vectors which are searched for most-similar values. For example, restrict_vocab=10000 would only check the first 10000 word vectors in the vocabulary order. (This may be meaningful if you've sorted the vocabulary by descending frequency.)

Example:

```
>>> trained_model.similar_by_vector([1,2])
[('survey', 0.9942699074745178), ...]
```

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

Find the top-N most similar words.

If topn is False, similar_by_word returns the vector of similarity scores.

restrict_vocab is an optional integer which limits the range of vectors which are searched for most-similar values. For example, restrict_vocab=10000 would only check the first 10000 word vectors in the vocabulary order. (This may be meaningful if you've sorted the vocabulary by descending frequency.)

Example:

```
>>> trained_model.similar_by_word('graph')
[('user', 0.9999163150787354), ...]
```

similarity(*w1*, *w2*)

Compute cosine similarity between two words.

Example:

```
>>> trained_model.similarity('woman', 'man')
0.73723527

>>> trained_model.similarity('woman', 'woman')
1.0
```

wmdistance(*document1*, *document2*)

Compute the Word Mover's Distance between two documents. When using this code, please consider citing the following papers:

Note that if one of the documents have no words that exist in the Word2Vec vocab, *float('inf')* (i.e. infinity) will be returned.

This method only works if *pyemd* is installed (can be installed via pip, but requires a C compiler).

Example

```
>>> # Train word2vec model.
>>> model = Word2Vec(sentences)
```

```
>>> # Some sentences to test.
>>> sentence_obama = 'Obama speaks to the media in Illinois'.lower().split()
>>> sentence_president = 'The president greets the press in Chicago'.lower().split()
```

```
>>> # Remove their stopwords.
>>> from nltk.corpus import stopwords
```

```
>>> stopwords = nltk.corpus.stopwords.words('english')
>>> sentence_obama = [w for w in sentence_obama if w not in stopwords]
>>> sentence_president = [w for w in sentence_president if w not in stopwords]
```

```
>>> # Compute WMD.
>>> distance = model.wmdistance(sentence_obama, sentence_president)
```

word_vec(*word*, *use_norm=False*)

Accept a single word as input. Returns the word's representations in vector space, as a 1D numpy array.

The word can be out-of-vocabulary as long as ngrams for the word are present. For words with all ngrams absent, a KeyError is raised.

Example:

```
>>> trained_model['office']
array([ -1.40128313e-02, ...])
```

words_closer_than(*w1*, *w2*)

Returns all words that are closer to *w1* than *w2* is to *w1*.

Parameters:
- **w1** (*str*) – Input word.
- **w2** (*str*) – Input word.

Returns: List of words that are closer to *w1* than *w2* is to *w1*.

Return type: list (str)

Examples

```
>>> model.words_closer_than('carnivore.n.01', 'mammal.n.01')
['dog.n.01', 'canine.n.02']
```

wv

gensim.models.wrappers.fasttext.compute_ngrams(*word*, *min_n*, *max_n*) gensim.models.wrappers.fasttext.ft_hash(*string*)

Reproduces [hash method](**https://github.com/facebookresearch/fastText/blob/master/src/dictionary.cc**) used in fastText.