models.fasttext – FastText model

Learn word representations via Fasttext: **Enriching Word Vectors with Subword Information**.

This module allows training word embeddings from a training corpus with the additional ability to obtain word vectors for out-of-vocabulary words.

This module contains a fast native C implementation of Fasttext with Python interfaces. It is **not** only a wrapper around Facebook's implementation.

This module supports loading models trained with Facebook's fastText implementation. It also supports continuing training from such models.

For a tutorial see **this notebook**.

Make sure you have a C compiler before installing Gensim, to use the optimized (compiled) Fasttext training routines.

Usage examples

Initialize and train a model:

```
>>> # from gensim.models import FastText  # FIXME: why does Sphinx dislike this import?
>>> from gensim.test.utils import common_texts  # some example sentences
>>>
>>> print(common_texts[0])
['human', 'interface', 'computer']
>>> print(len(common_texts))
9
>>> model = FastText(size=4, window=3, min_count=1)  # instantiate
>>> model.build_vocab(sentences=common_texts)
>>> model.train(sentences=common_texts, total_examples=len(common_texts), epochs=10)  # train
```

Once you have a model, you can access its keyed vectors via the *model.wv* attributes. The keyed vectors instance is quite powerful: it can perform a wide range of NLP tasks. For a full list of examples, see `FastTextKeyedVectors`.

You can also pass all the above parameters to the constructor to do everything in a single line:

```
>>> model2 = FastText(size=4, window=3, min_count=1, sentences=common_texts, iter=10)
```

Important

This style of initialize-and-train in a single line is **deprecated**. We include it here for backward compatibility only.

Please use the initialize-*build_vocab-train* pattern above instead, including using *epochs* instead of *iter*. The motivation is to simplify the API and resolve naming inconsistencies, e.g. the iter parameter to the constructor is called epochs in the train function.

The two models above are instantiated differently, but behave identically. For example, we can compare the embeddings they've calculated for the word "computer":

```
>>> import numpy as np
>>>
>>> np.allclose(model.wv['computer'], model2.wv['computer'])
True
```

In the above examples, we trained the model from sentences (lists of words) loaded into memory. This is OK for smaller datasets, but for larger datasets, we recommend streaming the file, for example from disk or the network. In Gensim, we refer to such datasets as "corpora" (singular "corpus"), and keep them in the format described in `LineSentence`. Passing a corpus is simple:

```
>>> from gensim.test.utils import datapath
>>>
>>> corpus_file = datapath('lee_background.cor')  # absolute path to corpus
>>> model3 = FastText(size=4, window=3, min_count=1)
>>> model3.build_vocab(corpus_file=corpus_file)  # scan over corpus to build the vocabulary
>>>
>>> total_words = model3.corpus_total_words  # number of words in the corpus
>>> model3.train(corpus_file=corpus_file, total_words=total_words, epochs=5)
```

The model needs the *total_words* parameter in order to manage the training rate (alpha) correctly, and to give accurate progress estimates. The above example relies on an implementation detail: the `build_vocab()` method sets the *corpus_total_words* (and also *corpus_count*) model attributes. You may calculate them by scanning over the corpus yourself, too.

If you have a corpus in a different format, then you can use it by wrapping it in an **iterator**. Your iterator should yield a list of strings each time, where each string should be a separate word. Gensim will take care of the rest:

```
>>> from gensim.utils import tokenize
>>> from gensim import utils
>>>
>>>
>>> class MyIter(object):
...     def __iter__(self):
...         path = datapath('crime-and-punishment.txt')
...         with utils.open(path, 'r', encoding='utf-8') as fin:
...             for line in fin:
...                 yield list(tokenize(line))
>>>
>>>
>>> model4 = FastText(size=4, window=3, min_count=1)
>>> model4.build_vocab(sentences=MyIter())
>>> total_examples = model4.corpus_count
>>> model4.train(sentences=MyIter(), total_examples=total_examples, epochs=5)
```

Persist a model to disk with:

```
>>> from gensim.test.utils import get_tmpfile
>>>
>>> fname = get_tmpfile("fasttext.model")
>>>
>>> model.save(fname)
>>> model = FastText.load(fname)
```

Once loaded, such models behave identically to those created from scratch. For example, you can continue training the loaded model:

```
>>> import numpy as np
>>>
>>> 'computation' in model.wv.vocab  # New word, currently out of vocab
False
>>> old_vector = np.copy(model.wv['computation'])  # Grab the existing vector
>>> new_sentences = [
...     ['computer', 'aided', 'design'],
...     ['computer', 'science'],
...     ['computational', 'complexity'],
...     ['military', 'supercomputer'],
...     ['central', 'processing', 'unit'],
...     ['onboard', 'car', 'computer'],
... ]
>>>
>>> model.build_vocab(new_sentences, update=True)  # Update the vocabulary
>>> model.train(new_sentences, total_examples=len(new_sentences), epochs=model.epochs)
>>>
>>> new_vector = model.wv['computation']
>>> np.allclose(old_vector, new_vector, atol=1e-4)  # Vector has changed, model has learnt something
False
>>> 'computation' in model.wv.vocab  # Word is still out of vocab
False
```

Important

Be sure to call the `build_vocab()` method with *update=True* before the `train()` method when continuing training. Without this call, previously unseen terms will not be added to the vocabulary.

You can also load models trained with Facebook's fastText implementation:

```
>>> cap_path = datapath("crime-and-punishment.bin")
>>> fb_model = load_facebook_model(cap_path)
```

Once loaded, such models behave identically to those trained from scratch. You may continue training them on new data:

```
>>> 'computer' in fb_model.wv.vocab  # New word, currently out of vocab
False
>>> old_computer = np.copy(fb_model.wv['computer'])  # Calculate current vectors
>>> fb_model.build_vocab(new_sentences, update=True)
>>> fb_model.train(new_sentences, total_examples=len(new_sentences), epochs=model.epochs)
>>> new_computer = fb_model.wv['computer']
>>> np.allclose(old_computer, new_computer, atol=1e-4)  # Vector has changed, model has learnt something
False
>>> 'computer' in fb_model.wv.vocab  # New word is now in the vocabulary
True
```

If you do not intend to continue training the model, consider using the `gensim.models.fasttext.load_facebook_vectors()` function instead. That function only loads the word embeddings (keyed vectors), consuming much less CPU and RAM:

```
>>> from gensim.test.utils import datapath
>>>
>>> cap_path = datapath("crime-and-punishment.bin")
>>> wv = load_facebook_vectors(cap_path)
>>>
>>> 'landlord' in wv.vocab  # Word is out of vocabulary
```

```
False
>>> oov_vector = wv['landlord']
>>>
>>> 'landlady' in wv.vocab  # Word is in the vocabulary
True
>>> iv_vector = wv['landlady']
```

Retrieve word-vector for vocab and out-of-vocab word:

```
>>> existent_word = "computer"
>>> existent_word in model.wv.vocab
True
>>> computer_vec = model.wv[existent_word]  # numpy vector of a word
>>>
>>> oov_word = "graph-out-of-vocab"
>>> oov_word in model.wv.vocab
False
>>> oov_vec = model.wv[oov_word]  # numpy vector for OOV word
```

You can perform various NLP word tasks with the model, some of them are already built-in:

```
>>> similarities = model.wv.most_similar(positive=['computer', 'human'], negative=['interface'])
>>> most_similar = similarities[0]
>>>
>>> similarities = model.wv.most_similar_cosmul(positive=['computer', 'human'], negative=['interface'])
>>> most_similar = similarities[0]
>>>
>>> not_matching = model.wv.doesnt_match("human computer interface tree".split())
>>>
>>> sim_score = model.wv.similarity('computer', 'human')
```

Correlation with human opinion on word similarity:

```
>>> from gensim.test.utils import datapath
>>>
>>> similarities = model.wv.evaluate_word_pairs(datapath('wordsim353.tsv'))
```

And on word analogies:

```
>>> analogies_result = model.wv.evaluate_word_analogies(datapath('questions-words.txt'))
```

Implementation Notes

These notes may help developers navigate our fastText implementation. The implementation is split across several submodules:

- gensim.models.fasttext: This module. Contains FastText-specific functionality only.
- gensim.models.keyedvectors: Implements both generic and FastText-specific functionality.
- gensim.models.word2vec: Contains implementations for the vocabulary and the trainables for FastText.
- gensim.models.base_any2vec: Contains implementations for the base. classes, including functionality such as callbacks, logging.
- gensim.models.utils_any2vec: Wrapper over Cython extensions.
- gensim.utils: Implements model I/O (loading and saving).

Our implementation relies heavily on inheritance. It consists of several important classes:

- Word2VecVocab: the vocabulary. Keeps track of all the unique words, sometimes discarding the extremely rare ones. This is sometimes called the Dictionary within Gensim.
- FastTextKeyedVectors: the vectors. Once training is complete, this class is sufficient for calculating embeddings.
- FastTextTrainables: the underlying neural network. The implementation uses this class to *learn* the word embeddings.
- FastText: ties everything together.

class gensim.models.fasttext.FastText(*sentences=None*, *corpus_file=None*, *sg=0*, *hs=0*, *size=100*, *alpha=0.025*, *window=5*, *min_count=5*, *max_vocab_size=None*, *word_ngrams=1*, *sample=0.001*, *seed=1*, *workers=3*, *min_alpha=0.0001*, *negative=5*, *ns_exponent=0.75*, *cbow_mean=1*, *hashfxn=<built-in function hash>*, *iter=5*, *null_word=0*, *min_n=3*, *max_n=6*, *sorted_vocab=1*, *bucket=2000000*, *trim_rule=None*, *batch_words=10000*, *callbacks=()*, *compatible_hash=True*)

Bases: gensim.models.base_any2vec.BaseWordEmbeddingsModel

Train, use and evaluate word representations learned using the method described in **Enriching Word Vectors with Subword Information**, aka FastText.

The model can be stored/loaded via its save() and load() methods, or loaded from a format compatible with the original Fasttext implementation via load_facebook_model().

wv

This object essentially contains the mapping between words and embeddings. These are similar to the embeddings computed in the Word2Vec, however here we also include vectors for n-grams. This allows the model to compute embeddings even for **unseen** words (that do not exist in the vocabulary), as the aggregate of the n-grams included in the word. After training the model, this attribute can be used directly to query those embeddings in various ways. Check the module level docstring for some examples.

Type: FastTextKeyedVectors

vocabulary

This object represents the vocabulary of the model. Besides keeping track of all unique words, this object provides extra functionality, such as constructing a huffman tree (frequent words are closer to the root), or discarding extremely rare words.

Type: `FastTextVocab`

`trainables`

This object represents the inner shallow neural network used to train the embeddings. This is very similar to the network of the `Word2Vec` model, but it also trains weights for the N-Grams (sequences of more than 1 words). The semantics of the network are almost the same as the one used for the `Word2Vec` model. You can think of it as a NN with a single projection and hidden layer which we train on the corpus. The weights are then used as our embeddings. An important difference however between the two models, is the scoring function used to compute the loss. In the case of FastText, this is modified in word to also account for the internal structure of words, besides their concurrence counts.

Type: `FastTextTrainables`

Parameters:

- **sentences** (*iterable of list of str, optional*) – 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. If you don't supply *sentences*, 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 *sentences* to get performance boost. Only one of *sentences* or *corpus_file* arguments need to be passed (or none of them, in that case, the model is left uninitialized).
- **min_count** (*int, optional*) – The model ignores all words with total frequency lower than this.
- **size** (*int, optional*) – Dimensionality of the word vectors.
- **window** (*int, optional*) – The maximum distance between the current and predicted word within a sentence.
- **workers** (*int, optional*) – Use these many worker threads to train the model (=faster training with multicore machines).
- **alpha** (*float, optional*) – The initial learning rate.
- **min_alpha** (*float, optional*) – Learning rate will linearly drop to *min_alpha* as training progresses.
- **sg** (*{1, 0}, optional*) – Training algorithm: skip-gram if *sg=1*, otherwise CBOW.
- **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.
- **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).
- **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).
- **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.
- **cbow_mean** (*{1,0}, optional*) – If 0, use the sum of the context word vectors. If 1, use the mean, only applies when cbow is used.
- **hashfxn** (*function, optional*) – Hash function to use to randomly initialize weights, for increased training reproducibility.
- **iter** (*int, optional*) – Number of iterations (epochs) over the corpus.
- **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 themodel.

  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.

- **sorted_vocab** (*{1,0}, optional*) – If 1, sort the vocabulary by descending frequency before assigning word indices.
- **batch_words** (*int, optional*) – Target size (in words) for batches of examples passed to worker threads (and thus cython routines).(Larger batches will be passed if individual texts are longer than 10000 words, but the standard cython code truncates to that maximum.)
- **min_n** (*int, optional*) – Minimum length of char n-grams to be used for training word representations.
- **max_n** (*int, optional*) – 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.
- **word_ngrams** (*{1,0}, optional*) – If 1, uses enriches word vectors with subword(n-grams) information. If 0, this is equivalent to `Word2Vec`.

- **bucket** (*int, optional*) – Character ngrams are hashed into a fixed number of buckets, in order to limit the memory usage of the model. This option specifies the number of buckets used by the model.
- **callbacks** – List of callbacks that need to be executed/run at specific stages during training.
- **compatible_hash** (*bool, optional*) – By default, newer versions of Gensim's FastText use a hash function that is 100% compatible with Facebook's FastText. Older versions were not 100% compatible due to a bug. To use the older, incompatible hash function, set this to False.

Examples

Initialize and train a *FastText* model:

```
>>> from gensim.models import FastText
>>> sentences = [["cat", "say", "meow"], ["dog", "say", "woof"]]
>>>
>>> model = FastText(sentences, min_count=1)
>>> say_vector = model.wv['say']  # get vector for word
>>> of_vector = model.wv['of']  # get vector for out-of-vocab word
```

`__getitem__`(***kwargs*)

Deprecated. Use self.wv.__getitem__() instead.

Refer to the documentation for `gensim.models.keyedvectors.KeyedVectors.__getitem__()`

`accuracy`(***kwargs*)

`bucket`

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

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

Parameters:

- **sentences** (*iterable of list of str, optional*) – 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.
- **corpus_file** (*str, optional*) – Path to a corpus file in `LineSentence` format. You may use this argument instead of *sentences* to get performance boost. Only one of *sentences* or *corpus_file* arguments need to be passed (not both of them).
- **update** (*bool*) – If true, the new words in *sentences* 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 `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.

- ****kwargs** – Additional key word parameters passed to `build_vocab()`.

Examples

Train a model and update vocab for online training:

```
>>> from gensim.models import FastText
>>> sentences_1 = [["cat", "say", "meow"], ["dog", "say", "woof"]]
>>> sentences_2 = [["dude", "say", "wazzup!"]]
>>>
>>> model = FastText(min_count=1)
>>> model.build_vocab(sentences_1)
>>> model.train(sentences_1, total_examples=model.corpus_count, epochs=model.epochs)
>>>
>>> model.build_vocab(sentences_2, update=True)
>>> model.train(sentences_2, total_examples=model.corpus_count, epochs=model.epochs)
```

`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.

Parameters:

- **word_freq** (*dict of (str, int)*) – A mapping from a word in the vocabulary to its frequency count.
- **keep_raw_vocab** (*bool, optional*) – If False, delete the raw vocabulary after the scaling is done to 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 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.

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

`clear_sims`()

Remove all L2-normalized word vectors from the model, to free up memory.

You can recompute them later again using the `init_sims()` method.

`cum_table`

`doesnt_match`(***kwargs*)

Deprecated, use self.wv.doesnt_match() instead.

Refer to the documentation for `doesnt_match()`.

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

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

Parameters:
- **vocab_size** (*int, optional*) – Number of unique tokens in the vocabulary
- **report** (*dict of (str, int), optional*) – A dictionary from string representations of the 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.

Return type: dict of (str, int)

`evaluate_word_pairs`(***kwargs*)

Deprecated, use self.wv.evaluate_word_pairs() instead.

Refer to the documentation for `evaluate_word_pairs()`.

`hashfxn`

`init_sims`(*replace=False*)

Precompute L2-normalized vectors.

Parameters: **replace** (*bool*) – If True, forget the original vectors and only keep the normalized ones to save RAM.

`iter`

`layer1_size`

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

Load a previously saved *FastText* model.

Parameters: **fname** (*str*) – Path to the saved file.
Returns: Loaded model.
Return type: `FastText`

> See also
>
> `save()`
>   Save `FastText` model.

`load_binary_data`(***kwargs*)

Load data from a binary file created by Facebook's native FastText.

Parameters: **encoding** (*str, optional*) – Specifies the encoding.

classmethod `load_fasttext_format`(***kwargs*)

Deprecated.

Use `gensim.models.fasttext.load_facebook_model()` or `gensim.models.fasttext.load_facebook_vectors()` instead.

`max_n`

`min_count`

`min_n`

`most_similar`(***kwargs*)

Deprecated, use self.wv.most_similar() instead.

Refer to the documentation for `most_similar()`.

`most_similar_cosmul`(***kwargs*)

Deprecated, use self.wv.most_similar_cosmul() instead.

Refer to the documentation for `most_similar_cosmul()`.

`n_similarity`(***kwargs*)

Deprecated, use self.wv.n_similarity() instead.

Refer to the documentation for `n_similarity()`.

`num_ngram_vectors`

`sample`

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

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

Parameters: fname (*str*) – Store the model to this file.

See also

`load()`
Load `FastText` model.

`similar_by_vector`(***kwargs*)

Deprecated, use self.wv.similar_by_vector() instead.

Refer to the documentation for `similar_by_vector()`.

`similar_by_word`(***kwargs*)

Deprecated, use self.wv.similar_by_word() instead.

Refer to the documentation for `similar_by_word()`.

`similarity`(***kwargs*)

Deprecated, use self.wv.similarity() instead.

Refer to the documentation for `similarity()`.

`syn0_lockf`

`syn0_ngrams_lockf`

`syn0_vocab_lockf`

`syn1`

`syn1neg`

`train`(*sentences=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 from a sequence of sentences (can be a once-only generator stream). For FastText, each sentence must be a list of unicode strings.

To support linear learning-rate decay from (initial) *alpha* to *min_alpha*, and accurate progress-percentage logging, either *total_examples* (count of sentences) or *total_words* (count of raw words in sentences) **MUST** be provided. If *sentences* 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:

- **sentences** (*iterable of list of str, optional*) – 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.
- **corpus_file** (*str, optional*) – Path to a corpus file in `LineSentence` format. If you use this argument instead of *sentences*, you must provide *total_words* argument as well. Only one of *sentences* or *corpus_file* arguments need to be passed (not both of them).
- **total_examples** (*int*) – Count of sentences.
- **total_words** (*int*) – Count of raw words in sentences.
- **epochs** (*int*) – 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*) – Count of words already trained. Set this to 0 for the usual case of training on all words in sentences.
- **queue_factor** (*int*) – Multiplier for size of queue (number of workers * queue_factor).
- **report_delay** (*float*) – Seconds to wait before reporting progress.
- **callbacks** – List of callbacks that need to be executed/run at specific stages during training.

Examples

```
>>> from gensim.models import FastText
>>> sentences = [["cat", "say", "meow"], ["dog", "say", "woof"]]
>>>
>>> model = FastText(min_count=1)
>>> model.build_vocab(sentences)
>>> model.train(sentences, total_examples=model.corpus_count, epochs=model.epochs)
```

`wmdistance`(***kwargs*)

Deprecated, use self.wv.wmdistance() instead.

Refer to the documentation for `wmdistance()`.

class `gensim.models.fasttext.FastTextTrainables`(*vector_size=100*, *seed=1*, *hashfxn=<built-in function hash>*, *bucket=2000000*)

Bases: `gensim.models.word2vec.Word2VecTrainables`

Represents the inner shallow neural network used to train `FastText`.

Mostly inherits from its parent (`Word2VecTrainables`). Adds logic for calculating and maintaining ngram weights.

`hashfxn`

Used for randomly initializing weights. Defaults to the built-in hash()

Type: function

`layer1_size`

The size of the inner layer of the NN. Equal to the vector dimensionality. Set in the `Word2VecTrainables` constructor.

Type: int

`seed`

The random generator seed used in reset_weights and update_weights.

Type: float

`syn1`

The inner layer of the NN. Each row corresponds to a term in the vocabulary. Columns correspond to weights of the inner layer. There are layer1_size such weights. Set in the reset_weights and update_weights methods, only if hierarchical sampling is used.

Type: numpy.array

`syn1neg`

Similar to syn1, but only set if negative sampling is used.

Type: numpy.array

`vectors_lockf`

A one-dimensional array with one element for each term in the vocab. Set in reset_weights to an array of ones.

Type: numpy.array

`vectors_vocab_lockf`

Similar to vectors_vocab_lockf, ones(len(model.trainables.vectors), dtype=REAL)

Type: numpy.array

`vectors_ngrams_lockf`

np.ones((self.bucket, wv.vector_size), dtype=REAL)

Type: numpy.array

`init_ngrams_weights`(*wv*, *update=False*, *vocabulary=None*)

Compute 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.

Parameters:

- **wv** (`FastTextKeyedVectors`) – Contains the mapping between the words and embeddings. The vectors for the computed ngrams will go here.
- **update** (*bool*) – If True, the new vocab words and their new ngrams word vectors are initialized with random uniform distribution and updated/added to the existing vocab word and ngram vectors.
- **vocabulary** (`FastTextVocab`) – This object represents the vocabulary of the model. If update is True, then vocabulary may not be None.

`init_post_load`(*model*, *hidden_output*)

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).

`prepare_weights`(*hs*, *negative*, *wv*, *update=False*, *vocabulary=None*)

Build tables and model weights based on final vocabulary settings.

`reset_weights`(*hs*, *negative*, *wv*)

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

`save`(*fname_or_handle*, *separately=None*, *sep_limit=10485760*, *ignore=frozenset([])*, *pickle_protocol=2*)

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.

`seeded_vector`(*seed_string*, *vector_size*)

Get a random vector (but deterministic by seed_string).

`update_weights`(*hs*, *negative*, *wv*)

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

class `gensim.models.fasttext.FastTextVocab`(*max_vocab_size=None*, *min_count=5*, *sample=0.001*, *sorted_vocab=True*, *null_word=0*, *max_final_vocab=None*, *ns_exponent=0.75*)

Bases: `gensim.models.word2vec.Word2VecVocab`

This is a redundant class. It exists only to maintain backwards compatibility with older gensim versions.

`add_null_word`(*wv*)

`create_binary_tree`(*wv*)

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

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).

`make_cum_table`(*wv*, *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()`.

`prepare_vocab`(*hs*, *negative*, *wv*, *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.

`save`(*fname_or_handle*, *separately=None*, *sep_limit=10485760*, *ignore=frozenset([])*, *pickle_protocol=2*)

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.

scan_vocab(*sentences=None*, *corpus_file=None*, *progress_per=10000*, *workers=None*, *trim_rule=None*)

sort_vocab(*wv*)

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

gensim.models.fasttext.load_facebook_model(*path*, *encoding='utf-8'*)

Load the input-hidden weight matrix from Facebook's native fasttext *.bin* output file.

Notes

Facebook provides both *.vec* and *.bin* files with their modules. The former contains human-readable vectors. The latter contains machine-readable vectors along with other model parameters. This function requires you to **provide the full path to the .bin file**. It effectively ignores the *.vec* output file, since it is redundant.

This function uses the smart_open library to open the path. The path may be on a remote host (e.g. HTTP, S3, etc). It may also be gzip or bz2 compressed (i.e. end in *.bin.gz* or *.bin.bz2*). For details, see **https://github.com/RaRe-Technologies/smart_open**.

Parameters:
- **model_file** (*str*) – 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 the load entire fastText model.
- **encoding** (*str, optional*) – Specifies the file encoding.

Examples

Load, infer, continue training:

```
>>> from gensim.test.utils import datapath
>>>
>>> cap_path = datapath("crime-and-punishment.bin")
>>> fb_model = load_facebook_model(cap_path)
>>>
>>> 'landlord' in fb_model.wv.vocab  # Word is out of vocabulary
False
>>> oov_term = fb_model.wv['landlord']
>>>
>>> 'landlady' in fb_model.wv.vocab  # Word is in the vocabulary
True
>>> iv_term = fb_model.wv['landlady']
>>>
>>> new_sent = [['lord', 'of', 'the', 'rings'], ['lord', 'of', 'the', 'flies']]
>>> fb_model.build_vocab(new_sent, update=True)
>>> fb_model.train(sentences=new_sent, total_examples=len(new_sent), epochs=5)
```

Returns: The loaded model.
Return type: **gensim.models.fasttext.FastText**

See also

load_facebook_vectors() loads the word embeddings only. Its faster, but does not enable you to continue training.

gensim.models.fasttext.load_facebook_vectors(*path*, *encoding='utf-8'*)

Load word embeddings from a model saved in Facebook's native fasttext *.bin* format.

Notes

Facebook provides both *.vec* and *.bin* files with their modules. The former contains human-readable vectors. The latter contains machine-readable vectors along with other model parameters. This function requires you to **provide the full path to the .bin file**. It effectively ignores the *.vec* output file, since it is redundant.

This function uses the smart_open library to open the path. The path may be on a remote host (e.g. HTTP, S3, etc). It may also be gzip or bz2 compressed. For details, see **https://github.com/RaRe-Technologies/smart_open**.

Parameters:
- **path** (*str*) – The location of the model file.
- **encoding** (*str, optional*) – Specifies the file encoding.

Returns: The word embeddings.

Return type: **gensim.models.keyedvectors.FastTextKeyedVectors**

Examples

Load and infer:

```
>>> from gensim.test.utils import datapath
>>>
>>> cap_path = datapath("crime-and-punishment.bin")
>>> fbkv = load_facebook_vectors(cap_path)
>>>
>>> 'landlord' in fbkv.vocab  # Word is out of vocabulary
False
>>> oov_vector = fbkv['landlord']
>>>
>>> 'landlady' in fbkv.vocab  # Word is in the vocabulary
True
>>> iv_vector = fbkv['landlady']
```

See also

`load_facebook_model()` loads the full model, not just word embeddings, and enables you to continue model training.